Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Danijel Maraž

Analiza problema k-strežnikov

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomskem delu se osredotočite na problem k-strežnikov, predstavite teoretične osnove in primere uporabe v praksi. Poiščite učinkovite algoritme za reševanje tega problema, algoritme implementirajte v programskem jeziku Java in jih poženite na izbranih testnih množicah. Rezultate meritev ovrednotite in predstavite ugotovitve o kvaliteti posamezne rešitve.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Opis problema **3**

2.1 Sprotne rešitve 5

2.2 Statični algoritem 7

2.3 Pomen obeh pristopov 10

3 Implementacija in testiranje **11**

3.1 Požrešni algoritem 15

3.2 Harmonični algoritem 17

3.3 Algoritem delovne funkcije 19

3.4 Optimalni algoritem 27

4 Rezultati testiranja **37**

4.1 Popolnoma naključne zahteve 38

4.2 Dva centra aktivnosti 39

4.3 Trije centri aktivnosti 40

4.4 Realni primer 41

4.5 Čas izvajanja 43

5 Zaključek **45**

Literatura **47**

Povzetek

Naslov: Analiza problema k-strežnikov

Avtor: Danijel Maraž

Problem k-strežnikov je optimizacijski problem, preproste in intuitivne narave. V tem problemu mora algoritem nadzorovati premike množice k strežnikov, ki so predstavljeni kot točke metričnega prostora in strežbo zahtev, ki so tudi v obliki točk prostora. Ob vsaki novi zahtevi mora algoritem izbrati ustrezni strežnik in ga premakniti na mesto zahteve. Glavni cilj algoritma je torej minimizirati skupno prepotovano razdaljo strežnikov ob obdelavi zahtev. Zahteve se pojavljajo ena za drugo, zato je algoritmu vidna le trenutna zahteva, ki jo mora v tistem koraku obdelati. S problemom se srečamo vsakič, ko moramo obdelovati zahteve v prostoru. Na primer pri podjetjih za dovažanje, policijskih upravah in v računalništvu pri omrežjih za posredovanje vsebin. Cilj diplomskega dela je preizkusiti učinkovitost najpopularnejših rešitev problema in jih primerjati. Pri tem smo uporabljali metode ocenjevanja, ki jih uporabljajo raziskovalci na tem področju in vsak algoritem tudi implementirali in testirali v programskem jeziku Java. V delu smo pri testiranju uporabljali realne podatke, česar ni bilo mogoče zaslediti pri ostalih podobnih študijah problema. Na koncu smo dobili zanimivo sliko slabosti in vrlin posameznih rešitev problema.

Ključne besede: optimizacija, k-strežnikov, algoritmi, metrični prostor, Java, testiranje, zahteve.

Abstract

Title: Analysis of the k-server problem

Author: Danijel Maraž

The k-server problem is an optimization problem of a relatively simple and intuitive nature. In this problem the algorithm must control the movements of a set of k servers, represented as points of a metric space, to service a set of requests, also represented by points of a metric space. As a new request arrives the algorithm must choose an appropriate server and move it to the requests location for servicing. The main goal of the algorithm is to minimize the total distance travelled by all the servers. Requests appear in a sequential manner so the algorithm only sees the current request at any one time. In practice it appears many times in the fields of servicing different kinds of requests. For example it is a common problem of delivery companies, police precincts and in the field of computer science by content delivery networks. The main goal of the thesis was to test the efficiency of the most popular solutions to the problem and make a comparison between these. To achieve this goal we made use of evaluation methods practiced by seasoned researches on this topic. We also implemented and tested each algorithm in the Java programming language. Noteworthy was also our usage of real data in contrast to similar studies which only utilized randomly generated data for testing. In the end we manage to get an interesting picture of the strong and weak points of the most popular solutions to the problem.

Keywords: optimization, k-servers, algorithms, metric space, Java, testing, requests.

Poglavje 1

Uvod

Problem k-strežnikov je del domene teoretičnega računalništva in sicer bolj podrobno spada pod kategorijo sprotnih algoritmov in je bistven del teorije metodike konkurenčne analize (*angl. competitive analysis*). V tem problemu mora algoritem nadzorovati premike množice k strežnikov, ki so predstavljeni kot točke metričnega prostora in strežbo zahtev, ki so v obliki točk prostora. Ob vsaki novi zahtevi mora algoritem izbrati ustrezni strežnik in ga premakniti na mesto zahteve. Glavni cilj algoritma je minimizirati skupno prepotovano razdaljo strežnikov ob obdelavi zahtev. Zahteve se pojavljajo ena za drugo, zato je algoritmu vidna le trenutna zahteva, ki jo mora v tistem koraku obdelati. V praksi se uporablja pri omrežjih za posredovanje vsebin (*angl. content delivery networks*) in sicer za izbiranje od kod bo prišla neka želena spletna vsebina, oziroma če bi bilo boljše, če bi celotno vsebino premaknili na bližji spletni strežnik. Druge bolj preproste uporabe so pri državnih ustanovah, ki skrbijo za red in varnost, torej tako za usmerjanje policijskih avtomobilov kot rešilcev. Najbolj intuitivna in preprosta rešitev problema je zagotovo požrešni algoritem [5], ki deluje po principu, da zahtevo postreže najbližji strežnik. Ta predstavlja osnovo za skoraj vse ostale sprotne rešitve problema. Med manj znanimi se najde harmonični algoritem [5], ki temelji na verjetnosti obdelave zahteve glede na oddaljenost od strežnika, torej bližje kot je zahteva, večja je verjetnost, da bo ta obdelana, vendar ni zagoto-

vljena kot pri požrešni rešitvi. V praksi se kot najuspešnejši algoritem izkaže tako imenovani algoritem delovne funkcije (*angl. work function algorithm*) [5], ki pri svojem odločanju upošteva razdaljo do trenutne zahteve kot tudi podatke o preteklih zahtevah. Pomemben del ocenjevanja učinkovitosti algoritma je primerjava prepotovane razdalje z razdaljo, ki jo prepotujemo pri optimalni obdelavi celotnega zaporedja zahtev ob predpostavki, da so nam nahajališča zahtev znana vnaprej. Ker ima tak algoritem že vse podatke o zahtevah in jih zato sproti ne sprejema, ga imenujemo statičen. Proces pridobivanja optimalne rešitve zajema pretvorbo točk problema v usmerjen graf ter reševanje problema iskanja največjega pretoka za minimalno ceno (*angl. min-cost max-flow*) na le tem [7]. V delu so implementirani vsi zgoraj omenjeni algoritmi ter nato testirani na ustvarjenih (naključnih) in pridobljenih (realnih) podatkih. Na temo testiranja teh in drugih algoritmov obstaja že veliko del, vendar niso v slovenščini. Omembe vreden je članek Measuring True Performance of the Work Function Algorithm for Solving the On-line k-Server Problem [8], ki se poslužuje drugačne metodike testiranja, ne meri časa izvajanja in ne testira na realnih podatkih. Za razliko se pri našem delu analizira povprečen čas izvajanja, časovno zahtevnost, prostorsko zahtevnost in učinkovitost posameznih algoritmov s poudarkom na ugotavljanju razlik med njimi.

Poglavje 2

Opis problema

Opišimo problem na preprostem primeru iz prakse. Imamo podjetje, ki se ukvarja s servisom računalnikov po vseh večjih krajih v Sloveniji. V podjetju so trenutno zaposleni trije serviserji, ki se na začetku delovnega dneva vsi nahajajo v Ljubljani. Ob prejemanju klicev se zmenijo, kdo bo potoval do katerega kraja na servis. Po vsakem uspešnem servisu nato prejmejo nov klic iz drugega kraja, ki ga morajo obdelati. Pri tem primeru so vozlišča grafa sestavljena iz večjih slovenskih mest, povezave pa iz prometnih povezav med temi. Serviserji so strežniki, ki so sprva vsi na istem mestu (v Ljubljani), na koncu dneva so lahko v popolnoma različnih mestih, odvisno od zaporedja klicev strank. Pri tem poslu se seveda pojavijo stroški za gorivo, zato vodstvo želi čim bolje razporediti zaposlene. Bistvo problema v metričnem prostoru ali obteženem grafu (npr. na slovenskem omrežju mest) je optimalno premikanje k-strežnikov k zahtevam, ki se sproti pojavljajo na točkah metričnega prostora. K-strežnikov se na začetku nahaja na k fiksnih točkah. Nato se ob vsakem naslednjem časovnem koraku v eni izmed točk metričnega prostora pojavi nova zahteva, katero mora tedaj obdelati eden izmed strežnikov. Pri vsem tem se predpostavi, da se zahteve pojavljajo ena za drugo in da nam je naslednja zahteva znana šele po obdelavi prejšnje. Cilj je minimizirati celotno prepotovano razdaljo strežnikov.

Bolj formalno M naj predstavlja metrični prostor in naj bo $d(a_1, a_2)$ razda-

lja med točkama a_1 in a_2 v M. Vsaka razdalja je nenegativna, simetrična in zadovoljuje trikotniško neenakost. Zanimajo nas podmnožice množice M velikosti k, ki se imenujejo konfiguracije (primer konfiguracije je zaporedje mest, v katerih se nahajajo serviserji). Naj $M^{(k)}$ predstavlja množico vseh konfiguracij množice M. Pojem razdalje razširimo iz M na $M^{(k)}$, in sicer če sta C_1 in C_2 konfiguraciji, naj $d(C_1, C_2)$ predstavlja minimalno prepotovano razdaljo ob spremembi konfiguracije iz C_1 v C_2 (se pravi najcenejši premik skupine strežnikov, ki so trenutno v konfiguraciji C_1, da dosežejo želeno konfiguracijo C_2). Problem definiramo z začetno konfiguracijo $C_0 \in M^{(k)}$ in zaporedjem zahtev $r = (r_1, ..., r_m)$ točk M. Rešitev je zaporedje konfiguracij $C_1, ..., C_m \in M^{(k)}$, da velja $r_t \in C_t$ za vsak $t = 1, ..., m$. Vseh k-strežnikov je sprva v konfiguraciji C_0 in obdela zahteve $r_1, ..., r_m$, s tem se sprehodi po konfiguracijah $C_1, ..., C_m$. Cena rešitve je celotna prepotovana razdalja strežnikov, torej $\Sigma_{t=1}^{m}\, d(C_{t-1}, C_t)$. Cilj je najti rešitev z najmanjšo ceno.
Sprotni algoritem si pri iskanju rešitve C_t lahko pomaga samo s podatki o preteklih zahtevah $r_1, ..., r_t$ in $C_0, ..., C_{t-1}$. Učinkovitost sprotnih algoritmov se ocenjuje s pomočjo rezultatov optimalnega (statičnega) algoritma, ki vnaprej vidi vse zahteve $r_{t+1}, ..., r_m$. Pri začetni konfiguraciji C_0 in $r = (r_1, ..., r_m)$ naj $cost_A(C_0, r)$ predstavlja ceno sprotnega algoritma A in $opt(C_0, r)$ ceno optimalne rešitve. Pravimo, da ima algoritem A konkurenčno razmerje ρ, če velja za vsaka C_0 in r

$$cost_A(C_0, r) \leq \rho \cdot opt(C_0, r) + \Phi(C_0)$$

za nek c. Prištetje konstante $\Phi(C_0)$, ki je neodvisna od zaporedja zahtev, nam omogoča, da se znebimo odvisnosti od začetne konfiguracije. Torej naj bi konstanta služila za zmanjševanje vpliva srečne začetne postavitve, če gre za enkratno testiranje. Konstante v delu nismo uporabljali, ker smo računali povprečje več serij zahtev, z naključnimi začetnimi konfiguracijami strežnikov, kar domnevam, da je bistveno omejilo vpliv začetne konfiguracije [5].

2.1 Sprotne rešitve

2.1.1 Požrešni algoritem

Algoritem za svoje odločanje izkorišča le trenutno konfiguracijo in nahajališče trenutne zahteve, zato spada med rešitve brez spomina (*angl. memoryless*). Ob neki konfiguraciji strežnikov C_1 mora algoritem najti

$$min\{d(C_1, C_2)\},$$

kjer velja C_1,$C_2 \in M^{(k)}$. Z drugimi besedami, algoritem poišče strežnik, ki je najbližji trenutni zahtevi in ga pomakne na njeno lokacijo in s tem dobi novo konfiguracijo točk. Glavna prednost tega algoritma je velika preprostost in hitrost izvajanja (v primerjavi z ostalimi). Njegova časovna zahtevnost je enaka $O(k \cdot m)$, kjer je k število strežnikov in m število zahtev.

2.1.2 Harmonični algoritem

Algoritem je zelo podoben požrešnemu z eno bistveno razliko. Požrešni algoritem ob vsakem koraku pomakne najbližji strežnik k zahtevi, harmonični pa pošlje vsak strežnik z verjetnostjo, ki je obratno sorazmerna razdalji med strežnikom in zahtevo. Naj bo A lokacija trenutne zahteve. Faktor normalizacije je v tem primeru $N = \Sigma_z 1/d(A, z)$, kjer zavzema z vrednosti lokacij vseh strežnikov. Strežnik na lokaciji x se premakne do A z verjetnostjo $1/(N \cdot d(A, x))$ [1]. Harmonični algoritem ima enako časovno zahtevnost kot požrešni z le malo dodanega računanja.

2.1.3 Algoritem delovne funkcije

Algoritem delovne funkcije (*angl. work function algorithm*) je metoda dinamičnega programiranja, ki velja za eno izmed najboljših sprotnih algoritmov problema. Omejimo se na prostor trenutne konfiguracije strežnikov in vseh prihajajočih zahtev. Za vsako konfiguracijo X definiramo $w(C_0, r_1, ..., r_t; X)$ kot ceno optimalne rešitve, ki sprva začne pri C_0, nato obdela vseh t zahtev

in se na koncu znajde v konfiguraciji X. Za določen C_0 in $r_1, ..., r_t$ je w realna funkcija na $M^{(k)}$. Takšni funkciji pravimo delovna funkcija w_t (*angl. work function*) ob določenih parametrih C_0 in r. S pomočjo metod dinamičnega programiranja se da vrednost delovne funkcije $w_t(X) = w(C_0, r_1, ..., r_t; X)$ z lahkoto izračunati. Za $i = 1, ..., t$ izračunamo

$$w_i(X) = min\{w_{i-1}(Z) + d(Z, X) : Z \in M^{(k)} \text{ ob } r_i \in Z\}$$

z osnovno vrednostjo $w_0 = d(C_0, X)$. Pri obdelavi neke zahteve r_t se WFA premakne iz konfiguracije C_{t-1} v C_t, ki vsebuje r_t in skuša minimizirati

$$w_t(C_t) + d(C_{t-1}, C_t).$$

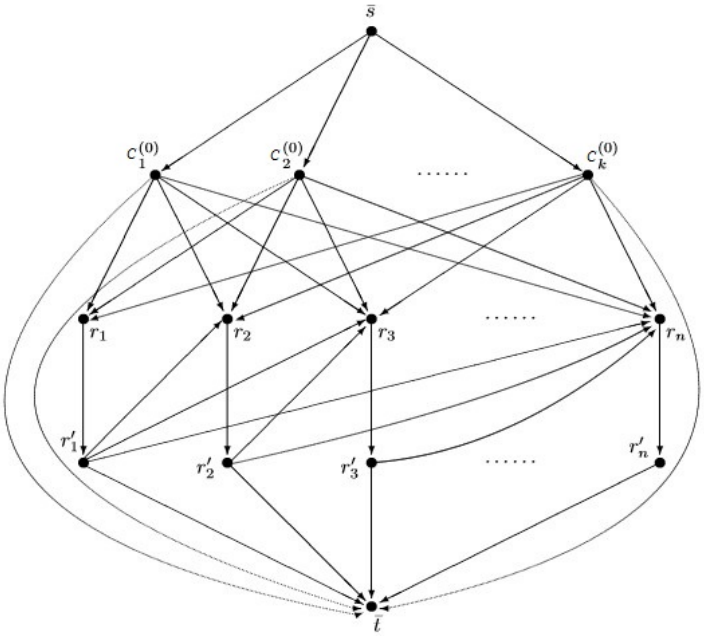
Opazimo lahko, da algoritem pri izbiri naslednjega strežnika pravzaprav upošteva trenutne razdalje (podobno kot požrešni algoritem) in zgodovino preteklih zahtev, s tem, da se poskuša prilagoditi na idealna pretekla zaporedja obdelave v upanju, da se bodo le ta ponovila. Njegova glavna prednost pred požrešno rešitvijo je torej, da išče vzorce, vendar ga to naredi šibkejšega pri popolnoma naključnih zaporedjih zahtev [5].

2.2 Statični algoritem

Kot smo že prej omenili, je bistvena značilnost statičnih algoritmov, da imajo dostop do vseh zahtev vnaprej, tako rekoč vidijo v prihodnost, kar jim omogoča, da dosežejo optimalno odzivanje na zahteve in posledično dosežejo minimalno ceno. Na prvi pogled se nam mogoče zdi problem podoben problemu trgovskega potnika in posledično nerešljiv v polinomskem času, vendar se izkaže, da se problem dobro prevede v problem iskanja maksimalnega pretoka za minimalno ceno v usmerjenem grafu.

2.2.1 Pretvorba v usmerjen graf

Usmerjen graf, na katerem se rešuje problem iskanja maksimalnega pretoka za minimalno ceno, ima $2n + k + 2$ vozlišč. Kot je razvidno iz slike 2.1, ima graf izvorno vozlišče s, ponorno vozlišče t in še tri dodatne nivoje vozlišč.



Slika 2.1: Usmerjen graf min cost max flow problema [7]

Besedna zveza nivo vozlišč v tem primeru predstavlja eno izmed horizontalnih zaporedij vozlišč med vozliščema s (izvorno vozlišče) in t (ponorno vozlišče). Nivo vozlišč, ki ga dosežemo ob potovanju po usmerjenih povezavah iz izvornega vozlišča predstavlja začetno konfiguracijo $C^{(0)}$, torej $C_j^{(0)}$ predstavlja začetno lokacijo j-tega strežnika. Ostala dva nivoja predstavljata celotno zaporedje zahtev, tako vozlišče r_p, kot $r_p^{'}$ predstavlja lokacijo p-te zahteve. Opazimo lahko, da niso vsa vozlišča povezana. Poljubno vozlišče r_p je povezano samo s sebi prirejenim vozliščem $r_p^{'}$. Poleg tega povezava med $r_p^{'}$ in r_q obstaja samo, če velja $q > p$. Vse povezave imajo kapaciteto 1. Cena povezav,ki izstopajo ali vstopajo v t ali s, je 0. Povezava med r_p in $r_p^{'}$ ima ceno $-L$, kjer L predstavlja poljubno veliko pozitivno število (biti mora bistveno večje od ostalih povezav, idealno pa največje možno pozitivno število). Vse ostale povezave imajo ceno, enako razdalji med točkami metričnega prostora. Opazimo lahko, da je največji pretok enak številu strežnikov, torej k ter da bo iz vsakega vozlišča $C_j^{(0)}$ potekal neodvisen enotski tok do ponora t. Sedaj je treba samo še pognati algoritem iskanja maksimalnega pretoka za minimalno ceno iz izvora s. Algoritem pri takem grafu privede do prave rešitve, ker prisili tok skozi najcenejše $-L$ povezave in posledično obdela vse zahteve [5].

2.2.2 Iskanje pretoka

Kot pri marsičem obstaja tudi pri iskanju pretoka več metod. V viru [5] je opisana metoda, pri kateri začnemo s tokom, ki nima maksimalne vrednosti, vendar ima minimalno ceno med tistimi s to vrednostjo. Nato v vsakem naslednjem koraku povečamo vrednost toka tako, da bo imel še vedno minimalno ceno med tokovi z enako vrednostjo. Po določenem številu korakov se doseže maksimalni pretok za minimalno ceno. Algoritem ima časovno zahtevnost $O(k \cdot |V|^2)$, kjer je $|V|$ število vozlišč $|V| = 2n + k + 2$
Naš izbrani algoritem izrablja obratno metodo, torej iskanje minimalne cene pri konstantnem maksimalnem pretoku. Izberemo poljubni maksimalni tok in vsem povezavam na toku spremenimo smer ter pomnožimo ceno z -1, tako

dobimo graf ostankov (*angl. residual graph*). Na grafu poženemo algoritem Bellman-Ford-Moore [3]. Ob koncu ga poženemo še enkrat, da preverimo, ali v grafu obstajajo negativni cikli. Ob pogoju, da v grafu ni negativnih ciklov, smo prišli do končne rešitve. Če ti obstajajo, jih bo treba odstraniti. Cikel se odstrani s tem, ko po njem porinemo vrednost minimalne kapacitete izmed vseh povezav. Ker imajo naše povezave vedno enako vrednost je porinjena vrednost vedno 1. V praksi to pomeni, da povezave na našem grafu ostankov spremenijo usmeritev. Na novo nastalem grafu ponovno poženemo algoritem Bellman-Ford in ves proces ponavljamo, dokler nam le-ta ne potrdi, da v grafu ni negativnih ciklov. Pri iskanju izbir strežbe (se pravi informacije o tem, kateri strežnik bo obdelal katero zahtevo) se nato sprehodimo iz posameznega vozlišča zahteve proti ustreznemu vozlišču strežnika po sledeh posameznega enotskega toka (glej podpoglavje 3.4). Časovna zahtevnost uporabljenega algoritma je $\frac{|V|}{4} \cdot |V| \cdot |E|$, z drugimi besedami $O(|V|^2 \cdot |E|)$, kjer je $|V|$ število vozlišč $|E|$ pa število povezav. Ta algoritem smo si izbrali, ker je bolj splošen in se ga da prilagoditi na grafe z različnimi pretoki. Grafi s pretokom, večjim od 1 nam sicer pri statičnem algoritmu niso uporabni, vendar smo se za omenjen algoritem vseeno odločili iz čiste vedoželjnosti.

2.3 Pomen obeh pristopov

Spoznali smo sprotni in statični algoritem. Prvi je v izvedbi navadno veliko hitrejši od drugega in njegova prostorska zahtevnost je nižja. Drugi namreč za svoje delovanje potrebuje podatkovne strukture za predstavitev več grafov. Zagotovo je statični algoritem uspešnejši, saj nam vedno vrne optimalno rešitev. Prav iz tega razloga je kakršnakoli primerjava med njima nesmiselna. Eden nam predstavlja absolutni ideal, ki ga najverjetneje nikoli ne bo mogoče preseči (razen ob iznajdbi algoritma, ki napoveduje prihodnost). Sprotni algoritem si prizadeva najti praktične rešitve na problem v realnih situacijah. Posledično nam statični problem služi le pri objektivnem ocenjevanju delovanja sprotnih rešitev in ga le redko (če sploh) vidimo uporabljenega v praksi.

Poglavje 3

Implementacija in testiranje

Algoritme, opisane v prejšnjem poglavju, smo želeli preizkusiti v praksi, da bi ugotovili pravilnost njihovega delovanja pri različnih vhodih in primerjali teoretične časovne zahtevnosti z izmerjenimi. V ta namen smo ustvarili simulator in v njem pognali naslednje algoritme:

- požrešni algoritem,
- harmonični algoritem,
- algoritem delovne funkcije,
- optimalni algoritem.

Del simulatorja je bil tudi razred `Ks.java`, v katerem so vse metode, ki poganjajo naše algoritme. Njegov konstruktor kot parametre prejme število strežnikov, število zahtev in način delovanja, ki določi vzorec pojavljanja zahtev v metričnem prostoru.

```
public Ks(int k, int r, int mode) throws FileNotFoundException {
    System.out.printf("Init k=%d r=%d\n", k, r);

    req = new Point[r];
    ks = new Point[k];

    char c = (char) 1000;

    for (int i = 0; i < 11; i++) {
```

```
        for (int j = 0; j < 11; j++) {
            cordnames[i][j] = (char) String.valueOf(c).charAt(0);
            cordpoints[i][j] = new Point(i, j);
            cordpoints[i][j].name = cordnames[i][j];
            np.put((int) cordnames[i][j], cordpoints[i][j]);
            c++;
        }
}
```

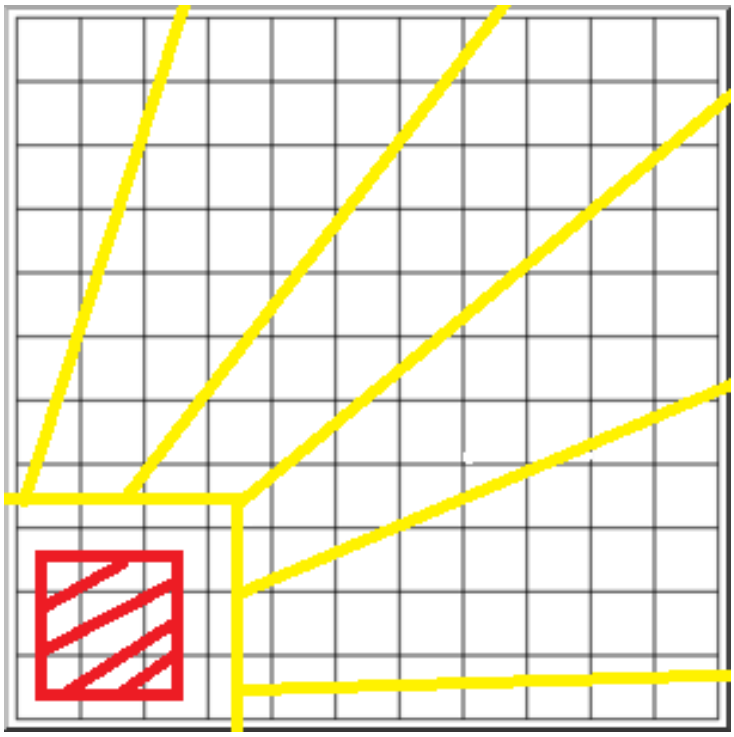
Ustvarili smo dva seznama objektov razreda `Point.java`, v katerih bodo shranjeni podatki o naših strežnikih in zahtevah. Odločili smo se, da se bodo zahteve pojavljale na polju velikosti 11×11, ker nam takšna velikost bistveno ne obremenjuje algoritmov z večjo prostorsko zahtevnostjo. Polje smo ustvarili in vsaki njegovi točki dodelili Unicode vrednost od 1000 naprej, kar nam služi predvsem pri ročnem preverjanju manjših zaporedij obdelave pri izbranih vhodih. To nam je služilo predvsem pri preverjanju pravilnosti metod pri začetnih fazah razvoja. Vrednost 1000 je bila izbrana, ker je vseh 121 možnih znakov po 1000 mogoče izpisati v konzoli našega razvojnega okolja. Končno smo lahko s pomočjo slovarja `np`, prevajali imena točk v njihove razredne vrednosti in posledično koordinate.

```
int rNum1 = 0, rNum2 = 0;
//Field is 11 x 11 = 121 different points
int f = 0;
for (int i = 0; i < r; i++) {
    f = ThreadLocalRandom.current().nextInt(0, 10);
    switch (mode) {

        //Points appear completely randomly.
        case 1:
            rNum1 = ThreadLocalRandom.current().nextInt(0, 10 + 1);
            rNum2 = ThreadLocalRandom.current().nextInt(0, 10 + 1);
            break;
```

Spremenljivka `mode` služi predvsem za izbiro načina postavitev zahtev in strežnikov. Za naše potrebe smo uporabili 4 načine (pri vseh smo se odločili za naključno začetno postavitev strežnikov):

- Zahteve prihajajo popolnoma naključno.

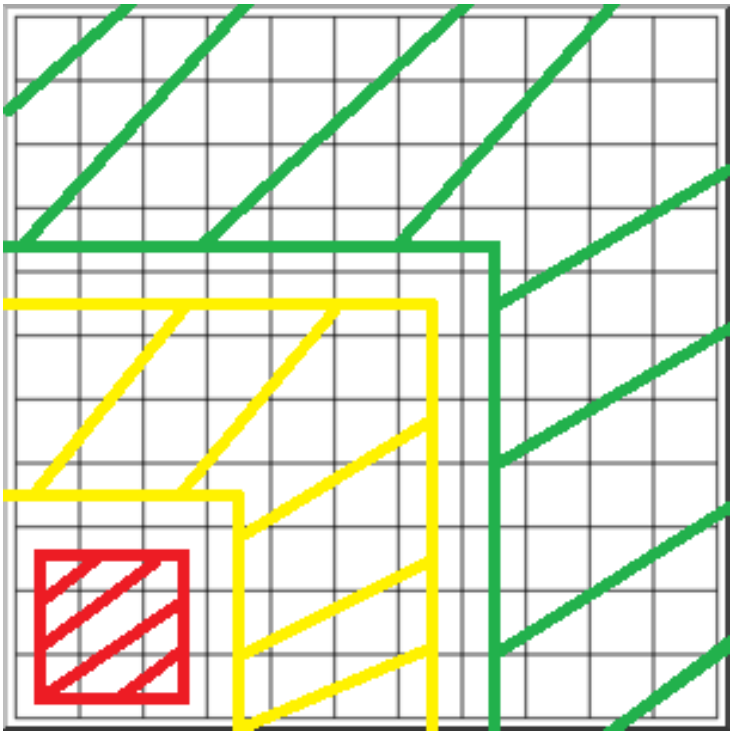
- 70 % zahtev se sprva pojavi naključno v celicah od cordpoints[0][0] do cordpoints[2][2]. Ostalih 30 % se pojavi naključno na preostanku polja.



Slika 3.1: Abstraktna slika dvo dimenzionalne tabele cordpoints. 70% zahtev se sprva pojavlja v rdeče pobarvanem območju, ostale v rumenem.

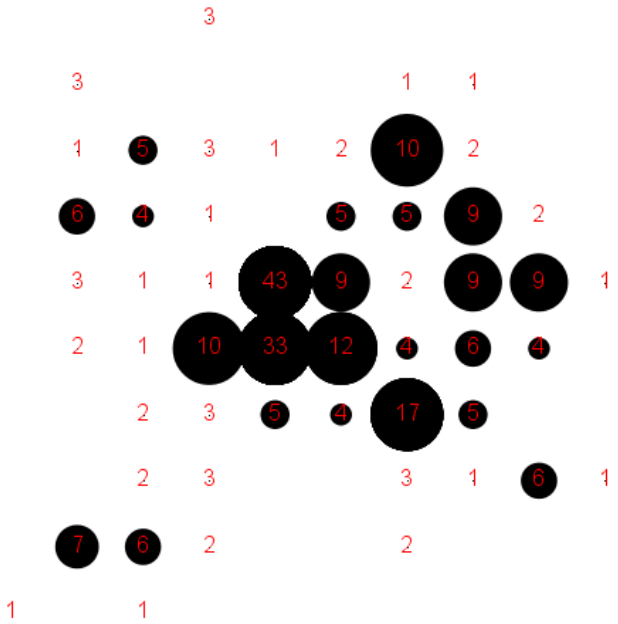
- 20 % zahtev se sprva pojavi naključno na celicah od cordpoints[0][0] do cordpoints[2][2]. Nato se 60 % zahtev pojavi naključno na celicah od cordpoints[3][3] do cordpoints[7][7]. Končno se zadnjih 20% zahtev pojavi naključno na celicah od cordpoints[7][7] do cordpoints[10][10].

- Zahteve se pojavljajo kot v primeru iz realnega življenja. Izbrali smo si tri vzorce podatkov klicev 911 v ameriškem mestu Seattle, in sicer popolnoma naključno za datume 24. 1. 2016, 17. 7. 2010 in 1. 8. 2010. Uporabili smo prvih 185 klicev dneva, razporejenih kronološko. Podatke je bilo treba normalizirati iz geolokacij na polje 11×11 [2].



Slika 3.2: Abstraktna slika dvo dimenzionalne tabele cordpoints. 20% zahtev se sprva pojavi v rdečem, nato 60% v rumenem in končno 20% v zelenem območju.

Testiranje je potekalo v okviru naše `KServer.java` glavne datoteke, kjer se je nahajala metoda `main`. Odločili smo se za sočasno testiranje požrešne, WFA in optimalne rešitve s številom strežnikov enako 2 ter številom zahtev enako 185. Tako število zahtev smo si izbrali, ker to predstavlja ravno en dan klicev na 911 v našem realnem primeru in tudi preveč ne obremenjuje naših počasnejših algoritmov. Poleg tega je po vsaki izvedbi vseh treh algoritmov potekala ponovna naključna postavitev strežnikov in zahtev. To smo ponovili 20 krat za vsak način delovanja. Ob vsaki uspešni izvedbi algoritma, se je na standardni izhod izpisala vrednost prepotovane razdalje strežnikov. Nato se je za vsako zaporedje poganjanja izračunalo faktor ρ in za vsak način delovanja povprečje faktorja ρ med posameznimi zaporedji izvajanja. Odločili



Slika 3.3: Normaliziran prikaz vseh 185 klicev dne 24.1.2016 [9]

smo se, da ne bomo uporabljali faktorja $\Phi(C_0)$, saj večkratna naključna postavitev strežnikov izključi možnost, da bodo začetne postavitve naklonjene enemu izmed algoritmov.

3.1 Požrešni algoritem

Izvajanje požrešnega algoritma je možno ob klicu metode `startGreedy`.

```
public void startGreedy(boolean v) {

    String solution = "";

    long startTime = System.nanoTime();

    for (Point r : req) {
        solution = solution + closest(r);

    }
```

```
    long estimatedTime = System.nanoTime() - startTime;

    System.out.println(solution + " dist");
    System.out.println(Math.round(globaldistC * 100) / 100);
    System.out.println(estimatedTime + " ns");
    resetK();

    if (v) {
        visualise(solution);
    }

}
```

Metoda sprva nastavi spremenljivko `solution`, ki predstavlja rešitev zaporedja zahtev v obliki zaporedja imen strežnikov, ki bodo zahteve obdelali. Na primer, če imamo pri primeru k=2 in r=3 strežnika z imeni A in B ter želimo povedati, da prvi dve zahtevi obdela strežnik A, zadnjo pa B, bi bila naša rešitev AAB. Imena so strežnikom dodeljena glede na njihovo prvotno začetno lokacijo. Jedro metode predstavlja zanka, ki iterira po zahtevah in preko pomožne metode `closest`, ki prejme zahtevo kot argument in vrne ime strežnika, ki bo obdelal dano zahtevo.

```
private char closest(Point r) {
    char c = (char) ks[0].name;
    Point x = r;
    double dist = Integer.MAX_VALUE;
    double mindist = Integer.MAX_VALUE;

    for (Point k : ks) {
        dist = calcDist(k.nx, k.ny, r.x, r.y);
        if (dist < mindist) {

            mindist = dist;
            c = (char) k.name;
            x = k;
        }
    }
    globaldistC = globaldistC + mindist;

    x.nx = r.x;
    x.ny = r.y;
    return c;
}
```

Poleg tega nam metoda s pomočjo statične razredne spremenljivke `globaldistC` omogoča štetje celotne prepotovane razdalje strežnikov. Posamezen razred `Point` vsebuje tudi pomožni spremenljivki `nx` in `ny`, ki služita za beleženje trenutne lokacije strežnikov. Končno nam prvotna metoda omogoča tudi štetje pretečenega časa med izvajanjem in ob potrebi ustrezno vizualizacijo, ki je implementirana preko javne knjižnice `StdDraw.java` [9].

3.2 Harmonični algoritem

Izvajanje harmoničnega algoritma je možno ob klicu metode `startHarmonic`. Ta je sestavljena enako kot metoda za izvajanje požrešnega algoritma, torej ima zanko, ki gre preko vseh zahtev in vsako posreduje metodi `closestH`.

```
private char closestH(Point r) {
    char c = (char) ks[0].name;
    Point x = r;
    double totaldist = 0;

    double N = 0;
    for (Point k : ks) {
        N=N+(1/calcDist(k.nx, k.ny, r.nx, r.ny));
    }

    for (Point k : ks) {
        k.prob = 1/(N*calcDist(k.nx, k.ny, r.nx, r.ny));
    }
```

Prvi del metode služi predvsem za določitev verjetnosti izbire posameznega strežnika s pomočjo načina, opisanega pri teoretičnih osnovah harmonične rešitve. Najprej se s pomočjo vseh strežnikov izračuna faktor N, nato se vsakemu strežniku dodeli verjetnost izbire na osnovi formule.

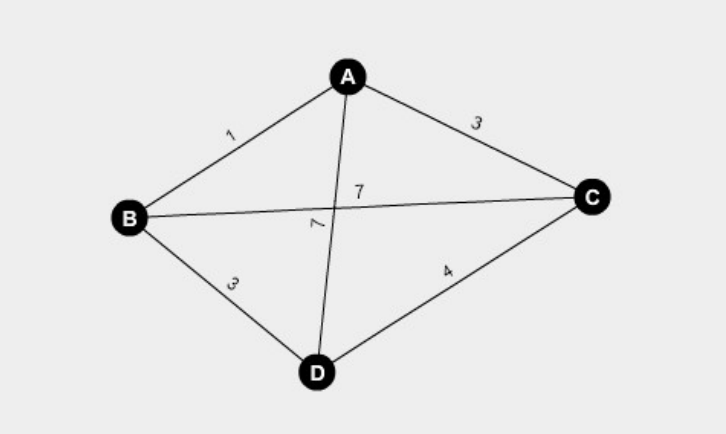
```
    Arrays.sort(ks, pointComparator);

      double roll = Math.random();
      double total = 0;
      for(Point k : ks){
          total+=k.prob;
          if(roll<=total){
              //Choose this one
              x=k;
```

```
                c=(char)k.name;
                globaldistC+=calcDist(k.nx,k.ny,r.x,r.y);
                break;
            }
        }

    x.nx = r.x;
    x.ny = r.y;
    return c;
}
```

V drugem delu uredimo seznam strežnikov, od tistega z največjo možnostjo da bo izbran, do tistega z najmanjšo. Poslužujemo se že vgrajene metode za urejanje, ki ji kot argument damo ustrezno prilagojen primerjalnik. Sledi klic metode za generiranje naključnih števil, ki nam vrne vrednost od vključno 0.0 do izključno 1.0. Nato seštevamo vrednosti verjetnosti posameznih strežnikov in preverjamo, v kateri interval pade generirano naključno število. Ob ugotovitvi ustreznega številskega intervala, ki vsebuje naše generirano število, se zanka prekine. Metoda končno vrne ime izbranega strežnika, ustrezno spremeni njegovi spremenljivki `nx` in `ny` ter razdaljo med njim in zahtevo prišteje h `globaldistC`.

3.3 Algoritem delovne funkcije

Ker je algoritem delovne funkcije težje razumljiv, bomo sprva opisali delovanje naše implementacije na preprostem primeru. Denimo, da imamo graf kot je na naslednji sliki. V našem primeru imamo dva strežnika, ki začneta



Slika 3.4: Primer grafa

na vozliščih a in b, sledi zaporedje treh zahtev D,C in A. Ustvarimo matriki w in s velikosti $\binom{n}{k} \times (r+1)$. Mali n nam predstavlja število vozlišč grafa, k število strežnikov in r število zahtev.

zahteva	indeks	ab	ac	ad	...	cd
$\emptyset$	0	0	7	3	...	6
d	1	6	7	3	...	6
c	2	12	7	11	...	6
a	3	12	7	11	...	14

Slika 3.5: Abstrakcija matrike w

Matrika w v sebi hrani cene premikov iz ene postavitve v drugo. Prva vrstica (w_0) , nam pove minimalno ceno za premik iz začetne postavitve ab v vse ostale. Na primer cena za premik iz ab v ac je enaka razdalji med

vozliščema c in b. Vrednosti ostalih vrstic se računa po formuli,

$$w_i(x,y) = min\{w_{i-1}(z,y) + d(z,x), w_{i-1}(x,z) + d(z,y)\}$$

za $i = 1, 2, ..., n$, kjer niz xy predstavlja poljubno postavitev strežnikov z pa vozlišče trenutne zahteve. Druga matrika s hrani imena strežnikov, izbranih pri računanju vrednosti posameznih celic matrike w, denimo, da je v zgornji formuli imela $w_{i-1}(zy)+d(z,x)$ najmanjšo vrednost. V takem primeru bi bila vrednost $s_{i-1}(xy)$ enaka x, torej enaka izbiri strežnika za obdelavo zahteve. Pri ugotavljanju izbir algoritma uporabimo spodnjo tabelo.

Stara postavitev	Zahteva	Premakni strežnik	Cena	Nova postavitev
ab	d	$s[d][ab] = b$	$d(b,d) = 3$	ad
ad	c	$s[c][ad] = a$	$d(a,c) = 3$	cd
cd	a	$s[a][cd] = c$	$d(a,c) = 3$	ad

Slika 3.6: Primer postopka pridobivanja rešitve

Če torej začnemo pri postavitvi ab z zahtevo d, bo to obdelal strežnik $s[d][ab]$ za ceno $d(b,d)$. Končno postane nova postavitev ad. Postopek nadaljujemo do zadnje spremembe postavitve.
Izvajanje algoritma delovne funkcije je možno ob klicu metode `startWFA`.

```
void startWFA(boolean v) {
    String solution = "";
    int xvrstica = binomial(121, ks.length).intValue();
    int yvrstica = req.length;
    wfa_w = new double[xvrstica][yvrstica + 1];
    wfa_s = new String[xvrstica][yvrstica];
    wfa_combos = new String[xvrstica];
    wfa_reqs = new String[yvrstica + 1];
    combc = 0;

    String starter = "";
    long startTime = System.nanoTime();

    for (int i = 0; i < 11; i++) {
        for (int j = 0; j < 11; j++) {
            starter += cordnames[i][j];
```

```
        }
    }

    String arr[] = starter.split("");
    //FIlls up wfa_combos with all the combinations
    //Also maps combs to indexes - ci
    setCombinations(arr, arr.length, ks.length);
    Arrays.sort(wfa_combos);
```

Najprej dodelimo vrednosti spremenljivkama `xvrstica` in `yvrstica`. Ti dve služita kot dolžina in širina pri inicializaciji naših statičnih razrednih tabel `wfa_w` in `wfa_s`, ki sta ekvivalentni prej opisanima matrikama w in s. Sledi vrsta inicializacij statičnih razrednih spremenljivk. `Wfa_combos` hrani vse možne kombinacije točk polja v obliki nizov. `Wfa_reqs` hrani imena zahtev v kronološkem vrstnem redu. S pomočjo pomožne metode `setCombinations`, ki kot argument prejme tabelo imen vseh točk `arr`, njeno dolžino in dolžino tabele vseh strežnikov, napolnimo tabelo kombinacij ter jo z vgrajeno metodo uredimo po abecednem vrstnem redu. Imena točk znotraj posameznih celic tabele kombinacij, so tudi urejena po abecednem vrstnem redu. Poleg tega nam metoda napolni statični razredni slovar `ci`, ki kot ključ prejme niz kombinacije, vrne pa njegov indeks v `wfa_combos`. To nam omogoča hitro ugotavljanje nahajališča posamezne kombinacije. Končno s pomočjo vgrajene metode uredimo tabelo `wfa_combos` po abecednem vrstnem redu.

```
    //Set up request values
    for (int k = 0; k < wfa_reqs.length; k++) {
        if (k == 0) {
            wfa_reqs[k] = "\n";
        } else {
            wfa_reqs[k] = "" + (char) req[k - 1].name;
        }
    }

    //Find points corresponding to the first configuration ABC and place them on position
    String sk = "";
    for (Point p : ks) {
        sk += (char) p.name;
    }

    sk = sortString(sk);
```

```
int first = ci.get(sk);
String holder = "";

//Switching nth column with 0th
holder = wfa_combos[first];
wfa_combos[first] = wfa_combos[0];
ci.put(wfa_combos[0], first);
wfa_combos[0] = holder;
ci.put(holder, 0);
```

V zgornjem segmentu dodelimo vrednosti tabeli `wfa_reqs`, poiščemo trenutno nahajališče začetne postavitve strežnikov in jo zamenjano s postavitvijo na prvem mestu v tabeli. S tem si bistveno olajšamo izvajanje nadaljnjih algoritmov.

```
//Getting first row done
for (int k = 0; k < wfa_combos.length; k++) {
    wfa_w[k][0] = wfa_D(wfa_combos[0], wfa_combos[k]);
}

//Getting the rest done
for (int i = 1; i < wfa_reqs.length; i++) {
    for (int k = 0; k < wfa_combos.length; k++) {
        wfa_w[k][i] = minWFA(k, i);
    }
}

//Getting final solution
String current = wfa_combos[0];
String combos = current;

for (int k = 1; k < wfa_reqs.length; k++) {
    int j = ci.get(current);
    solution = solution + wfa_s[j][k - 1];
    current = changeCombination(wfa_s[j][k - 1], current, wfa_reqs[k]);
    combos = combos + " " + current;
}
long estimatedTime = System.nanoTime() - startTime;
String[] combo_sequence = combos.split(" ");
```

Lotimo se iskanja vrednosti prve vrstice tabele cen premikanja. Sprehodimo se po celotni prvi vrstici in s pomočjo metode `wfa_D`, ki kot argument prejme prvo kombinacijo in k-to, nastavimo ceno vsaki celici. Takoj za tem se lotimo nastavljanja cen preostanku tabele. Sprehodimo se po vsaki celici in ji dodelimo ceno s pomočjo metode `minWFA`, ki kot argument prejme indeks

trenutne vrstice in stolpca. Končno s pomočjo prejšnjega na tabeli opisanega postopka, dobimo niz kombinacij nahajališč strežnikov, katere predstavljajo izbire strežnikov pri obdelavi posameznih zahtev.

3.3.1 Metoda `wfa_D`

```
private double wfa_D(String abc, String xyz) {
    double val = 0;
    double mindist;
    double curdist;
    Point p1;
    Point p2;
    char c1;
    char c2;

    int[] choicesABC = new int[abc.length()];
    int[] choicesXYZ = new int[xyz.length()];

    for (int i = 0; i < abc.length(); i++) {
        mindist = Double.MAX_VALUE;
        curdist = Double.MAX_VALUE;
        int curindx1 = 0;
        int curindx2 = 0;

        for (int k = 0; k < abc.length(); k++) {
            if (choicesABC[k] == 1) {
                continue;
            }

            for (int j = 0; j < xyz.length(); j++) {

                if (choicesXYZ[j] == 1) {
                    continue;
                }

                c1 = abc.charAt(k);
                c2 = xyz.charAt(j);
                p1 = np.get((int) c1);
                p2 = np.get((int) c2);

                curdist = calcDist(p1.x, p1.y, p2.x, p2.y);
                if (curdist < mindist) {
                    mindist = curdist;
                    curindx2 = j;
                    curindx1 = k;
```

```
                }

            }
        }

        choicesABC[curindx1] = 1;
        choicesXYZ[curindx2] = 1;
        val += mindist;

    }
    return val;
}
```

Metoda deluje po principu iskanja najbližjih dveh točk med postavitvama `abc` in `xyz`. Iskanje poteka v notranjih dveh zankah. Ko najdemo najbližji točki ju, po izstopu iz druge zanke, označimo v tabelah `choicesABC` in `choicesXYZ`, kot že izbrani. Že izbrane točke ne sodelujejo pri nadaljnjih iteracijah. Zunanja zanka se izvede tolikokrat, kot je dolžina naših nizov, kar zadostuje, saj ob vsaki iteraciji izberemo en par točk. Poleg tega vsakič znova nastavi vrednosti potrebnih spremenljivk.

3.3.2 Metoda `minWFA`

```
private double minWFA(int j, int i) {
    double val = 0;
    String c = "";
    String bestc = "";
    String abc = wfa_combos[j];
    String sr = wfa_reqs[i];
    double best_total = Double.MAX_VALUE;

    if (abc.contains(sr)) {
        wfa_s[j][i - 1] = sr;
        return wfa_w[j][i - 1];

    } else {

        for (int k = 0; k < abc.length(); k++) {

            c = "" + abc.charAt(k);
            String newc = createCombo(abc, k, sr);

            double dist = Double.MAX_VALUE;
            double wv = Double.MAX_VALUE;
```

```
            //Sorting our string alphabetically
            newc = sortString(newc);

            //Use dict to find comb id instantly
            int h = ci.get(newc);

            Point p1 = np.get((int) (c.charAt(0)));
            Point p2 = np.get((int) (sr.charAt(0)));
            dist = calcDist(p1.x, p1.y, p2.x, p2.y);
            wv = wfa_w[h][i - 1];
            if ((dist + wv) < best_total) {
                best_total = dist + wv;
                bestc = c;
            }

        }
        wfa_s[j][i - 1] = bestc;
        return best_total;
    }

}
```

Metoda najprej pridobi vrednosti j-te kombinacije in i-te zahteve. Nato preveri, če trenutna kombinacija vključuje i-to zahtevo. Ob pogoju, da je trenutna zahteva del trenutne kombinacije oziroma postavitve, metoda vrne privzeto vrednost trenutne celice tabele cen `wfa_w`, kar je pri javi vedno 0. Še prej pa dodeli, trenutni celici tabele izbir `wfa_s`, vrednost trenutne zahteve, katera je v tem primeru enaka vrednosti strežnika, s katerim si deli nahajališče. Če trenutna zahteva ni del trenutne kombinacije, se izvede `else` odsek. V sledeči zanki poiščemo, katera izbira strežnika je najcenejša. Spremenljivka `c` drži vrednost trenutnega poskusa izbiranja strežnika. Metoda `createCombo` nam vrne niz `abc`, ki ima kot k-to črko ime trenutne zahteve. Novo pridobljen niz `newc` se nato uredi s pomočjo vgrajene metode in pridobi njegov indeks iz slovarja `ci`. Sledi računanje vrednosti delovne funkcije in razdalje, pri katerem je spremenljivka `wv` enaka vrednosti delovne funkcije `dist` pa razdalji. Ob zadnji izvedbi zanke se tabeli izbir `wfa_s` dodeli vrednost izbrane rešitve. Končno metoda vrne tudi ceno rešitve.

3.3.3 Pretvorba v ustrezno obliko rešitve

```
double distvalue = 0;
  solution = "";
  String[] prev = combo_sequence[0].split("");
  String[] oldnames = combo_sequence[0].split("");
  for (int k = 1; k < combo_sequence.length; k++) {
      for (int j = 0; j < prev.length; j++) {
          if (!combo_sequence[k].contains(prev[j])) {
              solution += oldnames[j];
              //najdi razliko med prev in combo_sequence[k]
              for (int i = 0; i < prev.length; i++) {
                  if (i != j) {
                      combo_sequence[k] = combo_sequence[k].replace(prev[i], "");
                  }
               }
              Point A = np.get((int) prev[j].charAt(0));
              Point B = np.get((int) combo_sequence[k].charAt(0));
              distvalue += calcDist(A.x, A.y, B.x, B.y);
              prev[j] = combo_sequence[k];
              break;
          }
      }
  }
```

Tabelo nizov kombinacij strežnikov moramo sedaj samo še pretvoriti v ustaljeno obliko rešitve, torej v niz imen strežnikov. Najprej nastavimo vrednosti tabelama nizov `prev` in `oldnames`. Prva nam služi kot trenutna kombinacija, ki jo obdelujemo, druga predstavlja imena prvotnih strežnikov, ki jih bomo uporabljali pri gradnji spremenljivke `solution`. V naslednjih zankah se sprehodimo po vsaki kombinaciji naše rešitve in jo primerjamo s `prev`. S pomočjo pomožne metode `contains`, najdemo, katera črka niza `prev` se razlikuje od trenutne kombinacije. Z najglobljo zanko nastavimo vrednost `combo_sequence` na točko, ki jo je obdelal strežnik. Na koncu še izračunamo razdaljo med točkama in nastavimo novo vrednost spremenljivki `prev`.

3.4 Optimalni algoritem

Izvajanje optimalnega algoritma je možno ob klicu metode `min_max_Offline`.

```
void min_max_Offline(boolean b) {
    String solution = "";
    int negativeCycleC=0;
    long startTime = System.nanoTime();
    //Generate input for min cost max flow algorithm
    int nodeNumber = 2 * req.length + ks.length + 2;
    int[][] cap = new int[nodeNumber][nodeNumber];
    double[][] edge_cost_map = new double[nodeNumber][nodeNumber];

    //connect start to ks
    for (int i = 1; i <= ks.length; i++) {
        cap[0][i] = 1;
        edge_cost_map[0][1] = 0;
        edge_cost_map[1][0] = 0;
    }

    //connecting ks to first layer reqs
    for (int i = 1; i <= ks.length; i++) {

        cap[i][nodeNumber - 1] = 1;
        edge_cost_map[0][1] = 0;
        edge_cost_map[1][0] = 0;

        for (int k = 1; k <= req.length; k++) {
            cap[i][ks.length + k] = 1;
            edge_cost_map[i][ks.length + k] =
                calcDist(ks[i - 1].x, ks[i - 1].y, req[k - 1].x, req[k - 1].y);
            edge_cost_map[ks.length + k][i] = -1 * edge_cost_map[i][ks.length + k];

        }

    }
    //connecting first layer reqs to second layer reqs
    int e = ks.length + 1;
    for (int k = e; k < e + req.length; k++) {
        cap[k][k + req.length] = 1;
        edge_cost_map[k][k + req.length] = -1000;
        edge_cost_map[k + req.length][k] = 1000;
    }

    Point[] ps = new Point[1+ks.length+req.length];

    int ctr=1;
    for(int k = req.length-1;k>-1;k--){
```

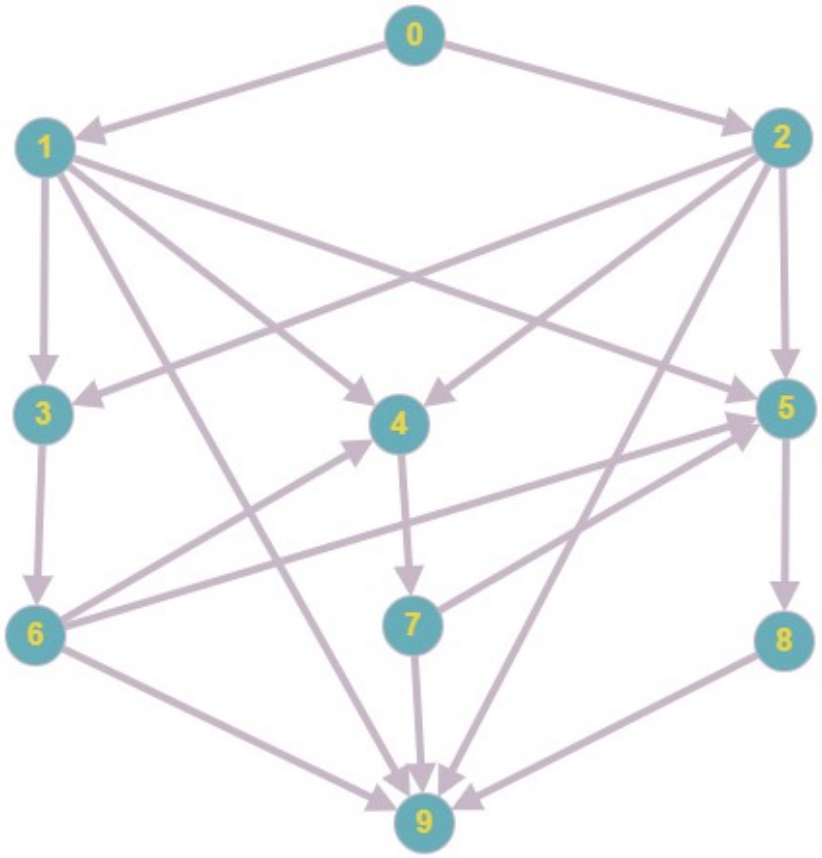
```
        ps[ps.length-ctr]=req[k];
        ctr++;
    }

    //connecting second layer of reqs to sink and to first layer of reqs
    int n = e + req.length;
    int c = 0;
    int s = 0;
    for (int k = n; k < req.length + n; k++) {
        c++;
        cap[k][nodeNumber - 1] = 1;
        for (int j = e + c; j < req.length + e; j++) {
            cap[k][j] = 1;

            edge_cost_map[k][j] = calcDist(req[s].x,req[s].y,ps[j].x,ps[j].y);
            edge_cost_map[j][k] = -1*edge_cost_map[k][j];


        }

        s++;
    }
```

Za vse operacije nad grafi se poslužujemo matrik sosednosti (*angl. adjecency matrix*). Najprej ustvarimo dve, in sicer `cap`, ki bo prikazovala obstoj začetnih povezav med vozlišči in `edge_cost_map`, ki bo opisovala cene posameznih povezav. V naslednjih korakih poteka postopek gradnje grafa, iz Slike 2.1. Ob nastavitvi matrike cen povezav upoštevamo tudi obratne povezave, ki niso del Slike 2.1. Z besedno zvezo obratna povezava označujemo povezave med istimi vozlišči z obratno usmerjenostjo. Tem dodelimo vrednosti osnovnih že v Sliki 2.1, obstoječih povezav pomnožene z -1. Za veliko pozitivno število L smo si izbrali 1000, kar je pri našem polju 11×11 dovolj velika nagrada, da so tokovi prisiljeni uporabiti te povezave. Izbrali smo si jo tudi, ker je skoraj sto krat večja od največje razdalje v našem grafu, ki poteka po diagonali in meri 14.142136 enot.

```
int[][] residual = cap;

int[][] original = new int[residual.length][residual[0].length];
original = copyArray(original, residual);
```

```
//cost test must be fixed
double[][] res_cost = getCostMatrix(edge_cost_map, residual);

int maxflow = findSimpleMaxFlow(residual,ks.length);

int[][] currentFlow = getFlow(residual, original);
```

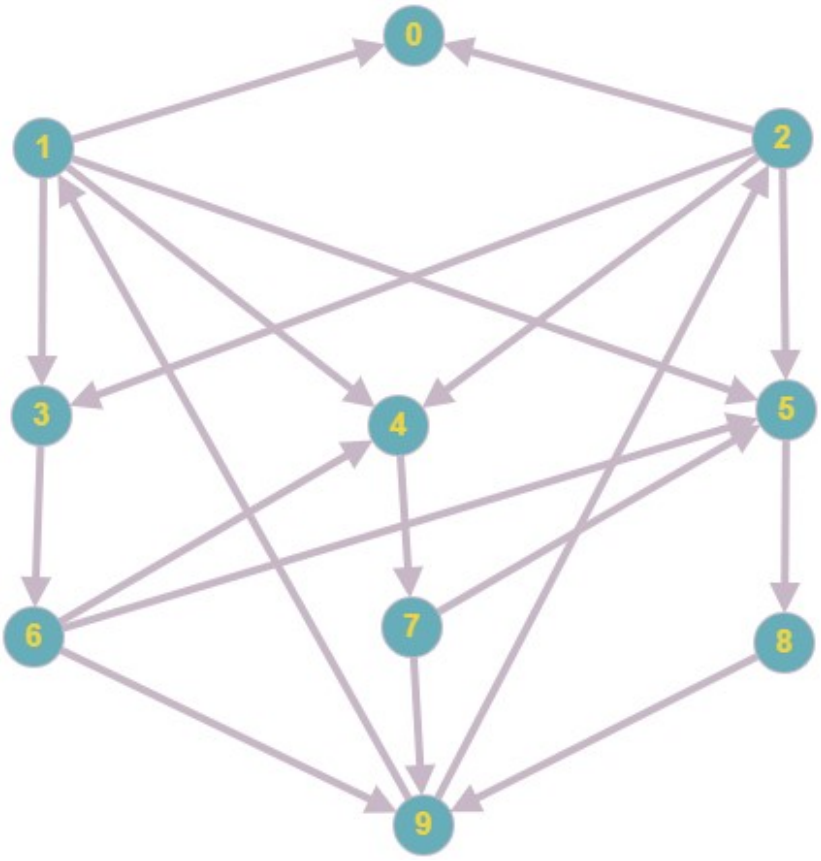
Naš graf `cap` preimenujemo v `residual` in v spremenljivko `original` shranimo kopijo tega. Na osnovi `edge_cost_map` ustvarimo cene trenutnih po-



Slika 3.7: Začetni graf pri k=2 in r=3, številke na vozliščih so enake indeksom vrstic matrik sosednosti. [6]

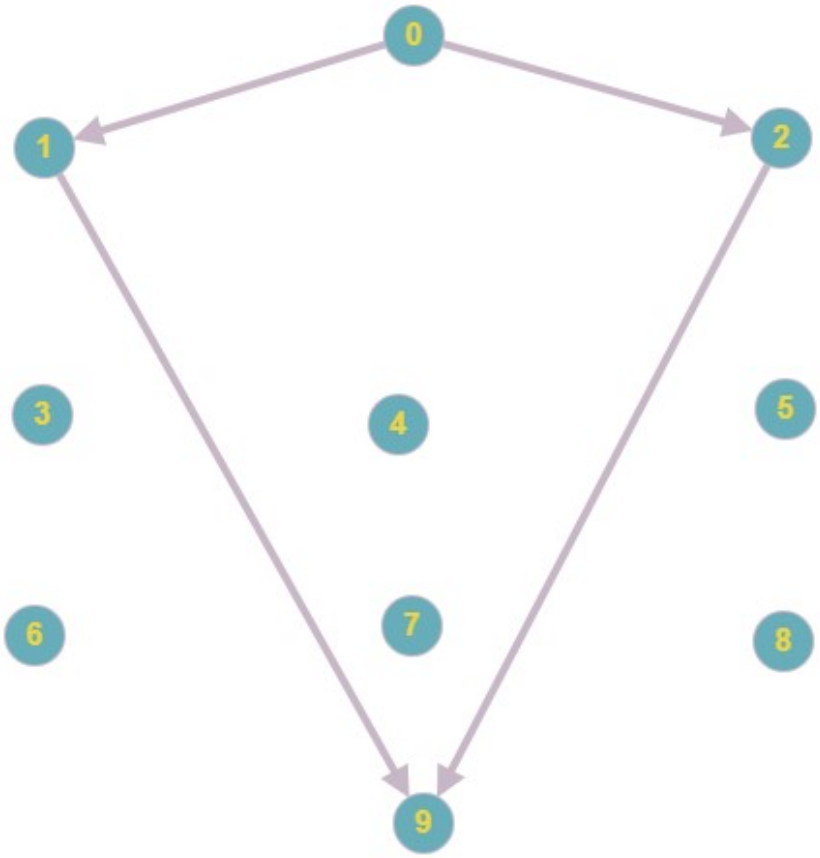
vezav v `residual` in novo matriko priredimo spremenljivki `res_cost`. Metoda `findSimpleMaxFlow` spremeni `residual` tako, da povezavam izbranim

kot potek toka od izvora do ponora, spremeni smer. Zaradi preprostosti in



Slika 3.8: Primer spremembe smeri povezav, začetna tokova potekata po vozliščih $0 \to 1 \to 9$ in $0 \to 2 \to 9$ [6].

enoličnosti našega grafa je možno najti največji tok samo s povezavo vsakega vozlišča strežnikov (na Sliki 3.4 je to prvi nivo vozlišč takoj po izvornem vozlišču) s ponornim vozliščem. Lahko bi se tudi lotili iskanja največjega toka s posebnim algoritmom, vendar bi bila to počasnejša rešitev.



Slika 3.9: Primer grafa, ki ga definira matrika sosednosti `currentFlow` [6]. Graf je primer enotskih tokov, že prej omenjenih v besedilu.

Končno funkcija `getFlow` nastavi spremenljivki `currentFlow` skoraj prazno matriko sosednosti z edinimi obstoječimi povezavami tistimi, ki prikazujejo potek trenutnega izbranega toka od izvora do ponora.

3.4.1 Glavna zanka

```
boolean negativeCycle = true;
double[] d;
int[][] pred;
int[][] cycle;

//LOOP
while (negativeCycle) {

    negativeCycle = false;

    //Make array of distances and fill it with infinity + sink node to 0
    d = new double[original.length];
    pred = new int[original.length][2];
    Arrays.fill(d, Double.MAX_VALUE / 2);
    d[d.length - 1] = 0;

    //Bellman ford d and pred is calculated
    //for every V
    for (int x = 0; x < residual.length; x++) {
        //for every E
        for (int k = 0; k < residual.length; k++) {

            for (int j = 0; j < residual.length; j++) {
                //if it's a valid edge
                if (residual[k][j] == 1) {
                    if (d[j] > d[k] + res_cost[k][j]) {
                        //predecessor of j is edge from {k to j}
                        d[j] = d[k] + res_cost[k][j];
                        pred[j] = new int[]{k, j};
                    }
                }
            }
        }
    }
```

Najprej definiramo vrsto uporabnih spremenljivk. Dokler bo `negativeCycle` imel vrednost `true`, se bo zanka iskanja in odpravljanja ciklov izvajala. Mali `d` nam predstavlja tabelo razdalj pri Bellman-Ford algoritmu [3]. `Pred` je tabela predhodnikov in `cycle` bo vsebovala naše najdene cikle. Sledi prva izvedba algoritma, ki se izvede $V \cdot E$ krat, kjer je V število vozlišč, E pa število povezav.

```
//Detecting if negative cycle exists
cycle = new int[residual.length * residual.length][2];

//for every V
outer:
for (int x = 0; x < residual.length; x++) {
    //for every E
    for (int k = 0; k < residual.length; k++) {

        for (int j = 0; j < residual.length; j++) {
            //if it's a valid edge
            if (residual[k][j] == 1) {
                if (d[j] > d[k] + res_cost[k][j]) {
                    //negative cycle exists
                    negativeCycleC++;
                    cycle[0] = new int[]{k, j};
                    pred[j] = new int[]{k, j};
                    negativeCycle = true;
                    break outer;
                }
            }
        }
    }
}

if (!negativeCycle) {
    //System.out.println("No negative cycles found!");
    break;
}
```

Ponovno izvedemo algoritem in ob kakršnikoli spremembi v tabeli razdalj `d` (kar je brez obstoja ciklov v grafu nemogoče) shranimo povezavo, ki se ji je znižala cena, v prvo celico tabele `cycle`. Takoj za tem končamo z izvajanjem obeh zank. Ob dogodku, da se tabela razdalj ni izboljšala, se bo izvedel prvi `if` stavek in imeli bomo našo rešitev.

```
else {
            int[] prvedg;
            int i = 0;

            while (!contains(cycle, pred[cycle[i][0]])) {
                prvedg = pred[cycle[i][0]];
                i++;
                cycle[i] = prvedg;
            }

            //Removing extra edges from cycle
            cycle = getEdges(cycle);

            //Creating new current flow
            for (int[] en : cycle) {

                if ((en[0] == 0 && en[1] == 0)) {
                    continue;
                }

                if (currentFlow[en[1]][en[0]] == 1) {
                    currentFlow[en[1]][en[0]] = 0;

                } else {

                    currentFlow[en[0]][en[1]] = 1;
                }
            }

            residual = modifyResidual(original, currentFlow);
            res_cost = getCostMatrix(edge_cost_map, residual);
        }
```

S pomočjo prve `while` zanke izsledimo celoten negativni cikel. Deluje po metodi iskanja predhodnih povezav, s pomočjo `pred` tabele, dokler se ponovno ne najde začetna povezava. Z metodo `contains` preverjamo, ali je v naši tabeli povezav `cycle` začetna povezava. Metoda `getEdges` nam nato odstrani robne povezave, ki same niso del cikla, iz tabele `cycle`. Nadalje sledi postopek spreminjanja vrednosti povezav iz 0 v 1, če so te 0, ali iz 1 v 0, če so te 1. v matriki `currentFlow`. To je ekvivalentno potiskanju maksimalnega toka skozi cikel. Končno na osnovi spremenjene matrike `currentFlow` generiramo novi matriki `residual` in `res_cost` s pomočjo pomožnih metod. Celoten postopek se nato ponovi, dokler v grafu ni več prisoten negativni cikel.

```
String seqs = "";
for(int k = 0;k<currentFlow[0].length;k++){
    if(currentFlow[0][k]==1){
        String seq = traversePath(k,currentFlow);
        if(seqs.equals("")){
            seqs+=seq;
        }else{
            seqs+="a"+seq;
        }
    }
}

    long estimatedTime = System.nanoTime() - startTime;
    //Convert them to known solution
    e = ks.length + 1;
    int ri=-1;
    String[] seqlist = {""};

    for(int k = 0;k<req.length;k++){

        ri=k+e;
        seqlist=seqs.split("a");
        String cchar = "";

        for(int j = 0;j<seqlist.length;j++){

            if(Integer.parseInt(seqlist[j])<e){
                cchar=seqlist[j];
            }else{
                if(Integer.parseInt(seqlist[j])==ri){

                    solution=solution+(char)ks[Integer.parseInt(cchar)-1].name;
                    break;

                }
            }
        }

    }

    for(int k = 0;k<seqlist.length-1;k++){
        int x = Integer.parseInt(seqlist[k]);
        int y = Integer.parseInt(seqlist[k+1]);
        if(edge_cost_map[x][y]!=-1000){
            globaldistC+=edge_cost_map[x][y];
        }
    }
```

Sledi še pretvorba `currentFlow` matrike v nam berljivo obliko končne rešitve. V prvi zanki prepotujemo, s pomočjo pomožne metode `traversePath`, ki vrača niz imen vozlišč, po katerih poteka enotski tok, iz izvornega vozlišča v ponorno po vseh enotskih tokovih. Niz `seqs` je na koncu sestavljen iz imen vozlišč ločenih s črko "a".
V naslednji funkciji je spremenljivka `ri` enaka imenu vozlišča posameznega strežnika (torej prvega nivoja vozlišč). Sprehodimo se po vsakem nizu zaporedij imen vozlišč in ob pogoju, da je ime enako indeksu `ri`, ga dodamo k naši končni rešitvi.

Poglavje 4

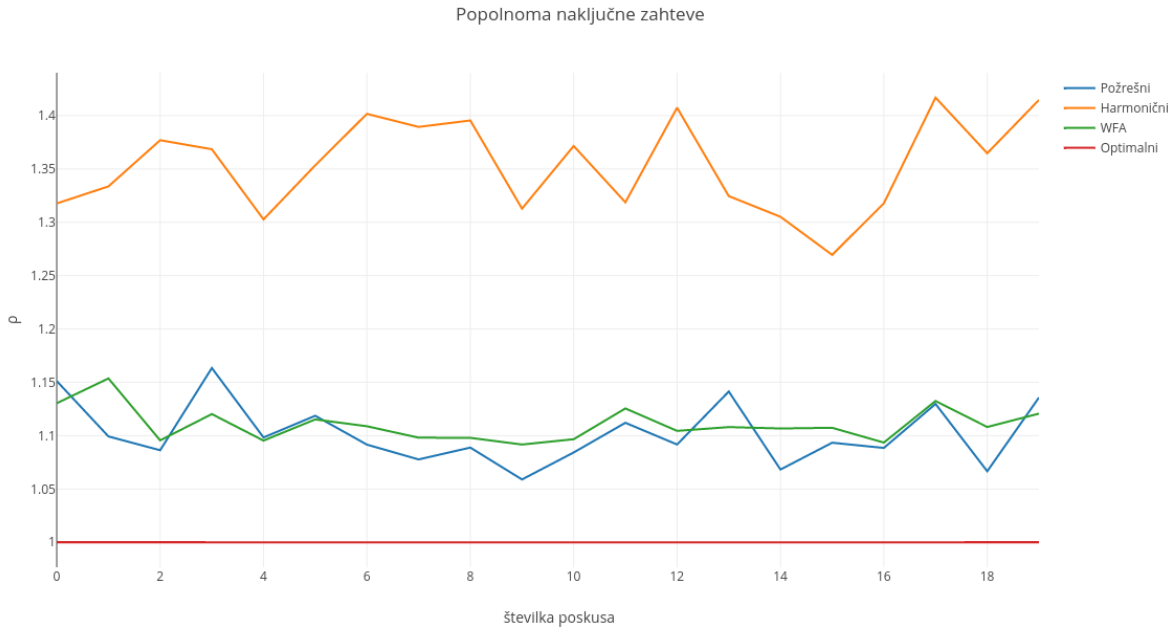
Rezultati testiranja

Algoritme, omenjene v prejšnjem poglavju smo testirali na računalniku Acer Aspire V3 771 (CPU: Intel Core i3 (2nd Gen) 2348M / 2.5 GHz, Dual core, 4GB DDR3 Memory), in sicer na operacijskem sistemu Linux Mint 17.2 "Rafaela". Javo smo poganjali za vsak način delovanja (glej Poglavje 3) posebej, in sicer v razvojnem okolju NetBeans IDE 8.2. Prepotovane razdalje algoritmov ne primerjamo neposredno, saj bi bilo to precej ne jasno in v veliki meri vezano na način delovanja, ampak jih pretvorimo v faktor ρ glede na rezultate optimalnega algoritma za posamezen način delovanja. Faktor ρ smo računali s pomočjo prilagojene formule iz Poglavja 2:

$$\rho = \frac{cost_A(C_0, r)}{opt(C_0, r)}$$

Za vsak način delovanja smo naredili 20 poskusov in njihove ρ vrednosti povprečili. Vrednosti posameznih poskusov smo opisali z grafom spreminjanja ρ vrednosti. Nato smo za čas izvajanja povprečili rezultate poskusov za vsak način delovanja. Lahko rečemo, da so nas ponekod rezultati celo presenetili, predvsem kar se tiče manj konkurenčnih algoritmov.

4.1 Popolnoma naključne zahteve



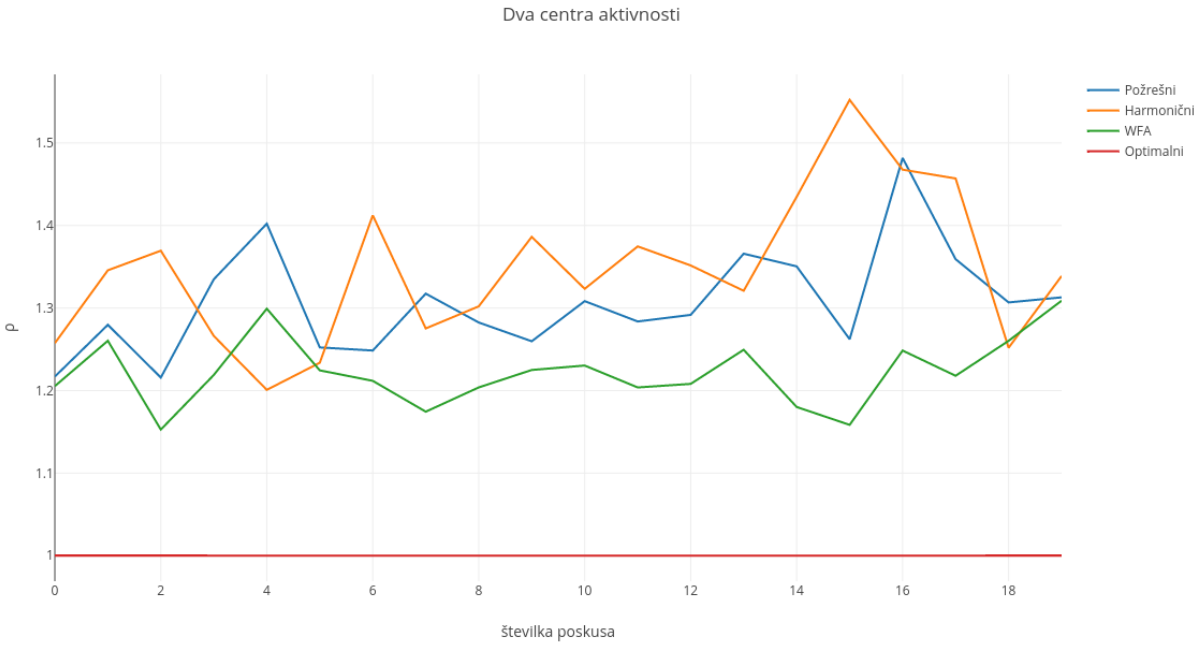
Slika 4.1: ρ vrednosti posameznih poskusov za vsak algoritem [4]

Požrešni	Harmonični	WFA
1.102	1.353	1.111

Slika 4.2: Povprečne ρ vrednosti 20 poskusov, k=2, r=185, popolnoma naključne zahteve, zaokroženo na 3 decimalna mesta

Rezultati kažejo, da se je v povprečju najbolje odrezal požrešni algoritem. To ni presenetljivo, saj nimajo naključno prihajajoče zahteve nikakršnega vzorca, zato se ob njegovem iskanju le zavajamo. Tukaj se pokaže šibkost Harmoničnega algoritma, saj že sam temelji na delnem ugibanju. Z dodatkom naključnih zahtev je možnost, da bi bil harmonični algoritem boljši, pogojena delno naključni izbiri strežnika in še naključnem nahajališču naslednje zahteve.

4.2 Dva centra aktivnosti



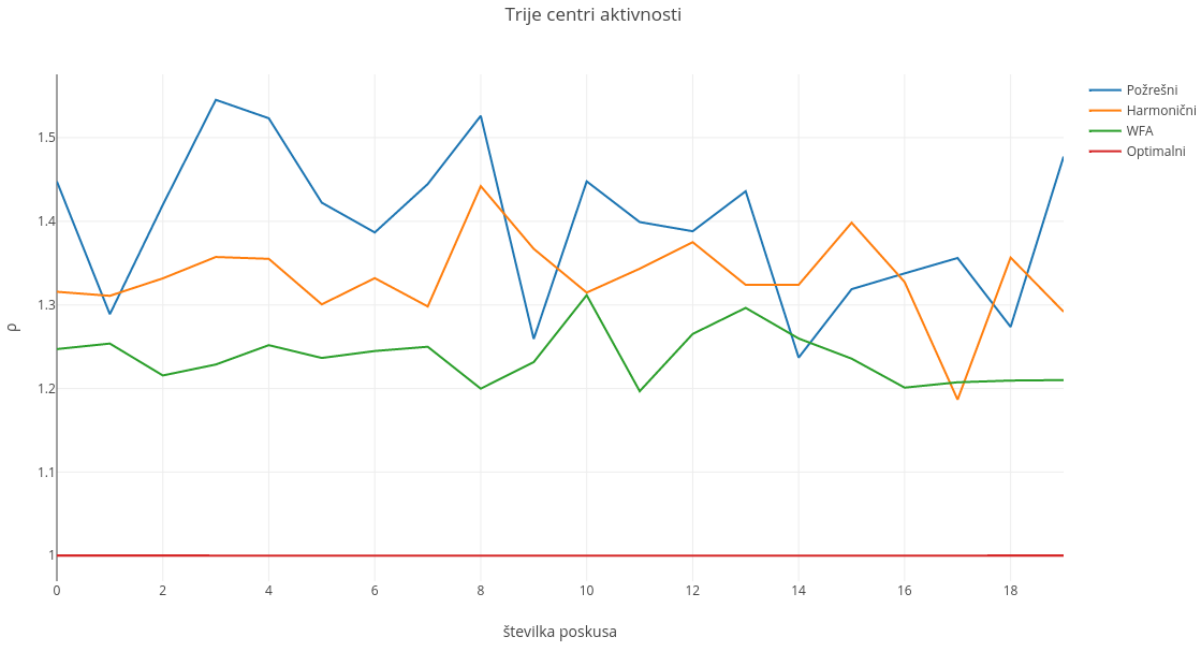
Slika 4.3: ρ vrednosti posameznih poskusov za vsak algoritem [4]

Požrešni	Harmonični	WFA
1.307	1.346	1.222

Slika 4.4: Povprečne ρ vrednosti 20 poskusov, k=2, r=185, dva centra aktivnosti, zaokroženo na 3 decimalna mesta

Tokrat se je najbolje odrezal WFA, saj je vsakič namreč premagal požrešni algoritem. To dokazuje, da je vendarle boljši, ko zahteve prihajajo v bolj predvidljivem vzorcu. Zanimivo je, da je bil dvakrat najboljši harmonični algoritem, kar si lahko razlagamo kot čisto naključje ali pa, da zadostuje samo ena srečna izbira strežnika, da se rezultat bistveno izboljša.

4.3 Trije centri aktivnosti



Slika 4.5: ρ vrednosti posameznih poskusov za vsak algoritem [4]

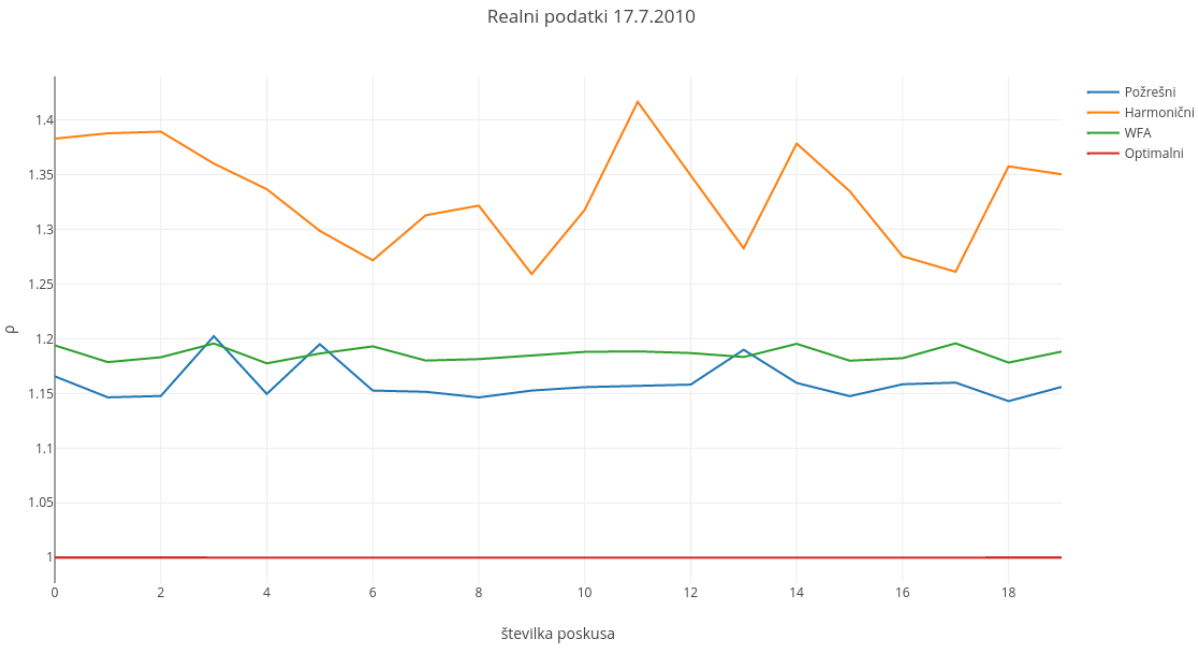
Požrešni	Harmonični	WFA
1.397	1.333	1.238

Slika 4.6: Povprečne ρ vrednosti 20 poskusov, k=2, r=185, trije centri aktivnosti, zaokroženo na 3 decimalna mesta

Rezultati so precej presenetljivi. Harmonični algoritem je v povprečju premagal požrešnega. Najboljše rezultate je tudi tokrat dosegel WFA. Možno je, da sta harmonični in WFA učinkovitejša na manjših prostorih. V našem primeru je to prvo območje pojavljanja zahtev, ki je velikosti 2×2, vendar bi to pomenilo, da sta boljša tudi v situacijah, ko so zahteve popolnoma naključne, kar smo s prvim poskusom zavrnili. Druga domneva je, da je imel

požrešni algoritem slabše rezultate preprosto, zato ker je samo en strežnik premaknil v polje aktivnosti. Boljše bi bilo, če bi tam imel dva hkrati, tako bi se drastično znižala prepotovana pot. WFA temelji na zaznavanju vzorcev, zato je ta hitro premaknil strežnika na področje aktivnosti. Harmonični je preprosto naključno premaknil en strežnik na območje aktivnosti, kar se mu je izplačalo v manjši prepotovani razdalji.

4.4 Realni primer



Slika 4.7: ρ vrednosti posameznih poskusov za vsak algoritem [4]

Izkazalo se je, da so rezultati pri teh realnih podatkih, precej podobni tistim pri popolnoma naključnih zahtevah. Najboljše se je odrezal požrešni algoritem, vendar mu WFA ni bil daleč za petami. Lahko domnevamo, da se WFA ni boljše odrezal, ker so ti vzorci ponavljanja bolj zapleteni. Možno

Datum	Požrešni	Harmonični	WFA
24.1.2016	1.138	1.312	1.143
17.7.2010	1.160	1.332	1.186
1.8.2010	1.117	1.338	1.151

Slika 4.8: Povprečne ρ vrednosti 20 poskusov, k=2, r=185, vsi datumi, zaokroženo na 3 decimalna mesta

je tudi, da ga je upoštevanje celotnega zaporedja preteklih zahtev pri izbiranju preveč zavedlo in bi bilo bolje uporabljati samo zadnjih w zahtev pri računanju delovne funkcije. Harmoničnemu ni uspelo najti srečnih potez, s pomočjo katerih bi postal bolj konkurenčen.

4.5 Čas izvajanja

Odločili smo se testirati čas za vsak algoritem posebej, in sicer za vse načine delovanja. Vsak algoritem smo izvedli dvajsetkrat in količino časa, porabljenega za vsako izvedbo algoritma prišteli globalni spremenljivki, ki smo jo na koncu delili s številom poskusov, da smo dobili končni rezultat. Merili smo v nanosekundah, ker naj bi bili tako rezultati bolj točni. Zaradi relativno majhne velikosti našega polja smo se merjenju časa, porabljenega za obdelavo posameznih zahtev odpovedali, saj bi bile le-te meritve preveč nenatančne.

Način delovanja	Požrešni	Harmonični	WFA	Optimalni
Naključne zahteve	0,000210042	0,000467997	2,449051918	13,432063301
Dva centra	0,000182938	0,000451194	2,551526506	13,181960546
Trije centri	0,000176828	0,000470663	2,408367960	12,830081770
Realni primer	0,000164878	0,000489030	2,327354943	12,638543302

Slika 4.9: Povprečni časi izvajanja 20 poskusov vsakega načina delovanja za $r = 2$,$k = 185$ v sekundah. Rdeče celice predstavljajo najslabši rezultat in zelene najboljšega.

Iz tabele hitro opazimo povezavo med zapletenostjo posameznega algoritma in časom izvajanja. Ti dve vrednosti sta premo sorazmerni. Kar se tiče načinov delovanja se da opaziti marsikaj. Pri realnem primeru imajo skoraj vsi algoritmi najnižji čas. Razlog za to je najverjetneje visok nivo predvidljivosti pri realnem problemu za razliko od velike količine naključnih zahtev pri ostalih načinih delovanja. Edini algoritem, ki pri realnem primeru ni imel najnižjega časa je harmonični, ki že sam pri svojem delovanju uporablja naključnost. Trditev o naključnosti prav tako podpirajo najslabši rezultati. Upamo si trditi, da je testiranje časa pri teh algoritmih nekoliko nesmiselno, saj nam pridobljeni surovi podatki ne povedo veliko in so v bistvu še najbolj zanimivi, ko jih primerjamo med seboj. Takšne primerjave se lahko zlahka lotimo iz samega vidika primerjanja formalne teoretične zahtevnosti algorit-

mov. Pravzaprav je edina stvar, ki je ne moremo preučiti v čisti teoriji, vpliv naključnosti na rezultate.

Poglavje 5

Zaključek

Videli smo, kako se lahko na videz preprost problem precej zaplete, kar se tiče raznovrstnosti metod reševanja. Pri delu smo se sicer samo dotaknili površja zanimivega in tudi še vedno zelo aktualnega problema k-strežnikov. Izkaže se, da so včasih najpreprostejše rešitve najboljše. Požrešni algoritem se je izkazal kot zelo praktičen in preprost za uporabo ter razumevanje. Ker je realni svet sestavljen iz mnogih, včasih na videz skorajda naključnih vzorcev in zaporedij, smo prisiljeni v stiski izbrati najpreprostejši algoritem. Nadalje se, v svojih spremenjenih oblikah, izkaže za precej bolj uporabnega pri obdelavi bolj predvidljivih in omejenih zaporedij. Harmonična implementacija, ki doda faktor naključnosti požrešnemu algoritmu, nam s pomočjo ugibanja včasih omogoča, da naredimo srečno potezo, ki nas privede do celotnega boljšega zaporedja obdelave. Najnovejša rešitev problema je algoritem delovne funkcije, ki pri svojem odločanju upošteva tako pretekle zahteve kot trenutno. V situacijah, ko so vse zahteve popolnoma naključne, se izkaže slabši oziroma skoraj enak preprosti požrešni rešitvi. Zagotovo je najboljši pri preprostih in predvidljivih vzorcih, takrat bistveno preseže rezultate drugih algoritmov. Za vse njegove rezultate je potrebno plačati višjo ceno prostorske zahtevnosti. V delu smo obravnavali le najpreprostejše implementacije navedenih algoritmov, obstaja namreč vrsto spremenjenih verzij, ki ponujajo različne izboljšave. Dve takšni sta w-wfa, ki je algoritem delovne funk-

cije, ki pri izračunu upošteva samo zadnjih w zahtev in spremenjena verzija statičnega algoritma, ki ima vrsto izboljšav pri računanju toka. Vredno se je tudi vprašati, ali bi se problem dalo še boljše obravnavati skozi oko umetne inteligence.

Literatura

[1] Yair Bartal and Eddie Grove. The harmonic k-server algorithm is competitive. *Journal of the ACM (JACM)*, 47(1):1–15, 2000.

[2] Seattle Police Department City of Seattle, Department of Information Technology. Seattle police department 911 incident response. `https://data.seattle.gov/Public-Safety/Seattle-Police-Department-911-Incident-Response/3k2p-39jp`, 2010.

[3] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[4] Plotly Inc. Plotly. `https://plot.ly/`, 2012.

[5] Elias Koutsoupias. The k-server problem. *Computer Science Review*, 3(2):105–118, 2009.

[6] Graph Online. Graph online. `http://graphonline.ru/en/create_graph_by_matrix`, 2012.

[7] Tomislav Rudec, Alfonzo Baumgartner, and Robert Manger. A fast implementation of the optimal off-line algorithm for solving the k-server problem. *Mathematical communications*, 14(1):119–134, 2009.

[8] Tomislav Rudec, Alfonzo Baumgartner, and Robert Manger. Measuring true performance of the work function algorithm for solving the online k-server problem. *Journal of computing and information technology*, 18(4):361–367, 2010.

[9] Princeton University. Standard draw. https://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html, 2008.