UNIVERZITA PALACKÉHO V OLOMOUCI

PEDAGOGICKÁ FAKULTA

Katedra technické a informační výchovy

Bakalářská práce

Radek Matoušek

Realizace základních algoritmů v programovacím jazyce Python s využitím stavebnice LEGO SPIKE Prime

Olomouc 2023

vedoucí práce: Mgr. Radim Děrda

Prohlášení

Prohlašuji, že jsem bakalářskou práci na téma *„Realizace základních algoritmů v programovacím jazyce Python s využitím stavebnice LEGO SPIKE Prime“* vypracoval samostatně a uvedl všechny použité literární a odborné zdroje, které uvádím v seznamu použitých zdrojů.

V Olomouci dne 20. června 2023 ……..

Radek Matoušek

Poděkování

Tímto bych rád poděkoval vedoucímu mé bakalářské práce Mgr. Radimu Děrdovi, za odborné vedení mé bakalářské práce, množství cenných a inspirativních rad, podnětů, doporučení, připomínek a zároveň za velkou trpělivost a ochotou při konzultacích poskytnutých ke zpracování této práce.

OBSAH

ÚVOD A CÍL BAKALÁŘSKÉ PRÁCE

V mé bakalářské práci se budu zabývat tématem realizace základních algoritmů v programovacím jazyce Python s využitím stavebnice LEGO SPIKE Prime. Pro lepší přehlednost svoji práci rozdělím na teoretickou a praktickou část.

Cílem teoretické části bude popis programovacího jazyka python, jeho historie a výhody plynoucí z jeho použití. Jako další se chci zabývat vysvětlením algoritmů a jejich použitím. Dále popíši v bakalářské práci stavebnicí lego, kterou chci v praktické části použít pro stavbu soutěžního robota určeného k průchodu čárového bludiště a srovnám ji s dalšími nejpoužívanějšími robotickými stavebnicemi. Stavba soutěžního robota musí splňovat pravidla soutěže. Soutěže tohoto typu se konají po celém světě.

V praktické části se chci zabývat samotným sestavením soutěžního robota ze stavebnice LEGO SPIKE Prime při použití pouze základní sady stavebnice. Po sestavení robota bych se chtěl zabývat naprogramováním jednotlivých funkcí pro ovládání robota a pro splnění úkolu spočívajícího ve vyhledání nejkratší cesty v čárovém bludišti
a následném projetí nalezené cesty.

I. TEORETICKÁ ČÁST

PROGRAMOVACÍ JAZYK PYTHON

Python je vysokoúrovňový programovací jazyk, který klade důraz na dobrou čitelnost kódu a intuitivnost. Je to dynamický jazyk, který automaticky maže nepotřebná data. V pythonu je možné psát malé i rozsáhlé projekty. Python podporuje programovací paradigmata, jako objektově orientovaná, strukturovaná a funkcionální. Python je velmi jednoduchý jazyk na naučení díky své syntaxi. Lze jej také velmi snadno rozšířit o nové datové typy a funkce, které byly implementovány v jiných programovacích jazycích jako je C++ a C. (Summerfield, 2021)

Paradigmata objektově orientovaná

Objektově orientované paradigma vytváří několik konstrukcí, pomocí kterých provádí abstrakci spolupracujících objektů.

- Objekt – je odvozený od nějakého vzoru. Podle tohoto vzoru také přebere své atributy a je přístupný pro volání.

- Třída – Pomocí třídy jsou objekty stejného typu spojeny a zároveň nepředává žádná svoje data (Pecinovský, 2020).

Paradigmata strukturovaná

Ve strukturovaném paradigmatu je problém rozdělen na několik pod problémů a pro každý podproblém je vytvořena funkce s parametry, která jej řeší, hovoříme tedy o takzvané funkcionální dekompozici. Ve strukturovaném paradigmatu se předpokládá že nemůžeme použít stejnou, nebo lehce modifikovanou funkci pro řešení dvou problémů (Pecinovský, 2020).

Paradigmata funkcionální

Funkcionální paradigma je založeno na postupném volání funkcí. V tomto paradigmatu se často setkáváme s rekurzí. Slovo rekurze pochází z latinského recurso – vrátit se. Rekurze v programování znamená, že v průběhu funkce zavolá funkce sama sebe. Jazyky, které používají funkcionální paradigma, vycházejí z lambda kalkulu. Programovací jazyky využívající funkcionální paradigmata se používají pro vývoj umělé inteligence, modelování nebo finanční analýzu (Pecinovský, 2020).

Historie Pythonu

Programovací jazyk Python byl vydán 20. února 1991 ve verzi Python 0.9.0. Od tohoto data do současnosti byly vydány celkem tři velké verze, které byly vzájemně nekompatibilní. Poslední verze Pythonu 1.0 byla vypuštěna v roce 2000, v tomto roce byla také vydána nová velká verze Pythonu s číslovkou 2.0. Tato verze skončila svůj vývoj 20. dubna 2020 s vydáním verze 2.7.18., ale na rozdíl od přechodu mezi verzí 1.0 a 2.0, nová verze 3.0 nebyla publikována ve stejném roce ve kterém se Python 2.0 přestal podporovat, ale byl vydán už v roce 2008. Bylo tu tedy dvanáctileté období, kdy byla podporována stará i nová verze pythonu (Python Software Foundation, 2001).

Za vznikem tohoto jazyku stál Nizozemský programátor Guido van Rossum, který jej vytvořil jako nástupce programovacího jazyka ABC. Guido van Rossum začal Python programovat v roce 1989 jako volnočasovou aktivitu. Hledal totiž nějaký koníček a už delší dobu měl myšlenku vytvořit nástupce jazyka ABC. Jednou z hlavních myšlenek pro směřování tohoto programovacího jazyka bylo vytvořit programovací jazyk, který by byl intuitivní, dobře se v něm prováděla výuka nováčků a funkčností by mohl konkurovat ostatním jazykům jako je C a Java. A bylo z něj lehké na tyto jazyky přejít (Python Software Foundation, 2001).

Pro splnění těchto požadavků Guido van Rossum využil svých zkušeností získaných při práci v Centrum voor Wiskunde en Informatica, kde pracoval se zmíněným jazykem ABC. Jednoho dne si začal tvořit svoji verzi tohoto jazyku, kde přejímal jen věci, které se mu líbili a implementoval je bez běžných problémů, které měl jazyk ABC. Také změnil syntax, kdy místo špičatých závorek a begin-end bloku, se příkazy seskupovaly do bloku pomocí odsazení. Tímto krokem zlepšil čitelnost a pochopitelnost jazyků pro nové programátory. Guido van Rossum brzo docílil vytvoření nového plně funkčního programovacího jazyku, kterému scházelo jen jméno. Přál si, aby jméno jeho jazyku bylo unikátní a krátké. Pojmenovala ho tedy Python. Ač si většina myslí že jazyk byl pojmenován podle hada, není tomu pravda. Jméno Python bylo zvoleno podle Monty Python's Flying Circus, komedie, kterou odvislá BBC. Guido van Rossum byl fanouškem této série (Python Software Foundation, 2001).

Výhody Pythonu

Mezi hlavní výhody Pythonu patří jeho popularita. Ovšem tu musel nějak získat. Narozdíl od ostatních jazyku se Python nezaměřuje na jednu nebo dvě informatické oblasti, ale v Pythonu lze vytvořit snad cokoli ať je to webová stránka, počítačová hra, firemní aplikace, umělá inteligence, Machine learning, analýza dat nebo GUI aplikace. Navíc je velmi intuitivní a lehký na naučení. Tuto vlastnost má hlavně kvůli jednoduchému syntaxu a absenci zbytečných zákonitostí. Většina funkcí má jméno podle toho, co dělají, samozřejmě v angličtině, ale pokud programátor ví, co chce udělat může si ve většině případů jméno funkce odvodit. Toto také pomáhá v obecné čitelnosti programu, i když programátor na funkci nenarazil může si lehce odvodit co dělá. Programovací jazyk Python také umožňuje využívání externích knihoven vytvořených ostatními programátory. U této vlastnosti se hodí už zmíněná popularita Pythonu. Čím je jazyk populárnější tím více bude nabízet kvalitních knihoven. Díky těmto knihovnám je možné si ulehčit svůj projekt. Například vytvoření jednoduchého GUI pomocí dvou řádku kódu nebo celou hru pomocí tří. Tento jazyk má také velmi silnou komunitu, která vytváří zdarma kurzy pro výuku programování. Kurzy jsou vytvořeny primárně pro jedince, tak i pro výuku ve školách. Na tuto komunitu se dá spoléhat i při zaseknutí na nějakém problému, stačí jen vybrat správný klíč který popisuje problém a většinou naleznete někoho, kdo měl stejný problém před pár lety i s několika způsoby, jak zmiňovaný problém vyřešit. Pokud ovšem řešení nenaleznete stačí se zeptat na stránkách k tomu určených. Jednou z takových a pravděpodobně i nejznámější je Stack Overflow. Tato stránka sice není určena jen pro Python, ale pro všechny programovací jazyky, i tak se tam vždy dozvíte několik způsobů řešení (Lutz, Ascher, 2003)

ALGORITMY

Algoritmus vyjadřuje přesný postup řešení úkolu nebo problému, který směřuje k určitému cíli.

„Musí splňovat následující podmínky:

- *musí mít začátek a konec – po konečném počtu kroků musí dojít od počátku do konce*
- *musí být věcně správný*
- *musí být jednoznačný – v každém jeho kroku musí být jasné, jaký bude jeho následující krok*
- *musí být obecný*
- *musí být opakovatelný*
- *musí být srozumitelný“* (Pšenčíková, 2009)

Vlastnosti algoritmu

- Elementárnost – Skládá se z konečného počtu jednoduchých kroků
- Konečnost – Algoritmus skončí pro provedení určitého počtu kroků
- Rezultativnost – Vždy vrátí odpověď na řešený problém.
- Determinovanost – Je jednoznačně definován a při stejných vstupních datech vrací stejný výsledek.
- Hromadnost – Neřeší jen jeden problém ale všechny problémy daného typu s rozdílem vstupních dat. (Pšenčíková, 2009)

Algoritmické myšlení

Algoritmické myšlení patří do obecného informatického myšlení. Algoritmické myšlení se používá u programování, a i v běžném životě. Patří do něj procesy, schopnosti a myšlenky, které používáme při porozumění algoritmů, vytváření algoritmů, jeho úpravě nebo zlepšení efektivity jeho provedení. (Heineman, Pollice, Selkow 2016)

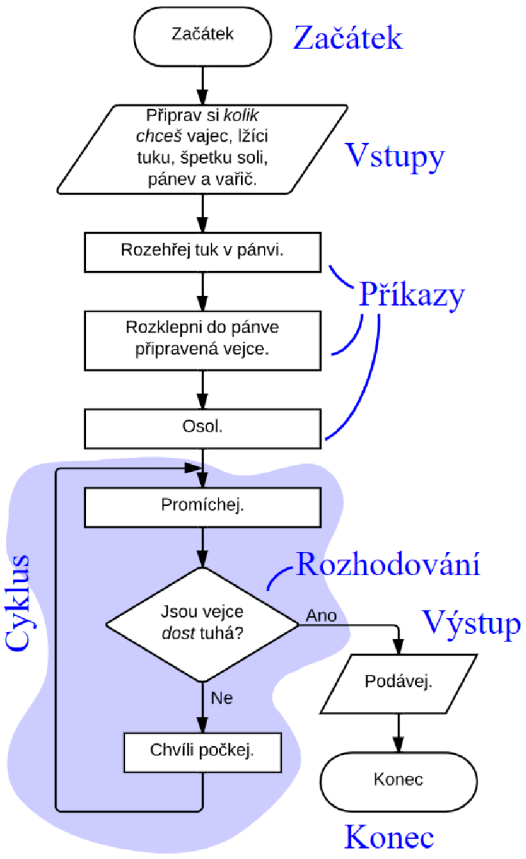
Zapisování algoritmů

Slovní popis

Slovní popis se používá v běžném životě. Patří k nim například kuchařky, návody pro instalaci. Výhody slovního popisu jsou možnosti domluvy. Zato se jedná o nejméně přehledný zápis algoritmů. Slovní zápis také neobsahuje nástroje ke kontrole, jestli algoritmus vede k cíli, jestli je jednoznačný, srozumitelný a přesný. (Pšenčíková, 2009)

Vývojový diagram

Vývojový diagram je grafické zapsání algoritmu, při kterém každá akce má své značky. Jedná se o jeden z nejdokonalejších zápisů algoritmů používaný i při vývoji softwaru. Jeho hlavní výhodou je přehlednost a snadná srozumitelnost, ovšem nedá se použít u všech typů úloh a u složitých úloh ztrácí přehlednost a přebírá nevýhody slovního zápisu. (Pšenčíková, 2009)



Obrázek 1: Vývojový diagram míchaná vajíčka (Dostupný z: https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/U%C4%8Debnice/Algoritmus/V%C3%BDvojov%C3%A9_diagramy)

Pseudokód

Pseudokód je člověkem vymyšlený kód, který je určen pro zápis algoritmů. Pseudokód je navržen tak aby i začátečníci v programování porozuměli zapsanému algoritmu. Pseudokód neobsahuje syntax jako programovací jazyky, příkazy jsou přebrány z angličtiny. Mezi výhody patří zlepšení čitelnosti algoritmu. Vysvětluje, co každý řádek kódu dělá a ulehčuje tedy psaní programu v programovacím jazyce. Pseudokód se může používat i v dokumentaci. (Bennett, 2015)

Počítačový program

Počítačový program se rozumí, že je algoritmus zapsán v jakémkoli programovacím jazyce, jako je Python, C++, C# nebo Java. Mezi výhody patří, že tomuto zápisu rozumí jak počítač, tak člověk (programátor) bez jakýchkoli úprav. Jako hlavní nevýhodou je že programu rozumí jen programátor, který se naučil programovací jazyk, ve kterém je program zapsán. Program je tedy nečitelný pro lajka. Další nevýhodou je,

že tento zápis není názorný a občas trpí i na přehlednost. Při dokumentacích programu se tedy provádí kombinace se zapisováním vývojovým diagramem. (Pšenčíková, 2009)

Druhy algoritmů

Jednoduché algoritmy

Vyhledání minimálního prvku v nesetříděném poli

Při spuštění algoritmu potřebujeme zadat pole ve kterém vyhledáváme a jeho velikost. V samotném algoritmu si uložíme první prvek v poli. Následně procházíme pozice v poli a pokud je na aktuální pozici v poli prvek menší, než náš uložený tak jej uložíme i s pozicí v poli. Pro projití celého pole vrátíme uloženou pozici nejmenšího prvku.

```
Find(A[1 .. n], n):
    min ← A[0]
    min_i ← 0
    for i ← 1 to n
        if A[i] < min
            min ← A[i]
            min_i ← i
    return min_i
```
(Prokop, 2015, s. 24)

Vyhledání zadaného prvku v nesetříděném poli

Při spuštění algoritmu potřebujeme zadat pole ve kterém vyhledáváme, jeho velikost a vyhledávanou hodnotu. Algoritmus prochází jednotlivé pozice v poli a pokud je na této pozici uložena hledaná hodnota funkce vrátí její pozici v poli. Když algoritmus došel na konec pole vrátí hodnotu False.

```
Search(A[1 .. n], n, x):
    for i ← 0 to n
        if x = A[i]
            return i
```

return False (Prokop, 2015, s 24)

Třídění pole

MergeSort

MergeSort je jeden s prvních algoritmů vytvořený pro počítače s možností uložení programů. Algoritmus byl navrhnut John von Neumannem v roce 1990.

Jedná se o rekurzní algoritmus. Algoritmus pole na setřízení rozdělí na 2 části a spustí rekurzní funkci na obě části zvlášť. Následně dvě setřízené části, které algoritmus vrátí sloučíme do jednoho setřízeného pole.

```
„MergeSort(A[1 .. n]):
    if n > 1
        m ← bn/2c
        MergeSort(A[1 .. m]) ⟨⟨Recurse!⟩⟩
        MergeSort(A[m + 1 .. n]) ⟨⟨Recurse!⟩⟩
        Merge(A[1 .. n], m)
Merge(A[1 .. n], m):
    i ← 1; j ← m + 1
    for k ← 1 to n
        if j > n
            B[k] ← A[i]; i ← i + 1
        else if i > m
            B[k] ← A[ j]; j ← j + 1
        else if A[i] < A[ j]
            B[k] ← A[i]; i ← i + 1
        else
            B[k] ← A[ j]; j ← j + 1
    for k ← 1 to n
        A[k] ← B[k]“ (Erickson, 2019, s. 27)
```

Quicksort

Quicksort je také rekurzivní algoritmus. Algoritmus provádí hlavní práci při rozdělování pole na dvě části před provedení rekurze. V roce 1959 algoritmus objevil Tony Hoare a o dva roky později byl vydán.

```
„QuickSort(A[1 .. n]):
    if (n > 1)
```

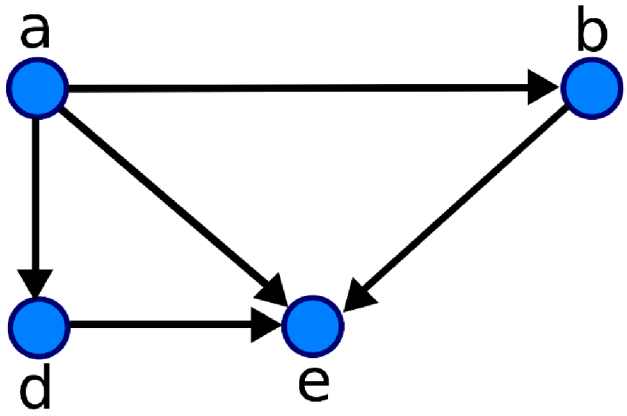
```
        Choose a pivot element A[p]
        r ← Partition(A, p)
        QuickSort(A[1 ..r − 1]) ⟨⟨Recurse!⟩⟩
        QuickSort(A[r + 1 .. n]) ⟨⟨Recurse!⟩⟩

Partition(A[1 .. n], p):
    swap A[p] ↔ A[n]
    l ← 0       ⟨⟨#items < pivot⟩⟩
    for i ← 1 to n − 1
        if A[i] < A[n]
                l ← l + 1
                swap A[l] ↔ A[i]
    swap A[n] ↔ A[l + 1]
    return l + 1“ (Erickson, 2019, s. 29)
```

Grafy

„Graf G = (V, E) je definován množinou vrcholů V a množinou hran E nad dvojicemi těchto vrcholů.“ (Heineman, Pollice, Selkow, 2016) Mezi tři běžně používané typy grafů patří: orientované grafy, ohodnocené grafy, neorientované, neohodnocené grafy (Heineman, Pollice, Selkow, 2016).

Orientované grafy

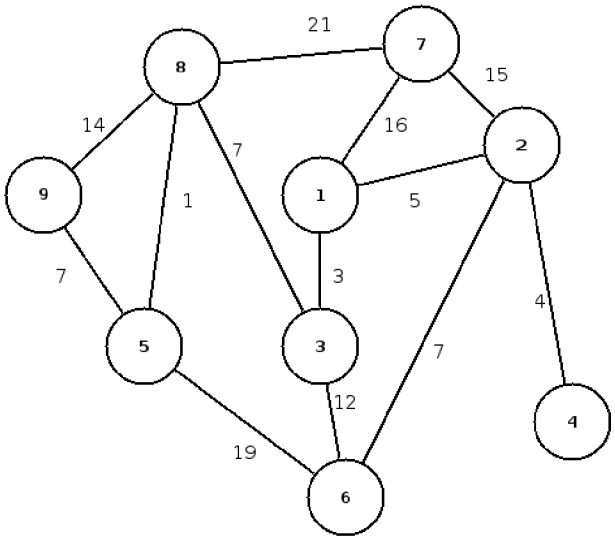
Orientované grafy vytváří vztahy mezi vrcholy, které se liší od vztahů mezi stejnými vrcholy v opačném směru, který nemusí existovat. Takový vztah můžeme porovnat k jednosměrným ulicím, kdy auto může projet z křižovatky na křižovatku jen v jednom směru (Neapolitan, 2015).



Obrázek 2: Orientovaný graf (Dostupný z: https://cs.wikipedia.org/wiki/Orientovan%C3%BD_graf)

Ohodnocené grafy

Ohodnocené grafy vytváří vztahy mezi vrcholy, kde se k tomuto vztahu přidá číselná hodnota. V některých případech může uchovávat i nečíselné informace. Pokud si vrcholy určíme jako města můžeme v této hodnotě například uložit vzdálenost v km mezi nimi nebo čas cesty. Pokud se jedná o nečíselnou informaci, může to být například jméno ulice, která spojuje dvě křižovatky (Neapolitan, 2015).



Obrázek 3: Ohodnocený graf (Dostupný z: https://ufal.mff.cuni.cz/sites/default/files/purl_legacy/vodrazka/public_html/vyuka/index.php?archiv=P2MLS1213_7)

Neorientované, neohodnocené grafy

Neorientované, neohodnocené grafy vytváří vztahy mezi vrcholy, které neberou ohled na směr nebo váhu cesty. Pokud mají dva vrcholy vztah platí stejný vztah i v opačném směru. Tento graf slouží k vytváření bludišti nebo mapování sociálních sítí (Neapolitan, 2015).

Hledání cesty v grafu

- Dijkstrův algoritmus řeší problém nejkratší cesty s jedním počátečním bodem a jen s pozitivními hodnotami uzlu
- Bellman–Ford algoritmus řeší problém nejkratší cesty s jedním počátečním bodem a hodnoty uzlu v mínusových hodnotách.

- A* search algoritmus řeší problém nejkratší cesty s pomocí heuristického výpočtu pro urychlení vyhledávání
- Floyd–Warshall algoritmus řeší všechny nejkratší cesty.
- Johnson's algoritmus řeší všechny nejkratší cesty, při použití na řídkých grafech může být rychlejší než Floyd–Warshall. (Erickson, 2019)

Dijkstrův algoritmus

Dijkstrův algoritmus zjišťuje nejkratší cestu ze startovního vrcholu do všech ostatních vrcholů v orientovaném ohodnoceném grafu. Využívá strukturu známou jako prioritní fronta. Prioritní fronta ke každé své položce přiřadí číslo, která určuje prioritu položky, kdy menší čísla mají větší prioritu. Algoritmus navrhl Edsger W. Dijkstra v roce 1956 a o tři roky později ho publikoval (Erickson, 2019).

Algoritmus každému vrcholu přiřadí prioritu nekonečno. Vyjme startovní vrchol z fronty. Algoritmus k sousedním vrcholům startovního vrcholu přiřadí prioritu jako vzdálenost od startovního vrcholu. Poté přejde na vrchol s nejmenší prioritou ve frontě, který z něj vyjme, ohodnotí sousedy svou hodnotou plus hodnotou hrany, pokud mají menší hodnotu a prochází, dokud není fronta prázdná. (Heineman, Pollice, Selkow, 2016).

„Dijkstra(s):
InitSSSP(s)
Insert(s, 0)
while the priority queue is not empty
u ← ExtractMin( )
for all edges u→v
if u→v is tense
Relax(u→v)
if v is in the priority queue
DecreaseKey(v, dist(v))
else
*Insert(v, dist(v))“ (*Erickson, 2019, s. 285*)*

Bellman–Ford

Bellman–Ford algoritmus řeší problém nejkratší cesty pro všechny vrcholy v grafu. Graf může obsahovat hrany s negativní hodnotou. Bellman–Ford algoritmus je časově složitější než Dijkstrův algoritmus. Algoritms byl poprvé navržen Alfonsem Shimbelem v roce 1955, ale byl pojmenován po Richardu Bellmanovi a Lesteru Fordovi Jr., kteří ho v roce 1958 a 1956 publikovali (Erickson, 2019).

```
„BellmanFord(s)
    InitSSSP(s)
    repeat V − 1 times
    for every edge u→v
        if u→v is tense
                Relax(u→v)
for every edge u→v
    if u→v is tense
        return "Negative cycle!"
```
(Erickson, 2019, s. 292)

ROBOTICKÉ STAVEBNICE

LEGO®

V roce 1979 se stal generálním ředitelem společnosti LEGO Group Kjeld Kirk Kristiansen. Do společnosti přináší spoustu nových nápadů. Má vizi toho, aby LEGO nebylo jen hračka na hraní, ale aby pomáhala i při vzdělávání. Proto v roce 1980 zakládá nové oddělení s názvem LEGO Institutional Department. V tomto novém oddělení se klade důraz na učení hrou. Společnost začne spolupracovat s předškolními, základními a středními školami a vyvíjet produkty LEGO přímo pro tyto instituce. V roce 1989 se změnil název oddělení LEGO Institutional Department na LEGO DACTA, následně v roce 2002 na LEGO Educational Division a v roce 2006 se název změnil na současný LEGO Education. LEGO Education má poslání učinit učení zábavným a účinným a vytvořit aktivní, spolupracující a celoživotní studenty. LEGO Education nabízí v průběhu let školám spoustu materiálů jak pro učitele, tak pro studenty. LEGO Technic je roku 1982 představen jako produkt zaměřený na matematiku, vědu, technologii a robotiku školským zařízením. Během let byly představeny první počítačově řízené produkty LEGO Technic Control I a II. Na pochopení základních vědeckých konceptů světa byly navržené Early Simple Machines (rané jednoduché stroje) a Simple and Powered Machines (jednoduché a poháněné stroje) byly pro zobrazené fyzického chování (LEGO, 2022).

LEGO Education je uzpůsobeno tak aby zapojovali aktivně studenty do výuky a podněcovat v nich zájem o studium (LEGO, 2022).

Tento přístup zajišťují 4C:

- Connect (připojení) – zajišťuje představené úkolu a umožňuje studentovi klást otázky na objasnění problému a možnost zapojení vlastních znalostí
- Construct (zkonstruování) – zahrnuje výstavbu a podporuje experimentování a následné připomínky
- Contemplate (uvažování) – studenti by měli zvážit co se naučili a vzájemně to konzultovat
- Continue (pokračování) – udržuje motivaci a zvídavost studentů tím, že na vyřešený úkol navazuje nový úkol a rozšiřuje jejich nově nabyté znalosti

Roku 1984 započne spolupráce mezi společností LEGO a profesorem z MIT Seymour Papert. Seymour Papert má stejnou vizi, jako majitel LEGO Kjeld Kirk Kristiansen, ve které se děti učí hrou. Již v roce 1987 díky této spolupráci je uveden na trh LEGO Technic Control 0 naprogramovatelný pomocí programovacího jazyka LOGO vyvinutým Seymour Papert a jeho týmem z MIT. Nejznámějším produktem, který vznikl díky této spolupráci je LEGO MINDSTORMSTM, který byl uveden na trh roku 1998. Pod LEGO Education jsou produkty MINDSTORMS zařazeny roku 1999 (LEGO, 2022).

Během let LEGO naváže spolupráci s mnoha firmami., uneverzitamia a organizacemi Jako je FIRST (For Inspiration and Recognition of Science and Technology), NASA (Národní úřad pro letectví a kosmonautiku), National InstrumentsTM, Tufts University a Fraunhofer. Díky spolupráci ze společností National InstrumentsTM vznikl software LabVIEW, který se používá v produktech LEGO MINDSTORMS dodnes (LEGO, 2022).

Lego SPIKE Prime

Lego Spike je v současnosti požívanou verzí stavebnic robotů Lego Education. Společnost Lego Education k této verzi stavebnice vydala 32 lekcí, jak pro učitele, tak pro žáky, kde vysvětlují jak se stavebnicí pracovat v různých předmětech (přírodní vědy, matematika, sociálních předmětech nebo při výuce jazyků). Stavebnice je primárně určena pro žáky 10+ (The brothers brick, 2020).

Řídící část Lego Spike je menší než Lego Mindstorms EV3. Software je u obou stejný, i motory a snímače mají podobý výkon a rozlišení. Ale hlavní výhodou Spike jsou obdelníkovější snímače a motory, které umožňují stavbu mnohem lepších robotů (Oitzman, 2022).

Kostičky jsou u této verze mnohem barevnější než u předchozích verzí, což je v dnešní době vítaná změna. Dochází tak k oslovení větší skupiny žáků a vzniklí roboti nejsou tak fádní. Lego Spike Prime je dodáván v plastové stohovatelné krabici pro opakované použití ve třídách. V této krabici se nachází i organizér na menší dílky, který je optřen obrázky dílků, které tam patří (The brothers brick, 2020).

Programovatelná kostka obsahuje celkem 6 vstupních a výstupních portů. Na horní straně se nachází světelná matrice o velikosti 5x5. Propojení s počítačem lze zajisti

přes Bluetooth nebo pomocí kabelu. Součástí je i 6osý gyroskop a dobíjecí baterie, která se připojuje pomocí MicroUSB (LEGO, 2022).

Základní balení s označením 45678 obsahuje:

- programovatelný HUB s baterií
- senzor vzdálenosti
- senzor síly
- barevný senzor
- velký motor
- 2x střední motor
- 523 dílků LEGO Technic

(EDUXE, 2023)

Obrázek 4: Lego stavebnice SPIKE Prime, (Dostupné z: https://www.lego.com/cs-cz/product/lego-education-spike-prime-set-45678)

Rozšiřující balení označené 45681 mimo jiné obsahuje:

- velká kola
- barevné čidlo

- velký motor

(insgraf, 2023, on-line)

LEGO® Mindstorms®

Název Mindstorms vychází z knihy Mindstorms od Seymour Papert, ve které popsal vzdělávací konstruktivismus. Stavebnice Mindstorms byla prvotně vyrobena pro potřeby vzdělávání ve školských zařízeních, ale její obliba byla tak velká, že se stala první domácí robotickou stavebnicí. LEGO Mindstorms bylo na trhu dostupné od roku 1998 do konce roku 2022. Každá stavebnice LEGO Mindstorms se skládá z kostky (mozek robota), do které jsou připojeny senzory a motory, samotné tělo je sestaveno z dílků stavebnice Lego Technic. Jako první programovací jazyk se rozhodli použít Visual, který vznikl ve spolupráci s laboratoří MIT Media. První demografické zaměření Lega bylo velmi úzké (chlapci 10–14 let). Tito chlapci byli podle průzkumu určeni jako skupina, kterou budou nejvíce přitahovat počítačové hračky. Prodej Lego Mindstorms byl spuštěn 1.9.1998, a již během prvních 3 měsíců byla vyprodána celá série (cca 100 tisíc kusů) (Watters, 2015, on-line).

Robotics Invention System RCX (1998) 1. generace

Součástí byla kostka RCX (Robotic Command eXplorers). Kostka měla 3 vstupní porty pro připojení senzorů a 3 výstupní porty, které umožňovaly komunikaci několika RCX kostek. Součástí kostky je i LCD displej, na kterém bylo možno zobrazovat stav baterie nebo stavy vstupních a výstupních portů, informace o současném výběru programu a další informace. Součástí RCX kostky 1.0 byla baterie i konektor pro nápajecí adaptér, ve verzi 2.0 byl tento konektor odstraněn. Oficiální programovací prostředí bylo ROBOLAB založeno na programu LabVIEW pro výukové účely a výukové sety a RCX Code, které bylo pro běžné zákazníky. Obsluha byla jednodušší, ale nevýhodou byla nemožnost rozsáhlejšího programování (Jandejsek, 2013).

Mindstorms NXT (2006) 2. generace

Byla to náhrada za stavebnici první generace. Dodával se ve dvou verzích – pro maloobchod a pro vzdělávání. Všechny moduly a senzory jsou do kostky připojeny pomocí speciálně upravených konektoru RJ-12. K připojení senzorů slouží čtyři vstupní porta a tři výstupní porty k připojení motorů. Napájení je zajištěno dvěma způsoby ve verzi pro školy je to Li-Ion dobíjecí baterií a pro běžné uživatele je napájení zajištěno 6 AA bateriemi. V sadě Lego Mindstorms 1.0 jsou obsaženy 3 stejné motory, dále se tam nachází dotykový senzor, světelný senzor, zvukový senzor a ultrazvukový senzor. Verze Lego Mindstorms 2.0 obsahuje ještě navíc barevný senzor, zvukový senzor. Základní sadu lze dále rozšířit o kompasový senzor, senzor akcelerometru, rotační senzor a RFID senzor, který zajišťuje komunikaci mezi více roboty. Lego Mindstorms NXT lze programovat na mnoha alternativních programovacích prostředí (Reich, 2008).

Mindstorms EV3 (2013) 3. generace

Mindstorms EV3 byla vyvinuta tak, aby byla zachována co největší kompatibilita s předchozí stavebnicí Mindstorms NXT. Kdy největší změnou prošla programovatelná kostka, a to hlavně co se týká výkonu procesoru, ten byl navýšen díky procesoru CPU ARM9 se systémem Linux. Programovatelná kostka obsahuje čtyři výstupní porty pro motory, čtyři vstupní porty pro senzory, reproduktor, čtečku MicroSD karet

Obrázek 5: Mindstorms EV3 (Dostupné z: https://lego.heureka.cz/lego-mindstorms-31313-ev3/#prehled/)

a dvoubarevnou LED diodu. V roce 2022 bylo LEGO Mindstorms ukončeno a nahradilo ho LEGO Spike (Kopecký, Szotkowski, Kubala, Krejčí, Havelka, 2021).

VEX

Robotické sady VEX jsou alternativou k robotickým stavebnicím od LEGO. VEX nabízí celo řadu robotických sad, které jsou vhodné mateřských škol až po vysoké školy (VEX Robotics, 2023).

VEX 123

Tato robotická sada je určena pro děti od 4 let. Obsahuje jednoduchý robot, který lze ovládat přímo tlačítky na těle. Starší žáci mohou použít programování pomocí programovací tabulky. Děti se zde naučí algoritmizaci a jednoduché příkazy (VEX Robotics, 2023).

VEX GO

Jedná se zjednodušenou sadu VEX IQ. Stavebnice VEX GO je určena pro I. stupeň základních škol. Stavba robota je v této sadě rozdělena do tří kroků. První krok je samotná stavba robota, kde žáci pochopí mechaniku a fyzikální principy. V druhém kroku doplní motory a ovládání a ve 3 kroku je samotné propojení s aplikací VEX CODE a samotné naprogramování (VEX Robotics, 2023).

VEX IQ

Jde o plnohodnotnou robotickou stavebnici, která je určena pro II. Stupeň základních škol a střední školy. Stavebnice obsahuje programovatelnou kostku, ke které se připojují veškeré senzory a motory. Dále se ke stavebnici dodává i ovladač, pomocí kterého jej můžeme snadno ovládat. Nebo ji lze naprogramovat pomocí aplikace VEX CODE – programování v blocích (založené na SCRATCHI) a textové (založené na PYTHONU) (VEX Robotics, 2023).

VEX EXP

Pokročilá robotická stavebnice pro žáky od II. Stupeňe základních škol. Stavebnice obsahuje mechaniku s kovových dílů. Stejně jako stavebnice VEX IQ

obsahuje ovladač pro snadné a rychlé otestování funkčnosti robota. A jeho odzkoušení naprogramovat pro robota autonomní pohyby (VEX Robotics, 2023).

VEX V5

Tato stavebnice dovoluje sestavit pokročilé roboty se složitými mechanismy. Stavebnice obsahoje návod, jak jednoduše sestavit prvního robota Clawbot IQ. Dále je součástí stavebnice modulární a projektově orientovaný učební pán, který učí designový proces praktickým a zajímavým způsobem (VEX Robotics, 2023).

FisherTechnik Robotics

Tato stavebnice je určena pro děti od 10 let. Stavebnice obsahuje řídící jednotku, motory a různé snímače a LED. Zajímavostí je, že tato stavebnice obsahuje i kameru pro zpracování obrazu. Programování robota lze provést pomocí grafického programovacího softwaru ROBO Pro Coding a ovladače ROBOTICS TXT 4.0. Pokročilí uživatelé mohou použít programovaní v blocích nebo textové v PYTHONU. FisherTechnik obsahuje také komplexní a volně přístupné výukové materiály (Fischertechnik, 2023, on-line).

Mbot

Ve své podstatě se nejedná o stavebnici jako takovou, ale jde o vzdělávacího robota. Tento robot je určen pro středoškolské vzdělávání. Mbot běži na CyberPi jde o výkonný a všestranný mikrokontrolér. Má integrované senzory, plnobarevný displej a možnost komunikovat přes Wi-Fi. Mbot lze rozšířit o širokou škálu inteligentních elektronických modulů a konstrukčních dílů mBuild od Makeblock Education. Programování s zde provádí pomocí blokového programování a nebo pomocí PYTHONU. (Makeblock, 2023, on-line)

BitBeam

Jde o robotikou stavebnici, která je založena na běžně dostupném Arduinu a na jeho různých čidlech a modulech. Dílky stavebni lze koupit nebo si je zdarma táhnout a vytisknout na 3D tiskárně díky OPEN SOURCE. Dále je stavebnice kompatibilní s LEGO MINDSTORMS a LEGO TECHNICS. (Feltl, 2023, on-line)

ROBOTICKÉ SOUTĚŽE

Nejznámější a největší soutěží je FIRST LEGO League, kterou založilo a zašťituje přímo společnost LEGO.

FIRST LEGO League

Soutěž funguje již od roku 1998, kdy ji založil Dean Kamen a majtel LEGO Group Kjeld Kirk Kristiansen. Soutěže se můžou zúčastnit děti ve věku od 4 do 16 let. Věk soutěžících se liší podle země, ve které se soutěže konají. Účastníci získávají zkušenosti s řešením problémů v reálném světě prostřednictvím řízeného globálního programu robotiky, který pomáhá dnešním studentům a učitelům společně budovat lepší budoucnost. (FIRST, 2023, on-line)

Děti jsou podle věku rozděleny do 3 kategorií:

- DISCOVER – Tato kategorie je pro ty nejmenší soutěžící ve věku 4-6 let. Používá se stavebnice LEGO DUPLO.
- EXPLORE – Je pro věkovou kategorii 6-10let. V této kategorii se začíná se základy inženýrství při zkoumání skutečných problémů. Používají k tomu stavebnici LEGO Education Spike Essential.
- CHALLENGE – Je pro soutěžící ve věku 9-16 let. Soutěžící se zabývají komplexním řešením robotů ze stavebnice Lego Education Spike.

Hlavní filozofií soutěže není výhra, ale přáteskla soutěživost, týmová práce a chuť se učit novým věcem.

V soutěžích se hodnotí tyto 4 kategorie:

- Robotgame – robot musí splnit v časovém limitu co nejvíce úkolů
- soutěžící musí prozkoumat možnosti využití určité technologie a svoje výsledky přednést porotě
- umění prezentovat schopnosti robota
- porota ocení týmovou práci jednotlivých skupin

V ČR je tato organizace zastoupena Českou ligou robotiky (FIRST, 2023, on-line).

ROBOTRIP

Je česká robotická soutěž, která má mnoho soutěžních kategorií.

*„**Soutěžní disciplíny***

RoboTrip – Olomoucké robokáry / Olomouc-robotaj aŭtoj

RoboTrip – Tulák po parku / Vaganto en la parko

RoboTrip – Olomoucká hřebenovka / Olomouc-montokresta transirejo

RoboTrip – Oslavy Slunovratu / Solsticaj festoj

Stopař / Line Follower

Stopař s překážkami / Line Follower enhanced

MiniSumo

Lego MiniSumo

Lego konstruktér / Lego Constructor

*Čárové bludiště / Line Maze“ (*Děrda, 2023, on-line)

V praktické části mé bakalářské práce se zabývám stavbou a naprogramováním robota pro pohyb v bludišti. Proto mě nejvíce zajímají propozice a průběh této soutěže.

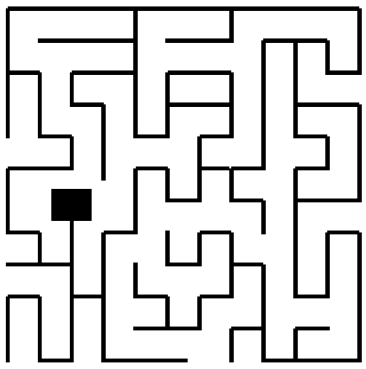
*„**Průběh soutěžních jízd***

- *místo startu – vstupu do bludiště, určí rozhodčí – vždy levý dolní roh,*
- *při zastavení, nebo ztrátě čáry, je pokus ukončen a pokračuje se z počátečního-startovního místa novým pokusem,*
- *pro průjezd každého bludiště je stanoven limit 10 minut, během nichž je možné absolvovat libovolný počet pokusů,*
- *pořadí se určí součtem nejlepších časů ze všech bludišť,*
- *neabsolvování nebo nenalezení cíle = 600 s,*
- *robot se smí pohybovat pouze po existujících cestách = je možný přesun jen na sousední křižovatku, na kterou vede přímá cesta,*

- *robot si může zapamatovat projetou trať a uložená data využít v následujících pokusech,*
- *cíl bludiště je vyznačen černým čtvercem 150x150mm, robot se v něm MUSÍ zastavit min. na 5 sekund. Následně může pokračovat v pokusu. Zaznamenán bude čas dosažení cíle.*
- *rozhodčí může rozhodnout o kontrolním průjezdu bludiště, tj. průjezdu bludiště z jiného startovního umístění, kdykoli v průběhu soutěže,*
- *soutěž bude probíhat ve 2-3 kolech – bludištích,*
- *po stanovení startovní pozice už nebude možné robota odnášet od tratě a zasahovat do jeho programu, dokud všichni soutěžící neodjezdí dané kolo (bludiště),*

Technické parametry robotů

- *Během soutěžní jízdy není povolena žádná vnější komunikace, tj. žádné dálkové řízení, ani komunikace s počítačem.*
- *Žádný rozměr robota nesmí překročit 300 mm.*
- *Pro pohon robota je možné použít pouze elektrické články s tuhým elektrolytem a maximálním napětím 24 V.“ (*Děrda, 2023, on-line)



Obrázek 6: Čárové bludiště se smyčkami (Dostupné z: http://robotrip.cz/discipliny/carove-bludiste-line-maze/)

V České republice se pořádá spousta dalších robotických soutěží, jako jsou:

- Robotický den 2022
- Robosoutěž ČVUT
- ROBOTIÁDA
- European Rover Challenge

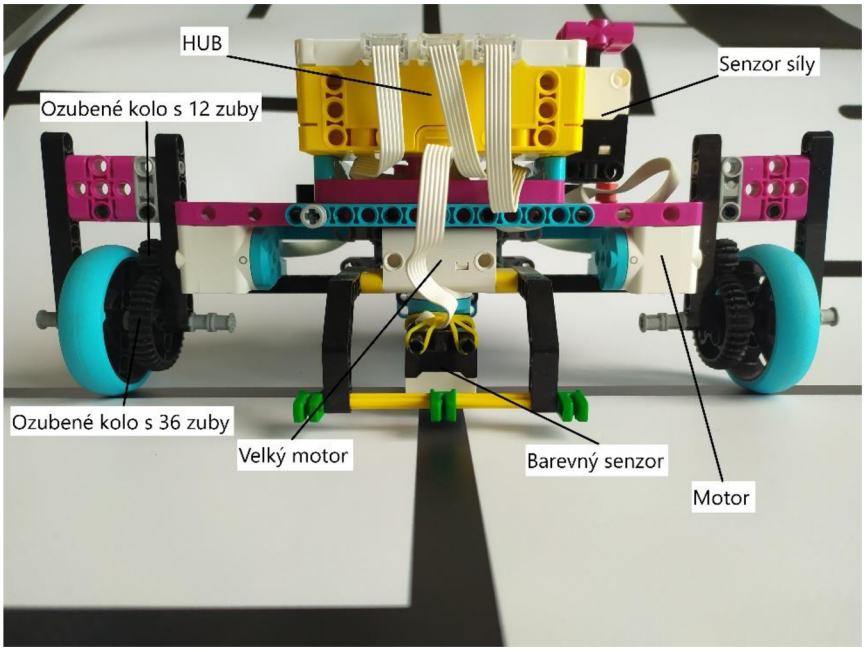
II. PRAKTICKÁ ČÁST

POSTUP SESTAVENÍ ROBOTA

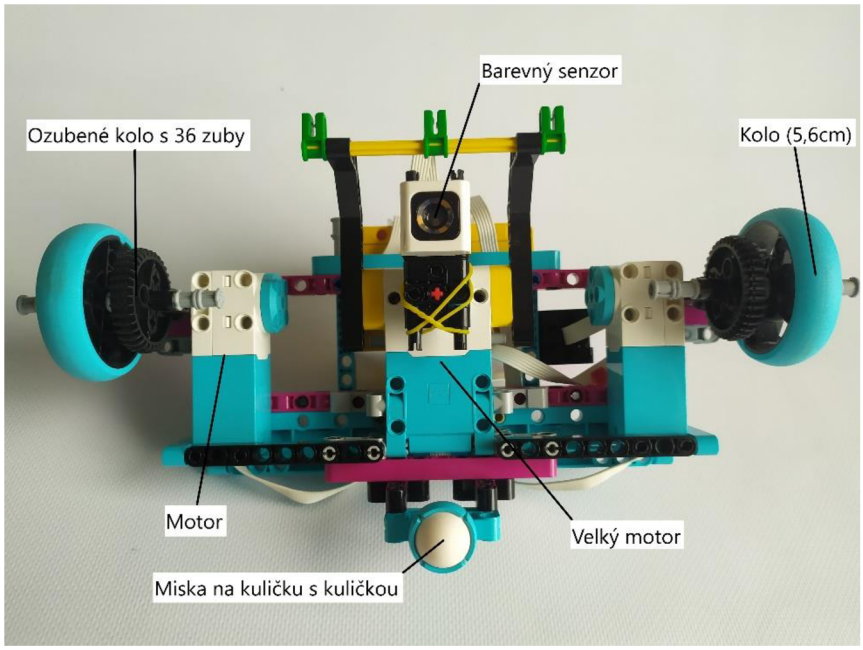
Cílem je popsat postup sestavení robota a jeho naprogramování od nejjednodušších robotických úloh až k algoritmům, které lze použít na robotických soutěžích.

Stavba robota

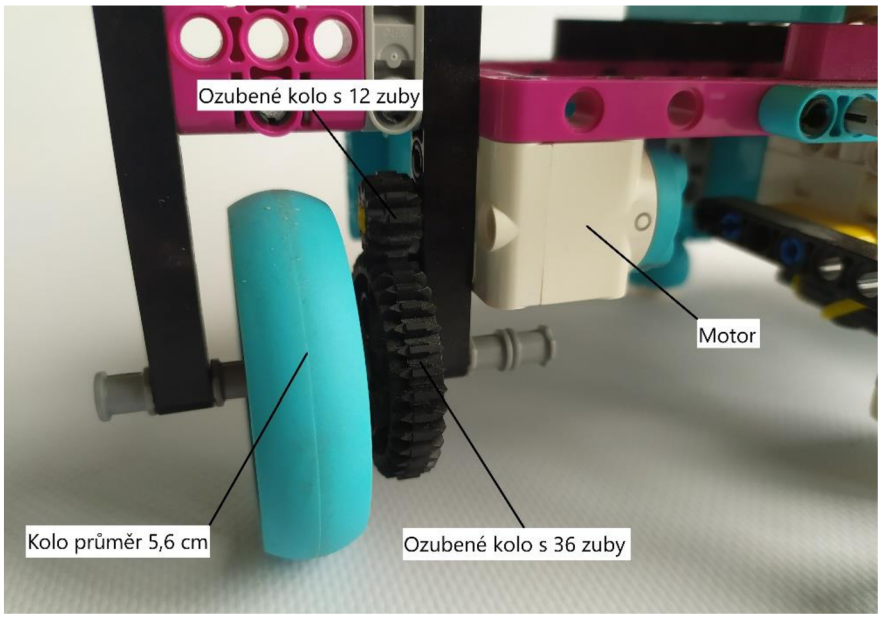
Před postavením robota se musíme rozhodnout pro to co robot musí umět a co máme k dispozici za součástky. V mé bakalářské práci se zabývám průchodem bludiště. Při stavbě robota pro průchod bludištěm potřebujeme minimálně 2 motory, 2 kola o průměru 5,7 cm, barevný senzor a spojovací bloky. Používání takového robota by bylo ovšem velmi nepohodlné. Pro zjištění cesty na křižovatce bychom museli otáčet celým robotem. Přidáme-li tedy další motor pro otáčení barevného senzoru. Ideálně motor se senzorem přiděláme na střed robota, tak aby se senzor mohl otáčet o 360 stupňů. Při montáži kabeláže musíme s tímto pohybem počítat. Následně máme i možnost přidání převodových kol k pohybu robota. Výhodou při převodu do rychla je vyšší rychlost robota, ale to má za následek ztrátu přesnosti motoru při zastavení. Robot má sice funkce pro zamknutí pohybu motoru, ale motor se může posunout o pár stupňů například kvůli jeho vůli při uložení. Dále při převodu do rychla má motor tendenci provádět trhavé pohyby při malých rychlostech. Taková stavba může být výhodná například u disciplín jako je Line Folower. Při použití převodu do pomala motor musí vykonat více otáček a tím pádem jsou pohyby robota plynulejší a celkový pohyb je přesnější. Proto v našem případě použijeme hnací kolo s 8 zuby a hnané kolo s 24 zuby pro přesnější pohybování robota. Dále přidáme senzor síly, který použijeme jako tlačítko pro spuštění robota.



Obrázek 7: Soutěžní robot z přední strany (Foto autor)



Obrázek 8: Soutěžní robot ze spodní strany (Foto autor)



Obrázek 9: Detail převodu soutěžního robota (Foto autor)

Programování robota

V práci vytvoříme robota pro průchod bludištěm. Tento úkol si rozdělíme do několika menších algoritmických úkonů, které na konec spojíme do funkčního celku.

Popis průchodu robota bludištěm. Robot dojede na křižovatku, zjistí cesty na okolní křižovatky a uloží si je do mapy. Následně vybere sousední křižovatku na které nebyl a přemístí se na ni. Pokud nenašel křižovatku na které nebyl v sousedních křižovatkách vypočítá nejkradší cestu k neprojeté křižovatce pomocí uložené mapy a algoritmu BFS. Toto chování opakuje dokud nenajde cíl. Následně robota přemístíme na start a spustíme robota pro vyhledávání zbytku mapy. Po zmapování celého bludiště robota přemístíme na start. Kde provede výpočet nejkratší cesty do cíle pomocí A-star algoritmu a projedeme bludiště po vypočítané cestě.

Pohyb robota

- Otočení senzoru před robota
- Pohyb robota po rovné cestě na další křižovatku
- Otočení robota (do leva, do prava, do zadu)
- Průjezd zadané cesty

Výběr cesty

- Zjištění cest z křižovatky
- Výběr další cesty robota

Úprava informací

- Uložení směru robota a přetečení jeho hodnot
- Úprava pozice robota podle jeho směru.
- Otočení bitového čísla podle směru robota.

Výpočet nejkratší cesty

- Algoritmus pro vyhledání nejkratší cesty BFS
- Kontrola jestli se prvek nachází ve frontě.
- Algoritmus pro vyhledání nejkratší cesty Astar
- Vypočítání Heuristické vzdálenosti použité v algoritmu Astar.

Pohyb robota

Vrátit senzor do absolutní nuly

V průběhu programu je nutné vrátit světelný senzor do polohy absolutní nuly (výchozí poloha). Vytvoříme algoritmus, který proces provede. Jelikož každý senzor je připojen kabelem musíme dávat pozor, aby se nedošlo k zastavení otáčení senzoru z důvodu namotání kabelu, délky kabelu nebo skřípnutí kabelu. Algoritmus tedy musí rozlišit, jestli se vrátit na absolutní hodnotu vrátit po nebo proti směru hodinových ručiček. Momentální pozici zjistíme pomocí příkazu motor.absolute_position(motor), který nám vrátí jeho pozici ve stupních.

```
def Return_To_Absolute(selected_motor):

    #Zjištění pozice motoru.
    position = motor.absolute_position(selected_motor)
```

```
        #Rozhodnutí kterým směrem se otočit.
        if position > 225:
        #Otočení po směru hodinových ručiček o zjištěnou polohu a
nastavení motoru na HOLD.
                motor.run_for_degrees(selected_motor, position, 400,
        stop = 2)
        else:
                motor.run_for_degrees(selected_motor, -1 * position,
        400, stop = 2)
        time.sleep_ms(500)
```

Pohyb po čáře

První věc, na kterou se musíme zaměřit jsou hodnoty naskenované senzorem (hodnotu kterou chceme dosáhnout). V případu bludiště na lesklém podkladu se nemůžeme spoléhat na rozpoznání barvy senzorem. Na lesklé černé i lesklé bílé senzor vrátí že se nachází na bílé barvě. Musíme tedy použít funkci pro odraz světla z knihovny color_sensor. Při použití odrazu světla na mapě bludiště senzor vrací hodnoty 100 na bílé barvě a 50 na černé. Senzor má ovšem široký záběr musíme tedy počítat s tím, že ideální černá se bude nacházet jen na středu čáry. V ideálním případě by senzor měl vracet hodnoty 100 na bílé a 0 na černé. Naše mapa má lesklí povrch a vrací nám tudíž jiné hodnoty.

```
while True:
        # Uložení hodnoty odraženého světla od podložky.
        scan = color_sensor.reflection(_senzor)
        # Pro úpravu korekce p zjistím rozdíl mezi hodnotou scanu a
ideální hodnoty po které se chceme pohybovat. Vypočítaný rozdíl
převedu do absolutní hodnoty a vynásobím hodnotou pi, která určuje
agresivitu korekce.
        p = abs(scan-ideal) * pi
        # Pokud je naskenovaná hodnota blíže ke středu černé čáry než
zadaná ideální hodnota. Začnu se otáčet doleva o korekci p.
        if ideal > scan:
        # Zpomalení levého motoru o korekci p a zrychlení pravého
motoru o korekci p
                motor.run(_left_motor, (-1 * (speed - p)))
                motor.run(_right_motor, speed + p)
```

```
        else:
        # Zpomalení pravého motoru o korekci p a zrychlení levého
motoru o korekci p
                motor.run(_right_motor, speed - p)
        motor.run(_left_motor, -1 * (speed + p))
```

Ujetá vzdálenost

Pro zjištění ujeté vzdálenosti robota potřebujeme znát vzdálenost, kterou urazí kolo při otočení o 360 stupňů. Pro výpočet této hodnoty potřebujeme znát průměr kola a počet zubů ozubených kol které se nachází mezi kolem a motorem. V našem případě má robot kola s průměrem 5,6 cm, která ujedou vzdálenost 17,6 cm za otočku a ozubené hnací kolo o velikosti 12 zubů a hnané kolo s 36 zuby.

Postup výpočtu je následující:

- Vypočet převodového pomeru 36/12=3
- Výpočet úhlové vzdálenosti otočení motoru pro ujetí kola o 15 cm. `(distance_cm / _wheel) * 360 * _gear = (15 / 17,6) * 360 * 3 = 920,5 stupňů`

```
wheel = 17.6
gear = 36/12
def Travel_To_Next_Crossroads():
        global _wheel, _gear, _left_motor, _right_motor,
_senzor_motor
        # Místní promněnné
        speed = 300
        ideal=60
        distance_cm = 15 #cm
        pi = 10

        # Výpočet vzdálenosti ve stupních.
        distance_degree = (distance_cm / _wheel) * 360 * _gear
        # Otočení motoru držícího barevný senzor na pozici absolutní
nuly.
        Return_To_Absolute(_senzor_motor)
```

```
    # Reset relativní vzdálenosti na pravém i levém motoru.
    motor.reset_relative_position(_left_motor, 0)
    motor.reset_relative_position(_right_motor, 0)

    # Pokud je ujetá vzdálenost menší než ⅔ vypočítané úhlové
vzdálenosti od další křižovatky, provádíme korekci rychlosti motorů
pro sledování čáry.
    while (motor.relative_position(_right_motor) + (-1 *
motor.relative_position(_left_motor))) / 2 < (distance_degree / 3 *
2):

    # Uložení hodnoty odraženého světla od podložky.
        scan = color_sensor.reflection(_senzor)
    # Pro úpravu korekce p zjistím rozdíl mezi hodnotou scanu a
ideální hodnoty po které se chceme pohybovat. Vypočítaný rozdíl
převedu do absolutní hodnoty a vynásobím hodnotou pi, která určuje
agresivitu korekce.
        p = abs(scan-ideal) * pi
    # Pokud je naskenovaná hodnota blíže ke středu černé čáry než
zadaná ideální hodnota. Začnu se otáčet doleva o korekci p.
        if ideal > scan:
    # Zpomalení levého motoru o korekci p a zrychlení pravého
motoru o korekci p
            motor.run(_left_motor, (-1 * (speed - p)))
            motor.run(_right_motor, speed + p)
        else:
    # Zpomalení pravého motoru o korekci p a zrychlení levého
motoru o korekci p
            motor.run(_right_motor, speed - p)
            motor.run(_left_motor, -1 * (speed + p))

    # Dojezd poslední ⅓ zadané vzdálenosti.
    motor.run_for_degrees(_right_motor, int(distance_degree / 3 +
5), speed)
    motor.run_for_degrees(_left_motor, int(distance_degree / 3 +
5) * -1, speed)
    time.sleep_ms(800)
```

Otočení robota

Při otáčení robota musíme brát na vědomý, kde se po otočení vyskytne barevný senzor. V našem případě robot pojede na další křižovatku po levé hraně černé čáry. Je tedy ideální na ní i zastavit otáčení robota. Proto provádíme dotočení robota z levé strany čáry.

Jako první vrátíme senzor na absolutní nulu. Určíme kam se má robot otočit a započneme rychlé otáčení před určenou cestu. Pokud se jedná o otočení doleva cestu o kousek přejedeme. Následně spustíme pomalé dotočení robota, které se zastaví při nalezení černé čáry.

```
def Robot_Turn(way):
      global _right_motor, _left_motor, _senzor, _senzor_motor,
_black
      # Rychlost dotáčení robota.
      speed = 200

      # Otočení motoru držícího barevný senzor na pozici absolutní
nuly.
      Return_To_Absolute(_senzor_motor)

      # Otočení do prava.
      if way == "right":
      # Pojetí do zatáčky
            motor.run_for_degrees(_right_motor, 15, 200)
            motor.run_for_degrees(_left_motor, -15, 200)
            time.sleep_ms(500)
            motor.run(_left_motor, -2000)
            motor.run(_right_motor, -2000)
            time.sleep_ms(700)

      # Otočení doleva. Přetočení přes cestu.
      if way == "left":
            motor.run_for_degrees(_right_motor, -30, 200)
            motor.run_for_degrees(_left_motor, 30, 200)
            time.sleep_ms(500)
            motor.run(_left_motor, 2000)
```

```
            motor.run(_right_motor, 2000)
            time.sleep_ms(1500)

      # Otočení zpět
      if way == "back":
            motor.run(_left_motor, -2000)
            motor.run(_right_motor, -2000)
            time.sleep_ms(2000)

      # Začínáme točit vpravo.
      motor.run(_left_motor, -1 * speed)
      motor.run(_right_motor, -1 * speed)

      # Čekáme dokud se senzor nenachází na černé barvě.
      while color_sensor.reflection(_senzor) > _black + 10:
            pass
      # Zastavíme otáčení robota.
      motor.run(_left_motor, 0)
      motor.run(_right_motor, 0)
```

Projetí definovanou cestou

Pro projetí nejkratší cestou potřebujeme funkci, která nám projede definovanou cestou. Cestu uložíme jako desítkové číslo, kde na každé pozici bude uložen směr, kterým se má vydat na křižovatce. Příklad čísla 221121881. Jako první číslo převedeme do stringu, následně číslo převedeme ze směřování na sever do směru robota a za každý upravený prvek provedeme akci přejetí na další křižovatku podle jeho hodnoty.

```
def Ride_Path(path):
      global _position_absolute

      # Převedeme path na string a provádíme for dokud ho
neprojdeme.
      for bin_cross in str(path):
      # Otočíme hodnotu křižovatky na směr robota
            bin_cross = Turn_Bin_Number(int(bin_cross), 4 -
      _position_absolute[2])

      # Pokud je cesta vpravo.
```

```
        if bin_cross == 1:
# Uprav směr robota o -1.
            _position_absolute[2] =
        Direction_Change(_position_absolute[2] - 1)
# Otoč do prava a pojeď na další křižovatku.
            Robot_Turn("right")
            Travel_To_Next_Crossroads()

# Pokud je cesta rovně.
        if bin_cross == 2:
# Jeď na další křižovatku
            Travel_To_Next_Crossroads()

# Pokud je cesta rovně.
        if bin_cross == 4:
# Uprav směr robota o +1.
            _position_absolute[2] =
        Direction_Change(_position_absolute[2] + 1)
# Otoč robota do prava a pojeď na další křižovatku.
            Robot_Turn("left")
            Travel_To_Next_Crossroads()
# Pokud je cesta za robotem.
        if bin_cross == 8:
# Uprav směr robota o +2.
            _position_absolute[2] =
        Direction_Change(_position_absolute[2] + 2)
# Otoč robota do zadu a pojeď na další křižovatku.
            Robot_Turn("back")
            Travel_To_Next_Crossroads()
```

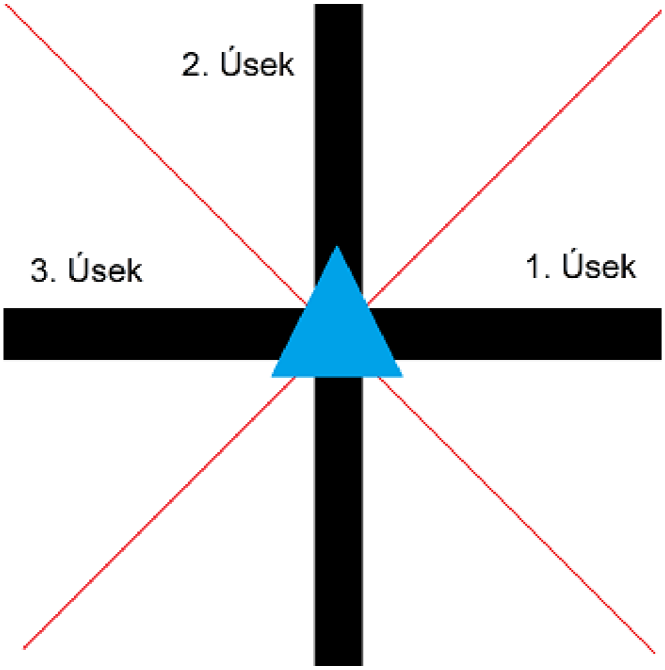
Výběr cesty

Cesty na křižovatce

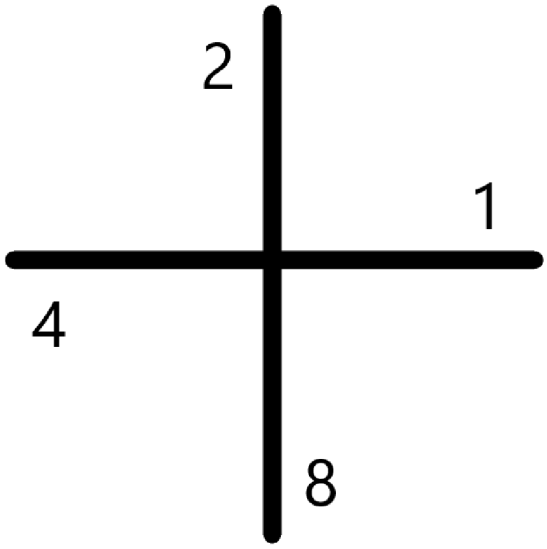
Křižovatku si rozdělíme do čtyř úseků o velikosti 90 stupňů. Kde se střed každého úseku bude nacházet na středu cesty. Jelikož robot vždy přijede po cestě, provedeme snímání zbylých tři úseků. Při psaní programu na snímání si musíme dát pozor na umístění kabelu na senzoru, aby se nezamotal, neskřípnul nebo měl dostatečnou délku. Z tohoto důvodu začínáme snímat od pravé části křižovatky. Pokud se na jakémkoliv místě v 90 stupňovém úseku nalezne černá, označíme že v tomto úseku se nachází křižovatka.

Zároveň v cyklu zkontrolujeme, jestli se černá nachází mezi úseky. Pokud se nachází, znamená to, že se robot vyskytuje v cíli, kde počkáme 10 sekund pro zapsání času dojetí do cíle.

Následně umístění cest uložíme do jedné proměnné pomocí binárních čísel. Do proměnné uložíme součet čísel, které představují desítková čísla 1, 2, 4 a 8, kdy číslo jedna nám označuje směr vpravo a číslo 8 směr za robotem. Číslo 8 přičteme vždy jelikož robot z této cesty přijel.



Obrázek 10: Rozdělení křižovatky do čtyř úseků (Zdroj autor)



Obrázek 11: Číselné ohodnocení křižovatky (Zdroj autor)

Tabulka 1: Převod desítkové soustavy do binární

Desítková soustava	Složení	Binární soustava
1	1	1
2	2	10
4	4	100
8	8	1000
9	8+1	1001
10	8+2	1010
11	8+2+1	1011
12	8+4	1100
13	8+4+1	1101
14	8+4+2	1110
15	8+4+2+1	1111

```
def Scan_Crossroads():
      global _senzor, _senzor_motor, _black, _position_finish,
_position_absolute
```

```
    # Pole pro zapisování cest kde na první pozici se nachází
levá křižovatka.
    paths = [0,0,0]
    # Otočení motoru držícího barevný senzor na pozici absolutní
nuly.
    Return_To_Absolute(_senzor_motor)

    # Pohyb senzoru na začátek levého úseku.
    motor.run_for_degrees(_senzor_motor, -135, 2000, stop = 2)
    time.sleep_ms(500)

    # Vynulování relativní pozice motoru, který pohybuje barevný
senzor
    motor.reset_relative_position(_senzor_motor, 0)
    # Spuštění pohybu motoru s barevným senzorem o velikost třech
dílů
    motor.run_for_degrees(_senzor_motor, 280, 1000, stop = 2)

    i = 1
    finish = 0
    # Proveď 3x
    while i < 4:
        motor_position = motor.relative_position(_senzor_motor)
    # Pokud se nachází černá barva na začátku úseku, přičti 1 k
hodnotě finish.
        if color_sensor.reflection(_senzor) < _black:
            finish = finish + 1
    # Provádí dokud je relativní poloha motoru menší než velikost
úseku krát pozice úseku zleva(1-3).
        while motor_position < 90 * i:
# Pokud senzor nalezne černou barvu, uložíme do proměnné paths
relativní polohu motoru na pozici úseku.
            if color_sensor.reflection(_senzor) < _black:
                paths[i - 1] = 1

            motor_position =
        motor.relative_position(_senzor_motor)
        i = i + 1
```

```
    # Převedení cesty v poli na binární hodnotu.
    paths_bin = 0
    # Pokud je cesta v prvním úseku přičteme 1.
    if paths[0]:
        paths_bin = paths_bin + 1
    # Pokud je cesta v druhém úseku přičteme 2.
    if paths[1]:
        paths_bin = paths_bin + 2
    # Pokud je cesta v třetím úseku přičteme 4.
    if paths[2]:
        paths_bin = paths_bin + 4

    # Pokud jsme zaznamenali černou mezi úseky třikrát,
zaznamenáme hodnotu 2048 pro cíl a počkáme 10 sekund podle pravidel
na zaznamenání dojetí do cíle.
    if finish == 3:
        paths_bin = 2048
        time.sleep_ms(10000)#cíl

    # Do cesty přičteme 8 pro zpáteční cestu která se vždy
nachází.
return paths_bin + 8
```

Vyhodnocení cest v křižovatce

Na každé křižovatce potřebujeme vyhodnotit cesty na křižovatce, uložit je do globální proměnné na aktuální pozici robota po bitovém otočení na sever a výběr další křižovatky na kterou robot pojede. Spustíme funkci pro naskenování křižovatky `Scan_Crossroads()`. Výsledek bitově otočíme a uložíme do _maze. Pokud se nacházíme v cíli upravíme promněnnou `_stop` a `_position_finish`, která ukončí cestování robota.

Pro výběr další křižovatky zjistíme jestli se k ní nachází cesta, když se nachází, zkontrolujeme jestli jsme na ní byly, pokud ne přejedeme na ni a ukončíme funkci. Pokud ano vybereme další křižovatku s cestou. Pro zjištění jestli se v křižovatce nachází cesta použijeme operátor &, který zjistí jestli se v čísle nachází stejné binární číslo. Pokud jsme byly na všech okolních křižovatkách, spustíme `BFS()` funkci s algoritmem pro vyhledání

nejbližší neprojeté křižovatky. Vrácené hodnoty uložíme jako novou pozici robota a spustíme funkci `Ride_Path()` pro projetí cesty k této křižovatce.

```
def Select_Path():
      global _position_absolute, _maze, _stop, _position_finish
      # Spustíme funkci Scan_Crossroads a vrácené hodnoty nahrajeme
do promněnné paths.
      paths=Scan_Crossroads()
      # Hodnotu paths bitově otočíme do směru 0 (sever) a uložíme
ji do globální promněnné _maze na souřadnice robota.
      _maze[_position_absolute[0]][_position_absolute[1]] =
Turn_Bin_Number(paths, _position_absolute[2])

      # Pokud je paths větší než 15, robot je v cíli.
      if paths>15:
      # Upravíme promněnnou pro zastavení hledání.
            _stop = True
      # Uložíme pozici robota do globální promněnné pro pozici
cíle.
            _position_finish[0] = _position_absolute[0]
            _position_finish[1] = _position_absolute[1]
      # Ukončíme funkci.
            return
      # Pokud je cesta vpravo.
      if paths & 1:
      # Ulož do next_c pozici křižovatky vpravo.
            next_c = Update_Robot_Position(-1, _position_absolute)
      # Pokud v _maze na pozici next_c je nula.
            if not(_maze[next_c[0]][next_c[1]]):
      # Ulož pozici křižovatky vpravo do pozice robota.
                  _position_absolute = next_c
      # Otoč se do prava a přejeď na další křižovatku.
                  Robot_Turn("right")
                  Travel_To_Next_Crossroads()
                  return
      # Pokud je cesta v rovně.
      if paths & 2:
            # Ulož do next_c pozici křižovatky před robotem.
            next_c = Update_Robot_Position(0, _position_absolute)
```

```
            # Pokud v _maze na pozici next_c je nula.
            if not(_maze[next_c[0]][next_c[1]]):
                  # Ulož pozici křižovatky vpravo do pozice robota.
                  _position_absolute = next_c
      # Přejeď na další křižovatku.
                  Travel_To_Next_Crossroads()
                  return
      # Pokud je cesta vlevo.
      if paths & 4:
      # Ulož do next_c pozici křižovatky vlevo.
            next_c = Update_Robot_Position(1, _position_absolute)
      # Pokud v _maze na pozici next_c je nula.
            if not(_maze[next_c[0]][next_c[1]]):
      # Ulož pozici křižovatky vpravo do pozice robota.
                  _position_absolute = next_c
      # Otoč se doleva a přejeď na další křižovatku.
                  Robot_Turn("left")
                  Travel_To_Next_Crossroads()
                  return
      # Spuštění funkce pro vyhledání nejkratší cesty k
neprohledané křižovatce a uložení výsledné hodnoty do proměnné
best_path_result.

      best_path_result = [0,0,0]
      best_path_result = BFS()


      # Uložení souřadnic nejbližší neprojeté křižovatky do
souřadnic robota.
      _position_absolute[0] = best_path_result[0]
      _position_absolute[1] = best_path_result[1]
      # Spuštění funkce pro projetí cesty s výsledkem funkce BFS.
      Ride_Path(best_path_result[2])
      Return
```

Úprava informací

Kompas robota (přetečení hodnot)

Vnitřní kompas funguje na principu očíslování směru od 0 po 3, kdy 0 nám označuje natočení robota na sever a uložení do globální proměnné. Při každém otočení

směr tedy změníme. Při otočení doprava odečteme jedničku, otočení doleva jedničku přičteme. Aby nedošlo k případům, kdy hodnota směru bude větší než tři nebo menší než nula si vytvoříme funkci která nám hodnotu směru bude udržovat v hodnotách od nuly do tří.

```
def Direction_Change(direction):

    if direction > 3:
        direction = direction - 4
    elif direction < 0:
        direction = direction + 4

    return direction

# Pro uložení směřování robota si vytvoříme globální
promněnnou ve které první dvě pozice použijeme pro uložení x
a y souřednice robota a poslední pozici sa směr robota.
_position_absolute = [0,0,0]
# Při přepisování směru zavoláme funkci pro zamezení
přetečení hodnot.
_position_absolute = Direction_Change(_position_absolute[2] +
change)
```

Binární otočení

Pokud chceme ukládat cesty do globální proměnné je nutné je mít všechny ve stejném směru. Vytvoříme tedy funkci která binární číslo otočí o určený směr. Funkci budeme používat při ukládání cest na křižovatce, kdy směr bude na sever, tak i u čtení kdy číslo budeme ze severu otáčet na aktuální směr robota.

Tabulka 2: Binární otočení čísla

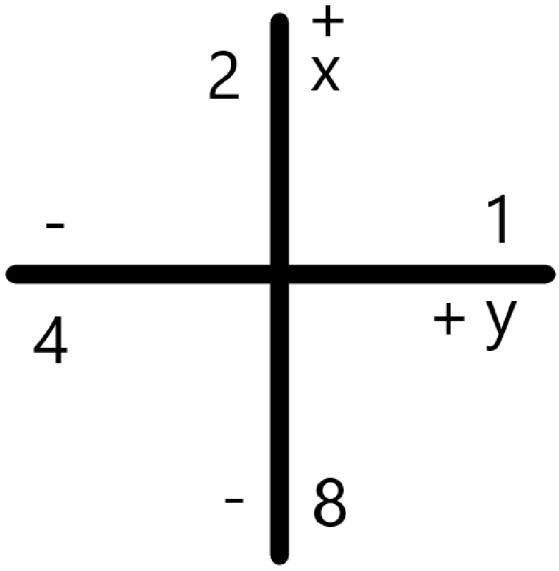
Lokální				Globální		
Směr Robota	Desítková	Složení	Binární	Desítková	Složení	Binární
1	1	1	0001	8	8	1000
3	1	1	0001	2	2	0010

2	12	8+4	1100	3	2+1	0011
0	14	8+4+2	1110	14	8+4+2	1110

```
def Turn_Bin_Number(bin_num, direction):

      turn = 0
      # Prováděj dokud je hodnota turn menší než směr robota.
      while turn < direction:
      # Zvěč hodnotu bin_num 2x.
            bin_num = bin_num * 2
            turn = turn + 1
      # Pokud je hodnota bin_num větší než 15 tak hodnotu o 15
zmenšíme.
            if(bin_num > 15):
                  bin_num = bin_num - 16
                  bin_num = bin_num + 1
      # Vrátíme upravené číslo.
      return bin_num
```

Úprava pozice robota

Při pohybu robota je nutná změna souřadnic robota. Ovšem směr robota nám v tomto dělá neplechu, kdy při cestě rovně nesměřuje vždy na sever. Vytvoříme si funkci, která nám podle směru a pozice robota provede vrácení (return) souřadnice další křižovatky. Jako první zjistíme, kam bude robot směřovat a podle této hodnoty upravíme souřadnice.



Obrázek 12: Číselné ohodnocení křižovatky směřující na sever (Zdroj autor)

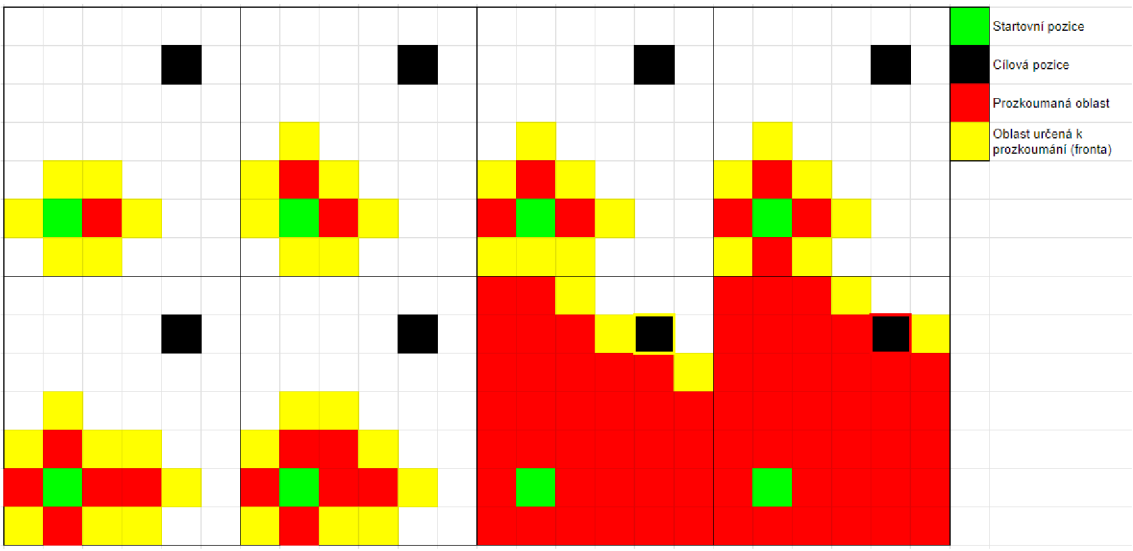
```
def Update_Robot_Position(way, position):
      # Změna směru uloženém v position o zadaný směr(way).
      position[2] = Direction_Change(position[2] + way)

      # Podle směru v position[2] změníme souřadnice. V position[0]
se nachází x souřadnice a v position[1] y souřadnice.
      # Pokud je směr nula, přičteme 1 k x souřadnici.
      if position[2] == 0:
            position[0] = position[0] + 1
      # Pokud je směr jedna, odečteme 1 od y souřadnice.
      if position[2] == 1:
            position[1] = position[1] - 1
      # Pokud je směr jedna, odečteme 1 od x souřadnice.
      if position[2] == 2:
            position[0] = position[0] - 1
      # Pokud je směr nula, přičteme 1 k y souřadnici.
      if position[2] == 3:
            position[1] = position[1] + 1
      return position
```

Výpočet nejkratší cesty

BFS – Vyhledání cesty k nejbližší neprojeté křižovatce

Breadth-First Search je algoritmus, který se snaží najít nejkratší cestu pomocí postupného procházení sousedních křižovatek. Algoritmus používá frontu, do které si uloží neprošlé sousedící křižovatky a set už prozkoumaných křižovatek pro zamezení jejich procházení. Algoritmus probíhá, dokud jsou prvky ve frontě nebo jsme došli do cílové hodnoty. (Laparra, 2019)

V naší funkci hledáme nejbližší křižovatku neprojetou robotem. Pro aktuální křižovatku si vytvoříme promněnou, ve které je uložena její souřadnice a cesta do této křižovatky. Cestu uložíme pomocí čísel 1, 2, 4, 8, které zapisujeme stejně jako hodnoty křižovatky při směru na sever. Vytvořenou proměnou uložíme do fronty a začneme vyhodnocovat sousední křižovatky, dokud není fronta prázdná. Při vyhodnocování křižovatky, křižovatku odebereme z první pozice fronty a nahrajeme ji do proměnné position. Zkontrolujeme, jestli robot na křižovatce byl, pokud ano skončíme funkci a vrátíme proměnnou position. Následně zkontrolujeme, jestli se nachází cesta k sousední křižovatce, pokud se nachází vytvoříme si proměnnou s upravenou pozicí křižovatky a cestou k ní. Proměnnou poté zkontrolujeme, jestli se nachází ve frontě nebo setu zpracovaných křižovatek. Pokud se nachází, uložíme ji do fronty. Pro projití všech sousedních křižovatek proměnnou position uložíme do setu zpracovaných křižovatek.



Obrázek 13:Průchod algoritmu BFS (Zdroj autor)

```
def BFS():
```

```
    global _maze, _position_absolute, _stop

    finished = []
    check = []

    # Do fronty check uložíme hodnotu pozici na které robot
stojí.
    tem = [0,0,0]
    tem[0] = _position_absolute[0]
    tem[1] = _position_absolute[1]
    check.append(tem.copy())

    # Provádíme dokud se nacházejí prvky v proměnné check.
    while check:
    # Vyjmeme první prvek z fronty check a nahrajeme ho do
promněnné position.
        position = check.pop(0)
    # Z _maze nahrajeme informace o křižovatce do paths.
        paths = _maze[position[0]][position[1]]
    # Pokud je v paths 0, funkce vrátí promněnnou position.
        if not(paths):
            return position
    # Pokud na křižovatce nachází cesta na východ.
        if paths & 1:
    # Uložíme prvek s pozicí východní křižovatky.
            tem[0] = position[0]
            tem[1] = position[1] + 1
    # Na poslední pozici aktuální cestu * 10 + 1.
            tem[2] = position[2] * 10 + 1
    # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem,
        finished):
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na sever.
        if paths & 2:
    # Uložíme prvek s pozicí severní křižovatky.
            tem[0] = position[0] + 1
            tem[1] = position[1]
```

```
            # Na poslední pozici aktuální cestu * 10 + 2.
                        tem[2] = position[2] * 10 + 2
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                        if In_Front(tem, check) and In_Front(tem,
                  finished):
                              check.append(tem.copy())
            # Pokud na křižovatce nachází cesta na západ.
                  if paths & 4:
            # Uložíme prvek s pozicí západní křižovatky.
                        tem[0] = position[0]
                        tem[1] = position[1] - 1
            # Na poslední pozici aktuální cestu * 10 + 4.
                        tem[2] = position[2] * 10 + 4
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                        if In_Front(tem, check) and In_Front(tem,
                  finished):
                              check.append(tem.copy())
            # Pokud na křižovatce nachází cesta na jih.
                  if paths & 8:
            # Uložíme prvek s pozicí jižní křižovatky.
                        tem[0] = position[0] - 1
                        tem[1] = position[1]
            # Na poslední pozici aktuální cestu * 10 + 8.
                        tem[2] = position[2] * 10 + 8
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                        if In_Front(tem, check) and In_Front(tem,
                  finished):
                              check.append(tem.copy())
            # Prošlý prvak uložíme do fronty finished.
                  finished.append(position)
            # Pokud nenalezne žádné neprohledané křižovatky, uložíme True
do globální promněnné _stop.
            _stop = True
```

Obsah souřadnic ve frontě

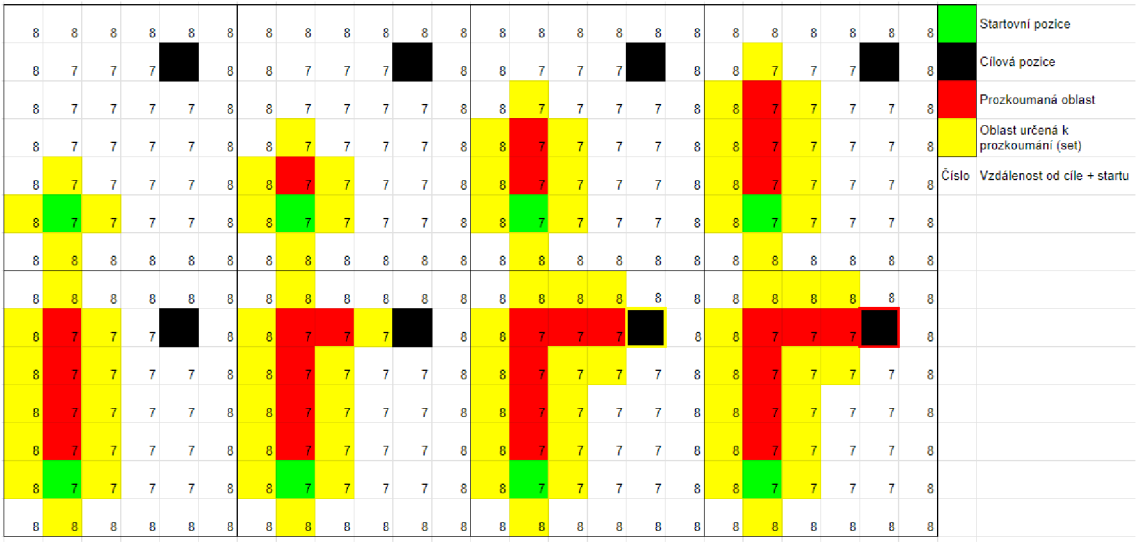
Pokud používáme vyhledávací algoritmy s frontou, potřebujeme zamezit případům s duplikací prvků ve frontě. Vytvoříme si tedy funkci, kterou použijeme před uložením prvku do fronty. Funkce zkontroluje, jestli se prvek už ve frontě nachází, vrátí False a pokud se nenachází vrátí True.

```
def In_Front(position, front):
      # Pro každý prvek ve frontě zkontroluji jestli se nachází
prvek se stejnou pozicí jako position.
      for i in front:
            if position[0] == i[0] and position[1] == i[1]:
      # Pokud se nachází vrátíme False.
                  return False
      return True
```

A-star – Vyhledání nejkratší cesty k cílové souřadnici

A-star algoritmus se snaží najít nejkratší cestu pomocí opakovaného prozkoumávání nejslibnější neprozkoumané křižovatky. Slibnost křižovatky vypočítá sečtením vzdálenosti od cíle a začáteční pozice prohledávání. Algoritmus skončí, když se aktuální pozice nachází v cíli nebo prozkoumal všechny sousedy. (Cui, Shi, 2010)

Funkce je podobná funkci pro BFS. Rozdíl mezi funkcemi je vyjmutí prvku z fronty a podmínka pro ukončení funkce. Z fronty vyjmeme nejslibnější prvek pro který nám jeho pozici vrátí funkce Heuristic_Distance(check, finish).



Obrázek 14: Průchod algoritmu A (Zdroj autor)*

```
def Astar(finish):
      global _maze, _position_absolute

      finished = []
      check = []
      tem = [0,0,0]
      # Do fronty check uložíme hodnotu pozici na které robot
stojí.
      tem[0] = position_absolute[0]
      tem[1] = position_absolute[1]
      check.append(tem.copy())

      # Provádíme dokud se nacházejí prvky v proměnné check.
      while check:
      # Vyjmeme prvek který je nejblíže k cíli z fronty check a
nahrajeme ho do proměnné position.
            position = check.pop(Heuristic_Distance(check, finish))

            paths = _maze[position[0]][position[1]]
      # Pokud pozice cíle je moje aktuální pozice, vrátíme
proměnnou position.
            if finish[0] == position[0] and finish[1] ==
      position[1]:
                  return position
```

```
        # Pokud na křižovatce nachází cesta na východ.
            if paths & 1:
        # Uložíme prvek s pozicí východní křižovatky.
                tem[0] = position[0]
                tem[1] = position[1] + 1
        # Na poslední pozici aktuální cestu * 10 + 1.
                tem[2] = position[2] * 10 + 1
        # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                if In_Front(tem, check) and In_Front(tem,
            finished):
                    check.append(tem.copy())
        # Pokud na křižovatce nachází cesta na sever.
            if paths & 2:
        # Uložíme prvek s pozicí severní křižovatky.
                tem[0] = position[0] + 1
                tem[1] = position[1]
        # Na poslední pozici aktuální cestu * 10 + 2.
                tem[2] = position[2] * 10 + 2
        # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                if In_Front(tem, check) and In_Front(tem,
            finished):
                    check.append(tem.copy())
        # Pokud na křižovatce nachází cesta na západ.
            if paths & 4:
                # Uložíme prvek s pozicí západní křižovatky.
                tem[0] = position[0]
                tem[1] = position[1] - 1
        # Na poslední pozici aktuální cestu * 10 + 4.
                tem[2] = position[2] * 10 + 4
        # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                if In_Front(tem, check) and In_Front(tem,
            finished):
                    check.append(tem.copy())
        # Pokud na křižovatce nachází cesta na jih.
            if paths & 8:
        # Uložíme prvek s pozicí jižní křižovatky.
```

```
                tem[0] = position[0] - 1
                tem[1] = position[1]
        # Na poslední pozici aktuální cestu * 10 + 8.
                tem[2] = position[2] * 10 + 8
        # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
                if In_Front(tem, check) and In_Front(tem,
            finished):
                    check.append(tem.copy())
        # Prošlý prvak uložíme do fronty finished.
            finished.append(position)
```

Heuristic

Pokud používáme Astar algoritmus potřebujeme najít prvek ve frontě, který je nejblíže k cílové hodnotě. Jelikož robot nemůže cestovat diagonálně stačí nám pouze jednoduchý součet rozdílu mezi x a y souřadnicí cíle a zvoleného prvku. Pozici prvku následně tato funkce vrátí.

```
def Heuristic_Distance(front, finish):
        k = 0
        select_distance = 1000
        select = 0
        # Projdeme všechny prvky ve frontě.
        for position in front:
        # Sečteme rozdíl mezi x souřadnicí cíle a pozice prvku ve
frontě a y souřadnicí cíle a pozice prvku ve frontě.
            distance = abs(finish[0] - position[0]) + abs(finish[1]
        - position[1])
        # Dále přičteme vzdálenost aktuální pozice robota od
startovní pozice robota.
            distance = distance + position[0] + position[1]
        # Pokud je vypočítaná vzdálenost menší než vybraná.
            if select_distance > distance:
        # Přepíšeme vybranou za vypočítanou a uložíme pozici ve
frontě do proměnné select.
                select_distance = distance
                select = k
            k = k + 1
        # Ve funkci vrátíme pozici prvku nejblíže k cíli ve frontě.
```

```
        return select
```

Spuštění robota

Mapu projedeme třikrát. Pro první projetí robot počká na stlačení tlačítka pro start. Po startu robot přejede na další křižovatku před sebou a upraví svou pozici. Následně spustíme cyklus pro prohledávání křižovatek, dokud není splněna podmínka pro zastavení. Podmínka je změna hodnoty `_stop`, která se změní při nalezení cíle nebo zmapování všech křižovatek. V druhém projetí robot zase počká na stlačení tlačítka, po kterém vynuluje pozici robota, spustí algoritmus pro vyhledání nejkratší cesty k neprohledané křižovatce a projede k ní. Následně pokračuje v mapování křižovatek, dokud neprohledá celé bludiště. V poslední části počkáme na spuštění tlačítkem. Vynulujeme pozici robota. Spustíme funkci pro vyhledání nejkratší cesty k cíli pomocí Astar algoritmu a následné projetí vyhledané cesty.

```
# Čekej na stlačení tlačítka
_maze[0][0]=2
while force_sensor.force(_force)==0:
      pass
Travel_To_Next_Crossroads()
position_absolute[0] = 1
while not(_stop):
      Select_Path()

# Čekej na stlačení tlačítka
while force_sensor.force(_force)==0:
      pass

position_absolute = [0,0,0]
best_path_result = BFS()
position_absolute[0] = best_path_result[0]
position_absolute[1] = best_path_result[1]
Ride_Path(best_path_result[2])
_stop = False

while not(_stop):
      Select_Path()

# Čekej na stlačení tlačítka
```

```
while force_sensor.force(_force) == 0:
      pass

position_absolute = [0,0,0]
best_path_result = Astar(_position_finish)

position_absolute[0] = best_path_result[0]
position_absolute[1] = best_path_result[1]
Ride_Path(best_path_result[2])
```

ZÁVĚR

V teoretické část práce jsem popsal programovací jazyk Python a jeho výhody. Dále jsem se zabýval popisem algoritmu, algoritmickým myšlením a zapisováním algoritmů. Nejvíce jsem se zabýval algoritmem na vyhledání nejkratší cesty v grafu. Dále jsem provedl zmapování nejčastěji používaných robotických stavebnic a provedl podrobný popis stavebnice LEGO SPIKE Prime, se které jsem následně sestavil soutěžního robota.

V praktické části práce jsem provedl samotné sestavení soutěžního robota a jeho naprogramování. Poté jsem jednotlivé funkce rozdělil do čtyř kategorií podle jejich funkce a podrobně popsal jejich fungování a místo v programu. Během realizace jsem narazil na problémy s vůlí motorů, u hnacích motorů robota jsem tomuto jevu zamezil pomocí převodu na menší rychlost, ovšem z důvodu umístění a pohybové oblasti převod nešel namontovat na poslední motor který pohybuje barevným senzorem. Musel jsem tedy s tímto chováním počítat v programu. Dalším problémem byla dokumentace k programování LEGO SPIKE, kdy verze 3.0 je celkem nová. Není k ní tedy rozsáhlá dokumentace a není zpětně kompatibilní s verzí 2.0. Dále jsem provedl praktickou zkoušku na mnou sestaveném soutěžním robotu. Tato zkouška spočívala v projetí čárového bludiště, jeho zmapování, vyhodnocení nejkratší cesty do cíle a následném projetí zjištěné cesty.

Zkouška robota proběhl úspěšně. Robot zmapoval všechny projeté křižovatky a našel nejkratší cestu do cíle.

SEZNAM POUŽITÝCH ZDROJŮ

Literatura

KOPECKÝ, Kamil, René SZOTKOWSKI, Lukáš KUBALA, Veronika KREJČÍ a Martin HAVELKA. Moderní technologie ve výuce: (o moderních technologiích ve výuce s pedagogy pro pedagogy). Olomouc: Univerzita Palackého v Olomouci, Pedagogická fakulta, 2021. ISBN 978-80-244-5925-7.

CHUN, Wesley J. Core Python Programming. Prentice Hall, 2007. ISBN 0-13-226993-7.

ERICKSON, Jeff. *Algorithms*. Illinois: Jeff Erickson, 2019. ISBN 978-1-792-64483-2.

NEAPOLITAN, Richard E. *Foundations of algorithms.* 5. vydání. Burlington: Jones & Bartlett Learning, 2015. ISBN 978-1-284-04919-0.

CHAUDHURI, A. B. *Flowchart and Algorithm Basics: The Art of Programming.* Dulles: Mercury Learning and Information ISBN: 978-1-68392-537-8

HEINEMAN, George T., Gary Pollice, a Stanley Selkow. *Algorithms in a Nutshell.* 2. vydání. Sebastopol: O'Reilly Media, 2016. ISBN: 978-1-491-94892-7

ROD, Stephens. *Essential Algorithms: A Practical Approach to Computer Algorithms*. Indianapolis: John Wiley & Sons, 2013. ISBN 978-1-118-61210-1.

PROKOP, Jiří. *Algoritmy v jazyku C a C++. 3.*, aktualizované a rozšířené vydání. Praha: Grada, 2015. ISBN 978-80-247-5467-3.

PŠENČÍKOVÁ, Jana. *Algoritmizace.* Vyd. 2. Kralice na Hané: Computer Media, 2009. ISBN 978-80-7402-034-6.

SUMMERFIELD, Mark. *Python 3: výukový kurz*. 2. vydání. Brno: Computer Press, 2021. ISBN 978-80-251-5030-6.

PECINOVSKÝ, Rudolf. *Python: kompletní příručka jazyka pro verzi 3.8*. První vydání. Praha: Grada Publishing, 2020. Myslíme v. Knihovna programátora. ISBN 978-80-271-2891-4.

LUTZ, Mark a ASCHER, David. *Naučte se Python: pohotová příručka*. 1. vyd. Praha: Grada Publishing, 2003. ISBN 80-247-0367-X.

Elektronické zdroje

BLAHO, Andrej, Ľubomír SALANCI a Václav ŠIMANDL. *Programování v jazyce Python pro střední školy* [online]. Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta, 2020. ISBN 978-80-7394-784-2. Dostupné také z: https://imysleni.cz/ucebnice/zaklady-programovani-v-jazyce-python-pro-stredni-skoly

LEGO education [online]. Denmark: LEGO System A/S, 2019 [cit. 2023-06-03]. Dostupné z: https://spike.legoeducation.com/

BENNETT, Nicholas. *Introduction to Algorithms and Pseudocode* [online]. Berlín: ResearchGate, 2015 [cit. 2023-05-18]. Dostupné z: https://www.researchgate.net/publication/309410533_Introduction_to_Algorithms_and_Pseudocode

CUI, Xiao a Hao SHI. *A*-based Pathfinding in Modern Computer Games* [online]. Berlín: ResearchGate, 2010 [cit. 2023-05-18]. Dostupné z: https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games

LEGO [online]. Denmark: LEGO System A/S, 2022 [cit. 2023-05-18]. Dostupné z: https://www.lego.com/en-us

The brothers brick [online]. Seattle: The Brothers Brick, 2020 [cit. 2023-05-18]. Dostupné z: https://www.brothers-brick.com

OITZMAN, Mike. *LEGO discontinues Mindstorms product line* [online]. Cleveland: WTWH Media, 2022 [cit. 2023-05-18]. Dostupné z: https://www.therobotreport.com/lego-discontinues-mindstorms-product-line/

LEGO® Education 45678 SPIKE™ Prime Set [online]. Praha: Růžovka.cz, 2023 [cit. 2023-05-18]. Dostupné z: https://ruzovka.cz/cs/2-stupen-zs-vxs/16469-lego-education-45678-spike-prime-set.html

RoboTrip [online]. Olomouc: Radim Děrda, 2023 [cit. 2023-05-18]. Dostupné z: http://robotrip.cz/

45678 LEGO® Education SPIKE™ Prime Základní souprava [online]. Velké Pavlovice: EDUXE, 2023 [cit. 2023-05-18]. Dostupné z: https://www.eduxe.cz/p/353/45678-spike-prime-zakladni-souprava

LEGO® Education SPIKE™ Prime – rozšiřující sada s podložkou [online]. Český Těšín: insgraf.cz, 2023 [cit. 2023-05-18]. Dostupné z: https://insgraf.cz/1121473/LEGO-Education-SPIKE-Prime-rozsirujici-sada-s-podlozkou/LEGO45681

WATTERS, Audrey. *Lego Mindstorms: A History of Educational Robots* [online]. New York: Audrey Watters, 2015 [cit. 2023-05-18]. Dostupné z: https://hackeducation.com/2015/04/10/mindstorms

VEX Robotics [online]. Greenville: VEX Robotics, 2023 [cit. 2023-05-18]. Dostupné z: https://www.vexrobotics.com/

Fischertechnik Robotics TXT 4.0 Base Set [online]. Bethel: Educational Innovations, 2023 [cit. 2023-05-18]. Dostupné z: https://www.teachersource.com/product/fischertechnik-robotics-txt-4-base-set?fbclid=IwAR1fm418FGyGYU-

Makeblock education [online]. Irwindale: Makeblock, 2023 [cit. 2023-05-18]. Dostupné z: https://education.makeblock.com

Robotická stavebnice m-BITBEAM [online]. Polička: Tomáš Feltl – TFSoft, 2023 [cit. 2023-05-18]. Dostupné z: https://www.e-mole.cz/diy/m-bitbeam?fbclid=IwAR1WS59HUWPP-JqP7sFXaPbnX5c4xp8yEmciTaS5f_NKIZ7QSg3_SDYjukc

First LEGO League [online]. Bedford: FIRST, 2023 [cit. 2023-05-18]. Dostupné z: https://www.firstlegoleague.org/

LAPARRA, Daniel Monzonís. *PATHFINDING ALGORITHMS IN GRAPHS AND APPLICATIONS* [online]. Barcelona: Universitat de Barcelona, 2019 [cit. 2023-05-18]. Dostupné z: https://diposit.ub.edu/dspace/bitstream/2445/140466/1/memoria.pdf

3.11.4 Documentation [online]. Delaware: Python Software Foundation, 2001 [cit. 2023-05-18]. Dostupné z: https://docs.python.org/

Kvalifikační práce

JANDEJSEK, Rostislav. *Inovace elektronické řídicí jednotky RCX*. Plzeň, 2013. Diplomová práce. Západočeská univerzita v Plzni, Fakulta elektrotechnická, Katedra aplikované elektrotechniky a telekomunikací. Vedoucí práce Ing. Petr Weissar Ph.D.

REICH, Jakub. *Ovládání laboratorního modelu Mindstorms NXT (spike) pomocí PC*. Zlín, 2008. Diplomová práce. Univerzita Tomáše Bati ve Zlíně, Fakulta aplikované informatiky. Vedoucí práce Ing. Roman Šenkeřík

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

Příloha č. 1: Program pro disciplínu „Čárové bludiště“

```
from hub import port
import motor, color_sensor, force_sensor, time

# Components

_left_motor = port.B
_right_motor = port.F
_senzor = port.C
_senzor_motor = port.D
_force = port.A

# Global variables

_position_absolute = [0,0,0]
_position_finish = [0,0]
_black = 54
_stop = False
_wheel = 17.6
_gear = 36/12

# Generace dvourozměrného pole pro uložení bludiště

_maze = []
for i in range(12):
    temp = []
    for j in range(12):
        temp.append(0)
    _maze.append(temp)


def Direction_Change(direction):

    if direction > 3:
        direction = direction - 4
    elif direction < 0:
        direction = direction + 4

    return direction




def Return_To_Absolute(selected_motor):

    #Zjištění pozice motoru.
    position = motor.absolute_position(selected_motor)
```

```
    #Rozhodnutí kterým směrem se otočit.
    if position > 225:
    #Otočení po směru hodinových ručiček o zjištěnou polohu a nastavení
motoru na HOLD.
        motor.run_for_degrees(selected_motor, position, 400, stop = 2)
    else:
        motor.run_for_degrees(selected_motor, -1 * position, 400, stop =
2)
    time.sleep_ms(500)


def Travel_To_Next_Crossroads():
    global _wheel, _gear, _left_motor, _right_motor, _senzor_motor
    # Místní proměnné
    speed = 300
    ideal=60
    distance_cm = 15 #cm
    pi = 10

    # Výpočet vzdálenosti ve stupních.
    distance_degree = (distance_cm / _wheel) * 360 * _gear
    # Otočení motoru držícího barevný senzor na pozici absolutní nuly.
    Return_To_Absolute(_senzor_motor)

    # Reset relativní vzdálenosti na pravém i levém motoru.
    motor.reset_relative_position(_left_motor, 0)
    motor.reset_relative_position(_right_motor, 0)

    # Pokud je ujetá vzdálenost menší než ⅔ vypočítané úhlové vzdálenosti
od další křižovatky, provádíme korekci rychlosti motorů pro sledování
čáry.
    while (motor.relative_position(_right_motor) + (-1 *
motor.relative_position(_left_motor))) / 2 < (distance_degree / 3 * 2):

    # Uložení hodnoty odraženého světla od podložky.
        scan = color_sensor.reflection(_senzor)
    # Pro úpravu korekce p zjistím rozdíl mezi hodnotou scanu a ideální
hodnoty po které se chceme pohybovat. Vypočítaný rozdíl převedu do
absolutní hodnoty a vynásobím hodnotou pi, která určuje agresivitu
korekce.
        p = abs(scan-ideal) * pi
    # Pokud je naskenovaná hodnota blíže ke středu černé čáry než zadaná
ideální hodnota. Začnu se otáčet doleva o korekci p.
        if ideal > scan:
    # Zpomalení levého motoru o korekci p a zrychlení pravého motoru o
korekci p
            motor.run(_left_motor, (-1 * (speed - p)))
            motor.run(_right_motor, speed + p)
        else:
```

```
    # Zpomalení pravého motoru o korekci p a zrychlení levého motoru o
korekci p
            motor.run(_right_motor, speed - p)
            motor.run(_left_motor, -1 * (speed + p))

    # Dojezd poslední ⅓ zadané vzdálenosti.
    motor.run_for_degrees(_right_motor, int(distance_degree / 3 + 5),
speed)
    motor.run_for_degrees(_left_motor, int(distance_degree / 3 + 5) * -1,
speed)
    time.sleep_ms(800)



def Scan_Crossroads():
    global _senzor, _senzor_motor, _black, _position_finish,
_position_absolute

    # Pole pro zapisování cest, kde na první pozici se nachází levá
křižovatka.
    paths = [0,0,0]
    # Otočení motoru držícího barevný senzor na pozici absolutní nuly.
    Return_To_Absolute(_senzor_motor)

    # Pohyb senzoru na začátek levého úseku.
    motor.run_for_degrees(_senzor_motor, -135, 2000, stop = 2)
    time.sleep_ms(500)

    # Vynulování relativní pozice motoru, který pohybuje barevný senzor
    motor.reset_relative_position(_senzor_motor, 0)
    # Spuštění pohybu motoru s barevným senzorem o velikost třech dílů
    motor.run_for_degrees(_senzor_motor, 280, 1000, stop = 2)

    i = 1
    finish = 0
    # Proveď 3x
    while i < 4:
        motor_position = motor.relative_position(_senzor_motor)
    # Pokud se nachází černá barva na začátku úseku, přičti 1 k hodnotě
finish.
        if color_sensor.reflection(_senzor) < _black:
                finish = finish + 1
    # Prováděj dokud je relativní poloha motoru menší než velikost úseku
krát pozice úseku z leva(1-3).
        while motor_position < 90 * i:
    # Pokud senzor nalezne černou barvu, uložíme do proměnné paths
relativní polohu motoru na pozici úseku.
            if color_sensor.reflection(_senzor) < _black:
                paths[i - 1] = 1
```

```
            motor_position = motor.relative_position(_senzor_motor)
        i = i + 1
    print(paths)
    # Převedení cesty v poli na bitovou hodnotu.
    paths_bit = 0
    # Pokud je cesta v prvním úseku přičteme 1.
    if paths[0]:
        paths_bit = paths_bit + 1
    # Pokud je cesta v druhém úseku přičteme 2.
    if paths[1]:
        paths_bit = paths_bit + 2
    # Pokud je cesta v třetím úseku přičteme 4.
    if paths[2]:
        paths_bit = paths_bit + 4

    # Pokud jsme zaznamenaly černou mezi úseky třikrát, zaznamenáme
hodnotu 2048 pro cíl a počkáme 5 sekund podle pravidel na zaznamenání
dojetí do cíle.
    if finish == 3:
        paths_bit = 2048
        time.sleep_ms(10000)#cíl

    # Do cesty přičteme 8 pro zpáteční cestu která se vždy nachází.
    return paths_bit + 8



def Robot_Turn(way):
    global _right_motor, _left_motor, _senzor, _senzor_motor, _black
    # Rychlost dotáčení robota.
    speed = 200

    # Otočení motoru držícího barevný senzor na pozici absolutní nuly.
    Return_To_Absolute(_senzor_motor)

    # Otočení doprava.
    if way == "right":
        # Pojetí do zatáčky
        motor.run_for_degrees(_right_motor, 15, 200)
        motor.run_for_degrees(_left_motor, -15, 200)
        time.sleep_ms(500)
        motor.run(_left_motor, -2000)
        motor.run(_right_motor, -2000)
        time.sleep_ms(700)

    # Otočení do leva. Přetočení přes cestu.
    if way == "left":
```

```
        motor.run_for_degrees(_right_motor, -30, 200)
        motor.run_for_degrees(_left_motor, 30, 200)
        time.sleep_ms(500)
        motor.run(_left_motor, 2000)
        motor.run(_right_motor, 2000)
        time.sleep_ms(1500)

    # Otočení zpět
    if way == "back":
        motor.run(_left_motor, -2000)
        motor.run(_right_motor, -2000)
        time.sleep_ms(2000)

    # Začínáme točit v pravo.
    motor.run(_left_motor, -1 * speed)
    motor.run(_right_motor, -1 * speed)

    # Čekáme dokud se senzor nenachází na černé barvě.
    while color_sensor.reflection(_senzor) > _black:
        pass
    # Zastavíme otáčení robota.
    motor.run(_left_motor, 0)
    motor.run(_right_motor, 0)


def Turn_Bit_Number(bit_num, direction):

    turn = 0
    # Prováděj dokud je hodnota turn menší než směr robota.
    while turn < direction:
    # Zvěč hodnotu bit_num 2x.
        bit_num = bit_num * 2
        turn = turn + 1
        # Pokud je hodnota bit_num větší než 15 tak hodnotu o 15
zmenšíme.
        if(bit_num > 15):
            bit_num = bit_num - 15
    # Vrátíme upravené číslo.
    return bit_num

def Find(array):
    min_num = array[0]
    i = 0
    min_i= 0
    for num in array[1:]:
        i= i + 1
        if num < min_num:
            min_num = num
            min_i = i
```

```
    return min_i


def Update_Robot_Position(way, position):
    # Změna směru uloženém v position o zadaný směr(way).
    position[2] = Direction_Change(position[2] + way)

    # Podle směru v position[2] změníme souřadnice. V position[0] se
nachází x souřadnice a v position[1] y souřadnice.
    # Pokud je směr nula, přičteme 1 k x souřadnici.
    if position[2] == 0:
        position[0] = position[0] + 1
    # Pokud je směr jedna, odečteme 1 od y souřadnice.
    if position[2] == 1:
        position[1] = position[1] - 1
    # Pokud je směr jedna, odečteme 1 od x souřadnice.
    if position[2] == 2:
        position[0] = position[0] - 1
    # Pokud je směr nula, přičteme 1 k y souřadnici.
    if position[2] == 3:
        position[1] = position[1] + 1

    return position



def Ride_Path(path):
    global _position_absolute

    # Převedeme path na string a provádíme for dokud ho neprojdeme.
    for bit_cross in str(path):
    # Otočíme hodnotu křižovatky na směr robota
        bit_cross = Turn_Bit_Number(int(bit_cross), 4 -
_position_absolute[2])

    # Pokud je cesta v pravo.
        if bit_cross == 1:
    # Uprav směr robota o -1.
            _position_absolute[2] =
Direction_Change(_position_absolute[2] - 1)
    # Otoč do prava a pojeď na další křižovatku.
            Robot_Turn("right")
            Travel_To_Next_Crossroads()

    # Pokud je cesta rovně.
        if bit_cross == 2:
    # Jeď na další křižovatku
            Travel_To_Next_Crossroads()
```

```
    # Pokud je cesta rovně.
        if bit_cross == 4:
    # Uprav směr robota o +1.
            _position_absolute[2] =
Direction_Change(_position_absolute[2] + 1)
    # Otoč robota do prava a pojeď na další křižovatku.
            Robot_Turn("left")
            Travel_To_Next_Crossroads()
    # Pokud je cesta za robotem.
        if bit_cross == 8:
    # Uprav směr robota o +2.
            _position_absolute[2] =
Direction_Change(_position_absolute[2] + 2)
    # Otoč robota do zadu a pojeď na další křižovatku.
            Robot_Turn("back")
            Travel_To_Next_Crossroads()
```

```
def Select_Path():
    global _position_absolute, _maze, _stop, _position_finish
    # Spustíme funkci Scan_Crossroads a vrácené hodnoty nahrajeme do
promněnné paths.
    paths=Scan_Crossroads()
    # Hodnotu paths bitově otočíme do směru 0 a uložíme ji do globální
promněnné _maze na souřadnice robota.
    _maze[_position_absolute[0]][_position_absolute[1]] =
Turn_Bit_Number(paths, _position_absolute[2])

    # Pokud je paths větší než 15, robot je v cíli.
    if paths>15:
    # Upravíme promněnnou pro zastavení hledání.
        _stop = True
    # Uložíme pozici robota do globální promněnné pro pozici cíle.
        _position_finish[0] = _position_absolute[0]
        _position_finish[1] = _position_absolute[1]
    # Ukončíme funkci.
        return
    # Pokud je cesta vpravo.
    if paths & 1:
    # Ulož do next_c pozici křižovatky vpravo.
        next_c = Update_Robot_Position(-1, _position_absolute)
    # Pokud v _maze na  pozici next_c je nula.
        if not(_maze[next_c[0]][next_c[1]]):
    # Ulož pozici křižovatky vpravo do pozice robota.
```

```
            _position_absolute = next_c
    # Otoč se do prava a přejeď na další křižovatku.
            Robot_Turn("right")
            Travel_To_Next_Crossroads()
            return
    # Pokud je cesta v rovně.
    if paths & 2:
        # Ulož do next_c pozici křižovatky před robotem.
        next_c = Update_Robot_Position(0, _position_absolute)
        # Pokud v _maze napozici next_c je nula.
        if not(_maze[next_c[0]][next_c[1]]):
            # Ulož pozici křižovatky vpravo do pozice robota.
            _position_absolute = next_c
            # Přejeď na další křižovatku.
            Travel_To_Next_Crossroads()
            return
    # Pokud je cesta v levo.
    if paths & 4:
    # Ulož do next_c pozici křižovatky vlevo.
        next_c = Update_Robot_Position(1, _position_absolute)
        # Pokud v _maze napozici next_c je nula.
        if not(_maze[next_c[0]][next_c[1]]):
            # Ulož pozici křižovatky vpravo do pozice robota.
            _position_absolute = next_c
            # Otoč se do leva a přejeď na další křižovatku.
            Robot_Turn("left")
            Travel_To_Next_Crossroads()
            return

    # Spuštění funkce pro vyhledání nejkratší cesty k neprohledané
křižovatce a uložení výsledné hodnoty do proměnné best_path_result.

    best_path_result = [0,0,0]
    best_path_result = BFS()


    # Uložení souřadnic nejbližší neprojeté křižovatky do souřadnic
robota.
    _position_absolute[0] = best_path_result[0]
    _position_absolute[1] = best_path_result[1]
    # Spuštění funkce pro projetí cesty s výsledkem funkce BFS.
    Ride_Path(best_path_result[2])
    return



def In_Front(position, front):
    # Pro každý prvek ve frontě zkontroluji jestli se nachází prvek se
stejnou pozicí jako position.
```

```
    for i in front:
        if position[0] == i[0] and position[1] == i[1]:
    # Pokud se nachází vrátíme False.
            return False
    return True



def Min_Distance_From(front, finish):
    k = 0
    select_distance = 1000
    select=0
    # Projdeme všechny prvky ve frontě.
    for position in front:
        # Sečteme rozdíl mezi x souřadnicí cíle a pozice prvku ve frontě
a y souřednicí cíle a pozice prvku ve frontě.
        distance = abs(finish[0] - position[0]) + abs(finish[1] -
position[1])
        # Pokud je vypočítaná vzdálenost menší než vybraná.
        if select_distance > distance:
            # Přepíšeme vybranou za vypočítanou a uložíme pozici ve
frontě do promněnné select.
            select_distance = distance
            select = k
        k = k + 1
    # Ve funkci vrátíme pozici prvku nejblíže k cíli ve frontě.
    return select


def BFS():
    global _maze, _position_absolute, _stop

    finished = []
    check = []

    # Do fronty check uložíme hodnotu pozici, na které robot stojí.
    tem = [0,0,0]
    tem[0] = _position_absolute[0]
    tem[1] = _position_absolute[1]
    check.append(tem.copy())

    # Provádíme dokud se nacházejí prvky v proměnné check.
    while check:
        # Vyjmeme první prvek z fronty check a nahrajeme ho do proměnné
position.
        position = check.pop(0)
        # Z _maze nahrajeme informace o křižovatce do paths.
        paths = _maze[position[0]][position[1]]
```

```
        # Pokud je v paths 0, funkce vrátí proměnnou position.
        if not(paths):
            return position
    # Pokud na křižovatce nachází cesta na východ.
        if paths & 1:
            # Uložíme prvek s pozicí východní křižovatky.
            tem[0] = position[0]
            tem[1] = position[1] + 1
            # Na poslední pozici aktuální cestu * 10 + 1.
            tem[2] = position[2] * 10 + 1
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na sever.
        if paths & 2:
            # Uložíme prvek s pozicí severní křižovatky.
            tem[0] = position[0] + 1
            tem[1] = position[1]
            # Na poslední pozici aktuální cestu * 10 + 2.
            tem[2] = position[2] * 10 + 2
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na západ.
        if paths & 4:
            # Uložíme prvek s pozicí západní křižovatky.
            tem[0] = position[0]
            tem[1] = position[1] - 1
            # Na poslední pozici aktuální cestu * 10 + 4.
            tem[2] = position[2] * 10 + 4
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na jih.
        if paths & 8:
            # Uložíme prvek s pozicí jižní křižovatky.
            tem[0] = position[0] - 1
            tem[1] = position[1]
            # Na poslední pozici aktuální cestu * 10 + 8.
            tem[2] = position[2] * 10 + 8
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
        # Prošlý prvak uložíme do fronty finished.
        finished.append(position)
```

```
    # Pokud nenalezne žádné neprohledané křižovatky, uložíme True do
globální promněnné _stop.
    _stop = True


def Astar(finish):
    global _maze, _position_absolute

    finished = []
    check = []
    tem = [0,0,0]
    # Do fronty check uložíme hodnotu pozici na které robot stojí.
    tem[0] = position_absolute[0]
    tem[1] = position_absolute[1]
    check.append(tem.copy())

    # Provádíme dokud se nacházejí prvky v promněnné check.
    while check:
        # Vyjmeme prvek který je nejblíže k cíli z fronty check a
nahrajeme ho do promněnné position.
        position = check.pop(Min_Distance_From(check, finish))
        #print(position)
        paths = _maze[position[0]][position[1]]
        # Pokud pozice cíle je moje aktuální pozice, vrátíme promněnnou
position.
        if finish[0] == position[0] and finish[1] == position[1]:
            return position

        # Pokud na křižovatce nachází cesta na východ.
        if paths & 1:
            # Uložíme prvek s pozicí východní křižovatky.
            tem[0] = position[0]
            tem[1] = position[1] + 1
            # Na poslední pozici aktuální cestu * 10 + 1.
            tem[2] = position[2] * 10 + 1
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na sever.
        if paths & 2:
            # Uložíme prvek s pozicí severní křižovatky.
            tem[0] = position[0] + 1
            tem[1] = position[1]
            # Na poslední pozici aktuální cestu * 10 + 2.
            tem[2] = position[2] * 10 + 2
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
```

```
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na západ.
        if paths & 4:
            # Uložíme prvek s pozicí západní křižovatky.
            tem[0] = position[0]
            tem[1] = position[1] - 1
            # Na poslední pozici aktuální cestu * 10 + 4.
            tem[2] = position[2] * 10 + 4
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
    # Pokud na křižovatce nachází cesta na jih.
        if paths & 8:
            # Uložíme prvek s pozicí jižní křižovatky.
            tem[0] = position[0] - 1
            tem[1] = position[1]
            # Na poslední pozici aktuální cestu * 10 + 8.
            tem[2] = position[2] * 10 + 8
            # Pokud se připravený prvek nenachází ve frontě check nebo
finished, uložíme ho na konec fronty check.
            if In_Front(tem, check) and In_Front(tem, finished):
                check.append(tem.copy())
        # Prošlý prvak uložíme do fronty finished.
        finished.append(position)

#Robot_Turn('left')




# Čekej na stlačení tlačítka
while force_sensor.force(_force)==0:
    pass

while not(_stop):
    Select_Path()

# Čekej na stlačení tlačítka
while force_sensor.force(_force)==0:
    pass

_maze[0][0]=2
position_absolute = [0,0,0]
best_path_result = BFS()
position_absolute[0] = best_path_result[0]
position_absolute[1] = best_path_result[1]
Ride_Path(best_path_result[2])
_stop = False
```

```
while not(_stop):
    Select_Path()

# Čekej na stlačení tlačítka
while force_sensor.force(_force) == 0:
    pass

position_absolute = [0,0,0]
best_path_result = Astar(_position_finish)

position_absolute[0] = best_path_result[0]
position_absolute[1] = best_path_result[1]
Ride_Path(best_path_result[2])
```

ANOTACE

Jméno a příjmení:	Radek Matoušek
Katedra:	Katedra technické a informační výchovy
Vedoucí práce:	Mgr. Radim Děrda
Rok obhajoby:	2023

Název práce:	Realizace základních algoritmů v programovacím jazyce Python s využitím stavebnice LEGO SPIKE Prime
Název v angličtině:	Implementation of basic algorithms in the Python programming language using the LEGO SPIKE Prime
Anotace práce:	V práci bude popsán postup sestavení robota a jeho naprogramování od nejjednodušších robotických úloh až k algoritmům používaným na robotických soutěžích.
Klíčová slova:	algoritmizace, algoritmus, lego, programovací jazyk, programování, python, robot, robotické stavebnice, spike prime
Anotace v angličtině	The paper will describe the process of building a robot and its programming from the simplest robotic tasks to algorithms used in robotic competitions.
Klíčová slova v angličtině:	algorithmization, algorithm, lego, programming language, programming, python, robot, robot kits, spike prime
Přílohy vázané v práci:	Příloha č. 1: Program pro disciplínu „Čárové bludiště“
Rozsah práce:	64 stran, 13 stran příloh
Jazyk práce:	Český jazyk