České vysoké učení technické v Praze
Fakulta strojní

Ústav přístrojové a řídící techniky
Odbor Automatického řízení

Adaptace algoritmů pro navigaci robota na základě apriorních informací

Adaptation of algorithms for robot navigation based on a priori information

BAKALÁŘSKÁ PRÁCE

Vypracoval: Daniel Čech
Vedoucí práce: Ing. Mgr. Jakub Jura, Ph.D.
Rok: 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Čech** Jméno: **Daniel** Osobní číslo: **509005**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Teoretický základ strojního inženýrství**

Studijní obor: **bez oboru**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Adaptace algoritmů pro navigaci robota na základě apriorních informací

Název bakalářské práce anglicky:

Adaptation of algorithms for robot navigation based on a priori information

Pokyny pro vypracování:

Úkoly:
1) Provést rešerši způsobů reprezentace vnějšího prostředí v robotu (robotické mapy)
2) Provést rešerši metod a knihoven pro Python pro hledání robotické cesty v mapě
3) Vyzkoušet jednoduchou neinteraktivní mapu na robotu ze stavebnice LEGO MINDSTORM.
4) Otestovat vybrané algoritmy hledání cesty

Seznam doporučené literatury:

[1] León Araujo, H.; Gulfo Agudelo, J.; Crawford Vidal, R.; Ardila Uribe, J.; Remolina, J.F.; Serpa-Imbett, C.; López, A.M.; Patiño Guevara, D. Autonomous Mobile Robot Implemented in LEGO EV3 Integrated with Raspberry Pi to Use Android-Based Vision Control Algorithms for Human-Machine Interaction. Machines 2022, 10, 193. https://doi.org/10.3390/machines10030193

[2] SAKAI, Atsushi; INGRAM, Daniel; DINIUS, Joseph; CHAWLA, Karan; RAF-FIN, Antonin; PAQUES, Alexis. PythonRobotics: a Python code collection of robotics algorithms. 2018. Dostupné z arXiv: 1808.10703 [cs.RO].

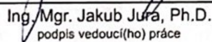
Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Mgr. Jakub Jura, Ph.D. U12110.3

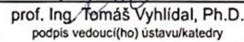
Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **26.04.2024** Termín odevzdání bakalářské práce: **31.05.2024**

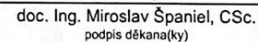
Platnost zadání bakalářské práce:



Ing. Mgr. Jakub Jura, Ph.D.
podpis vedoucí(ho) práce



prof. Ing. Tomáš Vyhlídal, Ph.D.
podpis vedoucí(ho) ústavu/katedry



doc. Ing. Miroslav Španiel, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

24.4.2024
Datum převzetí zadání

Čech
Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Golčově Jeníkově dne 20. 5. 2024

Daniel Čech

Poděkování

Děkuji Ing. Mgr. Jakubu Jurovi, Ph.D. za vedení mé bakalářské práce a za podnětné návrhy, které ji obohatily. Děkuji také panu Ing. Pavlu Trnkovi, Ph.D. za pomoc s praktickou částí práce. Dále děkuji rodině a blízkým za podporu a trpělivost.

Daniel Čech

Název práce:
Adaptace algoritmů pro navigaci robota na základě apriorních informací

Autor: Daniel Čech

Studijní program: Teoretický základ strojního inženýrství

Obor: bez oboru

Druh práce: Bakalářská práce

Vedoucí práce: Ing. Mgr. Jakub Jura, Ph.D.

Abstrakt: Obsahem této práce je sestrojení robota za použití stavebnice LEGO Mindstorms EV3. Na robotovi jsou vyzkoušeny různé algoritmy hledání cesty v robotických mapách. Řízení pohybu robota je programováno v Pythonu za použití dostupných Python knihoven, které se používají na poli robotiky. S tím souvisí rešerše robotických map, algoritmů pro hledání cesty a také knihoven pro Python, které jsou vhodné pro robotiku.

Klíčová slova: LEGO EV3, Python, robotické mapy, algoritmy hledání cesty

Title:
Adaptation of algorithms for robot navigation based on a priori information

Author: Daniel Čech

Abstract: The scope of this work is to build a robot using the LEGO Mindstorms EV3 kit. Different path finding algorithms in robot maps are tested on the robot. The motion control of the robot is programmed in Python using available Python libraries that are used in the field of robotics. This is related to the research of robotic maps, path finding algorithms and Python libraries that are suitable for robotics.

Key words: LEGO EV3, Python, robotics maps, pathfinding algorithms

Obsah

Seznam použitých zkratek **viii**

Seznam obrázků **ix**

Úvod **1**

1 Rešerše robotických map a algoritmů pro nalezení cesty v nich **2**

1.1 Druhy robotických map . . . 2

1.1.1 Metrické mapy . . . 2

1.1.2 Topologické mapy . . . 3

1.2 Algoritmy hledání trasy . . . 5

1.2.1 Potenciálová pole . . . 5

1.2.2 Algoritmus Breadth First Search . . . 6

1.2.3 Algoritmus Depth First Search . . . 7

1.2.4 Metoda Dhouib-Matrix-SPP (DM-SPP) . . . 8

1.2.5 Pseudo-Voronoi diagramy . . . 9

1.2.6 Dijkstrův algoritmus . . . 10

1.2.7 A* algoritmus . . . 10

2 Python v robotice **12**

2.1 Platformy pro Python . . . 12

2.1.1 Raspberry Pi . . . 12

2.1.2 Arduino . . . 13

2.2 Knihovny pro robotiku v Pythonu . . . 14

2.2.1 KLAMPT . . . 14

2.2.2 ROS (Robot Operating System) . . . 15

2.2.3 PythonRobotics . . . 15

2.2.4 PyRobot . . . 16

2.2.5 Pybotics . . . 16

2.2.6 PyRoboCOP . . . 17

2.2.7 PyLabRobot . . . 17

2.2.8 OSGAR (Open Source Garden/Generic Autonomous Robot) . . . 17

3 Praktická část **19**

3.1 Příprava EV3 pro práci s Pythonem . . . 19

3.1.1 Nahrání debiánu Linuxu do EV3 . . . 19

3.1.2 Propojení kostky s PC pomocí Bluetooth . . . 21

3.2 Koncepty konstrukce robota . . . 23

3.2.1 Konstrukce č.1 . . . 24

3.2.2 Konstrukce č.2 . . . 24

3.2.3 Konstrukce č.3 . . . 24

3.3 Lokalizace robota na mapě . . . 25

3.4 Programování robota . . . 26

3.4.1 Použitá mapa 26
3.4.2 Použité algoritmy 27
3.4.3 Modifikace kódu 29
3.5 Testování na robotovi 33
3.5.1 Zmenšení mapy 33
3.5.2 Zjednodušení mapy 33
3.5.3 32-bit x 64-bit 34
3.5.4 Debugování kódu 34

Závěr **36**

Bibliografie **37**

Seznam použitého SW **41**

Přílohy **42**
A Zdrojové kódy 42
B Videa 42

Seznam použitých zkratek

SBC Single Board Computer (*Jednodeskový počítač*)

IoT Internet of Things(*I*nternet věcí)

GPIO General-purpouse input/output(*O*becný vstup/výstup)

RTT Rapidly Expanding Random Tree(*R*ychle rostoucí náhodný strom)

LIDAR Light Detection And Ranging(*D*etekce a měření vzdálenosti pomocí světla)

GA Generic Algorithm(*G*enerický algoritmus)

C-space Configuration Space(*K*onfigurační prostor)

PRM Probabilistic Roadmap(*P*ravděpodobnostní plán cesty)

DFS Depth First Search(*P*rohledávání do hloubky)

BFS Breadth First Search(*P*rohledávání do šířky)

IDFS Iterative Depth First Search(*I*terativní prohledávání do hloubky)

LDFS Limited Depth First Search(*L*imitované prohledávání do hloubky)

SPI Serial Peripheral Interface(*R*ozhraní pro sériové periferní zařízení)

I2C Inter-Integrated Circuit(*S*ériový sběr dat)

Seznam obrázků

1.1 Tvorba mřížkové mapy pomocí LIDAR senzoru [5] 3
1.2 Geometrická mapa [6] 3
1.3 Transformace metrické mapy na mapu topologickou [8] 4
1.4 Lokální minimum [10] 6
1.5 Záplavové vyhledávání trasy [10] 6
1.6 Prohledávací strom BFS [16] 7
1.7 Prohledávací strom DFS [18] 8
1.8 Porovnání testovaných algoritmů [19] 9
1.9 Cesta nalezená algoritmem DM-SPP [19] 9
1.10 Voronoi diagram [20] 10
1.11 Hledání trasy pomocí kombinace Voroni diagramu a Dijkstrova algoritmu [5] 10
1.12 Posloupnost Dijkstrova algoritmu [4] 10

2.1 Spojení Raspberry Pi a LEGO EV3 [26] 13
2.2 Arduino UNO [28] 14
2.3 Příklad generování trajektorie pomocí Pybotics [36] [37] 17

3.1 Program Etcher 20
3.2 Obsah microSD karty 20
3.3 Bootování kostky EV3 20
3.4 Instalace rozšíření pro Python 21
3.5 Nainstalované rozšíření pro Python 22
3.6 Hledání zařízení pro připojení 22
3.7 Zařízení je připraveno k použití 23
3.8 Spouštění skriptu z kostky 23
3.9 Pohled zepředu 24
3.10 Pohled zleva 24
3.11 Pohled zepředu 24
3.12 Pohled zleva 24
3.13 Pohled zepředu 25
3.14 Pohled zleva 25
3.15 Mapa vytvořena v MS Excel 27
3.16 Fyzická mapa 27
3.17 Algoritmy bez diagonálního pohybu po mřížce 28
3.18 Algoritmy s diagonálním pohybem po mřížce 29
3.19 Původní stav 34
3.20 Zjednodušená mapa 34
3.21 Trasa nalezená v Pythonu 3.12 35
3.22 Trasa nalezená v Pythonu 3.6 35

3.23 Trasa nalezená v MicroPythonu 35

Seznam zdrojových kódů

3.1 Definice pohybu v grafu . . . 29
3.2 Vytvoření matice rx . . . 30
3.3 Spojení matice r__x a r__y do matice Q . . . 30
3.4 Import potřebných knihoven . . . 30
3.5 Začátek programu . . . 30
3.6 Definice absolutního pohybu robota . . . 31
3.7 Definice relativního pohybu robota . . . 31
3.8 Definice proměnné steer__pair . . . 32
3.9 Definice funkcí pro pohyb robota . . . 32
3.10 Odometrický výpočet pro řízení robota . . . 32
3.11 Definice proměnných `open_set` a `closed_set` . . . 34

Úvod

Tématem této práce je adaptace algoritmů pro plánování trasy mobilního robota na základě apriorně dodaných informací. Práce je rozdělena na rešeršní a praktickou část. V rešeršní části byly prozkoumány robotické mapy a různé algoritmy pro plánování trasy robotů v těchto mapách. Dále je obsahem práce rešerše knihoven pro robotiku v Pythonu. V praktické části je popsáno programování mobilního robota LEGO Mindstorms EV3 pomocí jazyka Python, za využití vybrané knihovny pro robotiku.

Roboti se v terénu - ve vnějším prostředí mohou pohybovat dle předem přidělených map - robotických map, nebo si je dokáží během chodu vytvářet.

Takováto robotická mapa je v nejužším slova smyslu informace o vnějším prostředím. Rozlišuje prostupný a neprostupný terén - překážky a topografické vztahy mezi nimi. Často se tyto informace zapisují do matice, která reprezentuje 2D terén reálného prostředí.

Kód, kterým je robot řízen, obsahuje algoritmus na vyhledávání trasy. Tento algoritmus hledá nejkratší cestu do cíle tak, aby se vyhnul všem překážkám, na které po cestě narazí.

Robot využívaný v této práci se jmenuje LEGO Mindstorms EV3. Tohoto robota lze programovat jak za využití SW od firmy LEGO - LEGO MINDSTORMS EV3 Home Edition - grafický programovací jazyk, který funguje na bázi funkčních bloků, tak i pomocí vyšších programovacích jazyků jako Python, Java a další.

Kapitola 1

Rešerše robotických map a algoritmů pro nalezení cesty v nich

Roboti se dají dělit na reaktivní a proaktivní. Reaktivní roboti fungují na principu reakce na okolní podněty, bez schopnosti plánovat následující akce. Tito roboti se vyznačují jednoduchými algoritmy řízení, které zpravidla fungují na principu "if-else".

Druhá skupina robotů je schopna aktivně pracovat na dosažení cíle. Právě touto skupinou robotů se zabývá tato práce.

Aby byl robot schopný samostatně vykonat úkol, v tomto případě se jedná o dojetí ze startovní pozice do cíle, musí být splněny určité podmínky. Robot, stejně jako člověk, který se svépomocí potřebuje dostat z bodu A do bodu B, musí mít představu o vnějším prostředí. Toho je dosaženo pomocí robotických map.

Pokud je tento předpoklad splněn - robot má představu o vnějším prostředí, nastává samotné hledání a plánování trasy. Toto hledání a plánování trasy se v širším slova smyslu dělí na informované a neinformované. Tato práce se v praktické části zabývá informovaným hledáním trasy. [1] [2] [3]

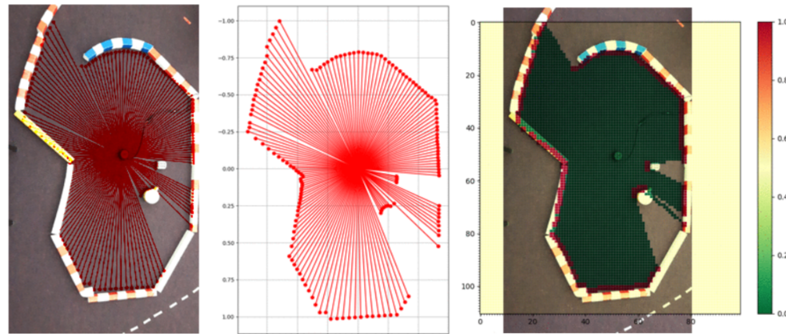
1.1 Druhy robotických map

Robotická mapa je způsob, jakým je robotu reprezentováno vnější prostředí. Potřebuje ji k tomu, aby mohl bezpečně dojít ze startovní pozice do cíle a aby se vyhnul překážkám, na které po cestě narazí. Existuje mnoho druhů těchto map - základním dělením těchto map je dělení na metrické a topologické mapy. [2] [4]

1.1.1 Metrické mapy

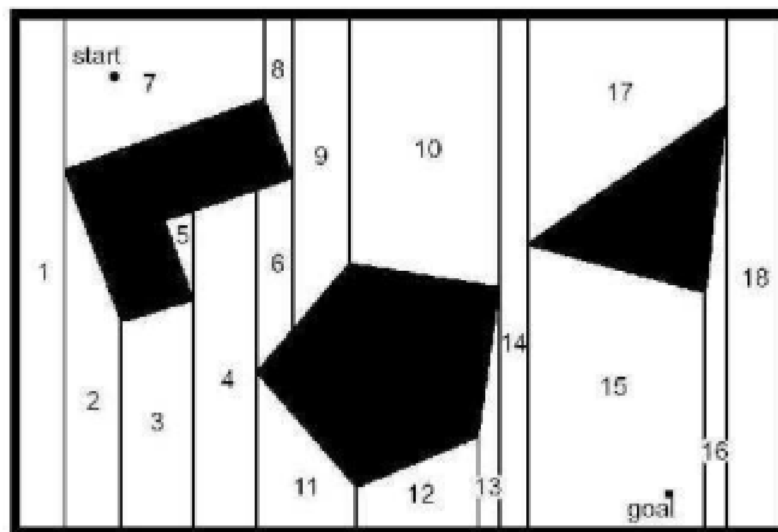
Metrické mapy jsou jedním z nejjednodušších způsobů vnímání vnějšího prostředí pro robota. Vnější prostředí je reprezentováno souřadnicemi na 2D prostoru. Díky těmto mapám je robot schopen celkem detailně vnímat okolní prostředí a pohybovat se v něm. Míra detailnosti vnímání prostředí závisí na rozlišení dané mapy. Metrické mapy se dělí také na několik druhů map.

- **Mřížková mapa** pracuje s reprezentací vnějšího prostředí ve formě mřížky - matice. Mapa je vhodná pro informované hledání trasy a jedná se o jeden z nejjednodušších způsobů reprezentace vnějšího prostředí pro mobilního robota. Robot je schopen si tuto mapu vytvořit sám pomocí senzorů, nebo mu může být dodána. [2] [5] [4]



Obrázek 1.1: Tvorba mřížkové mapy pomocí LIDAR senzoru [5]

- **Geometrické mapy** Tento způsob mapování pracuje s geometrickými útvary - vnější prostředí se dělí na geometrické útvary - například přímky, kružnice, lichoběžníky atd. Výsledná mapa je abstrakcí vnějšího prostředí, což se využívá k výslednému plánování trasy. [2] [4] [6]



Obrázek 1.2: Geometrická mapa [6]

1.1.2 Topologické mapy

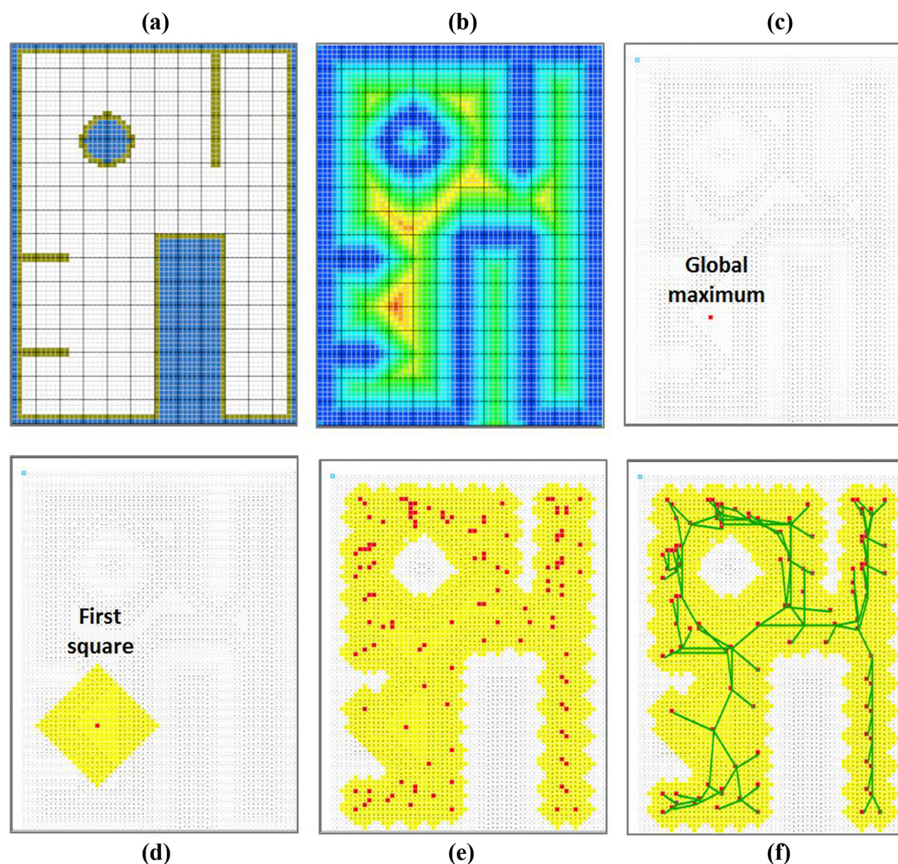
Topologické mapy jsou ještě více abstraktní než geometrické mapy. Jedná se o grafický model, který zprostředkovává informaci o vnějším prostředí pomocí uzlů a hran. Uzly grafu jsou významné oblasti, které jsou spojeny hranami. Tyto uzly často obsahují informace o své poloze nebo jiné významné charakteristiky prostředí, jako například typ oblasti nebo vlastnosti scény.

Topologický prostor je dvojice $(X, \mathcal{O})$, kde X je (typicky nekonečná) základní množina a $\mathcal{O} \subseteq 2^X$ je soustava množin, jejíž prvky se nazývají otevřené množiny, taková že platí následující podmínky:

1. $\emptyset \in \mathcal{O}$ (prázdná množina je otevřená množina),
2. $X \in \mathcal{O}$ (celá základní množina je otevřená množina),
3. Průnik konečně mnoha otevřených množin je otevřená množina,
4. Sjednocení libovolné kolekce otevřených množin je otevřená množina. [7]

Hrany grafu mohou být ohodnoceny metrickými údaji, jako je vzdálenost mezi uzly, nebo mohou být neohodnocené, představující pouze spojení mezi dvěma nebo více uzly. Pro lepší představu si lze uzly představit jako křižovatky a hrany jako silnice, které tyto křižovatky propojují.

Plánování trasy na topologické mapě obecně vyžaduje méně výpočetní síly než plánování na geometrické mapě, ale navigace je často omezena na opakování již známých tras. Zkracování tras na topologické mapě obvykle není možné, ale zlepšení efektivity navigace lze dosáhnout především optimalizací jednotlivých cest. Jednou z výhod topologických map je možnost transformace z metrické mapy na topologickou. To umožňuje využít výhod abstraktního reprezentace prostředí pro navigaci, přičemž lze stále využít přesného geometrického popisu prostoru. [2] [8] [4] [9]



Obrázek 1.3: Transformace metrické mapy na mapu topologickou [8]

1.2 Algoritmy hledání trasy

V rámci plánování trasy nad apriorními daty jsou robotu poskytnuty veškeré potřebné informace - o pozici startu, cíle a jednotlivých překážkách - jinými slovy má robot přesnou informaci o vnějším prostředí. Dále je implementován algoritmus, který hledá nejkratší a nejvhodnější cestu.

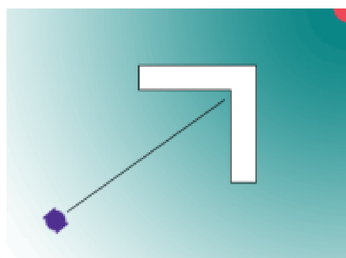
Při využití hledání trasy robot pracuje s reprezentací vnějšího prostředí - s robotickou mapou. Existuje velké množství různých algoritmů. Některé jsou použitelné pouze na metrických mapách, některé pouze na topologických a je tak důležité použít správný algoritmus pro odpovídající mapu.

V další části jsou zmíněny některé základní algoritmy vyhledávání trasy v robotické mapě, kdy je robotu známa jak pozice startu a cíle, tak i všech překážek, které se na mapě objevují. Některé dále zmíněné algoritmy jsou poté využity v praktické části této práce. [10] [11] [12]

1.2.1 Potenciálová pole

Metoda potenciálových polí je jedna z jednodušších metod pro hledání trasy na mřížce. Algoritmus spočívá v tom, že každému poli na mapě přiřazuje hodnotu, která v konečném důsledku slouží pro výběr finální trasy. Pro cíl se obvykle volí nejnižší hodnota a pro start nejvyšší - je zde snaha se dostat z místa s vysokým potenciálem do místa s potenciálem nízkým. V nejjednodušším případě je každé sousední buňce přiřazena její euklidovská hodnota - počet kroků, které je potřeba vykonat pro přemístění z počáteční polohy do aktuální polohy. Takto se pokračuje dokud algoritmus nedojde do startovního bodu, neboť se začíná v cíli. Robot se poté přesouvá vždy do buňky s nižší hodnotou tak dlouho, dokud nedojde do cíle nebo dokud se nezasekne v lokálním minimu.

Tato metoda je ale vhodná pouze pro překážky, u kterých nehrozí uvíznutí v lokálním extrému. V opačném případě se robot zasekne v lokálním minimu a uvízne. Je tedy nutné algoritmus upravit tak, aby k tomuto problému nedocházelo a bylo možné ho využít pro všechny typy překážek. Využije se tedy algoritmus záplavového vyplňování (wavefront-expansion). Algoritmus opět začne v cíli, vezme každou očíslovanou buňku a všem jejím sousedním buňkám přiřadí hodnotu a jednotku vyšší. Takto dochází k vytvoření pole, které má pouze jedno lokální minimum - cíl. [13] [10]



Obrázek 1.4: Lokální minimum [10]

9	8	7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2	1	1
9	8	7	6	5	4	3	2	2	2
9	8	7					3	3	3
9	8	8	8	9	10		4	4	4
9	9	9	9	9	9		5	5	5
10	10	10	10	9	8		6	6	6
11	11	11	10	9	8	7	7	7	7
	12	11	10	9	8	8	8	8	8
		11	10	9	9	9	9	9	9

Obrázek 1.5: Záplavové vyhledávání trasy [10]

1.2.2 Algoritmus Breadth First Search

Algoritmus prohledávání do šířky (anglicky BFS - Breadth First Searh) je spolu s algoritmem DFS (Depth First Search - Algoritmus prohledávání do hloubky) jedním ze základních algoritmů prohledávání stavového prostoru. Tento algoritmus je velmi často využívaný právě pro slepé prohledávání mapy. Jedná se o algoritmus neinformovaného vyhledávání - algoritmus postupuje od počátečního bodu bez určitého směru - prohledává všechny uzlové body v mapě dokud nenalezne cíl. Algoritmus pracuje tím způsobem, že uzly v mapě jsou prozkoumávány po úrovních - nejprve se začne v počátečním bodě a z něho se expanduje do všech sousedních bodů, pro které se vyhodnotí jejich cena - počet kroků, potřebných k jejich dosažení ze startovní pozice. Pro všechny tyto body algoritmus pokračuje stejným způsobem.

Je zde také využita fronta - část programu, do které se ukládají uzlové body, které čekají na zpracování. Algoritmus dělí uzly na tři skupiny:

• **Živé buňky** - uzly, které již byly navštíveny, ale sousední uzel nebo uzly ještě nebyly vyhodnoceny. Na začátku je živým uzlem pouze startovní pozice a dále se přidávají uzly ostatní.

• **Mrtvé uzly** - pozice, které již byly vyhodnoceny a již nemají sousední uzly, které čekají na vyhodnocení. Algoritmus již tyto uzly pro další hledání trasy nepotřebuje.

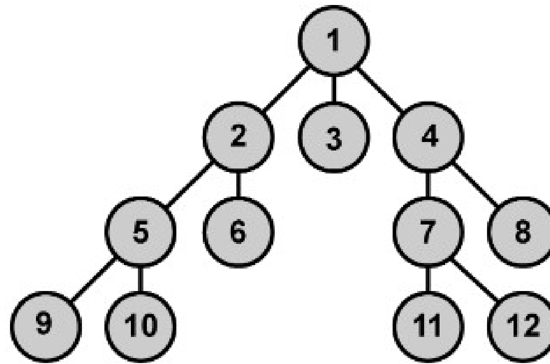
• **Nenavštívené** - uzly, jež stále nebyly vyhodnoceny - po vyhodnocení se z nich stanou živé uzly a po vyhodnocení sousedních uzlů se z nich stávají mrtvé uzly. Na začátku algoritmu se jedná o všechny uzly kromě počátečních.

Výhodou tohoto algoritmu je fakt, že pokud řešení existuje, bude vždy nalezeno, a pokud existuje různých řešení více, vždy bude vybráno to, které má nejmenší hloubku - nejmenší počet kroků. Cesta nalezená tímto algoritmem je často jedna z nejkratších a nejvýhodnějších. Při zvětšující se hloubce prohledávaného prostoru však algoritmus ztrácí na účinnosti. Maximální počet vyšetřených uzlů je totiž definován rovnicí (1.1), kde b značí větvící faktor a d je hloubka cíle.

$$O(BFS) = 1 + b^2 + b^3 + b^4 + ... + b^d \tag{1.1}$$

Pro prohledávání na matici, kde je uvažován pohyb pouze ve 4 směrech je tak větvící faktor roven pouze číslu 3, neboť algoritmus nevyhodnocuje již navštívené uzly. Při

relativně nízkých hloubkách cíle je tento algoritmus výhodný, ale při zvyšujících se hloubkách roste výpočetní čas pro tento algoritmus exponenciálně. [14] [15] [10] [16]



Obrázek 1.6: Prohledávací strom BFS [16]

1.2.3 Algoritmus Depth First Search

Algoritmus DFS - prohledávání do hloubky patří stejně jako BFS do základních algoritmů prohledávání stavového prostoru. Opět se jedná o způsob neinformovaného prohledávání. Na rozdíl od BFS ale funguje tak, že prohledává body do hloubky dokud nenarazí na cíl nebo dokud uzel nejde dále expandovat. Pokud nelze uzel dále expandovat, vrací se algoritmus o krok zpět a expanduje z dalšího uzlového bodu tak dlouho, dokud nenastane jedna ze dvou zmíněných situací. Tento algoritmus sice neprohledává celý stavový prostor mapy, ale je výpočetně mnohem méně náročnější, neboť maximální počet uchovaných uzlů je součtem počtu uzlů podél aktuální cesty a všech neexpandovaných sousedních uzlů. Výraz definující maximální počet vyšetřených uzlů je tedy výraz (1.2)

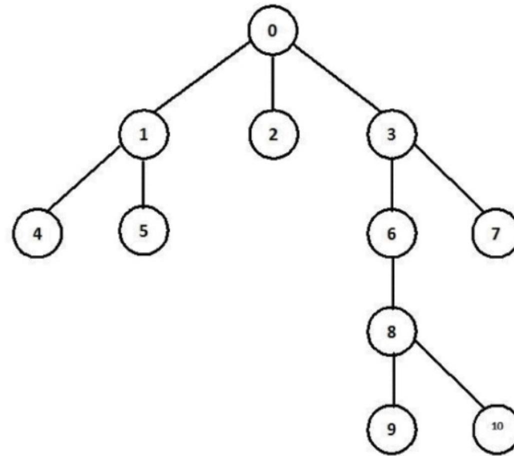
$$O(DFS) = bm \tag{1.2}$$

kde b opět značí větvící faktor a m značí maximální hloubku cíle. Za předpokladu, že jeden uzel zabere 100 B paměti a $b = m = 12$ lze dojít k výsledku, že BFS na prohledání stejné mapy využije 111 TB a DFS pouze 12 kB. V nejhorším případě je však časová složitost O (1.3) stejná jako u BFS (1.4):

$$O(b^m) \tag{1.3}$$

$$O(b^d) \tag{1.4}$$

U DFS je však nevýhodné to, že se může zaseknout na slepé větvi, která může být velmi hluboká a při tom cíl leží v mnohem výše položené vrstvě. K tomuto problému se dá přistoupit např. omezením prohledávané hloubky (LDFS) či iterativním prohlubováním (IDFS), což DFS přibližuje algoritmu BFS. [17] [15]



Obrázek 1.7: Prohledávací strom DFS [18]

1.2.4 Metoda Dhouib-Matrix-SPP (DM-SPP)

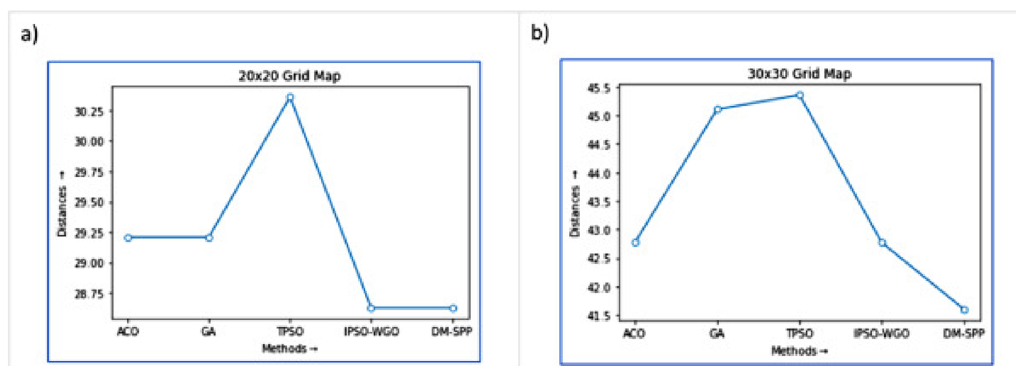
Jedná se o algoritmus vyhledávání trasy v prostředí, které je reprezentováno mřížkovou mapou (OGM - Occupancy Grid Map). Tento algoritmus byl publikován v roce 2023 a současné době se jedná o jeden z nejrychlejších algoritmů pro nalezení nejkratší a nejrychlejší trasy. Metoda uvažuje pohyb robota do 8 směrů - do všech okolních buněk mapy okolo pozice robota. Algoritmus pracuje s časovou komplexností, která je definována jako

$$O(n + m) \tag{1.5}$$

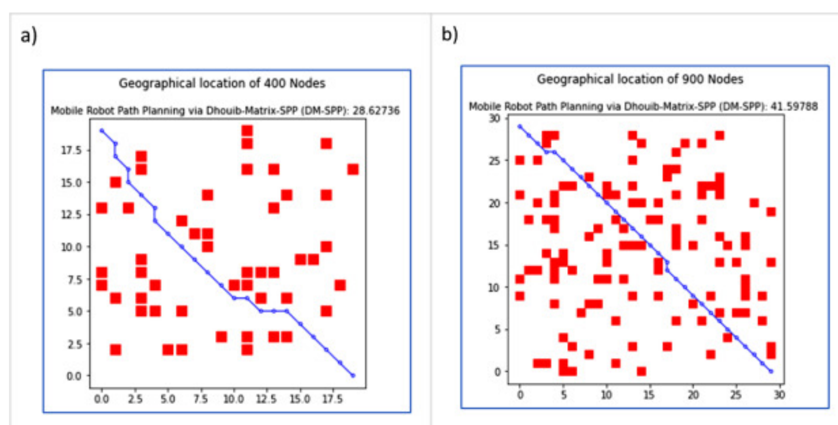
Proměnná m značí počet hran a n značí počet vrcholů. Při uvažování kompletního grafu - všechny vrcholy jsou propojeny, výsledkem je tedy výraz $m = n^2 - n$, což se odráží ve výsledné časové komplexnosti, která je tedy

$$O(n^2) \tag{1.6}$$

Tato hodnota odpovídá Dijkstrově algoritmu. Výhodou této metody je ale uvažování pohybu na mřížce, tudíž nejvyšší možná hodnota vrcholů je $n = 8$. Při uvažování překážek na mapě se celková hodnota časové komplexnosti ještě více snižuje, což v důsledku znamená, že s přibývajícím počtem překážek klesá počet vytvořených hran a tudíž je algoritmus rychlejší. Samotná rychlost algoritmu byla také testována na 11 různých mřížkových mapách a byla porovnána s 12 metaheurestickými algoritmy, jako například (RTT, GA (Generic Algorithm), (Ant Colony Optimization), ...). Algoritmus DM-SPP vyšel z tohoto porovnání jako nejrychlejší a tudíž je velmi výhodné jej použít při hledání trasy ve statickém prostředí, které lze reprezentovat mřížkovou mapou. [19]



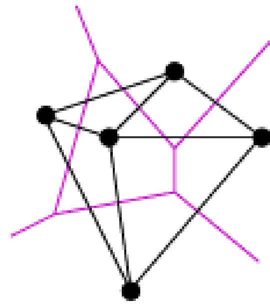
Obrázek 1.8: Porovnání testovaných algoritmů [19]



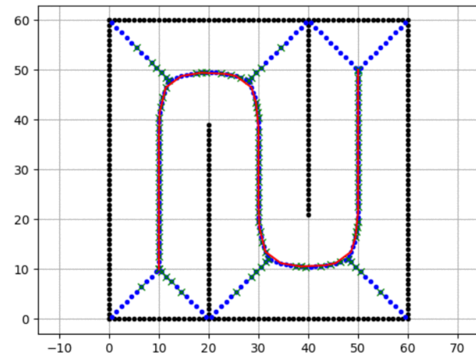
Obrázek 1.9: Cesta nalezená algoritmem DM-SPP [19]

1.2.5 Pseudo-Voronoi diagramy

Pseudo - Voronoi způsob hledání trasy využívá matematickou interpretaci prostředí v podobě Voronoi diagramů. Tento způsob spočívá v rozdělení překážek na n oblastí, které jsou rozděleny hranami. Tyto hrany reprezentují ekvidistantu právě mezi dvěma překážkami. Hrany jsou na Obrázku 1.10 znázorněny fialovou barvou, překážky černými body a spojnice mezi překážkami černými úsečkami. Výhodou je také fakt, že nemusí být uvažovány pouze bodové překážky, ale lze pracovat i s překážkami ve tvaru polygonů. V tomto případě akorát k řídícím bodům přibudou i řídící úsečky. Tato metoda se využívá pro hledání trasy s důrazem na bezpečnost při průchodu trasy - robot bývá interpretován jako kružnice o průměru největšího rozměru robota a při známých rozměrech robota i vzdálenosti překážek od hrany lze vybrat jen tu cestu po hraně, která zajistí bezproblémový průchod. Konstrukce samotných Voronoi diagramů se provádí pomocí několika jednoduchých algoritmů. Jeden z nejjednodušších algoritmů spočívá v redukci překážek pouze na dvě a rozdělení mapy na dvě oblasti s jednou hranou. Postupně se přidávají další překážky až algoritmus vyhodnotí konečný počet n překážek. Pro pohyb robota, který nemá počáteční polohu právě na jedné z hran diagramu je využit algoritmus vyhledávání do šířky či do hloubky, který prohledává oblast dokud nenarazí na bod, který leží na hraně. Pokud cíl také neleží na hraně, opakuje se stejný postup jako na začátku. [20] [21]



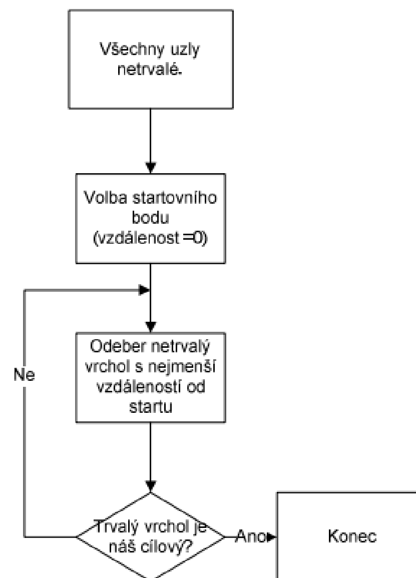
Obrázek 1.10: Voronoi diagram [20]



Obrázek 1.11: Hledání trasy pomocí kombinace Voroni diagramu a Dijkstrova algoritmu [5]

1.2.6 Dijkstrův algoritmus

Dijkstrův algoritmus využívá stejný princip prohledávání do šířky jako grafový přístup. Používá ale na rozdíl od výše zmíněných algoritmů tzv. prioritní frontu. Frontu, ve které jsou vrcholy řazeny podle jejich vzdálenosti a tudíž přesun robota z jednoho pole do druhého nestojí vždy jednu jednotku, ale počet jednotek je závislý na vzdálenosti mezi poli.[4] [10]



Obrázek 1.12: Posloupnost Dijkstrova algoritmu [4]

1.2.7 A* algoritmus

A* algoritmus je jakousi modifikací Dijkstrova algoritmu. Nevýhodou Dijkstrova algoritmu je fakt, že algoritmus prochází všechny pole, kterými se lze dostat do cíle a proto je časově náročnější. A* algoritmus řeší tuto skutečnost tak, že efektivněji třídí vrcholy ve frontě. Opět se jedná o algoritmus informovaného prohledávání. V Dijkstrově algoritmu jsou vrcholy řazeny podle jejich vzdálenosti od cílového bodu.

V A* algoritmu jsou vrcholy řazeny podle součtu vzdálenosti od cíle s odhadem vzdálenosti od začátku. Funkce, která představuje odhad nejvýhodnější - nejlevnější trasy, je značena $f(n)$ a má tento tvar:

$$f(n) = h(n) + g(n), \tag{1.7}$$

kde funkce $h(n)$ představuje heuristický odhad ceny přechodu z n-tého uzlu do cílového uzlu. Funkce $g(n)$ značí cenu vlastní cesty do uzlu n. V algoritmu je využita již zmíněná heuristika - funkce, která provádí samotný odhad pro funkci $h(n)$. Aby byl A* algoritmus A* algoritmem a ne pouze A algoritmem, je zde zavedena podmínka, která zaručuje, že heuristicky odhadnutá vzdálenost je menší nebo rovna skutečné vzdálenosti. Tato podmínka se odpovídá v rovnici 1.8

$$0 \leq h(n) \leq C* \tag{1.8}$$

Tato podmínka určuje, že žádná reálná cesta nemůže být kratší než použitá heuristika. Lze si to představit na vzdušné vzdálenosti mezi městy A a B - tato vzdálenost představuje heuristiku a samotná vzdálenost mezi těmito dvěma městy reprezentuje skutečnou vzdálenost. Skutečná vzdálenost je v nejlepším případě stejně dlouhá jako vzdušná vzdálenost - pokud by byla cesta z města A do města B naprosto rovná. V reálném světě bývá však silnice klikatá, a tak i v tomto případě by byla tato podmínka splněna. Důvod, proč musí být heuristika $h(n)$ vyšší než 0 je takový, že pokud by byla heuristická hodnota rovna 0, stal by se z A* algoritmu obyčejný Dijkstrův algoritmus. Tento algoritmus je velmi výhodný co se potřebné výpočetní paměti týče a proto se jedná o jeden z nejvíce používaných algoritmů v tomto odvětví. Algoritmus je také úplný a vždy nalezne nejkratší cestu. Přesnost tohoto algoritmu je také závislá na použité heuristice, která může průběh algoritmu značně ovlivnit. [10] [22] [23]

Kapitola 2

Python v robotice

Robotika patří mezi významné pilíře moderního technologického vývoje a pokroku. Jsou na ni kladeny nároky na snadné programování a přehlednost kódu. Proto se při vyvíjení robotů a robotických systémů stále více přechází k využití vyšších programovacích jazyků jako je Python, C, C++ či C#. Mnoho výrobců stále používá své nativní jazyky pro programování a řízení svých robotů, a tak může být složité programovat různé roboty se znalostí pouze jazyku jednoho výrobce. Proto se v posledních letech začíná přecházet k již zmíněným programovacím jazykům. Tato práce se zabývá využitím Pythonu v různých oblastech robotiky, zejména ale v oblasti plánování trasy v robotické mapě.

Python je jeden z nejpoužívanějších a nejuniverzálnějších programovacích jazyků současnosti a má velmi široké pole využití. Samotná všestrannost Pythonu spočívá ve snadné struktuře kódu, která je velmi dobře osvojitelná i laikům a další předností je, že Python nabízí velké množství knihoven, které funkce tohoto jazyka ještě více rozšiřují a prohlubují.

2.1 Platformy pro Python

Tato práce se zabývá především využitím stavebnice LEGO Mindstroms EV3, se kterou se dále pracuje v praktické části. Další dvě velmi rozšířené a velmi často používané HW platformy pro toto odvětví robotiky jsou zařízení Raspberry Pi a Arduino. Obě tato zařízení se vyznačují nízkou cenou, širokou škálou využití a obě podporují programování v Pythonu. Výhodou všech těchto tří platforem je možnost zapojení externích zařízení, senzorů a jejich ovládání a řízení za použití různých knihoven, které Python nabízí.

2.1.1 Raspberry Pi

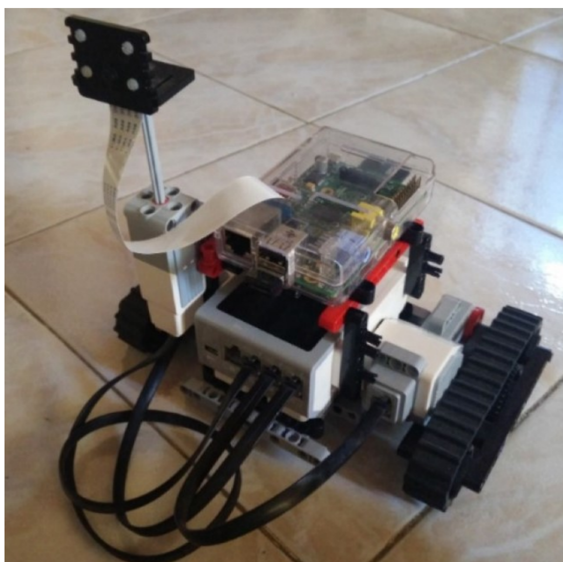
Raspberry Pi je mikropočítač od britské firmy Raspberry Pi Foundation. Původně byl tento SBC počítač zamýšlen k edukativním účelům, jakožto levná a snadno dostupná alternativa ke stolním PC, pro použití hlavně v rozvojových zemích. Pro jeho jednoduchý design a dostupnou cenu vyšlo postupem času mnoho modelů tohoto počítače. Raspberry Pi se nakonec rozšířil i mimo hranice původní cílové skupiny, neboť se jedná o ideální platformu jak pro domácí programování, tak i pro průmyslové použití v robotice. Počítač disponuje USB a HDMI porty a je vybaven OS Linux, což ho činí ideálním pro programování v jazyce Python. Tento SBC počítač lze také

libovolně HW upravovat podle potřeby jedince či firmy, a tak jej činí velmi vhodným prostředkem pro řízení všemožných průmyslových aplikací.

Právě využití v průmyslu, a hlavně v robotice je pro Raspberry Pi velmi významné. Díky malým rozměrům, solidnímu výpočetnímu výkonu a nízké ceně se tak tyto mikropočítače dostaly do průmyslových aplikací i do domácností. Výhodou těchto počítačů je možnost pracovat s nimi jako s normálním počítačem. Na desce tohoto počítače se vyskytuje spousta GPIO pinů, které umožňují ovládat elektronická zařízení, přijímat signály z různých senzorů a podporují také implementaci IoT (internet věcí).

Právě tyto GPIO piny jsou velmi výhodné pro využití Pi v robotice. Díky svým kompaktním rozměrům má Pi velmi rozsáhlou škálu použití. Od dávkování kávy v automatech až po ovládání robotických manipulátorů. S využitím Raspberry Pi a dodatečných senzorů a motorů lze postavit roboty, kteří jsou schopni mapovat vnější prostředí, navigovat a pohybovat se v něm. [24] [25]

Pro složitější aplikace robota EV3 je také možnost propojení EV3 s platformou Raspberry Pi. Na Obrázku 2.1 je zobrazeno spojení EV3 s Raspberry Pi, které bylo využito pro implementaci kamery, která byla využita pro vyvíjení autonomního robotického systému v práci [26].



Obrázek 2.1: Spojení Raspberry Pi a LEGO EV3 [26]

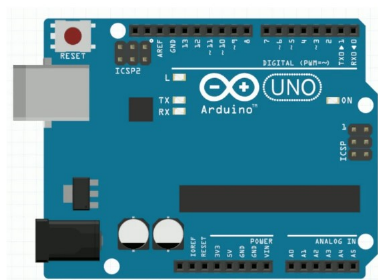
2.1.2 Arduino

Arduino představuje mikrokontrolerovou platformu, která díky své jednoduchosti a dostupnosti nabízí široké využití v oblasti robotiky. Samotný mikrokontroler byl vydán v roce 2005, jakožto učební pomůcka na Instituto de Diseño de Interacción Ivrea (IDII). Jedná se o jednoduchou open-source HW i SW platformu, která je vhodná jak pro začátečníky, tak i pro pokročilé programátory.

Arduino dokáže pracovat jak s analogovými, tak s digitálními signály. V této platformě je také integrováno velké množství komunikačních protokolů jako SPI, I2C a další. Díky svému všestrannému použití může být tento mikrokontroler využit od řízení

motorů až po vzdálenou robotickou laboratoř.

Samotný mikrokontroler používá vlastní programovací jazyk, který je velmi podobný C++, je ale možné pracovat také s Pythonem a dalšími vyššími programovacími jazyky. Python funguje s Arduinem velmi dobře a je vhodný pro úkoly, které vyžadují interakci s různými senzory a dalšími fyzickými zařízeními. [27] [28]



Obrázek 2.2: Arduino UNO [28]

2.2 Knihovny pro robotiku v Pythonu

Jak již bylo zmíněno výše, Python je velmi univerzální programovací jazyk, který nabízí velké množství softwarových rozšíření a knihoven. Existuje velké množství uplatnění Pythonu a téměř pro každou oblast lze najít spoustu rozšíření, které usnadňují orientaci v dané problematice. Většinou se jedná o open-sourcové softwarové balíčky třetích stran, které lze snadno nainstalovat přímo do Pythonu. V další části se tato práce věnuje knihovnám a rozšířením, které usnadňují práci a orientaci na poli robotiky. [29]

2.2.1 KLAMPT

Dostupné z: https://github.com/krishauser/Klampt

KLAMPT (Kris' Locomotion and Manipulation Planning Toolbox) je unifikovaný open-source softwarový balíček pro Python a C++, který je využíván k modelování a simulaci robotů, řeší také kinematiku a dynamiku, plánování pohybu a samotné řízení robota. Je zde také dostupná vizualizace uživatelského prostředí robota. Samotná knihovna je členěna na několik modulů, z nichž každý odpovídá výše zmíněným oblastem. Příkladem může být modul klampt - jedná se o hlavní modul celé knihovny, obsahuje příkazy a funkce, které slouží k simulování dynamiky a kinematiky robota. Dále jsou zde moduly jako klampt.plan, který slouží pro plánování trasy, modul klampt.control slouží pro ovládání a řízení robota a jeho modulů.

Jelikož se tato práce zabývá plánováním tras a navigací robota v terénu, musí být více dopodrobna zmíněn modul klampt.plan. V tomto modulu se nachází velké množství algoritmů pro plánování pohybu jako například metoda RTT (Rapidly-Exploring Random Tree), PRM (Probabilistic Roadmap), MMPRM (Multi-Modal PRM) a další. Je kladen důraz na pohyb robota bez kolizí s vnějším prostředím. Nejvýhodnější funkce v modulu klampt.plan.robotplanning jsou funkce planToConfig(), planToCartesianObjective() a planToSet(). Tyto funkce automaticky inicializují potřebné algoritmy a výsledkem je bezkolizní plán trasy robota do zadaného cíle.

Je také možnost použít libovolný algoritmus pro hledání trasy. Nejprve je nutné definovat tzv. C-space (Configuration Space) - abstraktní prostor, ve kterém je reprezentován pohyb robota. Poté je potřeba určit, jaký algoritmus bude využitý a také je nutné inicializovat koncový a počáteční bod robota. Příkladem je následující řádek kódu:

```
planner = cspace.MotionPlan(space,type="rrt")
planner.setEndpoints(qinit,qgoal)
#now the planner is ready to use...
```

Hodnoty proměnné "type"jsou názvy algoritmů. Např."rrt"označuje použití Rapidly-Exploring Random Tree algoritmu, "prm"znamená algoritmus Probabilistic Roadmap atd. [30] [31]

2.2.2 ROS (Robot Operating System)

Dostupné z: https://github.com/ifurusato/ros

Ačkoliv se Robot Operating System dá přeložit jako Robotický operační systém, nejedná se o operační systém v pravém slova smyslu. Jedná se open-sourcový framework, který disponuje více než 2000 různých balíčků nástrojů, kde každý tento balíček disponuje odlišnými funkcemi. Tato skutečnost činí z ROS velmi vhodný systém pro práci s roboty. Pole využití ROS není omezeno pouze na robotické aplikace, ale většina poskytovaných nástrojů je zaměřena na práci s periferním hardwarem.

ROS nabízí různé funkce pro abstrakci hardwaru, pro práci s ovladači zařízení, komunikaci mezi procesy na více strojích, nástroje pro testování a vizualizaci a spoustu dalších

Klíčovou vlastností ROS je způsob, jakým SW běží a také jakým způsobem komunikuje, což umožňuje navrhovat složité programy bez znalosti toho, jak funguje určitý používaný hardware. ROS poskytuje způsob, jak propojit síť procesů (uzlů) s centrálním uzlem. Uzly mohou být spuštěny na více zařízeních a k tomuto uzlu se připojují různými způsoby. ROS je tak velmi vhodný pro práci s více roboty najednou.

Samotné programování poté probíhá pomocí Pythonu či C++, avšak častěji bývá využíván jazyk C++ pro lepší výkonnost. [32] [33]

2.2.3 PythonRobotics

Dostupné z: https://github.com/AtsushiSakai/PythonRobotics

PythonRobotics je kolekce Python kódů pro robotiku. Jedná se o velmi obsáhlou knihovnu, která je postavená na spolupráci mnoha autorů, kde každý autor může knihovnu rozšířit či upravit nefunkční řádky kódu. Na tomto projektu se podílelo přes 110 autorů. Jedná se o soubor algoritmů, které se používají pro lokalizaci, tvorbu map, SLAM a plánování tras což je velmi vhodné pro využití v rámci této práce. Tato sbírka také obsahuje algoritmy pro řízení robotických manipulátorů a také pro navigaci leteckých zařízení - dronů či raket. Tato knihovna je přístupná všem uživatelům Githubu, kde jsou do přehledných složek uspořádány všechny části kódu, které jsou zde obsaženy. Kód je psán přehledně a obsahuje spoustu komentářů,

které usnadňují pochopení kódu. V online dokumentaci lze pro každou metodu najít přehlednou obrázkovou dokumentaci či animaci ve formě GIF. Proto je tato knihovna vhodná pro začátečníky, ale také pro pokročilé programátory.

Jelikož obsahem této práce je plánování pohybu robota pomocí robotické mapy, je tato knihovna vhodná na další použití v praktické části této práce. PythonRobotics podporuje využití algoritmů pro práci jak s mřížkovými mapami, tak i s geometrickými mapami. Jsou zde také obsaženy algoritmy pro mnohem komplexnější úlohy plánování trasy, než kterými se zabývá tato práce. Mezi zmiňované algoritmy patří např. Dubins path planning, Reeds Shepp planning, Bezier path planning a další.

V nejjednodušších případech bývá mapa robotu předem definována. V některých případech je ale možné, že si robot mapu definuje sám pomocí dostupných dat ze senzorů. Právě i tyto algoritmy nabízí tato knihovna. Mezi hlavní metody tvorby map patří například Normal Distance Transform, Ray casting grid či Lidar to grid. Lze také využít dostupné SLAM algoritmy, které pracují na principu současného pohybu robota a mapování okolního prostředí. [5]

2.2.4 PyRobot

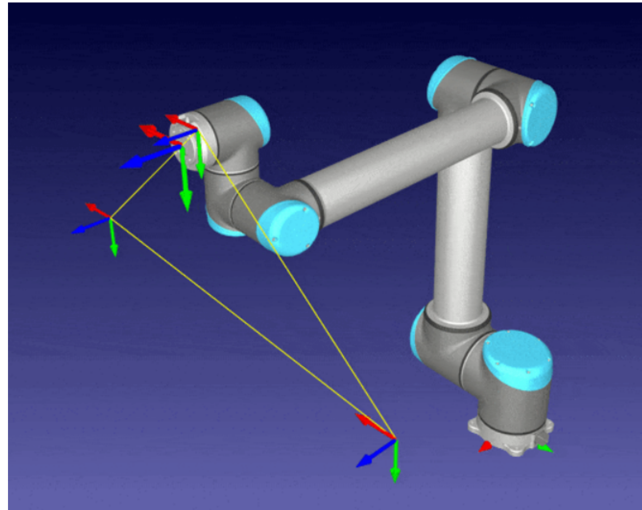
Dostupné z: https://pyrobot.org/docs/navigation

PyRobot je open-sourcová knihovna, která je využívána pro provádění experimentů a testování robotů. Primárně je tato knihovna navržena pro roboty LoCoBot a Sawyer, ale je možnost knihovnu přizpůsobit i pro jiné roboty, a tudíž její rozsah použití rapidně roste. Knihovna obsahuje funkce pro navigaci robota v prostředí a také pro řízení robotického ramena - manipulátoru. [34]

2.2.5 Pybotics

Dostupné z: https://github.com/engnadeau/pybotics

Knihovna Pybotics je open-sourcový softwarový balíček pro programovací jazyk Python, který je využíván především pro vyhodnocení konceptů kinematiky, dynamiky a kalibrace robotů. Podporuje také možnost využití jednoduchého strojového učení s využitím nasbíraných dat. Balíček Pybotics je v průmyslu i ve výzkumu hojně využíván pro kalibraci modelu robota s jeho skutečným protějškem, což ve výsledku vede ke zlepšení výkonnosti a přesnosti fyzického stroje. Tento balíček je navržen speciálně pro práci s konvencí modifikovaných Denavit-Hartenbergových parametrů. Tato konvence bývá hojně využívána pro modelování kinematiky a dynamiky robotických manipulátorů. Ve zkratce se jedná o sadu různých parametrů. které slouží k popisu geometrie sousedních článků manipulátoru, a tudíž zjednodušuje formulaci transformačních matic mezi sousedními souřadnicovými systémy. Tento balíček tak není použitelný pro navigování mobilního robota v prostoru, je ale vhodný pro řízení pohybu robotického manipulátoru. [35] [36] [37]



Obrázek 2.3: Příklad generování trajektorie pomocí Pybotics [36] [37]

2.2.6 PyRoboCOP

Dostupné z: https://github.com/merlresearch/PyRoboCOP

PyRoboCOP je Python knihovna, která řeší řízení, kontrolu a odhad pohybu robotických systémů, které jsou popsány nelineárními diferenciálními rovnicemi. Knihovna dokáže pracovat i se systémy, ve kterých dochází k dotyku s okolními předměty, které jsou popsány za pomoci komplementárních omezení. Knihovna tak poskytuje obecný rámec pro specifikaci omezení pro vyhýbání se překážkám. Výše zmíněné systémy - systémy popsané nelineárními diferenciálními rovnicemi jsou nejčastěji reprezentovány robotickými manipulátory. [38] [39]

2.2.7 PyLabRobot

Dostupné z: https://github.com/PyLabRobot/pylabrobot

PyLabRobot je další z příkladů využití Pythonu v robotice. Sice se nejedná o knihovnu, která řeší zkoumanou problematiku, ale je vhodné ji zde zmínit. Jedná se o open-source rozhraní pro Python, které je zaměřené na řízení liquid-handling robotů a jejich příslušenství. Jedná se o roboty, kteří pracují s kapalinami například v laboratorním či průmyslovém prostředí. Většina těchto robotů používá speciální software od svého výrobce, a tak může být problematické jejich řízení a programování v rámci automatizovaných linek. Proto byl vytvořen tento SW balíček, který podporuje většinu moderních operačních systémů jako Windows, Linux, macOS, či RaspberryPi OS. PyLabRobot podporuje také browser-based simulátor, který umožňuje provádět simulace bez potřebného HW. [40]

2.2.8 OSGAR (Open Source Garden/Generic Autonomous Robot)

Dostupné z: https://github.com/robotika/osgar

Jedná se o knihovnu pro Python vytvořenou českými výzkumníky Martinem Dlouhým, Zbyňkem Winklerem, Jakubem Lvem a Františkem Brabcem. Tato knihovna se zaměřuje na nahrávání a přehrávání dat z jednotlivých senzorů spojených do jednotlivé složky. Jedná se o jednoduchou knihovnu, která je určená pro fungování jak na běžných PC, tak i na mikrokontrolerech jako Raspberry Pi Zero. Knihovna si klade stejné cíle jako například ROS nebo ADTF. [41]

Kapitola 3

Praktická část

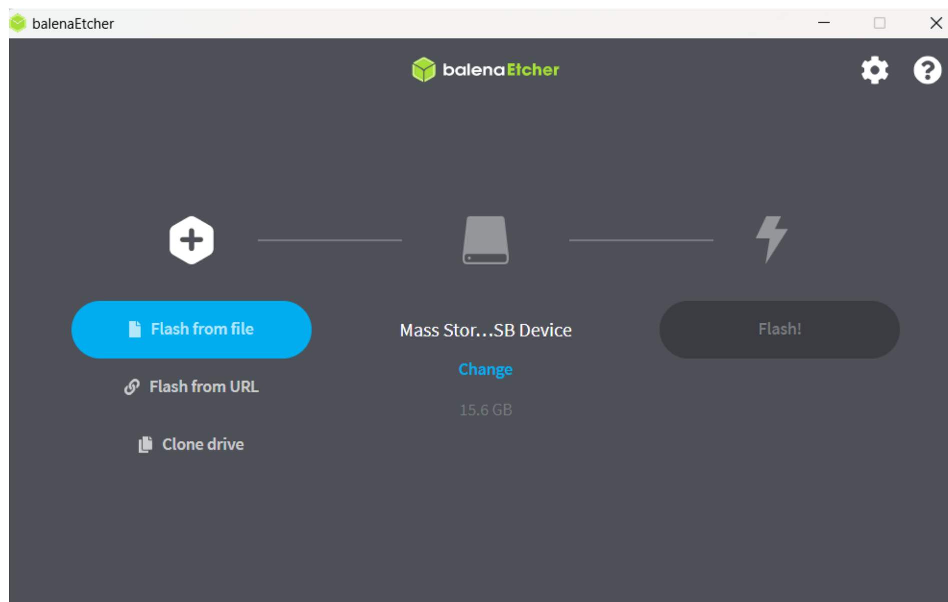
Praktická část této práce si klade za cíl implementovat jednu z již zmiňovaných Python knihoven pro úkol navigace mobilního robota. Robot, který bude použit pro tuto část je LEGO Mindstorms EV3. Robot EV3 je robotická sada od firmy LEGO vydána v roce 2013. Sada obsahuje mimo robotické kostky, která funguje jako centrální řídící systém robota, také servomotory a senzory jako např. Gyroskopický senzor, ultrazvukový senzor či infračervený senzor. Samotná kostka EV3 má následující technické parametry: procesor ARM9 o kmitočtu 300 MHz, Flash paměť 16 MB a paměť RAM 64 MB. Kostka také disponuje Bluetooth a USB 1.1 rozhraním a obsahuje slot pro microSD kartu, díky které lze do kostky nahrát dále zmíněný debian Linuxu. [42]

3.1 Příprava EV3 pro práci s Pythonem

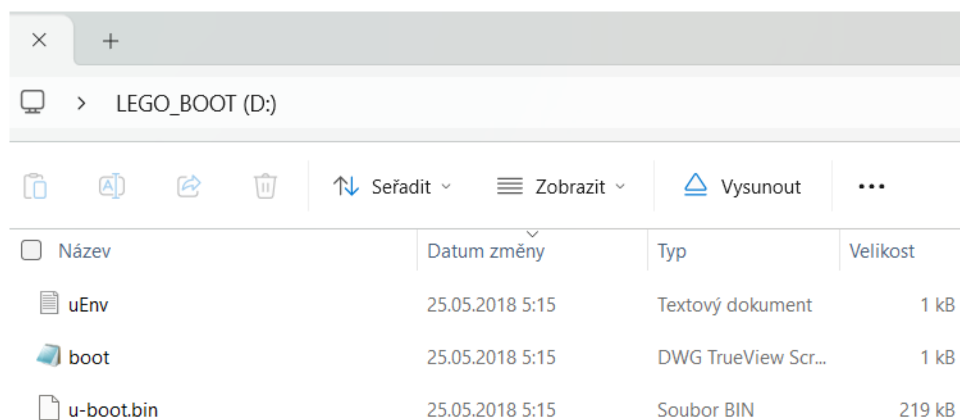
3.1.1 Nahrání debiánu Linuxu do EV3

Pro možnost práce s Pythonem přímo v robotické kostce LEGO MINDSTORM EV3 je potřeba provést určité kroky. Využiji oficiální stránky debianu www.ev3dev.org, kde stáhnu soubor (image), který poté budu muset "vypálit"na microSD kartu. Nejedná se sice o oficiální program od firmy LEGO, ale pro další práci se osvědčil nejlépe. [43]

Pro nahrání Pythonu do kostky je potřeba MicroSD karta, ne větší než 32 GB. V této praktické části byla využita karta o velikosti 16 GB. Na tuto kartu je poté potřeba "vypálit"soubor, který po vložení do kostky nabootuje sytém právě pro debián Linuxu. Využiji tedy například program Etcher, pomocí kterého dosáhnu požadovaného výsledku. MicroSD karta následně obsahuje soubory viz. Obrázek 3.2.



Obrázek 3.1: Program Etcher



Obrázek 3.2: Obsah microSD karty

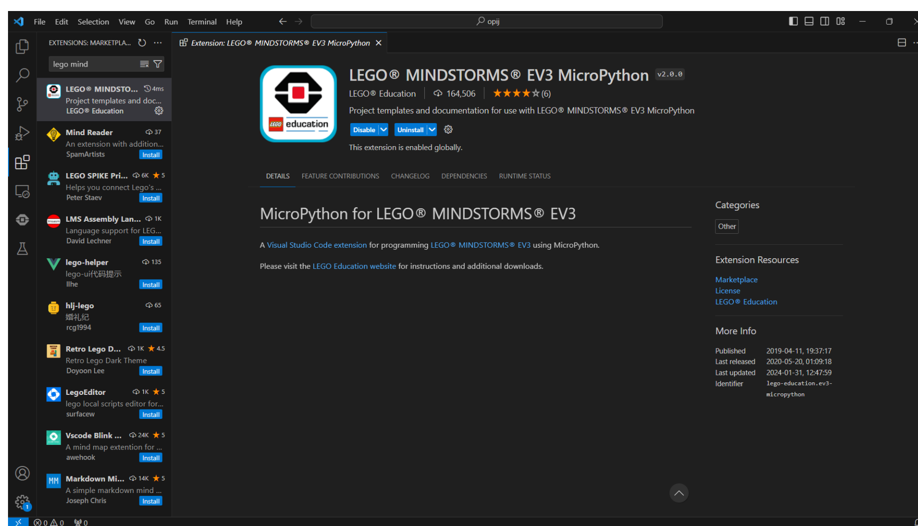
Obrázek 3.3: Bootování kostky EV3

Poté je vše připravené pro vložení karty do samotného robota LEGO. Karta se vloží do microSD slotu na levém boku kostky, když je robot vypnutý. Po zapnutí začne robot červeně blikat a začne proces bootování. Poté se robot zapne a lze už s ním dále pracovat.

3.1.2 Propojení kostky s PC pomocí Bluetooth

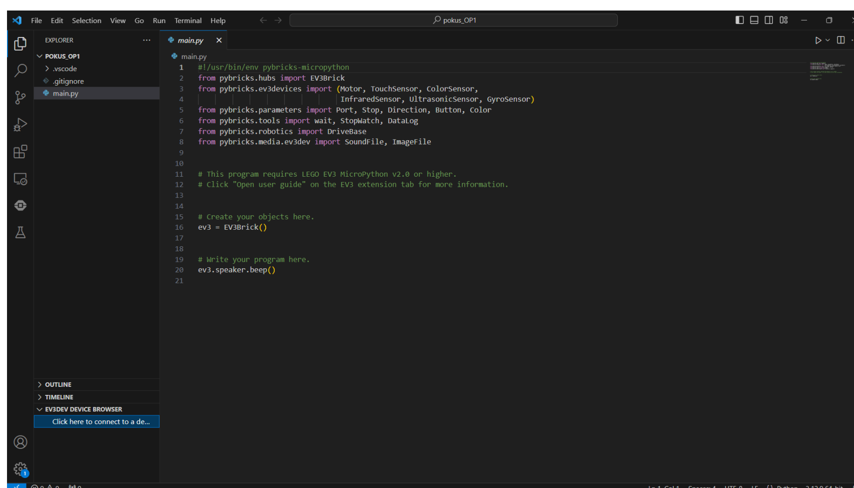
Aby bylo možné nahrávat a pouštět skripty přímo z PC, je nutné kostku připojit k PC. Lze tak učinit pomocí kabelu, pomocí Bluetooth anebo Wifi připojení. Tato práce využívá propojení kostky s PC za pomoci Bluetooth, neboť je to jednodušší než za pomoci Wifi a uživatelsky přívětivější než pomocí kabelu.

Pro práci se skripty je ale potřebný také nějaký PC program, ve kterém lze pracovat s Pythonem a s jeho knihovnami. Po zvažování pro a proti jsem se rozhodl pro práci s programem Visual Studio Code, který přesně vyhovuje mým požadavkům. Dále také musím nainstalovat rozšíření pro Python, který obsahuje příkazy přímo pro LEGO Mindstorms EV3 a také umožňuje spřažení kostky přímo s Visual Studio Code. Jedná se o doplněk, který se jmenuje LEGO® MINDSTORMS® EV3 MicroPython. Lze jej nainstalovat přímo v programu Visual Studio Code viz. Obrázek 3.4.



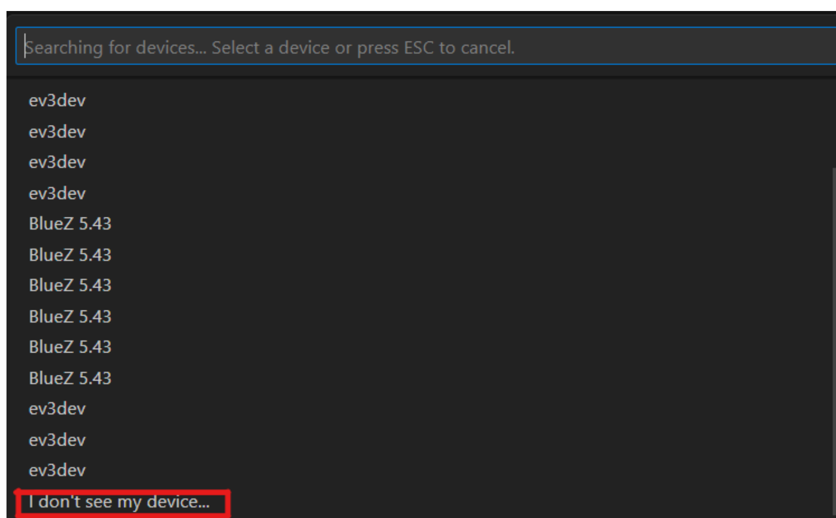
Obrázek 3.4: Instalace rozšíření pro Python

Po instalaci tohoto rozšíření už zbývá jen robota připojit. Stačí pouze rozkliknout ikonu rozšíření, zakliknout *"Create a new project"*, pojmenovat ho a dostávám se na obrazovku, kterou lze vidět na Obrázku 3.5.



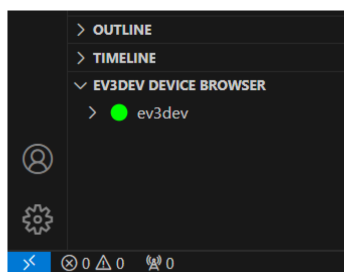
Obrázek 3.5: Nainstalované rozšíření pro Python

Dole na levé straně obrazovky viz. Obrázek 3.6 vidím nápis *"Click here to connect a device"*. Na tento nápis kliknu, nahoře se ukáže nabídka dostupných zařízení a dole je možnost přidat nové, pokud nevidím to své. Po zakliknutí možnosti přidat nové zařízení musím zadat jeho jméno, pod kterým je připojeno klasickým způsobem pomocí Bluetooth k PC a poté také jeho IP adresu, kterou lze odečíst přímo z displeje robota. [44]

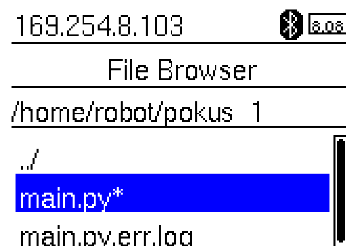


Obrázek 3.6: Hledání zařízení pro připojení

Po zadání jména a IP adresy se na stránce vlevo dole zobrazí jméno zařízení spolu se zelenou tečkou, která značí, že zařízení je připojené a je připravené k další práci.



Obrázek 3.7: Zařízení je připraveno k použití



Obrázek 3.8: Spouštění skriptu z kostky

Poté stačí nahrát skript do kostky pomocí tlačítka vedle jména zařízení. Skript lze spouštět jak z PC, tak i přímo z robotické kostky viz. Obrázek 3.8.

3.2 Koncepty konstrukce robota

Připravovaný robot má za úkol, pomocí předem dodaných robotických map projet terén. Proto je potřeba uvažovat konstrukci robota takovou, která bude pro tento úkol nejvhodnější. Robot provádí pouze tři pohyby - pohyb rovně, doprava a doleva. Každý pohyb je proveden jako jednotlivý krok a robot se při něm zastaví, je tedy nutné uvažovat, že se robot při rychlém zabrzdění rozkmitá. Proto je důležité robota zkonstruovat tak, aby byl stabilní i při těchto podmínkách. Dostatečné stability bude dosaženo, pokud třetí opěrný bod bude umístěn dostatečně daleko od hnané nápravy a pokud robot nebude vysoký tak, že při naklonění robota nedojde k převážení - je tedy nutné se pokusit dostat těžiště robota co nejníže.

Dále je podstatné dodržet rozměry robota. Jelikož se robot bude pohybovat po mřížkové mapě, kde každé pole má rozměr 100 x 100 mm, je nutné, aby se robot vešel celou svou styčnou plochou právě do této oblasti.

Po zvážení těchto faktů se dostávám k poznatku, že nejvhodnější konstrukcí bude použití diferenčně řízeného tříkolového podvozku. Na přední nápravě budou obě kola nezávisle řízená a jako třetí kolo bude použit omniwheel, který zaručí dostatečnou stabilitu a také hladké zatáčení robota. Hnanou nápravu je nejvhodnější osadit co nejmenšími a nejtenčími koly, aby se styk kola s podložkou mohl považovat za bodový. Co nejmenší průměr kol je vhodný k potlačení chyby, vzniklé nepřesným měřením natočení motorů. Ve finální konstrukci jsou tak použity kola s průměrem 30 mm, která jsou tenká 3 mm a rozchod mezi nimi je 90 mm.

Důležitými faktory jsou již zmiňovaný průměr a rozchod kol robota. Tyto faktory jsou důležité, neboť pomocí nich bude parametricky definován pohyb robota a bude tak zajištěno, že pouze díky znalosti rozchodu kol a jejich průměru bude přesně definován pojezd vpřed, doprava a doleva. Využiji totiž možnosti řídit motory definováním počtu otáček, o které se má motor otočit. Díky tomu jsem schopen celkem přesně definovat posuvný pohyb robota za pomoci rotační součástky - elektromotoru. Tohoto je možné dosáhnout díky tomu, že motor má v sobě zabudovaný senzor otáček - encoder, a lze řídit pomocí počtu otáček, o který se má motor otočit.

3.2.1 Konstrukce č.1

První koncept robota uvažuje kola umístěna dál od sebe, což je vhodnější pro stabilitu robota a robot tak může jet rychleji, viz. Obrázek 3.9 a 3.10. Jelikož jsou však kola dále od sebe, tak při pokročilejší fázi průjezdu trasou se může stát, že naakumulovaná chyba bude už tak velká, že robot bude koly vjíždět do překážek.

Obrázek 3.9: Pohled zepředu

Obrázek 3.10: Pohled zleva

3.2.2 Konstrukce č.2

Tento druhý koncept robota disponuje těžištěm umístěným mnohem výše než u prvního konceptu viz Obrázek 3.11 a 3.12. Zato je ale rozchod kol mnohem více příznivý pro zatáčení, což bude v rámci úkolu robota nejspíše výhodnější. Pro větší stabilitu třetího opěrného bodu robota byla použita masivnější konstrukce než u první verze konstrukce.

Obrázek 3.11: Pohled zepředu

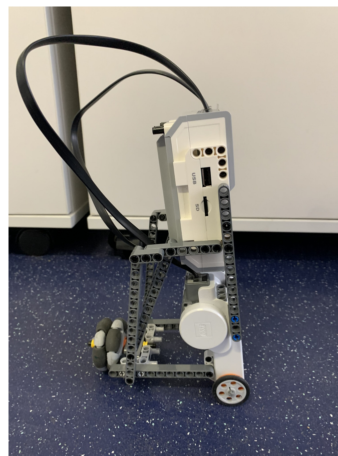
Obrázek 3.12: Pohled zleva

3.2.3 Konstrukce č.3

Třetí konstrukce robota je zobrazena na Obrázcích 3.13 a 3.14. Tato konstrukce je vybavena menšími koly o průměru 30 mm s rozchodem 90 mm. Jedná se o finální

koncept, který byl nakonec v této práci využit. Robot disponuje tuhou rámovou konstrukcí na zadní straně, na jejímž konci je uchycen omniwheel. Těžiště robota je umístěno výše než u konstrukce č. 1, ale díky omniwheelu umístěnému v dostatečné vzdálenosti slouží tato konstrukce dostatečně dobře.

Obrázek 3.13: Pohled zepředu



Obrázek 3.14: Pohled zleva

3.3 Lokalizace robota na mapě

Pro průjezd trasou je vhodné, aby robot věděl, na jakém místě v ní se vlastně nachází. Toho lze docílit několika možnostmi - lze využít senzory, díky kterým bude robot schopný vyhodnotit, na jakém místě se právě nachází. Stavebnice LEGO nabízí velkou spoustu senzorů, které by se takto daly použít - dotykový, ultrazvukový či infračervený. Pro exteriérové aplikace mobilních robotů se často využívá GPS lokalizace, to ale v případě LEGO stavebnice není možné. Dále je také možné využít možnost odometrie - při známém průměru kol a jejich rozchodu je pomocí odměřování úhlu natočení jednotlivých motorů možné dopočítat přesnou pozici robota na mapě. Tato metoda by byla nejvýhodnější pro tuto aplikaci. Jelikož ale stavebnice LEGO má velké vůle a odměřování natočení motorů také není na takové úrovni - jeho chyba se pohybuje mezi 2-6 stupni na otáčku, není tato metoda příliš přesná, a proto robota žádnou lokalizací nevybavím. Robot sice pojede podle odometrického výpočtu, nebude ale vybaven žádnou zpětnou vazbou o své pozici.

Pokud by měl být robot vybavený lokalizací v prostředí, dalo by se využít například kombinace odometrie s infračerveným či ultrazvukovým odměřováním vzdálenosti. Robot by jel podle odometrického výpočtu, ale po ujetí určitého počtu kroků by proběhlo měření vzdálenosti od překážek a porovnání naměřené vzdálenosti s odometricky spočtenou vzdáleností. Poté by se robot přesunul díky těmto korekcím na správnou pozici a mohl by pokračovat dále.

Případně by bylo možné použít Lidar, který by průběžně měřil vzdálenosti a pomocí regulátoru by byla jízda robota stále korigována. Pro mou potřebu je ale způsob řízení robota dostatečný, a proto se práce přesouvá k další části.

3.4 Programování robota

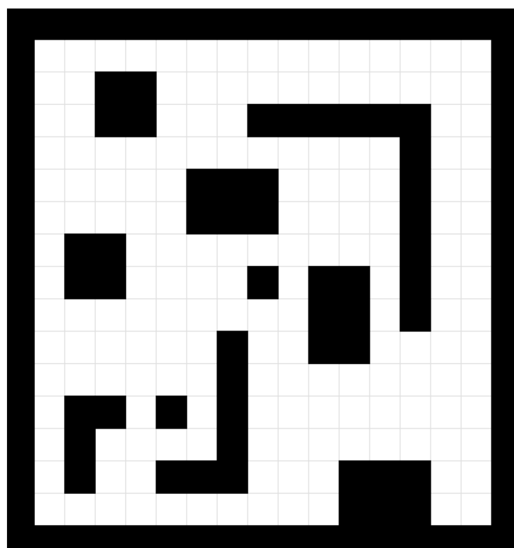
V další části této práce se přesouvám k samotnému programování robota. Cílem práce je využití již rešeršovaných knihoven pro Python pro řízení robota LEGO Mindstorms a vyzkoušení jednoduché robotické mapy - proto pracuji s mřížkovou mapou, která bude pro mé potřeby bohatě stačit.

Je potřeba najít vhodnou knihovnu, která bude srozumitelná a bude tak vhodná pro zkoumanou problematiku. Vhodné knihovny pro toto odvětví robotiky jsou např. KLAMPT, PythonRobotics a PyRobot. Po zvážení pro a proti volím knihovnu PythonRobotics, která obsahuje velké množství kódů právě pro hledání trasy v robotické mapě. Jedná se o velký adresář různých kódů, rozdělený do spousty podadresářů, kde každý obsahuje kódy pro určité odvětví robotiky. V adresáři pod názvem PathPlanning se nachází algoritmy pro hledání trasy v robotické mapě jako například algoritmy A*, D*, Dijkstrův algoritmus, RTT, vyhledávání do hloubky, vyhledávání do šířky a spousta dalších. Výhodou PythonRobotics je fakt, že tyto algoritmy se neschovávají pod názvem funkce, která musí být v programu zavolána jako například u KLAMPT a lze tak s nimi mnohem lépe pracovat. Každý z těchto kódů obsahuje samotný vyhledávací algoritmus a také definici robotické mapy. Mapa je definována pozicí startu, pozicí cíle, rozměry samotné mapy, pozicemi překážek a rádiusem robota. Se znalostí těchto informací lze sestavit mapu přesně podle potřeby. [5]

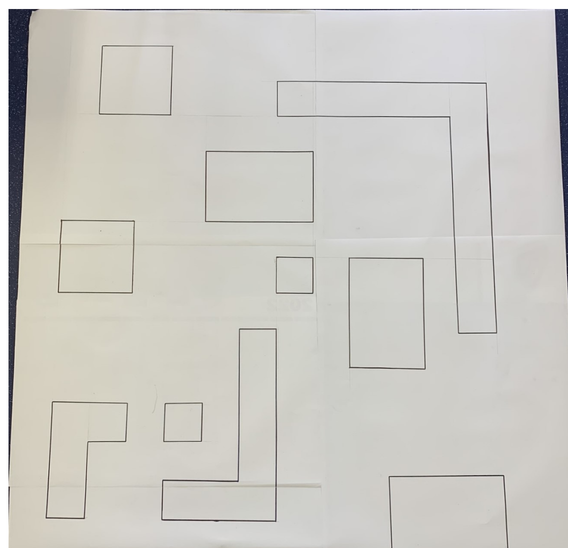
Výstupem této části kódu dostupného z knihovny PythonRobotics je nalezená cesta v mapě. Tímto končí práce s předem dodaným kódem a přesouvám se k vlastnímu programování

3.4.1 Použitá mapa

Původní mapa, kterou měl robot používat měla rozměry 1500 x 1500 mm. V následujících kapitolách však bude vysvětlen důvod proč robot nakonec využil jen mapu o rozměrech 1000 x 1000 mm. Jedná se o mřížkovou mapu, která je definována přímo v základním kódu, dostupném z PythonRobotics, pro tuto práci je ale modifikována tak, aby vyhovovala přesně mým potřebám. Startovní pozice robota je definována na souřadnici s_x = 50 mm a s_y = 50 mm, cíl má souřadnice g_x = 950 mm a g_y = 950 mm. Mapu definuji přímo v programu, který poběží v robotovi, a lze ji tak libovolně měnit. Mapa se skládá z jednotlivých polí, kde každé má rozměr 100 x 100 mm. Robot má délku kroku 100 mm, a tak by při každém kroku měl projet právě jedno políčko.



Obrázek 3.15: Mapa vytvořena v MS Excel



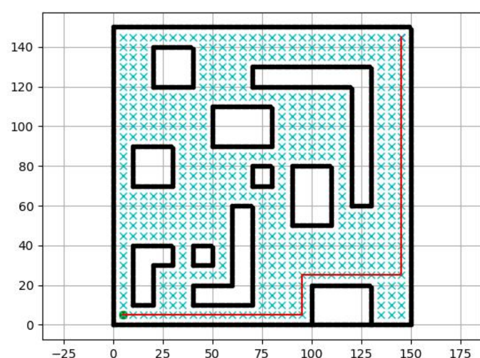
Obrázek 3.16: Fyzická mapa

3.4.2 Použité algoritmy

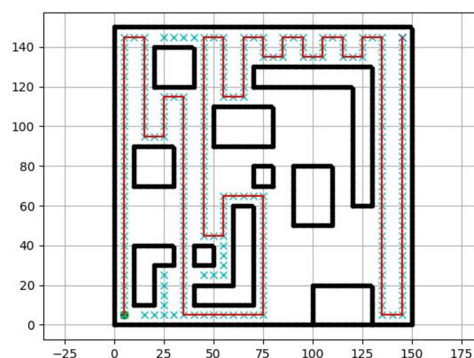
Pro praktickou část volím 4 různé algoritmy pro vyhledávání cesty v mřížkové mapě. Použit bude Dijkstrův algoritmus, A* algoritmus a algoritmy Depth-first search - prohledávání do hloubky a Breadth-first search - prohledávání do šířky. Všechny tyto algoritmy jsou dostupné z knihovny PythonRobotics. Pro každý algoritmus bude použita stejná robotická mapa. Právě tyto 4 algoritmy volím proto, že se využívají pro vyhledávání cesty v mapě a jsou vhodné pro použití v této práci. [5]

Je nutné zmínit, že pohyb robota je omezen pouze na pohyb doprava, doleva a rovně a neuvažuji žádný diagonální pohyb. Proto použitý algoritmus DFS generuje nejdelší a velmi nelogickou trasu, což je ale vlastnost samotného algoritmu. Pohyb do těchto směrů je omezen z důvodu nízkého výpočetního výkonu robota a také nebezpečí zacyklení algoritmu, které během pokusu naprogramovat jízdu v diagonálních směrech nastalo pokaždé. Zdrojové kódy všech 4 algoritmů nejprve upravím tak, aby ve všech byla nahrána mnou používaná mapa a upravím definici pohybu jen na jízdu doprava, doleva a rovně.

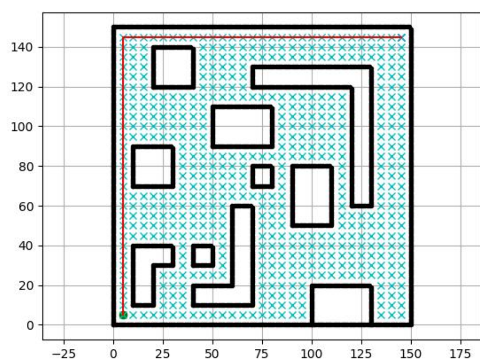
Na obrázcích níže (Obrázky 3.17 (a) - (d)) je vykreslena trasa, kterou vyhledaly jednotlivé algoritmy a kterou bude mít robot za cíl projet. Jak je zřejmé, v tomto případě jsou nalezené trasy všech algoritmů kromě DFS celkem jednoduché a intuitivní, ale navzájem se velmi liší. Důležité je si uvědomit, že každý algoritmus našel jinou cestu hlavně z toho důvodu, že není povolený diagonální pohyb. Při uvažování pohybu robota do všech 8 směrů, rozdíly mezi nalezenými trasami již nejsou tak markantní, viz. Obrázky 3.18 (a) - (d).



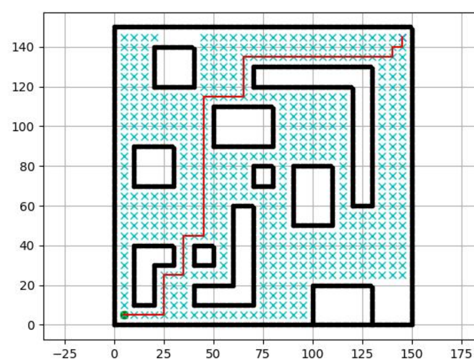
(a) Algoritmus BFS [5]



(b) Algoritmus DFS [5]

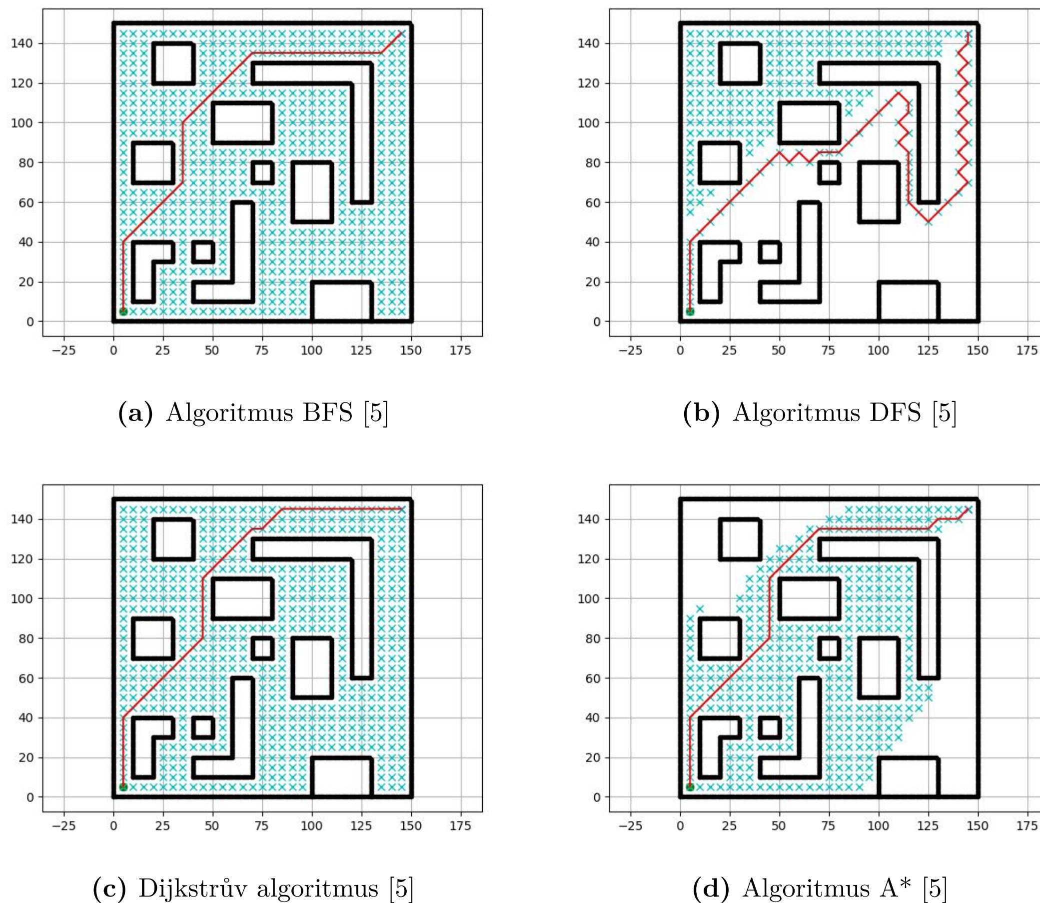


(c) Dijkstrův algoritmus [5]



(d) Algoritmus A* [5]

Obrázek 3.17: Algoritmy bez diagonálního pohybu po mřížce



(a) Algoritmus BFS [5]

(b) Algoritmus DFS [5]

(c) Dijkstrův algoritmus [5]

(d) Algoritmus A* [5]

Obrázek 3.18: Algoritmy s diagonálním pohybem po mřížce

3.4.3 Modifikace kódu

Nyní musím kód, který mám dostupný z Githubu upravit tak, aby byl schopný řídit robota. Toho dosáhnu zakomentováním některých částí kódu, zejména těch, které mají na starost tvorbu grafu. Původní kód také definuje cestu grafem v šikmém směru, což pro mou práci není potřeba, jelikož robot se bude pohybovat pouze rovně, doprava a doleva, a proto zakomentuji část kódu v rámci definice funkce:

```
def get_motion_model():
# dx, dy, cost
motion = [[1, 0, 1],
          [0, 1, 1],
          [-1, 0, 1],
          [0, -1, 1],
          #[-1, -1, math.sqrt(2)],
          #[-1, 1, math.sqrt(2)],
          #[1, -1, math.sqrt(2)],
          #[1, 1, math.sqrt(2)]
          ]
return motion
```

Zdrojový kód 3.1: Definice pohybu v grafu

Je potřeba zmínit, že při dalším programování můžu používat jen základní funkce Pythonu, neboť MicroPython, který běží v robotovi nepodporuje používání většiny dostupných knihoven pro Python z důvodu nízkého výpočetního výkonu. Je tak

důležité pracovat pouze s těmi knihovnami, které jsou dostupné. Příkladem tohoto problému může být použití knihovny Numpy, která je vhodná pro práci s maticemi, které používám v kódu. Naštěstí není kód příliš složitý, a tak si vystačím se zabudovanou knihovnou Math. [5]

Matici, kterou budu používat jako jakousi frontu, pojmenuji Q (Queue). Její vytvoření je jednoduché, stačí mi vytvořit matici z hodnot rx a ry. Nejprve definuji matici rx a ry pomocí příkazu:

```
1     rx_matrix = [item for item in rx]
```

Zdrojový kód 3.2: Vytvoření matice rx

To samé provedu pro matici ry a poté obě matice spojím do matice Q pomocí příkazu:

```
1     # Spojeni rx_matrix a ry_matrix do jedne matice Q
2    Q = []
3    for a, b in zip(rx_matrix, ry_matrix):
4        Q.append((a, b))
```

Zdrojový kód 3.3: Spojení matice r_x a r_y do matice Q

Toto mi zaručí uložení pozic, kterými má robot projít v sestupném směru, tj. od konečné pozice do startovní.

Nyní mám nalezenou cestu v robotické mapě a teď je na řadě robota rozhýbat. Proto kód, který jsem převzal z PythonRobotics musím upravit tak, aby byl funkční i s knihovnou MicroPython a tudíž s kostkou LEGO Mindstorms EV3. Jelikož robot nebude využívat žádné senzory, stačí mi importovat pouze část knihovny, která má na starost pohyb motorů. Proto na začátek kódu přidávám tyto řádky:

```
1 from ev3dev2.motor import MoveSteering, LargeMotor, OUTPUT_A,
    OUTPUT_D
```

Zdrojový kód 3.4: Import potřebných knihoven

Na samotný začátek skriptu je třeba přidat následující řádek, který zaručí, že skript bude správně zkompilován pro MicroPython běžící v robotovi.

```
1    #!/usr/bin/env python3
```

Zdrojový kód 3.5: Začátek programu

Teď je na čase robota rozhýbat. Jako první je potřeba vytvořit algoritmus, který bude procházet matici Q a určí, kterým směrem se má robot vydat, aby dojel do následujícího pole.

Nejprve je nutné zmínit, že algoritmus ukládá souřadnice cesty od konce do začátku, a tudíž budu pomocí příkazu *nynejsi_pozice = Q[-1]* vybírat prvky z této fronty od konce. Tento algoritmus je pak velmi jednoduchý. Pracuje tak, že porovnává x-ové a y-ové hodnoty z matice Q. Vyhodnocení probíhá tak, že algoritmus porovná sousední prvky v matici Q, poté první použitý prvek odstraní a prvek, který byl na druhém místě se označí za první a proces se opakuje.

Zároveň definuji 4 absolutní směry pohybu v grafu, které označím 0, 1, 2 a 3. Směr s číslem 3 značí pohyb robota v grafu směrem na sever, číslo 1 na jih, číslo 0 znamená pohyb směrem na východ a číslo 2 pohyb na západ. Toto celé je vloženo do while smyčky, která bude iterace provádět tak dlouho, dokud v matici Q nezbudou žádné

prvky. Původní směr robota je nastaven tak, aby robot směřoval na sever. Algoritmus tedy vypadá následovně:

```
smer = 3 # puvodni smer robota na sever
    nynejsi_pozice = Q[-1]
    nynejsi_pozice = Q.pop()  # Odstranuje prvni pouzity  prvek
   matice
    while len(Q) != 0:
        pristi_pozice = Q[-1]
        pristi_pozice = Q.pop()  # Odstrani prvni pouzity prvek
   matice
        novy_smer = smer

        if nynejsi_pozice[0] == pristi_pozice[0]:
            if pristi_pozice[1] > nynejsi_pozice[1]:
                novy_smer = 3 # pohyb smer sever
            elif nynejsi_pozice[1] > pristi_pozice[1]:
                novy_smer = 1 # pohyb smer jih
        elif nynejsi_pozice[1] == pristi_pozice[1]:
            if pristi_pozice[0] > nynejsi_pozice[0]:
                novy_smer = 0 # pohyb smer vychod
            else:
                novy_smer = 2 # pohyb smer zapad
```

Zdrojový kód 3.6: Definice absolutního pohybu robota

Nyní sice mám informace o absolutním pohybu robota po mřížce, ale ještě také musím definovat relativní pohyb, neboť robot se bude v mapě pohybovat rovně doprava a doleva, a tak potřebuji nástroj, který mi zaručí vždy správné zatočení robota do požadovaného směru. To zajistím přidáním následující části kódu do výše zmíněné while smyčky. Tato část algoritmu pracuje tak, že od sebe odečítá původní směr od směru nového a pro výsledné hodnoty tohoto výpočtu je buď definován pohyb doprava nebo doleva. Na konci této části kódu je přepsána hodnota *nynejsi_pozice* za pozici příští a algoritmus se díky vložené while smyčce opakuje.

Podmínka *if smer <0: smer = 3* a její obdoba v další části kódu zabezpečuje, že se hodnoty směrů pohybují v rozmezí 0 až 3, což jsou definované směry jízdy. Tuto podmínku je třeba dodat, neboť pomocí *smer -= 1* a *smer += 1* zvětšuji, resp. snižuji hodnotu proměnné *smer* a pokud by se dostal do nedefinované množiny čísel, kód by se zacyklil.

```
        while smer != novy_smer:
            novy_smer2 = smer - novy_smer
            if novy_smer2 == -3 or novy_smer2 == 1:
                print('Leva')
                robot.zatocit_doleva()
                smer -= 1
                if smer < 0: # podminka proti zacykleni
                    smer = 3
            else:
                print('Prava')
                robot.zatocit_doprava()
                smer += 1
                if smer > 3: # podminka proti zacykleni
                    smer = 0

        robot.rovne()
        print('Rovne')
```

```
18
19          nynejsi_pozice = pristi_pozice
```

Zdrojový kód 3.7: Definice relativního pohybu robota

Dále je v tomto kódu definováno, že při každém odbočení doprava robot vypíše na svůj displej "Prava", při odbočení doleva se na displeji zobrazí slovo "Leva"a při jízdě rovně se zobrazí slovo "Rovne".

V rámci výše zmíněného kódu jsou použity funkce *zatocit_doprava()*, *zatocit_-doleva()* a *rovne()*. Tyto funkce je potřeba na začátku programu definovat. Udělám tak vytvořením třídy *Rizeni*, kde definuji všechny tři funkce.

Nejprve si musím ujasnit, co od robota potřebuji. Využívám příkaz pro řízení obou motorů najednou, který dodefinuji tímto příkazem:

```
1     steer_pair = MoveSteering (OUTPUT_A, OUTPUT_D)
```

Zdrojový kód 3.8: Definice proměnné steer_pair

Tímto dosáhnu spřažení motorů dohromady a můžu tak definovat pohyb robota za pomoci jednoho řádku kódu. Celá definice třídy Rizeni bude tak vypadat takto:

```
1 #Definice tridy Rizeni
2 class Rizeni:
3     def __init__(self,steer_pair):
4         self.steer_pair = MoveSteering (OUTPUT_A, OUTPUT_D)
5
6     def zatocit_doprava(self):
7         steer_pair.on_for_rotations(steering=90, speed=15,
     rotations=pocet_otacek_2, brake=True, block=True) # Otoceni
     robota o 90 stupnu doprava
8
9     def zatocit_doleva(self):
10        steer_pair.on_for_rotations(steering=-90, speed=15,
     rotations=pocet_otacek_2, brake=True, block=True) # Otoceni
     robota o 90 stupnu doleva
11
12    def rovne(self):
13        steer_pair.on_for_rotations(steering=0, speed=-25,
     rotations=pocet_otacek, brake=True, block= True)  # jizda robota
      rovne o 100 mm
```

Zdrojový kód 3.9: Definice funkcí pro pohyb robota

Proměnné pocet_otacek a pocet_otacek2 jsou ještě před začátkem definice třídy Rizeni definovány takto:

```
1 #vypocet pro rizeni
2 prumer_kola_mm = 30
3 vzdalenost_mm = 100
4 obvod_kola_mm = math.pi * prumer_kola_mm
5 pocet_otacek = vzdalenost_mm / obvod_kola_mm # pocet otacek pro
     ujeti 100 mm
6 steer_pair = MoveSteering (OUTPUT_A, OUTPUT_D)
7 rozchod_kol_mm = 90
8
9 pocet_otacek_2 = (rozchod_kol_mm*math.pi/4)/obvod_kola_mm #pocet
     otacek pro zatoceni o 90 stupnu
```

Zdrojový kód 3.10: Odometrický výpočet pro řízení robota

3.5 Testování na robotovi

Po spuštění skriptů, které byly připraveny pro robota docházím ke zjištění, že robot sice trasou projíždí a cíl nachází, u algoritmů DFS a A* a Dijkstrova algoritmu nejede ale trasou, která byla vypočtena na PC ve verzi Pythonu 3.12.

Samotná jízda trasou je ale bez větších problémů. V pokročilejší fázi průjezdu robot vjíždí do vymezených překážek viz. Přílohy/B Videa/ 1,2,3,4. S touto skutečností ale bylo již od začátku počítáno, neboť vůle v elektromotorech robota jsou velmi velké, a také odměřování počtu otáček pomocí zabudovaného encoderu pracuje s jistou chybou.

Co se týče rozdílnosti generovaných tras mezi zařízeními, tak je třeba zjistit, čím je tato deviace způsobena.

Pozn.: Všechny skripty jsou umístěny v Přílohy/ A Zdrojové kódy

3.5.1 Zmenšení mapy

Jako první možnost vzniku této deviace jsem zkusil postupně zmenšovat mapu do té doby, než skript spuštěný v robotovi nalezne stejnou cestu, jako skript běžící na PC. Jelikož robot disponuje pamětí RAM pouze o velikosti 64 MB a jak bylo zmíněno v podkapitole 1.2.2 - výpočetní náročnost BFS i jiných algoritmů roste exponenciálně s narůstající hloubkou prohledávání, a tak při použití takto velké mapy docházelo k přepisování paměti v kostce. Nejprve jsem robota provozoval na mapě o rozměrech 1500 x 1500 mm. Poté jsem zkusil prostor, na kterém robot operuje zmenšit na rozměr 1000 x 1000 mm. Tato změna se ale téměř vůbec neprojevila, a tak byla použita mapa o rozměrech 800 x 800 mm. Tato změna sice úplné vyřešení problému nepřinesla, ale na rozdíl od původní velké mapy jel robot po trase, která se více podobala té vygenerované na PC a robot už alespoň dojel do cíle. Obě trasy disponovaly stejným počtem kroků a lišily se jen například v tom, že skript, který běžel v MicroPythonu zatáčel dříve doleva než skript z PC.

Poslední zkoumaná mapa měla rozměr 500 x 500 mm. V této mapě se trasy nalezené robotem a na PC shodovaly. Je potřeba ale zmínit, že na těchto rozměrech se jednalo o velmi jednodušší mapu než předešlé dvě.

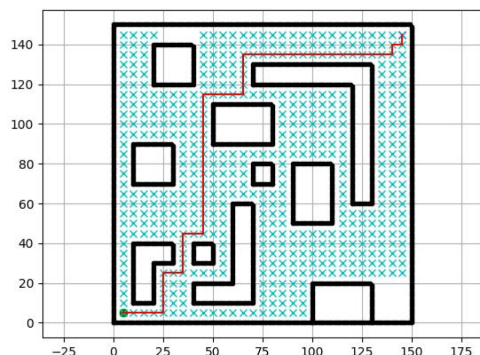
Dá se tedy usoudit, že problém může být zakořeněný ve výpočetních schopnostech robota, neboť s menším rozlišením mapy se jízda robota stále více blíží té trase, kterou generuje skript běžící na PC.

3.5.2 Zjednodušení mapy

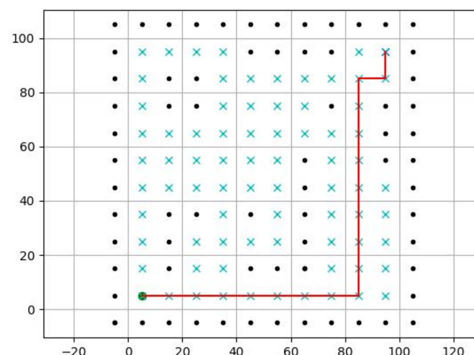
Další možností, která byla vyzkoušena je zjednodušení reprezentace mapy pro robota. V předešlých testech byly překážky na mapě definovány ve formě bodů, které byly umístěny téměř spojitě vedle sebe a tvořily tak hranice překážek. Dále algoritmus pracoval s velikostí mřížky mapy 50 mm, což zabíralo ještě více výpočetního výkonu.

Pro zjednodušení mapy byla proto využita nová velikost mřížky mapy - tentokrát 100 mm. Dále byly také překážky předefinovány pomocí svých středů. V rámci vykreslení

grafů je tato úprava méně přehledná, ale pro snazší průběh kódu je velmi výhodné tuto změnu provést. Změna je ilustrována na Obr. 3.19 a 3.20



Obrázek 3.19: Původní stav



Obrázek 3.20: Zjednodušená mapa

Po tomto zjednodušení však k žádnému výraznému zjednodušení v mapě o rozměrech 1500 x 1500 mm nedošlo. Po pokusu s mapou 1000 x 1000 mm se však algoritmus začal chovat mnohem lépe. Trasa, kterou algoritmus v robotovi nalezl stále sice není naprosto totožná s verzí trasy z PC. Jedná se ale o stejný případ jako v případě mapy o rozměrech 800 x 800 mm před zjednodušením - robot už nalézá bez problému cíl a všem překážkám se bez problémů vyhýbá.

3.5.3 32-bit x 64-bit

Jelikož robot LEGO EV3 má 32bitovou architekturu procesoru a PC, na kterém proběhlo programování skriptu má 64 bitů, rozhodl jsem se zjistit, zdali není problém způsoben touto odlišností. Pomocí programu Oracle VM VirtualBox jsem proto nainstaloval na PC virtuální 32 bitovou verzi Linuxu - MX Linux. V tomto prostředí jsem spustil stejný skript, který používám v robotovi i na PC a vyhodnocená trasa byla totožná s tou, kterou mi vrací PC verze Pythonu. Lze tedy usoudit, že odlišnost mezi 32 a 64bitovou verzí nehraje v tomto případě roli.

Pozn.: MX Linux dostupné z: https://mxlinux.org/

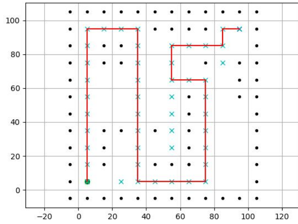
3.5.4 Debugování kódu

Po výše sepsaných pokusech jsem se pokusil skripty převzaté z PythonRobotics projít krok po kroku a zjistit, zdali neobsahují vadné části kódu. Algoritmus byl vždy spuštěn jak na PC, tak v robotovi. Dostal jsem se ke zjištění, že ve skriptu na řádku 91 je použita datová struktura *dict()* pro proměnné *open_set* a *closed_set*, která je definována uložením v paměti robota pod určitou adresou. Proměnné *open_set* i *closed_set* jsou nejprve definovány jako prázdná množina typu *dict()*. Poté je pomocí funkce *calc_grid_index* do proměnné *open_set* definována proměnná vztahující se na startovní pozici robota.

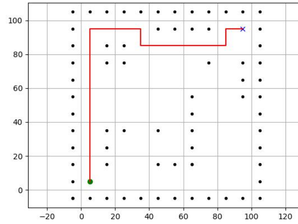
```
1 open_set, closed_set = dict(), dict()
```

Zdrojový kód 3.11: Definice proměnných `open_set` a `closed_set`

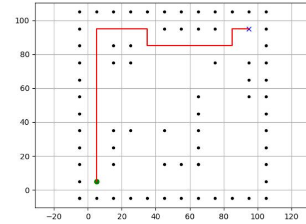
Datová struktura *dict()* je však definována tím, že není uspořádána a její uspořádání se může lišit i mezi verzemi Pythonu, tudíž nemusí být chyba v MicroPythonu. Proto jsem stejný skript, který pouštím na PC v Pythonu 3.12, spustil v Pythonu 3.6 a trasa, kterou tato verze Pythonu nalezla se shoduje s trasou, kterou generuje samotný MicroPython v robotovi.



Obrázek 3.21: Trasa nalezená v Pythonu 3.12



Obrázek 3.22: Trasa nalezená v Pythonu 3.6



Obrázek 3.23: Trasa nalezená v MicroPythonu

Na Obrázcích 3.21 a 3.22 je znázorněna nalezená trasa naprosto totožným algoritmem, jen v rozdílných verzích Pythonu. Po porovnání Obrázku 3.22 s Obrázkem 3.23 je patrné, že se cesta nalezená v Pythonu 3.6 a MicroPythonu shoduje.

Podle [45] je totiž od verze Pythonu 3.7 implementováno automatické řazení datového typu slovník. MicroPython ale podle [46] odpovídá verzi Pythonu 3.4 a disponuje některými vlastnostmi Pythonu 3.5, a tak není tato funkce implementována, a proto nachází Python 3.12 na PC a MicroPython v robotovi rozdílnou trasu. Docházím tak k závěru, že rozdílné chování algoritmů mezi jednotlivými verzemi, potažmo platformami, má za vinu implementace datového typu dict(). Robot ale v obou případech cestu do cíle najde, a tak je tato deviace pouze otázkou délky průjezdu robota mapou.

Závěr

Cílem této práce bylo prozkoumat možnosti adaptace algoritmů pro řízení robota na základě apriorních informací. Byla provedena rešerše nejběžněji používaných robotických map a algoritmů pro vyhledávání cesty v těchto mapách. Byla také provedena rešerše zaměřená na použití programovacího jazyka Python v robotice, se zvláštním důrazem na dostupné knihovny pro robotické aplikace.

V praktické části byl sestrojen robot ze stavebnice LEGO Mindstorms EV3, který byl programován v Pythonu. Byla vytvořena jednoduchá mřížková mapa o rozměrech 1500 x 1500 mm, která byla robotu předem dodána. Pro praktickou část byly vybrány 4 algoritmy z knihovny PythonRobotics: Dijkstrův algoritmus, A* algoritmus a BFS a DFS algoritmus. Skript spuštěný v robotovi byl poté schopen nalézt trasu v této mapě a robot tak mohl dojet do cíle pouze na základě apriorně dodaných informací o poloze startu, cíle a překážkách. Robot však následoval odlišnou trasu než tu, kterou generoval skript v Pythonu 3.12 na PC. Odchylka byla způsobena odlišností verzí Pythonu 3.12 a MicroPythonu.

Další vývoj v tomto tématu může směřovat k implementaci jiných HW platforem jako například Arduino, NVIDIA Jetson, Raspberry Pi a jiných mikrokontrolerů či SBC. Dále je vhodné prozkoumat i využití složitějších knihoven a algoritmů jako např. SLAM.

Bibliografie

1. *02_Reseni_problemu.pdf*. 2004. Dostupné také z: https://is.muni.cz/el/fi/podzim2006/PA161/um/02_Reseni_problemu.pdf.
2. MILFORD, Michael; SCHULZ, Ruth. Principles of goal-directed spatial robot navigation in biomimetic models. *Philosophical Transactions of the Royal Society B: Biological Sciences*. 2014, roč. 369, č. 1655, s. 20130484. Dostupné z DOI: 10.1098/rstb.2013.0484.
3. AL-SAGBAN, Madam; DHAOUADI, Rached. Reactive navigation algorithm for wheeled mobile robots under non-holonomic constraints. In: *2011 IEEE International Conference on Mechatronics*. 2011, s. 504–509. Dostupné z DOI: 10.1109/ICMECH.2011.5971338.
4. MACHEK, Ondřej. *Fuzzy modely map pro pohyb mobilních robotů*. Brno, 2011. Dostupné také z: http://hdl.handle.net/11012/1412.
5. SAKAI, Atsushi; INGRAM, Daniel; DINIUS, Joseph; CHAWLA, Karan; RAFFIN, Antonin; PAQUES, Alexis. *PythonRobotics: a Python code collection of robotics algorithms*. 2018. Dostupné z arXiv: 1808.10703 [cs.RO].
6. STĚPÁN, Petr. *Mobilní robotika ČVUT. Modely prostředí II.* 2008. Dostupné také z: https://cw.fel.cvut.cz/old/_media/mkr/lessons/mapovani-ii.pdf.
7. MATOUŠEK, Jiří. *Topology* [https://kam.mff.cuni.cz/Matematika++/topo.pdf]. 2014. A chapter for the Mathematics++ Lecture Notes, Rev. 15/XI/2014 JM.
8. KURIC, Ivan; BULEJ, Vladimir; SAGA, Milan; POKORNY, Peter. Development of simulation software for mobile robot path planning within multilayer map system based on metric and topological maps. *International Journal of Advanced Robotic Systems*. 2017, roč. 14, č. 6, s. 1729881417743029. Dostupné z DOI: 10.1177/1729881417743029.
9. XU, Song; ZHOU, Huaidong; CHOU, Wusheng. Visual Topological Mapping and Navigation for Mobile Robot in Large-Scale Environment. In: *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2019, s. 2589–2594. Dostupné z DOI: 10.1109/ROBIO49542.2019.8961726.
10. WINKLER, Zbyněk. *Plánování na mřížce (Robotika.cz > Průvodce)*. 2003. Dostupné také z: https://robotika.cz/guide/gridplan/cs.
11. ČIKEŠ, Mijo; ĐAKULOVIĆ, Marija; PETROVIĆ, Ivan. The path planning algorithms for a mobile robot based on the occupancy grid map of the environment — A comparative study. In: *2011 XXIII International Symposium on Information, Communication and Automation Technologies*. 2011, s. 1–8. Dostupné z DOI: 10.1109/ICAT.2011.6102088.

12. BRÄUNL, Thomas. Navigation. In: *Robot Adventures in Python and C*. Cham: Springer International Publishing, 2020, s. 109–123. ISBN 978-3-030-38897-3. Dostupné z DOI: 10.1007/978-3-030-38897-3_10.

13. BARRAQUAND, J.; LANGLOIS, B.; LATOMBE, J.-C. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, and Cybernetics*. 1992, roč. 22, č. 2, s. 224–241. Dostupné z DOI: 10.1109/21.148426.

14. XING, Hongbo; CHAI, Mengyao; SONG, Yaju. Artificial intelligence pathfinding based on Unreal Engine 5 hexagonal grid map. In: *2024 4th International Conference on Neural Networks, Information and Communication (NNICE)*. 2024, s. 1708–1711. Dostupné z DOI: 10.1109/NNICE61279.2024.10498463.

15. KUBALÍK, J. *Základy umělé inteligence ,2. Informované metody prohledávání stavového prostoru* [Stránky předmětu]. [B.r.]. https://cw.fel.cvut.cz/old/_media/courses/y33zui/informedsearchmethods_2010.pdf.

16. BLÁHA, Milan. *Matematická biologie*. [B.r.]. Dostupné také z: https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence--prohledavani-stavoveho-prostoru--metody-prohledavani--neinformovane-prohledavani--metoda-prohledavani-do-sirky-breadth-first-search-bfs.

17. BLÁHA, Milan. *Matematická biologie*. [B.r.]. Dostupné také z: https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence--prohledavani-stavoveho-prostoru--metody-prohledavani--neinformovane-prohledavani--metoda-prohledavani-do-hloubky-depth-first-search-dfs.

18. PALANISAMY, Vigneshwaran; VIJAYANATHAN, Senthooran. A Novel Agent Based Depth First Search Algorithm. In: *2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA)*. 2020, s. 443–448. Dostupné z DOI: 10.1109/ICCCA49541.2020.9250826.

19. DHOUIB, Souhail. Shortest path planning via the rapid Dhouib-Matrix-SPP (DM-SPP) method for the autonomous mobile robot. *Results in Control and Optimization*. 2023, roč. 13, s. 100299. ISSN 2666-7207. Dostupné z DOI: https://doi.org/10.1016/j.rico.2023.100299.

20. DLOUHÝ, Martin. *Exaktní plánování (Robotika.cz > Průvodce)*. 2003. Dostupné také z: https://robotika.cz/guide/exactplan/cs.

21. GARRIDO, Santiago; ABDERRAHIM, Mohamed; MORENO, Luis. PATH PLANNING AND NAVIGATION USING VORONOI DIAGRAM AND FAST MARCHING. *IFAC Proceedings Volumes*. 2006, roč. 39, č. 15, s. 346–351. ISSN 1474-6670. Dostupné z DOI: https://doi.org/10.3182/20060906-3-IT-2910.00059. 8th IFAC Symposium on Robot Control.

22. HOLOUBEK, Tomáš. *Plánování cesty mobilního robotu pomocí celulárních automatů*. 2020.

23. BLÁHA, Milan. *Matematická biologie*. [B.r.]. Dostupné také z: https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence--prohledavani-stavoveho-prostoru--metody-prohledavani--informovane-heuristicke-prohledavani--metoda-a.

24. GAY, Warren. *Raspberry Pi Hardware Reference*. 2014. ISBN 978-1-4842-0800-7. Dostupné z DOI: 10.1007/978-1-4842-0799-4.
25. WEST, Luke. *THE RISE OF THE RASPBERRY PI IN INDUSTRIAL SETTINGS*. [B.r.]. Dostupné také z: https://www.rowse.co.uk/blog/post/the-rise-of-the-raspberry-pi-in-industrial-settings.
26. LEÓN ARAUJO, Hernando; GULFO AGUDELO, Jesús; CRAWFORD VIDAL, Richard; ARDILA URIBE, Jorge; REMOLINA, John Freddy; SERPA-IMBETT, Claudia; LÓPEZ, Ana Milena; PATIÑO GUEVARA, Diego. Autonomous Mobile Robot Implemented in LEGO EV3 Integrated with Raspberry Pi to Use Android-Based Vision Control Algorithms for Human-Machine Interaction. *Machines*. 2022, roč. 10, č. 3. ISSN 2075-1702. Dostupné z DOI: 10.3390/machines10030193.
27. PLAZA, Pedro; SANCRISTOBAL, Elio; CARRO, German; BLAZQUEZ, Manuel; GARCÍA-LORO, Félix; MARTIN, Sergio; PEREZ, Clara; CASTRO, Manuel. Arduino as an Educational Tool to Introduce Robotics. In: *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. 2018, s. 1–8. Dostupné z DOI: 10.1109/TALE.2018.8615143.
28. CANDIDO, Renato. *Arduino With Python: How to Get Started*. 2022. Dostupné také z: https://realpython.com/arduino-python/.
29. SUMMERFIELD, Mark. *Python 3: výukový kurz*. 2. vyd. Přel. KREJČÍ, Lukáš. Brno: Computer Press, 2021. ISBN 978-80-251-5030-6.
30. HAUSER, Kris. *Krishauser/Klampt: Kris' locomotion and manipulation planning toolkit*. 2013. Dostupné také z: https://github.com/krishauser/Klampt.
31. *Klampt Python API*. 2023. Dostupné také z: https://motion.cs.illinois.edu/software/klampt/latest/pyklampt_docs/.
32. ADEMOVIC, Adnan. *An Introduction to Robot Operating System: The Ultimate Robot Application Framework*. 2010. Dostupné také z: https://www.toptal.com/robotics/introduction-to-robot-operating-system.
33. *ROS.org*. [B.r.]. Dostupné také z: https://wiki.ros.org/ROS/Introduction.
34. MURALI, Adithyavairavan; CHEN, Tao; ALWALA, Kalyan Vasudev; GANDHI, Dhiraj; PINTO, Lerrel; GUPTA, Saurabh; GUPTA, Abhinav. PyRobot: An Open-source Robotics Framework for Research and Benchmarking. *arXiv preprint arXiv:1906.08236*. 2019.
35. REDDY, A. DIFFERENCE BETWEEN DENAVIT - HARTENBERG (D-H) CLASSICAL AND MODIFIED CONVENTIONS FOR FORWARD KINEMATICS OF ROBOTS WITH CASE STUDY. In: 2014. Dostupné z DOI: 10.13140/2.1.2012.9607.
36. NADEAU, Nicholas A.; BONEV, Ilian A.; JOUBAIR, Ahmed. Impedance Control Self-Calibration of a Collaborative Robot Using Kinematic Coupling. *Robotics*. 2019, roč. 8, č. 2, s. 33. ISSN 2218-6581. Dostupné z DOI: 10.3390/robotics8020033.
37. NADEAU, Nicholas. Pybotics: Python Toolbox for Robotics. *Journal of Open Source Software*. 2019, roč. 4, č. 41, s. 1738. Dostupné z DOI: 10.21105/joss.01738.

38. RAGHUNATHAN, Arvind; JHA, Devesh K.; ROMERES, Diego. PYROBOCOP: Python-based Robotic Control & Optimization Package for Manipulation. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2022. ISBN 978-1-7281-9681-7. Dostupné z DOI: 10.1109/ICRA46639.2022.9812069.

39. RAGHUNATHAN, Arvind U; JHA, Devesh K; ROMERES, Diego. PYROBOCOP: Python-based robotic control & optimization package for manipulation and collision avoidance. *arXiv preprint arXiv:2106.03220*. 2021.

40. WIERENGA, Rick P.; GOLAS, Stefan M.; HO, Wilson; COLEY, Connor W.; ESVELT, Kevin M. PyLabRobot: An open-source, hardware-agnostic interface for liquid-handling robots and accessories. *Device*. 2023, roč. 1, č. 4, s. 100111. ISSN 2666-9986. Dostupné z DOI: https://doi.org/10.1016/j.device.2023.100111.

41. ROBOTIKA.CZ. *Osgar* [online]. 2023. [cit. 2024-05-05]. Dostupné z: https://github.com/robotika/osgar.

42. *Lego MINDSTORMS EV3 User Guide*. 2013. Dostupné také z: https://web.archive.org/web/20150911060701/http://cache.lego.com/r/www/r/mindstorms/-/media/franchises/mindstorms%5C%202014/downloads/user%5C%20guides/user%5C%20guide%5C%20lego%5C%20mindstorms%5C%20ev3%5C%2010%5C%20all%5C%20enus%5C%20(2).pdf?l.r2=-329554550.

43. *Getting Started with ev3dev*. Dostupné také z: http://www.ev3dev.org/docs/getting-started/.

44. *Connecting to the Internet via USB*. Dostupné také z: https://www.ev3dev.org/docs/tutorials/connecting-to-the-internet-via-bluetooth/.

45. TERCZYNSKI, Brian. *A Warning About Sorting Dictionaries and Python Versions*. 2020. Dostupné také z: https://bterczynski.medium.com/a-warning-about-sorting-dictionaries-and-python-versions-a285b7fef084.

46. GEORGE, Damien P.; SOKOLOVSKY, Paul. *MicroPython differences from CPython*. 2014. Dostupné také z: https://docs.micropython.org/en/latest/genrst/index.html.

Seznam použitého SW

1. Balena Etcher
2. ev3dev (debian Linuxu pro EV3)
3. LEGO® MINDSTORMS® EV3 MicroPython (rozšíření pro VS Code)
4. Visual Studio Code
5. MX Linux
6. Oracle VM Virtual Box

Přílohy

A Zdrojové kódy

1. a_star_do_robota.py
2. dijkstra_do_robota.py
3. BFS_do_robota.py
4. DFS_do_robota.py

B Videa

1. A_STAR.mp4
2. DIJKSTRA.mp4
3. BFS.mp4
4. DFS.mp4