EKONOMICKÁ UNIVERZITA V BRATISLAVE
NÁZOV FAKULTY

Evidenčné číslo: 10300/B/2023/36122176490650116

POROVNÁVANIE ZÁKLADNÝCH ALGORITMOV UMELEJ INTELIGENCIA NA RIEŠENIE HRY TYPU BLUDISKO

Bakalárska práca

2023 **RONALD SEDMÁK**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
NÁZOV FAKULTY

POROVNÁVANIE ZÁKLADNÝCH ALGORITMOV UMELEJ INTELIGENCIA NA RIEŠENIE HRY TYPU BLUDISKO

Bakalárska práca

Študijný program: Hospodárska informatika

Študijný odbor: Hospodárska Informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Rakovská, Eva, RNDr. PhD.

BRATISLAVA 2023 **RONALD SEDMÁK**

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Ronald Sedmák
Študijný program: hospodárska informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: ekonómia a manažment
Typ záverečnej práce: Bakalárska záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický
Názov: Porovnanie základných algoritmov umelej inteligencie na riešenie hry typu bludisko
Anotácia: Autor vyhľadá vhodnú úlohu hry typu bludisko, kde sa hľadá cesta k cieľu. Na nájdenie riešenia pre túto úlohu naprogramuje vybrané základné algoritmy umelej inteligencie a porovná ich efektivitu pre zvolenú úlohu.

Vedúci: RNDr. Eva Rakovská, PhD.
Katedra: KAI FHI - Katedra aplikovanej informatiky
Vedúci katedry: Ing. Mgr. Peter Schmidt, PhD.

Dátum zadania: 31.03.2021

Dátum schválenia: 17.04.2023 doc. Ing. Martin Mišút, CSc.
osoba zodpovedná za realizáciu študijného programu

Čestné vyhlásenie

Čestne vyhlasujem, že som záverečnú diplomovú prácu vypracoval samostatne, a že som uviedol všetku použitú literatúru.

Dátum: 12.05.2023

Poďakovanie

Za nesmiernu trpezlivosť, chcem poďakovať mojej školiteľke RNDr. Eve Rakovskej, PhD**.**

ABSTRAKT

SEDMÁK Ronald: *Porovnávanie základných algoritmov umelej inteligencie v hre maze.* – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; RNDr. Eva Rakovská, PhD. – Bratislava: FHI EU, 2023, počet strán.

Cieľom bakalárskej práce je porovnať algoritmy určené na vyhľadávanie riešenia v bludisku pomocou vlastnej implementácii algoritmov určených na túto úlohu.

V práci si opíšeme základné pojmy umelej inteligencie. Prejdeme si súčasný stav umelej inteligencie vo svete. Vysvetlíme si základné fungovania algoritmov, rôznorodý charakter umelej inteligencie, pojem inteligentný agent, špecifikácie stavového prostredia, agenta pri prehľadávaní, možné dátové štruktúry algoritmu a spôsoby merania výkonnosti algoritmu.

Ďalej si objasníme metodiku práce a výsledok skúmania kde si graficky znázorníme fungovanie vybraných algoritmov, konkrétne, algoritmus A* (A Star), Dijkstrov algoritmus, a algoritmus prehľadávania do šírky (BFS) a hĺbky (DFS), a hlbšie si popíšeme fungovanie dátových štruktúr ktoré tieto algoritmy využívajú.

A na záver, si ukážeme a vysvetlíme zdrojový kód a samotné výsledky porovnávania algoritmov v dvoch dvojrozmerných bludiskách.

Kľúčové slová: umelá inteligencia, algoritmy, breadth-first search, depth-first search, Dijkstra, A* algoritmus, bludisko, porovnanie, Programovací jazyk Python

ABSTRACT

SEDMÁK Ronald: Comparing basic artificial intelligence algorithms in the game maze. - University of Economics in Bratislava, Faculty of Economic Informatics; RNDr. Eva Rakovská, PhD. – Bratislava.

The aim of this bachelor thesis is to compare algorithms designed to search for a solution in the maze using own implementation of algorithms designed for this task.

In the thesis, we will describe the basic concepts of artificial intelligence. We will review the current state of artificial intelligence in the world. We will explain the basic workings of algorithms, the diverse nature of artificial intelligence, the concept of a intelligent agent, state environment specifications, agent in pathfinding, possible data structures of an algorithm, and ways to measure algorithm performance.

Next, we will explain the methodology of the work and the result of the research where we will graphically illustrate the functioning of the selected algorithms, specifically, the A* algorithm (A Star), Dijkstra's algorithm, and Breadth-first search (BFS) and Depth-first search (DFS) algorithms., and we will describe in more depth the functioning of the data structures used by these algorithms.

Finally, we will show and explain the source code and the actual results of comparing the algorithms in two two-dimensional mazes.

Keywords: artificial intelligence, algorithms, breadth-first search, depth-first search, Dijkstra, A* algorithm, maze, comparison, Python programming language

OBSAH

ÚVOD **9**

1. UMELÁ INTELGENCIA **10**

1.1 Sučasný stav AI vo svete **10**

1.1.1 Negatívny dopad AI na svet *11*

1.2 Algoritmy **13**

1.3 Využitie algoritmov **14**

1.4 Charakter AI **14**

1.4.1 Správanie sa ľudsky *15*

1.4.2 Rozmýšľanie ľudsky *15*

1.4.3 Správanie sa racionálne *15*

1.4.4 Rozmýšľanie racionálne *16*

1.5 Inteligentný agent **16**

1.5.1 Racionalita *17*

1.6 Úlohy v prostredí **18**

1.6.1 Špecifikovanie úloh *18*

1.6.2 Vlastnosti úloh *19*

1.6.3 Štruktúra agenta *19*

1.6.4 Fungovanie komponentov programu agenta *20*

1.7 Agenty na riešenie problémov hľadania cieľa **20**

1.7.1 Vyhľadávanie v priestore *21*

1.7.2 Algoritmy vyhľadávania ciest *21*

1.7.3 Dátové štruktúry *22*

1.7.4 Meranie výkonnosti *23*

2. CIEĽ PRÁCE **24**

3. METODIKA PRÁCE A METÓDY SKÚMANIA **26**

3.1 Využité dátové štruktúry **27**

3.1.1 Deque (double-ended queue) *27*

3.1.2 Stack *28*

3.1.3 PriorityQueue *28*

3.2 Breadth-first search 29
3.2.1 Grafické znázornenie 30
3.3 Depth-first search - recursive 32
3.3.1 Grafické znázornenie 32
3.4 Dijkstrov algoritmus 34
3.5 A* algoritmus 35
3.5.1 Prípustná heuristika 35
3.5.2 Grafické znázornenie A 37*
4. VÝSLEDKY PRÁCE 40
4.1 Pochopenie úlohy 40
4.2 Programovací jazyk Python 40
4.3 Zostrojenie programu 41
4.3.1 Priestor a cieľ - Bludisko 41
4.3.2 Akcie – Pohyb a hľadanie susedov 42
4.3.3 Meradlo výkonnosti – počet krokov, najkratšia cesta a čas 44
4.4 Pseudokód a implementácia algoritmov 46
4.4.1 Pseudokód BFS 46
4.4.2 Algoritmus BFS 47
4.4.3 Pseudokód DFS 48
4.4.4 Implementácia DFS 49
4.4.5 Pseudokód Dijkstrovho algoritmu 50
4.4.6 Implementácia pre Dijkstrov algoritmus 51
4.4.7 Pseudokód algoritmu A 52*
4.4.8 Implementácia algoritmu A 53*
4.5 Porovnávanie algoritmov 55
4.5.1 Výstup a jeho porovnanie. 55
ZÁVER 57
ZOZNAM POUŽITEJ LITERATÚRY 59

ÚVOD

Výber témy tejto práce bol výrazne ovplyvnený mojou zainteresovanosťou v sfére umelej inteligencie. Verím, že v dnešnej dobe daná tematika vzbudzuje u ľudí rastúci záujem. Umelá inteligencia je značne prepojená s fungovaním mnohých každodenných systémov. Umelá inteligencia je podstatná technológia, ktorá zlepšuje kvalitu práce ale aj života. Uľahčuje dianie vo viacerých smeroch, tak ako vedeckých, ekonomických a vojenských, tak jej uplatnenie nájdeme aj vo svete umenia.

Touto prácou by som chcel priblížiť svet algoritmov a umelej inteligencie pre všedných ľudí. Mojím cieľom je ukázať, že algoritmy, ktoré sa môžu javiť ako zložité pre všedného človeka, je možné aplikovať na riešenie triviálneho problému hľadania cieľa v bludisku.

Algoritmy, s ktorými budeme pracovať, sú vyvinuté na to, aby našli cestu z bodu „A“ do bodu „B“. Samotné algoritmy, ak sú správne implementované, dokážu v bludisku nájsť cestu, a tú najkratšiu cestu, v prípade ak je toho algoritmus schopný.

V tejto práci si graficky a taktiež aj pomocou kódu objasníme vybrané algoritmy prehľadávania a prejdeme si aké „zmýšľanie“ tieto algoritmy využívajú pri svojich činnostiach.

Nakoniec ich navzájom porovnáme a vyhodnotíme, ktorý algoritmus vykonal svoju prácu najefektívnejšie.

1. UMELÁ INTELGENCIA

Výskumníci sledovali niekoľko rôznych verzií definície AI. Niektorí definovali inteligenciu z hľadiska ľudského výkonu, zatiaľ čo ostatní definovali inteligenciu ako racionalitu, teda robenie tej správnej veci. Samotný subjekt inteligencie sa tiež líši. Môžeme brať do úvahy vlastnosť vnútorných myšlienkových procesov a uvažovania, teda vnútorná charakteristika, alebo inteligentného správania, teda vonkajšia charakteristika.

Ďalej môžeme rozdeliť umelú inteligenciu podľa vedných disciplín, v ktorých sa vyskytuje, ako napríklad vo filozofii, ekonómii , matematike a informatike. V každej tejto disciplíne bude s umelou inteligenciou spojený rôzny spôsob jej využitia.

Historicky je pre umelú inteligenciu najvplyvnejšia vízia Alana Turinga. V 50. rokoch predstavil Turingov test, strojové učenie, genetické algoritmy a reflexívne učenie. Taktiež navrhoval, že na vytvorenie umelej inteligencie na úrovni ľudského myslenia by bolo jednoduchšie vytvoriť algoritmy učenia a učiť tak stroj, namiesto programovania inteligencie manuálne. Ďalej varoval, že takýto postup by nemusel mať najlepší výsledok pre ľudskú rasu.
(Russel, 2021)

1.1 Sučasný stav AI vo svete

V dnešnej dobe ak človek počuje výraz umelá inteligencia, môže si pomyslieť o samojazdiacich autách a robotoch, ktorý napodobňujú chôdzu psa alebo aj človeka, ďalej aj chatovacích robotov a tzv. AI ilustrácie. Naj-lepším príkladom AI pre všedného človeka sa stávajú hlasoví asistenti v telefónnych zariadeniach alebo vo vyhľadávačoch ako Bing. (Diaz, 2023)

Medzi pozoruhodné pokroky v dnešnej dobre pre AI patrí určite vývoj a vydanie ChatGPT 3.5 a GPT 4. Ale pokrok AI sféry je momentálne tak rýchly, že takýchto podobných pokrokov existuje obrovské kvantum. (Diaz, 2023)

ChatGPT je AI chatovací robot, ktorý je schopný pracovať s prirodzeným jazykom, s prekladaním a s odpovedaním na rôzne otázky. ChatGPT sa zdá byť ako ten najznámejší AI nástroj, kvôli svojej rozšírenej dostupnosti. GPT je jazykový model, ktorý tento chatovací robot využíva. Pri vydaní GPT-3 v roku 2020, predstavoval najväčší jazykový model, s 175 miliárd parametrov. Terajší model GPT-4 má 1 bilión parametrov. (Diaz, 2023)

Obrázok č. 1 Ilustrácia vytvorená pomocou Midjourney
Zdroj: vlastné spracovanie

1.1.1 Negatívny dopad AI na svet

V dnešnej dobe, kvôli pokrokom AI, sa varovanie Alana Turinga stáva čo raz viac významnou súčasťou dnešných diskusiách o hrozbách ktoré AI predstavuje pre ľudský život. Hrozby ktoré AI predstavuje môžu mať dopad na človeka napríklad zo stránky právnej, politickej, pracovnej alebo ekonomickej.

Spoliehanie sa na AI služby a produkty má svoje potencionálne riziká.

Nadmerné a nedostatočné využitie

Nedostatočné využitie môže predstavovať nevyužité ekonomické príležitosti, ktoré by inak priniesli lepšie presadenie na medzinárodnom trhu, ekonomickú prosperitu a významnejšie príležitosti pre ľudí.

Nadmerné využitie predstavuje využitie AI pri riešení úloh pre ktoré nie je vhodná, alebo investícia do AI služieb alebo aplikácií, ktoré nemajú žiadnu aplikačnú hodnotu. (Europe, 2020)

Zodpovednosť za škodu

Dôležitou výzvou je určiť či výrobca vozidiel, programátor alebo koncoví užívateľ je zodpovedný za škody spôsobené v nehode v súvislosti so samojazdiacim vozidlom. Ak je produkt odvolený z akejkoľvek zodpovednosti, môže to viesť k nižšej motivácii o zlepšenie produktu alebo služby a mohlo by to znížiť vieru ľudí v technológiu. Nadmerná regulácia AI výrobkov môže taktiež obmedziť inovácie. (Europe, 2020)

Dopad na zamestnanosť

Využitie AI v pracovnom prostredí môže viesť k eliminovaniu veľkého množstva pracovných pozícií. Očakáva sa, že AI vytvorí nové pracovné možnosti, ale je potrebné zabezpečiť dostatočnú kvalifikovanosť pracovnej sily, aby sa zabránilo dlhodobej nezamestnanosti. (Europe, 2020)

Realita, osobné práva a demokracia

Dizajn a dáta AI produktu majú veľký vplyv na výsledky získané s jej pomocou, tieto výsledky môžu byť kvôli dizajnu či dátam úmyselne alebo neúmyselne neobjektívne. Reprezentácia reality pomocou AI tak môže byť mylná, čo potom vedie k chybným rozhodnutiam.

Osobné práva môže narušovať kombinovaním dát o jednotlivcovi, čo môže vytvoriť nepravdivé predpoklady o správaní či záujmoch, alebo pomocou týchto dát umožní ovplyvňovať ekonomické a politické správanie. Technológia rozpoznávania tváre môže byť využitá pre sledovanie a profilovanie ľudí čo narušuje právo na súkromie.

AI algoritmy sa snažia odporúčať obsah užívateľom taký, aký sa im najviac páči. Môže to viesť k vytvoreniu online komory ozvien kde neexistujú pluralistické názory. AI taktiež dokáže vytvoriť podobizeň reálneho človeka, čo môže viesť k šíreniu falošných informácií. Tieto možnosti AI dokážu vytvoriť separáciu a polarizáciu vo verejnosti, čo môže viesť k politickej nestabilite. (Europe, 2020)

Možnosti hackerských útokov

Hackeri môžu využiť AI pre lepšie optimalizované útoky. Napríklad implementovanie jazykových modelov za účelom phishing útokov alebo objavenie nových dier v obranných systémoch. Majú taktiež možnosť vytvorenia škodlivého kódu lepšie a rýchlejšie.

Fyzická bezpečnosť ľudí je taktiež ohrozená ak sú napríklad využité nedostatky v bezpečnostnom systéme samojazdiacich áut alebo iných zariadeniach.

Získanie prístupu k dátam ktoré AI využíva na svoje fungovanie môže ohroziť osobné údaje mnohých ľudí. Tieto dáta môžu byť taktiež úmyselne upravené, aby AI produkovalo nesprávne výsledky.

S využitím AI je možné vyprodukovať falošné realistické videá či fotografie, napríklad o únose blízkych, čo pridáva väčšiu presvedčivosť pri cielených ransomware útokoch. Falošné videá alebo fotografie môžu byť využité aj pre poškodenie dobrého mena spoločnosti alebo jednotlivca. (Malwarebytes, 2023)

1.2 Algoritmy

Neformálne, ***algoritmus*** je hocijaká dobre definovaná výpočtová procedúra, ktorá berie určité hodnoty, alebo set hodnôt, ako ***vstup*** a vyprodukuje nejakú hodnotu, alebo set hodnôt, ako ***výstup***. Algoritmus je teda sekvencia výpočtových krokov, ktoré premenia vstup na výstup. (Cormen, 2009)

Vieme taktiež vnímať algoritmus ako nástroj pre riešenie dobre špecifikovaných ***výpočtových problémov***. Príkaz problému vo všeobecnosti špecifikuje chcenú vstupovú/výstupovú súvislosť. Na dosiahnutie tejto súvislosti algoritmus opíše špecifickú výpočtovú procedúru. (Cormen, 2009)

Napríklad, budeme potrebovať zoradiť sekvenciu čísiel od najmenšieho po najväčšie. (deleted)Takto si formálne navrhneme ***triediaci problém***:

Vstup: Sekvencia *n* čísiel $\langle a_1, a_2, \ldots, a_n \rangle$.

Výstup: Permutácia (preskupenie) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ vstupnej sekvencie aby $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Napríklad, daná je vstupná sekvencia $\langle 31,41,59,26,41,58 \rangle$, triediaci algoritmus vráti ako výstup sekvenciu $\langle 26,31,41,41,58,59 \rangle$. Takáto vstupná sekvencia je nazývaná ako ***inštancia*** triediaceho algoritmu. Vo všeobecnosti, ***inštancia problému*** je zložená zo vstupu potrebného na výpočet riešenia pre daný problém. (Cormen, 2009)

Pretože mnoho programov využíva triedenie ako medzistupeň, stáva sa fundamentálnou funkciou v informatike. Ako výsledok, existuje k dispozícii obrovské množstvo dobrých triediacich algoritmov. Ktorý algoritmus je najlepší pre danú aplikáciu závisí od určitých faktorov, ako napríklad, množstvo predmetov, ktoré sa majú vytriediť, ako veľmi už sú predmety potriedené, možné reštrikcie na hodnoty predmetov, architektúra

počítača, a druh úložného miesta, ktorý môže byť využitý: hlavná pamäť, disky, alebo aj kazety. (Cormen, 2009)

O algoritme sa dá povedať, že je ***správny*** ak, pre každú vstupnú inštanciu, zastaví svoj proces pri správnom výstupe. Vieme povedať, že správny algoritmus ***vyrieši*** zadaný výpočtový problém. Nesprávny algoritmus sa nemusí zastaviť vôbec pri niektorých vstupných inštanciách, alebo sa zastaví s nesprávnom riešením. V rozpore s tým čo sa od algoritmu očakáva, môže aj nesprávny výsledok byť užitočný, ak vieme kontrolovať chybnosť algoritmu. (Cormen, 2009)

Algoritmus môže byť špecifikovaný, ako počítačový program, alebo aj ako hardvérový dizajn. Jediná požiadavka je, aby špecifikácie poskytovali precízny opis výpočtovej procedúry, ktorá je sledovaná. (Cormen, 2009)

1.3 Využitie algoritmov

Triedenie v žiadnom prípade nie je jediný výpočtový problém, pre ktorý boli algoritmy zostrojené. Praktická aplikácia algoritmov je všadeprítomná. Príkladmi môže byť:

- Vytvorenie sofistikovaných algoritmov pri práci s identifikovaním ľudského genómu, kde práca s dátami si vyžaduje ukladanie informácií do bázy dát, a vytvorenie nástrojov na dátovú analýzu. (Cormen, 2009)

- Manažment obrovského množstva dát na internete, algoritmy umožňujú nájsť efektívnu cestu pre tok informácií. (Cormen, 2009)

- Kryptografia na vytvorenie verejných kľúčov a digitálnych podpisov, ktorá závisí od číselných algoritmov a od teórie čísiel. (Cormen, 2009)

1.4 Charakter AI

Oblasť umelej inteligencie, sa zaoberá nielen s chápaním ale aj budovaním inteligentných entít, teda strojov, ktoré dokážu vypočítať ako efektívne a bezpečne postupovať v rôznych prostrediach. (Russel, 2021)

AI v dnešnej dobe zahŕňa obrovskú škálu podoblastí, od tých všeobecnejších, ako napríklad, učenie, uvažovanie a vnímanie, až po špecifické, ako hranie šachu, písanie

matematických teorém, písanie básní či tvorenie iných literárnych diel, alebo ovládaní motorového vozidla. AI je možné využiť v ktorejkoľvek oblasti; je to naozaj univerzálna oblasť.

Zo sledovania týchto charakteristík vyplývajú štyri prístupy k AI. (Russel, 2021)

1.4.1 Správanie sa ľudsky

Alan Turing v roku 1950 navrhol Turingov test. Bol navrhnutý ako myšlienkový pokus, ktorý by obišiel nejasnosť filozofickej otázky: *či dokáže stroj rozmýšľať*. Stroj skúšku prejde, ak ľudský vyšetrovateľ nedokáže rozoznať odpoveď stroja od odpovede človeka. (Russel, 2021)

Tomuto testu výskumníci nevenovali mnoho úsilia, keď že verili, že je dôležitejšie venovať sa základným princípom inteligencie. Hľadanie "umelého letu" sa podarilo, keď inžinieri a vynálezcovia prestali napodobňovať vtáky a začali používať veterné tunely a učiť sa o aerodynamike. (Russel, 2021)

1.4.2 Rozmýšľanie ľudsky

Aby sme mohli povedať, že stroj rozmýšľa ako človek, musíme najprv vedieť ako človek rozmýšľa. Ak sme dosiahli dostatočne presnú teóriu o fungovaní ľudskej mysle, dokážeme túto teóriu aplikovať pri fungovaní stroja. Ak sa správanie stroja zhoduje so správaním človeka, získame tak dôkaz, že niektoré mechanizmy stroja sa môžu nachádzať aj v človeku. (Russel, 2021)

1.4.3 Správanie sa racionálne

Na to, aby stroj dokázal využiť logiku, potrebuje určité vedomosti o svete. S využitím teórie pravdepodobnosti dokáže stroj fungovať aj v neistote.

V princípe, tieto informácie dokážu vybudovať komplexný model racionálneho myslenia, s ktorým vieme z hrubých vnemových poznatkov získať pochopenie o fungovaní sveta, a aj predpovede o budúcnosti. To, čo však stroj nerobí, je generovanie inteligentného správania. Preto potrebujeme teóriu racionálneho správania. Racionálne myslenie, samo o sebe, je nedostačujúce. (Russel, 2021)

1.4.4 Rozmýšľanie racionálne

Pre racionálny program, alebo teda racionálneho ***agenta***, je cieľom dosiahnuť ten najlepší výsledok, a pri neistote, najlepší očakávaný výsledok. Dosiahnutie tohto výsledku je z časti kvôli správnemu odvodzovaniu, čo predstavuje racionálne premýšľanie.

Problém nastáva, ak sa takýto agent snaží pohybovať v komplexnom prostredí, kde výber pre optimálnu cestu je veľmi výpočtovo náročný. (Russel, 2021)

1.5 Inteligentný agent

Agent je čokoľvek, čo vníma svoje prostredie a koná na základe svojho vnímania v tomto prostredí. Ľudský agent má oči, uši a iné orgány pre senzory a ruky, nohy, hlasivky a tak ďalej pre aktuátory. Robotický agent môže mať kamery a infračervené diaľkomery ako senzory a rôzne motory ako aktuátory. Softvérový agent prijíma obsah súborov, sieťové pakety a ľudské vstupy (klávesnica/myš/dotyková obrazovka/hlas) ako senzorické vstupy a pôsobí na prostredie zápisom súborov, odosielaním sieťových paketov a zobrazovaním informácie alebo generovanie zvukov. „Prostredie môže byť všetko – celý vesmír!“ V praxi je to len tá časť vesmíru, ktorej stav nás zaujíma, tá časť, ktorá ovplyvňuje to, čo agent vníma, a ktorú agent dokáže ovplyvniť. (Russel, 2021)

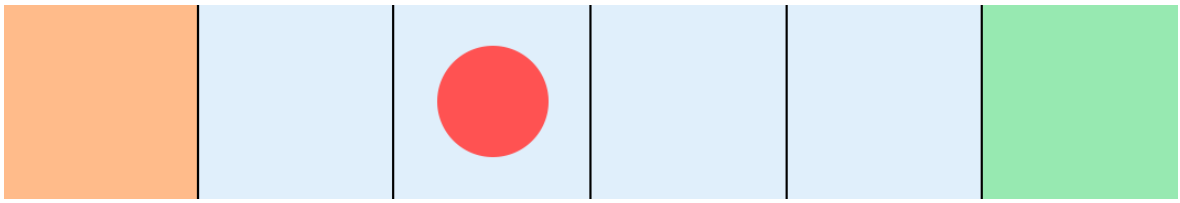
Agent koná jedine podľa toho čo dokáže vnímať a čo doposiaľ vníma. Tieto vedomosti, a vedomosti, ktoré sú v ňom zabudované, využije pre sledovanie a interakciu s rôznymi stavmi prostredia, v ktorom sa nachádza.

Implementácia agenta predstavuje program, zatiaľ čo funkcia agenta predstavuje jednotlivé možnosti, ktoré nariaďujú ako môže s prostredím interagovať. Pre našich agentov bude predstavovať ich prostredie bludisko. (Russel, 2021)

Majme jednoduché pole veľkosti 1x6, pre agenta predstavuje priestor, v ktorom môže konať. Cieľom agenta je dostať sa na koniec poľa. Agent vie, kde je štart a koniec. Agent vie, kde sa nachádza. Štart predstavuje oranžové políčko, koniec, zelené políčko. Agent sa môže pohybovať len doprava. Jeho funkciou predstavuje: ak nie som v cieli, pohnem sa o jedno políčko. Samotné prostredie musí byť reprezentované dátovou štruktúrou na to, aby algoritmus fungoval, ako napríklad pole, množina, zoznam, strom, zásobník a halda. (Russel, 2021)

Obrázok č. 2 Začiatočný stav 0
Zdroj: vlastné spracovanie

Každý pohyb agenta bude predstavovať samostatný stav prostredia.



Obrázok č. 3 Stav 2
Zdroj: vlastné spracovanie

Obrázok č. 4 Konečný stav
Zdroj: vlastné spracovanie

Dosiahnutie cieľa predstavuje pre agenta finálny stav, keďže sa nachádza v cieli, agent svoj proces terminuje. (Russel, 2021)

1.5.1 Racionalita

Čo je pre agenta racionálne závisí od štyroch vecí:

Meradlo výkonnosti, ktoré určuje kritérium úspechu,

Znalosti, ktoré agent má o prostredí,

Akcie, ktoré agent môže vykonať,

Vnímanie doterajšej sekvencie postupnosti agenta. (Russel, 2021)

Podľa predošlého príkladu, kde sa agent pohybuje v poli veľkosti 1x6, si ustanovíme či je agent racionálny.

- Pre **meradlo výkonnosti** je pre nás podstatné, či sa agent dostane do cieľa
- Pri **znalostí agenta**, agent vie aká je veľkosť poľa, kde začína a kde má skončiť

- Jeho **akcie** predstavujú len pohyb doprava
- Pri jeho **sekvenčnej postupnosti**, agent vie na akom políčku sa nachádza, podľa čoho vie určiť, či je v cieli

Podľa týchto okolností vieme určiť, že agent je naozaj racionálny. Podľa iných okolností napríklad, ak by bol cieľ nedostupný, agent by sa nevedel nikam pohnúť. Vedeli by sme povedať, že agent je neracionálny. V tomto prípade by lepším agentom, bol agent, ktorý vie, na akých políčkach cieľ nie je. Podľa čoho vie usúdiť, že do cieľa sa nedokáže dostať. (Russel, 2021)

1.6 Úlohy v prostredí

Úlohy v prostredí predstavujú problémy, pre ktoré sa agent snaží nájsť riešenie. Podľa toho ako sú úlohy vybudované, je potrebné spraviť program agenta, ktorý tieto úlohy bude schopný splniť. (Russel, 2021)

1.6.1 Špecifikovanie úloh

Pre vytvorenie racionálneho agenta sme potrebovali špecifikovať meradlo výkonnosti, jeho znalosti o prostredí, akcie, ktoré vie vykonať, a jeho vnímanie prostredia. Pre tieto špecifikácie sa využíva skratka PEAS (Performance, Environment, Actuators, Sensors). Špecifikovanie týchto úloh je prvý krok pri vytvorení agenta.
Berme ako príklad cestu do obchodu.

Ako meradlo výkonnosti môžeme mať, ako dlho nám trvá cesta, zastavíme sa, lebo sme unavení, koľko krokov nám bude trvať cesta do obchodu, a podobne.

Pri prostredí to je, koľko prechodov musíme prejsť, počet áut na ceste, a rôzne ďalšie faktory, s ktorými sa agent v prostredí vie stretnúť.

Pri akciách chápeme možnosti pohybu v priestore, napríklad ak sa na chodníku nachádza diera, môže ju preskočiť alebo obísť, ak zapadne do diery, tak z nej vylezie.

A pri jeho vnímaní si agent môže pri ceste všímať okolo idúce autá, počúvať z akých smerov prichádzajú, mať na mobile GPS, aby sa nestratil, a iné prostriedky, ktoré mu pomôžu. (Russel, 2021)

1.6.2 Vlastnosti úloh

Vlastnosti úloh nám ďalej určujú ako má byť agent vytvorený a ako agenta implementovať. Tieto vlastnosti rozdeľujeme na:

- Plne pozorovateľné a čiastočne pozorovateľné (Fully observable vs. Partially observable)

  Predstavujú dostupnosť k celkovému stavu prostredia. Šachová doska a pohyby súpera sú plne pozorovateľné. Pri ovládaní vozidla agent nevie, čo sa nachádza za ďalším rohom, takže je priestor len čiastočne pozorovateľný.
- S jedným agentom a s viac agentmi (Single-agent vs. Multiagent)

  Predstavujú či pre daný problém je potrebné implementovať jedného alebo viacerých agentov.
- Deterministické a nedeterministické (Deterministic vs. Nondeterministic)

  Deterministický charakter predstavuje závislosť ďalšieho stavu na momentálnom stave prostredia a akcií agenta. Nedeterministická charakteristika má náhodný charakter.
- Epizodické a sekvenčné (Episodic vs. Sequential)

  Určuje, či pre postup agenta sú potrebné predošlé znalosti o jeho akciách a či podľa nich má konať ďalej.
- Statické a dynamické (Static vs. Dynamic)

  Či priestor, v ktorom sa agent nachádza, sa neustále mení.
- Diskrétne a spojité (Discrete vs. Continuous)

  Ak počet celkových stavov v priestore je konečný, jedná sa o diskrétnu charakteristiku. Ak sa stavy v priestore nedajú numericky určiť, jedná sa o spojitú charakteristiku.
- Známe a neznáme (Known vs. Unknown)

  Závisí od agenta, nie od prostredia. V známom prostredí, agent vie posúdiť aký výsledok jeho akcie majú. V neznámom prostredí sa agent musí naučiť aký výsledok budú jeho akcie v prostredí mať. (Russel, 2021)

1.6.3 Štruktúra agenta

Cieľom AI je vytvoriť program agenta, ktorý implementuje funkcie agenta. Predpokladajme, že tento program beží na fyzickom stroji, pomocou ktorého agent dokáže

vnímať a konať. Takýto stroj nazývame ako architektúra agenta. Agent v celku teda predstavuje architektúru a samotný program. (Russel, 2021)

1.6.4 Fungovanie komponentov programu agenta

Jednotlivé komponenty v programe agenta predstavujú zmeny stavu priestoru, v ktorom sa agent nachádza.

Komponenty môžu mať rôzne reprezentácie. Rozdeľujú sa nasledovne:

Atomické,

Faktorizované,

Šturktúrované.

Pri atomickej reprezentácii každý stav vo svete je nedeliteľný, čo znamená že nemá žiadnu vnútornú štruktúru. Ak máme daný cieľ, kde sa máme dostať z mesta A do mesta B, vyhovovalo by zmenšiť škálu sveta len na jednotlivé mená miest. Ďalej by sa porovnávala aktuálna lokácia s naším cieľom. Mesto, samo o sebe v tejto reprezentácii neobsahuje žiadnu štruktúru, podľa ktorej by agent konal.

Na rozdiel od atomickej reprezentácie, faktorizovaná reprezentácia si rozdeľuje každý stav na určité množstvo množín premenných a atribútov, pri čom každý môže mať nejakú hodnotu. V prípade cestovania by agent musel sledovať ďalšie faktory, ktoré ovplyvňujú jeho pohyb. Ako príklady, agent môže sledovať momentálne množstvo benzínu vo vozidle, či fungujú smerovky vozidla, a či máme dosť peňazí na tankovanie.

Pri štruktúrovanej reprezentácii by bolo potrebné, aby agent bral znalosti z bázy dát kde sú vysvetlené všetky vzťahy faktorov, ktoré agent potrebuje na zmenu stavu v prostredí. (Russel, 2021)

1.7 Agenty na riešenie problémov hľadania cieľa

Väčšina agentov pri problémoch vyhľadávania využíva atomickú reprezentáciu, takže vnímajú stavy v priestore ako celky.

Pri práci s prostredím si agent bude prechádzať štvorfázovým procesom na to, aby sa dostal do cieľa.

- Formulácia cieľa.

  Agent si musí zadať svoj cieľ, podľa čo bude konať, aby cieľ dosiahol.

- Formulácia problému.

Agent si vytvorí opis stavov a akcií, ktoré sú potrebné na dosiahnutie cieľa. Pre nášho agenta je podstatné vedieť, že kvôli jeho akcii posunu na susedné políčko, sa zmení stav sveta, čo znamená že bude konať podľa svojej momentálnej pozície v priestore.

- Vyhľadávanie.

  Agent si sekvenčne simuluje postup akcií, ktorý ho dostane do cieľa. Tento postup sa nazýva riešením. Po určitom množstve akcií sa dostane do cieľa, ak to je v priestore možné.

- Vykonanie.

  Agent teraz môže vykonať akcie podľa simulovaného riešenia, krok po kroku. (Russel, 2021)

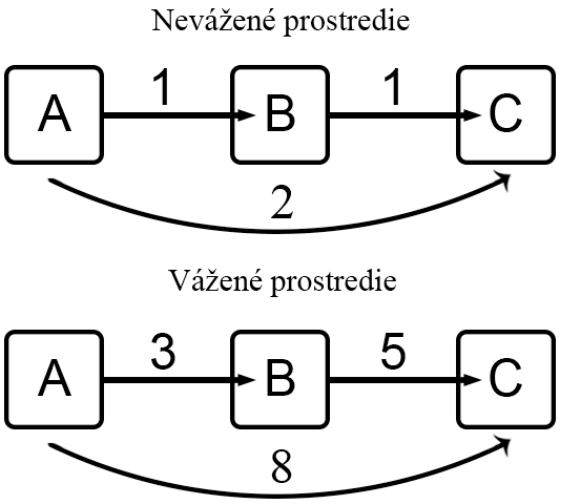
1.7.1 Vyhľadávanie v priestore

Na to, aby agent mohol v priestore vyhľadať cieľ, musí mať problém vyhľadávania definované množstvo možných stavov prostredia, tzv. stavové prostredie, začiatočný stav prostredia, cieľový stav prostredia, akcie, ktoré sú pre agenta možné, čo je výsledkom jeho akcie, a náklady vykonanie akcie. Sekvencia akcií vytvorí **cestu**, a riešením bude cesta zo začiatočného stavu až po konečný stav. (Russel, 2021)

1.7.2 Algoritmy vyhľadávania ciest

Breadth-first search, Depth-first search, Dijkstrov algoritmus a A * algoritmus.

Tieto základné algoritmy umelej inteligencie sú určené pre vyhľadávanie cesty v grafe, či inom prostredí kde sa potrebujeme dostať z bodu A do bodu B. Tieto prostredia môžeme rozdeliť na vážené a nevážené. Váha v týchto prostrediach predstavuje hodnotu pohybu z bodu A do bodu B. V neváženom prostredí je táto hodnota ekvivalentná hodnote počtu korkov z bodu A do bodu B a do bodu C, každý krok má hodnotu 1. Takže váha z bodu A do bodu C sa bude rovnať 2. Pri váženom prostredí má každý krok určitú hodnotu, ktorá je variabilná. Krok z bodu A do bodu B, by v príklade mal hodnotu 3, a krok z bodu B do bodu C, by mal hodnotu 5. Takto by celková váha cesty z bodu A do bodu C mala hodnotu 8. V bludisku je z princípu priestor nevážený, každý krok má rovnakú hodnotu. Určité algoritmy disponujú efektívnejšími metódami, ak sa berie do úvahy prostredie s váženými hodnotami kroku, ale taktiež sa dokážu pohybovať v neváženom prostredí. (Cormen, 2009) (Skiena, 2008)



Obrázok č. 5 Cesta vo váženom a neváženom prostredí
Zdroj: Vlastné spracovanie

1.7.3 Dátové štruktúry

Algoritmy prehľadávania priestoru potrebujú dátovú štruktúru, aby sledovali vyhľadávanie v priestore. Potrebujú dátovú štruktúru, aby algoritmy vedeli ukladať **hranicu**. Hranica predstavuje susedov aktuálnej pozície. Vhodný druh dátovej štruktúry môže byť fronta, tzv. *queue*. Operácie takejto štruktúry sú nasledovné:

IS-EMPTY(hranica) vráti hodnotu True ak fronta neobsahuje žiadne body.

POP(hranica) odstráni vrchný bod vo fronte a vráti ho.

TOP(hranica) vráti (ale neodstráni) vrchný bod vo fronte.

ADD(bod, hranica) vloží bod do fronty na svoje miesto. (Russel, 2021)

Tri ďalšie druhy fronty sú využívané v algoritmoch pre vyhľadávanie:

- **Prioritná fronta** (tzv. Priority queue) najprv využije operáciu pop na bod s minimálnou hodnotou, ktorá je určená nejakou hodnotiacou funkciou.
- **FIFO fronta** alebo First-in-First-out fronta najprv využije operáciu pop pre bod, ktorý bol pridaný ako prvý.
- **LIFO zásobník** alebo Last-in-First-out zásobník (inak nazývaný ako ***Stack***) využije operáciu pop pre bod, ktorý bol nedávno pridaný. (Russel, 2021)

1.7.4 Meranie výkonnosti

Výkonnosť predstavuje kritérium, podľa ktorého si vieme vybrať najlepší algoritmus. Výkonnosť algoritmu vieme odmerať štyrmi spôsobmi:

- **Kompletnosť** – Je algoritmus garantovaný nájsť riešenie ak nejaké existuje, a vie správne ohlásiť neúspech, ak riešenie neexistuje?
- **Nákladová optimálnosť** – Nájde algoritmus riešenie s tou najmenšou nákladovou hodnotou? (nákladová optimálnosť môže byť menovaná ako tzv. prípustnosť).
- **Časová komplexnosť** – Ako dlho trvá nájdenie riešenia? Môže byť merané v sekundách, alebo abstraktne pomocou počtu považovaných stavov a akcií.
- **Priestorová komplexnosť** – Koľko pamäte je potrebné na vykonanie vyhľadávania? (Russel, 2021)

Pre pochopenie kompletnosti, cieľ môže byť hocikde v stavovom priestore, takže kompletný algoritmus musí vedieť systematicky vyhľadávať každý stav, ktorý je dostupný zo začiatočného stavu. V konečnom stavovom priestore stačí, že algoritmus si bude pamätať svoju cestu, po čase sa algoritmus do cieľu dostane. (Russel, 2021)

V nekonečnom stavovom priestore by akcie algoritmu, ktorého jedinou možnosťou pohybu bol len krok dopredu, mal za následok nekonečnú cestu a množstvo nových stavov. V tomto prípade je algoritmus nekompletný, keďže ignoruje širšie rozširovanie v stavovom priestore. Aby bol algoritmus kompletný, musí pracovať **systematicky** v spôsobe vyhľadávanie nekonečného stavového priestoru, musí sa uistiť, že eventuálne dosiahne ľubovoľný stav, ktorý je pripojený k začiatočnému stavu. Žiaľ, aj dobrý algoritmus, v otvorenom nekonečnom priestore bez cieľa, by musel vyhľadávať do nekonečna, svoju činnosť nemôže zastaviť, lebo nevie, či nasledujúci stav nie je cieľový. (Russel, 2021)

Čas a priestor sa považujú za náročné meradlo, podľa špecifikácií problému. V teoretickej informatike, typickým meradlom je veľkosť stavového priestoru. Toto meradlo je dostačujúce ak sa jedná o explicitnú dátovú štruktúru, napríklad mapa. Dodatočne je v mnoho problémoch AI potrebné, aby priestor bol reprezentovaný iba implicitne podľa začiatočného stavu, akcií a dopad akcií na svet. Pre implicitný stavový priestor, komplexnosť môže byť meraná pomocou *d*, hĺbka alebo počet akcií pre optimálne riešenie; *m*, maximálne množstvo akcií v ceste; *b*, faktor vetvenia alebo počet susedných polí, ktoré treba brať do úvahy. (Russel, 2021)

2. CIEĽ PRÁCE

Cieľ bakalárskej práce je jednoznačný z názvu, hlavým účelom je výsledné porovnanie algoritmov v bludisku. Pre vykonanie porovnávania je prvým krokom vybranie algoritmov a prostredia, v ktorom ich budeme skúmať. Ďalším krokom je predstavenie zostavenia stavového priestoru, v ktorom budú vybrané algoritmy hľadať cieľ. Finálnym krokom v praktickej časti práce je samotné porovnávanie.

Pred samotnou implementáciou algoritmov je nevyhnutné oboznámiť sa s problematikou. Potrebujeme získať dostatočné znalosti teoretického fungovania algoritmov umelej inteligencie a ich spôsoby implementácie. Preto si objasníme pojmy umelá inteligencia a algoritmus, aby sme vedeli s čím vlastne pracujeme.

Týmto spôsobom taktiež získame znalosti o štruktúre programu, ktorý vytvárame. Predtým ako môžeme stavový priestor zostaviť, a algoritmy implementovať, musíme vybrať programovací jazyk, v ktorom bude program napísaný. Pre programovanie spomínaných algoritmov je ideálne vybrať programovací jazyk, v ktorom máme najväčšiu zásobu znalostí. Druhou možnosťou je výber jazyka, v ktorom sa tieto vyhľadávacie algoritmy často implementujú, čo znamená, že informácie o implementácii budú ľahko dostupne a vo veľkom množstve.

Získané znalosti o fungovaní algoritmov, ako aj o vybranom programovacom jazyku, aplikujeme pri implementácii a pri porovnávaní. Na kompletné oboznámenie sa s akciami, aké algoritmy vykonajú, je výhodné si ich implementáciu vo vývojovom prostredí neustále testovať a vyladiť. Pre jednoduchšie pochopenie fungovania algoritmov v práci uvádzame aj vizuálnu podobu algoritmov pomocou grafickej interpretácie a taktiež pseudokód, ktorý predstavuje zjednodušenú formu reálneho kódu.

Pri porovnávaní algoritmov máme možnosť implementovať do programu funkcie sledovania veličín, podľa ktorých vieme presne určovať efektívnosť algoritmov v stavovom priestore. Tieto sledované hodnoty prijmeme ako výstup na konzole vývojového prostredia.

Základnou vlastnosťou stavového priestoru je jeho statickosť a konečnosť pri porovnávaní diania algoritmov. Statický priestor je však možné manuálne modifikovať. Modifikácia sa vzťahuje na veľkosť a rozloženie priestoru, rovnako aj na manuálne nastavenie štartu a cieľa. Vďaka možnosti modifikácie môžeme otestovať správnosť práce algoritmov v rôznych variantoch stavového priestoru.

Pri implementácii samotných algoritmov je dôležité, aby v ich kóde existovali rovnaké funkcie, ktoré umožňujú agentovi meniť stav v priestore a tým sa v ňom pohybovať.

Po úspešnej implementácii je finálnym krokom porovnanie výstupy pozorovaných hodnôt pre všetky algoritmy. Z porovnania vieme jednoznačne určiť, ktorý algoritmus prehľadal stavový priestor najúspešnejšie.

3. METODIKA PRÁCE A METÓDY SKÚMANIA

Pri vypracovaní bakalárskej práci sme si ako prvý krok vybrali programovací jazyk Python kvôli osobnej preferencii. Následne sme sa bližšie oboznámili s pojmami spadajúcimi pod AI, a to v rozsahu vyhľadávania priestoru, ktoré sú podstatné na ovládanie problematiky.

Poznatky sme získali z dokumentácii jazyka Python, z verejného audio-vizuálneho, či textového obsahu na internete, a samozrejme z kníh určených pre danú problematiku. Podľa získaných poznatkov sme si vybrali algoritmy, ktoré budeme skúmať: Breadth-first search, Depth-first search, Dijkstra a A*.

Podľa vybraných algoritmov sme sa oboznámili s ich teoretickou a technickou implementáciou a pridelili im vhodné dátové štruktúry.

Predtým, ako sme začali vývoj programu, graficky sme znázornili beh jednotlivých algoritmov na fiktívnom stavovom priestore, pomocou poznatkov základných princípov pohybu algoritmov.

Po implementácii programu sme pomocou „debug“ funkcií sledovali, či je výstup rovnaký pri porovnaní s grafickým znázornením.

Pri vývoji Python programu sme dodatočne čerpali informácie z internetových fór, ktorým je napríklad Stackoverflow. Pomocou získaných informácií sme vedeli iteratívne zlepšovať celkovú funkcionalitu programu, tým, že sme implementovali lepšie metódy zostavovania špecifických častí kódu.

Dodatočne sme informácie preverovali s vybranými knižnými zdrojmi, ktoré poskytujú knižnicu funkčných reálnych implementácií algoritmov, o ktoré sme sa vedeli oprieť, ak sme pri vývoji programu narazili na nesprávny výstup algoritmu, či chybné hlásenie kvôli nesprávnej implementácii.

Po dokončení finálnej verzii programu sme sa posunuli na krok porovnávania, kde sme sledovali hodnoty výstupu a zapisovali ich do tabuľky (obrázok č. 23 a 24). Tabuľky slúžia na jednoduché zobrazenie rozdielov výstupov jednotlivých algoritmov.

3.1 Využité dátové štruktúry

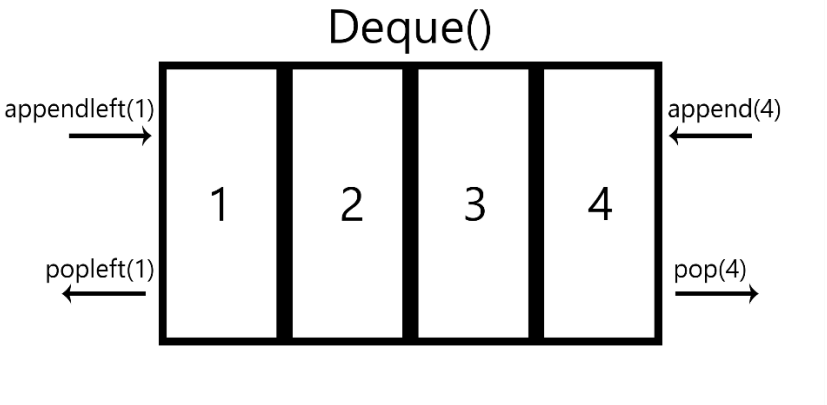
Queue, je jedna z mnohých dátových štruktúr, ktoré nám umožňujú uložiť dáta do zoznamu pre lepšiu funkcionality s dátami.

Na rozdiel od klasického zoznamu (*List*), tieto dátové štruktúry nám ponúkajú bezpečnejšie a efektívnejšie spracovanie dát. Zoznamy, ako také, sú vláknovo nebezpečné, čo znamená, že ak na zozname pracuje viacero vlákien systému, môže to viesť k závažnej nekonzistentnosti dát, je to preto, lebo v zoznamoch nie sú dáta zoradené. Taktiež, ich fungovanie pri väčších zoznamoch prináša vysoko pomalý výkon algoritmu. Táto pomalá efektívnosť je kvôli tomu, že ak vyberieme položku zo zoznamu, celý zoznam sa musí posunúť o jednu pozíciu, pri veľkom množstve dát uložených v takomto zozname, takýto postup prinesie stratu celkovej výkonnosti programu. (Zaczyński, 2022)

Queue je zoznam dát, ktorá ukladá položky v štýle *First-in-First-out (FIFO)*, čo znamená, že prvá položka, ktorá sa do Queue zoznamu vloží je tak tiež položka, ktorú vyberieme ako prvú. Je hlavne využívaný pri komunikácii z rôznymi vláknami.
(Zaczyński, 2022)

3.1.1 Deque (double-ended queue)

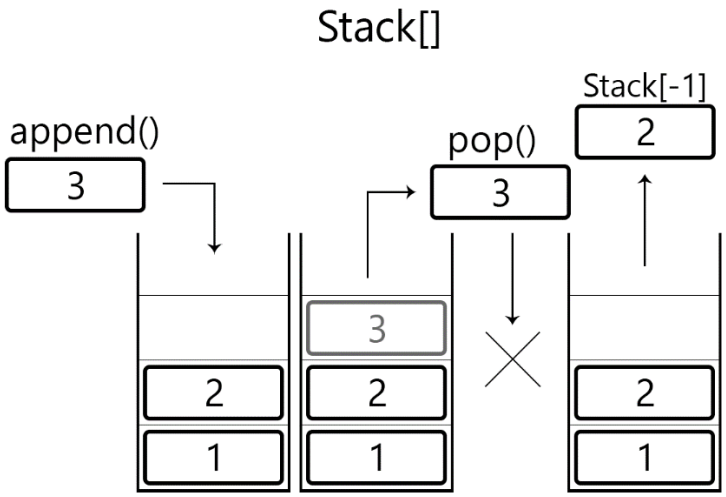
V našej implementácii algoritmu BFS miesto *Queue*, využívame *Deque*, teda **double-ended queue**. Z názvu si vieme odvodiť, že dokážeme vybrať a pridať položky z oboch strán, čo znamená, že vieme implementovať LIFO alebo FIFO metódu. Keďže dátová štruktúra *Queue* je najmä určená pri práci z vláknami, rozhodol som sa implementovať *Deque* pre algoritmus BFS. (Silveira, 2022) (Python, 01)



Obrázok č. 6 Deque znázornenie
Zdroj: vlastné spracovanie

3.1.2 Stack

Ukladá položky v štýle *Last-in-First-out (LIFO)*. Takže posledná pridaná položka, bude vybraná zo zoznamu. Ak vrchnú položku vybrať nepotrebujeme, využijeme index ***stack[-1]***, pomocou čoho vieme ukazovať na vrchnú položky uloženú v zozname. Funkcia *pop* odstráni vrchnú položku. Dátovú štruktúru *Stack* využívame pri algoritme *Depth-first search (DFS)*. (Zaczyński, 2022) (Python, 01)



Obrázok č. 7 Stack znázornenie
Zdroj: vlastné spracovanie

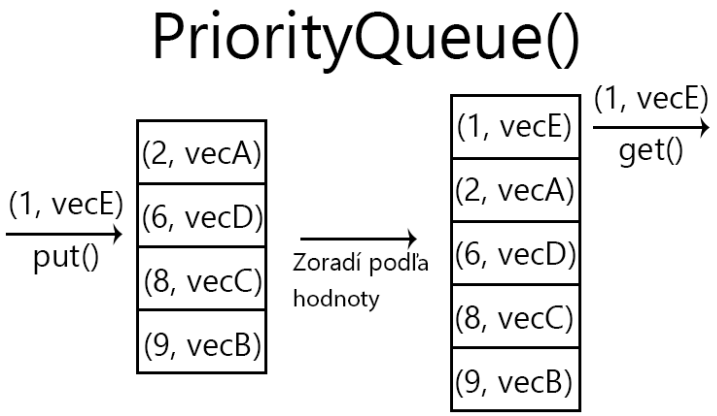
3.1.3 PriorityQueue

Pomocou tejto dátovej štruktúry, dokážeme porovnávať jednotlivé položky, ku ktorým je pridelená hodnota. Ďalej pomocou tejto hodnoty vie *PriorityQueue* zoradiť položky podľa priority. Položky v zozname budú mať takúto podobu: vec(priorita, položka). Položky nedokážeme porovnávať operátorom < pre porovnanie menšej ako x hodnoty, dodatočne, PriorityQueue nevie samostatne porovnať položky, ak sú hodnoty identické pre oboje položky. Potrebuje výpomoc „magickej metódy“ *__lt__*, ktorá tieto hodnoty bude porovnávať. Tieto metódy sú zabudované v programovacom jazyku Python. (Python, 01) (Zaric, 2022) (Geeksforgeeks, 2022) (Zaczyński, 2022)

```
def __lt__(self, other):
      return self.value < other.value
```

Táto metóda sa volá, ak sa do zoznamu vložia položky s identickou hodnotou priority, do zoznamu ich uloží podľa toho, ktorá položka bola skôr vybraná. Na vloženie

a vyberanie položiek sa využívajú funkcie *put()* a *get()*. Na vrchu zoznamu je vždy najvyššia priorita, teda položka, ktorú vyberieme pomocou funkcie *get()*. (Geeksforgeeks, 2022) (Kumar, 2020)



Obrázok č. 8 PriorityQueue znázornenie
Zdroj: vlastné spracovanie

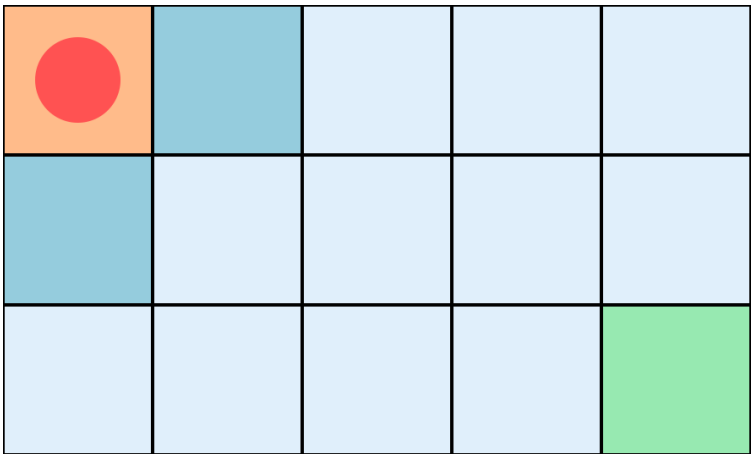
3.2 Breadth-first search

Keď každá akcia má rovnakú hodnotu, vhodnou voľbou bude BFS. Je jeden z najjednoduchších algoritmov vyhľadávania, predstavuje prototyp pre iné zložitejšie algoritmy. Primov algoritmus a Dijkstrov algoritmus fungujú na podobných princípoch ako BFS. Meno algoritmu je založené na jeho spôsobu vyhľadávania. Uniformne vyhľadáva svoje susedné polia podľa šírky. Algoritmus najprv vyhľadá nenavštívené políčka, ktorých vzdialenosť je najbližšie k momentálnej pozícii, až po prehľadaní týchto políčok, sa posunie o jednu vzdialenosť ďalej a proces opakuje. (Cormen, 2009) (Russel, 2021) (Zarembo, 2013)

BFS neuviazne, keď nájde slepú cestu. Ak z priestoru vyplýva riešenie, BFS je garantované ho nájsť. Okrem toho, ak exituje viacero riešení, tak nájde to minimálne. Je to kvôli tomu, že dlhšie cesty nie sú nikdy preskúmané dokým všetky kratšie cesty neboli preskúmané skôr. (Rich, 2009)

3.2.1 Grafické znázornenie

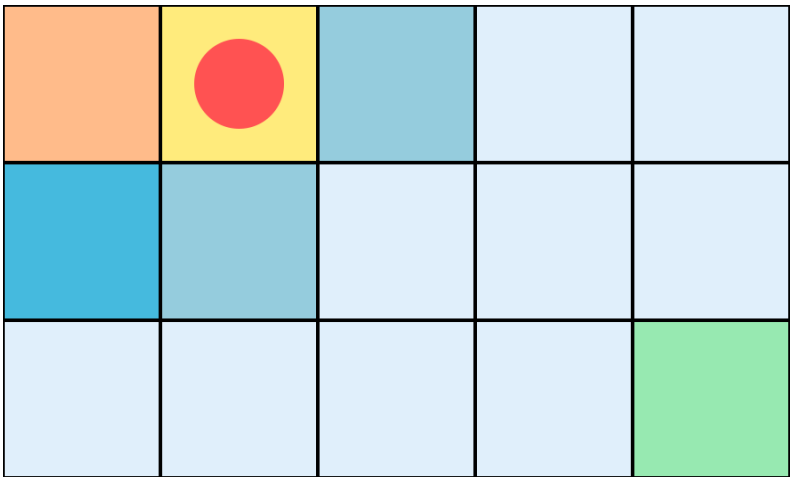
Pre názorné ukázanie funkcie algoritmu, majme zadané možnosti pohybu v tých to smeroch v danej rade: ***hore, vpravo, dole, vľavo.*** Algoritmus sa nemôže pohnúť mimo priestoru.



Obrázok č. 9 BFS Stav 0
Zdroj: vlastné spracovanie

V príklade máme znázornenú štartovaciu pozíciu oranžovým políčkom a momentálnu pozíciu červeným kruhom. Ďalej vidíme jeho susedné políčka vyznačené modrou farbou. Algoritmus si tieto susedné políčka uloží ako pozície, ktoré má vyhľadať a pridelí im hodnotu 1. Hodnota reprezentuje vzdialenosť od svojej momentálnej pozície. Susedov má uložených vo fronte, v ktorej sú susedia zoradený podľa poradia v akom sa do fronty vložili. V algoritme sú uložený susedia označení ako navštívení, aj keď v nich ešte nebol. Je to preto, aby sa nepridali susedia duplicitne.

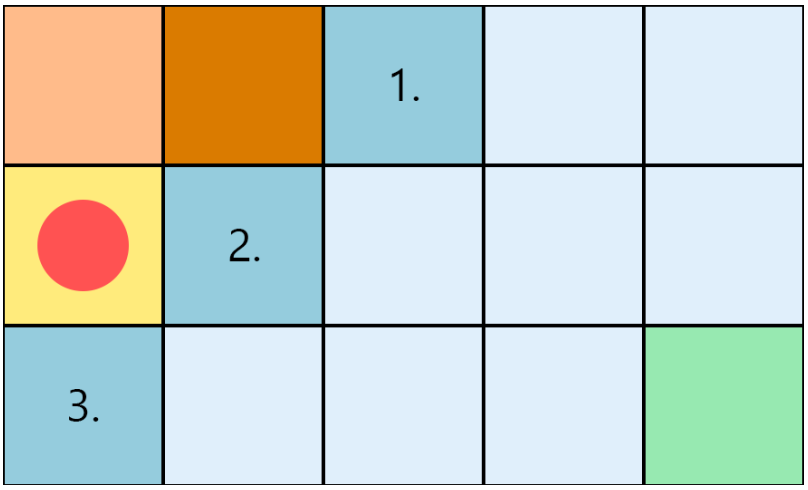
Algoritmus sa môže pohnúť o jedno políčko doprava alebo dole. Keďže koná podľa zadefinovanej postupnosti krokov, pohne sa najprv do pravého políčka. Toto políčko si označí ako navštívené, vyznačené žltou farbou.



Obrázok č. 10 BFS Stav 1
Zdroj: vlastné spracovanie

Z tejto novej pozície, jeho momentálnej pozície, sa pozrie na ďalšie susedné políčka a pridá im hodnotu vzdialenosti. Keďže pohyb na jeho momentálne políčko stálo jeden krok, hodnota vzdialenosti pre susedov bude rovná hodnote aktuálneho políčka + 1.

Sýtejšou modrou farbou je vyznačené políčko, ktoré je v sekvencii ďalšou prioritou na navštívenie. Keď že je druhé v poradí vo fronte, pohne sa naň.



Obrázok č. 11 BFS Stav 3 s postupom ďalších krokov
Zdroj: vlastné spracovanie

Sýtou oranžovou farbou sme vyznačili políčko, ktoré už algoritmus navštívil. Na svojej novej pozícii, vykoná tú istú akciu s jeho susednými políčkami. Políčko napravo od jeho pozície už je uložené vo fronte pre ďalšie políčka v poradí pre navštívenie, jeho pozíciu a hodnotu vzdialenosti si neuloží. Čísla na jednotlivých políčkach znázorňujú postupnosť vyhľadávania podľa políčok uložených vo fronte postupnosti. Algoritmus vo vyhľadávaní pokračuje, dokým sa jeho momentálna pozícia nebude rovnať pozícii cieľu. Cieľ je vyznačený zelenou farbou.

3.3 Depth-first search - recursive

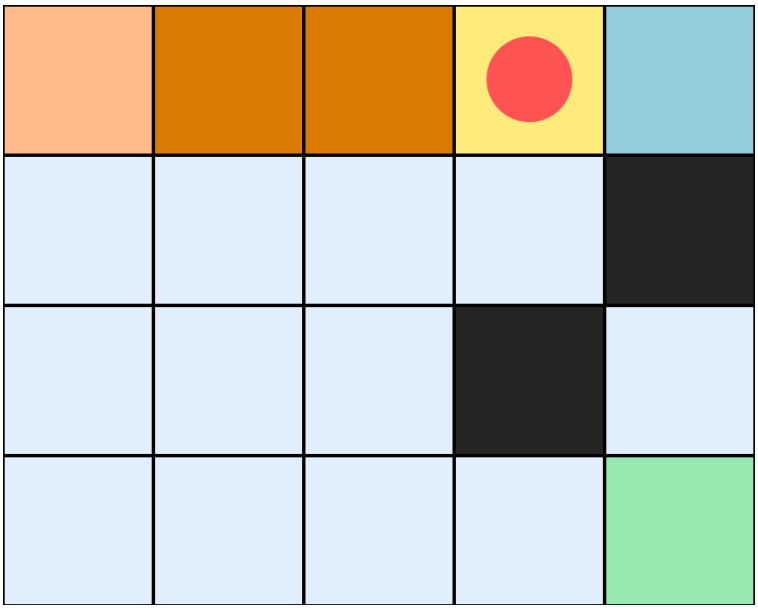
Podľa mena algoritmu vieme, že vyhľadáva čo najhlbšie, čo pre DFS znamená pokračovanie do jedného smeru, dokým nenarazí na problém. Rovnako ako pri algoritme BFS, najprv zistí aké má možnosti pohybu, ak sa vie pohnúť, pokračuje v rovnakom smere aj na ďalšej pozícii. Kvôli tomuto spôsobu vyhľadávania, algoritmus nie je garantovaný, že nájde najkratšiu možnú cestu v priestore. (Cormen, 2009)

Tento rekurzívny variant DFS pracuje s možnosťou vrátenia sa späť. Ak sa dostane do situácie kde nevie nájsť žiadnu cestu, jednoducho sa vráti späť na predošlé políčko a skúsi nájsť inú cestu.

DFS vyžaduje menej pamäte, keďže ukladá iba polia na svojej aktuálnej ceste. DFS má podľa prostredia šancu nájsť riešenie bez preskúmania väčšiny vyhľadávacieho priestoru. Toto predstavuje veľkú výhodu ak existuje viacej akceptovateľných riešení, stačí, že nájde jedno riešenie a môže svoj proces ukončiť. (Rich, 2009)

3.3.1 Grafické znázornenie

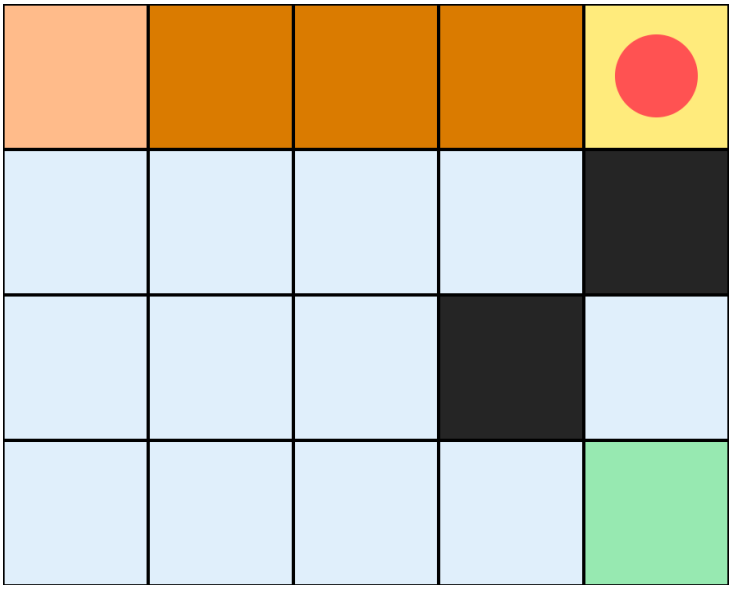
Pre ukázanie funkcie tohto algoritmu, tak isto ako pri BFS, predpokladajme, že jeho možnosti pohybu sú: ***hore, vpravo, dole, vľavo.*** Rad týchto akcií je dôležitým faktom pre DFS.



Obrázok č. 12 DFS Stav 3
Zdroj: vlastné spracovanie

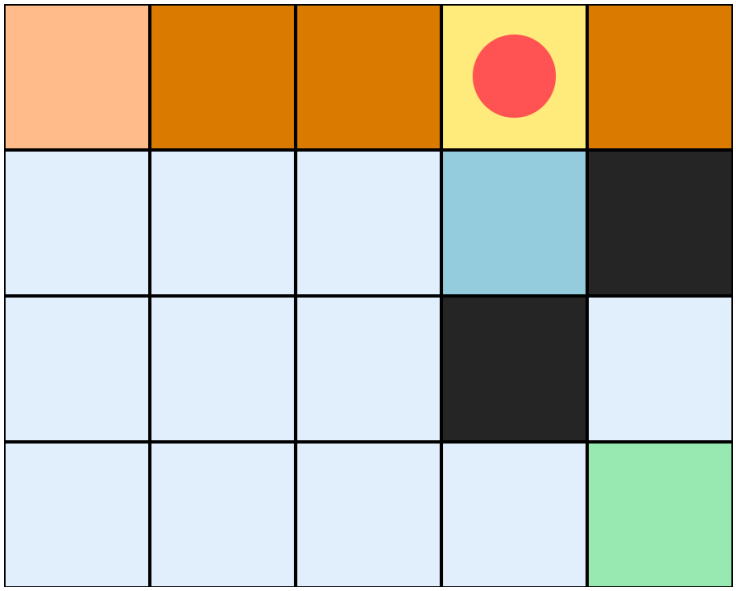
V názornej ukážke sa algoritmus pohol zo svojho začiatočného stavu, na stav 3. V porovnaní s ukážkou BFS, bol priestor zväčšený a boli pridané prekážka. Tieto prekážky,

či steny, sú vyfarbené tmavou farbou. Modrým políčkom je znázornené susedné políčko, do ktoré sa chce pohnúť. Políčko pod sebou neregistruje, keďže v rade svojich akcií je skôr pohyb vpravo.



Obrázok č. 13 DFS Stav 4
Zdroj: vlastné spracovanie

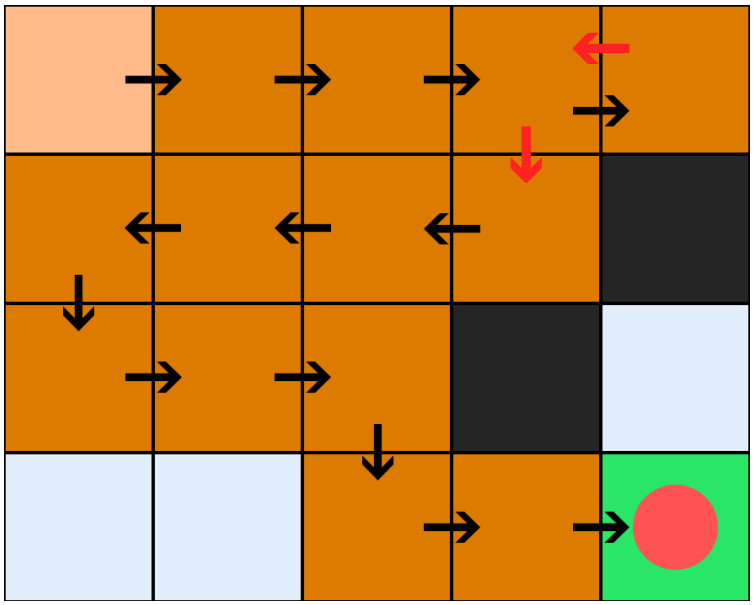
Algoritmus sa dostal do slepej cesty, v tomto prípade sa bude musieť vrátiť späť. Potom ako algoritmus pozrel všetkých svojich susedov, zistí, že jediná možnosť k dispozícii je vrátenie sa na svoje predchádzajúce políčko.



Obrázok č. 14 DFS Backtrack, Stav 5
Zdroj: vlastné spracovanie

Algoritmus sa vrátil späť. Predošlé políčko, z ktorého sa vrátil, označí ako navštívené. Jedinou možnosťou pre algoritmus je pohyb dole, keďže políčka vľavo a vpravo

už boli navštívené. Takýmto spôsobom bude pokračovať, dokým sa nedostane do cieľa, ak to je teda možné.



Obrázok č. 15 Postupnosť krokov po finálny stav DFS
Zdroj: vlastné spracovanie

Z finálneho stavu vieme vlastným okom posúdiť, že algoritmus síce nenašiel tú najkratšiu cestu, ale cieľ našiel.

3.4 Dijkstrov algoritmus

Dijkstrov algoritmus, pod iným menom známy ako Uniform-cost search. Je jeden z najznámejších algoritmov v informatike. Edsger Dijkstra prišiel na spôsob ako nájsť najkratšiu cestu v grafe, ktorého okraje mali nezáporné hodnoty. Následne publikoval algoritmus pod svojím menom v roku 1959. Fungovanie Dijkstrovho algoritmus je skoro identické algoritmu Breadth-first search. Rovnako ako BFS, posúva svoje vyhľadávanie o jednu hodnotu vzdialenosti ďalej. (Russel, 2021) (Zarembo, 2013) (Briliant, 2023)

Čim sa síce líšia je to, že Djikstrov algoritmus využíva miesto *Deque*, ***PriorityQueue***. Znamená to, že algoritmus pri porovnaní s BFS, by našiel lepšiu cestu, len ak by sa jednalo o priestor váženého charakteru. V našom prípade kde priestor je nevážený, budú výsledky oboch algoritmov takmer identické. Dijkstrov algoritmus je garantovaný nájsť najkratšiu cestu. (Skiena, 2008) (Briliant, 2023)

Ako referenciu na pohyb Dijkstrovho algoritmu, je vhodné využiť grafické znázornenie pohybu BFS.

3.5 A* algoritmus

Patrí medzi informované algoritmy vyhľadávania priestoru. Je to algoritmus široko zaužívaný pre vyhľadávanie a cestovanie grafom. Kombinuje funkcie algoritmu Uniform-cost search a čistej heuristiky, aby efektívne vypočítal optimálny výsledok. Je opisovaný ako vylepšenie Dijkstrovho algoritmu. Využíva best-first vyhľadávanie, aby našiel najkratšiu cestu. (Russel, 2021) (Zarembo, 2013) (Naeem, 2021)

Taktiež ako Dijkstrov algoritmus, jeho efektivita je najviac samozrejmá, ak sa jedná o priestor s váženými presunmi, keď že využíva dátovú štruktúru *PriorityQueue*.
Na vyhodnotenie cesty využíva funkciu

$$f(n) = g(n) + h(n)$$

kde $g(n)$ predstavuje vzdialenosť cesty od začiatočného stavu k n, a $h(n)$ predstavuje heuristiku, odhadovanú najkratšiu vzdialenosť z bodu n do konečného stavu,

f(n) = odhadovaná vzdialenosť najlepšej cesty, ktorá pokračuje z n do cieľa.

A* je garantovaný nájsť najkratšiu cestu. Ale či algoritmus dokáže vybrať optimálnu cestu závisí od vlastností heuristiky. Dôležitá vlastnosť heuristiky je prípustnosť, prípustná je vtedy ak nikdy neprecení náklady krokov na dosiahnutie cieľa. Ideálne je hodnota *h* rovnej hodnote potrebnej hodnoty na dosiahnutie cieľa. V takomto prípade by algoritmus vždy sledoval tú perfektnú cestu k cieľu, a nestrácal by čas preskúmavaním nepotrebných miest. Ak je zvolená heuristika, ktorá precení hodnotu *h*, cieľ je nájdený rýchlejšie, optimálnosť riešenia pri takejto heuristike klesá. Ak je zvolená heuristika, ktorá podcení hodnotu *h*, A* vždy nájde najlepšiu cestu. (Russel, 2021) (Naeem, 2021)

3.5.1 Prípustná heuristika

Pre prehľadávanie dvojdimenzionálneho stavového priestoru máme na výber dve prípustné heuristiky,
Euclidovská vzdialenosť,
kde Vzdialenosť = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Manhattanská vzdialenosť,
kde Vzdialenosť = $|x_2 - x_1| + |y_2 - y_1|$.

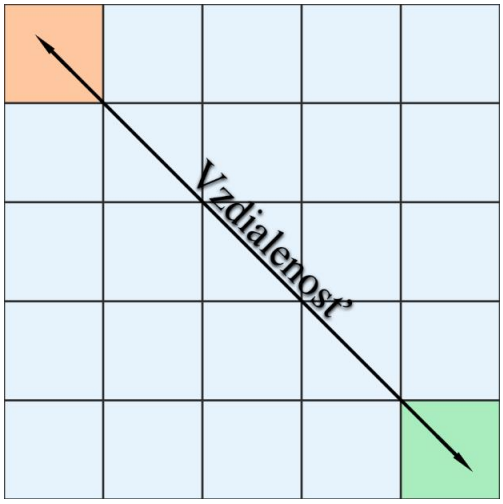
Euclidovská vzdialenosť predstavuje najkratšiu vzdialenosť medzi dvoma bodmi. Aplikuje pytagorovu vetu na vypočítanie vzdialenosti. Táto vzdialenosť predstavuje diagonálnu cestu priamo k cieľu. (Sharma, 2020) (Naeem, 2021)

Pre takúto vzdialenosť by bolo potrebné kalkulovať hodnotu jedného diagonálneho kroku ako:

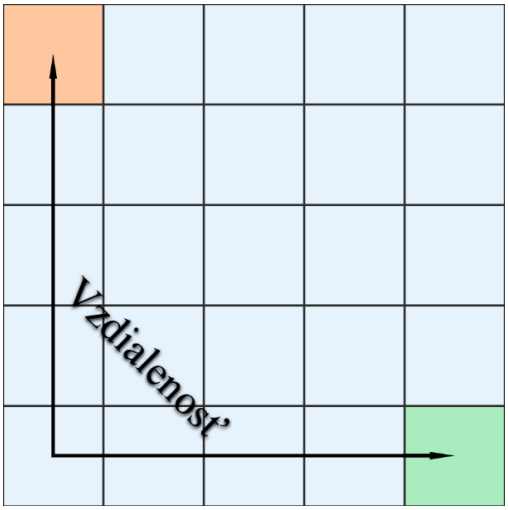
$$Vzdialenosť = \sqrt{2}$$

Každý diagonálny krok by v tomto prípade mal hodnotu 1.4.

Výsledkom výpočtu manhattanskej vzdialenosti je absolútna hodnota vzdialenosti medzi štartom a cieľom. (Sharma, 2020)

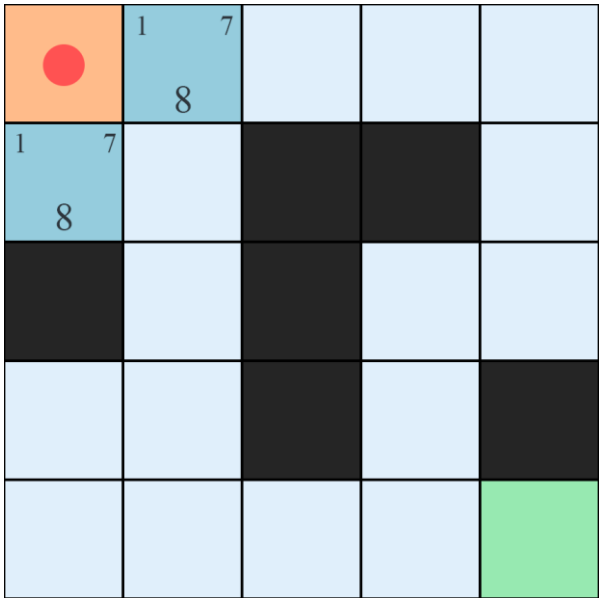


Obrázok č. 16 Euclidovská vzdialenosť znázornenie
Zdroj: vlastné spracovanie



Obrázok č. 17 Manhattanská vzdialenosť znázornenie
Zdroj: vlastné spracovanie

*3.5.2 Grafické znázornenie A**

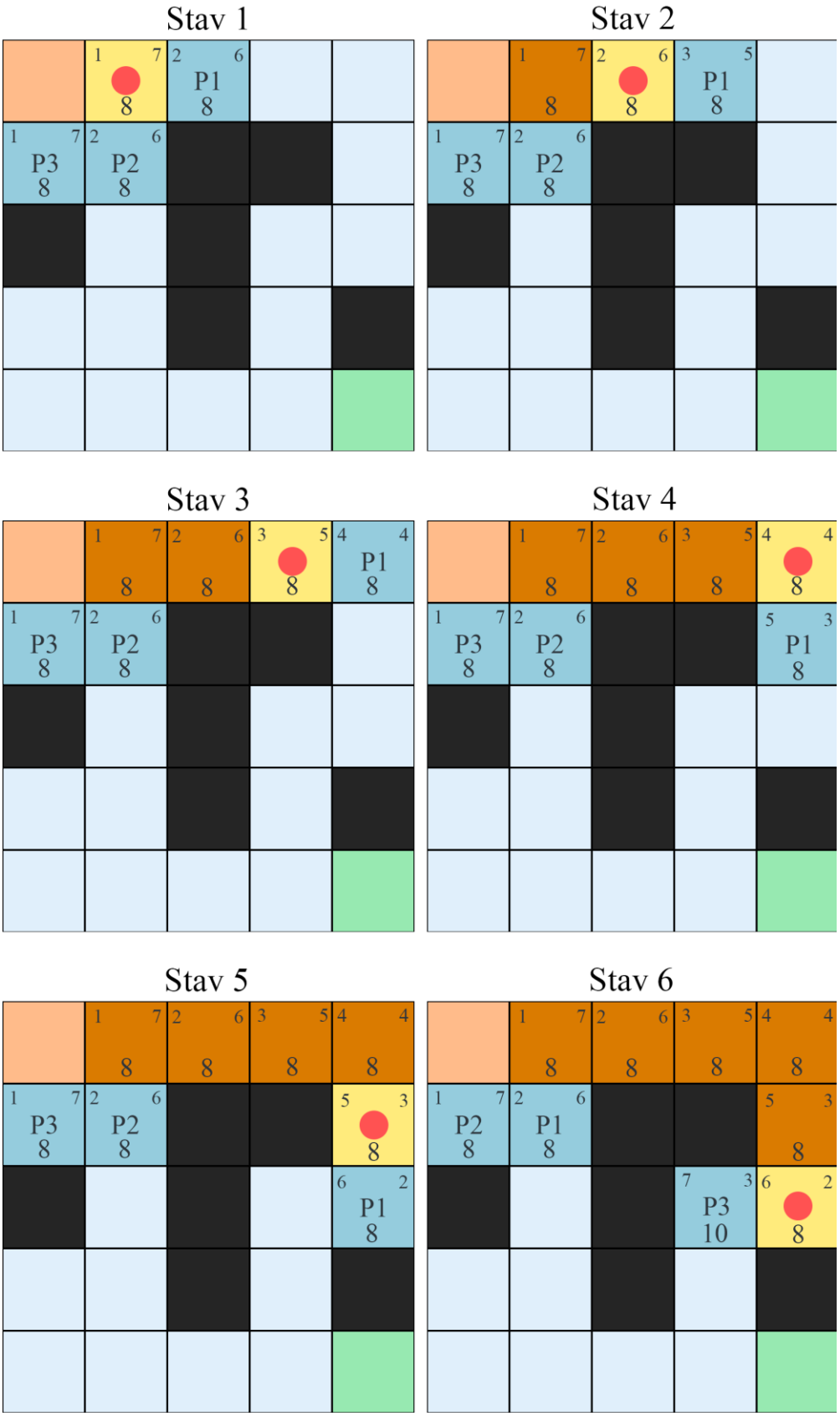


Obrázok č. 18 A Stav 0*
Zdroj: vlastné spracovanie

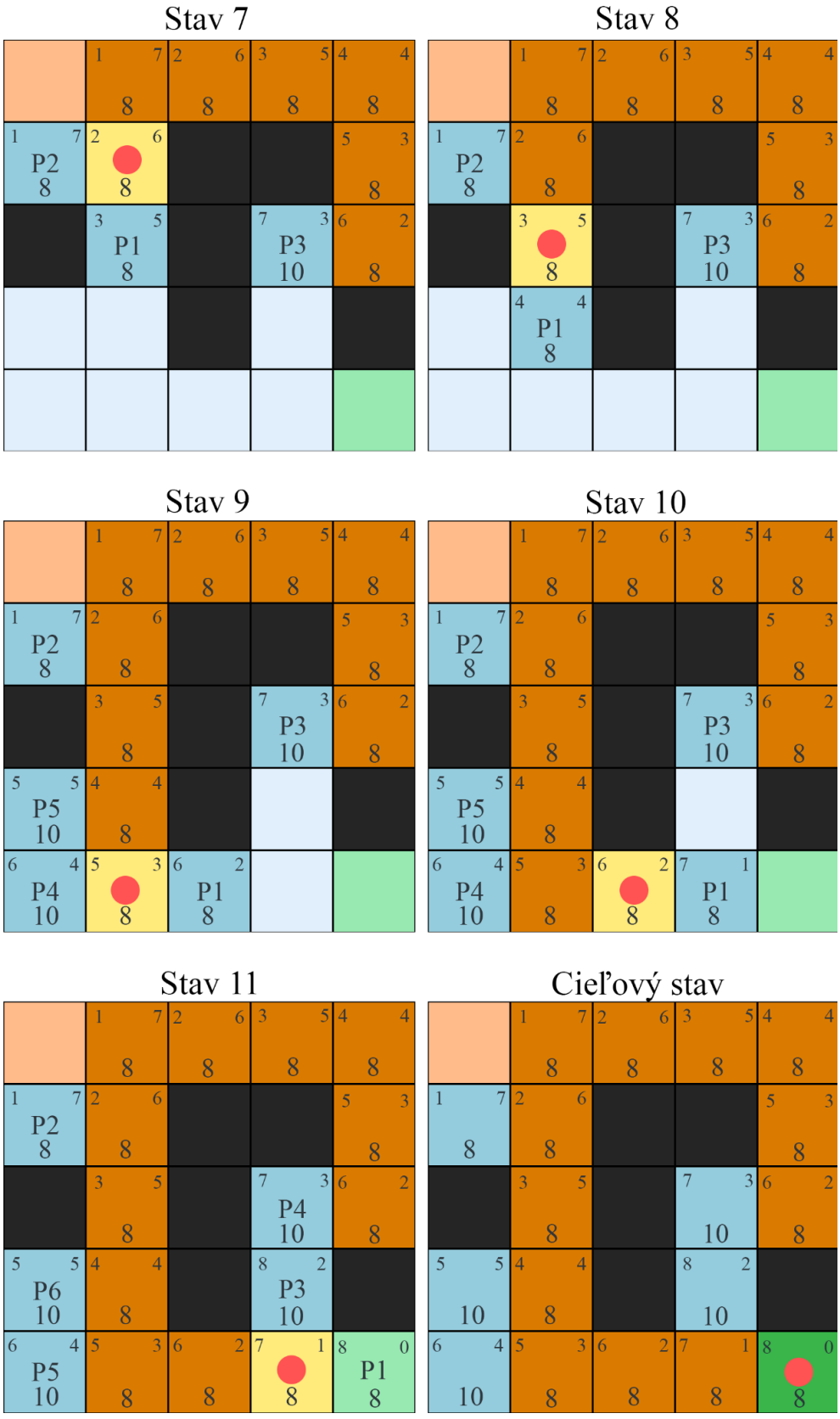
Vidíme, že algoritmus získal hodnotu $f(n) = g(n) + h(n)$ pre svojich susedov, kde f je rovné hodnote na spodku políčka, takže 8, g je rovné hodnote v ľavom hornom rohu políčka a hodnota h sa nachádza v pravom roku políčka.

Keď že f hodnoty, ktoré určujú pohyb sú rovnaké, algoritmus si vyberie políčko vpravo. Dôvodom je postupnosť krokov ***hore, vpravo, dole, vľavo***, kvôli ktorej vložilo políčko vpravo skôr do zoznamu. Primárnejšie je pre algoritmus vybrať najmenšiu heuristickú hodnotu h,

keď že tieto hodnoty sú tak isto rovnaké, nemá na výber a pokračuje podľa postupnosti.



Obrázok č. 19 A Stav 1 až 6*
Zdroj: vlastné spracovanie



Obrázok č. 20 A Stav 7 až finálny stav*
Zdroj: vlastné spracovanie

4. VÝSLEDKY PRÁCE

Ako výsledky práce si analyzujeme úlohu práce, ukážeme si dôležité časti kódu a kódy algoritmov ktoré sme implementovali. Následne prejdeme na porovnávanie algoritmov.

4.1 Pochopenie úlohy

Ako prvý krok sme si teda vybrali programovací jazyk a vývojové prostredie v ktorom budeme pracovať. Výber jazyka bol subjektívny. Zo zadania sme pochopili, že budeme sledovať činnosti algoritmov v bludisku.

Pre porovnávanie potrebujeme sledovať určité premenné, ktoré nám program vypíše, takže budeme sledovať len výstup na konzole vývojového prostredia, čo znamená, že vytvorenie grafického používateľského rozhrania alebo grafickej simulácie nebude potrebné. Ak by cieľom práce bolo vytvoriť aj grafické rozhranie pre program, využili by sme verejne dostupnú knižnicu pygame. Na porovnanie si potrebujeme vybrať meradlá výkonnosti, ktoré budeme sledovať.

Bludisko, náš priestor, musí byť spracované tak, aby sa všetky algoritmy dostali do cieľa a aby nebolo náhodne generované, keďže potrebujeme vedieť či implementácia pohybu algoritmov je správna. Pre porovnanie výsledkov potrebujeme, aby všetky algoritmy pracovali s rovnakým bludiskom, nedáva zmysel, aby každý algoritmus mal iné bludisko a rôzne hodnoty krokov v bludisku.

Ako návod na implementovanie algoritmov budeme potrebovať pseudokód a prideliť dátové štruktúry ktoré algoritmy využijú. Potrebujeme si zadeliť funkcie pre jednotlivé algoritmy, ktoré im umožnia bludisko čítať a pohybovať sa v ňom.

4.2 Programovací jazyk Python

Je viacparadigmový jazyk. Podporuje objektovo orientované aj štruktúrované programovanie. Python je vysokoúrovňový programovací jazyk, ktorý využíva dynamickú sémantiku. Jeho vysokoúrovňové zabudované dátové štruktúry v kombinácii s dynamickým

typingom a viazaním, robí Python atraktívnou voľbou na rýchly vývoj aplikácií, aj na použitie ako skriptovací alebo spojovací jazyk na prepojenie existujúcich komponentov. Jeho jednoduchý, ľahko čitateľný syntax kladie dôraz na celkovú čitateľnosť, čo vedie k nižším nákladom na údržbu kódu. Podporuje moduly a balíky, čím podporuje modularitu programu a ďalšie využitie rovnakého kódu. Interpretátor a rozsiahla štandardná knižnica sú dostupné open-source pre všetky hlavné platformy. (Python, 02)

Jednoduchá syntax, ktorá sa viac podobá všednej angličtine, robí Python prvou voľbou pre začiatočníkov, ktorý len vstupujú do sveta programovania. Tak tiež, tento jednoduchý syntax prináša jednoduché možnosti písania, čítania a debugovania kódu. Keďže jazyk Python je určený na všeobecné využitie, môže byť aplikovaný v mnohých oblastiach. Python môže byť využitý v mnoho aplikačných doménach. V reálnom svete je jeho využitie najmä pri analýze dát, vizualizácii dát, machine learningu, softvérovom a webovom vývoji a automatizácii úloh. Python predstavuje populárnu voľbu ak problematikou nastáva umelá inteligencia, či dátová veda. (Kosourova, 2022) (Python, 02)

4.3 Zostrojenie programu

Opíšeme ako hlavné funkcie v Python programe pracujú.

4.3.1 Priestor a cieľ - Bludisko

Bludisko bude predstavovať pre agenta jeho priestor, v ktorom môže konať. Bludisko je možné zostaviť rôznymi spôsobmi, ako napríklad náhodným generovaním, generovaním pomocou algoritmu hľadania ciest, s využitím knižnice alebo s načítaním obrázku alebo textového súboru, ktorý bludisko obsahuje. Naše bludisko bude mať jednoduchú podobu dvojrozmerného zoznamu jednotiek a núl, teda celých čísiel (dátový typ, int), kde jednotky sú voľné políčka a nuly sú steny. Agent má k dispozícii znalosti o celkovej veľkosti bludiska a či určité polia sú už navštívené. Nevie však, na ktorej pozícii sa nachádza stena.

Samotné rady pozícií sú tiež v zozname, pracujeme tak so zoznamom zoznamov.

```
def main():
    maze = [[1, 1, 1, 1, 1],
            [1, 1, 0, 1, 0],
            [1, 1, 1, 0, 1],
            [1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1]]
```

Dodatočne musíme definovať náš začiatočný bod a cieľový bod v bludisku.

```
goal = MazePosition(2, 4)
start = MazePosition(0, 0)
```

Tieto hodnoty sú neskôr volané pre jednotlivé definované funkcie algoritmov v tvare, def Algoritmus(maze, end, start). Podstatnou funkciou volaných funkcií nie je ich pomenovanie, ale pozícia v tele funkcie main.

Jednotlivé pozície na bludisku sú inicializované pomocou triedy MazePostition, ďalej využijeme konštruktora pre tieto pozície, kde x špecifikuje rad a y špecifikuje stĺpec.

```
class MazePosition:

    def __init__(self, x, y):
        self.x, self.y = x, y
```

[x0, y0]	[x0, y1]	[x0, y2]	[x0, y3]	[x0, y4]
[x1, y0]	[x1, y1]	[x1, y2]	[x1, y3]	[x1, y4]
[x2, y0]	[x2, y1]	[x2, y2]	[x2, y3]	[x2, y4]
[x3, y0]	[x3, y1]	[x3, y2]	[x3, y3]	[x3, y4]
[x4, y0]	[x4, y1]	[x4, y2]	[x4, y3]	[x4, y4]

Obrázok č. 21 Reprezentácia súradníc v priestore
Zdroj: vlastné spracovanie

Oranžové pole predstavuje pozíciu štart = [0,0], tmavé pole stenu a zelené pole pozíciu cieľ = [2,4].

4.3.2 Akcie – Pohyb a hľadanie susedov

Agent sa v priestore vie pohnúť ***hore, vpravo, dole, vľavo*** v danom poradí. Na vykonanie tejto akcie bude potrebovať vedieť svoju aktuálnu polohu.

```
class Node:

    def __init__(self, pos: MazePosition, cost):
```

```
        self.pos = pos
        self.cost = cost
```

Pomocou triedy Node, inicializujeme s konštruktorom pozíciu na bludisku a hodnotu cost, ktorá nám určuje počet krokov vykonaných na dosiahnutie danej pozície.
Pre definovanie orientácie chodu využívame zoznam s n-ticami, tzv. dátový typ, tuple.

```
adj_cells = [(-1, 0), (0, 1), (1, 0), (0, -1)]
```

Predstavuje zmenu aktuálnej pozície. Tuple (-1, 0) predstavuje prvú položku na zozname, znázorňuje zmenu pozície x o -1, posun hore, ďalej však pozíciu y nemeníme, keďže zmena je nulová. Ak by aktuálna pozícia bola [1, 1], nová pozícia by mala tvar [0, 1].

```
for adj_cell in adj_cells:
    x_pos = current_pos.x + adj_cell[0]
    y_pos = current_pos.y + adj_cell[1]
```

Pomocou for cyklu získame všetkých susedov. Keď že v zozname pre adj_cells máme 4 objekty, znamená to, že cyklus prejde všetky 4 možné pohyby, pomocou čoho vie susedom priradiť pozíciu. K aktuálnej pozícii sa pripočíta zmena pozície, daná podľa dátového typu tuple, kde pre novú x pozíciu potrebujeme nultý index v dátovom type tuple, vybraný pomocou adj_cell[0], keď že len ten pôsobí na zmenu x pozície. Rovnaká akcia sa vykoná pri získaní novej pozície y, kde miesto indexu 0 potrebuje index 1.

Ďalej v cykle máme pridelené príkazy if, ktoré sledujú či vybraná pozícia je mimo rozsahu bludiska, či pozícia na bludisku je voľná alebo či je stena. Posledný príkaz if zistí čo pozícia je uložená v zozname navštívených políčok, ak nie je, môžeme pokračovať ďalej.

Pri zisťovaní či pozícia je mimo hraníc bludiska, sa porovnávajú x a y pozície oproti celkovej pozície bludiska zadefinovaného v zozname maze. Funkcia *len*, nám vráti hodnotu dĺžky zoznamu. Funkcia len(maze) vráti množstvo radov x a funkcia len(maze[0]) vráti množstvo stĺpcov y.

```
if 0 <= y_pos < len(maze[0]) and 0 <= x_pos < len(maze):
    if maze[x_pos][y_pos] == 1:
        if not visited_nodes[x_pos][y_pos]:
```

Volaný zoznam visited_nodes je vytvorený pred začiatkom vyhľadávania priestoru. V zozname sa uložia všetky pozície v bludisku ako False. Označíme si tak pozície, ktoré

neboli navštívené. Pre štartovnú pozíciu túto hodnotu zmeníme na True, keďže predstavuje náš začiatočný stav.

```
visited_nodes = [[False for column in range(len(maze[0]))] for row in
range(len(maze))]
visited_nodes[start.x][start.y] = True
```

4.3.3 Meradlo výkonnosti – počet krokov, najkratšia cesta a čas

Ako výstup algoritmov, sledujeme najkratší počet krokov na dosiahnutie cieľa, počet navštívených polí a celkový čas trvania algoritmu. Pre algoritmus Depth-first search sledujeme dodatočne aj množstvo vrátení zo slepej cesty. Výstupom je aj, či algoritmus dokázal nájsť cestu, ak nenájde, vypíše tak.

```
Results:

BFS result:
Path found
Visited nodes =  22
Shortest path cost =  8
Time taken in milliseconds =  0.0525

DFS results:
Path found
Visited nodes =  13
Backtracks =  5
Shortest path cost found =  8
Path cost incl. backtracking =  18
Time taken in milliseconds =  0.0312

Dijkstra results:
Path found
Visited nodes =  22
Shortest path cost =  8
Time taken in milliseconds =  0.1331

A* results:
Path found
Visited nodes =  13
Shortest path cost =  8
Time taken in milliseconds =  0.094
```

Každý definovaný algoritmus musí pri nájdení cieľa vrátiť hodnoty sledovaných premenných.

```
return current_node.cost, taken_time
```

Pre získanie hodnoty najkratšej cesty jednoducho vypíšeme inkrementovanú hodnotu kroku cost ,pre posledné aktuálne políčko. Ako posledné políčko bude cieľ.

Pre celkový počet navštívených políčok, stačí inkrementovať túto hodnotu pri každej zmene aktuálnej pozície. Vypísanie počtu navštívených políčok a inkrementácia počtu sa nachádza priamo v tele jednotlivých algoritmov.

```
print("Visited nodes = ", visited)
```

Čas sledujeme pomocou funkcie time.perf_counter_ns. Je importovaná zo zabudovaného Python modulu menom ***time***. Vráti nám časovú hodnotu v nanosekundách, ktorú premeníme na milisekundy. Celkový čas behu algoritmu predstavuje časovú hodnotu od začiatku po koniec behu algoritmu.

```
end_time = time.perf_counter_ns()
taken_time = (end_time - start_time) / 1000000
```

Pri sledovaní časového diania algoritmu sme na výber mali viac možností.

1. time.clock (zastaralé)
2. time.monotonic
3. time.perf_counter
4. time.process_time
5. time.time

(Dunn, 2023)

Clock	Adjustable	Monotonic	Resolution	Tick Rate
process_time	False	True	1e-07	10,000,000
clock	False	True	4.665306263360271e-07	2,143,482
perf_counter	False	True	4.665306263360271e-07	2,143,482
monotonic	False	True	0.015625	64
time	True	False	0.015625	64

Obrázok č. 22 Porovnanie hodín
Zdroj: (Dunn, 2023) https://www.webucator.com/article/python-clocks-explained/

- Nastaviteľné (Adjustable)
  Hodiny môže meniť systémový administrátor. Nastaviteľné hodiny sú nespoľahlivé pre výpočet časovej delty.
- Monotonické (Monotonic)
  Tieto hodiny sú jednosmerné. Dokážu nám vrátiť len relatívny čas na základe dvoch udalostí.
- Rozlíšenie (Resolution)
  Čas medzi tickmi hodín (Tick predstavuje malú časovú jednotu, určuje korektný čas v počítači). Čím menšia hodnota rozlíšenia, tým väčší počet tickov za časovú jednotku.
- Tick Rate
  Počet tickov za sekundu. Hodiny s vysokým rozlíšením budú mať vysoký tick rate.

(Dunn, 2023)

Z týchto možností je time.perf_counter najpresnejšou funkciou pre meranie časového rozdielu začiatku a konca fungovania algoritmu. (Dunn, 2023)

4.4 Pseudokód a implementácia algoritmov

4.4.1 Pseudokód BFS

```
Input: s as the source node


BFS (G, s)
let Q be queue.
Q.enqueue( s )


mark s as visited
while ( Q is not empty)
v = Q.dequeue( )


for all neighbors w of v in Graph G
if w is not visited
Q.enqueue( w )
mark w as visited
```

G, priestor, v ktorom sa algoritmus pohybuje,

s, štartovacia pozícia,

Q, front pre ukladanie políčok na navštívenie, funkcia *dequeue* nám vyberie políčko v poradí na prehľadanie uložené vo fronte,

v, momentálna pozícia políčka, bude sa rovnať políčku vybraného z frontu pomocou *dequeue*,

w, susedné políčka, pridáme políčko do frontu pomocou *enqueue*

(Lateef, 2023)

4.4.2 Algoritmus BFS

```
def bfs(maze, end, start):

    adj_cells = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    visited_nodes = [[False for column in range(len(maze[0]))]
                                for row in range(len(maze))]
    visited_nodes[start.x][start.y] = True

    queue = deque()
    queue.append(Node(start, 0))
    visited = 0

    start_time = time.perf_counter_ns()

    while queue:
        current_node = queue.popleft()
        current_pos = current_node.pos
        visited += 1

        if current_pos.x == end.x and current_pos.y == end.y:
            end_time = time.perf_counter_ns()
            taken_time = (end_time - start_time) / 1000000

            print("BFS result:")
            print("Path found")
            print("Visited nodes = ", visited)
            return current_node.cost, taken_time

        for adj_cell in adj_cells:
            x_pos = current_pos.x + adj_cell[0]
            y_pos = current_pos.y + adj_cell[1]
            if 0 <= y_pos < len(maze[0]) and 0 <= x_pos < len(maze):
                if maze[x_pos][y_pos] == 1:
                    if not visited_nodes[x_pos][y_pos]:
                        visited_nodes[x_pos][y_pos] = True
                        queue.append(Node(MazePosition(x_pos, y_pos),
                                          current_node.cost + 1))

    return -1
```

4.4.3 Pseudokód DFS

```
Input: s as the source node


DFS (G, s)
let S be stack.
S.[s]


mark s as visited
while ( S is not empty)
v = S[-1]
dead end = true


for all neighbors w of v in G
if w is not visited
S.append( w )
mark w as visited
dead end = false
break

if dead end
S.pop
```

G, priestor, v ktorom sa algoritmus pohybuje,

s, štartovacia pozícia,

S, zoznam pre ukladanie políčok na navštívenie, dátová štruktúra *Stack*,

v, momentálna pozícia políčka, bude sa rovnať políčku na vrchu zoznamu, získaná pomocou indexu -1 pre dátovú štruktúru *Stack,*

w, susedné políčka, pridáme políčko do zoznamu pomocou *append*, ak všetky susedné políčka sú nevhodné, do zoznamu sa žiadne políčko nepridáva, z vrchu zoznamu sa odstráni vrchné políčko pomocou funkcie *pop*.

Adaptácia pseudokódu BFS, zo zdroja (Lateef, 2023)

4.4.4 Implementácia DFS

```
def dfs(maze, end, start):

    adj_cells = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    visited_nodes = [[False for column in range(len(maze[0]))]
                                for row in range(len(maze))]
    visited_nodes[start.x][start.y] = True

    stack = [Node(start, 0)]
    visited = 0
    backtracks = 0

    start_time = time.perf_counter_ns()

    while stack:
        current_node = stack[-1]
        current_pos = current_node.pos

        if current_pos.x == end.x and current_pos.y == end.y:
            end_time = time.perf_counter_ns()
            taken_time = (end_time - start_time) / 1000000
            total_moves = visited + backtracks
            print("DFS results:")
            print("Path found")
            print("Visited nodes = ", visited)
            print("Backtracks = ", backtracks)
            return current_node.cost, taken_time, total_moves

        dead_end = True

        for adj_cell in adj_cells:
            x_pos = current_pos.x + adj_cell[0]
            y_pos = current_pos.y + adj_cell[1]
            if 0 <= y_pos < len(maze[0]) and 0 <= x_pos < len(maze):
                if maze[x_pos][y_pos] == 1:
                    if not visited_nodes[x_pos][y_pos]:
                        visited_nodes[x_pos][y_pos] = True
                        visited += 1
                        stack.append(Node(MazePosition(x_pos, y_pos),
```

```
                                                    current_node.cost + 1))
                        dead_end = False
                        break

        if dead_end:
            stack.pop()
            backtracks += 1

    return -1
```

4.4.5 Pseudokód Dijkstrovho algoritmu

```
Input: s as the source node


Dijkstra (G, s)
let pQ be PriorityQueue.
pQ.put(c , s )

mark s as visited
while ( pQ is not empty)
v = pQ.get( )

for all neighbors w of v in Graph G
if w is not visited
pQ.put(c , w )
mark w as visited
```

G, priestor, v ktorom sa algoritmus pohybuje,

s, štartovacia pozícia,

pQ, front pre ukladanie políčok na navštívenie, funkcia *get* nám vyberie políčko v poradí vyhľadávania uložené vo fronte,

v, momentálna pozícia políčka, bude sa rovnať políčku vybraného z frontu pomocou funkcie *get*,

w, susedné políčka, pridáme políčko do frontu pomocou funkcie *put.*

c, váha kroku, pre štart je nulová, inkrementuje sa každým ďalším krokom.

Adaptácia pseudokódu BFS, zo zdroja (Lateef, 2023)

4.4.6 Implementácia pre Dijkstrov algoritmus

```
def Dijkstra(maze, end, start):

    adj_cells = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    priorityq = queue.PriorityQueue()

    visited_nodes = [[False for column in range(len(maze[0]))]
                                for row in range(len(maze))]
    visited_nodes[start.x][start.y] = True

    start = Node(start, 0)

    priorityq.put((0, start))
    visited = 0

    start_time = time.perf_counter_ns()

    while priorityq:
        current_node = priorityq.get()[1]
        current_pos = current_node.pos
        visited += 1
        if current_pos.x == end.x and current_pos.y == end.y:
            end_time = time.perf_counter_ns()
            taken_time = (end_time - start_time) / 1000000

            print("Dijkstra results:")
            print("Path found")
            print("Visited nodes = ", visited)
            return current_node.cost, taken_time

        for adj_cell in adj_cells:
            x_pos = current_pos.x + adj_cell[0]
            y_pos = current_pos.y + adj_cell[1]
            if 0 <= y_pos < len(maze[0]) and 0 <= x_pos < len(maze):
                if maze[x_pos][y_pos] == 1:
                    if not visited_nodes[x_pos][y_pos]:
                        neighbor = Node(MazePosition(x_pos, y_pos),
                                                    current_node.cost + 1)
                        visited_nodes[x_pos][y_pos] = True
```

```
                        c = neighbor.cost
                        priorityq.put((c, neighbor))

    return -1
```

*4.4.7 Pseudokód algoritmu A**

```
Input: s as the source node


Astar(G, s)
let pQ be PriorityQueue.
pQ.put( g, h , s )

mark s as visited
while ( pQ is not empty)
v = pQ.get( )

for all neighbors w of v in Graph G
if w is not visited
f = h + g
pQ.put(f, h , w )
mark w as visited
```

G, priestor, v ktorom sa algoritmus pohybuje,

s, štartovacia pozícia,

f, celková hodnota heuristiky a váhy kroku

g, váha kroku, pre štart je nulová, inkrementuje sa každým ďalším krokom.

h, manhattanská vzdialenosť

pQ, front pre ukladanie políčok na navštívenie, funkcia *get* nám vyberie políčko v poradí vyhľadávania uložené vo fronte,

v, momentálna pozícia políčka, bude sa rovnať políčku vybraného z frontu pomocou funkcie *get*,

w, susedné políčka, pridáme políčko do frontu pomocou funkcie *put*.

Adaptácia pseudokódu BFS, zo zdroja (Lateef, 2023)

*4.4.8 Implementácia algoritmu A**

```
def A_Star(maze, end, start):

    adj_cells = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    priorityq = queue.PriorityQueue()

    visited_nodes = [[False for column in range(len(maze[0]))]
                                        for row in range(len(maze))]
    visited_nodes[start.x][start.y] = True
    start = NodeAstar(start, 0, heuristic=abs(start.x - end.x)
                                       + abs(start.y - end.y))
    priorityq.put((0, start.heuristic, start))
    visited = 0

    start_time = time.perf_counter_ns()

    while priorityq:
        current_node = priorityq.get()[2]
        current_pos = current_node.pos
        visited += 1

        if current_pos.x == end.x and current_pos.y == end.y:
            end_time = time.perf_counter_ns()
            taken_time = (end_time - start_time) / 1000000
            print("A* results:")
            print("Path found")
            print("Visited nodes = ", visited)
            return current_node.cost, taken_time

        for adj_cell in adj_cells:
            x_pos = current_pos.x + adj_cell[0]
            y_pos = current_pos.y + adj_cell[1]
            if 0 <= y_pos < len(maze[0]) and 0 <= x_pos < len(maze):
                if maze[x_pos][y_pos] == 1:
                    if not visited_nodes[x_pos][y_pos]:
                        visited_nodes[x_pos][y_pos] = True
                        neighbor = NodeAstar(MazePosition(x_pos, y_pos),
                                             current_node.cost + 1,
                                       heuristic=abs(x_pos - end.x)
```

```
                                                + abs(y_pos - end.y))
                        f = neighbor.heuristic + neighbor.cost
                        priorityq.put((f, neighbor.heuristic, neighbor))
    return -1
```

Nová trieda pre NodeAstar

Pre algoritmus A* sme vytvorili svoju vlastnú triedu, pomocou ktorej dokáže pracovať s hodnotou heurisitky. Je to preto, aby algoritmus vedel lepšie vrátiť korektnú hodnotu z priorityq, keď z neho pýtame hodnotu pomocou funkcie priority.get().

```
class NodeAstar:

    def __init__(self, pos: MazePosition, cost, heuristic=0):
        self.pos = pos
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return self.cost < other.cost
```

4.5 Porovnávanie algoritmov

Pre porovnávanie sme si vybrali bludisko veľkosti 12x12. Ako štart sme si pridelili pozíciu [1, 0], a ako cieľ pozíciu [11, 9]. Bludisko má nasledovnú podobu v kóde:

```
maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0],
        [0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0],
        [0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0],
        [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0],
        [0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0],
        [0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0],
        [0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]]
```

4.5.1 Výstup a jeho porovnanie.

Výstup si vložíme do tabuľky, v ktorej môžeme vidieť jednotlivé fakty výstupu.

Bludisko 12x12	BFS	DFS	Dijkstra	A*
Počet navštívených polí	58	50	58	48
Hodnota najkratšej cesty	29	29	29	29
Trvanie (ms)	0.1295	0.1166	0.3302	0.2823

Obrázok č. 23 Tabuľka výstupu programu
Zdroj: vlastné spracovanie

Z tabuľky vieme zhodnotiť, že hodnota cesty je pri všetkých algoritmoch rovnaká, ale z počtu navštívených polí si vieme všimnúť rozdielnosti algoritmov. Pri porovnaní počtu navštívených polí vieme určiť, že algoritmus A* sa pohyboval najefektívnejšie oproti ostatným algoritmom. Pri časovom trvaní, vyhľadávanie trvalo najkratšie algoritmu DFS, veľmi blízko rovnakej hodnoty je BFS. Rozdiel je natoľko malý, že by sme mohli považovať tieto časy za rovnaké.

Vyskúšajme porovnanie vo väčšom bludisku.

Bludisko 30x10	BFS	DFS	Dijkstra	A*
Počet navštívených polí	130	100	130	121
Hodnota najkratšej cesty	48	54	48	48
Trvanie (ms)	0.2854	0.2397	0.7635	0.7347

Obrázok č. 24 Tabuľka výstupu programu 2
Zdroj: vlastné spracovanie

V časovom trvaní je najrýchlejší DFS, tak isto aj pri počte navštívených polí. Ale ak by sme pripočítali hodnotu počtu vrátenia sa na predošlé políčko, tak by bola hodnota 146. Avšak nenašiel optimálnu cestu. Ostatné algoritmy optimálnu cestu našli. A* využil, oproti BFS a Dijkstrovmu algoritmu, najmenej krokov. Medzi nimi má najlepší čas BFS.

Z výsledkov nedokážeme presne určiť, ktorý algoritmu je najlepší. Preto je výber efektívnejšieho algoritmu podstatný na vybraní si ľubovoľnej sledovanej veličiny. Je podstatné si z týchto veličín vybrať jednu alebo viacej výsledných hodnôt, podľa ktorých vieme upresniť algoritmus s lepším výsledkom.

#	Navštívený počet	
1.	DFS	100
2.	A*	121
3.	BFS	130
3.	Dijkstra	130
4.	DFS + Backtrack	146

#	Hodnota cesty	
1.	BFS	48
1.	Dijkstra	48
1.	A*	48
2.	DFS	54

#	Trvanie	
1.	DFS	0.2397
2.	BFS	0.2854
3.	A*	0.7347
4.	Dijkstra	0.7635

Obrázok č. 25 Tabuľky kategorizujúce sledované veličiny a zoradenie výsledkov
Zdroj: vlastné spracovanie

ZÁVER

Cieľom práce bolo vytvoriť program, ktorý implementuje algoritmy v bludisku a na základe výstupu algoritmy porovnať.

Počas pracovania na problematike sme získali dostatočné množstvo teoretických a praktických poznatkov, vďaka ktorým sme vedeli dokončiť praktickú časť práce. Docielili sme tak teda implementovanie jednotlivých algoritmom, a dokázali sme ich porovnať medzi sebou.

Fungovanie algoritmov sme si navrhli viacerými spôsobmi pre zjednodušené pochopenie algoritmov. Využili sme grafické znázornenia pohybu v priestore, ďalej, ako viac technický návrh, sme napísali hlavné telo algoritmu vo forme pseudokódu. Nakoniec, sme si ukázali implementáciu algoritmov v programovacom jazyku Python.

Z porovnávania sme dospeli k názoru, že ak by sme chceli niektorý algoritmus vybrať ako najlepší, tak to nie je možné v obecnom prípade. Keďže všetky algoritmy sú v niečom inom lepšie ako tie ostatné, treba vhodný algoritmus vybrať jedine podľa výsledkov, ktoré môžu byť v iných okolnostiach prioritnejšie .

Algoritmus DFS by sme vybrali, ak nepotrebujeme nájsť tú najkratšiu cestu a stačilo by nám len to, že sa dostane do cieľa.

Ak chceme, aby algoritmus vybral cestu do cieľa čo najrýchlejšie, máme na výber algoritmy BFS a DFS. Ak je pre nás dôležité nájdenie optimálnej cesty, zvolili by sme algoritmus BFS. Je samozrejmé, že náš výber je možné aplikovať len pri bludiskách v ktorých sme algoritmus pozorovali.

Z pozorovania Algoritmu A* vyplynulo, že síce pracoval dlhšie, ale vedel nájsť optimálnu cestu s menej navštíveným množstvom ostatných polí, a to jedine vďaka využitiu heuristickej funkcie.

Keďže sme pracovali s neváženým bludiskom, kde hodnota kroku z aktuálnej pozície na ďalšiu je rovná 1, nedokážeme spozorovať plný potenciál Dijkstrovhu algoritmu. Ak by teda pracoval v bludisku, ktorého hodnoty kroku na každom poli boli rôzne, počet navštívených polí by bol bližší k algoritmu A*.

Treba brať do úvahy veľkosť, zložitosť a typ priestoru, keďže tieto špecifikácie majú najväčší vplyv na výber vhodného algoritmu. Preto, výsledky porovnania nie je možné aplikovať ako všeobecnú príručku pri výbere algoritmu na priestor rozdielny od toho ktorý bol sledovaný v tejto práci.

Zameraním práce nebolo získanie podrobných technických informácií o časovej komplexnosti v matematickom tvare „Big *O* Notation“, keď že presné teoretické údaje o tomto tvare nie sú dostatočné, ak sa jedná o pohyb vybraných algoritmov v dvojdimenzionálnom prostredí. Samotná zložitosť „Big *O* Notation“ by mala, za pravdepodobný výsledok nesprávnu formuláciu v práci. Taktiež sme v práci ako parameter nesledovali presnú pamäťovú komplexnosť pri porovnávaní.

Na záver dúfam, že táto práca bude prínosom najmä pre čitateľa, ktorého daná tematika zaujíma, no nemusí mať nutne kompletné znalosti o fungovaní algoritmov v praxi.

Verím, že táto práca vzbudí záujem čitateľa získať hlbšie znalosti v sfére umelej inteligencie.

ZOZNAM POUŽITEJ LITERATÚRY

(Briliant, 2023) Dijkstra's Shortest Path Algorithm [online]. Briliant.org. 2023. [citované 06.05.2023]. Dostupné na internete: https://brilliant.org/wiki/dijkstras-short-path-finder/

(Cormen, 2009) CORMEN, T. H. – et. al.: Introduction to Algorithms [online]. 3. vyd. Cambridge, Massachusetts: MIT Press, 2009. ISBN 978-0-262-03384-8. Dostupné na internete: https://github.com/calvint/AlgorithmsOneProblems/blob/master/Algorithms/Thomas%20H.%20Cormen%2C%20Charles%20E.%20Leiserson%2C%20Ronald%20L.%20Rivest%2C%20Clifford%20Stein%20Introduction%20to%20Algorithms%2C%20Third%20Edition%20%202009.pdf

(Diaz, 2023) DIAZ, M., 2023: What is AI? Everything to know about artificial intelligence [online]. [citované 26.04.2023]. Dostupné na internete: https://www.zdnet.com/article/what-is-ai-heres-everything-you-need-to-know-about-artificial-intelligence/

(Dunn, 2023) DUNN, N.: Python Clocks Explained [online]. Webucator, Inc. [citované 08.05.2023]. Dostupné na internete: https://www.webucator.com/article/python-clocks-explained/

(Europe, 2020) Artificial Intelligence: Threats and opportunities: News: European parliament. In Artificial intelligence: threats and opportunities | News | European Parliament [online]. 2020. [citotavané 14.07.23]. Dostupné na internete: https://www.europarl.europa.eu/news/en/headlines/society/20200918STO87404/artificial-intelligence-threats-and-opportunities?&at_campaign=20234-Digital&at_medium=Google_Ads&at_platform=Search&at_creation=RSA&at_goal=TR_G&at_audience=artificial+intelligence&at_topic=Artificial_intelligence&at_location=SK&gclid=CjwKCAjw2K6lBhBXEiwA5RjtCTXht5tkUhdXBXCQbzGL9Xbr6xZB4gSWH-sLbvLBAHI2YMFHrxqnkhoCarUQAvD_BwE

(Geeksforgeeks, 2022) anon. 2022. Python - __lt__ magic method [online]. [citované 04.05.2023]. Dostupné na internete: https://www.geeksforgeeks.org/python-__lt__-magic-method/

(Kosourova, 2022) KOSOUROVA, E., 2022.: What is Python Used For? 7 Real-Life Python Uses [online]. [citované 05.05.2023]. Dostupné na internete: https://www.datacamp.com/blog/what-is-python-used-for

(Kumar, 2020) KUMAR, B. 2020. Priority queue in Python [online]. [citované 04.05.2023]. Dostupné na internete: https://pythonguides.com/priority-queue-in-python/

(Lateef, 2023) LATEEF, Z. 2023. All You Need To Know About The Breadth First Search Algorithm [online]. [citované 04.05.2023]. Dostupné na internete: https://www.edureka.co/blog/breadth-first-search-algorithm/

(Malwarebytes, 2023) Risks of AI & Cybersecurity: Risks of artificial intelligence. In *Malwarebytes* [online]. 2023. [citované 14.07.23]. Dostupné na internete: https://www.malwarebytes.com/cybersecurity/basics/risks-of-ai-in-cyber-security

(Naeem, 2021) NAEEM, M. A. 2021. A-Star (A*) Search for Solving a Maze using Python (with visualization) [online]. [citované 06.05.2023]. Dostupné na internete: https://levelup.gitconnected.com/a-star-a-search-for-solving-a-maze-using-python-with-visualization-b0cae1c3ba92

(Python, 01) Python 3.7 Documentation [online]. Python Software Foundation. 2023. Dostupné na internete: https://docs.python.org/3.7/

(Python, 02) In About Python [online]. Python Software Foundation. 2023. Dostupné na internete: https://www.python.org/

(Rich, 2009) RICH, E. - KNIGHT, K. - NAIR, S. B.: Artificial Intelligence [online]. 3. vyd. New Delhi: Tata McGraw Hill Education Private, Ltd., 2009. ISBN: 978-0-07-008770-5. Dostupné na internete: https://github.com/saranshbht/msc-books/blob/master/M.Sc.%20CS%20Sem-1/Artificial%20Intelligence/Kevin%20Knight%2C%20Elaine%20Rich%2C%20B.%20Nair%20-%20Artificial%20Intelligence%20(2010%2C%20Tata%20McGraw-Hill%20Education%20Pvt.%20Ltd.)%20-%20libgen.lc.pdf

(Russel, 2021) RUSSEL, S. - NORVIG, P.: Artificial Intelligence: A Modern Approach, 4. vyd. Hoboken, New Jersey: Pearson Education, Inc., 2021. ISBN: 978-0-13-461099-3

(Sharma2020) SHARMA, P. 2020. Understanding Distance Metrics Used in Machine Learning. In *Analytics Vidhya* [online]. [citované 11.05.2023]. Dostupné na internete:

https://www.analyticsvidhya.com/blog/2020/02/4-types-of-distance-metrics-in-machine-learning/

(Silveira, 2022) SILVEIRA, O. S. 2022 Python's Deque: How to Easily Implement Queue and Stacks [online]. [citované 04.05.2023]. Dostupné na internete: https://www.dataquest.io/blog/python-deque-queues-stacks/

(Skiena, 2008) SKIENA, S. S.: The Algorithm Design Manual [online]. 2. vyd. London: Springer Verlag, Ltd., 2008. ISBN: 978-1-84800-069-8. Dostupné na internete: https://github.com/aforarup/interview/blob/master/Data%20Structures%20and%20Algorithm/Algorithm%20Books/The%20Algorithm%20Design%20Manual%20by%20Steven%20S.%20Skiena.pdf

(Zaczyński, 2022) ZACZYŃSKI, B. 2022. Python Stacks, Queues, and Priority Queues in Practice [online]. [citované 04.05.2023]. Dostupné na internete: https://realpython.com/queue-in-python/

(Zarembo, 2013) ZAREMBO, I. - KODORS, S.: Pathfinding Algorithm Efficiency Analysis in 2D Grid. Proceedings of the 9th International Scientific and Practical Conference [online]. Rēzekne: RA publ., 2013. vol. 11 s 46-47. ISSN 1691-5402. Dostupné na internete: https://www.researchgate.net/publication/282488307_Pathfinding_Algorithm_Efficiency_Analysis_in_2D_Grid

(Zaric, 2022) ZARIC, D., 2022.: Priority Queue in Python [online]. [citované 04.05.2023]. Dostupné na internete: https://blogboard.io/blog/knowledge/priority-queue-in-python/