UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matjaž Verbole

Algoritem D*

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2012

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

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: 01859/2012

Datum: 03.09.2012

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

Kandidat: **MATJAŽ VERBOLE**

Naslov: **ALGORITEM D***

THE ALGORITHM D*

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Iskanje je osnovna metoda za reševanje problemov kombinatorične optimizacije pri navigaciji robotov. Cilj iskanja je v množici možnih poti od začetne do končne lokacije čim hitreje poiskati optimalno ali suboptimalno tako pot robota. Eden od možnih načinov preiskovanja je informirano iskanje, ki pri sestavljanju poti izkorišča že pridobljeno znanje dotedanjega iskanja. Predstavite in opišite algoritem D* za sestavljanje poti in njegove izpeljanke. Uvodoma opišite področje informiranih in neinformiranih preiskovalnih algoritmov. Algoritem D* in izpeljanke implementirajte v poljubnem programskem jeziku. Izvedite praktične meritve njihovih hitrosti.

Mentor:

prof. dr. Borut Robič

Dekan:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Matjaž Verbole,

z vpisno številko 63060307,

sem avtor diplomskega dela z naslovom:

Algoritem D*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Boruta Robiča
- 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 12.10.2012 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju prof. dr. Borutu Robiču za njegovo dosegljivost ter vsestransko strokovno in mentorsko podporo pri pripravi diplomske naloge.

Zahvaljujem se tudi mojima staršema, ki sta me podpirala ves čas mojega študija.

Kazalo

Povzetek **1**

Abstract **2**

1 Uvod **3**

2 Preiskovalni algoritmi **5**

2.1 Prostor stanj 5
2.2 Neinformirane tehnike 6
2.2.1 Iskanje v širino 7
2.2.2 Iskanje v globino 8
2.2.3 Razlike med BFS in DFS 8
2.2.4 Iterativno poglabljanje 9
2.3 Informirane tehnike 9
2.3.1 Kaj je hevristika 9
2.3.2 Algoritem najprej najboljši 11
2.3.3 Algoritem vzpenjanja 13
2.3.4 Algoritem A* 13
2.3.4.1 Popolnost algoritma A* 15
2.3.4.2 Optimalnost algoritma A* 15

3 Algoritem D* **18**

3.1 Inkrementalno iskanje 19
3.2 Ideja algoritma 19
3.3 Osnovni D* 20
3.3.1 Opis algoritma 21
3.3.2 Lastnosti algoritma 23
3.3.2.1 Pravilnost algoritma D* 25
3.3.2.2 Optimalnost algoritma D* 26
3.3.2.3 Popolnost algoritma D* 26

3.4 Focused D* 27
3.4.1 Motivacija za nadgradnjo osnovnega D* 27
3.4.2 Ideja algoritma Focused D* 28
3.4.3 Definicije in ostale lastnosti algoritma 29
3.4.4 Opis algoritma 30

4 Lifelong Planning A* **37**
4.1 Opis algoritma 37
4.2 Delovanje algoritma 38
4.3 Analitične lastnosti algoritma 42
4.3.1 Ustavljivost in pravilnost 42
4.3.2 Podobnost z A* 42
4.3.3 Učinkovitost 43

5 D* Lite **45**
5.1 Smer preiskovanja 45
5.2 Delovanje algoritma 46
5.3 Pohitritev algoritma 48
5.4 Izrek o ustavljivosti algoritma 51

6 Eksperimentalna primerjava algoritmov **52**
6.1 Svet v obliki mreže 52
6.2 Zaznavanje sveta s senzorji 53
6.3 Okolje za testiranje 54
6.4 Postopek testiranja 55
6.5 Rezultati 56

7 Zaključek **58**
7.1 Nadaljnje delo 59

Seznam slik **60**
Seznam tabel **61**

Literatura **62**

Seznam uporabljenih kratic in simbolov

- *NP* - Non-deterministic polynomial-time
- *BFS* - Breadth-first search (iskanje v širino)
- *DFS* - Depth-first search (iskanje v globino)
- *Iterative deepening* - Iterativno poglabljanje
- *Best-first search* - Najprej najboljši
- *Hill climbing* - Algoritem vzpenjanja
- *Brute force* - Groba sila
- *FD** - Focused D*
- *LPA** - Lifelong planning A*

Povzetek

Načrtovanje poti je pomemben del navigacije avtonomnih mobilnih robotov. Mobilni roboti pogosto delujejo v delno ali pa v celoti nepoznanem svetu. Pravzaprav bi bilo nepraktično, da bi moral robot v celoti poznati svet, preden bi se zapeljal po njem, saj bi moral pomniti zelo veliko količino podatkov. Res pa je tudi, da so zelo redki primeri, ko je informacija o okolju pred začetkom načrtovanja poti popolnoma neznana.

Leta 1994 je dr. Anthony Stentz odkril algoritem D*, ki združuje globalno in lokalno načrtovanje in omogoča učinkovito rabo informacije, ki jo dobimo iz že pripravljenega zemljevida okolja in iz robotovih senzorjev, ko se ta premika po prostoru. Algoritem je učinkovit, saj si tedaj, ko je treba poiskati novo optimalno pot (replanirati), pomaga z že izračunanimi informacijami, namesto da bi vse ponovno preračunaval.

V diplomskem delu smo si ogledali več različnih algoritmov in tehnik za iskanje optimalnih poti v grafih. Opisali smo algoritme, ki pri preiskovanju uporabljajo neinformirane tehnike (BFS, DFS in Iterativno poglabljanje), in tudi informirane tehnike, ki pri preiskovanju uporabljajo hevristične prijeme. Ker ima algoritem D* več različic, smo si tudi te pobliže ogledali. Tako smo poleg algoritma D* opisali tudi algoritma Focused D* in D* Lite. Algoritem D* Lite je izpeljan iz algoritma LPA*, zato smo opisali še tega.

Algoritme D*, Focused D*, D* Lite in LPA* smo implementirali v programskem jeziku Python, izmerili njihove hitrosti in rezultate meritev prikazali na koncu dela.

Ključne besede:

iskanje poti, hevristika, algoritem, D*, neznano okolje, replaniranje

Abstract

Path planning is an important part of the navigation of autonomous mobile robots. Mobile robots often operate in a partially or entirely unknown world. In fact, it would be impractical for a robot to fully know the world before driving through it, as the robot would have to memorise a large quantity of data. It is also true that cases when information about the environment is entirely unknown prior to planning the path are very rare.

In 1994 Dr Anthony Stentz discovered algorithm D*, which combines global and local planning, and enables efficient use of information obtained from the prepared map of the environment and from the robot's sensors as it moves around the area. The algorithm is effective, because when it needs to find a new optimal path (replan) it makes use of the already calculated information instead of recalculating everything.

The diploma thesis examined several algorithms and techniques for finding optimal paths in graphs. Algorithms have been described which use uninformed techniques (BFS, DFS, and Iterative Deepening) for exploring, as well as informed techniques which use heuristic approaches for exploring. Since algorithm D* has several versions, these were also given a closer look. Thus in addition to algorithm D*, algorithms Focused D* and D* Lite were described as well. The algorithm D* Lite has been derived from algorithm LPA*, hence the latter was described too.

Algorithms D*, Focused D*, D* Lite and LPA* were implemented in the Python programming language, their speeds measured, and the results of the measurements presented at the conclusion of the paper.

Key words:

path search, heuristic, algorithm, D*, unknown environment, replanning

Poglavje 1

Uvod

Iskanje je v računalništvu osnovna tehnika, ki se uporablja za reševanje računsko zahtevnih kombinatoričnih optimizacijskih problemov. Cilj iskanja je čim hitreje poiskati optimalno ali vsaj suboptimalno rešitev v končni ali neskončni množici potencialnih rešitev. Velikost prostora rešitev, ki ga mora pregledati iskalni algoritem, je pri NP-težkih problemih v najboljšem primeru eksponentna. Celo več, pri NP-težkih problemih mora algoritem tudi v povprečnem primeru preiskati eksponentno število potencialnih rešitev, da najde optimalno.

Preiskovalni algoritmi se delijo v dve skupini. V prvi skupini so tisti, ki najdejo optimalno rešitev. Takim algoritmom pravimo popolni (tudi natančni oz. eksaktni) algoritmi. V drugo skupino pa spadajo algoritmi, ki najdejo suboptimalno oziroma približno rešitev. Takšni algoritmi se imenujejo aproksimacijski algoritmi.

Probleme, ki jih bo reševal predstavljeni algoritem D*, se največkrat opiše s pomočjo grafa. Zato se bomo osredotočili le na algoritme, ki so sposobni preiskovati grafe. Preiskovati graf pomeni sistematično se sprehajati po njegovih povezavah od vozlišča do vozlišča. Največkrat nas bo zanimalo, ali obstaja pot od trenutnega do izbranega vozlišča. Kot bomo videli v nadaljevanju, se algoritmi kar precej razlikujejo po tehniki oziroma strategiji obiskovanja vozlišč.

Če se dotaknemo še robotike, lahko ugotovimo, da je tam eno izmed glavnih področij navigacija robotov po prostoru. Večina navigacijskih algoritmov predvideva, da ima robot že pred prvim premikom popoln in natančen model prostora. Zato lahko učinkovito preiščejo prostor in najdejo optimalno pot brez kakšnih večjih težav. Vendar v resničnem svetu to skoraj nikoli ne drži, saj se robot ponavadi nahaja v prostoru, za katerega ne obstaja natančen tloris ali zemljevid. Šele ko se robot premakne s trenutne lokacije na novo lokacijo,

lahko s svojimi senzorji zazna okolico in posodobi svojo sliko oziroma predstavo prostora.

Zato je treba v primerih, ko imamo opravka z raziskovalnim robotom, ubrati drugačne tehnike preiskovanja prostorov in iskanja optimalnih poti.

Algoritmi, ki imajo že pred začetkom iskanja poti na voljo popoln zemljevid prostora, za predstavitev tega sveta največkrat uporabijo graf s prostorom stanj. Nato brez ali s pomočjo hevristike poiščejo najcenejšo pot od začetnega položaja robota do cilja. Cena poti je lahko definirana kot prepotovana razdalja, potrošena energija, čas, v katerem je robot izpostavljen neki nevarnosti itd.

Algoritem D*, ki mu bomo v tem delu posvetili največ časa, je namenjen ravno iskanju poti v prostorih, v katerih se roboti največkrat znajdejo. Takšni prostori so lahko v celoti neznani, delno znani ali pa se skozi čas spreminjajo. Spoznali bomo, da algoritem D* rešuje takšne probleme na učinkovit in optimalen način.

Poglavje 2

Preiskovalni algoritmi

2.1 Prostor stanj

Prostor stanj je matematično predstavljen kot četverka [**N**,**A**,**S**,**GD**], kjer je:

- **N** množica stanj sistema. To ustreza množici vozlišč v grafu.
- **A** množica premikov v prostoru stanj. To ustreza posameznim korakom oziroma množici povezav v grafu.
- **S** neprazna podmnožica množice **N**. Vsebuje začetna stanja sistema.
- **GD** neprazna podmnožica množice **N**. V njej so končna stanja sistema.

Rešitvena pot v tem grafu je pot, ki gre od vozlišča v S do vozlišča v GD.

Na področju diskretnih dinamičnih sistemov je prostor stanj definiran kot usmerjen graf, kjer je vsako možno stanje danega sistema predstavljeno kot vozlišče [1]. Povezava v tem grafu ustreza možnemu prehodu med dvema stanjema sistema. Povezavam so lahko prirejene uteži, ki predstavljajo ceno prehoda iz enega stanja v drugo (npr. cestne razdalje). Funkcija f definira diskretni dinamični sistem V s preslikavo $f\colon V \to V$.

Ker bomo o grafih govorili skoraj ves čas, kar takoj navedimo nekaj definicij. Še več o teoriji grafov pa lahko bralec najde v [2] in v [3].

Definicija 2.1.1 (Graf)**.** Graf je trojica $G = (V(G), E(G), I_G)$, kjer je $V(G)$ neprazna množica in $E(G)$ množica, dobljena iz množice $V(G)$. Elementi množice $V(G)$ se imenujejo vozlišča grafa G, elementi množice $E(G)$ pa povezave grafa G. I_G je funkcija, ki vsaki povezavi priredi njeni krajišči.

Vozlišča *V(G)* grafa *G* predstavljajo stanja sistema. Ker pa sistemi lahko prehajajo iz enega stanja v drugo, predstavljajo povezave *E(G)* prehode v grafu *G*.

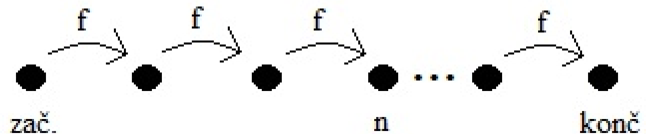
Definicija 2.1.2 (Povezava)**.** Med dvema vozliščema $a \in V$ in $b \in V$ obstaja usmerjena povezava, če in samo če obstaja tudi funkcija $f(a)=b$, kjer funkcija f definira diskretni dinamični sistem.

Definicija 2.1.3 (Usmerjenost)**.** Grafu $G=(V,E,I_a)$ pravimo, da je usmerjen, če je vsak element e $\in$ E urejen par vozlišč iz množice V.

Definicija 2.1.4 (Faktor razvejanosti)**.** Faktor razvejanosti predstavlja število otrok vsakega vozlišča.

Diskretni dinamični sistem se običajno nahaja v kakem *začetnem stanju* in s prehodi preide v kako *končno stanje*. Tema stanjema ustrezata *začetno* in *končno* vozlišče v ustreznem grafu.

Iskanje rešitve za dani problem v diskretnih dinamičnih sistemih je pogosto enako iskanju najcenejše poti v danem usmerjenem grafu med začetnim in ciljnim vozliščem. Rešitev problema je neka pot, torej zaporedja vozlišč grafa. Prehod iz enega stanja sistema v drugo se izvede tako, da se naredi *kompozicija* funkcije f: $konč = f(zač) = f(f(...f(zač)...))$ nad trenutnim stanjem sistema, ta pa nato izračuna novo stanje sistema.



Slika 2.1: Kompozicija funkcije f.

Zatem se ta postopek ponavlja, dokler kompozicija funkcije f ne izračuna iz trenutnega stanja končno stanje sistema. Z drugimi besedami lahko ponovimo, da je rešitev problema pot v usmerjenem grafu od začetnega do končnega (ciljnega) vozlišča.

2.2 Neinformirane tehnike

Ko preiskujemo nek prostor stanj (graf), si lahko, če premoremo poznavanje dane domene, pomagamo z njenimi značilnostmi in lastnostmi. Neinformirane

tehnike tega ne zmorejo in neodvisno od problema in domene določijo vrstni red obiskovanja vozlišč, z namero poiskati ciljno vozlišče. Poznamo tri takšne neinformirane tehnike preiskovanja [4] in zaradi boljšega razumevanja kasnejših algoritmov si jih podrobneje oglejmo.

2.2.1 Iskanje v širino

Iskanje v širino (Breadth-first search) za določitev naslednjega vozlišča, ki se bo obiskalo, uporablja podatkovno strukturo *vrsta*. Vsakokrat, ko algoritem obišče novo vozlišče v, se vsi njegovi sosedje dodajo na konec vrste. V usmerjenem grafu se na konec vrste dodajo le tisti sosedje, do katerih vodi usmerjena povezava iz trenutnega vozlišča. Za naslednje vozlišče se vzame vozlišče, ki je na začetku vrste in postopek se ponovi. Rezultat takega iskanja je, da se najprej razvijejo[1] vsa vozlišča, ki so od vozlišča v oddaljena i[2] korakov in šele nato vozlišča, ki so oddaljena $i + 1$ korakov. Ko trenutno vozlišče nima več nerazvitih otrok, ga ne moremo več razviti, zato se je treba vrniti korak nazaj in poskusiti po drugi poti (tj. z naslednjim bratom tega vozlišča). Vračanju za korak nazaj pravimo sestopanje.

Algoritem 1 Iskanje v širino

Vhod: Graf G in vozlišče $v \in G$
Izhod: Množica S vozlišč, ki so dosegljiva iz vozlišča v

```
1: function BFS(G, V)
2:     Množica S = {v}                        ▷ množica najdenih vozlišč
3:     Vrsta Q = vsi sosedje vozlišča v
4:     while Q ni prazna do
5:         odstrani v z začetka Q
6:         if v ni v S then
7:             dodaj v v S
8:             dodaj sosede v na konec Q
9:         end if
10:    end while
11:    return S
12: end function
```

[1] Razviti vozlišče pomeni postaviti njegove sosede v vrsto.

[2] Vozlišče u je od vozlišča v oddaljeno i korakov, če ima najkrajša usmerjena pot iz u do v i povezav.

2.2.2 Iskanje v globino

Iskanje v globino (Depth-first search) za določitev, katero vozlišče se bo obiskalo naslednje, uporablja podatkovno strukturo *sklad*. Vsakokrat, ko algoritem obišče novo vozlišče, se vsi njegovi sosedje dodajo na sklad. V usmerjenem grafu se na sklad dodajo le tista vozlišča, za katera obstaja povezava med trenutnim vozliščem in njegovimi sosedi, ki jih imenujemo otroci. Za naslednje vozlišče se določi vozlišče na skladu in postopek se ponovi. Rezultat tega je, da se najprej razvijejo vozlišča, ki so globlje v grafu.

Algoritem 2 Iskanje v globino

Vhod: Graf G in vozlišče $v \in G$
Izhod: Množica S vozlišč, ki so dosegljiva iz vozlišča v

```
1:  function DFS(G, V)
2:      Množica S = {v}                          ▷ množica najdenih vozlišč
3:      Sklad T = vsi sosedje vozlišča v
4:      while T ni prazen do
5:          pop v iz T
6:          if v ni v S then
7:              dodaj v v S
8:              push sosede od v na T
9:          end if
10:     end while
11:     return S
12: end function
```

2.2.3 Razlike med BFS in DFS

Čeprav sta si algoritma iskanja v širino in globino na prvi pogled precej podobna (razlikujeta se le v načinu pomnjenja obiskanih vozlišč in jemanja vozlišč iz podatkovne strukture), je med njima kar precej razlik. Nekaj jih je naštetih v nadaljevanju, še podrobnejši opis pa bralec najde v [5]:

- iskanje v širino zagotavlja optimalnost rešitve, če taka rešitev obstaja (najde najcenejšo pot od začetnega do končnega vozlišča, če ta obstaja),
- iskanje v globino ne zagotavlja optimalnosti rešitve,
- če je razvejanost možnih poti velika, je bolje uporabiti iskanje v globino, saj tedaj v pomnilniku ni treba hraniti vseh vozlišč na istem nivoju, ki so enako oddaljena od začetnega vozlišča,

- če že vnaprej vemo, da bo rešitev dolga pot, je bolje izbrati iskanje v globino,
- poraba pomnilnega prostora pri iskanju v širino je eksponentna: B^n, kjer je B faktor razvejanosti in število n nivo (oddaljenost od začetnega vozlišča),
- poraba pomnilnega prostora pri iskanju v globino je linearna: $B * n$, saj je na vsakem nivoju treba shraniti samo otroke trenutnega vozlišča.

2.2.4 Iterativno poglabljanje

Algoritem iterativno poglabljanje (iterative deepening) deluje podobno kot iskanje v globino. Pravzaprav je to iskanje v globino, ki mu kot argument podamo še, do katere oddaljenosti d naj preiskuje graf. Tako kot iskanje v globino tudi iterativno poglabljanje uporablja sklad. Vsakokrat, ko algoritem obišče novo vozlišče, se najprej vsem sosedom doda (kot atribut) še njihova oddaljenost (ki je za eno večja od oddaljenosti trenutnega vozlišča), nato pa se sosedje porinejo na sklad. V usmerjenem grafu se na sklad dodajo le tista vozlišča, za katera obstaja povezava med trenutnim vozliščem in njegovimi sosedi, ki jih imenujemo otroci. Za naslednje vozlišče se določi vozlišče na skladu in postopek se ponovi. Če je to vozlišče globlje od d, se s sklada vzame naslednje vozlišče. Rezultat tega je, da se razvijejo le vozlišča, ki so od začetnega vozlišča oddaljena največ d povezav.

2.3 Informirane tehnike

2.3.1 Kaj je hevristika

Cilj hevrističnega iskanja poti v grafu je zmanjšati število vozlišč, ki jih je pri preiskovanju grafa treba obiskati. To pomeni, da si pri reševanju nekega problema pomagamo z znanjem, informacijami, pravili, spoznanji, analogijami in poenostavitvami, ki jih za ta problem poznamo. Hevristika ne zagotavlja, da bomo rešitev vedno našli, če ta sploh obstaja. Splošna definicija hevristike ne obstaja, zato bomo navedli nekaj definicij, ki so jih navedli avtorji knjig s takšno tematiko [6]:

- Hevristika je praktična strategija, ki poveča učinkovitost reševanja kompleksnih problemov. (Feigenbaum, Feldman, 1963)

Algoritem 3 Iterativno poglabljanje

Vhod: Graf G, vozlišče $v \in G$ in globina d

Izhod: Množica S vozlišč, ki so dosegljiva iz vozlišča v in katerih globina je kvečjemu d

```
 1: function DFS(G, v, d)
 2:     v.globina = 0
 3:     Množica S = {v}                                  ▷ množica najdenih vozlišč
 4:     Sklad T = vsi sosedje vozlišča v in nastavi njihovo globino na 1
 5:     while T ni prazen do
 6:         pop v iz T
 7:         if v.globina > d then
 8:             continue
 9:         end if
10:         if v ∉ S then
11:             dodaj v v S
12:             push sosede od v na T, še prej pa nastavi globino
13:             sosedov na v.globina + 1
14:         end if
15:     end while
16:     return S
17: end function
```

- Hevristika vodi iskanje poti do rešitve po metodi, ki izpušča najmanj obetavne poti. (Amarel, 1968)
- Hevristika mora omogočiti, da se izognemo preiskovanju poti, ki vodijo v napačno smer in za dosego cilja uporablja že zbrane podatke. (Lenat, 1983)

Hevristika se lahko pri preiskovanju grafov uporabi na več mestih oziroma na več načinov:

- pri odločanju, katero vozlišče naj se razvije kot naslednje (namesto da se to počne kot pri algoritmih iskanja v širino ali globino),
- ko razvijamo vozlišče, hevristika določi, v kakšnem vrstnem redu se bodo preiskovali njegovi nasledniki,
- pri določanju vozlišč, ki se bodo pri nadaljevanju iskanja zavrgla.

Sodobne hevristične tehnike pogosto premoščajo razkorak med popolnostjo algoritma (da najde optimalno rešitev) in njegovo učinkovitostjo (da obišče čim manj vozlišč). Strategije preiskovanja grafov so se spremenile do te mere, da algoritmi namesto optimalne rešitve iščejo še sprejemljivo suboptimalno rešitev, za iskanje katere pa je potrebno precej manj časa in/ali prostora.

2.3.2 Algoritem najprej najboljši

Ko je govora o preiskovanju grafov, je eden izmed bolj priljubljenih načinov, kako to storiti, ravno preiskovanje s hevristiko, imenovano *najprej najboljši* (Best-first search) [7]. Splošen koncept tega algoritma je, da se vsako vozlišče oceni z neko oceno in na podlagi te ocene se pozneje določi, katero vozlišče se bo razvilo naslednje. To pomeni, da ves čas preiskovanja med seboj tekmuje več poti in vedno se kot naslednja razišče tista pot, ki vsebuje končno vozlišče z najmanjšo oziroma največjo (odvisno do problema) oceno. Iskanje se ustavi, ko ni več vozlišč, ki bi se jih dalo razviti. Za končno rešitev se določi pot, ki je bila najdena kot najboljša (največkrat je to najcenejša pot od začetnega vozlišča do enega od končnih vozlišč). Če takšne poti ni, algoritem to ustrezno javi kot napako.

V praksi se je razvilo več načinov, kako izboljšati to iskanje. Eden je npr. ta, da ko neko pot najdemo, jo lahko razglasimo za optimalno rešitev, ne da bi iskali še ostale potencialno zanimive poti. To bomo spoznali kasneje, ko bomo podrobneje opisali algoritem A*, ki je najbolj znan algoritem iz družine algoritmov najprej najboljši.

Algoritem 4 Najprej najboljši

Vhod: Usmerjen graf G, začetno vozlišče $s \in G$ in množica končnih vozlišč $F \subseteq G$

Izhod: Najboljša pot od začetnega vozlišča do enega izmed končnih vozlišč F, če ta sploh obstaja, sicer vrne napako

```
 1: function BF(G, s, F)
 2:     Množica OPEN = {s}
 3:     Množica CLOSED = {}
 4:     while OPEN ≠ ∅ do
 5:         odstrani vozlišče z najboljšo oceno iz OPEN, ga poimenuj
 6:         z n in ga dodaj v CLOSED
 7:         if n ∈ F then
 8:             rekonstruiraj pot tako, da se vračaš skozi starše vozlišč do
 9:             začetnega vozlišča in vrni pot
10:         end if
11:         zgeneriraj naslednike od vozlišča n
12:         for vse naslednike n do
13:             if naslednik ∉ CLOSED then
14:                 oceni naslednika
15:                 dodaj ga v OPEN
16:                 shrani njegovega starša, da se lahko kasneje vračaš
17:             else
18:                 spremeni povezavo do njegovega starša z n, če je ta
19:                 pot boljša
20:             end if
21:         end for
22:     end while
23:     return napaka
24: end function
```

2.3.3 Algoritem vzpenjanja

Algoritem vzpenjanja (Hill-climbing) je iterativni algoritem, ki mu kot vhod podamo začasno rešitev našega problema, algoritem pa nato s spremembo enega samega elementa rešitve (največkrat vozlišča) poskusi poiskati boljšo rešitev. Če sprememba elementa vrne boljšo rešitev (z višjo izračunano vrednostjo), potem se sprememba upošteva in postopek se ponovi. Postopek traja, dokler ni več mogoče doseči boljše rešitve. Ker se vedno viša vrednost trenutne rešitve, se ta algoritem imenuje algoritem vzpenjanja.

Algoritem med delovanjem ne vzdržuje celega grafa, temveč samo trenutno stanje in njegovo vrednost. Ta se izračuna s pomočjo hevristične funkcije. Hevristična funkcija izračuna pričakovano razdaljo do optimalne rešitve. Kakovost rešitve je odvisna zgolj od natančnosti te hevristične ocene. Algoritem vzpenjanja ne pregleduje vozlišč, ki so od trenutnega vozlišča oddaljene za več kot eno povezavo, zato ne zagotavlja optimalne rešitve (globalnega maksimuma, temveč rešitev, da je lokalni maksimum). Poznamo več različic tega algoritma:

- **Stohastično vzpenjanje:** Medtem ko osnovni algoritem za naslednjo potezo vedno izbere tisto, ki ustreza najbolj strmi poti, stohastično vzpenjanje naključno izbere eno izmed vzpenjajočih se poti. Zato algoritem počasneje konvergira h končni rešitvi in večinoma da boljše rezultate.

- **Brezglavo vzpenjanje (First-choice hill-climbing):** Ta algoritem posnema stohastično vzpenjanje, le da naključno generira naslednike trenutnega vozlišča, dokler se ne generira vozlišče, ki je boljše od trenutnega. Ta strategija je dobra, če imajo vozlišča veliko naslednikov (več sto ali celo tisoč).

- **Random-restart hill-climbing:** Obe opisani različici gotovo najdeta lokalni maksimum, ne pa vedno tudi globalnega. Tretji algoritem pa se drži načela ,,če ne uspeš v prvo, poskusi ponovno". Algoritem večkrat ponovi osnovno vzpenjanje z naključno izbranim začetnim stanjem in ko najde rešitev, ki ustreza nekim prej danim kriterijem, se ustavi.

Več o teh različicah najdemo v [8].

2.3.4 Algoritem A*

Algoritem A* je eden izmed najbolj znanih algoritmov na področju umetne inteligence nasploh. Opisan je bil leta 1986 v članku z naslovom "A Formal

Algoritem 5 Algoritem vzpenjanja

Vhod: Usmerjen graf G, začetno vozlišče $s \in G$
Izhod: Vozlišče, ki je lokalni maksimum

```
1: function HILL CLIMBING(G, s)
2:     Vozlišče current = {s}
3:     Vozlišče neighbor = {}
4:     while True do
5:         neighbor = najboljše ocenjeni naslednik vozlišča current
6:         if hevristična ocena vozlišča neighbor ≤ ocena vozlišča current then
7:             return vozlišče current
8:         else
9:             current = neighbor
10:        end if
11:    end while
12: end function
```

Basis for the Heuristic Determination of Minimum Cost Paths", avtorjev P. E. Hart, N. J. Nilsson in B. Raphael. Zaradi svoje učinkovitosti se danes zelo pogosto uporablja pri iskanju poti in preiskovanju grafov.

Da bi pri preiskovanju razvili kar najmanj vozlišč, mora preiskovalni algoritem v največji možni meri uporabiti vse informacije, ki jih ima na voljo, da se odloči, katero vozlišče razviti kot naslednje. Če slučajno razvije vozlišče, za katerega je očitno, da ne more ležati na optimalni poti, je to izguba časa. Po drugi strani, če zanemari vozlišče, ki bi lahko ležalo vzdolž optimalne poti, se lahko pripeti, da ne bo našel optimalne poti in zaradi tega ne bo popoln.

Definicija 2.3.1 (Popolnost)**.** Algoritem je *popoln*, če zagotovo najde optimalno pot od podanega vozlišča, ki ga označimo s $s \in G$, do končnega oziroma ciljnega vozlišča iz množice $F \subset G$, v kateri so zbrana vsa ciljna oziroma končna vozlišča.

Iz zgoraj napisanega je mogoče sklepati, da učinkovit algoritem potrebuje neko metodo, s katero lahko ovrednoti posamezno vozlišče in tako lažje in bolj zanesljivo izbere tisto, ki se bo razvilo naslednje. Algoritem A* ima takšno metodo. Definirana je kot

$$f(x) = g(x) + h(x) \tag{2.1}$$

kjer je $g(x)$ funkcija, ki izračuna ceno poti od začetnega vozlišča do trenutnega vozlišča, označenega z x, in $h(x)$ hevristična funkcija, ki izračuna oziroma *oceni* ceno poti, ki je potrebna do ciljnega vozlišča.

2.3.4.1 Popolnost algoritma A*

Izrek 2.3.1 (Popolnost)**.** Naj bo z *h(n)* izračunana cena poti, ki je potrebna za premik od trenutnega vozlišča n do ciljnega vozlišča. Naj bo *h*(n)* dejanska cena, ki je potrebna za premik od trenutnega vozlišča n do ciljnega vozlišča. Če je

$$h(n) \leq h^*(n) \tag{2.2}$$

za vse n, potem je algoritem A* popoln. Za dokaz tega izreka naj si bralec prebere [9].

Enačba (2.2) nam pravi, da hevristična funkcija ne sme precenjevati cene poti, ki je potrebna za prehod od trenutnega vozlišča do ciljnega vozlišča. Primer: če moramo npr. iskati pot na zemljevidu med dvema krajema, lahko za hevristično oceno uporabimo kar zračno razdaljo med tema krajema, saj je to najmanjša možna pot med tema krajema.

Če si izberemo za $h(n) = 0$ za vse n, potem bo A* popoln za vsak problem. To bi v praksi pomenilo, da ocenimo, da je vsako vozlišče za 0 enot oddaljeno do ciljnega vozlišča in enačbo (2.1) lahko preoblikujemo v

$$f(x) = g(x) \tag{2.3}$$

To nadalje pomeni, da algoritem kot naslednje vedno razvije tisto vozlišče, ki ima do sedaj najkrajšo pot (najmanjši $g(x)$). Ko algoritem najde ciljno vozlišče, je s tem tudi našel najkrajšo pot. Slabost tega je, da nikjer nismo uporabili ključnega dela algoritma A*, tj. informacije o domeni problema. Zaradi tega mora algoritem A* preiskati in razviti več vozlišč, kot bi to bilo potrebno, če bi vanj vključili še hevristično funkcijo.

2.3.4.2 Optimalnost algoritma A*

Obstajajo problemi, pri katerih lahko že zaradi samih omejitev, ki so v naravi, uporabimo hevristično funkcijo, ki nima vseh vozlišč n, določenih z 0. V naravi npr. ni možno priti po cesti iz enega kraja v drugega po krajši poti, kot bi to lahko storili, če bi med njima obstajala letalska povezava. Zato nam že intuicija pove, da lahko za spodnjo hevristično oceno uporabimo zračno razdaljo med njima. Če lahko to oceno postavimo še višje, toliko bolje, saj bomo s tem še natančneje ocenili potrebno pot do našega cilja.

V splošnem pri algoritmu A* velja pravilo, da če sta dani dve različni hevristični funkciji, ki sta tudi popolni (njun $h(n) \leq h^*(n)$), potem bo A*

moral razviti manj vozlišč z uporabo tiste hevristične funkcije, ki ima večjo spodnjo mejo.

Predpostavimo, da je hevristična funkcija monotona, kar pomeni, da zanjo velja:

$$h(x) \leq d(x, y) + h(y) \tag{2.4}$$

kjer je *d(x,y)* dolžina povezave med vozliščema x in y. V tem primeru lahko zapišemo še spodnjo trditev in izrek.

V [9] je predstavljen podrobnejši dokaz naslednje trditve:

Trditev 2.3.1. Če je neko vozlišče n, ko algoritem A* preiskuje graf, v množici zaprtih vozlišč, potem je izračunani $g(n)$ optimalen (najmanjši možen).

Posledica te trditve je, da algoritmu A*, ko enkrat zapre neko vozlišče, tega nikoli več ni treba ponovno odpreti oziroma razviti. To pomeni, da je algoritem našel optimalno pot od začetnega vozlišča do tega vozlišča.

Izrek 2.3.2 (Optimalnost)**.** Naj bo A poljuben popoln algoritem, ki ni bolj informiran od A*. Potem velja da, če A* razvije vozlišče n, ga razvije tudi A.

Tudi za ta izrek bo bralec našel temeljit dokaz v [9]. Posledica zgornjega izreka je, da ne obstaja algoritem, ki bi pri enaki informiranosti za iskanje optimalne poti moral razviti manj vozlišč, kot jih mora A*.

Algoritem 6 Algoritem A*

Vhod: Usmerjen graf G, zač. vozlišče *zacetek* $\in G$ in konč. vozlišče *cilj* $\in G$
Izhod: Optimalna pot od začetnega do končnega vozlišča

```
 1: function A*(G, zacetek, cilj)
 2:     Množ. zaprta = {}; Množ. odprta = {zacetek}; Slovar prisel_iz = {}
 3:     g_ocena[zacetek] = 0
 4:     h_ocena[zacetek] = hevristična_ocena(zacetek, cilj)
 5:     f_ocena[zacetek] = g_ocena[zacetek] + h_ocena[zacetek]
 6:     while odprta ≠ ∅ do
 7:         trenutno = vozlišče v odprta z najmanjšo f_oceno[]
 8:         if trenutno = cilj then
 9:             return rekonstruiraj_pot(prisel_iz, prisel_iz(cilj))
10:         end if
11:         odstrani trenutno iz odprta
12:         dodaj trenutno v zaprta
13:         for vsakega soseda sosed vozlišča trenutno do
14:             if sosed ∈ zaprta then
15:                 continue
16:             end if
17:             pogojna_g_ocena = g_ocena[trenutno] +
18:              razdalja_med(trenutno,sosed)
19:             if sosed ∉ odprta then
20:                 dodaj sosed v odprta
21:                 h_ocena[sosed] = hevristična_ocena(sosed, cilj)
22:                 pogojna_g_je_boljša = true
23:             else
24:                 if pogojna_g_ocena < g_ocena[sosed] then
25:                     pogojna_g_je_boljša = true
26:                 else
27:                     pogojna_g_je_boljša = false
28:                 end if
29:             end if
30:             if pogojna_g_je_boljša = true then
31:                 prisel_iz[sosed] = trenutno
32:                 g_ocena[sosed] = pogojna_g_ocena
33:                 f_ocena[sosed] = g_ocena[sosed] + h_ocena[sosed]
34:             end if
35:         end for
36:         return napaka
37:     end while
38: end function
```

Poglavje 3

Algoritem D*

Veliko strokovne literature je posvečene načrtovanju poti za enega ali več robotov skozi nek prostor z ovirami. Večina teh člankov predpostavlja, da je celotno okolje znano že pred samim premikom robota. Na žalost ima robot ponavadi le malo informacij o prostoru, v katerem se nahaja. Je pa zato opremljen s senzorji, ki so sposobni zaznati in izmeriti predmete, ki jih robot sreča med premikanjem po prostoru. Eden od pristopov pri načrtovanju poti v takšnem primeru bi bil ta, da izdelamo globalno pot z uporabo poznane informacije o prostoru in poskušamo zaobiti ovire, ki jih robot med premikanjem odkrije s pomočjo svojih senzorjev. Če je začrtana pot preveč ovinkasta z ovirami, se poišče nova globalna pot.

Lumelsky je predlagal, da robota pošljemo naravnost proti cilju in ko senzorji zaznajo oviro, se robot premika okrog ovire, dokler ne naleti na lokacijo, ki je najbližja končnemu cilju. Robot potem nadaljuje pot naravnost proti cilju. Pirzadeh si je zamislil strategijo, kjer robot tava po prostoru, dokler ne najde cilja. Robot gre vedno na sosednjo lokacijo, ki ima najmanjšo ceno in hkrati poveča njeno ceno z namenom, da jo kaznuje, če bo naslednjič spet prispel do nje.

Ti dve strategiji vedno najdeta pot, če ta obstaja, vendar je njuna rešitev suboptimalna v smislu, da ne najdeta najcenejše možne poti glede na informacije, ki jih pridobita iz senzorjev, saj je pri njima predpostavljeno, da je vsa informacija, ki je bila dana a priori, pravilna. Povedano preprosteje - ni nujno, da se podatki, ki so bili vnaprej podani, skladajo s tistimi, ki jih pridobi robot s pomočjo senzorjev (prostor se lahko med tem časom spremeni).

Tudi v tem primeru je možno poiskati optimalno pot glede na ceno. Robot se premika po prostoru po takšni poti, kot mu jo začrta algoritem s trenutnim znanjem o prostoru. Če robot med premikanjem naleti na oviro, ki ni bila

znana pred načrtovanjem te poti ali pa s senzorji zazna, da kjer bi morala biti ovira, te v resnici ni, se posodobi znanje o prostoru in na novo poišče optimalna pot od trenutnega položaja robota do končnega cilja. Čeprav ta pristop z "grobo silo" daje optimalen rezultat, je lahko skrajno neučinkovit v okoljih, kjer je cilj daleč od trenutnega položaja robota in/ali če je naše začetno znanje o prostoru skromno.

3.1 Inkrementalno iskanje

Inkrementalno iskanje je preiskovalna tehnika, ki pri iskanju ponovno uporabi informacijo iz prejšnjih iskanj za dosego cilja - poiskati rešitev problema s pomočjo zaporedja podobnih iskalnih problemov. Bistvo je, da to lahko naredi hitreje, kot če bi vsak takšen podproblem reševali posamično. Za razliko od nekaterih drugih preiskovalnih tehnik, ki pohitrijo iskanje, ta tehnika zagotavlja optimalnost najdene rešitve, če ta obstaja.

Mnogo sistemov za umetno inteligenco, katerih ena izmed nalog je tudi načrtovanje, mora velikokrat sproti popravljati svoje izračunane načrte zaradi sprememb, ki so se zgodile v svetu, v katerega so postavljeni, ali pa sprememb, ki so se zgodile na modelih resničnega sveta, ki jih ti sistemi simulirajo. Npr., včasih se izkaže, da se trenutna situacija samo malo razlikuje od tiste, ki smo jo sprva predpostavljali. V takšnem primeru morebiti naš prvotni načrt ni več uporaben in treba je ponovno poiskati nov načrt. V takem primeru večina preiskovalnih algoritmov poišče nov načrt od začetka, to pomeni, da poišče nov načrt neodvisno od prejšnjega. To je lahko zelo neučinkovito v velikih domenah, kjer se spremembe dogajajo pogosto. Na srečo imajo spremembe ponavadi samo lokalen vpliv na okolico. Zaradi tega ni potrebe, da bi morali še enkrat preiskati prostor v celoti, ampak je dovolj, da preiščemo le tisti del prostora, na katerega je imela dana sprememba vpliv. Inkrementalno iskanje počne ravno to.

Vsi algoritmi, ki bodo opisani v nadaljevanju, so inkrementalni algoritmi (D*, Focused D*, LPA*, D* Lite).

3.2 Ideja algoritma

Algoritem D* je algoritem za iskanje optimalnih poti v prostoru za robote, ki imajo senzorje in zemljevid okolja. Zemljevid je lahko popoln, prazen ali pa vsebuje delno informacijo o prostoru. Neznane regije na zemljevidu so lahko opisane s približno informacijo, stohastičnim modelom zasedenosti ali

celo hevristično oceno. D* je po funkcionalnosti enak algoritmom, ki delujejo po načelu "grobe sile", le da je veliko bolj učinkovit.

Algoritem je formuliran na enak način kot drugi algoritmi za iskanje poti. Prostor je predstavljen kot usmerjen graf, katerega povezave so označene z vrednostmi, ki predstavljajo ceno povezave. Že znane in ocenjene vrednosti povezav so vnešene v zemljevid, robotovi senzorji pa so sposobni izmeriti posamezne vrednosti med povezavami oziroma zaznati ovire, če so te v njegovem dometu, in s tem posodobiti zemljevid.

3.3 Osnovni D*

Algoritem je dobil ime D* zaradi podobnosti z algoritmom A*, pri čemer se tu lahko cene povezav med samim izvajanjem algoritma dinamično spreminjajo [10]. Robot začne v nekem stanju in se premika po povezavah (s tem povečuje ceno končne poti), ki vodijo v sosednja stanja, dokler ne doseže ciljnega stanja, označenega z G. Vsako stanje X razen stanja G ima kazalec, ki kaže v naslednje stanje Y, označeno z $b(X) = Y$. D* uporablja te kazalce za gradnjo poti do cilja. Cena poti od stanja X do stanja Y je pozitivno število $c(X, Y)$, ki ga določi funkcija c. Če neko vozlišče X nima povezave do vozlišča Y, potem je v tem primeru vrednost $c(X, Y)$ nedefinirana.

Definicija 3.3.1 (Sosednost)**.** Dve stanji X in Y sta sosednji, če je definiranana ena od vrednosti $c(X, Y)$ ali $c(Y, X)$.

Tako kot A* tudi D* vzdržuje seznam odprtih stanj z imenom *OPEN*. Seznam *OPEN* se uporablja, da se lahko ugotovijo spremembe, ki so se zgodile na cenah povezav, in izračuna cena poti do poljubnih stanj v prostoru. Vsakemu stanju X je pridružen tudi atribut $t(X)$, tako da je:

- $t(X) = NEW$, če X nikoli ni bil v seznamu *OPEN*,
- $t(X) = OPEN$, če je X trenutno v seznamu *OPEN*,
- $t(X) = CLOSED$, če X ni več v seznamu *OPEN*.

Za vsako stanje X algoritem D* vzdržuje tudi ceno vsote povezav na poti od X do G, ki jo izračuna funkcija $h(G, X)$. V idealnem primeru je ta ocena enaka optimalni ceni poti med stanjema X in G, ki jo izračuna implicitno podana funkcija o. Ta funkcija izračuna ceno poti tako, da se sprehodi po povezavah med stanjema G in X in sešteje cene povezav na tej poti.

Za vsako stanje X, ki je na seznamu *OPEN*, je definirana tudi vrednost $k(G,X)$. Ta vrednost je enaka minimumu vrednosti $h(G,X)$, preden se na povezavah zgodijo spremembe cen. Vrednost $k(G,X)$ razvrsti stanje X, ki je v seznamu *OPEN*, v enega izmed dveh stanj:

- v stanje *RAISE*, če $k(G,X) < h(G,X)$,
- v stanje *LOWER*, če $k(G,X) = h(G,X)$.

D* uporabi oznako *RAISE* na tistih stanjih v seznamu *OPEN*, ki se jim je povečala cena poti do cilja zaradi povečanja cene na eni ali več povezavah, oznako *LOWER* pa na tistih stanjih v seznamu *OPEN*, ki se jim je zmanjšala cena poti do cilja zaradi zmanjšanja cene na eni ali več povezavah ali pa je bila najdena nova, cenejša pot do cilja. Ko se neko stanje odstrani s seznama *OPEN*, se uporabita ti dve oznaki, da se prenesejo spremembe cen do njegovih sosedov, ki se jim potem posodobijo vsi zgoraj našteti parametri $t(X), h(G,X)$ in $k(G,X)$.

Stanja v seznamu *OPEN* so urejena v naraščajočem vrstnem redu glede na rezultat funkcije $k(G,X)$. Parameter k_{min} je definiran kot $min(k(X))$ za vse X, kjer je $t(X) = OPEN$. Parameter k_{min} predstavlja pomemben prag v algoritmu D*: poti, katerih cena je manjša ali enaka k_{min}, so optimalne poti in tiste, ki imajo ceno večjo od k_{min}, morda niso optimalne. Parameter k_{old} je definiran tako, da je enak parametru k_{min}, preden je bilo odstranjeno poljubno stanje s seznama *OPEN*. Če še nobeno stanje ni bilo odstranjeno s seznama *OPEN*, je parameter k_{old} nedefiniran.

Vrsta stanj, označena z $\{X_1,X_N\}$, je definirana kot *zaporedje*, če je $b(X_{i+1}) = X_i$ za vse i, $1 \leq i \leq N$, in $X_i \neq X_j$ za vse (i,j), $1 \leq i < j \leq N$. To pomeni, da *zaporedje* $\{X_1,X_N\}$ definira pot, pridobljeno iz kazalcev od X_N do X_1.

Zaporedje $\{X_1,X_N\}$ je *monotono*, če ($t(X_i)$ = CLOSED in $h(G,X) < h(G,X_{i+1})$) ali ($t(X_i)$ = OPEN in $k(G,X) < h(G,X_{i+1})$) za vse i, $1 \leq i < N$. Algoritem D* gradi in vzdržuje monotona zaporedja $\{G,X\}$, ki predstavljajo najnižjo ceno poti za vsako stanje X, ki je ali je bilo na seznamu *OPEN*.

Če je dano zaporedje stanj $\{X_1,X_N\}$, potem je stanje X_i prednik stanja X_j, če velja $1 \leq i < j \leq N$, oziroma je potomec stanja X_j, če velja $1 \leq j < i \leq N$.

3.3.1 Opis algoritma

Ker imamo opravka z okolji, ki so delno znana vnaprej, in roboti, ki imajo senzorje, velja nekaj zakonitosti [11]:

- Ker imajo robotovi senzorji omejen domet, se bodo zaznala le odstopanja med trenutnim stanjem zemljevida in izmerjenimi podatki, ki so v bližini robota. Posledica tega je, da se bo začrtana pot morala preurediti le lokalno.
- Večina ovir je majhnih in so zato potrebne le majhne preusmeritve začrtane poti, kar pomeni, da algoritmu ni treba izvajati zahtevnih računskih operacij, ko se vrača po začrtani poti in išče novo optimalno pot.
- Robot se skoraj ves čas monotono približuje cilju, zato je vsakokrat treba preurediti le preostanek poti, ki je ostal do cilja. To pomeni, da med izvajanjem algoritma postaja načrt, ki ga je treba spreminjati oziroma popravljati, vedno krajši in izračun s tem hitrejši.

Očitno je, da je takrat, ko imamo opravka z delno poznanimi okolji, kot del globalnega načrtovanja potreben tudi proces preurejanja poti. Procesa preurejanja poti se je treba lotiti, kadar robot s svojimi senzorji odkrije odstopanja med trenutno poznanim zemljevidom in resničnim okoljem.

Kot smo že omenili, je ključna vrednost $k(G, X)$, ki trenutno stanje X, ki je na seznamu *OPEN*, razvrsti v enega izmed dveh stanj, *RAISE* ali *LOWER*. Ko se neko stanje odstrani s seznama *OPEN*, se nato glede na njegovo razvrstitev prenesejo spremembe cen povezav do njegovih sosedov. Ti se nato dodajo na seznam *OPEN* in postopek se ponovi.

Algoritem D* je sestavljen iz petih korakov:

(I) Na začetku se vsem stanjem dodeli atribut *t()* kot *NEW*. $h(G)$ se nastavi na 0 in G se doda na seznam *OPEN*.

(II) Nato se v zanki kliče funkcija *PROCESS-STATE*, dokler se robotovo začetno stanje S ne odstrani s seznama *OPEN* (to je *t(S) = CLOSED*), kar pomeni, da pot obstaja, ali pa se vrne število *-1*, ki označuje, da pot ne obstaja.

(III) Robot se nato s pomočjo kazalcev, ki jih vrača funkcija $b(X)$, premika po začrtani poti, dokler ne doseže cilja ali zazna napake v ceni povezave na poti. Napako zazna s pomočjo svojih senzorjev, ko primerja izmerjene vrednosti s poznanim zemljevidom.

(IV) Če robot zazna napako, algoritem pokliče funkcijo *MODIFY-COST*, ki popravi cene na povezavah s pomočjo funkcije *c()*, in doda prizadeta stanja na seznam *OPEN*.

(V) Nato se ponovno v zanki kliče funkcija *PROCESS-STATE*, dokler se ne sestavi nova pot. Tedaj robot nadaljuje s točko (III).

Kot lahko opazimo, sta ključni funkciji pri delovanju algoritma *PROCESS-STATE* in *MODIFY-COST*, ostale pomožne funkcije pa so: *MIN-STATE*, ki vrne stanje na seznamu *OPEN*, ki ima minimalno vrednost *k()*, oziroma *NULL*, če je seznam prazen; *GET-KMIN*, ki vrne k_{min} za seznam *OPEN*, oziroma *-1*, če je seznam prazen; $DELETE(X)$, ki izbriše stanje X s seznama *OPEN* in nastavi njegov atribut $t(X) = CLOSED$; in funkcija $INSERT(X, h_{new})$, ki nastavi $k(X) = h_{new}$, če $t(X) = NEW$, $k(X) = min(k(X), h_{new})$, če $t(X) = OPEN$, in $k(X) = min(h(X), h_{new})$, če $t(X) = CLOSED$ in nastavi $h(X) = h_{new}$, ter doda ali popravi pozicijo stanja X na seznamu *OPEN*, ki je umeščen glede na vrednost $k()$.

Vloga stanj *RAISE* in *LOWER* je osrednjega pomena za delovanje algoritma. Stanje *RAISE* ($k(X) < h(X)$) prenese povečanje cene in stanje *LOWER* ($k(X) = h(X)$) prenese znižanje cene. Ko se poveča cena povezave na poti do cilja, se prizadeto stanje doda na seznam *OPEN* in prek stanja *RAISE* se prenese povečanje cene poti do vseh ostalih stanj, do katerih pridemo prek te povezave. Če ima stanje, ki je označeno z *RAISE*, za soseda stanje z nižjo ceno, se tudi to stanje *LOWER* doda na seznam *OPEN*. Če pa se cena povezave, ki je na poti do cilja, zmanjša, se prek stanja *LOWER* prenese zmanjšanje cene poti do vseh ostalih stanj, do katerih pridemo prek te povezave, vključno z njihovimi sosedi.

3.3.2 Lastnosti algoritma

Potem ko se na začetku vsem stanjem nastavi atribut $t(X) = NEW$ in se ciljno stanje G doda na seznam *OPEN*, se v zanki kliče funkcija *PROCESS-STATE*, da se kreira zaporedje stanj, ki tvorijo pot od ciljnega stanja do končnega stanja. Funkcija *MODIFY-COST* pa je zadolžena, da popravi vse cene povezav, ki se ne skladajo z izmerjenimi (s pomočjo robotovih senzorjev), in jih doda na seznam *OPEN*. Za D* veljajo naslednje lastnosti:

Lastnost 3.3.1 (Pravilnost)**.** Če $t(X) \neq NEW$, potem obstaja zaporedje $\{X, G\}$, ki je hkrati tudi monotono.

Lastnost 3.3.2 (Optimalnost)**.** Če funkcija *PROCESS-STATE* vrne vrednost k_{min}, ki je enaka ali presega vrednost $h(X)$, potem velja $h(X) = o(X)$, kjer $o(X)$ predstavlja optimalno pot od stanja X do stanja G.

Algoritem 7 PROCESS-STATE

Vhod: Nič

Izhod: Vrednost k_{min} za seznam *OPEN*

```
1: function PROCESS-STATE()
2:     X = MIN-STATE()
3:     if X = NULL then return -1
4:     end if
5:     k_old = GET-KMIN(); DELETE(X)
6:     if k_old < h(X) then
7:         for vsakega soseda Y od X do
8:             if h(Y) ≤ k_old and h(X) > h(Y) + c(Y,X) then
9:                 b(X) = Y; h(X) = h(Y) + c(Y,X)
10:            end if
11:        end for
12:    end if
13:    if k_old = h(X) then
14:        for vsakega soseda Y od X do
15:            if t(Y) = NEW or (b(Y) = X and h(Y) ≠ h(X) + c(X,Y) ) or
16:                (b(Y) ≠ X and h(Y) > h(X) + c(X,Y) ) then
17:                    b(Y) = X; INSERT(Y, h(X) + c(X,Y))
18:            end if
19:        end for
20:    else
21:        for vsakega soseda Y od X do
22:            if t(Y) = NEW or (b(Y) = X and h(Y) ≠ h(X)+c(X,Y) ) then
23:                b(Y) = X; INSERT(Y, h(X) + c(X,Y))
24:            else
25:                if b(Y) ≠ X and h(Y) > h(X) + c(X,Y) then
26:                    INSERT(X, h(X))
27:                else
28:                    if b(Y) ≠ X and h(X) Y h(Y) + c(Y,X) and
29:                        t(Y) = CLOSED and h(Y) > k_old then
30:                            INSERT(Y, h(Y))
31:                    end if
32:                end if
33:            end if
34:        end for
35:    end if
36:    return GET-KMIN()
37: end function
```

Algoritem 8 MODIFY-COST

Vhod: Vozlišči X in Y ter vrednost nove cene povezave *cval*
Izhod: Vrednost k_{min} za seznam *OPEN*

```
1: function MODIFY-COST(X,Y,cval)
2:     c(X,Y) = cval
3:     if t(X) = CLOSED then
4:         INSERT(X, h(X))
5:     end if
6:     return GET-KMIN()
7: end function
```

Lastnost 3.3.3 (Popolnost)**.** Če obstaja pot med X in G in preiskovalni prostor vsebuje končno število stanj, bo zaporedje $\{X, G\}$ kreirano po končnem številu klicev funkcije *PROCESS-STATE*. Če pot med X in G ne obstaja, funkcija *PROCESS-STATE* vrne rezultat -1 in atribut $t(X)$ je enak *NEW*.

V naslednjih treh podpoglavnjih bodo natančneje predstavljene vse tri lastnosti algoritma, podrobni dokazi pa so v [12].

3.3.2.1 Pravilnost algoritma D*

Lastnost 3.3.1 pravi, da ko je enkrat začetno stanje obiskano, obstaja končno zaporedje stanj, ki nas pripelje do cilja.

Kadarkoli funkcija *PROCESS-STATE* obišče vozlišče z vrednostjo atributa *NEW*, dodeli njen kazalec $b()$, da kaže v trenutno vozlišče X, in nastavi njen $h()$ tako, da se ohrani monotonost. Monotonost zaporedja se ohranja s pomočjo funkcij *t(), h(), k()* in *b()*. Ko se stanje X doda na seznam *OPEN*, se $k(X)$ nastavi na vrednost $h(X)$, da se ohrani monotonost za vsa stanja, katerih kazalci $b()$ kažejo na stanje X. Prav tako se vsem njegovim sosedom, če je treba, poviša vrednost $h()$, da se ohrani monotonost. Kazalec stanja $(X, b(X))$ se nastavi tako, da kaže na stanje Y le v primeru, če velja $h(Y) < h(X)$ in če zaporedje {Y, G} ne vsebuje stanja *RAISE*. Ker stanje Y ne vsebuje stanja *RAISE*, potem za vsa stanja v tem zaporedju velja, da je njihov $h()$ manjši od $h(Y)$. To pomeni, da stanje X nikakor ne more biti prednik stanja Y in zato s kazalci ne more ustvariti cikla. Posledica tega je, da ko algoritem obišče stanje X, se kreira zaporedje $\{X, G\}$ in kasneje se s pomočjo funkcij *t(), h(), k()* in *b()* to zaporedje tudi ohranja.

3.3.2.2 Optimalnost algoritma D*

Lastnost 3.3.2 definira pogoje, pri katerih veriga kazalcev do cilja tvori optimalno pot.

Vsakokrat, ko se stanje X doda ali odstrani s seznama *OPEN*, algoritem D* spremeni njegov $h(X)$ tako, da za vsak par (X, Y) velja $k(X) \leq h(Y)+c(Y, X)$, pri čemer je $t(X) = OPEN$ in $t(Y) = CLOSED$. Kadar je stanje X izbrano v funkciji *PROCESS-STATE*, da se razvije, njegovi sosedje, katerih atribut $t()$ je enak *CLOSED*, ne morejo zmanjšati njegovega $h(X)$ pod mejo k_{min}, niti tega ne morejo storiti sosedje, katerih atribut $t()$ je enak *OPEN*, saj mora biti zaradi monotonosti njihov $h()$ večji od k_{min}.

Stanja, ki se dodajo na seznam *OPEN* med razvijanjem vozlišča X, morajo imeti svoj $k()$ večji od $k(X)$. To pomeni, da vsakokrat, ko se pokliče funkcija *PROCESS-STATE*, postane k_{min} večji ali kvečjemu enak prejšnjemu. Če so stanja, ki imajo svoj $h()$ manjši ali enak k_{old}, optimalna, so optimalna tudi stanja, ki imajo svoj $h()$ med k_{old} in k_{min}, saj nobeno stanje, ki je na seznamu *OPEN*, ne more več zmanjšati njihove cene poti. Torej so stanja, ki imajo svoj $h()$ manjši ali enak k_{min}, optimalna. Funkcija *PROCESS-STATE* zgradi optimalna zaporedja do vseh dosegljivih stanj.

Če se med tekom algoritma spremeni cena povezave, $c(X, Y)$, potem funkcija *MODIFY-COST* doda stanje X na seznam *OPEN* in ko se ponovno pokliče funkcija *PROCESS-STATE*, je spremenljivka k_{min} manjša ali enaka $h(X)$. Ker ta sprememba cene povezave ne vpliva na nobeno stanje Y, pri katerem velja $h(Y) \leq h(X)$, lastnost optimalnosti še zmeraj drži.

3.3.2.3 Popolnost algoritma D*

Lastnost 3.3.3 pravi, da če obstaja pot med X in G, jo bo algoritem našel in sestavil. Če pot ne obstaja, bo to sporočil v končnem času.

Vsakokrat, ko funkcija *PROCESS-STATE* razvije vozlišče X, se vsi njegovi sosedje z vrednostjo atributa *NEW* dodajo na seznam *OPEN*. Torej, če obstaja zaporedje $\{X, G\}$, se bo to tudi zgradilo, razen če v tem zaporedju obstaja stanje Y, ki še ni bilo razvito. Ko se stanje doda na seznam *OPEN*, se njegov $k()$ ne more več povečati. In ravno zaradi monotonosti k_{min} se bo stanje Y enkrat izbralo za razvijanje in pot od X do G se bo lahko sestavila.

3.4 Focused D*

Ta algoritem je plod dela istega avtorja, ki je iznašel tudi algoritem D*, Anthonyja Stentza, profesorja z univerze v Pittsburghu [13].

Algoritem *Focused D** je razširitev algoritma *D**. Beseda *"Focused"* pove, da se algoritem osredotoča na tisti del algoritma *D**, ki je zadolžen za posodobitev cen povezav, potem ko robot s svojimi senzorji odkrije neskladnost med poznanim zemljevidom in resničnim svetom. Algoritem želi minimizirati število vozlišč, ki jih je treba razviti, in s tem zmanjšati čas, ki je potreben za računanje. Algoritem uporabi, podobno kot A*, hevristično funkcijo za prenos povečanja in zmanjšanja cene poti. Avtor tega algoritma navaja, da je čas, ki ga računalnik potrebuje za izračun poti, v primerjavi z osnovnim algoritmom D* od dva- do trikrat manjši.

3.4.1 Motivacija za nadgradnjo osnovnega D*

Osnovni algoritem D*, opisan v [12], prenese spremembo cene poti na vozliščih, ki imajo napačno ceno povezave do sosedov ne glede na to, kje se robot trenutno nahaja. Zaradi tega se včasih ponovno razvijejo vozlišča, pri katerih robot ne pridobi nobene koristi. Podobno kot A* lahko tudi D* uporabi hevristiko za usmerjanje iskanja poti in s tem zmanjšanja števila razvitih vozlišč.

Naj bo *osredotočena hevristika* $g(X, R)$ definirana kot ocena cene poti, od robotove lokacije R do lokacije X. Potem lahko definiramo *oceno cene poti* za premik robota od stanja R do končnega stanja G, ki gre prek stanja X kot:

$$f(X, R) = h(X) + g(X, R) \tag{3.1}$$

Če izpolnimo zahtevo, da je funkcija $g(X, R)$ monotona, in ker je $h(X)$, ko stanje X odstranimo s seznama *OPEN*, optimalen, bo optimalna tudi pot do stanja R.

V primeru stanja *RAISE* prejšnji $h()$ definira spodnjo mejo za vrednosti $h()$ vseh stanj *LOWER*, ki so še lahko odkrita. Torej, če za obe vrsti stanj (*RAISE* in *LOWER*) uporabimo enako *osredotočeno hevristiko* $g()$, potem prejšnja $f()$ ocena stanj *RAISE* definira spodnjo mejo za $f()$ oceno stanj *LOWER*, ki so še lahko odkrita. Posledica tega je, da če $f()$ ocena stanj *LOWER*, ki so na seznamu *OPEN*, presega prejšnjo $f()$ oceno stanj *RAISE*, potem je vredno razviti stanja *RAISE*, da bi odkrili boljša stanja *LOWER*. Na podlagi tega sklepanja je dobro, če so stanja *RAISE*, ki so na seznamu *OPEN*, urejena v skladu z enačbo 3.1. Da se izognemo ciklom, sestavljenim iz kazalcev, ki kažejo

na naslednje stanje, morajo biti stanja na seznamu *OPEN*, ki imajo enako $f()$ oceno, urejena po naraščajoči vrednosti $k()$.

Ko se glede na trenutno robotovo lokacijo izračuna nova optimalna pot do cilja, se robot premika po tej poti. Če njegovi senzorji zaznajo novo nepravilnost v cenah na povezavah, je treba iskanje poti osredotočiti na robotovo novo lokacijo. Ampak stanja, ki so na seznamu *OPEN*, so osredotočena na staro lokacijo in imajo zato napačni oceni $g()$ in $f()$. Ena od rešitev je, da za vsako stanje, ki je na seznamu *OPEN*, vsakokrat ko se robot premakne in doda na seznam novo stanje, na novo izračunata oceni $g()$ in $f()$. Kot bomo videli v nadaljevanju, zna algoritem Focused D* to narediti bolj elegantno in hitreje.

3.4.2 Ideja algoritma Focused D*

Algoritem Focused D* izkorišča dejstvo, da robot ponavadi naredi le nekaj korakov, preden je spet treba popraviti začrtano pot. Zaradi tega imata oceni *g()* in $f()$ nekega stanja le malo napako. Predpostavimo, da se stanje X doda na seznam *OPEN* v času, ko je robot na lokaciji R_0. Njegova ocena *f()* je takrat $f(X, R_0)$. Če se robot premakne na lokacijo R_1, je treba ponovno izračunati $f(X, R_1)$ in prilagoditi njegovo pozicijo na seznamu *OPEN*. Da se temu izognemo, lahko izračunamo le spodnjo mejo vrednosti $f(X, R_1)$

$$f_L(X, R_1) = f(X, R_0) - g(R_1, R_0) - \epsilon \tag{3.2}$$

$f_L(X,R_1)$ je spodnja meja $f(X,R_1)$, saj je predpostavljeno, da se robot premakne v smeri stanja X, to pomeni, da se odšteje neka vrednost od $g(X, R_0)$. Parameter ϵ je poljubno majhno pozitivno število. Če je pozicija stanja X na seznamu *OPEN* določena glede na $f_L(X,R_1)$ in ker je $f_L(X,R_1)$ spodnja meja prave ocene $f(X,R_1)$, bo stanje X izbrano za razvitje preden ali pa natanko takrat, ko bo to v resnici potrebno. Nikoli se to ne bo zgodilo prepozno. Šele ko bo prišel ta trenutek, se bo izračunala prava vrednost $f(X,R_1)$ in stanje X bo popravilo svojo lokacijo na seznamu *OPEN* glede na pravkar izračunano oceno $f(X,R_1)$.

Mogoče se sprva zdi ta pristop slab, saj je najprej treba urediti stanja na seznamu *OPEN* glede na $f_L()$ in potem po potrebi še enkrat zamenjati $f_L()$ s pravimi vrednostmi $f()$. Ampak ker je $g(R_1, R_0) + \epsilon$ odštet od vseh stanj na seznamu *OPEN*, se vrstni red ohrani in ga ni treba ponovno preurediti. Poleg tega se je prvemu koraku mogoče izogniti, če prištejemo $g(R_1, R_0) + \epsilon$ stanju, preden ga dodamo na seznam *OPEN*, namesto da to vrednost odštejemo od stanj, ki so že na tem seznamu. Tako nam ostane samo en korak, ki ga je treba izvršiti, in to je razvrstiti stanja na seznamu *OPEN* na pravilna mesta.

Za nameček je ta korak potreben samo za tista stanja, ki so obetavna, da jih bo robot dosegel. Za večino problemov to pomeni manj kot 2 % vseh stanj na seznamu *OPEN*.

3.4.3 Definicije in ostale lastnosti algoritma

Pri algoritmu Focused D* veljajo enake lastnosti kot pri osnovnem D* za *t(X), h(X), k(X)* in klasifikaciji stanj v *RAISE* in *LOWER*.

Razlikujeta se glede seznama *OPEN*, saj so pri tem algoritmu stanja urejena na seznamu *OPEN* glede na vrednost funkcije $f_B(X, R_i)$, kjer je X stanje na seznamu *OPEN* in R_i robotovo stanje v trenutku, ko je bil X dodan ali spremenjen na seznamu *OPEN*. Naj bo $\{ R_0, R_1, ..., R_N \}$ zaporedje stanj, ki jih je robot zavzemal, ko je algoritem dodajal stanja na seznam *OPEN*. Vrednost funkcije $f_B()$ se izračuna kot

$$f_B(X, R_i) = f(X, R_i) + d(R_i, R_0) \tag{3.3}$$

kjer je $f(X, R_i)$ ocena cene poti, ki jo robot potrebuje od trenutnega stanja X do končnega stanja G, izračunana kot

$$f(X, R_i) = h(X) + g(X, R_i) \tag{3.4}$$

in $d(R_i, R_0)$ pristranska funkcija, izračunana kot

$$d(R_i, R_0) = g(R_1, R_0) + g(R_2, R_1) + ... + g(R_i, R_{i-1}) + i\epsilon \tag{3.5}$$

če je $i > 0$. Če je $i = 0$ je $d(R_0, R_0) = 0$. Funkcijo *g(X,Y)* imenujemo *osredotočena hevristična funkcija* in predstavlja oceno cene poti od stanja Y do stanja X. Stanja na seznamu *OPEN* so razvrščena glede na naraščajočo vrednost funkcije $f_B()$. Če ima več stanj enako $f_B()$ vrednost, so razvrščena glede na naraščajočo vrednost $f()$ in če ima več stanj še to vrednost enako, so razvrščena glede na naraščajočo vrednost $k()$. To pomeni, da je na seznamu *OPEN* za vsako stanje shranjen vektor z vrednostjo $\langle f_B(), f(), k() \rangle$.

Kadarkoli se stanje odstrani s seznama *OPEN*, se pogleda njegova vrednost funkcije$f()$, da se ugotovi, ali je bila izračunana na podlagi zadnje robotove lokacije. Če ni bila, se vrednosti $f()$ in $f_B()$ ponovno izračunata glede na robotovo trenutno lokacijo in stanje se ponovno doda nazaj na seznam. Jemanje stanj s seznama *OPEN* glede na naraščajočo vrednost $f_B()$ nam zagotavlja, da je stanje, ki ima pravilno vrednost $f()$ glede na trenutno robotovo lokacijo, tudi minimum vseh stanj, označen z f_{min}. Naj bo s k_{val} označena njegova vrednost funkcije $k()$. Ti dve vrednosti predstavljata pomemben prag pri algoritmu D*.

Če jemljemo s seznama *OPEN* stanja, ki imajo pravilno izračunano vrednost $f()$, glede na trenutno robotovo lokacijo, v naraščajočem vrstnem redu in če slučajno imata dve stanji enako vrednost $f()$, še naraščajočo vrednost glede na njun $k()$, potem nam algoritem zagotavlja, da za vsa stanja X, če velja $f(X) < f_{min}$ ali ($f(X) = f_{min}$ in $h(X) \leq k_{val}$), je $h(X)$ optimalen. Za namene tega testiranja se uporablja parameter *val*, ki je vektor vrednosti $\langle f_{min}, k_{val} \rangle$.

Naj bo R_{curr} trenutno stanje, na katerega je osredotočeno iskanje. Na začetku je R_{curr} enak začetnemu stanju robota. Definirajmo funkcijo *r(X)*, ki vrne stanje, v katerem je bil robot, ko smo na seznam *OPEN* dodali ali spremenili pozicijo stanja X. Parameter d_{curr} predstavlja vrednost *pristranske funkcije* od robotovega začetnega stanja do trenutnega stanja. Je drugo ime za $d(R_{curr}, R_0)$ in na začetku je inicializirana na $d_{curr} = d(R_0, R_0) = 0$.

3.4.4 Opis algoritma

Algoritem Focused D* sestoji podobno kot osnovni algoritem D* iz dveh glavnih funkcij, *PROCESS-STATE* in *MODIFY-COST*, ter funkcije *MOVE-ROBOT*. Funkcija *PROCESS-STATE* izračuna optimalno pot do cilja, *MODIFY-COST* popravi cene na povezavah in doda prizadeta stanja na seznam *OPEN*, *MOVE-ROBOT* pa uporabi zgoraj navedeni funkciji, da premika robota po optimalni poti. Programer, ki implementira algoritem, mora napisati še osredotočeno hevristično funkcijo *GVAL(X,Y)*. Ta se razlikuje od problema do problema. V naslednjih nekaj odstavkih bomo opisali vse tri glavne funkcije, ter tri funkcije, ki skrbijo za upravljanje s seznamom *OPEN*. Poleg teh funkcij potrebuje algoritem za svoje delovanje še pomožne funkcije:

- *MIN(a,b)*, ki vrne minimum dveh skalarnih števil a in b,
- *LESS(a,b)*, ki za vhod dobi dva vektorja, $\langle a_1, a_2 \rangle$ za a in $\langle b_1, b_2 \rangle$ za b, in vrne TRUE, če $a_1 < b_1$ ali ($a_1 = b_1$ in $a_2 < b_2$),
- *LESSEQ(a,b)*, ki za vhod dobi dva vektorja, $\langle a_1, a_2 \rangle$ za a in $\langle b_1, b_2 \rangle$ za b, in vrne TRUE, če $a_1 < b_1$ ali ($a_1 = b_1$ in $a_2 \leq b_2$),
- *COST(X)*, ki izračuna $f(X, R_{curr}) = h(X) + GVAL(X, R_{curr})$,
- *DELETE(X)*, ki odstrani stanje X s seznama *OPEN* in nastavi atribut *t(X) = CLOSED*,
- *PUT-STATE(X)*, ki nastavi atribut *t(X) = OPEN* in doda stanje X na pozicijo na seznamu *OPEN* glede na vektor $\langle f_B(), f(), k() \rangle$,

- *GET-STATE*, ki vrne stanje v seznamu *OPEN* z najmanjšo vrednostjo vektorja $\langle f_B(), f(), k()\rangle$, oziroma NULL, če je seznam prazen.

Funkcija *INSERT(X)* spremeni vrednost *h(X)* na novo vrednost h_{new} in doda ali spremeni pozicijo stanja X na seznamu *OPEN*.

Algoritem 9 INSERT

Vhod: Vozlišče X in vrednost nove h ocene
Izhod: Nič

```
 1: function INSERT(X,h_new)
 2:     if t(X) = NEW then
 3:         k(X) = h_new
 4:     else
 5:         if t(X) = OPEN then
 6:             k(X) = MIN(k(X),h_new); DELETE(X)
 7:         else
 8:             k(X) = MIN(h(X),h_new)
 9:         end if
10:     end if
11:     h(X) = h_new; r(X) = R_curr
12:     f(X) = k(X) + GVAL(X,R_curr); f_B(X) = f(X) + d_curr
13:     PUT-STATE(X)
14: end function
```

Funkcija *MIN-STATE()* vrne stanje na seznamu *OPEN* z najmanjšo $f()$ oceno. To naredi tako, da vrne stanje na seznamu *OPEN* z najmanjšo $f_B()$ oceno. Če je bilo to stanje dodano na seznam *OPEN*, ko je robot zavzemal neko drugo lokacijo kot jo zdaj, se to stanje ponovno doda na seznam *OPEN*. Funkcija *MIN-STATE()* to počne tako dolgo, dokler ne naleti na stanje, ki je bilo dodano na seznam *OPEN*, ko je bil robot na trenutni poziciji.

Funkcija *MIN-VAL* vrne vrednosti *f()* in *k()* tistega stanja na seznamu *OPEN*, ki ima najmanjšo *f()* oceno.

V funkciji *PROCESS-STATE* se prenesejo nove cene povezav in izračuna se nova pot. S seznama *OPEN* se odstrani stanje X, ki ima najmanjšo oceno *f()*. Če je X stanje *LOWER* ($k(X) = h(X)$), potem je njegova že izračunana pot optimalna. V nadaljevanju algoritma se razvijejo vsi sosedje Y stanja X, da se ugotovi, če se lahko njegova cena poti še zmanjša. Če je atribut njegovih sosedov Y enak NEW, potem ti prejmejo svojo začetno oceno poti in njihov kazalec se nastavi na stanje X ne glede na to, ali je nova cena poti večja ali

Algoritem 10 MIN-STATE

Vhod: Nič

Izhod: Stanje z najmanjšo f() oceno

```
1:  function MIN-STATE()
2:      while X = GET-STATE() ≠ NULL do
3:          if r(X) ≠ R_curr then
4:              h_new = h(X); h(X) = k(X)
5:              DELETE(X); INSERT(X,h_new)
6:          else
7:              return X
8:          end if
9:      end while
10:     return NULL
11: end function
```

Algoritem 11 MIN-VAL

Vhod: Nič

Izhod: Vektor $\langle f_{min}, k_{val} \rangle$ stanja z najmanjšo f() oceno

```
1: function MIN-VAL()
2:     X = MIN-STATE()
3:     if X = NULL then
4:         return NO-VAL
5:     else
6:         return ⟨f(X), k(X)⟩
7:     end if
8: end function
```

manjša od prejšnje cene. Glede na to, da so ta stanja potomci stanja X, ima kakršna koli sprememba cene poti stanja X vpliv tudi na njihovo ceno poti. Vsa sosednja stanja Y, ki prejmejo novo oceno poti, se dodajo na seznam *OPEN*, da se bodo kasneje tudi njihove spremembe cene poti prenesle do njihovih sosedov.

Če je X stanje *RAISE*, potem njegova cena poti ni nujno optimalna. Preden stanje X prenese ceno poti do svojih sosedov, se njegovi optimalni sosedje razvijejo, da se ugotovi, če se lahko ocena $h(X)$ zmanjša. Zatem se sprememba cene prenese do vseh sosedov, ki imajo svoj atribut enak *NEW* na enak način, kot se to naredi pri stanju *LOWER*. Če lahko stanje X zmanjša ceno poti tudi tistim svojim sosedom, katerih kazalci ne kažejo na stanje X, se stanje X doda na seznam *OPEN* za kasnejše razvitje. Ta ukrep je potreben, da se prepreči ustvarjanje zanke v kazalcih. Če se cena poti stanja X lahko zmanjša s pomočjo suboptimalnega soseda, se ta sosed doda na seznam *OPEN* in posodobitev cene poti stanja X se preloži, dokler sosed nima optimalne cene poti.

Funkcija *MODIFY-COST* posodobi ceno povezave med stanjema X in Y s ceno, ki jo dobi kot argument. Ker se bo cena poti stanja Y spremenila, se stanje X doda na seznam *OPEN*. Ko se potem stanje X razvije v funkciji *PROCESS-STATE*, se izračuna nov $h(Y) = h(X) + c(X, Y)$ in stanje Y se doda na seznam *OPEN*.

Funkcija *MOVE-ROBOT* ponazarja, kako se uporabljata funkciji *PROCESS-STATE* in *MODIFY-COST* za premik robota od stanja S do stanja G po optimalni poti. Na začetku se nastavi atribut $t()$ na vrednost *NEW* za vsa stanja. Nastavi se tudi parameter d_{curr} in trenutno stanje, na katerega je iskanje osredotočeno R_{curr}. Vrednost $h(G)$ je nastavljena na 0 in stanje G se doda na seznam *OPEN*. Zatem se v zanki kliče funkcija *PROCESS-STATE*, dokler ni izračunana pot od cilja G do trenutnega robotovega stanja S (t(S) = CLOSED) ali se ugotovi, da pot ne obstaja (val = NO-VAL in t(S) = NEW). Robot se nato prek kazalcev pomika proti cilju toliko časa, dokler ne prispe na cilj ali odkrije razliko med izmerjeno ceno povezave $s()$ in ceno povezave, ki jo ima v zemljevidu $c()$. Do razlike pride zaradi na novo odkrite ovire ali ugotovitve, da ovire sploh ni tam, kjer bi morala biti. Pokliče se funkcija *MODIFY-COST*, da popravi ceno povezave in doda prizadeta stanja na seznam *OPEN*. Nato se v zanki ponovno kliče funkcija *PROCESS-STATE*, da se prenesejo nove cene povezav in izračuna nova optimalna pot do cilja. Robot zatem spet sledi kazalcem do cilja. Funkcija vrne *GOAL-REACHED*, če je bila najdena pot od začetnega stanja do končnega stanja, oziroma *NO-PATH*, če ta ne obstaja.

Algoritem 12 PROCESS-STATE

Vhod: Nič

Izhod: Vrednost funkcije MIN-VAL()

```
1: function PROCESS-STATE()
2:     X = MIN-STATE()
3:     if X = NULL then
4:         return NO-VAL
5:     end if
6:     val = ⟨f(X), k(X)⟩; k_val = k(X); DELETE(X)
7:     if k_val < h(X) then
8:         for vsakega soseda Y od X do
9:             if t(Y) ≠ NEW in LESSEQ(COST(Y),val)) in
10:                h(X) > h(Y) + c(Y,X) then
11:                    b(X) = Y; h(X) = h(Y) + c(Y,X)
12:            end if
13:        end for
14:    end if
15:    if k_val = h(X) then
16:        for vsakega soseda Y od X do
17:            if t(Y) = NEW ali (b(Y) = X in h(Y) ≠ h(X) + c(X,Y)) ali
18:                (b(Y) ≠ X in h(Y) > h(X) + c(X,Y)) then
19:                    b(Y) = X; INSERT(Y, h(X) + c(X,Y))
20:            end if
21:        end for
22:    else
23:        for vsakega soseda Y od X do
24:            if t(Y) = NEW ali (b(Y) = X in h(Y) ≠ h(X) + c(X,Y)) then
25:                b(Y) = X; INSERT(Y, h(X) + c(X,Y))
26:            else
27:                if b(Y)≠X in h(Y)>h(X) + c(X,Y) in t(X)=CLOSED then
28:                    INSERT(X, h(X))
29:                else
30:                    if b(Y) ≠ X in h(X) > h(Y) + c(Y,X) in t(Y) = CLOSED
31:                        in LESS(val, COST(Y)) then
32:                            INSERT(Y, h(Y))
33:                    end if
34:                end if
35:            end if
36:        end for
37:    end if
38:    return MIN-VAL()
39: end function
```

Algoritem 13 MODIFY-COST

Vhod: Vozlišči X in Y ter vrednost nove cene povezave c_{val}
Izhod: Vrednost funkcije MIN-VAL() za seznam *OPEN*

```
1: function MODIFY-COST(X,Y,c_val)
2:     c(X,Y) = c_val
3:     if t(X) = CLOSED then
4:         INSERT(X,h(X))
5:     end if
6:     return MIN-VAL()
7: end function
```

Algoritem 14 MOVE-ROBOT

Vhod: Začetno stanje S in končno stanje G
Izhod: Sporočilo ali je bila pot najdena

```
 1: function MOVE-ROBOT(S,G)
 2:     for vsako stanje X v grafu do
 3:         t(X) = NEW
 4:     end for
 5:     d_curr = 0; R_curr = S
 6:     INSERT(G,0)
 7:     val = ⟨0, 0⟩
 8:     while t(S) ≠ CLOSED in val ≠ NO-VAL do
 9:         val = PROCESS-STATE()
10:     end while
11:     if t(S) = NEW then return NO-PATH
12:     end if
13:     R = S
14:     while R ≠ G do
15:         if s(X,Y) ≠ c(X,Y) za kateri (X,Y) then
16:             if R_curr ≠ R then
17:                 d_curr = d_curr + GVAL(R, R_curr) + ϵ; R_curr = R
18:             end if
19:             for vsak (X,Y) če s(X,Y) ≠ c(X,Y) do
20:                 val = MODIFY-COST(X, Y, s(X,Y))
21:             end for
22:             while LESS(val, COST(R)) in val ≠ NO-VAL do
23:                 val = PROCESS-STATE()
24:             end while
25:         end if
26:         R = b(R)
27:     end while
28:     return GOAL-REACHED
29: end function
```

Poglavje 4

Lifelong Planning A*

Naslednji algoritem, ki ga želimo opisati kot enega izmed algoritmov D*, se imenuje D* Lite. Čeprav ima ta algoritem enako obnašanje kot D*, je izpeljan iz algoritma Lifelong Planning A*. Zato je na mestu, da si pred opisom algoritma D* Lite pobliže ogledamo algoritem Lifelong Planning A*, ki je bil opisan leta 2002 v članku [14].

Algoritem Lifelong Planning A* (LPA*) je inkrementalna verzija algoritma A*, ki uporablja hevristiko za iskanje poti in vedno najde optimalno pot.[1] Ko LPA* prvikrat preiskuje graf, je to enako hitro kot pri A*, vsa nadaljnja preiskovanja pa so precej hitrejša. Znatna pohitritev je možna zaradi tega, ker v nasprotju z A* pri drugem in vseh nadaljnjih preiskovanjih grafa ponovno, brez računanja, uporabi tiste dele v grafu, ki se ne spremenijo.

4.1 Opis algoritma

Pri opisu algoritma bomo uporabili naslednje oznake: S označuje končno množico vozlišč v grafu, $Succ(s) \subseteq S$ označuje množico naslednikov vozlišča $s \in S$. Podobno $Pred(s) \subseteq S$ označuje predhodnike vozlišča $s \in$ S. c(s,s') označuje ceno, ki je potrebna za premik od vozlišča s do vozlišča $s' \in Succ(s)$. Tu velja $0 < c(s, s') \leq \infty$. Začetno vozlišče je označeno s $s_{start} \in S$, končno pa s $s_{goal} \in S$.

Tako kot pri A*, tudi tukaj hevristična funkcija $h(s,s_{goal})$ ocenjuje dolžino poti, ki je potrebna za premik od vozlišča s do ciljnega vozlišča. Če želimo, da algoritem LPA* vedno najde optimalno pot, mora biti hevristika $h(s,s_{goal})$ konsistentna, kar pomeni, da mora veljati:

[1] Ob predpostavki, da je hevristična funkcija optimistična

$$\begin{aligned} h(s_{goal}, s_{goal}) &= 0 \\ h(s, s_{goal}) &\leq c(s, s') + h(s', s_{goal}) \end{aligned} \tag{4.1}$$

za vsa vozlišča $s \in S$ in $s' \in Succ(s)$, pri čemer $s \neq s_{goal}$.

LPA* vzdržuje tudi cene *g(s)*, ki so enake g cenam algoritma A*. Te cene se prenašajo med posameznimi iteracijami iskanja. Poleg teh cen algoritem LPA* vzdržuje tudi drugo vrsto cene, za določitev cene poti od začetnega do trenutnega vozlišča *s*. Imenuje se vrednost *rhs*. Za razliko od *g cene* vrednost *rhs* gleda en korak naprej in zato vsebuje več informacije kot *g cena*. Vrednost *rhs* je definirana kot

$$rhs(s) = \begin{cases} 0 & \text{če } s = s_{start}, \\ min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{sicer} \end{cases} \tag{4.2}$$

Vozlišču se reče, da je lokalno konsistentno, če ima enaki vrednosti $g(s)$ in $rhs(s)$. To je pomembno, saj če so vsa vozlišča konsistentna, potem je *g cena* vseh vozlišč enaka optimalni ceni poti od začetnega vozlišča do posameznega vozlišča. Vendar pa algoritem LPA* ne zagotovi vsem vozliščem lokalne konsistentnosti. Namesto tega uporabi hevristično funkcijo $h(s, s_{goal})$ za vodenje preiskovanja in posodobi samo tiste *g cene* vozlišč, ki so pomembne za izračunanje najcenejše poti od začetnega vozlišča do ciljnega vozlišča.

Algoritem LPA* skozi svoje delovanje vzdržuje prioritetno vrsto *U*, ki vedno vsebuje le tista vozlišča, ki *niso* lokalno konsistentna. To so vozlišča, katerih *g cena* mora biti posodobljena, da bodo lokalno konsistentna. Ključ, po katerem so urejena vozlišča v prioritetni vrsti *U*, je *f ocena*, ki je enaka f oceni algoritma A*. Algoritem LPA*, podobno kot A*, vedno razvije najprej tisto vozlišče, ki ima najmanjšo *f oceno* v prioritetni vrsti *U*. Ključ *k(s)* vozlišča *s*, je vektor z dvema komponentama:

$$k(s) = \langle k_1(s); k_2(s) \rangle, \tag{4.3}$$

kjer je $k_1(s) = min(g(s), rhs(s)) + h(s)$ in $k_2(s) = min(g(s), rhs(s))$.

4.2 Delovanje algoritma

Glavna funkcija algoritma LPA* je funkcija *Main()*, ki najprej pokliče funkcijo *Initialize()*, da se vzpostavi začetno stanje problema. Funkcija *Initialize()* nastavi *g ceno* vsem vozliščem na neskončno in njihovo vrednost *rhs* glede na

Algoritem 15 CalcKey

Vhod: Vozlišče $s \in S$
Izhod: Ključ $\langle k1(s), k2(s) \rangle$

1: **function** CalcKey(s)
2: **return** $\langle (min(g(s), rhs(s)) + h(s, s_{goal}); min(g(s), rhs(s)) \rangle$
3: **end function**

Algoritem 16 Initialize

Vhod: Nič
Izhod: Nič

1: **function** Initialize()
2: U = $\emptyset$
3: **for** vse $s \in S$ **do**
4: rhs(s) = g(s) = ∞
5: **end for**
6: rhs(s_{start}) = 0
7: U.Insert(s_{start}, CalcKey(s_{start}))
8: **end function**

Algoritem 17 UpdateVertex

Vhod: Povezava u $\in$ G
Izhod: Nič

1: **function** UpdateVertex(u)
2: **if** u $\neq s_{start}$ **then**
3: rhs(u) = $min_{s' \in Pred_{(u)}}(g(s') + c(s', u))$
4: **end if**
5: **if** u $\in$ U **then**
6: U.Remove(u)
7: **end if**
8: **if** g(u) $\neq$ rhs(u) **then**
9: U.Insert(u, CalcKey(u))
10: **end if**
11: **end function**

Algoritem 18 ComputeShortestPath

```
Vhod: Nič
Izhod: Nič
 1: function ComputeShortestPath()
 2:     while U.TopKey() < CalcKey(s_goal) ali rhs(s_goal) ≠ g(s_goal) do
 3:         u = U.Pop()
 4:         if g(u) > rhs(u) then
 5:             g(u) = rhs(u)
 6:             for vse s ∈ Succ(u) do
 7:                 UpdateVertex(s)
 8:             end for
 9:         else
10:             g(u) = ∞
11:             for vse s ∈ Succ(u) ∪ {u} do
12:                 UpdateVertex(s)
13:             end for
14:         end if
15:     end while
16: end function
```

Algoritem 19 Main

```
Vhod: Nič
Izhod: Nič
 1: function Main()
 2:     Initialize()
 3:     while True do
 4:         ComputeShortestPath()
 5:         Počakaj na spremembe cen na povezavah
 6:         for vse povezave (u,v), ki imajo spremembo v ceni do
 7:             Posodobi ceno povezave c(u,v)
 8:             UpdateVertex(u)
 9:         end for
10:     end while
11: end function
```

enačbo 4.2. Iz tega sledi, da je začetno vozlišče edino vozlišče, ki je lokalno nekonsistentno in je zato dodano v prazno prioritetno vrsto *U*. Za funkcijo *Initialize()* glavna funkcija *Main()* pokliče funkcijo *ComputeShortestPath()*, da se poišče najkrajša pot od začetnega do končnega vozlišča. Funkcija *ComputeShortestPath()* preračuna *g ceno* vsem lokalno nekonsistentnim vozliščem v nepadajočem vrstnem redu glede na njihov ključ.

Lokalno nekonsistentno vozlišče s, se imenuje *lokalno nadkonsistentno*, če velja $g(s) > rhs(s)$. Ko funkcija *ComputeShortestPath()* razvije vozlišče, ki je lokalno nadkonsistentno, potem velja, da $rhs(s) = g^*(s)$, kar pomeni, da velja $k(s) = \langle f(s); g^*(s)\rangle$, kje je $f(s) = g^*(s) + h(s, s_{goal})$. Med razvitjem vozlišča funkcija *ComputeShortestPath()* nastavi *g ceno* vozlišča na vrednost njegove *rhs* vrednosti, kar pomeni, da je *g cena* enaka vsoti cene poti od začetnega vozlišča s_{start} do tega vozlišča in s tem naredi vozlišče lokalno konsistentno. Njegova *g cena* se nato ne spreminja več, razen če se funkcija *ComputeShortestPath()* pokliče ponovno.

Lokalno nekonsistentno vozlišče s se imenuje *lokalno podkonsistentno*, če velja $g(s) < rhs(s)$. Ko funkcija *ComputeShortestPath()* razvije vozlišče, ki je lokalno podkonsistentno, potem zgolj nastavi njegovo *g ceno* na neskončno. To naredi vozlišče bodisi lokalno konsistentno bodiso nadkonsistentno.

Če je bilo razvito vozlišče lokalno nadkonsistentno, potem lahko sprememba njegove *g cene* vpliva na lokalno konsistentnost njegovih naslednikov. Podobno velja tudi za vozlišče, ki je lokalno podkonsistentno. Da se ohranja skladnost vrednosti *rhs*, prioritetne vrste in funkcije *k(s)*, funkcija *ComputeShortestPath()* posodobi vozliščem njihove vrednosti *rhs*, preveri njihovo lokalno konsistentnost in jih po potrebi doda ali odstrani iz prioritetne vrste.

Algoritem LPA* razvija vozlišča, dokler ni ciljno vozlišče s_{goal} lokalno konsistentno in ključ *k(s)* naslednjega vozlišča, ki bi moralo biti razvito, ni manjši od ključa $k(s_{goal})$ končnega vozlišča. Če $k(s_{goal}) = \infty$, potem ne obstaja pot od začetnega do končnega vozlišča. V nasprotnem primeru algoritem LPA* najde najkrajšo možno pot od začetnega do končnega vozlišča na naslednji način: algoritem se vedno premakne od trenutnega vozlišča s, začetek je v s_{goal}, do njegovega predhodnika *s'*, ki minimizira vrednost $g(s') + c(s', s)$, dokler ni doseženo začetno vozlišče s_{start}. Ko je bilo začetno vozlišče najdeno, je najdena tudi najkrajša pot med začetnim in končnim vozliščem, ki je enaka obratni poti, kot jo je prehodil algoritem.

Ko se funkcija *ComputeShortestPath()* konča, glavna funkcija *Main()* čaka na morebitne spremembe cen na povezavah. Če se to zgodi, je treba, da se ohrani skladnost vrednosti *rhs*, prioritetne vrste in funkcije *k(s)*, poklicati funkcijo *UpdateVertex()*, da se posodobijo vrednosti *rhs* in ključi vozlišč, ki

so bili morebiti prizadeti zaradi spremembe cene na povezavah. Zatem se ponovno pokliče funkcija *ComputeShortestPath()* in ko se ta konča, funkcija *Main()* spet čaka na morebitne spremembe cen na povezavah.

4.3 Analitične lastnosti algoritma

S pomočjo spodaj navedenih izrekov bomo pokazali, da se algoritem LPA* vedno ustavi ter da je pravilen in učinkovit. Dokaze izrekov lahko bralec najde v [14].

4.3.1 Ustavljivost in pravilnost

Izrek 4.3.1. Funkcija *ComputeShortestPath()* razvije vsako vozlišče kvečjemu dvakrat. Če je neko vozlišče lokalno podkonsistentno, ga razvije kvečjemu dvakrat, če pa je neko vozlišče lokalno nadkonsistentno, ga razvije kvečjemu enkrat in zato se funkcija tudi vedno ustavi.

4.3.2 Podobnost z A*

Že v opisu samega algoritma LPA* se lahko vidi precejšnja podobnost z algoritmom A*. Izrek 4.3.1 nam pravi, da funkcija *ComputeShortestPath()* razvije vsako vozlišče kvečjemu dvakrat, kar je podobno algoritmu A*, ki razvije vsako vozlišče kvečjemu enkrat. Naslednji izrek pravi, da imajo ključi vozlišč, ki jih zaporedoma razvija funkcija *ComputeShortestPath()*, monotono nepadajočo vrednost. To je spet podobno algoritmu A*, kjer imajo *f ocene* vozlišč, ko jih algoritem zaporedoma razvija, nepadajočo vrednost.

Izrek 4.3.2. Ključi vozlišč, ki jih zaporedoma razvija funkcija *ComputeShortestPath()*, imajo monotono nepadajočo vrednost, vse dokler se funkcija *ComputeShortestPath()* ne ustavi.

Iz naslednjih treh izrekov je razvidno, da funkcija *ComputeShortestPath()* razvije lokalno nadkonsistentna vozlišča na zelo podoben način, kot to stori algoritem A*. Izrek 4.3.3 nam pokaže, da je prva komponenta ključa, lokalno nadkonsistentnega vozlišča, v času, ko ga funkcija *ComputeShortestPath()* razvije, enaka *f oceni* tega vozlišča. Druga komponenta njegovega ključa je razdalja do začetnega vozlišča.

Izrek 4.3.3. Kadarkoli določi funkcija *ComputeShortestPath()* lokalno nadkonsistentno vozlišče s za razvitje, je njegov ključ enak vektorju

$$k(s) = \langle f(s); g^*(s) \rangle$$

Izreka 4.3.2 in 4.3.3 govorita o tem, da funkcija *ComputeShortestPath()* razvija lokalno nadkonsistentna vozlišča v monotonem nepadajačem vrstem redu glede na njihove *f ocene* in vozlišča z enakimi *f ocenami* v monotonem nepadajočem vrstnem redu glede na njihove razdalje od začetnega vozlišča. Podobno tudi algoritem A* upošteva razdaljo vozlišč od začetnega vozlišča, če imata dve ali več vozlišč enako *f oceno*.

Izrek 4.3.4. Funkcija *ComputeShortestPath()* razvija lokalno nadkonsistentna vozlišča, s končno *f oceno*, v enakem vrstem redu, kot to počne algoritem A*.

Naslednji izrek pravi, da funkcija *ComputeShortestPath()* razvije samo tista lokalno nadkonsistentna vozlišča, ki imajo *f oceno* manjšo, kot je *f ocena* ciljnega vozlišča, če pa je njihova *f ocena* enaka f oceni ciljnega vozlišča, razvije samo tista vozlišča, ki imajo krajšo ali enako razdaljo do začetnega vozlišča, kot jo ima ciljno vozlišče. To velja tudi za algoritem A*.

Izrek 4.3.5. Funkcija *ComputeShortestPath()* razvije samo tista lokalno nadkonsistentna vozlišča s, za katera velja: $\langle f(s); g^*(s) \rangle \leq \langle f(s_{goal}); g^*(s_{goal}) \rangle$.

4.3.3 Učinkovitost

Pokazali bomo, da v resnici algoritem LPA* razvije veliko manj vozlišč, kot to namiguje izrek 4.3.1. Naslednji izrek pravi, da je algoritem LPA* učinkovit zato, ker uporablja inkrementalno iskanje in ker izračunava samo tiste *g cene* vozlišč, ki so prizadete zaradi spremembe cen na povezavah ali pa niso bile izračunane že pri prejšnjih iskanjih.

Izrek 4.3.6. Funkcija *ComputeShortestPath()* ne razvije tistih vozlišč, katerih *g cena* je enaka dejanski razdalji do začetnega vozlišča, preden je bila funkcija *ComputeShortestPath()* poklicana.

Naš zadnji izrek pokaže, da je algoritem LPA* učinkovit tudi zaradi uporabe hevristike pri iskanju, ker zaradi tega izračunava samo tiste *g cene* vozlišč, ki so dejansko pomembne za določitev najkrajše poti. Že izrek 4.3.5 nam je pokazal, kako hevristika omeji število lokalno nadkonsistentnih vozlišč, ki jih mora funkcija *ComputeShortestPath()* razviti. Naslednji izrek posploši to trditev tudi na lokalno nekonsistentna vozlišča, ki jih mora funkcija *ComputeShortestPath()* razviti.

Izrek 4.3.7. Ključi vozlišč, ki jih funkcija *ComputeShortestPath()* določi za razvitje, nikoli ne presegajo vrednosti vektorja $\langle f(s_{goal}); g^*(s_{goal})\rangle$.

Za boljše razumevanje zgornjega izreka je dobro ponoviti, da je ključ $k(s)$ vozlišča s definiran kot

$$k(s) \doteq \langle min(g(s), rhs(s)) + h(s, s_{goal}); min(g(s), rhs(s))\rangle$$

To pomeni, da bolj kot bo neka hevristika informirana, večja bo vrednost ključa in *manj* bo vozlišč, ki bodo zadostila pogoju: $k(s) \leq \langle f(s_{goal}); g^*(s_{goal})\rangle$ in zato jih bo treba razviti manj.

Poglavje 5

D* Lite

Algoritem LPA*, ki smo ga opisali v prejšnjem poglavju, znova in znova določa najkrajšo pot med začetnim in končnim vozliščem, kadar se zgodi sprememba cene nekega vozlišča v grafu. Algoritem D* Lite, ki bo opisan v tem poglavju, je izpeljan iz algoritma LPA*, le da znova in znova določa najkrajšo pot med trenutnim vozliščem, na katerem je robot, in kočnim vozliščem, ko se zgodi sprememba cene nekega vozlišča v grafu med premikanjem robota proti končnemu vozlišču.

Algoritem D* Lite ne dela nobenih predpostavk o tem, kako se bodo spreminjale cene povezav, ali se bodo povišale ali znižale, ali bodo v bližini robota ali daleč stran, ter ali so se zgodile zaradi sprememb v svetu ali zaradi robotovih napačnih začetnih ocen. Algoritem D* Lite se ponavadi uporablja za reševanje navigacijskih problemov na neznanem terenu.

Algoritem D* Lite je delo dveh avtorjev, Svena Koeninga in Maxima Likhacheva ter je bil prvič predstavljen leta 2002 v članku [15].

5.1 Smer preiskovanja

Smer preiskovanja je nasprotna od smeri, ki jo za preiskovanje uporablja algoritem LPA*, kar pomeni, da algoritem vodi preiskovanje od končnega vozlišča proti začetnemu vozlišču. Algoritem LPA* preiskuje od začetnega vozlišča proti končnemu vozlišču in posledično njegove *g cene* predstavljajo ceno poti od začetnega do trenutnega vozlišča. D* Lite preiskuje od končnega vozlišča proti začetnemu vozlišču in zato njegove *g cene* predstavljajo ceno poti od končnega vozlišča do trenutnega vozlišča. Algoritem D* Lite je izpeljan iz algoritma LPA* tako, da se v psevdokodi obrnejo začetno in končno vozlišče ter vse povezave, ki tam nastopajo. Ko funkcija *ComputeShortestPath()* vrne rezul-

tat, lahko poiščemo najkrajšo pot od začetnega do končnega vozlišča tako, da začnemo v začetnem vozlišču s_{start} in se nato, dokler ne prispemo do končnega vozlišča s_{goal}, vedno premaknemo v tisto sosednje vozlišče, ki minimizira vrednost $c(s, s') + g(s')$, kjer je s' sosed trenutnega vozlišča s.

5.2 Delovanje algoritma

Da robot doseže želeno končno vozlišče iz začetnega vozlišča, funkcija *Main()* robota premika po poti, ki je bila določena s funkcijo *ComputeShortestPath()*. Funkcija *Main()* bi lahko preračunala prioritete (vrstni red) vozlišč v prioritetni vrsti vsakokrat, ko bi robot s senzorji zaznal spremembe na cenah povezav. Čeprav bi bile prioritete preračunane, ne bi izpolnile kriterija, da je prioriteta posameznega vozlišča, ki je v prioritetni vrsti, enaka njegovemu ključu $k(s) = \langle k_1(s); k_2(s) \rangle$, saj bi temeljile na hevristiki, izračunani glede na prejšnje vozlišče, na katerem je bil robot. Poleg tega je ponavljajoče se prerazporejanje vozlišč v prioritetni vrsti časovno zahtevno, saj je v njem ponavadi veliko število vozlišč.

Algoritem 20 CalculateKey

Vhod: Vozlišče $s \in S$
Izhod: Ključ $\langle k1(s), k2(s) \rangle$

1: **function** CALCULATEKEY(s)
2:     **return** $\langle (min(g(s), rhs(s)) + h(s, s_{goal} + k_m); min(g(s), rhs(s)) \rangle$
3: **end function**

Algoritem 21 Initialize

Vhod: Nič
Izhod: Nič

1: **function** INITIALIZE()
2:     U = $\emptyset$
3:     $k_m = 0$
4:     **for** vse $s \in S$ **do**
5:         rhs(s) = g(s) = ∞
6:     **end for**
7:     rhs(s_{goal}) = 0
8:     U.Insert(s_{goal}, $\langle h(s_{goal}, s_{goal}); 0 \rangle$)
9: **end function**

Algoritem D* Lite zato uporabi metodo, izpeljano iz algoritma D*, da se izogne prerazporejanju vozlišč v prioritetni vrsti. Hevristična funkcija $h(s, s')$ mora biti nenegativna in veljati mora: $h(s, s') \leq c^*(s, s')$ in $h(s, s'') \leq h(s, s') + h(s', s'')$ za vsa vozlišča $s, s', s'' \in S$, kjer $c^*(s, s')$ predstavlja ceno najkrajše povezave med vozliščema $s \in S$ ter $s' \in S$.

Ko se robot premakne iz vozlišča s v neko vozlišče s', kjer zazna spremembe cen na povezavah, se prva komponenta njegovega ključa $k(s)$ lahko poveča kvečjemu za $h(s, s')$. Na drugo komponento hevristika nima vpliva, zato ostane nespremenjena. To pomeni, da je treba za ohranitev pravilnosti spodnjih mej posameznih vozlišč od vseh vozlišč, ki so v prioritetni vrsti, njihovi prvi komponenti v ključu odšteti vrednost $h(s, s')$. Ker je vrednost hevristične funkcije $h(s, s')$ enaka za vsa vozlišča, ki so v prioritetni vrsti, ni treba popravljati vrstnega reda vozlišč, če odštevanje ni bilo izvedeno.

Algoritem 22 UpdateVertex

Vhod: Povezava u ∈ G
Izhod: Nič

```
1: function UpdateVertex(u)
2:     if g(u) ≠ rhs(u) in u ∈ U then
3:         U.Update(u, CalculateKey(u))
4:     end if
5:     if g(u) ≠ rhs(U) in u ∉ U then
6:         U.Insert(u, CalculateKey(u))
7:     end if
8:     if g(u) = rhs(U) in u ∈ U then
9:         U.Remove(u)
10:    end if
11: end function
```

Če pride do preračunavanja ključev vozlišč, je njihova prva komponenta za vrednost $h(s, s')$ manjša, gledano relativno na prioritete ostalih vozlišč v prioritetni vrsti. Zato je treba, kadarkoli se zgodi sprememba cene na povezavah, k prvi komponenti ključa prišteti vrednost $h(s, s')$.

Ko se robot spet premakne in če spet naleti na spremembo cen na povezavah, je treba prvo komponento ključa vozlišč povečati za konstanto k_m. Ker to naredimo vsem vozliščem, se ne spremeni vrstni red vozlišč v prioritetni vrsti in prioritetne vrste ni treba preurediti.

Vrednosti ključev vozlišč pri algoritmu D* Lite so vedno spodnje meje vrednosti ključev vozlišč algoritma LPA*, saj se prvi komponenti ključa algoritma

LPA* odšteje trenutna vrednost konstante k_m. To lastnost izkoristimo tako, da spremenimo funkcijo *ComputeShortestPath()* na sledeči način:

Ko funkcija *ComputeShortestPath()* odstrani vozlišče s z najmanjšim ključem $k_{old} = U.TopKey()$ iz prioritetne vrste, pokliče funkcijo *CalculateKey()*, da ta izračuna novo vrednost ključa, ki bi jo vozlišče s moralo imeti. Če velja, da je $k_{old} < CalculateKey(u)$, potem funkcija *ComputeShortestPath()* posodobi vrednost ključa vozlišča u z vrednostjo, ki jo izračuna funkcija *CalculateKey(u)* in ga vrne v prioritetno vrsto. S tem se zagotovi, da so vrednosti ključev vozlišč, ki so v prioritetni vrsti, spodnje meje vrednosti ključev algoritma LPA*, potem ko so bile prve komponente ključa pri algoritmu LPA* povečane za vrednost k_m.

Če velja $k_{old} \geq CalculateKey(u)$, potem je k_{old} kar enak vrednosti, ki jo vrne *CalculateKey(u)* ($k_{old} = CalculateKey(u)$), saj je k_{old} spodnja meja, ki jo vrne funkcija *CalculateKey()*. V tem primeru razvije funkcija *ComputeShortestPath()* vozlišče na enak način, kot to stori algoritem *LPA**.

5.3 Pohitritev algoritma

Algoritem D* Lite lahko pohitrimo, če izboljšamo učinkovitost funkcije *ComputeShortestPath()*. Eden od načinov, kako to storimo je, da popravimo ustavitveni pogoj funkcije. Osnovna različica funkcije *ComputeShortestPath()* se ustavi, ko je vozlišče lokalno konsistentno in je njegov ključ manjši ali enak *U.TopKey()*. Vendar pa se lahko funkcija *ComputeShortestPath()* ustavi že, če ključ vozlišča ni lokalno podkonsistenten in je njegov ključ manjši ali enak *U.TopKey()*. Da bi razumeli, zakaj je to tako, privzamimo, da je začetno vozlišče lokalno nadkonsistentno in je njegov ključ manjši ali enak *U.TopKey()*. Potem mora njegov ključ biti enak *U.TopKey()*, saj je *U.TopKey()* najmanjši ključ kateregakoli lokalno nekonsistentnega vozlišča. Zato lahko funkcija *ComputeShortestPath()* kot naslednje vozlišče razvije ravno začetno vozlišče in nastavi njegovo *g oceno* na vrednost, ki jo ima njegova vrednost *rhs*. S tem postane začetno vozlišče lokalno konsistentno in njegov ključ je manjši ali enak *U.TopKey()* in zato se funkcija *ComputeShortestPath()* ustavi. V tem trenutku je *g ocena* začetnega vozlišča enaka razdalji do končnega vozlišča.

Ampak funkcija *ComputeShortestPath()* se lahko ustavi, tudi če začetno vozlišče ni lokalno podkonsistentno in je njegov ključ manjši ali enak *U.TopKey()*. V tem primeru lahko ostane začetno vozlišče lokalno nekonsistentno tudi po izhodu iz funkcije *ComputeShortestPath()* in njegova *g ocena* morda ni enaka pravi razdalji do kočnega vozlišča (ampak njegova vrednost *rhs* je), kar pa ni

Algoritem 23 ComputeShortestPath

Vhod: Nič
Izhod: Nič

```
1: function ComputeShortestPath()
2:     while U.TopKey() < CalcKey(s_start) ali rhs(s_start) ≠ g(s_start) do
3:         u = U.Top()
4:         k_old = U.TopKey()
5:         k_new = CalculateKey()
6:         if k_old < k_new then
7:             U.Update(u, k_new)
8:         else
9:             if g(u) > rhs(u) then
10:                g(u) = rhs(u)
11:                U.remove(u)
12:                for vse s ∈ Pred(u) do
13:                    if s ≠ s_goal then
14:                        rhs(s) = min(rhs(s), c(s,u) + g(u))
15:                    end if
16:                    UpdateVertex(s)
17:                end for
18:            else
19:                g_old = g(u)
20:                g(u) = ∞
21:                for vse s ∈ Pred(u) ∪ {u} do
22:                    if s ≠ s_goal then
23:                        rhs(s) = min_{s'∈Succ(s)}(c(s, s') + g(s'))
24:                    end if
25:                    UpdateVertex(s)
26:                end for
27:            end if
28:        end if
29:    end while
30: end function
```

Algoritem 24 Main

Vhod: Nič

Izhod: Nič

1: **function** MAIN()
2: $s_{last} = s_{start}$
3: Initialize()
4: ComputeShortestPath()
5: **while** $s_{start} \neq s_{goal}$ **do**
6: $s_{start} = argmin_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$
7: Premakni se v s_{start}
8: S senzorji preveri, če je prišlo do sprememb na cenah povezav
9: **if** je bila zaznana sprememba cene **then**
10: $k_m = k_m + h(s_{last}, s_{start})$
11: $s_{last} = s_{start}$
12: **for** vse povezave (u, v) na katerih so se spremenile cene **do**
13: c_{old} = c(u, v)
14: Posodobi ceno povezave c(u, v)
15: **if** $c_{old} > c(u, v)$ **then**
16: **if** u $\neq s_{goal}$ **then**
17: rhs(u) = min(rhs(u),c(u,v) + g(v))
18: **end if**
19: **else**
20: **if** rhs(u) = $c_{old} + g(v)$ **then**
21: **if** u $\neq s_{goal}$ **then**
22: rhs(s) = $min_{s' \in Succ(u)}(c(u, s') + g(s'))$
23: **end if**
24: **end if**
25: **end if**
26: UpdateVertex(u)
27: **end for**
28: ComputeShortestPath()
29: **end if**
30: **end while**
31: **end function**

problem, saj se *g cena* ne uporabi za določanje naslednjega premika robota.

5.4 Izrek o ustavljivosti algoritma

Funkcija *ComputeShortestPath()* algoritma D* Lite je zelo podobna funkciji *ComputeShortestPath()* algoritma LPA* in zaradi tega deli z njo veliko njenih lastnosti. Ena od teh je, da funkcija *ComputeShortestPath()* razvije vsako vozlišče največ dvakrat, preden se zaključi. Naslednji izrek pravi, da se funkcija *ComputeShortestPath()* algoritma D* Lite vedno ustavi in je pravilna.

Izrek 5.4.1. Funkcija *ComputeShortestPath()* algoritma D* Lite se vedno ustavi in najkrajša pot od začetnega vozlišča s_{start} do končnega vozlišča s_{goal} se poišče tako, da se zmeraj premaknemo iz trenutnega vozlišča s, začenši s s_{start}, do tistega njegovega naslednika s', da bo vrednost $c(s, s') + g(s')$ minimalna, dokler ne dosežemo vozlišča s_{goal}.

Poglavje 6

Eksperimentalna primerjava algoritmov

Za potrebe tega diplomskega dela smo implementirali in testirali štiri opisane algoritme: Osnovni D*, Focused D*, LPA* in D* Lite. Prva dva in zadnji algoritem od omenjenih spadajo v isto družino in so bili tudi osrednji algoritmi v našem diplomskem delu, medtem ko je algoritem LPA* tukaj zgolj za primerjavo z algoritmom D* Lite, saj si delita veliko skupnih lastnosti.

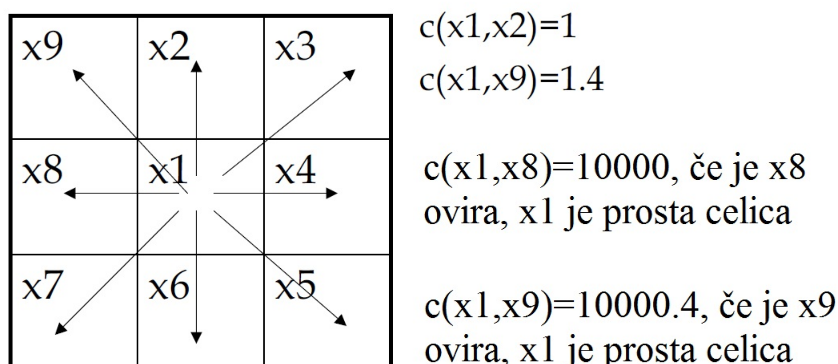
V tem poglavju bomo sprva opisali sestavine našega testiranja, kasneje pa podali še dobljene rezultate.

6.1 Svet v obliki mreže

Svet v obliki mreže je sestavljen iz celic, ki imajo povezave do svojih sosedov. V našem primeru je vsaka celica prosta celica ali pa predstavlja oviro. Povezava med dvema celicama je lahko enosmerna ali dvosmerna. V našem primeru je povezava dvosmerna, saj se robot lahko premika med dvema celicama poljubnokrat. V naši mreži ima vsaka celica, razen tistih na robu, 8 povezav do sosedov (gor, dol, levo, desno in 4 diagonalne povezave). Vsaka povezava ima tudi svojo ceno, ki se prišteje h končni poti, če se robot premakne po njej. V našem primeru, kot je to tudi prikazano na sliki 6.1, imajo povezave, ki vodijo do dosegljivih sosedov, ceno 1 in ceno 1.4, če so povezave diagonalne. Če sosednja povezava ni dosegljiva (celica predstavlja oviro), potem je njena cena enaka 10000 oziroma 10000.4, če je to diagonalna povezava. Ta cena ima toliko višjo vrednost zato, da je seštevek vseh povezav, ki predstavljajo mrežo, manjši od cene povezave, ki vodi do ovire. Če ne bi bilo tako, bi lahko v nekaterih izjemnih primerih algoritem našel krajšo pot, če gre robot skozi oviro,

kot pa če gre po poti, na kateri ni nobene ovire.

Za potrebe naših testiranj smo generirali različne velikosti mrež. Uporabili smo mreže velikosti 30 x 30, 60 x 60, 70 x 70, 100 x 100 in 120 x 120 celic. Vsem celicam smo že v času generiranja določili njihove cene povezav, ki vodijo do njihovih sosedov.



Slika 6.1: 8-smerna povezanost celic in cene povezav.

Začetno vozlišče smo postavili na sredino skrajnega levega konca mreže, končno vozlišče pa na sredino skrajnega desnega konca mreže. Tako bi bila v tem primeru, če na mreži ne bi bilo ovir, najkrajša pot kar vodoravna črta med tema dvema vozliščema, s ceno enako dolžini celotne mreže - 1 (pri mreži velikosti 30 x 30, bi bila cena 29 enot).

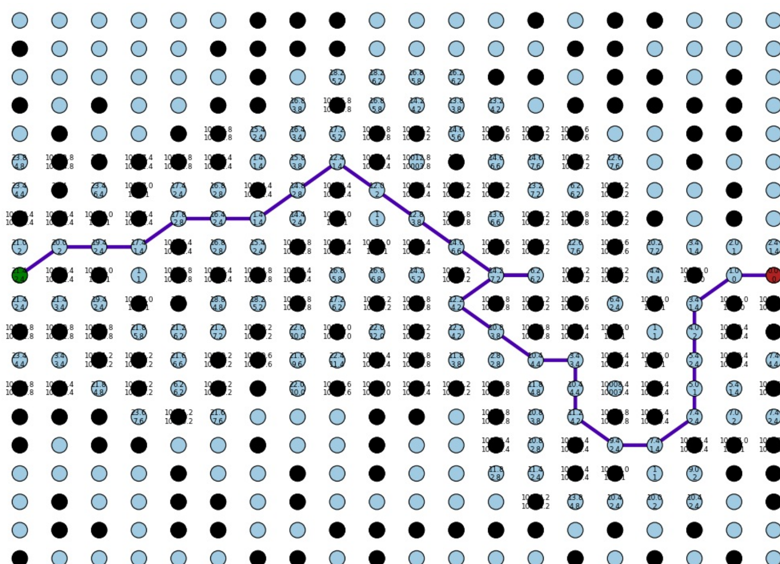
V vsakem testu smo nekatere celice že med generiranjem označili kot ovire. To je pomenilo, da so cene povezav iz teh celic do njihovih sosedov imele povišano ceno. Število teh celic se je gibalo med 30 % in 50 %.

6.2 Zaznavanje sveta s senzorji

Ker imamo opravka z algoritmi, ki omogočajo hitro iskanje najkrajših poti v okoljih, ki se dinamično spreminjajo, smo morali tudi to dejstvo na nek način simulirati. Naš robot je bil opremljen s senzorji, kar pomeni, da je imel sposobnost pogledati in preveriti ceno povezave za dve celici vstran od svoje trenutne lokacije, na vseh 8 strani. Povedano drugače, preveril je lahko vse cene povezav svojih sosedov in cene povezav sosedov od svojih sosedov. Torej je bila njegova globina gledanja enaka 2.

Preden smo pognali posamezen algoritem, ki smo ga testirali, smo ustvarili seznam celic, ki jih je robot kasneje, če je prišel v njihovo bližino, zaznal kot ovire. To je pomenilo, da so nekatere celice v resnici ovire, čeprav prvotni zemljevid tega ni vseboval in jih je označil kot proste. Nekatere celice na tem seznamu pa so se skladale z ovirami, ki so bile označene na začetnem zemljevidu.

Vsakokrat, ko je robot naletel na celico, ki je bila označena v nasprotju s stanjem na zemljevidu, je moral algoritem preračunati in poiskati novo najkrajšo pot od trenutne celice do končnega cilja. Primer najdene najkrajše poti je prikazan na sliki 6.2.



Slika 6.2: Mreža velikosti 20 x 20. Zelena celica predstavlja začetek poti, rdeča konec poti, črna oviro in modra prosto celico.

6.3 Okolje za testiranje

Za programski jezik, ki je služil za implementacijo algoritmov in izvajanje meritev, smo si izbrali Python. Izbrali smo ga zaradi treh razlogov, prva dva, ki ju bomo navedli, sta mogoče banalna, ampak ju opravičuje tretji razlog. Kot prvo je to moderen, zadnje čase precej priljubljen jezik in se precej uporablja kot učni pripomoček na univerzah. Kot drugo, omogoča pisanje elegantne kode in v našem primeru je pri večini algoritmov dejanska implementacija

skoraj povsem enaka, kot je psevdokoda. Skoraj povsod imata istoležni vrstici v Pythonovi kodi in psevdokodi enako funkcijo oziroma vlogo. In tretji, najbolj tehten in upravičen razlog je ta, da nam milisekunde ali sekunde, ki smo jih dobili kot rezultat meritev hitrosti delovanja algoritmov, ne povejo kaj dosti, če ne poznamo še strojne opreme, na kateri so tekli algoritmi.

Kar je pri rezultatih meritev veliko pomembneje, je razmerje med hitrostmi posameznih algoritmov. Če bomo to diplomsko delo brali čez 15 let, nam nič kaj dosti ne bo povedal rezultat, da je nek algoritem pri mreži, veliki 50 x 50 dosegel čas 12 milisekund, ker bo takrat ta ista implementacija algoritma potrebovala morda le nekaj nanosekund za rešitev navedenega problema. Še zmeraj pa se bo ohranilo razmerje med dvema algoritmoma. Če je danes nek algoritem npr. enkrat hitrejši od drugega algoritma, bo to razmerje ohranil tudi na močnejši in hitrejši računalniški opremi.

Zaradi zgoraj navedenega razloga počasnost Pythona ni nič kaj omejujoča.

6.4 Postopek testiranja

Pri meritvah smo se osredotočili na dve stvari, in sicer na število vozlišč, ki jih je moral algoritem razviti, da je našel cilj in skupen čas izvajanja algoritma. Svet v obliki mreže je bil različnih velikosti in z različnim številom začetnih ovir.

Za testiranje smo razvili nov razred, ki smo mu kot argument podali velikost mreže, ki jo naj samodejno zgenerira. Generiranje mreže je bilo naključno in vsaka mreža je vsebovala približno 40 % začetnih ovir (to število je variiralo, saj je bilo odvisno od funkcije, ki skrbi za naključnost). Pred samim zagonom vseh algoritmov se je zgeneriral tudi seznam celic, s katerim smo simulirali kasnejše zaznavanje sveta s pomočjo robotovih senzorjev. V tem seznamu je bilo približno 20 % vseh celic, ki so predstavljale ovire. Nekatere izmed njih so bile že označene na zemljevidu kot ovire, torej se ujemajo s svetom, nekatere pa bo mogoče robot šele odkril na svoji poti, če se bo premikal v njihovi bližini in bo zaradi njih moral algoritem ponovno poiskati novo najkrajšo pot.

Testirali smo 5 različnih velikosti mrež: 30 x 30, 60 x 60, 70 x 70, 100 x 100 in 120 x 120. Pri vsaki mreži je bil začetek poti na sredini leve strani mreže in konec na sredini desne strani mreže. Za vsako velikost mreže smo vsak algoritem pognali 20-krat in izračunali povprečni čas njegovega izvajanja.

6.5 Rezultati

V tabeli 6.1 so prikazani rezultati meritev hitrosti izvajanja posameznih algoritmov.

velikost mreže	D*	FD*	LPA*	D* Lite
30 x 30	0.18	0.14	1.09	0.17
60 x 60	1.73	0.99	12.21	1.56
70 x 70	3.47	2.02	3.93	3.19
100 x 100	16.09	5.49	129.93	8.96
120 x 120	33.60	6.07	232.14	12.09

Tabela 6.1: Hitrost izvajanja posameznih algoritmov v sekundah [s].

Pri vseh velikostih mreže je bil najhitrejši algoritem *Focused D** in najpočasnejši *LPA**. Izkazalo se je tudi, da je algoritem *D* Lite*, kot je omenjeno v literaturi, vsaj tako hiter, kot je algoritem *D**. Najmanjša razlika v hitrosti izvajanja je med algoritmoma *Focused D** in *D* Lite*. Ta je sicer v prid prvemu algoritmu, ampak ima drugi algoritem precej bolj preprosto in razumljivo zgradbo, zaradi česar je lažji za implementacijo.

velikost mreže	D*	FD*	LPA*	D* Lite
30 x 30	1012	573	3433	581
60 x 60	2560	1256	15249	1737
70 x 70	5214	2256	34123	3123
100 x 100	7317	2652	74335	3317
120 x 120	9987	5147	112738	7476

Tabela 6.2: Število razvitih vozlišč pri izvajanju posameznih algoritmov.

V tabeli 6.2 so prikazani rezultati merjenja števila razvitih vozlišč pri izvajanju posameznih algoritmov. Iz obeh tabel je mogoče razbrati, da je čas izvajanja algoritma v neposredni povezavi s številom razvitih vozlišč. Več vozlišč kot je bilo treba razviti, več časa je algoritem porabil za vodenje robota do cilja. Algoritem *Focused D**, ki se razlikuje od algoritma *D** po tem, da uporablja hevristiko, je zaradi tega moral razviti enkrat manj vozlišč kot pa algoritem *D**.

Izkazalo se je, da je hitrost izvajanja posameznega algoritma zelo odvisna od velikosti mreže, še bolj pa od števila ovir, ki so na poti do cilja. Če je

imel robot malo ovir na svoji poti, je bil algoritem tudi za od tri- do štirikrat hitrejši, kot je njegovo povprečje.

Poglavje 7

Zaključek

V tem diplomskem delu smo se osredotočili na algoritem *D**. Ta algoritem ima več različic oziroma izpeljank. Te so: osnovni algoritem *D**, algoritem *Focused D** in algoritem *D* Lite*. Ker se algoritem *D* Lite* po zgradbi precej razlikuje od preostalih dveh algoritmov, smo si zato še pobliže ogledali algoritem *LPA**, iz katerega je ta algoritem izpeljan.

Vsem štirim algoritmom je skupno to, da omogočajo inkrementalno preiskovanje, ki za trenutno preiskovanje izkorišča že pridobljeno informacijo iz prejšnjih preiskovanj. Najpogostejša uporaba teh algoritmov je v navigaciji robotov na neznanem ali delno neznanem terenu, ki se lahko med samim preiskovanjem spreminja.

Če še enkrat povzamemo tehnično plat teh algoritmov, lahko napišemo, da vsi navedeni algoritmi razen osnovnega algoritma *D** pri preiskovanju uporabljajo hevristiko. V našem primeru je bila to evklidska razdalja. Algoritmi *D*, Focused D* in D* Lite* preiskujejo od končnega vozlišča proti začetnemu vozlišču, medtem ko algoritem LPA* preiskuje od začetnega vozlišča proti končnemu.

Rezultati, ki smo jih dobili na podlagi meritev, so sovpadali z našimi pričakovanji. Manjša razlika, v njuno škodo, se je pojavila le pri algoritmih *LPA** in *D* Lite*, če smo za realizacijo prioritetnega seznama uporabili kopico (kot je to v originalu) namesto navadnega seznama. Razlogi za to so:

- brisanje poljubnega elementa ima za oba časovno zahtevnost O(n), s tem, da je treba kopico zatem še ponovno preurediti, navadnega seznama pa ne;

- dodajanje elementa na navaden seznam ima časovno zahtevnost O(1), kopico pa je treba pri dodajanju elementa še preurediti;

- jemanje prvega elementa ima za oba časovno zahtevnost O(1);
- navaden seznam je treba v funkciji *computeShortestPath()* vsakič preurediti, dokler se zanka ne izteče. Časovna zahtevnost za to je O(nlogn).

Torej lahko razberemo, da ima navaden seznam samo eno počasno operacijo v primerjavi s kopico, in sicer preurejanje v funkciji *computeShortestPath()*, kopico pa je treba preurediti pri brisanju in dodajanju poljubnega elementa v funkciji *updateVertex()*, ki se med tekom algoritma pokliče večkrat, kot pa se pokliče funkcija *computeShortestPath()*.

V eksperimentih, katerih rezultati so navedeni v prejšnjem poglavju, smo zaradi tega uporabili navaden seznam.

7.1 Nadaljnje delo

Nadaljevanje tega diplomskega dela se lahko usmeri v implementacijo teh algoritmov v kakega izmed hitrejših programskih jezikov. Dobra izbira bi bil jezik C. Pri tem bi vsekakor pridobili pri hitrosti, bi se pa povečalo število vrstic v implementaciji. Če bi želeli algoritme dejansko uporabiti za resnične robote, bi morali to tudi storiti.

Pobliže si lahko ogledamo še algoritem *Moving Target D* Lite*, ki lahko učinkovito najde najkrajšo pot, tudi če imamo poleg neznanega terena še premikajoči se cilj.

Slike

2.1 Kompozicija funkcije f 6

6.1 Svet v obliki mreže 53
6.2 Najkrajša pot 54

Tabele

6.1 Hitrost izvajanja algoritmov. 56
6.2 Število razvitih vozlišč. 56

Literatura

[1] W. Zhang, *State-Space Search*, New York: Springer-Verlag, 1999, pogl. 1.

[2] R. Balakrishnan, *A textbook of graph theory*, Springer-Verlag, 2000, pogl. 1.

[3] Robin J. Wilson, John J. Watkins, *Uvod v teorijo grafov*, knjižnica Sigma, DMFA, 1997.

[4] (2008) Graph search algorithms. Dostopno na:
http://www.cs.duke.edu/ josh/bfsdfs.html

[5] Trinity College (2011) Artificial Intelligence - Notes: State Space Search. Dostopno na:
http://www.cs.trincoll.edu/ ram/cpsc352/notes/search.html

[6] D. Kopec, (2008) Artificial Intelligence: Search Methods. Dostopno na:
http://spider.sci.brooklyn.cuny.edu/˜kopec/Publications/Publications/O_5_AI.pdf

[7] R. Dechter in J. Pearl, *Generalized Best-First Search Strategies and the Optimality of A**, JACM, 1985, zvezek št. 32.

[8] J. Russel in P. Norvig, *Artificial Intelligence: A Modern Approach*, Pearson Education, 2003, pogl. 4.

[9] P. E. Hart, N. J. Nilsson, B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, Systems Science and Cybernetics, 1968, zvezek št. 2.

[10] A. Stentz, *Optimal and Efficient Path Planning for Partially-Known Environments*, Carnegie Mellon University, 1994.

[11] J. Guo in L. Liu, *A study of improvement of D* algorithms for mobile robot path planning in partial unknown environments*, Wuhan: School of Automation, 2009.

[12] A. Stentz, *Optimal and Efficient Path Planning for Unknown and Dynamic Environments*, Carnegie Mellon University, 1993.

[13] A. Stentz, *The Focussed D* Algorithm for Real-Time Replanning*, Carnegie Mellon University, 1995.

[14] S. Koenig and M. Likhachev, *Incremental A**, Advances in Neural Information Processing Systems, 2001, strani 1539-1546.

[15] S. Koenig and M. Likhachev, *D* Lite*, Proceedings of the AAAI Conference of Artificial Intelligence, Edmonton 2002, strani 476-483.