Vizualizacija algoritama pretraživanja

Držaić, Dino

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:745320

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-04-01**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 335

VIZUALIZACIJA ALGORITAMA PRETRAŽIVANJA

Dino Držaić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 335

VIZUALIZACIJA ALGORITAMA PRETRAŽIVANJA

Dino Držaić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 335

Pristupnik: **Dino Držaić (0036521870)**

Studij: Računarstvo

Profil: Programsko inženjerstvo i informacijski sustavi

Mentorica: izv. prof. dr. sc. Josipa Pina Milišić

Zadatak: **Vizualizacija algoritama pretraživanja**

Opis zadatka:

U suvremenom podučavanju osnova programiranja pojavljuje se potreba za interaktivnim i grafički prilagođenim prikazom algoritama pretraživanja. Vizualizacijom algoritama korisnici mogu steći uvid u njihovu učinkovitost, usporediti različite algoritme, prepoznati njihove prednosti te uočiti nedostatke. Cilj ovog diplomskog rada je dati pregled, napraviti klasifikaciju te izraditi mrežni interaktivni alat temeljen na web tehnologiji za vizualizaciju različitih tehnika pretraživanja polja, grafova i stabala koje se mogu pronaći u egzaktnim metodama, metaheuristici i AI/planiranju. Web aplikacija za vizualizaciju predstavit će temeljnu logiku i osnovne razlike između najčešće korištenih algoritama pretraživanja. Napravit će se usporedba odabranih algoritama obzirom na njihovu prostornu i vremensku složenost s pripadnim prosječnim, najgorim i najboljim scenarijima.

Rok za predaju rada: 28. lipnja 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 335

VIZUALIZACIJA ALGORITAMA PRETRAŽIVANJA

Dino Držaić

Sadržaj

Uvod 1

1. Teorija grafova i složenosti algoritama – osnovni pojmovi 2

2. Algoritmi pretraživanja 9

2.1. Algoritmi pretraživanja polja 10

2.1.1. Linearno traženje 10

2.1.2. Linearno traženje korištenjem stražara (sentinel) 12

2.1.3. Binarno traženje 13

2.1.4. Algoritam ternarnog traženja (ternary search) 15

2.1.5. Algoritam traženja skokom (jump) 16

2.1.6. Eksponencijalno traženje 17

2.1.7. Fibonacci traženje 18

2.2. Algoritmi pretraživanja stabala 20

2.2.1. BFS 20

2.2.2. DFS 21

2.3. Algoritmi pretraživanja grafova 23

2.3.1. Dijkstra 23

2.3.2. Bellman-Ford 25

3. Vizualizacija algoritama pretraživanja 27

3.1. Odabir platforme 27

3.2. Odabir tehnologija 28

3.3. Komponente sučelja 28

3.3.1. Vizualizacija polja 28

3.3.2. Vizualizacija stabla 29

3.3.3. Vizualizacija grafa 30

3.4. Značajke aplikacije 31

3.4.1. Kontrole izvršavanja 32

Zaključak 34

Literatura 35

Sažetak 36

Summary 37

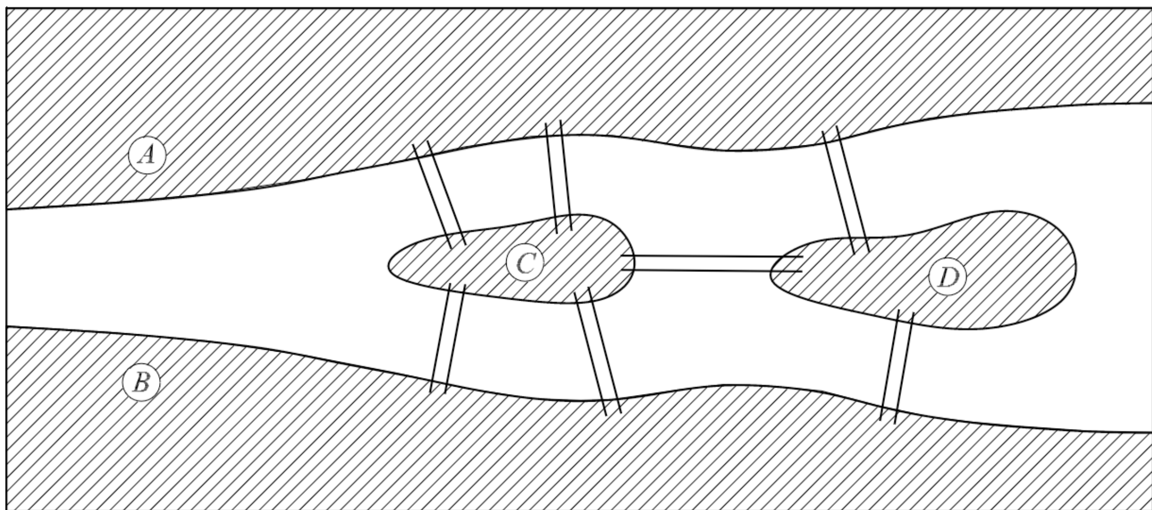
Skraćenice.. 38

Uvod

U području računalne znanosti i informatike, algoritmi pretraživanja igraju ključnu ulogu u pronalaženju određenih vrijednosti, resursa ili rješenja unutar različitih skupova podataka ili procesa. Oni su neophodni u raznim aplikacijama poput internetskih pretraživača, analize slika, strojnog učenja te kriptografije. Sama operacija pretraživanja može se primijeniti na razne strukture podataka, no mi ćemo se fokusirati na pretraživanje polja, stabala te grafova. Većina programskih inženjera upoznati su s algoritmima poput linearnog ili binarnog pretraživanja, međutim, postoje mnogi drugi manje poznati algoritmi koji nam također mogu biti zanimljivi. Kako bismo bolje razumjeli svaki od ovih algoritama, korisno je vizualizirati njihovo izvršavanje. Stoga je jedan od zadataka ovog diplomskog rada izraditi interaktivnu web aplikaciju koja će korisnicima vizualno prikazati izvršavanje svakog pojedinog algoritma i omogućiti međusobnu usporedbu algoritama. Osim što olakšava razumijevanje, vizualizacija će nam omogućiti i procjenu efikasnosti svakog algoritma, što je posebno važno u radu s velikim skupovima podataka koji su danas uobičajeni. Performanse pojedinog algoritma uvelike ovise o skupu ulaznih podataka. Razumijevanje prostornih i vremenskih složenosti algoritama traženja omogućuje inženjerima da izaberu optimalni algoritam za određene slučajeve korištenja. Ne postoji savršen algoritam koji je optimalan za svaki slučaj korištenja već je potrebno prilagoditi se zahtjevima rješenja koje razvijamo. Iz ovog možemo zaključiti da je odabir optimalnog algoritma moguć samo ako inženjer dobro poznaje ulazni skup podataka te dobro razumije kako pojedini algoritmi pretraživanja rade.

1. Teorija grafova i složenosti algoritama – osnovni pojmovi

U ovom poglavlju ćemo uvesti osnovne matematičke pojmove korištene u ovom radu. U tome će nam pomoći skripte [5] i [6]. Za razliku od mnogih drugih dijelova matematike, za teoriju grafova se točno može reći kada je zasnovana [5]. U svome članku iz 1736. godine švicarski matematičar Leonhard Euler riješio je jedan poznati stari problem. U današnjoj Rusiji postoji grad Kalinjingrad koji rijeka Pregel dijeli na četiri teritorija koji su u 18. stoljeću bili povezani sa 7 mostova (Sl. 1.1).



Sl. 1.1 Podjela grada na 4 područja [5]

Glavni problem bio je kako obići svaki dio grada, a da se svakim mostom pređe samo jedanput. Taj problem naziva se problem königsberških mostova. Grad se može prikazati kao neusmjereni graf, te se tako i svaka mreža iz stvarnoga svijeta kao npr. naftovod ili strujni krug, također mogu prirodno prikazati kao graf.

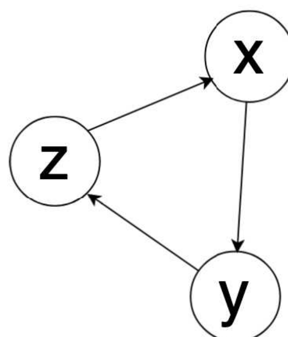
Jednostavni graf G sastoji se od nepraznog konačnog skupa V(G), čije elemente zovemo vrhovi (čvorovi) grafa G i konačnog skupa E(G) različitih dvočlanih podskupova skupa V(G) koje zovemo bridovi [5]. Oznaka V za skup vrhova dolazi od engleske riječi vertex za vrh, a oznaka E za skup bridova pak od engleske riječi edge za brid [5]. Za brid e = {v,w} kažemo da spaja vrhove v i w i bez mogućnosti zabune kraće ga pišemo vw. U toj situaciji kažemo da su vrhovi v i w grafa G susjedni [5]. Grafički su vrhovi grafova obično prikazani kružićima, a bridovi su prikazani spojnicama vrhova. Grafovi se najčešće zamišljaju kao

geometrijski oblici, te ih najčešće na neki način vizualiziramo, npr. skicom na papiru. Kad crtamo graf, potpuno je nebitno kako su čvorovi i bridovi raspoređeni u prostoru. Prema tome, slika (Sl. 1.2) prikazuje isti graf prikazan na dva različita načina.

Sl. 1.2 Različiti prikazi istog grafa

Za svaki čvor možemo odrediti stupanj čvora koji je jednak broju grana kojima je taj čvor krajnja točka.

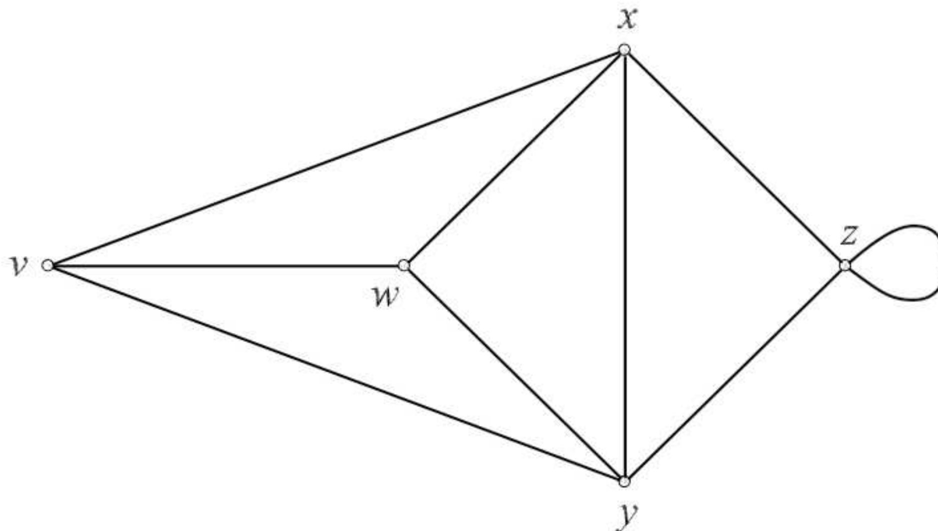
Osim neusmjerenih grafova, postoje i usmjereni grafovi kod kojih je svaki brid uređeni par čvorova. Usmjereni grafovi još se nazivaju i digrafovima. Stoga, za brid e = (x, y) kažemo da povezuje čvor x sa čvorom y, ali ne nužno i obratno. Kažemo da brid e počinje u čvoru x, a završava u čvoru y. Primjer usmjerenog dan je slikom (Sl. 1.3) gdje možemo vidjeti bridove (x, y), (y, z) te (z, x). Kod usmjerenih grafova, za svaki čvor možemo odrediti ulazni i izlazni stupanj čvora. Ulazni stupanj čvora je broj bridova koji završavaju u tom čvoru. Izlazni stupanj čvora je broj bridova koji počinju u tom čvoru. Čvor kojemu je ulazni stupanj jednak nuli nazivamo izvorom, a čvor kojemu je izlazni stupanj jednak nuli nazivamo ponorom. Usmjereni je graf čvrsto povezan ako za njega vrijedi da se iz svakog čvora može doći u svaki drugi čvor. Slabo povezan usmjereni graf je onaj graf koji postaje povezan graf kad bi se bridovi zamijenili neusmjerenima.



Sl. 1.3 Primjer usmjerenog grafa

Za zadane disjunktne grafove $G_1 = (V(G_1), E(G_1))$ i $G_2 = (V(G_2), E(G_2))$ definiramo njihovu uniju $G_1 \cup G_2$ kao graf $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ [5]. Graf je povezan ako se ne može prikazati kao unija neka dva grafa. U suprotnom kažemo da je graf nepovezan [5]. Šetnja u G je konačan slijed bridova oblika $v_0, v_1, v_2, . . , v_{m-1}, v_m$, također često u oznaci $v_0 \to v_1 \to \cdot \cdot \cdot \to v_m$, u kojem su svaka dva uzastopna brida ili susjedna ili jednaka [5].

Ako pogledamo graf (Sl. 1.4), jedna šetnja u tom grafu je na primjer $v \to w \to x \to y \to z \to z \to y \to w$. To je šetnja duljine 7.



Sl. 1.4 Primjer grafa iz skripte [5]

Šetnju u kojoj su svi bridovi različiti zovemo staza. Ako su, uz to, i svi vrhovi $v_0, v_1, . . . , v_m$ različiti (osim eventualno početni vrh v_0 i krajnji vrh v_m), onda takvu stazu zovemo put. Za stazu ili put kažemo da su zatvoreni ako je $v_0 = v_m$. Zatvoreni put koji sadrži barem jedan brid zovemo ciklus [5]. Težinski graf je jednostavan povezan graf u kojemu svaki brid e ima pridružen realan broj w(e) koji zovemo težina brida. Ponekad se umjesto pojma težina koristi pojam cijena prolaska. Ta težina se u stvarnom svijetu može interpretirati na razne načine, npr. duljina pojedine ceste neke prometne mreže, kapacitet cijevi nekog transportnog sustava i slično (Sl. 1.5). Oni također mogu biti usmjereni ili neusmjereni. Težina nekog puta jednaka je zbroju težina bridova od kojih se put sastoji.

Sl. 1.5 Primjer težinskog grafa u stvarnom svijetu

Udaljenost dva vrha u težinskom grafu možemo definirati kao duljinu najkraćeg puta između njih [5]. Povijesno najpoznatiji algoritam koji rješava problem najkraćeg puta je Dijkstrin algoritam. Svakom vrhu v zadanog težinskog grafa pridružimo realni broj l(v) koji označava gornju među za traženu udaljenost $d(u_0, v)$. Najprije početnom vrhu u_0 pridružimo $l(u_0) = 0$, te $l(v) = \infty$, za $v \neq u_0$. Skup S je pravi podskup skupa vrhova V(G). U početnom koraku je $S = \{u_0\}$, te $i = 0$. Drugi korak algoritma je pronalazak najkraćeg puta do svakog vrha iz skupa komplementarnog skupu S. To radimo na sljedeći način: Za svaki vrh $v \in \overline{S_i}$, zamijeni l(v) s $\min\{l(v), l(u_i) + w(u_i v)\}$. Izračunaj $\min\{l(v)\}$ gdje je $v \in \overline{S_i}$, te odredi u_{i+1} kao onaj vrh za koji se taj minimum postiže. Stavi $S_{i+1} = S_i \cup \{u_{i+1}\}$. Treći korak je povećanje broja i za jedan. Ako je $i = n - 1$, zaustavljamo se. Ako je $i < n - 1$, vraćamo se na drugi korak. Nakon izvršenja algoritma, svim vrhovima v zadanog težinskog grafa pridijeljena je vrijednost l(v) koja predstavlja duljinu najkraćeg puta od zadanog početnog vrha u_0 do njih samih [5].

Staze koje prolaze svim bridovima grafa su posebno važno strukturalno svojstvo grafa. Takve staze zovemo eulerovske staze, a graf koji posjeduje takvu stazu nazivamo eulerovski graf. Ako se znaju stupnjevi svih vrhova grafa, jednostavno je odrediti je li graf eulerovski ili ne. Povezani graf G je eulerovski onda i samo onda ako je stupanj svakog vrha paran [5]. Za pronalazak neke eulerovske staze u eulerovskom grafu najjednostavnije je koristiti

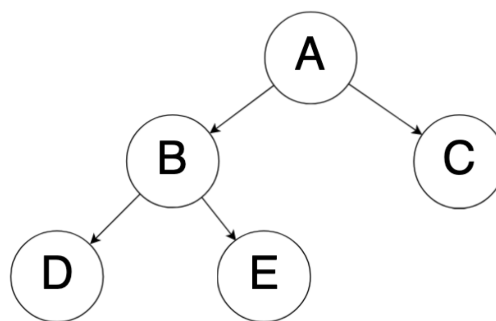
Fleuryev algoritam koji glasi ovako: Započni u bilo kojem vrhu *u* i prolazi vrhovima u bilo kojem redoslijedu, pazeći pritom samo na sljedeća pravila:

(i) prebriši bridove kojima si prošao, a ako nakon prolaska vrh ostane izoliran, pobriši i njega

(ii) prijeđi mostom trenutnog grafa samo ako nemaš druge mogućnosti [5].

Ciklus koji prolazi svim vrhovima nekog grafa nazivamo hamiltonovski ciklus, a graf koji posjeduje takav ciklus nazivamo hamiltonovski graf. Ustanoviti je li neki graf s n vrhova hamiltonovski možemo dakle tako da u njemu nađemo ciklus duljine n [5]. Takvi algoritmi načelno su faktorijelne složenosti jer je potrebno provjeriti svaku permutaciju skupa od n vrhova. Nije poznato postoji li algoritam polinomijalne složenosti za nalaženje hamiltonovskog ciklusa [5]. Po Diracovom teoremu možemo provjeriti je li neki graf hamiltonovski pomoću stupnjeva vrhova. Ako je G jednostavni graf s n ($n \geq 3$) vrhova, te ako je $\deg(v) \geq n/2$ za svaki vrh v iz G, onda je G hamiltonovski [5].

Jedna vrlo specifična klasa grafova su stabla. Šuma je graf bez ciklusa, a povezanu šumu zovemo stablo [5]. Stabla su po mnogočemu najjednostavniji grafovi. Ukorijenjeno stablo je stablo koje ima poseban čvor koji se naziva korijenom (Sl. 1.6).



Sl. 1.6 Primjer ukorijenjenog stabla

Svi bridovi ukorijenjenog stabla su usmjereni. Njihov je smjer prirodno uvijek od čvora bližeg korijenu prema čvoru daljem od korijena. Ako su dva čvora povezana, bližeg korijenu zovemo roditeljem, a daljeg djetetom. Čvorove koji nemaju djece nazivamo listovi. Općenito za grafove vrijedi da je stupanj vrha v grafa G jednak broju bridova koji su incidentni s v [5]. Iz te definicije možemo zaključiti da je stupanj čvora u stablu jednak broju djece koje ima čvor. Najveći stupanj čvora u stablu nazivamo stupanj stabla. Jedan od najčešće korištenih oblika stabala su binarna stabla. To su stabla u kojima svaki čvor ima stupanj najviše 2. Binarna stabla mogu se poopćiti na k-stabla koja predstavljaju stabla u kojima je

stupanj čvora jednak najviše k. Zato se binarna stabla nazivaju i 2-stabla. Za svaka dva vrha može se pronaći jedinstveni put koji ih povezuje. To se može dokazati kontradikcijom. Ako je G povezan graf i stablo, između svaka dva para vrhova postoji put. Kada bi postojala dva različita puta između neka dva vrha, unija tih putova bi bila zatvorena šetnja koja sigurno sadrži barem jedan ciklus, što je kontradikcija jer je stablo povezana šuma, a šuma je graf bez ciklusa. Stabla su neizostavan dio računalne znanosti. Neke od važnijih primjena su u strukturama podataka, računalnoj grafici, parsiranju podataka, umjetnoj inteligenciji i strojnom učenju, bazama podataka i još mnogim drugim granama računarstva. Stabla su također i temelj Kruskalovog algoritma koji se koristi za dizajniranje učinkovitih mreža poput telekomunikacijskih i računalnih mreža. Mnogi algoritmi koji rade nad stablima zahtijevaju da je stablo uravnoteženo. Stablo je uravnoteženo ako razlika u visini između lijevog i desnog podstabla bilo kojeg čvora ne prelazi unaprijed određenu granicu. Tako postoje više vrsta uravnoteženih stabala, a neke od poznatijih su AVL stablo, crveno-crno stablo i B-stablo. Uravnotežena stabla su važna zbog efikasnosti pretraživanja, brzih operacija umetanja i brisanja čvorova te predvidivosti performansi.

Točno vrijeme trajanja rada nekog algoritma i potrebnog prostora je gotovo nemoguće unaprijed izračunati, stoga ga procjenjujemo. Jedan prikladan oblik procjene u kojem pokušavamo ocijeniti vrijeme trajanja rada stroja je trajanje rada za velike ulazne podatke, koji zovemo asimptotska analiza [6]. Proučavamo ponašanje broja operacija kad veličina ulaznog skupa neograničeno raste. Neka su $f, g : N \rightarrow R+$ dvije funkcije. Kažemo da je funkcija g asimptotska gornja međa za funkciju f ako postoji $c > 0$ i $n_0 \in N$ tako da za svaki $n \geq n_0$ vrijedi $f(n) \leq cg(n)$ [6]. To označavamo s $f(n) = O(g(n))$.

Pravila notacije veliko O iz skripte [6]:

1. Ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$ tada je $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
2. Ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$ tada je $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$
3. Ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$ tada je $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
4. Ako je $p : N \rightarrow R+$ polinom stupnja k tada vrijedi $p(n) = O(n^k)$

Tablicom (Tablica 1.1) želimo naglasiti koliko su zapravo velike (približne) vrijednosti nekih funkcija.

n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n
5	3	15	25	125	32
10	4	40	100	10^3	10^3
100	7	700	10^4	10^6	10^{30}
1000	10	10^4	10^6	10^9	10^{300}

Tablica 1.1 Približne vrijednosti važnijih funkcija [6]

2. Algoritmi pretraživanja

Povijest algoritama pretraživanja

Ljudska vrsta ima potrebu efikasnog pretraživanja još od doba starih Egipćana. Još tada su ljudi pohranjivali stotine zapisa u arhivima na način da se što efikasnije može pronaći određeni zapis [8]. Ovaj problem su naslijedila računala kad smo počeli spremati zapise na njihove diskove. Područje algoritama traženja ima daleku prošlost u matematici, ali je doživjelo brzi rast nakon izuma računala. Počelo je izumom jednostavnih algoritama kao što su linearno i binarno traženje, a kasnije su izumljeni i napredniji algoritmi. Algoritmi traženja su bitno dobili na značaju izumom interneta kojeg ljudi danas pretražuju nekoliko puta dnevno. Najpoznatija internetska tražilica Google, nastala je 1998. godine, a od tada je promijenila mnoštvo algoritama koje koristi za pretraživanje [8]. Problem koji ti algoritmi danas pokušavaju riješiti mnogo je kompliciraniji od početnog problema pretraživanja zapisa u arhivu[8]. Od početaka algoritama traženja, bila je jasna podjela na dvije vrste algoritama, informirano i neinformirano traženje. Informirano traženje koristi heuristiku, odnosno dodatne informacije o stanju unutar algoritma. To je na primjer A* algoritam, koji je jedan od najstarijih i osnovnih informiranih algoritama traženja. Neinformirano traženje je slijepo traženje svih mogućih putanja bez ikakvog znanja o ulaznim podacima. Takvi algoritmi su na primjer BFS i DFS. BFS i DFS su jedni od najstarijih i najosnovnijih algoritama pretraživanja.

Karakteristike algoritama

Dvije najvažnije karakteristike algoritama pretraživanja su vremenska složenost i prostorna složenost.

Vremenska složenost algoritma opisuje kako vrijeme izvođenja programa varira s obzirom na veličinu ulaznih podataka, odnosno koliki utjecaj veličina ulaznih podataka ima na ukupan broj operacija potrebnih za rješavanje problema. Drugim riječima, to pokazuje koliko puta treba pristupiti svakom elementu i obaviti odgovarajuće operacije koje se tiču svakog pojedinačnog elementa, kao što je, na primjer, pretraživanje niza. Tipično se promatra najgori mogući slučaj poredaka elemenata, a pored toga zanima nas i kako se algoritam ponaša u najboljem mogućem slučaju, te kako se ponaša u nekom prosječnom

slučaju. Za opis najgoreg mogućeg slučaja koji nas najčešće zanima, koristi se takozvana O notacija. Iako postoje specifične notacije i za najbolji i prosječan slučaj, u praksi se O notacija koristi za sva tri slučaja pri čemu se navede o kojem slučaju se radi.

Kada pričamo općenito o složenosti algoritma, uglavnom se to misli na vremensku složenost, dok se prostorna složenost stavlja u drugi plan.

Prostorna složenost algoritma je količina memorije potrebna za izvršavanje algoritma. U prošlosti je prostorna složenost bila puno važnija nego danas jer je memorija bila skupa i poprilično nedostupna, dok danas to nije slučaj. Međutim, niti danas nije loše obratiti pažnju i na prostornu složenost, pogotovo ako radimo s velikim skupovima podataka. Za prikaz prostorne složenosti, također se koristi O notacija.

Neki primjeri O notacije su:

- O(1) – konstantna složenost, ne ovisi o veličini ulaznog skupa podataka
- O(n) – linearna složenost, složenost raste linearno ovisno o veličini ulaznog skupa podataka
- O(logn) – logaritamska složenost
- $O(n^2)$ – kvadratna složenost

2.1. Algoritmi pretraživanja polja

Polje je struktura podataka koja predstavlja kolekciju elemenata organiziranih na način da svaki element ima jedinstveni indeks. Indeksi su obično cijeli brojevi koji počinju od nule ili jedinice, ovisno o konvenciji jezika ili kontekstu. Najpopularniji algoritmi pretraživanja polja su linearno traženje i binarno traženje [1].

2.1.1. Linearno traženje

Linearno traženje je osnovni algoritam pretraživanja koji se koristi za pronalaženje određenog elementa u kolekciji podataka, najčešće polja. Ovaj algoritam prolazi kroz svaki element u polju jedan po jedan, počevši od početka, dok ne pronađe traženi element ili dođe do kraja polja. Lako je zaključiti da je najbolji slučaj ovog algoritma kada se traženi element nalazi na prvom mjestu polja, a najgori slučaj kada se traženi element nalazi na posljednjem mjestu polja.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(n), prostorna složenost O(1)
- Najgori slučaj: vremenska složenost O(n), prostorna složenost O(1)

Dokaz za vremensku složenost se može prikazati analizom broja operacija koje algoritam (Kôd 2.1) izvršava:

1. Inicijalizacija petlje: jedna operacija
2. Usporedba arr[i] === x: izvodi se n puta u najgorem slučaju
3. Usporedba i < arr.length: izvodi se n puta u najgorem slučaju
4. Povećanje brojila i: izvodi se n puta u najgorem slučaju

Ukupan broj operacija u najgorem slučaju je:

$$1 + n + n + n = 1 + 3n \tag{1}$$

Kada koristimo notaciju velikog O, zanemarujemo konstante i manje značajne članove:

$$O(1+3n)=O(n) \tag{2}$$

Algoritam koristi konstantnu količinu dodatne memorije potrebne za varijablu i, te eventualno za povratnu vrijednost što znači da je prostorna složenost O(1).

```
function linearSearch(arr, x) {
    for (let i = 0; i < arr.length; i++) {
      if (arr[i] === x) {
        return i;
      }
    }

    return -1;
}
```

Kôd 2.1 Izvedba linearnog traženja u programskom jeziku JS

2.1.2. Linearno traženje korištenjem stražara (sentinel)

Osnovna ideja ovog algoritma je dodati graničnu vrijednost na kraj niza koja je jednaka traženoj vrijednosti. To omogućuje izbjegavanje provjere jesmo li stigli do kraja niza u svakoj iteraciji petlje jer nam ta granična vrijednost služi kao stoper petlje. Iako su složenosti podjednake onima kod standardnog linearnog traženja, ovaj algoritam je optimiziraniji u prosječnom slučaju jer je ukupan broj usporedbi manji [1].

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(n), prostorna složenost O(1)
- Najgori slučaj: vremenska složenost O(n), prostorna složenost O(1)

Vremenska složenost analizira se na temelju broja operacija koje algoritam izvršava u odnosu na veličinu ulaznog niza n. Analizirat ćemo najgori slučaj u kojemu se element x nalazi na posljednjem mjestu ili nije u nizu. Broj operacija koje algoritam (Kôd 2.2) izvršava:

1. Inicijalizacije varijabli: 4 operacije
2. Usporedba uvjeta petlje: n puta
3. Povećanje brojila i: n puta
4. Sve ostale naredbe nakon izlaska iz petlje algoritam izvršava maksimalno jedanput pa je za njih potreban broj operacija jednak 3

Ukupan broj operacija je dakle:

$$4 + n + n + 3 = 2n + 7 \qquad (3)$$

Vremenska složenost je:

$$O(2n+7) = O(n) \qquad (4)$$

Kao i algoritam linearnog traženja, ovaj algoritam koristi konstantu veličinu dodatne memorije pa je tako prostorna složenost jednaka O(1).

```
function sentinelLinearSearch(arr, x) {
```

```
    const n = arr.length;
    const last = arr[n - 1];
    arr[n - 1] = x;
    let i = 0;

    while (arr[i] !== x) {
      i++;
    }

    arr[n - 1] = last;

    if (i < n - 1 || arr[n - 1] === x) {
      return i;
    }

    return -1;
}
```

Kôd 2.2 Izvedba linearnog traženja korištenjem stražara u programskom jeziku JS

2.1.3. Binarno traženje

Osnovna ideja binarnog traženja je smanjivanje vremenske složenosti uzastopnom podjelom intervala traženja na polovice. Ovaj algoritam radi samo pod uvjetom da je niz podataka sortiran. Algoritam dijeli interval traženja na polovice sve dok tražena vrijednost nije pronađena ili dok interval traženja ne postane prazan. U svakoj iteraciji se provjerava je li vrijednost koja se nalazi u sredini intervala traženja jednaka traženoj vrijednosti. Ako ta vrijednost nije jednaka traženoj vrijednosti, interval traženja dijelimo na dva manja intervala traženja. Prvi od ta dva intervala uključuje sve vrijednosti do vrijednosti koju smo provjeravali, a drugi uključuje sve vrijednosti koje slijede nakon provjeravane vrijednosti.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(logn), prostorna složenost O(1)
- Najgori slučaj: vremenska složenost O(logn), prostorna složenost O(1)

Dokaz složenosti za najgori slučaj kada element nije u nizu ili se nalazi na početku ili kraju niza ćemo pokazati analizom broja operacija koje će algoritam (Kôd 2.3) izvesti:

1. Broj iteracija petlje: $\log_2(n)$ zato što će se nakon $\log_2(n)$ koraka smanjiti veličina podniza na 1 što označava kraj izvođenja.
2. Broj operacija po iteraciji:
    a. Usporedba uvjeta petlje
    b. Izračunavanje srednjeg elementa
    c. Usporedba arr[mid] === x
    d. Usporedba arr[mid] < x
    e. Ažuriranje left ili right varijable

Ukupan broj operacija u najgorem slučaju:

$$\log2(n) * 5 = 5\log_2(n) \tag{5}$$

Vremenska složenost u tom slučaju je

$$O(5\log n) = O(\log n) \tag{6}$$

```
function binarySearch(arr, x) {
    let left = 0;
    let right = arr.length - 1;

    while (left <= right) {
      const mid = Math.floor((left + right) / 2);

      if (arr[mid] === x) {
        return mid;
      }

      if (arr[mid] < x) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
    }

    return -1;
```

```
}
```

Kôd 2.3 Izvedba binarnog traženja u programskom jeziku JS

2.1.4. Algoritam ternarnog traženja (ternary search)

Algoritam ternarnog traženja radi na isti princip kao algoritam binarnog traženja, samo što se interval traženja dijeli na tri umjesto na dva dijela. Također, radi samo pod uvjetom da je niz sortiran.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost $O(\log_3 n)$, prostorna složenost O(1)
- Najgori slučaj: vremenska složenost $O(\log_3 n)$, prostorna složenost O(1)

Složenosti ternarnog traženja mogu se dokazati na isti način kao i za binarno traženje. Iako bismo iz navedenih složenosti mogli zaključiti da je algoritam ternarnog traženja bolji od algoritma binarnog traženja, u praski to najčešće nije slučaj. Razlog tomu je duplo veći broj usporedbi kod algoritma ternarnog traženja, što u praksi ima značaj, a kod notacije veliko O se zanemaruje.

```
function ternarySearch(arr, x) {
    let left = 0;
    let right = arr.length - 1;

    while (left <= right) {
      const mid1 = left + Math.floor((right - left) / 3);
      const mid2 = right - Math.floor((right - left) / 3);

      if (arr[mid1] === x) {
        return mid1;
      }

      if (arr[mid2] === x) {
        return mid2;
      }

      if (arr[mid1] > x) {
```

```
      right = mid1 - 1;
    } else if (arr[mid2] < x) {
      left = mid2 + 1;
    } else {
      left = mid1 + 1;
      right = mid2 - 1;
    }
  }

  return -1;
}
```

Kôd 2.4 Izvedba ternarnog traženja u programskom jeziku JS

2.1.5. Algoritam traženja skokom (jump)

Osnovna ideja ovog algoritma je u prosjeku provjeriti manji broj elemenata od linearnog traženja na način da se rade određeni skokovi po listi te se tako preskoči određeni broj usporedbi. Algoritam traženja skokom zahtjeva da je ulazni niz sortiran. Radi jako dobro na nizovima koji su uniformno distribuirani.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost $O(\sqrt{n})$, prostorna složenost O(1)
- Najgori slučaj: vremenska složenost $O(\sqrt{n})$, prostorna složenost O(1)

Prvi korak ovog algoritma su skokovi unaprijed za korak veličine $\sqrt{n}$ sve dok ne pronađemo podniz u kojemu bi se mogao nalaziti traženi element ili dok ne dođemo do kraja niza. Broj koraka u ovoj petlji je u najgorem slučaju $\frac{n}{\sqrt{n}} = \sqrt{n}$. Drugi dio algoritma je linearno pretraživanje unutar podniza veličine $\sqrt{n}$ što u najgorem slučaju može zahtijevati $\sqrt{n}$ koraka. Ukupna vremenska složenost je dakle $O(\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n})$, što je značajno ubrzanje u odnosu na standardno linearno traženje. U većini slučajeva je binarno traženje brže od traženja skokom, međutim algoritam traženja skokom može biti pogodniji u slučajevima kada nam je operacija kretanja unatrag po nizu „skupa“, jer se kod algoritma traženja skokom u najgorem slučaju samo jedanput vraćamo unatrag, dok kod binarnog traženja možemo puno puta ići unatrag po nizu ako se traženi element nalazi blizu početka niza.

```
function jumpSearch(arr, x) {
    const n = arr.length;
    let step = Math.floor(Math.sqrt(n));
    let prev = 0;

    while (arr[Math.min(step, n) - 1] < x) {
      prev = step;
      step += Math.floor(Math.sqrt(n));

      if (prev >= n) {
        return -1;
      }
    }

    while (arr[prev] < x) {
      prev++;

      if (prev === Math.min(step, n)) {
        return -1;
      }
    }

    if (arr[prev] === x) {
      return prev;
    }

    return -1;
  }
```

Kôd 2.5 Izvedba traženja skokom u programskom jeziku JS

2.1.6. Eksponencijalno traženje

Eksponencijalno traženje je dobilo ime po tome što pretražuje podnizove čija se veličina eksponencijalno povećava. Prvo pretražuje podniz veličine jedan te provjerava je li zadnji element jednak traženom elementu, zatim pretražuje podniz veličine dva, pa veličine četiri, itd. sve dok zadnji element podniza nije veći od traženog elementa. Jednom kad pronađe podniz u kojem se nalazi traženi element, izvršava se binarno traženje na tom podnizu. Eksponencijalno traženje je izrazito korisno u slučaju neograničene pretrage gdje je veličina ulaznog niza beskonačna. Ovaj algoritam radi samo za sortirane nizove.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(logi), prostorna složenost O(1)
- Najgori slučaj: vremenska složenost O(logi), prostorna složenost O(1)

gdje je i pozicija traženog elementa u nizu [2].

Ovime zaključujemo da su performanse ovog algoritma izvrsne kad je traženi element blizu početka niza. Ako gledamo standardni zapis vremenske složenosti gdje nas zanima ovisnost o veličini ulaznog niza, ovaj algoritam u prosječnom i najgorem slučaju ima složenost O(logn) što je lako i dokazati. U najgorem slučaju `while` petlja se izvršava logn puta jer se brojilo `i` unutar petlje množi s dva. Nakon izvršavanja te petlje, provodi se binarno traženje nad podnizom veličine maksimalno n / 2, što znači da je vremenska složenost binarnog traženja u tom slučaju O(log (n/2)) što je jednako O(logn). Ukupna vremenska složenost je dakle O(logn) + O(logn) = O(logn).

```
function exponentialSearch(arr, x) {
    if (arr[0] === x) {
      return 0;
    }

    let i = 1;
    const n = arr.length;

    while (i < n && arr[i] <= x) {
      i *= 2;
    }

    return binarySearch(arr, x, i / 2, Math.min(i, n));
  }
```

Kôd 2.6 Izvedba eksponencijalnog traženja u programskom jeziku JS

2.1.7. Fibonacci traženje

Fibonacci traženje dijeli niz u nekoliko podnizova nejednake veličine. Ovaj algoritam zahtijeva da je ulazni niz sortiran. Fibonaccijev niz brojeva je slijed prirodnih brojeva kod kojega je svaki član, izuzevši prva dva, zbroj dvaju prethodnih brojeva, tj. niz

1,1,2,3,5,8,13,… [3]. Prvi korak algoritma je pronaći najmanji broj koji pripada Fibonaccijevom nizu, a koji je veći ili jednak dužini ulaznog niza. Nazovimo taj broj m-ti broj Fibonaccijevog niza. Koristimo broj koji prethodi m-tom broju za dvije pozicije kao indeks niza. Ako je traženi broj veći od broja koji se nalazi na poziciji s tim indeksom, ponavljamo postupak za desni podniz, a ako je traženi broj manji od broja koji se nalazi na poziciji s tim indeksom, ponavljamo postupak za lijevi podniz.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(logn), prostorna složenost O(1)
- Najgori slučaj: vremenska složenost O(logn), prostorna složenost O(1)

```
function fibonacciSearch(arr, x) {
    let fibMMm2 = 0;
    let fibMMm1 = 1;
    let fibM = fibMMm2 + fibMMm1;
    const n = arr.length;

    while (fibM < n) {
      fibMMm2 = fibMMm1;
      fibMMm1 = fibM;
      fibM = fibMMm2 + fibMMm1;
    }

    let offset = -1;

    while (fibM > 1) {
      const i = Math.min(offset + fibMMm2, n - 1);

      if (arr[i] < x) {
        fibM = fibMMm1;
        fibMMm1 = fibMMm2;
        fibMMm2 = fibM - fibMMm1;
        offset = i;
      } else if (arr[i] > x) {
        fibM = fibMMm2;
        fibMMm1 -= fibMMm2;
        fibMMm2 = fibM - fibMMm1;
```

```
      } else {
        return i;
      }
    }

    if (fibMMm1 && arr[offset + 1] === x) {
      return offset + 1;
    }

    return -1;
  }
```

Kôd 2.7 Izvedba Fibonacci traženja u programskom jeziku JS

2.2. Algoritmi pretraživanja stabala

Složenosti ovih algoritama više ne ovise samo o broju čvorova, već i o broju grana. Broj čvorova ćemo označavati s V, a broj grana s E.

2.2.1. BFS

BFS je skraćenica od Breadth First Search, tj. Pretraživanje u širinu. To je osnovni algoritam obilaska stabla. BFS se koristi i za pretraživanje grafova, jedina razlika to što grafovi mogu sadržavati cikluse pa moramo voditi računa o već obiđenim čvorovima. Ovaj algoritam prvo posjećuje sve čvorove na jednoj razini prije nego prijeđe na sljedeću razinu. Za to se koristi struktura podataka red. Struktura podataka red (engl. queue) se koristi za organiziranje elemenata na način da se pristupa elementima po principu "prvi unutra, prvi van" (FIFO - First In, First Out). To znači da se prvi element koji je dodan u red prvi i uklanja, slično kao što ljudi stoje u redu u trgovini. Tako je i sa čvorovima stabla, prvo se u red dodaje korijen stabla, zatim njegova djeca. Ako korijen ima više djece, prvo će se obići sva neposredna djeca korijena prije nego što se obiđu njihova djeca jer su neposredna djeca korijena prva dodana u red. Tako se postupak ponavlja dok traženi element nije pronađen ili dok se ne obiđe cijelo stablo.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(V), prostorna složenost O(V)

- Najgori slučaj: vremenska složenost O(V), prostorna složenost O(V)

Gore navedene složenosti vrijede za konkretnu implementaciju algoritma (Kôd 2.8), gdje nije cilj obići cijelo stablo već pronaći određeni čvor stabla. Složenost za općenit BFS algoritam gdje obilazimo cijeli graf jednaka je O (V + E).

Dokaz složenosti za najgori slučaj: u najgorem slučaju petlja `while` se izvršava onoliko puta koliko postoji čvorova.

```
function bfs(root, item) {
    const queue = [root];

    while (queue.length) {
      const node = queue.shift();

      if (node.value === item) {
        return node;
      }

      if (node.left) {
        queue.push(node.left);
      }

      if (node.right) {
        queue.push(node.right);
      }

    }


    return null;
  }
```

Kôd 2.8 Izvedba BFS traženja u programskom jeziku JS

2.2.2. DFS

DFS je skraćenica od Depth First Search, tj. Pretraživanje u dubinu. Ovaj algoritam kao i BFS koristi se i za pretraživanje grafova, te se također mora paziti na cikluse vođenjem evidencije o posjećenim čvorovima. Za razliku od BFS-a, DFS prvo pretražuje koliko god

može u dubinu počevši od korijena. To se postiže korištenjem strukture podataka stog. Stog (engl. stack) se koristi za organiziranje elemenata na način da se elementima pristupa po principu "zadnji unutra, prvi van" (LIFO - Last In, First Out). To znači da se posljednji element koji je dodan u stog uklanja prvi, slično kao što se knjige slažu jedna na drugu, a skida se ona koja je zadnja stavljena. Tako se na stog prvo dodaje korijen stabla, a zatim njegova djeca. Čim se obiđe prvo dijete, na stog se dodaju djeca tog djeteta, a pošto stog radi na principu LIFO, ta djeca će biti sljedeća na redu za obilazak. Na taj način se krećemo po stablu što je više moguće u dubinu prije nego krenemo u širinu. Postupak ponavljamo dok ne pronađemo traženi element ili dok ne obiđemo cijelo stablo.

Složenosti:

- Najbolji slučaj: vremenska složenost O(1), prostorna složenost O(1)
- Prosječan slučaj: vremenska složenost O(V), prostorna složenost O(V)
- Najgori slučaj: vremenska složenost O(V), prostorna složenost O(V)

Sve što smo spomenuli vezano za složenosti BFS algoritma vrijedi i za DFS.

```
function dfs(root, item) {
    const stack = [root];

    while (stack.length) {
      const node = stack.pop();

      if (node.value === item) {
        return node;
      }

      if (node.right) {
        stack.push(node.right);
      }

      if (node.left) {
        stack.push(node.left);
      }
    }

    return null;
```

```
}
```

Kôd 2.9 Izvedba DFS traženja u programskom jeziku JS

2.3. Algoritmi pretraživanja grafova

2.3.1. Dijkstra

Dijkstrin algoritam pronalazi najkraći put od jednog vrha grafa do svih ostalih vrhova. Izmislio ga je Nizozemac Edsger W. Dijkstra 1956. godine [7]. Dijkstrin algoritam radi na način da uzastopno odabire najbliži neobiđeni vrh, te računa udaljenost do svih ostalih neobiđenih vrhova. Ulazni skup podataka sastoji se od usmjerenog ili neusmjerenog grafa, te početnog vrha. Dijkstrin algoritam radi samo na grafovima s nenegativnim težinama bridova. U prolasku kroz graf, algoritam gradi listu posjećenih vrhova koja je na početku prazna, te gradi tablicu u koju zapisuje najmanju udaljenost do svakog vrha, te ažurira podatke u toj tablici ako pronađe kraći put do nekog vrha. Na početku je vrijednost udaljenosti za svaki vrh osim početnog jednaka beskonačnosti.

Složenosti:

- vremenska složenost $O(V^2)$
- prostorna složenost O(V)

```
function dijkstra(graph, start, end) {
    const distances = {};
    const parents = {};
    const visited = [];
    let path = [];
    let smallest;

    for (let vertex in graph) {
      if (vertex === start) {
        distances[vertex] = 0;
      } else {
        distances[vertex] = Infinity;
      }

      parents[vertex] = null;
    }
```

```
    while (visited.length !== Object.keys(graph).length) {
      smallest = getSmallestNode(distances, visited);
      visited.push(smallest);

      for (let neighbor in graph[smallest]) {
        let newDistance = distances[smallest] +
graph[smallest][neighbor];

        if (newDistance < distances[neighbor]) {
          distances[neighbor] = newDistance;
          parents[neighbor] = smallest;
        }
      }
    }

    let node = end;

    while (node) {
      path.push(node);
      node = parents[node];
    }

    return path.reverse();
  }

  function getSmallestNode(distances, visited) {
    let smallest = null;

    for (let node in distances) {
      if (smallest === null || distances[node] <
distances[smallest]) {
        if (!visited.includes(node)) {
          smallest = node;
        }
      }
    }

    return smallest;
```

```
}
```

Kôd 2.10 Izvedba Dijkstra algoritma u programskom jeziku JS

2.3.2. Bellman-Ford

Da bismo mogli objasniti složenosti ovog algoritma, prvo moramo objasniti što je povezan graf. Povezani graf je graf u kojemu postoji put između bilo koja dva vrha u grafu [4]. Ovaj algoritam je praktičan kada imamo graf s negativnim vrijednostima težina bridova jer u tom slučaju ne možemo koristiti Dijkstrin algoritam. Bellman-Ford algoritam možemo koristiti i za grafove s nenegativnim težinama bridova, međutim tada je bolje koristiti Dijkstrin algoritam jer je brži. Ovaj algoritam provjerava za svaki brid može li se doći do nekog vrha na brži način koristeći taj brid. Tu provjeru svih bridova ponavlja V – 1 puta gdje je V broj vrhova.

Složenosti za slučaj kada je graf povezan:

- Najbolji slučaj: vremenska složenost O(E), prostorna složenost O(V)
- Prosječan slučaj: vremenska složenost O(V*E), prostorna složenost O(V)
- Najgori slučaj: vremenska složenost O(V*E), prostorna složenost O(V)

Složenosti kada graf nije povezan:

- Vremenska složenost: $O(E*V^2)$
- Prostorna složenost: O(V)

```
function bellmanFord(graph, start, end) {
    const distances = {};
    const parents = {};
    const visited = [];
    let path = [];
    let smallest;

    for (let vertex in graph) {
      if (vertex === start) {
        distances[vertex] = 0;
      } else {
        distances[vertex] = Infinity;
      }
```

```
      parents[vertex] = null;
    }

    for (let i = 0; i < Object.keys(graph).length - 1; i++) {
      for (let vertex in graph) {
        for (let neighbor in graph[vertex]) {
          let newDistance = distances[vertex] +
graph[vertex][neighbor];

          if (newDistance < distances[neighbor]) {
            distances[neighbor] = newDistance;
            parents[neighbor] = vertex;
          }
        }
      }
    }

    let node = end;

    while (node) {
      path.push(node);
      node = parents[node];
    }

    return path.reverse();
  }
```

Kôd 2.11 Izvedba Bellman-Ford algoritma u programskom jeziku JS

3. Vizualizacija algoritama pretraživanja

Kao što je u uvodu napomenuto, vizualizacija algoritama je ključna za lakše razumijevanje kako svaki pojedini algoritam radi. Elemente niza, stabala ili grafova je potrebno na neki grafički način prikazati uz mogućnost naglašavanja svakog pojedinog elementa unutar niza, stabla ili grafa. Dodatno je potrebno moći naglasiti liniju koda koja se trenutno izvršava kako bismo znali u kojoj je trenutno algoritam fazi. Veoma je praktično da su izvršni kod algoritma i grafička vizualizacija prikazani jedno uz drugo da se može istovremeno pratiti izvršavanje koda i promatrati kakav utjecaj to izvršavanje ima na grafičke elemente. Korisno bi bilo i moći prikazati više algoritama odjednom kako istovremeno rade da bi ih se lakše moglo usporediti. Sada nam je već jasno da mnoštvo toga mora biti istovremeno vidljivo na ekranu što će biti jedan od važnijih faktora prilikom odabira tehnologija za razvoj aplikacije.

3.1. Odabir platforme

Prvi korak u razmišljanju bio je odabir platforme za koju razvijamo aplikaciju. U razmatranju su bile tri opcije:

1. Mobilna aplikacija
2. Desktop aplikacija
3. Web aplikacija

Mobilna aplikacija je prva ispala iz razmatranja jer jednostavno nije praktično prikazati toliko puno informacija na tako malenom ekranu. Desktop aplikacija je također bila odbačena kao opcija jer bi desktop aplikacijom ograničili ciljanu skupinu samo na ljude koji koriste stolna ili prijenosna računala, a cilj je da aplikacija bude što dostupnija, uključujući mobilne uređaje. Iako smo zaključili da nije praktično prikazati toliko puno informacija na malenom ekranu, i dalje ne želimo potpuno onemogućiti korisnicima pametnih mobitela da koriste aplikaciju. Web aplikacija tu dolazi kao rješenje koje omogućuje svima da koriste aplikaciju, te da sami odaberu hoće li aplikaciji pristupiti pomoću desktop računala gdje će sve biti pregledno, ili će aplikaciju koristiti preko mobilnih telefona svjesno žrtvujući preglednost. Još jedna prednost web aplikacije je činjenica da se ne mora instalirati na uređaj, već joj se može pristupiti preko bilo kojeg web preglednika kojeg ljudi već imaju instaliranog na svojim uređajima.

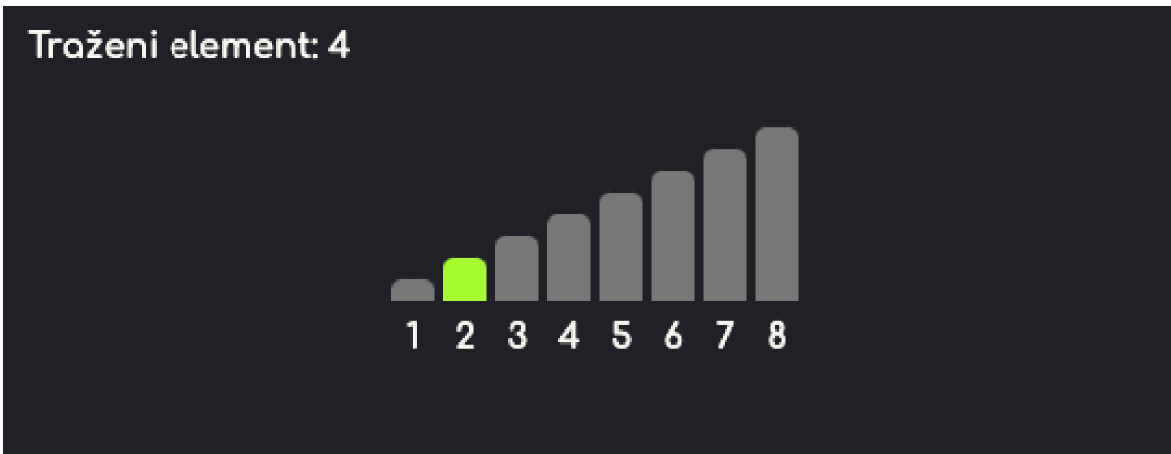
3.2. Odabir tehnologija

Nakon što je odabrana web platforma, slijedi odabir tehnologija koje će se koristiti u skladu s uslugama koje će aplikacija pružati. Za razvoj sučelja odabran je radni okvir React iz mnogo razloga. Glavni razlog je popularnost React radnog okvira. Razvio ga je Facebook 2013. godine, a do danas je stekao ogromnu popularnost. React se koristi za razvoj jednostrukih web aplikacija (SPA) gdje se podaci mogu mijenjati bez ponovnog učitavanja cijele stranice. React je daleko najpopularniji radni okvir za razvoj sučelja te samim time ima najveću podršku zajednice i najviše odgovora na Internetu ako dođe do kakvih nepoznanica u izradi. React omogućuje izradu složenih korisničkih sučelja putem komponentnog pristupa, gdje se korisničko sučelje dijeli na manje cjeline koje se mogu jednostavno ponovno upotrebljavati. React koristi virtualni DOM za efikasno upravljanje promjenama u korisničkom sučelju, što poboljšava performanse aplikacija. Još jedna bitna značajka je mogućnost renderiranja aplikacija na strani poslužitelja što omogućuje odlične performanse i optimizaciju za tražilice.

3.3. Komponente sučelja

3.3.1. Vizualizacija polja

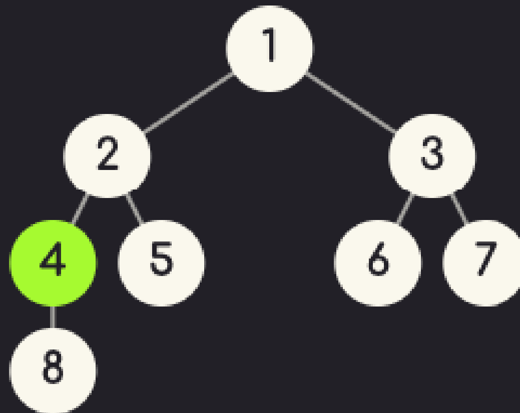
Vizualizirati polje kao strukturu podataka je poprilično jednostavno. To postižemo korištenjem kvadratića unutar kojih su brojevi koji označavaju vrijednosti pohranjene u polju. Da bi bilo lakše primijetiti vrijednosti, odlučujemo prilagoditi visinu svakog pravokutnika ovisno o vrijednosti koju on predstavlja (Sl. 3.1). Vrijednost koja je trenutno aktivna u algoritmu koji se izvršava naglašena je različitom bojom od ostalih vrijednosti.



Sl. 3.1 Vizualizacija polja

3.3.2. Vizualizacija stabla

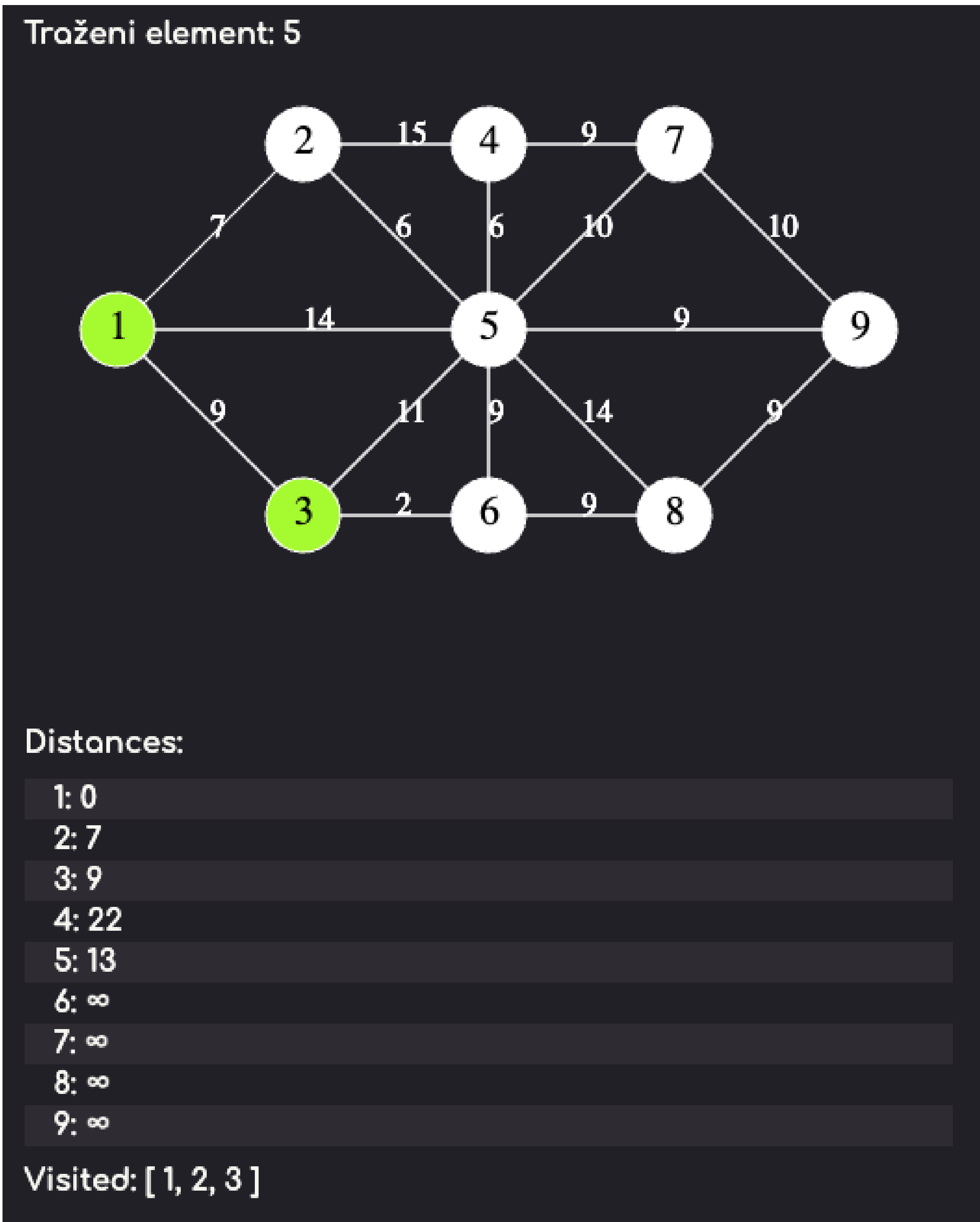
Za vizualizaciju stabla inicijalno planiramo koristiti neku od gotovih knjižica za crtanje stabala, međutim sve isprobane na kraju ne daju željene rezultate. Nakon nekoliko neuspješnih pokušaja, koristimo Canvas aplikacijsko programsko sučelje koje omogućuje crtanje grafika pomoću JavaScripta i HTML-a. Canvas je HTML element koji omogućuje skriptnom jeziku, poput JavaScript-a, crtanje grafike izravno na web stranicu. Prvi put je uveden u Appleov Safari preglednik 2004. godine. Može se koristiti za izradu grafova, animacija, igara i drugih vizualnih elemenata. W3C je uključio Canvas u HTML5 specifikaciju 2010. godine, što je rezultiralo masovnom prihvaćenošću među preglednicima. Canvas se može koristiti za izradu složenih animacija, uključujući igre i interaktivne vizualne efekte. Taj pristup iziskuje malo više vremena za razvoj rješenja, ali zato pruža maksimalnu razinu prilagodljivosti našim potrebama. Element koji je trenutno aktivan u algoritmu koji se izvršava prikazan je različitom bojom od ostalih čvorova (Sl. 3.2).



Sl. 3.2 Vizualizacija stabla

3.3.3. Vizualizacija grafa

Za vizualizaciju grafa koristimo Canvas aplikacijsko programsko sučelje kao i za vizualizaciju stabla. Međutim, za vizualizaciju onoga što se događa kod izvršavanja algoritama za pronalazak najkraćeg puta, potrebno je prikazati još neke dodatne informacije kao npr. listu obiđenih vrhova ili određene interne tablice koje algoritam gradi (Sl. 3.3). Aktivni vrhovi su prikazani različitom bojom od ostalih vrhova.



Sl. 3.3 Vizualizacija grafa

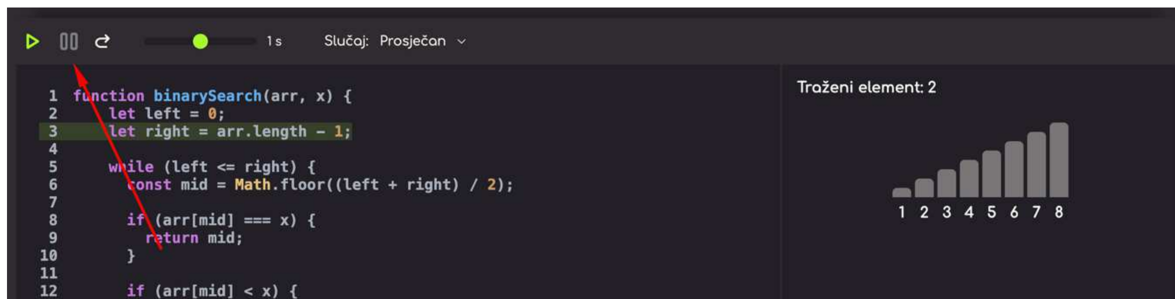
3.4. Značajke aplikacije

Neke od značajki aplikacije već smo spomenuli, a ovdje ćemo ih nabrojati sve. Osnovna značajka aplikacije je izvršavanje izvornog koda algoritma, te grafički prikaz prolaska algoritma kroz određenu strukturu podataka. Dodatne značajke su ubrzavanje ili usporavanje

izvršavanja algoritma, pauziranje izvršavanja i izvršavanje korak po korak klikom na gumb za sljedeći korak, nastavak automatskog izvršavanja nakon pauziranja, istovremeno izvršavanje i vizualizacija dva različita algoritma za direktnu usporedbu, označavanje linije koda na kojoj se algoritam trenutno nalazi, te promjena slučaja algoritma (najgori, prosječan ili najbolji slučaj).

3.4.1. Kontrole izvršavanja

Komponenta za kontrolu izvršavanja omogućuje pokretanje izvršavanja, pauziranje izvršavanja, izvršavanje korak po korak klikom na gumb, te promjenu brzine izvršavanja (Sl. 3.4).



Sl. 3.4 Komponenta za upravljanje izvršavanjem algoritma

Jedan od težih dijelova implementacije bio je omogućavanje korisniku izvršavanje korak po korak klikom na gumb. To postižemo korištenjem JavaScript Promise-a (Kôd 3.1).

```
import React from "react";

const pause = (ms: number) => new Promise((resolve) =>
setTimeout(resolve, ms));

export const usePausable = () => {
  const shouldPause = React.useRef(false);
  const resolve = React.useRef<any>();

  const nextTick = () =>
    new Promise((res) => {
      resolve.current = res;
      setTimeout(() => {
        if (!shouldPause.current) {
          resolve.current();
        }
      });
```

```
  });

  const handlePlay = () => {
    if (shouldPause.current) {
      shouldPause.current = false;
      resolve.current();
    }
  };

  const handlePause = () => {
    shouldPause.current = true;
  };

  const handleNext = () => {
    if (resolve.current) {
      resolve.current();
    }
  };

  const pauseAndWait = async (ms: number) => {
    await pause(ms);
    await nextTick();
  };

  return {
      pauseAndWait,
      handlePlay,
      handlePause,
      handleNext
  };
};
```

Kôd 3.1 React hook koji omogućuje pauziranje izvođenja i izvršavanje korak po korak

Zaključak

Algoritmi traženja neizostavan su dio gotovo svakog softverskog rješenja današnjice. Ključni su za efikasno pretraživanje podataka u raznim aplikacijama, od najjednostavnijih baza podataka do jako složenih sustava umjetne inteligencije. Ne postoji savršen algoritam traženja koji je najbolji za svaku vrstu ulaznih podataka, već su različiti algoritmi preferirani ovisno o vrsti ulaznih podataka. Ako je inženjeru cilj maksimalno optimizirati brzinu izvođenja softverskog koda, potrebno je dobro poznavati karakteristike ulaznih podataka, te dobro poznavati različite algoritme traženja i njihova ponašanja. Ponekad nije jednostavno razumjeti kako neki algoritam zapravo radi samo gledajući u izvorni kod algoritma. Čak i najiskusniji inženjeri mogu imati poteškoća s praćenjem složenih petlji i rekurzivnih poziva koji su vrlo često prisutni u naprednim algoritmima traženja. Tu nam u pomoć dolazi vizualizacija izvođenja algoritma koja nam prikazuje prolazak algoritma kroz neku strukturu podataka, na primjer niz, stablo ili graf. Bitno je prikazati izvođenje algoritma s različitim ulaznim podacima koji prikazuju najbolji, prosječan i najgori slučaj karakterističan za taj algoritam. Za izradu takvog alata, odabrana je web platforma zbog dostupnosti na svim vrstama uređaja i jednostavnosti za korištenje bez potrebe instalacije. Web aplikacije su jako praktične jer omogućuju korisnicima da pristupe aplikaciji s bilo kojeg mjesta putem internetskog preglednika, neovisno o operacijskom sustavu ili specifikacijama uređaja. Dodatno, današnji web preglednici podržavaju napredne vizualizacijske biblioteke i tehnologije koje omogućuju izradu složenih i interaktivnih vizualizacija. U konačnici, cilj je stvoriti alat koji će biti koristan ne samo inženjerima, već i studentima, istraživačima i svima koji žele bolje razumjeti algoritme traženja i njihovu primjenu.

Literatura

[1] GeeksforGeeks, *Searching Algorithms*, (2024, travanj). Poveznica: www.geeksforgeeks.org/searching-algorithms; pristupljeno 20. svibnja 2024.

[2] Entrustech Inc, *Harnessing the Power of Exponential Search Algorithm*, Medium, (2023, srpanj). Poveznica: https://medium.com/@entrustech/harnessing-the-power-of-exponential-search-algorithm-1c2bc9f882df; pristupljeno 21. svibnja 2024.

[3] Fibonaccijev niz. *Hrvatska enciklopedija*, *mrežno izdanje.* Leksikografski zavod Miroslav Krleža, 2013. – 2024. Poveznica: https://www.enciklopedija.hr/clanak/fibonaccijev-niz; pristupljeno 21. svibnja 2024.

[4] mrežni graf. *Hrvatska enciklopedija*, *mrežno izdanje.* Leksikografski zavod Miroslav Krleža, 2013. – 2024. Poveznica: https://enciklopedija.hr/clanak/mrezni-graf; pristupljeno 24. svibnja 2024.

[5] Kovačević, D., Krnić, M., Nakić, A., Osven Pavčević, M. *Diskretna matematika 1*, FER, Zagreb, 2020.

[6] Vuković, M. *Složenost algoritama*, PMF, Zagreb, 2019.

[7] GeeksforGeeks, *DSA Dijkstra's Algorithm*. Poveznica: www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php; pristupljeno 11. lipnja 2024.

[8]Urševskis, R., Pocola T.O. *The evolution of search algorithms over time,* Delft University of Technology, Delft

Sažetak

Vizualizacija algoritama pretraživanja

Dino Držaić

U ovom radu objašnjava se važnost poznavanja algoritama traženja te se radi pregled onih najpoznatijih. Prikazuje se analiza njihove vremenske i prostorne složenosti te objašnjenje na koji način pojedini algoritam radi. Iznose se njihove prednosti i nedostaci, te ograničenja ako postoje. Objašnjava se i važnost vizualizacije algoritma za bolje i lakše razumijevanje načina rada pojedinog algoritma. Prezentira se realizacija interaktivnog rješenja za vizualizaciju algoritama traženja. Za izradu rješenja koriste se web tehnologije zbog jednostavnosti korištenja, mogućnosti pristupa s bilo koje lokacije te podržavanja velikog spektra uređaja. Rad zaključuje da vizualizacija nije korisna samo inženjerima, već i studentima i istraživačima koji žele bolje upoznati načine rada algoritama traženja.

Ključne riječi: algoritmi pretraživanja; vizualizacija; notacija veliko O

Summary

Search algorithm visualization

Dino Držaić

This paper explains the importance of understanding search algorithms and provides an overview of the most well-known ones. It presents an analysis of their time and space complexity, along with an explanation of how each algorithm works. The advantages and disadvantages of these algorithms are discussed, as well as any limitations they may have. The importance of visualizing algorithms for a better and easier understanding of their functionality is also explained. An interactive solution for visualizing search algorithms is presented. Web technologies are used for developing this solution due to their ease of use, the ability to access them from any location, and their support for a wide range of devices. The paper concludes that visualization is beneficial not only for engineers, but also for students and researchers who wish to better understand the workings of search algorithms.

Keywords: search algorithms; visualization; big O notation

Skraćenice

HTML	*HyperText Markup Language*	prezentacijski jezik za izradu web stranica
JS	*JavaScript*	skriptni programski jezik