Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matjaž Hegedič

Vizualizacija delovanja preiskovalnih algoritmov v umetni inteligenci

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: akad. prof. dr. Ivan Bratko

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: 00031/2013
Datum: 09.04.2013

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

Kandidat: **MATJAŽ HEGEDIČ**

Naslov: **VIZUALIZACIJA DELOVANJA PREISKOVALNIH ALGORITMOV V UMETNI INTELIGENCI**

VISUALITION OF THE EXECUTION OF SEARCH ALGORITHMS IN ARTIFICIAL INTELLIGENCE

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Algoritmi preiskovanja prostora stanj sodijo med osnovna orodja umetne inteligence in jih zato tudi obravnavamo med osnovnimi temami pri poučevanju umetne inteligence. Grafični prikaz z animacijo delovanja teh algoritmov močno olajša njihovo razlago in razumevanje. V tem diplomskem delu implementirajte izbrane preiskovalne algoritme ter izdelajte program za vizualizacijo njihovega delovanja za potrebe poučevanja. Implementacija naj vključuje naslednje algoritme: iskanje v globino, v širino, iterativno poglabljanje, A*, IDA* in RBFS. Pri tem oblikujte čim bolj nazorne grafične ponazoritve, primerne za (1) sledenje izvajanju posameznih korakov algoritmov ter (2) globalno ilustracijo učinkovitosti preiskovanja z raznimi hevrističnimi ocenami.

Mentor:

prof. dr. Ivan Bratko

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

za

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matjaž Hegedič, z vpisno številko **63100230**, sem avtor diplomskega dela z naslovom:

Vizualizacija delovanja preiskovalnih algoritmov v umetni inteligenci

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom akad. prof. dr. Ivana Bratka,
- 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 10. septembra 2013 Podpis avtorja:

Kazalo

Povzetek

Preiskovalni algoritmi so eno izmed ključnih orodij za reševanje problemov v umetni inteligenci pa tudi drugih področjih računalništva. Naučiti se in intuitivno razumeti njihovo delovanje pa je lahko težavno, še posebej za začetnike. V okviru tega diplomskega dela je bila razvita spletna aplikacija, ki vizualizira delovanje nekaterih najpomembnejših preiskovalnih algoritmov. V besedilu je najprej opisan koncept reševanja problemov s preiskovanjem prostorov stanj, sledijo predstavitev aplikacije in navodila za uporabo. Nadaljni dve poglavji bolj podrobno razložita delovanje vsakega izmed obravnavanih algoritmov in lastnosti dveh implementiranih prostorov stanj - mreže in igre osmih kvadratov. Sedmo poglavje opiše uporabljene tehnologije, arhitekturo aplikacije in sam postopek razvoja. Zaključno poglavje poda spoznanja o nekaterih problemih, ki so se tekom diplomske naloge pojavili in navede nekaj idej za nadaljne razširitve in nadgradnje aplikacije. Vključena sta dva dodatka, prvi vsebuje dokumentacijo kode, drugi pa navodila za pisanje novih modulov za aplikacijo.

Abstract

Search algorithms are one of the key tools for solving problems in Artificial Intelligence as well as other fields in Computer Science. Learning and intuitively understanding these algorithms can be hard, especially for beginners. As a part of this BSc thesis, a web application which visualizes some of the most important search algorithms, was developed. The text begins with a description of the concept of solving problems using state-space search, followed by an introduction to the application and instructions for its use. Subsequent two chapters explain the workings of each of included algorithms as well as properties of both implemented state-spaces - grid and 8-puzzle. The seventh chapter describes technologies used, the application architecture and the development process itself. The final chapter addresses some of the problems encountered during the making of the thesis and suggests some ideas for extensions and upgrades for the application. Two appendices are included, the first covering code documentation and the second instructions for writing new application modules.

Poglavje 1

Uvod

V mnogih panogah računalništva, še posebej pa na področju umetne inteligence se pogosto soočamo s problemi, ki se jih da reševati zgolj s sistematičnim preizkušanjem in vrednotenjem različnih alternativnih možnosti. Takšnemu pristopu k reševanjo pravimo preiskovanje, postopke in načine, na katere ga opravljamo, pa imenujemo preiskovalni algoritmi.

Čeprav je preiskovanje kot orodje za reševanje problemov najpogostejše uporabljano prav na področju umetne inteligence, pa je zaradi svoje vsestranskosti pomembno tudi za računalničarje na drugih področjih. Tako so tekom svojega študija z njim seznanjeni in ga morajo znati ter tudi razumeti malodane vsi študentje računalništva.

Med drugim tudi iz lastnih izkušenj vem, da je učenje in razumevanje algoritmov naporno, sploh če se z njimi seznanjamo prvič in če njihovo delovanje ni najbolj intuitivno. Literature na področju preiskovalnih algoritmov obstaja precej in je večinoma enostavno dostopna tudi preko spleta, a če si želimo, da so naše učenje, razumevanje in pomnenje delovanja teh algoritmov učinkoviti, za večino ljudi zgolj formalni opisi ne zadostujejo. Zaradi tega opise delovanja v člankih in učbenikih pogosto spremljajo tudi ilustracije, na predavanjih pa predavatelji razlago običajno popestrijo z ilustracijo vsaj nekaj korakov predstavljenih preiskovalnih algoritmov. A ko pridemo do orodij, s katerimi bi si lahko delovanje posameznih algoritmov vizualizirali

in animirali, je izbira precej bolj skopa. Velikokrat smo omejeni na animacije, kjer je prikazano reševanje enega samega problema na en sam način. Uporabnik tako ne more videti, kako bi se reševanje razlikovalo na malenkost drugačnem problemu ali morda z drugačnim algoritmom. Obstajajo nekateri programi za vizualizacijo, ki so pisani za točno določeno platformo in posledično trpijo za nezdružljivostjo in težko dostopnostjo uporabnikom. Za marsikatere preiskovalne algoritme pa zaradi premajhnega povpraševanja vizualizacije delovanja enostavno ne obstajajo.

Zaradi tega sem se v sklopu svoje diplomske naloge odločil napisati aplikacijo za vizualizacijo delovanja preiskovalnih algoritmov v umetni inteligenci, ki na dveh različnih vrstah problemov animira delovanje naslednjih algoritmov: iskanje v globino, iskanje v širino, iterativno poglabljanje, A*, IDA*, RBFS in RTA*.

Za aplikacijo sem si zastavil naslednje cilje: da je uporabna in ne sama sebi namen, da je zelo nazorna, da je uporabniku prijazna in enostavno dostopna ter da je karseda enostavno nadgradljiva (v kolikor bo v prihodnosti to potrebno). Odločitvi za takšno diplomo pa so botrovali naslednji osebni razlogi: želel sem si jo opravljati na računalniškem področju, ki me najbolj zanima (se pravi, umetni inteligenci), se seznaniti z načrtovanjem in izgradnjo arhitekture večjih programov oziroma aplikacij in se hkrati naučiti tudi novega programskega jezika oziroma novih tehnologij, ki jih tekom študija še nisem spoznal.

V naslednjih poglavjih bom opisal koncept reševanja problemov s preiskovanjem prostorov stanj, navedel nekaj že obstoječih rešitev in jih na kratko opisal, predstavil svoj program iz uporabniškega vidika in podal navodila za uporabo, se natančneje poglobil v implementirane preiskovalne algoritme in implementirane prostore stanj, natančneje opisal zgradbo in arhitekturo programa, zapisal krajšo dokumentacijo kode, podal napotke za nadaljno nadgrajevanje programa in v zadnjem poglavju navedel nekaj svojih idej, kako bi se dalo program še dodatno izboljšati.

Poglavje 2

Reševanje problemov s preiskovanjem

Preiskovanje je pristop, s katerim lahko rešujemo probleme, za katere znamo določiti prostor stanj. V splošnem je prostor stanj določen z množico stanj (torej enolično določenih konfiguracij, v katerih se problem lahko nahaja) in pa množico prehodov med stanji.

Glede na iskano rešitev, lahko v grobem probleme uvrstimo v enega izmed treh različnih razredov [4]:

- iskanje poti do znanega ciljnega vozlišča
- iskanje najboljše poteze v igri dveh igralcev
- iskanje (neznanega) ciljnega vozlišča, ki ustreza danim pogojem

V tej diplomski nalogi so obravnavani problemi, ki spadajo v prvega od teh, to se pravi, problemi, kjer imamo v splošnem podana začetno stanje in nek željeni cilj, ugotoviti pa moramo pot (zaporedje prehodov, potez, premikov, dejanj ...), ki vodi iz prvega v drugega.

Kot enostaven konkretni primer problema lahko vzamemo dvodimenzionalni prostor v katerem se nahaja robot. Stanja takega prostora stanj so kar vsi možni položaji v dejanskem prostoru (določeni s koordinatama višine in

širine), prehodi pa koraki gor, dol, levo ali desno, ki jih lahko naredi robot, da preide v nov položaj (novo stanje). V takšnem prostoru stanj lahko določimo začetni položaj in ciljni položaj, nato pa uporabimo preiskovalni algoritem, da poiščemo pot oziroma zaporedje korakov, ki bo robota pripeljalo do cilja.

Primer z iskanjem v dejanskem prostoru lahko razširimo še na bolj vsakdanje primere, recimo cestno navigacijo: kot stanja določimo množico vseh križišč na danem zemljevidu cest, kot prehode pa množico vseh cest, ki jih povezujejo. V takšnem prostoru stanj lahko določimo naše trenutno križišče in križišče, ki ga želimo obiskati. Rezultat preiskovanja je nato pot, ki jo moramo prevoziti.

Resnična vsestranskost reševanja s preiskovanjem pa leži v tem, da deluje tudi na precej bolj abstraktnih prostorih stanj. Vzemimo na primer Rubikovo kocko: imamo množico vseh možnih stanj te kocke, prehodi pa naj bodo vrtljaji poljubne vrstice ali stolpca na njej. Tak prostor stanj si je morda težko predstavljati, a je povsem veljaven. Če preiskovalnemu algoritmu kot začetek podamo trenutno stanje kocke, kot cilj pa stanje rešene kocke bo rezultat preiskovanja zaporedje vrtljajev, ki to kocko rešijo (če tako zaporedje obstaja).

Dve pomembni lastnosti prostorov stanj, ki bistveno vplivata na učinkovitost njihovega preiskovanja sta "faktor vejanja" (angl. *branching factor*), ki pove, koliko različnih prehodov v povprečju obstaja iz enega stanja in "globina rešitve" (angl. *solution depth*), ki označuje dolžino najkrajše poti od začetka do cilja glede na število prehodov [5].

Poglavje 3

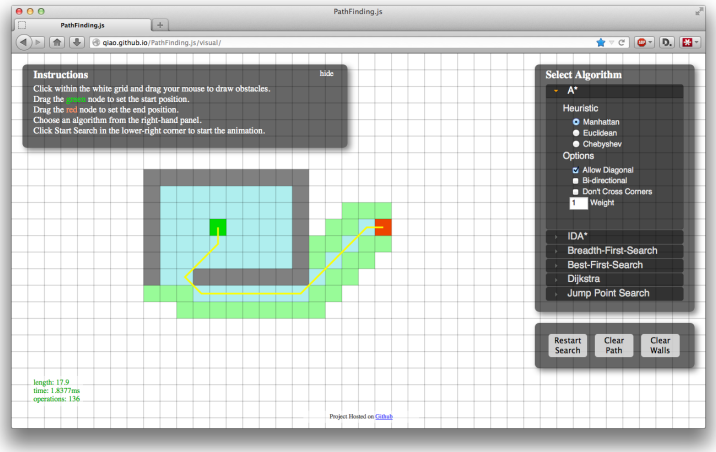
Obstoječe rešitve

V tem poglavju sta na kratko predstavljena dva vizualizacijska programa, na katera sem naletel med raziskovanjem snovi za diplomsko nalogo. Prav verjetno jih obstaja več, vendar sta le opisana med odkritimi zares podobna aplikaciji, ki je bila razvita v okviru te diplomske naloge.

3.1 Pathfinding.js

Pathfinding.js [7] je odprtokodna knjižnica preiskovalnih algoritmov napisana v programskem jeziku JavaScript in namenjena na mreži osnovanim igram, ki tečejo v spletnih brskalnikih. Vsebuje pet različnih preiskovalnih algoritmov: *A**, *iskanje v širino*, *Best-First Search*, *Dijkstrov algoritem* in *Jump Point Search*. Za predstavitev delovanja ima priložen preprost vizualizator, katerega vmesnik (slika 3.1) je na prvi pogled zelo podoben mojemu in to ni naključje: mreža obarvanih kvadratov, ki predstavljajo preiskovalni prostor ter plavajoče okno z nastavitvami sta mi služila kot navdih za izgled prvega prototipa mojega programa (vendar zgolj idejno, njegova koda in programska arhitektura sta povsem različni).

Vizualizator teče v spletnem brskalniku, uporabnik z miško določi začetno in končno polje (zelene oziroma oranžne barve), nariše morebitne ovire (sive barve), izbere enega izmed algoritmov in ga požene. Njegovo delovanje je



Slika 3.1: Vmesnik aplikacije Pathfinding.js

predstavljeno modrimi in zelenimi kvadrati, ki predstavljajo polja, ki so že bila obiskana oziroma polja, ki so kandidati za obisk. Ob koncu preiskovanja izriše pot, če ta seveda obstaja.

Kljub temu pa je prvinski namen vizualizatorja predstaviti delovanje knjižnice in njene vključitve v druge aplikacije, ne pa podrobna predstavitev delovanja posameznih algoritmov. Temu primerne so tudi omejitve:

- deluje samo na mreži, ki je končna, omejena z velikostjo brskalnikovega okna in vsebuje nabor zgolj takih algoritmov, ki so na mreži (relativno) učinkoviti;
- algoritmi se izvajajo s stalno hitrostjo, pomikanje po korakih naprej ali nazaj ni mogoče in ni izpisa o hevristični ocenih posameznih polj, zaradi tega je podrobnostim izvajanja težko slediti;
- ne deluje na mobilnih napravah;

V času pisanja te diplome je Pathfinding.js sicer še v aktivnem razvoju, zato se morda njegov aktualen nabor funkcij razlikuje od zapisanega tukaj.

Slika 3.2: Pathvisual v ukazni vrstici operacijskega sistema Ubuntu

3.2 Pathvisual

Pathvisual [6] je odprtokodni program za operacijski sistem Linux danskega avtorja Dana Amlunda Thomsena. Gre za v jeziku OCaml napisan program, ki teče izključno v ukazni vrstici, preiskovalne algoritme pa vizualizira s simboli.

Podobno kot prejšnji primer pathvisual vizualizira ravninski prostor stanj, ta pa je naključno generiran. Uporabnik ga ne more spreminjati, lahko pa zahteva, da se zgenerira nov prostor. Določi lahko začetek in cilj, nato pa na njem sproži enega izmed treh algoritmov: *Dijsktra*, *A** in *požrešni A**.

Čeprav je vizualizacija preprosta, pa omogoča prikaz po korakih (le naprej) s hevrističnimi ocenami vozlišč naslednikov. Glavni namen programa je sicer implementacija preiskovalnih algoritmov v funkcionalnem programskem jeziku in ne prav vizualizacija.

Pathvisual ni več v razvoju, druga (in hkrati zadnja) različica pa je izšla leta 2011.

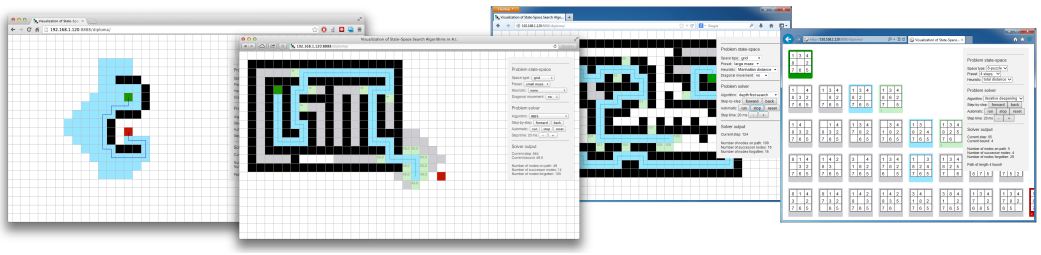
Poglavje 4

Vizualizator z vidika uporabnika

To poglavje služi kot neke vrste navodila za uporabo vizualizacijskega programa. V njem bom opisal kako do njega dostopati in kako ga uporabljati, ne bom pa se spuščal v podrobnosti in delovanje predstavljenih prostorov stanj in preiskovalnih algoritmov. Temu so namenjena kasnejša poglavja.

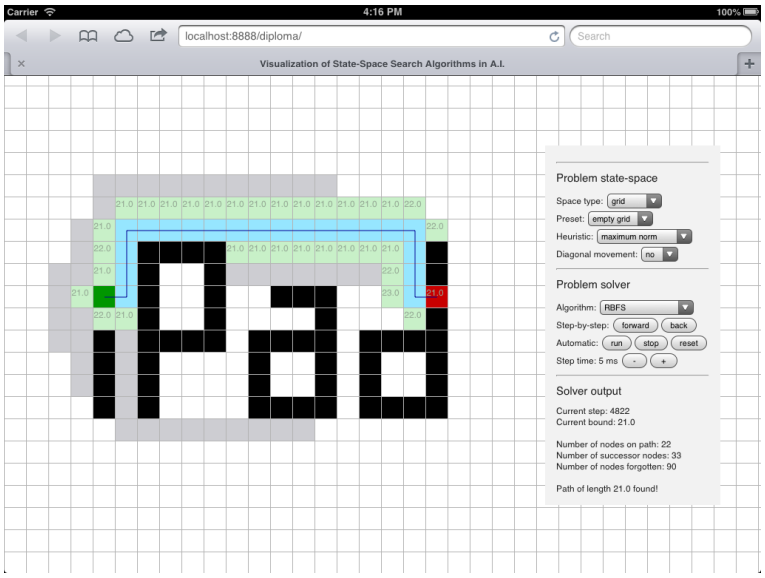
4.1 Sistemske zahteve

Z namenom, da bi bil uporabniku kar najbolj dostopen, sem vizualizator zasnoval kot spletno aplikacijo, ki je zgrajena zgolj na tehnologijah, ki so del spletnih standardov *World Wide Web Consortium (W3C)*. Zaradi tega je aplikacija povsem neodvisna od platforme na kateri teče in potrebuje le sodoben brskalnik, ki upošteva spletne standarde. Potrebna ni namestitev nobenega programa ali razširitve za brskalnik.



Slika 4.1: Aplikacija brez težav teče v brskalnikih Chrome, Safari, Firefox in Internet Explorer 9

Posledično jo je mogoče poganjati ne le na klasičnih osebnih računalnikih, temveč tudi na tabličnih računalnikih in pametnih mobilnih telefonih, ki imajo zmogljive in sodobne spletne brskalnike. Obetam si, da bo brez težav delovala tudi na napravah vsaj bližnje prihodnosti. Smernice nakazujejo, da bodo te naprave v vse večji meri občutljive na dotik, zato aplikacija omogoča tudi upravljanje na ta način (poleg kombinacije miške in tipkovnice).



Slika 4.2: Razvojna različica vizualizatorja, ki teče na tabličnem računalniku iPad

Slika 4.3: Razvojna različica vizualizatorja, ki teče na telefonu Galaxy Nexus in žepnem računalniku iPod touch

Kot pa je razvidno s slike 4.3, pa aplikacija vseeno potrebuje razumljivo velik ekran, saj njen vmesnik ni posebej prilagojen za manjše naprave, kot so na primer mobilni telefoni. Čeprav na njih brez težav deluje, pa je uporaba nekoliko nerodna.

4.2 Dostop do aplikacije

Kljub temu, da je moj vizualizator napisan kot spletna aplikacija pa njegovo izvajanje teče samo na uporabnikovi napravi in za to ne potrebuje sodelovanja oddaljenega strežnika. Uporabnik ima za dostop dve možnosti: lahko uporabi spletni naslov na katerem je aplikacija gostovana, ali pa jo prenese na lastno napravo in požene kar od tam, tako da odpre datoteko `index.html` v korenskem imeniku programa.

Velja omeniti, da nekateri spletni brskalniki zaradi varnostnih razlogov privzeto ne dovolijo poganjanja spletnih aplikacij direktno z naprave same,

tak je na primer Google Chrome. V takšnih primerih se je potrebno pozanimati kako to nastavitev izklopiti ali zaobiti.

4.3 Uporabniški vmesnik

Ob zagonu aplikacije se na zaslonu prikaže vizualizacija privzetega prostora stanj (mreže), ki zaseda celotno okno brskalnika, na desni strani pa je delno prekrita s plavajočim oknom, ki vsebuje gumbe in nastavitve za upravljanje z njo. Vmesnik je z namenom širše dostopnosti v angleščini.

Opomba: Razen če je omenjeno posebej, se na napravah občutljivih na dotik namesto klika miške uporablja tapkanje s prstom ali peresom, uporaba pa je sicer enaka.

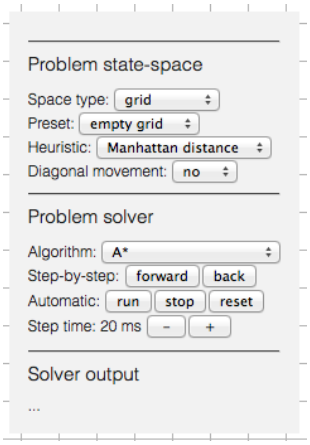
4.3.1 Nadzorno okno

Nadzorno okno je plavajoče, kar pomeni, da ga lahko postavimo na poljuben položaj v brskalnikovem oknu. To storimo tako, da nanj kliknemo na obrobi in ga z miško povlečemo na želeno mesto. Razdeljeno na tri logične razdelke (glej sliko 4.4):

1. upravljanje s prostorom stanj (“Problem state-space”)
2. nadzor nad preiskovalnim algoritmom (“Problem solver”)
3. izpis informacij o trenutnem stanju algoritma (“Solver output”)

Upravljanje s prostorom stanj

V prvem razdelku uporabnik določi kateri prostor stanj želi vizualizirati, naloži še morebitno začetno konfiguracijo v tem prostoru, določi hevristično funkcijo za informirane preiskovalne algoritme in morebitne preostale nastavitve danega prostora:



Slika 4.4: Nadzorno okno kot izgleda v programu

- `space type`: izbira prostora stanj, na voljo sta mreža (grid) in igra osmih kvadratov (8-puzzle)
- `preset`: izbira nekaj vnaprej določenih konfiguracij prostora stanj, a uporabnik lahko naredi tudi svojo, brez uporabe tega
- `heuristic`: hevristična funkcija za oceno stanj, ki jo uporabljajo informirani preiskovalni algoritmi, lahko je konstanta 0 (none)
- `diagonal movement`: samo na prostoru mreže, če je vklopljeno, dovoli gibanje po diagonali

Nadzor nad preiskovalnim algoritmom

Drugi razdelek omogoča izbiro preiskovalnega algoritma in nadzor nad njegovim izvajanjem: to je lahko korak za korakom ali pa samodejno z danim tempom:

- `step-by-step`: gumba naprej (forward) in nazaj (back) omogočata vizualizacijo algoritma po korakih, tako naprej kot tudi nazaj

- `automatic`: gumbi poženi (run), ustavi (stop) in ponastavi (reset) omogočajo samodejno izvajanje algoritma brez uporabnikovega klikanja

- `step time`: tukaj uporabnik lahko določi hitrost samodejnega izvajanja v času med posameznima korakoma algoritma, podanem v milisekundah; največja možna vrednost je 1000 ms (korak na sekundo), najmanjša pa 0 ms, kjer se algoritem izvaja kolikor hitro dopušča zmogljivost uporabnikovega računalnika

Izpis o stanju preiskovalnega algoritma

V zadnjem razdelku uporabnik ne more vnašati ali spreminjati ničesar, saj je namenjeno zgolj izpisu. Tekom delovanja preiskovanja se tudi izpisujejo informativni podatki, ki so merodajni za dan algoritem (izpis je torej za vsakega izmed njih nekoliko drugačen). Če se preiskovanje še ni začelo, ali pa je bilo stanje prostora/algoritma ponastavljeno, je v tem razdelku zgolj tropičje (...).

4.3.2 Vizualizacija prostora stanj

V aplikaciji sta implementirana dva različna prostora stanj z dvema različnima vizualizacijama: mreža, kjer so stanja predstavljena s kvadrati na njej in igra osmih kvadratov, kjer so predstavljena s konfiguracijami igre.

Čeprav se vizualizaciji precej razlikujeta, pa sem zavoljo enotnosti na obeh uporabil enako barvno shemo za stanja:

- s temno zeleno je označeno začetno stanje

- s temno rdečo je označeno ciljno stanje

- s črno barvo so (samo na mreži) označeni tako imenovani “zidovi”, to so stanja oziroma položaji, ki niso dovoljeni

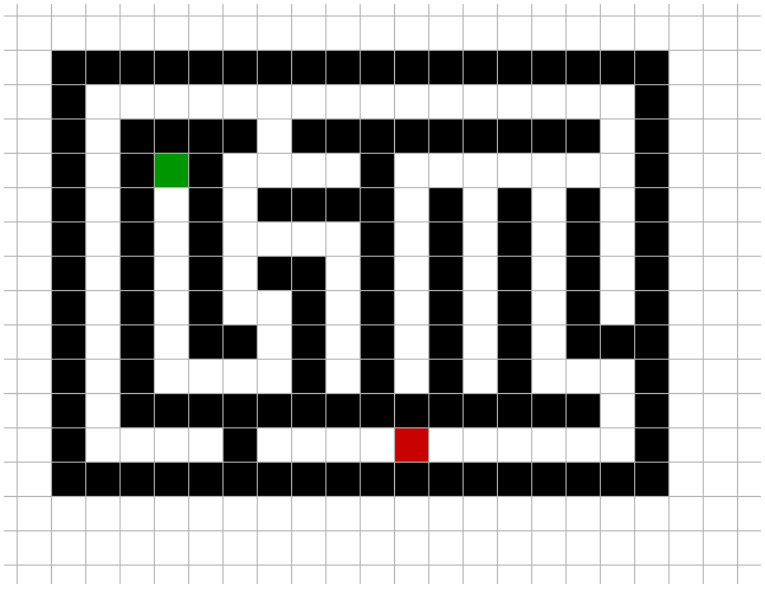
- s svetlo modro so označena stanja, ki jih je preiskovalni algoritem že obiskal in so še shranjena v njegovem spominu
- s sivo so označena stanja, ki jih je preiskovalni algoritem že obiskal, vendar pa niso več shranjena v njegovem spominu (so "pozabljena")
- s svetlo zeleno so označena stanja, ki jih algoritem pozna in so shranjena kot kandidati za obisk; ta stanja imajo dodatno izpisano številsko vrednost, ki predstavlja globino pri neinformiranih in skupno oceno pri informiranih algoritmih
- z vijolično pa je označen trenutni položaj; ta označba je potrebna in prisotna samo pri algoritmu RTA*

Ob koncu preiskovanja se na vizualizaciji izriše pot temno modre barve, če ta seveda obstaja. Nekateri algoritmi (iskanje v globino, iterativno poglabljanje, IDA* in RBFS) tekom delovanja v spominu držijo samo trenutno pot po kateri preiskujejo. Zaradi nazornosti je ta pri teh med izvajanjem izrisana s turkizno zeleno, če pa ob koncu prava pot ni najdena, se obarva rdeče.

Mreža

Z mrežo je predstavljen (v teoriji) neskončen dvodimenzionalni diskreten prostor pri čemer vsak kvadrat označuje položaj oziroma stanje v tem prostoru. Začetek in cilj lahko uporabnik spreminja tako, da nanju klikne in ju povleče na želena mesta. Poleg tega lahko na mrežo prosto riše zidove, polja, ki jih ni mogoče obiskati. Tako na enostaven način zariše labirinte in ovire ali celo omeji preiskovanje na končen prostor (primer na sliki 4.5). Zidove zbriše tako, da nanje klikne. Risanje in spreminjanje mreže bo vedno povzročilo, da se morebitno izvajanje preiskovalnega algoritma ustavi in vrne na začetek.

Ker je mreža neskončna, okno brskalnika pa seveda ne, lahko mrežo premikamo tako, na tipkovnici med držanjem tipke SHIFT kliknemo nanjo in miško z zadržanim gumbom premikamo v želeno smer. Naprave občutljive na



Slika 4.5: Na mrežo zarisan labirint

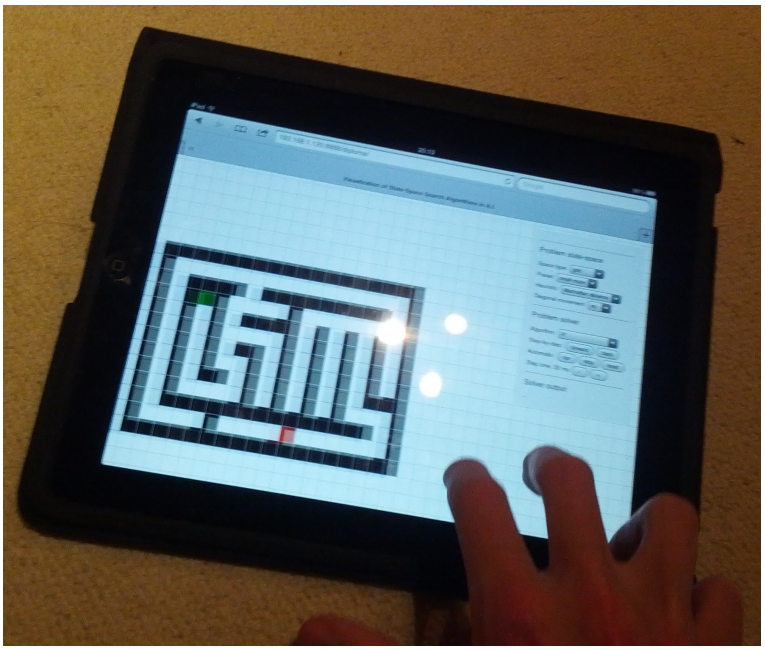
dotik tipkovnice (običajno) nimajo; namesto tega se mreže lahko dotaknemo z dvema prstoma naenkrat in ju premikamo vzporedno (slika 4.6).

Igra osmih kvadratov

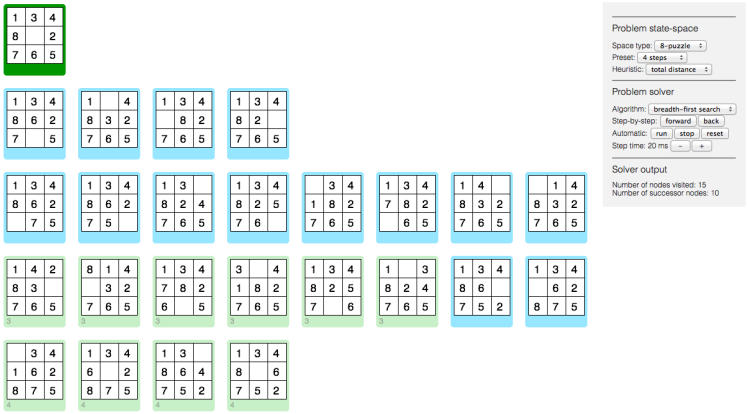
Nekoliko bolj abstrakten prostor stanj, igra osmih kvadratov, je predstavljena z drevesnim grafom, kjer vsako vozlišče v drevesu prikazuje možno stanje igre. Vizualizacija je zelo preprosta (slika 4.7) in ne omogoča toliko spreminjanja kot mreža.

Uporabnik lahko izbere eno izmed začetnih stanj iz danega nabora, nato pa opazuje delovanje algoritmov na drevesu – ko so vozlišča enkrat tvorjena, se njihov položaj ne spremeni. Ob koncu iskanja se izriše pot.

V večini primerov se zgodi, da velikost grafa preseže velikost prikaznega okna. Uporabnik se lahko po njem premika tako, da ga klikne in povleče v poljubno smer.



Slika 4.6: Dvoprstno premikanje mreže na iPadu



Slika 4.7: Razvojna različica vizualizatorja, ki prikazuje iskanje v širino pri igri osmih kvadratov

Poglavje 5

Preiskovalni algoritmi

Preiskovalni algoritmi so postopki za iskanje poti (rešitve) v danem prostoru stanj za neko začetno in končno stanje. So generični in od prostora neodvisni. Najprej bodo predstavljeni nekateri izzivi ki jih je predstavljala implementacija algoritmov, nato pa se posvetil podrobnostim vsakega posameznega algoritma v vizualizacijski aplikaciji posebej. Formalni opise in dokazi nekaterih njihovih lastnosti bodo izpuščeni, saj so v raznorazni literaturi zapisani bolje in bolj natančno, kot bi to bilo smiselno narediti tu [1] [4].

Opomba: V izogib dvoumnosti se bo v preostanku poglavja za prostorska stanja uporabljala sopomenska beseda *vozlišče*. Vsak problemski prostor stanj lahko namreč predstavimo z grafom v katerem so stanja njegova vozlišča, povezave med njimi pa prehodi.

5.1 Izzivi ob implementaciji

Prvi korak implementacije vsakega algoritma je seveda bil preučevanje delovanja; s tem večinoma ni bilo težav vendar je vredno omeniti, da avtorji pogosto pod istim imenom opisujejo med seboj nekoliko različne variante algoritmov in kar je še pomembneje, predstavljajo jih v različnih programskih jezikih, ki sledijo različnim paradigmam.

To v tej nalogi ni prišlo v upoštev, zato sem moral vsako implementacijo

napisati nekoliko po svoje (ter ob tem seveda paziti, da je ta še vedno pravilna). Ker sem hotel algoritme vizualizirati, pa je to prineslo nekaj dodatnih izzivov in omejitev, katerim sem jih moral prilagoditi:

5.1.1 Delitev na korake

Aplikacija uporabniku omogoča izvajanje algoritma po korakih, kjer delovanje med vsakima korakoma miruje in je po vsakem videna neka smiselna sprememba oziroma napredek. Pri tem je bilo potrebno premisliti, kaj je smiselno vključiti v en sam korak, kar pri nekaterih algoritmih ni čisto samoumevno. Hkrati je zaradi te zahteve tudi odpadla možnost uporabe rekurzije; a to samo po sebi ni velika težava, saj jo lahko "simuliramo" z uporabo dodatnega sklada.

5.1.2 Hranjenje informacij o vseh obiskanih stanjih

Da si uporabnik lahko nazorno predstavlja trenutno stanje preiskovanja, je nujno potrebno, da je na vizualizaciji razviden status vsakega obiskanega vozlišča: je to že obiskano, ali je morda kandidat za obisk, kakšna je njegova vrednost, in tako naprej. Zato je potrebno, da algoritem vsako vozlišče ob obisku ustrezno označi (ali po potrebi to označbo spremeni).

Nekateri preiskovalni algoritmi v spominu hranijo vsa obiskana vozlišča od začetka pa do konca izvajanja, drugi pa obiskana vozlišča zavržejo oziroma jih "pozabijo", če se ta ne izkažejo za obetavna. V praktični rabi je ta lastnost lahko zelo dobrodošla, saj je kritičnega pomena pri zmanjševanju pomnilniške zahtevnosti. Ravno zaradi tega, ker pa želimo pozabljena vozlišča prikazati, pa jih morajo algoritmi (ironično) nujno ohraniti v spominu, kar dodatno zaplete implementacijo.

Seveda se hkrati s tem izgubi tudi prostorska učinkovitost teh algoritmov, vendar to za namene vizualizacije ni tako pomembno.

5.1.3 Korak nazaj

Za morda najbolj zahteven del programiranja izvajanja algoritmov za vizualizacijo pa se je izkazala potreba po koraku nazaj, ki preiskovanje vrne v stanje pred zadnjim korakom izvajanja (ena izmed bistvenih funkcij aplikacije). Zelo hitro sem namreč prišel do ugotovitve, da se pri nobenem od obravnavanih algoritmov iz stanja v trenutnem koraku ne da enolično določiti predhodnika.

Prva ideja, ki se mi je porodila ob reševanju tega problema je bila uporaba programskega sklada, na katerega bi ob vsakem koraku algoritma shranil kopijo celotnega stanja vozlišč in preiskovanja. V primeru koraka nazaj bi se v prejšnji korak vrnil enostavno tako, da bi ga vzel s sklada. Ta način bi bil generična rešitev za vse algoritme, a se je izkazal za skrajno neučinkovitega: nenehno kopiranje podatkovnih struktur v vsakem koraku je izjemno upočasnilo izvajanje, hkrati pa zaradi eksponentne rasti zelo hitro zasedlo celoten pomnilnik.

Tako ni preostalo drugega, kot da sem za vsak algoritem napisal lastno rešitev, pri čemer sem se opiral na kodo koraka naprej in ugotavljal, kakšne informacije se ob njem izgubijo. Algoritem sem nato razširil s shranjevanjem teh v dodatne podatkovne strukture, in jih nato uporabil pri programiranju koraka nazaj. Kot omenjeno v prejšnjem paragrafu mora algoritem tudi dodeljevati in spreminjati vizualizacijske oznake vozlišč in za potrebe koraka nazaj je nujno hraniti tudi zgodovino teh sprememb. Cena tega sta veliko večja zapletenost programske kode in občutno večja poraba pomnilnika.

5.2 Neinformirani algoritmi

Za razred neinformiranih algoritmov velja, da vozlišča prostora med obiskovanjem vrednotijo izključno na podlagi njihove oddaljenosti od začetnega stanja po številu korakov. So enostavni, vendar pa preiskujejo povsem “na slepo”.

5.2.1 Iskanje v globino

Pri iskanju v globino (angl. *depth-first search*) se znana vozlišča obiskujejo tako, da imajo prednost tista z največjo oddaljenostjo od začetnega. Gre za pomnilniško zelo učinkovit algoritem, saj zanj potrebujemo samo dva seznama vozlišč, ki tekom preiskovanja naraščata linearno:

- trenutna pot – seznam obiskanih vozlišč
- kandidati – seznam vozlišč, ki so kandidati za obisk

Korak algoritma

V vsakem koraku vzamemo prvo vozlišče s seznama kandidatov; v kolikor je ciljno, izvajanje uspešno zaključimo in vrnemo pot, sicer pa ga postavimo na trenutno pot, hkrati pa z nje pobrišemo in kot pozabljena označimo vozlišča z enako ali večjo globino. Nato v prostoru stanj tvorimo njegove naslednike in jih vrinemo na začetek seznama kandidatov. Če je ta na začetku koraka prazen, potem pot ne obstaja in se iskanje zaključi.

Korak nazaj

Pri koraku nazaj pa je potrebno storiti naslednje: iz seznama kandidatov izbrišemo vozlišča dodana v zadnjem koraku, vanj vrnemo zadnji element s poti in nanjo vrnemo morebitne pozabljene. To zahteva, da vodimo trenutno število korakov in z njim tudi označujemo kdaj natanko smo spreminjali označbe vsakega vozlišča.

Učinkovitost

Iskanje v globino ima nekaj slabosti, ki na prostorih stranj v mojem vizualizatorju še posebej pridejo do izraza:

- v neskončnem prostoru ni zagotovila, da se bo izvajanje končalo
- ni zagotovila, da bo našlo najkrajšo pot do cilja

- če obišče vozlišče, ki je že na poti, se "zacikla" in teče v neskončnost

Prvo in drugo se da enostavno vizualizirati na mreži (ki je neskončna). Tretjo pa je nujno potrebno odpraviti, sicer na algoritem na taki mreži sploh ne bi bil delujoč, cikel bi naredil že v štirih korakih. K sreči je mehanizem za to enostaven: pred dodajanjem naslednika vozlišča med kandidate za obisk preverimo, če se nemara že nahaja na trenutni poti in v kolikor se, ga zavržemo.

5.2.2 Iskanje v širino

Iskanje v širino (angl. *breadth-first search*) vozlišča obiskuje ravno v nasprotnem vrstnem redu: najprej obišče tista najbližje začetnemu.

Korak algoritma

V enem koraku algoritma vzamemo prvo vozlišče s seznama kandidatov. V prostoru stanj tvorimo njegove naslednike in vsakemu damo referenco na svojega predhodnika (trenutno vozlišče). Nato jih postavimo na konec seznama kandidatov. S pomočjo referenc na predhodna vozlišča hranimo vse obiskane poti v obliki vezanih seznamov. Ko naletimo na ciljno vozlišče, lahko preko njih enostavno rekonstruiramo tudi pot.

Korak nazaj

Razveljavljanje koraka pri iskanju v širino je enostavno: s seznama kandidatov izbrišemo nazadnje dodane naslednike in na njegov začetek vrnemo njihovega predhodnika.

Učinkovitost

Zaradi potrebe po hranjenju poti do vseh znani vozlišč nobenega od obiskanih ne moremo zavreči oziroma pozabiti. Posledica tega je, da ima iskanje v širino v primerjavi z iskanjem v globino precej večjo pomnilniško zahtevnost, po drugi strani pa zanj ne velja nobena od zgoraj omenjenih slabosti.

5.2.3 Iterativno poglabljanje

Kot kompromis med prejšnjima dvema preiskovalnima algoritmoma se ponuja iterativno poglabljanje (angl. *iterative deepening*), ki je v bistvu različica iskanja v globino. Pri njem je globina preiskovanja omejena (začenši z 1) in se iterativno povečuje v primeru, da preiskovanje s trenutno omejitvijo ni naletelo na cilj.

Korak algoritma

Posamezen korak iterativnega poglabljanja je skorajda enak navadnemu iskanju v globino, a z dvema razlikama:

- naslednike, katerih globina presega trenutno globinsko omejitev zavržemo
- ko se seznam kandidatov izprazni (in pot ni najdena), omejitev povečamo in iskanje pričnemo od začetka

Korak nazaj

Korak nazaj prav tako implementiran podobno, pravzaprav enako; posebej je potrebno obravnavati samo razveljavitev povečanja omejitve. Ker je iskanje takrat vrnjeno na začetek, se v prejšnje stanje ne moremo vrnit le z majhno spremembo. To sem rešil tako, da celotno stanje algoritma in prostora ob povečavi globinske omejitve naložim na sklad (in ga ob koraku nazaj od tam vrnem). Gre za relativno redek dogodek, zato je to sprejemljiva rešitev.

5.3 Informirani algoritmi

Z razliko od neinformiranih pa se informirani preiskovalni algoritmi med preiskovanjem opirajo na informacijo, ki izvira iz prostora samega. Namesto, da bi bilo preiskovanje na slepo, se prednost daje obiskovanju najprej tistih vozlišč za katera si najbolj obetamo, da nas bodo pripeljala do iskanega cilja.

Obetavnost vsakega izmed vozlišč kandidatov za obisk predstavlja tako imenovana f-vrednost (angl. *f-value*), ki jo tvorita dve komponenti ($f = g + h$):

- g: znana vrednost vozlišča - bodisi v obliki števila korakov od začetnega vozlišča, bodisi v obliki vsote cen teh korakov
- h: ocena oddaljenosti tega vozlišča do cilja

O oceni h v splošnem ne moremo povedati nič, saj izhaja iz hevristične funkcije, ki je vezana na dan prostor stanj. Najbolj obetajoča so vozlišča z najmanjšo f-vrednostjo.

5.3.1 A*

A* je zaradi svoje enostavnosti in relativne učinkovitosti danes široko uporabljan in zastopan preiskovalni algoritem. Podobno kot pri iskanju širino vsako vozlišče nosi referenco na svojega predhodnika iz katere se lahko rekonstruira pot do njega, poleg tega pa uporablja še množico vozlišč kandidatov, ki so urejeni po svoji f-vrednosti.

Korak algoritma

V vsakem koraku A* se iz množice kandidatov vzame vozlišče z najnižjo oceno in ga označi kot obiskanega. Nato se v prostoru stanj tvori množica njegovih naslednikov, ki se dodajo v množico kandidatov. Če se tam že nahajajo jih se jih zamenja zgolj v primeru, da je ocena obstoječega vozlišča v množici večja od ocene ravnokar tvorjenega.

Korak nazaj

Kot pri iskanju v širino je razveljavitev koraka algoritma razmeroma enostavna. Nazadnje dodana vozlišča iz množice kandidatov se pobriše in vanjo vrne njihovega predhodnika. Ker pa so ta vozlišča morda nadomestila kakšna

slabše ocenjena, jih je potrebno zaradi vizualizacije vrniti. V množici kandidatov morajo vozlišča po potrebi torej nositi referenco na nadomeščeno vozlišče.

Popolnost

Za preiskovalni algoritem rečemo da je popoln (angl. *admissible*), če vedno najde optimalno rešitev oziroma pot. A* je (po izreku o popolnosti) popoln, če za vsako vozlišče v prostoru stanj velja, da je njegova ocena poti do cilja h manjša ali enaka dejanski ceni $h*$ te poti.

Takšnemu pogoju je zadostiti trivialno (z oceno, ki je vedno enaka 0), v splošnem pa je izziv poiskati hevristično funkcijo, ki algoritem ohranja popoln in hkrati čim bolje vodi njegovo preiskovanje.

Učinkovitost

V primerjavi iskanju v širino A* v splošnem obišče precej manjše število vozlišč in je v praksi manj pomnilniško zahteven. Kljub temu pa je njegova pomnilniška zahtevnost v najslabšem primeru še vedno eksponentna, kar je za nekatere velike probleme (ali omejene sisteme) še vedno nesprejemljivo. Vsi ostali informirani preiskovalni algoritmi obravnavani v tej nalogi so različice A*, ki to pomanjkljivost odpravljajo na račun časovne zahtevnosti ali popolnosti.

5.3.2 IDA*

IDA* (krajšava za angl. *iterative deepening A**) je v bistvu različica neinformiranega preiskovalnega algoritma iterativno poglabljanje in se od njega deluje razlikuje samo v tem, da se globinsko preiskovanje omejuje na podlagi f-ocene vozlišč namesto na podlagi globine. Delovanje algoritma se tako usmerja z mejo preiskovanega podprostora.

Korak algoritma

Kot pri iterativnem poglabljanju je korak IDA* enak koraku iskanje v globino, z dvema razlikama. Naslednike obiskanega vozlišča se zavrže, če njihova ocena presega trenutno mejo, kljub temu pa se hrani najmanjša ocena zavrženih vozlišč. Če preiskovanje cilja s trenutno omejitvijo ne najde, se omejitev poveča na to vrednost in iskanje ponovi.

Korak nazaj

Korak nazaj je povsem enak kot pri iterativnim poglabljanjem, vključno z uporabo sklada za morebitno vrnitev v stanje algoritma in prostora pred povečavo omejitve.

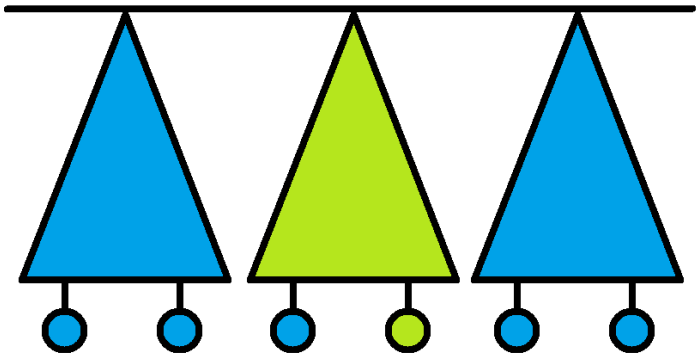
Učinkovitost

Za popolnost IDA* velja podoben izrek o popolnosti kot pri A*, vendar z dodatnim pogojem, da mora biti hevristična funkcija monotona. To zadošča, da bo algoritem našel optimalno rešitev. Prednost IDA* je v zelo majhni rabi pomnilnika, hraniti mora le trenutno pot in seznam kandidatov za naslednike, oba naraščata samo linearno.

Cena za to je precej večja časovna zahtevnost, saj so mnoge poti obiskane večkrat. To še posebej pride do izraza, če si enako f-vrednost deli le malo vozlišč; v takem primeru se območje preiskovanje ob višanju omejitve veča počasi, preiskati pa ga je potrebno vedno znova. V nekaterih problemih je povečanje v časovni zahtevnosti tako veliko, da se uporaba IDA* enostavno ne izplača.

5.3.3 RBFS

Preiskovalni algoritem RBFS (krajšava za angl. *recursive best-first search*) je različica algoritma A*, ki ima podobno kot IDA* samo linearno pomnilniško zahtevnost. Razliko med A* in RBFS je opisati relativno preprosto, vendar pa je implementacija slednjega veliko bolj zapletena.

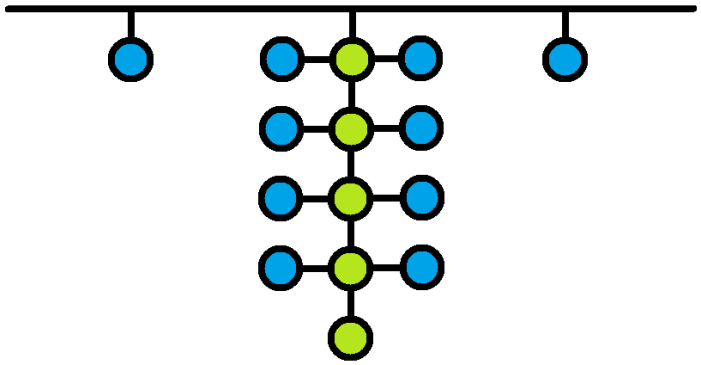


Slika 5.1: A* poleg trenutno najbolj obetavne poti (zelena) hrani tudi vsa ostala obiskana poddrevesa (modra)

Za primerjavo si lahko A* zamislimo kot algoritem, ki v spominu hrani sosednja poddrevesa in preiskuje najbolj obetavnega. Če ocene naslednikov trenutno preiskovanega poddrevesa postanejo večje kot ocene naslednikov sosednjega poddrevesa, začne algoritem preiskovati slednjega, trenutno poddrevo pa ostane v spominu in ga, če se k njemu vrnemo, ni potrebno ponovno preiskovati (skicirano na sliki 5.1).

Delovanje RBFS lahko predstavimo na enak način, vendar z eno pomembno razliko: da zmanjša porabo pomnilniškega prostora, preiskovanje ob zamenjavi preiskovanega poddrevesa trenutnega enostavno pozabi in nadomesti samo z njegovim korenskim vozliščem. Ker pa je potrebno vedeti, kdaj in če bo vrnitev v to poddrevo spet obetavna se njegovo korensko vozlišče oceni z najmanjšo oceno njegovih listov (skicirano na sliki 5.2). Cena ponovnega obiska je, da se morajo vsa pozabljena vozlišča tvoriti ponovno.

RBFS je v tej nalogi predstavljal še posebej velik izziv; v literaturi so implementacije predstavljene v rekurzivni obliki, ki delovanju tega preiskovalnega algoritma tudi najbolj ustreza, a takšna oblika ob zahtevi po delitvi na korake ne pride v upoštev. Za rešitev se je izkazala uporaba skladovnih podatkovnih struktur, ki hranijo trenutno stanje (preiskovano poddrevo, ko-



Slika 5.2: RBFS poleg trenutno najbolj obetavne poti (zelena) hrani samo korene preostalih poddreves (modra)

reni sosedov in najmanjša ocena sosed starša) v vsaki cepitvi na poddrevesa (torej vsakem vozlišču na poti).

Korak algoritma

Poleg skladov algoritem hrani naslednje spremenljivke: trenutni seznam sosednjih neobiskanih vozlišč in dano mejo ocene (najmanjšo oceno sosed starša ali sosed starega starša in tako naprej). V posameznem koraku se (zaradi nazornosti vizualizacije) zgodi točno ena izmed dveh stvari; pozabljanje ali obiskovanje.

Če ocene neobiskanih vozlišč presegajo mejo, trenutno poddrevo ni več najobetavnejše in ga potrebno pozabiti. To se stori tako, da se s skladovnih struktur vzame stanje prejšnjega (višjega) nivoja, v katerem se korenu ravnokar pozabljenega poddrevesa dodeli nova ocena – ta je enaka najmanjši oceni pozabljenih listov.

V nasprotnem primeru pa se trenutno stanje porine na sklade in iz najobetavnejšega neobiskanega vozlišča tvori njegove naslednike. Te se shrani v nov seznam neobiskanih vozlišč, kot nova meja pa se določi ali meja trenutnega nivoja ali pa najmanjša ocena sosed vozlišč ravnokar obiskanega, kar

je pač manjše.

Korak nazaj

Razveljavljanje koraka RBFS je odvisno od njegove vrste. Če je šlo za obiskovanje novega vozlišča je postopek enak kot pri pozabljanju: trenutno stanje se pozabi (v tem primeru zares) in s skladov naloži prejšnje, a v tem primeru brez spreminjanja ocen.

Korak nazaj pri pozabljanju poddreves pa zahteva, da se vodi množica vseh pozabljenih vozlišč vključno s številkami korakov v katerih so bili odstranjeni. Ker pa ima vsako pozabljeno vozlišče tudi svoje stanje (množico zgoraj omenjenih spremenljivk), ga je treba ohraniti. Temu služi še en dodaten nabor skladovnih struktur, kamor se potisne stanje nivoja, ko je to označeno kot pozabljeno. V koraku nazaj se pozabljena vozlišča označi nazaj v prejšnje stanje (obiskano ali naslednik), trenutno stanje porine nazaj na glavne sklade in prejšnje vrne s sklada pozabljenih.

Učinkovitost

RBFS ravno tako kot IDA* ne žrtvuje svoje optimalnosti in je popoln ob enakem pogoju (ustrezni hevristični funkciji). Ima enako linearno pomnilniško zahtevnost in v splošnem zaradi nekoliko manjšega popolnega obiskovanja tudi manjšo časovno zahtevnost. Njegova slabost leži v zapletenosti, prav tako pa se tudi RBFS lahko zgodi, da ponovno obiskovanje pozabljenih vozlišč preveč poveča časovno zahtevnost.

5.3.4 RTA*

Od ostalih obravnavanih preiskovalnih algoritmov je RTA* [3] (krajšava za angl. *real-time A**) poseben tako v namenu kot tudi delovanju.

Ideja tega algoritma je, da omogoča sočasno gibanje agenta v prostoru še preden je znana globalna pot do cilja. Algoritem hrani trenutno stanje

in vedno izvaja samo lokalno preiskovanje do neke fiksne globine, nato pa se odloči za premik v najbolj obetaven položaj v tej preiskani okolici.

Ker bi morebitna vrnitev v že obiskan položaj povzročila ciklanje (saj bi algoritem za naslednika spet izbral isto vozlišče), je za odpravljanje te težave mehanizem spreminjanja ocene trenutnega vozlišča, priredi se ji vrednost ocene drugega najbolj obetavnega naslednika.

Korak algoritma

V koraku RTA* se vzame trenutni položaj in tvori njegove naslednike v prostoru stanj. Za vsakega izmed njih se preveri, ali se nahaja v množici obiskanih vozlišč; v kolikor je odkrit prvič, njegovo oceno določa hevristična funkcija prostora stanj, če pa je v preteklosti že bil obiskan, pa se uporabi že določena vrednost iz množice.

Iz naslednikov se izbere tistega z najmanjšo oceno, hkrati pa se shrani tudi ocena drugega najbolj obetavnega naslednika. Ta se v množici obiskanih priredi trenutnemu položaju. Kot nov trenutni položaj se izbere najboljši naslednik, ostali pa se zavržejo.

Korak nazaj

Za izvedbo vrnitve v prejšnje stanje algoritma se uporabljata dva dodatna sklada, na prvem so prejšnji trenutni položaji, na drugem pa prejšnji zavrženi nasledniki. Vozlišča v množici obiskanih pa vsebujejo še reference na svoje predhodno stanje (vezan seznam). Ob koraku nazaj se prejšnje stanje algoritma enostavno naloži s skladov, ocena v množici obiskanih vozlišč pa se po referenci vrne na prejšnje stanje ali pa izbriše.

Učinkovitost

Preiskovanje z RTA* ni popolno in zanj ni nobenih zagotovil, da bo najdena pot zares optimalna. Kljub temu pa bo pot, če je prostor preiskovanja končen in če ta obstaja algoritem vedno našel. V neskončnih prostorih ali v prostorih kjer poti ni, se lahko zgodi, da se ne bo ustavil nikoli.

Ker mora za svoje delovanje hraniti množico vseh že obiskanih položajev, pomnilniška zahtevnost raste vsaj polinomsko. Zaradi posebne namembnosti pa direktna primerjava RTA* z ostalimi zgoraj opisanimi algoritmi ni najbolj smiselna. V praksi se zelo dobro obnese na vizualizacijskih primerih z mrežo.

Poglavje 6

Prostori stanj

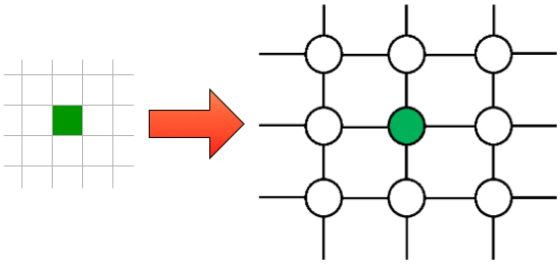
Kot je opisano v drugem poglavju, je prostor stanj določen z množico stanj in množico prehodov med stanji. V splošnem se da vsak prostor stanj predstaviti kot graf, na katerem vozlišča predstavljajo stanja, povezave med njimi pa prehode. Za vozlišči, ki sta medsebojno povezani, rečemo, da sta sosednji. Povezave so lahko obtežene, kar označuje ceno prehodov med posameznimi stanji.

Namen tega poglavja je natančneje opisati in predstaviti dva prostora stanj, ki sta vključena v vizualizacijsko aplikacijo. Na kratko sta bila predstavljena že v četrtem poglavju. Tukaj so razložene nekatere njune lastnosti, kako se ju lahko konfigurira in kakšne hevristične funkcije vsebujeta za ocenjevanje svojih vozlišč.

6.1 Mreža

Morda najbolj intuitivna predstavitev prostora je kar z dejanskim evklidskim prostorom, enostavna različica tega pa je mreža. Mreža je diskretiziran dvodimenzionalni prostor (torej ravnina) s števno mnogo položaji oziroma stanji.

Položaj na mreži je enolično določen s celoštevilskima koordinatama višine in širine. Njemu sosednje položaje tvorimo tako, da natanko eni izmed koor-



Slika 6.1: Že brez diagonal se v okolici enega vozlišča na prazni mreži nahaja veliko število ciklov in poti

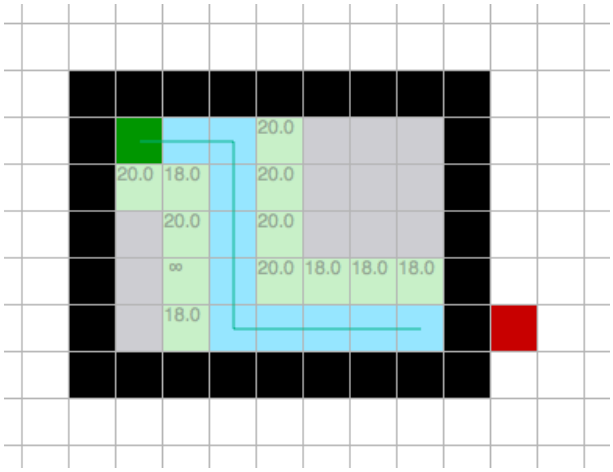
dinat prištejemo ali odštejemo 1 (lahko tudi obema, v primeru, da je soseščine definirana tudi "po diagonali").

Za določene položaje lahko velja, da niso veljavno stanje prostora stanj (v tej nalogi se jih opisuje z intuitivnih izrazom "zidovi"). Teh se kot sosedov ne da tvoriti.

6.1.1 Lastnosti

Mreža ima nekaj posebnosti. Razen v konfiguracijah kjer je omejena z zidovi v vseh smereh, predstavlja neskončen prostor. Če je predstavljana z grafi je mogoče opaziti, da vsebuje zelo veliko ciklov (slika 6.1). Brez zidov je njen povprečni faktor vejanja 4 (oziroma 8 z diagonalno sosednostjo).

Te lastnosti precej vplivajo na učinkovitost delovanja preiskovalnih algoritmov. Pri iskanju v globino (in v nekaterih primerih, RTA*) se lahko zgodi, da se odpravi v neskončnost in cilja nikoli ne najde. Pri drugih, ki obiskana vozlišča "pozabljajo" (IDA*, RBFS, iterativno poglabljanje ...), pa težave povzroča, da do vsakega vozlišča zaradi velikega števila ciklov obstaja ogromno različnih in pogosto enakovrednih poti (z oddaljenostjo vozlišča njihovo število kombinatorično raste). Četudi se to vozlišče izkaže za neobetavnega, bodo preiskane vse poti do njega, kar je izjemno potratno (za primer slika 6.2).



Slika 6.2: Preiskovalni algoritem RBFS v precej majhnem prostorčku preišče več 1000 poti na istih vozliščih preden ugotovi, da pot do cilja ne obstaja

Pomembno pa je omeniti, da so lastnosti prostora mreže povsem odvisne od konfiguracije. Koncept zidov namreč pomeni, da se prostor lahko omeji, faktor vejanja zmanjša in cikle povsem odpravi (na primer z labirinti).

Takšna prilagodljivost prostora je velika prednost, saj lahko na enostaven način predstavimo kako njegove lastnosti vplivajo na učinkovitost preiskovanja.

6.1.2 Gibanje po diagonali

Uporabnik ima možnost vklopa gibanja po diagonali; to pomeni, da se kot nasledniki položaja na mreži tvorijo tudi njegove diagonalne sosede.

Diagonalne povezave so obtežene s ceno $\sqrt{2}$ z razliko od ostalih povezav, katerih cena je kar 1. Razlog zato izhaja iz dejstva, da mreža predstavlja evklidski prostor.

Neinformirani algoritmi kot je na primer iskanje v širino cene povezav ne upoštevajo, zato se lahko zgodi, da najdejo rešitev, ki je optimalna zgolj po številu korakov, ne pa tudi po evklidski dolžini poti.

6.1.3 Prednastavljene konfiguracije

V aplikacijo so vključene naslednje prednastavljene konfiguracije mreže:

- `small maze`: majhen labirint brez ciklov, na njem je še posebej nazorno prikazano delovanje algoritmov IDA* in RBFS
- `large maze`: nekoliko večji labirint s cikli in večimi potmi do cilja, primeren na primer za prikaz delovanja iskanja v globino
- `yin-yang`: konfiguracija, kjer sta začetek in cilj precej blizu, pa se zaradi njenih lastnosti večina algoritmov odreže zelo slabo (časovno gledano)
- `empty space`: konfiguracija, ki s prostora izbriše vse zidove

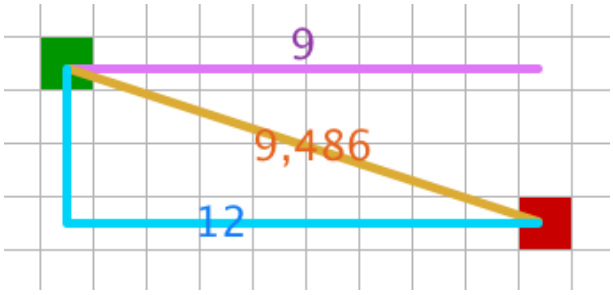
Vse konfiguracije lahko uporabnik seveda sam spreminja s premikanjem začetka in cilja ter dodajanjem in odvzemanjem zidov.

6.1.4 Hevristične funkcije

Vključene so štiri različne hevristične funkcije za ocenjevanje vozlišč, ki jih informirani algoritmi uporabljajo za usmerjanje preiskovanja:

- `Manhattan distance`: manhattanska razdalja, vsota absolutnih razlik koordinat ocenjevanega vozlišča od ciljnega vozlišča
- `euclidean distance`: evklidska razdalja, kvadratni koren vsote kvadratov razlik koordinat ocenjevanjega vozlišča od ciljnega vozlišča
- `Chebyshev distance`: večja od obeh absolutnih razlik po koordinatah
- `none`: hevristična funkcija, ki vedno vrne oceno 0

Povedano drugače, absolutno razliko koordinat vozlišča in cilja lahko vzamemo kot vektor. Manhattanska razdalja je prva norma tega vektorja, evklidska razdalja druga norma, tretja možnost pa je njegova neskončna oziroma maksimalna norma [9]. Skicirane so na sliki 6.3.



Slika 6.3: Manhattanska razdalja (modra), evklidska razdalja (rumena) in neskončna norma (roza)

Omeniti je treba, da ob kombinaciji hevristične funkcije manhattanske razdalje in dovoljenega gibanja po diagonali pogoj za popolnost algoritmov A*, RBFS in IDA* ni izpolnjen in zato ni zagotovila, da bodo te našli optimalno pot.

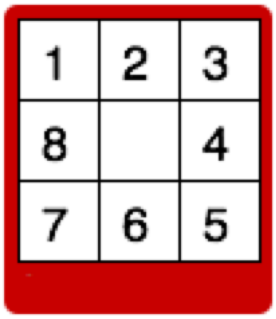
6.2 Igra osmih kvadratov

Igra osmih kvadratov temelji na igrači, kjer so s številkami označene ploščice vpete v okvir in se jih da premikati z drsenjem. Na mreži ploščic je manjka natanko ena, na prazen prostor pa se lahko premakne katerokoli od njenih sosed (zgornja, spodnja, leva ali desna). Cilj igre je z ustreznim zaporedjem premikov ploščic doseči stanje, ko so te urejene po vrstnem redu (točno tako kot na sliki 6.4).

Gre za klasičen način predstavitve abstraktnega prostora stanj, zato je vključena tudi v to aplikacijo.

6.2.1 Lastnosti

Prostor stanj igre osmih kvadratov ni neskočen (vsebuje natanko 362 880 različnih vozlišč), a v njem vseeno obstajajo cikli. Njegova stanja so vse možne konfiguracije igre, prehodi pa vsi možni premiki v posameznem stanju. Te si lahko zamišljamo kot premike praznega prostorčka gor, dol, levo ali



Slika 6.4: Rešena igra osmih kvadratov, oziroma ciljno vozlišče

desno, vendar pa si velja zapomniti, da ne gre za evklidski prostor. Medtem ko nas na mreži drugačno zaporedje enakih premikov vedno pripelje v isto vozlišče, pa v igri osmih kvadratov to še zdaleč ni res, primer je ilustriran na 6.5.

Cena vsake povezave je kar 1.

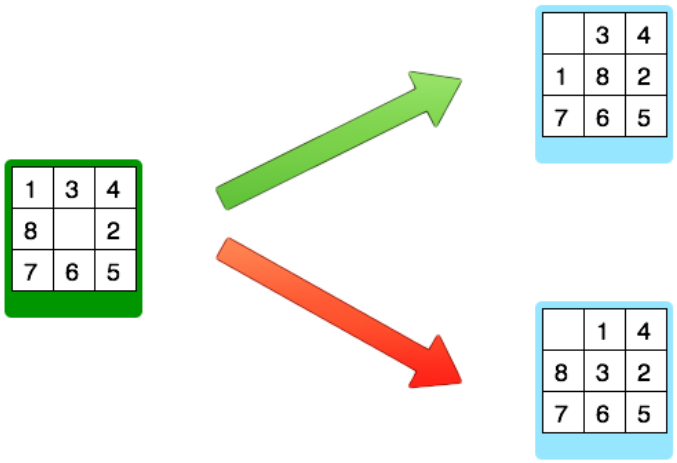
6.2.2 Vključene konfiguracije

Prostor igre osmih kvadratov je zelo nefliksibilen, saj ga točno določajo pravila igre. Po literaturi [1] so vključene 4 prednastavljene začetne konfiguracije, ki so poimenovane po svoji najkrajši oddaljenosti do ciljnega stanja (v številu korakov).

6.2.3 Hevristične funkcije

Za prostor igre osmih kvadratov so vključene tri hevristične funkcije: [1]

- `Manhattan distance`: podobno kot pri mreži, vsota oddaljenosti vsake ploščice na trenutni konfiguraciji do svojega ciljnega položaja na ciljni konfiguraciji
- `Manhattan distance + sequential score (man+seq)`: prejšnji funkciji prištet trikratnik "vrstne ocene", ki se računa na podlagi vrstnega



Slika 6.5: Premik levo-in-gor (zelena) vodi do drugačnega stanja kot premik gor-in-levo (rdeča)

reda ploščic; za vsako ploščico, ki nima pravega soseda v smeri urinega kazalca se pribije ena dodatna točka

- `none`: hevristična funkcija, ki vedno vrne oceno 0

Z razliko od ostalih dveh funkcij, vsota Manhattanske razdalje in vrstne ocene ne izpolnjuje pogoja za popolnost A*, vendar pa v splošnem zelo hitro najde rešitev.

6.2.4 Izzivi ob vizualizaciji

Kot rečeno, je igra osmih kvadratov klasičen prostor stanj v umetni inteligenci in ga zasledimo v precej literature, a je tam ilustriran z zgolj nekaj razvitimi stanji. V tej aplikaciji pa je zaželjeno dinamično tvoriti sliko celotnega preiskanega prostora, kar se je izkazalo za zelo težak problem.

Prvi problem predstavlja velika rast drevesa, ki ga je težko v celoti prikazati na zaslonu, hkrati pa je v splošnem na vsakem novem nivoju večje število

vozlišč kot na prejšnjih. Tu se pojavi drug problem – ali naj se ob tem višji nivoji prilagodijo in centrirajo, ali pa naj ostane njihov položaj fiksen? Ker bi prva možnost pomenila stalno premikanje vozlišč, ki jih algoritem v danem koraku ni obravnaval, je bila uporabljena druga.

Izbira fiksnega položaja vozlišč pa je pripeljala do tretje težave: kako risati povezave? Če vozlišča niso poravnana pod svoje starše na grafu pride do zmede povezav, ki se križajo in niso pregledne. Kot kompromis se izrišejo povezave le po koncu preiskovanja in le na rešitveni poti.

Poglavje 7

Zgradba in arhitektura aplikacije

Vizualizator je zgrajen kot spletna aplikacija, ki teče v spletnem brskalniku uporabnika. Da bi bila zagotovljena čim večja združljivost z napravami danes in v prihodnje za delovanje uporablja samo standardne spletne tehnologije in ne uporablja dodatnih vtičnikov za brskalnik.

7.1 Uporabljene tehnologije

Uporabnikov brskalnik mora podpirati standard HTML5 [11] in programski jezik JavaScript [12], za pomoč pri izrisovanju pa je v program vključena odprtokodna programska knjižnica EaselJS [10].

7.1.1 HTML5 in ⟨canvas⟩

V času pisanja najnovejša različica spletnega jezika HTML se imenuje HTML5. Dočim so bile njene predhodnice namenjene večinoma opisovanju dokumentov (statičnih spletnih strani) pa je ta namenjena predvsem dinamičnim spletnim aplikacijam. V jezik prinaša širok nabor novosti, med drugim večnitno delovanje, multimedijske elemente, uporabo strojnega pospeševanja, dostop naprav in več.

Čeprav standard HTML5 uvaja na desetine novih elementov, pa vizualizacijska aplikacija uporablja le enega – tako imenovan element ⟨canvas⟩ ali platno. Platno je element spletne strani, ki kaže dinamično tvorjeno sliko, se pravi sliko, ki jo tvori programska koda (v nasprotju od ostalih elementov, katerih vsebina je v splošnem določena vnaprej). V primeru te aplikacije je platno edini uporabljen spletni element (poleg plavajočega nadzornega okna) in služi izrisu celotnega vmesnika.

Platno je sicer zelo hiter in učinkovit element, vendar pa samo po sebi nudi zgolj primitivne metode za manipulacijo pikslov na sebi kar visokonivojsko programiranje precej otežuje. Rešitev je uporaba dodatnih programskih knjižnic.

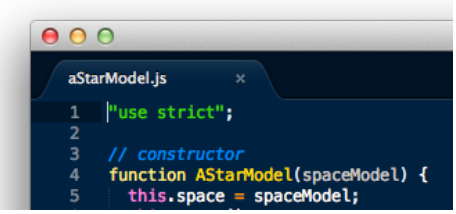
7.1.2 JavaScript

JavaScript je visokonivojski interpretiran in dinamično tipiziran programski jezik, ki sledi večim paradigmam: je objektno usmerjen (vendar ne pozna koncepta razredov) in omogoča programiranje tako v imperativnem kot v funkcionalnem slogu.

Med danes uporabljanimi spletnimi brskalniki je široko zastopan in podprt ter hkrati edini jezik, ki ga podpira večina (zato je tudi ta vizualizator napisan v njem). Vsestranskost in vseprisotnost JavaScripta pa je privedla do tega, da zanj obstaja mnogo implementacij in interpreterjev, medsebojna konkurenčnost med njimi pa do tega, da je ta programski jezik danes med najhitrejšimi v razredu interpretiranih dinamično tipiziranimih jezikov.

Dedovanje s prototipi

Čeprav je JavaScript objektno usmerjen jezik, pa ne pozna koncepta razredov ali kakršnihkoli statičnih struktur. Namesto tega uporablja sistem prototipov: vsak objekt nosi referenco na svoj prototipni objekt in tako deduje njegove metode, atribute in lastnosti. Referenca se lahko dinamično spreminja, prototipni objekt pa tudi. To po eni strani omogoča izgradnjo zelo



Slika 7.1: Za uporabo strogega načina v JavaScript je to potrebno na vrhu datoteke označiti.

unikatnih programskih arhitektur, po drugi pa lahko prototipov nevajenemu programerju delo tudi oteži.

Dedovanje s prototipi se uporablja tudi v tej aplikaciji in bralcu, ki jo želi spreminjati, je priporočano, da se s tem konceptom seznani v literaturi [13].

Strogi način

Ker je JavaScript zgrajen z enostavnostjo programiranja v mislih, je zelo fleksibilen: prevajalnik skuša samodejno odpraviti nekatere sintaktične in semantične napake ter dopušča določene dvoumnosti. Napačne stavke preskoči in nadaljuje z izvajanjem. Tako programiranje je morda sicer lažje za začetnika, a zelo oteži razhroščevanje, ki je pri razvoju velikih aplikacij kritičnega pomena.

Rešitev za to je strogi način [14] (angl. *strict mode*), v katerem se samodejno odpravljanje napak izklopi in ko interpreter nanje naleti, raje vrže napako in izvajanje konča. Strogi način se vklopi tako, da se na začetek vsake JavaScript datoteke zapiše ukaz `‘‘use strict’’;` (kot na sliki 7.1).

7.1.3 EaselJS

EaselJS je knjižnica (del paketa knjižnic CreateJS) in programsko ogrodje, ki zgoraj opisani element platna nadgradi z visokonivojskimi abstrakcijami in omogoča manipulacijo slike na ravni grafičnih objektov (likov, besedila, množic, vektorjev, rastrskih slik in tako naprej) namesto na ravni pikslov.

```
// izris rdečega kvadrata v HTML5
var canvas  = document.getElementById("canvas");
var context = canvas.getContext("2d");
context.fillStyle = "#FF0000";
context.fillRect(10, 10, 100, 100);
```

```
// izris rdečega kvadrata v EaselJS
var stage = new createjs.Stage("canvas");
var shape = new createjs.Shape();
shape.graphics.beginFill("#FF0000").drawRect(10, 10, 100, 100);
stage.addChild(shape);
stage.update();
```

Slika 7.2: Primerjava: nižjenivojsko risanje naravnost platno (HTML5) in višjenivojsko risanje v objekt (EaselJS)

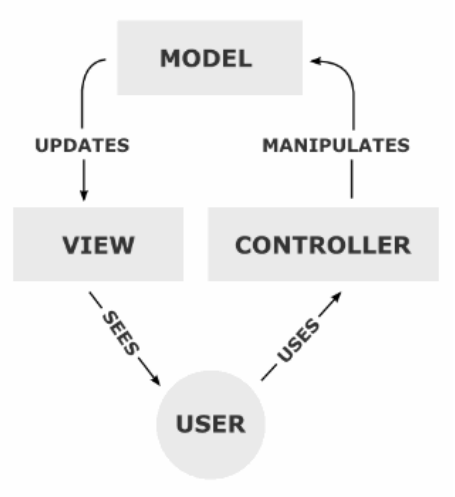
Napisan je v JavaScriptu in se ga vključi v kodo aplikacije (se pravi, ni vtičnik ali razširitev za brskalnik).

V EaselJS osrednji in najpomembnejši koncept predstavlja oder (angl. *stage*), objekt, ki razširja delovanje platna z interno hiearhijo grafičnih elementov in širokim naborom za manipulacijo in izris le teh. Elementi v hiearhiji imajo lastnosti, ki določajo njihov položaj in rotacijo v koordinatnem sistemu. Ta je relativen na njihove starševske elemente v hiearhiji. Ob klicu metode `update()` se oder sprehodi čez hiearhijo in programsko tvori sliko, ki jo nato izriše na platno (primer kode na sliki 7.2). Metoda je lahko klicana ročno ali v naprej nastavljenih intervalih.

Deli aplikacije, ki skrbijo za izris prostorov stanj so bili napisani v EaselJS, kar je znatno olajšalo njihovo implementacijo in omogočilo, da se je razvoj čim bolj osredotočil na vizualizacijo samo.

7.2 Arhitektura in delitev kode

Kot omenjeno v uvodu je bil eden glavnih ciljev pri razvoju aplikacije za vizualizacijo delovanja preiskovalnih algoritmov da bi ta bila čim bolj enostavno popravljiva, nadgradljiva in razširljiva. Da bi temu bilo karseda dobro zadoščeno, je bilo še pred programiranjem potrebno zastaviti programsko arhitekturo, ki posamezne dele kode smiselno ločuje glede na naloge, ki jih opravljajo (delitev dela, angl. *separation of concerns*) in hkrati spodbuja čim večjo generičnost pri pisanju teh delov, da se jih da ponovno uporabiti v čim več različnih kontekstih (reciklaža kode, angl. *code reusability*). Zato



Slika 7.3: Ideja delitve kode po vzorcu MVC [8]

je prišlo do odločitve za uporabo arhitekture, ki temelji na vzorcu "model-pogled-krmilnik" ali krajše, MVC [2] (angl. *model-view-controller*).

7.2.1 MVC na splošno

Arhitekturni vzorec MVC v grobem ločuje kodo programa v tri ločene kategorije:

- koda, ki dela s hranjenjem in obdelavo podatkov (modeli)
- koda, ki skrbi za prikaz podatkov v človeku berljivi obliki (pogledi)
- koda, ki sprejema uporabnikov vnos in pošilja ukaze ostalim delom in skrbi za njihovo medsebojno komunikacijo (krmilniki)

MVC ni strogo definiran, njegovi opisi in implementacije se precej razlikujejo. Pri vseh pa je poudarjena stroga ločitev delov kode glede na njihove naloge, ki medsebojno sicer tesno sodelujejo, a so vsak zase večinoma samostojni in neodvisni.

Velja omeniti, da MVC ne določa števila uporabljenih modelov, krmilnikov in pogledov. Na primer, v velikih projektih je lahko vseh treh ogromno. V

enostavnih vmesniških programih je lahko izpuščen model, v programih s tekstovnim ali neobstoječim vmesnikom pa izpuščen pogled. Končna odločitev je v rokah programerja.

Model

Model vsebuje glavno logiko programa in modelira delovanje nekega koncepta (v primeru te naloge, prostora stanj ali algoritma). Njegova naloga je hranjenje in manipulacija svojega notranjega stanja.

Običajno ima nabor javnih metod, ki usmerjajo njegovo delovanje, rezultat tega pa je na voljo v obliki izhodnih podatkov, dostopnih skozi prav tako javne podatkovne strukture ali metode. Hkrati je povsem neodvisen od izpisa, "ne zanima" ga ali so njegovi podatki predstavljeni grafično, v obliki besedila ali na primer datoteke.

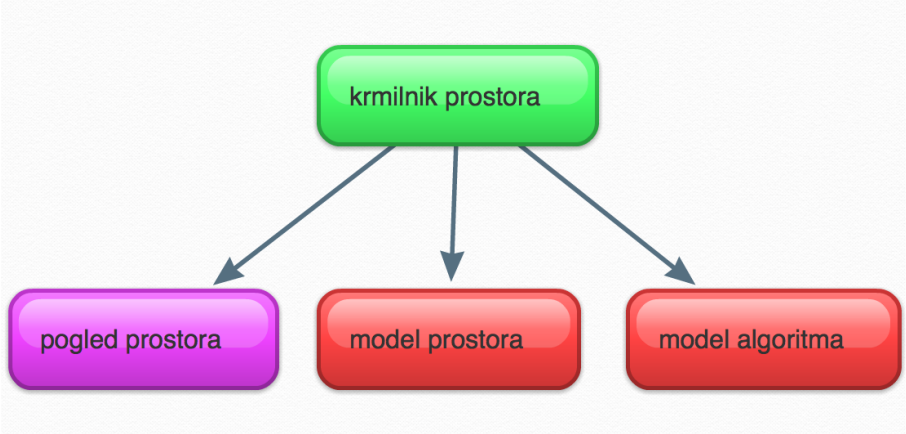
Pogled

V splošnem pogled ne izvaja nobene obdelave podatkov in v večini MVC arhitektur teh tudi ne hrani. Njegova naloga je skoz prikaz in izris danih podatkov na nek vnaprej določen način. Vezan je na krmilnik, od katerega prejema vsebino in morebitne dodatne parametre.

Pogled je lahko hkrati tudi zadolžen za prejemanje vnosa uporabnika, a ob tem ne sprejema nobenih odločitev. Prejet vnos (recimo miških klik, pritisk na tipkovnico ali dotik na zaslon) enostavno pošlje krmilniku, ta pa ukrepa naprej.

Krmilnik

Krmilnikova naloga je voditi delovanje nanj priključenih pogledov, modelov podkrmilnikov ter omogočati njihovo medsebojno komunikacijo, istočasno se mora odzivati na uporabniški vnos (recimo s spremembo prikaza v pogledu ali sprožitvijo dejanja v modelu). Prebirati mora izhodne podatke modelov in jih po potrebno prevajati v obliko razumljivo pogledom. Skrbeti mora za vzpostavitev in morebitno zamenjavo svojih pogledov in modelov.



Slika 7.4: Krmilnik prostora ima dva priključena modela

Razen v izjemnih primerih je torej naloga krmilnikov zgolj krmiljenje.

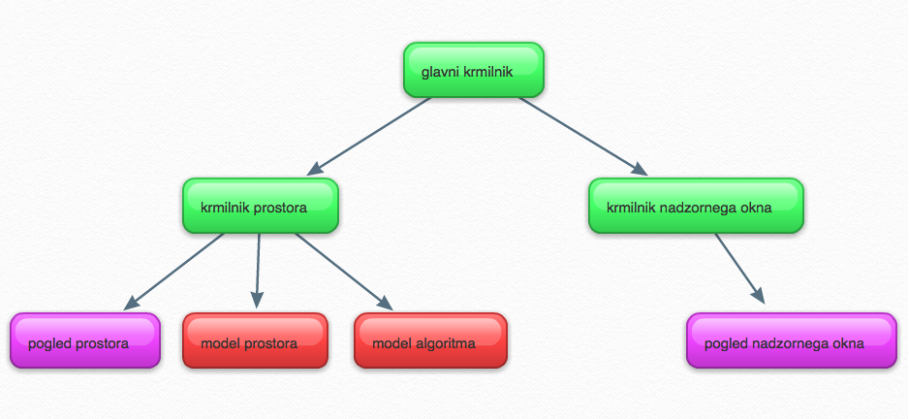
7.2.2 Hierarhija zgradbe aplikacije

Vizualizacijsko aplikacijo v grobem sestavljajo tri komponente:

- prostor stanj s prikazom
- preiskovalni algoritem
- vmesnik za nadzor programa

Za prostor stanj je potreben njegov model, ki hrani vse potrebne informacije in pa pogled, ki ga izrisuje. Povezovati ju mora krmilnik. Preiskovalni algoritem je zamenljiv, torej mora biti njegova logika shranjena v lastnem ločenem modelu. Po drugi strani se izrisuje na vizualiziran prostor stanj, torej nima smisla uporabljati svojega lastnega pogleda. Priključen je lahko kar na krmilnik prostora (predstavljeno na sliki 7.4).

A ker aplikacija omogoča preklapljanje tudi med prostori stanj, je potrebno vpeljati še statičen del programa, ki vsebuje mehanizem za to: glavni krmilnik.



Slika 7.5: Arhitektura celotne aplikacije

V to arhitekturo je končno potrebno vključiti še nadzorno okno, s katerim uporabnik vodi program. Njegova vsebina je statična, zato uporaba modela ni smiselna, potrebuje pa pogled, ki skrbi za izris. Poleg tega rabi svoj krmilnik, ki ta pogled vzpostavi in se odziva na uporabnikov vnos. Ker je nadzorno okno statičen del programa, je priključen kar na glavni krmilnik. Celotna arhitektura aplikacije je prikazana na sliki 7.5.

Tekom delovanja programa v pomnilniku torej biva 7 programskih modulov hkrati, a je število vseh, ki tvorijo vizualizacijsko aplikacijo višje:

- vsak od dveh prostorov ima svoj model, pogled in krmilnik, kar nanese skupaj 6 modulov
- vsak od implementiranih algoritmov ima svoj model, teh je 7
- statični moduli programa so trije

Celotno aplikacijo torej tvori 16 različnih programskih modulov.

7.2.3 MVC prostora stanj in preiskovalnega algoritma

Posamezni prostori stanj imajo zelo različne lastnosti; prostora stanj igre osmih kvadratov na primer se ne da na noben način prikazat kot dvodimenzionalni evklidski prostor. Iz tega izhaja, da se vizualizacije/pogleda ne

da narediti generično za poljuben prostor stanj (oziroma to ni smiselno) in posledično ima vsak prostor lasten pogled, krmilnik in model.

Algoritmi so po drugi strani lahko zaradi svoje vsestranskosti implementirani povsem neodvisno od prostora stanj (to se pravi, ni jih potrebno napisati za vsak prostor posebej).

Pogled prostora

Pogled izrisuje trenutno stanje modela prostora kot tudi stanje v modelu algoritma. Gre za objekt, ki prototipno deduje od objekta `Container` v knjižnici EaselJS. `Container` je sam po sebi neviden objekt, ki pa vsebuje svojo lastno hiearhijo grafičnih elementov.

Kot vhodni podatek prejme množico vozlišč, ki jim je algoritemski model določil trenutno stanje. Za vsakega izmed vozlišč je tvorjen pripadajoč grafični element (kvadratek v mreži, razporeditev številk v igri osmih kvadratov) in glede na vsebino postavljen na nek položaj v pogledu.

Uporabnik lahko s klikom ali dotikom na izrisano vizualizacijo spreminja prostor stanj ali njegov prikaz, zato pogled prostora vsebuje poslušalce: metode, ki ob uporabniškem vnosu o tem obveščajo krmilnik.

Model prostora

V modelu prostora je shranjena celotna logika prostora stanj. Vsebuje kaj je začetno in kaj končno vozlišče in metodo, ki tvori naslednike danega vozlišča.

Prav tako vsebuje trenutno konfiguracijo prostora, ki določa uporabljeno hevristično funkcijo za ocenjevanje vozlišč glede na njihovo oddaljenost od cilja in morebitne ostale stvari, ki so lastne tej vrsti prostora stanj (na primer zidovi na mreži).

Zaradi lažje implementacije modelov preiskovalnih algoritmov pa arhitektura od modela prostora pričakuje še dve nalogi. Prva služi primerjanju dveh vozlišč; algoritem, ki je povsem generičen, namreč o vozliščih razen ocene ne ve nič in tega ne more početi, zato za primerjavo opira na metodo prostora stanj.

Algoritmi obiskana in označena vozlišča hranijo v množicah, v katerih se do njihovih elementov (vozlišč) dostopa kar najbolj učinkovito. To je lažje narediti, če lahko o vozliščih predpostavimo določene lastnosti, zato takšna množica ni generična in jo nudi model prostora.

Model algoritma

V modelu algoritma morajo biti implementirane štiri metode:

- metoda za korak naprej, ki opravi korak algoritma
- metoda za korak nazaj, ki ta korak razveljavi
- metoda, ki algoritem vrne v začetno stanje (ponastavitev)
- metoda, ki vrne trenutni izpis algoritma (prikazan v nadzornem oknu)

Za vizualizacijske namene pa morajo biti javno dostopne tudi množice vozlišč (za vsako vrsto oznake po ena), do katerih dostopa krmilnik prostora in jih predaja pogledu prostora.

Preiskovalni algoritmi generični in njihovo delovanje ni pogojeno s prostorom, ki ga preiskujejo. Algoritmi z modelom prostora komunicirajo in sodelujejo direktno preko vnaprej določenega vmesnika (nabora metod), ki mora biti enak vsem implementacijam prostora stanj.

Krmilnik prostora

Ob vzpostavitvi krmilnika prostora mora ta vzpostaviti tudi svoj pogled, svoj model in model algoritma. Podati jim mora referenco nase in hkrati ohraniti referenco nanje (da lahko teče komunikacija).

Prva naloga krmilnika je, da se odziva na uporabnikov vnos, informacijo o katerem prejme od pogleda. Krmilnik prostora mreže, na primer, se mora na podlagi vrste vnosa odločiti ali gre za manipulacijo pogleda (premikanje mreže) ali prostora (risanje zidov, premikanje začetka in cilja). Odvisno od tega kliče metode v enem ali drugem, ki spremembo dejansko izvedejo.

Opomba: JavaScript klike z miško in dotike s prstom obravnava kot ločene dogodke. Ker je cilj programa, da deluje na napravah občutljivih na dotik, mora krmilnik implementirati dodatno logiko za odziv na dotike. To je hkrati edini del kode, ki ga tablični računalniki za poganjanje aplikacije potrebujejo dodatno.

Druga naloga krmilnika je, da vsebuje metodo, ki osveži pogled (vizualizacijo). Ob klicu ta metoda prebere trenutno stanje iz javnih podatkovnih struktur modelov prostora in algoritma ter ga prevede v primerno obliko prevedenega poda pogledu.

7.2.4 Nadzorno okno

Nadzorno okno je statičen del aplikacije in je tekom njegovega delovanja stalno prisoten. Ker ni smiselno govoriti o njegovem notranjem stanju, modela ne vsebuje, potrebuje pa pogled za izris in krmilnik, ki pogled vzpostavi in se odziva na uporabnikov vnos.

Pogled nadzornega okna

Pogled nadzornega okna z razliko od ostalih ne uporablja HTML5 platna oziroma knjižnice EaselJS. Razlog je v tem, da prikazuje pretežno gumbe in izbirnike, ki so zelo pogost element spletnih strani in bi jih bilo nesmiselno in potratno napisati posebej.

Uporablja torej navadne HTML elemente. Nosi referenco na spletni element, ki okno predstavlja in vsebuje metode, ki vanj dodajajo vsebino (besedilo, gumbe, izbirne menije) ter kodo, ki ob uporabnikovem kliku na to vsebino o tem obvesti svoj krmilnik.

Krmilnik nadzornega okna

Krmilnik nadzornega okna je zaradi enostavnosti napisan tako, da je vanj statično zapisana vsebina okna in jo ob vzpostavitvi samo poda pogledu - izjema so seznami prostorov stanj, algoritmov in hevrističnih funkcij za

katere pred tem prosi glavni krmilnik. Vsebuje še metodo, po kateri ga pogled obvešča o vnosu, krmilnik pa nato to sporoči ustreznemu naslovniku (o spremembi prostora stanj je na primer obveščen glavni krmilnik, o spremembi hevristične funkcije pa krmilnik prostora).

Odzivati pa se mora tudi na zunanja sporočila: ko je o tem obveščen, mora na primer v pogledu zamenjati izpis preiskovalnega algoritma, ob zamenjavi prostora stanj pa mora popraviti tiste dele pogleda, ki so nanj vezani.

7.2.5 Glavni krmilnik

Glavni krmilnik je statičen del programa, dodeljene pa so mu tri naloge:

- nalaganje in zamenjava prostora stanj
- omogočanje komunikacije med aktivnimi krmilniki aplikacije
- krmiljenje algoritma

Menjava prostora stanj

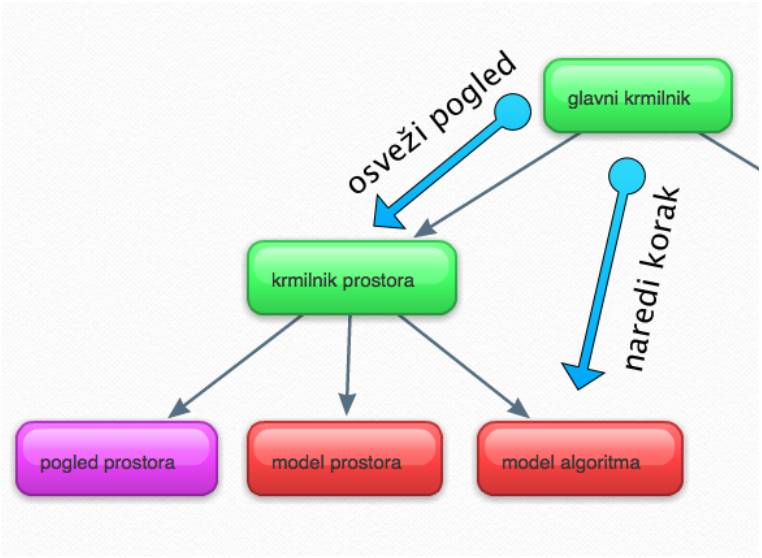
Mehanizem za menjavo prostora stanj je precej enostaven. Najprej krmilnik z odra (v EaselJS) odstrani vse grafične elemente trenutnega prostorskega pogleda, nato ustavi morebitno izvajanje algoritma in končno odstrani referenco na krmilnik prostora. Želeni nov prostor stanj naloži enostavno tako, da kliče njegovo konstruktorsko funkcijo.

Medkrmilniška komunikacija

Za medkrmilniško komunikacijo skrbi sistem sporočanja, katerega "center" je implementiran v glavnem krmilniku. Ta sistem je bolj podrobno opisan v naslednjem podpoglavju.

Krmiljenje algoritma

Model preiskovalnega algoritma je zaradi svoje tesne povezanosti delovanja z modelom prostora priključen kar na njegov krmilnik. Kljub vsemu pa je

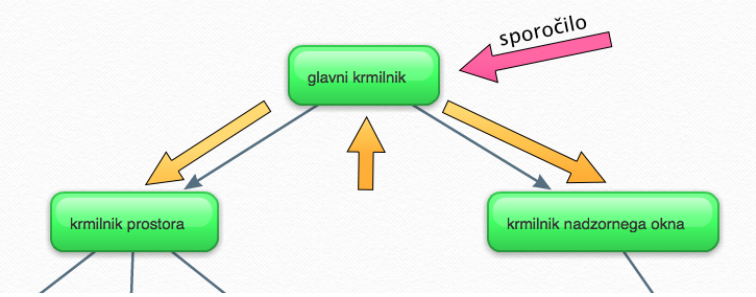


Slika 7.6: Krmilnik algoritma najprej sproži spremembo v modelu, nato pa naroči osvežitev pogleda

nekam potrebno dati mehanizem, ki skrbi za samodejno izvajanje algoritma oziroma njegovo izvajanje po korakih. Ta je za vse prostore in vse algoritme enak in ga torej ni smiselno vključiti v krmilnik vsakega prostora; prišlo bi do nepotrebnega podvajanja kode.

Krmilnik algoritma je zato kar del glavnega krmilnika. S tem je sicer malenkost kršeno načelo strogega ločevanja kode, vendar ni zaradi tega nobene škode. Skrbi za menjavo med algoritmi in samo izvajanje algoritma (tako po korakih kot samodejno).

Odziva se na sporočila, ki jih prejme od nadzornega okna (tam pa jih sproži uporabnik) in na podlagi njih ukrepa. Ob menjavi algoritma samo zamenja referenco nanj v krmilniku prostora. Če dobi prejme ukaz za en sam korak naprej ali nazaj, kliče za to ustrezno metodo v modelu algoritma, nato pa še metodo za osvežitev pogleda v krmilniku prostora (kot na sliki 7.6). Pri samodejnem izvajanju se isto dogaja v zanki, vendar je ta osveževanje pogleda omejeno na 25 Hz (zaradi večje učinkovitosti).



Slika 7.7: Poslano sporočilo prejmejo vsi krmilniki, vključno s pošiljateljem in glavnim

7.2.6 Sistem sporočanja

Vzorec "model-pogled-krmilnik" jasno določa delitev kode na posamezne module. Te moduli morajo za delovanje kot celota občasno komunicirati in si izmenjevati informacije, o čemer pa MVC ne določa ničesar. Rešitev v tej aplikaciji je sistem za medkrmilniško sporočanje.

Opis

Gre za enostaven koncept, ki znatno poenostavi določene stvari. Ideja je v tem, da lahko vsak krmilnik pošlje sporočilo, ki ga prejmejo vsi preostali krmilniki (slika 7.7), te ga lahko ignorirajo ali pa nanj reagirajo in morda celo vrnejo odgovor. Vsako sporočilo ima ime (ki označuje vrsto), lahko pa tudi dodatno vsebino. V aplikaciji za vizualizacijo je namen sistema sporočil komunikacija med nadzornim oknom in prostorom stanj/algoritmom ter večja nadgradljivost same aplikacije.

Implementacija

V kodi je sistem napisan tako, da glavni krmilnik vsebuje metodo za pošiljanje sporočil (`sendMessage`), ki v vsakem izmed priključenih krmilnikov kliče metodo za sprejemanje (`getMessage`).

Uporaba

Sistem sporočanja je uporabljen v naslednjih primerih:

- Ob svoji vzpostavitvi in ob menjavi prostora nadzorno okno pošlje sporočilo s poizvedbo o hevrističnih funkcijah prostora in ta odgovori s seznamom;
- Ob pritisku gumba ali izbirnika v oknu njegov krmilnik pošlje sporočilo, na katerega se odzove ali glavni krmilnik (upravljanje z algoritmom) ali pa krmilnik prostora (upravljanje s prostorom);
- Ko algoritem opravi en korak izvajanja, njegov krmilnik nadzornemu oknu pošlje sporočilo z zadnjim izpisom modela algoritma;

Prednosti in slabosti

Sporočila so enostavna ideja, ki ima kar nekaj prednosti, med drugim lahko veliko število preprostih metod nadomestimo z eno samo, klicatelja oziroma pošiljatelja sporočila pa dejansko ne zanima, kdo nanje odgovarja (torej kje v arhitekturi se mora nekaj zgoditi). Posledično so deli aplikacije med seboj bolj neodvisni, kar pomeni lažje programiranje in večjo popravljivost oziroma razširljivost.

Slabost sporočil je v tem, da sam sistem terja nek manjši pribitek pri izvajalnem času in zaradi tega na primer ni primeren za komunikacijo med modelom prostora in modelom algoritma, kjer je učinkovitost bistvenega pomena in se bolj splača klicati javne metode direktno.

Poglavje 8

Sklepne ugotovive

8.1 Analiza delovanja programa

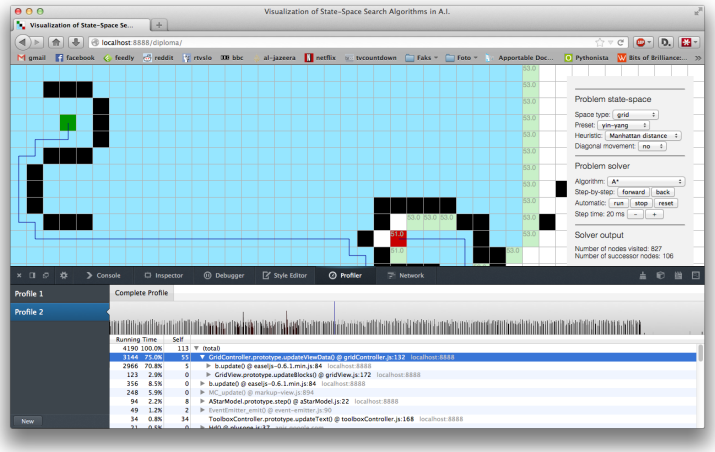
Spletni brskalnik Firefox [15] ima vgrajena številna orodja, ki močno olajšajo razvoj spletnih aplikacij, najpomembnejša med njimi sta razhroščevalnik (ki omogoča izvajanje kode po korakih) in pa profilnik, ki analizira delovanje aplikacije v smislu časa, ki ga vsaka izmed klicanih funkcij ali metod porabi (slika 8.1).

Napisana aplikacija je bila analizirana, da bi se ugotovilo, kateri njeni deli so računsko najbolj zahtevni. Izkazalo se je, da izrisovanje porabi daleč največ časa:

- `izris`: 95-97% procesorskega časa, od tega 15% sestavljanja "plana za risanje" (koncept v EaselJS) in preostalih 85% dejanskega risanja v HTML5

- `algoritem`: 2-3% procesorskega časa

Preostali čas je porabljen v krmilnikih, vendar gre za zanemarljiv delež, manjši od odstotka. Če bi želeli izboljšati učinkovitost, bi morali za najboljše rezultate poseči v delovanje pogledov.



Slika 8.1: Firefoxov profilnik delovanje programa analizira in rezultate predstavi v drevesni obliki

8.2 Ideje za izboljšave

Aplikacija je bila razvita z nadgradljivostjo v mislih, saj se jo gotovo da narediti še veliko bolj uporabno, hitrejšo in morda uporabniku še bolj prijazno. Različica narejena v okviru te diplomske naloge je bila zgrajena v omejenem času in z omejenim znanjem. Za dodatno pomoč pri razširjanju služita dodatka, ki sledita temu poglavju, tukaj pa je navedenih nekaj idej:

Optimizacija delovanja

Kot kaže prvi del tega poglavja, aplikacija ob izvajanju preiskovalnih algoritmov veliko večino procesorskega časa porabi za vizualizacijo samo, pri čemer daleč največji pribitek nosi izrisovanje (pri 25 slikah na sekundo precej zahtevna naloga), manjšega pa dejstvo, da so preiskovalni algoritmi za vizualizacijo prilagojeni in uporabljajo dodatne podatkovne strukture.

Sam nimam velikega znanja o učinkovitem izrisovanju grafike, zato pri moji implementaciji nedvomno marsikaj da izboljšati, algoritme pa hkrati zapisati na boljši oziroma bolj učinkovit način. Po drugi strani pa aplikacija

že sedaj teče precej dobro – merilo za tekoče delovanje je ob razvoju bil tablični računalnik iPad, računalniki v prihodnosti pa bodo zgolj hitrejši, zato morda optimizacija sama ni tako ključnega pomena.

Izboljšava prikaza igre osmih kvadratov

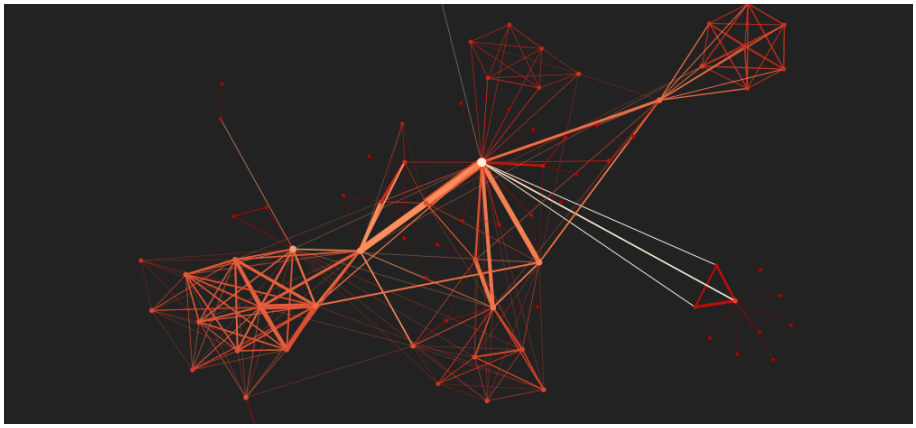
Kot se je izkazalo, je predstaviti prostor stanj igre osmih kvadratov precej težko, še posebej, če vizualizacija "ne ve", kako se bo preiskovanje odvijalo. Dobra vizualizacija bi potrebovala zares inovativno idejo, ki bi rešila ali ublažila težave ob "velikem prostoru" z inteligentnim izrisovanjem drevesa stanj.

Dodatni preiskovalni algoritmi in novi prostori stanj

V vizualizatorju je predstavljenih sedem preiskovalnih algoritmov, vendar pa jih v resnici obstaja in se jih tudi uporablja mnogo, mnogo več. Napisati nove je relativno enostavno, saj so hranjeni v lastnih modulih.

Nekoliko zahtevnejše je dodajanje novih prostorov stanj: potrebno si je dobro zamisliti vizualizacijo in jo ustrezno napisati. Morda najbolj zanimiv (in seveda zelo zahteven) podvig bi bil implementacija prostora stanj, kateremu bi lahko uporabnik določil poljubna stanja in prehode, vključno s poljubnimi hevrističnimi funkcijami. To bi aplikacijo naredilo ne le pedagoško, temveč tudi raziskovalno zanimivo.

Ideji sem tekom diplomske naloge posvečal precej časa, vendar je zaradi ogromne zahtevnosti nisem mogel izvesti. Prikaz poljubnega prostora stanj bi namreč zahteval zelo inteligentno knjižnico za izrisovanje grafov. Nekaj kandidatov bi lahko bilo: *Raphael.js* [16], *Sigma.js* [17] (predstavljen na sliki 8.2), *Processing.js* [18], *jit.js* [19] ali *D3.js* [20] (poimenovanje s končnico .js je za JavaScript knjižnice očitno precej priljubljeno).



Slika 8.2: Graf izrisan s funkcijami knjižnice Sigma.js

8.3 Ugotovitve

Tekom razvoja aplikacije se je platforma HTML5 izkazala za izredno dovršeno in zrelo. Omogoča programiranje spletnih aplikacij v visokonivojskem jeziku, ki zelo hitro in učinkovito tečejo na skorajda vsakem sodobnem osebnem računalniku, neodvisno od oblike, operacijskega sistema ali procesorske arhitekture.

Vizualizator sam pa je delno odgovoril na vprašanje, zakaj je tovrstnih izdelkov precej malo; služijo namreč zgolj majhnemu krogu ljudi (večinoma študentom računalništva) in jih je razmeroma zahtevno narediti. Algoritme je potrebno zastaviti drugače, za skoraj vsake prostor stanj pa je potrebna kreativna ideja, kako ga predstaviti.

Končni rezultat te diplomske naloge je izdelek, za katerega kot avtor upam, da bo koristil in pomagal vsaj manjšemu številu ljudi, ki se s preiskovalnimi algoritmi želijo seznaniti. Z njim sem zadovoljen - verjamem, da gre za stvar, ki že zdaj ni sama sebi namen, hkrati pa je lahko tudi odskočna deska za nekaj še večjega in še boljšega.

Dodatek A

Dokumentacija kode

Celotna izvorna koda (ki je hkrati tudi celotna aplikacija) je na voljo na tem spletnem repozitoriju: *https://bitbucket.org/matjazh/diploma-bsc-thesis*

Ta dodatek vsebuje seznam vseh delov programa po datotekah in na kratko opisuje delovanje in nalogo vsakega. Podrobna razlaga delovanja se nahaja v kodi sami, ki je dobro komentirana (v angleščini).

Korenski imenik (/)

V korenskem imeniku se nahajata dve datoteki in en podimenik. To je edini del programa, ki ni napisan v JavaScriptu.

- `index.html`: HTML dokument, katerega edina naloga je, da naloži vsebuje referenco na vse dele programa (*.js* datoteke), nastavi naslov in ikono strani in vsebuje statični HTML element (platno)
- `icon.png`: ikona strani, ki je prikazana zraven naslova v spletnem brskalniku
- `js/`: imenik, ki vsebuje vso nadaljno kodo programa

JavaScript koren (/js/)

V korenu imenika /js/ se nahajajo datoteke, ki niso del vzorca MVC in podimeniki, ki vsebujejo MVC elemente glede na kategorijo.

- `main.js`: vsebuje začetno funkcijo, ki naloži MVC arhitekturo in njen glavni krmilnik
- `easeljs-0.6.1.min.js`: vsebuje v času pisanja najnovejšo različico knjižnice EaselJS
- `data.js`: vsebuje konstantne podatke, ki se jih ne da uvrstit v preostale dele programa
- `prototypes.js`: vsebuje razširitvene metode za obstoječe standardne JavaScript in EaselJS objekte
- `model/`: imenik, ki vsebuje vse modele programa
- `view/`: imenik, ki vsebuje vse poglede programa
- `controller/`: imenik, ki vsebuje vse krmilnike programa

Krmilniki (/js/controller/)

Vsi krmilniki so shranjeni v imeniku /js/controller/. Njihovo delovanje je podrobno opisano v poglavju o arhitekturi aplikacije.

- `mainController.js`: glavni krmilnik programa
- `toolboxController.js`: krmilnik nadzornega okna
- `gridController.js`: krmilnik mreže
- `puzzleController.js`: krmilnik igre osmih kvadratov

Pogledi (/js/view/)

Vsi pogledi so shranjeni v imeniku /js/view/. Njihovo delovanje je podrobno opisano v poglavju o arhitekturi aplikacije.

- `toolboxView.js`: pogled, ki izrisuje nadzorno okno
- `gridView.js`: pogled, ki izrisuje mrežo

- `puzzleView.js`: pogled, ki izrisuje igro osmih kvadratov

Modeli (/js/model/)

Vsi modeli so shranjeni v imeniku /js/model/. Njihovo delovanje je podrobno opisano v poglavju o arhitekturi aplikacije in v pripadajočih poglavji o algoritmih oziroma prostorih stanj.

- `gridModel.js`: model, ki vsebuje stanje mreže
- `puzzleModel.js`: model, ki vsebuje stanje igre osmih kvadratov
- `DFSModel.js`: model, ki vsebuje kodo iskanja v globino
- `BFSModel.js`: model, ki vsebuje kodo iskanja v širino
- `IDModel.js`: model, ki vsebuje kodo iterativnega poglabljanja
- `aStarModel.js`: model, ki vsebuje kodo algoritma A*
- `IDAStarModel.js`: model, ki vsebuje kodo algoritma IDA*
- `RBFSModel.js`: model, ki vsebuje kodo algoritma RBFS
- `RTAModel.js`: model, ki vsebuje kodo algoritma RTA*

Dodatek B

Razširjanje programa

V tem dodatku je na kratko orisan postopek pisanja dodatnih modulov za vizualizacijsko aplikacijo. Ker je popolnoma odprtokodna, se jo seveda da razširjati na poljuben način. Tukaj sta opisana zgolj dva najverjetnejša scenarija, kjer morajo razširitve komunicirati s preostalo arhitekturo aplikacije in zato uporabljati ustrezen nabor metod oziroma vmesnikov.

Navedene so samo potrebne lastnosti in javne metode modulov, za podrobnosti implementacije pa je bralcu priporočeno, da si pogleda programsko kodo že napisanih.

Za vse razširitvene programske module velja, da jih je treba postaviti v ustrezni imenik glede na nalogo (glej *Dokumentacija kode*) in jih vključiti v seznam datotek v `index.html`.

Implementacija preiskovalnega algoritma

Ob dodajanju novega preiskovalnega algoritma je potrebno vključiti samo eno dodatno datoteko – algoritmov model. Konstruktor modela dobi podano referenco na model prostora, do metod katerega dostopa med svojim delovanjem. Na voljo mora imeti naslednje javne spremenljivke in metode:

- `start, goal, visited, successors, forgotten, path` so javne spremenljivke (množice), ki vsebujejo sezname vozlišč glede na ustrezno ka-

tegorijo; posamezno vozlišče je lahko v večih množicah, te pa krmilnik prostora uporabi za vizualizacijo

- `step` in `stepBack` sta javni metodi, katerih naloga je narediti oziroma razveljaviti en korak algoritma
- `reset` je javna metoda, ki stanje modela algoritma vrne v začetno stanje
- `output` je javna metoda, ki izpis algoritma vrne kot seznam vrstic

Implementacija prostora stanj

Prostor stanj tvorijo trije moduli: model, pogled in krmilnik, ki skupaj tvorijo zaokroženo celoto. Model prostora mora nuditi javne metode, ki jih nato uporablja model preiskovalnega algoritma:

- `start` je referenca na začetno vozlišče v prostoru
- `isGoal` je metoda, ki kot argument prejme vozlišče in pove, ali je ciljno ali ne
- `successors` je metoda, ki sprejme vozlišče in tvori njegove naslednike, te pa vrne kot seznam; vsako izmed vrnjenih vozlišč mora imeti lastnosti knownValue in estimatedValue, ki vsebujeta njegovo globino oziroma njegovo hevristično oceno
- metoda `nodeSet` mora vrniti objekt, množico vozlišč, ki ima svoje metode `add, remove` in `contains`

Krmilnik ima manj zahtev, vsebovati mora le:

- referenco na model algoritma, poimenovano `algorithm` (preko nje komunicira glavni krmilnik)
- metodo `updateViewData`, ki prebere stanje priključenih modelov in jih poda svojemu pogledu

- metodo `getMessage`, ki sprejme sporočilo poslano vsem krmilnikom in se po potrebi nanj odzove

Pogled pa je lahko implementiran popolnoma poljubno, ker je njegovo delovanje odvisno samo od krmilnika.

Literatura

[1] I. Bratko. *Prolog Programming for Artificial Intelligence, Fourth Edition*. Addison-Wesley, 2012.

[2] I. Davis. "What are the benefits of MVC?", 9. december 2008
Dostopno na:
http://blog.iandavis.com/2008/12/09/what-are-the-benefits-of-mvc/

[3] R. E. Korf, "Real Time Heuristic Search", Artificial Intelligence, 1990, str. 189–211

[4] R. E. Korf. "Artificial Intelligence Search Algorithms", Computer Science Department, UCLA, 5. julij 1995

[5] R. E. Korf, S. Edelkamp. "The Branching Factor of Regular Search Spaces", American Association for Artificial Intelligence, 1998

[6] D. A. Thomsen. pathvisual (2010)
Dostopno na: *http://danamlund.dk/*

[7] X. Xu. Pathfinding.js (2012)
Dostopno na: *http://qiao.github.io/PathFinding.js/visual/*

[8] R. Frey. MVC Process (2010)
Dostopno na:
http://en.wikipedia.org/wiki/File:MVC-Process.png

[9] Norm (mathematics) — Wikipedia
Dostopno na: *http://en.wikipedia.org/wiki/Norm_(mathematics)*

[10] EaselJS — A JavaScript library for making HTML5 Canvas easy
Dostopno na: *http://www.createjs.com/#!/EaselJS*

[11] HTML5 Working Group
Dostopno na: *http://www.w3.org/html/wg/*

[12] JavaScript — Mozilla Developer Network
Dostopno na:
https://developer.mozilla.org/en-US/docs/Web/JavaScript

[13] Prototype-base programming — Wikipedia
Dostopno na:
http://en.wikipedia.org/wiki/Prototype-based_programming

[14] Strict Mode — Mozilla Developer Network
Dostopno na:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode

[15] Mozilla Firefox
Dostopno na: *http://getfirefox.com*

[16] Raphael — JavaScript Library
Dostopno na: *http://raphaeljs.com/*

[17] sigma.js — A lightweight drawing library
Dostopno na: *http://sigmajs.org/*

[18] Processing.js
Dostopno na: *http://processingjs.org/*

[19] jit.js — JS InfoVist Toolkit
Dostopno na: *http://philogb.github.io/jit/*

[20] d3.js — Data-Driven Documents
Dostopno na: *http://d3js.org*