UNIVERZITET U SARAJEVU
ELEKTROTEHNIČKI FAKULTET
SARAJEVO - ODSJEK ZA
RAČUNARSTVO I INFORMATIKU

Eksperimentalna analiza vremenske kompleksnosti Dijkstra algoritma

ZAVRŠNI RAD
-PRVI CIKLUS STUDIJA-

Mentor:
Red. prof. dr. Haris Šupić

Kandidat:
Tarik Velić

Sarajevo, Juli 2024.

Univerzitet u Sarajevu
Elektrotehnički fakultet
Odsjek za računarsvo i informatiku

Izjava o autentičnosti radova

ZAVRŠNI RAD

-PRVI CIKLUS STUDIJA-

Ime i prezime: Tarik Velić

Naslov rada: Eksperimentalna analiza vremenske kompleksnosti Dijkstra algoritma

Vrsta rada: Završni rad prvog ciklusa studija

Broj stranica: X

Potvrđujem:

- da sam pročitao dokumente koji se odnose na plagijarizam, kako je to definirano Statutom Univerziteta u Sarajevu, Etičkim kodeksom Univerziteta u Sarajevu i pravilima studiranja koja se odnose na I i II ciklus studija, integrirani studijski program I i II ciklusa i III ciklus studija na Univerzitetu u Sarajevu, kao i uputama o plagijarizmu navedenim na web stranici Univerziteta u Sarajevu;
- da sam svjestan univerzitetskih disciplinskih pravila koja se tiču plagijarizma;
- da je rad koji predajem potpuno moj, samostalni rad, osim u dijelovima gdje je to naznačeno;
- da rad nije predat, u cjelini ili djelimično, za stjecanje zvanja na Univerzitetu u Sarajevu ili nekoj drugoj visokoškolskoj ustanovi;
- da sam jasno naznačio prisustvo citiranog ili parafraziranog materijala i da sam se referirao na sve izvore;
- da sam dosljedno naveo korištene i citirane izvore ili bibliografiju po nekom od preporučenih stilova citiranja, sa navođenjem potpune reference koja obuhvata potpuni bibliografski opis korištenog i citiranog izvora;
- da sam odgovarajuće naznačio svaku pomoć koju sam dobio pored pomoći mentora i akademskih tutora/ica;

Sarajevo, Juli 2024

Potpis:_____

Izjava o autentičnosti je preuzeta iz sljedećih dokumenata: *Izjava autora koju koristi Elektrotehnički fakultet Sarajevo, Izjava o autentičnosti završnog rada Centra za interdisciplinarne studije – master studij – „Evropske studije“, Izjava o plagijarizmu koju koristi Fakultet političkih nauka u Sarajevu.*

Postavka završnog rada na I ciklusu studija

Tema: Eksperimentalna analiza vremenske kompleknosti Dijkstra algoritma

Ciljevi rada:

- Generalni opis Dijkstra algoritma i problema najkraćeg puta u grafu.
- Izučiti vremensku složenost Dijkstra algoritma u ovisnosti od implementacije.
- Implementacija Dijkstra algoritma na više načina.
- Eksperimentalna analiza brzine izvršavanja algoritma u ovisnosti od parametara

Opis:

Na početku teorijskog dijela ćemo se fokusirati na generalni princip rada Dijkstra algoritma, gdje ćemo iznijeti njegove prednosti i mane u odnosu na druge algoritme za pronalazak najkraćeg puta. S obzirom da radimo na problemu teorije grafova objasnit ćemo i nekoliko pojmova vezanih za istu kako bismo čitaocima približili terminologiju koja će se prožimati kroz čitav rad. Dodatno u teoretskom dijelu treba izučiti i vremensku kompleksnost implementacija Dijkstra algoritma uz dodatna pojašnjenja za ključne strukture za ubrzavanje izvršavanja. U implementacijskom dijelu će biti implementiran algoritam u programskom jeziku C++, te ćemo mjeriti vrijeme izvršavanja algoritma za različite verzije u ovisnosti od više parametara.

Početna literatura:

1. H
2. J
3. J
4. J
5. G

Mentor:

Red. prof. dr. Haris Šupić

_____

Sažetak

Na početku ovog rada govorimo generalno o fundamentalnim principima Dijkstra algoritma, predstavljajući njegov pseudokod i ključne korake pri sprovođenju algoritma na papiru. Također spominjemo realne probleme koji se rješavaju pomoću teorije grafova u kojoj se nerijetko treba pronaći najkraći put od jednog čvora ka drugom. Istraživajući područja kojima se bave inženjeri često ćemo naići na povezane strukture koje je najpogodnije predstavljati pomoću grafova. Također, predstavili smo dokaz Dijkstra algoritma kako bi se uvjerili u njegovu ispravnost, te smo uradili jedan primjer da bismo pokazali kako se algoritam izvodi na papiru. Situacija je puno interesantnija kada se govori o implementaciji algoritma na računaru s obzirom da moramo uzeti u obzir brzinu izvršavanja koja je jako bitan faktor bilo kojeg računarskog algoritma. Analizirali smo razne načine implementacije Dijkstra algoritma kako bismo se odlučili koja je najbolja, odnosno najbrža, za implementaciju na računaru. Pored detaljne analize vremenske kompleksnosti različitih implementacija, spomenuli smo i prostornu kompleksnost kako bismo stekli osjećaj i za tu vrstu kompleksnosti kada je u pitanju Dijkstra algoritam. Detaljno smo obradili i Fibonacci gomile - posebnu implementaciju prioritetnog reda. Fibonacci gomile predstavljaju najbolji način implementacije Dijkstra algoritma zbog njihove fleksibilnosti pri operacijama koje su ključne za ovaj algoritam, stoga je bitno da ih dobro razumijemo i objasnimo način na koji one djeluju zajedno sa formulama algoritma. Pored toga spomenuli smo i ostale načine implementacije : preko matrice susjedstva, preko liste susjedstva bez prioritetnog reda, preko liste susjedstva sa prioritetnim redom i ostale. Također generalizovali smo formulu za vremensku kompleksnost ukoliko koristimo bilo koju od implementacija prioritetnog reda (binarne gomile, binarno stablo pretraživanja, gomile sa prioritetog minimuma, Fibonacci gomile i ostale).

Što se tiče implementacijskog dijela, implementirali smo dva različita koda za Dijkstra algoritam, te smo iz literature preuzeli dva koda, stoga smo na njih stavili reference na njihov izvor, tako da smo imali četiri različita koda algoritma od kojih svaki ima različitu vremensku kompleksnost što nam daje odličnu podlogu za eksperimentalnu analizu. Analiza se temelji na tome da po volji povećavamo broj čvorova grafa i na osnovu parametra određujemo gustinu grafa. Uzimajući u obzir dosta slučajeva sa ulaznim podacima različitih karakteristika i proslijeđivajući te podatke različitim načinima implementacija, osigurali smo se da naš uzorak bude dovoljno velik i sveobuhvatan. Također randomizirali smo i težine grana postavljajući ih na vrijednosti željenih opsega što čini naš model analize još pouzdanijim. Nakon izvršene analize rezultate smo prikazali na graficima kako bi se stekao uvid u ponašanja različitih implementacija u odnosu na zadane parametre. Pored gustih i rijetkih grafova, te grafova sa različitim težinama grana napravili smo i paralelu između usmjerenih i neusmjerenih grafova. Na kraju naše analize objasnili smo dobivene rezultate te iznijeli argumente zašto su rezultati očekivani (osim u nekim graničnim slučajevima za koje opet ima valjano objašnjenje da ih opravda). U zaključku ovog završnog rada obrazložili smo zašto i kada je neka implementacija najpovoljnija za sveobuhvatne performanse algoritma uzimajući u obzir i vremensku i prostornu kompleksnost, te kompleksnost sa gledišta upotrebe memorije.

Abstract

At the beginning of this research, we discuss the fundamental principles of the Dijkstra algorithm, presenting its pseudocode and the key steps for performing the algorithm on paper. We also mention real problems solved by using the graph theory, in which often the shorthest path from one node to another needs to be found. In exploring the fields in which engineers operate, we frequently encounter connected structures that are most conveniently represented through graphs. Additionally, we have presented the proof of the Dijkstra algorithm in order to verify its correctness, and we have done one example to demonstrate how the algorithm is performed on paper. The situation is much more intriguing when discussing the implementation of the algorithm on the computer since we have to consider the running time which is an important factor of any computer algorithm. We have analyzed various ways of implementing the Dijkstra algorithm in order for us to determine which is the best, that is, the fastest, for the implementation on a computer. In addition to a detailed analysis of the time complexity of different implementations, we have also mentioned space complexity in order to get a feel for that type of complexity when it comes to Dijkstra algorithm. We have also covered the Fibonacci heaps in detail – a special implementation of the priority queue. Fibonacci heaps represent the most effective way of implementing the Dijsktra algorithm because of their flexibility in operations that are key to this algorithm. Therefore, it is important for us to thoroughly understand them and explain how they operate in conjunction with the algorithm's formulas. In addition, we have mentioned the other methods of the implementation: via the adjacency matrix, via the adjacency list without priority queue, via the adjacency list with priority queue and others. We have also generalized the formula for time complexity when using any of the priority queue implementations (binary heaps, binary search tree, minimum priority heaps, Fibonacci heaps and others).

Regarding the implementation part, we have implemented two different codes for Dijkstra algorithm, and we have also adopted two codes from the literature, providing the reference to the original source, resulting in four different algorithm codes, each with different time complexity providing an excellent basis for experimental analysis. The analysis was based on increasing the number of graph nodes at will and determining the graph density based on parameters. Considering numerous cases with input data of different characteristics and passing that data through different implementation methods, we have ensured that our sample is sufficiently large and comprehensive. Additionally, we have randomized the edge weights by setting them within desired ranges, making our analysis model even more reliable. After the analysis, we presented the results in graphs in order to gain insight into the behaviour of different implementations concerning the given parameters. In addition to dense and sparse graphs, and graphs with different edge weights, we have also made a comparison between directed and undirected graphs. At the end of our analysis, we have explained the obtained results and provided arguments for why the results were expected (expect in some limited cases for which there is still a valid explanation to justify them). In the conclusion of this research, we justified why and when a certain implementation is most favorable for comprehensive algorithm performances, taking into consideration both time and space complexity, as well as memory usage.

1. Uvod

U okviru inžinjeringa i računarskih nauka, suočavamo se s raznovrsnim problemima koji zahtijevaju efikasno i efektivno rješenje. Jedno od ključnih područja koje nude bogat spektar alata za rješavanje ovakvih problema jeste teorija grafova. Grafovi nam omogućavaju da modeliramo složene sisteme i odnose među njihovim komponentama, pružajući nam moćan način za analizu i optimizaciju. Jedan od fundamentalnih problema teorije grafova je pronalazak najkraćeg puta između dva čvora kako u netežinskim tako i u težinskim grafovima. U netežinskim grafovima pod najkraćim putem podrazumjevamo najmanji broj grana kroz koje moramo proći kako bismo došli do odredišta. Za tu svrhu najpogodniji je algoritam pretraživanja u širinu (BFS, Breadth-First-Search). Razlog za to je što BFS sistematski istražuje sve čvorove na jednoj dubini prije nego što pređe na čvorove na sljedećoj dubini. To osigurava da kada prvi put dođete do odredišnog čvora, put koji ste pronašli je zaista najkraći put od početnog do odredišnog čvora u okviru netežinskih grafova. U ovom radu mi ćemo se baviti pronalaskom najkraćeg puta u težinskim grafovima.

Među mnogobrojnim algoritmima za pronalazak najkraćeg puta, Dijkstra algoritam se ističe svojom efikasnošću i jednostavnošću. Njega je prvi put objavio nizozemski matematičar i istraživač Edsger Wybe Dijkstra 1959. godine po kojem je ovaj algoritam i dobio ime. U ovom radu fokusirat ćemo se na eksperimentalnu analizu Dijkstra algoritma s ciljem da detaljno razumijemo kako algoritam funkcioniše, kao i da istražimo njegove performanse u različitim uslovima i na različitim tipovima grafova. Da bismo postigli ovaj cilj, prvo ćemo se upoznati sa osnovnim principima algoritma, uključujući njegov način rada, njegovu strukturu i matematičku osnovu koja dokazuje njegovu ispravnost. Neizostavan dio analize je i razumijevanje pojmova kao što su vremenska kompleksnost, jer nam ona predstavlja ključni faktor u određivanju kvalitete nekog algoritma. Čitatelje ćemo upoznati sa osnovnim pojmovima koji se tiču vremenske kompleksnosti, te ćemo spomenuti i glavnu ideju prostorne kompleksnosti.

Dijkstra algoritam nalazi primjenu u mnogim naučnim oblastima, krenuvši od računarskih mreža, operacionih sistema, preko geoinformatike i urbanog planiranja, vještačke inteligencije i optimizacije algoritama. Ova široka primjenljivost je dokaz njegove snage i univerzalnosti, kao i njegove sposobnosti da pruži efikasna rješenja za naizgled kompleksne probleme koji se mogu javiti u različitim sferama nauke. Kroz ovaj rad, težimo da pružimo sveobuhvatan pogled na Dijkstra algoritam, počevši od njegove historije i teorijskih osnova, preko detaljne analize i njegove strukture i performansi, do praktične primjene i eksperimentalne verifikacije njegove efikasnosti. Na taj način, čitaoci će steći duboko razumijevanje važnosti Dijsktra algoritma u teoriji grafova i računarskim naukama, kao i praktične vještine potrebne za njegovu implementaciju i optimizaciju u različitim aplikacijama. Također, čitaoci će moći uočiti očiglednu razliku između performansi algoritama koji imaju različite vremenske kompleksnosti, čime ćemo steći osjećaj za brzine algoritama u ovisnosti od reda njihove kompleksnosti, broja čvorova i broja grana grafa.

1.1 Motivacija

Motivacija za dubinsko istraživanje i analizu Dijkstra algoritma proizilazi iz njegove fundamentalne uloge u teoriji grafova i široke primjene u različitim tehničkim i inženjerskim disciplinama. Razumijevanje rada algoritma, kao i njegovih performansi u praktičnim aplikacijama, krucijalno je za rješavanje kompleksnih problema koji zahtijevaju efikasno pronalaženje najkraćeg puta u grafu. Pored teorijskog značaja, motivacija za ovaj rad proizilazi i iz praktične potrebe da se poboljšaju algoritmi koji leže u osnovi brojnih aplikacija koje svakodnevno koristimo. S obzirom da svijet postaje sve digitalizovaniji i povezaniji, efikasnost u obradi i analizi podataka dobiva sve više na značaju. Navest ćemo još neke od mnogobrojnih upotreba Dijkstra algoritma, da bismo stvorili sliku o tome koliko je bitno da ovaj algoritam radi na optimalan način sa aspekta brzine izvršavanja. Neke od konkretnih primjena su: mrežno rutiranje i OSPF protokol – u mrežama koje su povezane ruterima jako je bitno odrediti najkraći put od nekog ka ostalim ruterima, planiranje puta u navigacionim sistemima – GPS sistemi koriste ovaj algoritam, optimizacija transportnih mreža – upravljanje lancima snadbijevanja, robotika – navigacija robota, video igre i virtualne realnosti – navigacija likova i druge brojne primjene.

Dodatna motivacija leži u želji da se prevaziđu ograničenja postojećih implementacija Dijkstra algoritma. Iako je algoritam izuzetno efikasan, postoji prostor za optimizaciju, posebno u kontekstu specifičnih aplikacija i okruženja. Istraživanje različitih implementacija i struktura podataka, poput Fibonacci gomila, može pružiti uvide u to kako dalje poboljšati performanse algoritma čineći ga još prilagodljivijim i efikasnijim. Ovaj rad predstavlja priliku za istraživače i studente da doprinesu zajednici kroz eksperimentalnu analizu i optimizaciju jednog od najvažnijih algoritama u računarskim mrežama.

Na kraju, motivaciju za ovaj rad sam pronašao i u ljubavi prema matematici i algoritmima, te želji za proširivanjem znanja. Također, od velike je koristi implementirati neki algoritam na više različitih načina kako bi proširili spektar naših ideja u generalnom rješavanju zadataka. Dodatno, proći ćemo kroz više kodova, tako da ćemo se upoznati sa nekim novim tehnikama i novim strukturama podataka, što će nam predstavljati olakšanje za razumijevanje i implementaciju kodova u budućnosti.

1.2 Ciljevi rada

Pored već navedenih ciljeva rada u postavci rada, čitaoce želimo podstaknuti na „inspektorsko“ razmišljanje u smislu detaljnije analize bilo kojeg algoritma kojeg implementiraju. Obično je slučaj da u realnim problemima sa kojima se susrećemo nastojimo da samo implementiramo algoritam ne uzimajući u obzir njegove performanse i kompleksnost. Cilj ovog rada je naglasiti važnost performansi algoritma posmatrajući stvarne podatke, dobivene eksperimentima, za svaku implementaciju algoritma te objašljavajući zašto su podaci takvi kakvi jesu. Konkretno, ovaj radi ima za cilj da:

- detaljno razloži teorijske osnove
- prouči različite implementacije
- eksperimentalno evaluira performanse implementacija
- ilustruje praktične primjene
- identifikuje prednosti i ograničenja svake od implementacija
- predlaže pravce za buduća istraživanja
- podstakne čitaoce na dublje razmišljanje o nefunkcionalnim zahtjevima aplikacija

Sveukupno, ovaj završni rad teži da doprinese akademskoj i profesionalnoj zajednici, pružajući duboko i praktično razumijevanje Dijkstra algoritma, teorije grafova, vremenske kompleksnosti i struktura podataka, čime se olakšava njegova dalja primjena i razvoj u širokom spektru tehničkih i inženjerskih disciplina.

1.3 Struktura rada

Struktura rada se sastoji od četiri glavna dijela, koja će površno biti objašnjena u ovom poglavlju, a detaljnija obrada svakog od dijelova slijedi u nastavku.

Uvodni dio rada predstavlja osnove i objašnjenje problema kojim se bavimo, uključujući motivaciju za rad, ciljeve rada i strukturu rada. Pomoću njega čitaoc može steći uvid u sami problem tako da mu bude lakše pratiti naredna izlaganja.

U **teorijskom dijelu**, govorimo konkretno o Dijkstra algoritmu, njegovoj matematičkoj definiciji, pseudokodu i objašnjenju toka algoritma. Također, predstavit ćemo njegov dokaz, te ćemo uraditi jedan primjer rješavanja problema najkraćeg puta pomoću Dijkstra algoritma, na način pogodan za rad na papiru. Pored toga analizirat ćemo vremensku kompleksnost različitih implementacija algoritma i posebno se osvrnuti na Fibonacci gomile. Priložit ćemo implementaciju Dijkstra algoritma preko Fibonacci gomila da bismo čitaocima razjasnili zašto su baš one najbolji izbor za implementaciju. Predstavit ćemo još jedan kod iz literature u ovom dijelu da bismo efikasnije usporedili vremenske kompleksnosti i dodatno objasnili zbog kojih operacija u implementaciji algoritma Fibonacci gomile imaju prednost u odnosu na druge strukture podatak.

Praktični dio rada se svodi na samostalnu implementaciju algoritma na dva načina, te eksperimentalnu analizu četiri algoritma sa različitim vremenskim kompleksnostima. Rezultate istraživanja ćemo predstavljati na graficima. Istraživanje će se vršiti na osnovu parametara i imat ćemo po jedan grafik za svaku kombinaciju ulaza. Konkretni detalji implementacije će biti prikazani u nastavku, te će čitaocima biti puno jasnije o čemu se radi, jer ćemo detaljno objasniti svaki korak i svrhu eksperimenta.

U **zaključku** rada ćemo proanalizirati rezultate dobivene iz eksperimenata, te ćemo ih argumentovano opravdati, pored toga spomenut ćemo i neke varijante Dijkstra algoritma sa određenim ograničenjima te ćemo vidjeti kako ta ograničenja mogu uticati na vremensku kompleksnost. Na kraju, dat ćemo konačan sud o tome koja je implementacija najbolja za specifičnu upotrebu algoritma koja odgovara našim zahtjevima.

2. Problem najkraćeg puta u grafu i Dijkstra algoritam

U ovom poglavlju fokusirat ćemo se na jedan od osnovnih problema teorije grafova – pronalazak najkraćeg puta. U tu svrhu razvijeni su mnogi algoritmi od kojih svaki ima svoje prednosti i mane. Kroz ovaj rad mi ćemo se fokusirati na objašnjenje i eksperimentalnu analizu Dijkstra algoritma. On kombinira ideje Bellman-Fordovog i Dantzigovog algortima - koji se danas više ne koristi. Razlog zbog kojeg se Dantzigov algoritam ne koristi je to što posjeduje ista ograničenja na nenegativnost grana kao i Dijkstra algoritam, a sporiji je do *n* puta, gdje *n* predstavlja broj čvorova u grafu. Za potrebe pronalaska najkraćeg puta bez ikakvih ograničenja koristi se Bellman-Fordov algoritam, koji po cijenu brzine izvođenja eliminira potrebu za ograničenjem na nenegativnost grana. S obzirom da vrijedi *max(-f) = - min(f),* Bellman-Fordov algoritam omogućava i pronalazak najdužeg puta od jednog čvora ka drugom. Iako je Bellman-Fordov algoritam „jači“ od Dijkstra algoritma u smislu mogućnosti, dosta je sporiji, te s obzirom da se u realnim problemima rijetko pojavljuju negativne težine grana, možemo reći da je Dijkstra algoritam primarni algoritam za pronalazak najkraćeg puta u savremenim aplikacijama.

Prema principu koji slijedi ovaj algoritam, ukoliko su sve dužine grana u grafu nenegativne, tada ukoliko je *(i,j)* najkraća od svih grana koje vode iz čvora *i,* ta grana ujedno mora biti i najkraći put iz čvora *i* u čvor *j*. Posljedica ovog principa je da će postojati čvorovi za koje se vrijednosti f_i u daljim iteracijama neće mjenjati, te ćemo na osnovu njih iterativno određivati najkraći put u grafu. Vrijednosti f_i se nazivaju potencijali čvorova, te predstavljaju najkraći put u grafu do određenog trenutka, odnosno svakom iteracijom sve više „istražujemo“ graf, sve dok ne dođemo do odredišta. Dijkstra algoritam pronalazi najkraći put u smislu „jedan do nekih“ ili „jedan do svih“ zavisno od toga da li ćemo prekinuti algoritam nakon što dođemo do odredišta ili ćemo nastaviti sve dok ne istražimo sve čvorove i njihove susjede, odnosno dok svaki od čvorova u jednom trenutku ne postane referentan u smislu da je njegov potencijal f_i konačan (što znači da se više ne može promijeniti). Nakon što potencijal f_i postane konačan za neki čvor možemo biti sigurni da smo pronašli najkraći put od početnog čvora do datog čvora. U nastavku ćemo se uvjeriti u validnost ove tvrdnje ali prije toga ćemo dati detaljan opis Dijkstra algoritma korak po korak, te njegov pseudokod za izvedbu na računaru.

Koraci Dijkstra algoritma pogodnog za izvođenje na papiru su sljedeći:

1. Postaviti $f_1 = 0$ i $f_j = \infty$ za $j = 2 .. n$.
2. Proglasiti čvor 1 za referentni čvor ($r = 1$).
3. Proglasiti potencijal f_r referentnog čvora konačnim. Ukoliko je $r = n$ preći na korak 6.
4. Za sve susjede referentnog čvora r čiji potencijali nisu konačni obaviti algoritamski korak $f_j \leftarrow \min \{f_j, f_r + l_{r,j}\}$. Drugim riječima, ukoliko je vrijednost $f_r + l_{r,j}$ manja od tekuće vrijednosti f_j, potrebno je postaviti f_j na novu vrijednost. Također korisno je da svaki put kada izvršimo prepravku u zagradu stavimo preko kojeg čvora smo došli na određeni čvor, u ovom slučaju ako smo vršili prepravku za neki čvor p, tako da je njegov novi potencijal $f_p = f_r + l_{r,p}$, trebamo napisati u $f_p(r)$, kako bismo znali kako smo došli do tog čvora, što će nam pomoći pri očitavanju puta.

5. Od svih čvorova čiji potencijal još nije konačan, pronaći čvor k sa najmanjim potencijalom. Proglasiti čvor k za novi referentni čvor (r ← k) i vratiti se na korak 3.
6. Traženi najkraći put je pronađen. Njegova dužina iznosti f_n, pri čemu sam put možemo očitati koristeći brojeve u zagradama koje smo naveli u četvrtom koraku, odnosno najkraći put očitavamo razmotavanjem unazad

Primjetimo da ukoliko ne postoji put od jednog čvora ka drugom vrijednost potencijala odredišnog čvora će biti beskonačna vrijednost, na osnovu prvog koraka. Da bismo čitaocima dodatno približili ovaj algoritam. Predstavit ćemo njegov pseudokod, tako da imaju osnovno razumijevanje za implementacija algoritma koje slijede u nastavku. Na osnovu koraka koje smo naveli iznad, pseudokod Dijkstra algoritma dat je u nastavku:

```
function Dijkstra(Graph, source):

   for each vertex v in Graph.Vertices:
         dist[v] ← INFINITY
         prev[v] ← UNDEFINED
         add v to Q
   dist[source] ← 0

   while Q is not empty:
          u ← vertex in Q with min dist[u]
          remove u from Q

          for each neighbor v of u still in Q:
             alt ← dist[u] + Graph.Edges(u, v)
             if alt < dist[v]:
                 dist[v] ← alt
                 prev[v] ← u

   return dist[], prev[]
```

U suštini, ovaj pseudokod je osnova svake implementacije Dijkstra algoritma, samo je pitanje u kojoj strukturi podataka Q ćemo čuvati podatke, kako ćemo iz te strukture izvlačiti minimalni element i kako ćemo ubacivati elemente u tu strukturu. Računarske implementacije uglavnom predstavljaju „od jednog do svih“ varijantu Dijkstra algoritma, što se vidi i iz ovog pseudokoda, jer algoritam terminira tek kada je Q prazan odnosno nakon što je svaki čvor postao referentni (što znači da je potencijal svakog čvora postao konačan). O tome koje sve strukure podataka možemo koristiti kao kontejner Q bit će više govora u narednim poglavljima.

Da bismo čitaocima još više približili kako algoritam radi, u nastavku ćemo uraditi primjer u kojem je graf predstavljen pomoću matrice susjedstva. U matrici susjedstva vrijednost na poziciji *(i,j)* u matrici nam određuje težinu grane od *i* ka *j*. U našem primjeru svi elementi na glavnoj dijagonali su beskonačne vrijednosti, što znači da nemaju granu koja ih vodi ka sebi. Čak i da to nije slučaj takve grane ne bi uticale na najkraći put jer kada bismo se kretali tim granama mogli bismo samo povećati dužinu puta što nam nije u cilju, također algoritam vješto

izbjegava takve grane. One mogu predstavljati problem u Bellman-Fordovom algoritmu prilikom traženja najdužeg puta jer kada naiđemo na takvu granu možemo se „vrtiti“ do beskonačnosti povećavajući tako dužinu puta po volji. U takvim situacijama onda kažemo da najduži put ne postoji, odnosno da je u beskonačnosti.

Primjer: Za graf zadan preko matrice susjedstva nađite najkraći put od početnog čvora ka svim ostalim, uzimajući da je početni čvor prvi čvor.

$$L = \begin{pmatrix} \infty & 9 & \infty & 21 & 25 & \infty & 17 & 28 \\ 11 & \infty & 18 & 6 & 17 & 12 & \infty & 11 \\ 5 & 8 & \infty & 7 & \infty & 4 & 2 & \infty \\ 16 & \infty & 7 & \infty & 9 & 14 & \infty & 12 \\ \infty & 9 & 11 & 9 & \infty & 10 & 8 & \infty \\ 8 & \infty & 13 & 14 & 10 & \infty & 2 & 12 \\ \infty & 13 & 7 & 6 & 7 & 2 & \infty & 1 \\ 8 & 11 & \infty & 7 & 9 & 5 & 14 & \infty \end{pmatrix}$$

Referentni čvor (r)	fr	1	2	3	4	5	6	7	8
		0	∞	∞	∞	∞	∞	∞	∞
1	**0**		9(1)		21(1)	25(1)		17(1)	28(1)
2	**9**			27(2)	15(2)	25(1)	21(2)	17(1)	20(2)
4	**15**			22(4)		24(4)	21(2)	17(1)	20(2)
7	**17**			22(4)		24(4)	19(7)		18(7)
8	**18**			22(4)		24(4)	19(7)		
6	**19**			22(4)		24(4)			
3	**22**					24(4)			
5	**24**								

Pronašli smo najkraći put od početnog čvora do svih ostalih, odnosno potencijal svakog čvora je postao konačan. U prvom koraku smo u tabelu unijeli sve vrijednosti grana od prvog (početnog) čvora ka ostalim (osim za vrijednosti beskonačno, jer to označava da ne postoji grana od jednog čvora ka drugom), te smo tim vrijednostima pridružili referentni čvor – u ovom slučaju to je prvi čvor. Nakon toga biramo polje sa minimalnom vrijednosti unutar reda, te čvor koji se nalazi u toj koloni proglašavamo za referentni, čime tvrdimo da je vrijednost u presjeku spomenute kolone i reda do tada referentnog čvora zapravo najkraći put od početnog čvora do novog referentnog čvora. U našem primjeru novi referentni čvor je čvor 2, dalje ponavljamo postupak, gledamo sve grane koje izlaze iz referentnog čvora i na njihove težine dodajemo vrijednost potencijala – u ovom slučaju to je 9. Posmatramo da li sa novim skupom grana možemo povoljnije doći do nekih čvorova, ukoliko možemo, prepravljamo vrijednost i prepravljamo referentni čvor. U našem primjeru to možemo vidjeti za čvor 4, gdje je do tada najpovoljniji put do njega bio 21 – direktno preko prvog, međutim sada nam je povoljnije da dođemo do čvora 4; idemo prvo u čvor 2 iz početnog, pa iz njega u čvor 4. Time smo postigli značajnu uštedu, stoga ćemo kao referentni čvor čvoru 4 postaviti 2. Ovaj postupak ponavljamo sve dok svaki čvor ne postane referentni u jednom trenutku.

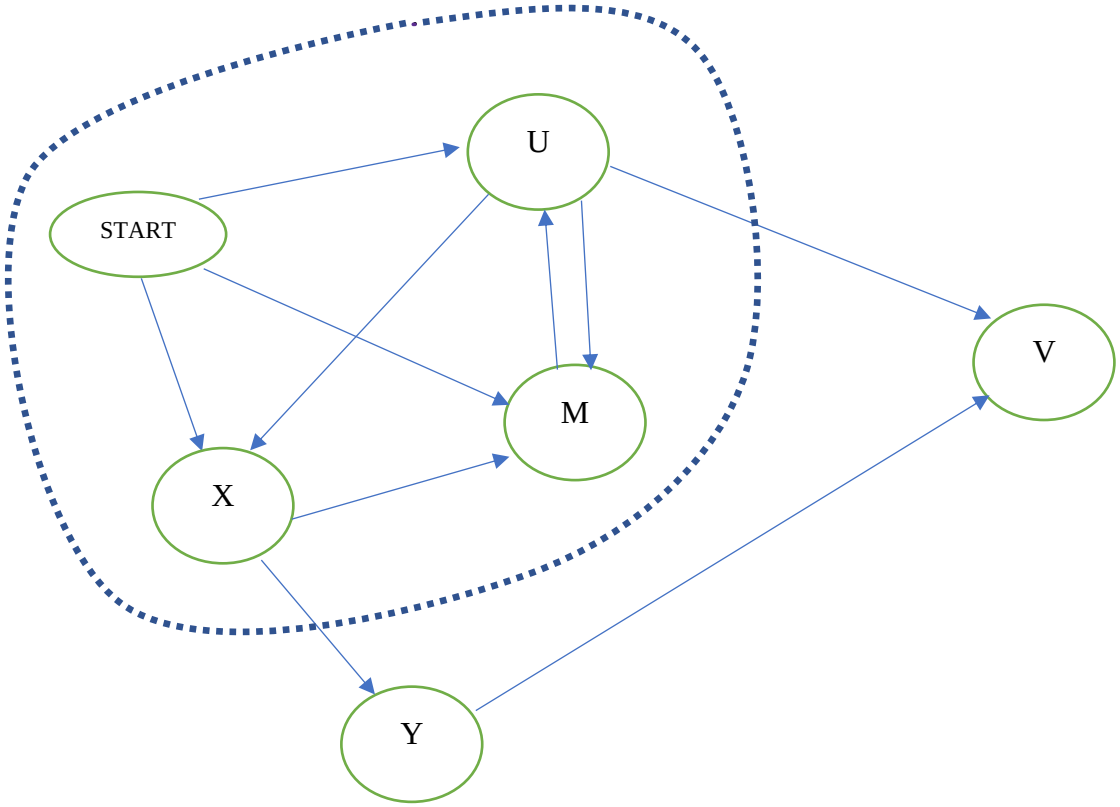
Nakon što algoritam terminira, možemo lako očitati vrijednosti najkraćih puteva od početnog čvora do svih ostalih gledajući prve dvije kolone, a konkretan put možemo vidjeti razmotavanjem unazad. Naprimjer, najkraći put iz početnog do čvora 5 je 24, vidimo da smo došli u čvor 5 preko čvora 4 (zato što mu je 4 referentni), dalje gledamo kako smo došli u čvor 4, u našem slučaju to je preko čvora 2, a u čvor 2 iz početnog. Dakle, na osnovu referentnih čvorova očitavamo put, a on glasi: $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$. Primjetimo i da nakon što proglasimo neki čvor za referentni, ne moramo više gledati grane koje vode ka njemu, jer sigurno nećemo moći naći povoljniji put. Da je to zaista tako uvjerit ćemo se u narednom poglavlju gdje ćemo predstaviti dokaz Dijkstra algoritma.

2.1 Dokaz Dijkstra algoritma

U ovom poglavlju ćemo se uvjeriti u validnost Dijkstra algoritma. Za dokazivanje ćemo koristiti matematičku indukciju. Pretpostavimo da neki čvor V treba da postane referentni čvor na osnovu već objašnjenih principa, dakle on je među susjedima već referentnih čvorova najpovoljnija opcija za nastavak iteriranja kroz sve grane grafa. Ono što sa sigurnošću možemo tvrditi na osnovu koraka i principa algoritma je to da ukoliko gledamo samo čvorove koji su do sada bili referentni i njihove susjede, nakon što smo proglasili čvor V za referentni ujedno smo i pronašli najkraći put od početnog čvora do čvora V.

Preostaje nam da dokažemo da ne postoji niti jedan povoljniji put do čvora V takav da će uključivati čvorove koji do tada nisu postali referentni. Ukoliko to dokažemo, dokazali smo i činjenicu da kada čvor postane referentan, da se njegov potencijal više ne može mijenjati, a iz toga direktno slijedi i potvrda valjanosti Dijkstra algoritam. Vizuelno ćemo predstaviti ovu ideju, gdje ćemo obilježiti čvorove koji su do sada bili referentni čvorovi.

Opisani scenario je prikazan na sljedećem grafu:



Čvorovi koji su zaokruženi predstavljaju posjećene čvorove, odnosno one čvorove koji su postali referentni. U ovom grafu, čvorovi od interesa su X,Y i V. Čvor Y je onaj čvor preko kojeg se može doći do čvora V (direktno ili indirektno), a koji ne pripada posjećenim čvorovima. S obzirom da je pretpostavka da postoji takav put koji ide van okvira posjećenih čvorova i vodi do V, zaključujemo da čvor Y mora postojati, jer ukoliko ne postoji, validnost algoritma slijedi iz pronalaska minimuma među vrijednostima grana, što smo već uradili kroz algoritam. Dalje, čvor X je najbliži čvoru Y od čvorova koji su posjećeni (čiji je potencijal konačan). Opet, na osnovu pretpostavke da postoji put iz posjećenog dijela grafa do čvora Y, zaključujemo da čvor X također mora postojati (može biti i početni čvor).

Pomoću matematičke indukcije ćemo pokazati da ne postoji put koji ide van okvira posjećenih čvorova, takav da je povoljniji od puta kojeg smo dobili posmatrajući samo posjećene čvorove.

Bazni slučaj indukcije za $n=2$ je trivijalan. Dakle, imamo samo dva čvora od kojih jedan mora biti početni tako da se najkraći put do drugog čvora može direktno očitati.

Pretpostavka indukcije je da za $n=k$, gdje je k broj posjećenih čvorova, vrijedi da su za njih dobiveni najkraći putevi zaista najkraći, odnosno da ne postoji neki drugi povoljniji put igdje u grafu. Takve vrijednosti, za koje bez sumnje možemo tvrditi njihovu validnost ćemo označavati sa $D[i]$, gdje je i čvor unutar skupa posjećenih čvorova.

Preostaje nam još da dokažemo da ovo vrijedi i za $n=k+1$, tako da ćemo reći da je čvor V novi čvor koji ulazi u skup posjećenih čvorova na osnovu algoritma. Kako bismo ovo dokazali pretpostavit ćemo suprotno, odnosno da postoji bolji put do čvora V, kojeg ćemo označiti sa *BP[V] (Best Path).* Naravno, za čvorove koji su posjećeni, na osnovu pretpostavke vrijedi da je BP[V]=D[V]. Grane od nekog čvora *x* do nekog čvora *y* ćemo označavati kao *l(x,y).*

Na osnovu ove pretpostavke i svega izloženog dobijamo sljedeće uslove:

1. $BP[V] < D[V]$
2. $D[X] + l(x,y) \leq BP[V]$
3. $D[Y] \leq D[X] + l(x,y)$
4. $D[Y] \geq D[V]$

Što se tiče prvog uslova, on direktno slijedi iz pretpostavke suprotnog, da postoji neki bolji put BP[V] od onoga puta D[V] kojeg smo dobili pomoću algoritma. Drugi uslov je trivijalan i slijedi iz definicija čvorova X i Y. S obzirom da od čvora X idemo ka čvoru Y, a iz čvora Y dolazimo u čvor V, te imajući u vidu da su sve grane nenegativne, lako je uvidjeti da je drugi uslov valjan. U trećem uslovu smo iskoristili činjenicu da iz čvora X možemo doći u čvor Y, pa automatski imamo i jednu vrijednost puta za Y, a najkraći put ka Y u najgorem slučaju može biti jednak jednom od tih puteva. Na kraju, kao četvrti uslov smo iskoristili to da je algoritam odlučio da je čvor V sljedeći čvor koji treba postati referentan, ukoliko bi bilo da je D[Y] < D[V], algoritam bi na osnovu traženja minimuma zaključio da je čvor Y taj koji treba da postane sljedeći referentni čvor. Stoga na osnovu početnih definicija čvorova i početne pretpostavke zaključujemo da je i četvrti uslov valjan.

Kombinirajući uslove 2. i 3. dobijamo:

5. $D[Y] \leq BP[V]$

Ukoliko sada kombiniramo uslove 1. i 5. dobijamo:

6. $D[Y] < D[V]$

Na kraju kombinirajući uslove 4. i 6. dobijamo:

7. ***D[V] < D[V]***

Sedmi uslov je očigledna kontradikcija, stoga naša pretpostavka da postoji put koji prolazi kroz neposjećene čvorove takav da je povoljniji od puta kojeg je ponudio algoritam je neispravna. Drugim riječima, ne postoji takav put koji je povoljniji od onog puta koji je ponudio Dijkstra algoritam uzimajući samo do tada referentne čvorove. Ovime smo dokazali treći korak matematičke indukcije. Na osnovu toga možemo reći da je Dijkstra algoritam matematički validan, odnosno u četvrtom koraku indukcije utvrđujemo validnost algoritma za svaki prirodan broj *n,* gdje *n* predstavlja broj čvorova u grafu, koji svakako mora biti prirodan broj.

U narednom poglavlju fokusirat ćemo se na vremensku kompleksnost Dijkstra algoritma. Objasnit ćemo koje implementacije su najpogodnije za maksimizaciju performansi u određenim slučajevima.

2.2 Analiza vremenske kompleksnosti Dijkstra algoritma

Različite implementacije ovog algoritma pokazuju različitu vremensku kompleksnost zavisno od struktura podataka koje se koriste za upravljanje skupom neposjećenih čvorova. Osnovna ideja u svim implementacijama ostaje ista: u svakom koraku se bira čvor sa najmanje procjenjene udaljenosti koji još nije obrađen, ažuriraju se udaljenosti susjednih čvorova, te se ovaj proces ponavlja dok svi čvorovi ne budu obrađeni ili dok čvor koji je nama od interesa ne bude obrađen ukoliko ne radimo implementaciju „jedan do svih“.

U najjednostavnijoj formi, Dijkstra algoritam se može implementirati koristeći listu, red ili matricu za čuvanje čvorova čije se udaljenosti tek trebaju ažurirati. Neovisno od toga da li je graf predstavljen pomoću matrice susjedstva ili liste susjedstva, vremenska kompleksnost u najgorem slučaju iznosi $O(n^2)$, gdje n predstavlja broj čvorova u grafu. Pretpostavimo da imamo potpuno povezan graf (sve polja matrice imaju konačne pozitivne brojeve). U prvom koraku moramo uzeti u razmatranje n vrijednosti, nakon što odaberemo najpovoljniju vrijednost za dalje iteriranje u sljedećem koraku tražimo minimum među n-1 vrijednosti, prateći ovaj princip zaključujemo da vremenska kompleksnost u najgorem slučaju iznosi:

$$O(n + n - 1 + n - 2 + \ldots + 1) = O(\frac{n(n+1)}{2}) = O(n^2).$$

Dakle vremenska kompleksnost za „naivne“ implementacije je kvadratnog reda. Ovo je prihvatljiva kompleksnost za guste grafove, međutim problem je što kompleksnost zavisi od broja čvorova grafa ali ne i od broja grana, odnosno ukoliko imamo rijedak graf sa puno čvorova jasno je da je ovakva implementacija poprilično neefikasna.

U nastavku ćemo prikazati implementaciju Dijkstra algoritma gdje se čvorovi čuvaju u matrici, tako da prilikom svake iteracije u obzir moramo uzeti sve čvorove u određenom redu.

KOD 1

```
#include <iostream>
using namespace std;
#include <limits.h>
// Number of vertices in the graph
#define V 9

int minDistance(int dist[], bool sptSet[])
{

    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}
```

```
void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] << endl;
}


void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the
                 // shortest distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is
                    // included in shortest path tree  or
    // shortest distance from src to i isfinalized

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet,
            // there is an edge from u to v, and total
            // weight of path from src to  v through u is
            // smaller than current value of dist[v]

            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];

    }

    printSolution(dist);

}
```

```
U navedenom kodu nula predstavlja odsustvo grane

// driver's code
int main()
{
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}

// This code is contributed by shivanisinghss2110
```

U navedenom kodu je definisana matrica koja čuva podatke o povezanosti grafa, nakon toga definisan je i niz u kojem se čuvaju najkraće udaljenosti svakog od čvorova do datog trenutka, te niz bool vrijednosti koje označavaju da li je neki čvor posjećen ili nije. Nakon što čvor bude posjećen, njegov potencijal postaje konačan odnosno ne može se više promijeniti što možemo vidjeti i uslovima ovog koda. Pozitivna strana ovakve implementacije je jednostavnost, a negativna strana je to što troši previše vremena, te to vrijeme ne ovisi od gustine grafa, tako da gust i rijedak graf se tretiraju skoro pa isto. Da bismo postigli to da brzina izvršavanja zavisi i od broja grana morat ćemo koristiti drugačiji pristup. Ideja je da u prioritetnom redu čuvamo potencijale čvorova kako bismo ubrzali pronalazak minimalnog elementa te izbjegli čuvanje veza koje ne postoje.

2.2.1 Implementacija Dijkstra algoritma pomoću prioritetnih redova

Implementacija Dijkstra algoritma koristeći prioritetne redove, posebno gomile, predstavlja značajno poboljšanje u odnosu na naivnu implementaciju zasnovanu na prostom pretraživanju minimalnog potencijala čvorova. Razlozi za ovo poboljšanje su mnogobrojni i uključuju kako efikasnost u upravljanju skupom nepostjećenih čvorova, tako i smanjenje vremena potrebnog za ažuriranje i pronalaženje čvorova s najmanjim potencijalom. U nastavku su detaljnije objašnjeni ključni aspekti i prednosti korištenja prioritetnih redova u kontekstu Dijkstra algoritma.

- *Efikasno upravljanje neposjećenim čvorovima*

Korištenjem prioritetnih redova, algoritam u svakom koraku može brzo pronaći i ukloniti čvor sa najmanjim potencijalom. Nakon toga će se razmatrati susjedni čvorovi pronađenog čvora i opet nakon toga će među tim čvorovima biti pronađen onaj čvor sa najmanjim potencijalom. Ovo je značajno efikasnije u odnosu na naivnu implementaciju koja zahtijeva pregled svih čvorova da bi se pronašao onaj s najmanjim potencijalom.

- *Dinamičko ažuriranje potencijala*

Kada se otkrije put do nekog čvora koji je kraći od trenutno poznatog, potrebno je ažurirati njegov potencijal. U naivnoj implementaciji, ovo ažuriranje donosi dodatnu složenost i dodatno gubljenje vremena. Međutim, kada se koriste prioritetni redovi, posebno binarna gomila ili **Fibonacci gomila**, ažuriranje potencijala može se obaviti efikasno, čime se dodatno skraćuje vrijeme izvršavanja algoritma.

- *Smanjenje vremena izvršavanja u zavisnosti od gustine grafa*

Jedna od ključnih prednosti korištenja prioritetnih redova je sposobnost algortima da se prilagodi gustini grafa. U rijetkim grafovima, gdje broj veza nije kvadratno proporcionalan broju čvorova, upotreba prioritetnih redova omogućava algortimu da brže pronađe najkraće puteve. Ovakav pristup omogućava to da manji broj veza direktno utiče na vremensku kompleksnost, odnosno algoritmu će trebati više vremena da obradi više veza, a manje vremena da obradi manje veza. Ovime dobijamo vremensku kompleksnost koja zavisi i od gustine grafa, a to nismo imali kada smo koristili naivnu implementaciju.

- *Fleksibilnost u izboru strukture podataka*

Korištenje prioritetnih redova nudi veliku fleksibilnost u izboru strukture podataka koja najbolje odgovara problemu. Generalizirat ćemo red vremenske kompleksnosti za sve implementacije Dijkstra algoritma koje koriste gomile kao prioritetni red, te ćemo onda predstaviti tabelu vremenske kompleksnosti izvršavanja određenih implementacija za svaku od gomila. Pomoću tih informacija moći ćemo direktno očitati vremensku kompleksnost implementacije Dijkstra algoritma za bilo koju vrstu gomile. Pored gomila, spomenut ćemo i još jednu posebnu strukturu podataka, za koju ćemo koristiti njen izvorni naziv na engleskom jeziku, a to je *bucket*.

- *Skalabilnost i adaptabilnost*

Ovakav način implementacije omogućava bolju skalabilnost, što je ključno za upotrebu algoritma u velikim sistemima, kao što su internet mreže, velike baze podataka, kompleksni simulacioni sistemi i brojnim drugim. Također, u dinamičkim sistemima, gdje se graf može mijenjati tokom vremena, implementacija pomoću prioritetnih redova omogućava brže adaptiranje algoritma na novonastalu situaciju, odnosno omogućeno je efikasnije ažuriranje potencijala bez potrebe za velikim korekcijama i mnogobrojnim ponovnim izračunavanjima. Adaptabilnost i skalabilnost algoritma su jako bitni faktori u *real-time* aplikacijama.

2.2.1.1 Bucket i double-bucket

Prije nego što pređemo na gomile, koje predstavljaju primarnu implementaciju prioritetnog reda u vezi sa Dijkstra algoritmom, pomenut ćemo i specijalnu strukturu podataka koja se naziva *bucket*. S obzirom da je jedini prijevod koji je dostupan u literaturi za ovu strukturu podataka „kofičarske strukture“, te uzimajući u obzir dosljednost ovog završnog rada koristit ćemo termin *bucket* u nastavku izlaganja. Također ovaj termin je pogodan zbog literature dostupne na internetu, jer na ovu temu ima jako malo materijala na našem jeziku i njemu srodnim jezicima. Ukoliko čitatelji budu željeli samostalno istražiti ovu strukuru podataka, ovaj pasus će dati dobar uvod za to. Iako bucket ne možemo poistovijetiti sa prioritetnim redom, spominjemo ga na ovom mjestu jer uz prioritetne redove daje jedan od najboljih načina za implementaciju Dijkstra algoritma te se nerijetko koristi zajedno u implementaciji sa prioritetnim redovima s obzirom da i bucketi i prioritetni redovi imaju svoje vrline i mane u pogledu implementacije ovog algoritma, pa njihova kombinacija najvjerovatnije daje optimalno rješenje.

Bucket struktura podataka organizuje elemente na osnovu njihovih ključeva. Svaki bucket odgovara određenoj vrijednosti ključa. Naprimjer, ako imamo čvorove označene sa A,B,C,D,E u skupu, gdje su trenutno poznate najkraće udaljenosti od izvora, te one iznose 4,4,9,7,9 respektivno, možemo kreirati buckete na sljedeći način:

- B[4]={A,B}
- B[9]={C,E}
- B[7]={D}

Dakle svaki bucket sadrži čvorove sa istom udaljenosti u odnosu na izvor, što omogućava efikasno ažuriranje udaljenosti i relativno brzo izbacivanje čvorova iz strukure. Ključne operacije za bucket strukuru i njihove vremenske kompleksnosti su:

- Ubaci (insert) - O(1)
- Ukloni (remove) - O(1)
- Smanji ključ (decrease key) - O(1)
- Pronađi minimum (find minimum) - O(c) (*c* je maksimalna vrijendost ključa u strukturi)

Vremenska kompleksnost Dijkstra algoritma implementiranog preko bucketa je ***O(m+nc),*** gdje *m* predstavlja broj grana, *n* predstavlja broj čvorova, dok je već pomenuto *c* maksimalna vrijednost ključa u strukutri. Na osnovu ovih činjenica možemo naslutiti problem implementacije algoritma preko bucketa u generalnom slučaju. Ukoliko imamo najkraće udaljenosti od izvora takve da se jako razlikuju, naprimjer 2,6,187,23 za neke čvorove, primjetimo da je će *c* biti jednako 187. U datom primjeru imamo samo četiri čvora i kompleksnost koja je velika za ovako mali graf. Dakle implementacija preko bucketa je pogodna u specifičnim situacijama, međutim najbolja implementacija algoritma takva da ne ovisi od konkretne strukure grafa i vrijednosti grana je putem prioritetnih redova, odnosno gomîla kao načina predstavljanja prioritetnog reda.

Kako bismo poboljšali performanse ubacivanja, brisanja i ažuriranja ključeva unutar bucketa, možemo definisati takav bucket da su svi njegovi elementi također bucketi. Ovakva struktura podataka se naziva *double-bucket*. Iako ima već objašnjene prednosti u odnosu na „obični“ bucket, uvođenjem double-bucketa možemo naići na dodatne probleme poput složenosti implementacije, dodatnog memorijskog troška i ostalih. U specifičnim slučajevima i grafovima sa velikim brojem čvorova ove mane možemo zanemariti, međutim ukoliko imamo neujednačene težine grana, što će vjerovatno rezultirati time da imamo neujednačene vrijednosti najkraćih puteva svakog od čvorova u odnosu na izvor, dobit ćemo preveliku razliku u vrijednostima ključeva što će uticati na performanse. Pored toga ukoliko nemamo dovoljno veliki graf i dovoljno dobru raspodjelu težina, uvodimo nepotrebnu složenost u našu implementaciju, odnosno ako imamo dovoljno dobru raspodjelu težina grana, pogodnu za korištenje bucketa, a da je graf sa malim brojem čvorova, onda su vjerovatno „obični“ bucketi bolja opcija. Ukoliko imamo graf sa velikim brojem čvorova i sa takvom raspodjelom težina grana da je pogodna za korištenje bucketa, u tom slučaju su double-bucketi najvjerovatnije optimalna opcija, ali to je jako specifičan slučaj. Dakle, iako double-bucketi predstavljaju optimizaciju bucketa u smislu performansi za osnovne operacije, sa sobom donose i dodatnu kompleksnost u drugim aspektima, te detaljno moramo proanalizirati situaciju ukoliko se odlučimo da koristimo ovu strukuru podataka. U zaključku, bucketi i double-bucketi su interesantna strukutura podataka koja može imati odlične performanse u određenim situacijama, međutim ukoliko nam je cilj da generaliziramo problem najkraćeg puta, konkretno Dijkstra algoritam, onda je implementacija pomoću prioritetnog reda najbolja opcija. Kao strukture koje će predstavljati prioritetni red koristit ćemo gomile, te su one naša sljedeća tema.

2.2.1.2 Implementacija algoritma pomoću gomila kao prioritetnih redova

U prethodnim poglavljima površno smo objasnili razne načine implementacije Dijkstra algoritma i naveli njihove vremenske kompleksnosti, te smo rekli koja od njih je najbolja. Sada ćemo detaljno objasniti najbolji način implementacije Dijsktra algoritma na računaru, a to je preko prioritetnih redova, te ćemo objasniti zašto je baš on najbolji. Kao što je već rečeno kada koristimo matricu susjedstva ili listu susjedstva za predstavljanje grafa, a ne koristimo prioritetne redove za čuvanje informacija potrebnih za algoritam, kao što su praćenje najmanjeg trenutnog elementa, odnosno udaljenosti, tada je vremenska kompleksnost proporcionalna kvadratu broja čvorova. Dakle moramo proći kroz cijelu matricu kako bismo odredili minimum

u svakoj iteraciji, što je loše jer „gubimo vrijeme“ i na grane koje ne postoje. Situacija je malo bolja ukoliko koristimo listu susjedstva jer tada naprimjer možemo u prvi niz koji je svojstven prvom čvoru dodati parove koji se sastoje od imena čvora i udaljenosti tog čvora od prvog čvora. Iako je ovo poboljšanje u odnosu na matricu susjedstva opet u najgorem slučaju (kada je potpuno povezan graf) moramo detaljno obraditi svaku moguću granu iako neke grane mogu biti irelevantne. Na osnovu svega napisanog vremenska kompleksnost u najgorem slučaju za pomenute implementacije iznosi $O(n^2)$ kao što je već rečeno u prethodnim poglavljima, što je prihvatljivo za guste grafove, međutim za rijetke grafove ova kompleksnost je problematična jer ne ovisi od broja grana. Prirodno se postavlja pitanje možemo li nekako ubrzati algoritam, tako da on zavisi i od gustine grafa.

Odgovor na prethodno pitanje je potvrdan zahvaljujući prioritetnim redovima. Kao što već znamo prioritetni redovi su strukture podataka koje se zasnivaju na prioritetu – element sa najvećim prioritetom je na „vrhu“ i lako je dokučiv, te je on prvi kandidat za napuštanje strukture. Najpopularniji način implementacije prioritetnog reda je putem gomila. Postoje mnoge vrste gomila, od kojih svaka ima svoje prednosti i mane zavisno od slučajeva upotrebe. Prije nego počnemo detaljnije govoriti o gomilama prezentovat ćemo osnovnu ideju implementacije Dijkstra algoritma preko gomila:

- Inicijalizacija – svi čvorovi u grafu dobivaju početnu vrijendost udaljenosti koja je postavljena na beskonačno, osim početnog čvora čija je vrijednost udaljenosti postavljena na nulu. Također sve čvorove dodajemo u gomilu.
- Izvlačenje čvora sa minimalnom udaljenosti – u svakoj iteraciji iz gomile izvlačimo čvor sa najmanjom udaljenosti (počinjemo od početnog čvora). Iako lako možemo dokučiti do takvog čvora, ponovno uspostavljanje gomile nakon izvlačenja čvora na vrhu (čvor je na vrhu jer je u pitanju gomila sa prioritetom minimuma) je vremenski zahtjevno.
- Ažuriranje vrijednosti čvorova – nakon što smo izvukli čvor sa najmanjom udaljenosti, posmatramo njegove susjedne čvorove, te računamo nove vrijednosti svakog od čvorova, odnosno nove najkraće udaljenosti od početnog čvora. Ukoliko su nove udaljenosti bolje od prethodnih vršimo ažuriranje vrijednosti čvorova. S obzirom da gomila ima svoju jasno definisanu strukturu jasno nam je da se mora izvršiti neka transformacija gomile kako bi se novonastala situacija opet mogla predstaviti preko gomile. Jasno je da je ovaj postupak vremenski zahtjevan.
- Ponavljanje – prethodne korake ponavljamo sve dok prođemo kroz sve čvorove.

Analizirajući prethodno opisani postupak primjetimo da su operacije ažuriranja vrijednosti čvorova i izvlačenja čvora sa minimalnom udaljenosti ključne za implementaciju algoritma pomoću gomila kao strukture koja predstavlja prioritetni red. U literaturi na engleskom jeziku operacije izvlačenja čvora sa minimalnom udaljenosti i ažuriranje vrijednosti čvorova se nazivaju *Extract-Min* i *Decrease-Key* respektivno, tako da ćemo u nastavku koristiti i te izraze. Da bismo postigli to da Dijkstra algoritam da odgovor na pitanje najkraće udaljenosti od jednog čvora do svih, jasno nam je da moramo primjeniti operaciju Extract-Min onoliko puta koliko ima čvorova, također trebamo imati na umu da u najgorem slučaju ćemo za svaku granu unutar grafa morati vršiti ažuriranje odnosno operaciju Decrease-Key, što nas dovodi do zaključka da je vremenska kompleksnost Dijkstra algoritma pomoću gomila kao prioritetnih redova jednaka:

$$O(m \cdot T_{DK} + n \cdot T_{EM})$$

- m – broj grana u grafu
- n – broj čvorova u grafu
- DK – vremenska kompleksnost operacije Decrease-Key
- EM – vremenska kompleksnost operacije Extract-Min

Dakle dobili smo generaliziranu formulu za vremensku kompleksnost implementacije Dijkstra algoritma preko gomila – a one predstavljaju prioritetne redove. Preostaje nam još da za različite vrste gomila odredimo vremenske kompleksnosti traženih operacija, te ćemo na osnovu toga dobiti odgovor na pitanje koja je vrsta gomile najpovoljnija za implementaciju s aspekta vremenske kompleksnosti. U nastavku ćemo prikazati vrememenske kompleksnosti za najviše korištene operacije za binarne, binomne i Fibonacci gomile.

Procedure	Binary Heap (worst case)	Binomial Heap (worst case)	Fibonacci Heap (amortized)
Make-Heap	Θ(1)	Θ(1)	Θ(1)
Insert	Θ(log(n))	O(log(n))	Θ(1)
Find-Minimum	Θ(1)	O(log(n))	Θ(1)
Union (Merge)	Θ(n)	O(log(n))	Θ(1)
Decrease-Key	Θ(log(n))	Θ(log(n))	Θ(1)
Extract-Min	Θ(log(n))	Θ(log(n))	O(log(n))
Delete	Θ(log(n))	Θ(log(n))	O(log(n))

Na osnovu prikazane tabele možemo vidjeti da **Fibonacci gomile** imaju najpovoljniju kombinaciju vremenskih kompleksnosti za operacije od interesa za implementaciju Dijkstra algoritma. Pored toga prikazali smo i kompleksnosti operacija za binarnu i binomnu gomilu. Binarna gomila je jedna od najpopularnijih vrsta gomila, što dokazuje i činjenica da je bibliotečni priority_queue u C++ implementiran preko binarne gomile, ali ona očigledno nije najbolji izbor za ovaj specifičan slučaj. Još jedan interesantan detalj u vezi sa prethodnom prikazanom tabelom je to što u koloni gdje je Fibonacci gomila imamo amortizovano vrijeme pa ćemo sada objasniti šta je to zapravo amortizovano vrijeme.

Amortizovano vrijeme je koncept koji se koristi za analizu performansi algoritma. Umjesto da se analizira najgori slučaj vremenske složenosti algoritma, amortizovana analiza pruža analizu prosječnog slučaja algoritma uzimajući u obzir trošak nekoliko operacija izvedenih tokom vremena. Ključna ideja amortizovane analize je raspodjela troškova skupe operacije preko nekoliko operacija. Ona nam pruža garanciju da je prosječna vremenska složenost operacije konstantna, čak i ako su neke operacije skupe.

Druga interesantna stvar u prikazanoj tabeli je i oznaka Θ. Iako je poprilično poznato šta to zapravo predstavlja notacija *O* u vidu vremenske kompleksnosti, manje je poznato šta zapravo znači Θ. S obzirom da se ovaj rad prvenstveno bavi proučavanjem vremenske kompleksnosti, objasnit ćemo detaljno šta one zapravo znače. Prvo ćemo objasniti šta u teoriji zapravo predstavlja big-O notacija:

Za funkciju *f(n)* kažemo da je *O(g(n))*, ako postoje pozitivne konstante *c* i n_0, tako da je:

$$f(n) \leq cg(n), za\ svako\ n \geq n_0$$

U biti ovo predstavlja gornju granicu složenosti, dakle ako je vremenska kompleksnost O(n), to znači da ona nikako ne može preći O(n), odnosno to je najgori mogući slučaj.

Što se tiče notacije big- Θ, nju možemo izraziti sljedećom tvrdnjom:

Za funkciju *f(n)* kažemo da je *Θ(g(n))*, ako postoje pozitivne konstante c_1, c_2 i n_0, tako da je:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), za\ svako\ n \geq n_0$$

U prijevodu ova notacija nam garantuje da ukoliko je vremenska kompleksnost neke funkcije jednaka Θ(n), da će ona zaista i biti reda n, za razliku od O(n) koja nam govori da će u najgorem slučaju biti reda n, ali ništa je ne spriječava da bude O(log(n)) naprimjer u nekom od slučajeva. Postoji još i big-Ω notacija koja predstavlja najbolji slučaj, u njene detalje nećemo dublje ulaziti, jer je poprilično slična predhodnim notacijama u smislu formalne definicije, a i nigdje je ne koristimo u ovom radu.

Na osnovu svega prikazanog u ovom dijelu, zaključili smo da nam je najbolje implementirati Dijkstra algoritam koristeći Fibonacci gomile ukoliko želimo da maksimiziramo brzinu izvršavanja algoritma. U narednom poglavlju fokusirat ćemo se na detaljno obrazloženje Fibonacci gomila, te ćemo predstaviti implementaciju algoritma koristeći njih. Prije nego što pređemo na njihovu obradu predstavit ćemo i pseudokod za implementaciju Dijkstra algoritma koristeći prioritetne redove uz proizvoljan izbor prioritetnog reda:

Algorithm 3: DIJKSTRA

input : An arc weighted graph $G = (V, A)$, and source vertex $s \in V$
output: A populated label array L and predecessor array P

```
1  For all u ∈ V, set L(u) = ∞, set D(u) = false, and set P(u) = NULL
2  Set L(s) = 0 and insert the ordered pair (L(s), s) into Q
3  while Q is not empty do
4      Let (L(u), u) be the element in Q with the minimum value for L(u)
5      Remove the element (L(u), u) from Q
6      Set D(u) = true
7      foreach v ∈ Γ(u) such that D(v) = false do
8          if L(u) + w(u, v) < L(v) then
9              if L(v) < ∞ then
10                 Decrease the key of (L(v), v) to L(u) + w(u, v). That is, replace the element (L(v), v) in Q
                       with the element (L(u) + w(u, v), v)
11             else
12                 Insert the element (L(u) + w(u, v), v) into Q
13             Set L(v) = L(u) + w(u, v) and set P(v) = u
```

2.3 Fibonacci gomile