Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Rok Sokol

Vizualizacija iskanja z algoritmom LRTA*

diplomsko delo

univerzitetni interdisciplinarni študijski program prve stopnje računalništvo in matematika

Mentor: izr. prof. dr. Marko Robnik Šikonja

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00029/2013
Datum: 12.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **ROK SOKOL**

Naslov: **VIZUALIZACIJA ISKANJA Z ALGORITMOM LRTA***
VISUALIZATION OF LRTA* SEARCH ALGORITHM

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Pri poučevanju preiskovalnih algoritmov nam njihovo razumevanje bistveno olajšajo orodja za vizualizacijo njihovega delovanja. To še posebej velja za algoritme, ki lahko delujejo vzporedno in asinhrono. Algoritem iskanja najkrajših poti LRTA* je primer algoritma, ki pohitritev doseže z vzporednim delovanjem več agentov, in služi kot primer usklajenega delovanja in posredovanja informacij med računskimi enotami.
Proučite delovanje algoritma LRTA*(n) za iskanje najkrajših poti v usmerjenem grafu in njegovo delovanje ilustrirajte na nekaj primerih grafov. Izdelajte spletno animirano interaktivno aplikacijo, ki bo uporabnika poučila o delovanju algoritma in mu omogočila izdelavo lastnih konfiguracij.

Mentor:
izr. prof. dr.Marko Robnik Šikonja

Dekan Fakultete za računalništvo in informatiko:
prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:
akad. prof. dr. Franc Forstnerič

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Rok Sokol, z vpisno številko **63060287**, sem avtor diplomskega dela z naslovom:

*Vizualizacija iskanja z algoritmom LRTA**

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Robnika Šikonje,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 18. september 2013 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Marku Robniku Šikonji za pomoč in usmerjanje pri izdelavi diplomskega dela. Za podporo in razumevanje ob pomanjkanju časa se zahvaljujem tudi punci Luciji Eržen, poleg nje pa tudi mami in očetu, ki sta bila tekom mojega študija potrpežljiva in me vzpodbujala.

Samemu dragemu sebi.

Kazalo

Povzetek

V okviru diplomskega dela smo izdelali spletno aplikacijo, ki interaktivno, korak za korakom, prikazuje delovanje algoritma LRTA* in poleg vnaprej sestavljenih primerov omogoča tudi izdelavo novih problemov. V delu najprej predstavimo področje preiskovalnih algoritmov. Poleg preiskovalnega algoritma LRTA* si ogledamo tudi nekaj povezanih algoritmov. Predstavimo orodja in tehnologije, ki so bile uporabljene za izdelavo spletne aplikacije, predstavimo uporabo in delovanje spletne aplikacije in vizualiziramo nekaj primerov, kjer bi lahko uporabljali algoritem LRTA*. V zaključku povzamemo delo in navedemo ideje za izboljšave spletne aplikacije.

Ključne besede

preiskovalni algoritmi, asinhrono preiskovanje, algoritem LRTA*(n), spletna aplikacija, vizualizacija

Abstract

We developed a web application interactively demonstrating LRTA* algorithm by solving pre-assembled problems and allowing creation of new tasks. We first present search algorithms related to LRTA* algorithm and explore LRTA* by defining intelligent agents, which are used by it. Step by step presentation of algorithm is shown. We present tools and technologies that were used to create the web application, present how the web application works, and visualize a few cases where LRTA* algorithm can be used. We summarize the work and present ideas for improvements of the web application.

Keywords

search algorithms, asynchronous search, LRTA*(n) algorithm, web application, visualization

Poglavje 1

Uvod

Vsakodnevno se srečujemo z uporabo transportnih sredstev. Pri tem velikokrat uporabljamo navigacijske naprave, mobilne telefone ter spletne aplikacije, na primer Google Maps. Nemalokrat iščemo najkrajšo razdaljo med dvema krajema, časovno najhitrejšo pot med dvema krajema itn.

Pri iskanju poti s temi kriteriji naprave uporabljajo algoritme za iskanje najkrajših poti. Teh algoritmov je precej, zato si jih bomo v nadaljevanju nekaj pogledali, osredotočili pa se bomo predvsem na algoritem LRTA* (angl. *learning real-time A star algorithm*).

Za razumevanje iskalnih algoritmov je pomembna njihova vizualizacija – namesto, da prostor stanj opisujemo z besedami, ga raje predstavimo v obliki grafa. Koristna je slikovna predstava spreminjanja stanj tekom delovanja algoritma. Za hitro razumevanje delovanja algoritmov je po navadi bolje, da imamo namesto učbenika, v katerem je naveden problem in njegova rešitev, spletno aplikacijo, ki nam omogoča interaktivnost, nam razlaga postopek delovanja korak za korakom ter nas pouči o razlogih za določene izbire. Navedeno je motivacija za to diplomsko delo – izdelava spletne aplikacije za pomoč izobraževanju, ki zagotovi hitro razumevanje in nazornejše delovanje algoritma LRTA*.

V drugem poglavju se seznanimo z neinformiranimi in informiranimi preiskovalnimi algoritmi. Naštejemo nekaj pomembnejših algoritmov, si v gro-

bem pogledamo njihovo delovanje ter naštejemo njihove dobre in slabe lastnosti. Za vsak preiskovalni algoritem navedemo časovno in prostorsko zahtevnost. Definiramo prostor stanj in hevristične funkcije, ki jih uporabljajo informirani preiskovalni algoritmi. Seznanimo se z osnovnimi pojmi kot so dopustnost, optimalnost in popolnost algoritmov. Nekaj besed namenimo optimalnim rešitvenim potem in cenam povezav. S tem si pripravimo podlago za lažje in hitrejše razumevanje tretjega poglavja, kjer govorimo o jedru diplomskega dela – algoritmu LRTA*.

To je popoln algoritem, ki mu pri iskanju najkrajše poti do cilja nikoli ne spodleti. Ker pri preiskovanju uporablja hevristično funckijo, se uvršča med informirane preiskovalne algoritme. Gre za algoritem, ki deluje asinhrono in doseže pohitritve pri iskanju rešitev z uporabo delovanja več vzporednih agentov. Predstavimo algoritem LRTA*, njegove lastnosti, prednosti, slabosti, ter ga primerjamo z izbranimi algoritmi iz drugega poglavja. Delovanje algoritma LRTA* ilustriramo na dveh primerih, kjer skozi korake spoznavamo njegovo delovanje.

Četrto poglavje je namenjeno predstavitvi strežnikov, uporabljenih orodij ter tehnologij, ki so omogočila nastanek spletne aplikacije, ki rešuje probleme s pomočjo algoritma LRTA*.

V petem poglavju predstavimo strukturo spletne aplikacije, si ogledamo možnosti, ki jih aplikacija ponuja, ter se podučimo o njeni uporabi. Predstavimo zgradbo celotne spletne strani in jo opišemo s pomočjo zaslonskih slik spletne aplikacije.

V zaključnem poglavju povzamemo delo, podamo nekaj idej za nadaljnje delo in izboljšave v obliki dodatkov k spletni aplikaciji.

Poglavje 2

Preiskovalni algoritmi

S preiskovanjem se ukvarja umetna inteligenca, saj je to ena izmed osnovnih tehnik za reševanje problemov. Preiskovalne algoritme razdelimo na dve vrsti in sicer na algoritme, ki preiskujejo izčrpno (neinformirano, angl. *brute-force search*) in na algoritme, ki preiskujejo hevristično (informirano).

Obema skupinama preiskovalnih algoritmov je skupno, da je problem predstavljen s prostorom stanj, s katerim pretvorimo problem v usmerjeni graf, pogosto v drevo (nemalokrat imamo sicer tudi probleme, kjer grafi oz. drevesa niso usmerjena). Vozlišča grafa predstavljajo stanje problema, usmerjene povezave pa definirajo prehode med stanji. Za delovanje preiskovalnih algoritmov definiramo eno začetno stanje in eno (ali več) končnih stanj. Potrebna je tudi funkcija za pridobivanje naslednikov, ki za dano vozlišče vrne množico z njim povezanih stanj [3]. Povezavam priredimo vrednosti, ki jih imenujemo cene. Ceni vozlišča n priredimo oznako g(n) in jo definiramo kot vsoto cen povezav, ki ležijo na poti od začetnega stanja do vozlišča n. Podroben opis prostora stanj in hevrističnih fukncij definiramo v poglavju 2.1.

Pred pregledom prostora stanj in preiskovalnih algoritmov definirajmo štiri kriterije, ki se uporabljajo za merjenje učinkovitosti preiskovalnih algoritmov:

- popolnost – ali algoritem vedno najde rešitev, če le ta obstaja,

- optimalnost – ali algoritem najde najboljšo rešitev,
- časovna zahtevnost – čas, ki ga porabi algoritem za najdeno rešitev,
- prostorska zahtevnost – pomnilniški prostor, ki je potreben algoritmu za preiskovanje.

2.1 Prostor stanj

V nadaljevanju govorimo o preiskovanju prostora, zato ga definirajmo. Prostor stanj je definiran kot peterček (N, A, k, s, G), ki predstavlja preprost graf (N, A), kjer je N neprazna končna množica stanj (vozlišč) in A množica povezav, ki ustreza pogoju A $\subset$ N$\times$N – $\{$(n, n) | n $\in$ N$\}$. Povezave so označene s števili, ki predstavljajo ceno poti med danima vozliščema, ki jo predpisuje cenovna funkcija (angl. *cost function*) k : A $\mapsto$ [0, ∞). Ostala dva elementa peterice sta s, ki predstavlja začetno stanje, za katerega velja s $\in$ N in G $\subseteq$ N, ki definira množico ciljnih vozlišč. Prostor stanj je neusmerjen, če v A velja simetrična relacija, kjer je k(n_i, n_j) = k(n_j, n_i) za vsak par (n_i, n_j) $\in$ A, sicer pa je prostor stanj usmerjen.

Označimo množico neposrednih naslednikov (angl. successors) vozlišča n z oznako Succ(n) ali Succ(n) = $\{$ n_j | (n_i, n_j) $\in$ A $\}$. Pot v grafu je neprazno zaporedje (n_0, n_1, ...) stanj, kjer vsak par (n_i, n_{i+1}) v zaporedju pripada množici A.

Predpostavljamo, da za vsako ne-ciljno vozlišče n $\in$ N – G v prostoru stanj obstaja vsaj ena pot od vozlišča n do ciljnega vozlišča. Dolžina rešitvene poti je vsota števila povezav, ki so vključena v pot od začetnega vozlišča do končnega vozlišča. Cena rešitvene poti pa je vsota povezav med pari vozlišč, ki ležijo na rešitveni poti.

Povzemimo lastnosti prostora stanj v petih točkah:

- moči množic vozlišč |N| in povezav |A| sta končni,
- vsaka povezava med dvema vozliščema ima pozitivno vrednost,

- za vsako stanje (vozlišče) grafa obstaja vsaj ena pot do ciljnega vozlišča,
- graf (N, A) je enostaven, kar pomeni, da v grafu ni vzporednih (večkratnih) povezav med dvema vozliščema,
- prostor stanj ne vsebuje zank (cikel dolžine 1) tj. $(n, n) \notin A$ za katerikoli vozlišče $n \in N$.

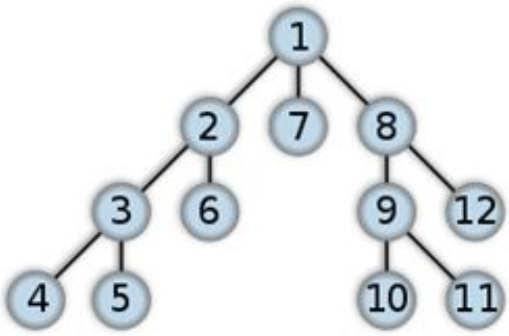
2.2 Neinformirano preiskovanje

Preiskovalni algoritmi, ki za preiskovanje ne uporabljajo dodatnih informacij o problemskem prostoru, so neinformirani. Tako preiskovanje sistematično in neusmerjeno pregleduje celoten graf. Pogledali si bomo tri neinformirane preiskovalne algoritme in sicer iskanje v globino, iskanje v širino in iterativno poglabljanje.

Za potrebe izračuna časovne in prostorske zahtevnosti definirajmo parametre b, d in m. Parameter b naj bo faktor vejitve oz. največje število možnih naslednikov kateregakoli vozlišča, parameter d predstavlja globino ciljnega vozlišča, ki je glede na nivoje v drevesu najbližje začetnemu vozlišču, in parameter m naj bo najdaljša pot v prostoru stanj. Število generiranih stanj izraža časovno zahtevnost algoritma, prostorska zahtevnost pa je izražena kot število stanj, ki jih naenkrat hranimo v pomnilniku.

2.2.1 Iskanje v globino

Algoritem iskanja v globino oz. DFS (angl. *Depth-first search*) izmed množice naslednikov vozlišča n izbere prvega in se vrača, ko naslednikov zmanjka, razen v primeru, če najde pot do ciljnega vozlišča (cilja) in se potemtakem ni potrebno vračati do vozlišča n. Ker algoritem iz množice naslednikov vedno izbere prvega, si lahko delovanje algoritma predstavljamo kot sprehod skozi drevo od leve proti desni. Najprej torej preišče celotno vejo na levi strani in, če ne najde rešitve, pot postopoma nadaljuje po desni veji. Delovanje algoritma DFS je prikazano na sliki 2.1 [14].



Slika 2.1: Zaporedje, po katerem algoritemm DFS preiskuje drevo.

Časovna zahtevnost algoritma DFS je O(b^m) (to število je lahko neskončno, če globina prostora stanj ni omejena).

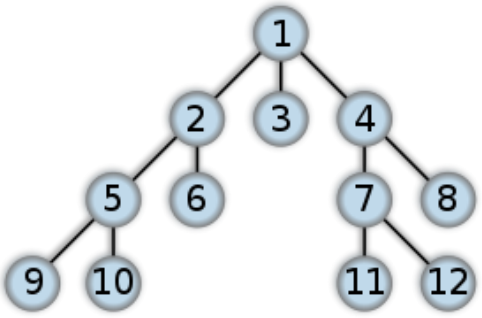
Prostorska zahtevnost algoritma je reda O(bm), saj se v danem trenutku na skladu hranijo le tista vozlišča, preko katerih algoritem prispe v dano vozlišče. Posledica tega je, da algoritem porabi zelo malo pomnilnika, kar je pozitivna lastnost iskanja v globino.

Iskanje v globino ima tudi slabe lastnosti. Kot prvo, algoritem ne zagotavlja optimalne rešitve, torej najkrajše poti od začetnega do končnega vozlišča. Slabost iskanja v globino je tudi, da lahko algoritem med izvajanjem zaide v neskončno zanko (t.j. zankanje), če niso implementirani mehanizmi za prepoznavanje zank [1].

2.2.2 Iskanje v širino

Algoritem iskanja v širino oz. BFS (angl. *Breadth-first search*) deluje tako, da začne svojo pot pri korenskem vozlišču (začetno vozlišče) in generira vse direktne naslednike vozlišča. Nato obišče vsako vozlišče v množici naslednikov in preveri, ali je katero izmed njih ciljno vozlišče. V primeru, da je obiskano vozlišče ciljno, kot rešitev vrne pot od začetnega do končnega vozlišča, sicer pa se iskanje poglobi za en nivo, torej se generira množica naslednikov naslednikov korenskega vozlišča. Postopek se ponavlja, dokler algoritem BFS ne prispe v ciljno stanje [12]. Delovanje algoritma BFS je prikazano na sliki

2.2.



Slika 2.2: Zaporedje, po katerem algoritemm BFS preiskuje drevo.

Časovna zahtevnost algoritma BFS je O(b^d), kar sledi iz ugotovitve, da imamo na prvem nivoju eno vozlišče, na drugem nivoju b vozlišč, na tretjem nivoju b^2 vozlišč itn. Red časovne zahtevnosti dobimo po razmisleku, da je

$$1 + b + b^2 + \ldots + b^{(d-1)} = O(b^d) \tag{2.1}$$

Prostorska zahtevnost algoritma BFS je reda O(b^d), kar sledi iz razmisleka, da tudi, če se ciljno vozlišče nahaja na nivoju d-1, je algoritem vnaprej generiral vozlišča nivoja d.

Dobra lastnost algoritma iskanja v širino je, da zagotavlja najkrajšo rešitev, torej rešitev, ki je glede nivojske oddaljenosti najbližja korenskemu vozlišču.

2.2.3 Iterativno poglabljanje

Preiskovalni algoritem iterativnega poglabljanja oz. IDDFS (angl. *Iterative deepening depth-first search*) deluje kot kombinacija algoritmov iskanja v širino in iskanja v globino. Uvedemo ga z namenom, da se izognemo prekomerni prostorski porabi algoritma iskanja v širino. Pri algoritmu IDDFS gre za iskanje v globino, kjer globino omejimo na določen nivo. V primeru, da

ne dosežemo ciljnega vozlišča, se nivo globinske omejitve poveča. Izvajanje se ustavi, ko na d-ti globini dosežemo cilj oz. iskano končno vozlišče.

Slabost algoritma je, da se vozlišča do mejne globine preiščejo vsakič znova. Posledica tega je, da manjši preiskovalni prostor zamenjamo z večjim preiskovalnim časom.

Izračunajmo časovno zahtevnost algoritma IDDFS: vozlišča na končni (največji) globini d, kjer se nahaja ciljno vozlišče, razvijemo enkrat. Vozlišča na zadnjem koraku izvajanja algoritma, kjer cilj ni bil najden (torej globina d-1), se razvijejo dvakrat. Tako pridemo do korenskega vozlišča, ki je bilo razvito d+1 krat. Te ugotovitve lahko strnemo v formulo in dobimo slednje [25]:

$$(d+1) + db + (d-1)b^2 + \ldots. + 3b^{(d-2)} + 2b^{(d-1)} + b^d \tag{2.2}$$

$$\sum_{i=0}^{d} (d+1-i) * b^i \tag{2.3}$$

Iz formul (2.2) in (2.3) sledi, da je časovna zahtevnost algoritma IDDFS reda O(b^d).

Prostorska zahtevnost algoritma IDDFS je O(bd).

2.2.4 Lastnosti neinformiranih preiskovalnh algoritmov

Za večjo preglednost strnimo ugotovitve v tabelo 2.1.

	Časovna zahtevnost	Postorska zahtevnost	Najdba najkrajše rešitvene poti
Iskanje v globino (DFS)	O(b^m)	O(bm)	ne
Iskanje v širino (BFS)	O(b^d)	O(b^d)	da
Iterativno poglabljanje (IDDFS)	O(b^d)	O(bd)	da

Tabela 2.1: Primerjava osnovnih neinformiranih algoritmov.

2.3 Informirano preiskovanje

Informirano preiskovanje imenujemo tudi hevristično preiskovanje, saj se tekom reševanja v problemskem prostoru uporablja hevristična ocena, ki usmerja preiskovanje v predvidoma bolj obetavno smer.

Informirano preiskovanje se uporablja, kjer so klasične preiskovalne metode prepočasne. Uporablja se tudi za pridobivanje približnih rešitev, kadar klasične metode ne najdejo eksaktne rešitve. S takim preiskovanjem dosežemo, da optimalnost, popolnost, točnost ali natančnost zamenjamo za hitro reševanje danega problema [20].

Kot preprost primer hevurističnega preiskovanja si, povzeto po [3], oglejmo igro križcev in krožcev. Pravila igre so preprosta: v polje dimenzije 3x3 dva igralca izmenično na poljubno nezasedeno mesto označita svojo potezo, eden igralec s križcem, drugi s krožcem. Zmaga igralec, ki prvi vodoravno, navpično ali diagonalno postavi svoje znake. Recimo, da sta igralca na začetku igre, kjer so vsa polja še prazna. Na sliki 2.3 je v posameznih celicah zapisano število možnih zmag, če igralec postavi prvi znak v dano celico.

3	2	3
2	**4**	2
3	2	3

Slika 2.3: Število možnih zmag, odvisno od začetne postavitve znaka.

Iz slike 2.3 je razvidno, da je največja možnost zmage v primeru, ko igralec postavi potezo v sredinsko polje. Ta začetna ocena pripomore k tem, da bi informirano (hevristično) preiskovanje izbralo sredinsko pozicijo kot prvo potezo.

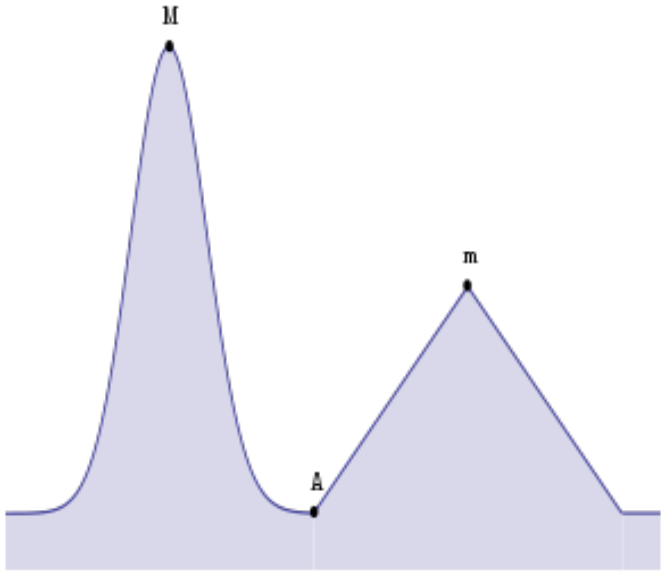
2.3.1 Požrešno preiskovanje

Osnovni način izkoriščanja informiranosti je požrešno preiskovanje. Najpreprostejša primera požrešnega preiskovanja sta algoritma požrešno iskanje in Hill-climbing. Ogledali si bomo algoritem požrešnega iskanja.

2.3.1.1 Algoritem požrešnega iskanja

Algoritem požrešnega iskanja (*angl. Greedy search*) rešuje probleme na način, da na vsakem koraku sledi lokalno najvišji hevristični oceni [19], ter upa, da najde globalni optimum. V mnogih primerih požrešno preiskovanje ne prinese optimalne rešitve, vendar pa najde lokalno optimalno rešitev, ki se včasih približa globalni optimalni rešitvi.

Na sliki 2.4. je predstavljen primer, na katerem požrešnemu algoritmu spodleti najti globalni maksimum. Nahajamo se v točki A in želimo doseči najvišjo točko krivulje. S črko M je označen globalni maksimum, s črko m pa lokalni maksimum. Desno od točke A se vrednost koordinate y poveča bolj kot levo, zato požrešni algoritem začne svojo pot v desno stran. Po končnem številu korakov pride v točko m, ki je od sosednjih točk najvišja glede na koordinato y. Algoritem zmotno sporoči drugačno rešitev, kot bi jo pričakovali.



Slika 2.4: Primer, na katerem požrešnemu algoritmu spodledi.

2.3.2 Najprej najboljši

Algoritmi po principu najprej najboljši (angl. *best first search*) bolje izkoriščajo hevristično oceno. Postopek delovanja je sledeč: vsa vozlišča, ki jih algoritem generira, so shranjena v prioritetni vrsti. Za izbiro naslednjega pregledanega vozlišča služi hevristična ocena. Torej, če iščemo najkrajšo pot v grafu, bo vedno najprej razvito vozlišče z najmanjšo hevristično oceno. Pomembnejši posebni primeri algoritmov najprej najboljši so A, A* in B*.

2.3.3 Algoritem A

Algoritem A je poseben primer iskanja najprej najboljšega. Denimo, da lahko informacijo o hevristični oceni vozlišča n razdelimo na vsoto dveh delov:

$$f(n) = g(n) + h(n) \tag{2.4}$$

V enačbi je pot do vozlišča n označena z g(n) in pomeni seštevek vseh povezav od začetnega vozlišča do vozlišča n. Hevristična funkcija h je preslikava h: N $\mapsto$ [0, ∞]. Hevristična funkcija vsakemu stanju dodeli oceno dolžine poti do cilja, zato vrednosti h(n) pravimo hevristična ocena stanja n. Algoritem A uporablja hevristiko iz enačbe (2.4).

2.3.4 Algoritem A*

Razlika med algoritmom A* in prejšnjim algoritmom A je le v uporabi dopustne hevristike. Algoritem je dopusten (angl. *admissible*), če zanj velja, da zagotovo najde najkrajšo pot od začetnega korenskega vozlišča do ciljnega vozlišča (če taka pot obstaja).

Za lažje razumevanje vpeljimo enačbo:

$$f^*(n) = g^*(n) + h^*(n) \tag{2.5}$$

Dolžina najkrajše poti od začetnega vozlišča do vozlišča n je označena z g*(n), dolžino najkrajše poti od vozlišča n do ciljnega vozlišča pa označujemo z h*(n). Točna ocena h*(n) vozlišča n je minimalna vsota cen povezav, ki se nahajajo na rešitveni poti od vozlišča n do ciljnega vozlišča. Za vsako ciljno vozlišče g $\in$ G velja h*(g) = 0, kar pomeni, da hevristika napoveduje, da je dolžina poti od (ciljnega) vozlišča do ciljnega vozlišča enaka 0. Algoritmi, ki uporabljajo hevristiko f*(n) so dopustni, ker zaradi uporabe prioritetne vrste najprej razvijejo vozlišča, ki so bližje cilju. Pot, ki je enaka hevristični oceni vozlišča n, se imenuje optimalna pot iz n. V primeru, da h*(n) podcenjuje dolžino poti do cilja, se razvije vozlišče iz prioritetne vrste, ki je obetavnejše in na ta način postopoma zanesljivo najdemo najkrajšo pot. Žal po navadi ne znamo določiti funkcije f*(n) vozlišča n, ki sestoji iz vsote členov g*(n) in h*(n).

g*(n) ocenjujemo s približkom g(n), kjer velja neenakost g*(n) $\leq$ g(n). V primeru, da smo do danega vozlišča n že odkrili najkrajšo pot, velja, da med g*(n) in g(n) nastopa enakost. V takšnem primeru bi bil algoritem dopusten tudi ob uporabi enačbe (2.6).

$$f(n) = g(n) + h^*(n) \tag{2.6}$$

Podobno velja za h*(n), ki za približek uporabi h(n) iz enačbe (2.4). Do tega zaključka pridemo z razmislekom da, če je h(n) dopustna (podcenjuje razdaljo do cilja tj. h*(n) $\geq$ h(n)), potem smemo uporabiti enačbo (2.4) in imamo še vedno dopustni algoritem, ki najde najkrajšo pot do cilja.

Časovna zahtevnost algoritma A* je odvisna od hevrističnih ocen. V najslabšem primeru (angl. *worst case scenario*) je rast eksponentna. Rast je polinomska, če problem ustreza naslednjim kriterijem:

- preiskovalni prostor je drevo,
- obstaja samo eno ciljno vozlišče,
- hevristična funkcija h ustreza pogojem enačbe (2.7), kjer je h* optimalna hevristika, torej točna vrednost povezav od vozlišča n do ciljnega vozlišča.

$$|h(n)–h^*(n)| = O(log\, h^*(n)) \tag{2.7}$$

Največja težava algoritma A* je velika poraba pomnilniškega prostora, saj v prioritetni vrsti hrani veliko število vozlišč, ki hitro polnijo pomnilnik [9]. Čeprav je algoritem A* eden najboljših splošno namenskih iskalnih algoritmov, obstajajo izboljšave A*, ki jih bomo omenili v poglavju 2.3.5.

2.3.5 Iterativno poglabljanje A*

Algoritem iterativnega poglabljanja na način A* oz. IDA* (angl. *iterative deepening A**) rešuje težave velike porabe prostora, ki se pojavijo pri A* [24]. Ideja je v združitvi algoritmov iterativnega poglabljanja in A*, s poglabljanjem vrednosti hevristične ocene f(n). Algoritem deluje tako, da začne izvajanje pri vrednosti F, ki je na začetku enaka f(s), kjer je s začetno vozlišče. Preiskujejo se vozlišča, ki imajo vrednosti F' (tj. f(n) za dano vozlišče

n) manjše ali enake F. Ko takih vrednosti zmanjka, se vrednost F poveča oz. poglobi za najmanjšo prekoračitev F'.

Algoritem IDA* ima majhno porabo pomnilniškega prostora, ker ne hrani vozlišč. Zapomni si le tista, ki ležijo na trenutni poti preiskovanja. Posledično je algoritem IDA* v primerjavi z algoritmom A* počasnejši, saj številna vozlišča preišče večkrat. Glede na prirastke F' algoritma IDA* ni smiselno uporabljati, kadar se F poglablja tako, da se v vsaki iteraciji razvije samo eno novo vozlišče, saj nova iteracija pomeni, da se poleg vozlišč, ki ustrezajo meji F, ponovno razvijejo vsa vozlišča prejšnjih iteracij. To povzroči neučinkovitost algoritma. V takem primeru bi se raje odločili za drug preiskovalni algoritem.

Poglavje 3

Preiskovalni algoritem LRTA*

V prejšnjem poglavju smo preiskovalne algoritme razdelili na neinformirane in informirane. Pri vsaki skupini smo obravnavali nekaj pomembnih pripadnikov skupine. Predstavili smo prostor stanj, kako hevristično ocenjevanje pripomore k boljšemu reševanju problemov, težave pri iskanju optimalnih rešitev in težave s prostorsko zahtevnostjo.

V tem poglavju bomo obravnavali hevristični algoritem LRTA* (angl. *learning real time A* algorithm*), ki je eden temeljnih in najpopularnejših algoritmov izmed vseh, ki so sposobni učenja [4].

V nadaljevanju naštejemo lastnosti algoritma LRTA*, zapišemo psevdokodo algoritma, pogledamo učni proces, definiramo inteligentne agente in si podrobno ogledamo delovanje algoritma na dveh primerih.

3.1 Algoritem LRTA* in psevdokoda algoritma

Pri iskanju poti se dogajanje začne v začetnemu stanju prostora stanj, nato se poišče optimalna pot, ki vodi do cilja, tj. pot, ki ima najmanjšo vrednost vsote povezav. Iskanje v realnem času zahteva, da se posamezne odločitve preiskovanja sprejemajo v realnem času, kar je mogoče, če preiskovanje temelji zgolj na informacijah, ki so na voljo v bližnji okolici trenutnega stanja. V diplomskem delu omejimo lokalno iskanje po načelu gledanja vnaprej (angl.

look-ahead) in sicer na globino 1, torej pregledovati smemo le vozlišča, ki so neposredno povezana z danim vozliščem, ki ga v danem trenutku obravnavamo.

Prikažimo psevdokodo algoritma LRTA*, ki je skladen z zgornjim preiskovalnim modelom:

1. Za vsako vozlišče n ∈ N nastavi h(n) na njegovo začetno vrednost $h_0(n)$.
2. Algoritem prične z izvajanjem v začetnem stanju. Nastavimo $n_{cur} \leftarrow$ s.
3. Ponavljaj sledeča koraka (a) in (b) dokler n_{cur} ni ciljno vozlišče; takrat se premakni v 4.
   (a) Pogled vnaprej in posodabljanje hevristične ocene h(n_{cur}) trenutnega stanja n_{cur} po pravilu: h(n_{cur}) ← max{h(n_{cur}), min {[k(n_{cur}, m) + h(m)]} | ∀ m; (n_{cur},m) ∈ A}.
   (b) Premakni se v naslednika m ∈ Succ(n_{cur}) vozlišča n_{cur}, ki ima v točki (a) najmanjšo vsoto.
4. Če sta rešitveni poti zadnjih dveh iteracij enaki, se ustavi, sicer nadaljuj v 2.

Za razumevanje zgornjega algoritma je potrebno definirati oznako n_{cur}. To je spremenljivka, v katero algoritem shranjuje trenutno stanje, torej mesto v grafu, kjer se algoritem trenutno nahaja. V točki 1 se nastavijo začetne vrednosti hevurističnih ocen (npr. na 0). V resnici hranimo tabelo hevurističnih ocen, ki se za dano vozlišče posodobi le, če je prišlo do spremembe vrednosti ocene. Ko imajo vsa vozlišča nastavljene hevristične ocene, se algoritem prične izvajati v izhodiščnem položaju s tako, da se spremenljivka n_{cur} nastavi na s. To se zgodi v koraku 2. Koraka 3a in 3b se ponavljata, dokler ne pridemo do ciljnega vozlišča. V koraku 3a pogledamo vnaprej množico vseh direktnih naslednikov trenutnega vozlišča n_{cur}. Izmed množice naslednikov Succ(n_{cur}) izberemo tistega, katerega vsota cene povezave (n_{cur}, m) ∈ A

in hevristične ocene h(m) je najmanjša. Pred napredovanjem v vozlišče m osvežimo podatek v tabeli hevurističnih ocen za stanje n_{cur}. V točki 3b se premaknemo v to vozlišče in preverimo, ali je vozlišče končno. V primeru, da ni, se vrnemo na korak 3a, sicer gremo na korak 2.

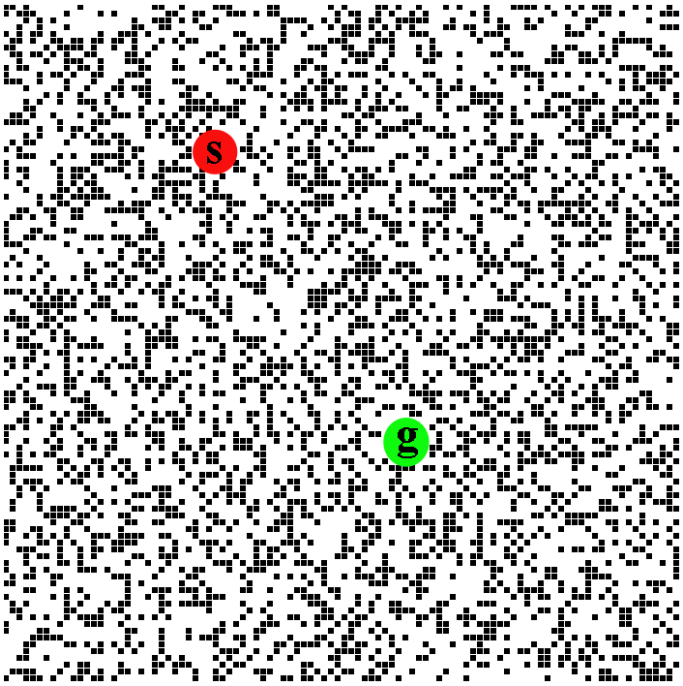
Zgornji algoritem je popoln, saj mu pri iskanju rešitve nikoli ne spodleti. Algoritem se vrne na korak 2 po koncu izvajanja koraka 3 in prične znova. Tako obnašanje delovanja algoritma je dopustno, saj je algoritem popoln. Vsaki ponovitvi sprehoda skozi graf pravimo epizoda (angl. *episode*). V vsaki epizodi LRTA* konvergira k optimalni rešitvi, torej najkrajši rešitveni poti iz začetnega vozlišča s k vozlišču g ∈ G.

Omenimo, da včasih pride do situacije, ko velja (n_{cur}, m_1) + h(m_1) = (n_{cur}, m_2) + h(m_2). V tem primeru algoritem naključno izbere enega od naslednikov m_1 ali m_2.

Algoritem konča izvajanje in vrne rešitev, ko v dveh zaporednih epizodah pride do ekvivalentnih rešitvenih poti, zaradi česar si vsako prejšnjo in trenutno rešitveno pot zapomnimo. Če sta enaki, se postopek zaključi, če nista, se trenutna rešitvena pot shrani kot prejšnja. Procedura se ponavlja, dokler ne dosežemo optimalne rešitvene poti.

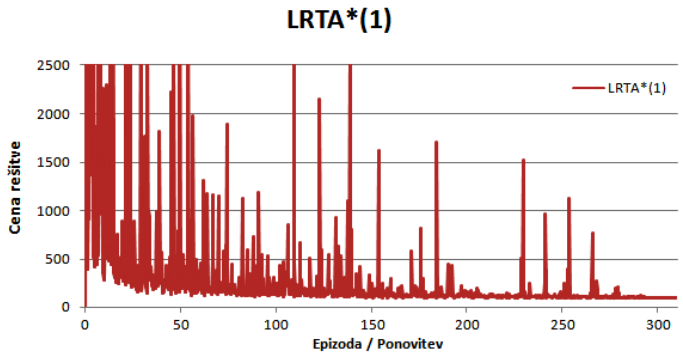
3.2 Učni proces algoritma LRTA*

Poglejmo si primer procesa konvergence algoritma LRTA* [8]. Imamo polje dimenzije 100×100 (glej sliko 3.1), na katerem so postavljene ovire označene kot črna polja. Ovire predstavljajo 35% celotne površine polja. Na sliki je označeno začetno stanje s in končno stanje g. Na poti med omenjenima stanjema so dovoljene poteze vertikalnih in horizontalnih premikov v vse smeri, razen, če je na poziciji želene smeri premika postavljena ovira. Začetno in končno stanje sta oddaljeni 100 enot (glede na Manhattansko razdaljo), med njima pa obstaja samo ena optimalna pot, katere cena znaša 122 enot.



Slika 3.1: Polje dimenzije 100×100 z ovirami (35%) ter stanji s in g.

Na opisanem in prikazanem problemu je bil pognan algoritem LRTA*. Rezultate si oglejmo na sliki 3.2.



Slika 3.2: Učni proces algoritma LRTA*(1).

Algoritem je bil pognan 300 krat (tj. 300 epizod). Na osi y so vrednosti cen rešitvenih poti (v tem primeru gre za število potez, ki so potrebne, da

iz stanja s preidemo v stanje g), na osi x pa zaporedna številka epizode. Kot vidimo na grafu, konvergenca ni monotona. Število potrebnih potez je v nekaterih primerih občutno večje kot v prejšnjih primerih. Uspešnost reševanja problema se lahko drastično spremeni, kljub temu, da je algoritem v prejšnjem koraku skorajda našel optimalno rešitveno pot.

Ta pojav ni značilen samo za primer 3.1, pač pa je neločljivo povezan z algoritmom LRTA*. Strmimo karakteristike in razlage za videno obnašanje algoritma v dve točki:

1. Četudi algoritem najde pot, ki je glede na ceno blizu optimalne poti, vseeno preišče še ostali prostor stanj z namenom odkritja boljše poti.

2. Algoritem LRTA* ne jamči stabilnega izboljševanja kakovosti rešitve, ker uporablja dopustno začetno hevristično funkcijo, ki ne precenjuje cen poti za nobeno stanje, kar je vzrok za znatno manjše ocenjene vrednosti od točnih vrednosti cen poti. Posledično se, tekom procesa učenja, hevristične ocene obiskanih vozlišč povečujejo, medtem ko hevristične ocene ne-obiskanih stanj ostanejo majhne. Ker algoritem LRTA* išče optimalne rešitve, to privede do preiskovanja še neraziskanih stanj (ki imajo manjše hevristične ocene), to pa je vzrok, da algoritem zaide na poti, ki so slabše od že najdenih.

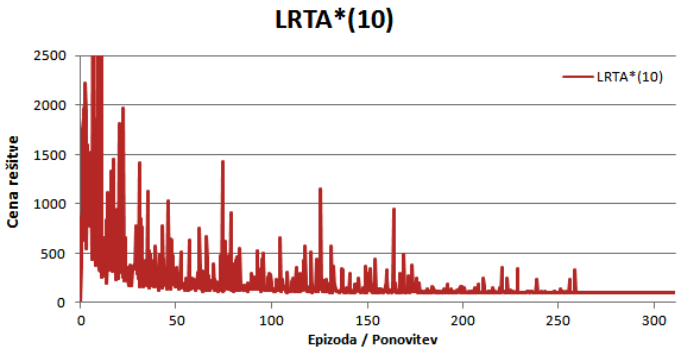
3.3 Inteligentni agenti

Ko govorimo o algoritmu LRTA*, moramo omeniti tudi agente. Definirajmo agenta kot entiteto, ki izvršuje akcije, potrebne za reševanje določenih problemov. Do sedaj smo govorili o delovanju algoritma, ki sledi korakom iz poglavja 3.1. Če se istočasno izvaja več pojavitev algoritma iz poglavja 3.1, pomeni, da imamo več agentov, torej entitet, ki izvajajo iste operacije. Agenti delujejo na način, da vsak izmed njih rešuje svoj problem, vendar pa rezultati enega agenta botrujejo odločitvam drugega agenta in na ta porazdeljen način računanja zagotavljajo hitrejše rešitve glede na število epizod. Agenti delujejo samostojno ter se učijo iz izkušenj. V splošnem je sicer priporočeno, da

se probleme razdeli na skupine, s katerimi se ukvarjajo specializirani agenti. V tem delu in spletni aplikaciji uporabljamo večagentni sistem, kjer so agenti ekvivalentni in vsi opravljajo isto nalogo – iskanje najkrajše poti.

Kadar algoritem izvaja več agentov hkrati, ga označimo z LRTA*(n), kjer je n število agentov, ki delujejo vzporedno. Primer delovanja algoritma LRTA*(2) je podan v poglavju 3.6.

Za primerjavo izboljšav z uporabo večagentnega algoritma, smo na primeru, ki je prikazan na sliki 3.1 pognali algoritem LRTA* z desetimi agenti. Rezultati so prikazani na sliki 3.3.



Slika 3.3: Učni proces algoritma LRTA*(10).

3.4 Asinhrono dinamično programiranje

Asinhrono dinamično programiranje temelji na spoznanju, da če vozlišče n leži na optimalni poti med začetnim s in končnim vozliščem g, potem je tudi pot iz s do n in pot iz n do g optimalna. Razdelitev problema na podprobleme nam omogoča postopno reševanje od spodaj navzgor [3].

V splošnem se pri dinamičnem programiranju pojavita rekurzivni relaciji, ki sta poimenovani pristop naprej in pristop nazaj [16]. Prednost dinamičnega programiranja je v tem, da za potrebe za delo z več objekti (agenti) ne potrebujemo vnaprej prihranjenih točnih količin pomnilnika, ampak ga lahko dodeljujemo dinamično v realnem času.

Zaradi primerjave z algoritmom LRTA* razložimo primer, ki uporablja rekurzivno relacijo pristop nazaj (algoritem začne pri cilju, kjer je znana točna vrednost h*(n) = 0); vsako vozlišče n ∈ N ima shranjeno hevristično oceno h(n), ki je približek točne vrednosti cene od vozlišča n do končnega vozlišča - h*(n). Vrednosti so na začetku postavljene na ∞, med izvajanjem algoritma pa približne vrednosti h(n) konvergirajo k pravim h*(n) in sicer tako, da ocenjene vrednosti vozlišč, ki so od ciljnega vozlišča oddaljene en nivo, konvergirajo v prvem koraku, drugi nivo v drugem itn. V najslabšem primeru je potrebno n korakov, da ocena začetnega vozlišča h(s) konvergira k točni vrednosti h*(s).

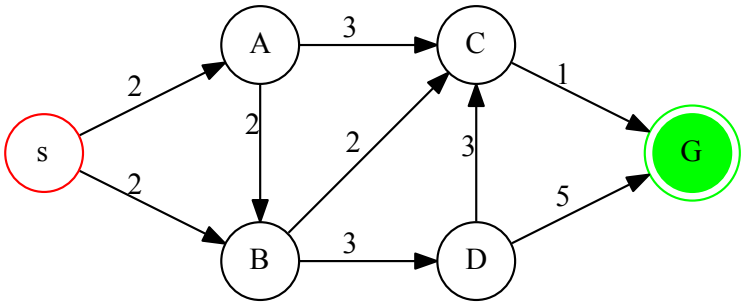
V opisanem primeru potrebuje algoritem za konvergenco vsakega vozlišča svojega agenta, kar pa ni sprejemljivo na realnih primerih, ker so števila stanj prevelika.

Boljši pristop preiskovanja s pomočjo agentov predstavlja algoritem LRTA*. Ta za razliko od zgoraj opisanega primera preiskovanje začne v začetnem vozlišču s, ocenjene vrednosti h(n) vozlišč pa so na začetku nastavljene na 0 s čimer zagotovimo najdbo optimalne rešitvene poti. Prednost algoritma LRTA*(n) je tudi v tem, da je n poljubno število (večje od nič). Za preiskovanje lahko torej uporabimo enega agenta, v primeru potrebe pohitritve delovanja, pa lahko uporabimo na primer 100 agentov.

Predstavitev delovanja algoritma LRTA* si ogledamo v poglavjih 3.5 (primer z enim agentom) in 3.6 (primer z dvema agentoma).

3.5 Predstavitev algoritma LRTA* na primeru

V tem razdelku bomo delovanje algoritma LRTA* komentirali po korakih. V pomoč nam je tabela, ki nazorno kaže spremembe hevrističnih ocen.



Slika 3.4: Graf, na katerem bomo obravnavali delovanje LRTA*.

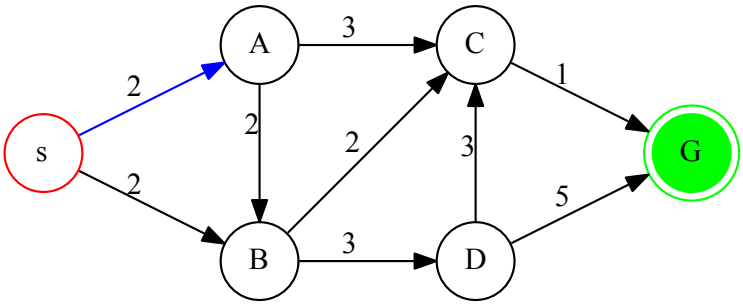
Primer grafa, na katerem si bomo ogledali delovanje algoritma LRTA*, je na sliki 3.4. Začetno vozlišče s je označeno z rdečo barvo, končno vozlišče G pa z zeleno barvo. Vrednosti na povezavah predstavljajo cene povezav, začetne hevristične ocene pa so prikazane v tabeli 3.1.

stanje	s	A	B	C	D	G
h(x)	0	0	0	0	0	0

Tabela 3.1: Začetne hevristične ocene grafa iz slike 3.3

3.5.1 Prva iteracija

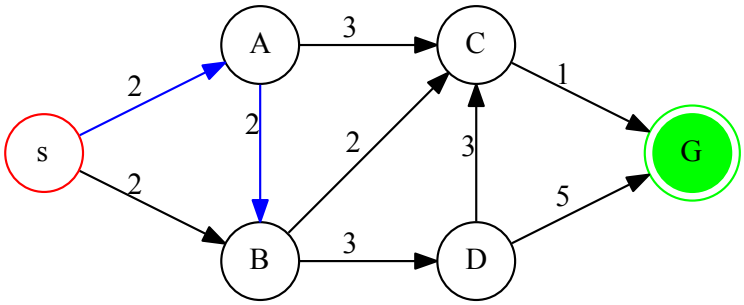
Hevristične ocene vseh stanj so nastavljene na 0, kot je prikazano v tabeli 3.1. Začetno stanje preiskovalnega prostora je prikazano na sliki 3.4. Izvajanje se prične v stanju s. Ker algoritem uporablja pogled vnaprej, vidi stanji A in B ter njuni hevristični oceni. Ker sta hevristični oceni vozlišč A in B enaki, razdalja do stanj A in B pa je tudi enaka, se algoritem naključno odloči, v katero stanje bo prešel. Recimo, da se odloči za stanje A. Pot je prikazana na sliki 3.5.



Slika 3.5: Prvi korak prve iteracije algoritma LRTA*.

Hevristična ocena stanja s se posodobi na vrednost 2, kot je prikazano v tabeli 3.2, trenutno stanje algoritma pa je A.

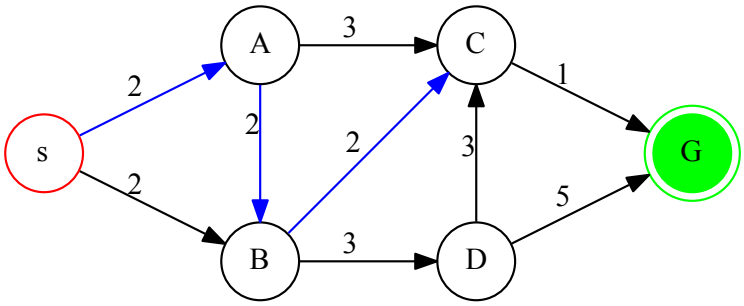
Algoritem lahko preide v stanje B ali v stanje C. Razdalja iz A v B je d(A, B) = 2, hevristična ocena pa je h(B) = 0. Vsota razdalje in hevristične ocene je 2. Če naredimo enak izračun za prehod v stanje C, dobimo vrednost 3. Ker je $2 < 3$, je pot predvidoma boljša, zato se algoritem odloči za prehod v stanje B, kot je prikazano na sliki 3.6.



Slika 3.6: Drugi korak prve iteracije algoritma LRTA*.

Hevristična ocena h(A) se spremeni na vrednost 2, kar je razvidno v tabeli 3.2.

Ko se algoritem nahaja v stanju B, ima na izbiro stanji C in D. Ker je bolj optimalno stanje C, algoritem nadaljuje pot v tej smeri, kar je razvidno iz slike 3.7.

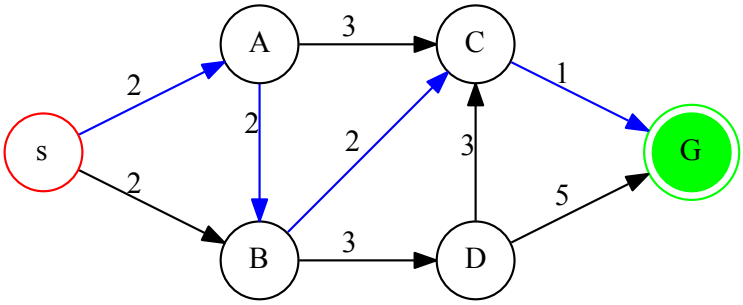


Slika 3.7: Tretji korak prve iteracije algoritma LRTA*.

Hevristična ocena stanja B se spremeni na vrednost 2, kar je vidno v

tabeli 3.2.

Za nadaljevanje iz stanja C je na izbiro eno izhodno stanje, ki pa je hkrati tudi ciljno, v katerega algoritem napreduje, kot vidimo na sliki 3.8.



Slika 3.8: Četrti korak prve iteracije algoritma LRTA*.

Hevristična ocena stanja C se spremeni na vrednost 1, kar je podano v tabeli 3.2. Ker se algoritem nahaja v ciljnem stanju, se izvajanje procedure ponovno začne v stanju s. Ponovno je na voljo prostor stanj, ki je prikazan na sliki 3.4, hevristične ocene vozlišč pa so tokrat enake tistim, ki so prikazane v tabeli 3.2.

stanje	s	A	B	C	D	G
h(x)	2	2	2	1	0	0

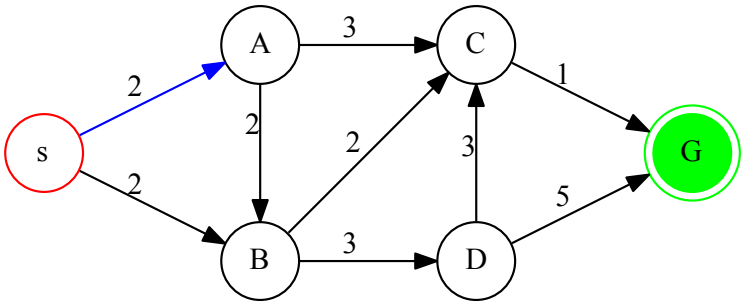
Tabela 3.2: Hevristične ocene stanj po prvi iteraciji

Definirajmo ceno rešitvene poti i-te iteracije z oznako c(i). Po prvi iteraciji je cena rešitvene poti ekvivalentna $c(1) = 7$.

Rešitvena pot prve iteracije: s $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ G.

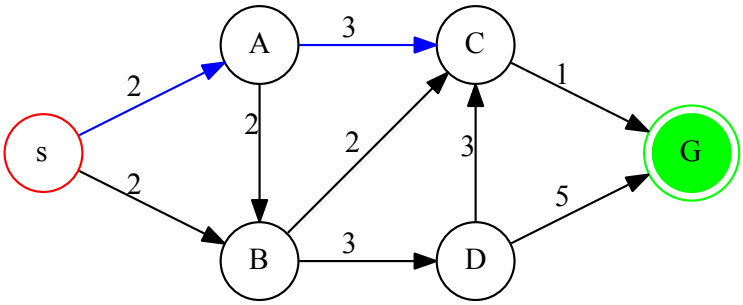
3.5.2 Druga iteracija

Izvajanje se ponovno začne v stanju s in pogledamo naslednike stanja s. Na voljo sta stanji A in B. Ker je pot do obeh stanj enaka, prav tako pa je enaka njuna hevristična ocena, se algoritem naključno odloči za eno izmed stanj. Recimo, da se odloči za stanje A. Slednji korak je prikazan na sliki 3.9.



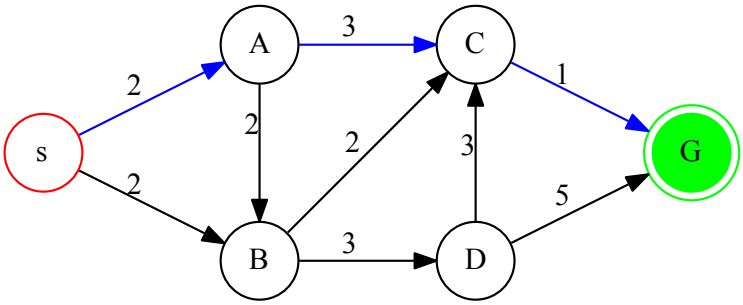
Slika 3.9: Prvi korak druge iteracije algoritma LRTA*.

Nahajamo se v stanju A, hevristična ocena stanja s se spremeni na vrednost 4, kot je prikazano v tabeli 3.3. Na izbiro za nadaljevanje sta stanji B(2+2) in C(3+1). Algoritem se naključno odloči za prehod v stanje C. Spremembe so prikazane na sliki 3.10.



Slika 3.10: Drugi korak druge iteracije algoritma LRTA*.

Hevristična ocena stanja A se spremeni na vrednost 4, kot je prikazano v tabeli 3.3. Iz stanja C vodi le ena izhodna povezava, katero algoritem prehodi v naslednjem koraku, ki je prikazan na sliki 3.11.



Slika 3.11: Tretji korak druge iteracije algoritma LRTA*.

Hevristična ocena stanja C se ne spremeni, saj je že točna. Posledično bo ostala enaka 1 tekom vseh preostalih iteracij, videli bomo tudi, da bodo počasi

vsa stanja konvergirala k svoji točni vrednosti. Nove hevristične vrednosti po izteku druge iteracije so prikazane v tabeli 3.3.

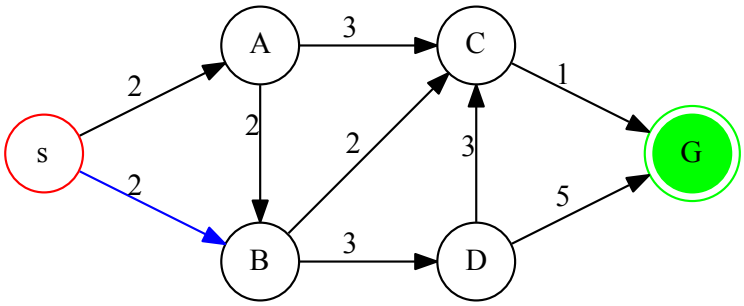
stanje	s	A	B	C	D	G
h(x)	4	4	2	1	0	0

Tabela 3.3: Hevristične ocene stanj po drugi iteraciji

Po drugi iteraciji je cena rešitvene poti enaka c(2) = 6.
Rešitvena pot druge iteracije: s → A → C → G.

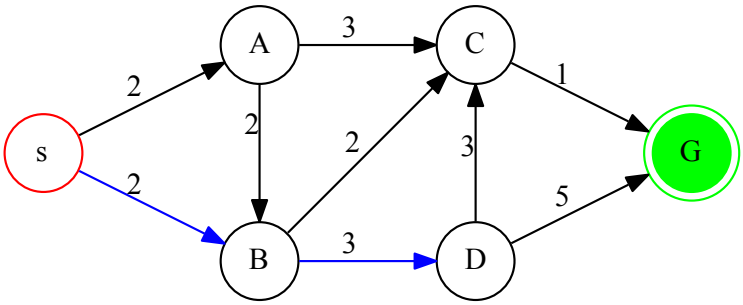
3.5.3 Tretja iteracija

Izvajanje se ponovno začne v stanju s (glej sliko 3.3). Tokrat je pot skozi stanje A ocenjena na dolžino 6, pot skozi stanje B pa na dolžino 4, zato LRTA* preide v stanje B. Prehod je označen na sliki 3.12.



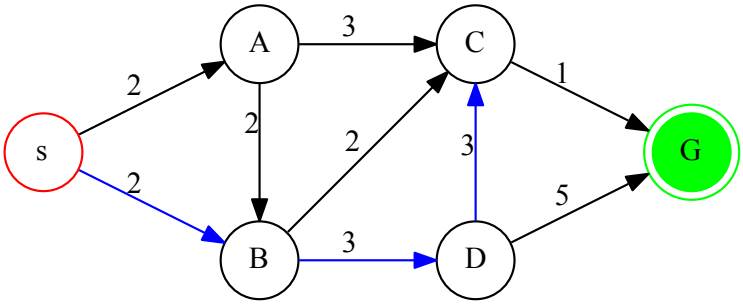
Slika 3.12: Prvi korak tretje iteracije algoritma LRTA*.

Hevristična ocena stanja s ostane nespremenjena, kot je razvidno iz tabele 3.4. Za izhod iz stanja B sta na izbiro stanji C(2+1) in D(3+0). Optimalna pot je predvidoma enaka v obeh smereh, naključno pa se za nadaljevanje izbere stanje D, kot je prikazano na sliki 3.13.



Slika 3.13: Drugi korak tretje iteracije algoritma LRTA*.

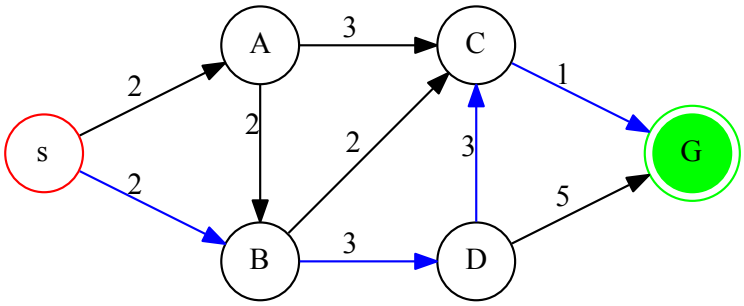
Hevristična ocena stanja B se spremeni na vrednost 3, kot je razvidno iz tabele 3.5. Za nadaljevanje iz trenutnega stanja D sta na izbiro stanji C(3+1) in stanje G(5+0). Ker je predvidoma bolj optimalna pot skozi stanje C, algoritem nadaljuje v stanje C. Sprememba je prikazana na sliki 3.14.



Slika 3.14: Tretji korak tretje iteracije algoritma LRTA*.

Hevristična ocena stanja D se spremeni na vrednost 4, kot je razvidno iz tabele 3.6. Iz stanja C se preiskovanje nadaljuje v stanje G, kot je prikazano

na sliki 3.15.



Slika 3.15: Četrti korak tretje iteracije algoritma LRTA*.

Hevristična ocena stanja C ostane nespremenjena.

stanje	s	A	B	C	D	G
h(x)	4	4	3	1	4	0

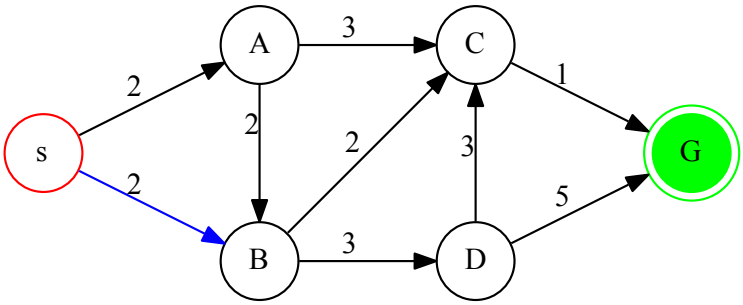
Tabela 3.4: Hevristične ocene stanj po tretji iteraciji

Po tretji iteraciji je cena rešitvene poti enaka c(3) = 9.

Rešitvena pot tretje iteracije: s → B → D → C → G.

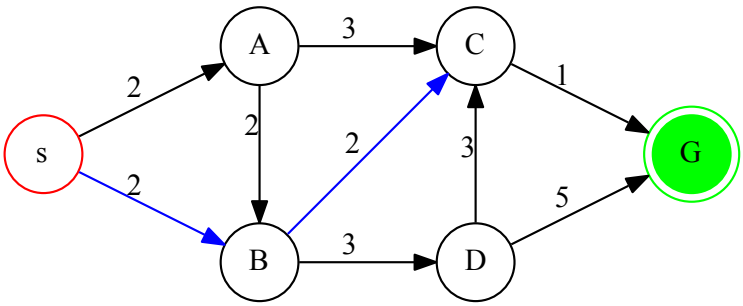
3.5.4 Četrta iteracija

Preiskovanje se prične v stanju s (glej sliko 3.4). Izhodne povezave vodijo v stanji A(2+4) in B(2+3). Zaradi iskanja optimalne poti algoritem nadaljuje v stanje B, kot je prikazano na sliki 3.16.



Slika 3.16: Prvi korak četrte iteracije algoritma LRTA*.

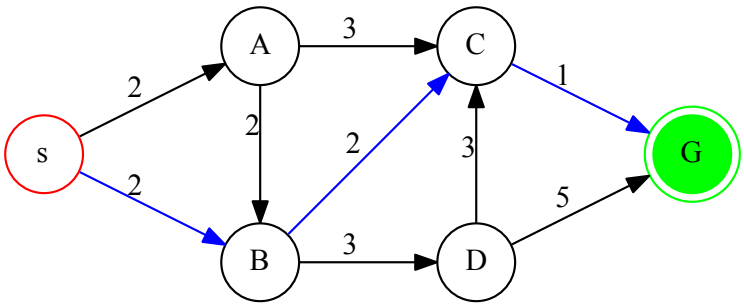
Hevristična ocena stanja s se poveča na vrednost 5, kot je razvidno iz tabele 3.5. Naslednika stanja B sta C(2+1) in D(3+4). Pot se nadaljuje skozi bolj optimalno stanje, torej skozi vozlišče C, kot je prikazano na sliki 3.17.



Slika 3.17: Drugi korak četrte iteracije algoritma LRTA*.

Hevristična ocena stanja B se ne spremeni. V stanju C je na izbiro samo ena izhodna povezava, stanje G, kamor se pot nadaljuje, kot je prikazano na

sliki 3.18.



Slika 3.18: Tretji korak četrte iteracije algoritma LRTA*.

Hevristična ocena stanja C je nespremenjena, kot je razvidno iz tabele 3.5, kjer so podane hevristične ocene za vsa stanja po končani četrti iteraciji.

stanje	s	A	B	C	D	G
h(x)	4	4	3	1	4	0

Tabela 3.5: Hevristične ocene stanj po četrti iteraciji

Po četrti iteraciji je cena rešitvene poti enaka $c(4) = 5$.

Rešitvena pot četrte iteracije: $s \rightarrow B \rightarrow C \rightarrow G$.

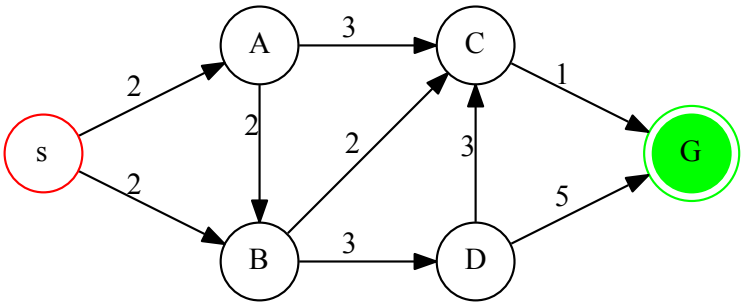
3.5.5 Peta iteracija

Potek iskanja optimalne poti je v peti iteraciji identičen iskanju v četrti iteraciji, zato ga ne bomo obravnavali posebej. Algoritem LRTA* si zapomni zadnji dve obiskani poti (zadnjo in predzadnjo), ki pa ju na koncu vsake iteracije primerja. Če sta enaki, pomeni, da je algoritem našel najkrajšo pot med vozlišči s in G. Tedaj se njegovo izvajanje konča.

Opozorimo še, da so hevristične ocene konvergirale k točnimi cenam že v prejšnji iteraciji, kot se lahko prepričamo iz slike 3.4 in tabele 3.5.

3.6 Predstavitev algoritma LRTA*(n) na primeru

V tem poglavju je opisano delovanje algoritma LRTA*(n). Za spremenljivko n vzamemo vrednost 2, kar pomeni, da algoritem vzporedno poganjata dva agenta, ki imata skupno tabelo hevurističnih ocen. Programsko se vzporedni agenti realizirajo kot niti, ki delujejo neodvisno in asinhrono. Ker koraki algoritmov (v splošnem) ne trajajo vedno enako časa, se zgodi, da, na primer, druga nit z izvajanjem konča prej kot prva. Zaradi lažjega razumevanja delovanja algoritma LRTA*(2), v tem razdelku predpostavljamo, da prvi agent (nit) konča pred drugim.



Slika 3.19: Graf, na katerem bomo obravnavali delovanje LRTA*(2).

Primer grafa, na katerem si ogledamo delovanje algoritma LRTA*(2) je podan na sliki 3.19. Enako, kot v poglavju 3.5, je začetno vozlišče s označeno z rdečo barvo, končno vozlišče G pa z zeleno barvo. Vrednosti na povezavah predstavljajo cene povezav med vozlišči, začetne hevristične ocene imajo

vrednosti 0 in so prikazane v tabeli 3.6. Zaradi nazornosti delovanja, je vsak agent predstavljen z drugo barvo. Pot prvega agenta je označena z rdečo barvo, pot drugega pa z mordro barvo. Z zeleno barvo so označeni deli poti, ki jih prehodita oba agenta.

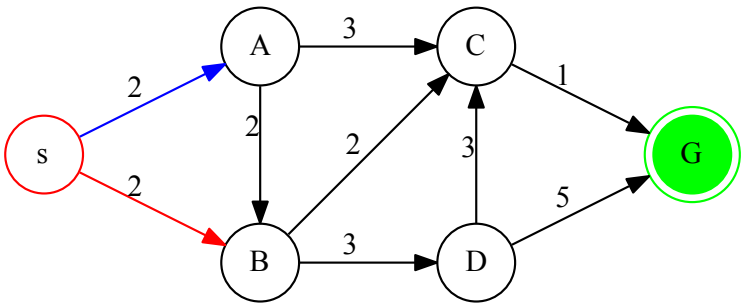
stanje	s	A	B	C	D	G
h(x)	0	0	0	0	0	0

Tabela 3.6: Hevristične ocene začetnega grafa iz slike 3.18

3.6.1 Prva iteracija

Agenta se nahajata v začetnem stanju s. Izhodne povezave vodijo v stanji A(2+0) in B(2+0). Ker sta izbiri enaki, se prvi agent premakne po naključni izbiri v stanje B. Hevristična ocena stanja s postane 2, kot je podano v tabeli 3.7.

Na vrsti je drugi agent (v realnosti se niti izvajajo neodvisno, tako da drugi agent ne čaka prvega), ki ima na izbiro stanji A(2+0) in B(2+0). Naključno izbere prehod v stanje A. Vrednost hevristične ocene stanja s ostane 2. Poti, ki jih opravita agenta, sta vidni na sliki 3.20.

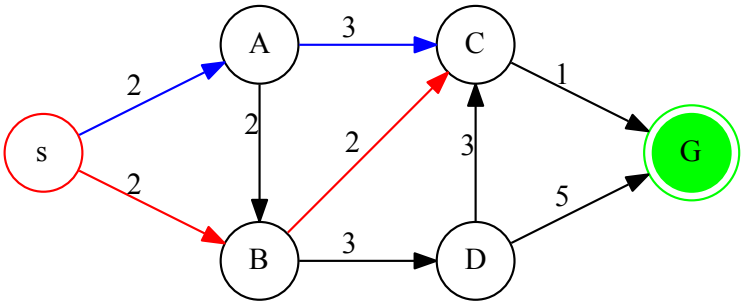


Slika 3.20: Prvi korak prve iteracije algoritma LRTA*(2).

Na vrsti je prvi agent, ki se nahaja v stanju B, iz katerega lahko nadaljuje v stanje C(2+0) ali stanje D(3+0). Ker imata obe vozlišči enako hevristično oceno, se agent za nadaljevanje odloči za prehod krajše poti - preide v stanje C. Hevristična ocena vozlišča B postane 2.

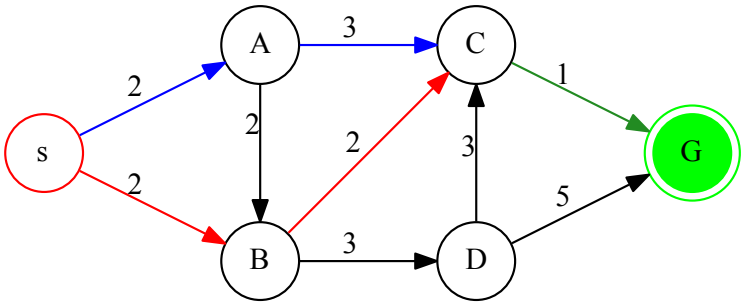
Drugi agent se nahaja v stanju A. Na izbiro ima stanje B(2+2), kateremu je ravno prvi agent spremenil hevistično oceno na vrednost 2, in stanje C(3+0). Ker je stanje C bolj optimalno, se drugi agent premakne v stanje C. Hevristična ocena stanja A postane 3, kot je prikazano v tabeli 3.7.

Poti agentov so prikazane na sliki 3.21.



Slika 3.21: Drugi korak prve iteracije algoritma LRTA*(2).

Agenta nahajata v stanju C. Na voljo je le izhodna povezava v stanje G. Ker gresta agenta po isti poti, je ta pot označena z zeleno barvo. Hevristična ocena stanja C se spremeni na vrednost 1, kar sledi iz G(1+0). Tretji korak prve iteracije je prikazan na sliki 3.22, vrednosti hevrističnih ocen po končani prvi iteraciji pa so podane v tabeli 3.7.



Slika 3.22: Tretji korak prve iteracije algoritma LRTA*(2).

stanje	s	A	B	C	D	G
h(x)	2	3	2	1	0	0

Tabela 3.7: Hevristične ocene stanj po končani prvi iteraciji.

Rešitvena pot prve iteracije (agent 1): s → B → C → G.

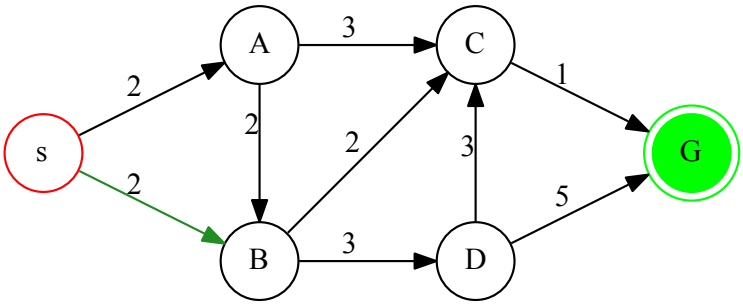
Cena rešitvene poti prve iteracije (agent 1): c(1) = 5.

Rešitvena pot prve iteracije (agent 2): s → A → C → G.

Cena rešitvene poti prve iteracije (agent 2): c(1) = 6.

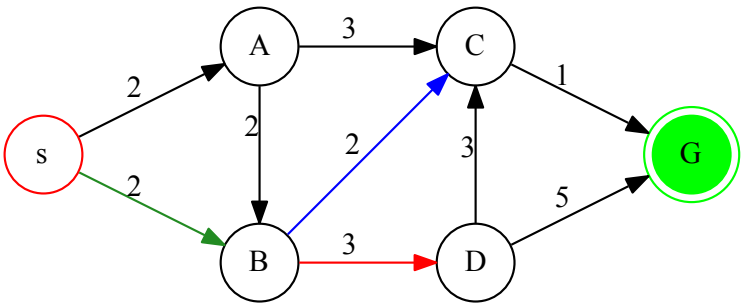
3.6.2 Druga iteracija

Agenta se vrneta v stanje s in nadaljujeta preiskovanje. Za nadaljevanje izbirata med vozlišči A(2+3) in B(2+2). Ker je pot do cilja predvidoma bolj optimalna skozi stanje B, preideta vanj oba. Korak je označen na sliki 3.23. Hevristična ocena stanja s se spremeni na vrednost 4.



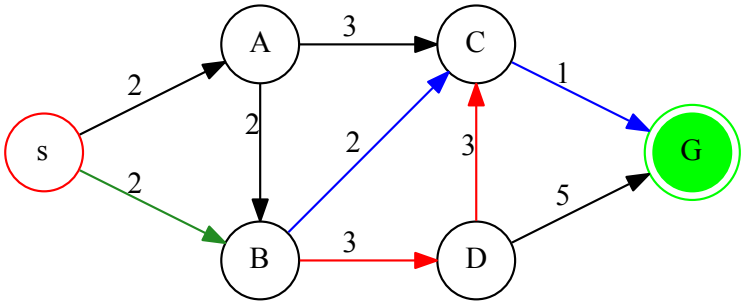
Slika 3.23: Prvi korak druge iteracije algoritma LRTA*(2).

Agenta se nahajata v stanju B, ki ima izhodne povezave do stanj C(2+1) in D(3+0). Seštevka vrednosti stanj in hevrističnih ocen sta enaka, zato se agenta ponovno naključno odločita za nadaljevanje. Prvi agent nadaljuje v stanje D. Hevristična ocena stanja B se poveča na vrednost 3. Drugi agent naključno izbere nadaljevanje skozi stanje C. Hevristična ocena stanja B ostane 3. Korak je prikazan na sliki 3.24.



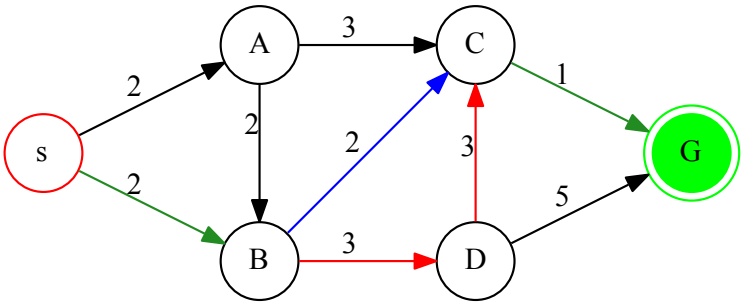
Slika 3.24: Drugi korak druge iteracije algoritma LRTA*(2).

V tretjem koraku druge iteracije se prvi agent nahaja v stanju D, drugi pa v stanju C. Prvi agent za nadaljevanje izbira med stanji C(3+1) ter G(5+0). Nadaljuje v stanje C. Hevristična ocena vozlišča D postane 4, kar je prikazano v tabeli 3.8. Drugi agent nadaljuje pot v končno vozlišče - stanje G. Hevristična ocena stanja C ostane 1. Korak iteracije je prikazan na sliki 3.25.



Slika 3.25: Tretji korak druge iteracije algoritma LRTA*(2).

Prvi agent se v četrtem koraku tretje iteracije nahaja v stanju C. Pot nadaljuje po edini izhodni povezavi v stanje G. Hevristična ocena stanja C ostane nespremenjena. Drugi agent se v četrtem koraku tretje iteracije nahaja v vozlišču G in čaka prvega agenta, da preide v ciljno vozlišče. Opozorimo, da v realnem svetu drugi agent zaradi neodvisnosti nadaljuje s preiskovanjem in ne čaka prvega. Ker želimo vizualno čim bolj jasno prikazati potek korakov, smo drugega agenta namerno zaustavili. Korak je prikazan na sliki 3.26, nove vrednosti hevrističnih ocen pa so podane v tabeli 3.8.



Slika 3.26: Četrti korak druge iteracije algoritma LRTA*(2).

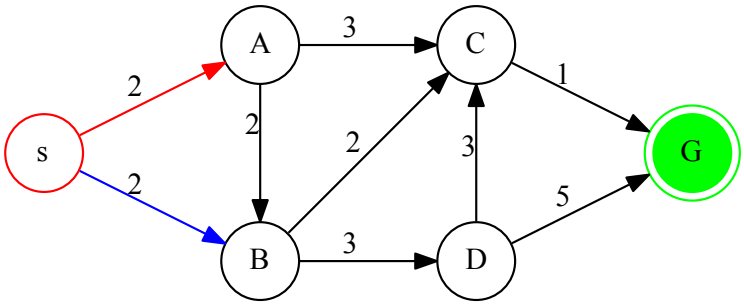
stanje	s	A	B	C	D	G
h(x)	4	3	3	1	4	0

Tabela 3.8: Hevristične ocene stanj po končani drugi iteraciji.

Rešitvena pot druge iteracije (agent 1): s → B → D → C → G.
Cena rešitvene poti druge iteracije (agent 1): c(2) = 9.
Rešitvena pot druge iteracije (agent 2): s → B → C → G.
Cena rešitvene poti druge iteracije (agent 2): c(2) = 5.

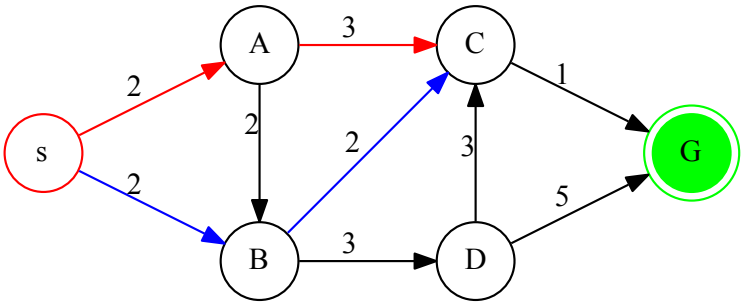
3.6.3 Tretja iteracija

Prvi korak tretje iteracije se ponovno prične v začetnem stanju s. Agenta izbirata med stanji A(2+3) in B(2+3). Prvi agent se naključno odloči za nadaljevanje preiskovanja skozi stanje A. Hevristična ocena stanja s postane 5, kot je prikazano v tabeli 3.9. Drugi agent za nadaljevanje naključno izbere vozlišče B. Hevristična ocena stanja s se ne spremeni. Koraka agentov sta prikazana na sliki 3.27.



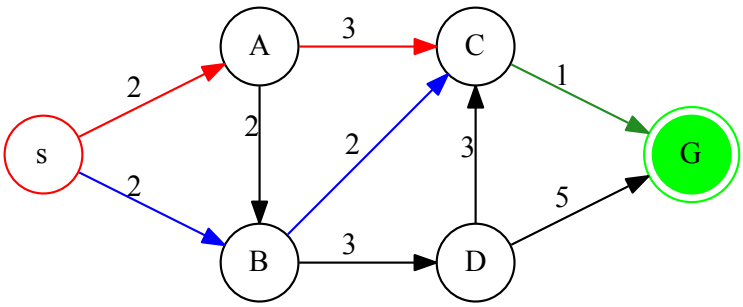
Slika 3.27: Prvi korak tretje iteracije algoritma LRTA*(2).

V drugem koraku tretje iteracije se prvi agent nahaja v stanju A. Izhodne poti vodijo v stanji B(2+3) in C(3+1). Ker je stanje C bolj optimalno, se preiskovanje nadaljuje v njegovi smeri. Hevristična ocena stanja A postane 4, kot je podano v tabeli 3.9. Drugi agent ima na izbiro stanji C(2+1) in D(3+4). Ker je vrednost seštevka stanja C manjša od vrednosti seštevka stanja D, se preiskovanje nadaljuje v smeri stanja C. Hevristična ocena stanja B ostane nespremenjena. Korak iteracije je prikazan na sliki 3.28.



Slika 3.28: Drugi korak tretje iteracije algoritma LRTA*(2).

Agenta se sedaj nahajata v stanju C. Nadaljujeta lahko le v stanje G(1+0), kot je prikazano na sliki 3.29. Ker se po tem koraku agenta nahajata v ciljnem vozlišču, se iteracija zaključi. Spremenjene vrednosti hevrističnih ocen po tretji iteraciji so podane v tabeli 3.9.



Slika 3.29: Tretji korak tretje iteracije algoritma LRTA*(2).

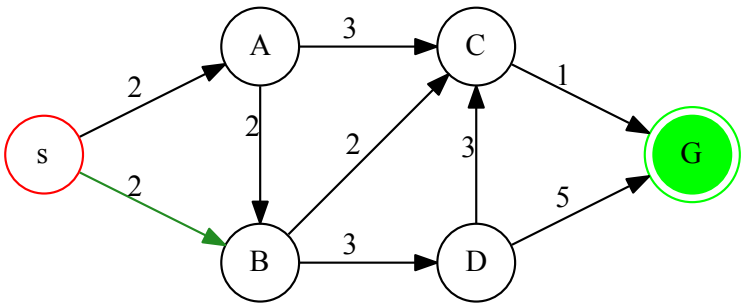
stanje	s	A	B	C	D	G
h(x)	5	4	3	1	4	0

Tabela 3.9: Hevristične ocene stanj po končani tretji iteraciji.

Rešitvena pot tretje iteracije (agent 1): s → A → C → G.
Cena rešitvene poti tretje iteracije (agent 1): c(3) = 6.
Rešitvena pot tretje iteracije (agent 2): s → B → C → G.
Cena rešitvene poti tretje iteracije (agent 2): c(3) = 5.

3.6.4 Četrta iteracija

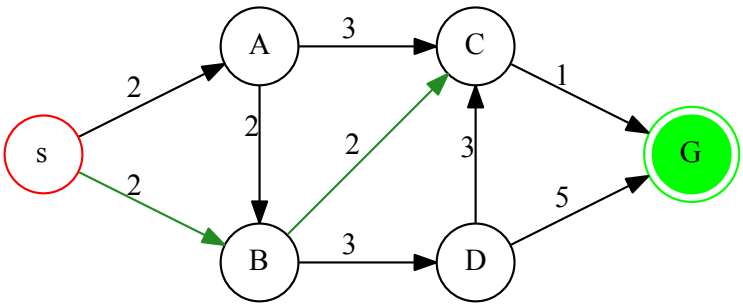
Agenta preiskovanje začneta v začetnem stanju s. Izbirata med stanji A(2+4) in B(2+3). Ker je stanje B bolj optimalno, oba nadaljujeta po poti v stanje B. Hevristična ocena stanja s ostane nespremenjena, kot je razvidno iz tabele 3.10. Korak je prikazan na sliki 3.30.



Slika 3.30: Prvi korak četrte iteracije algoritma LRTA*(2).

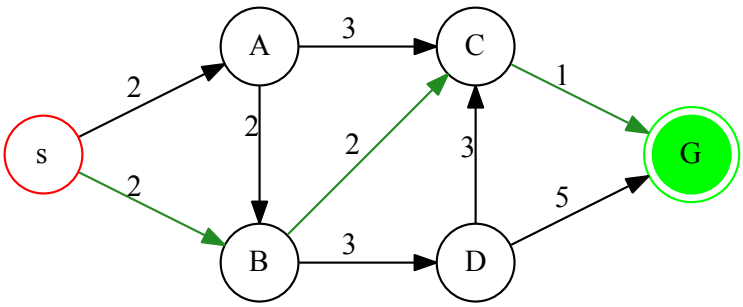
Agenta se nahajata v stanju B. Izhodne povezave omogočajo prehod v stanji C(2+1) ter D(3+4). Odločita se za prehod v stanje C, ker je bolj opti-

malno, kot stanje D. Hevristične ocene se pri drugem koraku četrte iteracije ne spreminjajo. Korak agentov je prikazan na sliki 3.31.



Slika 3.31: Drugi korak četrte iteracije algoritma LRTA*(2).

Agenta se nahajata v stanju C. Na izbiro imata le prehod v stanje G(1+0), ki je končno vozlišče. Korak je prikazan na sliki 3.32, hevristične ocene po končani četrti iteraciji pa so podane v tabeli 3.10.



Slika 3.32: Tretji korak četrte iteracije algoritma LRTA*(2).

stanje	s	A	B	C	D	G
h(x)	5	4	3	1	4	0

Tabela 3.10: Hevristične ocene stanj po končani četrti iteraciji.

Rešitvena pot četrte iteracije (agent 1): s $\rightarrow$ B $\rightarrow$ C $\rightarrow$ G.
Cena rešitvene poti četrte iteracije (agent 1): c(4) = 5.
Rešitvena pot četrte iteracije (agent 2): s $\rightarrow$ B $\rightarrow$ C $\rightarrow$ G.
Cena rešitvene poti četrte iteracije (agent 2): c(4) = 5.

3.6.5 Peta iteracija

Rešitvini poti četrte in pete iteracije sta enaki, zato pete iteracije ne obravnavamo posebej. Tudi vrednosti hevrističnih ocen zadnjih dveh iteracij so enake. Izvajanje algoritma LRTA*(2) se po peti iteraciji konča zaradi enakih končnih rezultatov (rešitvene poti) dveh zaporednih iteracij. Opozorimo, da morajo biti za zaustavitev delovanja algoritma rešitvene poti vseh agentov enake v zadnjih dveh ponovitvah algoritma.

3.7 Posebni primeri algoritma LRTA*(n)

Lahko se zgodi, da hevristične ocene nekaterih vozlišč ne konvergirajo k točnim cenam. Situacija delne konvergence nastane zaradi naključne izbire vozlišča v primeru, ko imamo v nahajajočem vozlišču več enakovrednih potomcev. Če se algoritem naključno vedno odloči za istega, bo pot skozi preostala vozlišča ostala neraziskana. Primer ne vpliva na algoritem pri iskanju najkrajše poti, če je le-ta dopusten (hevristična ocena neraziskanih vozlišč se lahko le poveča ali ostane enaka).

Problem se pojavi, če od začetnega do končnega vozlišča obstaja več kot ena optimalna pot. Algoritem na razpotju enakih potomcev izbira naključno, zato obstaja možnost, da bo vedno izbral drugo pot kot v prejšnji iteraciji.

Posledično dve zaporedni iteraciji nista enaki, kar povzroči neskončno izvajanje. Za reševanje tega problema po določenem številu iteracij primerjamo cene rešitvenih poti zaporednih iteracij. Če se ne spreminjajo, algoritem konča in vrne eno izmed rešitev.

Poglavje 4

Uporabljena orodja in tehnologije

V tem poglavju bomo našteli jezike (programske, opisne in skriptne) in tehnologije, strežnike ter orodja, ki so omogočila nastanek spletne aplikacije za delo z algoritmom LRTA*. Pri vsaki komponenti bomo pogledali njeno zgodovino, današnjo uporabnost ter našteli dobre in slabe lastnosti.

Spletna aplikacija je shranjena na aplikacijskem strežniku Apache Tomcat 7. Za prikazovanje spletne vsebine so uporabljene novejše tehnologije kot sta HTML5 in kaskadne stilske podloge CSS3. Skriptni jezik javascript je uporabljen na mestih, kjer se zahteva preverjanje regularnih izrazov. Nosilec kontrolnih funkcij je programski jezik java. Ta izvaja algoritem LRTA*, poleg tega pa je njegova funkcija še nadzor toka izvajanja spletne aplikacije, tj. določanje metod, ki naj se izvedejo ob klikih na gumbe. V javi je ustvarjen tudi razred, ki piše v dokumente s končnico ".dot". Te se shranjujejo na spletni strežnik Apache2, potrebne pa so za risanje grafov v označevalnem jeziku DOT. Datoteke so posredovane na javni spletni strežnik WebDot, ki izriše grafe opisane v prejetih datotekah in vrne slike grafov, ki jih prikažemo v spletni aplikaciji. Izdelana aplikacija omogoča shranjevanje lastnosti grafov, kar je realizirano z jezikom SQL, ki shranjuje podatke na podatkovni strežnik MySQL 5.1. Podrobnejšo strukturo spletne aplikacije in povezavo komponent

si bomo ogledali v poglavju 5.

4.1 Jeziki in tehnologije

V tem razdelku si bomo ogledali jezike in tehnologije, ki so bile uporabljene za izdelavo spletne aplikacije učenja delovanja algoritma LRTA*. Pogledali si bomo jezike, kot so java, javascript, SQL, DOT in tehnologije, kot sta HTML(5) in CSS3.

4.1.1 Programski jezik java

Programski jezik java je splošno namenski, objektno orientiran programski jezik. Je eden izmed najbolj uporabljenih programskih jezikov, zlasti v spletnih aplikacijah, ki delujejo po načelu odjemalec-strežnik [26].

Nastanek programskega jezika java sega v leto 1990, ko so v podjetju Sun Microsystems (sedaj pod okriljem Oracle Corporation) začeli razvijati nov objektni programski jezik, ki bi omogočal programiranje različnih elektronskih naprav, ne glede na raznolikost operacijskih sistemov in strojne opreme naprav. Ciljna ideja je bila, da se razvije programski jezik, ki bi razvijalcem aplikacij omogočal princip 'napiši enkrat, poganjaj povsod' ali WORA (angl. *write once, run anywhere*), kar pomeni, da bi razvijalci napisali programsko kodo ne eni platformi, za poganjanje programa na drugi platformi pa ne bi bilo potrebno ponovno prevajati (angl. *compile*) izvorne kode.

Leta 1998 je izšla različica 1.2, ki se je glede na prejšnjo različico razcvetela. Primerjalni podatki so prikazani v tabeli 4.1.

Različica	Leto izida	Število paketov	Števili razredov in vmesnikov	Število metod in lastnosti
1.1	1997	23	504	5.478
1.2	1998	62	1.781	20.935

Tabela 4.1: Primerjava različic jave 1.1 in 1.2

Tako se je java s prihodom različice 1.2 preimenovala v Java 2 SE (angl. *standard edition*). Java 2 pa je poleg standardne različice prinesla še dve različici. Poglejmo si vse tri različice:

- Java SE (angl. *Java Standard Edition*) – je platforma, ki vsebuje vse potrebne knjižnice in vmesnike oz. API (angl. *application programming interface*) za delo z javo.

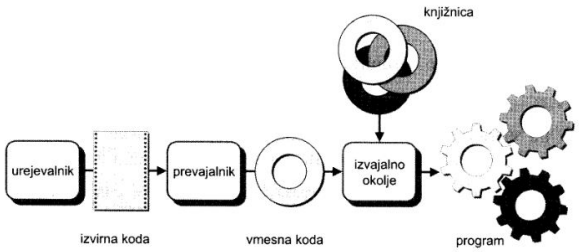
- Java ME (angl. *Java Micro Edition*) – je platforma, ki je namenjena razvijanju aplikacij za mobilne telefone, dlančnike in vgradne sisteme. Gre za okrnjeno različico jave SE z namenom hitrejšega delovanja na manj zmogljivih sistemih.

- Java EE (angl. *Java Enterprise Edition*) – je platforma, ki je namenjena razvijanju aplikacij za programske strežnike. Vsebuje vse knjižnice in vmesnike, kot jih vsebuje Java SE, poleg tega pa vsebuje še dodatne vmesnike, knjižnice in metode, ki so v uporabi pri spletnem programiranju.

Za razvoj naše spletne aplikacije uporabljamo različico Java EE. Ker gre za spletno aplikacijo, potrebujemo različico, ki je sposobna komunicirati s strežniki in vsebuje vse potrebne knjižnice in metode za delo. Uporabljamo tudi servlete, skriplete in javine strežniške strani JSP (angl. *Java Server Pages*). Servleti omogočajo dinamično gradnjo spletnih strani z javansko kodo [7]. Servlet je javanski objet, ki se odzove na zahtevek HTTP (angl. *Hypertext Transfer Protocol*) in se izvede na strežniku. Skripleti omogočajo vstavljanje programske kode direktno v JSP dokumente, ki vsebujejo kodo jave HTML.

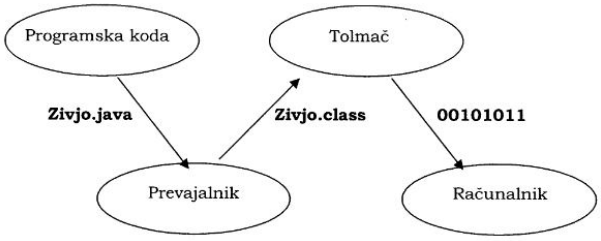
Prednost jave je v tem, da izvorne kode prevajalnik ne prevede direktno v strojno kodo, temveč jo prevede v vmesno kodo, ki se izvaja v javanskem izvajalnem okolju oz. JVM (angl. *java virtual machine*), vmesna koda pa se nato prevede v strojno kodo. Ta potek omogoča, da prevedeni program

poganjamo povsod, kjer je na voljo izvajalno okolje oz. JRE (angl. *java runtime environment*). Proces prevajanja izvorne kode do končnega programa je prikazan na sliki 4.1 (povzeto po [5]).



Slika 4.1: Življenjski ciklus javanske kode od urejevalnika do programa.

Poglejmo si, kako je slika 4.1 predstavljena v računalniku. Vsak program, ki ga napišemo v programskem jeziku java ima končnico '.java'. Temu pravimo izvorna koda, ki se s pomočjo javanskega prevajalnika prevede v vmesno kodo, ki ima končnico '.class'. Nato nastopi JVM, ki mu pravimo tudi tolmač. Negova naloga je prevesti vmesno kodo v strojno kodo (torej zaporedje ničel in enic), katero poganjajo različne naprave. Poudarimo, da se izvorna koda napiše na katerikoli napravi, ki pa mora imeti tolmača za prevajanje vmesne kode v strojno kodo. Razlog je v tem, da se enaka strojna koda ne bi zagnala na računalnikih z drugačno arhitekturo ali drugačnim operacijskim sistemom. Delovanje opisanega postopka je prikazano na sliki 4.2.



Slika 4.2: Prehod iz izvorne kode k strojni kodi (povzeto po [6]).

Naštejmo še nekaj lastnosti jave [26]: enostavnost, predmetnost, neodvisnost, trdoživost, dinamičnost, visoka zmogljivost.

4.1.2 Skriptni jezik javascipt

Skriptni jeziki javascript igra veliko vlogo v spletnih brskalnikih, saj njegova vgradnja v spletne strani omogoča izvajanje skript na odjemalčevi strani (angl. *client-side scripts*), ki skrbijo za interakcijo z uporabniki, nadzorovanje obnašanja brskalnikov ter dovoljuje spreminjanje prikazane vsebine znotraj spletne strani [27]. Javascript je pogosto uporabljen s strani programerjev, ki skrbijo za izvajanje na strani strežnika (angl. *server-side programming*). Javascript se uporablja tudi izven okvirov spleta, na primer v dokumentih PDF in v namiznih aplikacijah.

Nastanek skriptnega jezika javascript sega v leto 1995, ko je bil ustvarjen pod vodstvom podjetja Netscape. Sprva se je javascript imenoval LiveScript, ko pa se je pojavil programski jezik java in bil vgrajen v spletni brskalnik podjetja Netscape (Netscape Navigator), se je skriptni jezik preimenoval v javascript in do danes ohranil svoje ime. Vredno si je še zapomniti, da je bil jezik javascript razvit neodvisno od jezika java, s katero si deli številne lastnosti in strukture [2].

V naši spletni aplikaciji je jezik javascript uporabljen za preverjanje regularnih izrazov. Na področju računalništva je regularni izraz definiran kot zaporedje znakov, ki tvorijo nek vzorec. Uporaba regularnih izrazov je v spletu pogosto uporabljena v kombinaciji z vnosnimi polji (na primer, ali je vnešeno besedilo veljaven elektronski naslov ali ne).

4.1.3 Jezik SQL

Strukturirani povpraševalni jezik za delo s podatkovnimi bazami SQL (angl. *Structured Query Language*) je standardiziran in zelo razširjen povpraševalni jezik za delo s podatkovnimi bazami [30]. Razvijati so ga začeli leta 1970 v podjetju IBM, danes pa predstavlja standard za delo s podatkovnimi bazami

oz. zbirkami. Z jezikom lahko gradimo poizvedbe in manipuliramo s podatki v bazah. Poizvedbe so programski stavki namenjeni pridobivanju podatkov iz baz. S podatki manipuliramo preko stavkov, ki vstavljajo in brišejo podatke iz baz ali pa jih posodabljajo.

V spletni aplikaciji so programski stavki poizvedb in manipulacij s podatki, za shranjevanje podatkov na strežnik MySQL, napisani v jeziku SQL. Potreba po shranjevanju podatkov v izdelani spletni aplikaciji nastane zaradi zagotavljanja možnosti shranjevanja grafov in dela z vnaprej pripravljenimi grafi. Več o strežniku MySQL sledi v poglavju 4.2.3.

4.1.4 Jezik DOT

Jezik DOT se uporablja za opisovanje grafov. Velja za preprost jezik, ki ga ljudje hitro osvojimo. Sintakso, ki opisuje sestavo grafov, shranimo v dokumente, ki imajo kočnico ".gv"ali ".dot"[15]. Te dokumente bere kar nekaj programov, izdelana spletna aplikacija pa se posluži programa oz. paketa GraphViz, katerega nastanek sega v leto 1988.

V spletni aplikaciji je uporabljen javni spletni dot strežnik (angl. *public WebDot server*), ki je na voljo od 20. avgusta 2013 [29]. Izdelovanje slik grafov v spletni aplikaciji poteka v treh korakih; najprej z java kodo napišemo dokument s končnico ".dot", ki se posreduje spletnemu strežniku WebDot, ki vrne sliko opisanega grafa. Prejeto sliko vključimo v spletno aplikacijo na primerno mesto.

4.1.5 Označevalni jezik HTML5

Označevalni jezik HTML (angl. *hyper text markup language*) je narejen z namenom ustvarjanja spletnih strani in drugih informacij, ki jih lahko prikažemo v spletnih brskalnikih [21]. Namen spletnih brskalnikov je torej branje dokumentov HTML in njihovo pretvarjanje v ljudem razumljivo obliko. Elementi označevalnega jezika HTML so gradniki vseh spletnih strani. Z njegovo pomočjo poleg prikazovanja elementov v spletnih brskalnikih obe-

nem določimo tudi semantični pomen delov dokumenta in strukturo dokumenta.

Označevalni jezik HTML je leta 1990 razvil Tim Berners-Lee. V dolgih letih obstoja jezika se je (do sedaj) zadnja različica pojavila decembra 2012, in nosi ime HTML5. Najnovejša izdaja HTML je postregla z veliko novimi funkcijami, na primer: nove vrednosti atributov, celovita podpora kaskadnim stilskim podlogam inačice 3, boljša podpora za vgradnjo video in zvočnih zapisov, enostavno risanje, podpora 2D in 3D grafiki, lokalne baze podatkov itn. Podrobnosti si lahko bralec pogleda na spletu [18, 19].

Novosti, ki jih prinaša HTML5, so zelo koristne, saj omogočajo hitrejše in bolj enostavo izdelovanje spletnih strani in nadomečajo uporabo knjižnic jezika javascript in vtičnikov, ki so bili potrebni pred prihodom nove inačice jezika HTML [23]. Opozorimo, da nova različica še ne predstavlja standarda za HTML [22]. Posledično vse novosti niso podprte v vseh spletnih brskalnikih, se pa razvijalci le-teh pri izdaji posodobitev trudijo za vključitev vse večjega nabora vsebovanih funkcij.

4.1.6 Kaskadne stilske podloge CSS

Kaskadne stilske podloge oz. CSS (angl. *cascading style sheets*) so podloge, predstavljene s slogovnim jezikom, ki skrbijo za predstavitev spletnih strani. Ključna lastnost uporabe kaskadnih stilskih predlog je v tem, da ločijo strukturo strani – gre za ločitev predstavitve spletne strani od označevalnega jezika HTML in njegove vsebine. S tem pridobimo lažje in hitrejše urejanje spletnih strani, poleg tega pa lahko isti stil uporabimo na več različnih mestih, kar povzroči tudi zmanjšanje dolžine dokumenta HTML [13]. Sintaksa CSS je torej ločena od sintakse HTML, vendar obstaja možnost vgradnje obeh sintaks v isti dokument.

4.2 Strežniki

Strežniki so sestavljeni iz računalnikov oz. programske kode na računalnikih, ki vzdržujejo spletna mesta na internetu. Če želimo objaviti spletno stran ali aplikacijo na spletu, potrebujemo spletni strežnik. Njegova naloga je, da na zahtevo odjemalcev posreduje zahtevane podatke, ki jih hrani.

Spletna aplikacija algoritma LRTA* uporablja štri strežnike in sicer: aplikacijski strežnik Apache Tomcat, spletni strežnik Apache HTTP, podatkovni strežnik MySQL in javni spletni strežnik WebDot. Na aplikacijskem strežniku poganjamo spletno aplikacijo. Lastnosti izdelanih grafov hranimo v podatkovni bazi na podatkovnem strežniku MySQL. Strežnik, ki oskrbuje aplikacijo s slikami grafov je WebDot. Ta za delovanje potrebuje dokumente ".dot", ki se hranijo na strežniku Apache HTTP.

V podpoglavjih bomo opisali prve tri naštete strežnike. Delo s strežnikom WebDot je zelo enostavno (strežniku posredujemo samo lokacijo shranjenega dokumenta), zato o njem ne bomo govorili.

4.2.1 Strežnik Apache Tomcat

Strežnik Apache Tomcat (v nadaljevanju Tomcat) je odprtokodni spletni strežik in vsebovalnik servletov, ki se razvija s strani ASF (angl. *Apache Software Foundation*). V strežnik Tomcat so vgrajeni servleti jave in JSP-ji [11]. S tem omogoča poganjanje čiste javanske kode na spletnih straneh. To pomeni, da je lahko datoteka HTML napisana zgolj v javi, brez uporabe označevalnega jezika HTML, potrebno pa je zagotoviti, da ga izpisuje java.

Programska koda spletne aplikacije algoritma LRTA* je napisana v programskem jeziku java, zato smo se za njeno predstavitev odločili za strežnik, ki vsebuje javanske servlete, hkrati pa je odprtokoden in prosto dostopen.

4.2.2 Strežnik Apache HTTP

Spletni strežnik Apache HTTP (v nadaljevanju Apache), je odprtokodni strežnik, ki so ga začeli razvijati v začetku leta 1995 in od aprila 1996 velja za

najbolj popularen spletni strežnik. Leta 2009 je postal prvi spletni strežnik, na katerem je gostovalo več kot 100 miljonov spletnih strani in aplikacij, junija 2013 pa je bilo ocenjeno, da je 54,2% vseh aktivnih spletnih strani streženih iz strežnika Apache [10].

Strežnik Apache ne vsebuje javanskih servletov in JSP-jev, zato ga v izdelani spletni aplikaciji nismo uporabili kot glavni strežnik. Njegova naloga je zgolj hranjenje dokumentov ".dot", ki morajo biti prosto dostopne na spletu.

4.2.3 Strežnik MySQL

Strežnik MySQL (angl. *My Sequel*) je od julija 2013 svetovno najbolj uporabljen odprtokodni relacijski sistem (za uporavljanje podatkovnih baz), ki deluje kot strežnik in omogoča večuporabniški dostop [28]. Podatkovni strežnik MySQL je bil ustvarjen leta 1995 v podjetju MySQL AB (Švedska).

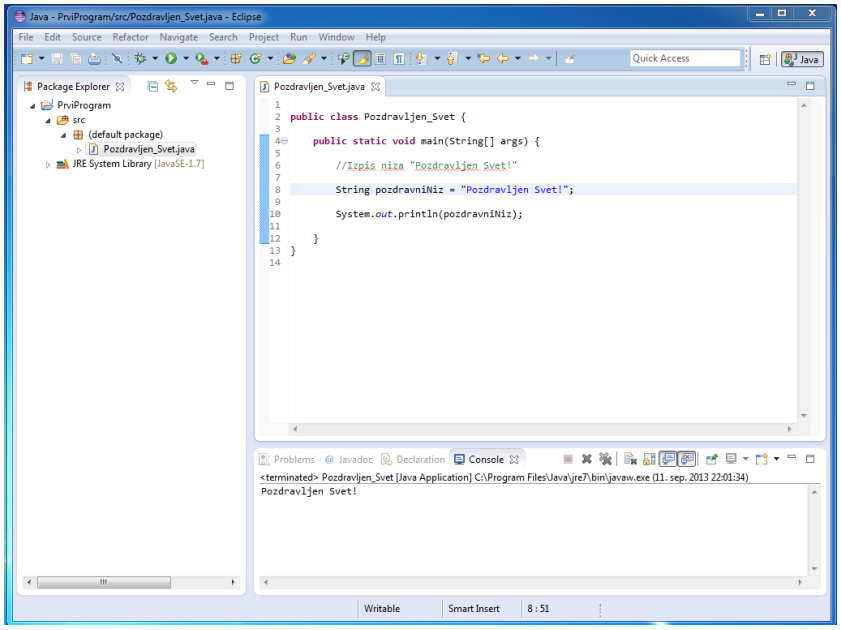
Izdelana spletna aplikacija za potrebe shranjevanja uproablja strežnik MySQL, s katerim komunicira preko jezika SQL. Aplikacija uporablja zastonjsko različico strežnika (community server), obstajajo pa tudi različice za komercialno uporabo, ki so plačljive. Take različice se uporabljajo na obsežnih spletnih straneh kot so Wikipedia, Facebook, Twitter, Flickr, Youtube itd.

4.3 Orodja

Pri izdelavi spletne aplikacije smo uporabili dve orodji; v razvojnem okolju Eclipse je napisana programska koda, za delo s podatkovno bazo pa se uporablja delovno okolje MySQL Query Browser. V podpoglavjih ju bomo na kratko opisali in pogledali njune zaslonske posnetke.

4.3.1 Razvojno okolje Eclipse

Eclipse je odprtokodno razvojno okolje, ki nudi podporo številnim programskim jezikom [18, 17], kot so java, javascript, python, r, ruby, c, c++, haskell, php, perl itn. Programski jezik java je na voljo že ob namestitvi razvojnega okolja, medtem ko za ostale potrebujemo vtičnike (angl. *plug-in*), ki pa jih ni težko pridobiti, saj ima Eclipse vgrajen center za pridobivanje in posodabljanje programske opreme. Prvič je razvojno okolje izšlo leta 2001, navadno pa je vsako leto (meseca junija) izdana nova inačica, ki nosi tudi svoje razvojno ime (zadnja inačica (2013) se imenuje Kepler).



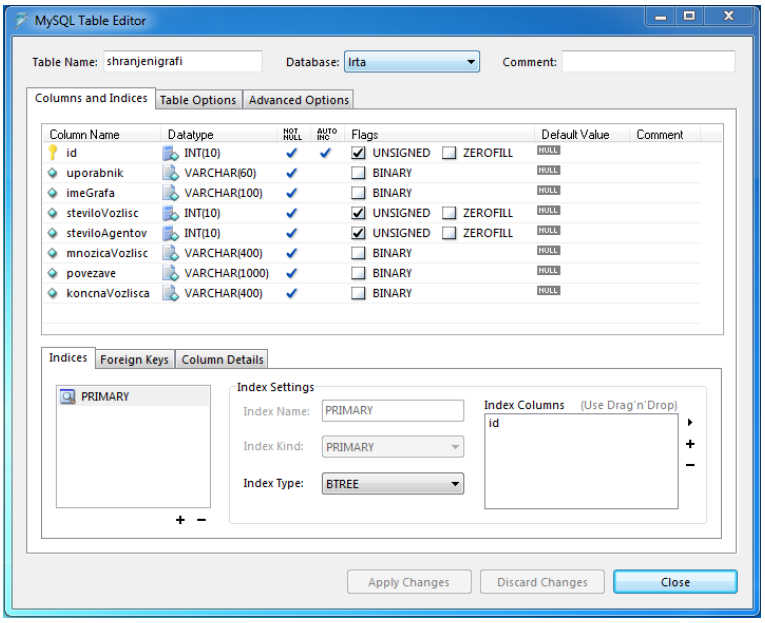
Slika 4.3: Zaslonski posnetek okolja Eclipse.

Na sliki 4.3 je prikazana zaslonski posnetek okolja Eclipse, kjer je v programskem jeziku java napisan program "Pozdravljen svet!". V največjem polju slike se nahaja programska koda, ki je pobarvana glede na rezervirane besede, nize, komentarje itn. Pod kodo je prikazovalno okno (angl. *console*), kjer se nahaja izpis zagnanega programa. V sekciji na levem delu slike so nanizani odprti projekti, ki jih ustvarimo, pripadajoči razredi, paketi, knjižnice

itn. Omenimo, da okolje Eclipse omogoča samodejno dokončanje progamskih stavkov, kar pohitri delo.

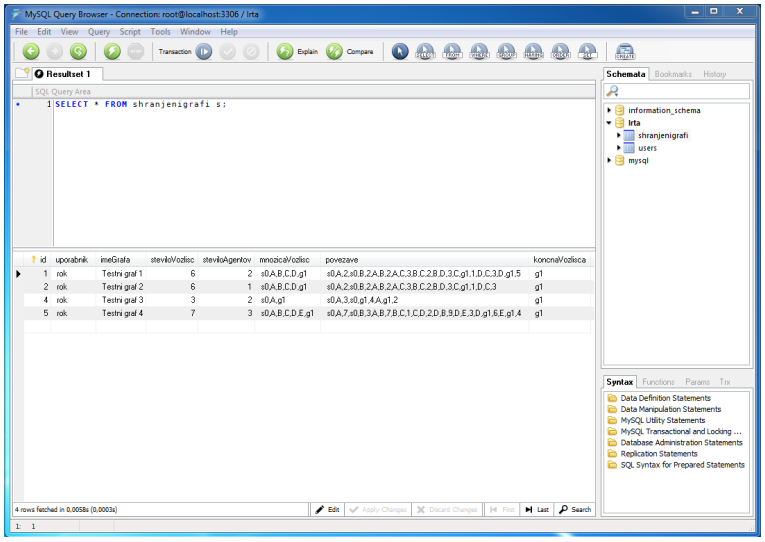
4.3.2 Okolje MySQL Query Browser

Okolje za delo s podatkovno bazo uporabljamo za kreiranje shem in tabel. Podatki so shranjeni v tabelah, ki so razdeljene na vrstice in stolpce. Podateki entitet so shranjeni v svojih vrsticah, stolpci pa predstavljajo atribute, ki jim je moč določiti tipe in privzete vrednosti. Predstavitev lastnosti tabele z imenom "shranjenigrafi", ki jo uporablja spletna aplikacija algoritma LRTA*, so predstavljene na sliki 4.4.



Slika 4.4: Lastnosti tabele spletne aplikacije z imenom"shranjenigrafi".

Zaslonska slika okolja MySQL Query Browser je prikazana na sliki 4.5.



Slika 4.5: Zaslonska slika okolja MySQL Query Browser.

Na zaslonski sliki so vidna štiri polja. Desno zgoraj so vidne uporabljene sheme, ki imajo strukturo drevesa, v katerih se nahajajo shranjene tabele. Kot vidimo, spletna aplikacija uporablja shemo z imenom "lrta" v kateri sta dve tabeli in sicer "shranjenigrafi" in "users". Levo od shematske predstavitve sta dve polji. V spodnjem so podatki, ki so shranjeni v podatkovni bazi (v izbrani tabeli), zgornje pa je interaktivno in omogoča pisanje stavkov (poizvedb in stavke za manipulacijo s podatki) v jeziku SLQ.

Okolje MySQL Query Browser omogoča hiter in jasen pregled podatkov, ki so shranjeni v podatkovni bazi oz. pripadajočih tabelah. Pogosto se uporablja za preverjanje pravilnosti stavkov v jeziku SQL, ki jih nato prenesemo v kodo programskega jezika java in si tako zagotovimo, da je njihovo delovanje pravilno.

Poglavje 5

Struktura in predstavitev spletne aplikacije

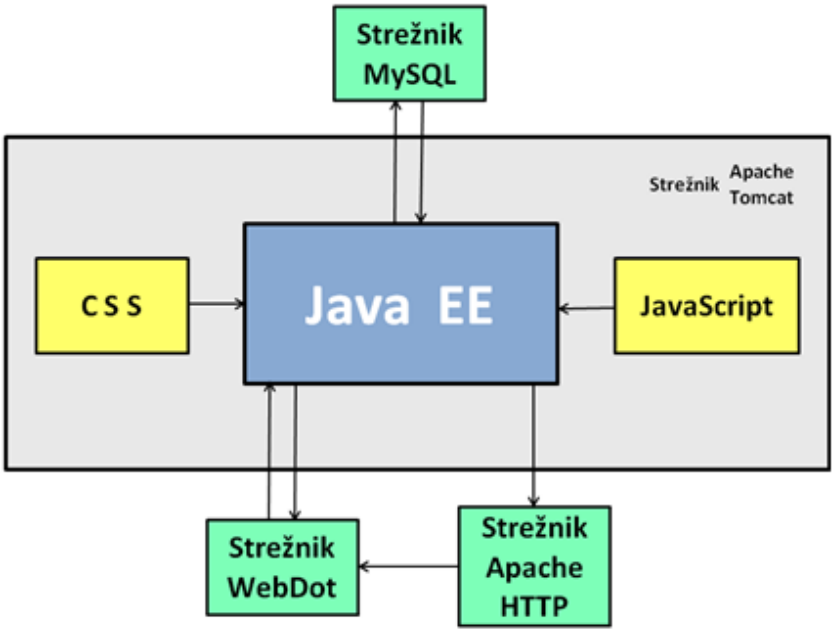
V tem poglavju si bomo pogledali arhitekturno zgradbo aplikacije - naštejemo sestavne razrede napisane v javi, na kratko opišemo njihovo funkcijo in omenimo, katere pomožne datoteke so uporabljene v strukturi spletne aplikacije. Drugi del poglavja je namenjen predstavitvi spletne aplikacije. Prikazali bomo zaslonske posnetke strani in podstrani aplikacije, videli pa bomo tudi, kako je realiziran prikaz postopka reševanja algoritma LRTA*.

5.1 Struktura spletne aplikacije

V 4. poglavju so naštete uporabljene tehnologije in jeziki ter zakaj in kje so uporabljeni, stuktura spletne aplikacije pa ni podana, zato jo predstavljamo tu. V podpoglavjih so predstavljene strukture znotraj posameznih komponent.

Na sliki 5.1 je prikazana struktura programskega dela spletne aplikacije, na kateri so podane njene komponente. Vsa programska koda algoritma in pomožnih razredov je napisana v programskemu jeziku java (na sliki označeno z modro barvo). Tu se poleg kode jave nahajajo tudi servleti in strežniške strani jave (JSP). Puščice iz rumenih pravokotnikov (CSS in JavaScript)

ponazarjajo potrebo po njihovi vključitvi v spletno aplikacijo, saj so potrebni za oblikovanje spletne strani in validacije. Vse omenjene komponente se nahajajo znotraj aplikacijskega spletnega strežnika Apache Tomcat (slika 5.1).



Slika 5.1: Struktura programskega dela spletne aplikacije.

Z uporabo razredov jave gradimo poizvedbe v jeziku SQL, te pa se prenašajo do podatkovnega strežnika MySQL. Ta javi (oziroma strežniku Tomcat) vrača rezultate poizvedb, če pa prejme stavke za manipulacijo s podatki iz baze, pa le te spremeni, doda ali odstrani.

Na sliki sta prikazana tudi strežnika WebDot in Apache HTTP, ki se uporabljata za izrisovanje grafov, ki so opisani (definirani) v opisnem jeziku DOT in shranjeni v datotekah končnice ".dot". Izvajanje deluje tako, da programska koda jave na vsakem koraku iteracije ustvari potreben dokument DOT in ga shrani na strežnik Apache HTTP. Nato aplikacijski strežnik Tomcat pošlje zahtevek strežniku WebDot (k zahtevku se doda spletno mesto, kjer

se nahaja dokument opisanega grafa - torej naslov strežnika Apache), ta preveri, ali datoteka obstaja in nato zgradi potreben graf, ki je opisan v datoteki iz prejetega naslova. Zgenerirana slika se posreduje strežniku Tomcat, ki pa jo prikaže v brskalniku s pomočjo skripletov.

5.1.1 Struktura programske kode jave

Programska koda java je razdeljena v programske pakete (angl. *package*), ki so med seboj povezani. Imamo štiri pakete v katerih se nahajajo razredi, ki predstavljajo servlete, ali pa so sestavljeni iz običajne programske kode značilne za javo SE. V tabeli 5.1 so podani vsi razredi jave, ki niso servleti.

Ime paketa	Algoritem	DataBase	Tree
Vsebovani razredi	LRTA	BazaQuery	PrintTree

Tabela 5.1: Prikaz paketov in vsebovanih razredov.

V vsakem izmed treh paketov je en razred. V paketu Algoritem se nahaja datoteka "LRTA.java", ki vsebuje spremenljivke in metode za delo z algoritmom LRTA*. Paketa Algoritem in Tree sta prepletena z metodami, saj razred PrintTree skrbi za ustvarjanje dokumentov ".dot", za kar potrebuje podatke o grafu (število vozlišč, agentov, množica povezav, število iteracij itn), ki se nahajajo v razredu LRTA. Najmanj uporabljen razred, BazaQuery, se nahaja v paketu DataBase in vsebuje programsko kodo za povezovanje na podatkovni strežnik, stavke poizvedb in stavke za manipulacijo s podatki.

Četrti paket, ki predstavlja servlete, se imenuje Servleti in vsebuje naslednje razrede:

- CreateNewGraphServlet - servlet za določanje parametrov novo ustvarjenega grafa. Komunicira z razredom LRTA, kjer se posodabljajo informacije o številu vozlišč in povezavah.
- DataBaseHandlerServlet - servlet se uporablja za pridobivanje podatkov o ustvarjenih grafih iz podatkovne baze. Povezan je z metodami

razreda BazaQuery, kjer so vsebovane ustrezne poizvedbe.

- LoadGraphHandlerServlet - servlet se uporabi, kadar želimo iz baze podatkov naložiti predhodno shranjene grafe in za spreminjanje ter shranjevanje le teh. Servletu so na voljo metode razredov LRTA in BazaQuery.

- RunGraphHandlerServlet - servlet se izvaja, kadar v spletni aplikaciji pregledujemo učni proces algoritma LRTA. Skrbi za ustrezno spreminjanje spletne vsebine, s pomočjo skripletov pa prikazuje delovanje korak za korakom.

K strukturi programske kode java se uvrščajo tudi strežniške strani jave ali JSP (angl. *Java Server Pages*), ki služijo prikazu informacij uporabniku spletne aplikacije. Naštejemo in opišemo jih v abecednem vrstnem redu:

- contact - stran vsebuje podatke o avtorju in aplikaciji ter elektronski naslov avtorja.

- createNewGraph - spletna stran, na kateri se ustvarjajo novi grafi. Vsebuje možnosti, kot so dodajanje določenega števila vozlišč in agentov, dodajanje, spreminjanje in brisanje povezav ter shranjevanje grafa v podatkovno bazo.

- howToUseAlgorithm - predstavitev delovanja algoritma. Vsebina strani je pridobljena iz izdelanega diplomskega dela.

- howToUseWebApp - opis uporabe spletne aplikacije (ustvarjanje novih grafov, shranjevanje, nalaganje ustvarjenih grafov ter predstavitev strani “runGraph.jsp”).

- index - pozdravna stran spletne aplikacije.

- loadGraph - stran, kjer so na voljo predogledi shranjenih grafov. Stran nam ponuja tudi možnosti sprememb povezav med vozlišči (dodajanje, odstranjevanje, spreminjanje) ter shranjevanje grafa v podatkovno bazo.

- menuTop - stran, ustvarjena z namenom, da se isti odseki programske kode ne ponavljajo. Gre za meni na vrhu spletne strani, ki ga vključimo v vsako stran aplikacije.

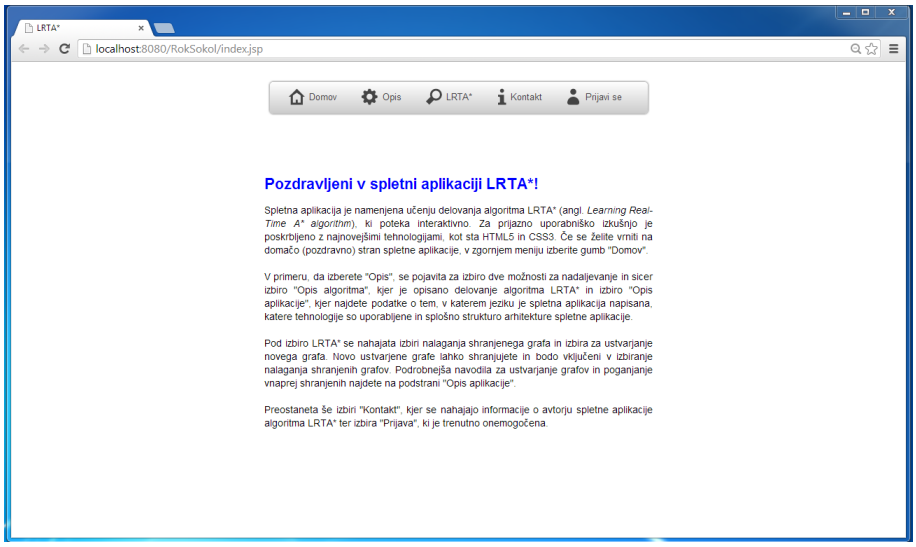
- runGraph - interaktivno izvajanje algoritma LRTA*, kjer se poleg slik grafov izpisuje tudi razlaga koraka, prikazanega na pripadajoči sliki.

V straneh JSP so povezave do dokumentov, ki so shranjeni v datoteki css (nahajajo se znotraj datoteke z dokumenti JSP). Dokumenti v tej mapi pripomorejo k izgledu spletne strani, saj vsebujejo kodo CSS3. Definirajo izbirno vrstico (meni spletne aplikacije) gumbe, barve, postavitve strani itn.

5.2 Predstavitev spletne aplikacije

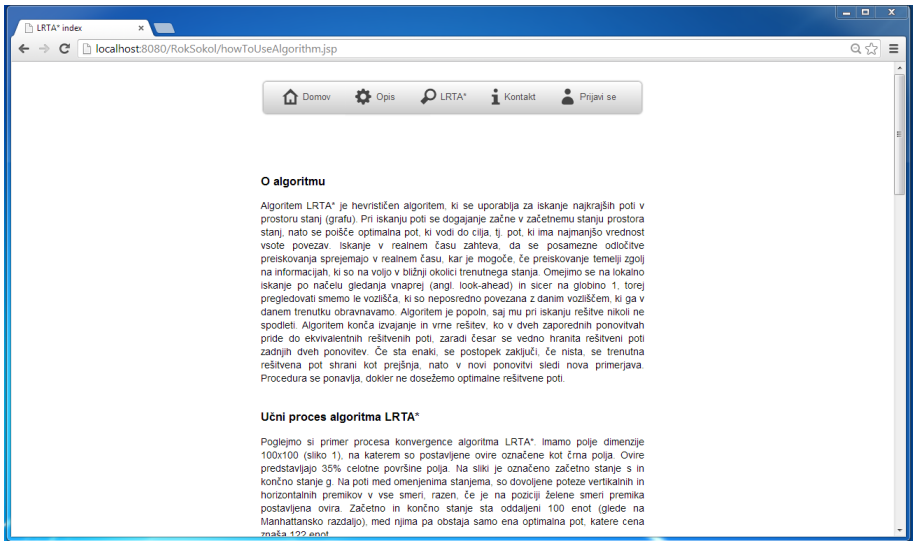
Na slikah si bomo ogledali delovanje aplikacije v spletnem brskalniku Chrome. Opišemo elemente vsake strani, kaj je na slikah prikazano ter navedemo funkcije gumbov. Omenimo, da HTML5 in CSS3 še nista standarda in zato vsi brskalniki ne podpirajo vseh uporabljenih elementov. Največ težav ima pri prikazovanju Internet Explorer, ki ne uspe stolmačiti menijske vrstice spletne aplikacije. Ostali brskalniki (Google Chrome, Mozilla FireFox, Opera, Safari, Maxthon) s prikazovanjem spletne vsebine nimajo težav.

Na sliki 5.2 je prikazana začetna stran spletne aplikacije. Na vrhu je meni, ki vsebuje direktne povezave do strani Domov in Kontakt, preko spustnega menija (angl. *drop-down menu*) se pod opisom pojavita strani Opis algoritma in Opis aplikacije. Spustni meni se prikaže še pod elementom LRTA*, ki vsebuje strani Ustvari nov graf in Naloži ustvarjen graf.



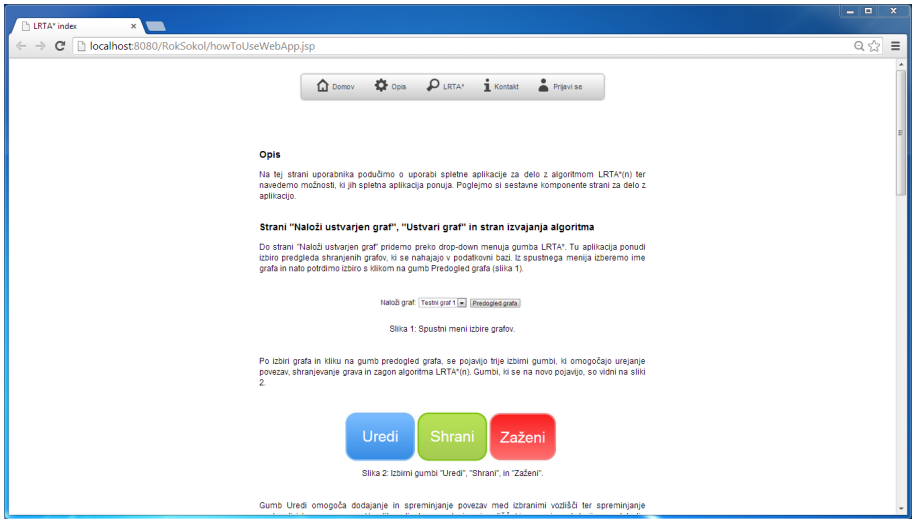
Slika 5.2: Začetna (indeksna) stran spletne aplikacije.

Na začetni strani je besedilo z vsebino kratke predstavitve aplikacije. V besedilu je omenjen algoritem LRTA*, tehnologije, s katerimi je aplikacija izdelana ter opis povezav, ki se nahajajo v meniju.



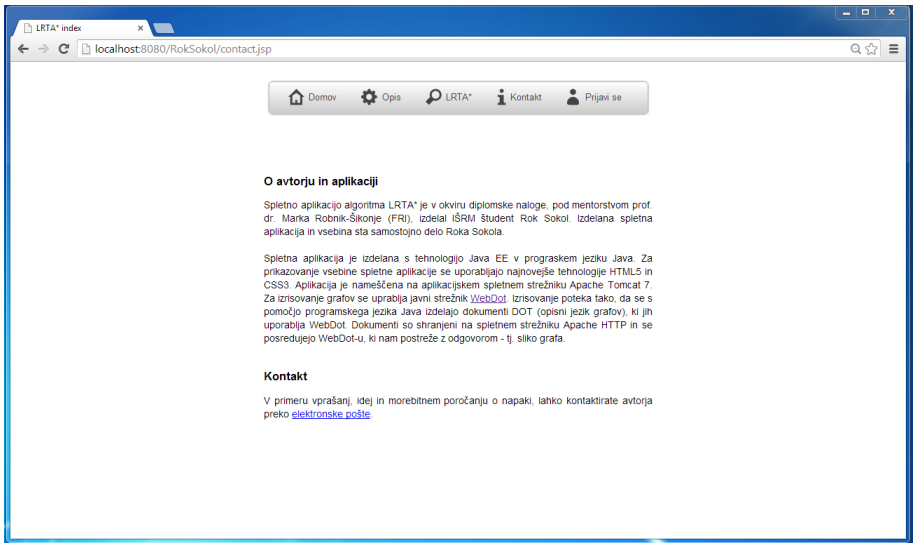
Slika 5.3: Stran “Opis algitma”.

Stran z opisom algoritma je prikazana na sliki 5.3. Na njej najdemo besedilo s podrobnim opisom algoritma, učnega procesa in inteligentnih agentov. Vsebina te strani je povzeta po tretjem poglavju te diplomske naloge.



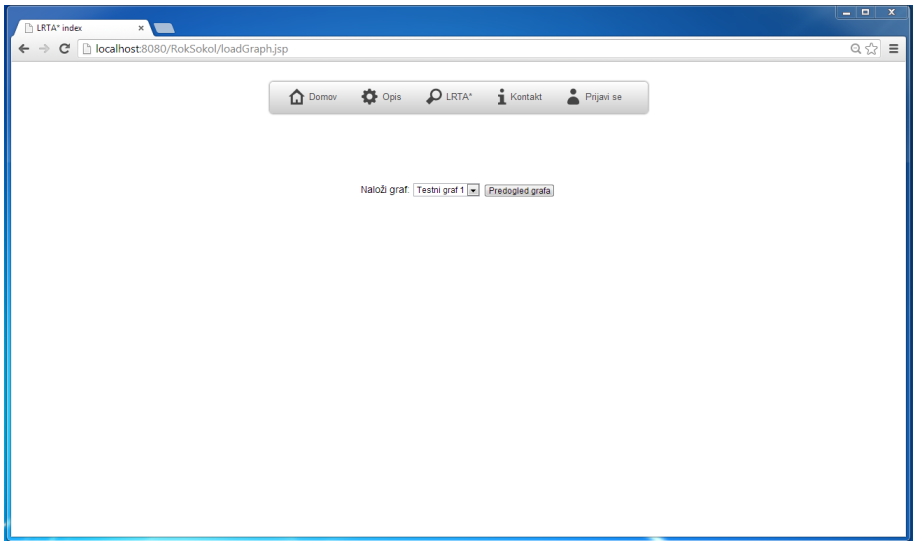
Slika 5.4: Stran “Opis aplikacije”.

Stran z navodili za uporabo spletne aplikacije je podana na sliki 5.4. Vsebuje opise postopkov za gradnjo novega drevesa, nalaganje že obstoječega in upravljanje aplikacije tekom učnega procesa, ki ga zagotavlja algoritem.



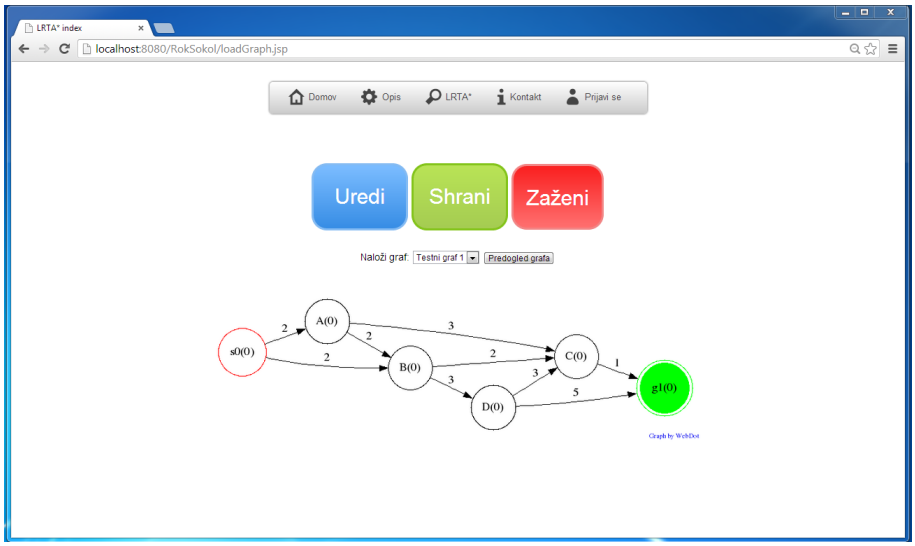
Slika 5.5: Stran “Kontakt”.

Kontaktna stran (slika 5.5) vsebuje podatke o avtorju spletne aplikacije. Omenjena je groba struktura spletne aplikacije, za morebitna vprašanja uporabnikov pa je priložena povezava elektronskega naslova avtorja.



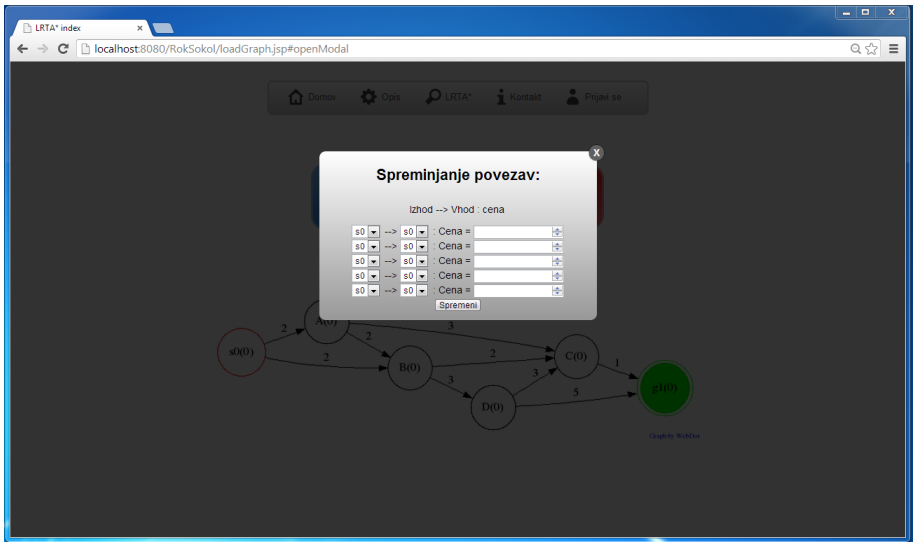
Slika 5.6: Stran “Naloži ustvarjen graf”.

Stran "Naloži ustvarjen graf"je prikazana na sliki 5.6 in se med izbirami spreminja. Aplikacija nam ponudi izbiro predogleda shranjenih grafov, ki se nahajajo v podatkovni bazi. Iz spustnega menija izberemo ime grafa in nato potrdimo izbiro s klikom na gumb Predogled grafa. Podatki, ki so se ob prihodu na stran prenesli iz podatkovne baze, se preko servleta pošljejo razredu LRTA, ki obdela podatke grafa in jih posreduje razredu PrintTree, ki ustvari dokument opisnega jezika DOT in z njim iz strežnika WebDot pridobi sliko. Rezultat tega procesa je viden na sliki 5.7.



Slika 5.7: Stran "Naloži ustvarjen graf"po izbiri grafa.

Ob izbiri grafa se na spletni strani pojavi slika izbranega grafa ter trije gumbi, ki so oblikovani s pomočjo CSS3. V primeru klika na gumb Uredi, se nad obstoječo stranjo odpre novo okno. Prikazano je na sliki 5.8.



Slika 5.8: Stran "Naloži ustvarjen graf"po kliku na gumb Uredi.

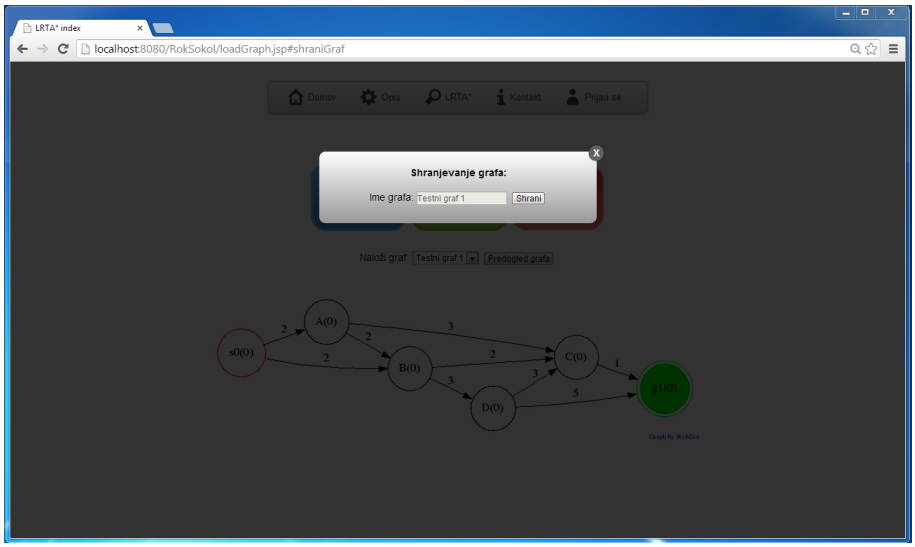
Gumb Uredi omogoča dodajanje in spreminjanje povezav med izbranimi vozlišči ter spreminjanje vrednosti izbranih povezav. V petih vrsticah so podani pari vozlišč, kjer prvo izmed dvojice predstavlja izhodno vozlišče, drugo pa vhodno vozlišče. Potrebno je vnesti vrednost povezave (tj. cena), ki pa ne sme biti negativno število. Z validatorjem regularnih izrazov se preverja, da je pogoj pozitivne vrednosti izpolnjen, hkrati pa onemogoča vpisovanje nizov.

Če želimo povezavo med dvemi vozlišči izbrisati, v oknu Spreminjanje povezav izberemo pripadajoči vozlišči povezave in določimo vrednost cene na 0. Dopustno je tudi, da izberemo samo par vozlišč, polja s ceno pa ne izpolnimo - rešitvi data isti rezultat. Po kliku na potrditveni gumb Spremeni, se slika grafa osveži.

O oknu spreminjanja povezav povejmo še, da ni potrebna izpolnitev vseh petih parov vozlišč in cen. Več jih je na voljo z razlogom hitrejšega hkratnega

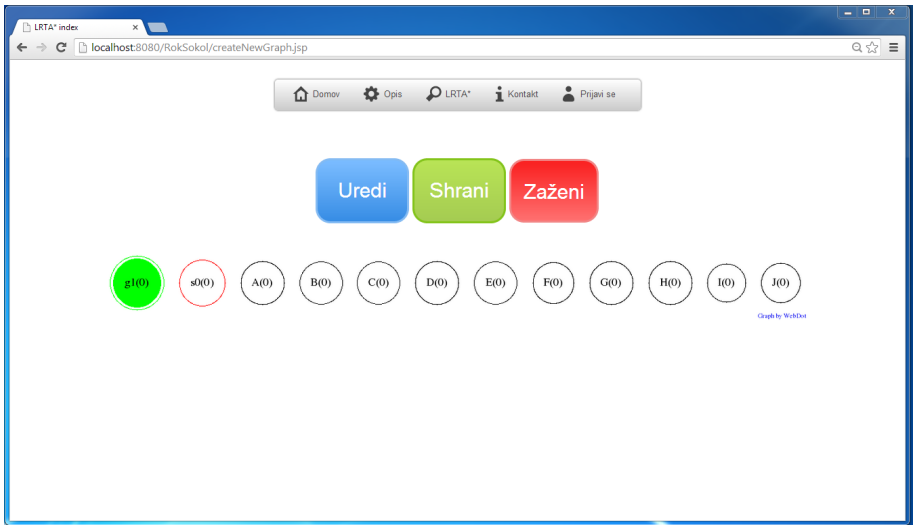
dodajanja večih povezav. Pomembno je tudi, da algoritem LRTA* deluje na enostavnih grafih (brez zank in ciklov dolžin 2), kar upošteva tudi programska koda, ki se nahaja za vmesnikom spreminjanja povezav.

Gumb Shrani (slika 5.6) omogoča shranjevanje spremenjenega grafa v podatkovno bazo. Ob kliku nanj se odpre okno, ki je prikazano na sliki 5.9.



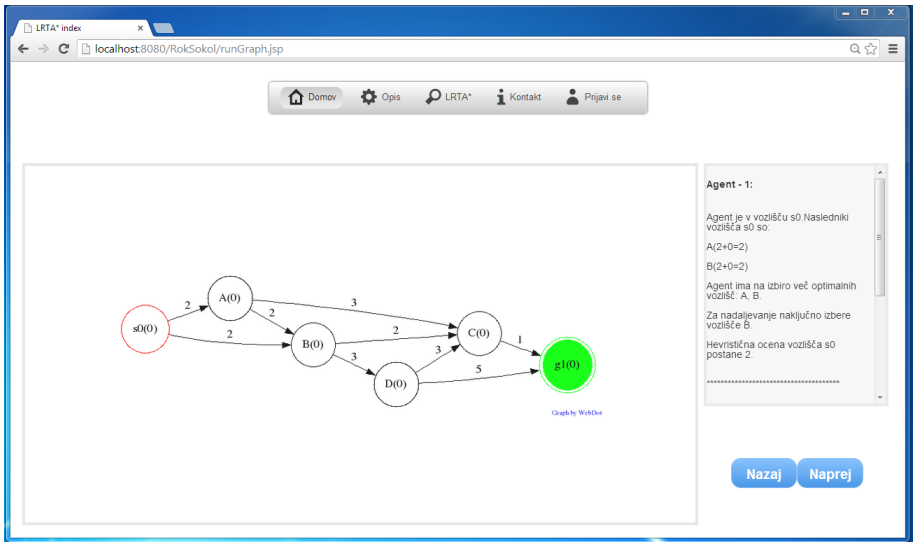
Slika 5.9: Stran "Naloži ustvarjen graf"po kliku na gumb Shrani.

Izbira je na strani "Naloži ustvarjen graf"podobna kot na strani "Ustvari nov graf". Po prihodu na stran najprej izberemo število vozlišč in število agentov. S klikom na gumb kreiraj dobimo stanje, ki je prikazano na sliki 5.10.



Slika 5.10: Stran "Ustvari graf".

Na sliki 5.10 je vidno, da smo izbrali dvanajst vozlišč, ki pa so nepovezana. Povezave dodajamo enako, kot je opisano pri sliki 5.8. Graf lahko po končanem urejanju shranimo, lahko pa na njem poženemo izvajanje algoritma LRTA*(n), kar storimo s klikom na gumb Zaženi.



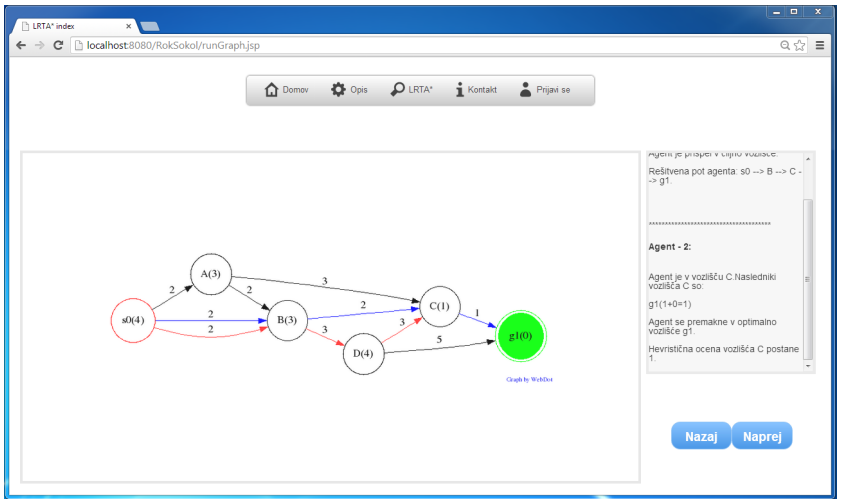
Slika 5.11: Izvajanje algoritma LRTA*(n) v spletni aplikaciji.

Prehod v stanje na sliki 5.11 je možen na dva načina in sicer s klikom na gumb Zaženi na straneh "Naloži ustvarjen graf"in "Ustvari graf". Stran je sestavljena iz treh delov in sicer sredinske slike, opisa koraka in kontrolnih gumbov.

Na sliki so v vozliščih ob imenu zapisane hevristične ocene stanj. Na začetku izvajanja so po pravilu delovanja algoritma LRTA* vedno enake 0, skozi delovanje algoritma pa se spreminjajo. Ob povezavah so napisane tudi vrednosti oz. cene poti. Začetno vozlišče je označeno z rdečo barvo, končno pa je obarvano z zeleno.

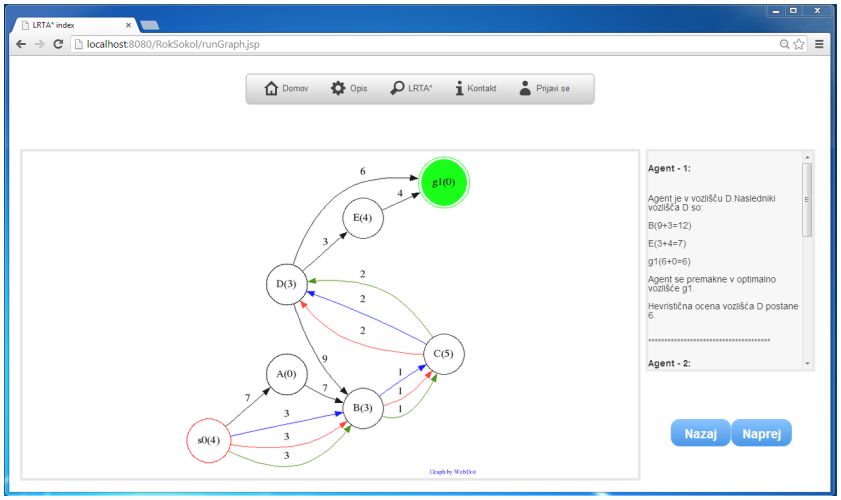
Desno od slike se nahaja vsebnik z drsnikom, ki vsebuje opis trenutnega koraka - tistega, ki je prikazan na sliki. Opis koraka vsebuje informacije, kot so imena agentov (nanizana zaporedno in odebeljena), trenutno stanje (vozlišče) vsakega agenta, nasledniki trenutnega stanja, hevristične ocene in cene povezav vsakega naslednega stanja ter stanje, v katerega se agent premakne. Kadar je agent na razcepu med več enakimi stanji (tj. enake vsote hevrističnih ocen naslednikov in cene povezav do naslednikov), je podana tudi ta informacija skupaj z opisom agentove naključne izbire stanja. Na vsakem koraku je dana tudi informacija o spremembi hevristične ocene stanja po prehodu agenta v drugo stanje. Ko algoritem konča izvajanje, je to opisano v vsebniku razlag, podana pa je tudi rešitev - najkrajša pot od začetnega stanja s0 do končnega stanja g.

Pod razlago koraka se nahajata gumba Nazaj in Naprej. Omogočata prehod med koraki tako, da enostavno pregledujemo prejšnje in naslednje korake oz. iteracije izvajanja. Na prvem koraku je klik na gumb Nazaj onemogočen, po končanem izvajanju pa se onemogoči klik na gumb Naprej.



Slika 5.12: Izvajanje algoritma LRTA*(2) v spletni aplikaciji.

Na sliki 5.12 je prikazano vzporedno delovanje dveh agentov. Vsak je označen s svojo barvo. Zaradi preglednosti je število agentov omejeno na največ 5.



Slika 5.13: Izvajanje algoritma LRTA*(3) v spletni aplikaciji.

Na sliki 5.13 je prikazano še vzporedno delovanje treh agentov.

Poglavje 6

Zaključek

V diplomski nalogi smo se ukvarjali s preiskovalnimi algoritmi. Na začetku smo jih razdelili na informirane in neinformirane. Našteli smo kriterije za merjenje učinkovitosti preiskovalnih algoritmov ter definirali pojme, ki so skupni vsem preiskovalnim algoritmom. V nadaljevanju smo si ogledali osnovne neinformirane algoritme, za vsakega smo podali informacijo o časovni in prostorski zahtevnosti. S slikovnim gradivom smo prikazali njihovo delovanje ter našteli dobre in slabe lastnosti. Pogledali smo informirane preiskovalne algoritme, ki med preiskovanjem uporabljajo hevristične funkcije, ter definirali pojme dopustnosti, popolnosti in optimalnosti. S tem smo si zagotovili boljše razumevanje osrednje teme diplomskega dela - delovanje asinhronega preiskovalnega algoritma LRTA*.

Zapisali smo psevdokodo algoritma LRTA* ter opisali njegovo delovanje. Govorili smo o inteligentnih agentih, ki jih za preiskovanje uporablja algoritem. Podrobno smo pogledali delovanje algoritma na dveh primerih. V enem primeru je deloval en agent, v drugem primeru pa smo prikazali vzporedno delovanje dveh agentov. Primera sta bila slikovno opremljena. S tem je bil potek delovanja jasen na vsakem koraku algoritma.

V diplomskem delu smo našteli dve orodji, tri strežnike in šest jezikov, ki so bili potrebni za izdelavo spletne aplikacije. Za vsako uporabljeno tehnologijo smo na kratko opisali njeno zgodovino in navedli, kje in zakaj smo jo

uporabili.

V zadnjem poglavju smo podrobno opisali strukturo izdelane spletne aplikacije ter opisali, kako so komponente povezane in kakšna so funkcije posameznih komponent. Ogledali smo si tudi dvanajst zaslonskih posnetkov spletne aplikacije in razložili njeno delovanje.

Cilj diplomskega dela je bilo predstaviti preiskovalne algoritme in se osredotočiti na asinhroni preiskovalni algoritem LRTA*(n). Stranski produkt je bila izdelava spletne aplikacije v najnovejših tehnologijah, ki uporabnike poučuje o delovanju algoritma LRTA* na interaktiven način. Aplikacija nudi možnosti ustvarjanja in shranjevanja novih grafov.

Motivacija za obdelavo preiskovanja prostora s pomočjo algoritma LRTA* je splošnost in uporabnost algoritma ter dejstvo, da kljub temu ne obstaja interaktivna spletna aplikacija, ki bi učila njegovo delovanje. Poleg tega je o algoritmu, ki smo ga vzeli pod drobnogled, na svetovnem svetu zelo malo informacij v primerjavi z ostalimi preiskovalnimi algoritmi.

Menim, da je bil cilj dosežen in aplikacija uspešno realizirana. Za nadaljnje delo bi priporočili izdelavo spletne aplikacije v več svetovnih jezikih. Koristno bi bilo, če bi v spletno aplikacijo vključili še druge preiskovalne algoritme in bi tako uporabnikom ponudili možnost primerjave različnih algoritmov na istih primerih na enem spletnem mestu.

Literatura

[1] I. Bratko, M. Možina, J. Žabkar, *"Prostor stanj in osnovni algoritmi preiskovanja"*, Ljubljana, 2013. Dostopno na: https://ucilnica.fri.uni-lj.si/mod/resource/view.php?id=11519 (dostop: 11. 9. 2013)

[2] D. Goodman, *"JavaScript Bible, Gold Edition"*, Hungry Minds, Inc, New York, NY, 2001.

[3] I. Kononenko, M. Robnik-Šikonja, *"Inteligentni sistemi"*, Založba FE in FRI, Ljubljana, 2010.

[4] R. E. Korf, *"Real-time heuristic search"*, Elsevier, 1990.

[5] U. Mesojedec, B. Fabjan, *"Java2 - temelji programiranja"*, Založba Pasadena, Ljubljana, 2004.

[6] P. Mrhar, *"Java 2 - prvi korak"*, Flamingo Založba, Nova Gorica, 2002.

[7] R. W. Sebesta, *"Programming the World Wide Web"*, Pearson, Boston, 2011.

[8] M. Simbo, *"Controlling the learning process of real-time heuristic search"*, Elsevier, 2002.

[9] A* search algorithm. Dostopno na: http://en.wikipedia.org/wiki/A*_search_algorithm (dostop: 11. 9. 2013)

[10] Apache HTTP Server. Dostopno na:
http://en.wikipedia.org/wiki/Apache_HTTP_Server
(dostop: 11. 9. 2013)

[11] Apache Tomcat. Dostopno na:
http://en.wikipedia.org/wiki/Apache_Tomcat
(dostop: 11. 9. 2013)

[12] Breadth-first search. Dostopno na:
http://en.wikipedia.org/wiki/Breadth-first_search
(dostop: 11. 9. 2013)

[13] Cascading Style Sheets. Dostopno na:
http://en.wikipedia.org/wiki/Cascading_Style_Sheets
(dostop: 11. 9. 2013)

[14] Depth-first search. Dostopno na:
http://en.wikipedia.org/wiki/Depth-first_search
(dostop: 11. 9. 2013)

[15] DOT language. Dostopno na:
http://en.wikipedia.org/wiki/DOT_language
(dostop: 11. 9. 2013)

[16] Dynamic programming. Dostopno na:
http://en.wikipedia.org/wiki/Dynamic_programming
(dostop: 11. 9. 2013)

[17] Eclipse. Dostopno na:
http://www.eclipse.org/org/
(dostop: 11. 9. 2013)

[18] Eclipse. Dostopno na:
http://en.wikipedia.org/wiki/Eclipse_(software)
(dostop: 11. 9. 2013)

[19] Greedy algorithm. Dostopno na:
http://en.wikipedia.org/wiki/Greedy_algorithm
(dostop: 11. 9. 2013)

[20] Heuristic (computer science). Dostopno na:
http://en.wikipedia.org/wiki/Heuristic_(computer_science)
(dostop: 11. 9. 2013)

[21] HTML. Dostopno na:
http://en.wikipedia.org/wiki/HTML
(dostop: 11. 9. 2013)

[22] HTML5 - new elements. Dostopno na:
http://www.w3schools.com/html/html5_new_elements.asp
(dostop: 11. 9. 2013)

[23] HTML5. Dostopno na:
http://en.wikipedia.org/wiki/HTML5
(dostop: 11. 9. 2013)

[24] (2013) IDA*. Dostopno na:
http://en.wikipedia.org/wiki/IDA*
(dostop: 11. 9. 2013)

[25] Iterative deepening depth-first search. Dostopno na:
http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
(dostop: 11. 9. 2013)

[26] Java (programming language). Dostopno na:
http://en.wikipedia.org/wiki/Java_(programming_language)
(dostop: 11. 9. 2013)

[27] JavaScript. Dostopno na:
http://en.wikipedia.org/wiki/JavaScript
(dostop: 11. 9. 2013)

[28] MySQL. Dostopno na:
hhttp://en.wikipedia.org/wiki/MySQL
(dostop: 11. 9. 2013)

[29] Public server WebDot. Dostopno na:
http://api.graphviz.org/webdot/
(dostop: 11. 9. 2013)

[30] SQL. Dostopno na:
http://en.wikipedia.org/wiki/SQL
(dostop: 11. 9. 2013)