Algoritmi traženja optimalnog puta u prostoru s preprekama

Dokladal, Darko

Sveučilište u Zagrebu
Fakultet strojarstva i brodogradnje

ZAVRŠNI RAD

Darko Dokladal

Zagreb, 2010.

Sveučilište u Zagrebu
Fakultet strojarstva i brodogradnje

ZAVRŠNI RAD

Voditelj rada:
Prof. dr. sc. Mario Essert

Darko Dokladal

Zagreb, 2010.

Sažetak

U okviru ovog završnog rada opisan je algoritam pronalaska puta. Algoritam je sposoban pronaći put od početne pozicije do cilja, a da pritom izbjegava prepreke uz uvjet optimalnosti.

U prvom poglavlju dan je kratak opis razvojnog okruženja algoritma u Pythonu, zatim opis rada algoritama. Algoritam koristi heuristiku, te je kratko objašnjeno što je heuristika i njena važnost. Načinjeni program može učitati realnu sliku prostora te iz slike prikupiti informacije o objektima koje je potrebno izbjeći da bi se došlo do traženog cilja. Za obradu slike korišten je OpenCV u sprezi s Pythonom. Također su opisana dosadašnja istraživanja u području traženja puta.

Sadržaj

1. Uvod 1
1.1. Računalni vid 1
2. Python 3
3. A* (a star) - algoritam pronalaska puta 5
3.1. Heuristika 5
3.2. Odabir heuristike 6
3.3. Podjela površine pretraživanja 6
3.4. Opis rada A* algoritma 9
3.5. Bodovanje puta 11
3.6. Druge metode algoritama pronalaska puta 16
3.6.1. D* (D star) algoritam 16
3.6.2. Genetski algoritmi 17
3.6.3. Metoda potencijalnih polja 17
4. Učitavanje realnog modela 18
4.1. Pretraživanje područja realnog modela 21
5. Zaključak 24
6. Literatura 25

Popis slika

Slika 1: Okolina u kojoj algoritam traži put 7
Slika 2: Prvi korak početka pretraživanja, u sredini se nalazi početni čvor 10
Slika 3: Rezultati prvog koraka pretraživanja ploče 12
Slika 4: Prebacivanje čvora na zatvorenu listu 13
Slika 5: Stanje na ploči nakon stavljanja trećeg člana na zatvorenu listu 14
Slika 6: Prvi korak početka pretraživanja, u sredini se nalazi početni čvor 15
Slika 7: Konačan izgled pronađenog puta na ploči 16
Slika 8: Primjena Canny algoritma, lijevo – Canny, desno – original 20
Slika 9: Izgled ploče nakon učitavanja rubova predmeta 21
Slika 10: Rubovi premeta dobro su nađeni 21
Slika 11: Pronalazak puta: a) Euclide, b) Manhatten 22
Slika 12: Prvi korak početka pretraživanja, u sredini se nalazi početni čvor 22
Slika 13: Rezultati nađenog puta: a) Euclide, b) Manhatten 23

Izjavljujem da sam ovaj rad izradio samostalno sa stečenim znanjem i navedenom literaturom.

Zahvaljujem prof. dr. sc. Mariu Essertu na ukazanom povjerenju prihvaćanjem mentorstva za ovaj rad i korisnim savjetima.

1. Uvod

Jedan od glavnih ciljeva moderne robotike je načiniti autonomnog robota, robota kojem bi mogli reći što da napravi bez da mu kažemo kako da to napravi. Između ostalog, to znači da robot mora moći samostalno se kretati. Da bi se kretanje znalo planirati, robot mora znati nešto o okolini u kojoj će se kretati. Uzmimo za primjer mobilnog robota u tvornici. Takav robot mora znati gdje mu se prepreke nalaze. Neke od tih informacija mogu mu se zadati tlocrtom prostora. Za druge informacije robot će se morati osloniti na vlastite senzore. Što znači da bi trebao moći detektirati prepreke koje nisu na tlocrtu. Koristeći informacije o okolini, robot bi se trebao kretati do svog cilja bez sudaranja u prepreke. U ovom radu je pretpostavljeno da se takva informacija dolazi iz videokamere.

1.1. Računalni vid

U današnje vrijeme svjedoci smo brzog razvoja informatičkih tehnologija. Jedno od područja koje se naglo razvija je i računalni vid. Zahtjevi mnogih proizvodnih procesa nadilaze sposobnosti ljudskog vida. Proizvodnja se odvija prebrzo ili su tolerancije odstupanja toliko male da ljudsko oko ne može raspoznati loš proizvod. Zbog toga je računalni vid vrlo raširen u modernoj serijskoj proizvodnji za primjerice - pozicioniranje predmeta ili robotske ruke, prepoznavanja uzoraka, upravljanja kakvoćom itd. U budućim tehnologijama kao što su: bespilotne letjelice, samoupravljajuća vozila, medicinska analiza, sigurnosno motrenje računalni vid također će imati važnu ulogu.

Računalni vid (eng. computer vision) je grana računalne znanosti koja se bavi teorijom i izradom sustava koji služe za prikupljanje informacija iz slike. Slike se mogu prikupljati iz različitih izvora kao što je kamera, određeni medicinski uređaji (CT skener, ultrazvuk), itd. Neke od poddisciplina računalnog vida su praćenje, detekcija i prepoznavanje objekata, detekcija događaja, restauracija slike i slično. Računalni vid je srodan mnogim drugim znanstvenim disciplinama kao što je optika, obrada slike i analiza slike, raspoznavanja uzoraka, robotika, umjetna inteligencija itd.

Biološki i računalni vid su blisko povezani. Biološki vid se bavi istraživanjem vizualnih percepcija ljudi i životinja s ciljem izgradnje modela funkcioniranja tih sustava s aspekta fizioloških procesa. S druge strane računalni vid se bavi istraživanjem i opisivanjem umjetnih vizijskih sustava koristeći znanja dobivena proučavanjem biološkog vida.

Postoji i pojam „ strojni vid“ (eng. machine vision) koji nije sinonim za računalni vid. Pojam strojni vid definira primjenu računalnog vida u industriji i proizvodnji, zbog toga pojam „strojni vid“ spada u inženjerske discipline. Cilj strojnog vida je da određeni stroj odnosno robot dobije sposobnost „gledanja“. Koristeći strojni vid, strojevi vrše inspekciju proizvoda

detektirajući prisutnost, odnosno odsutnost dijelova, uzimajući mjere, čitajući bar kodove. Strojni vid veliku pažnju posvećuje hardveru posebice kamerama.

Kamere koje se najčešće upotrebljavaju u industriji za strojni vid su kompaktne, lagane sa ugrađenom video memorijom i procesorom za obradu slika. Sastoje se od CCD senzora visoke rezolucije sa brzim (DSP) signalnim procesorom za obradu slike. Koristi se RAM memorija za spremanje podataka i slika. Na taj način mnogi problemi računanog vida riješeni su u samoj kameri pomoću hardvera. Osim toga sastoje se od sučelja za komunikaciju sa drugim uređajima koristeći već standardne komunikacijske protokole kao što su Profibus, CAN bus, Industrial Ethernet. Zbog toga se za takve kamere koristi pojam „pametne kamere“ (eng. smart camera).

U sustavima strojnog vida računalo od kamere prima niz brojeva i ništa više. Ljudi na temelju te slike mogu raspoznati određene dijelove automobila, ali računalo „vidi“ samo niz brojeva. Svaki broj osim informacije o slici sadrži i određenu količinu smetnji. Smetnje mogu nastati zbog promjene u osvjetljenju, određenih refleksija, neželjenih pomaka promatranog objekta. Mehanički dijelovi kamere, posebice leća, također uzrokuje određenu distorziju zbog nesavršenosti u izradi. Elektronički dijelovi također uzrokuju određeni električni šum koji utječe na kvalitetu dobivene slike. Zadatak računalnog vida je izdvojiti korisne informacije iz slike kako bi pomoću određenih algoritama mogli odrediti što slika predstavlja.

Problem dodatno otežava i način na koji kamera prikuplja informacije. Kamera prikuplja informacije iz trodimenzionalnog prostora te ih sprema u obliku brojeva u dvodimenzionalni prostor. Pomoću dobivenih informacija nije moguće u potpunosti rekonstruirati trodimenzionalni sliku. Izgled snimljenog dvodimenzionalnog objekta drastično ovisi o položaju kamere.

Navedeni problem se može riješiti pomoću kontekstualnih informacija. Što se tiče smetnji njih rješavamo pomoću statističkih metoda. Neke smetnje možemo u potpunosti kompenzirati. Primjerice, distorziju koju uzrokuje leća na kameri je dobro poznata te ju je moguće matematički opisati na temelju mjernih podataka.

2. Python

Python je visoki programski jezik opće namjene, a glavni naglasak je na čitljivosti kôda. Cilj programskog jezika Python je imati moćan jezik s jasnom čitljivosti kôda, njegova standardna biblioteka je velika i laka za korištenje. Interpreter, interaktivan, objektno orijentiran, inkorporira module, iznimke, vrlo visoke dinamičke tipova podataka i klase (razrede) sve su to odlike Python jezika. Python je lako prenosiv na druge platforme kao što su Unix, MacOS i Windows.

Python je programski jezik, kojeg je 1990. godine prvi razvio Guido van Rossum. Već do konca 1998., Python je imao bazu od 300.000 korisnika, a od 2000. Već su ga prihvatile ustanove kao MIT, NASA, IBM, Google, Yahoo i druge. Python ne donosi neke nove revolucionarne značajke u programiranju, već na optimalan način ujedinjuje sve najbolje ideje i načela rada drugih programskih jezika. On je jednostavan i snažan istodobno. Više nego drugi jezici on omogućuje programeru više razmišljanja o problemu nego o jeziku. U neku ruku možemo ga smatrati hibridom: nalazi se između tradicionalnih skriptnih jezika (kao što su Tcl, Schema i Perl) i sistemskih jezika (kao što su C, C++ i Java). To znači da nudi jednostavnost i lako korištenje skriptnih jezika (poput Matlab-a), uz napredne programske alate koji se tipično nalaze u sistemskim razvojnim jezicima.

Python je besplatan (za akademske ustanove i neprofitnu upotrebu), softver otvorenog kôda, s izuzetno dobrom potporom, literaturom i dokumentacijom.

Jezik visoke razine
Osim standardnih tipova podataka (brojevi, nizovi znakova i sl.) Python ima ugrađene tipove podataka visoke razine kao što su liste, n-terci i rječnici.

Interaktivnost
Python se može izvoditi u različitim okruženjima. Za razvitak programa najlakši je interaktivni način rada u kojem se programski kôd piše naredbu za naredbom. Ne postoji razlika u razvojnom i izvedbenom (engl. runtime) okolišu: u prvom se izvodi naredba za naredbom, a u drugom odjednom čitava skripta.

Čista sintaksa
Sintaksa jezika je jednostavna i očevidna. Uvlake zamjenjuju posebne znakove za definiranje blokova kôda, pa je napisani program vrlo pregledan i jednostavan za čitanje.

Napredne značajke jezika
Python nudi sve značajke očekivane u modernom programskom jeziku: objektu orijentirano programiranje s višestrukim nasljeđivanjem, dohvaćanje izuzetaka ili iznimki (engl.

exception), redefiniranje standardnih operatora, pretpostavljene argumente, prostore imena (engl. namespaces), module i pakete.

Proširivost
Python je pisan u modularnoj C arhitekturi. Zato se može lako proširivati novi značajkama ili API-ima (engl. application programming interface).

Bogate biblioteke programa
Pythonova biblioteka (engl. library), koja uključuje standardnu instalaciju, uključuje preko 200 modula, što pokriva sve od funkcija operacijskog sustava do struktura podataka potrebnih za gradnju web servera. Glavni Python web site (www.python.org) nudi sažeti indeks mnogih Python projekata i različitih drugih biblioteka.

Potpora
Python ima veliku entuzijastičku zajednicu korisnika koja se svake godine udvostručuje.

Jedna od standardnih biblioteka koja je korištena u ovom radu za izradu grafičkog sučelja je Tkinter. Tkinter nije jedini alat za izradu grafičkog sučelja u Pythonu ali je jedan od najčešće korištenih. Sastoji se od većeg broja modula, a neki ovdje korištenih su: `tkFileDialog`, `tkMessageBox`, `tkSimpleDialog`. Najvažniji modul je sam modul `Tkinter`.

3. A* (a star) - algoritam pronalaska puta

A* algoritam jednostavan je za implementaciju, vrlo je efikasan i uvijek ima dovoljno prostora za optimizaciju. A* je dizajniran da pronađe put od točke do točke, za razliku od Dijkstra algoritma koji služi za pronalazak najkraćeg puta u teoriji grafova. Također A*, za razliku od Dijkstre, koristi heuristiku.

3.1. Heuristika

Heuristika je aproksimativno rješenje koje u puno situacija je točno, ali mala je vjerojatnost da će biti točno za sve situacije. Ljudi u svakodnevnom životu koriste heuristiku. Ne razmišljamo o svim posljedicama naših djela već se pouzdamo na nekom generalnom principu koji je u prošlosti funkcionirao za određenu stvar. To može biti nešto jednostavno kao „ako si izgubio nešto, vrati se koracima kojima si išao da bi našao tu stvar“. Heuristika se sastoji da mi zapravo ne znamo gdje smo izgubili ali pogađamo da se to dogodilo negdje gdje smo bili. „Nikad ne vjeruj prodavaču rabljenih vozila“ je heuristika kojom donosimo odluku o kupnji auta.

Što je točnija heuristika to će biti potrebno manje prostora pretražiti tj. brže će se izvesti algoritam. Kada bi imali savršenu heuristiku (ona koja će uvijek dati točnu minimalnu vrijednost između dva čvora), A* bi odmah krenuo ravno prema cilju. Nažalost, da bi dobili točnu udaljenost između dva čvora moramo naći najkraći put između njih. To bi značilo da je potrebo riješiti problem pronalaska puta između njih – što je ono što i pokušavamo učiniti. Samo u nekim slučajevima će heuristika biti točna. Za nesavršenu heuristiku, A* će se ponašati drugačije, ovisno je li heuristika preniska ili previsoka.

Ako je heuristika preniska, tako da podcjenjuje stvarnu dužinu puta, A* će se izvoditi nešto duže. Razlog tome je što će A* pretraživati čvorove koji su bliži početnom čvoru prije od onih koji su bliže cilju, jer je vrijednost heuristike manja od realne. To će povećati vrijeme pretraživanja puta do cilja. U aplikacijama gdje je točnost puta važnija od performanse izvođenja, bitno je osigurati da heuristika precjenjuje. Ponekad se ne radi samo o optimalnom putu već o realnom.

Ako je heuristika viša od realne, tako da precjenjuje stvarnu udaljenost, postoji mogućnost da A* ne vrati najbolji put. Ukupna vrijednost čvora će biti pristrana heuristici. A* će posvetiti manje pažnje na vrijednost „cijena čvora do sad“, a favorizirat će čvorove koji imaju manju udaljenost do cilja. To će pomaknut fokus potrage prema cilju brže, ali s izgledom da ne nađe najbolji put do cilja. To znači da ukupna duljina puta može biti veća od optimalnog puta. Srećom, to ne znači da će davati jako loše putove. Može se pokazati da ako heuristika precjenjuje za najviše *x* (*x* je najveća vrijednost kojom precjenjuje čvor), tada će konačni put

biti ne viši od *x* predugačak. Precijenjena heuristika se ponekad se naziva i „nedopustiva heuristika“. To znači da ju ne možemo koristiti, a odnosi se na to da A* algoritam više ne vraća najkraći put. Ako se malo precjenjuje, generirani put će često biti identičan najboljem putu, tako da kvaliteta nije veliki problem.

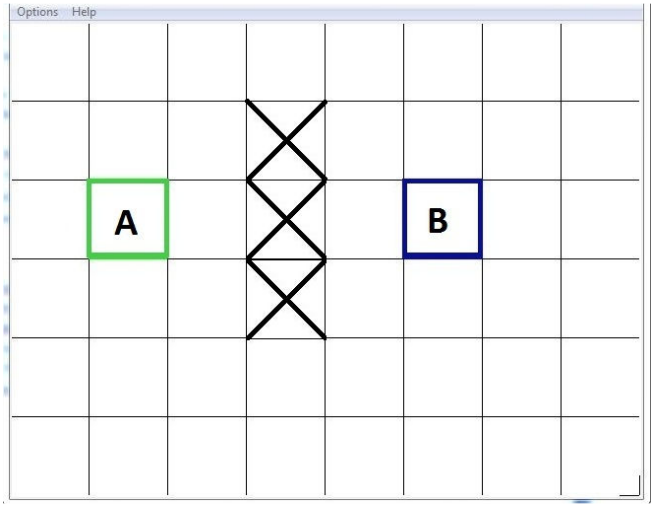
Ali margina za pogrešku je mala. Kako heuristika precjenjuje sve više, performanse A*-a rapidno postaju sve lošije. Ukoliko je heuristika konstantno blizu savršene, može biti više efikasno imati podcijenjenu heuristiku.

3.2. Odabir heuristike

Izrada heuristike je više umjetnost nego znanost. Njen značaj je podcijenjen u grani umjetne inteligencije. Jedini siguran način da se dobije kvalitetna heuristika je tako da se vizualizira „popunjenost“ pretraživanja ploče tj. koliko je čvorova algoritam morao posjetiti prije nego je našao put. Često se moguće prevariti da heuristika koja će dodatno poboljšati svojstva rezultira suprotnim efektom. Najčešće se koriste dvije vrste heuristike – Euclideova i tzv. Manhattan. Provedena su određena istraživanja gdje se heuristika generira automatski bazirano na analizi strukture grafa. Takva vrsta heuristike može rezultirati boljim performansama od Euclideove. Većina programera ipak se odluči za heuristiku koja procjenjuje blizu, ali vuče na stranu podcjenjivanja. Ali za sad heuristika je Euclideova udaljenost i tako će ostati još neko vrijeme.

3.3. Podjela površine pretraživanja

Zamislimo da želimo na nekakvoj površini doći iz jedne točke do druge točke, recimo – od točke A, do točke B. Recimo da su te dvije točke odijeljene zidom. Ta situacije je ilustrirana na slici 1 – zeleni kvadratić je početna točka A, plavi kvadratić je ciljna točka B, prekriženi kvadratići su prepreka odnosno zid.



Slika 1: Okolina u kojoj algoritam traži put

To nam pojednostavljuje pretraživanje polja, a ujedno je i prvi korak prema algoritmu za pronalazak puta. Ova metoda svodi našu površinu koju pretražujemo na jednostavno dvodimenzionalno polje. Svaki kvadratić predstavljen je objektom i status toga objekta govori nam je li prohodan ili nije. Put je pronađen tako što zaključimo kojim kvadratićima treba ići od A do B. Jednom kad je put pronađen, mičemo se od centra jednog kvadratića do centra drugog kvadratića dok ne dođemo do konačnog cilja.

Centri kvadratića se zovu čvorovi (eng. nodes). Kada se čita literatura o algoritmima za pronalazak puta, često se može primijetiti da se govori o čvorovima. To je iz razloga što površinu koju pretražujemo ne moramo podijeliti na kvadratiće, već to mogu biti pravokutnici, heksagoni, trokuti, bilo koji oblik.

Jednom kad smo podijelili našu površinu koju pretražujemo na savladiv broj čvorova slijedeći korak je provesti pretraživanje koje će dati najkraći put. To radimo tako što počinjemo od točke A, pregledavamo sve kvadratiće oko nje i općenito tražimo dalje dok ne dođemo do konačne točke. A* je tip algoritma koji ne pretražuje slijepo cijelo područje dok ne nađe put već uputno (zbog heuristike) počinje pretraživati najbolji smjer.

Počnimo s modeliranjem okoline. Poznato je da dobar algoritam bez dobre strukture podataka ne rezultira dobrim programom. Pa tako je i u ovom slučaju. Okolina se najčešće prikazuje nekom vrstom strukture u programskom kôdu. Prostor pretrage predstavljen je objektom razreda „Ploca“. Jedna od klasa koja jako pridonosi pojednostavljenju pisanja i snalaženju u kôdu je klasa „Koord“ koja potpuno brine o uvedenom koordinatnom sustavu. Definicija klase Koord izgleda ovako:

```
class Koord:
    def __init__(self, m = 0, n = 0):
        self.x = m
        self.y = n

    def set(self, m, n):
        self.x = m
        self.y = n

    def dim(self, sirina_ploce):
        return self.y * sirina_ploce + self.x

    def u_koord(self, ploca_sirina, polje_duljina):
        self.x = polje_duljina % ploca_sirina
        self.y = polje_duljina / ploca_sirina
        return self
```

Usluge koje objekt klase Koord može tražiti da se na njemu izvedu su funkcije „set“, „dim“ i „u_koord“. Konstruktor klase je jednostavan, proslijeđena su mu dva argumenta *m* i *n* koji postavljaju početne vrijednosti objekta. Funkcija „set“ mijenja vrijednosti objekta. Kako je ploča u memoriji zapisana kao jednodimenzionalno polje, funkcije „dim“ i „u_koord“ pojednostavljuju baratanje prebacivanje iz jednodimenzionalne u dvodimenzionalnu

veličinu. Nakon uvedenog koordinatnog sustava imamo temelje za klasu „Ploca“ koja predstavlja prostor pretrage tj. okolinu u kojoj A* algoritam radi.

```
class Ploca:
    def __init__(self, m, n):
        self.buff = []
        self.sirina = m
        self.visina = n
        for i in range(m * n):
            self.buff.append('.')

    def zidovi(self, koordinata):
        self.buff[koordinata.dim(self.sirina)] = 'x'

    def iscrtaj_plocu(self):
        for x in range(self.visina * self.sirina):
            if(x % self.sirina == 0 and x != 0):
                print '\n'
            print self.buff[x],
        print '\n'
```

Konstruktor klase „Ploca“ dobiva kao argumente dimenzije 2D ploče. Klasa sadrži dvije funkcije od kojih jedna naziva „zidovi“ postavlja prepreke u područje pretraživanja algoritma. Prazna, odnosno prohodna, mjesta u memoriji bit će predstavljena . (točkom), a zidovi tj. neprohodna mjesta će imati znak „x“. Tako ćemo pozivom funkcije iscrtaj_plocu dobiti trenutno stanje prostora pretrage. Nakon tako modelirane okoline, možemo se posvetiti izradi algoritma pretrage.

3.4. Opis rada A* algoritma

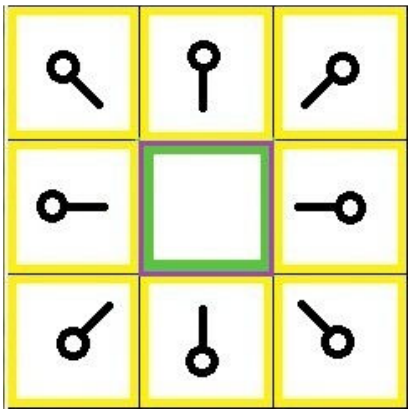
Algoritam je napisan u obliku klase. Tako ga je lakše za koristiti za druge korisnike, npr. ako se uveze kao modul u neki drugi program, a i lakše je podložan izmjenama. Ako se želi promijeniti funkcioniranje algoritma, potrebno je mijenjati samo njegovu implementaciju, ne i dio kôda koji se odnosi na manipulacijom objektom klase.

```
class A:
    def __init__(self):
        self.F, self.G, self.H = 0, 0, 0
        self.parent = Koord()
        self.open = True

class Astar:
    def __init__(self, vel):
        self.broj_clanova_open = 0
        self.openL = { }
        self.closeL = []
        self.Apolje = []
        for i in range(vel.x * vel.y):
            self.Apolje.append(A())

    def add_to_open_list(self, ploca, pocetni, zavrsni):
            .
            .
            .
    def find(self, ploca, pocetni, zavrsni):
            .
            .
            .
    def add_to_close_list(self, ploca_sirina):
            .
            .
            .
    def path(self, ploca, pocetni, zavrsni):
            .
            .
            .
```

Kako je prikazano implementacijom klase `Astar` algoritam je podijeljen u četiri osnovne funkcije. Bitni dijelovi klase su i njeni članovi. Svaki kvadrat na koji je prostor pretrage podijeljen predstavljen je objektom klase `A`. Klasa sadrži informaciju kao što su bodovanje puta. U tu svrhu služi član `Apolje`. Tako imamo pojam „otvorena lista“ koja je predstavljena članom `openL`. Otvorena lista sadrži informacije o čvorovima koji bi mogli činiti put do cilja ali i možda ne. Općenito to je lista čvorova koje treba provjeriti. Funkcija koja će se baviti stavljanjem članova na otvorenu listu zove se `add_to_open_list`. Upravo suprotno otvorenoj listi, postoji i zatvorena lista. Na nju će biti stavljeni svi čvorovi koji su provjereni i nema potrebe ih gledati ponovno. Kao i otvorena lista, zatvorena ima svoju funkciju koja vodi računa o stavljanju čvorova na zatvorenu listu stoga joj od tuda i ime `add_to_close_list`. Funkcije `find` i `path` su zapravo jedine koje se koriste od strane

korisnika. Funkciji `find` je zadaća naći put, dok `path` crta put koji je pronađen na ekran. Kako sve to radi, analizirat će se na primjeru sa slike 1.

Algoritam analizira čvor po čvor i to tako što pregledava sve čvorove oko sebe. Kakvo je stanje na početku prikazano je na slici 2.



Slika 2: Prvi korak početka pretraživanja, u sredini se nalazi početni čvor

Dakle, točka od koje počinjemo tražiti put stavljena je na otvorenu listu čvorova koje ćemo razmotriti. Na početku je samo jedan čvor na otvorenoj listi ali ćemo ih više imati kasnije. Slijedeći korak je provjeriti sve dostupne i prohodne čvorove oko trenutnog čvora (na početku je to čvor A), ignorirajući sve ostale čvorove koje predstavljaju prepreku. Njih također stavimo na otvorenu listu. Za svaki od tih čvorova spremamo informaciju tko im je roditelj. Ta informacija je spremljena u podatkovnom članu klase A. Ilustrativno svaki čvor pokazuje na svog roditelja crnom zastavicom. Tko je roditelj je bitna informacija kada želimo slijediti put prema natrag. Nakon što smo sve čvorove oko početnog smjestili na otvorenu listu, ispuštamo početnu točku A s otvorene liste i smještamo u zatvorenu listu što znači da taj čvor više nećemo trebati provjeravati. Do sada imamo nešto kao što je prikazano na slici 2. Na slici je prikazano: zeleni kvadratić u sredini je početni kvadratić, obojen je u ljubičaste rubove što nam govori da je premješten na zatvorenu listu. Svi čvorovi koji ga okružuju su sada na otvorenoj listi, moraju biti provjereni, i imaju žuti okvir. Svaki čvor ima crnu „zastavicu“ koja pokazuje tko joj je roditelj, a to je početni čvor A.

Slijedeće, izaberemo jedan od okolnih čvorova koji je na otvorenoj listi i ponovimo proces. Potrebno je izabrati čvor s najmanjom F vrijednosti.

3.5. Bodovanje puta

Ključ određivanja koji kvadratić upotrijebiti da bi došli do cilja je slijedeća jednadžba:

$$F = G + H$$

gdje je:

G = cijena pomicanja s čvora A do slijedećeg čvora na mreži, prateći put generiran do tamo

F = cijena pomicanja od čvora kojeg promatramo do konačnog cilja, točke B. Ovo se naziva heuristikom tj. procjena preostale udaljenosti. Mi u biti ne znamo stvarnu udaljenost dok ne pronađemo put, zato što razne prepreke mogu biti na putu do cilja.

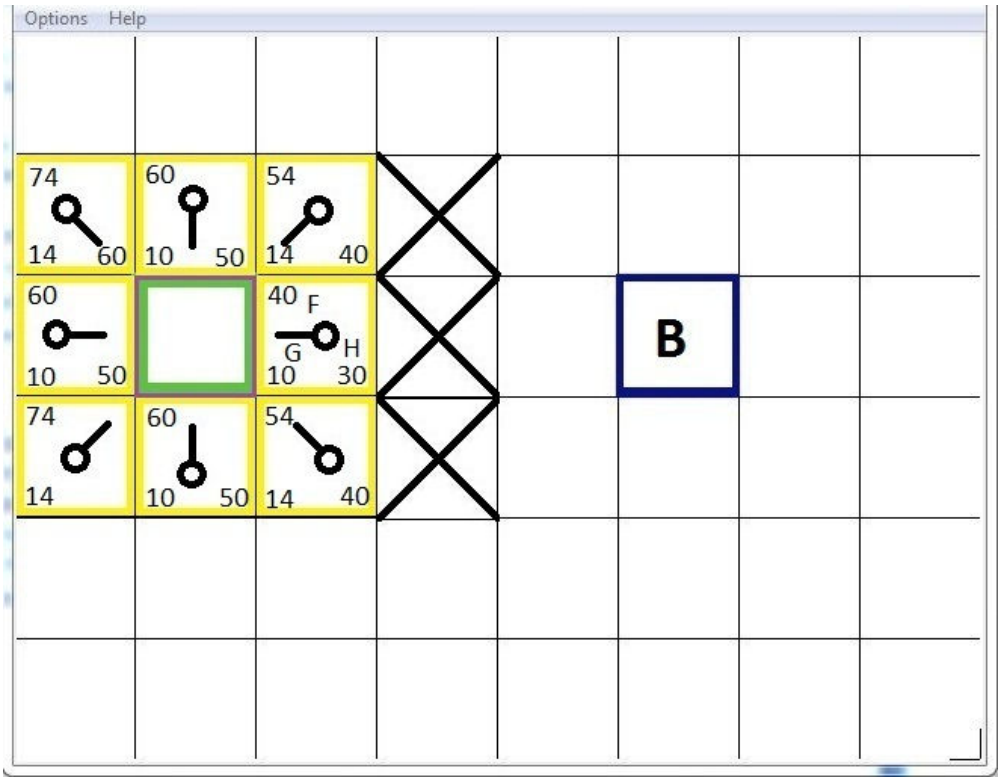
Naš put se generira tako što konstantno pretražujemo otvorenu listu i biramo čvorove s najmanjom F vrijednosti. Proces će biti detaljnije opisan. Prvo ćemo pogledati kako izračunavamo jednadžbu.

Kao što je opisano, G je cijena pomaka od početnog do trenutnog čvora koristeći put generiran do njega. U ovom primjeru, dodijelit ćemo cijenu 10 svakom horizontalnom i vertikalnom pomaku, a cijenu 14 svakom dijagonalnom pomaku. Koristimo ove brojeve jer stvarna udaljenost za pomak po dijagonali je drugi korijen od 2, što iznosi 1.414. Koristimo 10 puta veće brojeve zbog jednostavnosti, a i izbjegavamo rad s decimalnim brojevima da bi računalo moglo brže računati.

G cijenu računamo tako što uzmemo G cijenu roditelja čvora kojem računamo cijenu i zbrojimo 10 ili 14 ovisno je li dijagonalno i ortogonalno od roditelja. Potreba za ovom metodom će doći malo kasnije u primjeru, kad više nećemo biti samo jedan čvor od početnog.

H se može odrediti na razne načine. Metoda koju koristimo ovdje zove se Manhattan metoda, izračunavamo broj čvorova potrebnih da dođemo do ciljnog čvora tako što se mičemo horizontalno i vertikalno po ploči, ignorirajući dijagonalne pomake i bilo kakve zapreke koje su na putu. Zatim pomnožimo taj broj s 10. Ovakva heuristika je gruba estimacija preostale udaljenosti između trenutnog položaja i cilja. Što smo bliži s procjenom stvarne udaljenosti, to će algoritam biti brži. Ako precijenimo udaljenost, nije garantirano da ćemo dobiti najkraći put. U tom slučaju imamo „nedopustivu heuristiku“.

F se računa tako što se zbroje G i H. Rezultat prvog koraka pretrage se može vidjeti na slici 3. F, G i H bodovi su zapisani u svaki čvor. F je u gornjem lijevom kutu, G je u donjem lijevom kugu, a H je u donjem desnom.



Slika 3: Rezultati prvog koraka pretraživanja ploče

Analizirat ću sada čvor sa slovima upisanim na njemu. Njemu je G = 10. To je zato što je samo za jedan čvor od početnog kvadratića u horizontalnom smjeru. Čvorovi koji su odmah iznad, ispod i lijevo od početnog – svi imaju jednaki G, 10. Dijagonalni čvorovi imaju G da im je 14.

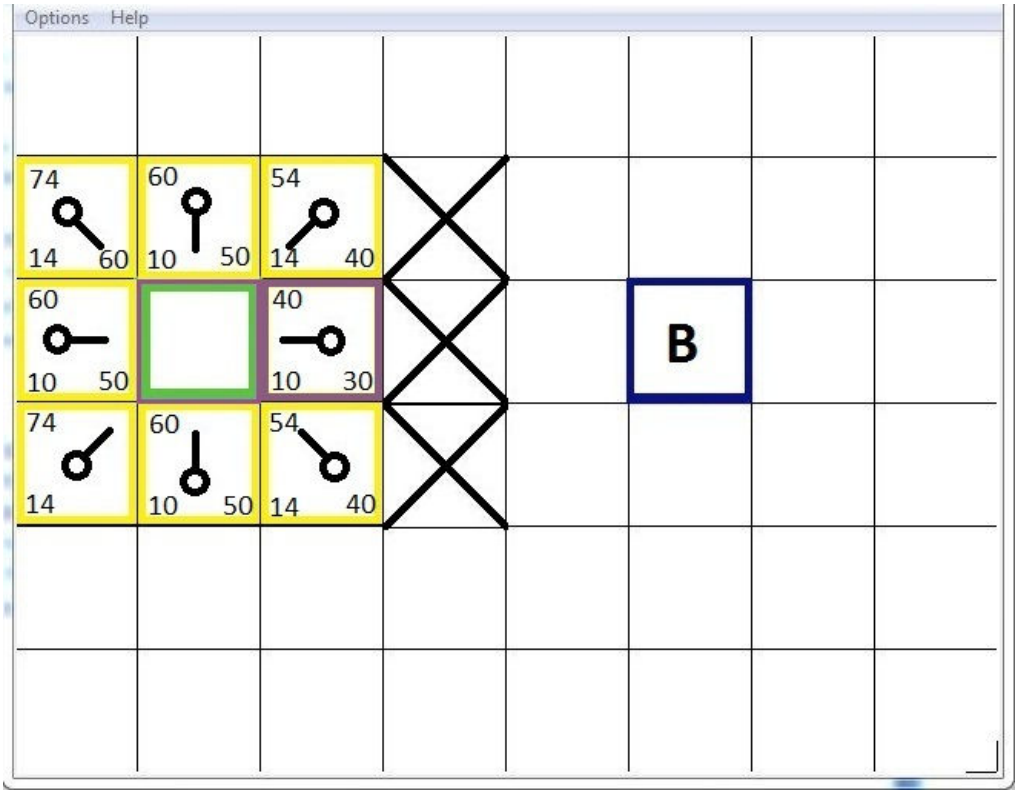
H cijena se proračunava Manhattan metodom, mičući se samo horizontalno i vertikalno do cilja, ignorirajući sve zapreke na putu. Tako će kvadratić koji je odmah desno od početnog kvadratića imati H = 30, zato što je 3 kvadratića udaljen od plavog kvadratića. Kvadratić odmah iznad njega je 4 kvadratića udaljen i zato mu je H cijena 40.

F cijena se računa, za svaki kvadratić nanovo, tako što se zbroji G i H.

Za nastavak pretrage, jednostavno izaberemo čvor s najmanjim F-om od svih koji su na otvorenoj listi. Upravo iz tog razloga za otvorenu listu najbolje je koristiti kao podatkovni član rječnik (eng. dict) u Pythonu. Tako automatski dobivam sortiranu listu čvorova po njihovim vrijednostima F od najmanjeg prema najvećem, što mi osigurava da uvijek na početku (prvi član) bude čvor s najmanjom vrijednošću. Prvi čvor na otvorenoj listi (to je onaj koji je desno od početnog čvora A) premještamo na zatvorenu listu. Zatim je potrebno provjeriti sve čvorove koji ga okružuju. Ignorirajući sve čvorove koji su na zatvorenoj listi ili su neprohodni dodajemo čvorove na otvorenu listu osim ako već nisu tamo. U ovom trenutku je potrebno promijeniti podatkovni član klase A koji se zove `parent` za taj čvor, i to tako da zapamti koordinate čvora s kojeg smo došli provjeriti njega. Zato ga zovemo roditeljem tog čvora. Ukoliko su okolni čvorovi već na otvorenoj listi (ili neki od njih), potrebno je provjeriti je li put do tog čvora bolji od onog koji već postoji. To se radi tako što gledamo vrijednost G

za taj čvor, ako je ona manja ako koristimo trenutni čvor da dođemo do njega onda je to kraći (bolji) put. Ako je vrijednost veća, sve ostaje po starom, za slučaj da je manja potrebno je promijeniti vrijednost podatka `parent` u koordinate čvora na kojem smo bili malo prije. Konačno ponovno izračunamo F i G vrijednosti za taj čvor.

Situacija na ploči je slijedeća: od početnih devet čvorova, imamo osam na otvorenoj listi nakon što je početni prebačen na zatvorenu listu. Prvi na otvorenoj listi je onaj čvor s najmanjom F vrijednosti, a to je odmah desno od početnog. Stoga je taj čvor slijedeći na kojem se izvršava nastavak algoritma. Označen je ljubičastim na slici 4.



Slika 4: Prebacivanje čvora na zatvorenu listu

Prvo, ispustimo ga s otvorene liste i stavimo na zatvorenu listu (zato je označen ljubičastim). Zatim provjerimo čvorove koji ga okružuju. Oni odmah desno od tog čvora su čvorovi koji predstavljaju zid, stoga njih zanemarimo. Čvor koji je odmah lijevo je početni kvadratić. On je na zatvorenoj listi, njega isto ignoriramo.

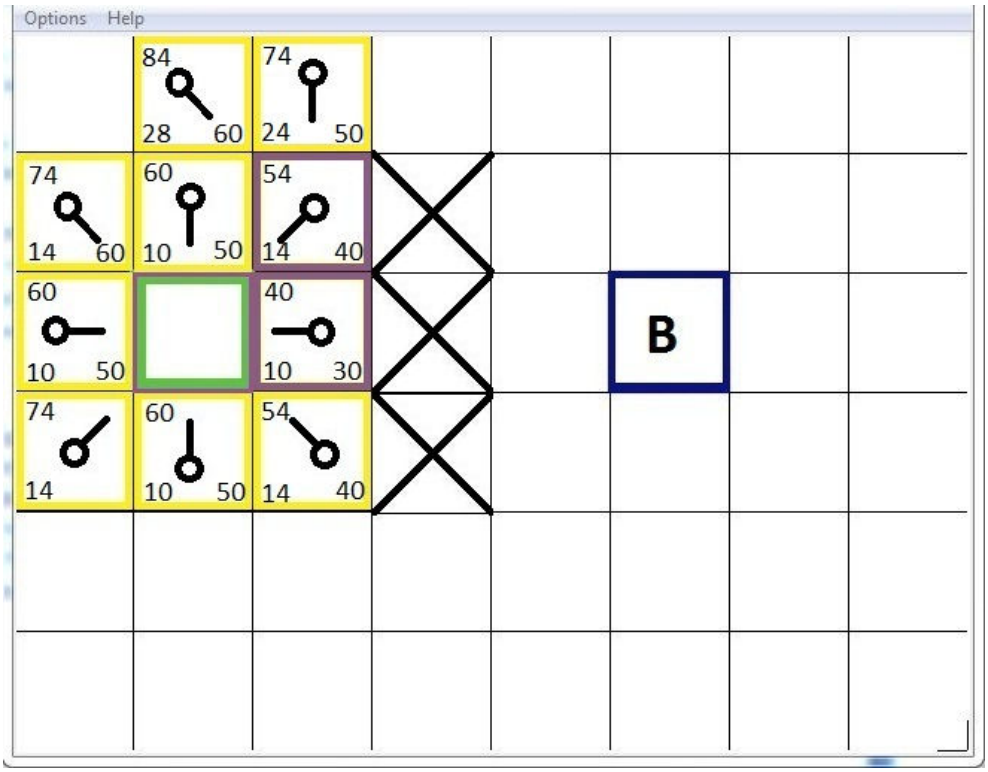
Ostalih četiri čvora su već na otvorenoj listi, pa njih trebamo provjeriti tj. trebamo provjeriti je li put kojim možemo doći do njih bolji od onog koji već postoji. Za tu provjeru koristimo G cijenu kao referencu. Pogledajmo gornji čvor, odmah iznad našeg odabranog kojeg smo stavili na zatvorenu listu, njegova G cijena je 14. Ako bi do njega išli preko našeg trenutnog čvora, G cijena bi bila jednaka 20 (10, što je G cijena da bi došli do trenutnog kvadratića, plus 10 više da bi došli vertikalno do njega). G cijena je 20 i veća je od 14, stoga to nije bolji put.

Kada pogledamo sliku, to i ima smisla. Više je direktno doći do tog čvora dijagonalno preko početnog A kvadratića nego pomakom horizontalno jedan čvor i zatim vertikalno jedan.

Kada ponovimo ovaj proces za sva četiri okolna čvora koja su već na otvorenoj listi, vidimo da ni jedan put neće biti bolji od onog koji već postoji. Sad smo pogledali sve okolne čvorove, gotovi smo s tim čvorom i spremni smo prijeći na slijedeći.

Stoga, prolazimo kroz otvorenu listu, koja je sad na sedam čvorova, i izaberemo onaj s najmanjom F cijenom. U ovom slučaju, tu su dva čvora s jednakom F cijenom 54. U korist brzine, recimo da izaberemo zadnji koji je ušao na otvorenu listu.

Pretpostavimo da je zadnji na otvorenu listu došao gornji čvor te je on naš slijedeći početni čvor kao što je nacrtano na slici 5.



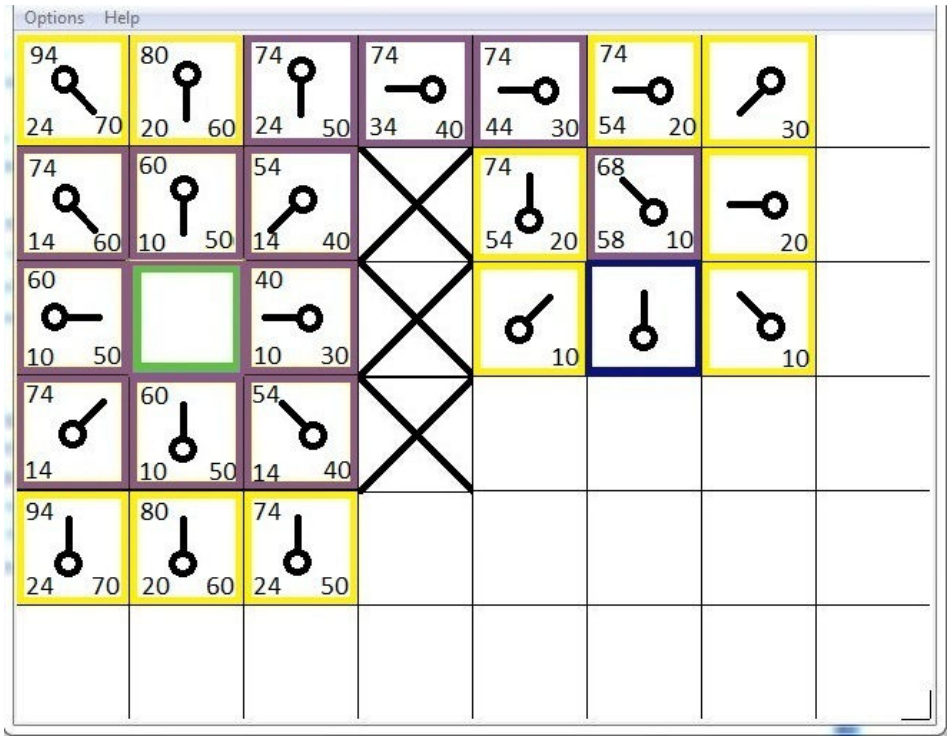
Slika 5: Stanje na ploči nakon stavljanja trećeg člana na zatvorenu listu

Ovaj put kad provjeravamo okolne čvorove pronalazimo da je odmah desni čvor zid, njega ignoriramo. Jednako se odnosi na donji čvor (on je na zatvorenoj listi). Također ignoriramo i čvor koji je odmah iznad zida. Iz razloga što ne možemo proći pokraj zida, već ga moramo zaobići. Prolazak pokraj kuta je proces. Ovo je više opcionalno, nije nužno.

To nam ostavlja pet drugih čvorova. Od toga dva čvora iznad trenutnog nisu još na otvorenoj listi, stoga dodamo njih na listu i trenutni čvor postaje njihov roditelj. Od preostalih tri čvora, dva su već na zatvorenoj listi (početni, kojeg zovemo A, i jedan upravo ispod trenutnog čvora, oba imaju ljubičaste okvire na slici), njih ignoriramo. I zadnjem čvoru, odmah lijevo od trenutnog čvora, moramo provjeriti njegovu G cijenu je li niža ako idemo do tog čvora

koristeći trenutni čvor oko kojeg provjeravamo ostale. Nije taj slučaj. Stoga smo gotovi sa svim čvorovima oko trenutnog i spremni smo pogledati na otvorenu listu za slijedeći s najmanjom F cijenom.

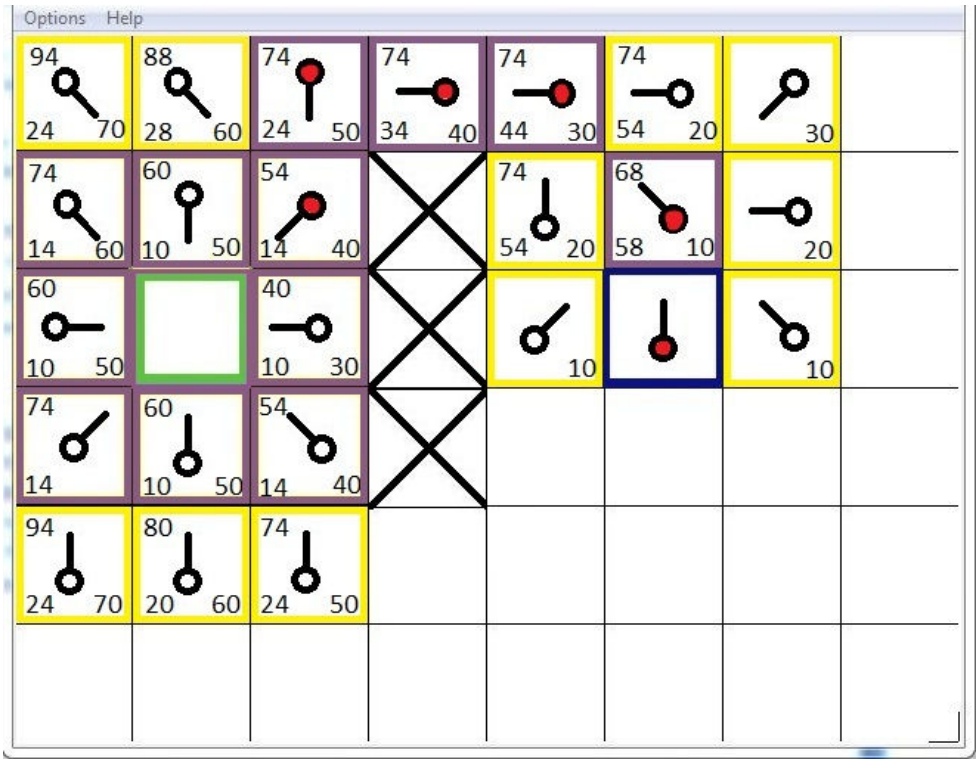
Ponovimo ovaj proces sve dok ne dodamo završni čvor (točka B, plavi kvadratić) na zatvorenu listu, što bi izgledalo nešto poput slike 6.



Slika 6: Pronalazak puta do cilja

Možemo uočiti da je za čvor dva kvadratića iznad početne točke A promijenjen roditelj. Prije je imao G cijenu 28 i zastavicu koja je pokazivala natrag na čvor dolje lijevo od njega. Sada ima cijenu 20 i pokazuje na čvor odmah ispod njega. To se dogodilo negdje kroz proces traženja puta, gdje je G cijena koju smo provjeravali ispala manja koristeći novi put – pa je došlo do promjene roditelja i G i F cijene su ponovo izračunate. U ovom primjeru ta promjena se ne čini važnom, ali postoje mnoge situacije u kojima bi ova provjera napravila razliku u određivanju najkraćeg puta do cilja.

Končan put odredimo jednostavno, počnemo od plavog čvora i vraćamo se od jednog čvora pa do njegovog roditelja, slijedimo zastavice. Ovo će nas s vremenom vratiti do početnog čvora i to će biti naš put. Trebalo bi izgledati nešto poput ovoga što je na slici 7.



Slika 7: Konačan izgled pronađenog puta na ploči

3.6. Druge metode algoritama pronalaska puta

3.6.1. D* (D star) algoritam

D* je poboljšana verzija A* algoritma, a spada u kategoriju algoritama koji se služe inkrementalnim pretragama (eng. incremental search algorithms). Takvi algoritmi koriste se informacijama iz prijašnjih pretraga da bi ubrzali trenutnu i riješili problem pronalaska puta puno brže nego da pretražuje kao da je prvi put. Algoritmi koji koriste heuristiku, koriste heurističko znanje u obliku aproksimativne udaljenosti od cilja te tako fokusiraju svoju pretragu prema rješavanju traženja puta puno brže od algoritama koji pretražuju cijelo područje općenito. Također postoji podvrsta D* algoritma koji koristi i heuristiku i inkrementalno pretraživanje (eng. incremental heuristic search) te njihovom kombinacijom rješavaju problem pretraživanja puta u dinamičnim okolinama. U takvim slučajevima robotu je moguće zadati cilj u neistraženom području. On će pretpostaviti da to područje nema prepreka i izračunati će put uz tu pretpostavku. Zatim će slijediti put sve dok se to ne pokaže nemogućim. Nove informacije o terenu dodat će na mapu i ponovno izračunati put ukoliko je to potrebno. Dok prolazi kroz nepoznati teren, nove prepreke mogu biti česte, stoga ponovno izračunavanje puta mora biti brzo. D* algoritam je općenito brži i efikasniji od A*.

3.6.2. Genetski algoritmi

Oponašanje biološke evolucije i pokušaj primjene njene snage intrigirala je računalne stručnjake zadnjih nekoliko desetljeća. Genetski algoritmi spadaju u adaptivno stohastički optimizirajuće algoritme u svrhu pretraživanja i optimiziranja. Ovaj tip algoritma John Holland predstavio je 1960. godine. Osnovna ideja genetskih algoritama je pokušaj imitacije prirodne selekcije s ciljem pronalaska dobrog ili boljeg rješenja od postojećeg. Prvi korak je mutacija, ili bilo koji oblik nasumične varijacije, početnog stanja programa. Drugi korak je selekcija koja se obično vrši tako što se rezultat usporedi s funkcijom cilja. Proces se ponavlja sve dok se ne pronađe pogodno rješenje.

Holland je pokušao shvatiti i povezati raznolike tipove prirodnih fenomena ali je isto tako uočio potencijalnu inženjersku primjenu genetičkih algoritama. Od objave Hollandove knjige, genetski algoritmi su izrasli u značajno područje umjetne inteligencije i strojnog učenja. U današnje vrijeme organizira se nekoliko internacionalnih konferencija godišnje posvećenih genetskim algoritmima. Istraživanja genetskih algoritama su se proširila s računalnih znanosti na primjenu kod robota. Dvodimenzionalno planiranje trajektorije genetskim algoritmom može se vrlo jednostavno proširiti na problem trodimenzionalnog planiranja trajektorije. Put je potrebno razložiti u tri ravnine tj. *xy, xz, yz* projekcije. Tako je putanja u trodimenzionalnom prostoru jedinstveno prikazana parom putanja u dvodimenzionalnom prostoru. Jedna je *xy* projekcija, a druga *xz* projekcija (odnosno *yz* projekcija). Trodimenzionalno planiranje trajektorije se sastoji od nalaženja para dvodimenzionalne trajektorije. Takav tip algoritma se može primijeniti kod planiranja trajektorije podvodnog vozila (eng. autonomous underwater vehicle – AUV) koje istražuje morsko dno.

3.6.3. Metoda potencijalnih polja

U prethodna dva desetljeća, metoda potencijalnih polja postala je vrlo popularna u robotici, posebno u području mobilne robotike, prvenstveno zbog svoje matematičke jednostavnosti i elegancije. U metodi potencijalnih polja cilj je predstavljen atraktivnim potencijalom koji privlači mobilnog robota, a prepreke su predstavljene kao uzvišenja odnosno odbojni potencijal. Linearnom superpozicijom atraktivnog i odbojnog potencijala dobiva se ukupni potencijal. Metoda potencijalnih polja promatra mobilnog robota kao točku, odnosno kuglicu koja se giba po potencijalu pod utjecajem gravitacijske sile.

4. Učitavanje realnog modela

Za manipulaciju slikama koristi se već dobro poznati modul OpenCV. OpenCV je „ open source“ biblioteka za računalni vid. Biblioteka je pisana u programskom jeziku C i C++ -u te se može koristiti na operativnom sustavu Windows, Linux i Mac OS X. Biblioteku je također moguće koristiti sa programskim jezicima Python, Ruby i Matlab.

Kod dizajniranja OpenCV – a težilo se postići što veću računalnu učinkovitost, pogotovo kod aplikacija u stvarnom vremenu „ real – time application“. Zbog toga je pisan u optimiziranom C – u te je u mogućnosti koristiti prednosti višejezgrenih procesora.

Ako je potrebna dodatna optimizacija može se koristiti Intelova biblioteka IPP (Integrated Performance Primitives). Ova biblioteka sadrži rutine napisane na vrlo niskoj razini što omogućava izradu još bržih i još učinkovitijih aplikacija.

Jedan od ciljeva OpenCV – a je stvaranje jednostavne infrastrukture koja omogućava ljudima brzu izradu složenih vizijskih aplikacija. OpenCV biblioteka sadrži više od 500 funkcija koje pokrivaju područja kao što su: procesiranje slike, kalibracija kamere, stereo vid, vizijski sustavi u robotici, kontrola tolerancija proizvoda itd. OpenCV također sadrži podbiblioteku za strojno učenje (Mashine Learning Library – MLL).

Projekt OpenCV započeo je u Intelu gdje su razvijali aplikacije u stvarnom vremenu koje traže veliku procesorsku moć. U projekt su se uključila i vodeća sveučilišta kao što je MIT gdje su studenti razvijali algoritme namijenjene računalnom vidu. Stručnjaci za optimizaciju računalnog kôda također su pridonijeli razvoju OpenCV – a.

Bilo je nekoliko ciljeva koji su se htjeli postići OpenCV – om:

• Razviti ne samo besplatan nego i optimiziran računalni kôd za osnovne operacije u računalnom vidu, tako da ih novi korisnik može odmah početi koristiti u svom radu, umjesto da ponovno sam sve piše ispočetka.

• Pružiti znanje o računalnom vidu osiguravajući jednostavno razvojno okruženje kako bi računalni kôd bio lakše čitljiv i prenosiv.

• Razviti napredne aplikacije koje su besplatne ali po kvaliteti i mogućnostima ne zaostaju za komercijalnim rješenjima.

Pojavom višejezgrenih procesora vrijednost OpenCV je još više narasla. Danas mnoge institucije aktivno razvijaju OpenCV za tehnike računalnog vida koje tek dolaze u budućnosti kao što su: percepcija dubine, složeno prepoznavanje uzoraka, integracija kamere sa laserom za procjenu udaljenosti, kalibracija sustava sa više kamera, robotski vid itd.

Neke od korištenih funkcija iz ove biblioteke su:

- `CreateImage`
- `GetSize`
- `CvtColor`
- `Canny`

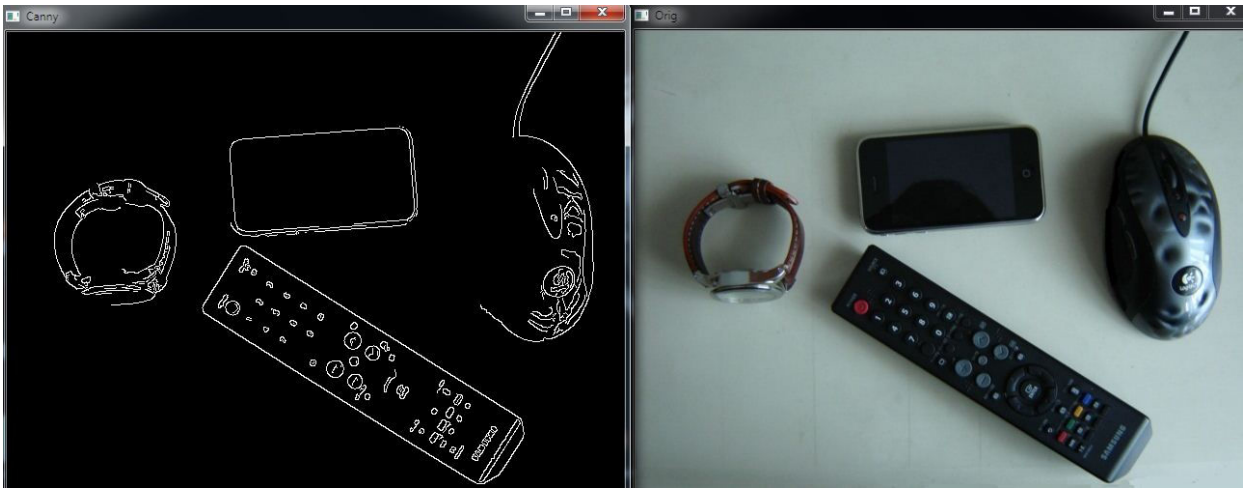
Funkcija `CreateImage` alocira memoriju potrebnu za sliku koja će imati parametre koje podesimo preko argumenata te funkcije. Prvi argument se tiče veličine slike, a kako želimo napraviti sliku jednake veličine kao original koristimo funkciju `GetSize` te njen rezultat proslijedimo kao prvi argument. Drugi i treći argument se tiču dubine boje i broj kanala po pikselu. `CvtColor` je funkcija kojom konvertiramo kanale slike iz jedne boje u drugu. Sve ove funkcije smo iskoristili da bi pripremili podatke za funkciju Canny koja će nam vratiti rubove objekata sa slike. Prije nego pogledamo funkciju Canny, valjalo bi se osvrnut na jednostavniju funkciju Laplace. Laplaceova funkcija implementira Laplaceov operator:

$$Laplace(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Laplace funkcija uzima kao argumente izvornu i odredišnu sliku. Laplaceov operator je zbroj drugih derivacija funkcije u *x* i *y* smjeru osi. To znači da će točka okružena s višim vrijednostima maksimalizirati funkciju i obrnuto. Takav Laplaceov operator može biti korišten za detekciju rubova. Da bi vidjeli kako je to postignuto, promotrimo prvu derivaciju funkcije, koja će biti velika kadgod se funkcija mijenja ubrzano (rapidno). Jednako bitno, rast će rapidno kako se bližimo rubu ili nečemu što nalikuje na rub i smanjivati kako se udaljujemo od toga. Stoga će derivacija biti lokalni maksimum unutar nekog područja. Tako možemo gledati nule druge derivacije za područja takvih lokalnih maksimuma. Rubovi u originalnoj slici će biti nule kao rezultat primjene Laplacea. Nažalost, bitni i manje značajni rubovi će biti nule.

Laplaceovu metodu za pronalazak rubova preradio je J. Canny zbog toga se njegova metoda naziva Canny detektor rubova (eng. Canny edge detector). Jedna od razlika između Canny algoritma i jednostavnijeg, temeljen na Laplaceu, je što Canny algoritam prvu derivaciju izračunava u *x* i *y* smjeru, a zatim kombinirana u četiri usmjerene derivacije. Točke gdje su te usmjerene derivacije lokalni maksimum su kandidati da budu dio ruba. Važnija razlika između dva algoritma je što Canny pokušava složiti kandidata za rub (piksel) u konturu. Postoje dva praga, gornji i donji. Ako piksel ima gradijent veći od gornjeg praga, prihvaćen je kao dio ruba; ako je piksel ispod donjeg praga, odbacuje se mogućnost da je dio ruba. Ako je piksel između gornje i donje vrijednosti, bit će prihvaćen samo ako spojen (neposredno blizu) drugim pikselom koji je iznad gornje vrijednosti granice. Canny preporuča omjer gornja:donja granica između 2:1 i 3:1. Ulazni argumenti funkcije su originalna slika koja mora biti boje sivih tonova i izlaznu sliku, koja je također u sivim tonovima (ali će u zapravo biti Boolean slika). Slijedeća dva argumenta su donji i gornji prag piksela.

Algoritam je onoliko dobar koliko mu je proslijeđena dobra originalna slika. Osvjetljenje ima veliki utjecaj, tj. sjene čine veliki problem. Rubovi koji su u sjeni mogu ostati neprepoznati tj. algoritam neće uspjeti prepoznati da je riječ o rubu predmeta. Također bi bilo poželjno da je podloga jednolična. Na slici 8 možemo vidjeti kako rub predmeta fali upravo zbog sjene.

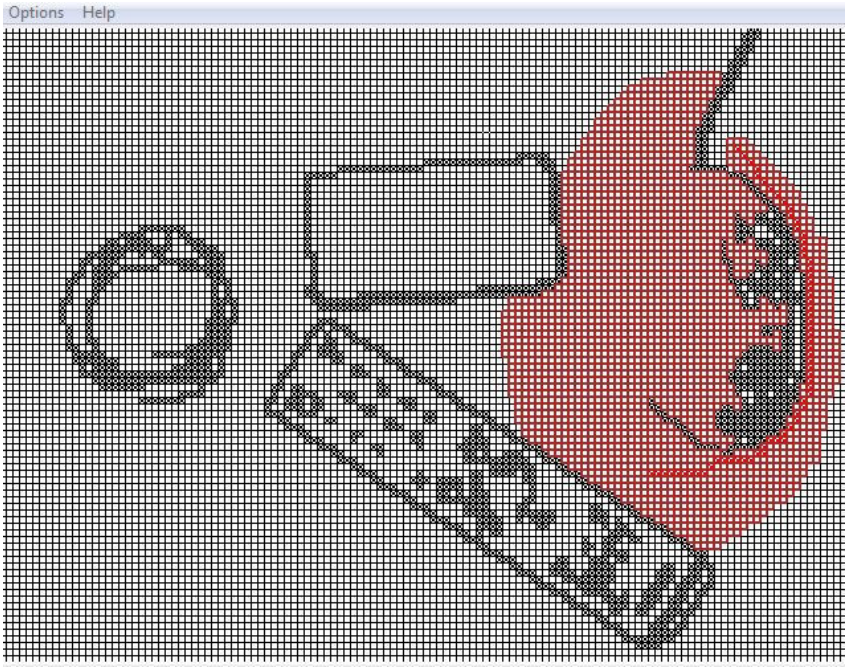


Slika 8: Primjena Canny algoritma, lijevo – Canny, desno – original

Zbog nedostatka ruba predmeta, algoritam traženja puta dodatno se usporava jer istražuje područje unutar predmeta te tako gubi vrijeme na nešto što nije potrebno.

4.1. Pretraživanje područja realnog modela

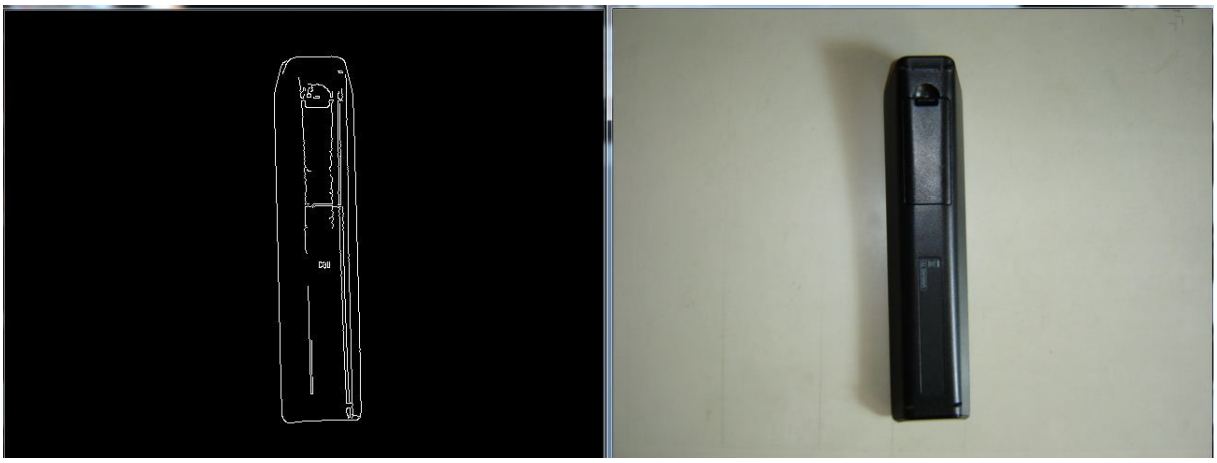
Nakon što učitamo sliku i dobijemo rubove predmeta koji su prikazani na slici 8, slijedeći korak je učitavanje tih informacija u prostor algoritma traženja puta gdje se on može snaći. Na slici 9 možemo vidjeti kako bi algoritam traženja puta gubio vrijeme na pretraživanje područja koje je svakako neprohodno, ali problem je što on to ne zna jer je izgubio informaciju o rubu predmeta.



Slika 9: Izgled ploče nakon učitavanja rubova predmeta

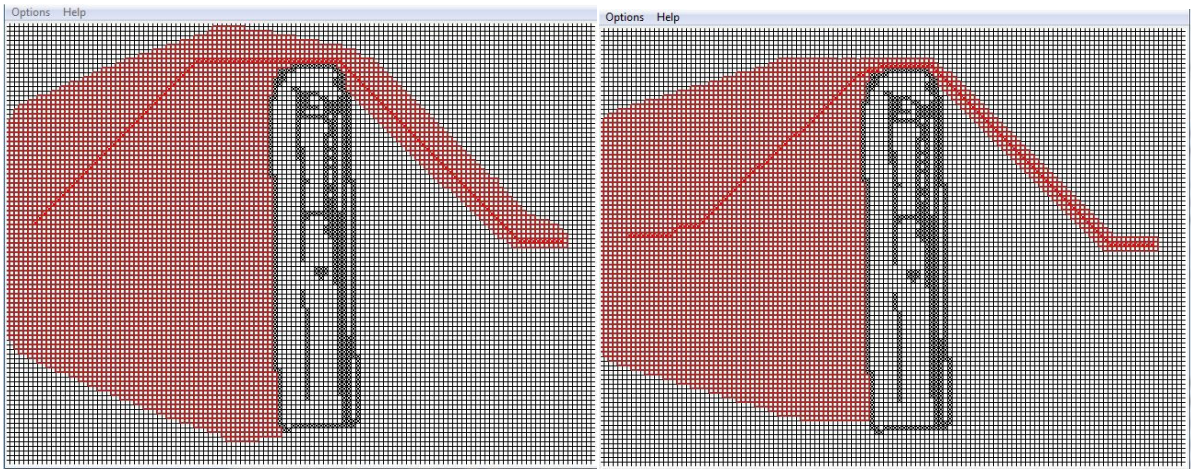
Algoritam je izveden interaktivno, tako da i kad je put pronađen još uvijek može doći do promjene ako se postavi zapreka na trenutnu putanju.

Primjerom će biti pokazano koliko je važan izbor heuristike tj. koliko heuristika utječe na brzinu pretraživanja, a čak je vidljiv njen utjecaj na oblik puta. Slika 10 je primjer kojim će biti pokazane dvije heuristike, a to su Euclideova i Manhatten.



Slika 10: Rubovi premeta dobro su nađeni

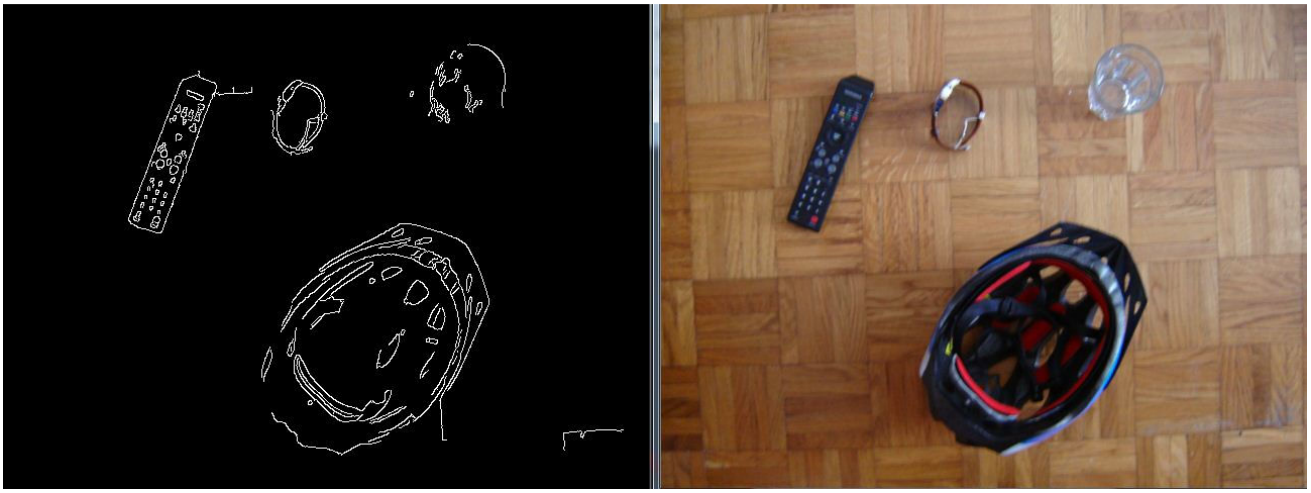
U ovom primjeru nema prevelike razlike između dvije heuristike. Iako se veličina pretraženog područja i oblik puta malo razlikuju, obje metode su našle put u približno jednakom vremenu. Tako je za Manhatten heuristiku trebalo prosječno 1.83 sekunde, dok je Eucideova to izvršila za prosječno 1.71 sekundi. To je u prvom redu zbog veličine podjele ploče (odabrano je 5 piksela razmaka između čvorova). Što je podjela finija (manja) to će trebati više vremena za pronalazak puta i obrnuto – što je podjela grublja (veća) to će rezultati pronalaska puta biti sličniji pa i skoro jednaki. U ovom primjeru odabrano je 5 piksela razmaka između čvorova.



a) b)

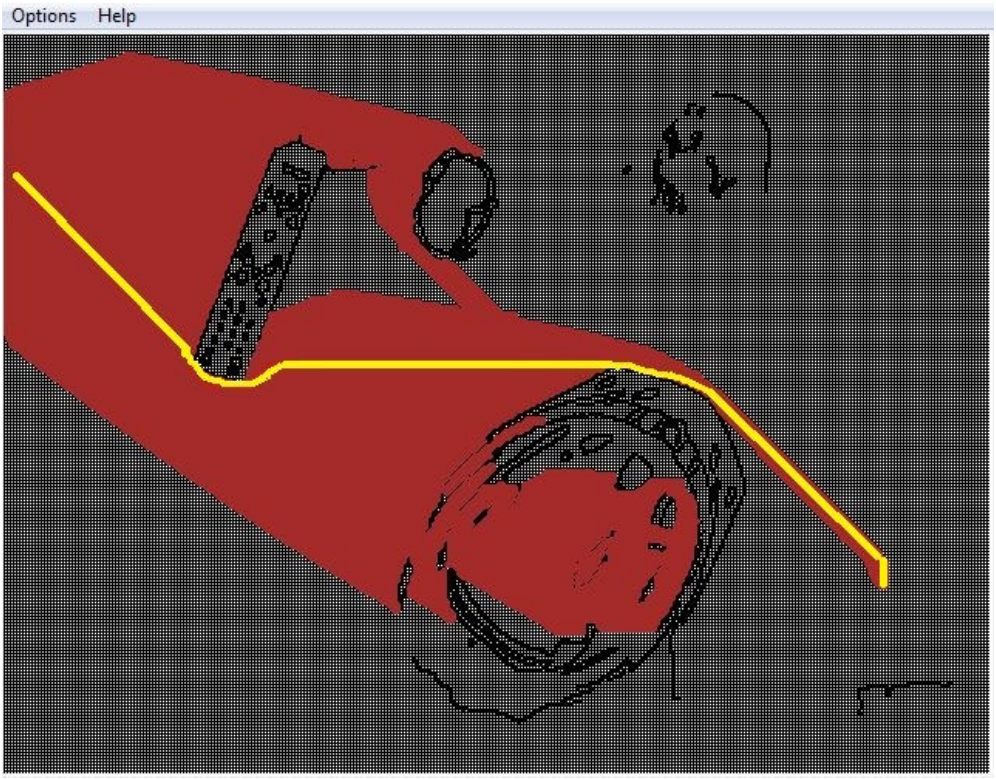
Slika 11: Pronalazak puta: a) Euclide, b) Manhatten

Slijedeći primjer (slika 12) pokazuje manju podjelu ploče, a vidjet ćemo kako to utječe i na vrijeme pretrage. Također je vidljivo kako algoritam pronalaska rubova ima problema s nejednoličnom podlogom te tako neke rubove koji stvarno postoje gubi, dok su neki koji postoje ostali neprepoznati. Može se primijetiti veliki problem pri prepoznavanju stakla.

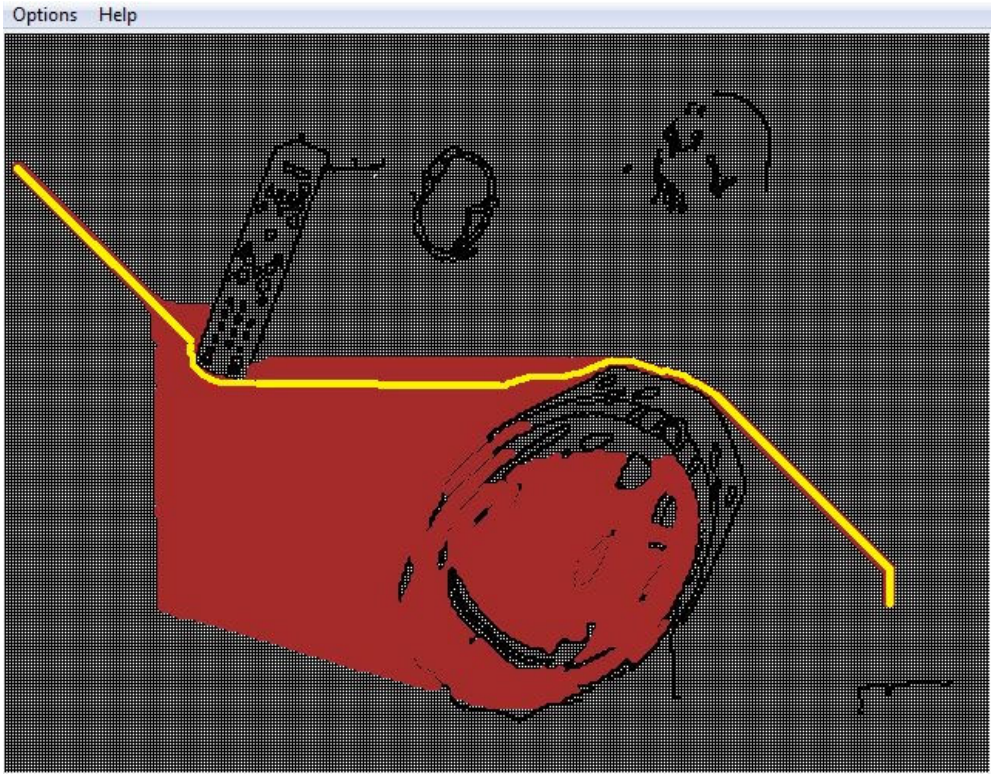


Slika 12: Primjer pronalaska rubova predmeta na nejednoličnoj podlozi

Manhatten heuristika je pronašla put za prosječno 34.2 sekundi, a Euclideova za 15.1 sekundi. Ovdje je to poboljšanje od 2.2 puta, a ima primjera kada je to i više. U prosjeku ispadne da je Euclideova heuristika bolja za 2-5 puta. Razmak između čvorova je 2 piksela.



a)



b)

Slika 13: Rezultati nađenog puta: a) Euclide, b) Manhatten

5. Zaključak

U ovom radu naglasak je bio na algoritmu pronalaska puta. Puno vremena i truda je uloženo u područje znanosti o traženju puta. Algoritam je pokazao solidne rezultate. Što je finija podjela ploče to vrijeme pretraživanja eksponencijalno raste. Algoritam daje optimalne rezultate za razmak između čvorova od 10 piksela. Tu je postignu kompromis između jasnoće slike i brzine algoritma (put je pronađen u svega stotinu milisekundi). Poboljšanja su svakako moguća. Prvenstveno u količini potrebne memorije. Od velike pomoći algoritmu bi bio kvalitetan algoritam za prepoznavanje rubova predmeta. Smatram da Canny algoritam odlično radi svoj posao ali potrebno je pribaviti kvalitetne slike. Problem pronalaska puta je kompleksan iako se čovjeku čini trivijalan (kao svakodnevna aktivnost). Natjecanje poput DARPA (Defense Advanced Research Projects Agency) Grand Challenge, koje se održavalo od 2004 – 2007, i NASA-ino istraživanje Marsa najviše su doprinijeli praktičnoj primjeni te vrste znanosti. Ako uzmemo u obzir da je prvo osvojeno mjesto u DARPA natjecanju 2 milijuna američkih dolara jasno je čemu se isplati potruditi. Mjesta za napredak u tom području još uvijek ima. Vjerujem da bi i u NASA-i bili sretniji kada bi se mobilni roboti na Marsu mogli kretati brže i sigurnije.

6. Literatura

1. Millington I., “Artificial Intelligence For Games“, Elsevier Inc., San Francisco CA, 2006.
2. Bradski G., Kaehler A., „Learning OpenCV“, O’Reilly Media, United States of America, 2008.
3. Cizelj I., Diplomski rad „Neizrazito analitičko upravljanje mobilnim robotom u nepoznatoj okolini pomoću metode potencijalnih polja“, Fakultet strojarstva i brodogradnje, 2008.
4. Python Programming Language -- Official Website, http://www.python.org/, lipanj 2010.
5. Wikipedia, http://en.wikipedia.org/wiki/A*_search_algorithm, lipanj 2010.
6. Wikipedia, http://en.wikipedia.org/wiki/D*, lipanj 2010.