VIŠJA STROKOVNA ŠOLA ACADEMIA

MARIBOR

PRIMERJAVA ALGORITMOV ISKANJA POTI IN UMETNE INTELIGENCE V NAKLJUČNO GENERIRANEM LABIRINTU

Kandidat: Žan Žerak

Vrsta študija: študent izrednega študija

Študijski program: Informatika

Mentor predavatelj: mag. Ervin Schaff

Mentor v podjetju: Vitomil Selič, dipl. inž. inf.

Lektorica: Karin Vidmar, prof. slov. in angl.

Maribor, 2025

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisani Žan Žerak sem avtor diplomskega dela z naslovom Primerjava algoritmov iskanja poti in umetne inteligence v naključno generiranemu labirintu, ki sem ga napisal pod mentorstvom mag. Ervina Schaffa.

S svojim podpisom zagotavljam, da:

- je predloženo delo izključno rezultat mojega dela;
- sem poskrbel, da so dela in mnenja drugih avtorjev, ki jih uporabljam v predloženi nalogi, navedena oz. citirana skladno s pravili Višje strokovne šole Academia Maribor;
- se zavedam, da je plagiatorstvo – predstavljanje tujih del oz. misli kot mojih lastnih, kaznivo po Zakonu o avtorskih in sorodnih pravicah (Uradni list RS, št. 16/07 – uradno prečiščeno besedilo, 68/08, 110/13, 56/15 in 63/16 – ZKUASP); prekršek pa podleže tudi ukrepom Višje strokovne šole Academia Maribor skladno z njenimi pravili;
- skladno z 32.a členom ZASP dovoljujem Višji strokovni šoli Academia Maribor objavo diplomskega dela na spletnem portalu šole.

Maribor, oktober 2025 Podpis študenta:

ZAHVALA

Zahvaljujem se mentorju mag. Ervinu Schaffu za mentorstvo pri diplomskem delu. Zahvaljujem se svoji družini, ki me je podpirala v času študija.

POVZETEK

V diplomskem delu je predstavljena primerjava algoritmov iskanja poti in umetne inteligence pri reševanju naključno generiranih labirintov. Namen diplomskega dela je ugotovitev učinkovitosti in hitrosti različnih pristopov k reševanju poti v kompleksnih strukturah.

V teoretičnem delu so obravnavani algoritmi iskanja najkrajše poti, in sicer algoritem A*, preiskovanje v širino in Dijkstrov algoritem. Predstavljena je tudi teorija grafov in mrež ter vloga hevristike pri iskanju poti. Posebno poglavje je namenjeno umetni inteligenci in njenemu pomenu pri reševanju problema iskanja poti.

Za empirično raziskavo je bila razvita aplikacija v programskem jeziku Python z grafičnim uporabniškim vmesnikom. Aplikacija omogoča generiranje naključnih labirintov različnih velikosti ter vizualizacijo delovanja posameznih algoritmov. Za simulacijo umetne inteligence je bil implementiran pristop globokega Q-učenja z uporabo knjižnice PyTorch.

Testiranje je potekalo na tisoč naključno generiranih labirintih za naslednje velikosti, in sicer 11x11, 21x21 in 31x31. Merjeni so bili čas iskanja, število obiskanih vozlišč in uspešno doseganje cilja. Algoritmi iskanja poti so pokazali 100-% uspešnost pri vseh velikostih labirintov. Algoritem A* se je izkazal kot najbolj optimiziran z najmanjšim obiskom vozlišč, medtem ko je Dijkstrov algoritem najhitreje našel končno vozlišče. Umetna inteligenca je dosegla 60,3-% uspešnost na labirintih velikosti 11x11, vendar se je njena učinkovitost občutno zmanjšala pri večjih labirintih, in sicer 8,1-% uspešnost na labirintih velikosti 21x21 in 2,8-% uspešnost na labirintih velikosti 31x31.

Rezultati nakazujejo, da so tradicionalni algoritmi iskanja poti bolj zanesljivi in učinkoviti za praktične aplikacije, kjer je potrebna 100-% učinkovitost in uspešnost. Umetna inteligenca ponuja potencialno boljše rezultate, vendar zahteva obsežnejše učenje na raznolikih podatkih za doseganje primerljivih rezultatov z algoritmi iskanja poti. Diplomsko delo prispeva k boljšemu razumevanju prednosti in omejitev različnih pristopov iskanja poti v kompleksnih strukturah. V prihodnosti bi bilo smiselno raziskati še druge metode umetne inteligence ter optimizirati procese učenja, da bi se dosegla večja uspešnost pri reševanju zahtevnejših labirintov.

Ključne besede: *umetna inteligenca, strojno učenje, mreže, grafi, algoritmi iskanja poti.*

ABSTRACT

COMPARISON OF PATH FINDING ALGORITHMS AND ARTIFICIAL INTELLIGENCE IN A RANDOMLY GENERATED MAZE

This thesis presents a comparison of pathfinding algorithms and artificial intelligence in solving randomly generated mazes. The purpose of the thesis is to determine the effectiveness and speed of different approaches to solving paths in complex structures.

The theoretical part deals with algorithms for finding the shortest path, namely the A* algorithm, breadth-first search, and Dijkstra's algorithm. Graph and network theory and the role of heuristics in pathfinding are also presented. A special chapter is devoted to artificial intelligence and its importance in solving the pathfinding problem.

For empirical research, an application was developed in the Python programming language with a graphical user interface. The application enables the generation of random mazes of various sizes and the visualization of the operation of individual algorithms. To simulate artificial intelligence, a deep Q-learning approach was implemented using the PyTorch library.

The testing was conducted on a thousand randomly generated mazes of the following sizes: 11x11, 21x21, and 31x31. The search time, number of visited nodes, and successful goal achievement were measured. The pathfinding algorithms showed 100% success for all maze sizes. The A* algorithm proved to be the most optimized with the fewest node visits, while Dijkstra's algorithm found the final node the fastest. Artificial intelligence achieved 60.3% success on 11x11 mazes, but its efficiency decreased significantly in larger mazes, with 8.1% success in 21x21 mazes and 2.8% success in 31x31 mazes.

The results indicate that traditional pathfinding algorithms are more reliable and effective for practical applications where 100% efficiency and success are required. Artificial intelligence offers potentially better results but requires more extensive learning on diverse data to achieve comparable results with pathfinding algorithms. This thesis contributes to a better understanding of the advantages and limitations of different approaches to pathfinding in complex structures.

Keywords: *artificial intelligence, machine learning, networks, graphs, pathfinding algorithms*

KAZALO VSEBINE

1 UVOD 9

1.1 Opis področja in opredelitev problema 9

1.2 Namen, cilji in osnovne trditve 10

1.3 Predpostavke in omejitve 11

1.4 Uporabljene raziskovalne metode 12

2 ALGORITMI ISKANJA NAJKRAJŠE POTI 13

3 GRAFI IN MREŽE 15

3.1 Graf 15

3.2 Mreže 17

3.3 Hevristika 20

4 ALGORITEM A* 22

4.1 Prednosti algoritma A* 22

4.2 Slabosti algoritma A* 23

4.3 Psevdokoda in načrt implementacije algoritma A* 24

5 ALGORITEM BREADTH-FIRST SEARCH 26

5.1 Prednosti algoritma BFS 26

5.2 Slabosti algoritma BFS 27

5.3 Psevdokoda in načrt implementacije algoritma BFS 28

6 DIJKSTROV ALGORITEM 30

6.1 Prednost Dijkstrovega algoritma 31

6.2 Slabost Dijkstrovega algoritma 31

6.3 Psevdokoda in načrt implementacije Dijkstrovega algoritma 32

7 UMETNA INTELIGENCA 34

7.1 Umetna inteligenca in razumevanje 35

7.2 Umetna inteligenca in strojno učenje 35

7.3 Umetna inteligenca in iskanje 36

7.4 Umetna inteligenca in iskanje najkrajše poti 37

7.5 Stroški razvoja in implementacije umetne inteligence 38

8 PRIPRAVA OKOLJA 40

8.1 Struktura izvorne kode 40

8.2 Uporabniški vmesnik aplikacije 42
8.3 Izvorna koda labirinta 49
8.4 Izvorna koda algoritma BFS 51
8.5 Izvorna koda algoritma A* 53
8.6 Izvorna koda algoritma Dijkstra 54
8.7 Izvorna koda strojnega učenja 55

9 PRIMERJAVA ALGORITMOV IN UMETNE INTELIGENCE V NAKLJUČNO GENERIRANEM LABIRINTU 69

9.1 Rezultati algoritmov iskanja poti in umetne inteligence v naključno generiranem labirintu 69
9.2 Analiza hitrosti algoritmov iskanja najkrajše poti 71
9.3 Analiza zmogljivosti umetne inteligence pri reševanju najkrajše poti 73

10 POTRDITEV ALI ZAVRNITEV HIPOTEZ 74

10.1 H1: A* je najpopularnejši algoritem 74
10.2 H2: Obstaja umetna inteligenca, ki najde najkrajšo pot tudi iz takega algoritma, pri katerem imajo ostali algoritmi omejitve 74
10.3 H3: A* algoritem bo najhitrejši algoritem in bolj učinkovit pri iskanju najkrajše poti v naključno generiranih labirintih v primerjavi z drugimi algoritmi 74
10.4 H4: Umetna inteligenca bo najučinkovitejša pri iskanju najkrajše poti v naključno generiranem labirintu v primerjavi z algoritmi A*, Dijkstrovim algoritmom in BFS 75

11 SKLEP 76

12 VIRI IN LITERATURA 78

KAZALO SLIK

SLIKA 1 USMERJEN GRAF 16
SLIKA 2 USMERJEN GRAF 16
SLIKA 3 OBTEŽEN GRAF 17
SLIKA 4 PRIKAZ MREŽE 18
SLIKA 5 PRIKAZ ŠESTEROKOTNE MREŽE 19
SLIKA 6 PRIKAZ PREPROSTE MREŽE 20
SLIKA 7 PSEUDO KODA ALGORITMA A STAR 25
SLIKA 8 PSEUDO KODA ALGORITMA BFS 29
SLIKA 9 PSEUDO KODA ALGORITMA DIJKSTRA 33
SLIKA 10 STRUKTURA IZVORNE KODE 41
SLIKA 11 UPORABNIŠKI VMESNIK APLIKACIJE 42
SLIKA 12 PRIKAZ ISKANJA ALGORTIMA V APLIKACIJI 43
SLIKA 13 IZVORNA KODA RAZREDA APP.PY 43
SLIKA 14 IZVORNA KODA RAZREDA APP.PY 44
SLIKA 15 IZVORNA KODA METODE START() RAZREDA APP.PY 44
SLIKA 16 IZVORNA KODA METODE UPDATE_TABLE() RAZREDA APP.PY 44
SLIKA 17 IZVORNA KODA DRAW_CELL() RAZREDA APP.PY 45
SLIKA 18 IZVORNA KODA METODE CREATE_WIDGETS() RAZREDA APP.PY 45
SLIKA 19 IZVORNA KODA METODE RUN_MAZE() RAZREDA APP.PY 46
SLIKA 20 IZVORNA KODA OBJEKTA ALGORITHMS 47
SLIKA 21 IZVORNA KODA METODE DRAW_MAZE() RAZREDA APP.PY 48
SLIKA 22 IZVORNA KODA RAZREDA APP.PY 48
SLIKA 23 IZVORNA KODA RAZREDA MAZE.PY 50
SLIKA 24 IZVORNA KODA ALGORITMA BFS 52
SLIKA 25 IZVORNA KODA ALGORITMA A STAR 53
SLIKA 26 IZVORNA KODA ALGORITMA DIJKSTRA 55
SLIKA 27 IZVORNA KODA ZA STROJNO UČENJE 56
SLIKA 28 IZVORNA KODA RAZREDA DQN 56
SLIKA 29 IZVORNA KODA RAZREDA MAZEENV 57
SLIKA 30 IZVORNA KODA RAZREDA MAZEENV 58
SLIKA 31 KONSTRUKTOR RAZREDA SOLVINGMAZEENV 59
SLIKA 32 IZVORNA KODA METODE _GET_STATE() 59

SLIKA 33 IZVORNA KODA METODE RESET() 60
SLIKA 34 IZVORNA KODA METODE STEP() 60
SLIKA 35 IZVORNA KODA METODE SOLVE_WITH_DQN() 61
SLIKA 36 IZVRONA KODA METODE SOLVE_WITH_DQN() 61
SLIKA 37 IZVRONA KODA METODE SOLVE_WITH_DQN() 62
SLIKA 38 IZVRONA KODA METODE SOLVE_WITH_DQN() 62
SLIKA 39 IZVRONA KODA METODE SOLVE_WITH_DQN() 63
SLIKA 40 IZVORNA KODA METODE TRAIN_DQN() 64
SLIKA 41 IZVORNA KODA METODE TRAIN_DQN() 65
SLIKA 42 IZVORNA KODA METODE TRAIN_DQN() 65
SLIKA 43 IZVORNA KODA METODE TRAIN_DQN() 66
SLIKA 44 IZVORNA KODA METODE TRAIN_DQN() 66
SLIKA 45 IZVORNA KODA METODE TRAIN_DQN() 67
SLIKA 46 IZVORNA KODA METODE TRAIN_DQN() 67
SLIKA 47 IZVORNA KODA METODE TRAIN_DQN() 68

KAZALO TABEL

TABELA 1 REZULTATI ALGORITMOV ISKANJA POTI IN UI V LABIRINTU VELIKOSTI 11X11 69
TABELA 2 REZULTATI ALGORITMOV ISKANJA POTI IN UI V LABIRINTU VELIKOSTI 21X21 70
TABELA 3 REZULTATI ALGORITMOV ISKANJA POTI IN UI V LABIRINTU VELIKOSTI 31X31 70

KAZALO GRAFIKONOV

GRAFIKON 1 PRIKAZ ČASA ISKANJA KONČNEGA VOZLIŠČA ALGORITMOV IN UI V LABIRINTU 72
GRAFIKON 2 PRIKAZ ŠTEVILA OBISKANIH VOZLIŠČ ALGORITMOV IN UI V LABIRINTU 72
GRAFIKON 3 PRIKAZ USPEŠNOSTI ISKANJA ALGORITMOV IN UI V LABIRINTU 73

1 UVOD

V dobi napredne računalniške tehnologije in umetne inteligence je iskanje optimalnih poti v kompleksnih strukturah postalo ključno za reševanje številnih problemov. Od navigacijskih sistemov, robotike in videoiger se algoritmi iskanja poti nenehno razvijajo in izpopolnjujejo. Labirint kot abstraktna struktura predstavlja idealno okolje za primerjavo različnih pristopov, saj omogoča kontrolirano analizo učinkovitosti algoritmov iskanja poti in umetne inteligence. Naključno generirani labirinti zagotavljajo nepristranske pogoje za ovrednotenje delovanja algoritmov v nepredvidljivih okoljih.

1.1 Opis področja in opredelitev problema

Cormen idr. (2022) definirajo algoritem kot računalniški postopek, ki vzame poljubno vrednost ali niz vrednosti kot vhod in ustvari izhod ali niz vrednosti v določenem času. Torej gre za zaporedje računalniških postopkov, ki pretvorijo vhod v izhod.

Algoritem je orodje, ki opisuje postopek reševanja računalniških problemov. Določa želeno razmerje med vhodi in izhodi. Definiran je kot računalniški program, ki vsebuje postopek za proceduro (Cormen, Leiserson, Rivest in Clifford, 2022).

Algoritem je pravilen, če za vsak problem, ki ga dobimo kot vhodni podatek, ustvari in konča računanje v končnem času in poda pravilen odgovor. Pravilen algoritem reši trenutni problem. Nepravilen algoritem lahko poteka neskončno ali poda nepravilen rezultat (Cormen, Leiserson, Rivest, in Clifford, 2022).

Računalniški viri so zaradi številnih dejavnikov omejeni. Učinkovitost algoritmov merimo z uporabo računalniških virov. Učinkovit algoritem optimalno uporabi časovne in prostorske vire. Prostorska učinkovitost se na primer uporablja za merjenje količine pomnilnika, ki je potrebna za izvajanje algoritma. Hitreje kot algoritem konvergira, večja je natančnost algoritma in manjše je število iteracij. Na učinkovitost algoritma vplivajo tudi pomnilniške zahteve programa. S povečevanjem števila iteracij lahko poraba pomnilnika poslabša učinkovitost sistema (Choudhury, Ghose, Islam in Yogita, 2024).

Za učinkovite algoritme so potrebne podatkovne strukture. Te nam omogočajo shranjevanje podatkov za lažji dostop in spreminjanje le-teh. Podatkovna struktura ne deluje dobro za vse

namene, zato je njihova ustrezna izbira ključna za optimalno delovanje algoritmov (Cormen, Leiserson, Rivest in Clifford, 2022).

Internet omogoča ljudem po vsem svetu hiter dostop in iskanje velike količine informacij. S pomočjo algoritmov spletna mesta upravljajo velike količine podatkov. Primer uporabe algoritmov lahko najdemo v iskalnikih, kot je iskalnik Google (angl. Google search engine), in v iskanju najkrajše poti do cilja (Cormen, Leiserson, Rivest in Clifford , 2022).

V številnih primerih želimo primerjati dva algoritma. Na splošno lahko učinkovitost algoritma ocenimo glede na čas izvajanja kot funkcijo velikosti vhodnih podatkov. Ko govorimo o učinkovitosti algoritma vedno upoštevamo najslabši možni rezultat. Za učinkovitost algoritmov uporabljamo notacijo Big O.

Zapis Big O je matematični zapis, ki opisuje mejno obnašanje funkcije, ko argument teži k določeni vrednosti ali neskončnosti. V računalništvu se zapis velikega O uporablja za razvrščanje algoritmov glede na to, kako se njihov čas izvajanja ali prostorske zahteve povečujejo z rastjo velikosti vhoda (Cormen, Leiserson, Rivest in Clifford, 2022).

V računalništvu se zapis vrstnega reda uporablja predvsem za primerjavo učinkovitosti algoritmov. Pri analizi učinkovitosti algoritmov je še posebej uporaben zapis Big O. V tem primeru je n velikost vhoda, f (n) pa je čas delovanja algoritma glede na velikost vhoda.

V diplomskem delu se bomo osredotočili na primerjavo algoritmov iskanja poti in umetne inteligence v naključno generiranem labirintu. Cilj diplomskega dela je primerjava algoritmov in umetne inteligence ter ugotovitev, kateri pristop k reševanju problema je najhitrejši in optimalen.

Rezultati diplomskega dela bodo pomagali razvijalcem programske opreme pri izbiri primernih algoritmov iskanja poti za njihove aplikacije. Diplomsko delo bo prispevalo k boljšemu razumevanju prednosti in omejitev algoritmov iskanja poti in umetne inteligence pri reševanju problemov navigacije in iskanja poti v kompleksnih strukturah.

1.2 Namen, cilji in osnovne trditve

Namen diplomskega dela je pregled obstoječe literature na področju algoritmov iskanja poti in primerjava učinkovitosti iskanja poti algoritmov A star, Dijkstrovega algoritma, algoritma preiskovanja v širino in umetne inteligence v naključno generiranem labirintu. Primerjava bo

temeljila na več kriterijih, vključno s časom in uspešnostjo reševanja ter številom obiskanih vozlišč.

Cilji diplomskega dela so naslednji, in sicer:

- Ugotoviti, ali je algoritem A star najpopularnejši algoritem za iskanje poti.
- Primerjati algoritme iskanja poti in umetne inteligence v naključno generiranem labirintu.
- Ugotoviti, ali obstaja umetna inteligenca za iskanje najkrajše poti v naključno generiranem labirintu.
- Ugotoviti, ali je umetna inteligenca najučinkovitejša pri iskanju poti v naključno generiranem labirintu v primerjavi z algoritmi A*, Dijkstrovim algoritmom in algoritmom BFS.

Hipoteze, ki jih bo diplomsko delo preverilo, so naslednje, in sicer:

Hipoteza 1: A* je najpopularnejši algoritem.

Hipoteza 2: Obstaja umetna inteligenca, ki najde najkrajšo pot tudi iz takega labirinta, pri katerem imajo ostali algoritmi omejitve.

Hipoteza 3: A* algoritem bo hitrejši in bolj učinkovit pri iskanju najkrajše poti v naključno generiranih labirintih v primerjavi z drugimi algoritmi.

Hipoteza 4: Umetna inteligenca bo najučinkovitejša pri iskanju najkrajše poti v naključno generiranem labirintu v primerjavi z algoritmi A*, Dijkstrovim algoritmom in BFS.

1.3 Predpostavke in omejitve

V diplomskem delu se soočamo s predpostavkami in omejitvami, ki vplivajo na rezultate.
Predpostavke:

- Predpostavljamo, da so vse meritve algoritmov in umetne inteligence izvedene na naključno generiranih labirintih velikosti 11x11, 21x21 in 31x31. To omogoča, da so testi nepristranski in objektivni pri analizi hitrosti ter učinkovitosti algoritmov in umetne inteligence.
- Predpostavljamo, da so vse informacije pridobljene iz akreditiranih virov informacij in so relevantne za to področje raziskave.

Omejitve:

- V diplomskem delu bomo uporabili strojno učenje za simulacijo umetne inteligence. Učenje bo izvedeno na labirintu velikosti 11x11, kar bo vplivalo na rezultate umetne inteligence.
- Rezultati analize so odvisni od specifičnih testov in meritev, ki se izvajajo. Drugačne metode analize bi lahko privedle do drugačnih rezultatov.
- Algoritem A* je odvisen od hevristike, boljša kot je ta, boljši so rezultati. V diplomskem delu ne bomo obravnavali različnih možnosti vpliva hevristike na algoritem A*.

1.4 Uporabljene raziskovalne metode

V diplomskem delu smo uporabili več raziskovalnih metod, ki so pomagale pri pridobitvi podatkov ter primerjavi algoritmov iskanja poti in umetne inteligence. Te metode so:

- Pregled literature: Izvedli smo pregled trenutno znane znanstvene literature z namenom pridobitve verodostojnih in preverjenih informacij. Pregled literature je pomagal razumeti in primerjati algoritme iskanja poti in umetno inteligenco.
- Eksperimentalno testiranje: V okviru diplomskega dela smo razvili aplikacijo v programskem jeziku Python. Aplikacija nam je omogočala primerjavo in analizo algoritmov iskanja poti in umetne inteligence v naključno generiranem labirintu. To nam je omogočalo izvesti analizo učinkovitosti in hitrosti algoritmov in umetne inteligence na labirintih različnih velikosti. Večanje labirinta nam je omogočalo simulacijo kompleksnejših problemov, in s tem ugotoviti, kako se obnašajo algoritmi in umetna inteligenca pri reševanju le-teh.
- Analiza: S pomočjo trenutno znane znanstvene raziskave in rezultatov eksperimentalnega testiranja smo lahko odgovorili na zastavljena vprašanja v okviru diplomske naloge.

2 ALGORITMI ISKANJA NAJKRAJŠE POTI

Ljudje uporabljajo zemljevide za veliko stvari, na primer za iskanje krajev, restavracij, bencinskih črpalk ali iskanje poti do želenega cilja. Ko zahtevamo pot od začetne točke do končnega cilja, vedno dobimo najkrajšo pot. Problem najkrajše poti se preučuje že vrsto let. Problem najkrajše poti je problem, ki najde najmanjšo razdaljo ali pot med vozlišči ali vrhovi v grafu. Graf je abstraktni matematični objekt, ki vsebuje množice vrhov in robov (Kairanbay in Jani, 2013), (Rachmawati in Gustin, 2020).

Algoritmi za iskanje poti se uporabljajo za reševanje problema najkrajše in optimalne poti. Običajno se uporabljata algoritma A* in Dijkstra kot metoda rešitve za iskanje najkrajše poti. Iskanje poti je izrisovanje vozlišč. Cilj algoritma je iskanje najkrajše poti med dvema točkama od začetka do cilja. Iskanje poti je glavna sestavina številnih pomembnih aplikacij na področjih videoiger, robotike (Hunkeler, Schär, Dornberger in Hanne, 2016), simulacije množic (Wolsey, 1998) in GPS (Carr, 2014) (Rafiq, Tuty Asmawaty in Ihsan, Pathfinding Algorithms in Game Development, 2020).

Iskalni algoritmi so že dolgo časa v središču zanimanja na področju računalništva. Iskanja morajo biti hitra, natančna in učinkovita, vsako odstopanje od teh lastnosti pa se šteje za veliko napako. Iskanje lahko delimo na informirano in neinformirano. Najpogostejše iskanje je informirano. Pri informiranem iskanju se uporablja hevristično funkcijo, ki meri oddaljenost od cilja za sprejemanje boljših odločitev (Foead , Ghifari, Kusuma, Hanafiah in Gunawan , 2021).

Iskanje optimalne poti je sicer zaželeno, vendar ni vedno nujno, odvisno od končne uporabe. V nekaterih primerih, na primer pri sistemih za upravljanje GPS v realnem času, bodo uporabniki bolj cenili takojšen in hiter odziv kot vedno optimalno pot (Aria, 2018), (Foead , Ghifari, Kusuma, Hanafiah in Gunawan , 2021).

Informirana iskanja, kot so IDA*, A* in Jump Point Search IDA*, običajno uporabljajo nekatere zunanje podatke za povečanje učinkovitosti in temeljijo na zmožnosti pretvorbe ciljev v podatke. Hevristične funkcije delujejo izjemno dobro, kadar je iskalno območje dobro znano, na primer zemljevid, vendar lahko trpijo, če so podatki netočni ali neznani (Foead , Ghifari, Kusuma, Hanafiah in Gunawan , 2021).

Nasprotno pa neinformirana iskanja slepo sledijo svojemu algoritmu do zaključka, kar jih običajno naredi počasnejša, a manj odvisna od zunanjih dejavnikov. Na primer v igrah se je neinformirano iskanje, kot je Dijkstrov algoritem, izkazalo za izjemno neučinkovito v primerjavi s HPA*, saj je optimalno pot našlo skoraj trikrat počasneje (Noori in Moradi, 2015), (Foead , Ghifari, Kusuma, Hanafiah in Gunawan , 2021).

3 GRAFI IN MREŽE

Algoritmi iskanja so predstavljeni s pomočjo grafov in mrež. V matematiki in računalništvu je teorija grafov študija grafov, ki so matematične strukture, uporabljene za modeliranje parnih odnosov med objekti.

3.1 Graf

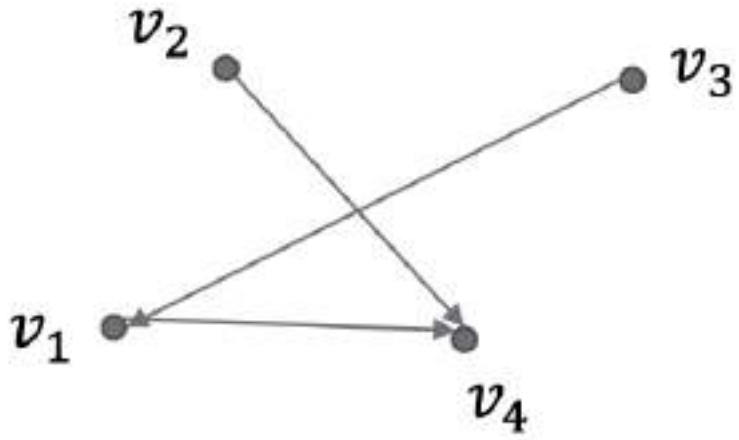
Graf je sestavljen iz vrhov, imenovanih tudi vozlišča ali točke, ki so povezani z robovi, imenovanimi tudi loki, povezave ali črte. Razlikujemo med neusmerjenimi grafi, kjer robovi simetrično povezujejo dva vrhova, in usmerjenimi grafi, kjer robovi asimetrično povezujejo dva vrhova.

V enem omejenem, a zelo pogostem pomenu izraza je graf urejen par $G = (V, E)$, ki vsebuje:

- V - množico vozlišč;
- E - $E \subseteq \{\{x, y\} \mid x, y \in V \text{ in } x \neq y\}$, množico povezav, imenovanih tudi robovi ali črte, kjer so povezave neurejeni pari vozlišč, torej je povezava določena z dvema različnima vozliščema (Bender in Williamson, 2010), (Berge, 1958).

Graf je lahko usmerjen ali neusmerjen. V primeru usmerjene povezave velja, da:

$$\{u, v\} \neq \{v, u\}.$$

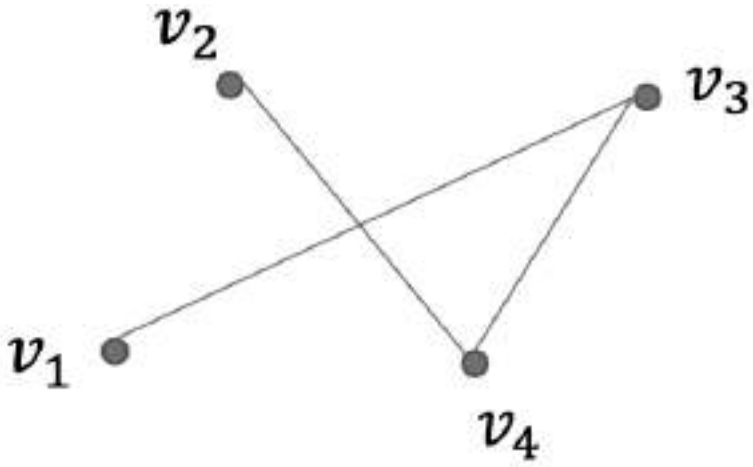


Slika 1: Usmerjen graf

Vir: Diskretna matematika: teorija grafov. Višja strokovna šola Academia Maribor, 2024.

V primeru neusmerjene povezave velja, da:

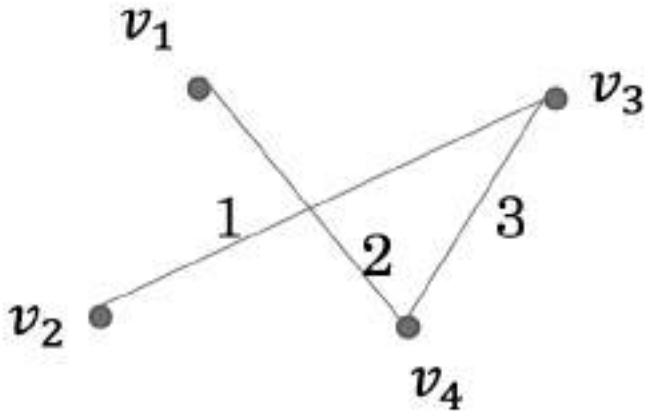
$$\{u, v\} = \{v, u\}.$$



Slika 2: Usmerjen graf

Vir: Diskretna matematika: teorija grafov. Višja strokovna šola Academia Maribor, 2024.

Obtežen graf G je graf, kjer vsaki povezavi dodamo zahtevnost oziroma težo. Zapišemo $G = (V, E, C)$, kjer je $C(G)$ množica obtežitev povezav grafa.



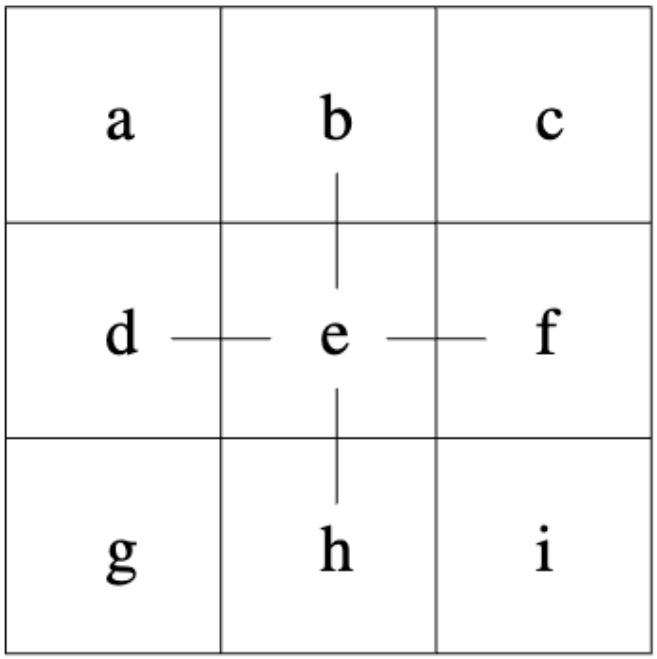
Slika 3: Obtežen graf

Vir: Diskretna matematika: teorija grafov. Višja strokovna šola Academia Maribor, 2024.

3.2 Mreže

Algoritme iskanja optimalne poti najlažje predstavimo na mrežnih površinah. Da lažje primerjamo pot in učinkovitost algoritma, je potrebno razumeti mrežne površine.

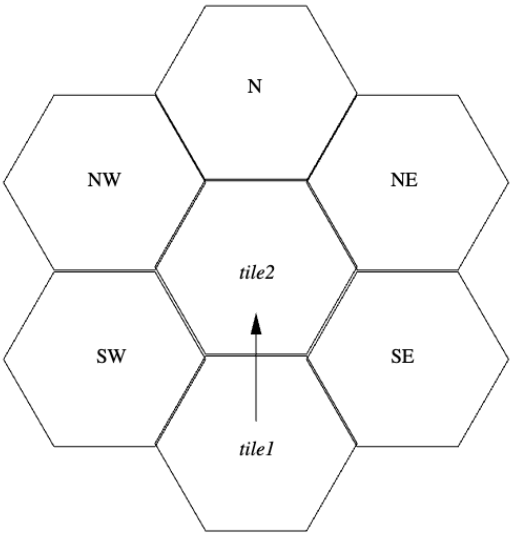
Iskanje poti je pomemben problem za številne aplikacije, vključno z omrežnim prometom, načrtovanjem robotov, vojaškimi simulacijami in računalniškimi igrami. Najpogostejši scenarij je uporaba mreže ploščic. Ploščica ima štiri sosednja vozlišča (b = 4). Zato je treba pri iskanju poti upoštevati štiri sosednje ploščice, ki jih je treba raziskati. Ker se nikoli ne vrnemo nazaj vzdolž optimalne poti, ni treba upoštevati smeri, ki smo jo že obiskali. Zato je število vozlišč, ki jih je treba raziskati tri (b = 3) (Yap, 2002).



Slika 4: Prikaz mreže

Vir: Yap, P. Grid-Based Path-Finding: lecture notes in computer science, 2002.

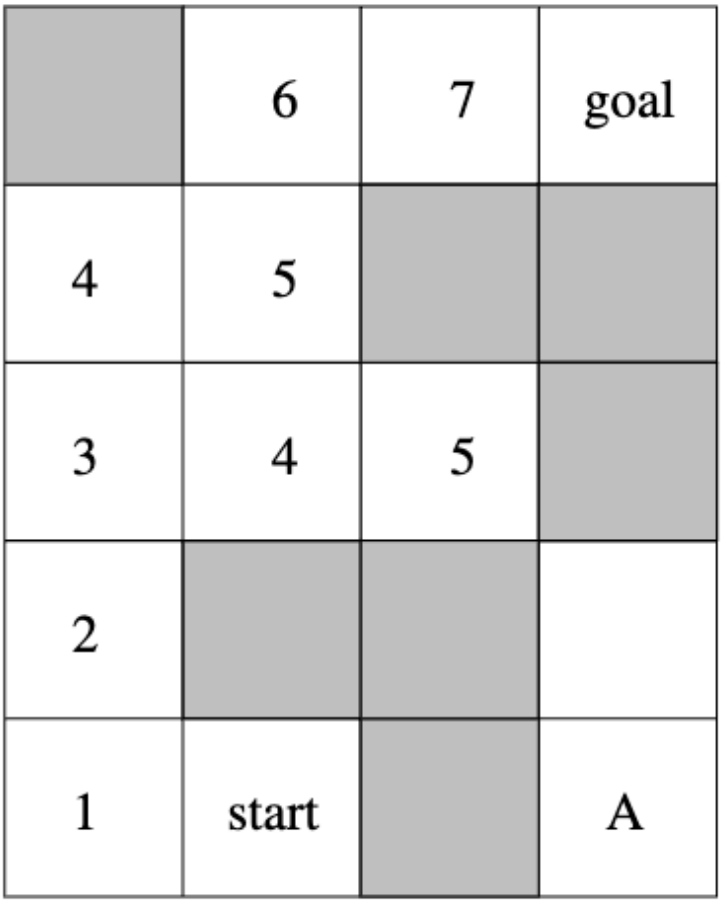
Oglejmo si šesterokotno mrežo, kjer ima vsako vozlišče šest možnih smeri gibanja, in sicer sever, severovzhod, jugovzhod, jug, jugozahod in severozahod. Na prvi pogled bi lahko sklepali, da ima takšna mreža faktor razvejanosti enak 5, saj na vsakem koraku obstaja pet možnih poti naprej, če ne štejemo poti nazaj. Vendar pa lahko to število zmanjšamo z upoštevanjem optimalnega iskalnega algoritma. Recimo, da se premaknemo iz ploščice 1 na ploščico 2 v smeri sever. Ko razmišljamo o nadaljnjih korakih z nove pozicije, upoštevamo samo tista gibanja, ki so smiselna v kontekstu iskanja najkrajše poti. Gibanja nazaj ne štejemo, prav tako izločimo smeri SV in SZ, ker bi v primeru optimalne poti te smeri izhajale iz drugih odločitev že prej. Tako na vsaki ploščici ostanejo le tri smiselne možnosti za nadaljnje gibanje. Zaradi tega je efektivni faktor razvejanosti šesterokotne mreže zmanjšan na tri (b = 3) (Yap, 2002).



Slika 5: Prikaz šesterokotne mreže

Vir: Yap, P. Grid-Based Path-Finding: lecture notes in computer science, 2002.

Za diplomsko delo bomo uporabili preprosto mrežo, ki omogoča štirismerno premikanje. Mreža bo sestavljena iz belih in sivih kvadratov. Siv kvadrat bo predstavljal oviro, ki je algoritem ne sme prečkati. Vsak kvadrat predstavlja svoje vozlišče.



Slika 6: Prikaz preproste mreže

Vir: Yap, P. Grid-Based Path-Finding: lecture notes in computer science, 2002.

3.3 Hevristika

Običajno se pri iskanju poti uporablja hevristika. Hevristika je nekaj, kar daje grobo oceno, kako daleč je do cilja. Običajno se neposredno uporablja realna evklidska razdalja do cilja, saj je ta hitra in običajno daje razmeroma dobre rezultate. Težava je v tem, da v primerih, ki niso samo odprt prostor z nekaj majhnimi ovirami, raztresenimi naokoli, ta ocena ni preveč dobra. Posledica tega je, da gre iskanje skozi veliko več vozlišč, kot bi bilo potrebno, če bi bila hevristika boljša.

Hevristična funkcija h(n) podaja ocenjeni strošek od trenutnega vozlišča do ciljnega vozlišča, deluje kot informirano ugibanje algoritma o preostali poti.

V mrežnih ali kartografskih problemih se uporablja Manhattanska in Evklidska funkcija za računanje razdalje od začetnega do končnega vozlišča.

Manhattansko razdaljo lahko izračunamo z naslednjo funkcijo:

$h(n) = |x_1 - x_2| + |y_1 - y_2|$.

Evklidsko razdaljo lahko izračunamo z naslednjo funkcijo:

$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Končno oceno poti lahko naredimo s funkcijo:

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

Kjer je:

$f(n)$ - ocenjevalna funkcija,

$g(n)$ - razdalja od začetnega vozlišča do trenutnega vozlišča,

$h(n)$ - hevristika od vozlišča n.

4 ALGORITEM A*

Algoritem A* je preizkušena metoda, ki se uporablja kot osnova za reševanje problemov iskanja poti. Običajno se algoritem A* uporablja za splošne in strateške igre, njegova stabilnost in hitrost pa sta se v zadnjem desetletju dodatno izboljšali (Foead , Ghifari, Kusuma, Hanafiah in Gunawan, 2021).

Foead idr. (2021) navajajo, da se algoritem že dolgo uporablja v raziskovalni skupnosti za iskanje poti. Njegova učinkovitost, preprostost in modularnost so pogosto poudarjene kot prednosti v primerjavi z drugimi algoritmi. Zaradi svoje vsesplošne in razširjene uporabe je A* postal standard za raziskovalce, ki poskušajo rešiti probleme iskanja poti. Vendar ni zanesljiv, saj v številnih primerih potrebuje dodaten algoritem ali spremembo svojih osnovnih funkcij, da lahko opravi te zapletene naloge. Algoritem ni optimalen pri Multi-agent pathfinding problemih, saj se srečuje s številnimi ovirami, kot so nasprotujoče si poti med agenti (Felner idr., 2018).

Na splošno je A* običajno dosleden pri reševanju različnih problemov, vendar zahteva prilagoditev, da doseže najboljše rezultate. Kljub temu je še vedno izjemno učinkovit pri širokem naboru iskalnih nalog, dokler njegova glavna slabost, odvisnost od hevristične funkcije, ni bistvena ali ne predstavlja težave (Foead , Ghifari, Kusuma, Hanafiah in Gunawan, 2021).

*4.1 Prednosti algoritma A**

Pomemben dejavnik iskanja z algoritmom A* je njegova učinkovitost. Medtem ko lahko večina dobro zasnovanih algoritmov za iskanje poti najde rešitev, bodo mnogi pri tem porabili več časa, virov ali obojega v primerjavi z A*. Razlika v splošni učinkovitosti lahko znaša več kot 40 %, v nekaterih primerih pa do 30 % (Barnouti, Al-Dabbagh in Naser, 2016), (Foead , Ghifari, Kusuma, Hanafiah in Gunawan, 2021).

Algoritem A* lahko najde pot veliko hitreje kot neinformirano iskanje, vendar ne zagotavlja, da bo rezultat najkrajša pot. Raziskava je pokazala, da bo algoritem A* v nekaterih primerih strateških zemljevidov in labirintov pokazal le 85 % časa rezultat, pri katerem najde najkrajšo pot (Barnouti, Al-Dabbagh in Naser, 2016).

Poleg tega je A* izjemno modularen in ga je mogoče prilagoditi širokemu spektru potreb. Vendar pa niso vse prilagoditve nujno izboljšave, nekatere optimizacije, ki dajejo prednost hitrosti, lahko vodijo do tega, da ne najdejo najboljše poti. To je pokazal eksperiment z 29 vozlišči in 4 križišči, v katerem so primerjali algoritme HPA*, A*, Dijkstra in IDA*. HPA* sicer ni našel najhitrejše poti, a je zaključil izračun izjemno hitro, medtem ko so bili drugi algoritmi počasnejši, vendar so dosegli optimalen rezultat.

Algoritem A* je samostojno še vedno bolje opravil kot Dijkstrov algoritem, saj je optimalno pot našel dvakrat hitreje. Ta primer jasno prikazuje potencialno povečano učinkovitost hevristične funkcije (Foead, Ghifari, Kusuma, Hanafiah in Gunawan , 2021).

A* lahko najde optimalne in skoraj optimalne rešitve učinkoviteje, tako da usmeri iskanje k cilju s pomočjo hevurističnih funkcij, s čimer se bistveno zmanjša njegova časovna zahtevnost (Soltani, Tawfik, Goulermas in Fernando, 2002).

*4.2 Slabosti algoritma A**

Ena glavnih slabosti algoritma A* je njegova slabša zmogljivost pri dvosmernem iskanju. V eksperimentu o dvosmernem iskanju v grafih so raziskovalci ugotovili, da se je dvosmerni A* odrezal zelo slabo v primerjavi z dvosmernim BFS. Še posebej se je BFS izkazal pri večjih mrežah velikosti 16×16, medtem ko je A* včasih imel prednost pri manjših mrežah velikosti 8×8 (Kumar, 2019), (Foead, Ghifari, Kusuma, Hanafiah in Gunawan, 2021).

Na splošno tradicionalni osnovni algoritem A* ne more slediti vse večjim zahtevam iskanja poti. Vendar pa lahko s pravilnimi prilagoditvami in izboljšavami še vedno konkurira drugim algoritmom.

Zaradi ogromnega števila različnih situacij pri iskanju poti ni mogoče razviti univerzalne rešitve, ki bi delovala v vseh primerih. Jasno je, da klasični A* postopoma izgublja priljubljenost pri reševanju kompleksnih problemov, medtem ko njegove izboljšane različice še vedno dosegajo visoko hitrost in večjo učinkovitost. Prihodnji razvoj algoritma A* bo moral vključevati prilagoditve, kot je uporaba zgoščevalnih tabel (angl. hash tables) ali zanašanje na druge algoritme za natančnejše hevristike, saj te spremembe neposredno odpravljajo njegove slabosti (Foead, Ghifari, Kusuma, Hanafiah in Gunawan, 2021).

Uspešnost in učinkovitost A* iskalnega algoritma je močno odvisna od kakovosti hevristične funkcije. Zato postane oblikovanje optimalne hevristične funkcije glavni cilj pri razvoju iskalnega algoritma (Yiu, Du in Mahapatra, 2018).

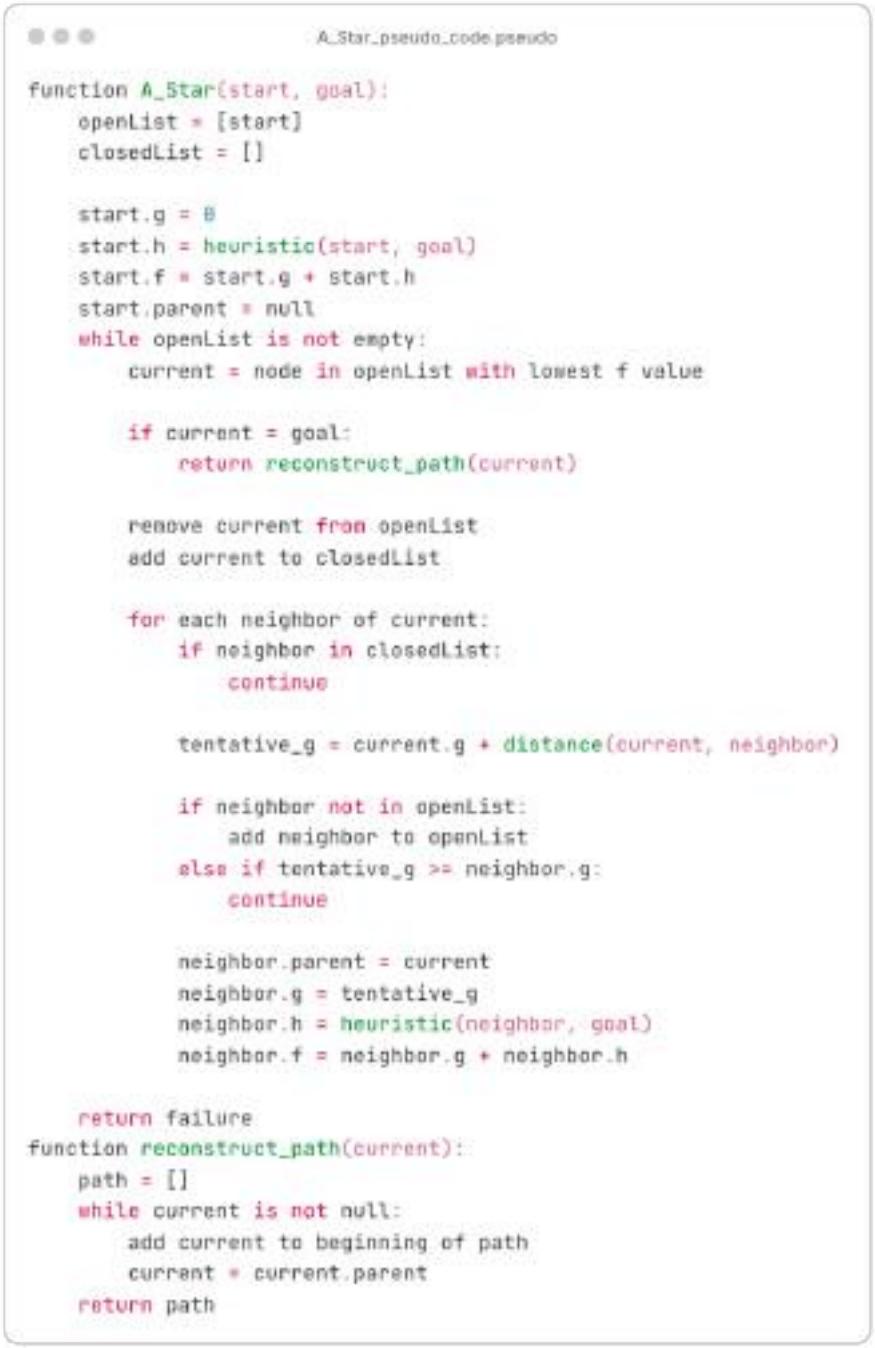
*4.3 Psevdokoda in načrt implementacije algoritma A**

Algoritem A* uporablja dve funkciji, in sicer funkcijo razdalje g(n) in hevristično funkcijo h(n). Ti sta ključni za oceno uspešnosti algoritma.

Končno oceno poti lahko naredimo s funkcijo:

$f(n) = g(n) + h(n)$.
Kjer je:
$f(n)$ - ocenjevalna funkcija,
$g(n)$ - razdalja od začetnega vozlišča do trenutnega vozlišča,
$h(n)$ - hevristika od vozlišča n.

Algoritem implementiramo s pomočjo psevdokode, zapisane v programskem jeziku Python.



Slika 7: Psevdokoda algoritma A star

Vir: https://www.datacamp.com/tutorial/a-star-algorithm, 2025

5 ALGORITEM BREADTH-FIRST SEARCH

Algoritem preiskovanja v širino (angl. Breadth-First Search) je algoritem za iskanje vozlišča v drevesni podatkovni strukturi, ki izpolnjuje določen pogoj. Začne pri korenu drevesa in preišče vsa vozlišča na trenutni globini, preden preide na vozlišča na naslednji globinski ravni. Za shranjevanje otroških vozlišč, ki so bila odkrita, a še niso raziskana, uporablja dodaten pomnilnik, običajno vrsto (angl. queue). Preiskovanje v širino lahko posplošimo na neusmerjene in usmerjene grafe z določenim začetnim vozliščem.

Preiskovanje v širino je pomemben gradnik mnogih algoritmov na grafih. Pogosto se uporablja za preverjanje povezanosti ali izračun najkrajših poti z enim virom v neuteženih grafih (Beamer, Asanović in Patterson, 2013).

Algoritmi za grafe (angl. Graph algorithms) postajajo vse pomembnejši. V velikih računalniških sistemih se izvajajo algoritmi za analizo ogromnih količin podatkov. Na mobilnih aplikacijah so algoritmi za grafe uporabljeni za strojno učenje. Žal so aplikacije pogosto omejene s skupnim pomnilnikom (angl. shared-memory systems). Iskanje po širini, ki je pomemben gradnik številnih drugih grafnih algoritmov, ima nizko računsko zmogljivost, kar še poslabša pomanjkanje lociranosti in posledično nizko skupno zmogljivost. Za pospešitev BFS je bilo veliko predhodnega dela, kjer so spreminjali algoritme in podatkovne strukture, v nekaterih primerih tudi z dodajanjem dodatnega računalniškega dela, da bi izboljšali lokalnost in povečali splošno zmogljivost. Vendar pa nobena od teh metod ni poskušala zmanjšati števila pregledanih povezav. Da bi pospešili BFS, je bilo v preteklosti opravljeno veliko dela za spremembo algoritma in podatkovnih struktur, v nekaterih primerih z dodatnim računskim delom, da bi povečali lokacijo in skupno zmogljivost (Agarwal, Petrini, Pasetto in Bader, 2010), (Buluç in Madduri, 2011), (Hong, Ogunteb in Olukotun, 2011), (Yoo in drugi, 2005), (Beamer, Asanović in Patterson, 2013).

5.1 Prednosti algoritma BFS

Delovanje BFS se začne pri izvorni točki, nato pa se iskalna meja postopoma širi navzven, pri čemer na vsakem koraku obišče vsa vozlišča na isti globinski ravni, preden preide na globljo raven (Beamer, Asanović in Patterson, 2013).

Pri klasičnem pristopu od zgoraj navzdol vsako vozlišče preveri vse svoje sosednje točke, da ugotovi, ali so še neobiskane. Vsaka neobiskovana točka se doda v iskalno mejo in se označi kot obiskana tako, da se nastavi spremenljivka starša. Ta algoritem ustvari BFS drevo, ki pokriva povezano komponento z izhodiščnim vozliščem (Beamer, Asanović in Patterson, 2013).

Večina računske obremenitve pri iskanju v širino je preverjanje povezav na sosednji točki, da se ugotovi, ali je ciljno vozlišče že bilo obiskano. Skupno število preverjanj povezav v klasičnem algoritmu od zgoraj navzdol (angl. top-down) je enako številu povezav v povezani komponenti, ki vsebuje izvorno vozlišče, saj se pri vsakem koraku preveri vsaka povezava na vozlišču (Beamer, Asanović in Patterson, 2013).

Pristop od spodaj navzgor (angl. bottom-up) odpravi potrebo po nekaterih operacijah v paralelni implementaciji. Pri pristopu od zgoraj navzdol (angl. top-down) bi lahko več niti hkrati pisalo v istega otroka, zato so potrebne atomske operacije za zagotovitev medsebojne izključitve. Pri pristopu od spodaj navzgor pa piše otrok sam vase, s čimer se odpravi vsakršno tekmovanje. Ta prednost, skupaj z morebitnim zmanjšanjem števila pregledanih povezav, pride na račun serijske obdelave dela za posamezno vozlišče, vendar še vedno obstaja velika mera paralelizma med deli za različna vozlišča. Pristop od spodaj navzgor je prednosten, ko je velik delež vozlišč v fronti, vendar povzroči več dela, če je fronta majhna (Beamer, Asanović in Patterson, 2013).

Zato mora učinkovita implementacija iskanja v širino združevati tako pristop od zgoraj navzdol kot tudi pristop od spodaj navzgor. Če je graf neusmerjen, izvajanje pristopa od spodaj navzgor ne zahteva nobenih sprememb v podatkovnih strukturah grafa, saj sta že predstavljeni obe smeri povezav. Če pa je graf usmerjen, bo korak od spodaj navzgor zahteval inverzni graf, kar lahko skoraj podvoji pomnilniški odtis grafa (Beamer, Asanović in Patterson, 2013).

Algoritem je učinkovit, kadar je cilj blizu začetnega vozlišča, kar je značilno zanj, saj obiskuje vsa vozlišča na istem vrhu (Elkari idr., 2024).

5.2 Slabosti algoritma BFS

Iskanje po širini, ki zagotavlja najkrajšo pot v natehtanih grafih, ima v navigaciji po labirintu precejšnje omejitve. Njegova glavna pomanjkljivost je velika poraba pomnilnika, saj shrani vsa vozlišča na trenutni globini, preden nadaljuje na naslednjo raven. Ta značilnost povzroči eksponentno rast pomnilnika, zlasti v širokih ali zapletenih labirintih. Poleg tega BFS nima

hevrističnega vodenja, zaradi česar raziskuje številne nepomembne poti in v velikih okoljih povečuje računski čas (Elkari idr., 2024).

5.3 Psevdokoda in načrt implementacije algoritma BFS

Algoritem BFS raziskuje graf, tako da najprej obišče vsa sosednja vozlišča. Začne na začetni točki (angl. Root level) in nadaljuje pot na istem nivoju, dokler ne obišče vse točke. To lahko zapišemo na naslednji način:

$$BFS(v) = 1 + \sum_{u \in sosednje\ vozlišče(v)} BFS(u).$$

Kjer je:

- v - trenutni vrh (angl. Vertex),
- u - sosednje vozlišče,
- BFS(u) - vrstni red obiska vrha (Elkari, in drugi, 2024).

S pomočjo psevdokode lahko implementiramo algoritem BFS.

Slika 8: Psevdokoda algoritma BFS

Vir: https://www.datacamp.com/tutorial/breadth-first-search-in-python, 2025

6 DIJKSTROV ALGORITEM

Dijkstrov algoritem se uporablja za iskanje najkrajše poti na grafu, kjer imamo eno izhodno točko (Wahyuningsih in Syahreza , 2018), (Rachmawati in Gustin, 2020).

Dijkstrov algoritem najde najkrajšo pot od danega izvornega vozlišča do vsakega vozlišča. Uporabimo ga lahko za iskanje najkrajše poti do določenega ciljnega vozlišča, tako da po določitvi najkrajše poti do ciljnega vozlišča algoritem zaključimo. Če na primer vozlišča grafa predstavljajo mesta, stroški robov pa razdalje med pari mest, ki jih povezuje neposredna cesta, lahko Dijkstrov algoritem uporabimo za iskanje najkrajše poti med enim mestom in vsemi drugimi mesti. Pogosta uporaba algoritmov za najkrajše poti so omrežni usmerjevalni protokoli, predvsem IS-IS (angl. Intermediate System to Intermediate System) in OSPF (angl. Open Shortest Path First). Uporablja se tudi kot podprogram v algoritmih, kot je Johnsonov algoritem (Kurt in Sanders, 2008).

Dijkstra (Dijkstra, 1959) je optimizacijski algoritem, ki se predvsem uporablja za določanje najkrajših poti. Dijkstrov algoritem je neinformiran iskalni algoritem za iskanje najkrajših poti, ki se zanaša zgolj na lokalne stroške poti in zagotavlja najkrajšo pot od začetnega do ciljnega vozlišča v grafu (Soltani, Tawfik, Goulermas in Fernando, 2002).

Algoritem uporablja podatkovno strukturo čakalne vrste z najmanjšo prioriteto za izbiro najkrajših do zdaj znanih poti. Preden so bile odkrite naprednejše strukture prioritetnih čakalnih vrst, je Dijkstrov izvirni algoritem deloval v $\theta(|E| + |V| \log|V|)$ času, kjer $|V|$ predstavlja število vozlišč (Schrijver, 2012).

Fredman in Tarjan (1984) sta predlagala prednostno čakalno vrsto s Fibonaccijevo kupo za optimizacijo časovne zapletenosti delovanja $\theta(|E| + |V| \log|V|)$. To je asimptotično najhitrejši znani algoritem najkrajše poti z enim virom za poljubne usmerjene grafe z neomejenimi nenegativnimi utežmi. Če je dovoljena predhodna obdelava, so lahko algoritmi, kot so hierarhije krčenja (angl. contraction hierarchies), bistveno hitrejši.

Dijkstrov algoritem se običajno uporablja na grafih, kjer so uteži robov pozitivna cela ali realna števila. Lahko ga posplošimo na katerikoli graf, kjer so uteži robov delno urejene, če so

zaporedne oznake monotono napadajoče (angl. monotonically non-decreasing) (Szcześniak, Jajszczyk in Woźna-Szcześniak, Generic Dijkstra for optical networks, 2019), (Szcześniak in Woźna-Szcześniak, Generic Dijkstra: correctness and tractability, 2023).

Na številnih področjih, zlasti na področju umetne inteligence, Dijkstrov algoritem ali njegova različica ponuja iskanje po enotnih stroških in je oblikovan kot primer splošnejše zamisli o iskanju po načelu najboljši prvi (angl. best-first search) (Felner, Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm, 2011).

6.1 Prednost Dijkstrovega algoritma

Prednost Dijkstrovega algoritma je, da v nasprotju z nekaterimi osnovnimi hevrističnimi algoritmi zagotavlja najkrajšo pot. Algoritem je precej učinkovit, saj deluje v času O(E log(V)), kjer E pomeni število robov v grafu, V pa število vrhov v grafu. To učinkovitost je mogoče nekoliko povečati z uporabo čakalne vrste z najmanjšo prioriteto za shranjevanje vozlišč (Nico, 2020).

V raziskavi, ki sta jo naredila Noto in Sato leta 2000, sta predlagala, da algoritem začne z iskanjem na začetnem in končnem vozlišču. To zniža območje iskanja vozlišč in s tem zmanjša računski čas algoritma. Ta sprememba algoritma omogoča, da se število obiskanih vozlišč zmanjša za polovico. Z zmanjšanjem števila obiskanih vozlišč se je čas iskanja zmanjšal za petino.

6.2 Slabost Dijkstrovega algoritma

Glavna težava Dijkstrovega algoritma je, da izvaja slepo iskanje, ki je lahko dolgotrajno in potratno v smislu izračunavanja. Dijkstrov algoritem je pogosto uporabljen za reševanje problema najkrajše poti. Če je pot kompleksna in velika, kar se običajno zgodi v praktičnem okolju, algoritem pri iskanju najkrajše poti vzame preveč časa. Večina iskanih vozlišč je nepomembnih, saj ta ne morejo biti del rešitve. Zato algoritem zapravi veliko računskega časa (angl. computation time). Čeprav je ta algoritem učinkovit, bo iskanje celotnega grafa s tisoči vozlišč, da bi našli najkrajšo pot, še vedno trajalo dolgo časa (Liu, idr., 1994).

Pomanjkljivost algoritma je, da če do željenega cilja ni vozlišča, se mora algoritem pred zaključkom ponovno sprehoditi po celotnem grafu. Algoritem je pohlepen (angl. greedy

alghorithm), kar pomeni, da bo vedno izbral možnost, ki je na prvi pogled vidna kot optimalna. To lahko vodi v iskanje poti, ki ne obstaja (Nico, 2020).

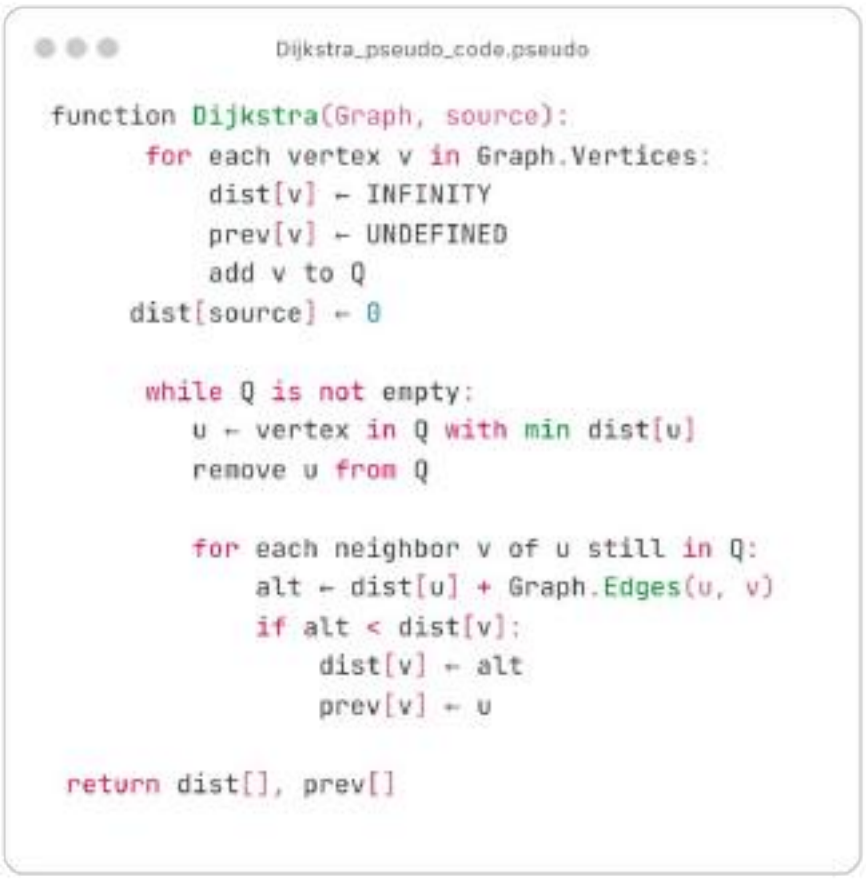
Dijkstrov algoritem lahko najde optimalne rešitve s sistematičnim ustvarjanjem vozlišč in njihovim testiranjem glede na cilj, vendar postane neučinkovit za obsežne probleme (Soltani, Tawfik, Goulermas in Fernando, 2002).

6.3 Psevdokoda in načrt implementacije Dijkstrovega algoritma

Časovna kompleksnost algoritma je odvisna od števila vozlišč, ki jih mora obiskati. Ker algoritem išče na slepo, je večje število vozlišč omejitveni dejavnik. Omejitve časa delovanja Dijkstrovega algoritma na grafu z robovi E in vrhovi V je mogoče izraziti kot funkcijo števila robov, označeno z $|E|$, in številom vozlišč$|V|$. Časovna kompleksnost je odvisna od podatkovne strukture, ki se uporablja za predstavitev množice Q.

Zgornje meje lahko poenostavimo, ker $|E|$ je $O(|V|^2)$ za vsak preprost graf. Za katerokoli podatkovno strukturo za množico vrhov Q je čas delovanja $\theta(|E| * T_{dk} + |V| * T_{em})$, kjer T_{dk} in T_{em} predstavljata kompleksnost operacij algoritmov (Cormen idr., 2022).

S pomočjo psevdokode lahko implementiramo Dijkstrov algoritem.



Slika 9: Psevdokoda algoritma Dijkstra

Vir: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, 2025

7 UMETNA INTELIGENCA

Umetna inteligenca se nanaša na sposobnost računalniških sistemov, da opravljajo naloge, ki so običajno povezane s človeško inteligenco, kot so učenje, sklepanje, reševanje problemov, zaznavanje in odločanje. Je področje v računalništvu, ki razvija in preučuje metode in programsko opremo, ki strojem omogočajo zaznavanje okolja ter uporabo učenja in inteligence za ukrepanje, ki povečuje možnosti za doseganje določenih ciljev (Russell in Norvig, 2022).

Različna podpodročja umetne inteligence se osredotočajo na posebne cilje in uporabo posebnih orodij. Primarni cilji raziskav umetne inteligence vključujejo učenje, sklepanje, predstavitev znanja, načrtovanje, obdelavo naravnega jezika, zaznavanje in podporo robotiki. Eden izmed dolgoročnih ciljev področja umetne inteligence je doseči raven splošne umetne inteligence (angl. general intelligence). Splošna umetna inteligenca je zmožnost opravljanja naloge z enako zmogljivostjo kot človek (Russell in Norvig, 2022).

Za doseganje teh ciljev so raziskovalci umetne inteligence prilagodili in povezali številne tehnike, vključno z iskanjem in matematično optimizacijo (angl. search and mathematical optimization), formalno logiko, umetnimi nevronskimi mrežami in metodami, ki temeljijo na statistiki, operacijskih raziskavah in ekonomiji. Umetna inteligenca se tudi osredotoča na psihologijo, jezikoslovje, filozofijo, nevroznanost in druga področja (Russell in Norvig, 2022).

Aplikacije in naprave, opremljene z umetno inteligenco, lahko vidijo in prepoznajo predmete. Razumejo lahko človeški jezik in se nanj odzivajo. Učijo se lahko na podlagi novih informacij in izkušenj. Uporabnikom in strokovnjakom lahko pripravijo podrobna priporočila. Delujejo lahko samostojno in nadomestijo potrebo po človeški inteligenci ali posredovanju, na primer samovozeči avtomobili.

Neposredno pod umetno inteligenco spada strojno učenje, ki vključuje ustvarjanje modelov z usposabljanjem algoritma za napovedovanje ali odločanje na podlagi podatkov. Zajema široko paleto tehnik, ki računalnikom omogočajo učenje in sklepanje na podlagi podatkov, ne da bi bili izrecno programirani za določene naloge.

7.1 Umetna inteligenca in razumevanje

Russell in Norvig (2019) opisujeta, da predstavljanje in inženiring znanja (angl. knowledge engineering) omogočata programom umetne inteligence, da inteligentno odgovarjajo na vprašanja in sklepajo o dejstvih iz resničnega sveta. Formalne predstavitve znanja se uporabljajo pri indeksiranju in iskanju na podlagi vsebine, interpretaciji prizora, podpori kliničnim odločitvam in odkrivanju znanja iz velikih podatkovnih baz.

Predstavljanje znanja in inženiring znanja je področje umetne inteligence, ki poskuša posnemati presojo in vedenje človeka na določenem področju. Inženiring znanja je tehnologija, ki stoji za ustvarjanjem sistemov za pomoč pri vprašanjih, povezanih z njihovim področjem znanja. Sistemi vključujejo veliko, razširljivo bazo znanja, integrirano z mehanizmom pravil, ki določa, kako uporabiti informacije v vsaki posamezni situaciji. Inženirji znanja lahko ustvarijo sistem, ki vključuje strojno učenje (angl. machine learning), tako da se lahko uči iz izkušenj na enak način kot ljudje. Strokovni sistemi se uporabljajo na različnih področjih, vključno z zdravstvom, storitvami za stranke, finančnimi storitvami, proizvodnjo in pravom (Lutkevich, 2022).

Da umetna inteligenca poskuša posnemati človeško vedenje, potrebuje bazo podatkov. Baza znanja je zbirka znanja, predstavljena v obliki, ki jo lahko uporablja umetna inteligenca. Ontologija je niz predmetov, odnosov, pojmov in lastnosti, ki se uporabljajo v določeni domeni znanja. Baze znanja morajo predstavljati stvari, kot so predmeti, lastnosti, kategorije in odnosi med predmeti, situacije, dogodki, stanja in čas, vzroki in posledice, znanje, privzeto sklepanje, ter številne druge vidike in domene znanja (Russell in Norvig, 2022).

7.2 Umetna inteligenca in strojno učenje

Strojno učenje je študija programov, ki lahko samodejno izboljšajo svoje delovanje pri določeni nalogi. Je del umetne inteligence že od samega začetka njenega razvoja (Russell in Norvig, 2022).

Strojno učenje vsebuje ustvarjanje modelov za usposabljanje algoritma za napovedovanje ali določanje rezultatov na podlagi vnosnih podatkov. Zajema širok nabor tehnik, ki računalnikom omogočajo učenje in sklepanje na podlagi podatkov, ne da bi bili neposredno razviti za določene naloge (Stryker in Kavlakoglu, 2024).

Strojno učenje vsebuje veliko vrst tehnik ali algoritmov strojnega učenja, vključno z linearno regresijo (angl. linear regression), logistično regresijo (angl. logistic regression), drevesi odločanja (angl. decision trees), naključnim gozdom (angl. random forest), podpornimi vektorskimi stroji (angl. support vector machines), k-najbližjim sosedom (angl. k-nearest neighbor) in grozdenjem (angl. clustering) (Stryker in Kavlakoglu, 2024).

Ena izmed najbolj uporabljenih vrst algoritmov za strojno učenje je nevronska mreža. Nevronske mreže so oblikovane po strukturi in delovanju človeških možganov. Nevronsko omrežje je sestavljeno iz medsebojno povezanih plasti vozlišč, ki sodelujejo pri obdelavi in analizi kompleksnih podatkov. Nevronske mreže so primerne za naloge, ki vključujejo prepoznavanje zapletenih vzorcev in povezav v velikih količinah podatkov (Stryker in Kavlakoglu, 2024).

Obstaja več vrst strojnega učenja. Nenadzorovano učenje analizira tok podatkov in išče vzorce ter napoveduje brez drugih navodil. Nadzorovano učenje zahteva označevanje učnih podatkov s pričakovanimi odgovori in se deli na dve glavni vrsti, in sicer klasifikacijo, kjer se mora program naučiti napovedati, v katero kategorijo spada vhodni podatek, in regresijo, kjer mora program na podlagi številčnega vnosa določiti številčno funkcijo (Russell in Norvig, 2022).

7.3 Umetna inteligenca in iskanje

UI lahko reši veliko problemov z inteligentnim iskanjem številnih možnih rešitev. V UI se uporabljata dve različni vrsti iskanja, in sicer iskanje v prostoru stanj in lokalno iskanje (Russell in Norvig, 2022).

Iskanje v prostoru išče po drevesu možnih stanj, da bi našlo ciljno stanje. Na primer algoritmi za načrtovanje iščejo po drevesih ciljev in podciljev, pri čemer poskušajo najti pot do ciljnega stanja, kar se imenuje analiza sredstev in ciljev (angl. means-ends analysis) (Russell in Norvig, 2022).
Preprosto izčrpno iskanje (angl. simple exhaustive searches) redko zadostuje za večino realnih problemov, saj iskalni prostor hitro naraste. Rezultat je iskanje, ki je prepočasno ali pa se nikoli ne konča. Za izboljšanje časa iskanja nam lahko pomaga hevristika (Russell in Norvig, 2022).

Lokalno iskanje uporablja matematično optimizacijo za iskanje rešitve problema. Začne se z neko obliko ugibanja in ga postopoma izpopolnjuje (Russell in Norvig, 2022).

Gradientno spuščanje je vrsta lokalnega iskanja, ki optimizira niz numeričnih parametrov z njihovim postopnim prilagajanjem, da se čim bolj zmanjša funkcija izgube. Različice gradientnega spuščanja se pogosto uporabljajo za usposabljanje nevronskih mrež z algoritmom povratnega širjenja (angl. backpropagation) (Russell in Norvig, 2022).

Druga vrsta lokalnega iskanja je evolucijsko računanje, katerega cilj je iterativno izboljšati niz kandidatnih rešitev z njihovo mutacijo in rekombinacijo, pri čemer se v vsaki generaciji izberejo le najprimernejši, ki preživijo (Russell in Norvig, 2022).

7.4 Umetna inteligenca in iskanje najkrajše poti

UI je ena od ključnih delov videoiger (Ostrowski, 2015). Yannakakih (2018) opisuje, da so zgodnje raziskave UI v iskanju najkrajše poti vključevale šah in druge družinske namizne igre. V igrah je opredeljeno, kako se obnaša računalniški nasprotnik do igralca. Obnašanje računalniškega nasprotnika je segalo od preprostih do kompleksnih vzorcev gibanja s pomočjo algoritmov iskanja poti. Ena izmed morebitnih implementacij UI v videoigrah je posnemanje igralca na podlagi njegovih naključnih dejanj in posledično računalniški nasprotnik lahko z uporabo UI prilagodi svoje gibanje (Iskandar, Diah in Ismail, 2020).

Igralniška industrija še naprej hitro raste zaradi tehnološkega napredka, predvsem na področju umetne inteligence. UI se v sodobnih videoigrah uporablja za različne igralne like. Primarni cilj UI je igralcu zagotoviti izziv pri sprejemanju odločitev in povečati stopnjo težavnosti. UI omogoča prilagajanje raznih igralnih likov znotraj videoigre na igralčeve odločitve (Lawande, Jasmine, Anbarasi in Izhar, 2022).

Iskanje poti se nanaša na koncept iskanja optimalne poti od izvornega do ciljnega vozlišča v najkrajšem času. Za iskanje najkrajše poti od izvornega do ciljnega vozlišča je bilo zasnovanih več algoritmov, ki se poskušajo izogniti vsem oviram na poti. Za iskanje poti lahko uporabljajo tudi UI (Rafiq in Kadir, Pathfinding Algorithms in Game Development, 2020), (Lawande, Jasmine, Anbarasi in Izhar, 2022).

Razvoj UI in iskanja poti je dosegel velik napredek, vendar ima še vedno določene probleme. Eden izmed takšnih problemov je zahteva po visoki zmogljivosti, ki jo morajo ti algoritmi zagotoviti v videoigrah. Ob visoki zahtevi zmogljivosti morajo pogosto ti algoritmi izračunati poti za več komponent, in ker so viri, dodeljeni tem algoritmom, omejeni, obstaja povpraševanje po algoritmih z visoko zmogljivostjo v krajšem času reševanja problema iskanja poti (Lawande, Jasmine, Anbarasi in Izhar, 2022).

7.5 Stroški razvoja in implementacije umetne inteligence

Pri razvoju in implementaciji umetne inteligence v interne procese podjetja naletijo na nepričakovano povečane stroške. Hitro naraščajoči stroški računalništva lahko ovirajo nadaljnji razvoj in inovacije podjetja (Brodsky, 2024).

Ekonomski pritisk čutijo tudi vodilni pri razvoju umetne inteligence. Podjetje OpenAI naj bi zabeležil eksponentno rast prihodkov, saj je v mescu avgustu leta 2024 doseglo 300 milijonov ameriških dolarjev prihodka. V začetku oktobra je podjetje OpenAI objavilo, da so v novem krogu financiranja zbrali 6,6 milijarde ameriških dolarjev, s čimer bi pokrili rast podjetja in stroške razvoja ter implementacije UI (Brodsky, 2024).

Ekonomski vidik UI postaja ključni dejavnik pri določanju njenega poslovnega učinka. Številna podjetja zaradi naraščajočih stroškov UI prestavljajo svoje interne produkte na hibridno oblačno arhitekturo (angl. hybrid cloud architectures) (Brodsky, 2024).

Eden glavnih dejavnikov stroškov je vrsta rešitve UI. Vsak sistem UI ni zgrajen z enako tehnologijo, razlike med njimi so velike. Poznamo več sistemov, in sicer:

- Sistemi, ki temeljijo na pravilih (angl. rule-based systems), so preprosti za razvoj in implementacijo. Ti sistemi upoštevajo preprosta pravila in zahtevajo minimalno računalniško moč (Le, 2025).
- Rešitve za strojno učenje, ki jih uporabljajo podjetja, se s časom izboljšujejo. Te potrebujejo kakovostne podatke in stalno prilagajanje, kar vodi do višjih stroškov razvoja in implementacije (Le, 2025).

- Modeli globokega učenja so odlični za reševanje zapletenih nalog, kot je prepoznavanje slik ali glasu. Zahtevajo veliko količino podatkov, napredne algoritme in vrhunsko strojno opremo, zato je njihov razvoj najdražji (Le, 2025).

Na stroške vplivata tudi obseg in zapletenost projekta. Projekti UI so različni, pri čemer so število funkcij, točke integracije in zahtevane ravni zmogljivosti zelo pomembne. Enostavno orodje za analizo povratnih informacij stranke običajno stane med dvajset tisoč in štirideset tisoč ameriških dolarjev. Srednje zahtevne aplikacije, ki potrebujejo naprednejše algoritme za analizo vedenja in preferenc uporabnikov, stanejo med petdeset tisoč in sto tisoč ameriških dolarjev. Napredne aplikacije UI, kot so večjezični modeli, lahko presežejo sto petdeset tisoč ameriških dolarjev za implementacijo v podjetje. Te aplikacije so zapletene zaradi zahtev po visoki zmogljivosti večjezičnih modelov (Le, 2025).

Število zaposlenih je pogosto najbolj spremenljiv dejavnik pri določanju stroškov razvoja in implementacije UI v podjetju. Velikost stroškov je odvisna od števila zaposlenih in lokacije podjetja. Interni razvoj omogoča popolni nadzor in tesnejše sodelovanje razvijalcev UI. Vendar to vodi do višjih stroškov razvoja. Najem zunanjih razvijalcev omogoča dostop do specializiranega kadra, kar vodi do dolgoročno nižjih stroškov dela (Le, 2025).

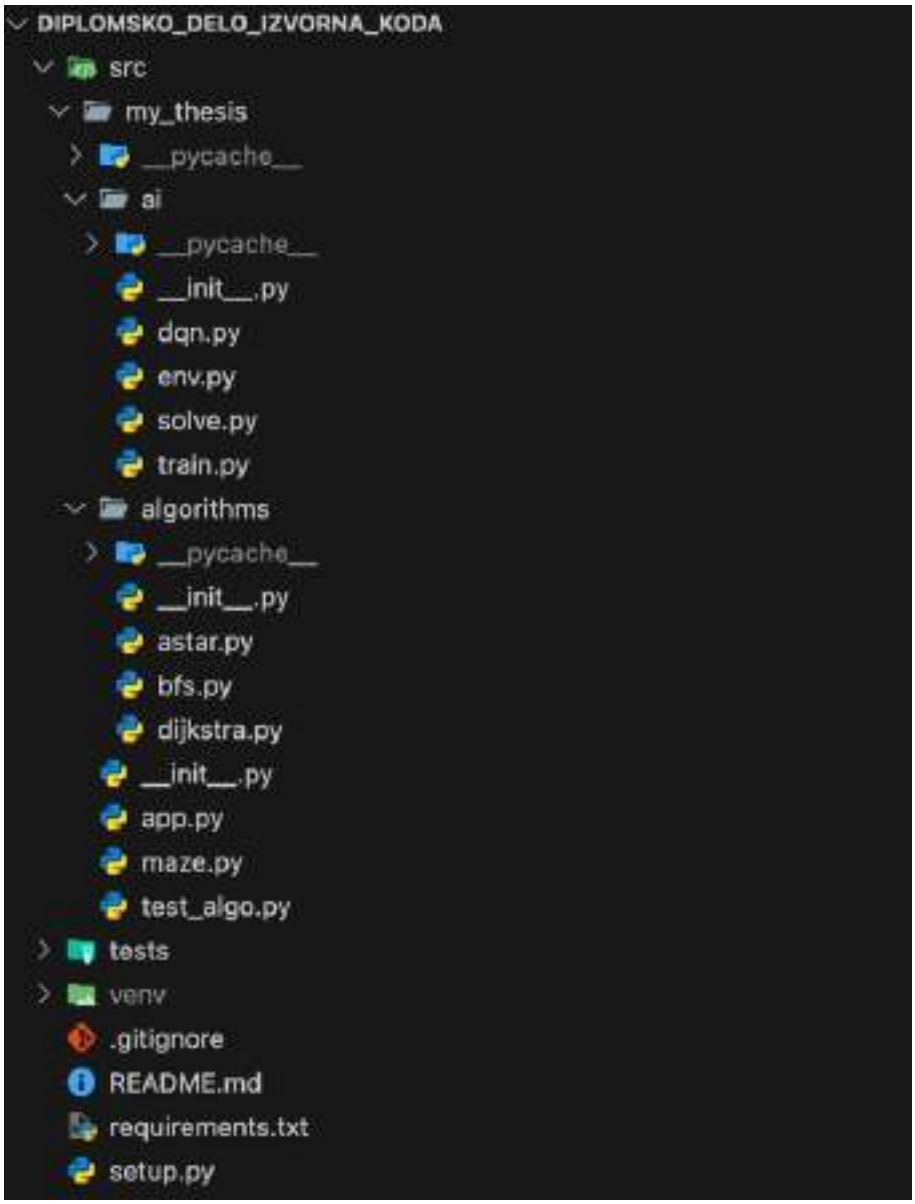
8 PRIPRAVA OKOLJA

Za testiranje in primerjavo algoritmov iskanja najkrajše poti smo razvili aplikacijo za vizualizacijo mreže in iskanje poti do končnega vozlišča. Aplikacija je razvita v programskem jeziku Python. Aplikacija bo vsebovala grafični uporabniški vmesnik (angl. graphical user interface), kjer bo uporabnik lahko generiral naključno mrežo z ovirami. Na voljo bo možnost izbire algoritma za iskanje najkrajše poti. Končni rezultat iskanja bo zapisan v tabeli. Časovna enota merjenja je milisekunda. Za programiranje uporabniškega vmesnika bomo uporabili knjižnico Tkinter.

Ker v diplomskem delu primerjamo umetno inteligenco in algoritme iskanja najkrajše poti, bomo za to uporabili knjižnico PyTorch. Knjižnica nam omogoča, da program učimo reševati problem s pomočjo nevronske mreže.

8.1 Struktura izvorne kode

Struktura izvorne kode je razdeljena v več logičnih enot. To omogoča lažji pregled logike. Razdelili smo kodo na več manjših delov, ki imajo vsak svojo logiko. Modularna koda omogoča lažje sledenje logiki, lažje nadgrajevanje in njeno razhroščevanje.

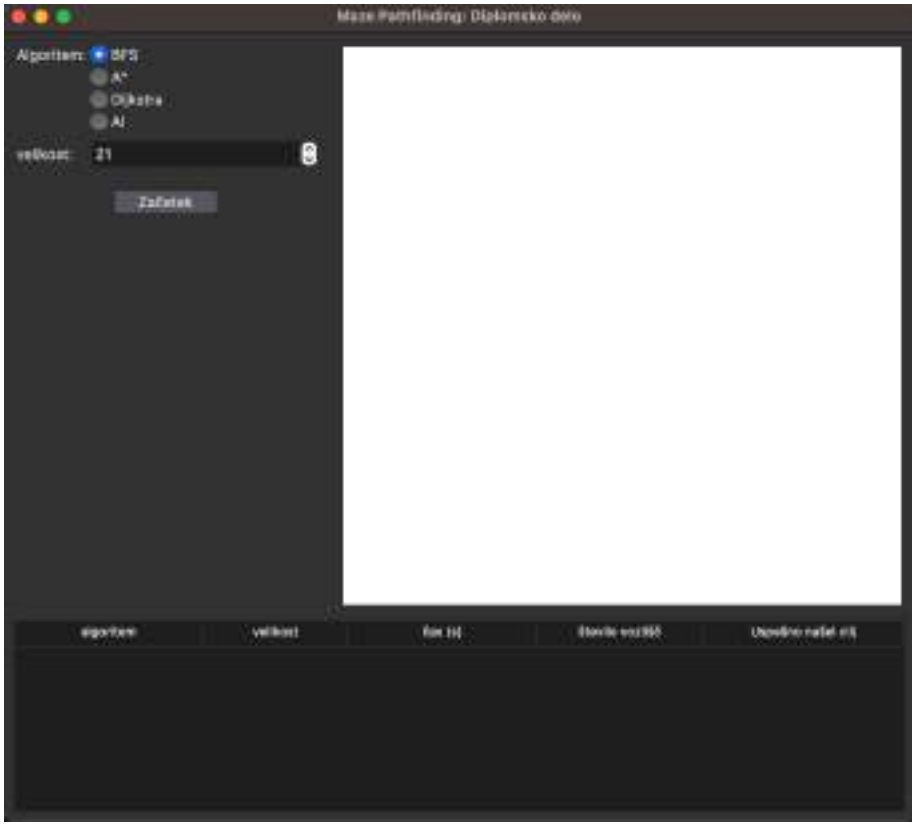


Slika 10: Struktura izvorne kode

Vir: Lasten

8.2 Uporabniški vmesnik aplikacije

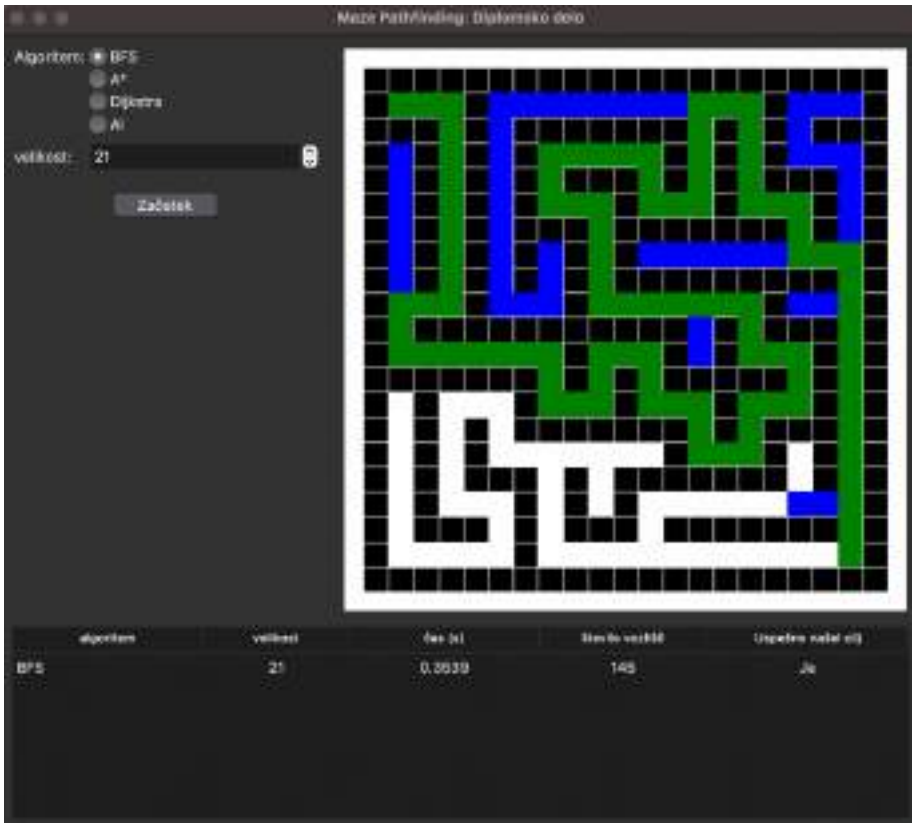
Grafični vmesnik vsebuje možnost izbora algoritma, prikaz labirinta in tabelo za zapis podatkov.



Slika 11: Uporabniški vmesnik aplikacije

Vir: Lasten

Za prikaz uporabniškega vmesnika smo razvili razred App. Razred implementira grafični uporabniški vmesnik z uporabo knjižnice Tkinter, ki omogoča uporabniku izbiro algoritma, velikosti labirinta in vizualno spremljanje izvajanja iskanja poti. Ob kliku na gumb Start se ustvari nov labirint iz razreda Maze, nato se izvede izbrani algoritem iz objekta algorithms, medtem ko se obiski vozlišč sproti prikazujejo z modro barvo, najdena pot pa z zeleno. Rezultati se prikažejo v tabeli Treeview. Vizualizacija in logika iskanja sta ločeni z uporabo niti, da GUI ostane odziven. Canvas prikazuje mrežo, ovire in pot, s čimer aplikacija omogoča interaktivno primerjavo učinkovitosti različnih algoritmov.



Slika 12: Prikaz iskanja algortima v aplikaciji

Vir: Lasten

__init__.py

```
from .app import App

if __name__ == "__main__":
    app = App()
    app.mainloop()
```

Slika 13: Izvorna koda razreda App.py

Vir: Lasten

app.py

```
class App(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Maze Pathfinding: Diplomsko delo")
        self.create_widgets()
        self.run_times = []
```

Slika 14: Izvorna koda razreda App.py

Vir: Lasten

```
class App(tk.Tk):
    def start(self):
        try:
            size = int(self.size_spin.get())
        except ValueError:
            messagebox.showerror("Napravilen vnos", "Velikost mora biti celo število.")
            return
        algo_name = self.algo_var.get()
        self.start_btn.config(state="disabled")
        threading.Thread(
            target=self.run_maze, args=(size, algo_name), daemon=True
        ).start()
```

Slika 15: Izvorna koda metode start() razreda App.py

Vir: Lasten

```
class App(tk.Tk):
    def update_table(self):
        for i in self.tree.get_children():
            self.tree.delete(i)

        for algo_name, size, duration, nodes, success in self.run_times:
            self.tree.insert(
                "", "end", values=(algo_name, size, f"{duration:.4f}", nodes, success)
            )
```

Slika 16: Izvorna koda metode update_table() razreda App.py

Vir: Lasten

app.py

```
class App(tk.Tk):
    def draw_cell(self, x, y, color):
        size = len(self.grid)
        padding = 20
        canvas_w = self.canvas.winfo_width() - 2 * padding
        canvas_h = self.canvas.winfo_height() - 2 * padding
        cell_w = canvas_w / size
        cell_h = canvas_h / size
        x1 = padding + x * cell_w
        y1 = padding + y * cell_h
        x2 = x1 + cell_w
        y2 = y1 + cell_h
        self.canvas.create_rectangle(x1, y1, x2, y2, fill=color, outline="")
```

Slika 17: Izvorna koda draw_cell() razreda App.py

Vir: Lasten

app.py

```
class App(tk.Tk):
    def create_widgets(self):
        frm = ttk.Frame(self, padding=10)
        frm.grid(row=0, column=0, sticky="nw")

        ttk.Label(frm, text="Algoritem:").grid(row=0, column=0, sticky="w")
        self.algo_var = tk.StringVar(value=list(algorithms.keys())[0])
        row = 0
        for algo in algorithms.keys():
            rbtn = ttk.Radiobutton(frm, text=algo, variable=self.algo_var, value=algo)
            rbtn.grid(row=row, column=1, sticky="w")
            row += 1

        ttk.Label(frm, text="velikost:").grid(row=row, column=0, sticky="w")
        self.size_spin = tk.Spinbox(frm, from_=5, to=101, increment=2)
        self.size_spin.delete(0, "end")
        self.size_spin.insert(0, "21")
        self.size_spin.grid(row=row, column=1, pady=5)

        self.start_btn = ttk.Button(frm, text="Začetek", command=self.start)
        self.start_btn.grid(row=row + 1, column=0, columnspan=2, pady=10)

        self.canvas = tk.Canvas(self, width=500, height=500, bg="white")
        self.canvas.grid(row=0, column=1, rowspan=row + 2, padx=10, pady=10)

        columns = ("algoritem", "velikost", "čas", "vozlišča", "uspešnost")
        self.tree = ttk.Treeview(self, columns=columns, show="headings", height=8)
        self.tree.grid(
            row=row + 2, column=0, columnspan=2, sticky="nsew", padx=10, pady=(0, 10)
        )

        self.tree.heading("algoritem", text="algoritem")
        self.tree.heading("velikost", text="velikost")
        self.tree.heading("čas", text="čas (s)")
        self.tree.heading("vozlišča", text="število vozlišč")
        self.tree.heading("uspešnost", text="Uspešno našel cilj")

        self.tree.column("algoritem", width=120)
        self.tree.column("velikost", width=80, anchor="center")
        self.tree.column("čas", width=120, anchor="center")
        self.tree.column("vozlišča", width=100, anchor="center")
        self.tree.column("uspešnost", width=130, anchor="center")
```

Slika 18: Izvorna koda metode create_widgets() razreda App.py

Vir: Lasten

app.py

```
class App(tk.Tk):
    def run_maze(self, size, algo_name):
        maze = Maze(size)
        grid = maze.generate()
        self.grid = grid
        start = maze.start
        end = maze.end
        self.after(0, lambda: self.draw_maze(grid, start, end))
        visited_count = [0]

        def visit(x, y):
            visited_count[0] += 1
            self.after(0, lambda: self.draw_cell(x, y, "blue"))
            time.sleep(0.001)

        algo_fn = algorithms[algo_name]
        start_time = time.time()
        path = algo_fn(grid, start=start, goal=end, visit_callback=visit)
        duration = time.time() - start_time

        reached_goal = len(path) > 0 and path[-1] == end
        success_text = "Ja" if reached_goal else "Ne"

        if path and len(path) > 1:
            for x, y in path:
                self.after(0, lambda x=x, y=y: self.draw_cell(x, y, "green"))

        self.run_times.append(
            (algo_name, size, duration, visited_count[0], success_text)
        )
        self.after(0, self.update_table)

        if reached_goal:
            result_msg = f"✅ {algo_name} je uspešno našel cilj!\n"
            result_msg += f"Čas: {duration:.4f} sekund\n"
            result_msg += f"Obiskana vozlišč: {visited_count[0]}\n"
            result_msg += f"Dolžina poti: {len(path)} korakov"
        else:
            result_msg = f"❌ {algo_name} ni našel poti do cilja.\n"
            result_msg += f"Čas: {duration:.4f} sekund\n"
            result_msg += f"Obiskano vozlišč: {visited_count[0]}"

        self.after(0, lambda: messagebox.showinfo("Rezultat", result_msg))
        self.after(0, lambda: self.start_btn.config(state="normal"))
```

Slika 19: Izvorna koda metode run_maze() razreda App.py

Vir: Lasten

app.py

```python
from .bfs import bfs
from .astar import astar
from .dijkstra import dijkstra
from ..ai.solve import solve_with_dqn

algorithms = {
    "BFS": bfs,
    "A*": astar,
    "Dijkstra": dijkstra,
    "AI": solve_with_dqn,
}
```

Slika 20: Izvorna koda objekta algorithms

Vir: Lasten

app.py

```
class App(tk.Tk):
    def draw_maze(self, grid, start, end):
        self.canvas.delete("all")
        size = len(grid)
        padding = 20
        canvas_w = self.canvas.winfo_width() - 2 * padding
        canvas_h = self.canvas.winfo_height() - 2 * padding
        cell_w = canvas_w / size
        cell_h = canvas_h / size

        self.canvas.create_rectangle(
            padding,
            padding,
            padding + cell_w * size,
            padding + cell_h * size,
            outline="gray",
        )

        for y in range(size):
            for x in range(size):
                x1 = padding + x * cell_w
                y1 = padding + y * cell_h
                x2 = x1 + cell_w
                y2 = y1 + cell_h
                if (x, y) == start:
                    color = "green"
                elif (x, y) == end:
                    color = "red"
                elif grid[y][x] == 1:
                    color = "black"
                else:
                    continue
                self.canvas.create_rectangle(x1, y1, x2, y2, fill=color)
```

Slika 21: Izvorna koda metode draw_maze() razreda App.py

Vir: Lasten

app.py

```
if __name__ == "__main__":
    App().mainloop()
```

Slika 22: Izvorna koda razreda App.py

Vir: Lasten

8.3 Izvorna koda labirinta

Za prikaz naključno generiranega labirinta smo naredili razred Maze, ki omogoča ustvarjanje mreže s prehodnimi in neprehodnimi polji. Razred vsebuje metode __init__(self, size) in generate(self). Razred vsebuje spremenljivke size, grid, start in end. S spremenljivkami določamo velikost labirinta z začetno in končno točko.

Metoda __init__(self,size) je konstruktor razreda, ki ustvari mrežo poljubne velikosti. Definira začetno in končno točko razreda.

Metoda generate(self) s pomočjo algoritma ustvari prehodni labirint. S pomočjo stack in visited spreminjamo vozlišča iz neprehodnih v prehodne. V vsakem koraku zanke se preveri, ali obstaja še neobiskano sosednje vozlišče, ki ga nato naključno izbere in nadaljuje z ustvarjanjem labirinta. Rezultat razreda je dvodimenzionalen labirint z začetno in končno točko.

Maze.py

```python
import random


class Maze:
    def __init__(self, size):
        self.size = size
        self.grid = [[1 for _ in range(size)] for _ in range(size)]
        self.start = (1, 1)
        self.end = (size - 2, size - 2)

    def generate(self):
        stack = []
        visited = set()
        x, y = self.start
        stack.append((x, y))
        visited.add((x, y))
        self.grid[y][x] = 0

        while stack:
            x, y = stack[-1]
            neighbors = []
            for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                nx, ny = x + dx * 2, y + dy * 2
                if (
                    1 <= nx < self.size - 1
                    and 1 <= ny < self.size - 1
                    and (nx, ny) not in visited
                ):
                    neighbors.append((nx, ny, dx, dy))
            if neighbors:
                nx, ny, dx, dy = random.choice(neighbors)
                self.grid[y + dy][x + dx] = 0
                self.grid[ny][nx] = 0
                visited.add((nx, ny))
                stack.append((nx, ny))
            else:
                stack.pop()

        end_x, end_y = self.end
        self.grid[end_y][end_x] = 0
        for dx, dy in [(-1, 0), (0, -1)]:
            nx, ny = end_x + dx, end_y + dy
            if 1 <= nx < self.size - 1 and 1 <= ny < self.size - 1:
                self.grid[ny][nx] = 0

        return self.grid
```

Slika 23: Izvorna koda razreda Maze.py

Vir: Lasten

8.4 Izvorna koda algoritma BFS

Algoritem iskanje po širini smo implementirali z metodo bfs(grid, start, goal, visit_callback=lambda x, y; None). Na začetku iteracije metoda ustvari prazno množico visited, vrsto queue ter objekt prev. V vsaki iteraciji se iz vrste vzame trenutno vozlišče in preveri, ali ustreza cilju. Če je cilj dosežen, se iteracija prekine. Če cilj ni dosežen, se preverijo vsa sosednja vozlišča. Če je vozlišče neobiskano in je prehodno, se doda v vrsto queue in se označi kot obiskano. Nato se ta povezava shrani v objekt prev. Ko se iskanje zaključi, se rekonstruira pot do cilja na podlagi vozlišč v objektu prev. Rezultat metode je seznam vozlišč, ki predstavljajo najkrajšo pot.

bfs.py

```python
from collections import deque


def bfs(grid, start, goal, visit_callback=lambda x, y: None):
    size = len(grid)
    visited = set()
    queue = deque([start])
    prev = {}

    visited.add(start)

    while queue:
        current = queue.popleft()
        x, y = current
        visit_callback(x, y)

        if current == goal:
            break

        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            nx, ny = x + dx, y + dy
            neighbor = (nx, ny)

            if (
                0 <= nx < size
                and 0 <= ny < size
                and grid[ny][nx] == 0
                and neighbor not in visited
            ):
                visited.add(neighbor)
                prev[neighbor] = current
                queue.append(neighbor)

    path = []
    node = goal
    while node in prev:
        path.append(node)
        node = prev[node]
    if path:
        path.append(start)
        path.reverse()
    return path
```

Slika 24: Izvorna koda algoritma BFS

Vir: Lasten

8.5 *Izvorna koda algoritma A**

A star algoritem smo implementirali z metodo astar(grid, start={1,1}, goal=None, visit_callback=None). Funkcija uporablja strošek poti od začetka in oceno razdalje do cilja. Na začetku se določi cilj in pripravi prioritetno vrsto, kamor vstavi začetno vozlišče. Zanka nato vedno izbere vozlišče z najmanjšo skupno oceno.

```python
def astar(grid, start=(1, 1), goal=None, visit_callback=None):
    import heapq

    n = len(grid)
    if goal is None:
        goal = (n - 2, n - 2)

    def h(a, b):
        return abs(a[0] - b[0]) + abs(a[1] - b[1])

    open_set = []
    heapq.heappush(open_set, (h(start, goal), 0, start, [start]))
    g_score = {start: 0}
    visited = set()

    while open_set:
        _, cost, current, path = heapq.heappop(open_set)
        if current in visited:
            continue
        visited.add(current)

        if visit_callback:
            visit_callback(*current)

        if current == goal:
            return path

        x, y = current
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n and grid[ny][nx] == 0:
                tentative_g = cost + 1
                neighbor = (nx, ny)
                if tentative_g < g_score.get(neighbor, float("inf")):
                    g_score[neighbor] = tentative_g
                    f = tentative_g + h(neighbor, goal)
                    heapq.heappush(
                        open_set, (f, tentative_g, neighbor, path + [neighbor])
                    )

    return []
```

Slika 25: Izvorna koda algoritma A star

Vir: Lasten

8.6 Izvorna koda algoritma Dijkstra

Funkcija Dijkstra išče najkrajšo pot v mreži, pri čemer za vsako vozlišče vodi slovar dist z doslej poznanimi najnižjimi stroški od začetka in uporablja prioritetno vrsto za izbiro vozlišča z najmanjšim trenutno znanim stroškom. Začne pri začetku z razdaljo 0, nato v vsaki iteraciji iz vrste vzame vozlišče z najmanjšim cost, ga označi kot obiskano, in pokliče visit_callback. Če je cilj dosežen, prekine zanko, sicer pregleda vse sosednje običajne premike, izračuna nov potencialni strošek (new_cost = cost + 1) in, če je ta manjši od prej znanega za soseda, posodobi dist, shrani predhodnika v prev in ga vstavi v vrsto. Ko zanka konča, se najdena pot rekonstruira iz slovarja prev, se obrne in vrne kot seznam koordinat od začetka do cilja.

dijkstra.py

```python
import heapq


def dijkstra(grid, start, goal, visit_callback=lambda x, y: None):
    size = len(grid)
    visited = set()
    dist = {start: 0}
    prev = {}
    heap = [(0, start)]

    while heap:
        cost, current = heapq.heappop(heap)
        if current in visited:
            continue
        visited.add(current)

        x, y = current
        visit_callback(x, y)

        if current == goal:
            break

        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < size and 0 <= ny < size and grid[ny][nx] == 0:
                neighbor = (nx, ny)
                new_cost = cost + 1
                if neighbor not in dist or new_cost < dist[neighbor]:
                    dist[neighbor] = new_cost
                    prev[neighbor] = current
                    heapq.heappush(heap, (new_cost, neighbor))

    path = []
    node = goal
    while node in prev:
        path.append(node)
        node = prev[node]
    path.append(start)
    path.reverse()
    return path
```

Slika 26: Izvorna koda algoritma Dijkstra

Vir: Lasten

8.7 Izvorna koda strojnega učenja

Implementacija strojnega učenja simulira umetno inteligenco. Za reševanje naključno generiranega labirinta smo uporabili DQN pristop (angl. Deep Q-Network). Izvorna koda je sestavljena iz več razredov, in sicer DQN, MazeEnv, SolvingMazeEnv in samostojne funkcije train_dqn.

Slika 27: Izvorna koda za strojno učenje

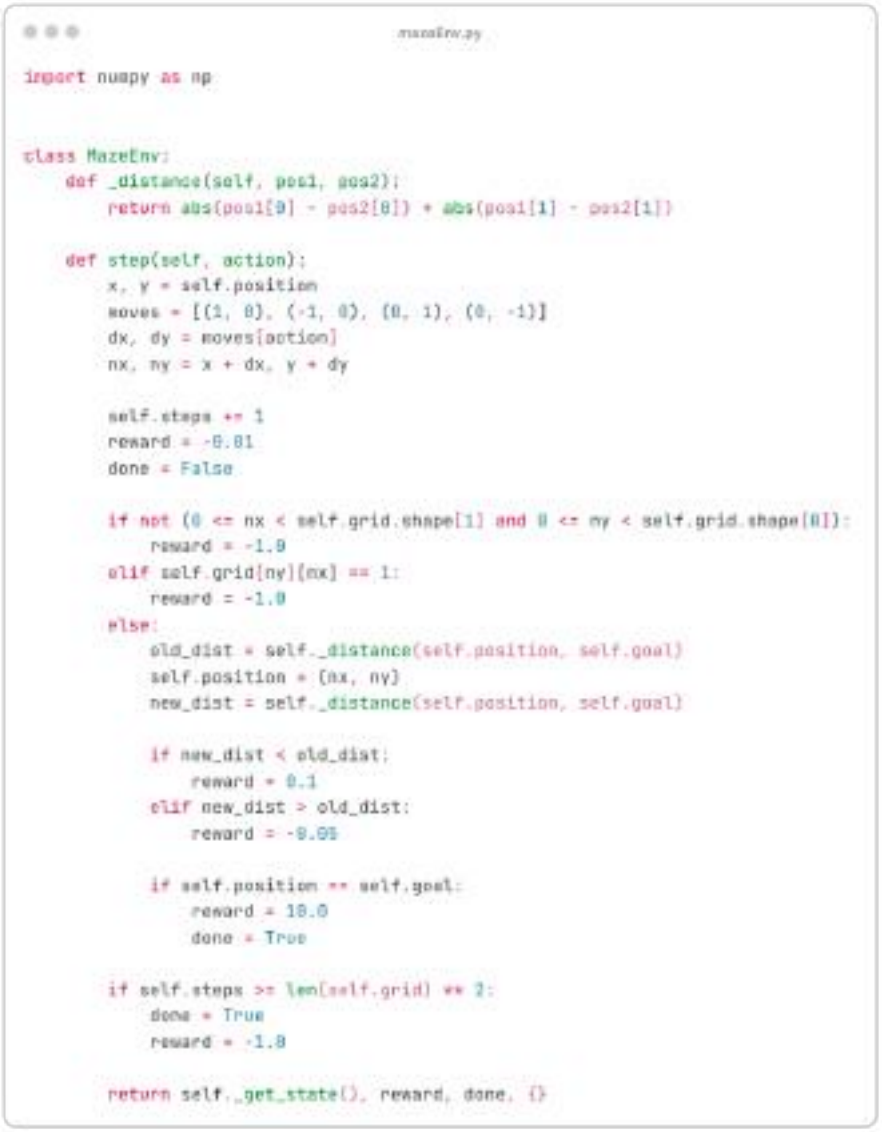
Vir: Lasten

Razred DQN implementira globoko Q-mrežo DQN kot nevronsko mrežo z veliko povezanimi sloji. Za implementacijo smo uporabili knjižnico PyTorch. Konstruktor razreda vsebuje parametre self, input_size=13, hidden_size=128, output_size=4. Ti parametri definirajo arhitekturo mreže. Vhodni sloj sprejme 13-razsežni vektor, dva skrita sloja z ReLU funkcijami procesirata informacije. Izhodni sloj vrne Q-vrednost štiri možne akcije.

Slika 28: Izvorna koda razreda DQN

Vir: Lasten

Metoda forward(self) definira prehod podatkov skozi mrežo in vrne Q-vrednost za vse možne akcije v danem stanju.



```python
import numpy as np


class MazeEnv:
    def _distance(self, pos1, pos2):
        return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

    def step(self, action):
        x, y = self.position
        moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        dx, dy = moves[action]
        nx, ny = x + dx, y + dy

        self.steps += 1
        reward = -0.01
        done = False

        if not (0 <= nx < self.grid.shape[1] and 0 <= ny < self.grid.shape[0]):
            reward = -1.0
        elif self.grid[ny][nx] == 1:
            reward = -1.0
        else:
            old_dist = self._distance(self.position, self.goal)
            self.position = (nx, ny)
            new_dist = self._distance(self.position, self.goal)

            if new_dist < old_dist:
                reward = 0.1
            elif new_dist > old_dist:
                reward = -0.05

            if self.position == self.goal:
                reward = 10.0
                done = True

        if self.steps >= len(self.grid) ** 2:
            done = True
            reward = -1.0

        return self._get_state(), reward, done, {}
```

Slika 29: Izvorna koda razreda MazeEnv

Vir: Lasten

Razred MazeEnv predstavlja okolje za učenje pri reševanju labirintov s strojnim učenjem. Razred implementira vmesnik za okolje učenja z metodami reset, step in get_state. Konstruktor razreda vsebuje parametre self, grid, start in goal. To omogoča inicializacijo labirinta za učenje z začetno in končno točko.

```python
import numpy as np


class MazeEnv:
    def __init__(self, grid, start, goal):
        self.grid = np.array(grid)
        self.start = start
        self.goal = goal
        self.reset()

    def reset(self):
        self.position = self.start
        self.steps = 0
        return self._get_state()

    def _get_state(self):
        x, y = self.position
        size = 3
        half = size // 2

        padded = np.pad(self.grid, pad_width=half, mode="constant", constant_values=1)

        patch = padded[y : y + size, x : x + size].flatten()

        norm_pos = np.array(
            [x / (self.grid.shape[1] - 1), y / (self.grid.shape[0] - 1)],
            dtype=np.float32,
        )
        norm_goal = np.array(
            [
                self.goal[0] / (self.grid.shape[1] - 1),
                self.goal[1] / (self.grid.shape[0] - 1),
            ],
            dtype=np.float32,
        )

        state = np.concatenate([patch, norm_pos, norm_goal]).astype(np.float32)
        return state
```

Slika 30: Izvorna koda razreda MazeEnv

Vir: Lasten

Metoda reset(self) poenostavi labirint v začetno stanje. Pozicijo agenta postavi na začetno točko, število korakov spremeni na število nič in vrne okolje v prvotno stanje.

Metoda _get_state(self) ustvari predstavitev trenutnega stanja okolja, kot 13-razsežni vektor, ki služi kot vhod za nevronsko mrežo. Metoda določi trenutno pozicijo agenta in definira velikost lokalnega okna velikosti 3x3. K velikosti okna se doda še eno polje, ki predstavlja neprehodni rob mreže. Izvlečeni lokalni pogled se nato splošči iz 2D-matrike v 1D-vektor z 9 elementi, kjer vsak element predstavlja vrednost posameznega polja v okolici agenta. Trenutna pozicija agenta se normalizira z delitvijo s skupno velikostjo mreže minus ena, kar zagotavlja vrednosti v intervalu [0, 1]. Ta normalizacija omogoča boljše delovanje nevronske mreže, saj so vsi vhodni podatki v enotnem merilu.

SolvingMazeEnv.py

```
class SolvingMazeEnv:
    def __init__(self, grid, start, goal):
        self.grid = np.array(grid)
        self.start = start
        self.goal = goal
        self.reset()
```

Slika 31: Konstruktor razreda SolvingMazeEnv

Vir: Lasten

```
class SolvingMazeEnv:
    def _get_state(self):
        x, y = self.position
        size = 3
        half = size // 2

        padded = np.pad(self.grid, pad_width=half, mode="constant", constant_values=1)

        patch = padded[y : y + size, x : x + size].flatten()

        norm_pos = np.array(
            [x / (self.grid.shape[1] - 1), y / (self.grid.shape[0] - 1)],
            dtype=np.float32,
        )
        norm_goal = np.array(
            [
                self.goal[0] / (self.grid.shape[1] - 1),
                self.goal[1] / (self.grid.shape[0] - 1),
            ],
            dtype=np.float32,
        )

        state = np.concatenate((patch, norm_pos, norm_goal)).astype(np.float32)
        return state
```

Slika 32: Izvorna koda metode _get_state()

Vir: Lasten

SolvingMazeEnv.py

```
class SolvingMazeEnv:
    def reset(self):
        self.position = self.start
        self.steps = 0
        return self._get_state()
```

Slika 33: Izvorna koda metode reset()

Vir: Lasten

SolvingMazeEnv.py

```
class SolvingMazeEnv:
    def step(self, action):
        x, y = self.position
        moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        dx, dy = moves[action]
        nx, ny = x + dx, y + dy

        self.steps += 1
        done = False

        if (
            0 <= nx < self.grid.shape[1]
            and 0 <= ny < self.grid.shape[0]
            and self.grid[ny][nx] == 0
        ):
            self.position = (nx, ny)

        if self.position == self.goal:
            done = True
        elif self.steps >= len(self.grid) ** 2 * 3:
            done = True

        return self._get_state(), 0, done, {}
```

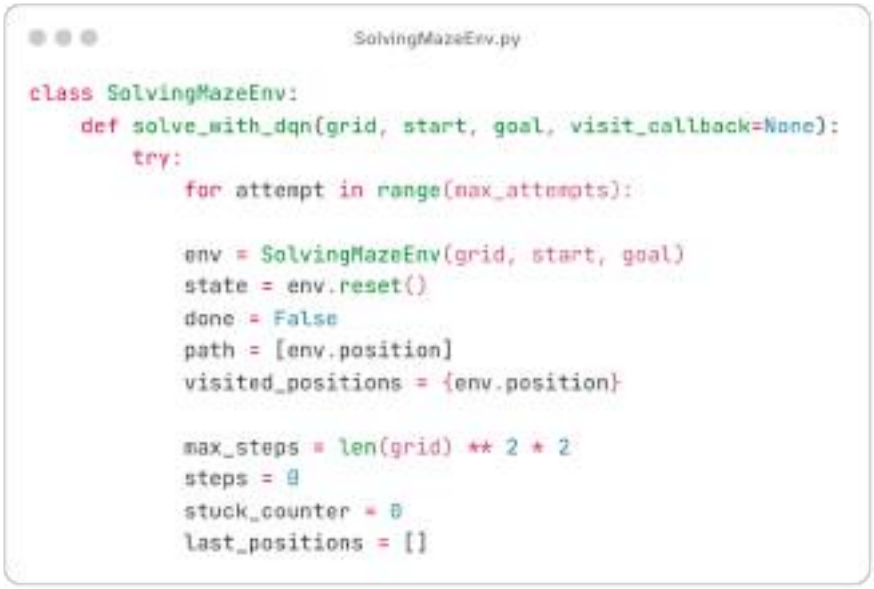
Slika 34: Izvorna koda metode step()

Vir: Lasten

Metoda step(self, action) izvede premik agenta v okolju glede na podano akcijo. Za razliko od učnega okolja ta metoda vrača poenostavljeno nagrado 0 in ne izračunava kompleksnih nagrad, saj se uporablja le za evalvacijo naučenega modela.

Slika 35: Izvorna koda metode solve_with_dqn()

Vir: Lasten



Slika 36: Izvrona koda metode solve_with_dqn()

Vir: Lasten

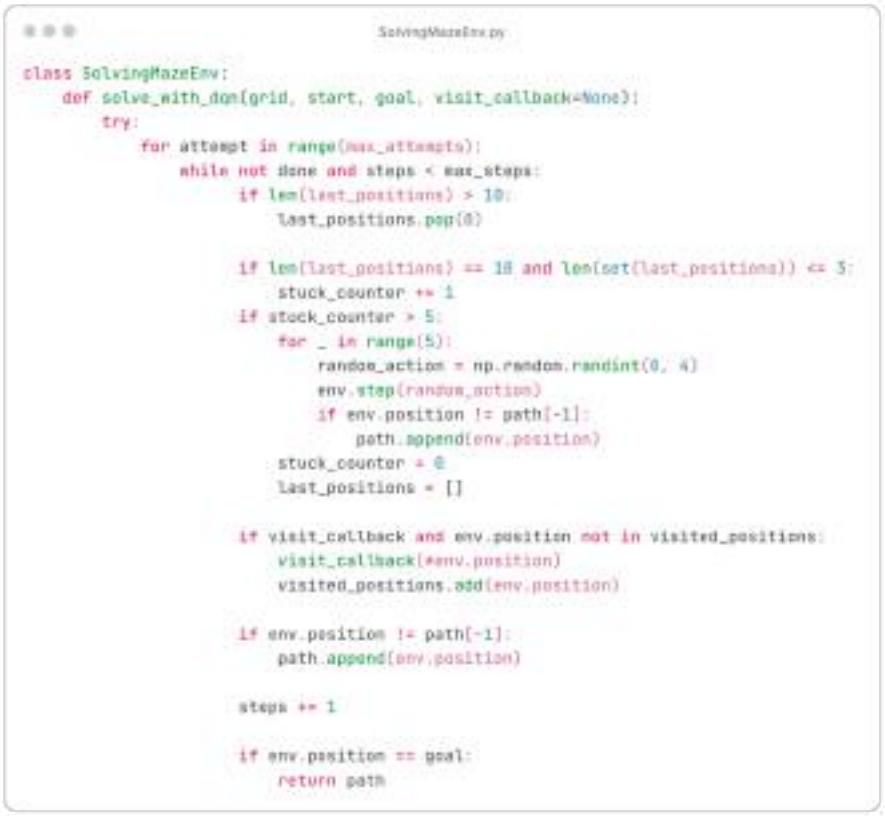
```
class SolvingMazeEnv:
    def solve_with_dqn(grid, start, goal, visit_callback=None):
        try:
            for attempt in range(max_attempts):
                while not done and steps < max_steps:
                    with torch.no_grad():
                        q_values =
                        model(torch.tensor(state,dtype=torch.float32).unsqueeze(0))

                        if steps > 10:
                            noise = torch.randn_like(q_values) * 0.1
                            q_values = q_values + noise
                            action = torch.argmax(q_values).item()

                    old_position = env.position
                    next_state, reward, done, _ = env.step(action)
                    state = next_state
                    last_positions.append(env.position)
```

Slika 37: Izvrona koda metode solve_with_dqn()

Vir: Lasten



```
class SolvingMazeEnv:
    def solve_with_dqn(grid, start, goal, visit_callback=None):
        try:
            for attempt in range(max_attempts):
                while not done and steps < max_steps:
                    if len(last_positions) > 10:
                        last_positions.pop(0)

                    if len(last_positions) == 10 and len(set(last_positions)) <= 3:
                        stuck_counter += 1
                    if stuck_counter > 5:
                        for _ in range(5):
                            random_action = np.random.randint(0, 4)
                            env.step(random_action)
                            if env.position != path[-1]:
                                path.append(env.position)
                        stuck_counter = 0
                        last_positions = []

                    if visit_callback and env.position not in visited_positions:
                        visit_callback(*env.position)
                        visited_positions.add(env.position)

                    if env.position != path[-1]:
                        path.append(env.position)

                    steps += 1

                    if env.position == goal:
                        return path
```

Slika 38: Izvrona koda metode solve_with_dqn()

Vir: Lasten

Slika 39: Izvrona koda metode solve_with_dqn()

Vir: Lasten

Funkcija solve_with_dqn(grid, start, goal, visit_callback=None) uporablja predhodno naučen model DQN za iskanje poti skozi labirint. Funkcija naloži shranjeni model iz datoteke dqn_model.pth, ga postavi v način evalvacije in izvede do tri poskuse iskanja poti. Med iskanjem uporablja epsilon-greedy strategijo z dodanim šumom za izboljšanje raziskovanja in implementira mehanizem zaznavanja, ko se agent zatakne v zanki. Če agent ostane na istem območju predolgo, funkcija izvede naključne premike za izhod iz zanke.

train_dqn.py

```python
import torch
import torch.nn.functional as F
import torch.optim as optim
import random
from collections import deque
import numpy as np
from ai.dqn import DQN
from ai.env import MazeEnv
from maze import Maze

def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=200,
    memory_size=20000
):
```

Slika 40: Izvorna koda metode train_dqn()

Vir: Lasten

```python
def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=200,
    memory_size=20000,
):
    model = DQN(input_size=13)
    target_model = DQN(input_size=13)
    target_model.load_state_dict(model.state_dict())

    optimizer = optim.Adam(model.parameters(), lr=lr)
    memory = deque(maxlen=memory_size)

    episode_rewards = []
    success_count = 0
    best_success_rate = 0

    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=1000, gamma=0.8)
```

Slika 41: Izvorna koda metode train_dqn()

Vir: Lasten

```python
def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=200,
    memory_size=20000,
):
      for episode in range(episodes):
          maze = Maze(size=11)
          grid = maze.generate()
          env = MazeEnv(grid, maze.start, maze.end)
          state = env.reset()
          epsilon = max(epsilon_min, epsilon_start * (epsilon_decay ** episode))
          done = False
          total_reward = 0
          episode_steps = 0
```

Slika 42: Izvorna koda metode train_dqn()

Vir: Lasten

```
def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=250,
    memory_size=20000,
):
    for episode in range(episodes):
        while not done:
            if random.random() < epsilon:
                action = random.randint(0, 1)
            else:
                with torch.no_grad():
                    q_values = model(torch.tensor(state, dtype=torch.float32).unsqueeze(0))
                    if epsilon > 0.1:
                        noise = torch.randn_like(q_values) * 0.01
                        q_values = q_values + noise
                    action = torch.argmax(q_values).item()

            next_state, reward, done, _ = env.step(action)
            memory.append([state, action, reward, next_state, done])
            state = next_state
            total_reward += reward
            episode_steps += 1
```

Slika 43: Izvorna koda metode train_dqn()

Vir: Lasten

```
def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=200,
    memory_size=20000,
):
    for episode in range(episodes):
        while not done:
            if len(memory) >= batch_size and episode % 2 == 0:
                batch = random.sample(memory, batch_size)
                states, actions, rewards, next_states, dones = zip(*batch)

                states = torch.tensor(np.array(states), dtype=torch.float32)
                actions = torch.tensor(actions, dtype=torch.int64)
                rewards = torch.tensor(rewards, dtype=torch.float32)
                next_states = torch.tensor(np.array(next_states), dtype=torch.float32)
                dones = torch.tensor(dones, dtype=torch.bool)

                current_q_values = model(states).gather(1, actions.unsqueeze(1)).squeeze()
```

Slika 44: Izvorna koda metode train_dqn()

Vir: Lasten

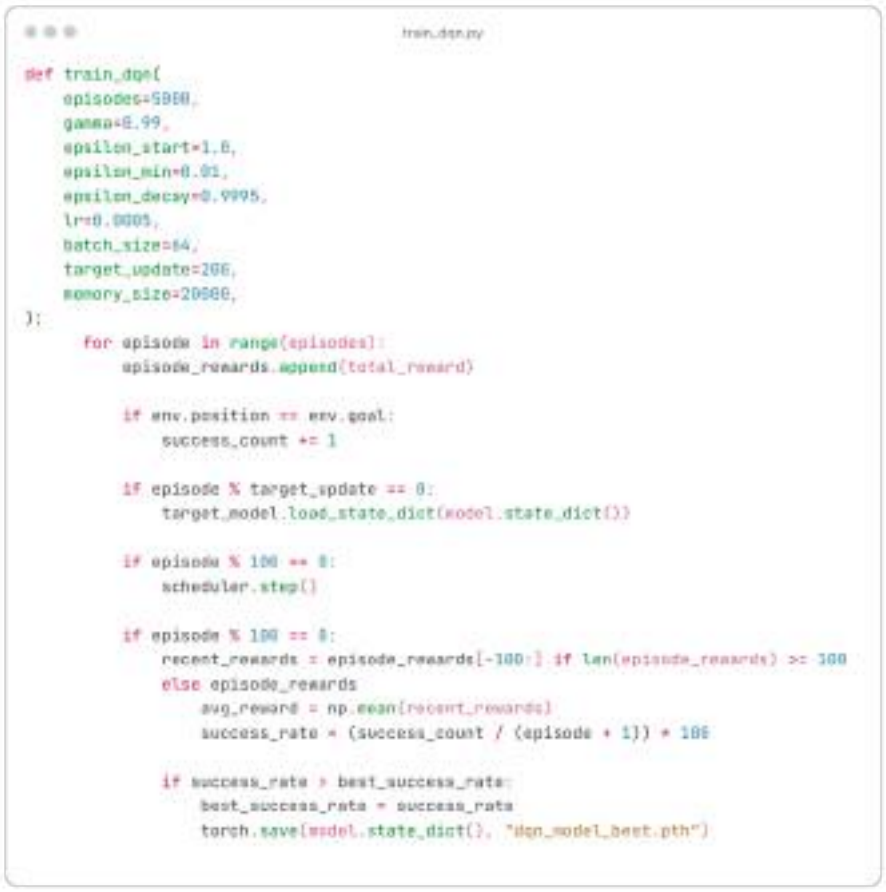
```
def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=200,
    memory_size=20000,
):
      for episode in range(episodes):
          while not done:
                with torch.no_grad():
                    next_actions = model(next_states).max(1)[1]
                    next_q_values =
                    target_model(next_states).gather(1, next_actions.unsqueeze(1)).squeeze()
                    target_q_values = rewards + gamma * next_q_values * (~dones)

                loss = F.smooth_l1_loss(current_q_values, target_q_values)

                optimizer.zero_grad()
                loss.backward()
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                optimizer.step()
```

Slika 45: Izvorna koda metode train_dqn()

Vir: Lasten



```
def train_dqn(
    episodes=5000,
    gamma=0.99,
    epsilon_start=1.0,
    epsilon_min=0.01,
    epsilon_decay=0.9995,
    lr=0.0005,
    batch_size=64,
    target_update=200,
    memory_size=20000,
):
      for episode in range(episodes):
          episode_rewards.append(total_reward)

          if env.position == env.goal:
              success_count += 1

          if episode % target_update == 0:
              target_model.load_state_dict(model.state_dict())

          if episode % 100 == 0:
              scheduler.step()

          if episode % 100 == 0:
              recent_rewards = episode_rewards[-100:] if len(episode_rewards) >= 100
              else episode_rewards
                  avg_reward = np.mean(recent_rewards)
                  success_rate = (success_count / (episode + 1)) * 100

              if success_rate > best_success_rate:
                  best_success_rate = success_rate
                  torch.save(model.state_dict(), "dqn_model_best.pth")
```

Slika 46: Izvorna koda metode train_dqn()

Vir: Lasten

Slika 47: Izvorna koda metode train_dqn()

Vir: Lasten

Funkcija train_dqn implementira algoritem globokega Q-učenja za treniranje nevronske mreže. Funkcija uporablja replay memory za shranjevanje izkušenj, ciljno mrežo za stabilizacijo učenja in epsilon-greedy strategijo za ravnovesje med raziskovanjem in izkoriščanjem. Parametri vključujejo število epizod učenja, diskontni faktor gamma, učno hitrost in velikost batcha. Med učenjem funkcija uporablja Double DQN pristop, kjer glavna mreža izbere akcije, ciljna mreža pa ocenjuje njihove vrednosti. Funkcija implementira postopno zmanjševanje učne hitrosti in shrani model z najboljšo uspešnostjo med učenjem.

9 PRIMERJAVA ALGORITMOV IN UMETNE INTELIGENCE V NAKLJUČNO GENERIRANEM LABIRINTU

Za empirično primerjavo algoritmov iskanja poti smo oblikovali testno okolje s štirimi naključno generiranimi labirinti različnih velikosti, in sicer 11x11, 21x21 in 31x31. Posamezne velikosti predstavljajo različne stopnje kompleksnosti in omogočajo sistematično primerjavo učinkovitosti algoritmov v različnih primerih. Rezultate smo dokumentirali v tabelah za lažjo interpretacijo. Pri vsakem testu smo spremljali ključne parametre, in sicer čas iskanja, dimenzije labirinta, število obiskanih vozlišč in uspešno obiskano ciljno vozlišče. Za objektivno primerjavo algoritmov smo vsak algoritem testirali na tisoč naključno generiranih labirintih za vsako velikost.

V okviru diplomske naloge smo implementirali pristop globokega Q-učenja DQN (angl. Deep Q-network) za simulacijo inteligentnega iskanja. Proces učenja je potekal na obsežnem naboru pet tisoč naključno generiranih labirintov v standardizirani velikosti 11x11. Uspešnost naučenega agenta dosega 32,4 %, kar je pomemben dejavnik pri interpretaciji in analizi nadaljnjih rezultatov diplomskega dela. Pričakujemo lahko, da bo umetna inteligenca imela najboljši rezultat na velikosti labirinta 11x11.

9.1 Rezultati algoritmov iskanja poti in umetne inteligence v naključno generiranem labirintu

Tabela 1: Rezultati algoritmov iskanja poti in UI v labirintu velikosti 11x11

Algoritem	Velikost	Število testiranj	Uspešno najden cilj	Neuspešno najden cilj	Uspeh algoritma(%)	Povprečen čas iskanja (ms)	Povprečno število obiskanih vozlišč
BFS	11	1000	1000	0	100.0	0.029	29
A*	11	1000	1000	0	100.0	0.028	26
Dijkstra	11	1000	1000	0	100.0	0.024	29
UI	11	1000	603	397	60.3	13.47	28

Vir: Lasten

Tabela 2: Rezultati algoritmov iskanja poti in UI v labirintu velikosti 21x21

Algoritem	Velikost	Število testiranj	Uspešno najden cilj	Neuspešno najden cilj	Uspeh algoritma(%)	Povprečen čas iskanja (ms)	Povprečno število obiskanih vozlišč
BFS	21	1000	1000	0	100.0	0.071	115
A*	21	1000	1000	0	100.0	0.107	105
Dijkstra	21	1000	1000	0	100.0	0.089	120
UI	21	1000	81	919	8.1	55.80	84

Vir: Lasten

Tabela 3: Rezultati algoritmov iskanja poti in UI v labirintu velikosti 31x31

Algoritem	Velikost	Število testiranj	Uspešno najden cilj	Neuspešno najden cilj	Uspeh algoritma(%)	Povprečen čas iskanja (ms)	Povprečno število obiskanih vozlišč
BFS	31	1000	1000	0	100.0	0.16	267
A*	31	1000	1000	0	100.0	0.26	242
Dijkstra	31	1000	1000	0	100.0	0.22	266
UI	31	1000	28	297	2.8	288.94	105

Vir: Lasten

9.2 Analiza hitrosti algoritmov iskanja najkrajše poti

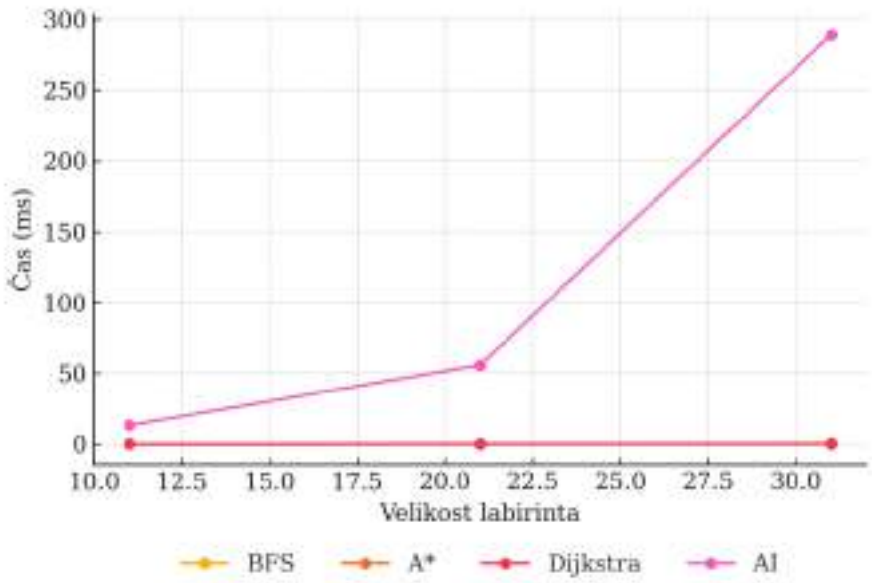
Rezultati diplomskega dela so pokazali na razlike med algoritmi iskanja poti in umetno inteligenco. Algoritmi iskanja poti izkazujejo hitro delovanje v vseh testnih primerih, medtem ko umetna inteligenca zahteva več časa za morebitno doseganje cilja. Umetna inteligenca je bila učena na labirintu velikosti 11x11, kar tudi odraža njen najboljši rezultat izmed vseh testnih primerov.

Pri labirintih velikosti 11x11 je povprečni čas iskanja 0,02 ms, pri labirintih 21×21 pa med 0,07 in 0,10 ms. Pri labirintih velikosti 31x31 je najhitrejši Dijkstrov algoritem, in sicer 0,024 ms. Hitrost UI upada z večanjem števila vozlišč in velikosti labirinta. Iz tega lahko sklepamo, da je učinkovitost algoritmov odvisna od velikosti labirinta in števila vozlišč.

Kljub podobnim časom iskanja je algoritem A* obiskal najmanjše število vozlišč za dosego cilja, kar nakazuje na bolj optimizirano delovanje algoritma v primerjavi z BFS, Dijkstro in umetno inteligenco.

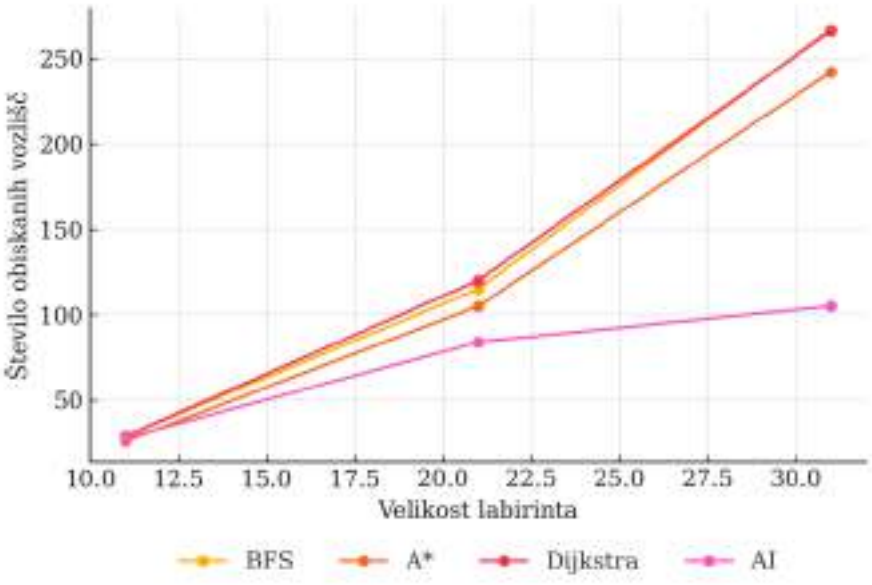
Najhitrejši algoritem je Dijkstrov algoritem. Zabeležil je najkrajši čas iskanja, vendar je treba poudariti, da je obiskal največ vozlišč. To nakazuje na potencialno poslabšanje optimalnosti algoritma na večjih labirintih, kjer je število vozlišč večje. Rezultati hitrosti in obiskanih vozlišč algoritmov in UI so razvidni v spodnjih grafih.

Grafikon 1: Prikaz časa iskanja končnega vozlišča algoritmov in UI v labirintu



Vir: Lasten

Grafikon 2: Prikaz števila obiskanih vozlišč algoritmov in UI v labirintu

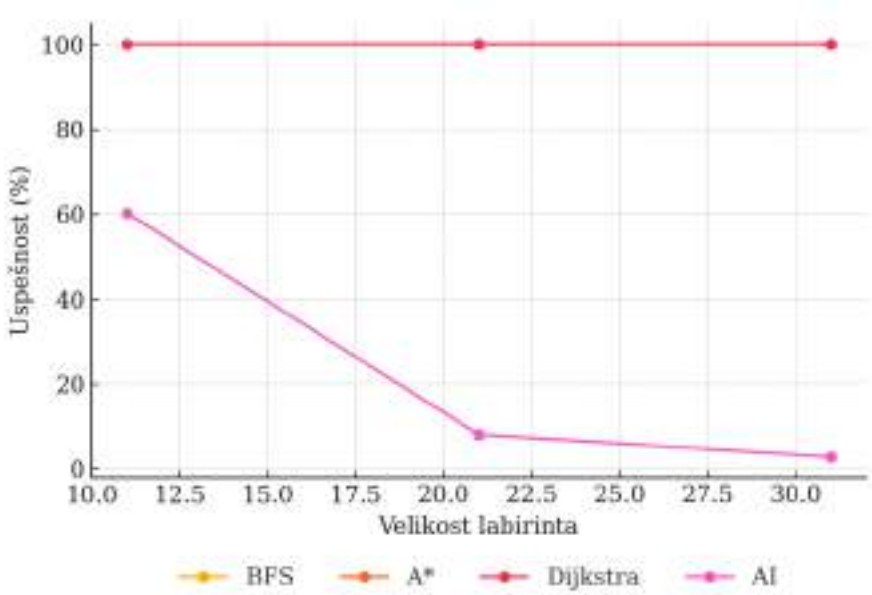


Vir: Lasten

9.3 Analiza zmogljivosti umetne inteligence pri reševanju najkrajše poti

Umetna inteligenca je bila v primerjavi z drugimi algoritmi najpočasnejša in neučinkovita. Rezultati nakazujejo na slabše rezultate v labirintih, kjer velikost ni 11x11. To je bilo pričakovano, saj je umetna inteligenca bila učena izključno na labirintih velikosti 11x11. Stopnja uspešnosti UI se zmanjšuje z naraščajočo kompleksnostjo problema. Pri majhnih labirintih velikosti 11x11 dosega 60,3-% uspešnost s 603 uspešnimi rešitvami od 1000 testov. Ta uspešnost se zmanjša pri srednjih labirintih velikosti 21×21 na 8,1-% z 81 uspešnimi rešitvami. Pri največjih labirintih velikosti 31×31 UI dosega le 2,8-% uspešnost z zgolj 28 uspešnimi rešitvami od 1000 testov, kar pomeni, da ne uspe rešiti preko 97,2 % testnih primerov. Uspešnost UI in algoritmov je razvidna v spodnjem grafu.

Grafikon 3 :Prikaz uspešnosti iskanja algoritmov in UI v labirintu



Vir: Lasten

10 POTRDITEV ALI ZAVRNITEV HIPOTEZ

10.1 H1: A je najpopularnejši algoritem*

V diplomskem delu smo pregledali znanstveno literaturo na področju algoritmov iskanja poti. Pregled literature nakazuje na vsesplošno razširjenost algoritma A* v raziskovalni skupnosti. Foed idr. (2021) so navedli, da je algoritem A* zaradi svoje priljubljenosti in razširjenosti standard za raziskovalce, ki raziskujejo in rešujejo problem iskanja poti. Trditev, da je algoritem A* najpopularnejši, zasledimo tudi v raziskavi, ki so jo opravili Iskandar, U. A. S. idr. (2020), kjer navedejo, da je algoritem A* najpogosteje uporabljen algoritem iskanja poti v videoigrah.

10.2 H2: Obstaja umetna inteligenca, ki najde najkrajšo pot tudi iz takega algoritma, pri katerem imajo ostali algoritmi omejitve

Rezultati diplomskega dela so pokazali, da UI najde najkrajšo pot v labirintu kljub predpostavljenim omejitvam. Literatura nakazuje, da je razvoj UI pri problemu iskanja najkrajše poti pomemben dejavnik. V raziskavi, ki so jo opravili Iskandar, U. A. S. idr. (2020), ugotovimo, da je UI prisotna pri iskanju poti v videoigrah. V raziskavi so zabeležili, da agenti najdejo pot do igralca, vendar je učinkovitost algoritmov slabša.

10.3 H3: A algoritem bo najhitrejši algoritem in bolj učinkovit pri iskanju najkrajše poti v naključno generiranih labirintih v primerjavi z drugimi algoritmi*

V poglavju Primerjava algoritmov in umetne inteligence v naključno generiranem labirintu smo ugotovili, da je algoritem A* najučinkovitejši algoritem z obiskom najmanjšega števila vozlišč za dosego cilja. Rezultati diplomskega dela nakazujejo, da je algoritem A* najučinkovitejši algoritem v primerjavi z ostalimi testnimi algoritmi in umetno inteligenco.

Rezultati diplomskega dela so pokazali, da algoritem A* kljub svoji učinkovitosti ni najhitrejši algoritem izmed testiranih algoritmov in umetne inteligence.

10.4 H4: Umetna inteligenca bo najučinkovitejša pri iskanju najkrajše poti v naključno generiranem labirintu v primerjavi z algoritmi A, Dijkstrovim algoritmom in BFS*

Rezultati diplomskega dela so pokazali, da umetna inteligenca ni najučinkovitejša v naključno generiranem labirintu. Hitrost reševanja problema je bila znatno višja v pram algoritmov iskanja poti. Število obiskanih vozlišč je bilo v povprečju višje od algoritmov iskanja poti. Pomemben podatek je, da UI ni bila vedno uspešna pri reševanju problema, kar nakazuje na splošno slabo učinkovitost UI. Kljub rezultatom je pomembno povedati, da je v okviru diplomskega dela UI bila učena na manjši količini podatkov, in sicer pet tisoč naključno generiranih labirintov velikosti 11x11.

11 SKLEP

V okviru diplomskega dela smo primerjali algoritme iskanja poti BFS, A*, Dijkstrov algoritem in umetno inteligenco pri reševanju naključno generiranega labirinta. Cilj diplomskega dela je analiza literature in analitična primerjava uspešnosti algoritmov pri reševanju naključno generiranega labirinta. Analiza algoritmov in umetne inteligence je zajemala čas reševanja labirinta, število obiskanih vozlišč in sposobnost algoritma, da najde končno vozlišče. Algoritme smo primerjali na različnih velikostih labirintov, in sicer 11x11, 21x21 in 31x31, z namenom simulacije različnih kompleksnosti problemov.

Algoritmi iskanja poti so v vseh testnih primerih pokazali izjemno zanesljivost s 100-% uspešnostjo pri iskanju cilja. Algoritem A* se je izkazal kot najbolj optimiziran, saj je dosegel cilj z najmanjšim številom obiskanih vozlišč, kar nakazuje na uspešno uporabo hevristične funkcije. Dijkstrov algoritem je bil najhitrejši, vendar je obiskal največ vozlišč, kar nakazuje na potencialne težave pri večjih in zahtevnejših problemih. Algoritem BFS je pokazal konsistentno delovanje z zmernim obiskom vozlišč in hitrostjo reševanja problema.

Umetno inteligenco smo implementirali s pomočjo strojnega učenja na osnovi globokega Q-učenja DQN. Za namen diplomskega dela smo umetno inteligenco učili na naključno generiranih labirintih velikosti 11x11. Dosežen uspeh učenja UI je 32,4 %, kar moramo upoštevati pri končnih rezultatih primerjave algoritmov iskanja poti in umetne inteligence. Velikost labirinta je močno vplivala na uspeh reševanja problema UI. Najboljši časovni rezultat UI je zabeležen na labirintih velikosti 11x11, na katerih je bila UI učena. Uspešno dosežen cilj UI je bil najvišji na velikosti 11x11, kjer je število vozlišč minimalno. UI je imela znatno slabše rezultate na labirintu velikosti 21x21 in 31x31, kar nakazuje na slabšanje učinkovitosti UI na kompleksnejših problemih. Takšni rezultati so bili pričakovani, saj je strojno učenje najučinkovitejše na podatkih, na katerih je bilo učeno.

Algoritmi iskanja poti so hitreje reševali problem v pram UI. Čas reševanja problema smo beležili v milisekundah. Primerjavo algoritmov iskanja poti in UI smo izvedli na tisoč primerih na naključno generiranih labirintih različne velikosti. Algoritmi iskanja poti so bili bistveno hitrejši v primerjavi z UI, saj je morala izračunati Q-vrednost za vsako novo stanje.

Uporaba algoritmov iskanja poti v praktičnem okolju je še vnaprej zaželena, saj je potrebna 100-% učinkovitost in hitra odzivnost pri reševanju problema. Njihova deterministična narava

in dokazana učinkovitost jih delata zanesljive za kritične aplikacije. Vendar pa pristopi umetne inteligence ponujajo potencial za učenje iz izkušenj, kar lahko postane prednost v dinamičnih okoljih, kjer se pogoji spreminjajo.

Pomembno je poudariti, da so rezultati umetne inteligence odvisni od kakovosti učnega procesa, količine učnih podatkov in arhitekture nevronske mreže. Z dodatnim učenjem na raznolikih labirintih različnih velikosti bi se lahko njena splošna uspešnost občutno izboljšala.

Izbira med algoritmi iskanja poti in UI je odvisna od problema, ki ga želimo rešiti, zahtevane zanesljivosti, časovnih omejitev in narave problema. Algoritmi iskanja poti so primerni za aplikacije, ki zahtevajo 100-% uspešnost in hitro reševanje. UI ponuja potencialno prilagajanje pri reševanju kompleksnih dinamičnih problemov, vendar z večjimi računskimi zahtevami in manjšim zagotavljanjem uspešnosti. Prednost UI je neprestano učenje na vnosnih podatkih, kar omogoča izboljšanje časa reševanja problema in učinkovitost rešitve.

Za nadaljnje raziskovanje UI priporočamo testiranje z drugimi algoritmi strojnega učenja, drugačnimi parametri učenja, večjim številom labirintov in različnimi testnimi okolji. Prav tako je smiselno raziskati hibridne pristope, ki bi kombinirali prednosti deterministčnih algoritmov in prilagodljivosti UI.

12 VIRI IN LITERATURA

Agarwal, V., Petrini, F., Pasetto, D., &in Bader, D. (2010). Scalable Graph Exploration on Multicore Processors. *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*.

Aria, M. (2018). New algorithm for digital way-finding map. *IOP Conference Series Materials Science and Engineering*.

Barnouti, N., Al-Dabbagh, S., &in Naser, M. (2016). *Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm.* Journal of Computer and Communications.

Beamer, S., Asanović, K., &in Patterson, D. (2013). Direction-Optimizing Breadth-First Search. *Scientific Programming*.

Bender, E., &in Williamson, S. (2010). *Lists, Decisions and Graphs With an Introduction to Probability.*

Berge, C. (1958). *Sculptor of Graph Theory.* Trends in Mathematics.

Brodsky, S. (14. Oktober 2024). *The hidden costs of AI: How generative models are reshaping corporate budgets*. Pridobljeno iz IBM: https://www.ibm.com/think/insights/ai-economics-compute-cost

Buluç, A., &in Madduri, K. (2011). Parallel breadth-first search on distributed memory systems. *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*.

Carr, J. (2014). *An Introduction to Genetic Algorithms.* Journal of Computer and Communications.

Chassignol, M., Khoroshavin , A., Klimova, A., &in Bilyatdinova , A. (2018). Artificial Intelligence trends in education: a narrative overview. *Procedia Computer Science*.

Choudhury, A., Ghose, M., Islam, A., in& Yogita. (2024). *Machine learning-based computation offloading in multi-access edge computing: A survey.* Journal of Systems Architecture.

Cormen, T. H., Leiserson, C. E., Rivest, L. R., in& Clifford , S. (2022). *Introduction to Algorithms.* MIT Press.

Dijkstra, E. W. (1959). *A note on two problems in connexion with graphs.* Numerische Mathematik .

Elkari, B., Ourabah, L., Sekkat, H., Hsaine, A., Essaiouad, C., Bouargane, Y., &in El Moutaouakil, K. (2024). Exploring Maze Navigation: A Comparative Study of DFS, BFS, and A* Search Algorithms. *Statistics, Optimization &in Information Computing* .

Felner, A. (2011). Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm. *Proceedings of the International Symposium on Combinatorial Search*.

Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T., &in Koenig, S. (2018). *Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding.* PKP Publishing Services Network.

Foead , D., Ghifari, A., Kusuma, M. B., Hanafiah, N., &in Gunawan , E. (2021). *A Systematic Literature Review of A* Pathfinding.* Procedia Computer Science.

Fredman, M., &in Tarjan, R. (1984). Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. *25th Annual Symposium onFoundations of Computer Science, 1984.*

Harabor, D., &in Grastien, A. (2011). *Online Graph Pruning for Pathfinding on Grid Maps.* NICTA and The Australian National University.

Hong, S., Oguntеb, T., &in Olukotun, K. (2011). Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. *Parallel Architectures and Compilation Techniques(PACT)*.

Hunkeler, I., Schär, F., Dornberger, R., &in Hanne, T. (2016). *fairGhosts — Ant colony controlled ghosts for Ms. Pac-Man.* IEEE.

Iskandar, U., Diah, N., &in Ismail, M. (2020). Identifying Artificial Intelligence Pathfinding Algorithms for Platformer Games. *2020 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*.

Kairanbay , M., &in Jani, H. (2013). A Review and Evaluations of Shortest Path Algorithms. *International Journal of Scientific &in Technology Research*.

Kumar, N. (2019). Bidirectional Graph Search Techniques for Finding Shortest Path in Image Based Maze Problem. *Guru Nanak Dev University* .

Kurt, M., &in Sanders, P. (2008). *Algorithms and Data Structures.*

Lawande, S., Jasmine, G., Anbarasi, J., &in Izhar, L. (2022). A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games. *Applications of Evolutionary Computation to Machine Learning and Data Mining*.

Le, H. (27. April 2025). *How Much Does AI Cost in 2025? A Complete Breakdown for Businesses*. Pridobljeno iz ekotek: https://ekotek.vn/how-much-does-ai-cost

Liu, B., Choo, S.-H., Lok, S.-L., Leong, S.-M., Lee, S.-C., &in Poon, F.-P. (1994). Integrating case-based reasoning, knowledge-based approach and Dijkstra algorithm for route finding. *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*.

Lutkevich, B. (12. Maj 2022). *knowledge engineering*. Pridobljeno iz techtarget: https://www.techtarget.com/searchenterpriseai/definition/knowledge-engineering

Nico. (29. October 2020). *An Analysis of Dijkstra's Algorithm*. Pridobljeno iz medium: https://medium.com/@nico_72892/an-analysis-of-dijkstras-algorithm-fcd64fc827c7

Noori, A., &in Moradi, F. (2015). Simulation and Comparison of Efficency in Pathfinding algorithms in Games. *Pproceedings of the 8th international conference on engineering, technology, and industrial applications 2021 (8th icetia 2021)*.

Noto, M., &in Sato, H. (2000). A method for the shortest path search by extended Dijkstra algorithm. *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0*.

Rachmawati , D., &in Gustin, L. (2020). Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem. *IOP Publishing Ltd*.

Rafiq , A., &in Kadir, T. (2020). Pathfinding Algorithms in Game Development. *IOP Conference Series Materials Science and Engineering*.

Rafiq, A., Tuty Asmawaty, K. A., in& Ihsan, S. N. (2020). *Pathfinding Algorithms in Game Development.* IOP Publishing Ltd.

Russell , S., &in Norvig, P. (2022). *Artificial Intelligence A Modern Approach 4th edition.* Pearson Education Limited.

Schrijver, A. (2012). *On the history of the shortest path problem.* EMS Press eBooks.

Soltani, A., Tawfik, H., Goulermas, J., &in Fernando, T. (2002). Path planning in construction sites: performance evaluation of the Dijkstra, A∗, and GA search algorithms. *Advanced Engineering Informatics*.

Stryker, C., &in Kavlakoglu, E. (9. Avgust 2024). *What is artificial intelligence (AI)?* Pridobljeno iz IBM: https://www.ibm.com/think/topics/artificial-intelligence

Szcześniak, I., &in Woźna-Szcześniak, B. (2023). Generic Dijkstra: correctness and tractability. *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*.

Szcześniak, I., Jajszczyk, A., &in Woźna-Szcześniak, B. (2019). Generic Dijkstra for optical networks. *Journal of Optical Communications and Networking*.

Wahyuningsih, D., &in Syahreza , E. (2018). Shortest Path Search Futsal Field Location With Dijkstra Algorithm. *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)*.

Wolsey, L. A. (1998). *Integer Program.* A Willey-Interscience Publication.

Yap, P. (2002). Grid-Based Path-Finding. *Department of Computing Science, University of Alberta*.

Yiu, Y., Du, J., &in Mahapatra, R. (2018). Evolutionary Heuristic A* Search: Heuristic Function Optimization via Genetic Algorithm. *IEEE*.

Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., &in Umit , C. (2005). A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. *Association for Computing Machinery*.