UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

Robotické zařízení pro testování algoritmů umělé inteligence

Bc. Bohumil Lotz

Diplomová práce

2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Bohumil Lotz**
Osobní číslo:	**I22187**
Studijní program:	**N0714A150005 Automatické řízení**
Téma práce:	**Robotické zařízení pro testování algoritmů umělé inteligence**
Zadávající katedra:	**Katedra řízení procesů**

Zásady pro vypracování

Cílem diplomové práce je návrh a realizace konstrukce robotického mechatronického zařízení pro testování algoritmů umělé inteligence, s využitím strojového vidění. Zařízení bude umožňovat autonomní funkce pro manipulaci např. řešení problému skládání "Rubikovy kostky". Konstrukce bude využívat kamerový sytém pro snímání aktuálního stavu cílového objektu a nástroje umělé inteligence pro určení řešení zadaného problému.
Hlavní cíle práce jsou:

- Rešerše zadaného tématu.
- Návrh konstrukce mechatronického zařízení, který bude plnit požadované funkce.
- Modulární implementace softwaru pro kamerový systém se zaměřením na snímání stavu cílového objektu s následnou digitalizací.
- Tvorba algoritmu, nebo modelu, umělé inteligence, pro analýzu dat z kamerového systému a návrhu řešení manipulace s cílovým objektem.

Řídicí systém bude realizován vybraným typem mikropočítačového vývojového kitu. Nedílnou součástí práce bude podrobná výrobní dokumentace a zdrojové kódy řídicího mikropočítače a obslužného software.

Rozsah pracovní zprávy: **60**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

MAŘÍK, Vladimír; ŠTĚPÁNKOVÁ, Olga a LAŽANSKÝ, Jiří. Umělá inteligence. Praha: Academia, 1993-. ISBN 978-80-200-1470-2.
BÍLA, Jiří. Umělá inteligence a neuronové sítě v aplikacích. 2. vyd., přeprac. Praha: Vydavatelství ČVUT, 1998. ISBN 80-01-01769-9.

Vedoucí diplomové práce: **Ing. Libor Havlíček, Ph.D.**
Katedra řízení procesů

Datum zadání diplomové práce: **8. listopadu 2023**
Termín odevzdání diplomové práce: **17. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Daniel Honc, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 14. listopadu 2023

Prohlášení:

Prohlašuji:

Práci s názvem Robotické zařízení pro testování algoritmů umělé inteligence jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 31. 01. 2024 Bc. Bohumil Lotz

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat Ing. Liboru Havlíčkovi Ph.D za odborné vedení mé diplomové práce, cenné rady a věcné připomínky, které mi pomohly práci dokončit.

ANOTACE

Tato diplomová práce je věnována návrhu Robotické zařízení pro testování algoritmů umělé inteligence s využitím strojového vidění. Zařízení bude umožňovat funkce pro manipulaci např. řešení problému skládání „Rubikovy kostky.“ Konstrukce bude využívat kamerový systém pro snímání aktuálního stavu cílového objektu a nástroje umělé inteligence pro určení řešení zadaného problému.

KLÍČOVÁ SLOVA

Umělá inteligence, Strojové učení, Strojové vidění, Robot, Rubikova kostka

TITLE

ROBOTIC DEVICE FOR TESTING ARTIFICIAL INTELIGENCE ALGORITHMS

ANNOTATION

This diploma thesis is devoted to the design of a robotic device for testing artificial intelligence algorithms using machine vision. The device will enable manipulation functions, e.g. solving the problem of solving the "Rubik's cube." The construction will use a camera system to capture the current state of the target object and artificial intelligence tools to determine the solution to the given problem.

KEYWORDS

Artificial intelligence, Machine Learning, Machine vision, Robot, Rubik´s cube

OBSAH

SEZNAM ZKRATEK A ZNAČEK

AI, UI – Artificial inteligence, Umělá inteligence

ANN – Artificial neural network

HSV – Hue, Saturation, Value – barevný model

CW – ClockWise – Po směru hodinových ručiček

CCW – Counter ClockWise – Proti směru hodinových ručiček

I/O – Input / Output – Vstup / Výstup

SEZNAM ILUSTRACÍ A TABULEK

ÚVOD

V dnešní éře digitální transformace a rapidního pokroku v oblasti informatiky a technologií se algoritmy umělé inteligence stávají klíčovým prvkem naší společnosti. Tato diplomová práce se zaměřuje na zkoumání a analýzu algoritmů umělé inteligence, které hrají nezastupitelnou roli ve zpracování dat, automatizaci, predikci a řešení komplexních úkolů.

Algoritmy umělé inteligence představují technologický pokrok, který umožňuje vytvářet systémy schopné učení, adaptace a inteligentního rozhodování. Tyto algoritmy se stávají stěžejním nástrojem v oblastech jako jsou strojové učení, neuronové sítě, genetické algoritmy, evoluční výpočty a mnoho dalších. Jsou klíčovými složkami systémů, které dokáží analyzovat obrovská množství dat, identifikovat vzory a odvozovat relevantní informace.

Cílem této práce je otestovat a porovnat různé algoritmy umělé inteligence, jejich výhody a omezení, a jejich konkrétní použití. Dále se budeme věnovat technikám optimalizace a ladění těchto algoritmů s cílem dosáhnout co nejlepších výsledků.

1. RUBIKOVA KOSTKA

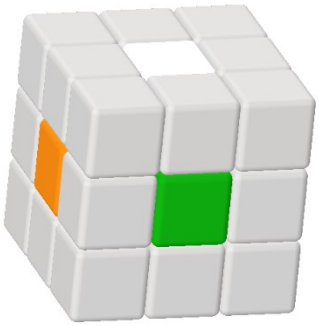
1.1. Úvod

Rubikova kostka je mechanický hlavolam, který se skládá ze 26 menších krychlí, které jsou za pomocí jádra složeny do větší krychle. Každá strana velké krychle má jednu ze šesti různých barev, typicky červená, zelená, modrá, oranžová, bílá a žlutá. Středy každé strany jsou připojeny k jádru kostky, díky kterému můžeme otáčet každou stranu a tím měnit pozice menších krychlí mezi sebou. Rubikova kostka má svou strukturou, až 43 252 003 489 856 000 kombinací. Díky tomu je matematicky velmi náročné získat její řešení.

Obrázek 1.1 - Rubikova kostka

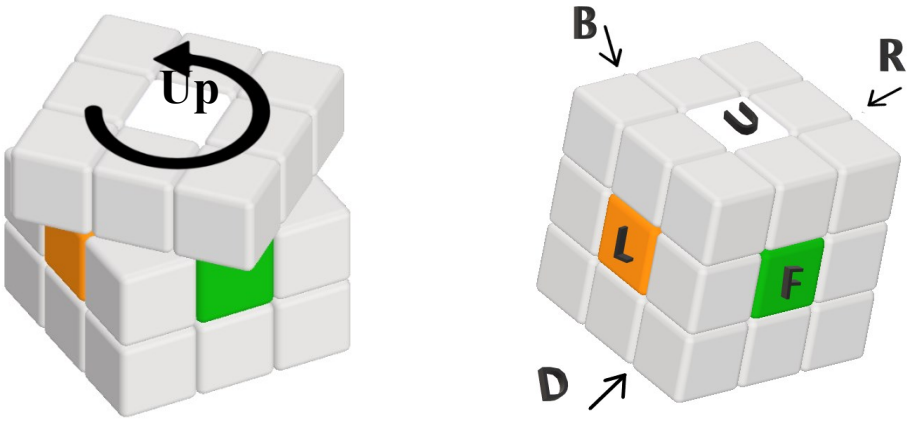
1.2. Terminologie

Každá strana kostky má na poli skládání kostky své značení, avšak to záleží na tom, jak se na kostku řešitel dívá, avšak obecné základní značení, podle nějž se i kostka rozkládá je značeno tak, že zelený střed je na přední straně a na vrchní straně kostky je střed bílý.



Obrázek 1.2 - Rubikova kostka středy

Značení je vždy značeno podle toho, jak se se stranou otáčí. Můžeme nalézt vícero značení, avšak platí vždy totéž a to, zda točíme po směru hodinových ručiček anebo proti. Každá strana má svou značku a též jestli se otáčíme po nebo proti směru ručiček. Značení stran vychází z anglického pojmenování, R – Right, L – Left, U – Up, D – Down, F – Front, B – Back. Značení, zda otáčíme po směru (Clockwise – CW) nebo proti (Counter ClockWise – CCW)



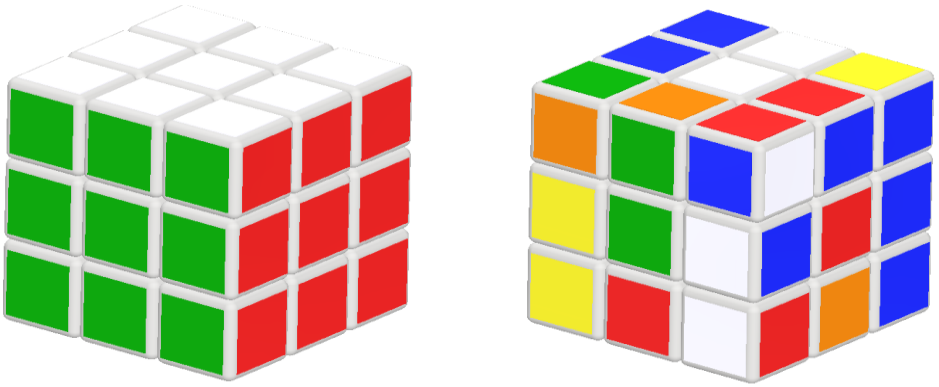
Obrázek 1.3 - Rubikova kostka značení

vychází z toho, jak se na kostku díváme. Např. Pravá strana proti směru (Rp, R', Rccw) bude pravá strana dolů – pokud se díváme ze předu. Naopak Levá protisměru (Lp, L', Lccw) bude levá nahoru.

Pokud tedy budeme mít nějaký scramble (tj. jak kostku rozložíme), jako například tento

D F2 D R2 D2 F2 L2 B2 U' R2 D R2 L' D' B R' D2 F2 U B'

,tak z počátečního stavu dostaneme:



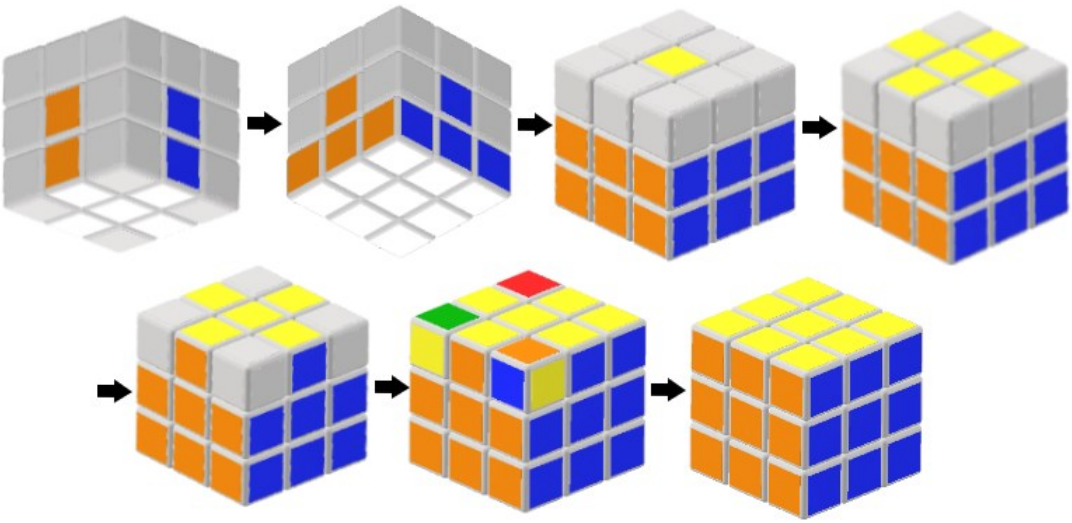
Obrázek 1.4 - Rubikova kostka scrambled

1.3. Metody složení:

Teď když známe značení kostky, tak se můžeme začít bavit o jejím skládání. Existuje několik metod pro složení. Některé se soustředí na rychlost složení, některé na počet tahů a některé na to, aby se kostka dala složit i po slepu. Každá je tedy něčím výjimečná.

1.3.1. Metoda vrstva po vrstvě

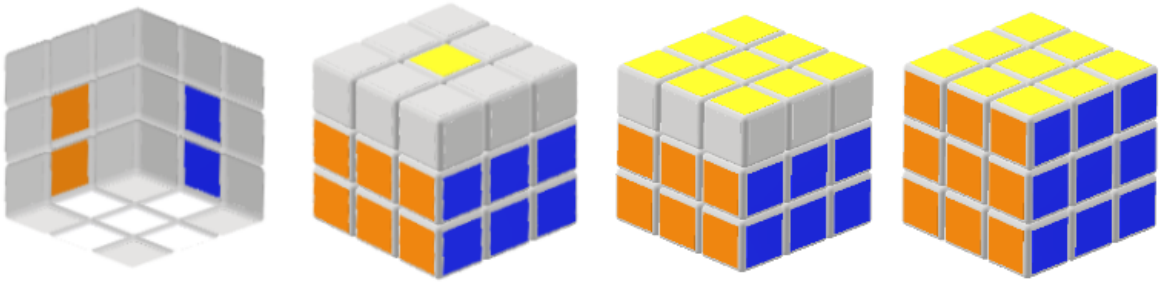
Tato metoda je jedna z nejjednodušších a většina lidí s ní a jejími variantami začíná. V metodě se lze naučit základní principy, které jsou zásadní při používání složitějších, rychlejších metod.



Obrázek 1.5 - Metoda vrstva po vrstvě

1.3.2. Metoda CFOP (Fridrich metod)

Název metody je odvozen od kroků, jimiž při skládání procházíte a také podle české profesorky Jessica Fridrich působící na Binghamptonské Univerzitě v New Yorku, která tuto

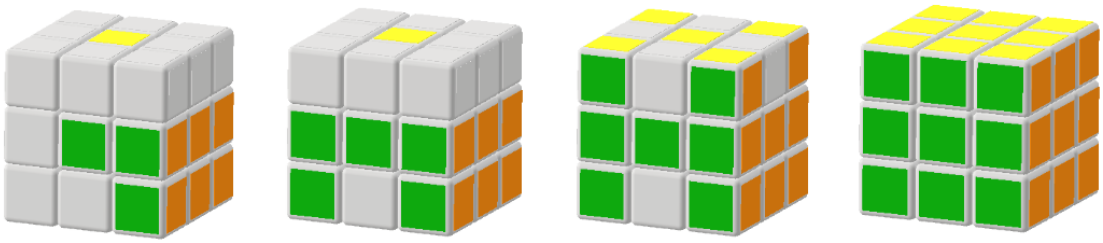


Obrázek 1.6 - Metoda CFOP

metodu publikovala v roce 1997. Metoda je značně rychlejší než metoda vrstva po vrstvě. Nejenže přeskakuje kroky této metody, ale ke po složení dvou vrstev nejdříve naorientuje poslední vrstvu a poté kostku složí. Z tohoto postupu také vychází název metody CFOP – Cross, First two layers, Orientation of the last layers, Permutation of the last layer. Metoda se pohybuje průměrně 55-60 tahů na složení. Nevýhodou metody je nepřeberné množství algoritmů, které je potřeba znát pro „perfektní“ složení.

1.3.3. Metoda Roux

Metoda Roux je dělaná přímo pro kostku 3x3x3 na rozdíl od CFOP, které je možno aplikovat i na další typy hlavolamů. Výhodou této metody je, že se v průměru pohybuje kolem 45-50 tahů na složení, tedy je metoda rychlejší než CFOP. Další výhodou je, že není potřeba kostkou rotovat, jako v dalších metodách. Principem metody, je složit blok 1x2x3, poté druhý na druhé straně, poté složit rohy v poslední vrstvě a poté LSE (Last six edges) a kostka bude složená.

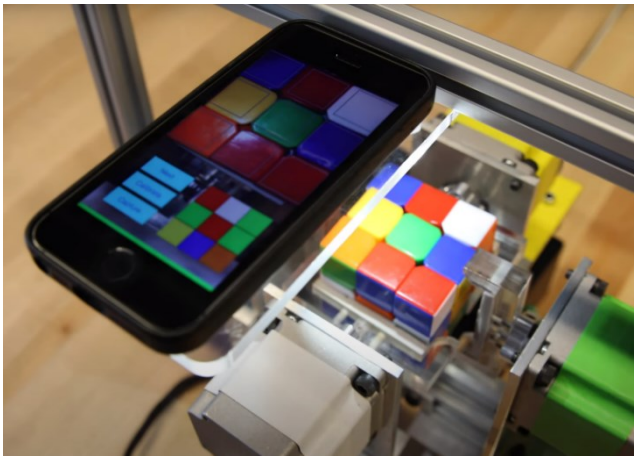


Obrázek 1.7 - Metoda Roux

2. Rubik solvery

2.1. RUKU

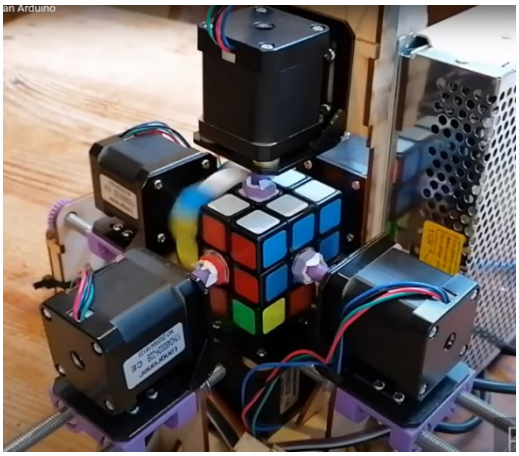
Robot vytvořen na vývojovém kitu Raspberry Pi. Pro snímání stavu kostky se využívá aplikace v telefonu, která poté odesílá data do Raspberry. Motory jsou opatřeny nástroji pro uchycení kostky, však po každém tahu se musí „přehmátnout“, aby nedošlo ke kolizi. Robot byl navrhnut pro výukové účely.



Obrázek 2.1 - Robot RUKU

2.2. DIY - Freddie Meinertzhagen

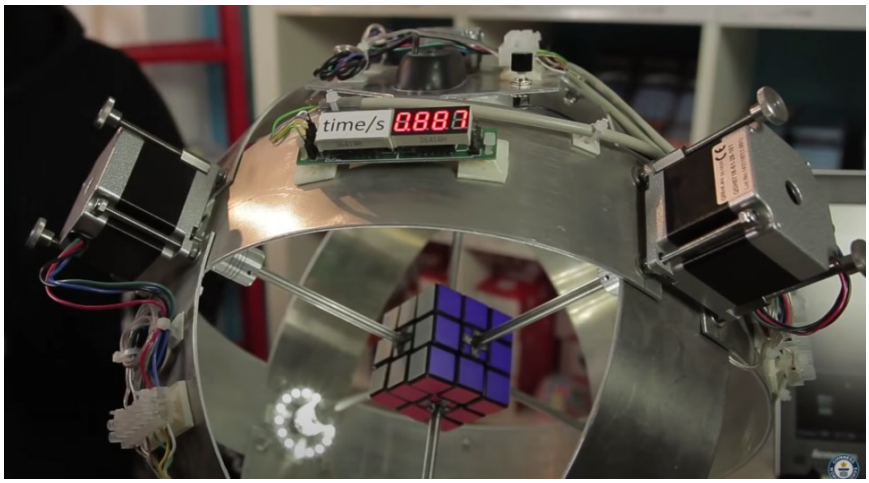
Tento robot je unikátní v tom, že barvy se musejí zadat ručně do konzole Arduina, protože robot nemá kamerový systém, přes který by kostku skenoval. Dále tento robot používá „konstantní“ postup, tj. kostku vždy skládá stejně – stejné kostky i stejné barvy – vždy ve stejném pořadí.



Obrázek 2.2 - Solver od Freddie Meinertzhagen

2.3. SUB1

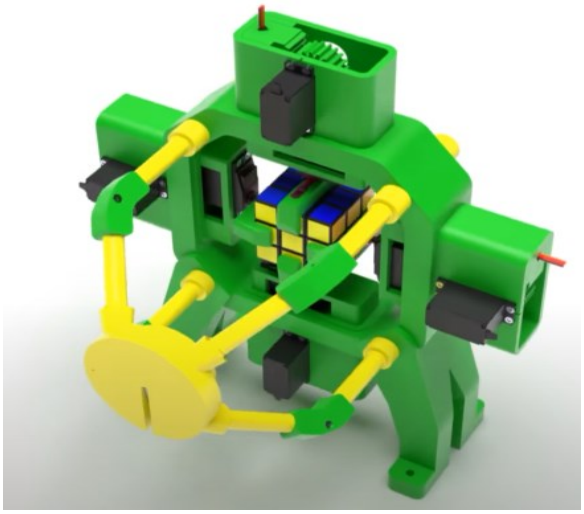
HW část je postavena na vývojovém kitu Arduino, však výpočetní výkon, spolu s kamerovým systémem je zde zpracováván v PC, který poté jen předává příkazy přes sériovou komunikaci do Arduina. Robot je postaven primárně pro rychlostní skládání, proto je zde využíván Herbert Kociemba's Two-Phase algoritmus, který je považován za jeden z nejrychlejších, co se týče kombinačního skládání u Rubikovy kostky.



Obrázek 2.3 - Solver SUB1

2.4. O.T. Vinta

Tento solver je opět postaven na Raspberry PI. Využívá USB kamery pro skenování stavu kostky a poté servomotory umožňující pohyb. Je zde stejný mechanizmus jako u robota RUKU, a to je přehmatávání při otočení kostkou, aby nedošlo ke kolizi. Zajímavostí na tomto robotu je, že konstrukce byla kompletně vytisknuta na 3D tiskárně.



Obrázek 2.4 - Solver O.T. Vinta

3. Umělá inteligence

3.1. Úvod

Umělá inteligence (AI) je bezesporu jedním z nejrevolučnějších oborů v oblasti informatiky a technologie. Tento oddíl se věnuje podrobnému zkoumání a analýze konceptů, technik a aplikací, které tvoří jádro umělé inteligence. Umělá inteligence představuje disciplínu, která se snaží vytvářet systémy a algoritmy, které dokáží simulovat nebo napodobit lidskou inteligenci a schopnosti, jako je učení, rozpoznávání vzorů, rozhodování a řešení problémů. Tento oddíl se zaměří na různé aspekty umělé inteligence, včetně strojového učení, hlubokého učení a dalších.

3.2. Neinformované algoritmy

První velkou skupinou umělé inteligence jsou neinformované algoritmy, do nichž patří velká část odvětví umělé inteligence. Neinformované algoritmy jsou také známé jako "hrubá síla" nebo "slepé" algoritmy, protože nepoužívají žádné specifické informace o problému k tomu, aby rozhodly, který krok udělat. Jsou obecně založeny na systematickém prozkoumávání možných řešení a jsou vhodné pro situace, kdy neexistuje mnoho informací o struktuře problému.

3.2.1. Prohledávání do šířky BFS

Prohledávání do šířky (Breadth-First Search) je algoritmus používaný v oblasti umělé inteligence a informatiky k prohledávání stavového prostoru. Tento algoritmus se používá k prozkoumání všech uzlů nebo stavů v grafu od kořene stromu, postupně do větší hloubky. Jeho základní myšlenkou je, že se nejdříve prozkoumávají všechny sousední uzly kořene, pak všechny sousední uzly těchto sousedních uzlů, a tak dále, dokud nejsou prozkoumány všechny dosažitelné uzly nebo nalezeno řešení.

3.2.2. Obousměrné prohledávání do šířky BiDi

Obousměrné prohledávání do šířky (BiDi – BiDirectional BFS) je modifikací algoritmu BFS. Na rozdíl od BFS, který prohledává stavový prostor pouze od kořene (počátečního bodu), tak BiDi BFS prohledává stavový prostor současně z obou uzlů, od počátečního bodu a také od cílového bodu. Algoritmus pokračuje ve vyhledávání ve dvou směrech, dokud se tyto dvě fronty

nespojí nebo se cílový uzel neprohledá. Tímto způsobem lze najít nejkratší cestu mezi počátečním a cílovým uzlem výrazně efektivněji.

3.2.3. Prohledávání do hloubky DFS

Prohledávání do hloubky (DFS – Depth First Search) je algoritmus, který začíná v kořenovém uzlu a postupuje do hloubky podél jedné větve grafu, než se vrátí zpět, a pokračuje v prohledávání další větve. DFS je hloubkový algoritmus, což znamená, že prochází co nejhlouběji do grafu, než se vrátí zpět.

3.2.4. Prohledávání do hloubky s omezením DLS

Depth-Limited Search je varianta algoritmu prohledávání do hloubky (Depth-First Search, DFS). DLS je omezený podle maximální hloubky, do které se může prohledávání dostat. To znamená, že DLS neprochází do nekonečna do hloubky, ale pouze do určité úrovně. Pokud cíl není dosažen v této omezené hloubce, algoritmus se vrátí a pokračuje v prohledávání jinými cestami.

3.2.5. Iterativní prohledávání do hloubky IDS

Iterativní prohledávání do hloubky (IDS - Iterative Deepening Search) je algoritmus, který kombinuje vlastnosti prohledávání do šířky (Breadth-First Search, BFS) a prohledávání do hloubky (Depth-First Search, DFS). Tento algoritmus je používán pro nalezení nejkratší cesty nebo řešení v grafu, ale umožňuje postupně zvyšovat hloubku prohledávání s každým pokusem, dokud nenajde řešení.

IDS v kontextu prohledávání grafu je obzvláště užitečný, když nevíte, jaká je optimální hloubka prohledávání, a chcete najít nejbližší řešení. Tím, že postupně zvyšujete hloubku, IDS umožňuje řešit problémy, které by jinak vyžadovaly nekonečný časový nebo paměťový zdroj s použitím tradičního BFS.

3.3. Informované algoritmy

Informované algoritmy, známé také jako heuristické algoritmy, jsou typy algoritmů používaných v oblasti umělé inteligence a hledání řešení problémů. Na rozdíl od neinformovaných algoritmů, informované algoritmy využívají specifické informace o problému, které jim pomáhají efektivněji navigovat v prohledávacím prostoru a najít optimální nebo přibližně optimální řešení.

Základním prvkem informovaných algoritmů je heuristika, což je funkce nebo strategie, která poskytuje odhad, jak blízko jsme ke správnému řešení nebo k optimu. Tato heuristická informace umožňuje algoritmu určovat, které cesty nebo uzly v prohledávacím stromu je vhodné zkoumat nejdříve.

3.3.1. Best-First Search

Best-First Search je algoritmus, který se používá k hledání nejlepšího uzlu pro prozkoumání na základě nějaké heuristiky nebo odhadu. Tento algoritmus se zaměřuje na výběr uzlu, který má největší pravděpodobnost, že je blízko cíli, a tím se snaží najít optimální řešení nebo co nejlepší řešení problému.

Na rozdíl od BFS a DFS, kde se využívala fronta hledání, tak zde se nachází také, ovšem s tím rozdílem, že fronta je rozšířena o prvek priority. Ta je určena heuristickou funkcí. To znamená, že fronta se při každém prozkoumaném uzlu přerovná sestupně podle heuristické funkce, aby dalším prozkoumávaným prvkem byl prvek s nejvyšší prioritou, tj. prvek s nejvyšší pravděpodobností, že je blízko cíle.

3.3.2. A* algoritmus

A* (A-Star) je algoritmus, který kombinuje výhody algoritmu prohledávání do šířky (Breadth-First Search, BFS) a algoritmu prohledávání do hloubky (Depth-First Search, DFS) s heuristikou. Tento algoritmus je používán pro hledání nejkratší cesty nebo optimálního řešení problému v grafu nebo stromu.

Stejně jako Best-First-Search má frontu založenou na prioritě určenou heuristickou funkcí. Ta je zde ovšem určena více parametry než jen vzdálenost od cíle. Můžeme do ní zakomponovat také například cenu, kterou stanovíme na základě toho kudy se řešení vydá.

3.4. Neuronové sítě

Neuronové sítě jsou matematický model inspirovaný strukturou a funkcí lidského mozku, který se používá k řešení úloh zpracování informací, učení a rozhodování. Tyto sítě se skládají z jednotek nazývaných neurony, které jsou propojeny váhami. Neurony přijímají vstupní signály, zpracovávají je a generují výstupní signály. Nejjednodušší neuronovou sítí je síť s jedním neuronem zvaným perceptron.

3.4.1. Perceptron

Perceptron je jednoduchý model neuronové sítě, který byl navržen v 50. letech 20. století Frankem Rosenblattem. Jedná se o základní stavební blok v oblasti neuronových sítí a představuje nejjednodušší formu učícího se algoritmu pro binární klasifikaci.

$$Z = \sum_{i=1}^{N} a_i w_i - \theta \tag{3.1}$$

Kde a je hodnota na vstupu i

w je váha vstupu

θ je bias,

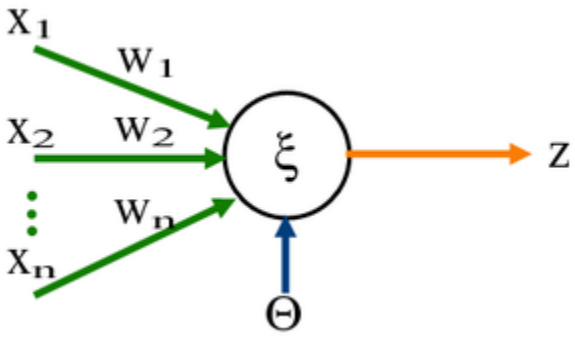
N je počet vstupů

Z je potenciál neuronu – agregační funkce

$$y = akt(Z) \tag{3.2}$$

akt je aktivační funkce

y je výstup perceptronu



Obrázek 3.1 – Perceptron (Haykin, 1999)

Aktivační funkce

Aktivační funkce je základní stavební blok neuronových sítí, zejména v hlubokém učení. Jedná se o matematickou funkci, která přijímá váženou sumu vstupních hodnot a přidává do výstupu neuronu nelinearitu. Existuje několik různých aktivačních funkcí, z nichž každá má své vlastnosti a vhodnost pro různé typy úloh a architektur neuronových sítí.

Mezi nejběžněji používané aktivační funkce patří:

Sigmoidní funkce

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

Kde x je vstupní hodnota

σ je výstupní hodnota v rozsahu 0 až 1

Hyperbolický tangens

$$\sigma(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{3.4}$$

Kde x je vstupní hodnota

σ je výstupní hodnota v rozsahu -1 až 1

ReLu

Je definována v intervalech

$$\sigma(x) = \begin{Bmatrix} 0 & pro\ \mathrm{x} \in (-\infty; 0) \\ x & pro\ \mathrm{x} \in [0; \infty) \end{Bmatrix} \tag{3.5}$$

SoftMax

SoftMax algoritmus normalizuje výstupy aktivačních funkcí vzhledem k celku, proto se často používá jako aktivační funkce u poslední vrstvy v neuronové síti, aby bylo ihned zřejmé, který výstup se percentuálně zdá jako „nejsprávnější“.

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{3.6}$$

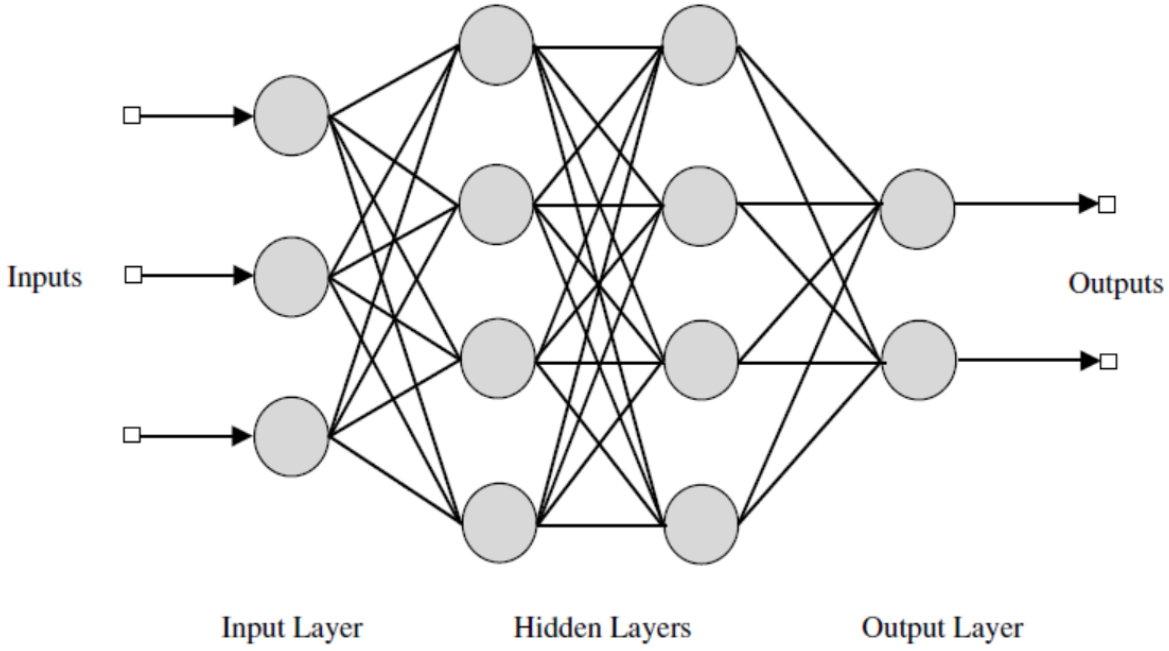
Kde x_i je vstupní hodnota neuronu

x_j je suma vstupních hodnot neuronu

σ je výstupní hodnota v rozsahu 0 až 1

3.4.2. Vícevrstvé neuronové sítě

Vícevrstvé neuronové sítě jsou rozšířením jednoduchého perceptronu a představují základní architekturu pro moderní hluboké učení. Základním rozdílem oproti jednovrstvým perceptronům je přidání jedné nebo více skrytých vrstev neuronů mezi vstupní a výstupní vrstvou.



Obrázek 3.2 - Sktruktura neurální sítě (Haykin, 1999)

Základní rovnice je totožná, ale výstup y je poté vstupem další vrstvě neuronové sítě, kde opět každý vstup má svou váhu a po sumaci opět volíme vhodnou aktivační funkci.

$$Z_1 = A \cdot W - \theta \tag{3.7}$$

$$Y_1 = akt(Z_1) \tag{3.8}$$

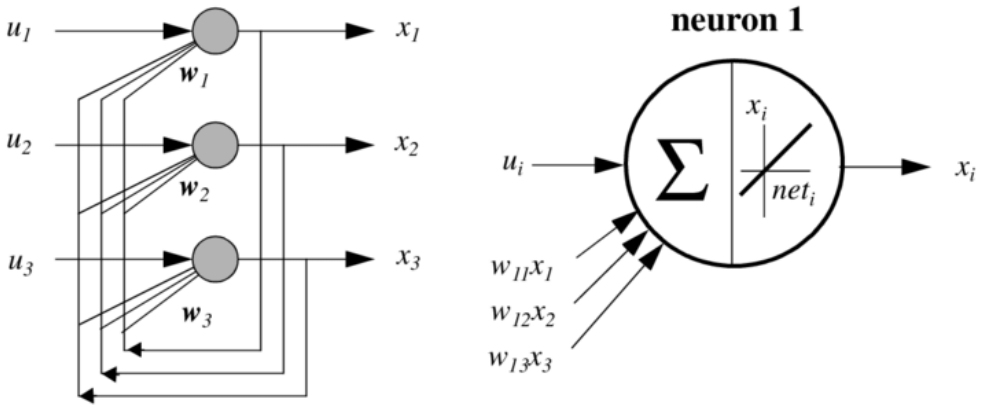
$$Z_2 = Y_1 \cdot V - \vartheta \tag{3.9}$$

$$Y_2 = akt(Z_2) \tag{3.10}$$

*V tomto zápisu se jedná o maticové násobení, proto se zde již nevyskytují sumy.

3.4.3. Hopfieldova neuronová síť

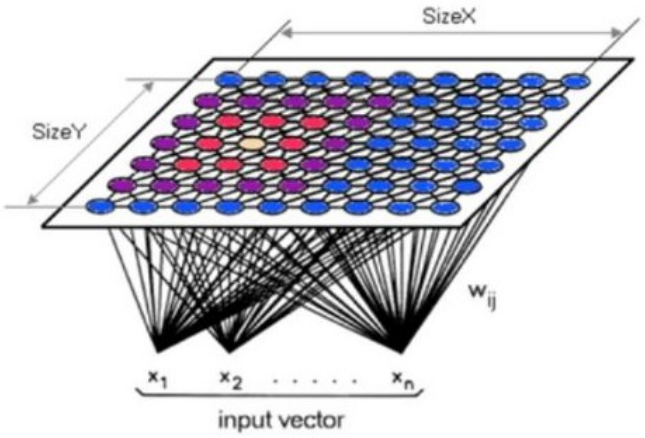
Hopfieldova neuronová síť je typ umělé neuronové sítě navržený Johnem Hopfieldem v roce 1982. Jedná se o rekurentní neuronovou síť, což znamená, že obsahuje zpětné vazby, tj. spojení, která umožňují signálům cirkulovat v síti. Hopfieldova síť má specifickou strukturu a je obvykle používána pro řešení problémů optimalizace, ukládání a obnovování vzorů. (George G. Lendaris, 1999)



Obrázek 3.3 - Struktura Hopfieldovy sítě (George G. Lendaris, 2019)

3.4.4. Kohonenova mapa

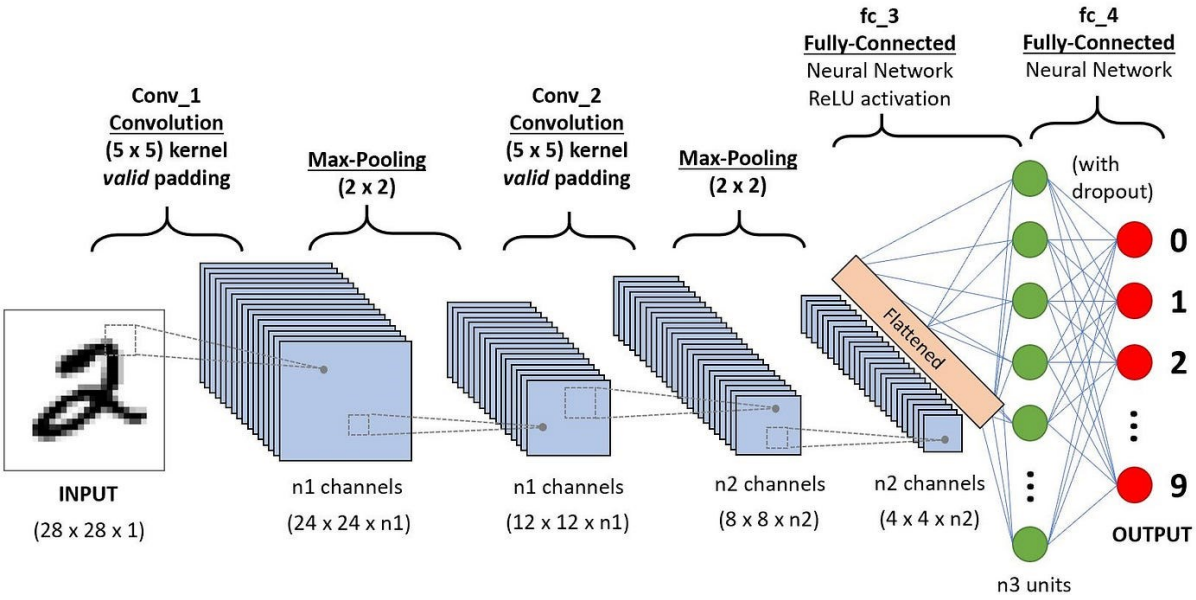
Kohonenova mapa, známá také jako samoorganizační mapa je typ umělé neuronové sítě, který byl vyvinut Finským vědcem Teuvo Kohonenem v roce 1982. Kohonenova mapa patří mezi učení bez učitele a je často používána k shlukování a vizualizaci dat. Kohonenova mapa je specifický typ vícevrstvé neuronové sítě s jednou vrstvou váhových neuronů (též nazývaných Kohonenovy neurony). Tyto neurony jsou uspořádány v 2D nebo 3D mřížce, což umožňuje efektivní vizualizaci dat. (Ciaburro G., Venkateswaran B., 2017)



Obrázek 3.4 - Struktura Kohonenovy mapy

3.4.5. Konvoluční neuronová síť

Konvoluční neuronová síť (Convolutional Neural Network - CNN) je typ neuronové sítě, který byl navržen zejména pro efektivní zpracování vizuálních dat, jako jsou obrázky a videa. CNN se stal klíčovým nástrojem v oblasti počítačového vidění a dosáhl vynikajících výsledků při úlohách jako rozpoznávání objektů, klasifikace obrazu, segmentace obrazu a další.



Obrázek 3.5 - Struktura konvoluční neuronové sítě (Saha Sumit 2018)

Namísto propojení všech jednotek ve vrstvě se všemi jednotkami v předchozí vrstvě konvoluční sítě organizují každou vrstvu do map prvků, které si můžete představit jako paralelní roviny nebo kanály. V konvoluční vrstvě se vážené součty provádějí pouze v rámci malého lokálního okna a váhy jsou identické pro všechny pixely. (Szeliski R, 2010)

Konvoluční sítě jsou schopny zachytit hierarchii příznaků a vzorů v datech a jsou invariantní vůči translaci, což znamená, že jsou schopny rozpoznávat vzory, i když jsou umístěny na různých místech ve vstupních datech. Tato vlastnost je klíčová pro efektivní zpracování vizuálních informací.

3.5. Strojové vidění

Strojové vidění je obor umělé inteligence, který se zabývá vývojem systémů, schopných interpretovat a porozumět vizuálním informacím ze světa okolo nich. Cílem strojového vidění je umožnit počítačům vidět a chápat své okolí. Tato technologie se opírá o různé metody a algoritmy z oblasti počítačového vidění, obrazového zpracování a hlubokého učení.

Klíčové techniky a úkoly ve strojovém vidění zahrnují:

- Klasifikace obrázků: Rozpoznání objektů na obrázku.
- Detekce objektů: Určení umístění a ohraničení objektů v obraze.
- Segmentace obrazu: Rozdělení obrázku do jednotlivých segmentů nebo regionů s podobnými vlastnostmi.
- Rozpoznávání obličejů: Identifikace a analýza obličejů na obrazech a v obrazech.
- Sledování objektů: Sledování pohybu a chování objektů v čase na základě sérií snímků nebo videa.

4. Praktická část

Tabulka 1 - Seznam použitých součástí

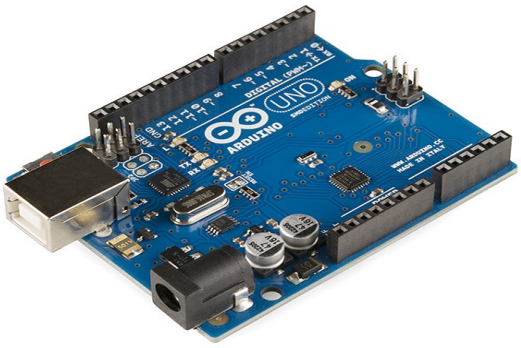
Typ zařízení	Název zařízení	Počet
Vývojový kit	Arduino UNO	1 ks
Řadič krokového motoru	A4988	8 ks
Krokový motor	NEMA-17	8 ks
Kondenzátor	Elektrolytický 10uF	8 ks
Zdroj napájení	LONGWEI LW-K3010D Laboratorní zdroj	

4.1. Hardwarový návrh

4.1.1. Vývojový kit

Arduino UNO je open-source platforma pro vývoj elektronických projektů. Spojuje mikrokontrolér ATmega328P s uživatelsky přívětivým vývojovým prostředím. Díky svým digitálním a analogovým vstupům a výstupům umožňuje snadnou interakci s různými senzory, aktuátory a dalším periferiím.

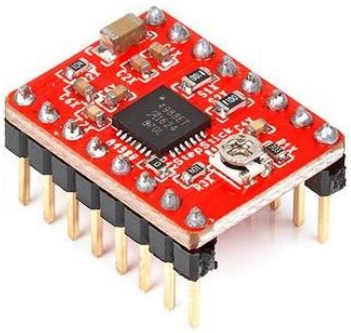
V tomto případě využijeme I/O piny k připojení řadičů pro řízení motorů, kterými budou ovládány jednotlivé motory. Jedná se o piny určující rotaci a směr rotace motorů. Dále bude potřeba sériová komunikace, která pro zjednodušení bude zajištěna přes napájecí USB kabel. Nebudeme však využívat Arduino IDE seriový monitor. Příkazy budeme totiž posílat již z python GUI, tak musí být port (COMx) nepoužívaný.



Obrázek 4.1 - Arduino UNO (LaskaKit, nedatováno)

4.1.2. Motor Driver A4988

Pro řízení motoru bude využít řadič pro krokové motory A4988. Piny 7 (STEP) a 8 (DIR) budou připojeny k Digitálním pinům na Arduinu. Tyto piny slouží k určení směru rotace motoru a k odesílání impulzů pro krokování do motoru. Pro napájení logiky řadiče připojíme pin 10 (VCC) a 9 (GND) na 5V. Piny 2, 3 a 4 pro mikro-krokování budou též napájeny 5V, neboť při plných krocích motoru v plné, což odpovídá 1,8° vznikají obrovské vibrace, které mohou v plné sestavě znemožnit správnou funkčnost zařízení. Pro napájení řadiče i motoru připojíme piny 16 (VMOT) a 15 (GND) k druhému zdroji napětí o hodnotě 12V.



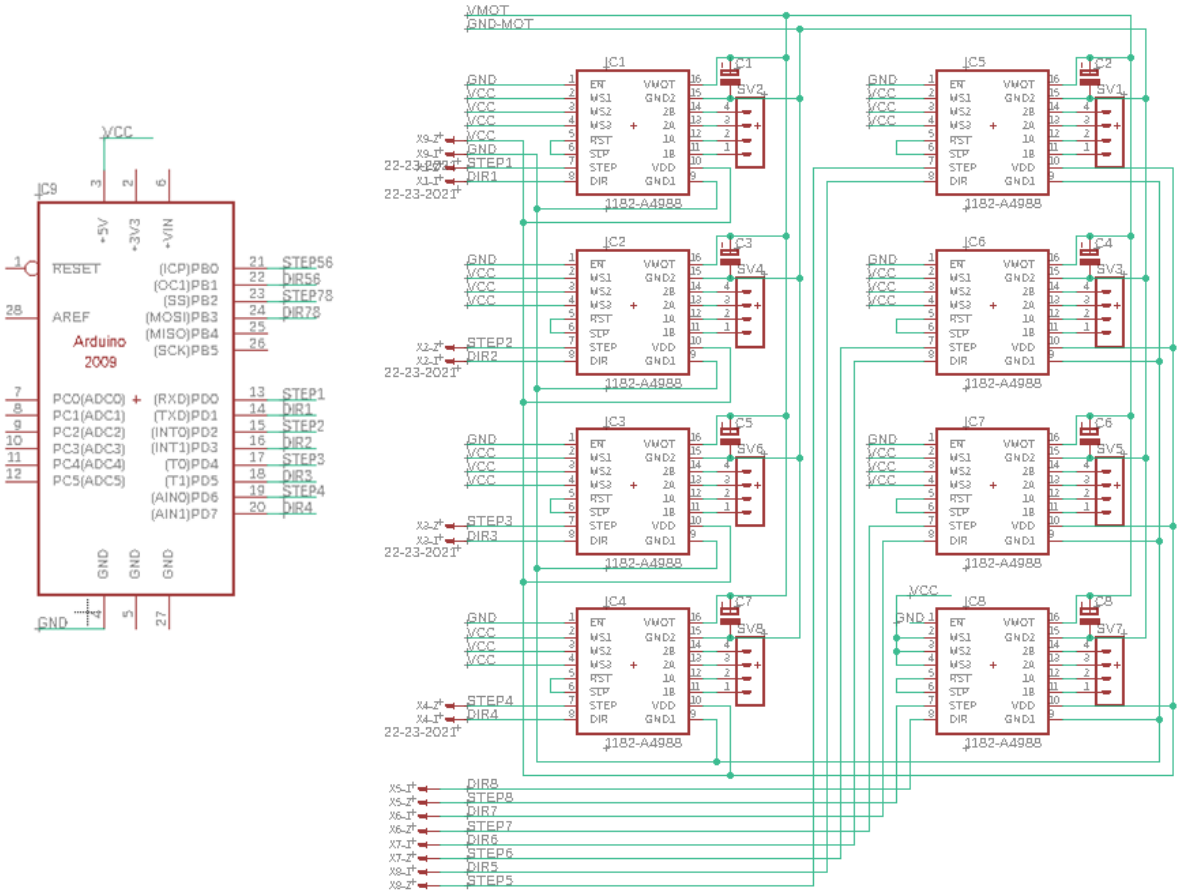
Obrázek 4.2 - Řadič A4988 (Dratek, nedatováno)

4.1.3. Krokový motor NEMA-17

Tento krokový motor svou cenou/výkonem odpovídá našim potřebám. Má statický moment cca 400mN.m, což je dostatečné pro naše potřeby. Motor připojíme k řadiči na piny 11,12,13,14 vždy po dvojících – na piny 11 a 12 půjdou konce cívky A a na piny 13 a 14 konce cívky B. Kvůli problémům s vibracemi bylo nutné u řadiče zapojit mikro-krokování, což vysoce zpomalilo otáčení motoru. Mikro-krokování kvůli co nejhladšímu průběhu bylo nastaveno na 1/16 původní rychlosti.

Obrázek 4.3 - Motor NEMA-17 (LaskaKit, nedatováno)

4.1.4. Schéma zapojení



Obrázek 4.4 - Schéma zapojení

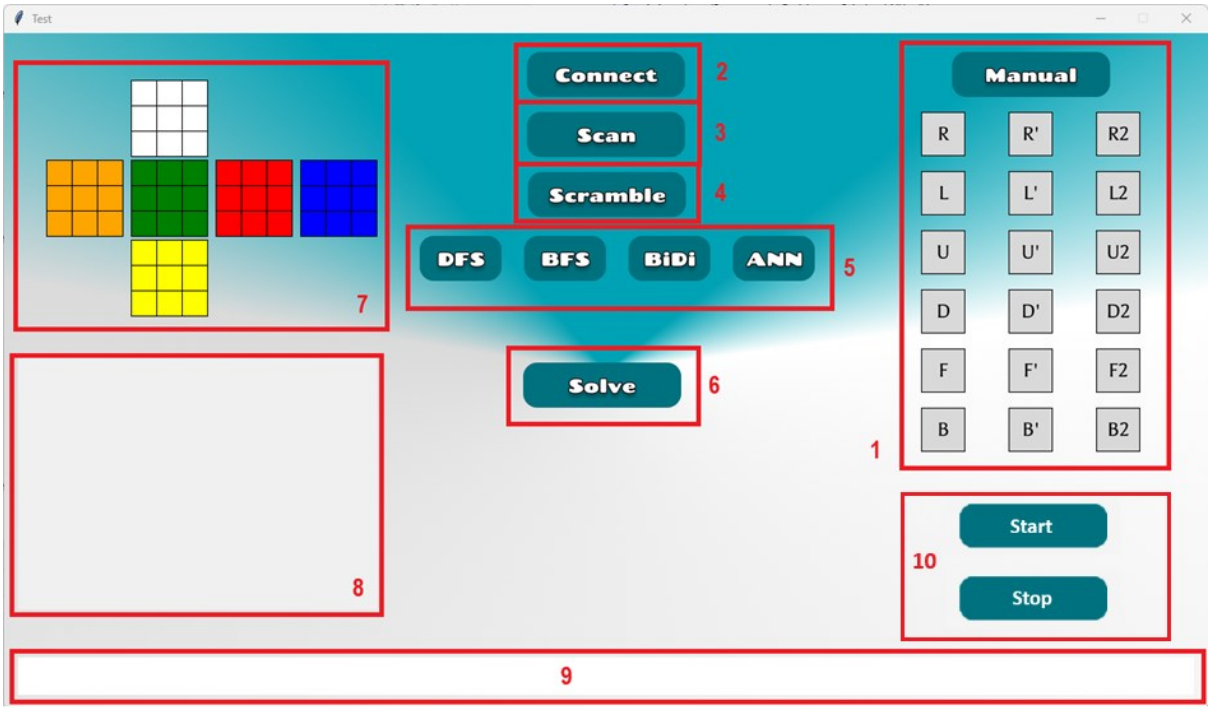
V celé konstrukci je 8 motorů, to znamená, že budeme potřebovat 16 pinů, avšak při otáčení celé kostky je potřeba, aby dva motory „ustoupily“, aby bylo možné kostkou otočit a tento „ústup“ je jednotný pro protilehlé motory, takže bude stačit pro protilehlé motory řídící piny sloučit a ušetříme tím 2 piny pro řízení. Směrové piny sloučit nemůžeme, protože se protilehlé motory netočí stejným směrem.

4.2. Softwarový návrh

Do softwarové části zařadíme implementaci algoritmů, grafické rozhraní umožňující ovládání robota (GUI) a kamerový systém spolu se zpracováním obrazu, který následně předá data některému algoritmu.

4.2.1. GUI

Grafické uživatelské rozhraní (GUI) umožňující uživateli ovládání robota. Níže jsou popsány úkoly jednotlivých komponent.



Obrázek 4.5 - Grafické uživatelské rozhraní

- Část pro manuální ovládání motorů
- Tlačítko pro připojení kamery
- Tlačítko pro naskenování stavu kostky (jestli je)
- Tlačítko pro „zamíchání“ kostky
- Seznam dostupných metod
- Tlačítko pro složení kostky – poté co najde řešení
- Aktuální stav kostky
- Zobrazení kamery
- Konzole – zobrazení řešení / chyb
- Start / Stop – Pro uchopení, puštění kostky

Grafické rozhraní bylo naprogramováno přes knihovnu tkinter. Návrh uživatelského rozhraní bylo vytvořeno ve webové aplikaci www.figma.com. Výsledný návrh byl poté přes python aplikaci Tkinter-Designer, který je volně dostupný na GitHubu na této adrese:

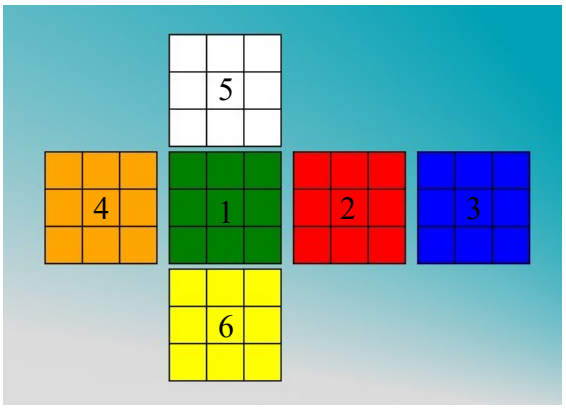
https://github.com/ParthJadhav/Tkinter-Designer, transformován na aplikaci s klasickou tkinter strukturou, do níž se aplikuje funkčnost ostatních skriptů.

```
window = Tk()

window.geometry("1280x720")
window.configure(bg="#FFFFFF")
window.title("Diplomova prace")

canvas = Canvas(
    window,
    bg="#FFFFFF",
    height=720,
    width=1280,
    bd=0,
    highlightthickness=0,
    relief="ridge"
)

canvas.place(x=0, y=0)
image_image_1 = PhotoImage(
    file=relative_to_assets("image_1.png"))
image_1 = canvas.create_image(
    640.0,
    360.0,
    image=image_image_1
)
...
```

Obrázek 4.6 - Struktura programu v Tkinteru

4.2.2. Zobrazení stavu kostky

Pro zobrazení aktuálního stavu kostky bylo potřeba využít lineární algebry, jelikož se výsledný stav nachází v maticové podobě viz Obrázek 4.7. Pro skenování se použije řešení




```
initState = [[["G", "G", "G"], 1
             ["G", "G", "G"],
             ["G", "G", "G"]],
            [["R", "R", "R"], 2
             ["R", "R", "R"],
             ["R", "R", "R"]],
            [["B", "B", "B"], 3
             ["B", "B", "B"],
             ["B", "B", "B"]],
            [["O", "O", "O"], 4
             ["O", "O", "O"],
             ["O", "O", "O"]],
            [["W", "W", "W"], 5
             ["W", "W", "W"],
             ["W", "W", "W"]],
            [["Y", "Y", "Y"], 6
             ["Y", "Y", "Y"],
```

Obrázek 4.7 - Maticové zobrazení kostky

obsažené v kapitole Strojové vidění, avšak to nám zajistí jen aktuální stav kostky. Pro použití v algoritmech je zapotřebí, aby kostka mohla provádět tahy.

Pro umožnění tahů kostky je nutno představit si, co se děje, když se stěna kostky otáčí. Např. když se otáčí přední stěna (1 - Zelená) po směru hodinových ručiček (F), tak není jen zapotřebí otočit barvy na přední stěně, ale je také zapotřebí vzít hodnoty z posledního řádku vrchní stěny (5 - bílá) a transponovat ji na levý první sloupec druhé stěny (2 - červené). Hodnoty červené zase transponovat na první řádek spodní strany (6 - žluté) a hodnoty žluté transponovat na poslední sloupec čtvrté stěny (4 – oranžová) a tu opět reverzně transponovat (transponovat a poskládat od konce na začátek). Zde v programu je tato metoda interpretována, ale pro zjednodušení převodu sloupce na řádky a naopak, zde transponujeme celou stěnu.

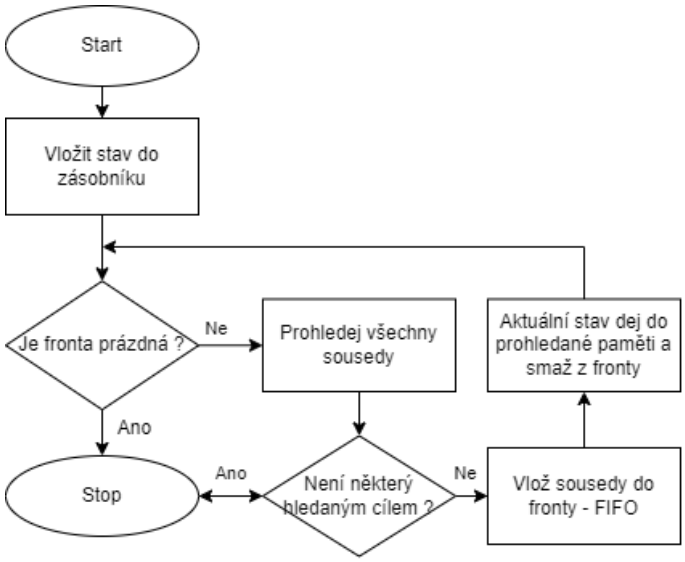
```
def F_CW(self):
    # F row clockwise
    temp = deepcopy(self.cube[1, :, 0])

    self.cube[1] = self.cube[1].T
    self.cube[3] = self.cube[3].T

    self.cube[1, 0] = self.cube[4, 2]
    self.cube[4, 2] = np.flip(self.cube[3, 2], 0)
    self.cube[3, 2] = self.cube[5, 0]
    self.cube[5, 0] = np.flip(temp, 0)

    del temp

    self.cube[1] = self.cube[1].T
    self.cube[3] = self.cube[3].T

    # F side clockwise
    temp = deepcopy(self.cube[0])
    self.cube[0] = np.rot90(temp, 3)
```

Obrázek 4.8 - Ukázka kódu pro změnu stavu kostky

4.2.3. Neinformované algoritmy – programová implementace

Jako příklad neinformovaného algoritmu jsme zde implementovali BFS algoritmus, tedy prohledávání do šířky a BiDi algoritmus, tedy obousměrné prohledávání do šířky, abychom porovnali jejich vlastnosti. Implementace algoritmů je programově napsána níže v jazyce Python.

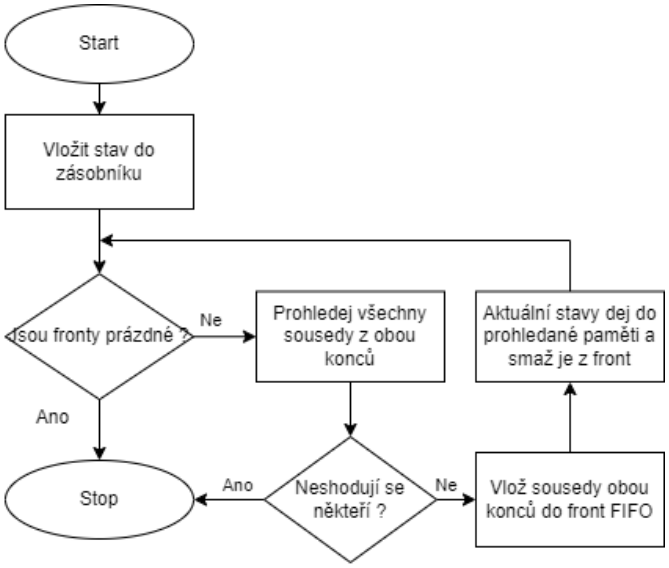
Pro BFS algoritmus můžeme z programu vidět, že je zavedena vždy jedna proměnná pro paměť (self.memory) a jedna proměnná pro prohledávanou frontu (self.fifo), na rozdíl od BiDi algoritmu, kde jsou zavedeny paměti pro oba směry (self.memory ,self.bidi_memory) a (self.fifo, self.bidi_fifo), což je základním rozdílem v algoritmech. Avšak v BiDi algoritmu je též nutné kontrolovat, zda oba seznamy již neobsahují stejný prvek, protože pokud ano, tak již je možné nalézt řešení.



Obrázek 4.9 - Blokové schéma BFS algoritmu

```
def FindSolution(self):
    i = 1

    while len(self.fifo) > 0:
        self.memory.append(self.fifo[0])

        if self.CompareMatrix(self.fifo[0].cube, self.finalPos):
            break

        for actNum in range(0, self.numAct):
            temp = Node(i, self.fifo[0].depth + 1, actNum, self.fifo[0].id, self.Action(actNum, self.fifo[0].cube))

            if ~self.Check(temp, self.memory):
                if ~self.Check(temp, self.fifo):
                    self.fifo.append(temp)
                    i = i + 1
            del temp

        del self.fifo[0]

    if self.fifo[0]:
        return self.Solution()
    return "Řešení nenalezeno!"
```

Obrázek 4.10 – Programová implementace algoritmu BFS

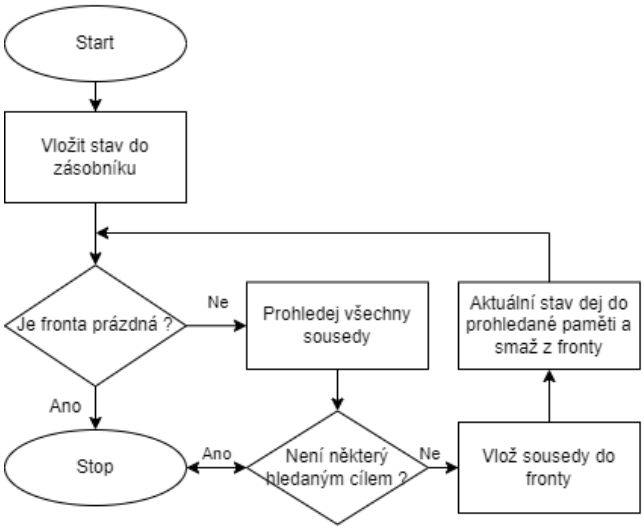


Obrázek 4.11 - Blokové schéma BiDi algoritmu

```
def FindSolution(self):
    i = 1
    j = 1

    while len(self.fifo) > 0:
        # Vlozi hodnotu fifo[0] do memory
        self.memory.append(self.fifo[0])
        self.bidi_memory.append(self.bidi_fifo[0])

        for actNum in range(0, self.numAct):
            # First direction
            temp = Node(i, self.fifo[0].depth + 1, actNum, self.fifo[0].id, self.Action(actNum, self.fifo[0].cube))

            if ~self.Check(temp, self.memory):
                if ~self.Check(temp, self.fifo):
                    self.fifo.append(temp)
                    i = i + 1
            del temp
            # Second direction
            bidi_temp = Node(j, self.bidi_fifo[0].depth + 1, actNum, self.bidi_fifo[0].id, self.Action(actNum, self.bidi_fifo[0].cube))

            # Kontrola, zda se již nenachází v paměti či zásobníku
            if ~self.Check(bidi_temp, self.bidi_memory):
                if ~self.Check(bidi_temp, self.bidi_fifo):
                    self.bidi_fifo.append(bidi_temp)
                    j = j + 1
            del bidi_temp

        del self.fifo[0]
        del self.bidi_fifo[0]

        if self.memory[-1].depth < self.fifo[0].depth or self.bidi_memory[-1].depth < self.bidi_fifo[0].depth:
            ret = self.findMatchingNodes(self.fifo, self.bidi_fifo)
            if ret:
                path = self.Solution(self.memory + self.fifo, self.bidi_memory + self.bidi_fifo, ret[0])
                return path
    return ("Nepodarilo se najit reseni") #self.Solution("Nan", 0)
```

Obrázek 4.12 – Programová implementace algoritmu BiDi

Algoritmus DFS nebylo vhodné pro tento problém implementovat, jelikož by nastat stav, kdy by pokračoval pořád dále a nikdy nenalezl řešení. Proto zde bylo lepší implementovat iterativní metodu DFS, kde se po každé iteraci zvedne hloubka prohledávání. Kvůli tomu je zde zavedena proměnná self.iterationDepth, která se pokud je self.lifo prázdné a nenalezlo se řešení, tak se zvedne o 1 a prohledávání začne od začátku.



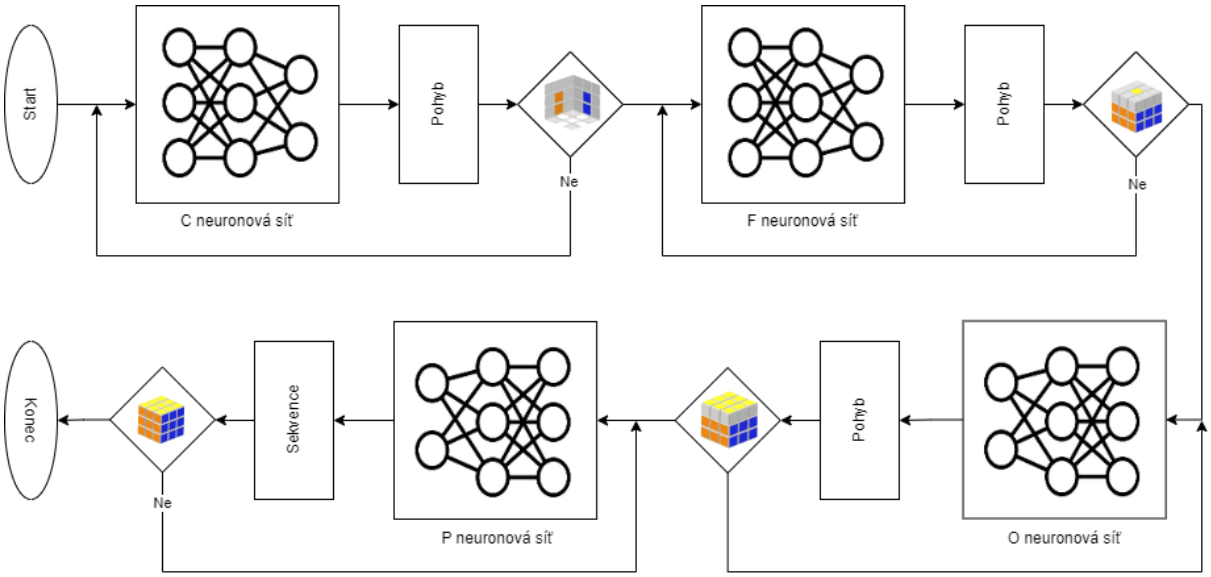
Obrázek 4.13 - Blokové schéma DFS algoritmu

```
def FindSolution(self):
    i = 1
    while len(self.lifo) > 0:
        self.memory.append(self.lifo[0])
        if self.CompareMatrix(self.lifo[0].cube, self.finalPos):
            break

        for actNum in range(0, self.numAct):
            temp = Node(i, self.lifo[0].depth + 1, actNum, self.lifo[0].id, self.Action(actNum, self.lifo[0].cube))
            if ~self.Check(temp, self.memory):
                if ~self.Check(temp, self.lifo):
                    if temp.depth <= self.iterationDepth:
                        self.lifo.append(temp)
                        i = i + 1
            if self.CompareMatrix(temp.cube, self.finalPos):
                self.lifo.insert(1, temp)
                break
            del temp

        del self.lifo[0]
        self.lifo = self.sortByDepthInNodes(self.lifo)

    if len(self.lifo) > 0:
        return self.Solution()

    self.iterationDepth += 1
    self.lifo = []
    self.memory = []
    self.lifo.append(Node(0, 0, 0, 0, self.initPos))
    return self.FindSolution()
```

Obrázek 4.14 - Programová implementace iterativního algoritmu DFS

4.2.4. Neuronové sítě – programová implementace

Pro implementaci neuronových sítí využíváme soustavy vícevrstvých dopředných neuronových sítí se zpětným šířením chyby. Cílem soustavy je implementace metody řešení CFOP. Každá je svým způsobem jedinečná, protože má jiný cíl, ke kterému se musí dostat. Tím, že je každá síť jiná, tak také potřebuje jiné množství trénovacích dat. Nejsložitější na natrénování je neuronová síť F, která skládá 2. vrstvu kostky. V tomto kroku je totiž zapotřebí udělat velké množství tahů oproti ostatním krokům, což znamená větší množství trénovacích dat. To je vidět v Tabulce 1, kde je napsáno, jak jsou sítě strukturovány a kolik dat bylo potřeba pro jejich natrénování. Jak bylo zmíněno výše, tak nejvíce dat bylo zapotřebí u sítě F, která má nakonec 4 skryté vrstvy.



Obrázek 4.15 - Struktura neurální sítě pro metodu CFOP

4.2.5. Zpětné šíření chyby (Gradient descent) s programovou implementací

U metody zpětného šíření chyby se vychází z teze, jak moc daný prvek je ovlivněn prvkem předchozím. U metody se vychází z toho, jak je uspořádána neuronová síť. Pokud vezmeme v úvahu klasickou dopřednou neuronovou síť s jednou vstupní vrstvou, jednou skrytou a jednou výstupní vrstvou, tak budeme postupovat následovně:

$$Z_1 = A \cdot W - \theta \tag{4.1}$$

$$Y_1 = akt(Z_1) \tag{4.2}$$

$$Z_2 = Y_1 \cdot V - \vartheta \tag{4.3}$$

$$Y_2 = akt(Z_2) \tag{4.4}$$

$$L = \frac{1}{2} \cdot (Y_2 - E)^2 \tag{4.5}$$

,kde E je předpokládaný výsledek

L je rozdíl očekávaného výstupu a aktuálního výstupu

Z jsou agregační funkce

Y jsou výstupní funkce

Akt(x) jsou aktivační funkce

```
def feedForward(self):

    # INPUT
    self.AgrHid = np.dot(self.W, self.InpX) + self.BI
    self.AktHid = self.softmax(self.AgrHid)

    # OUTPUT
    self.AgrOut = np.dot(self.V, self.AktHid) + self.BH
    self.AktOut = self.softmax(self.AgrOut)
```

Obrázek 4.16 - Feed forward s 1 skrytou vrstvou

Pokud budeme chtít zjistit, jak moc ovlivňuje výstup váha vstupu W, tak postupujeme naopak od konce řešení, tudíž se ptáme na změnu odchylky L, když změníme hledanou veličinu (váhu). Tudíž výpočet pro hledání změny L při změně W, uvažujeme-li neuronovou síť s jednou skrytou vrstvou, vypadá následovně:

$$\frac{dL}{dW} = \frac{dZ_1}{dW} \cdot \frac{dY_1}{dZ_1} \cdot \frac{dZ_2}{dY_1} \cdot \frac{dY_2}{dZ_2} \cdot \frac{dL}{dY_2} \tag{4.6}$$

$$\frac{dL}{dV} = \frac{dZ_2}{dV} \cdot \frac{dY_2}{dZ_2} \cdot \frac{dL}{dY_2} \tag{4.7}$$

$$\frac{dL}{d\theta} = \frac{dZ_1}{d\theta} \cdot \frac{dY_1}{dZ_1} \cdot \frac{dZ_2}{dY_1} \cdot \frac{dY_2}{dZ_2} \cdot \frac{dL}{dY_2} \tag{4.8}$$

$$\frac{dL}{d\vartheta} = \frac{dZ_2}{d\vartheta} \cdot \frac{dY_2}{dZ_2} \cdot \frac{dL}{dY_2} \tag{4.9}$$

$$\frac{dL}{dW} = A \cdot \frac{dAkt(Z_1)}{dZ_1} \cdot V \cdot \frac{dAkt(Z_2)}{dZ_2} \cdot (Y_2 - E) \tag{4.10}$$

$$\frac{dL}{dV} = Y_1 \cdot \frac{dAkt(Z_2)}{dZ_2} \cdot (Y_2 - E) \tag{4.11}$$

$$\frac{dL}{d\theta} = -1 \cdot \frac{dAkt(Z_1)}{dZ_1} \cdot V \cdot \frac{dAkt(Z_2)}{dZ_2} \cdot (Y_2 - E) \tag{4.12}$$

$$\frac{dL}{d\vartheta} = -1 \cdot \frac{dAkt(Z_2)}{dZ_2} \cdot (Y_2 - E) \tag{4.13}$$

$$W_{New} = W_{Old} + \frac{dL}{dW} * LR \tag{4.14}$$

$$V_{New} = V_{Old} + \frac{dL}{dV} * LR \tag{4.15}$$

$$\theta_{New} = \theta_{Old} + \frac{dL}{d\theta} * LR \tag{4.16}$$

$$\vartheta_{New} = \vartheta_{Old} + \frac{dL}{d\vartheta} * LR \tag{4.17}$$

Postup se vždy stejný a s postupným zesložiťováním si všimneme jisté pravidelnosti, čemuž se říká řetězové pravidlo (chain rule). Princip řetězového pravidla spočívá v tom, že chyba se šíří z výstupu sítě zpět k vstupům, přičemž se rozkládá na jednotlivé vrstvy sítě a přizpůsobuje váhy tak, aby minimalizovala tuto chybu. Řetězové pravidlo se opírá o derivace, které určují směr a velikost změn váhových koeficientů potřebných k minimalizaci chyby.

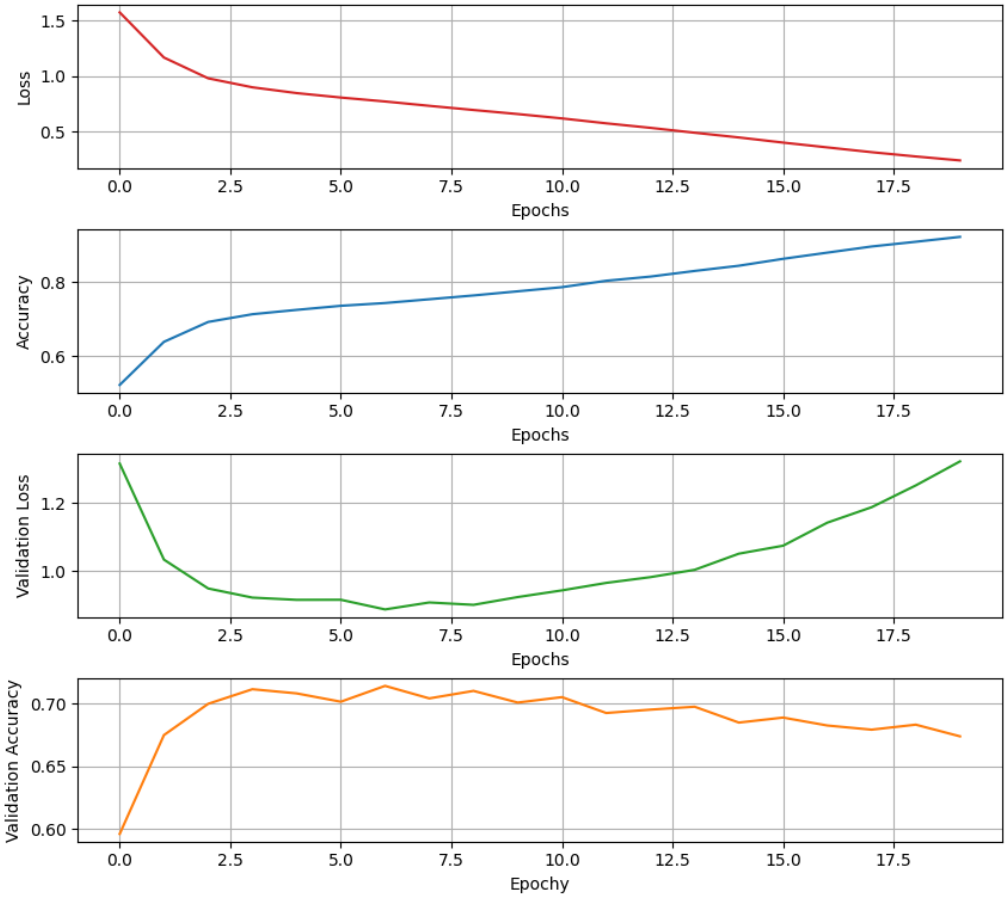
Avšak s každou vrstvou přibývá derivací / proměnných a postup se zesložiťuje. Také ovšem záleží na zvolených aktivačních funkcích. Takto poté probíhá trénování vah z testovacích dat.

```
def backPropagation(self, inp, target, leRa):
    # inputCube = self.flatCube(inp)
    self.InpX = self.flatCube(inp)
    self.feedForward()
    # matrix - matrix

    dA1 = self.der_softmax(self.AgrHid)
    dA2 = self.der_softmax(self.AgrOut)

    # error = 0.5*((self.AktOut - target)**2)
    derror = (self.AktOut - target)

    dW = np.dot(self.InpX, np.multiply(np.dot(self.V.T, np.multiply(dA2, derror)), dA1).T)
    dV = np.dot(self.AktHid, np.multiply(dA2, derror).T)

    dBI = np.multiply(dA1, np.dot(self.V.T, np.multiply(dA2, derror)))
    dBH = np.multiply(dA2, derror)


    self.W -= leRa * dW.T
    self.V -= leRa * dV.T
    self.BI -= leRa * dBI
    self.BH -= leRa * dBH
```

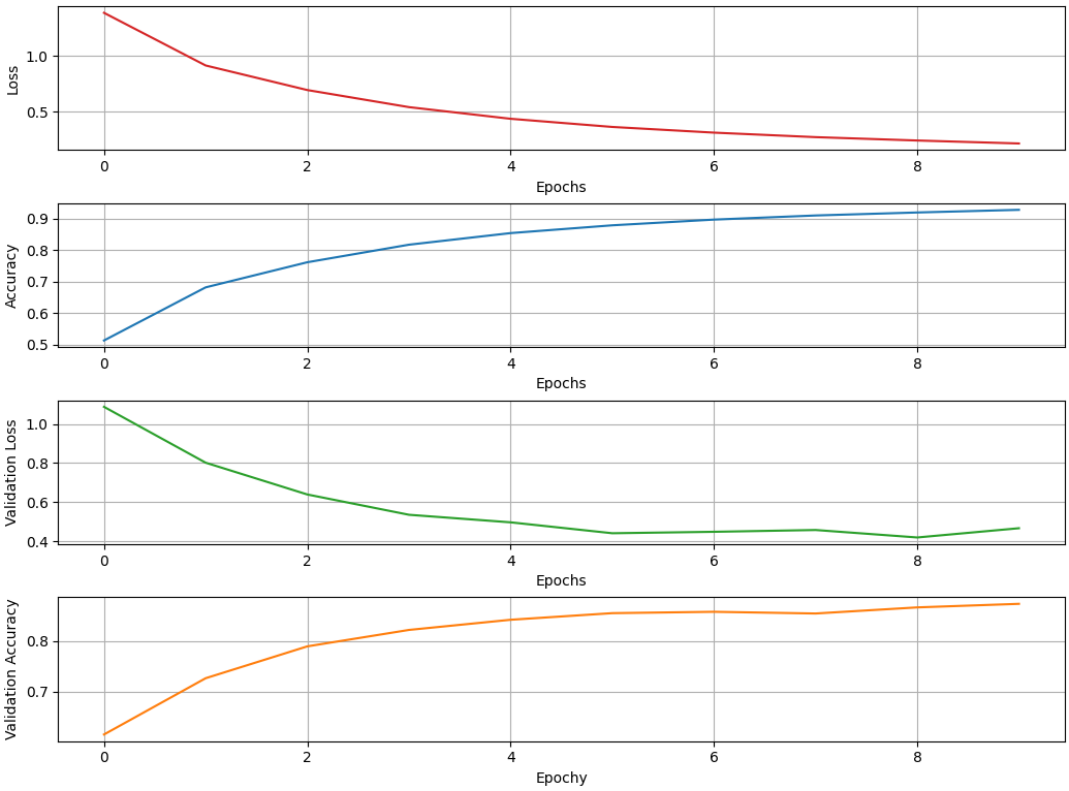
Obrázek 4.17 – Zpětné šíření chyby s 1 skrytou vrstvou

Kvůli vysoké výpočetní náročnosti při trénování, bylo nutné optimalizovat výpočetní proces. Toho bylo dosaženo využitím knihovny tensorflow. Ta ve třídě keras obsahuje nástroje pro vytvoření libovolného počtu skrytých vrstev s potřebnou optimalizací (optimalizační algoritmy Adam, RMSprop, SGD…). Kvůli této knihovně bylo též nutná trénovací data předem transformovat do podoby [n, 1], aby se dala použít na vstup neuronové sítě rovnou, a ne až v průběhu trénování. To byl také jeden z aspektů, proč výše uvedený algoritmus byl v porovnán s tímto velmi pomalý. Uvedené optimalizační algoritmy mají umožňovat lepší postupování gradientu, čímž se urychlí průběh testování.

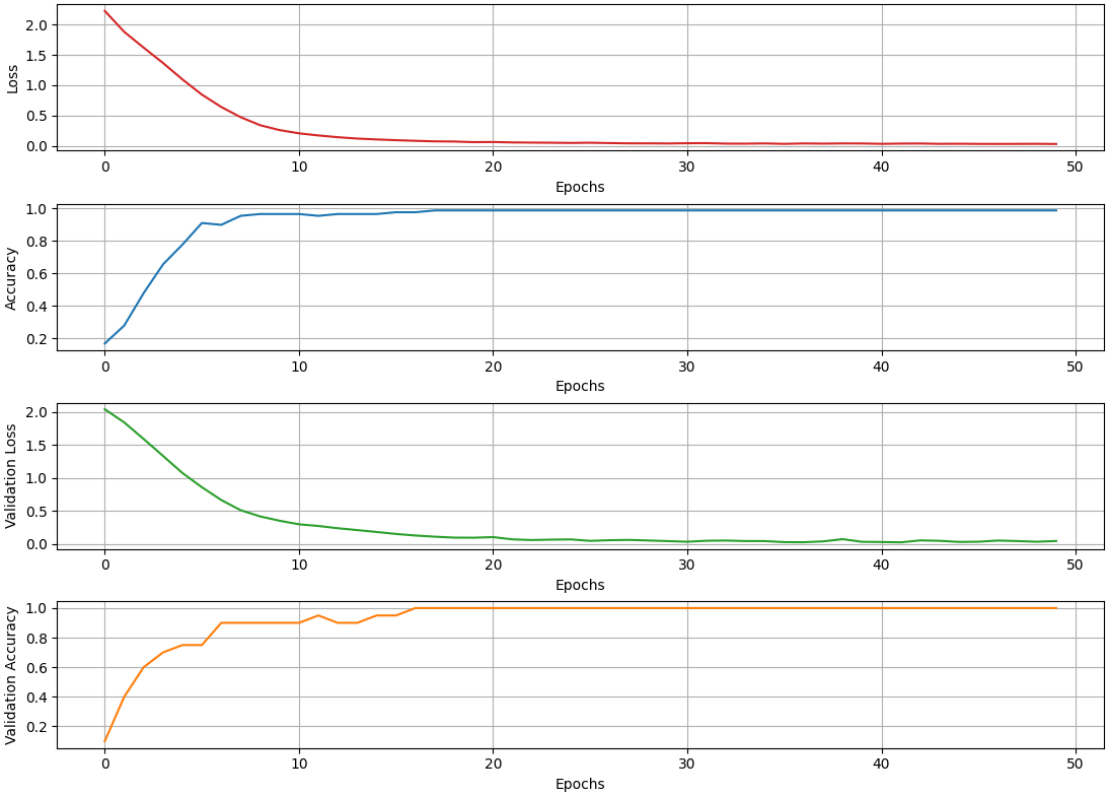
Výsledné modely poté uložíme a při opětovném spuštění programu, pokud je model správně uložený ho načteme. Pokud by nebyl, tak se opět spustí trénování.



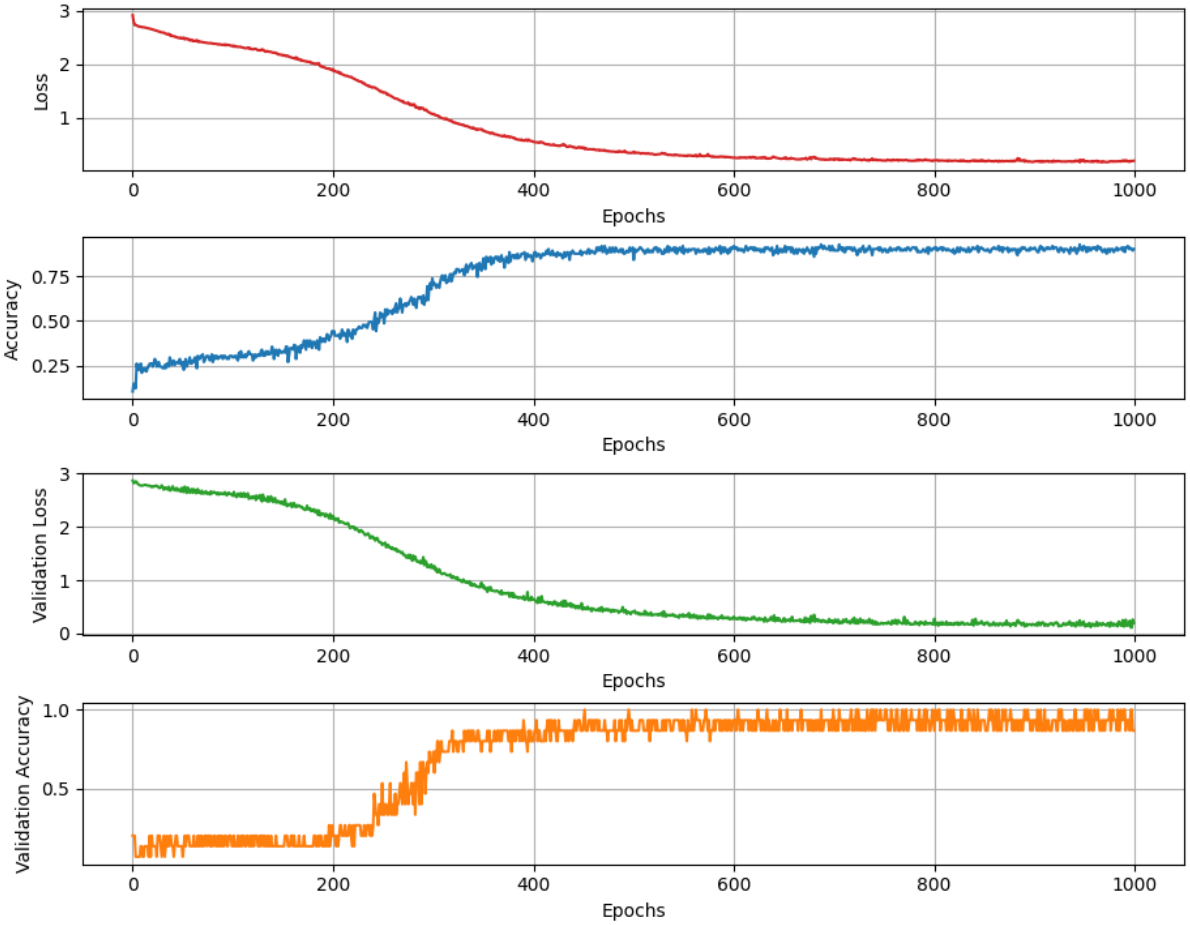
Obrázek 4.18 - Trénování C neurální sítě



Obrázek 4.19 - Trénování F neurální sítě



Obrázek 4.20 - Trénování O neurální sítě



Obrázek 4.21 - Trénování P neurální sítě

4.2.6. Systém strojového vidění

Jako systém strojového vidění (kameru) byla zvolena Trust Spotlight Pro, která obsahuje senzor s rozlišením 1280x1024px, což je plně dostačující pro tuto úlohu. Obsahuje přídavné osvětlení a je možno nastavovat zaostření objektivu. Do PC ji připojíme přes USB-A. Pro připojení do GUI systému stačí v kódu přenastavit proměnnou videoSource v hlavičce souboru gui.py na hodnotu 1 (hodnota 0 je defaultní připojená kamera – u notebooku webkamera). Pokud žádnou kameru připojenou nemáte, tak ponechte na hodnotě 0.

Obrázek 4.22 - Webkamera Trust Spotlight Pro

Metodologie

V tomto případě, tedy při rozeznávání barev z rubikovy kostky a jejich následné uložení do matic můžeme použít tyto metody:

- První metodou pro získání barevných matic je ruční zápis. Tento způsob se jeví sic jako nejjednodušší, ale také zároveň časově nejnáročnější a pro obecné použití zcela nepoužitelný. Je to tím, že při ručním zadávání začne hrát roli i lidský faktor – únava, špatný zrak, překlepy.
- Další metodou je filtrace barev za pomocí masek pro jednotlivé barvy. Obraz vyfiltrujeme podle požadovaných barev a výsledné obrazy poté sjednotíme. Tím dostaneme pouze obraz kostky a okolí stejných barev. Poté využijeme konturování, kde můžeme zjistit velikost konturovaných oblastí. Pokud bude kostka statická, tak můžeme postupně dojít k takovým filtrům, které odstraní menší oblasti anebo větší oblasti. Takto vyfiltrované oblasti potom můžeme analyzovat a uložit do matic.

- Další metodou je vytvoření konvoluční neuronové sítě (CNN) pro detekci objektů a jejich následné segmentace, tj. vytvoření sítě, která v zorném poli nalezne kostku a tu poté analyzuje – přečte jednotlivé barvy kostek. Tato metoda by byla zcela zásadní, pokud bychom měli obraz dynamický, protože síť by našla kostku, kdekoliv na obrázku, v jakékoliv velikosti a jakémkoliv rozpoložení (natočenou, rozloženou, s otočenou stranou). Ovšem pro vytvoření by na rozdíl od předchozích metod byla potřeba datová sada, ze které by se síť učila.

Provedení

Jelikož se jedná o komplikovaný systém, tak by bylo zbytečné do něj zakomponovat další systém s neuronovou sítí – v tomto případě složitější konvoluční neuronovou síť, která by rozpoznávala objekty a také extrahovala vlastnosti z těchto objektů (barvy). Navíc v našem případě bude objekt statický a vždy na stejném místě. Proto zde bylo využito jednodušší filtrování barev za pomocí barevných masek.

```
import cv2
import numpy as np

def nothing(x):
    pass

cap = cv2.VideoCapture(1)

cv2.namedWindow("Trackbar")
cv2.createTrackbar("H-L", "Trackbar", 0, 255,nothing)
cv2.createTrackbar("S-L", "Trackbar", 0, 255,nothing)
cv2.createTrackbar("V-L", "Trackbar", 0, 255,nothing)
cv2.createTrackbar("H-H", "Trackbar", 0, 255,nothing)
cv2.createTrackbar("S-H", "Trackbar", 0, 255,nothing)
cv2.createTrackbar("V-H", "Trackbar", 0, 255,nothing)

while True:

    _, frame = cap.read()
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    mask_lower = np.array([cv2.getTrackbarPos('H-L','Trackbar'),cv2.getTrackbarPos('S-L','Trackbar') ,cv2.getTrackbarPos('V-L','Trackbar')])
    mask_upper = np.array([cv2.getTrackbarPos('H-H','Trackbar'),cv2.getTrackbarPos('S-H','Trackbar') ,cv2.getTrackbarPos('V-H','Trackbar')])
    mask = cv2.inRange(hsv, mask_lower, mask_upper)

    cv2.imshow('mask',  mask)

    cv2.imshow("frame", frame)
    key = cv2.waitKey(1)

    if key == 27:
```

Obrázek 4.23 - Konfigurační funkce pro filtraci barev

Pro vytvoření masky byla využita funkce inRange v knihovně OpenCV. Ta za pomocí horních a spodních hranic barev v HSV spektru a vstupního snímku vytvoří masku filtrace pro jednotlivé barvy. Pro nalezení hranic byla vytvořena malá testovací funkce za jejíž pomoci byl

nalezen rozsah barev pro masky. Masky barev jsou velmi závislé na světle proto bude nejspíše nutné před každým spuštěním kalibrovat barvy.

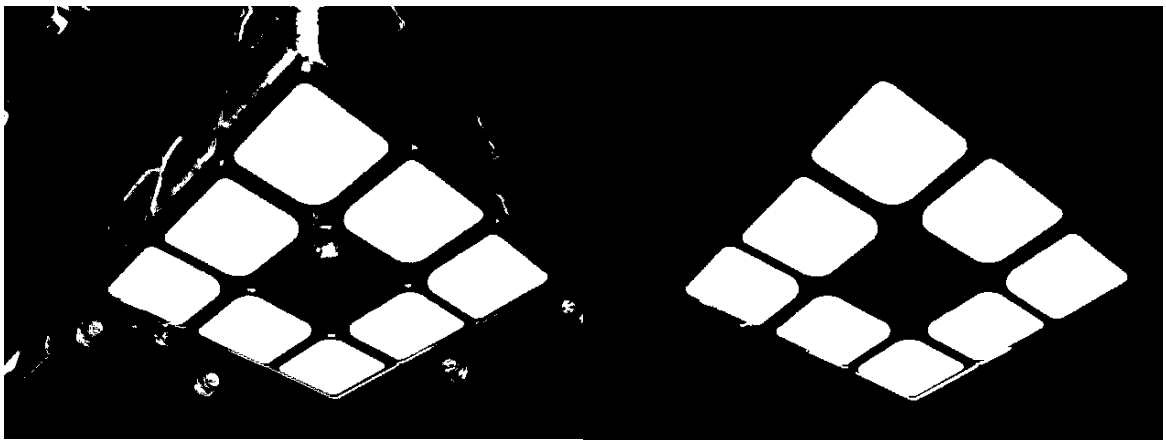
Výsledkem je např. takováto maska, která filtruje pouze modrou barvu.

```
blue_lower = np.array([90, 80, 100], np.uint8)
blue_upper = np.array([130, 255, 255], np.uint8)
```

Obrázek 4.24 - Deklarace masky barevného filtru

Tuto masku je nutno aplikovat na původní obraz., abychom dostali filtrovaný obraz pro jednotlivé barvy. Na již filtrovaném obrázku pro červenou je vidět, že jako červená byla detekována i některá další místa na obrázku. Abychom se jich zbavili je potřeba si „pohrát“ s nastavením odstínů barev pro masku nebo je filtrovat jiným způsobem.

Po aplikace funkce dilate, která rozšíří světelné oblasti (opakem je funkce erode) a následném filtrování oblastí od určité velikosti dostáváme obrázek uvedený níže. Jde o tentýž obrázek masky modré barvy, avšak byl odfiltrován „šum.“ Vyfiltrovaný obrázek, se může zdát „výborný“, avšak šum se vždy objeví a nemůžeme perfektně odfiltrovat jednotlivé barevné složky, takže po složení obrazu se šum opět objeví.

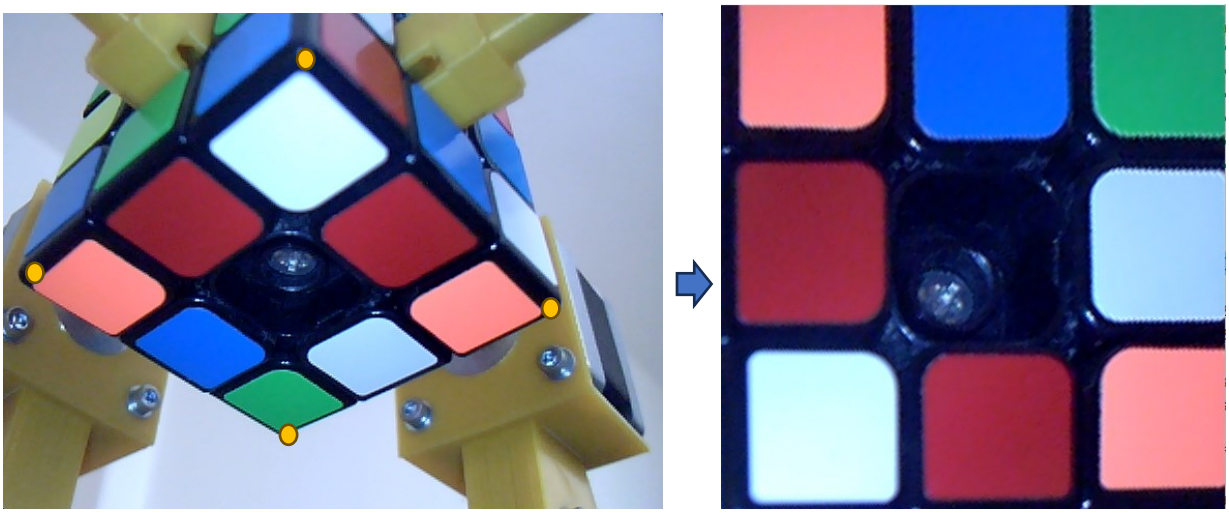


Obrázek 4.25 - Filtrace šumu

```python
def noiseFilter(mask):
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Odfiltrujte malé oblasti
    min_contour_area = 200  # nastavte minimální plochu pro odfiltrování
    max_contour_area = 2000
    filtered_contours = [cnt for cnt in contours if min_contour_area < cv2.contourArea(cnt) < max_contour_area]

    # Vytvořte prázdnou masku pro výsledné oblasti
    filtered_mask = np.zeros_like(mask)

    # Nakreslete filrované kontury na prázdnou masku
    cv2.drawContours(filtered_mask, filtered_contours, -1, (255, 255, 255), thickness=cv2.FILLED)

    return filtered_mask
```

Obrázek 4.26 - Funkce pro odstranění šumu

Pro vylepšení filtrace barev a odstranění působení pozadí, transformujeme obraz do podoby na obrázku 4.17 – vpravo. Využijeme funkcí *getPerspectiveTransform* a *warpPerspective*, které jsou součástí knihovny opencv.



Obrázek 4.27 - Transformace pohledu kamery

Pro konfiguraci bodů transformace byla vytvořena Transformační funkce, ve které ručně zadáme oblast z obrázku, kterou chceme transformovat. Oblast vybereme (naklikáme) za pomocí 4 bodů viz obrázku 4.17 - vlevo. Tím získáme souřadnice bodů původního obrázku a druhou sadu bodů zadáme manuálně, tj. určení, kde bude bod v novém obrázku. Obě sady zadáme do *getPerspectiveTransform*, která nám vrátí transformační matici, kterou poté zadáme do *warpPerspective* funkce. Výsledkem je transformovaný obrázek, který usnadní zobrazování dat a konfiguraci barevného spektra.

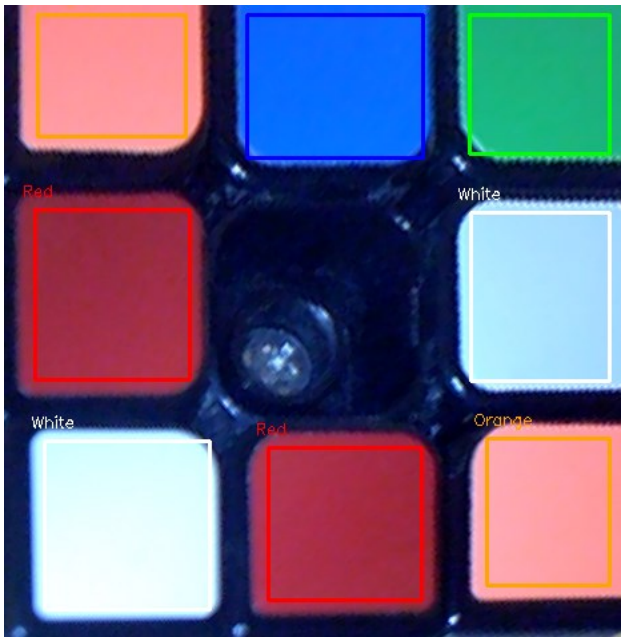
Nyní využijeme funkce *findContours*, která najde v masce barevné oblasti a poté využijeme funkce boundingRect, která nám najde parametry „obdélníkové oblasti“, tj. souřadnice x a y a délku a šířku obdélníku, který se do této oblasti vejde. Tyto parametry

využijeme a do obrazu zakreslíme oblasti. Každou barvu musíme udělat zvlášť a poté obraz opět složit.

```
import cv2
import numpy as np

array = np.zeros((4,2), int)
counter = 0

def click(event, x ,y ,flags, param):
    global counter
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(image, (x,y), 5, (255, 0, 0), -2)
        array[counter] = x,y
        counter += 1
        print(x,y)

cap = cv2.VideoCapture(1)

while True:

    _, image = cap.read()

    if counter == 4:
        points1 = np.float32([array[0],array[1],array[2],array[3]])
        points2 = np.float32([[0,0],[image.shape[0], 0], [0, image.shape[0]], [image.shape[0], image.shape[0]]])

        matrix = cv2.getPerspectiveTransform(points1, points2)
        final = cv2.warpPerspective(image, matrix, (image.shape[0], image.shape[0]))
        cv2.imshow("Output",final)


    cv2.namedWindow("Input")
    cv2.setMouseCallback("Input", click)
    cv2.imshow("Input", image)

    key = cv2.waitKey(1)

    if key == 27:
        break
```

Obrázek 4.28 - Transformační funkce

Po složení „obdélníkových oblastí“ musíme určit, zda máme všechny oblasti na straně kostky, tj. pokud program správně „profiltroval“ všech 8 kostek na stěně. Středové kostky kvůli uchopení v robotu byly odstraněny, tudíž se musí zachovávat při skenování sekvenční postup a kostka musí být do robotu vkládána vždy stejně, aby se mohl do stavové matice při skenu vždy zapsat barevný střed dané strany.



Obrázek 4.29 - Rozeznání barev stěny

GUI - Zobrazení kamery

Aby bylo možné kameru zasadit do GUI a zároveň pracovat s GUI, tak bylo nutné vše, co se týče kamerového systému spustit na samostatném vlákně. Tím je zajištěno to, že můžeme pracovat s GUI a zároveň se obnovuje obraz z kamery, a to v reálném čase. To ale způsobuje, že cokoliv budeme dělat na kameře a bude se týkat celé aplikace, což je např. skenování nebo sériová komunikace, tak musíme opět vracet zpět.

Pro skenování stačí do update funkce přidat návratovou hodnotu, která v sobě bude mít naskenovanou matici kostky. Pro komunikaci bylo nutno přidat proměnnou již při vytváření funkce, protože nemůžeme vytvořit dvě spojení na jednu sériovou linku. Pokud by se tak dělo, tak nám v konzoli vyskočí chybová hláška. Takže byla vytvořena instance sériové komunikace při spuštění aplikace a jejíž variace v jiné proměnné byla uložena do proměnných ve funkci kamery.

4.3. Sériová komunikace

Pro sériovou komunikaci bylo využito USB připojení mezi PC a Arduinem, přes které se klasicky programuje i Arduino. Při startu aplikace v pythonu musíme však v aplikaci, zatím manuálně, nastavit COM port, ke kterému Arduino připojujeme. Komunikace s Arduinem probíhá přes UART komunikaci.

```
serialComm = serial.Serial(port='COM7', baudrate=115200, timeout=1)
```

Obrázek 4.30 - Ukázka nastavení připojení COM portu

Univerzálně asynchronní přijímač/vysílač (UART) je sériové komunikační rozhraní, které poskytuje datový převod z paralelního na sériové a naopak. Je univerzální proto, že parametry jako přenosová rychlost, datová rychlost atd. jsou nastavitelné (Campbell, 2016.)

Využíváme zde knihovnu pyserial, která je pro sériovou komunikaci vytvořena. Pro zjednodušení přenosu posíláme pouze bytové informace – informace v rozsahu (0x00 – 0xFF), což je pro naše potřeby dostačující.

```
serialComm.flushInput()
serialComm.flushOutput()
serialComm.write(bytes(selectInstruction(move), 'utf-8'))
```

Obrázek 4.31 - Ukázka komunikace po sériové lince

Aby též nedošlo k přepsání instrukcí, tak aplikace vždy čeká, než zpětně z Arduina nedostane potvrzení o dokončení instrukce (0x78 → 120 → „utf-8 → x“). Teprve poté se může zaslat další instrukce. Též bylo nutno před čtením čistit buffer sériové komunikace, protože se v něm občas uložilo potvrzení o dokončení instrukce (x) a program poté zaslal další instrukci, která, z důvodů toho, že bylo Arduino „zaneprázdněno“, tak byla ztracena.

Po správném připojení, již stačí pracovat s GUI, které poté samostatně pracuje se sériovou komunikací a při stisknutí tlačítka pošle příkaz do Arduina. Příkaz jsou pouze čísla, která reprezentují instrukce, jež jsou v Arduinu definovány.

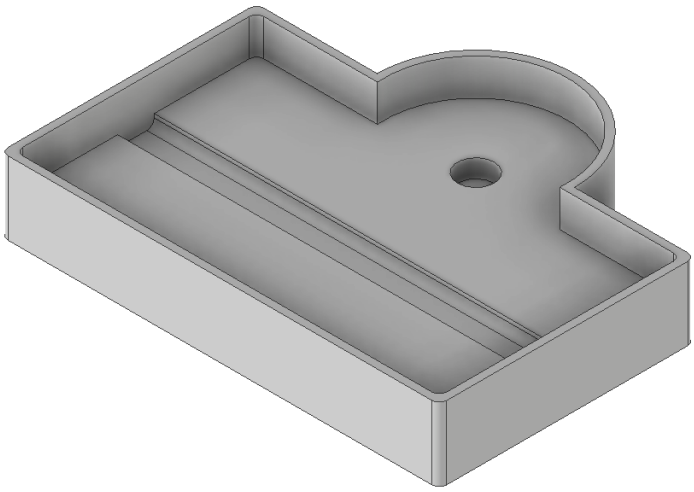
```
def selectInstruction(move):
    match(move):
        case "R":
            return "1"
        case "Rp":
            return "2"
        case "R2":
            return "3"
        case "L":
            return "4"
        case "Lp":
            return "5"
        case "L2":
            return "6"
        case "B":
            return "7"
        case "Bp":
            return "8"
```

```
while (Serial.available()){
  int Move = Serial.readString().toInt();
  switch (Move){
    case 1:
      Move_Side(pin_side_R, pin_dir_R ,CW, numOfTicsMove);
      break;
    case 2:
      Move_Side(pin_side_R, pin_dir_R ,CCW, numOfTicsMove);
      break;
    case 3:
      Move_Side(pin_side_R, pin_dir_R ,CW, 2 * numOfTicsMove);
      break;
    case 4:
      Move_Side(pin_side_L, pin_dir_L ,CW, numOfTicsMove);
      break;
    case 5:
      Move_Side(pin_side_L, pin_dir_R ,CCW, numOfTicsMove);
      break;
    case 6:
      Move_Side(pin_side_L, pin_dir_R ,CW, 2*numOfTicsMove);
      break;
    case 7:
      Move_Side(pin_side_B, pin_dir_R ,CW, numOfTicsMove);
      break;
```

Obrázek 4.32 - Přetypování příkazů z Pythonu na příkazy v Arduinu

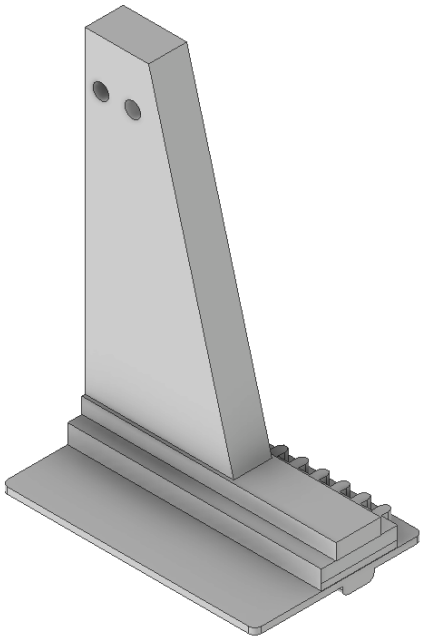
5. Konstrukční návrh

Konstrukce je rozdělena na 5 dílů. Spodní díl spojuje ostatní části konstrukce do jednoho celku a poskytuje tak celé konstrukci větší stabilitu a nosnost. Ostatní 4 díly jsou totožné. Umožňují uchycení a manipulaci s motory, které jsou umístěny na posuvníku a na vrchním krytu základny.



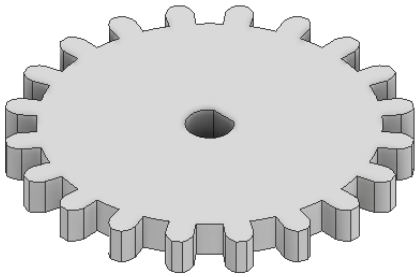
Obrázek 5.1 - Návrh základny pro pohyb s motory

Základna posuvníku pro motory, která je na obrázku 5.1. je vymodelována, tak aby do drážky ve středu základny mohl být umístěn posuvník viz obrázek 5.2, a do otvoru vpravo bude

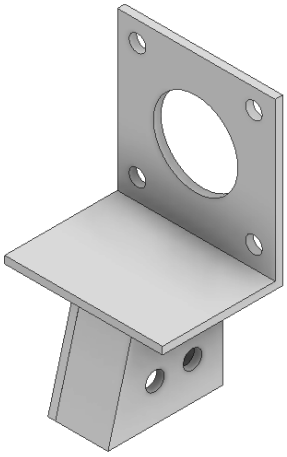


Obrázek 5.2 - Návrh posuvníku pro motory

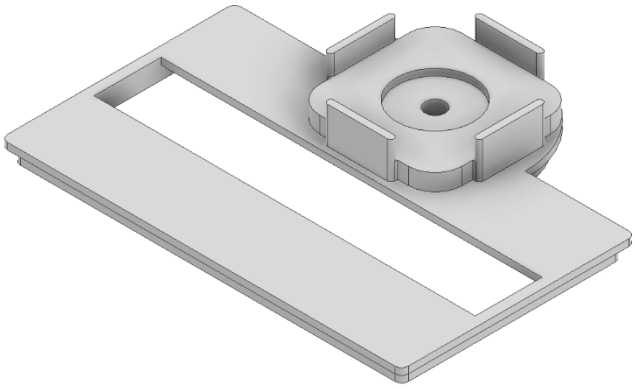
umístěno ozubené kolo, které je na obrázku 5.3. To spolu s posuvníkem přes ozubený převod bude převádět rotační pohyb na posuvný. Na posuvníku jsou otvory, které slouží k připevnění držáku motoru viz obrázek 5.4, který na něm bude umístěn. Poslední částí bočních částí je kryt, který upevní posuvník uvnitř základny spolu s ozubeným převodem. Na krytu je umístění pro další motor, jehož hřídel bude umístěna přes kryt do ozubeného kola.



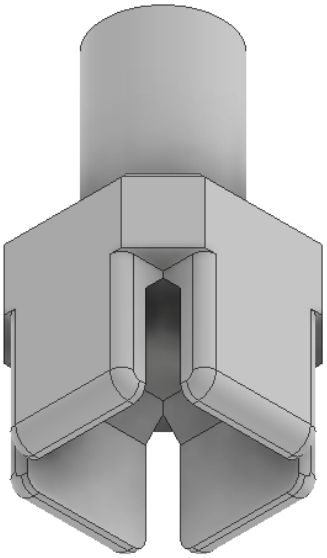
Obrázek 5.3 - Návrh ozubeného převodu



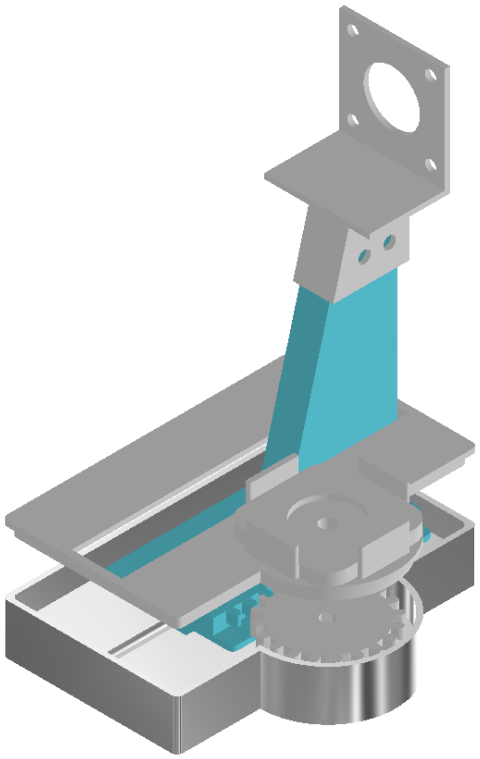
Obrázek 5.4 - Návrh držáku motoru



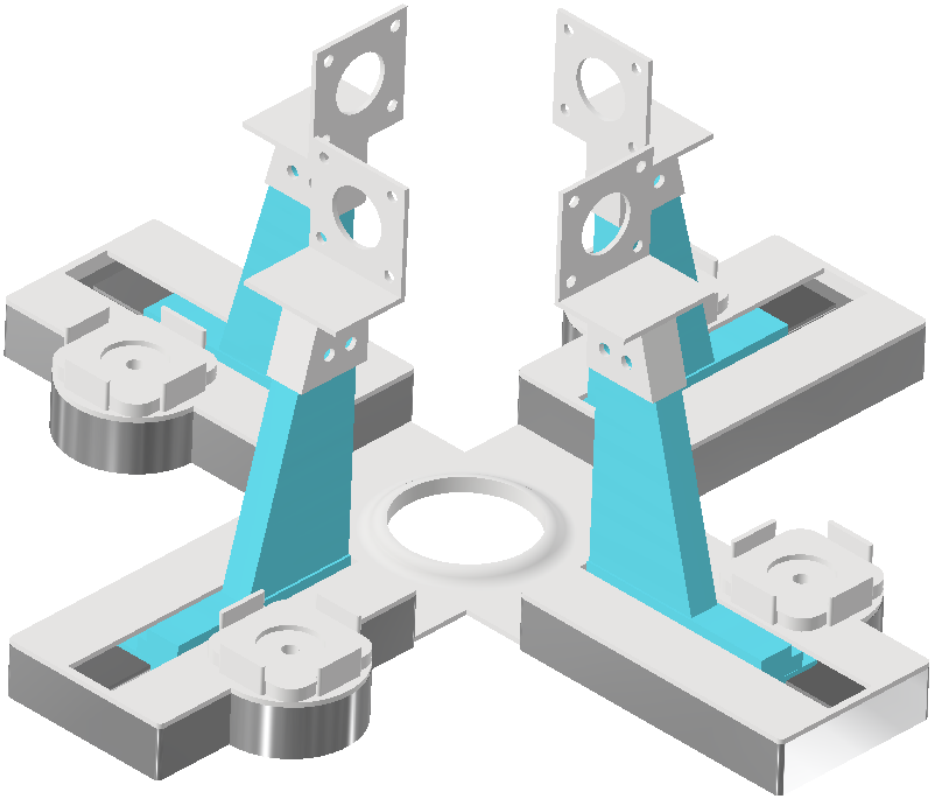
Obrázek 5.5 - Návrh krytu základny



Obrázek 5.6 - Návrh uchopovací součásti



Obrázek 5.7 - Sestava konstrukce



Obrázek 5.8 - Kompletní sestava

6. Závěr

Závěrem lze podotknout, že v umělé inteligenci je budoucnost. Z toho vychází, že by měla být možnost implementace do jakéhokoliv zařízení, ať už z části nebo pro úplné nahrazení dosavadní činnosti. Tato práce měla být pokusem o to, zda je toho umělá inteligence schopna a jakým způsobem ji lze implementovat.

Pro umělou inteligenci, kterou má tento robot testovat zde byly implementovány tři známé neinformované algoritmy, jež prohledávají stavový prostor a soustava neuronových sítí, která byla naučena na algoritmickou metodu skládání Rubikovy kostky, kde výsledkem je vždy řešení, doba trvání a paměťová náročnost.

Pro neinformované algoritmy je výsledek nedostačující z důvodů vysoké výpočtové náročnosti kostky. Navíc při testování bylo zjištěno, že knihovna tkinter, která vytváří GUI, na kterém vše běží, i přes fakt, že algoritmy běží na nezávislých vláknech, tak je zpomalována procesem výpočtu a občas se stane, že aplikace v průběhu výpočtu spadne.

Tabulka 2 - Přehled časových závislostí algoritmů

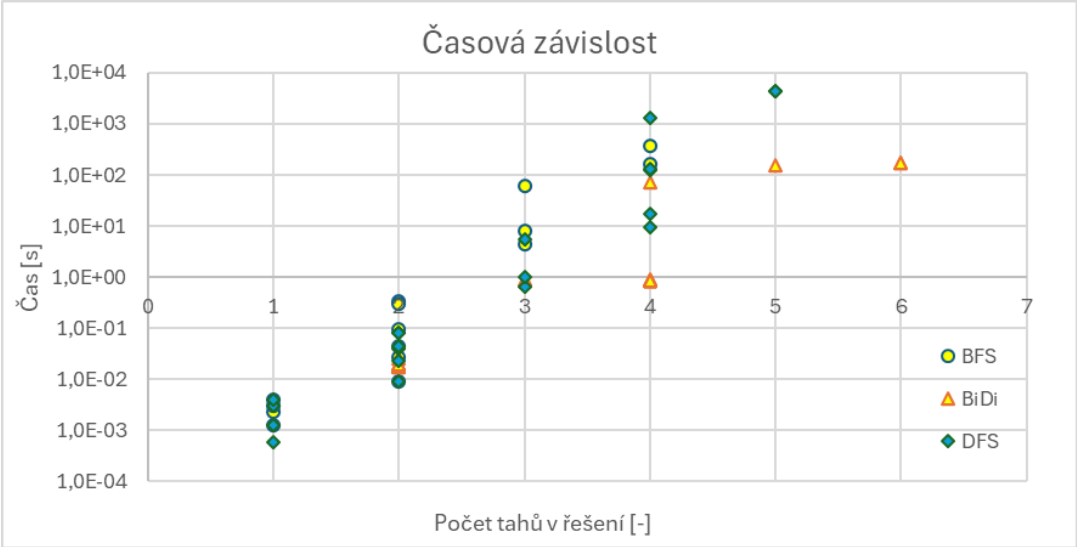
Počet tahů	BFS - Tmin	BFS - Tmax	BiDi - Tmin	Bidi - Tmax	DFS - Tmin	DFS - Tmax
1	0,00227165	0,00399971	0,01505415	0,01698954	0,00058474	0,004005194
2	0,0275588	0,29534483	0,01750565	0,01977754	0,00933862	0,080070496
3	8,06714392	60,790894	0,79210337	0,89567675	0,99069071	5,600423098
4	361,77215	-	0,83710337	0,91245675	17,4823961	1275,595304

Data v Tabulce 2 a Tabulce 3 byla získána spuštěním řešení kostky, které mělo algoritmu zabrat nejméně/nebo nejvíce času. Tento časový rámec byl založen na pořadí prohledávání jednotlivých tahů: pro nejrychlejší časy byly použity tahy na začátku pole, zatímco pro nejpomalejší časy tahy na konci pole. Drastické rozdíly mezi jednotlivými algoritmy byly patrné již po 3 tazích.

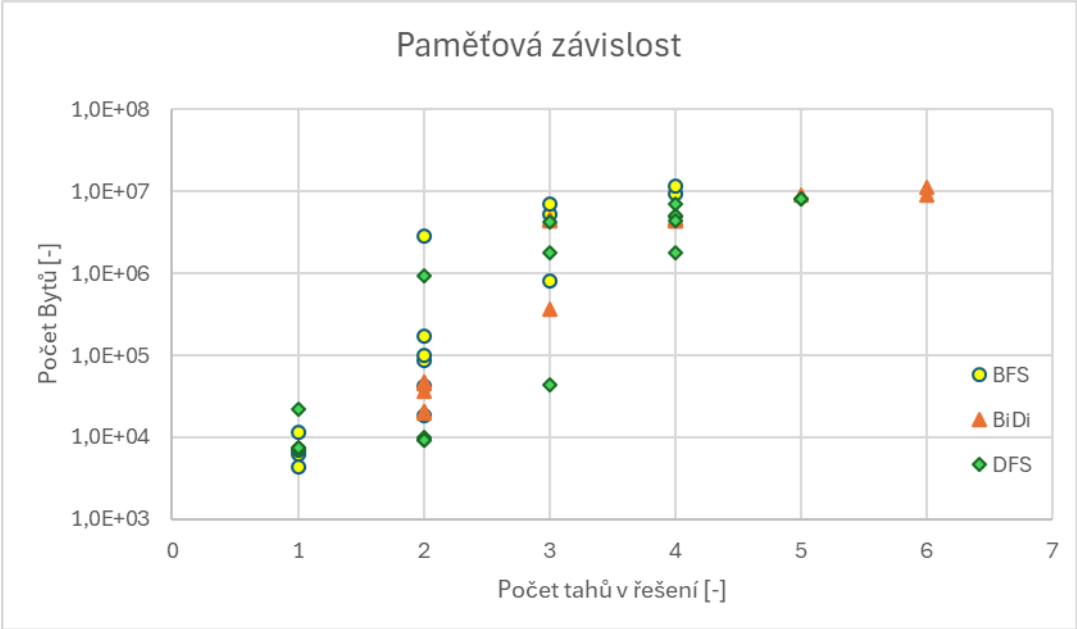
Tabulka 4 - Přehled paměťových závislostí algoritmů

Počet tahů	BFS - Mmin	BFS - Mmax	BiDi - Mmin	Bidi - Mmax	DFS - Mmin	DFS - Mmax
1	6232	11568	19901	36900	7501	21997
2	42520	2803586	20790	44661	9252	933230
3	5206554	7083562	4480001	4453780	1779160	4176171
4	11549856		4448195	4441935	6880615	8149850

Tabulka s paměťovou závislostí byla odhadována podobně jako Tabulka 2. V aplikaci byly zaznamenány případy, kdy byla paměťová závislost výrazně vyšší, než bylo očekáváno. Stačilo program spustit znovu a paměťová závislost se dostala do přijatelných mezí. To naznačuje, že na pozadí mohou být spuštěny procesy, které omezují výpočetní proces nebo nutí program alokovat více paměti.



Obrázek 6.2 – Časová závislost zaznamenaných hodnot



Obrázek 6.3 - Paměťová závislost zaznamenaných hodnot

Z pohledu neuronových sítí, jež tvoří značnou část umělé inteligence, je výsledek celkem slušný. Implementace metody CFOP nejvíce upadá na prvních dvou krocích, na kroku C a na kroku F, což je pochopitelné, jelikož jsou to matematicky nejvíce náročné části kostky, avšak zarážejícím faktem bylo, že se síť byla schopna lépe učit metodu F, která je náročnější než metodu C. Ta i přes různé změny faktorů (počet vrstev, neuronů, aktivačních funkcí, ztrátových funkcí nebo trénovacích dat) nebyla schopna natrénovat více jak 75%, což ve finálním výsledku naznačuje, že celkový výsledek dopadl velmi špatně, protože se jedná o první fázi metody a když selže první, tak ostatní selžou též.

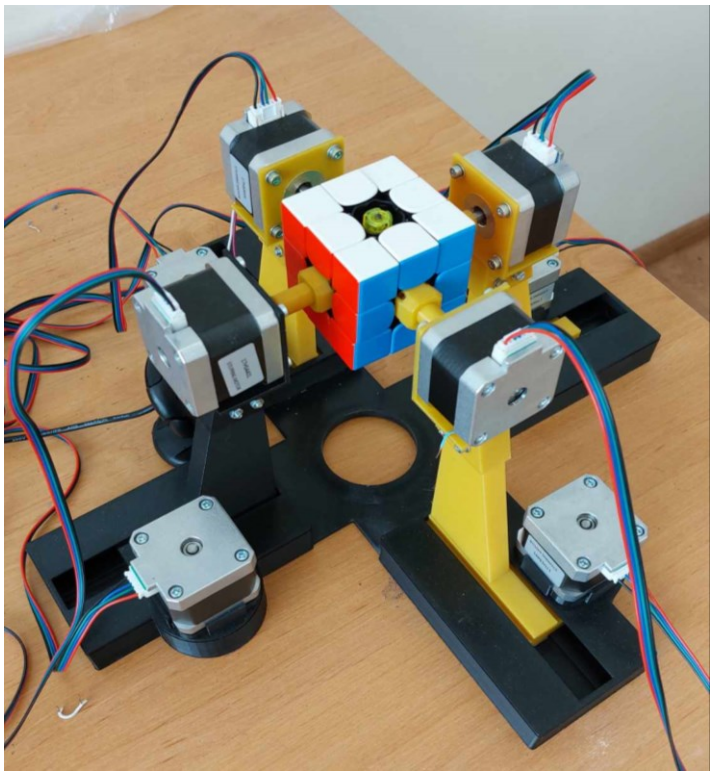
Co se týče výpočtu řešení, tak se velmi liší, v závislosti na řešení kostky.

Tabulka 6 - Neuronové sítě

Síť	Počet skrytých vrstev	Počet trénovacích dat	Úspěšnost / ztrátovost testovacích dat
Neurální síť C	3	50000	75,23%, 0.9685
Neurální síť F	4	300000	87%, 0.4341
Neurální síť O	2	100	95,45%, 0.1317
Neurální síť P	2	120	92,31%, 0.1541

Další částí je zde systém strojového vidění. Ten výsledkem dopadl velmi dobře, avšak je velmi náchylný na světlo, takže stačí změnit osvětlení a celý systém se může konfigurovat znovu, což ve finálním výsledku přináší více práce než užitku, takže jistě dalším vylepšením by byl systém, který by auto-konfiguroval barvy pro lepší detekci a rovněž systém pro rozpoznání tvarů a následné transformaci, jež je zobrazena na Obrázku 4.21.

Celkově pro tento model by se dalo najít mnoho vylepšení, jak hardwarových, tak softwarových



Obrázek 6.4 - Finální podoba zařízení

LITERATURA

LENDARIS, G; MATHIA, K 1999 : *"Linear Hopfield networks and constrained optimization"*. [cit 24-03-06]. Dostupné z článku: January 1999, IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics: a publication of the IEEE Systems, Man, and Cybernetics Society 29(1):114-118

SAHA Sumit, 2018 „*A Comprehensive Guide to Convolutional Neural Networks"*. [online]. [cit 24-03-06] Dostupné z: https://towardsdatascience.com

CIABURRO, G; VENKATESWARAN, B 2017 : *„Neural Networks with R: Smart models using CNN, RNN, deep learning, and artificial intelligence principles."* ISBN: 9781788397872

SZELISKI, R 2010. *„Computer Vision: Algorithms and Applications*." [cit 24-02-10]. ISBN 9781848829350.

VIRTUAL LABS, 2019 *„Artificial Neural Networks"* [cit 24-03-14] Dostupné z: https://cse22-iiith.vlabs.ac.in/Introduction.html

CAMPBELL, S 2016. „*Basics Of UART Communication*". [online]. [cit 24-04-5]. Dostupné z: https://www.circuitbasics.com/basics-uart-communication/

HAYKIN, S 1999. „*Neural networks: a comprehesive foundation. 2nd ed."* [cit 24-02-14]. ISBN 978-0132733502.

BRILLIANG.ORG. „*Backpropagation"* [online]. [cit. 24-04-06]. Dostupné z: https://brilliant.org/wiki/backpropagation/.

CHOI R., COYNER A., KALPATHY J., CHIANG F., CAMPBELL P., 2020 „*Introduction to Machine Learning, Neural Networks and Deep Learning*" [online]. [cit. 24-3-3]. Dostupné z: https://tvst.arvojournals.org/article.aspx?articleid=2762344

PŘÍLOHY

A – CD

B – Uživatelská příručka

C – Výrobní dokumentace

Příloha A

Příloha k Diplomové práci

Robotické zařízení pro testování algoritmů umělé inteligence

Bc. Bohumil Lotz

CD

OBSAH

1. Text diplomové práce ve formátu PDF.
2. Firmware aplikace
3. 3D modely tisknutelných dílů
4. Schémata zapojení, DPS

Příloha B

Příloha k Diplomové práci

Robotické zařízení pro testování algoritmů umělé inteligence

Bc. Bohumil Lotz

UŽIVATELSKÁ PŘÍRUČKA

OBSAH

SEZNAM ILUSTRACÍ

ÚVOD

Tato příloha poskytuje podrobný návod a doporučení ohledně softwarových požadavků nutných k úspěšnému spuštění a bezpečné manipulaci s robotickým zařízením. Obsahuje informace o potřebném softwaru, jeho instalaci a konfiguraci, stejně jako o nezbytných bezpečnostních opatřeních, která je třeba dodržovat při práci s robotem.

1. POŽADAVKY NA SW

Pro správnou funkčnost aplikace je zapotřebí nainstalovat Python 3.10 nebo novější. Ve verzi 3.10 bylo implementována funkce match-case, která je v programu hojně využívána kvůli koncepci označení jednotlivých tahů kostky.

Pokud program není spouštěn přes jakékoliv Python IDE, ale pouze přes python konzoli, tak je nutné, aby do každého souboru byla dopsána absolutní cesta k souboru viz obr. XX. Nynější implementace totiž počítá s projektovým otevřením v IDE, čímž se absolutní cesta nastaví automaticky.

```
import sys
sys.path.append('/home/pi/Desktop/Diplom_Prace')
```

Obrázek 1.1 - Nastavení absolutní cesty k projektu

Dále pro funkčnost je potřeba do pythonu nainstalovat řadu knihoven, které jsou použiti v kódu aplikace. Zde je seznam:

- Pillow 10.1.0
- opencv-python – 4.8.1.78
- matplotlib – 3.8.1
- pyserial – 3.5
- numpy – 1.26
- tensorflow – 2.16.1
- threading (součást IDE pycharm)
- tkinter (součást IDE pycharm)
- pathlib (součást IDE pycharm)
- time (součást IDE pycharm)
- json (součást IDE pycharm)
- copy (součást IDE pycharm)
- math (součást IDE pycharm)

Dále pro funkčnost aplikace je zapotřebí mít dostupné 2 USB porty. Jeden pro sériovou komunikaci s Arduinem a druhý pro připojení kamery.

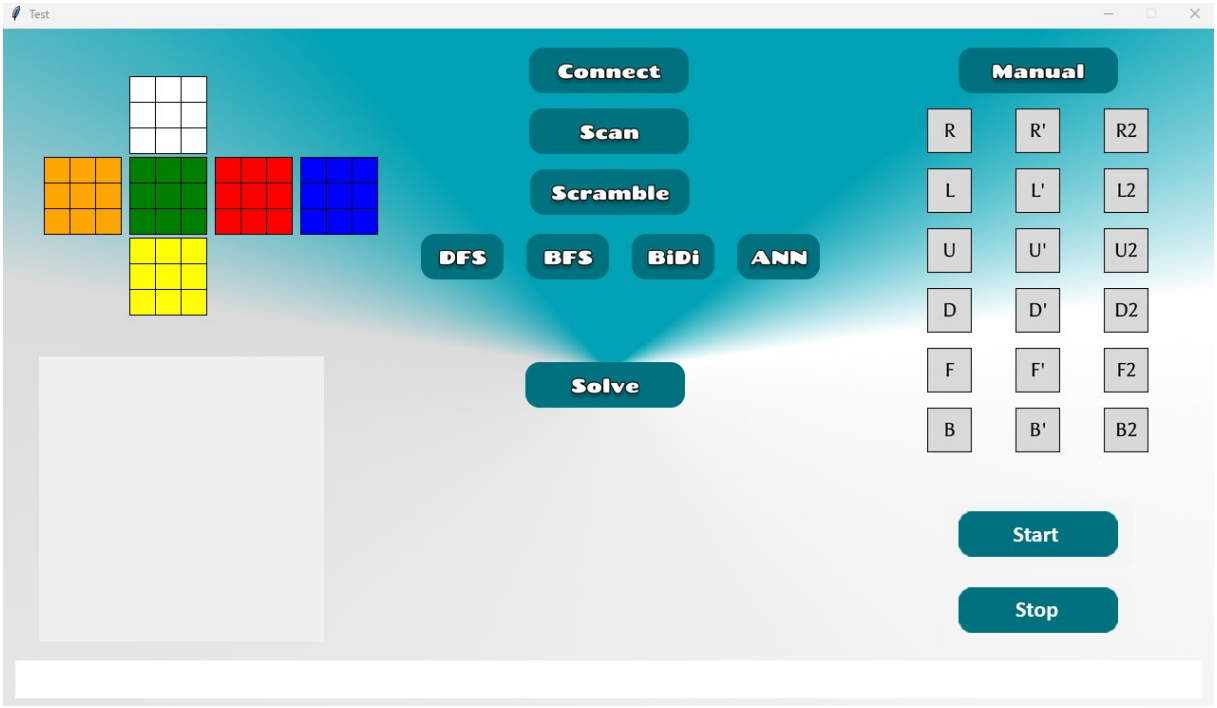
2. SPUŠTĚNÍ APLIKACE

Pro spuštění aplikace je v souboru gui.py ve složce GUI nastavit, na který port budeme připojovat sériovou komunikaci, aby vše bezpečně fungovalo. Na každém zařízení bude jiné číslo portu. Pro zjištění otevřete Správce zařízení ve Windows a v sekci Porty (COM a LTP) se po připojení objeví číslo portu, na kterém je zařízení připojeno.

```
serialComm = serial.Serial(port='COM7', baudrate=115200, timeout=1)
```

Obrázek 1.2 - Nastavení seriového portu

Poté stačí spustit gui.py, aby se otevřelo na uživatelské rozhraní, které se při používání aplikace používá. Při stisknutí tlačítka Connect připojíme / odpojíme kameru. Tlačítko Scan spustí sekvenci, která zmapuje celou kostku a její povrch zobrazí vlevo nahoře v zobrazení kostky. Tlačítko Scramble vybere z databáze náhodný scramble pro kostku a následně ji zamotá. Následující tlačítka vezmou aktuální stav kostky a hledají řešení podle zvoleného algoritmu. Po nalezení řešení se řešení, spolu s délkou hledání vypíší do konzole. Tlačítko solve vezme řešení nalezené algoritmem a přes sériovou komunikaci ho aplikuje na reálnou kostku.



Obrázek 1.3 – GUI aplikace

3. OCHRANA A BEZPEČNOST

Toto zařízení pracuje s nízkým napětím 12 V, což může působit bezpečně, avšak je důležité si uvědomit, že proud procházející konstrukcí může dosahovat až 3 A. I při nízkém napětí může takový proud způsobit zranění nebo jiné škody. Proto je zásadní respektovat bezpečnostní pokyny.

Je důrazně nedoporučeno manipulovat s konstrukcí zařízení, pokud je pod proudem. Vkládání rukou do konstrukce nebo jakékoliv další zásahy by měly být prováděny pouze tehdy, když je zařízení odpojeno od napájení a bezpečně vypnuté.

Stejně tak není vhodné sahat do konstrukce zařízení, pokud se pohybuje nebo je ve stavu provozu. Pohybující se části mohou být nebezpečné a představovat riziko úrazu. Je nutné dodržovat opatření k ochraně před nebezpečím, jako je vzdálenost od rotujících částí nebo mechanismů.

Celkově je důležité být opatrný a dodržovat bezpečnostní pokyny, aby se minimalizovalo riziko úrazu nebo poškození zařízení.

ZÁVĚR

Zařízení je navrženo pro intuitivní ovládání, avšak je nutno dbát bezpečnostních pokynů, aby se předešlo jakýmkoliv úrazům, které mohou vzniknout.

Příloha C

Příloha k Diplomové práci

Robotické zařízení pro testování algoritmů umělé inteligence

Bc. Bohumil Lotz

VÝROBNÍ DOKUMENTACE

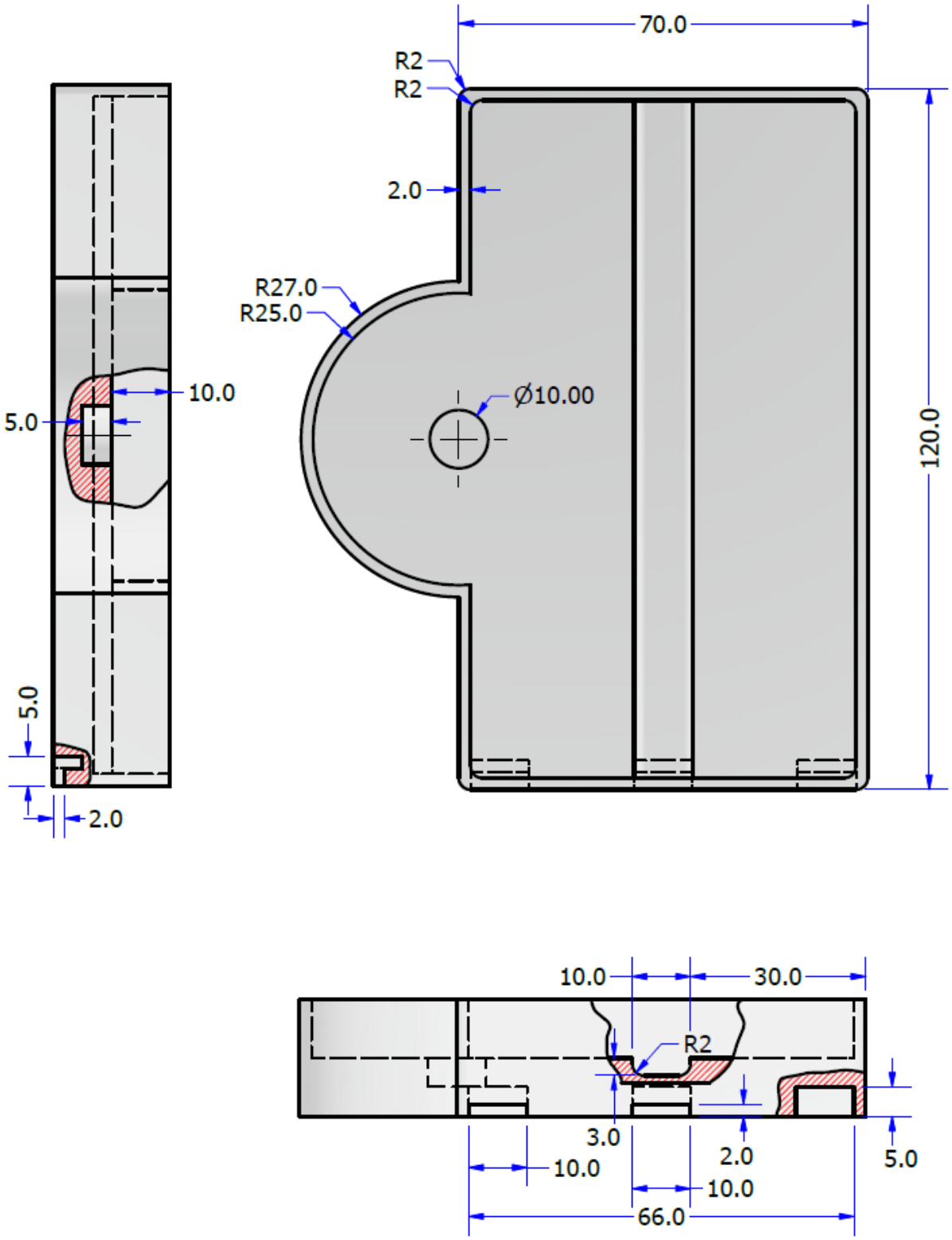
OBSAH

SEZNAM ILUSTRACÍ A TABULEK

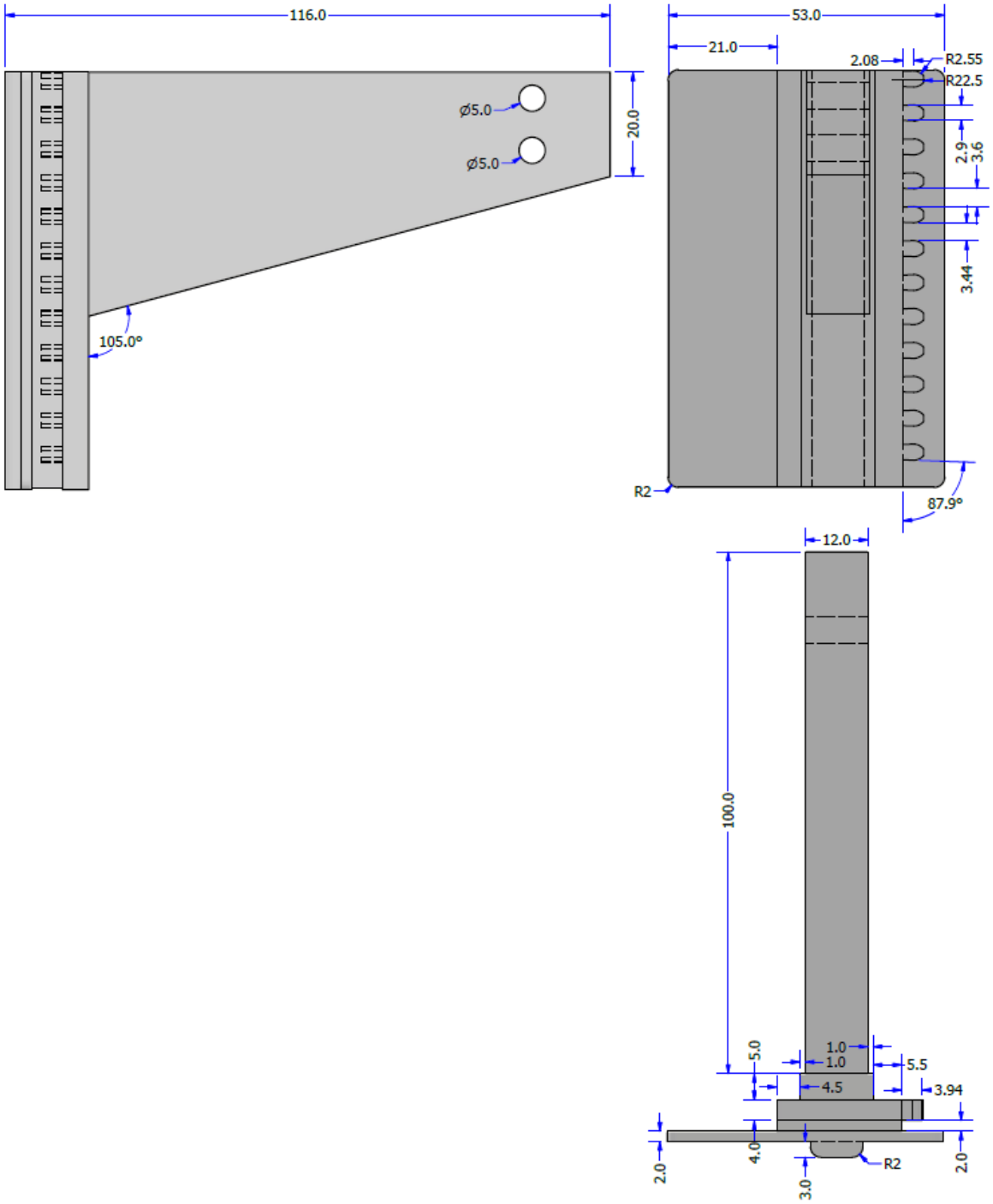
ÚVOD

V této příloze jsou výkresy a schémata potřebná pro výrobu vlastního zařízení. Jednotlivé komponenty jsou navržené pro 3D tiskárnu, proto je u některých komponent užita malá tolerance, která vzniká chybou při 3D tisku.

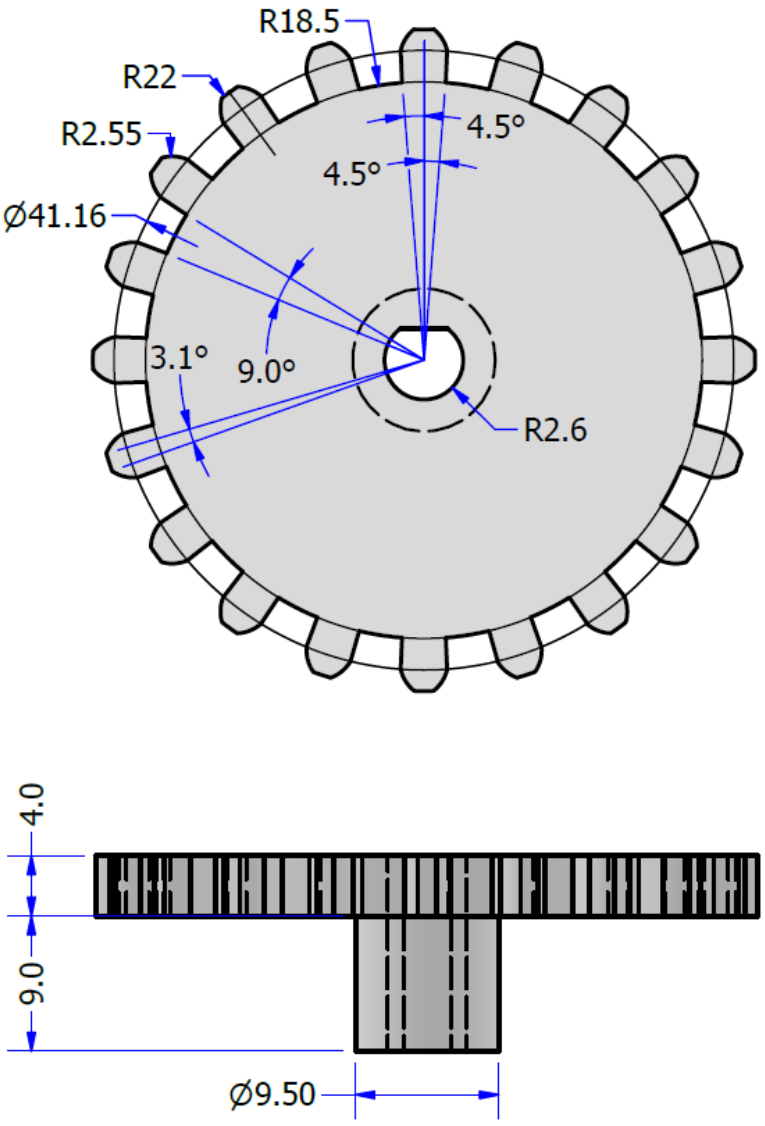
1. KONSTRUKCE ZAŘÍZENÍ



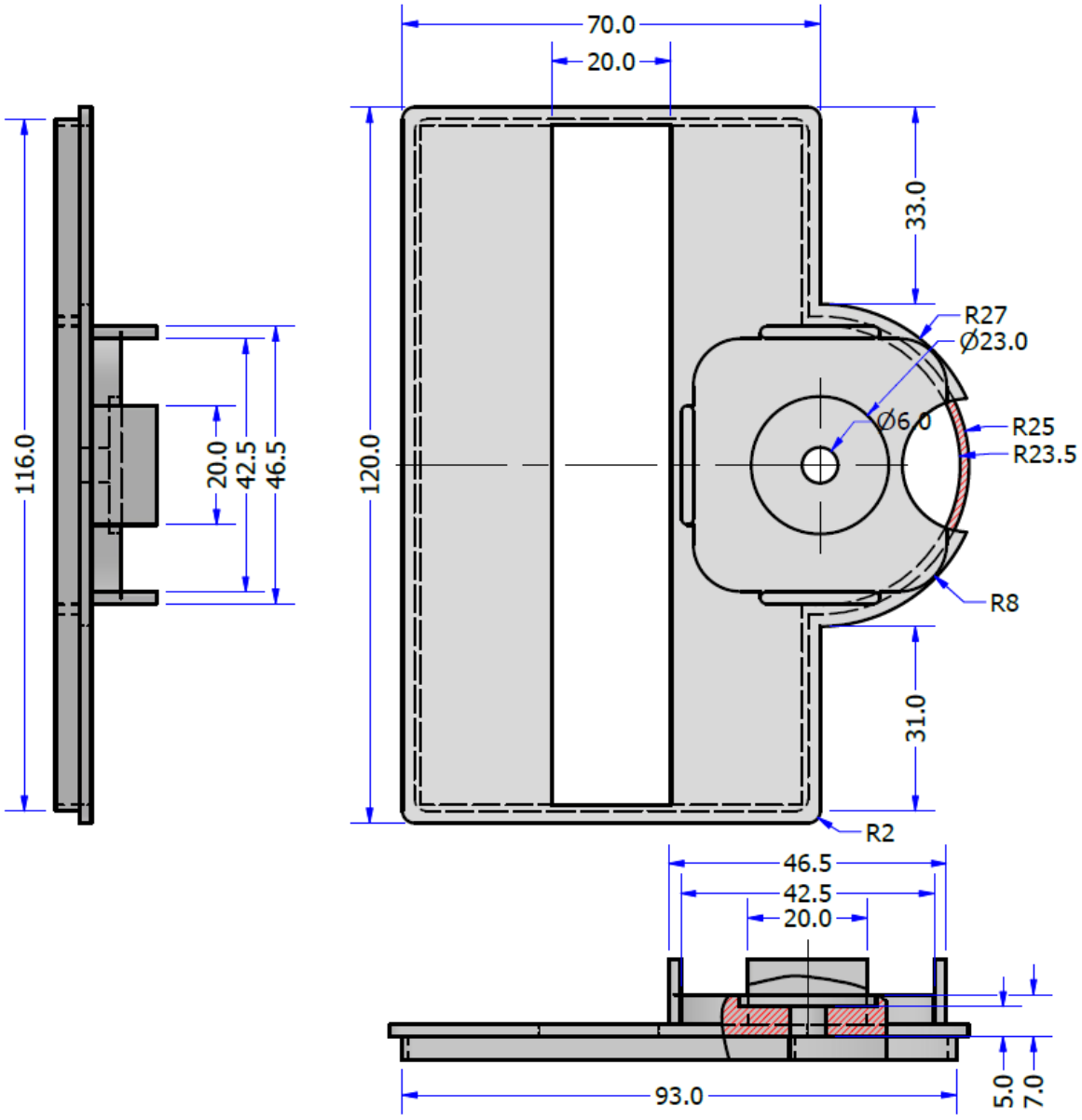
Obrázek 1.1 – Výkresová dokumentace - Základna



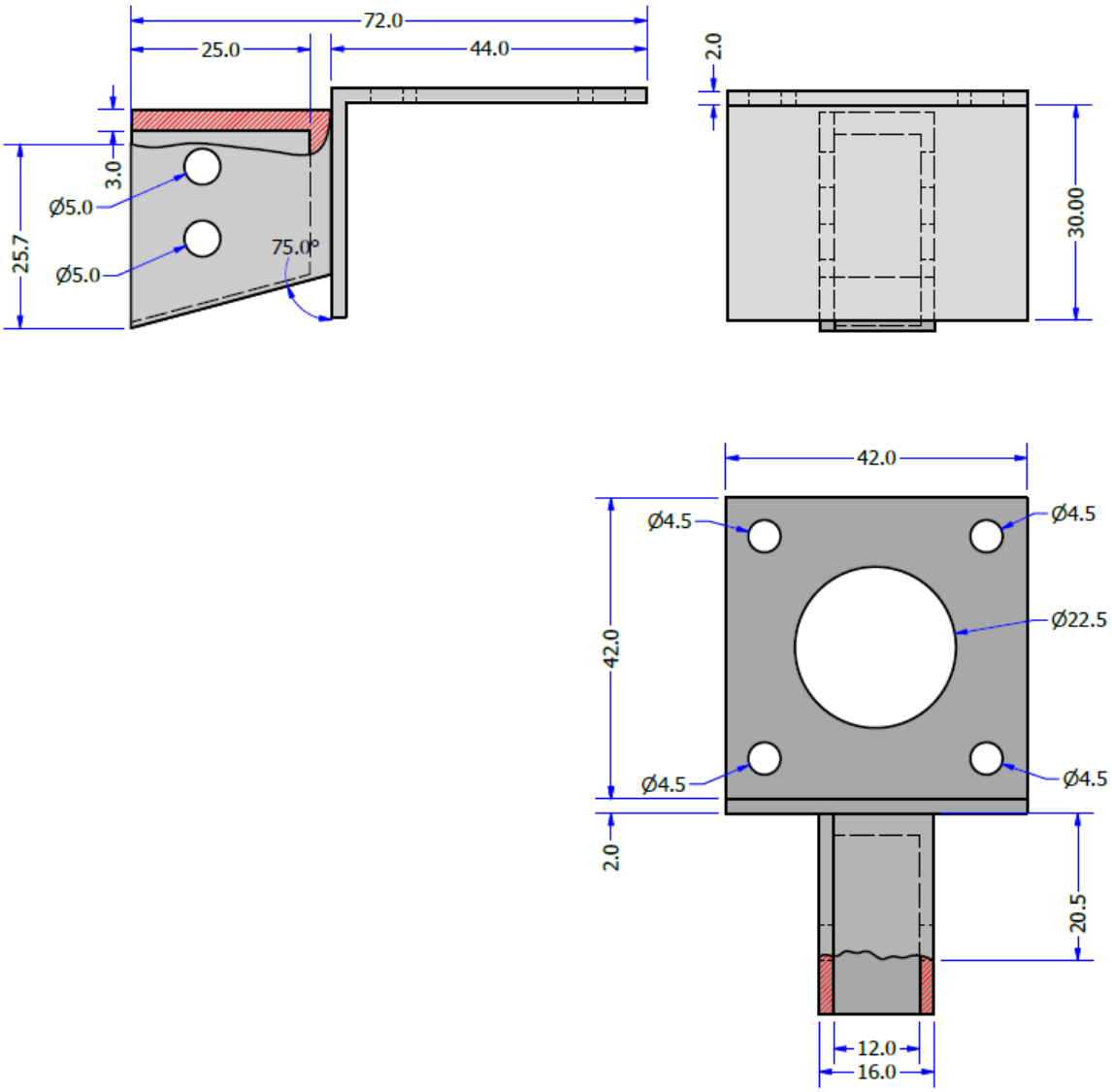
Obrázek 1.2 - Výkresová dokumentace - Posuvník



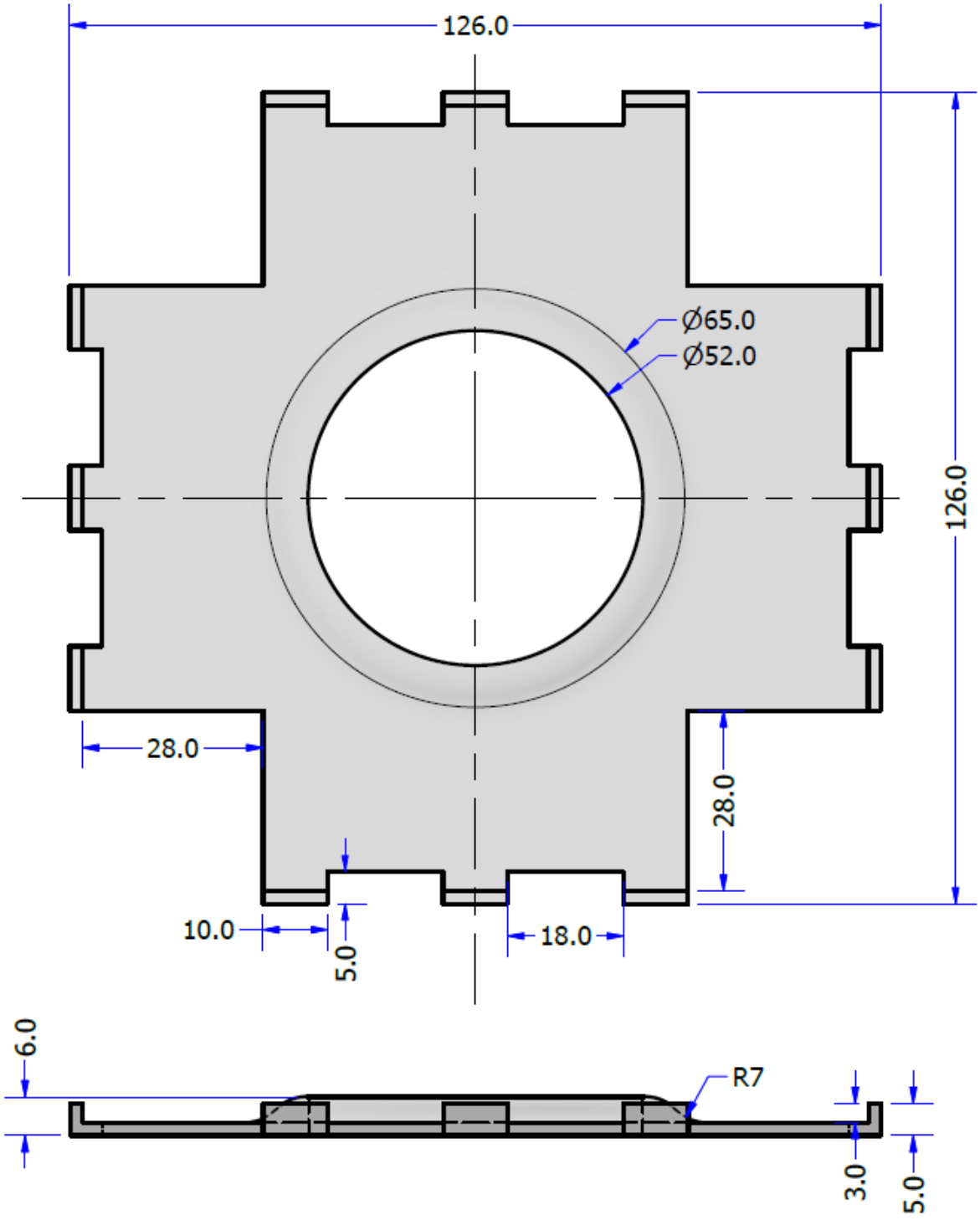
Obrázek 1.3 - Výkresová dokumentace - Ozubené kolo



Obrázek 1.4 - Výkresová dokumentace - Kryt základny



Obrázek 1.5 - Výkresová dokumentace - Držák motoru



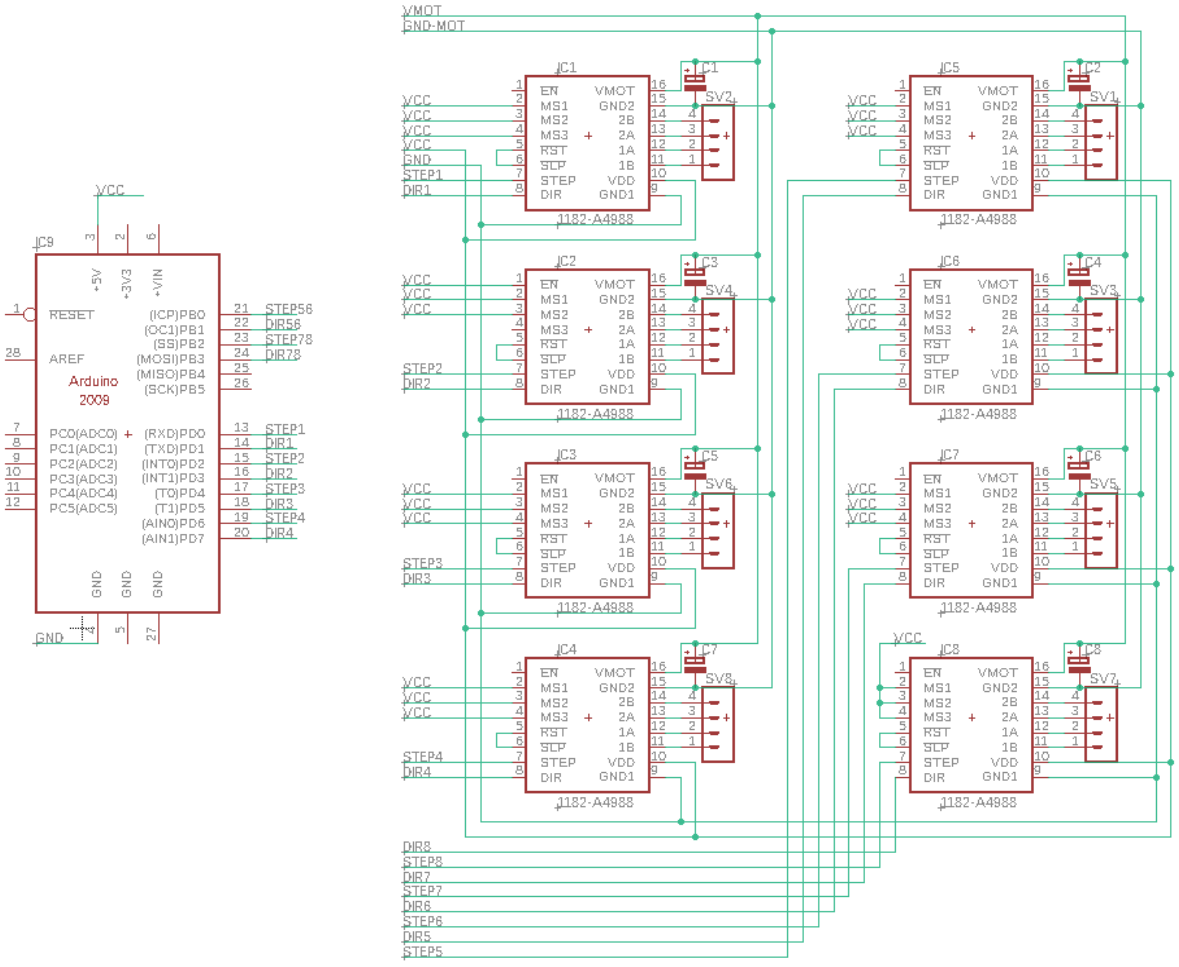
Obrázek 1.6 - Výkresová dokumentace - Středová část

2. KONSTRUKCE ELEKTRONIKY

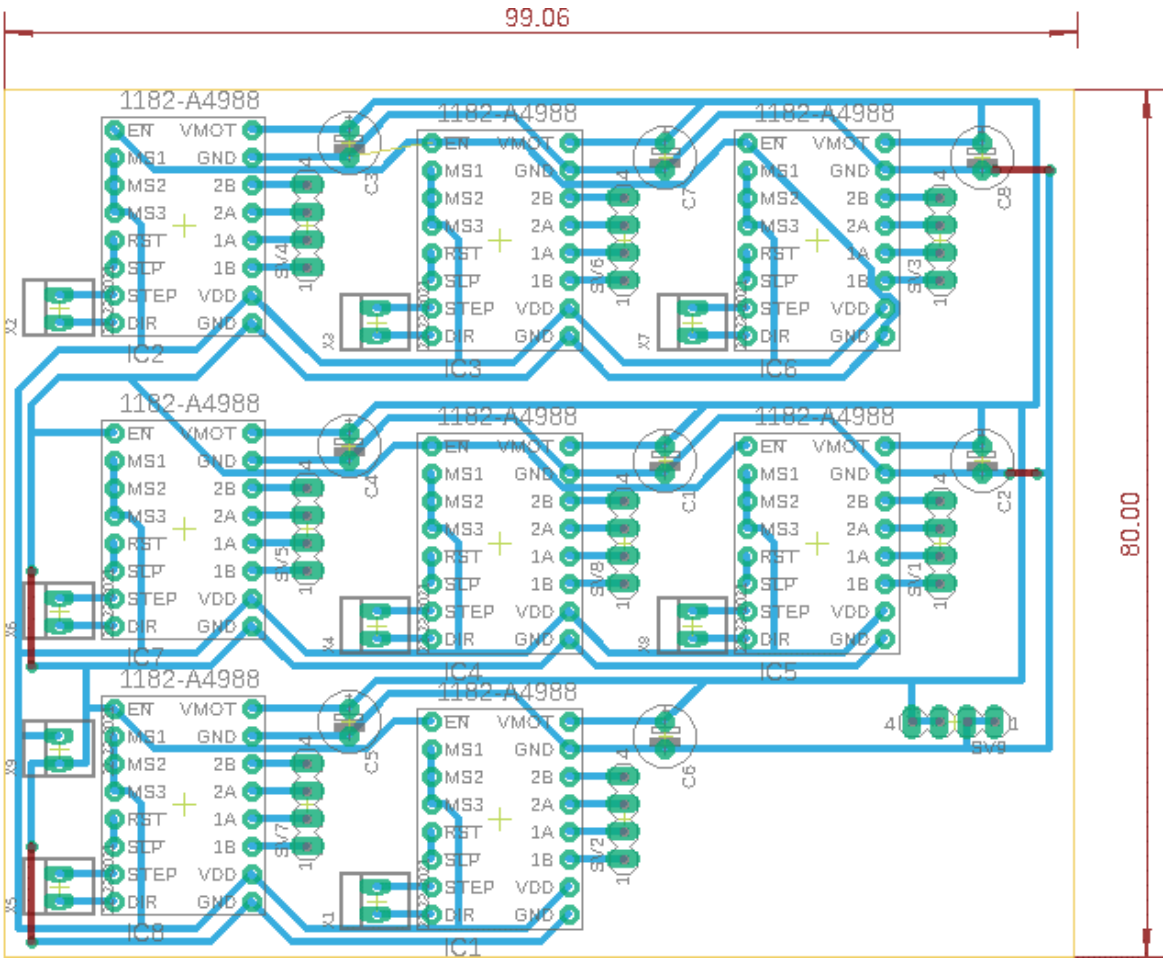
V této kapitole je uveden seznam použitých součástek spolu se schématem zapojení a návrhem desky plošných spojů. Pro detailnější schéma zapojení můžete využít datasheet řadičů či krokových motorů.

Tabulka 1 - Seznam použitých součástí

Typ zařízení	Název zařízení	Počet
Vývojový kit	Arduino UNO	1 ks
Řadič krokového motoru	A4988	8 ks
Krokový motor	NEMA-17	8 ks
Kondenzátor	Elektrolytický 10uF	8 ks
Zdroj napájení	LONGWEI LW-K3010D Laboratorní zdroj	



Obrázek 2.1 - Schéma zapojení robotického zařízení



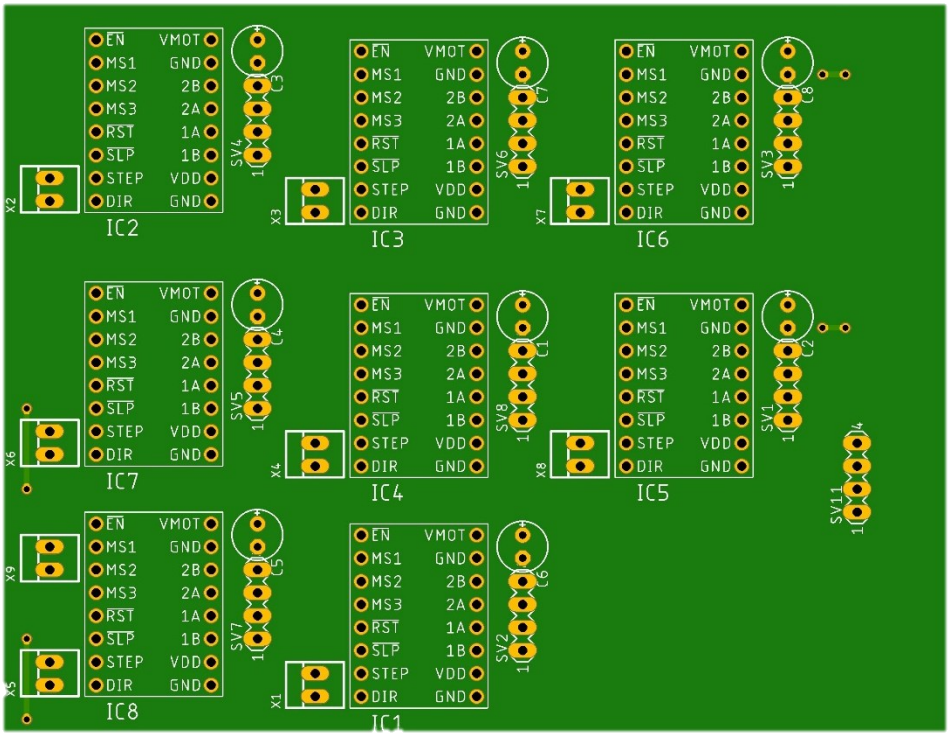
Obrázek 2.2 - Návrh DPS – Top view

Bottom view šířka cest 24 mils*

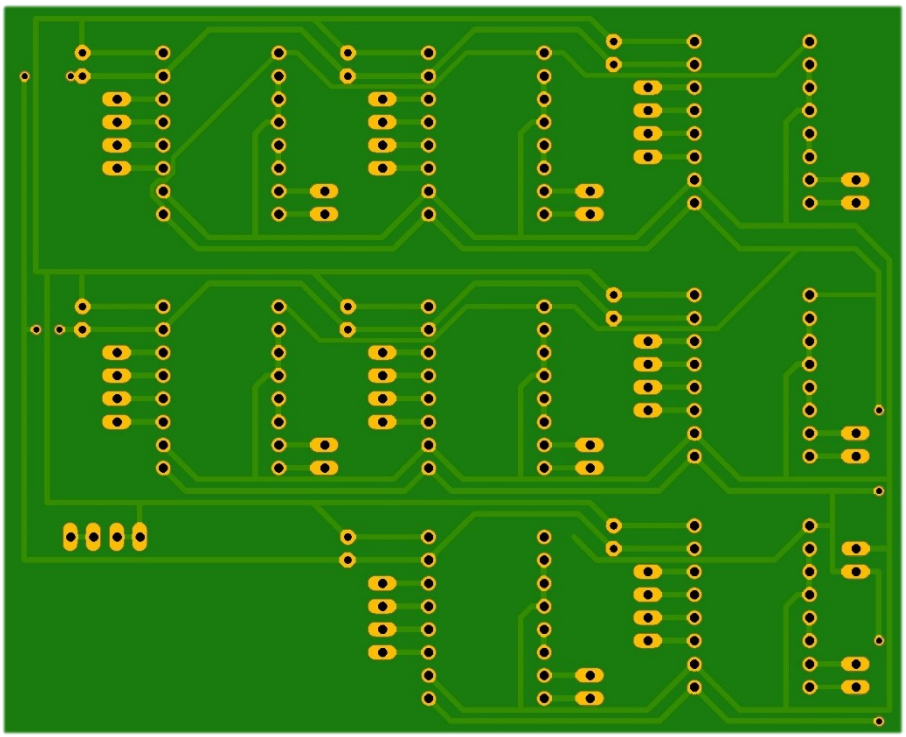
Top view šířka cest 24 mils

Rozměr desky 99x80mm

*mils = 0.001 z palce → 0.0254mm



Obrázek 2.3 - Zobrazení DPS - Top viev



Obrázek 2.4 - Zobrazení DPS - Bottom view