УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
Трг Доситеја Обрадовића 6, 21000 Нови Сад, Република Србија
Деканат: 021 6350-413; 021 450-810; Централа: 021 485 2000
Рачуноводство: 021 458-220; Студентска служба: 021 6350-763
Телефакс: 021 458-133; e-mail: ftndean@uns.ac.rs
ИНТЕГРИСАНИ СИСТЕМ МЕНАЏМЕНТА СЕРТИФИКОВАН ОД:

MEHATRONIKA

NAZIV PROJEKTA:

Autonomni robot

TEKST ZADATKA:

Projektni zadatak se sastoji iz više celina:

- kreiranje okruženja, odnosno sveta u kome se nalazi objekat, prepreke i robot. Odrađeno u okviru softverskog okruženja *Webots*. Modifikacija robota prema potrebi
- kreiranje kontrolera za kameru. Uslikati okruženje u kome se nalazi robot i proslediti datu sliku na obradu. Za obradu slike koristi se *OpenCV*, dok se sve poruke šalju pomoću *ROS*-a (eng. *Robot Operating System*). Kreira se matrica u kojoj su označene prepreke, robot, objekat, ciljevi
- uz pomoć obrađenih slika kreiramo matricu potencijala i nalazimo optimalnu putanju od robota do objekta, kao i od objekta do cilja 1 ili 2, u zavisnosti od boje putokaza
- napisati kontroler za robota koji sadrži neophodne funkcije za njegovo kratanje

Zadatak se svodi na to da:

- robot krene sa startne pozicije
- dođe do kocke i odgurne je, kako bi video boju putokaza koji određuje da li ide na cilj 1 ili 2
- najkraćim putem ide do cilja 1 iili 2

MENTOR PROJEKTA:

prof. dr Mirko Raković
prof. dr Kalman Babković
MSc Lazar Milić
BSc Miroslav Bogdanović

PROJEKAT IZRADILI:

Vanja Lazarević EE149/2019
Đorđe Drozgović EE43/2018

Sadržaj:

1 Uvod

Projektni zadatak se sastoji iz više celina:

- kreiranje okruženja, odnosno sveta u kome se nalazi objekat, prepreke i robot. Odrađeno u okviru softverskog okruženja *Webots*. Modifikacija robota prema potrebi
- kreiranje kontrolera za kameru. Uslikati okruženje u kome se nalazi robot i proslediti datu sliku na obradu. Za obradu slike koristi se *OpenCV*, dok se sve poruke šalju pomoću *ROS*-a (eng. *Robot Operating System*). Kreira se matrica u kojoj su označene prepreke, robot, objekat, ciljevi
- uz pomoć obrađenih slika kreiramo matricu i nalazimo optimalnu putanju od robota do objekta, kao i od objekta do cilja 1 ili 2, u zavisnosti od boje putokaza
- napisati kontroler za robota koji sadrži neophodne funkcije za njegovo kratanje

Zadatak se svodi na to da:

- robot krene sa startne pozicije
- dođe do kocke i odgurne je, kako bi video boju putokaza koji određuje da li ide na cilj 1 ili 2
- najkraćim putem ide do cilja 1 iili 2

Startna pozicija je zelene boje i dimenzije su 20x20cm. Cilj 1 i cilj 2 su istih dimenzija, ali boje im se razlikuju. Cilj 1 je crvena boje, a cilj 2 je ljubičaste boje. Putokaz ka ciljevima stranice dužine 15cm može biti ili crvene ili ljubičaste boje. Dati putokaz je postavljen ispod kocke, kako bi tokom guranja objekta, mogla da se otkrije boja. Prostor po kome se kreće robot veličine je 2x3m sa ogradom visine od 10cm. Ova celokupna struktura u daljem tekstu nazvaće se stolom. Na datom stolu se, pored pomenutih ciljeva, nalaze prepreke žute boje čija je visina 10cm i proizvoljne dužine kao i kocka koja treba da se odgurne kako bi otkrili putokaz. Kocka je u datom zadatku svetlo plave boje (#20D0E0).

Robot koji se koristi u datom projektu jeste *TurtleBot3 Burger*. Na njega je potrebno postaviti plavu ploču kvadratnog oblika, proizvoljnih dimenzija. U konkretnom projektu dimenzija te ploče je 14x14cm.

Svi programi i kontroleri su napisani u programskom jeziku P*ython3*.

Dalja izrada projekta biće objašnjena u nastavku.

2 Kreiranje okruženja u Webots-u

RGB je model boja koji predstavlja boje kao kombinaciju tri osnovne boje (crvena, zelena i plava). Obično se svaka primarna boja predstavljena vrednošću između 0 (minimalni intenzitet) i 255 (maksimalni intenzitet). Postoji i način gde se predstavlja sa vrednostima od 0 do 1, što je naš slučaj. U daljem tekstu je prikazan ovaj način umesto prethodno spomenutog, gde je opseg od 0 do 255. Ovim je omogućena veća preciznost u predstavljanju boja i manipulaciji.

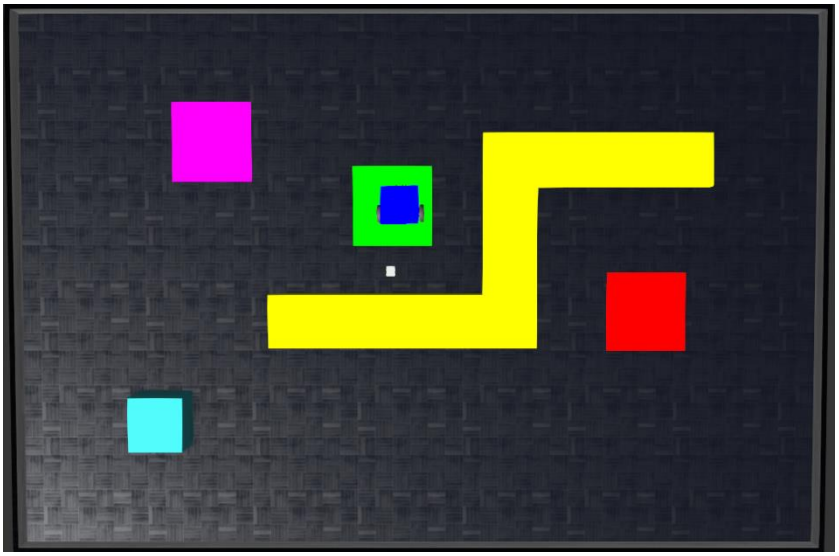
Na početku potrebno je kreirati „svet“ u *Webots*-u. Robot se kreće po prethodno spomenutom stolu. Na tom istom stolu su postavljene sve neophodne komponente. Ovaj sto je nađen pod stavkom *RectangleArena (Solid)* i dodata je ograda visine 10cm. Zbog lakšeg uočavanja elemenata postavljena je podloga tamne boje (RGB(0.0, 0.0, 0.0)).

Prepreke na podlozi su kreirane kao *Solid* objekti pravougaonog oblika, širine 20cm i visine 10cm, dok je dužina proizvoljna. Položaj prepreka je takođe proizvoljan. Koji god da je raspored izabran, robot je razvijen tako da može da obiđe date prepreke.

Kamera se postavlja iznad terena na visina na kojoj može da obuhvati ceo sto. Empirijskim postupkom dolazi se do konkretne vrednosti visine kamere koja iznosi 1.8m iznad terena.

Zadati objekat je kocka čija je dužina stranice 20cm i mase 250g. Zadati heksadecimalni kod boje objekta je #20D0E0. Vrednost crvene iznosi 32, zelene 208 i plave 224. Kao što je prethodno spomenuto, date vrednosti moramo da prebacimo u opseg između 0 i 1. Konverzijom dobijamo da je R=0.125 , G= 0.816 i B= 0.878 i date vrednosti ubacujemo u „naš svet“.

Na slici 1 data je slika „sveta“ sa postavljenom startnom pozicijom, preprekama, ciljevima 1 i 2, putokazom do cilja i objektom.



Slika 1. Okruženje u Webots-u

3 TurtleBot3 Burger

Webots sadrži velik broj modifikovanih modela robota i objekata. Pored toga moguće je izraditi nove modele počevši od nule. Dizajner modela robota navodi grafička i fizička svojstva. Grafička svojstva uključuju oblik, dimenzije, položaj i orijenzaciju, boje i teksturu objekata. Fizička svojstva uključuju masu, faktor trenja, konstantne opruge i amortizere.

Roboti u datom okruženju mogu da sadrže skup senzora i aktuatora koji se često koriste u robotskim ekperimentima. Primer nekih od senzora jeste lidar senzor, kamera, akcelometar, kompas, senzor distance, itd. Aktuatori koji mogu da se dodaju jesu kočnice, konektor, displej, propeler, zvučnik...

Robot koji se koristi u našem projektu jeste *TurtleBot3 Burge* kompanije *Robotis*. Radi se o novoj generaciji mobilnog robota. Ima kompaktan i modularan dizajn. Moguće su modifikacije datog robota po potrebama korisnika. U spomenutom robotu je uključen prethodno spomenut lidar senzor. Dati senzor se koristi za navigaciju. Aktuatori na ovom robotu su točkovi koji su neophodni za kretanje robota, žiroskop i akcelerometar po sve tri ose kao i troosni magnetometar. Na slici 2 je prikazan fizički izgled upotrebljenog robota.

Slika 2. TurtleBot3 Burger

3.1 Kontroler robota

Unutar fajla *my_controler* napisan je program za simulaciju kretanja robota u *Webots* simulacionom okruženju.

Na početku ubacujemo nekoliko *Python* modula i klase iz *Webots* kontrolera:

- Robot i Motor su klase koje se koriste za kontrolu robota u *Webots* simulaciji
- *threading* koji se koristi za stvaranje paralelnih niti kako bi se omogućilo istovremeno izvršavanje različitih delova koda
- *time* za kontrolu vremena i pravljenje pauza tokom izvršavanja neke akcije
- *math* za matematičke operacije
- *std_msgs.msg* koristi se za ROS poruke
- *rospy* je *Python* biblioteka koja omogućava komunikaciju sa ROS-om

Nakon toga, definišemo konstantu *TIME_STEP* koja predstavlja vremenski korak simulacije. Ovo predstavlja osnovnu vremensku jedinicu u *Webots*-u i koristi se za kontrolu brzine simulacije.

Definišemo klasu *DifferentialDriveRobot* koja sadrži logiku za upravljanje diferencijalnim pogonom robota. Data klasa ima sledeće funkcionalnosti:

- inicijalizuje parametre robota kao što su maksimalna brzina, radijus točkova, ubrzanje, poziciju i brzinu točkova
- implementira funkcije za pomeranje robota napred, nazad, levo, desno. Ove funkcije koriste akceleraciju kako bi postigle željenu brzinu i put

Kreiramo instancu ove klase koja se zove *myRobot*.

Klasa *WebotsThread* se koristi za kreiranje niti koja će izvršavati *Webots* simulaciju.
U okviru ove niti, robot prati komande i izvršava odgovarajuće akcije (kretanje napred, nazad, levo, desno) na osnovu komandi koje dobija preko ROS-a. Kreiramo instancu ove klase koja se zove *webots_thread* i pokreće se u pozadini.

Definiše se *sub_callback* funkcija koja se poziva kada stigne ROS poruka na temu *turtlebot_motion*. Ova funkcija čita komande iz poruke i dodaje ih u listu komandi robota.

Inicijalizuje se čvor i pretplaćuje se na temu *turtlebot_motion* sa funkcijom *sub_callback*. Program ostaje u petji *rospy.spin* kako bi se čekale nove ROS poruke i omogućilo interaktivno upravljanje robotom preko komandi.

```
def sub_callback(data):
    command_str = data.data
    action = command_str[0]
    print("Action:", action)
    webots_thread.robot.commands.append(data.data)

rospy.init_node('motion_driver')
rospy.Subscriber('turtlebot_motion', msg.String, sub_callback)
rospy.spin()
```

Slika 3. Funkcija sub_callback

4 Program za obradu slike

U fajlu node.py se definiše čvor camera_filter_node koji se pretplaćuje na temu webots_camera. Njegov zadatak je preuzimanje slike sa kamere. Preuzeta slika se kasnije obrađuje.

Na slici 4 prikazane su potrebne biblioteke za obradu slike u ROS okruženju, kao i način njihovog importovanja u program. *OpenCV* biblioteka se u program unosi kao *cv2*. Pored navedene biblioteke unosi se i biblioteka *cv_bridge* kao i *cvb*. Radi se o biblioteci koja omogućava da se ROS format slike prilagodi formatu slike koji odgovara funkcijama iz *OpenCV* formata. Biblioteka *numpy* služi za rad sa višedimenzionalnim nizovima i matricma. Imamo biblioteku *math* za matematičke operacije i *matplotlib.pyplot* koja se koristi za vizuelizaciju podataka i grafičko crtanje (histogrami, dijagrami rasejanja, trakasti grafikoni..)

```
from cv2 import waitKey
import rospy
from sensor_msgs.msg import Image
import cv2
import cv_bridge as cvb
from cv_bridge import CvBridgeError
import numpy as np #za matrice itd
from enum import Enum
import time
import std_msgs.msg as msg
import math
```

Slika 4. Potrebne biblioteke

Definisali smo klasu *TableFieldType* tipa *Enum* koja je data na slici ispod. Svaki objekat, prepreka od značaja koja se generiše u matricu označena je posebnim brojem, kako bi lakše ispratili šta se ispisuje.

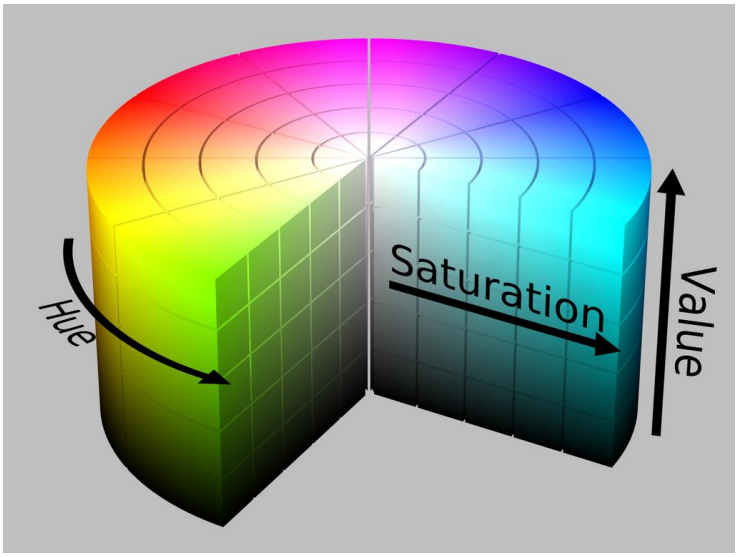
```
class TableFieldType(Enum):
    FIELD_EMPTY = 0
    FIELD_OBSTACLE = 1
    FIELD_ENDING1 = 2
    FIELD_ENDING2 = 3
    FIELD_ROBOT = 4
    FIELD_OBJECT = 5
```

Slika 5. Funkcija TableFieldType

Sledeće što definišemo jeste klasa *CameraFilter* i čija je slika data u nastavku. Prvo definišemo objekat iz klase *cvBridge*. Cilj njihovog definisanja je prilagođavanje slike iz ROS-a tako da može da radi sa funkcijama iz *OpenCv* biblioteke. *Cut_image* metoda se koristi za sečenje slike. Uzima argument *data*, a to predstavlja sliku u određenom formatu. Data metoda pokušava da konvertuje datu sliku u *OpenCV* format koristeći *cvb.imgmsg_to_cv2*. Koristimo samo deo slike, konkretno regiju od 50. reda do 670. reda kao i od 80. kolone do 1000 kolone.

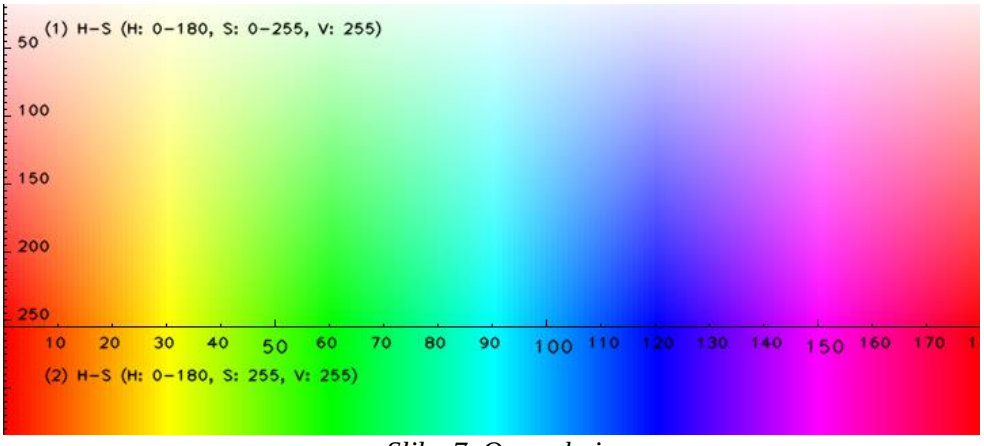
Prikazuje se slika, sačeka naša reakcija i na kraju vraća našu sliku. Nakon toga imamo funkciju *populate_matrix* koja služi za popunjavanje matrice određenim vrednostima (prikazane na slici 5) na osnovu filterisane slike.

Za tačan prikaz postavljenih komponenti u „našem svetu“, potrebno je odrediti njihov položaj na stolu. Mora da se posmatra i pozicija robota, objekata (u našem slučaju kocke), ciljeva kao i putokaza ka datim ciljevima. Navedeni objekti su predstavljeni različitim bojama i zbog toga neće biti problem identifikovati ih na stolu. Za izdvajanje objekata iz lavirinta na osnovu njihove boje neophodno je kreirati odgovarajuće bit maske za svaku boju koja je prisutna u lavirintu. Slike u *OpenCV* biblioteci predstavljaju se u *RGB* (*eng. Red Green Blue*) formatu. Ovaj format je previše kompleksan za kreiranje bit maski i iz tog razloga potrebno je izvršiti konverziju u *HSV* (*eng. Hue Saturation Value*) format. Prethodno spomenuta funkcija *cv2.cvtColor* izvršava datu konverziju.



Slika 6. HSV model boja

Parametar H govori o kojoj boji se radi. Drugi parametar definiše sadržaj bele svetlosti, odnosno koliko je boja svetla. Boja je čistija ukoliko ima manji sadržaj bele boje u sebi. Zadnji parametar, treći, predstavlja meru koliko je boja svetla ili tamna. *HSV* format boja prikazan je na slici 6. Biblioteka *OpenCV* omogućava prepoznavanje boja na osnovu vrednosti parametara *Hue*. Za svaku boju se definiše opseg vrednosti ovog parametara što je prikazano na slici 7.



Slika 7. Opseg boja

U skladu sa prikazanim opsezima, moguće je definisati gornju i donju granicu određene boje. O tome će biti kasnije reči.

Na slici 8 nalazi se klasa *CameraNode* unutar koje se definiše *subscriber* čvor. Pomoću naredbe *rospy.init_node(„camera_filter_node“)* definišemo dati čvor. Naredba *rospy.Subscriber* definiše temu na koju se čvor pretplaćuje. Prvi parametar predstavlja ime teme, drugi tip poruke koji čvor treba da preuzme i na kraju, navodi se *callback* funkcija. Zadnji parametar se poziva svaki put kada *subscriber* preuzme podatke sa date teme.

```
class CameraNode:
    matrix = np.zeros((620, 920))
    def __init__(self):

        rospy.init_node("camera_filter")
        rospy.loginfo("Starting camera_filter node.")

        self.camera_sub = rospy.Subscriber("webots_camera", Image, callback = self.callback_camera) #kad dobije sliku, nzm odakle, onda pozove callback_camera

        print('slikanje')

        self.cf_class = CameraFilter()
```

Slika 8. Klasa CameraNode

U našem slučaju, tema na koju se čvor pretplaćuje, je *webots_camera*, a tip poruke koju preuzi je *Image*. *Image* predstavlja sliku koju je potrebno obraditi. *Callback* funkcija poziva metodu *callback_camera* koja će biti definisana u nastavku koda.

Downscale_matrix je funkcija koja nam služi za smanjivanje dimenzije matrice sa slikom, tako da se iz svakog većeg bloka od 20x20 piksela uzima samo jedan piksel i postavja u manju matricu. Funkcija *printMatrix* služi za ispisivanje elemenata matrice u formatu prikladnom za vizuelno prikazivanje. Date funkcije zajedno sa primerom kako izgleda data matrica date su na slici 9.

```
def downscale_matrix(self, matrix):
    downscaledMatrix = np.zeros((31, 46))
    for i in range(31):
        for j in range(46):
            if matrix[i * 20][j * 20] == 1:
                for dx in range(-1, 2):
                    for dy in range(-1, 2):
                        new_i = i + dx
                        new_j = j + dy

                        # Proveri da li su novi indeksi u granicama matrice
                        if 0 <= new_i < 31 and 0 <= new_j < 46:
                            downscaledMatrix[new_i][new_j] = 1
            else:
                downscaledMatrix[i][j] = matrix[i * 20][j * 20]
    return downscaledMatrix
```

Slika 9. Funkcije downscale_matrix i printMatrix

Zadnja klasa u ovom fajlu zove se *callback_camera*. Prvo stvaramo objekat *cf_class* koji je baziran na klasi *CameraFilter*. Za sečenje slike ispisali smo liniju *self.cf_class.cut_image(image)* čije metode se nalaze u CameraFilter.

Slika 10. Prikaz dobijene matrice

U prethodnom delu smo spominjali gornju i donju granicu određene boje. Funkcija *filter_image* iz *cf_class* filtrira sliku *cutImage* tako da se identifikuje oblast sa određenom bojom. Unutar te funkcije smo definisali npr da identifikuje zelenu boju. Nakon toga uz pomoć funkcije *populate_matrix* popunjavamo matricu određenim bojama, što je prikazano na slici gore (ubaci broj umesto gore). Nakon popunjavanja, smanjimo dimenziju matrice, ispišemo, sačuvamo i resetujemo na nulu.

```
def callback_camera(self, image):
    cutImage = self.cf_class.cut_image(image)
     #prepreka(zuta)
    filteredImage = self.cf_class.filter_image(cutImage, np.array([22, 93, 0]), np.array([45, 255, 255]), 'yellow')
    self.matrix = self.cf_class.populate_matrix(filteredImage, self.matrix, TableFieldType.FIELD_OBSTACLE.value)
    #objekat(svetlo plava)
    filteredImage = self.cf_class.filter_image(cutImage, np.array([87,85,185]), np.array([180, 255, 255]), 'lightblue')
    self.matrix = self.cf_class.populate_matrix(filteredImage, self.matrix, TableFieldType.FIELD_OBJECT.value)
    #cilj1(crvena)
    filteredImage = self.cf_class.filter_image(cutImage, np.array([0, 50, 50]), np.array([12, 255, 255]), 'red')
    self.matrix = self.cf_class.populate_matrix(filteredImage, self.matrix, TableFieldType.FIELD_ENDING1.value)
    #cilj2(ljubicasta)
    filteredImage = self.cf_class.filter_image(cutImage, np.array([125, 50, 50]), np.array([150, 255, 255]), 'purple')
    self.matrix = self.cf_class.populate_matrix(filteredImage, self.matrix, TableFieldType.FIELD_ENDING2.value)
    #robot(plava)
    filteredImage = self.cf_class.filter_image(cutImage, np.array([118, 50, 50]), np.array([125, 255, 255]), 'blue')
    self.matrix = self.cf_class.populate_matrix(filteredImage, self.matrix, TableFieldType.FIELD_ROBOT.value)
    downscaled_matrix = self.downscale_matrix(self.matrix)
    self.printMatrix(downscaled_matrix)
    print(self.matrix.shape)
    #print('usao u callback')
    print(cutImage.shape)
    np.savetxt('/home/vanja/MEHATRONIKA/project_ws/src/camera_filter/src/camera_filter/nodes/matricaa.txt', downscaled_matrix, fmt = '%.2d') #int od 2 broja
    self.matrix = np.zeros((620, 920))
```

Slika 11. Funkcija callback_camera

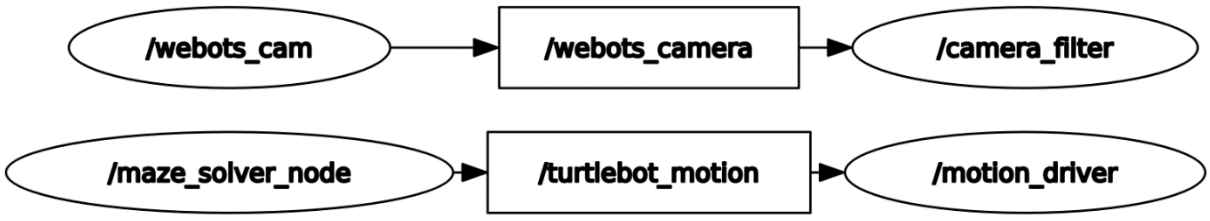
5 ROS

ROS, ili Robot Operating System se koristi za razmenu podataka između čvorova (*eng node*). U svakoj skripti je moguće napraviti zaseban čvor koji može biti ili objavljivač podataka (*eng publisher*) ili pretplatnik (*eng subscriber*). Svi podaci se šalju ili preuzimaju sa teme (*eng topic*). Jedan čvor može biti ili pretplatnik ili objavljivač na jednu temu. Moguće je da jedan čvor postavlja poruke na više tema, ili da preuzima poruke sa više tema, ali ne sme u isto vreme da vrši obe funkcije na jednu temu. *ROS Master* čini jezgro celog sistema i omogućava komunikaciju između čvorova.

Konkretno, u našem primeru projekta, pomoću ROS-a definišu se dve teme sa svojim objavljivačima i pretplatnicima. Prva tema se koristi za obradu slike, dok se druga koristi za kretanje robota po terenu.

Prva tema naziva se *webots_camera*. Čvor koji objavljuje sliku na tu temu jeste *webots_cam*, dok čvor koji se pretplaćuje na rečenu temu jeste *camera_filter*. Dati čvor preuzima sliku sa teme *webots_camera* i obrađuje je.

Druga tema, kao što je već rečeno, definiše kretanje robota. Stoga je naziv teme *turtlebot_motion*, gde je naveden naziv korišćenog robota. Objavljivač na navedenu temu je čvor *maze_solver_node,* koji ima zadatak da na temu objavi sekvencu kretanja koju robot treba da izvrši u odgovarajućem trenutku. Čvor *motion_driver* je pretplaćen na temu koja opisuje kretanje robota. Uloga ovog čvora je pokretanje samog robota onda kada je podatak preuzet sa teme.



Slika 12. Graf

6 Dijkstra algoritam

Graf težine je graf čije ivice imaju „težinu“ ili „cenu“. Težina ivice može predstavljati rastojanje, vreme ili bilo šta što modelira „vezu“ između para čvorova koje povezuje.

Uz pomoć Dijkstra algoritma možemo pronaći najkraći put između čvorova na grafu. Konkretno, možemo pronaći najkraći put od izvornog čvora do ostalih čvorova u grafu. Na ovaj način se stvara stablo najkraćeg puta.

Ovaj algoritam se koristi u GPS (*eng. Global Positioning System*) uređajima za pronalaženje najkraćeg puta između trenutne lokacije i odredišta. Ima široku primenu u industriji, posebno u domenima koji zahtevaju mreže za modeliranje.

Dijkstra algoritam počinje od čvora koji se izabere kao izvorni i analiza graf kako bi pronašao najkraći put između tog čvora i svih ostalih čvorova na grafu. Algoritam prati trenutno poznatu najkraću udaljenost od svakog čvora do izvornog čvora i ažurira ove vrednosti ako pronađe najkraću putanju. Kada algoritam pronađe najkraću putanju između izvornog čvora i drugog čvora, taj čvor se označava kao „posećen“ i dodaje se putanji. Proces se nastavlja sve dok se svi čvorovi na grafu ne dodaju putanji. Na ovaj način, imamo putanju koja povezuje izvorni čvor sa svim ostalim čvorovima, prateći najkraći mogući put do svakog čvora.

Ovaj algoritam može da radi samo sa grafovima koji imaju pozitivne težine. Pod težinom se smatra sve ono što smo naveli na početku poglavlja. Radi samo sa pozitivnim težinama, zato što se tokom procesa moraju dodati težine ivica, da bi se pronašao najkraći put. Kada je čvor označen kao „posećen“, trenutna putanja do tog čvora je označena kao najkraća putanja do tog čvora. Negativne težine mogu promeniti ovo ako se ukupna težina smanji, nakon što se ovaj korak dogodio.

Dati algoritam se nalazi u fajlu *path_finder.py*. Kod koji je vezan za Dijkstra algoritam dat je na slici ispod.

```python
def dijkstra(maze, start, end):
    rows, cols = maze.shape
    distances = {(i, j): float('inf') for i in range(rows) for j in range(cols)}
    distances[start] = 0

    pq = [(0, start)]  # Priority queue of (distance, position) tuples
    parents = {}

    while pq:
        dist, current = heapq.heappop(pq)

        if current == end:
            path = []
            while current != start:
                path.append(current)
                current = parents[current]
            path.append(start)
            return path[::-1]

        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and maze[neighbor] != 1:
                new_dist = dist + 1
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    parents[neighbor] = current
                    heapq.heappush(pq, (new_dist, neighbor))

    return None  # No path found
```

Slika 13. Funkcija dijkstra

Moramo napomenuti da se ovaj algortiam koristi za rešavanje grafova, a mi u ovom slučaju radimo sa matricama, te je zbog toga bilo neophodno prvo naš "lavirint" koji je predstavljen matricom posmatrati kao jedan graf čije tačke naš robot obilazi.

Ova metoda možda nije najidealnija za rešavanje ovakvih vrsta problema, ali se pokazala kao jedna od najlakših za realizaciju.

Path_finder.py

Path_finder sadrži više celina:

- importovanje biblioteka i modula
- inicijalizacija promenljivih
- funkcija za pronalaženje pozicija
- Dijkstra algoritam
- učitavanje i čuvanje rute
- izračunavanje pokreta robota i čuvanje pokreta
- upoređivanje boje putokaza i cilja
- glavna funkcija main
- pokretanje glavne funkcije

Dati *Python* fajl koristi prethodno spomenuti Dijkstra algoritam za navigaciju robota kroz sto, sa preprekama kako bi došao do ciljnih tačaka, na osnovu informacija koje se dobijaju iz kamere.

Importujemo različite biblioteke i module koji su neophodni za njegov rad, kao npr. *NumPy* (za manipulaciju matrica), ROS (za komunikaciju sa robotom). Koristimo različite strukture podataka i funkcije za manipulaciju podacima i slikama. Inicijalizujemo različite promenljive, redove i kolone stola sa preprekama, redove za čekanje za skladištenje podataka, kao i inicijalnu orijentaciju robota i listu čuvanja sekvence puta.

Funkcije za pronalaženje pozicija čine funkcije *findTargetCenter* i *findRobotPosistion*. Prva funkcija se koristi za pronalaženje pozicije objekta sa određenim brojem. Druga funkcija se koristi za pronalaženje pozicije robota na stolu, takođe označen određenim brojem. Kod *findTargetCenter* koristimo funkciju *numpy.where()* kako bi se pronašle sve koordinate na stolu gde se vrednost podudara sa brojem koji predstavlja naš objekat. Rezultat spomenute funkcije jeste niz sa kooridinatama svih pronađenih ćelija. Nakon toga se odradi proračunavanje srednje koordinate. Funkcija *findRobotPosistion* realizuje se na isti način kao i *findTargetCenter*. Na slici 14 dat je deo koda koji je vezan za ove dve funkcije.

```python
def findTargetCenter(target_num, maze): #pronalazenje srednje koordinate kocke i cilja
    #a = np.array(maze)
    indices = np.where(maze == target_num)
    coordinates = list(zip(indices[0], indices[1]))

    if not coordinates:
        print("No matching coordinates found.")
    else:
        middle_index = len(coordinates) // 2
        middle_x, middle_y = coordinates[middle_index]

    return int(middle_x), int(middle_y)


def findRobotPosition(target_num, maze): #pronalazenje srednje koordinate robota
    #a = np.array(maze)
    indices = np.where(maze == target_num)
    coordinates = list(zip(indices[0], indices[1]))

    if not coordinates:
        print("No matching coordinates found.")
    else:
        middle_index = len(coordinates) // 2
        middle_x, middle_y = coordinates[middle_index]

    return int(middle_x), int(middle_y)
```

Slika 14. Funkcije findTargetCenter i findRobotPosistion

Funkcija *calculate_movement* koristi se za izračuvanje pokreta robota na osnovu trenutne pozicije, sledeće pozicije i prethodne orijentacije robota. Data funkcija vraća i informaciju o novoj orijentaciji, nakon što je izvršen određeni pokret. Pored potrebnih argumenata, imamo deo gde se računa razlika u koordinatama. Funkcija prvo računa razlike između x i y koordinata trenutne pozicije i sledeće pozicije. Izračunata vrednost se smešta u promenljive *row_diff* i *col_diff* koje predstavljaju razlike u redovima i kolonama. Na osnovu izračunate razlike u koordinatama i prethodne orijentacije, funkcija određuje koji pokret treba izvršiti, odnosno, koja je nova orijentacija robota nakon izvršenja tog pokreta. Odluka o pokretima i orijentaciji zavisi od toga u kojem smeru se robot kreće u odnosu na trenutnu orijentaciju i poziciju. Povratna vrednost ove funkcije čija dva stringa. Prvi string predstavlja pokret koji robot treba da izvrši, a drugi novu orijentaciju robota nakon izvršenja pokreta. Pokret koji može da izvrši jeste napred (f), nazad (p), levo (l), desno (r). Nova orijentacija može da bude gore, dole, levo ili desno.

```python
#calculating movements based on position and orientation of robot
def calculate_movement(current_position, next_position, previous_orientation): #returns movement and orientation
    row_diff = current_position[0] - next_position[0]
    col_diff = current_position[1] - next_position[1]

    if previous_orientation == "down":
        if row_diff == 1:
            return "p", "down" #using sing p for reverse because r is taken for the right turn
        elif row_diff == -1:
            return "f", "down"
        elif col_diff == 1:
            return "r", "left"
        elif col_diff == -1:
            return "l", "right"
    elif previous_orientation == "up":
        if row_diff == 1:
            return "f", "up"
        elif row_diff == -1:
            return "p", "up"
        elif col_diff == -1:
            return "r", "right"
        elif col_diff == 1:
            return "l", "left"
    elif previous_orientation == "right":
        if row_diff == 1:
            return "l", "up"
        elif row_diff == -1:
            return "r", "down"
        elif col_diff == -1:
            return "f", "right"
        elif col_diff == 1:
            return "p", "right"
    elif previous_orientation == "left":
        if row_diff == 1:
            return "r", "up"
        elif row_diff == -1:
            return "l", "down"
        elif col_diff == 1:
            return "f", "left"
        elif col_diff == -1:
            return "p", "left"
    else:
        return None, None  # Handle invalid orientation
```

Slika 15. Funkcija calculate_movement

Funkcija *save_movements_to_file* koristi se za čuvanje liste pomenutih pokreta u tekstualnoj datoteci.

Za proveru, da li boja putokaza odgovara nekim od boja ciljeva unutar određenog radijusa, definisali smo funckiju *check_color_around_object*. Ukratko, postoji matrica koja predstavlja sliku/kartu na kojoj se vrši provera boje. Proveravamo da li se izračunate koordinate adj_x i adj_y nalaze unutar prethodno zadatog prečnika (radius). Ako su tačke unutar granice matrice, uzimamo vrednost boje za tu tačku i smeštamo je u promenljivu color, što predstavlja boju putokaza. Nakon što dobijemo boju putokaza, funkcija proverava da li ta boja odgovara ciljnoj boji (*target_color*), odnosno cilju 1 ili 2. Ako postoji podudaranje, onda funkcija vraća *True*, suprotno vraća *False*.

```python
#check if the color around object match colors of finish zone
def check_color_around_object(matrix, position, target_color, radius):
    x, y = position
    for i in range(-radius, radius + 1):
        for j in range(-radius, radius + 1):
            adj_x, adj_y = x + i, y + j
            if 0 <= adj_x < len(matrix) and 0 <= adj_y < len(matrix[0]):
                color = matrix[adj_x][adj_y]
                if color == target_color:
                    return True
    return False
```

Slika 16. Funkcija check_color_around_object

Predzadnja funkcija koja čini ovaj fajl jeste main koja predstavlja glavni deo programa za rešavanje kretanje robota kroz razne prepreke. Na početku inicijalizujemo čvor *maze_solver_node* i objavljivača *turtle_motion*. Dalje pronalazimo početne pozicije robota, ciljeva i objekta na stolu, ispisujemo njihove koordinate i postavljemo ih kao tuple sa odgovarajućim koordinatama. Nalazimo najkraće rastojanje od početne pozicije robota do objekta, ako je pronađena ruta, čuva se u tekstualnom fajlu, a ako nije, onda onda se ispisuje „Route not found“. Nakon toga, program čeka da kamera učita odgovarajuću sliku stola sa preprekama. Proveravamo boju putokaza i na osnovu boje putokaza računa se nova ruta do odgovarajućeg cilja. Koraci se čuvaju i objavljuju kao i u prethodnom delu koda. Na samom kraju pokrećemo ovu glavnu funkciju.

7 Zaključak

Osvrnuvši se na sve prethodno rečeno možemo reći da realizovani projekat za autonomno kretanje robota u okviru *Webots* okruženja zajedno sa *ROS* i *OpenCV* obradom slike ispunjava sva željena očekivanja. Prednost ovakve realizacije projekta omogućava robotu da nađe optimalnu putanju kretanja od starta do objekta i na kraju nazad do ciljne podloge bez obzira na izgled zadatih prepreka.

Jedan od nedostataka i mana ovog projekta predstavlja neusklađenost brzine kojom se slike sa kamere prosleđuju na zadatu temu u ROS okruženju kao i njihovo kasnije ispisivanje u txt formatu kao već prikazana matrica.

Takođe zbog nepreciznosti kretanja robota u nekim slučajevima može doći da se zbog netačnog koraka njegovog kretanja pogreši pri pozicioniranju na zadato polje ili da se promaši željena unapred predviđena ruta.

Ovaj problem se može unaprediti korišćenjem enkodera, žiroskopa ili lidara i ultrazvučnih senzora kojim bi se:

- Utvrdilo tačno i precizno kretanje robota i smanjila greška koraka
- Sprečilo da robot udari u prepreku ili u zid