Univerza v Ljubljani

Fakulteta za elektrotehniko

David Vodnik

Planiranje gladke poti mobilnega vozila z uporabo hibridnega algoritma A*

Diplomsko delo univerzitetnega študija

Mentor: izr. prof. dr. Gregor Klančar

Ljubljana, 2016

Zahvala

Zahvaljujem se izr. prof. dr. Gregorju Klančarju za napotke in pregled diplomskega dela.

Zahvaljujem se tudi družini in prijateljem za vso podporo.

Vsebina

Seznam slik

Povzetek

Delo opisuje implementacijo hibridnega A* algoritma za iskanje gladke poti mobilnega vozila. Cilj je bil poiskati in uspešno prevoziti pot, ki zadosti neholonomičnim omejitvam realnega vozila v prostoru obdanem z ovirami. Izdelana je bila simulacija v programskem okolju Unity, ki je pogosto uporabljeno za izdelavo računalniskih iger in omogoča dobro vizualizacijo sledenja poti mobilnega vozila. Simulacija vsebuje primer realnega prostora z ovirami, med katerimi se mora vozilo prebiti od začetne do ciljne pozicije. Izdelana je bila preprosta simulacija laserskega senzorja za kartiranje okolja, Ackermannov model vozila, hibridno A* iskanje in vodenje vozila po dobljeni poti.

Ključne besede: iskanje poti, hibridni algoritem A*, mobilni robot, Ackermann model

Abstract

The thesis describes the implementation of the hybrid A* algorithm with the aim of finding a smooth path in an obstacle dense environment. The goal was to find and successfully folow a path that meets the nonholonomic restrictions of a real vehicle. A simulation was developed in the Unity framework that is often used in game development and allows for a good visualization of mobile vehicle path following. The simulation consists of an example of a space with obstacles among which the vehicle has to travel from a start to a goal location. A simple laserscanner simulation was developed for mapping the environment, an Ackermann vehicle model, hybrid A* search and path following.

Key words: pathfinding, hybrid A* algorithm, mobile robot, Ackermann model

1 Uvod

Ena od pomembnejših nalog vodenja mobilnega vozila je prav iskanje primerne poti. Vendar pa ima pot realnega vozila določene omejitve, ki jih standardni algoritmi ne upoštevajo.

Prva takšna omejitev je neholonomična narava vozila, kar pomeni da ima sistem manj vhodnih spremenljivk, kot pa prostostnih stopenj. Stanje Ackermannovega (avtomobilskega) modela vozila lahko opišemo v treh dimenzijah: z dvodimenzionalno pozicijo in smerjo proti kateri je obrnjeno. Vplivamo pa lahko le na dve vhodni spremenljivki: smer gibanja (naprej ali vzvratno) in smer zavijanja. Ali povedano drugače: vozilo se v danem trenutku ne more gibati v poljubni smeri.

Druga omejitev poti pa je zveznost poti. Realno vozilo se ne giblje po diskretnih celicah, ki jih predpostavljajo klasični algoritmi, temveč v svetu zveznih koordinat. Dobljena pot mora bit definirana v vseh točkah od začetne pozicije do cilja.

Klasični algoritmi, kot so Dijkstrov ali osnovni A* razdelijo prostor na mrežo celic in iščejo pot od celice do celice. Takšna pot je le odsekoma zvezna in ne upošteva neholonomičnih omejitev, zato ji ni mogoče dobro slediti. Te omejitve rešuje hibridni A* algoritem, ki prostor še vedno razdeli v mrežo celic, vendar vsaki priredi zvezno stanje vozila. Ohranja tudi relacije med sosednjimi zveznimi stanji, tako da najdena pot zadosti neholonomičnosti vozila.

Opisani algoritem je implementacija algoritma, ki je bil uporabljen na tekmovanju Darpa Grand Challenge 2005, kjer je vozilo Junior prvič prevozilo celotno pot. Implementacija bazira na članku, ki opisuje algoritem uporabljen na tem tekmovanju[1]. Algoritem je opisan tudi v širšem delu, ki opisuje celotno arhitekturo vozila[2]. V pomoč pri modelu Ackermannovega vozila pa je bila skripta o avtonomnih mobilnih sistemih[3].

2 Klasični algoritmi iskanja poti

V tem poglavju bo opisanih nekaj osnovnih algoritmov za iskanje poti. Ti algoritmi iščejo pot po diskretnem grafu ali mreži celic in zato njihove rešitve niso primerne za vodenje Ackermannovega vozila v zveznih koordinatah. Vendar pa služijo kot uvod in predstavo principov, ki jih bomo pozneje potrebovali pri hibridnem A*.

2.1 Prostor iskanja

Sledeči algoritmi bodo opisani s pomočjo primerov na diskretni kvadratni mreži celic, saj je to priročen način za razumevanje in predstavo algoritmov z namenom iskanja poti v nekem virtualnem ali realnem svetu. Ker pa osnovni principi izhajajo iz teorije grafov, bomo v tem poglavju opredelili reprezentacijo mreže celic v obliki grafa in definirali analogne pojme, saj bodo ti pozneje zaradi lažje razlage uporabljeni izmenično.

2.1.1 Graf

Graf je set vozlišč in povezav med njimi. Je upodobitev množice objektov, pri čemer so določeni pari objektov povezani med seboj. Objekte v grafu predstavimo z vozlišči, vozlišča, ki so med seboj v relaciji, pa so povezana s povezavami.

Povezave v grafu so lahko usmerjene ali neusmerjene. Primer usmerjene povezave v svetu iskanja poti je na primer enosmerna cesta, ki jo lahko prevozimo le v eno smer.

Graf je lahko obtežen ali neobtežen. Neobtežen graf lahko predstavlja preprosto dvodimenzionalno mrežo, v kateri so sosednje celice vedno enako oddaljene ena od druge. Obtežen graf pa lahko predstavlja skupek cestnih povezav, pri čemer vozlišča grafa predstavljajo križišča, ki niso enakomerno oddaljena. Lahko pa utež grafa predstavlja povišano ceno vzvratne vožnje ali ostrega zavijanja v grafu zveznih stanj vozila.

2.1.2 Predstavitev kvadratne mreže z grafom

Iskanje poti se pogosto (predvsem pri računalniških igrah) uporablja za iskanje poti po kvadratni mreži. Celice mreže si lahko predstavljamo kot vozlišča grafa, možne poti med sosednjimi celicami pa kot povezave med vozlišči. V najbolj preprostem primeru ima vsaka celica štiri sosede: zgornjega, spodnjega, levega in desnega. V tem primeru lahko uporabimo neobtežen graf, saj so razdalje med vsemi povezanimi celicami enake. Če pa želimo povezati še diagonalne celice, potrebujemo predstavitev z obteženim grafom, saj so razdalje med diagonalnimi celicami za koren iz 2 krat daljše kot tiste med neposrednimi sosedi.

2.2 Iskanje v širino (Breadth first search)

Najosnovnejši algoritem je iskanje v širino. Osnovna ideja je enakomerno širjenje iskanja od začetne točke v vse smeri. Primeren je samo za iskanje po neobteženem grafu, saj izkorišča dejstvo, da se v vsakem krogu širitve oddaljimo od začetne lokacije točno za eno enoto. Na ta način je ciljna točka dosežena prek najmanjšega števila celic, kar nam zagotavlja najkrajšo najdeno pot (oz. eno iz množice najkrajših, katero, je odvisno od vrstnega reda odpiranja sosedov). Slabost tega algoritma (poleg zahteve po neobteženosti) je, da pregleda precej veliko območje grafa in je zato računsko precej neučinkovit.

Algoritem implementiramo s pomočjo čakalne vrste na katero dodajamo kandidate za širjenje iskanja in listo obiskanih celic na naslednji način:

1. Doda začetno celico na čakalno vrsto in jo označi kot obiskano.
2. Vzame (najbolj zgodaj dodano) celico iz čakalne vrste.
3. Obišče vse sosede trenutne celice. Če še niso bili obiskani, jih doda v čakalno vrsto in označi kot obiskane. Trenutno obiskani celici priredi referenco na izvorno celico, to je na celico iz točke 2.
4. Skoči na korak 2 in ponavlja dokler ni obiskana ciljna celica.

Za algoritem je podana tudi psevdokoda (vsa psevdokoda v tem dokumentu je veljavna Python koda):

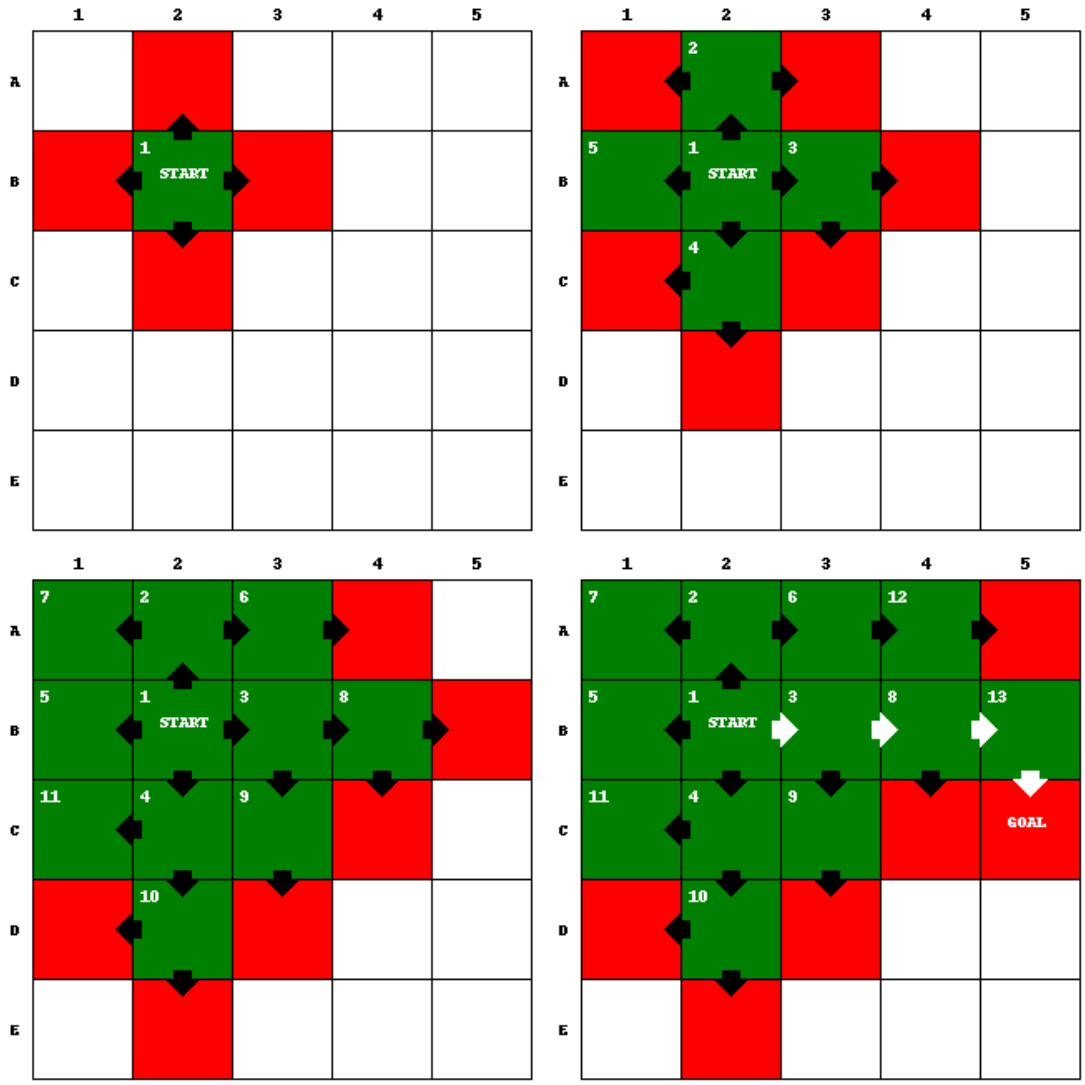
```
frontier = Queue()
frontier.put(start )
opened = []
opened.Add(start)

while not frontier.empty():
      current = frontier.get()

if current == goal:
      break

for next in neighbors(current):
      if next not in opened:
      next.parent = current
      frontier.put(next)
      opened.Add(next)
```

Ker se celice iz čakalne vrste jemljejo v istem vrstnem redu kot so vanjo vstopile, je zagotovljeno, da se iskanje širi najprej na celice z najmanjšo oddaljenostjo.

V koraku 2 se je vsaki odprti celici priredila referenca na prejšnjo (izvorno) celico. Na ta način lahko vzamemo ciljno celico, se sprehodimo po referencah navzgor do začetne celice in sestavimo dobljeno najkrajšo pot:

```
current = goal
path = [current]
while current != start:
      current = current.parent
      path.append(current)
path.reverse()
```

Poglejmo si še primer algoritma na sliki **1**. Imamo preprosto mrežo celic dimenzij 5x5. Označimo vrstice s črkami A-E in stolpce s številkami 1-5. Začetna točka (označena z S) leži v celici B2, ciljna točka (označena s C) pa v celici C5. Z rdečo so označene celice v čakalni vrsti, z modro pa še neobiskane celice. Na sliki so narisani štirje krogi širjenja po celicah pri čemer se zadnji krog predčasno zaključi, saj je bila dosežena ciljna točka.



Slika 1: Primer iskanja v širino

Takšno pa je stanje čakalne vrste za trinajst korakov, ki so potrebni za doseg cilja:

1. [~~**B2**~~, **A2**, **B3**, **C2**, **B1**]

V začetku je na čakalni vrsti le začetna točka. V prvem koraku jo vzamemo iz čakalne vrste in odpremo njene štiri sosede. Ker noben od sosedov še ni bil odprt, jih vse dodamo na čakalno vrsto.

2. [~~**A2**~~, B3, C2, B1, **A3**, **A1**]

V drugem koraku nadaljujemo z naslednjo celico iz čakalne vrste. Ker celica leži ob zgornjem robu ima le tri sosede. Spodnji sosed pa je že odprt (to je namreč začetna celica), zato na čakalno vrsto postavimo le levega in desnega soseda.

3. [~~**B3**~~,C2, B2, A3, A1, **B4**, **C3**]
4. [~~**C2**~~, B2, A3, A1, B4, C3, **D2**, **C1**]
5. [~~**B2**~~, A3, A1, B4, C3, D2, C1]

…

12. [~~A4~~, B5, C4, D3, E2, D1, A5]
13. [~~B5~~, C4, D3, E2, D1, A5, C5]

V zadnjem koraku odpremo le še ciljno celico, s tem se algoritem zaključi.

Kot že omenjeno je omejitev tega algoritma, da lahko z njim iščemo pot samo po neobteženem grafu. V prejšnjem primeru so imeli vsi premiki enako ceno – eno enoto v vsako stran. Delali smo lahko le horizontalne in vertikalne premike – diagonalni premik ima večjo ceno. Pri določenih problemih pa bi si radi različno obtežili različne dele grafa – lahko bi imeli teren, ki je težje prehoden, ali pa bi težje ocenili določene vrste premika, recimo vzvratno vožnjo ali ostro zavijanje.

2.3 Dijkstrov algoritem

Dijkstrov algoritem je nadgradnja iskanja v širino, ki upošteva različno ceno premika med sosednjima vozliščema grafa. Ideja je, da se iskanje enakomerno širi po grafu v smereh z enako skupno dolžino. Ker dolžina (oz. cena) poti ni več preprosto enaka številu prepotovanih celic, je potrebno za vsako odprto celico pomniti vsoto cen vseh celic, ki vodijo do nje. Celic se ne dodaja na čakalno vrsto, temveč na prioritetno vrsto, ki je urejena glede na akumulirano ceno celice.

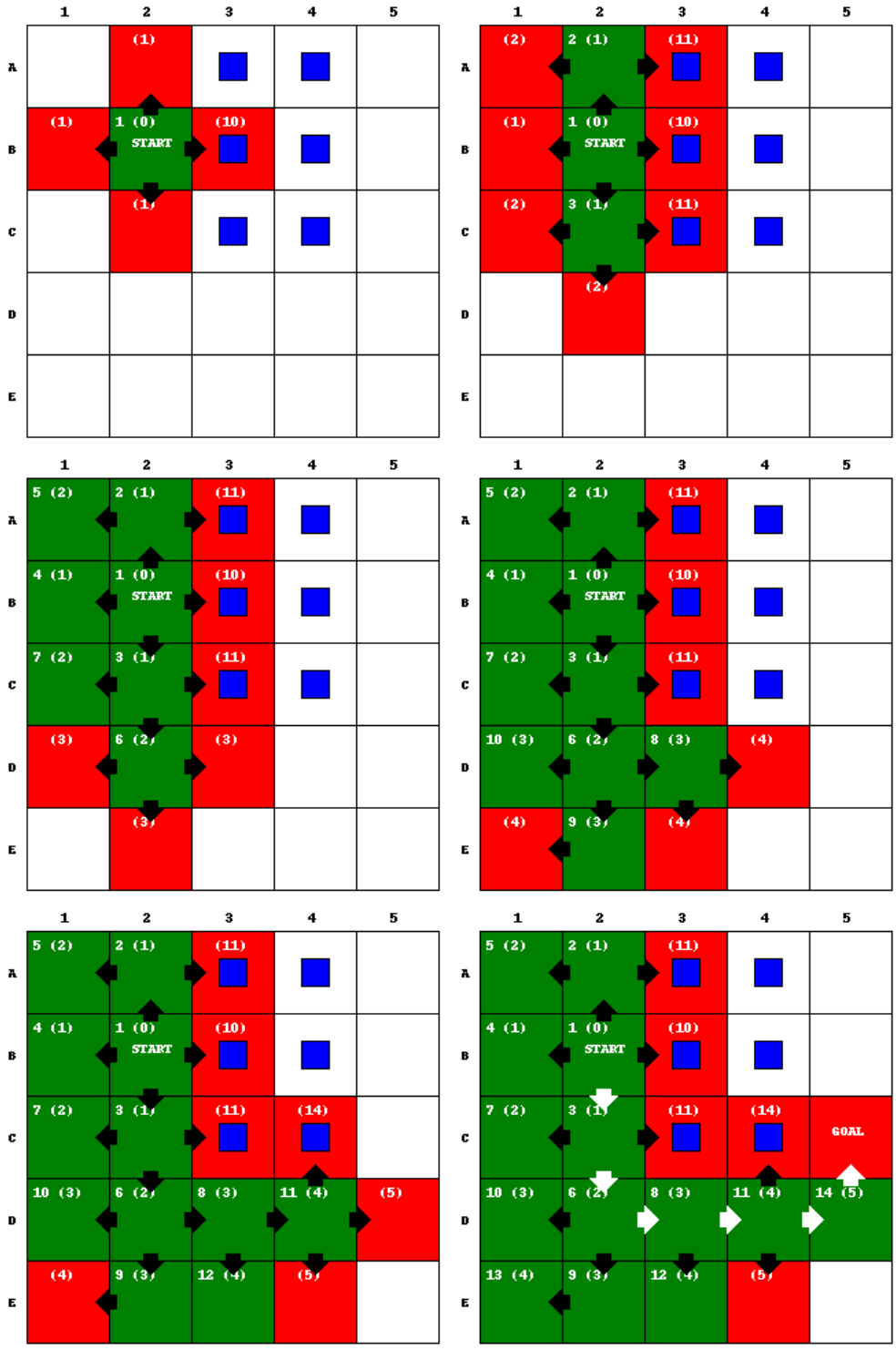
Dijkstrov algoritem se tako od iskanja po širini razlikuje v naslednjih detajlih (označeno z debelo):

1. Doda začetno celico na prioritetno vrsto in jo označi kot obiskano.
2. Vzame celico z najnižjo ceno iz prioritetne vrste.
3. Obišče vse sosede trenutne celice. Če še niso bili obiskani, ali pa je trenutna cena manjša od že najdene po drugih poteh, jih doda v prioritetno vrsto in označi kot obiskane. Cena celice za prioritetno vrsto je enaka vsoti cene izvorne celice in cene premika na novo celico. Trenutno obiskani celici priredi referenco na izvorno celico, to je na celico iz točke 2.

Algoritem opisuje naslednja psevdokoda:

```
frontier = PriorityQueue()
frontier.put(start, 0)
opened = []
cost = {}
opened.Add(start)
cost[start] = 0

while not frontier.empty():
      current = frontier.get()

if current == goal:
      break

for next in graph.neighbors(current):
      new_cost = cost[current] + cell_cost(next)
      if next not in opened or new_cost < cost[next]:
            next.parent = current
            cost[next] = new_cost
            frontier.put(next, new_cost)
            opened.Add(next)
```

Če so vse cene prehodov med celicami enake, je vrstni red celic v prioritetni vrsti enak vrstnemu redu v čakalni vrsti, tako da se Dijkstrov algoritem v takem primeru obnaša enako kot iskanje v širino. V primeru na sliki **2** so podane celice z različnimi cenami prehoda. Med začetno (START) in končno točko (GOAL) je polje celic z utežjo 10 krat večjo od ostalih celic. Pot z najnižjo ceno v tem primeru pelje okrog tega polja, česar z iskanjem v širino ne bi mogli doseči. Polje z večjo težo je označeno z modrimi pikami.



Slika 2: Primer iskanja z algoritmom Dijkstra

Stanje v prioritetni vrsti za dani primer je naslednje:

1. [~~B2(0)~~, A2(1), C2(1), B1(1), B3(10)]

Prvi korak algoritma je podoben kot pri iskanju v širino. Iz prioritetne vrste vzamemo začetno točko, obiščemo sosede in jih dodamo na prioritetno vrsto.

Pomembna razlika pa je v vrstnem redu, v katerem so postavljeni v vrsto. Celica B3, ki je bila sicer obiskana druga po vrsti, je postavljena na zadnje mesto, saj ima večjo ceno od ostalih (10).

2. [~~A2(1)~~, A2(1), C2(1), B1(1), **A1(2)**, B3(10), **A3(11)**]
3. [~~C2(1)~~, B1(1), A1(1), **D2(2)**, C1(2), B3(10), A3(10), **C3(11)**]
4. [~~B1(1)~~, A1(1), D2(2), C1(2), B3(10), A3(10), C3(11)]
5. [~~A1(1)~~, D2(2), C1(2), B3(10), A3(10), C3(11)]
6. [~~D2(2)~~, C1(2), D3(3), **E2(3)**, **D1(3)**, B3(10), A3(10), C3(11)]
7. [~~C1(2)~~, D3(3), E2(3), D1(3), B3(10), A3(10), C3(11)]
8. [~~D3(3)~~, E2(3), D1(3), **D4(4)**, **E3(4)**, B3(10), A3(10), C3(11)]
9. [~~E2(3)~~, D1(3), D4(4), E3(4), **E1(4)**, B3(10), A3(10), C3(11)]
10. [~~D1(3)~~, D4(4), E3(4), E1(4), B3(10), A3(10), C3(11)]
11. [~~D4(4)~~, E3(4), E1(4), **D5(5)**, **E4(5)**, B3(10), A3(10), C3(11), **C4(14)**]
12. [~~E3(4)~~, E1(4), D5(5), E4(5), B3(10), A3(10), C3(11), C4(14)]
13. [~~E1(4)~~, D5(5), E4(5), B3(10), A3(10), C3(11), C4(14)]
14. [~~D5(5)~~, E4(5), **C5(6)**, B3(10), A3(10), C3(11), C4(14)]

Dijkstrov algoritem nam torej omogoča iskanje poti po različno obteženem grafu. Različne uteži pa nam ne omogočajo le iskanja okrog težje prehodnih terenov kot v prejšnjem primeru. Dovolijo nam uporabo grafov, ki niso oblikovani v pravilne mreže, na primer opis ulic in križišč ali pa zveznih stanj mobilnega vozila.

2.4 Požrešna metoda (Greedy best first)

Dosedanja algoritma širita iskanje po grafu v vse smeri, kar je računsko precej potratno. Če poznamo lokacijo cilja, bi bilo bolj smotrno iskanje širiti v smeri proti cilju. V tem poglavju bo opisana najbolj naivna metoda, ki izkorišča to informacijo.

Požrešna metoda se tako imenuje, ker v vsakem koraku algoritma odpira tisto celico, ki leži najbližje v smeri cilja. Metoda poišče rešitev v veliko manjšem številu korakov, vendar pa ne zagotavlja najkrajše možne poti.

Ta algoritem, podobno kot Dijkstra oceni vsako obiskano celico, vendar namesto prepotovane dolžine uporablja oceno preostale dolžine do cilja. Tej oceni pravimo hevristika. Za hevristiko pogosto uporabljamo Evklidovo (zračno) razdaljo ali razdaljo Manhattan.

Algoritem poteka podobno kot Dijkstra (označeno z debelo):

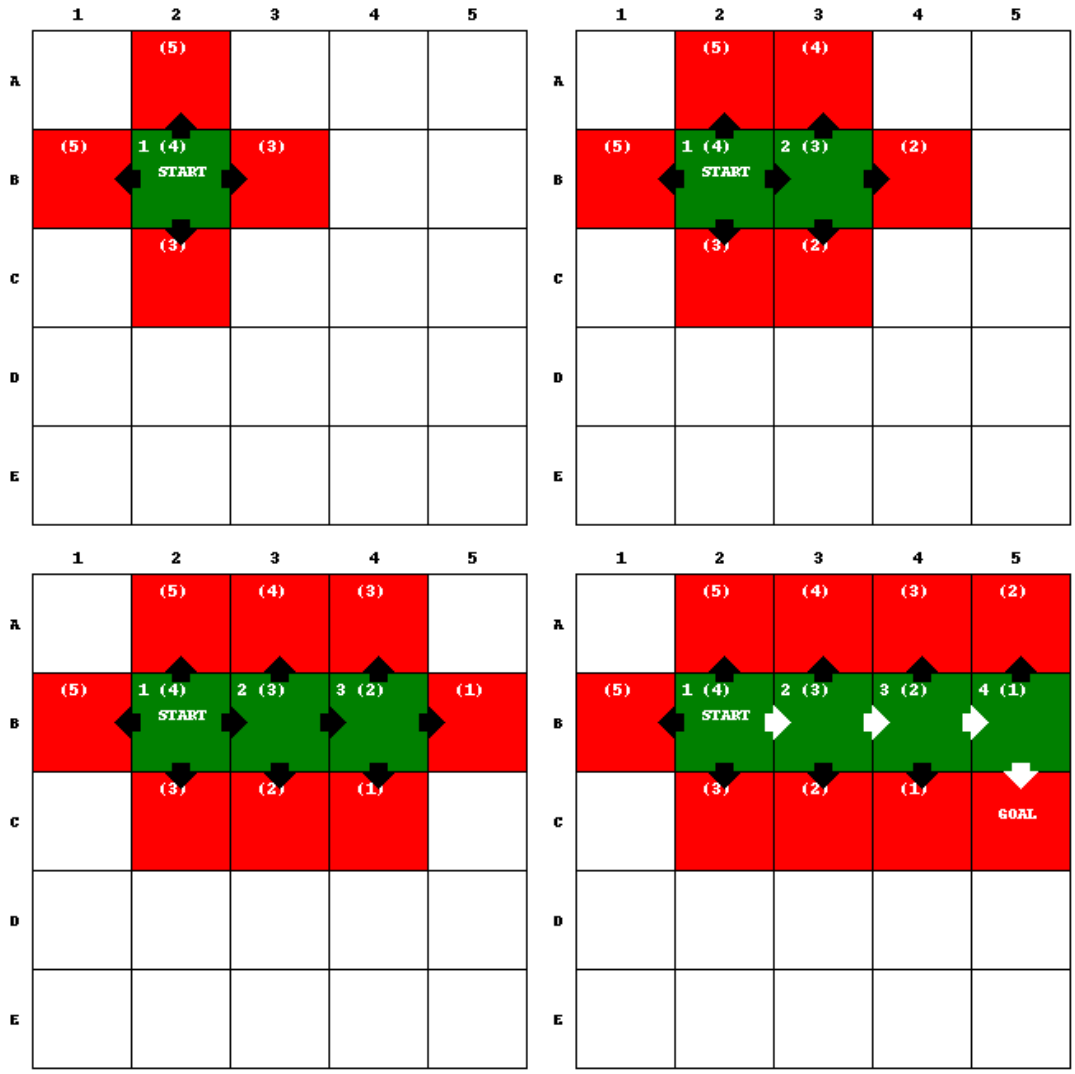
1. Doda začetno celico na prioritetno vrsto in jo označi kot obiskano.
2. Vzame celico z najnižjo ceno iz prioritetne vrste.

3. Obišče vse sosede trenutne celice. Če še niso bili obiskani, jih doda v prioritetno vrsto in označi kot obiskane. Cena celice za prioritetno vrsto je enaka oceni preostale dolžine do ciljne celice (hevristika). Trenutno obiskani celici priredi referenco na izvorno celico, to je na celico iz tocke 2.
4. Skoči na korak 2 in ponavlja dokler ni obiskana ciljna celica.

Psevdokoda se od Dijkstre razlikuje le v ceni dodani v prioritetno vrsto:

```
frontier = PriorityQueue()
frontier.put(start, 0)
opened = []
opened.Add(start)

while not frontier.empty():
      current = frontier.get()

if current == goal:
      break

for next in neighbors(current):
      if next not in opened:
            next.parent = current
            frontier.put(next, heuristic(goal, next))
            opened.Add(next)
```

Na sliki **3** si lahko ogledamo primer, ki smo ga uporabili že pri iskanju v širino. Uporabljena hevristika je razdalja Manhattan.



Slika 3: Primer iskanja s požrešno metodo

Namesto 13-ih korakov kot pri iskanju v širino, se ta algoritem zaključi že po 4-h korakih:

1. [~~**B2(4)**~~, **B3(3)**, **C2(3)**, **A2(5)**, **B1(5)**]

Kot v prejšnjih algoritmih v prvem koraku na prioritetno listo dodamo vse štiri sosede začetne celice. Vrstni red dodanih celic pa je sedaj spremenjen tako, da je na prvem mestu celica, ki leži najbližje cilju (celici B3 in C2 imata najmanjšo Manhattan razdaljo do cilja).

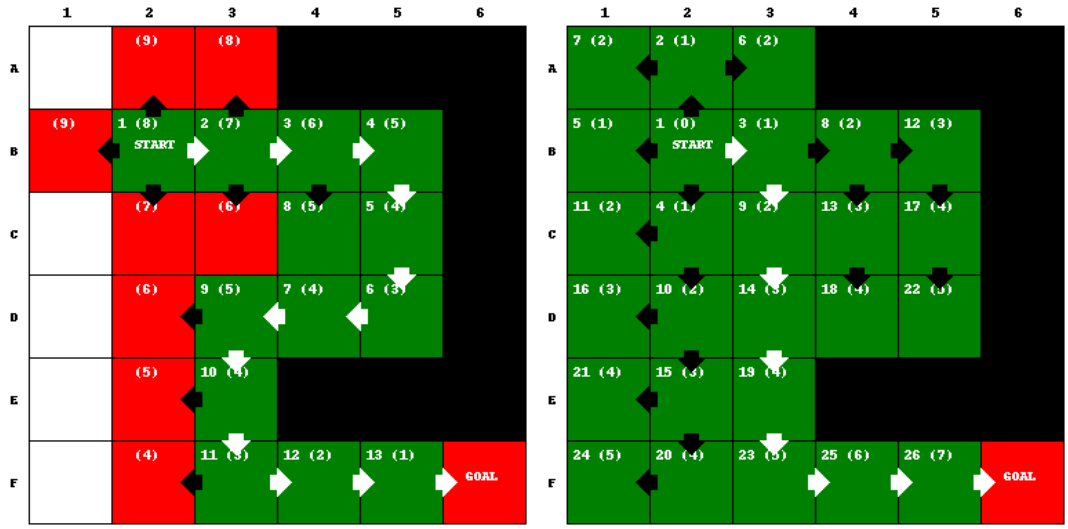
2. [~~**B3(3)**~~, **B4(2)**, **C3(2)**, C2(3), **A3(4)**, A2(5), B1(5)]

V drugem koraku odpiramo eno od celic najbližje cilju. Na prvo mesto v vrsti spet dodamo celici, ki sta se približali cilju za dodatno enoto.

3. [~~**B4(2)**~~, **B5(1)**, **C4(1)**, C3(2), C2(3), **A4(3)**, A3(4), A2(5), B1(5)]
4. [~~**B5(1)**~~, **C5(0)**, C4(1), C3(2), **A5(2)**, C2(3), A4(3), A3(4), A2(5), B1(5)]

V četrtem koraku odpremo le še celico neposredno ob cilju, s čemer zaključimo algoritem.

Vidimo lahko, da je požrešna metoda precej bolj učinkovita v številu korakov potrebnih za dosego cilja. Vendar pa ima ta metoda slabost pri iskanju bolj kompleksnih poti, konkretno v primerih, kjer direktno pot sekajo konkavne ovire (slepe ulice). V teh primerih metoda najprej razišče slepi odsek (ki je najbližje cilju) nato pa zavije okrog ovire in na končni poti ohrani nepotrebne celice iz tega odseka. Končna pot je zato daljša kot je potrebno, saj spotoma zavije še v konkavni predel, kot je prikazano v primeru na sliki **4**.



Slika 4: Primerjava požrešne metode (levo) in Dijkstre (desno)

2.5 Osnovni A* algoritem

A* algoritem ponuja idealen kompromis med številom odprtih vozlišč in zahtevo po najkrajši poti. Pogosto se uporablja za načrtovanje poti v računalniskih igrah, kjer je prostor razdeljen na diskretna kvadratna polja. A* zagotavlja optimalno rešitev in je popularen zaradi svoje računske učinkovitosti.

A* izbira pot, ki minimizira:

$$F(n) = g(n) + h(n) \tag{2.1}$$

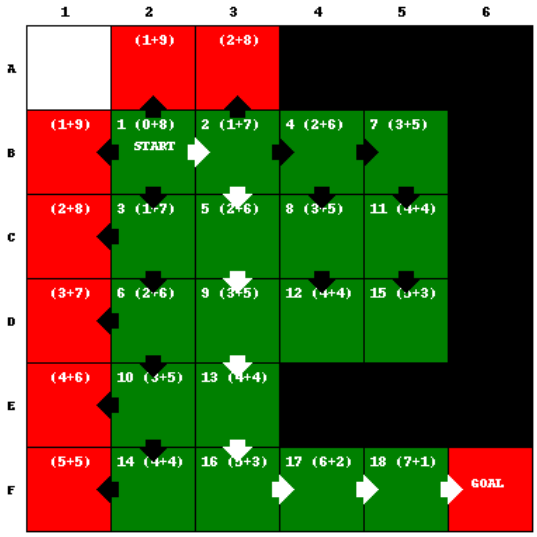
Kjer je g(n) cena (dolžina) poti med začetno in trenutno celico, h(n) pa hevristika, ki estimira ceno med trenutno in končno celico. G(n) je enaka ceni, ki jo uporablja Dijkstra, h(n) pa je enaka hevristiki požrešne metode.

1. Doda začetno celico na prioritetno vrsto in jo označi kot obiskano.
2. Vzame celico z najnižjo ceno iz prioritetne vrste. Če imata celici enako ceno, izberi celico z manjšo ceno brez hevristike.

3. Obišče vse sosede trenutne celice. Če še niso bili obiskani, ali pa je trenutna cena manjša od že najdene po drugih poteh, jih doda v prioritetno vrsto in označi kot obiskane. Cena celice za prioritetno vrsto je enaka vsoti cene izvorne celice in cene premika na novo celico povečani za oceno oddaljenosti od cilja. Trenutno obiskani celici priredi referenco na izvorno celico, to je na celico iz točke 2.
4. Skoči na korak 2 in ponavlja dokler ni obiskana ciljna celica.

Psevdokoda je torej združitev Dijkstre in požrešne metode:

```
frontier = PriorityQueue()
frontier.put(start, 0)
opened = []
cost = {}
opened.Add(start)
cost[start] = 0

while not frontier.empty():
      current = frontier.get()

if current == goal:
      break

for next in graph.neighbors(current):
      new_cost = cost[current] + cell_cost(next)
      if next not in opened or new_cost < cost[next]:
            next.parent = current
            cost[next] = new_cost
            priority = new_cost + heuristics(next, goal)
            frontier.put(next, priority)
            )
            opened.Add(next)
```

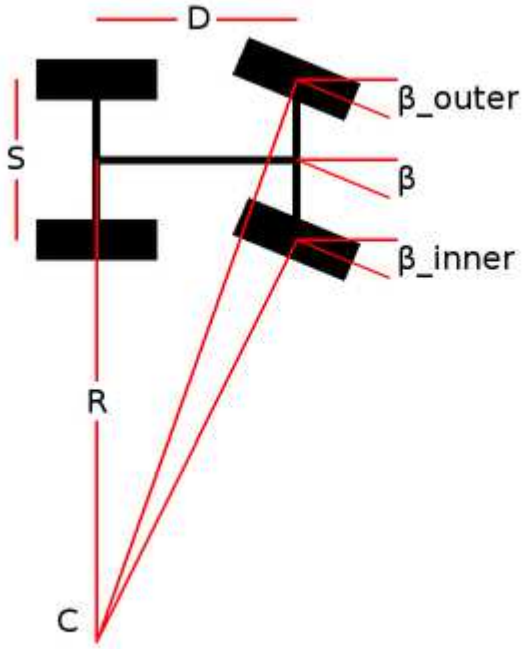


Slika 5: Primer iskanja z algoritmom A*

Na sliki 5 je algoritem A* prikazan na primeru, v katerem je požrešna metoda našla predolgo pot. Opazimo, da je algoritem našel enako pot kot Dijkstra, hkrati pa se je bolj optimalno širil v smeri cilja.

3 Ackermannov model vozila

Ackermannov model vozila je model, ki je uporabljen pri današnjem avtomobilskem pogonu. Lastnost takega pogona je, da se prednji kolesi obračata vsak pod svojim kotom. Notranje kolo se obrača pod večjim kotom kot zunanje, tako da se center obračalnega kroga vozila prestavi nazaj in leži na premici, ki vodi skozi zadnjo os vozila. To omogoča, da zadnji kolesi ne zdrsujeta pri zavijanju.



Slika 6: Ackermannov model

Na sliki **6** so označeni koti prednjih koles (β_inner in β_outer), medosna razdalja in širina vozila (D in S), ter radij in center obračanja vozila (R in C). Zaradi razlike v kotih β_outer in β_inner leži center obračanja na premici, ki vodi skozi zadnjo os. Radij obračanja je odvisen od dolžine in širine vozila ter kotov prednjih koles. Na

sliki je označen tudi povprečni kot prednjih koles (β). S poenostavitvijo modela je radij obračanja odvisen samo še od dolžine vozila in povprečnega kota krmiljenja.

Za Ackermannov model lahko definiramo relacije med kotom krmiljenja, medosno razdaljo in radijem obračalnega kroga na naslednji način:

$$\tan\left(\frac{\pi}{2}-\beta_{\text{outer}}\right)=\frac{R+\frac{S}{2}}{D} \tag{3.1}$$

$$\tan\left(\frac{\pi}{2}-\beta_{\text{inner}}\right)=\frac{R-\frac{S}{2}}{D} \tag{3.2}$$

Za potrebe simulacije bomo enačbo poenostavili z uporabo povprečnega kota prednjih koles:

$$\tan\left(\frac{\pi}{2}-\beta\right)=\frac{R}{D} \tag{3.3}$$

S tem ne izgubimo na splošnosti modela, saj sta posamezna kota odvisna le od povprečnega kota ter konstante S in ju lahko po potrebi še vedno poiščemo.

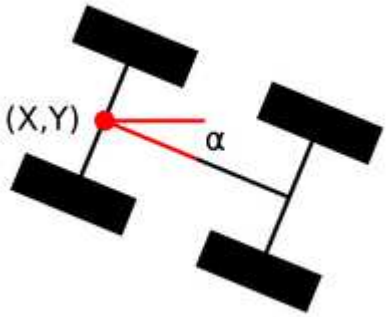
Krmilni kot pa lahko izrazimo:

$$\beta=\frac{\pi}{2}-\arctan\left(\frac{R}{D}\right) \tag{3.4}$$

3.1 Prostor parametrov Ackermannovega vozila

Iskanje poti bomo izvajali v prostoru parametrov Ackermannovega vozila. Prostor ima poleg dveh prostorskih dimenzij še rotacijo vozila:

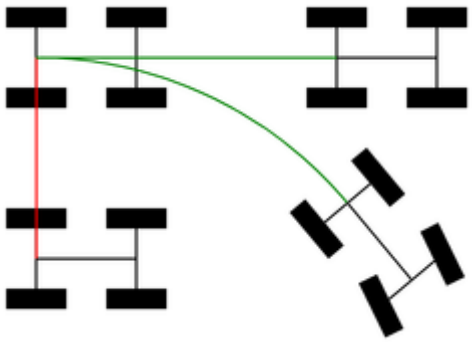
1. horizontalna koordinata (X)
2. vertikalna koordinata (Y)
3. rotacija vozila (α) – enaka 0 v horizontalni smeri in narašča v nasprotni smeri urinega kazalca



Slika 7: Prostor parametrov Ackermannovega vozila

Na sliki **7** so prikazani vsi trije parametri. Center vozila bomo postavili na sredino zadnje osi, ničelni kot vozila pa bo pomenil smer v X osi.

3.2 Neholonomičnost



Slika 8: Primeri gibanja za neholonomično vozilo

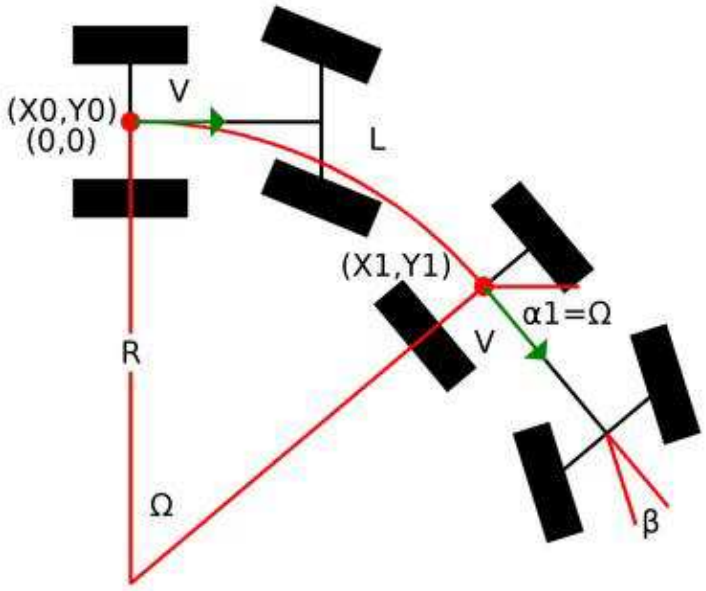
Ackermannov model vozila je neholonomičen sistem. To pomeni, da se vozilo v danem trenutku ne more gibati v poljubni smeri. Vozilo se recimo ne more premakniti vstran brez spremembe lastnega kota (slika **8**), vozila, katerih maksimalni krmilni kot je manjši od pravega kota, pa poleg tega ne morejo na mestu spremeniti svojega kota. Če želimo spremeniti kot vozila moramo hkrati tudi spremeniti njegovo pozicijo (primer desno spodaj na sliki **8**).

3.3 Parametri vodenja Ackermannovega vozila

Vozilo ima dva parametra vodenja:

1. hitrost (V)
2. kot krmiljenja (β) – pozitiven za levi obrat in negativen za desni obrat

Zanima nas sprememba koordinat in kota vozila pri določenih parametrih vodenja v nekem časovnem obdobju T. Enačbe bomo izpeljali za izhodiščno pozicijo (X0=0, Y0=0, α0=0), rešitev za vse ostale pozicije (X0', Y0', α0') nato dobimo z rotacijo s kotom α0' in translacijo na (X0', Y0').



Slika 9: Ackermannov premik pri danih parametrih vodenja

Potrebovali bomo radij obračalnega kroga pri danem kotu krmiljenja:

$$R = D * \tan\left(\frac{\pi}{2} - \beta\right) \tag{3.5}$$

Nato nas zanima dolžina loka, ki ga vozilo prepotuje z dano hitrostjo v času T:

$$L = V * T \tag{3.6}$$

Kot tega loka je nato:

$$\Omega = L/R \tag{3.7}$$

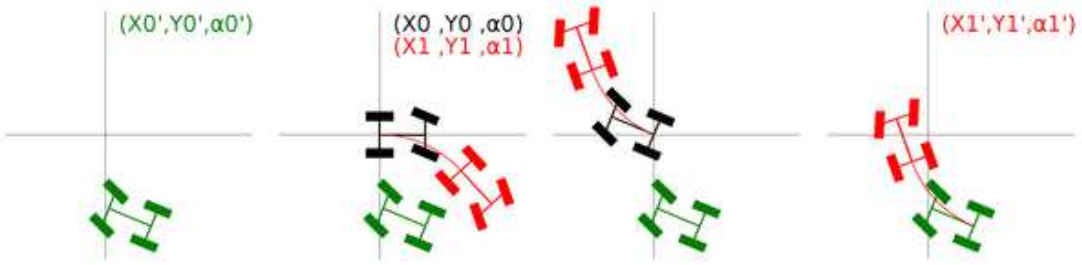
Kot vozila ($\alpha 1$) je zaradi podobnih trikotnikov enak kotu loka:

$$\alpha 1 = \Omega \tag{3.8}$$

Nove koordinate pa dobimo s pomočjo preproste trigonometrije:

$$X1 = R * \sin(\Omega) \tag{3.9}$$

$$Y1 = R * (1 - \cos(\Omega)) \tag{3.10}$$



Slika 10: Ackermannov premik iz poljubnega stanja

Ackermannov premik lahko na poljubnem stanju (X0' ≠ 0, Y0' ≠ 0, α0' ≠ 0) izvedemo tako, da nove koordinate rotiramo z začetnim kotom (α0') in transliramo z začetno pozicijo (X0', Y0'):

$$X1' = \cos(\alpha 0') * X1 + \sin(\alpha 0') * X2 + X0' \tag{3.11}$$

$$Y1' = \sin(\alpha 0') * X1 - \cos(\alpha 0') * X2 + Y0' \tag{3.12}$$

$$\alpha 1' = \alpha 1 + \alpha 0' \tag{3.13}$$

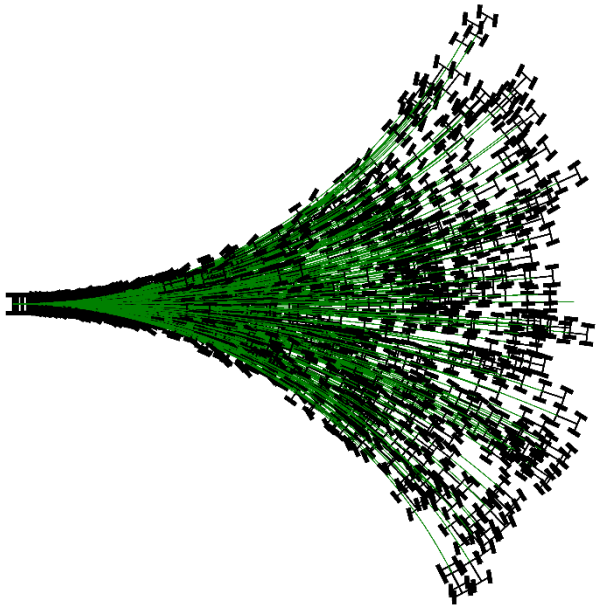
Na sliki 10 je prikazan premik iz poljubnega stanja. Z zeleno je označeno izvorno stanje. Najprej izvedemo premik iz ničelnega stanja (drugi del slike), nato dobljeno stanje rotiramo za kot izvornega stanja (tretji del slike) in transliramo za pozicijo izvornega stanja (četrti del slike).

4 Hibridni A* algoritem

Hibridni A* je v osnovi A* algoritem, ki išče pot po zveznih vozliščih grafa, vendar prostor iskanja hkrati diskretizira. S pomočjo diskretiziranega prostora se graf iskanja zmanjša na tak način, da v vsaki celici diskretnega prostora lahko obstane samo eno vozlišče grafa, ki pa še vedno hrani zvezno informacijo. Vsa ostala vozlišča se zavržejo, iskanje po grafu se pri njih konča. Če v celico pade več vozlišč, se odločimo za tisto vozlišče, ki ima najnižjo ceno, glede na izbrano hevristiko. Tako graf iskanja še vedno predstavlja stanja v zveznem prostoru, največje število preiskanih vozlišč pa je omejeno s številom celic diskretnega prostora.

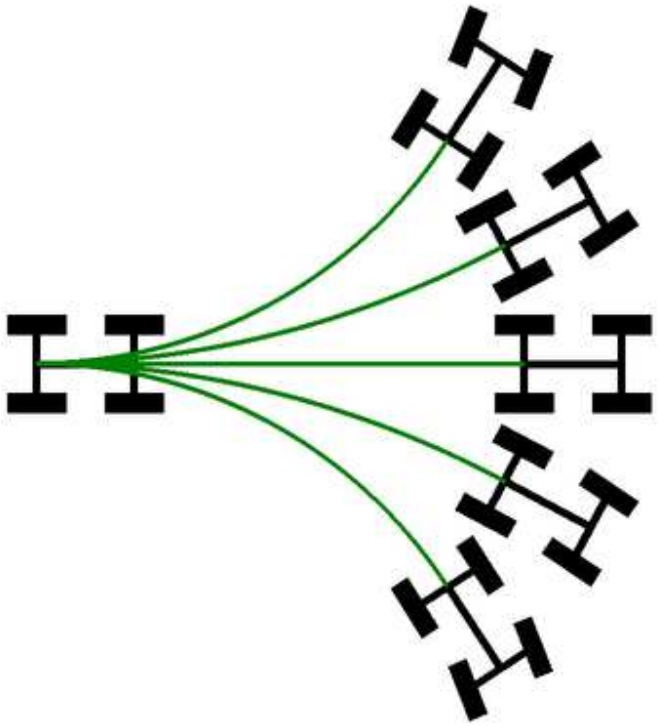
4.1 Vozlišče grafa iskanja

Če vozlišče grafa definiramo s tremi zveznimi parametri Ackermannovega modela, je število vozlišč grafa za iskanje po taksnem zveznem prostoru neskončno. Še huje, število vozlišč je neskončno na poljubno majhnem delu prostora.



Slika 11: Nekaj od neskončno stanj dosegljivih v končnem prostoru

Za iskanje po grafu moramo biti sposobni prostor parametrov predstaviti v neki diskretizirani obliki. Namesto diskretiziranja samih parametrov (kar vodi v le odsekoma zvezne poti) bomo najprej diskretizirali premik iz enega stanja v drugega. Diskretni premik bo legalen premik Ackermannovega vozila za eno enoto dolžine s konstantnim obračalnim radijem. Set vseh možnih premikov iz enega stanja bomo dodatno zmanjšali s končnim številom kotov krmiljenja.

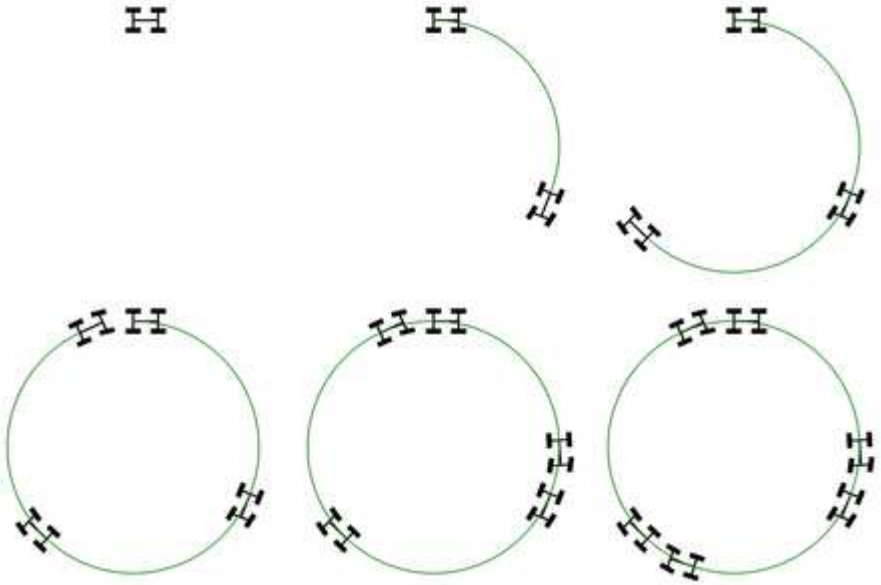


Slika 12: Set diskretnih premikov med vozlišči grafa v zveznem prostoru

Na sliki 12 je prikazano izhodiščno vozlišče grafa, ki vsebuje polno informacijo o poziciji in rotaciji v prostoru. Iz njega izhaja končno število enako dolgih poti, ki zadoščajo Ackermannovemu premiku s konstantnim kotom krmiljenja. Kote krmiljenja izberemo enakomerno iz intervala, ki ga določajo omejitve vozila (od najmanjšega do največjega kota, ki ga vozilo zmore). Poleg vožnje naprej dodamo še enako število vzvratnih poti. Set teh premikov vodi v nova vozlišča grafa. Dolžina poti in število poti je konstanta določene implementacije.

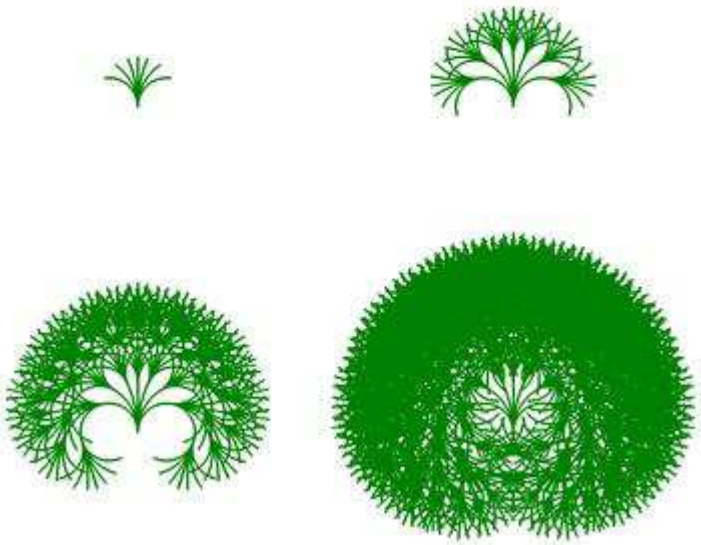
Na ta način smo v vsakem vozlišču ohranili polno informacijo o stanju vozila, v vsakem vozlišču bomo sledili končnemu številu povezav, iz vsake povezave pa bomo lahko pozneje rekonstruirali zvezen odsek poti. Dobljeni graf pa še ni končen na omejenem delu prostora. Najbolj očiten primer za to je kroženje vozila po krožnici (Slika 13). Če je razmerje med obsegom kroga in dolžino koraka iracionalno število,

lahko isti krog obkrožimo poljubno mnogokrat brez da bi se dvakrat ustavili na isti točki na krožnici.



Slika 13: Primer neskončnega števila stanj v končnem območju

Naslednji problem pa je eksponentno večanje števila vozlišč grafa. Na sliki 14 so prikazane štiri globine širjenja. V vsakem širjenju se število odprtih vozlišč poveča za faktor diskretnih Ackermannovih premikov, kar vodi v vedno večjo gostoto vozlišč na majhnem delu prostora.



Slika 14: Eksponentna rast prostora iskanja

Običajni algoritem A* bi zato pregledal izredno velik del grafa, saj bi odprl mnogo zelo podobnih stanj. Cilj hibridnega A* algoritma je omejiti graf iskanja z

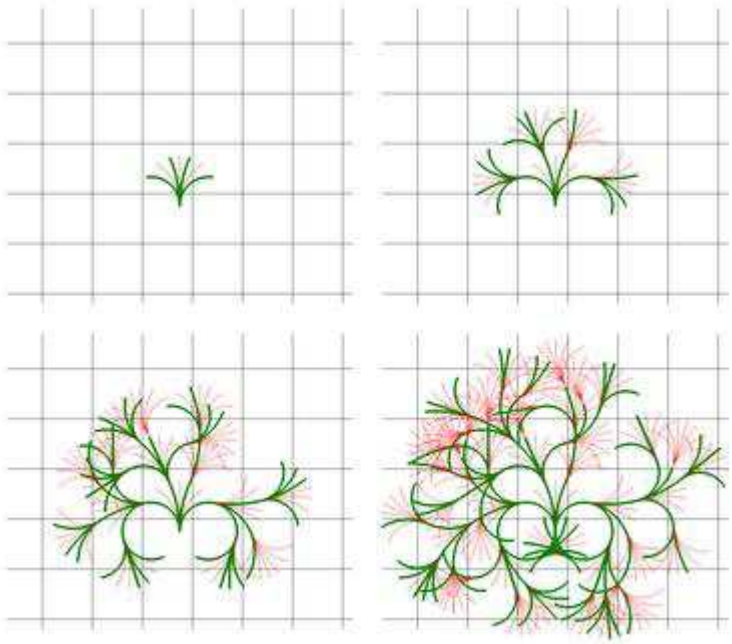
izločanjem vseh razen enega od stanj, ki so si med seboj preveč podobna. Stanja, ki so si podobna definiramo kot tista, ki jim priredimo isto celico diskretiziranega prostora, kot je razloženo v nadaljevanju.

4.2 Izločanje z diskretizacijo

Izločanje izvajamo z namenom zmanjšanja gostote vozlišč v prostoru parametrov. To dosežemo z diskretizacijo 3-dimenzionalnega prostora parametrov. Lokacijo v prostoru diskretiziramo s kvadratno mrežo z določeno dolžino stranic celic. Orientacijo v prostoru pa diskretiziramo na končno število intervalov rotacij vozila. Diskretni prostor podvojimo še za vzvratno vožnjo, ker ne želimo, da bi vožnja naprej vplivala na zmanjšanje prostora iskanja vzvratne vožnje. Vsakemu vozlišču grafa bomo priredili eno celico diskretnega prostora. Dva vozlišča, ki ležita v isti diskretni lokaciji, se še vedno lahko nahajata v ločenih celicah, če se njuni orientaciji diskretizirata v različna intervala orientacij ali pa, če smo do enega vozlišča prispeli z vožnjo naprej, do drugega pa z vzvratno vožnjo.

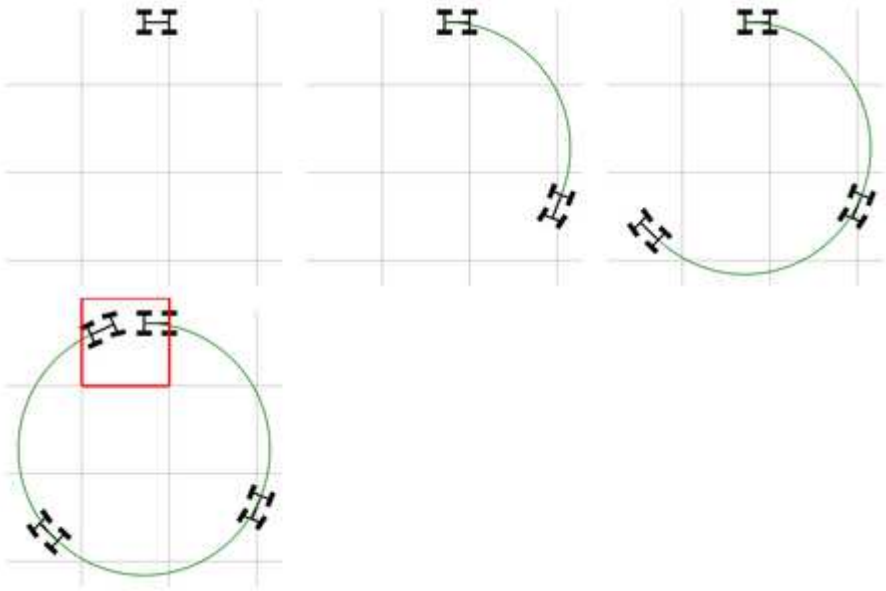
V vsaki celici diskretnega prostora bomo dovolili širjenje le enemu vozlišču, ki leži v tej celici. Takoj ko iskanje razširimo prek prvega vozlišča (ki ima najmanjšo ceno – je najboljši kandidat za optimalno pot), se celica označi. Če pride v iskanju na vrsto vozlišče, ki leži v označeni celici, se širjenje tam prekine in vozlišče zavrže.

Nas sliki 15 je prikazana diskretizacija s štirimi intervali rotacij vozila. V vsaki prostorski celici zato lahko ležijo največ štiri vozlišča (eno iz intervala $(0,\pi/2)$, eno iz intervala $(\pi/2,\pi)$ itn.).



Slika 15: Enakomerna gostota prostora iskanja z uporabo izločanja

Na sliki 16 je prikazano izločanje vozlišč na primeru kroženja. Ker je v četrtem koraku novo vozlišče prispelo v diskretno celico, ki jo že zaseda eno od predhodnih vozlišč, je bilo zavrženo. Pomembno je poudariti, da do izločanja ni prišlo samo zaradi iste diskretizirane X in Y koordinate, temveč imata vozlišči tudi enako smer vožnje in isto diskretizirano rotacijo. Če bi povečevali število rotacijskih celic, bi v nekem trenutku obe vozlišči ležali v različnih celicah in ne bi prišlo do izločanja. Večanje števila celic sicer vodi v bolj optimalno najdeno pot, vendar pa to pomeni večje število preiskanih vozlišč in s tem večjo računsko zahtevnost algoritma.



Slika 16: Izločanje vozlišč z diskretnimi celicami

4.3 Cena vozlišca

Cena vozlišča je v najpreprostejšem primeru skupna dolžina vseh poti, ki vodijo do tega vozlišča. Takšna cena bo iskanje vodila v odpiranje vozlišč, po katerih je dobljena pot najkrajša. Pogosto pa dolžina poti ni edini kriterij optimalnosti, saj bi radi da pot ne dela naglih zavojev ali pogosto prehaja v vzvratno vožnjo. Zato bomo vozlišča dodatno obtežili za naslednje primere:

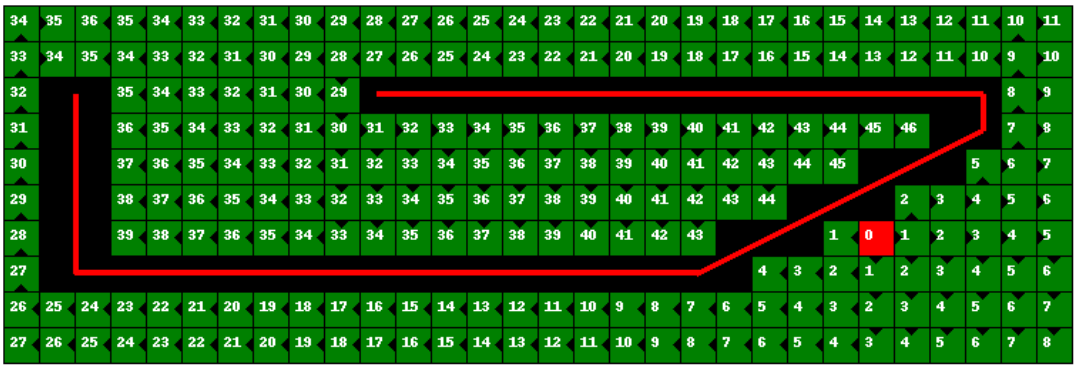
1. Vzvratna vožnja
2. Sprememba vzvratne vožnje
3. Sprememba kota krmiljenja

4.4 Hevristika

Najpreprostejša izbira za hevristiko je kar Evklidova razdalja. Problem take preproste hevristike je v tem, da vodi iskanje poti v slepe ulice, ki ležijo v smeri cilja.

Ker lahko iskanje v zveznih koordinatah odpre precej veliko število vozlišč je bolje izbrati bolj zahtevno hevristiko, ker tako vodimo iskanje stran od slepih poti in zmanjšamo prostor iskanja.

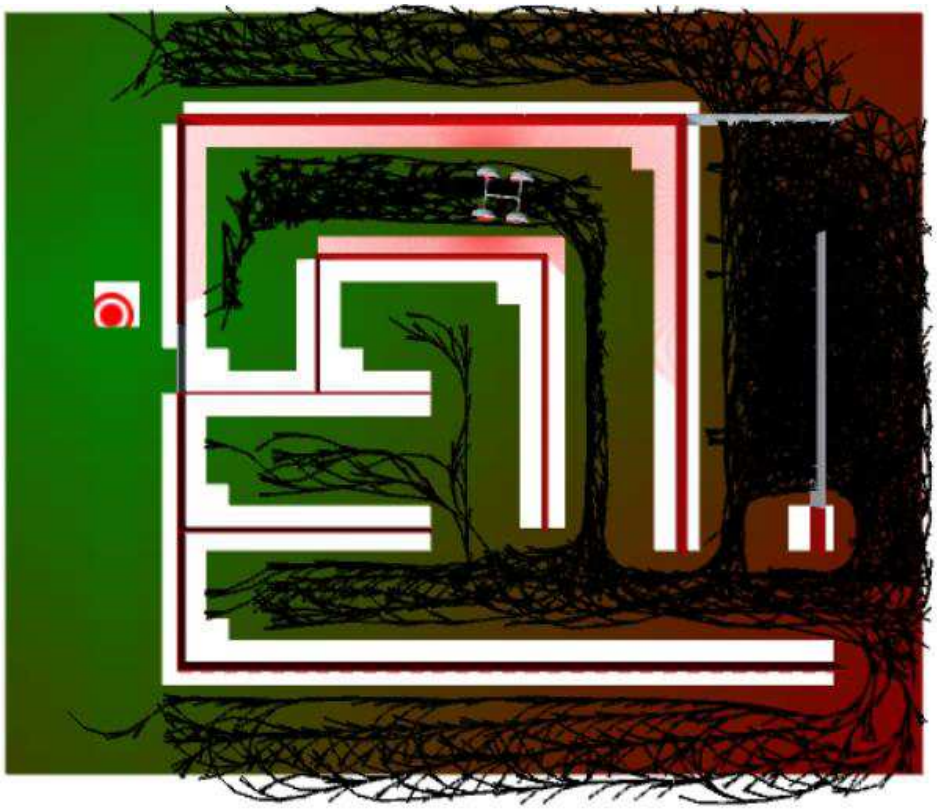
Prostor v okolici vozila bomo razdelili na kvadratno mrežo, ki naj objema vse znane ovire in cilj. Vse celice v katerih ležijo ovire bomo označili kot nedosegljive. V vsaki celici mreže bomo določili najkrajšo pot do cilja s pomočjo algoritma Dijkstra s pričetkom v ciljni celici. Širjenje algoritma bomo nadaljevali po vseh dosegljivih celicah mreže in v vsaki shranili trenutno ceno poti, ki jo bomo v hibridnem algoritmu uporabili za hevristiko. Ta korak se morda zdi potraten zaradi širjenja po celotnem prostoru, vendar je diskretni graf precej manjši od zveznega in vpliv na vodenje zveznega iskanja v optimalno smer odtehta začetni vložek.



Slika 17: Diskretna hevristika z upoštevanjem ovir

Primer iskanja hevristike je prikazan na sliki 17. Rdeče črte prikazujejo neprehodne ovire, rdeča celica pa označuje območje v katerem leži cilj. Črno obarvane celice so tiste, ki jih prečkajo ovire in jih zato označimo kot nedosegljive. Vse preostale celice so označene z dolžino poti, ki je potrebna za doseg celice z algoritmom Dijkstra. Vidimo, da je slepi odsek, ki sicer leži blizu ciljne celice, visoko obtežen, zaradi česar bo zvezno iskanje vodeno v bolj optimalno smer.

Na slikah 18 in 19 lahko vidimo primerjavo med širjenjem iskanja pod vplivom preproste hevristike z Evklidovo razdaljo in hevristike s pomočjo diskretnega grafa. Barve opisujejo vrednost hevristike (zelena manjšo, rdeča večjo).



Slika 18: Evklidova hevristka v simulaciji

Slika 19: Hevristika z diskretnim grafom v simulaciji

4.5 A*

Algoritem ob dosedaj naštetih detajlih poteka enako kot običajni A*. Najprej povzemimo razlike med hibridnim in diskretnim A*:

1. Vozlišče je podano v zveznem prostoru Ackermannovih parametrov.
2. Sosednja vozlišča so le tista v katera lahko prispemo z enim iz seta Ackermannovih premikov iz območja možnih krmilnih kotov z isto razdaljo gibanja.
3. Set vozlišč dodatno zmanjšamo tako, da v diskretni celici 3-dimenzionalnega prostora parametrov ohranimo le eno vozlišče.
4. Graf je utežen glede na vrsto premika, vsaka utež je enaka vsoti dolžine premika in cene vzvratne vožnje, spremembe smeri in spremembe krmilnega kota.
5. Celotna hevristika se določi v predkoraku algoritma za celotno okolico znanega prostora v diskretni obliki.

Rezultat hibridnega algoritma A* je drevo vozlišč povezanih z Ackermannovimi premiki, ki ima v korenu začetno pozicijo vozila v enem od končnih vozlišč pa ciljno pozicijo. S prehodi od ciljnega do začetnega vozlišča lahko nato ustvarimo listo zaporednih Ackermannovih premikov, ki opisujejo gladko pot od začetne do ciljne pozicije.

Dobljena pot opisuje pozicijo zadnje osi vozila, za vodenje pa bomo potrebovali pot, ki jo mora prepeljati prednja os. V ta namen bo zadnji korak algoritma translacija krivulje zadnje osi v krivuljo prednje osi na tak način, da se vsaka točka krivulje prestavi za dolžine medosne razdalje v smeri tangente na krivuljo v tej točki.

Konkretna implementacija hibridnega A* je opisana v poglavju 6.7.2 Iskanje poti na strani 50, implementacija translacije v prednjo os pa v poglavju 6.7.3 Rekonstrukcija poti sprednje in zadnje osi na strani 51.

5 Vodenje vozila po gladki poti

Za vodenje vozila bomo uporabili metodo uporabljeno pri vozilu Stanley[4]. Vodenje vozila poteka po referenčni poti prednje osi. Regulator je nelinearna funkcija pogreška e, ki je določen z oddaljenostjo centra prednje osi (x,y) od najbližje točke na referenčni poti (p_x,p_y):

$$e = \sqrt{(p_x - x)^2 + \left(p_y - y\right)^2} \tag{5.1}$$

Prvi člen regulatorja samo skrbi, da so prednja kolesa obrnjena v smeri referenčne poti tako, da nastavi krmilni kot enak razliki kota poti (α_p) in rotacije vozila (α):

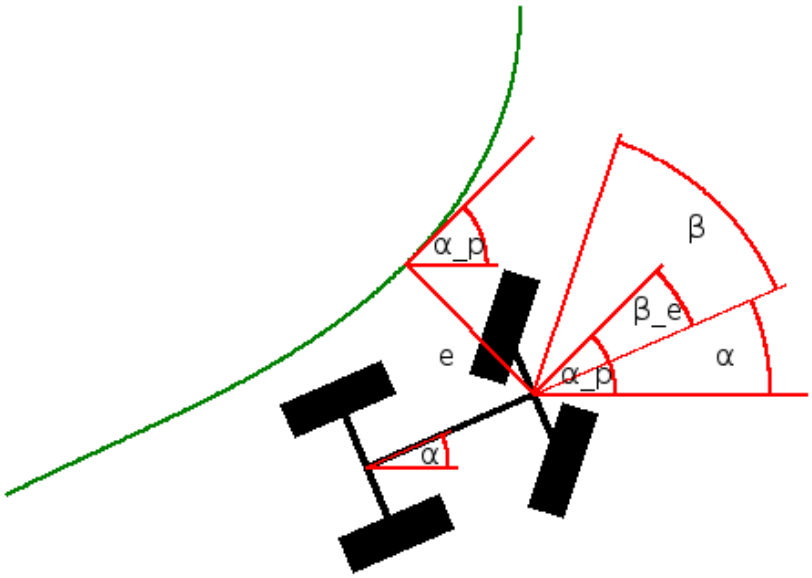
$$\beta_e = \alpha_p - \alpha \tag{5.2}$$

Za neničelne pogreške pa drugi člen prilagodi krmilni kot β v smeri tangente na referenčno pot:

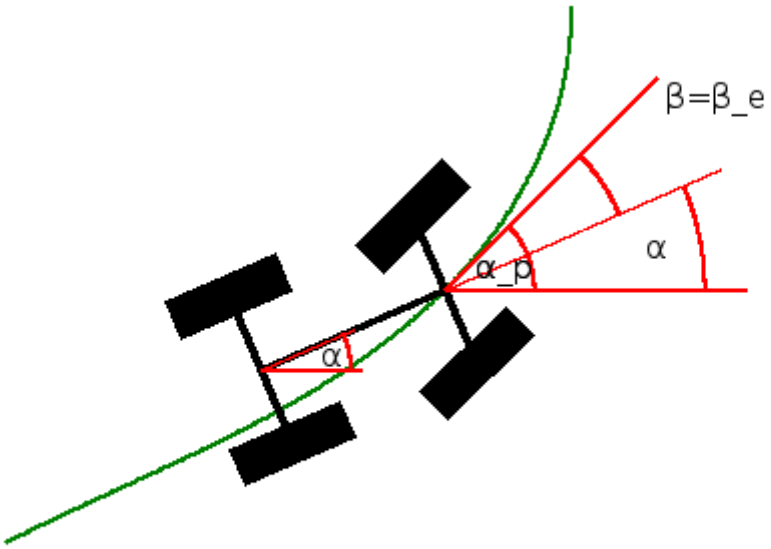
$$\beta = \beta_e + \tan^{-1}(k * e) \tag{5.3}$$

kjer je k ojačenje regulatorja. Izpeljava same zaprtozančne funkcije presega namen tega dela, opisana pa je v članku [5].

Na sliki 20 so prikazani koti vozila, pogrešek in krmilni koti. Vidimo, da prvi člen regulatorja (β_e) poravna krmilni kot s smerjo poti, drugi člen (dodatek do β) pa ga usmeri dlje proti poti. Kolikšen vpliv ima drugi člen, je odvisno od ojačanja k. Na sliki 21 pa je prikazan primer regulatorja za ničelni pogrešek, ko je dejaven le prvi člen.

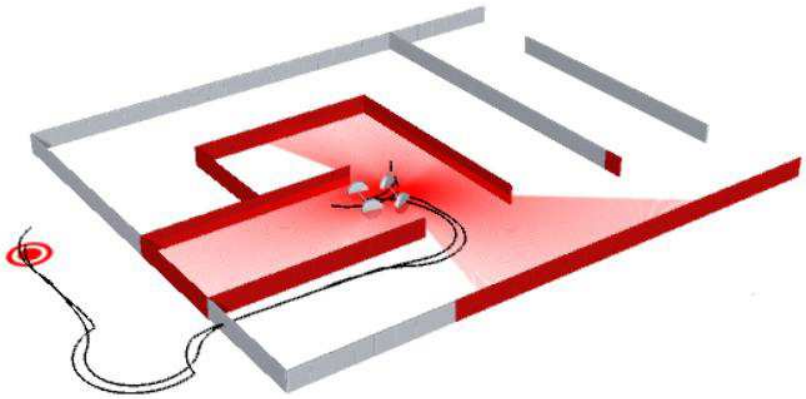


Slika 20: Regulator vozila Stanley



Slika 21: Regulator vozila Stanley v primeru ničelnega pogreška

6 Simulacija



Slika 22: Simulacija

V okviru naloge je bila izdelana simulacija vozila z Ackermannovim pogonom, katere cilj je najti in prevoziti pot od podane začetne lokacije do ciljne točke. V simulaciji je vključen predefiniran prostor z ovirami, ki sestavljajo preprost labirint. Vozilo je izvedeno kot samostojen objekt, ki simulira dinamiko Ackermannovega modela, nanj pa lahko vplivamo z dvema vhodoma: kotom prednjih koles in hitrostjo (pozitivno ali negativno za vzvratno vožnjo). Za bolj zanimivo predstavitev je na vozilu izvedena tudi imitacija laserskega senzorja. V simulaciji tako obstaja tudi interna predstavitev prostora, v katerem so le tiste ovire, ki so kdaj bile v dosegu senzorja. Na ta način vozilo postopoma pridobiva informacijo o svojem okolju in si v zgodnejših fazah svoje vožnje poišče pot, ki jo v resnici seka še neznana ovira. Tako mora vozilo vsakič, ko naleti na taksno oviro zavreči staro pot in znova pognati iskanje.

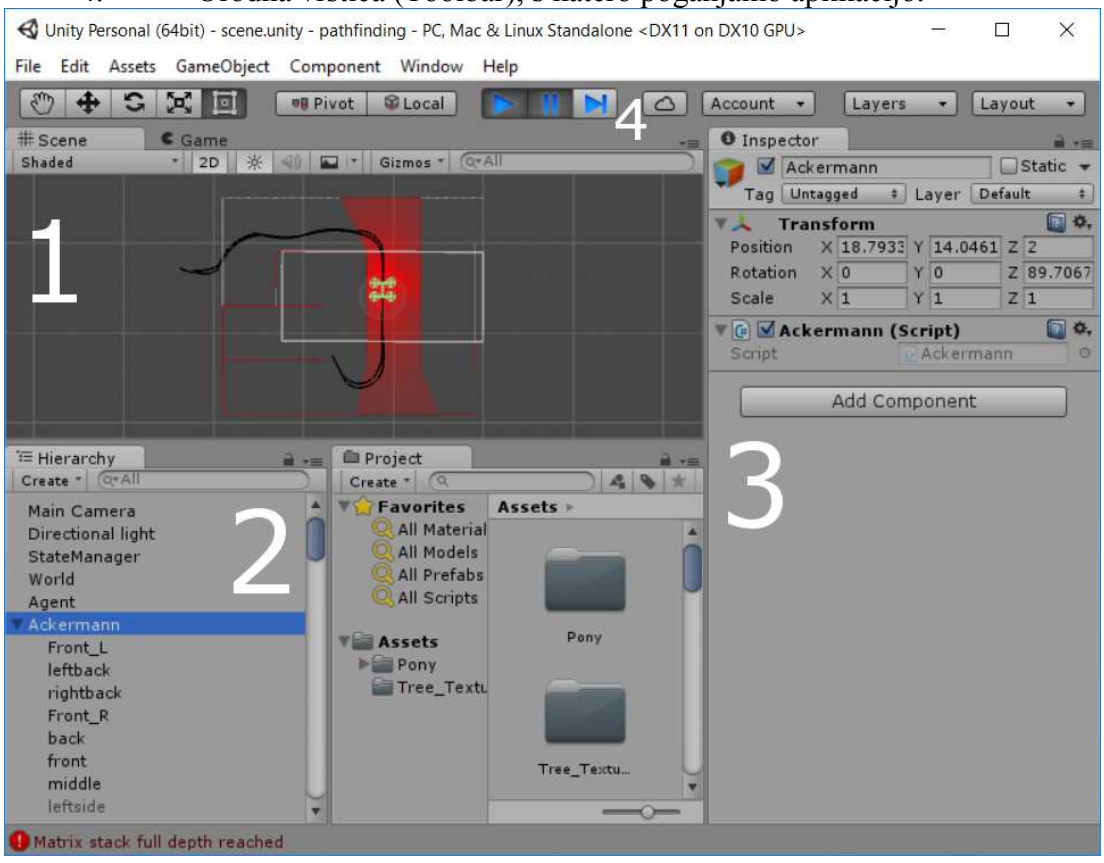
6.1 Uvod v Unity

Unity Engine je grafični pogon, ki se primarno uporablja za razvoj 2D in 3D računalniških iger in aplikacij. Zaradi uporabniku prijazne uporabe in široke funkcionalnosti pa se uporablja tudi na drugih področjih, kot so virtualna realnost, arhitekturna vizualizacija, medicinske simulacije... Podpira preprosto vnašanje 3D modelov, samo okolje pa že vsebuje osnovne geometrijske elemente, kot so kvadri, krogle, valji in podobno, kar dovoljuje hitro prototipiranje ali pa razvoj minimalističnih vizualizacij brez potrebe po drugih programskih paketih. Omogoča

vstavljanje skriptnih datotek v javascript-u ali C#-u, mogoče pa je tudi linkanje C++ knjižnic za najbolj zahtevne aplikacije.

Prvi korak v svet Unity-ja je Unity Editor, ki je urejevalnik, v katerem lahko prek intuitivnega grafičnega vmesnika manipuliramo objekte, ki sestavljajo aplikacijo. Glavni deli urejevalnika so:

1. Pregledovalnik scene (Scene View), ki dovoljuje pogled v 3D svet naše aplikacije,
2. Okno hierarhije (Hierarchy window), v katerem so razvrščeni objekti, ki sestavljajo našo aplikacijo,
3. Okno lastnosti (Inspector window), kjer lahko urejamo lastnosti izbranega objekta,
4. Orodna vrstica (Toolbar), s katero poganjamo aplikacijo.



Slika 23: Unity editor

Osnovni elementi v Unity-u so objekti tipa GameObject, prek katerih lahko gradimo vso ostalo funkcionalnost, ki jo pogon omogoča.

6.1.1 GameObject

GameObject je osnovna enota v Unity-ju. Vsak element v programu, naj bo to 3D model, kamera ali kos kode, mora pripadati objektu tipa GameObject. Ta osnovni

objekt sam po sebi nosi le informacijo o poziciji in orientaciji v prostoru in ne omogoča nobene druge funkcionalnosti. Lahko pa mu dodamo različne komponente ali druge objekte. Ko objekt v hierarhiji lezi pod drugim objektom, sta njegova pozicija in orientacija relativna glede na starševski objekt. To nam omogoča, da celotni objekt sestavimo iz več delov, pri manipulaciji starševskega objekta pa Unity sam poskrbi, da se spremembe pozicije prenesejo tudi na pod-objekte.

Nekaj osnovnih komponent, uporabljenih pri simulaciji je:

• Kamera (Camera): GameObject s to komponento definira pogled koncne aplikacije na 3D sceno.

• Luč (Light): GameObject s to komponento definira lokacijo vira svetlobe.

• Mesh: ta komponenta hrani 3D model objekta, ki bo izrisan na zaslonu. V simulaciji so bili uporabljeni predefinirani objekti tipa kvader in valj, ki so na voljo v Unity-ju.

• Script: ta komponenta je osnova za klicanje kode v Unity Engine-u. Pri simulaciji sem uporabil skripte v jeziku C\#.

6.1.2 Script

Skriptne komponente vsebujejo razrede z dvema osnovnima funkcijama, ki ju za vsak object kliče pogon Unity. Na začetku programa se enkrat pokliče funkcija Setup(), v kateri lahko inicializiramo svojo kodo. Nato pa se v rednih intervalih (vendar ne nujno v enakomernih časovnih zamikih) kliče funkcija Update(), v kateri posodobimo stanje sistema za trenutni časovni korak. V vsakem Update-u lahko tudi dostopamo do časa, ki je pretekel od prejšnjega klica funkcije, tako da lahko zagotovimo fizikalno pravilnost simulacije glede na pretečen čas.

6.2 Hierarhija simulacije

Simulacija je zgrajena iz naslednjih Unity objektov:

• StateManager: objekt, ki skrbi za grafični vmesnik in diktira začetek vožnje,

• World: objekt, ki populira sceno s predefiniranimi ovirami,

• Ackermann: model vozila z Ackermannovim pogonom,

• LaserScanner: laserski senzor,

• Agent: objekt, ki zbira podatke iz laserskega senzorja, po potrebi poganja iskanje poti s hibridnim A* algoritmom in krmili vozilo,.

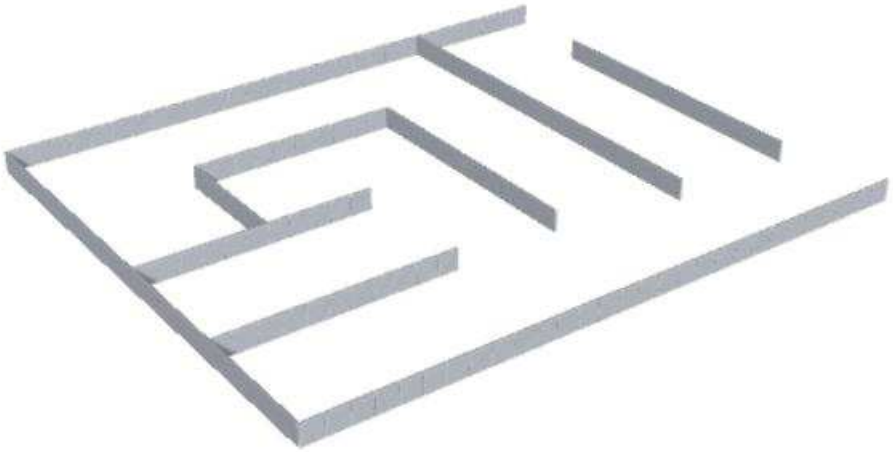
Objekt Agent pa vsebuje se naslednje razrede:

• Mapper

• PathFinder
• Driver

6.3 Objekt World

Ta objekt je dejaven le na začetku simulacije in skrbi za to, da v Unity okolju ustvari ovire, ki sestavljajo prostor. Vsebuje dvodimenzionalno matriko, katere elementi predstavljajo tip ovire na določenem delu prostora. Vsak element nosi informacijo o 2x2 metra velikem območju in vsebuje 4 bite. Prvi bit pomeni 1 meter dolg zid od sredine do desnega robu območja, drugi bit pomeni zid od sredine do vrha, tretji od sredine do levega robu, četrti pa zid od sredine do spodnjega robu območja. Na ta način je možno opisati različne konfiguracije horizontalnih in vertikalnih zidov (vendar je v simulaciji uporabljena fiksna konfiguracija). Objekt World pregleda celotno matriko in instancira kvadre, ki ustrezajo informaciji v elementih.



Slika 24: Ovire

6.4 Objekt Ackermann

Objekt Ackermann skrbi za samo vozilo v simulaciji. Vsebuje preprost grafični model vozila in implementira Ackermannove premike pri danih parametrih vodenja.

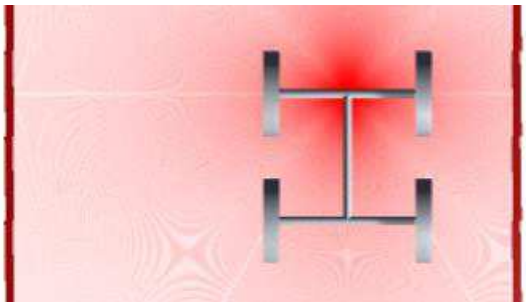
Vsebuje parametre in omejitve vozila:

- wheelBase – medosna razdalja (1.5 metra)
- maxSteering – maksimalni krmilni kot (0.6 rad)

Hrani tudi stanja parametrov vodenja:

- steering – krmilni kot
- speed – hitrost gibanja

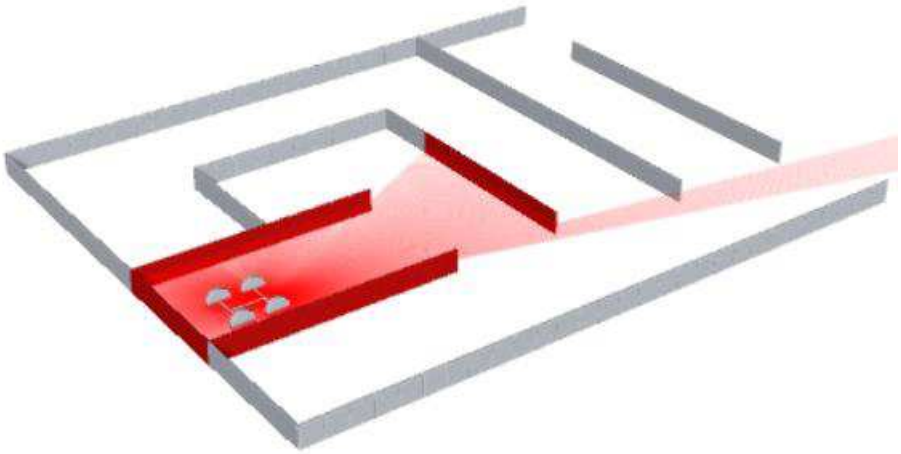
Objekt v vsakem koraku simulacije posodobi svojo pozicijo glede na Ackermanov premik pri danih parametrih vodenja v času, ki je pretekel od prejšnjega koraka. Ponuja funkcijo setControls, ki dovoli nastavljanje parametrov vodenja.



Slika 25: Model vozila

6.5 Razred LaserScanner

LaserScanner simulira laserski senzor na vozilu. Skrbi za to, da prenaša informacijo o okolici vozila v interno rekonstrukcijo prostora. Vozilo namreč nima dostopa do vseh elementov prostora, ki ga zgradi razred World, temveč le do tiste okolice, ki jo med vožnjo razbere z laserjem. Laser v simulaciji obstaja kot element objekta Ackermann, tako da se njegova lokacija avtomatsko posodablja skupaj z vozilom. V vsakem koraku simulacije se simulira 1000 žarkov v vse smeri okrog vozila. To je izvedeno s funkcijo Raycast, ki jo ponuja okolje Unity. Funkciji podamo izvor žarka (lokacijo laserja) in smer. Rezultat nam vrne prvi objekt, ki seka pot žarka. Za vsak zadeti objekt obvestimo razred Mapper s klicem funkcije AddObstacle(). Razred Mapper uporabi to informacijo za posodobitev internega zemljevida prostora in morebitno zahtevo za vnovičen zagon iskanja poti.



Slika 26: Simulacija laserskega senzorja

6.6 Razred Mapper

Razred Mapper imitira rekonstrukcijo ovir iz podatkov laserskega senzorja, nudi dostop do znanega prostora in opravlja zaznavanje trčenja (collision detection). Algoritem ne izvaja prave rekonstrukcije prostora iz laserskih točk, saj to presega namen naloge. Simulator laserskega senzorja ne detektira le točke, temveč Mapper-ju poda celotno oviro, ki jo je zadel, ta pa jo doda na listo ovir, ki bodo uporabljene za iskanje poti. Tak način sicer ne bi bil primeren za realne podatke, vendar pa služi svojemu namenu v simulaciji za proženje iskanja poti, ko vozilo naleti na novo oviro. Za dodajanje ovir razred ponuja metodo AddObstacle, ki prejme oviro tipa GameObject, ki jo je instanciral objekt World in zadel LaserScanner prek funkcije RayCast.

Za potrebe iskanja poti pa ponuja funkcijo collides(…), ki prejme pozicijo Ackermannovega vozila. Funkcija pregleda listo znanih ovir ,za vsako preveri če se vozilo in ovira prekrivata ter vrne informacijo o trčenju. To funkcijo uporablja iskanje poti, da se ne razširi čez ovire.

6.7 Razred PathFinder

PathFinder implementira hibridni algoritem A*. Ponuja funkcijo FindPath, ki prejme ciljno pozicijo. Algoritem sestavlja iskanje hevristike in iskanje gladke poti.

6.7.1 Iskanje hevristike

Hevristika algoritma je optimalna dolžina poti za nek diskretni 1x1 meter velik del prostora. Hevristika se izračuna za celotno okolico ovir na območju, ki ga opisuje kvader, ki za vsaj 5 metrov objema vse ovire, ciljno točko in lokacijo vozila. Vrednost vsake celice hevristike je določena kot najkrajša pot po vertikalnih in horizontalnih premikih prek centrov celic do cilja, pri čemer pot ne seka ovire. Vrednosti dobimo s potovanjem po grafu celic v širino z začetkom v ciljni celici. Računanje hevristike se morda zdi potratno, ker prehodi celoten prostor v okolici ovir, vendar pa je diskretni graf precej manjši od zveznega in prihrani veliko število iskanj v kasnejšem delu algoritma, saj vodi iskanje stran od slepih odsekov k optimalni smeri proti cilju.

6.7.2 Iskanje poti

Za iskanje poti uporabljamo naslednje strukture:

- frontier – prioritetna vrsta vseh obiskanih a še ne odprtih stanj

- opened – zemljevid vseh odprtih celic diskretiziranega prostora

Potek algoritma:

1. Odpre prvo stanje:
   - Pozicija in orientacija: trenutna pozicija in orientacija vozila
   - Cena: 0
   - Hevristika: vrednost celice zemljevida hevristike iz prvega dela algoritma v kateri leži trenutna pozicija

   Stanje doda na prioritetno vrsto frontier s prioriteto enako vsoti cene in hevristike.
2. Iz prioritetne vrste jemlje stanja z najnižjo prioriteto. Zavrača stanja dokler eno ne zadosti naslednjim pogojem:
   - Diskretna celica zemljevida opened, ki pripada stanju je še neoznačena.
   - Stanje se ne prekriva z oviro.

   Na tej točki se algoritem konča, če je stanje v bližini cilja (manj kot 1 meter oddaljen), ali pa če je prioritetna vrsta prazna (neuspešen zaključek).
3. Iz stanja izvede 32 Ackermannovih premikov in jih doda na prioritetno vrsto. Za vsak krmilni kot od minimalnega do maksimalnega kota krmiljenja s korakom ene osmine maksimalnega krmilnega kota in za vožnjo naprej ter vzvratno doda na prioritetno vrsto novo stanje:
   - Pozicija in orientacija: je enaka premiku iz trenutnega stanja za 1 meter z danim krmilnim kotom in smerjo gibanja.
   - Cena: enaka vsoti cene trenutne celice in cene premika. Cena premika je enaka dolžini premika (1 meter), ki ji dodamo ceno spremembe smeri (5 metrov), ceno vzvratne vožnje (dodatni meter) in ceno spremembe krmilnega kota (razlika kotov * 1 meter).
   - Hevristika: vrednost celice zemljevida hevristike iz prvega dela algoritma v kateri leži trenutna pozicija

   Stanje doda na prioritetno vrsto frontier s prioriteto enako vsoti cene in hevristike. Algoritem se vrne na korak 2 in ponavlja odpiranje stanj.

6.7.3 Rekonstrukcija poti sprednje in zadnje osi

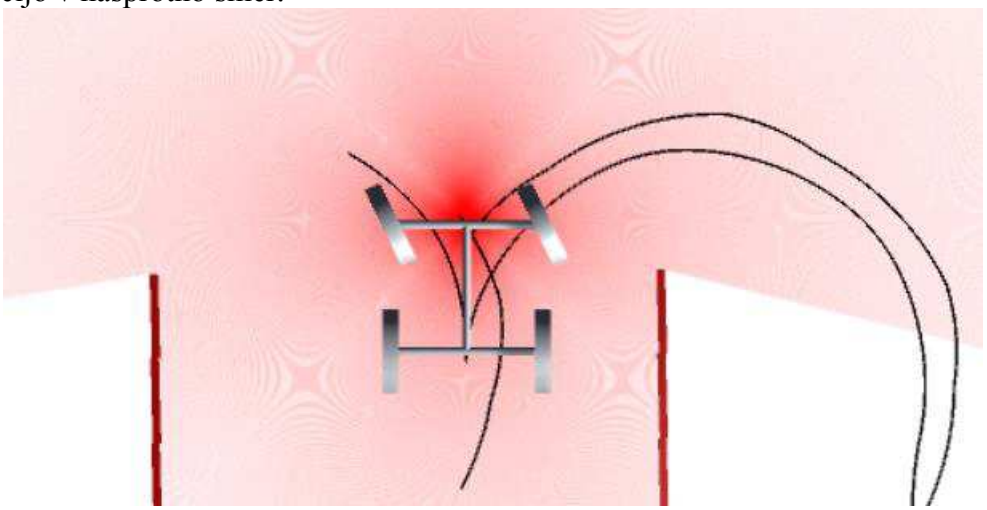
Če je algoritem uspešno našel ciljno stanje, potem sledi predhodnim stanjem do začetnega in ustvari končno pot zadnje osi. Pot sestavljajo odseki z enako smerjo (vsaka sprememba iz vožnje naprej v vzvratno naprimer povzroči nov odsek – to je pomembno za kasnejše vodenje vozila), vsak odsek pa sestavlja zaporedje pozicij, ki so vzorčene iz zaporednih Ackermannovih premikov.

Potek rekonstrukcije zadnje osi:

1. Poišče predhodnika trenutnega stanja (v prvem koraku je to ciljno stanje) in določi Ackermannov premik med stanjema.
2. Vzorči 100 točk na premiku od trenutnega do predhodnega stanja in jih dodaja na začetek liste odseka. To da zaporedje pozicij, ki so med seboj oddaljene 1 centimeter.
3. Če je predhodnik začetno stanje zaključi rekonstrukcijo. Če je premik v predhodno stanje različen od premika v trenutno stanje, se odsek zaključi in na listo odsekov doda nov prazen odsek. Predhodno stanje postane trenutno stanje in rekonstrukcija se nadaljuje v koraku 1.

Naslednji korak je rekonstrukcija prednje osi. Rekonstrukcija poteka za vsak odsek posebej in se razlikuje za vzvratno vožnjo in vožnjo naprej. Za vožnjo naprej potrebujemo dejansko prednjo os, za vzvratno vožnjo pa regulator vodenja potrebuje prednjo os zrcaljeno prek zadnje osi.

Rekonstrukcija prednje osi za vsak odsek vožnje translira vsako pozicijo v smer tangente na pot v tej poziciji. Smer tangente dobimo s pomočjo dveh pozicij, ene pred in druge za trenutno pozicijo (pri robnih pozicijah, ki nimajo predhodnika ali naslednika uporabimo trenutno pozicijo). Trenutno pozicijo transliramo v smeri obeh okoliških pozicij za medosno razdaljo (1.5 metra). Na vzvratnem odseku transliramo pozicijo v nasprotno smer.



Slika 27:Referenčna pot prednje in zadnje osi

6.8 Razred Driver

Razred driver implementira regulator sledenja referenčni poti prednje osi. V vsakem koraku simulacije pridobi novo pozicijo vozila in jo primerja z referenčno potjo. Določi odsek poti (par zaporednih točk), ki leži najbližje poziciji. Iz dobljenega odseka določi kot tangente (ki je kar kot odseka) in oddaljenost od pozicije. Iz razlike kota tangente in rotacije vozila ter oddaljenosti izračuna krmilni kot vozila na način opisan v poglavju 5 Vodenje vozila po gladki poti na strani 43. Krmilni kot nastavi razredu Ackermann, ki ga v naslednjem koraku simulacije uporabi za premik vozila. Ko zaključi celotno pot in doseže cilj, ustavi vožnjo.

6.9 Razred Agent

Razred Agent skrbi za proženje iskanja poti in vožnje. Deluje v režimu naslednjih stanj:

- Pathfinding
  V tem stanju sproži razred Pathfinder in čaka na rezultat hibridnega A*. Če je bilo iskanje uspešno se premakne v stanje Driving.
- Driving
  V tem stanju sproži razred Driver. Sproti spremlja stanje razreda Mapper, od katerega izve ali trenutna pot seka na novo detektirano oviro. V tem primeru ustavi vožnjo in se vrne v stanje Pathfinding. Če je vožnja zaključena (cilj dosežen) se premakne v stanje Idle.
- Idle
  V tem stanju izklopi ustavi vožnjo vozila in postane neaktiven. To je tudi začetno stanje razreda.
- Paused
  V tem stanju izklopi ustavi vožnjo vozila in postane neaktiven.
- Manual
  V tem stanju izklopi ustavi vožnjo vozila in omogoči razredu StateManager, da nastavlja parametre vodenja.

6.10 Razred StateManager

Razred StateManager skrbi za preprost uporabniški vmesnik s tremi gumbi:

- Start
  Če je Agent v stanju Paused ali Manual, ga prestavi v stanje Driving.

Če je Agent v stanju Idle, ga prestavi v stanje Pathfinding.

- Stop
  Če je razred Agent v stanju Driving, ga prestavi v stanje Paused.
- Manual
  Če je razred Agent v stanju Driving ali Paused, ga prestavi v stanje Manual in omogoči vodenje vozila prek tipkovnice.

Literatura

[1] D. Dolgov, S. Thrun, M. Montemerlo, J. Diebel, Practical Search Techniques in Path Planning for Autonomous Driving, Dosegljivo: http://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf

[2] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, A. Petrovskaya, M. Pflueger, G. Stanek, D. Stavens, A. Vogt, S. Thrun, Junior: The Stanford Entry in the Urban Challenge, Dosegljivo: http://robots.stanford.edu/papers/junior08.pdf

[3] G. Klančar, Avtonomni mobilni sistemi, Dosegljivo: http://msc.fe.uni-lj.si/Download//klancar/AMS_knjiga2014ver1.pdf

[4] J. M. Snider, Automatic Steering Methods for Autonomous Automobile Path Tracking, Dosegljivo: https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf

[5] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, P. Mahoney, Stanley: The Robot that Won the DARPA Grand Challenge, Dosegljivo: http://isl.ecst.csuchico.edu/DOCS/darpa2005/DARPA%202005%20Stanley.pdf