VEGOVA
ELEKTROTEHNIŠKO-RAČUNALNIŠKA
STROKOVNA ŠOLA IN GIMNAZIJA
LJUBLJANA

Seminarska naloga pri računalništvu

Izdelava iger: primerjava algoritmov iskanja poti

Mentor: Aleš Volčini, prof.

Avtor: Lovro Hauptman, G 4. B

Ljubljana, april 2024

Povzetek

Algoritmi za iskanje poti imajo ključno vlogo na različnih področjih, kot so na primer navigacija, robotika, video igre in matematika. S pomočjo teh algoritmov nam lahko aplikacije, kot je Google Maps, napišejo natančna navodila, kako priti iz lokacije A na lokacijo B. V tej seminarski nalogi sem predstavil in razložil delovanje petih različnih algoritmov za iskanje poti. Z uporabo grafične predstavitve na problemu labirinta sem prikazal njihovo delovanje na bolj intuitiven način. Poleg tega sem izvedel poglobljeno primerjavo med njimi.

Ključne besede: algoritmi iskanja poti, Dijkstrov algoritem, A*, DFS, BFS.

Abstract

Pathfinding algorithms play a crucial role in various fields, such as navigation, robotics, video games, and mathematics. With the help of these algorithms, applications like Google Maps can provide precise instructions on how to navigate from location A to location B. In this seminar paper, I will present and explain the operation of five different pathfinding algorithms. Using graphical representations on the problem of a maze, I will illustrate their operation in a more intuitive manner. Additionally, I will conduct an in-depth comparison between them.

Keywords: pathfinding algorithms, Dijkstra's algorithm, A*, DFS, BFS.

Kazalo

1. Uvod

Algoritmi za iskanje poti so že dolgo del različnih aplikacij in video iger. Znanstvena literatura že zajema številne raziskave in primerjalne analize teh algoritmov, vendar se njihova uporaba in učinkovitost še vedno raziskujejo in izboljšujejo, saj je njihova učinkovitost odvisna od raznih faktorjev. V tej seminarski nalogi se bom osredotočil na analizo in primerjavo algoritmov iskanja poti na problemu labirinta.

Namen seminarske naloge je sistematično primerjati pet različnih algoritmov iskanja poti in razumeti, kako delujejo ter identificirati najustreznejši algoritem za določeno vrsto aplikacije oz. igre.

Glavni cilji te seminarske naloge vključujejo:

- razumevanje osnovnih konceptov in delovanja petih izbranih algoritmov za iskanje poti,
- implementacijo teh algoritmov v aplikacijo za reševanje labirintov in
- izvedbo primerjalne analize njihove učinkovitosti ter zmogljivosti.

Pri raziskovanju bom obravnaval labirinte z eno možno rešitvijo in labirinte z več rešitvami ter uporabil različne metode za ocenjevanje učinkovitosti algoritmov, vključno z analizo časa izvajanja, številom obiskanih vozlišč, dolžino najkrajše poti ter porabo pomnilnika.

Za doseganje teh ciljev bom razvil aplikacijo za reševanje labirintov, ki omogoča preizkušanje in primerjavo izbranih algoritmov v različnih scenarijih. S pomočjo te aplikacije bom pridobil vpogled v delovanje algoritmov in določil njihove prednosti in slabosti v praksi.

2. Teoretično ozadje

V tem delu seminarske naloge bom na kratko predstavil teoretično ozadje, ki nam bo pomagalo pri razumevanju in primerjavi izbranih algoritmov iskanja poti.

2.1 O-notacija (ang. Big O notation)

O-notacija nam pove, koliko časa se bo nek program oz. algoritem pri danem vhodu (označen z n) izvajal, preden bo vrnil rešitev. Čas običajno merimo v številu operacij, ki jih program izvede. Ker je natančno število operacij težko dobiti, uporabimo O-notacijo, ki označuje red rasti (funkcijo) problema. Nekaj pogostih časovnih zahtevnosti je:

- $O(1)$ – konstantna,
- $O(\log n)$ -logaritemska,
- $O(n)$ – linearna,
- $O(n \log n)$ – vmesna,
- $O(n^2)$ – kvadratna.

2.2 Teorija grafov

Teorija grafov je matematična disciplina, ki se ukvarja z raziskovanjem grafov. Graf je matematična struktura, ki je sestavljen iz množice točk, imenovanih vozlišča (ang. nodes) ter povezav, ki jih povezujejo. Vozlišča lahko predstavljajo različne strukture, kot so mesta na zemljevidu, osebe v socialnem omrežju ali računalnik v računalniškem omrežju.

Grafi so lahko usmerjeni ali neusmerjeni, če so povezave med vozlišči usmerjene oz. neusmerjene. Prav tako so lahko povezani, če obstaja pot med vsakim parom vozlišč oz. nepovezani, če to ne velja.

Grafe lahko delimo tudi na uteženе (ang. weighted) in neuteženе. Povezave uteženih grafov so opremljene z določenimi vrednostmi (na primer razdaljo, kapaciteto, itd.), ki vplivajo na določitev najkrajše poti med vozlišči (prometnost ceste pri navigaciji).

2.3 Klasifikacija algoritmov iskanja poti

Algoritmi iskanja poti so natančno določena navodila (algoritmi), ki določijo najkrajšo pot med dvema točkama v danem omrežju ali okolju. V okviru teorije grafov ti algoritmi igrajo ključno vlogo pri identifikaciji najbolj učinkovite poti od začetnega vozlišča do ciljnega vozlišča. Teorija grafov nam pomaga razumeti, kako so različne točke oz. vozlišča povezana med seboj, kar je zelo koristno pri iskanju najkrajše poti med njimi.

Algoritmi iskanja poti sistematično prehajajo skozi vozlišča in povezave grafa, raziskujejo različne poti in nam vrnejo najkrajšo pot.

Uporabljajo se v različnih področjih, od računalništva in robotike, do prometa in logistike. V računalništvu so algoritmi iskanja poti bistveni sestavni deli navigacijskih sistemov (Google Maps), pri računalniških omrežjih ter pri programiranju likov, ki niso igralci (NPC - Non-Player Character).

Eden izmed glavnih izzivov pri iskanju poti je vzpostaviti ravnovesje med optimizacijo kakovosti rešitve in uporabo računalniških virov. Nekateri algoritmi se osredotočajo na zagotavljanje visoke kakovosti rešitve, ne glede na to, koliko računalniških virov je za to potrebih (na primer, število operacij ali količino pomnilnika), medtem ko se drugi algoritmi bolj osredotočajo na učinkovito uporabo računalniških virov, pri čemer lahko rešitve niso nujno najkrajše.

2.4 Teoretični pregled izbranih algoritmov

2.4.1 Dijkstrov algoritem (ang. Dijkstra's Algorithm)

Dijkstrov algoritem, imenovan po Edsgerju W. Dijkstri, je algoritem iskanja najkrajših poti med vozlišči v grafu. Algoritem deluje tako, da sistematično raziskuje sosede določenega vozlišča (v primeru labirinta je to začetek labirinta) in določi najkrajšo pot do vsakega drugega vozlišča v grafu.

Algoritem začne z grafom G, predstavljenim kot množica vozlišč V in povezav E. Vsakemu vozlišču priredi začasno vrednost razdalje - razdaljo začetnega vozlišča nastavi na 0, ostale pa ne neskončno.

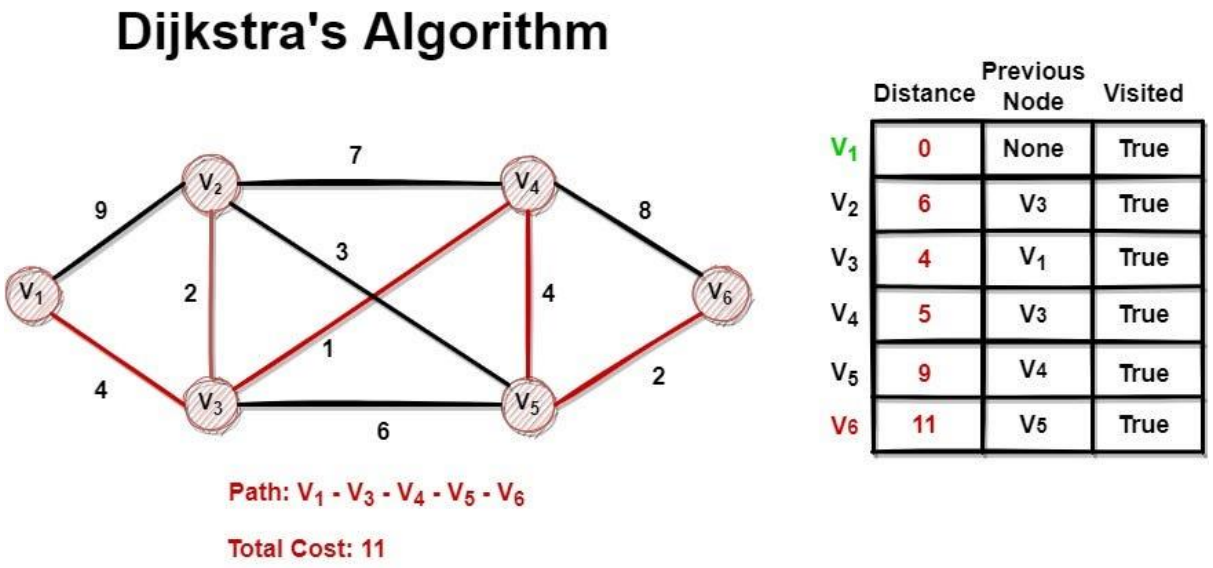
Za vsako vozlišče preuči vse njegove sosede in izračuna njihovo začasno razdaljo preko trenutnega vozlišča. Če je izračunana razdalja manjša od trenutne vrednosti, jo posodobi. Razdaljo izračuna po enačbi:

$$d(v) = min\big(d(v), d(u) + w(u,v)\big)$$

Tu $d(v)$ predstavlja trenutno najkrajšo razdaljo od začetnega vozlišča do ciljnega vozlišča v, $d(u)$ predstavlja najkrajšo razdaljo od začetnega vozlišča do trenutnega vozlišča u, preko katerega preverjamo pot do ciljnega vozlišča v in $w(u,v)$ predstavlja utež povezave med vozliščema u in v.

Nato algoritem iz množice vozlišč, ki še niso bile obiskane, izbere tisto vozlišče, ki ima najmanjšo začasno razdaljo, kot naslednje vozlišče, ki ga bo obiskal.

Postopek ponavlja za vsa vozlišča, dokler niso obiskana vsa in določene njihove končne najkrajše razdalje. Če algoritem išče neko določene vozlišče, se postopek konča, ko ga najde. Postopek Dijkstrovega algoritma je prikazan na spodnji sliki.



Slika 1: prikaz Dijkstrovega algoritma
*(vir:https://miro.medium.com/v2/resize:fit:720/format:webp/0*3FMvRMEr3Sv6NkWt.jpeg)*

2.4.2 Algoritem A*

2.4.2.1 Definicija osnovnih pojmov

- Hevristika (ang. heuristics) - funkcija, ki oceni strošek dosega ciljnega vozlišča iz trenutnega vozlišča (ocena razdalje). Pogosto se za to uporablja Evklidova enačba za izračun razdalje med dvema točkama.
- Iskanje z enotno ceno (ang. uniform-cost search) - strategija iskanja, ki išče pot z najnižjim skupnim stroškom, brez upoštevanja hevristike
- Pohlepno iskanje po najboljšem vozlišču (ang. greedy best-first search) - strategija iskanja, ki se osredotoča na razširjanje vozlišč (preverba sosednjih vozlišč tega vozlišča) na podlagi najmanjše ocene hevristike.

Algoritem A* je hevristični iskalni algoritem, ki poišče najkrajšo pot med začetnim in končnim vozliščem v grafu. Kombinira lastnosti iskanja z enotno ceno in pohlepnega iskanja po

najboljšem vozlišču. Algoritem združuje prednosti obeh strategij, kar mu omogoča učinkovito raziskovanje iskalnega prostora.

Algoritem A* vsakemu vozlišču dodeli dve vrednosti stroškov:

- $g(n)$ - dejanski strošek od začetnega vozlišča do vozlišča n,
- $h(n)$ - predvideni strošek (hevristična funkcija) najcenejše poti od vozlišča n do cilja.

Skupni strošek nekega vozlišča je vsota teh dveh vrednosti:

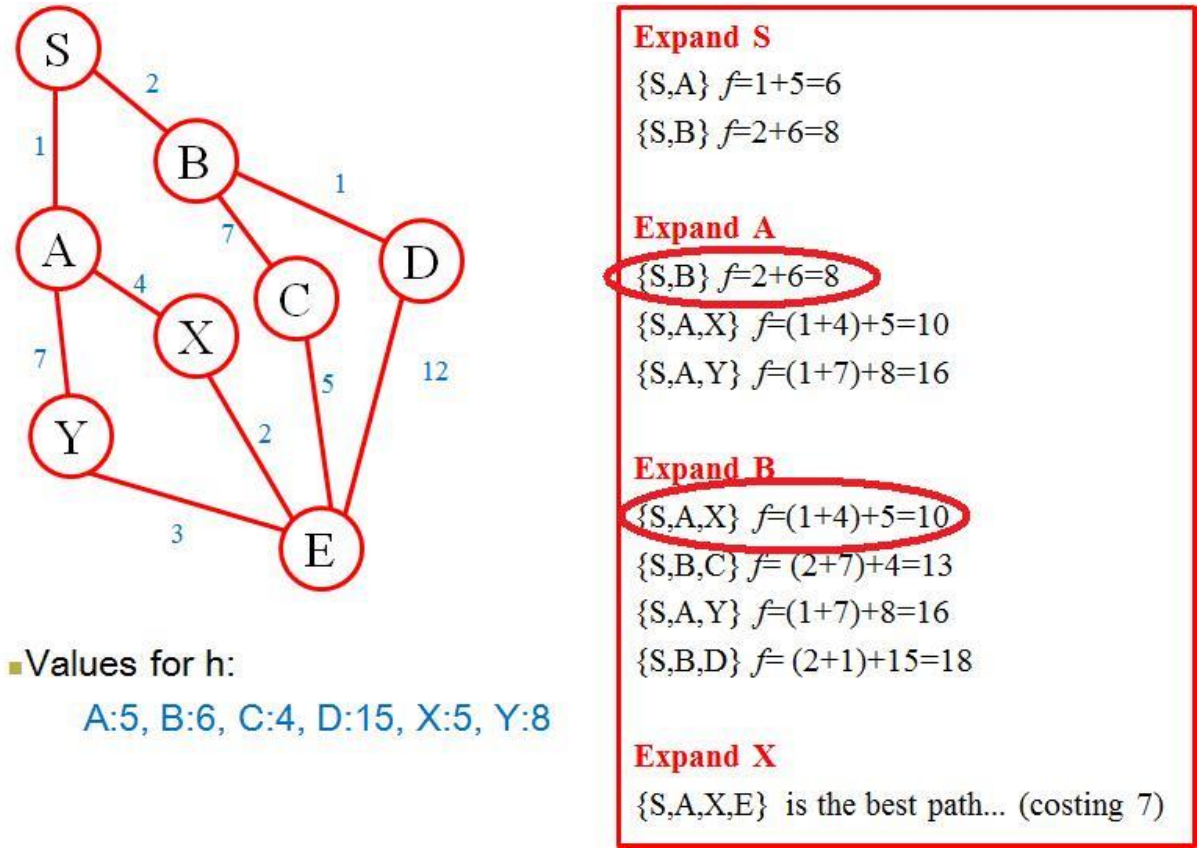
$$f(n) = g(n) + h(n)$$

Pri tem vzdržuje dva seznama:

- odprt seznam - vsebuje vozlišča, ki so bila obiskana, a ne razširjena,
- zaprt seznam - vsebuje vozlišča, ki so bila obiskana in razširjena.

Pri vsakem koraku algoritem izbere vozlišče, ki ima najmanjšo $f(n)$ vrednost iz odprtega seznama in se nanj razširi - izračuna vse $f(n)$ vrednosti za sosede tega vozlišča ter jih doda na odprt seznam, če tam še niso, trenutno vozlišče pa premakne na zaprt seznam.

Postopek se nadaljuje, dokler se ne doseže ciljno vozlišče ali če na odprtem seznamu ni več nobenega vozlišča, kar kaže, da ni poti od začetnega vozlišča do cilja. Postopek algoritma A* je prikazan na spodnji sliki.



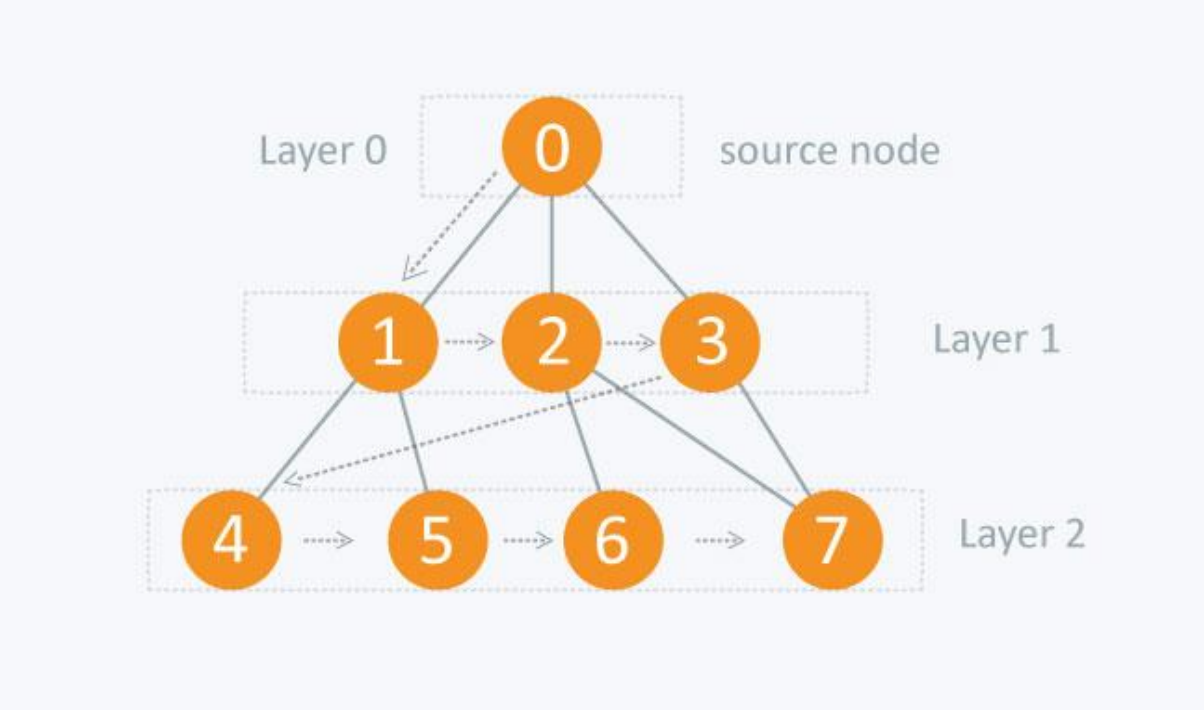
Slika 2: prikaz algoritma A (vir: https://miro.medium.com/v2/resize:fit:720/format:webp/0*LSXBwjDHVA3csu7C.jpg)*

2.4.3 Iskanje najprej v širino (ang. Breadth-First Search - BFS)

BFS je algoritem iskanja poti v grafu, ki sistematično raziskuje oz. preveri vsa vozlišča v grafu, pri čemer začne z začetnim vozliščem in se nato premika po vse bolj oddaljenih vozliščih, dokler ne najde ciljnega vozlišča ali pa je obiskal vsa vozlišča.

Deluje tako, da najprej poišče vsa sosednja vozlišča od začetnega vozlišča, nato pa se premika na vozlišča, ki so eno vozlišče stran od začetnega vozlišča, nato na dve vozlišči stran od začetnega in tako naprej. Algoritem torej obišče vsa vozlišča na ravni k, preden se premakne na vozlišča $k + 1$.

Pri izvajanju si pomaga z vrsto (ang. queue), ki hrani vozlišča, ki jih je treba pogledati. Vrsta zagotavlja, da algoritem vozlišča pregleda v pravilnem vrstnem redu. Postopek algoritma BFS je prikazan na spodnji sliki.



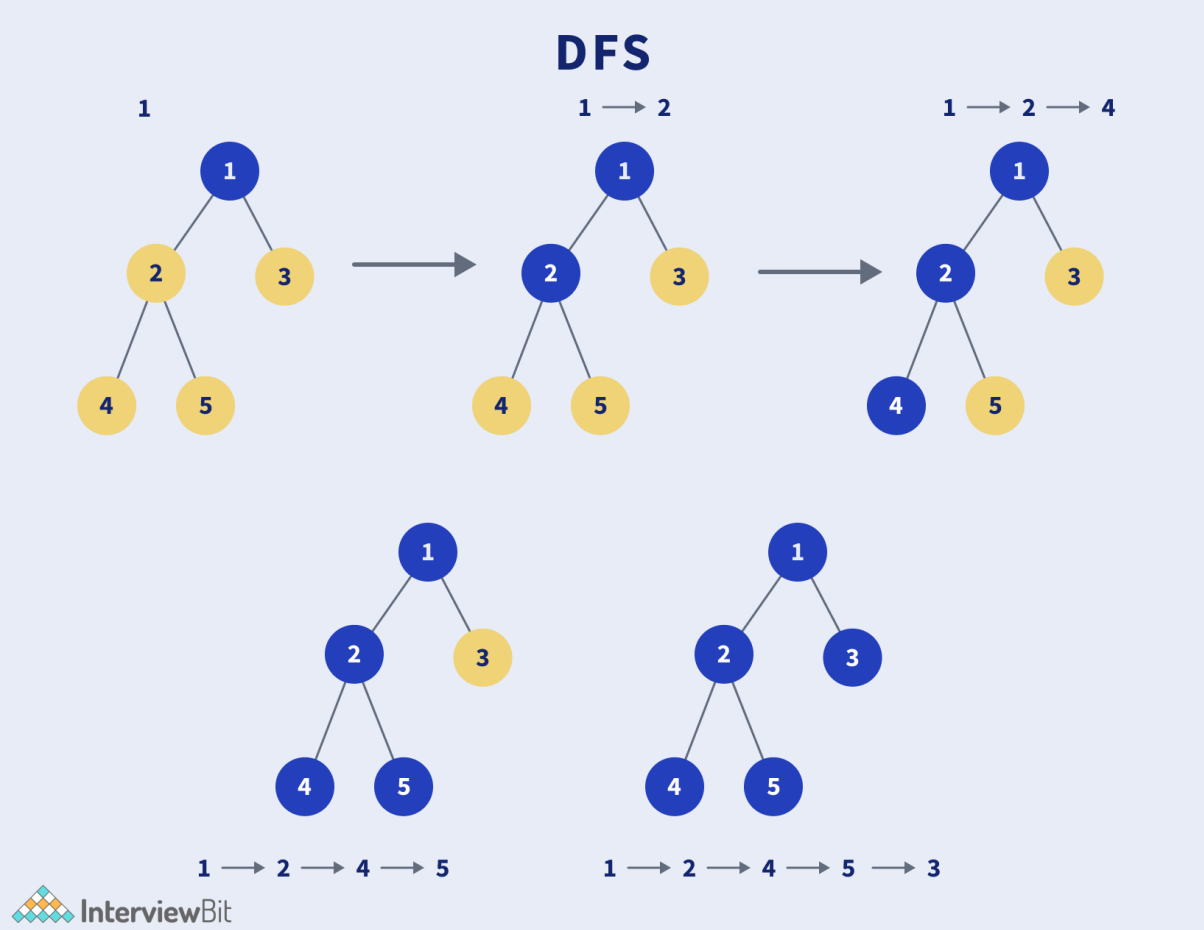
Slika 3: prikaz delovanja BFS (puščice označujejo kako obiskuje vozlišča) (vir: https://he-s3.s3.amazonaws.com/media/uploads/fdec3c2.jpg)

2.4.4 Iskanje najprej v globino (ang. Depth-First Search - DFS)

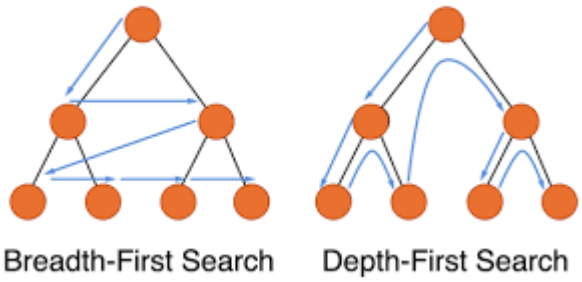
DFS je prav tako kot BFS algoritem iskanja poti v grafu, ki sistematično obiskuje vsa vozlišča v grafu, le da DFS sledi povezavam tako daleč, kot je le mogoče, preden se vrne nazaj in nadaljuje z naslednjim vozliščem.

Deluje tako, da najprej preveri vse povezave vozlišča v določeni smeri, preden preide na naslednje vozlišče. Lahko se izvaja s pomočjo sklada (ang. stack - vozlišča se shranjujejo na sklad) ali pa rekurzivno (funkcija kliče samo sebe in za argument poda vsako sosednje vozlišče).

Pri DFS algoritmu je pomembno, da graf nima ciklov, saj algoritem lahko preide v neskončno zanko. Postopek DFS algoritma je prikazan na spodnji sliki 4, slika 5 pa prikazuje primerjavo načina obiskovanja vozlišč med algoritmoma BFS in DFS.



Slika 4: prikaz DFS algoritma (vir: https://www.interviewbit.com/blog/wp-content/uploads/2021/12/DFS-Algorithm-768x595.png)



Slika 5: primerjava načina obiskovanja vozlišč med BFS in DFS (vir: https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcQ3WT_MTklL9LgqryZ_U41KYIriXRYLubO8Zz3R6ddrNfpF0w6_)

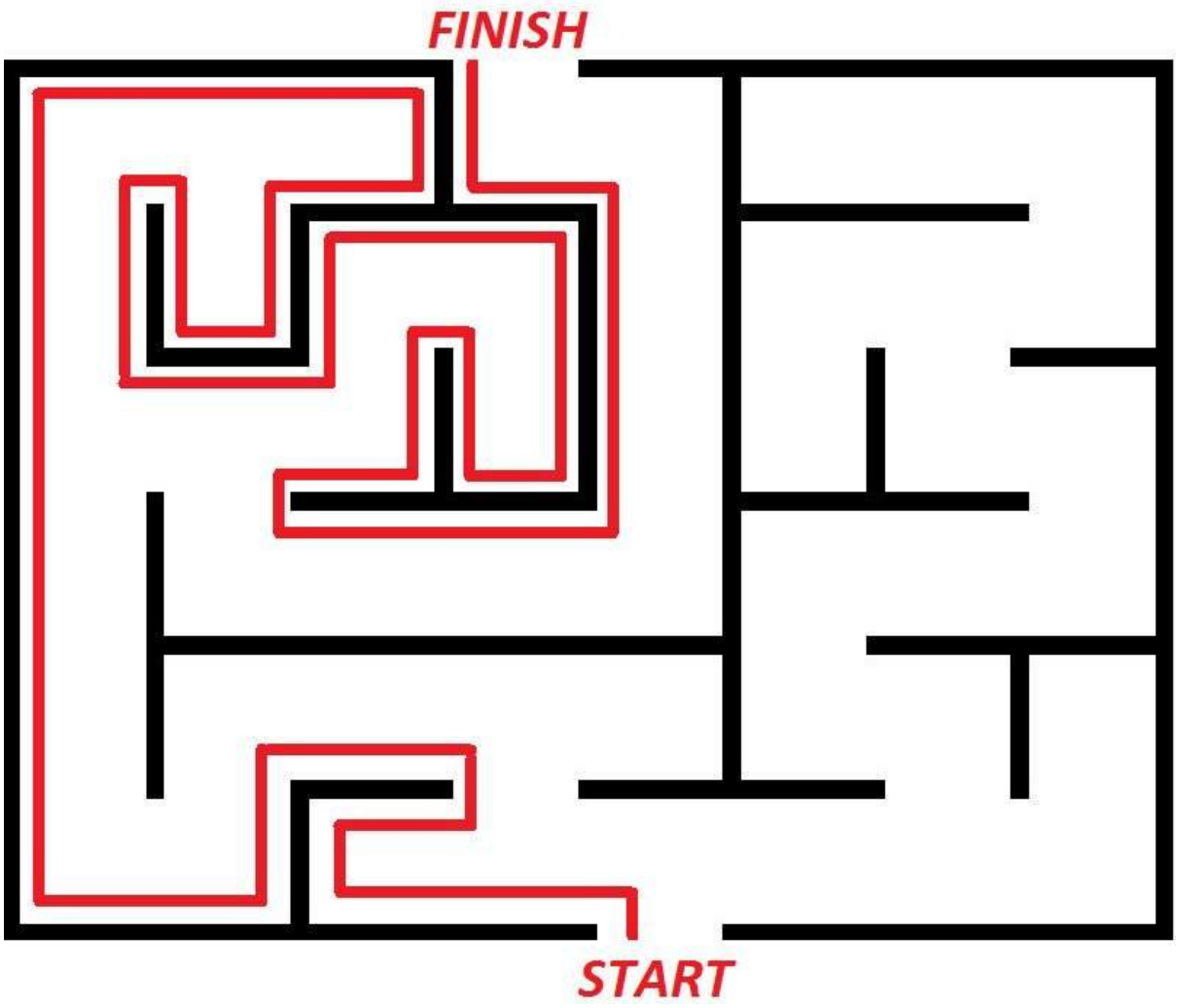
2.4.5 Algoritem sledilca steni (ang. The Wall Follower Algorithm)

Algoritem sledilca steni je preprosta strategija reševanja labirintov, ki temelji na ideji, da če neprestano držimo roko ob steni na naši levi strani, bomo na koncu prehodili celoten labirint in dosegli izhod.

Deluje tako, da na začetku izbere stran, kateri bo sledil (levo ali desno). Nato se začne premikati po labirintu. Ko naleti na križišče, vedno da prednost tisti strani, za katero se je na začetku

odločil, recimo, da je to leva. Tako vsakič, ko naleti na križišče, poskusi zaviti levo, če je leva pot blokirana, nadaljuje naravnost, drugače pa zavije na desno.

To ponavlja dokler ne pride do cilja labirinta. Postopek algoritma sledilca steni je prikazan na spodnji sliki.



Slika 6: prikaz rešenega algoritma s pomočjo ideje algoritma sledilca steni (vir: https://www.researchgate.net/publication/315969093/figure/fig7/AS:668320751685645@1536351492897/Left-Wall-Follower-solvable-maze-7.jpg)

3. Implementacija algoritmov

Zaradi enostavnosti prikazovanja izbire naslednje celice, sem algoritme implementiral v programskem jeziku JavaScript s pomočjo orodja ReactJS. V veliko pomoč pri implementaciji in učenju orodja ReactJS sta bili tudi orodji Chat GPT in Google Gemini. Algoritme sem testiral na unikatnih labirintih, ki sem jih prav tako generiral s pomočjo JavaScripta.

Vsaka celica v mreži je predstavljena kot komponenta *Node* z lastnostmi, ki določajo njen položaj v mreži (x in y koordinati), ali je začetna ali končna točka, ali je zid, ter unikaten identifikator mreže, s pomočjo katerega sem rešil težavo, kjer so se vsi algoritmi izvajali v isti mreži, namesto vsak v svoji (izsek kode 1).

```
import React, { Component } from 'react';
import './Node.css';

export default class Node extends Component {
    render() {
        const { col, row, isEnd, isStart, isWall, gridId } = this.props;
        const extraClassName = isEnd ? 'node-end' : isStart ? 'node-start' : isWall ?
'node-wall' : '';
        return (
            <div
                id={`grid${gridId}-node-${row}-${col}`}
                className={`node ${extraClassName}`}></div>
        );
    }
}
```

Izsek kode 1: programska koda JavaScript za celico (vir: Lovro Hauptman)

Implementiral sem tudi logiko za vizualizacijo delovanja algoritmov s pomočjo programskega jezika CSS (izseki kode 2, 3, 4). S tem sem animiral prikazovanje obiskanih celic, najkrajše poti ter drugih ključnih korakov vsakega algoritma.

```
.node{
    width: 10px;
    height: 10px;
    border: 1px solid #484D6D;
    display: inline-block;
}
.node-start {
    background-color: green;
}
.node-end {
    background-color: red;
}
.node-wall {
    background-color: #4d009a;
}
```

Izsek kode 2: programska koda CSS za celico (vir: Lovro Hauptman)

```
// Shortest path
```

```
    animateShortestPath(nodesInShortestPathOrder) {
        for (let i = 0; i < nodesInShortestPathOrder.length; i++) {
            setTimeout(() => {
                const node = nodesInShortestPathOrder[i];
                document.getElementById(`grid${node.gridId}-node-${node.row}-
${node.col}`).className = 'node node-shortest-path';
            }, 3 * i);
        }
    }

    // Dijkstra's algorithm
    animateDijkstra(visitedNodesInOrder, nodesInShortestPathOrder) {
        for (let i = 0; i <= visitedNodesInOrder.length; i++) {
            if (i === visitedNodesInOrder.length) {
                setTimeout(() => {
                    this.animateShortestPath(nodesInShortestPathOrder);
                }, 5 * i);
                return;
            }
            setTimeout(() => {
                const node = visitedNodesInOrder[i];
                document.getElementById(`grid${node.gridId}-node-${node.row}-
${node.col}`).className = 'node node-visited';
            }, 5 * i);
        }
    }

    visualizeDijkstra() {
        const { gridDijkstra, gridDijkstraStartNode, gridDijkstraEndNode } = this.state;
        const startTime = performance.now();
        const visitedNodesInOrder = dijkstra(gridDijkstra, gridDijkstraStartNode,
gridDijkstraEndNode);
        const endTime = performance.now();
        const nodesInShortestPathOrder = getNodesInShortestPathOrder(gridDijkstraEndNode);
        const totalNodes = gridDijkstra.length * gridDijkstra[0].length;
        const wallNodes = gridDijkstra.flat().filter(node => node.isWall).length;
        const nonWallNodes = totalNodes - wallNodes;
        this.setState({
            dijkstraTime: endTime - startTime,
            dijkstraVisitedNodes: visitedNodesInOrder.length,
            dijkstraVisitedPercentage: (visitedNodesInOrder.length / nonWallNodes) * 100,
            pathLengthDijkstra: nodesInShortestPathOrder.length
        });
        this.animateDijkstra(visitedNodesInOrder, nodesInShortestPathOrder);
    }
```

Izsek kode 3: koda JavaScript za vizualizacijo Dijkstrovega algoritma (vir: Lovro Hauptman)

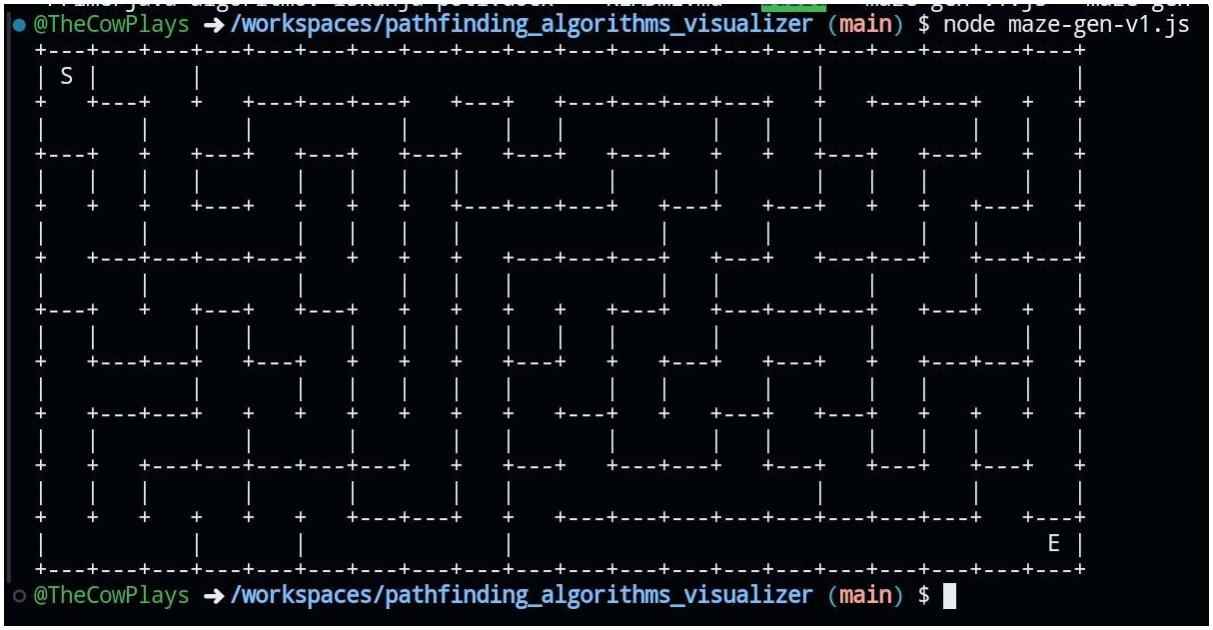
```
.node-visited {
    animation-name: visitedAnimation;
    animation-duration: 1s;
    animation-timing-function: cubic-bezier(1, 0, 0, 1);
    animation-delay: 0;
    animation-fill-mode: forwards;
    animation-play-state: running;
}

@keyframes visitedAnimation {
    0% {
```

```
        background-color: #29315a;
    }
    100% {
        background-color: #4b8f8c;
    }
}

.node-shortest-path {
    animation-name: shortestPath;
    animation-duration: 1s;
    animation-timing-function: cubic-bezier(1, 0, 0, 1);
    animation-delay: 0;
    animation-fill-mode: forwards;
    animation-play-state: running;
}
@keyframes shortestPath {
    0% {
        transform: scale(0.6);
        background-color: #C5979D;
    }
    50% {
        transform: scale(1.2);
        background-color: #C5979D;
    }
    100% {
        transform: scale(1);
        background-color: #C5979D;
    }
}
```

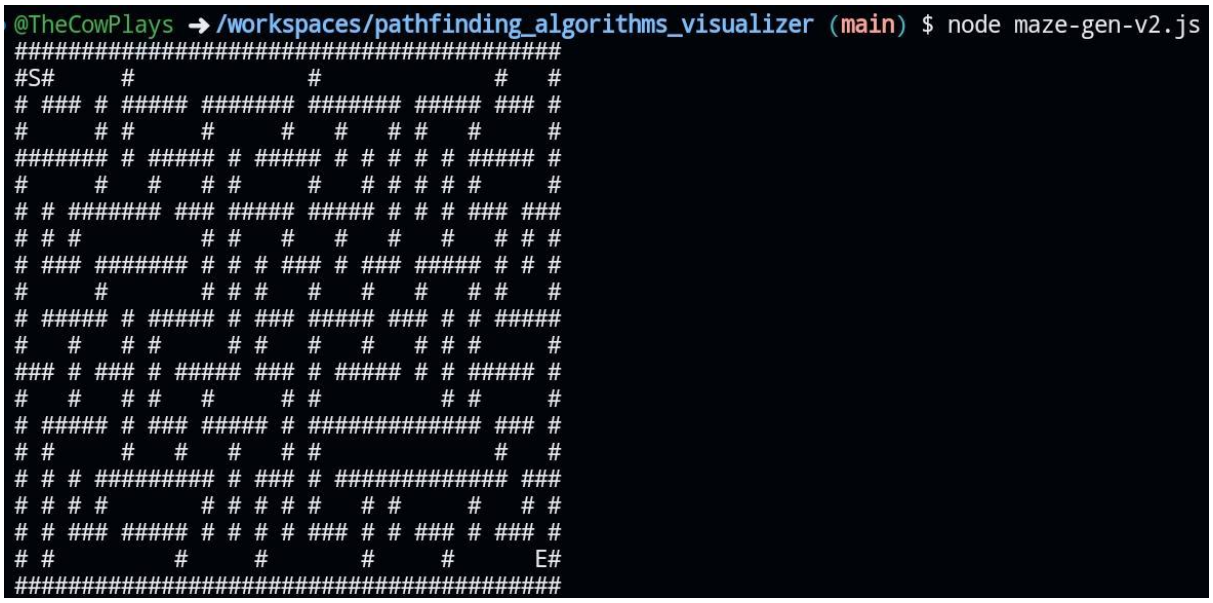
Izsek kode 4: koda CSS za animacijo celic (vir: Lovro Hauptman)

3.1 Problem generiranja labirintov primernih za testiranje

Labirinte sem generiral s pomočjo različice DFS algoritma (algoritme za iskanje poti lahko uporabimo tudi za generacijo labirintov). Pri implementaciji sem se soočil s težavo, da je algoritem na začetku generiral labirinte tako, da je vsaka celica lahko imela od 0 do 4 zidove okoli sebe (slika 7), v moji implementaciji mreže pa sem naredil, da je celica lahko zid ali pa ne (slika 8). Tako sem moral algoritem prilagoditi na svoj način prikazovanja zidov.



Slika 7: prvotna različica generatorja labirintov, kjer ima vsaka celica lahko do 4 zidove (vir: Lovro Hauptman)



Slika 8: popravljena različica generatorja labirintov, kjer je celica lahko ali zid ali pa ne (vir: Lovro Hauptman)

Generator labirintov deluje tako, da nastavi začetno (levo zgoraj) in končno (desno spodaj) celico. Nato od začetne celice dalje rekurzivno raziskuje še ne obiskane celice, pri čemer je vmes dodajal in odstranjeval zidove in s tem zagotovil, da so vsi deli labirinta dostopni, pri čemer si je pomagal z skladom.

Za dodajanje kompleksnosti in raznolikosti v labirintu, se je algoritem občasno vrnil na prej obiskane celice (ang. backtrack). S tem je v labirint dodal zanke in slepe ulice (izsek kode 5).

```
generateMazeNotGlobal() {
  this.currentCell.visited = true;
```

```
    let nextCell = this.getNeighbors(this.currentCell);

    if (nextCell) {
      nextCell.visited = true;

      this.stack.push(this.currentCell);

      // Remove the wall between the current cell and the next cell
      let wallX = (this.currentCell.x + nextCell.x) / 2;
      let wallY = (this.currentCell.y + nextCell.y) / 2;
      this.grid[wallY][wallX].isWall = false;

      this.currentCell = nextCell;
    } else if (this.stack.length > 0) {
      // 40% chance to backtrack and carve a new path
      if (Math.random() < 0.4) {
        let backtrackCell = this.stack[Math.floor(Math.random() * this.stack.length)];
        this.currentCell = backtrackCell;
      } else {
        this.currentCell = this.stack.pop();
      }
    }
  }
```

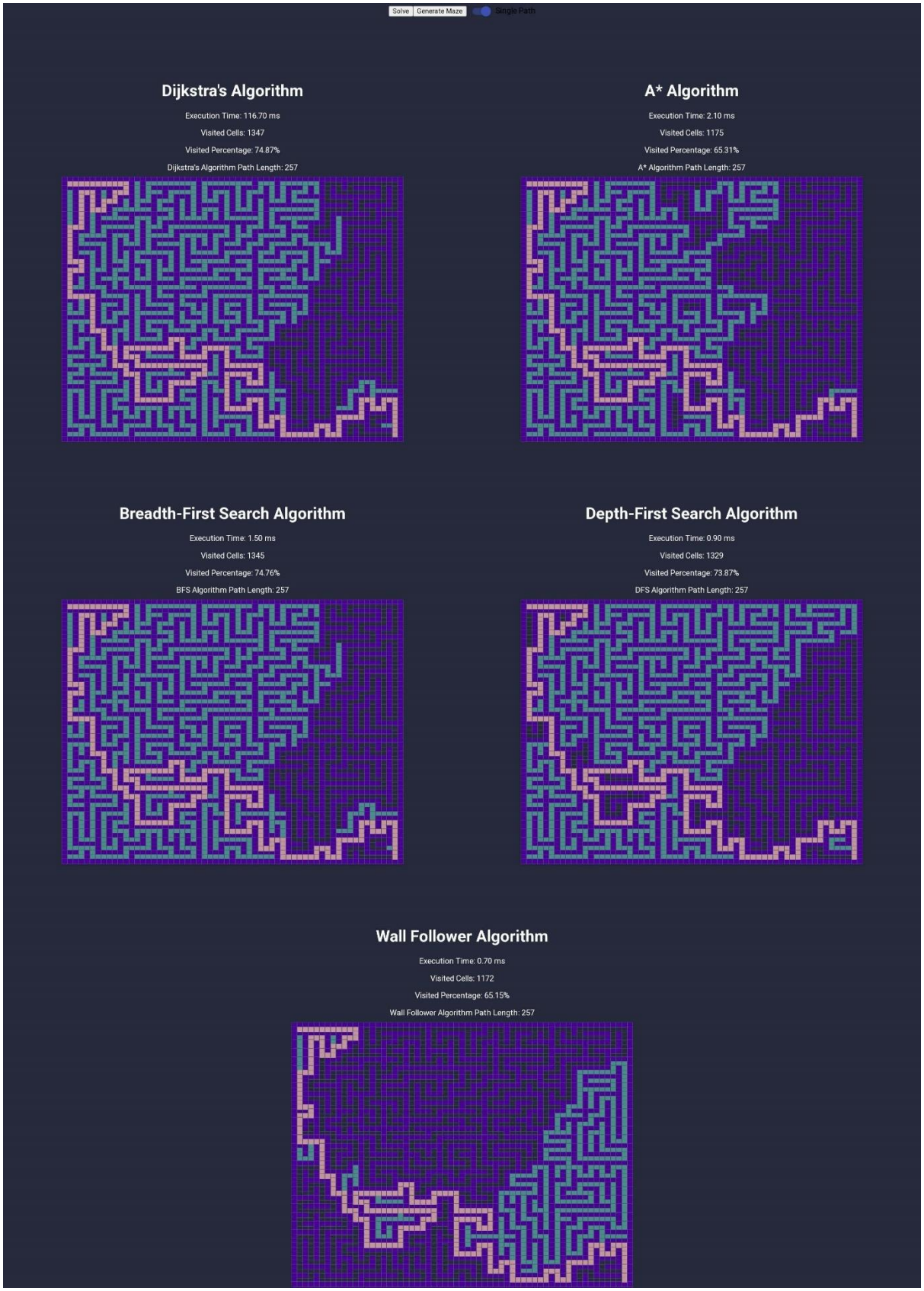
Izsek kode 5: koda za generiranje labirinta (vir: Lovro Hauptman)

Prav tako sem dodal opcijo generacije labirintov, ki nimajo samo ene možne rešitve. To sem storil tako, da sem iz labirinta izbrisal naključnih 10% zidov, ki pa niso hkrati najbolj zunanji zidovi (izsek kode 5).

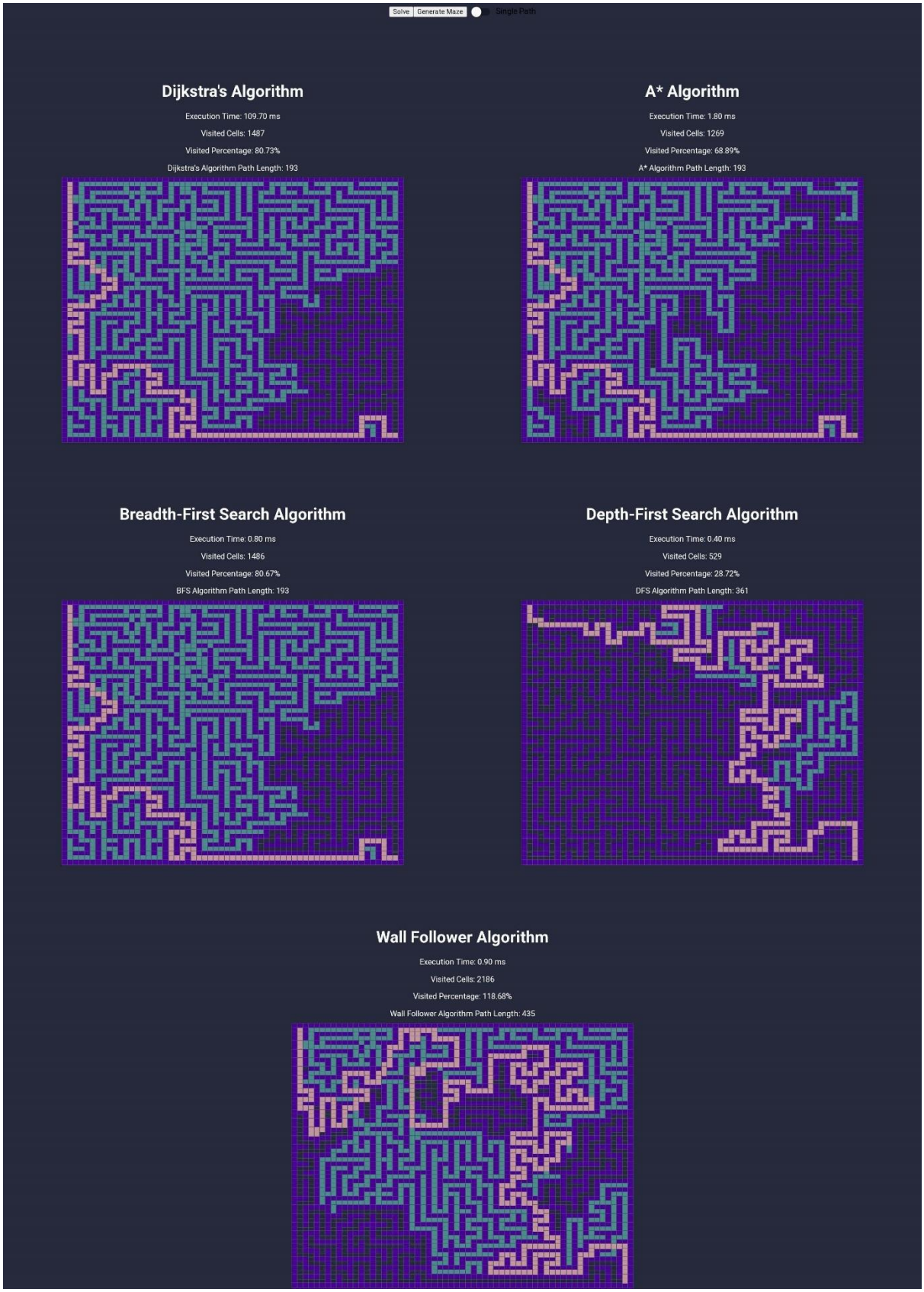
```
  if(!singlePath){
    for (let i = 0; i < numRows * numCols * 0.1; i++) { // remove 10% of the walls
        let x = Math.floor(Math.random() * (numRows - 2)) + 1;
        let y = Math.floor(Math.random() * (numCols - 2)) + 1;
        maze.grid[x][y].isWall = false;
    }
```

Izsek kode 6: koda za labirinte, ki imajo več kot 1 rešitev (vir: Lovro Hauptman)

Kočni prikaz rešenega labirinta z eno in z več rešitvami je viden na spodnjih slikah:



Slika 9: vizualni prikaz rešenega labirinta z eno rešitvijo



Slika 10: vizualni prikaz rešenega labirinta z več rešitvami

3.2 Implementacija posameznega algoritma

3.2.1 Dijkstrov algoritem (ang. Dijkstra's Algorithm)

Za implementacijo Dijkstrovega algoritma (izsek kode 6) sem napisal funkcijo *dijkstra()*, ki sprejme mrežo, začetno in končno celico. Algoritem sledi preprostemu postopku. Postavi začetno celico na razdaljo 0 in določi ne obiskane celice, ki jih shrani v seznam. Nato se izvaja zanka, dokler obstajajo ne obiskane celice. V vsaki iteraciji algoritem izbere najbližjo ne obiskano celico ter preveri njegove sosednje celice. Če je sosednja celica zid, se preskoči; če je dosežen cilj ali pa ni več ne obiskanih celic, se algoritem konča. V nasprotnem primeru se razdalje sosednjih ne obiskanih celic posodobijo. Na koncu algoritem vrne najkrajšo pot od začetne do končne celice.

```
export function dijkstra(grid, startNode, endNode){
    const visitedNodesInOrder = [];
    startNode.distance = 0;
    const unvisitedNodes = getAllNodes(grid);
    while(!!unvisitedNodes.length){
        sortNodesByDistance(unvisitedNodes);

        const closestNode = unvisitedNodes.shift();

        //if there is a wall, skip it
        if(closestNode.isWall) continue;
        // if distance is infinity, we are trapped and should stop
        if(closestNode.distance === Infinity) return visitedNodesInOrder;

        closestNode.isVisited = true;
        visitedNodesInOrder.push(closestNode);
        if(closestNode === endNode) return visitedNodesInOrder;
        updateUnvisitedNeighbors(closestNode, grid);
    }
}

function sortNodesByDistance(unvisitedNodes){
    unvisitedNodes.sort((nodeA, nodeB) => nodeA.distance - nodeB.distance);
}

function updateUnvisitedNeighbors(node, grid){
    const unvisitedNeighbors = getUnvisitedNeighbors(node, grid);
    for(const neighbor of unvisitedNeighbors){
        neighbor.distance = node.distance + 1;
        neighbor.previousNode = node;
    }
}

function getUnvisitedNeighbors(node, grid){
    const neighbors = [];
    const {col, row} = node;
    if(row > 0) neighbors.push(grid[row - 1][col]);
```

```
    if(row < grid.length - 1) neighbors.push(grid[row + 1][col]);
    if(col > 0) neighbors.push(grid[row][col - 1]);
    if(col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
    return neighbors.filter(neighbor => !neighbor.isVisited);
}

function getAllNodes(grid){
    const nodes = [];
    for(const row of grid){
        for(const node of row){
            nodes.push(node);
        }
    }
    return nodes;
}
```

Izsek kode 6: koda Dijkstrovega algoritma (vir: Lovro Hauptman)

3.2.2 Algoritem A*

Funkcija *astar()* (izsek kode 7) izvaja algoritem A* za iskanje najkrajše poti od začetne do končne celice v mreži. Pri tem uporablja odprt in zaprt seznam celic ter hevristično funkcijo ocenjevanja razdalje med njimi. Med izvajanjem algoritma se obiskane celice dodajajo v seznam *visitedNodesInOrder()*, ki sledi vrstnemu redu obiska. Algoritem iterativno izbira celice z najnižjo vrednostjo f. Hevristična funkcija *heuristic()* oceni razdaljo med določeno celico in končno celico s pomočjo evklidove formule za izračun razdalje med točkama.

```
export function astar(grid, startNode, endNode) {
    const openList = [startNode];
    const closedList = [];
    const visitedNodesInOrder = [];
    startNode.distance = 0;
    startNode.h = heuristic(startNode, endNode);
    startNode.f = startNode.h;
    while (openList.length > 0) {
        const currentNode = getLowestFScore(openList);
        openList.splice(openList.indexOf(currentNode), 1);
        closedList.push(currentNode);
        visitedNodesInOrder.push(currentNode);
        if (currentNode === endNode) {
            return visitedNodesInOrder;
        }
        const neighbors = getUnvisitedNeighbors(currentNode, grid);
        for (const neighbor of neighbors) {
            if (closedList.includes(neighbor) || neighbor.isWall) continue;
            const gScore = currentNode.distance + 1;
            const hScore = heuristic(neighbor, endNode);
            if (!openList.includes(neighbor)) {
                openList.push(neighbor);
                neighbor.distance = gScore;
                neighbor.h = hScore;
                neighbor.f = gScore + hScore;
                neighbor.previousNode = currentNode;
            } else if (gScore < neighbor.distance) {
                neighbor.distance = gScore;
```

```
                neighbor.f = gScore + hScore;
                neighbor.previousNode = currentNode;
            }
        }
    }
    return visitedNodesInOrder;
}

// f = g + h
function getLowestFScore(nodes){
    return nodes.reduce((lowest, node) => node.f < lowest.f ? node : lowest, nodes[0]);
}

function heuristic(node, endNode){
    return Math.abs(node.row - endNode.row) + Math.abs(node.col - endNode.col);
}

function getUnvisitedNeighbors(node, grid){
    const neighbors = [];
    const {col, row} = node;
    if(row > 0) neighbors.push(grid[row - 1][col]);
    if(row < grid.length - 1) neighbors.push(grid[row + 1][col]);
    if(col > 0) neighbors.push(grid[row][col - 1]);
    if(col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
    return neighbors.filter(neighbor => !neighbor.isVisited);
}
```

Izsek kode 7: koda algoritma A (vir: Lovro Hauptman)*

3.2.3 Iskanje najprej v širino (ang. Breadth-First Search - BFS)

Funkcija *bfs()* (izsek kode 8) izvaja iskanje najkrajše poti z algoritmom BFS od začetne do končne celice v mreži, pri čemer uporablja vrsto za sledenje obiskanih celic. Med izvajanjem algoritma se celice dodajajo v seznam *visitedNodesInOrder()*. Algoritem nadaljuje z obiskovanjem sosednjih celic, dokler ne najde cilja oz. dokler ne pregleda vseh celic v mreži. Vsako celico preveri, če je zid ali ne in če je že obiskana. Če je že obiskana, jo doda na seznam obiskanih celic. Nato preveri sosednje celice, ki še niso bile obiskane. Razdalja od začetne celice do sosednjih celic trenutne celice se posodablja med vsakim korakom, da se izračuna najkrajša pot.

```
export function bfs(grid, startNode, endNode) {
    const visitedNodesInOrder = [];
    const queue = [];
    startNode.distance = 0;
    queue.push(startNode);
    while (queue.length !== 0) {
        const currentNode = queue.shift();
        if (currentNode.isWall) continue;
        if (currentNode.distance === Infinity) return visitedNodesInOrder;
        currentNode.isVisited = true;
        visitedNodesInOrder.push(currentNode);
        if (currentNode === endNode) return visitedNodesInOrder;
        const unvisitedNeighbors = getUnvisitedNeighbors(currentNode, grid);
        for (const neighbor of unvisitedNeighbors) {
```

```
            neighbor.distance = currentNode.distance + 1;
            neighbor.previousNode = currentNode;
            neighbor.isVisited = true;  // Mark as visited here
            queue.push(neighbor);
        }
    }
}

function getUnvisitedNeighbors(node, grid) {
    const neighbors = [];
    const {col, row} = node;
    if (row > 0) neighbors.push(grid[row - 1][col]);
    if (row < grid.length - 1) neighbors.push(grid[row + 1][col]);
    if (col > 0) neighbors.push(grid[row][col - 1]);
    if (col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
    return neighbors.filter(neighbor => !neighbor.isVisited);
}
```

Izsek kode 8: koda za algoritem BFS (vir: Lovro Hauptman)

3.2.4 Iskanje najprej v globino (ang. Depth-First Search - DFS)

Izsek kode 9 prikazuje implementacijo algoritma DFS. DFS začne svoje iskanje iz začetne celice, pri čemer nastavi njeno razdaljo na 0 in jo potisne na sklad. Nadaljuje dokler sklad ni prazen. Pri vsakem koraku algoritem vzame iz sklada celico in preveri, če je zid. Če je, jo preskoči in nadaljuje na naslednjo. Če je razdalja celice enaka neskončno, pomeni da še ni bila obiskana. Če je trenutna celica enaka končni celici, oz. če je sklad prazen, algoritem zaključi z izvajanjem in vrne seznam obiskanih celic.

```
export function dfs(grid, startNode, endNode) {
    const visitedNodesInOrder = [];
    startNode.distance = 0;
    const stack = [startNode];
    while (stack.length) {
        const currentNode = stack.pop();
        if (currentNode.isWall) continue;
        if (currentNode.distance === Infinity) return visitedNodesInOrder;
        currentNode.isVisited = true;
        visitedNodesInOrder.push(currentNode);
        if (currentNode === endNode) return visitedNodesInOrder;
        const unvisitedNeighbors = getUnvisitedNeighbors(currentNode, grid);
        for (const neighbor of unvisitedNeighbors) {
            neighbor.distance = currentNode.distance + 1;
            neighbor.previousNode = currentNode;
            stack.push(neighbor);
        }
    }
}

function getUnvisitedNeighbors(node, grid) {
    const neighbors = [];
    const { col, row } = node;
    if (row > 0) neighbors.push(grid[row - 1][col]);
    if (row < grid.length - 1) neighbors.push(grid[row + 1][col]);
    if (col > 0) neighbors.push(grid[row][col - 1]);
```

```
    if (col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
    return neighbors.filter(neighbor => !neighbor.isVisited);
}
```

Izsek kode 9: koda za algoritem DFS (vir: Lovro Hauptman)

3.2.5 Algoritem sledilca steni (ang. The Wall Follower Algorithm)

Izsek kode 10 prikazuje implementacijo algoritma sledilca steni. Algoritem začne svoje iskanje iz začetne celice, pri čemer nastavi njeno razdaljo na 0. Nato se premika skozi mrežo, dokler ne doseže cilja. Med premikanjem vsako obiskano celico označi kot obiskano in jo doda na seznam obiskanih celic. Algoritem preveri ne obiskane sosednje celice trenutne celice in izbere naslednjo celico, ki jo bo obiskal. Pri tem se skuša izogniti že obiskanim celicam ter celicam, ki so zidovi ter se premika v smeri, ki sledi steni. Če naslednje celice ni mogoče izbrati (slepa ulica), se vrne nazaj in nadaljuje od tam naprej. To pomeni, da lahko eno celico obišče večkrat. Ta postopek ponavlja, dokler ne doseže cilja. Na koncu vrne seznam obiskanih celic.

```
export function wallFollower(grid, startNode, endNode) {
    const visitedNodesInOrder = [];
    startNode.distance = 0;
    let currentNode = startNode;
    let previousNode = null;
    while (currentNode !== endNode) {
        currentNode.isVisited = true;
        visitedNodesInOrder.push(currentNode);
        const neighbors = getUnvisitedNeighbors(currentNode, grid);
        let nextNode = null;
        for (const neighbor of neighbors) {
            if (neighbor !== previousNode && !neighbor.isWall) {
                nextNode = neighbor;
                break;
            }
        }
        if (nextNode) {
            nextNode.distance = currentNode.distance + 1;
            nextNode.previousNode = currentNode;
            previousNode = currentNode;
            currentNode = nextNode;
        } else {
            if (previousNode) {
                currentNode = previousNode;
                previousNode = currentNode.previousNode;
            } else {
                break;
            }
        }
    }
    return visitedNodesInOrder;
}

function getUnvisitedNeighbors(node, grid) {
    const neighbors = [];
    const { col, row } = node;
    if (row > 0) neighbors.push(grid[row - 1][col]);
```

```
    if (row < grid.length - 1) neighbors.push(grid[row + 1][col]);
    if (col > 0) neighbors.push(grid[row][col - 1]);
    if (col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
    return neighbors.filter(neighbor => !neighbor.isVisited);
}
```

Izsek kode 10: koda za algoritem sledilca steni (vir: Lovro Hauptman)

4. Podrobna analiza izbranih algoritmov

V nadaljevanju bom podrobno analiziral izbrane algoritme, kar nam bo pomagalo pri boljšem razumevanju njihove uporabe.

4.1 Merila ocenjevanja

Za merjenje učinkovitosti vsakega algoritma pri reševanju labirinta bom uporabil naslednja merila:

- Čas izvajanja [ms] - meri čas, ki ga vsak algoritem porabi za izvedbo. Izračuna se z beleženjem začetnega in končnega časa izvajanja s pomočjo metode *performance.now()*.
- Odstotek obiskanih celic [%] - odstotek celic, ki jih je algoritem obiskal.
- Ali je pot najkrajša? - ali je algoritem našel najkrajšo možno pot, ali samo eno izmed poti.

Na žalost nisem uspel izmeriti količine pomnilnika, ki ga vsak algoritem uporablja v spletni aplikaciji, ki algoritem vizualizira, saj JavaScript in ReactJS ne omogočata neposrednega dostopa do podatkov o porabi pomnilnika v brskalniku. Vendar sem napisal program *testRunner.js* (izseki kode 11, 12, 13, 14, 15), ki poganja te algoritme brez vizualizacije, obenem pa jih izvaja v ločenih nitih (ang. thread). S pomočjo orodja Node.js sem v program dodal še sledenje porabe pomnilnika. Pomembno je omeniti, da so to le ocene, saj JavaScript uporablja mehanizem čiščenja spomina (ang. garbage collection), ki vpliva na natančnost meritev.

testRunner.js deluje s pomočjo JavaScript delovnih niti (Worker Threads), ki omogočajo vzporedno izvajanje algoritmov. To pomeni, da lahko hkrati izvajamo več algoritmov, ne da bi pri tem blokirali glavno nit izvajanja. S tem pristopom lahko bolje izkoriščamo več jedrne procesorje in izboljšamo učinkovitost izvajanja testov.

```
if (isMainThread) {
    // This code will be executed in the main thread
    function runAlgorithmInWorker(workerData) {
        return new Promise((resolve, reject) => {
            const worker = new Worker(__filename, { workerData });

            worker.on('message', resolve);
            worker.on('error', reject);
            worker.on('exit', (code) => {
                if (code !== 0) reject(new Error(`Worker stopped with exit code ${code}`));
            });
```

```
        });
    }
```
Izsek kode 11: koda za vzpostavitev delovnih niti (vir: Lovro Hauptman)

```
algorithms.forEach(algorithm => {
    const promise = runAlgorithmInWorker({
        algorithm,
        grid: grids[algorithm],
        startNode: startNodes[algorithm],
        endNode: endNodes[algorithm],
    }).then(({ algorithm, time, visitedNodesInOrder, nodesInShortestPathOrder, memoryUsed,
visitedPercentage }) => {
        const visitedNodes = visitedNodesInOrder.length;
        const pathLength = nodesInShortestPathOrder.length;
        metricsSPOn[algorithm].time.push(time);
        metricsSPOn[algorithm].visitedNodes.push(visitedNodes);
        metricsSPOn[algorithm].visitedPercentage.push(visitedPercentage);
        metricsSPOn[algorithm].pathLength.push(pathLength);
        metricsSPOn[algorithm].memoryUsed.push(memoryUsed);
    });
    promises.push(promise);
});

await Promise.all(promises);
console.log(`Completed test ${i + 1} of ${numMazes} for mazes with a single path`);
```
Izsek kode 12: koda za zagon algoritmov na labirintu z eno rešitvijo (vir: Lovro Hauptman)

```
function calculateAverages(metrics) {
    const averages = {
        dijkstra: { time: 0, visitedNodes: 0, visitedPercentage: 0, pathLength: 0,
memoryUsed: 0 },
        astar: { time: 0, visitedNodes: 0, visitedPercentage: 0, pathLength: 0, memoryUsed:
0 },
        bfs: { time: 0, visitedNodes: 0, visitedPercentage: 0, pathLength: 0, memoryUsed: 0
},
        dfs: { time: 0, visitedNodes: 0, visitedPercentage: 0, pathLength: 0, memoryUsed: 0
},
        wallFollower: { time: 0, visitedNodes: 0, visitedPercentage: 0, pathLength: 0,
memoryUsed: 0 }
    };


    for (const algorithm in metrics) {
        const numTests = metrics[algorithm].time.length;
        for (const metric in metrics[algorithm]) {
            const sum = metrics[algorithm][metric].reduce((a, b) => a + b, 0);
            let average = sum / numTests;
            if (metric === 'time') {
                average = average.toFixed(4); // round to 4 decimal places
            } else if (metric === 'visitedPercentage') {
                average = (average).toFixed(2); // convert to percentage and round to 2
decimal places
            } else if (metric === 'memoryUsed') {
                average = (average / 1024 / 1024).toFixed(2); // convert to MB and round to
2 decimal places
            }
            averages[algorithm][metric] = average;
        }
```

```
    }

    return averages;
}
```
Izsek kode 13: koda za izračun povprečja meril ocenjevanja (vir: Lovro Hauptman)

```
function writeResultsToCsv(filename, averagesSPOn, averagesSPOff) {
    const header = ['Algorithm', 'SinglePath', 'Time', 'VisitedNodes', 'VisitedPercentage',
'PathLength', 'MemoryUsed'];
    const rows = [];

    for (const algorithm in averagesSPOn) {
        const row = [
            algorithm,
            'true',
            averagesSPOn[algorithm].time,
            averagesSPOn[algorithm].visitedNodes,
            averagesSPOn[algorithm].visitedPercentage,
            averagesSPOn[algorithm].pathLength,
            averagesSPOn[algorithm].memoryUsed
        ];
        rows.push(row.join(','));
    }

    for (const algorithm in averagesSPOff) {
        const row = [
            algorithm,
            'false',
            averagesSPOff[algorithm].time,
            averagesSPOff[algorithm].visitedNodes,
            averagesSPOff[algorithm].visitedPercentage,
            averagesSPOff[algorithm].pathLength,
            averagesSPOff[algorithm].memoryUsed
        ];
        rows.push(row.join(','));
    }

    const csv = [header.join(','), ...rows].join('\n');

    // Write to the CSV file
    fs.writeFile(filename, csv, (err) => {
        if (err) {
            console.error('Error writing to CSV file', err);
        } else {
            console.log('Successfully wrote to CSV file');
        }
    });
}
```
Izsek kode 14: koda za zapisovanje podatkov v .csv datoteko (vir: Lovro Hauptman)

```
} else {
    // This code will be executed in the worker thread
    const { algorithm, grid, startNode, endNode } = workerData;
    const startTime = performance.now();
    const initialMemoryUsage = process.memoryUsage().heapUsed;
    const visitedNodesInOrder = algorithms[algorithm](grid, startNode, endNode);
    const finalMemoryUsage = process.memoryUsage().heapUsed;
    const endTime = performance.now();
```

```
    const nodesInShortestPathOrder = getNodesInShortestPathOrder(endNode);
    const time = endTime - startTime;
    const memoryUsed = finalMemoryUsage - initialMemoryUsage;

    // Calculate the total number of visitable nodes
    const totalNodes = grid.length * grid[0].length;
    const wallNodes = grid.flat().filter(node => node.isWall).length;
    const nonWallNodes = totalNodes - wallNodes;

    // Calculate the visited percentage based on the number of visitable nodes
    const visitedPercentage = (visitedNodesInOrder.length / nonWallNodes) * 100;

    parentPort.postMessage({ algorithm, time, visitedNodesInOrder,
nodesInShortestPathOrder, memoryUsed, visitedPercentage });
}
```

Izsek kode 15: koda za vsako nit (vir: Lovro Hauptman)

4.2 Primerjava izbranih algoritmov

4.2.1 Primerjava O-notacije

Na primeru reševanja labirintov imajo izbrani algoritmi sledeče O-notacije:

- Dijkstrov algoritem: $O\big((V+E)\,log\,V\big)$; poenostavimo ga lahko na $O(2V\,log\,V)$
- algoritem A*: $O\big((V+E)\,log\,V\big)$; poenostavimo ga lahko na $O(2V\,log\,V)$
- BFS: $O(V+E)$; poenostavimo ga lahko na $O(2V)$
- DFS: $O(V+E)$; poenostavimo ga lahko na $O(2V)$
- Algoritem sledilca steni: podobno kot BFS in DFS, a se ta ocena lahko zelo razlikuje glede na kompleksnost labirinta

Tu je V število celic v labirintu in E število povezav med celicami. Pomembno je poudariti, da gre pri teh O-notacijah za ocene najslabših primerov reševanja. Kljub temu, da imata Dijkstrov algoritem in algoritem A* enako O-notacijo, lahko uporaba hevristične funkcije pri algoritmu A* izboljša časovno zahtevnost.

4.2.2 Primerjava rezultatov programa testRunner.js

Program *testRunner.js* je izvajal algoritme na labirintih velikost 100x100. Vsak algoritem je rešil 20 labirintov z eno rešitvijo in 20 labirintov z več rešitvami. Program je bil izveden znotraj orodja Github Codespaces s konfiguracijo s štirimi procesnimi jedri in 16 GB pomnilnika, kar je prispevalo k zmanjšanju možnosti napak zaradi drugih programov ali operacijskega sistema med testiranjem.

Rezultate je zapisal v .csv tabelo (slika 11):

Algorithm	Single Path	Time	Visited Nodes	Visited Percentage	Path Length	Memory Used
dijkstra	true	21418.86	18134.85	90.68	1007.8	20.49
astar	true	600.67	16353.35	81.77	1007.8	4.57
bfs	true	34.75	18133.65	90.67	1007.8	1.77
dfs	true	19.25	9776	48.88	1007.8	3.16
wallFollower	true	26.3	20889.6	104.45	1007.8	3.67
dijkstra	false	21751.43	19509.85	95.28	896.9	20.14
astar	false	717.68	18768.4	91.66	896.9	5.32
bfs	false	32.54	19508.05	95.27	896.9	1.99
dfs	false	19.34	9445.35	46.13	1500.6	2.53
wallFollower	false	23.24	19672.3	96.07	2215.8	4.35

Slika 11: .csv tabela rezultatov - velikost labirinta 100x100 (vir: Lovro Hauptman)

Iz tabele lahko razberemo, da je Dijkstrov algoritem najpočasnejši, ne glede na število poti, prav tako uporabi največ pomnilnika.

Drugi najdlje se izvaja algoritem A*, prav tako ne glede na število poti. Algoritem A* tudi porabi drugo največ pomnilnika, a bistveno manj kot Dijkstrov algoritem. Najmanj pomnilnika porabi BFS, ki pa je poleg Dijkstrovega algoritma in algoritma A* edini, ki ne glede na število poti vedno vrne najkrajšo pot do cilja. BFS porabi za izvedbo več časa kot DFS in sledilec steni, vendar manj kot Dijkstrov algoritem in A*.

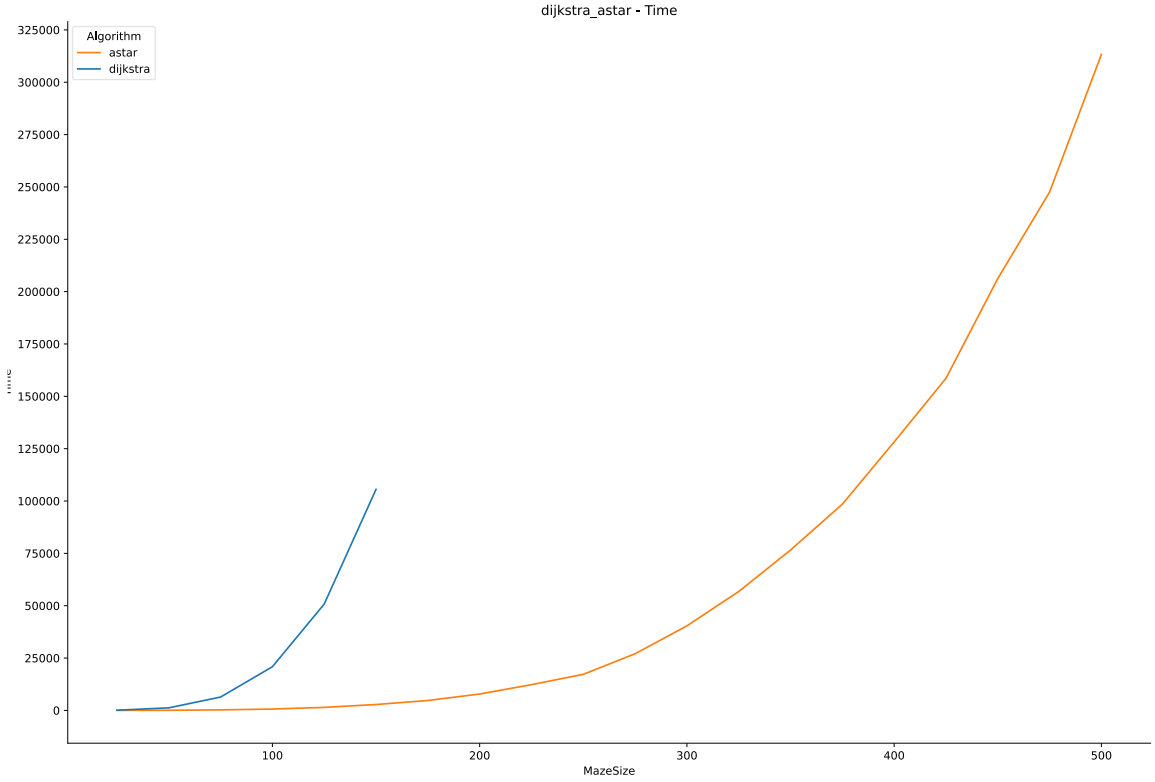
Najhitrejši algoritem je DFS, ki v obeh scenarijih porabi najmanj časa za izvedbo. To je morda zato, ker je labirint generiran s pomočjo variacije DFS algoritma. DFS je prav tako drugi najbolj učinkovit pri porabi pomnilnika. Drugi najhitrejši algoritem je algoritem sledilca steni, porabi pa manj pomnilnika kot Dijkstrov algoritem in A*, vendar več kot BFS in DFS.

Pomembno je tudi, da Dijkstrov algoritem, A* in BFS najdejo najkrajšo pot v labirintu, medtem ko DFS in sledilec steni najdeta samo eno izmed poti, ki ni nujno najdaljša. Sledilec steni prav tako v povprečju preišče več kot 100% celic oz. nekatere celice preišče večkrat, medtem ko DFS le okoli 40% celic (morda zaradi okoliščin generacije labirinta).

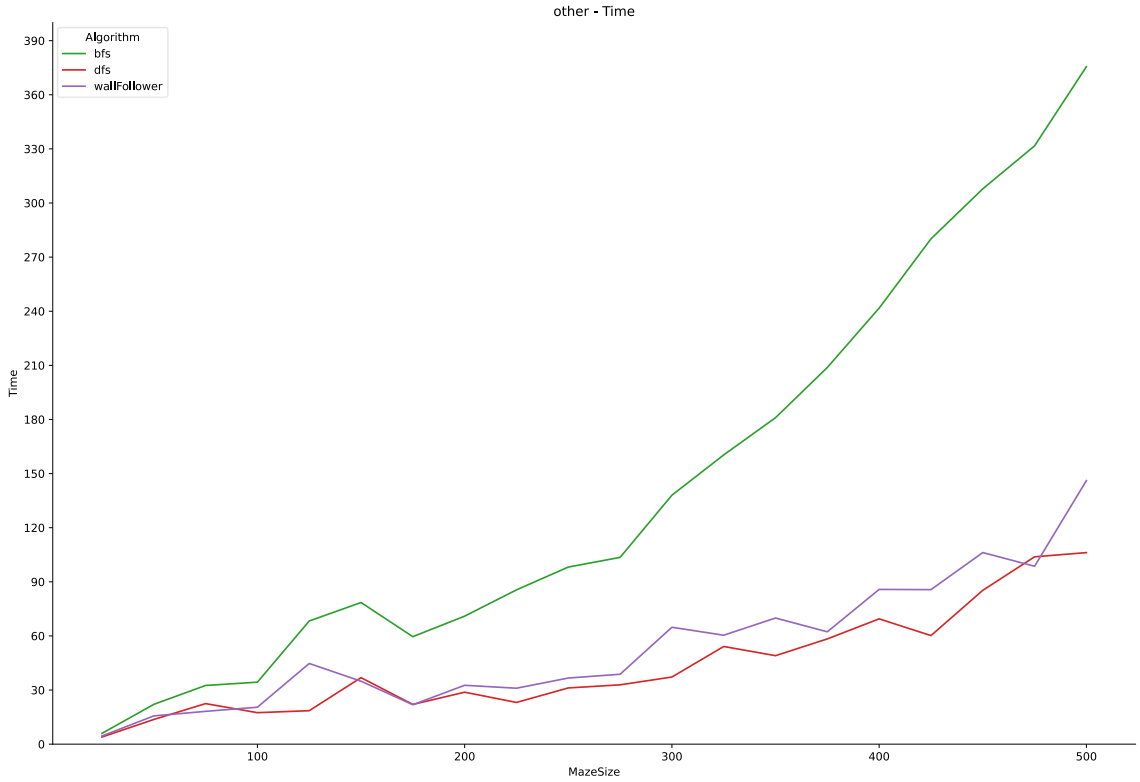
Odločil sem se izvesti *testRunner.js* še na večjih labirintih, vendar pa, ker se Dijkstrov algoritem izvaja opazno dlje kot ostali, sem program, kjer je so bili labirinti večji od 150x150, izvajal brez njega (povprečno 21 sekund na 100x100 labirint v primerjavi z ostalimi, ki so pod sedem sekund). Program sem izvedel na labirintih velikosti od 25x25 do 500x500 (velikosti sem povečeval za 25), a sem tokrat pri vsaki velikosti testiral le tri različne labirinte z eno rešitvijo in tri z več rešitvami, saj bi se drugače program izvajal več dni. Nato sem napisal program (izsek kode 16) v programskem jeziku Python, ki je iz pridobljenih podatkov narisal grafe (slike 12, 13, 14, 15):

```
import os
import glob
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.ticker import MaxNLocator

files = glob.glob('./data/averages*.csv')
all_data = []

for file in files:
    data = pd.read_csv(file)
    maze_size = int(os.path.basename(file).split('x')[0].replace('averages', ''))
    data['MazeSize'] = maze_size
    all_data.append(data)

all_data = pd.concat(all_data)
data_grouped = all_data.groupby(['Algorithm', 'MazeSize'])[['Time',
'MemoryUsed']].mean().reset_index()
data_dijkstra_astar = data_grouped[data_grouped['Algorithm'].isin(['dijkstra', 'astar'])]

data_other = data_grouped[~data_grouped['Algorithm'].isin(['dijkstra', 'astar'])]
colors = plt.get_cmap('tab10')
color_map = {algorithm: colors(i) for i, algorithm in
enumerate(data['Algorithm'].unique())}

for data, group in [(data_dijkstra_astar, 'dijkstra_astar'), (data_other, 'other')]:
    for metric in ['Time', 'MemoryUsed']:
        fig, ax = plt.subplots(figsize=(15, 10))  # Increase the figure size
        sns.lineplot(data=data, x='MazeSize', y=metric, hue='Algorithm', ax=ax,
palette=color_map)
        ax.set_title(f'{group} - {metric}')
        if metric == 'Time':
            if group == 'other':
                ax.set_ylim(bottom=0, top=400)
            ax.yaxis.set_major_locator(MaxNLocator(nbins=15))
        else:
            ax.set_ylim(bottom=0, top=data_grouped[metric].max() * 1.1)

        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)
        plt.subplots_adjust(left=0.05, right=0.95, top=0.95, bottom=0.05)
        plt.savefig(f'graphs/{group}_{metric}.svg', format='svg')
```

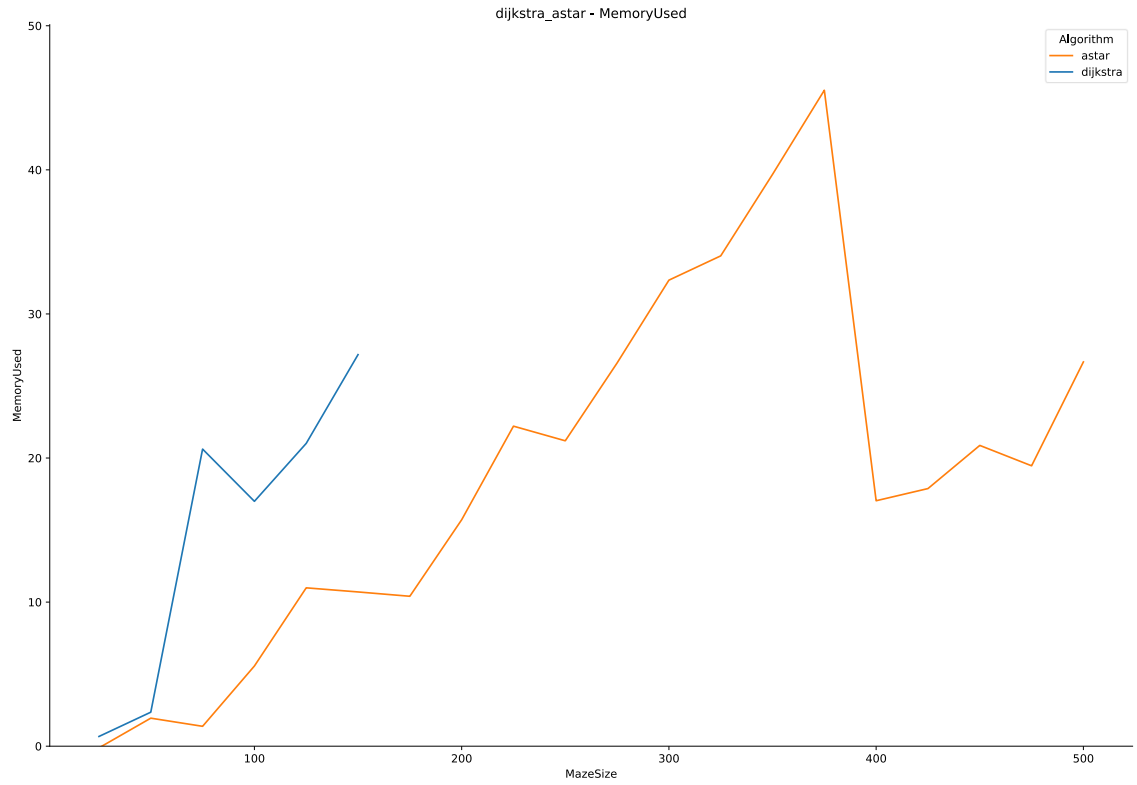
Izsek kode 16: Python koda, ki s pomočjo knjižnic pandas, matplotlib in seaborn izriše grafe (vir: Lovro Hauptman)



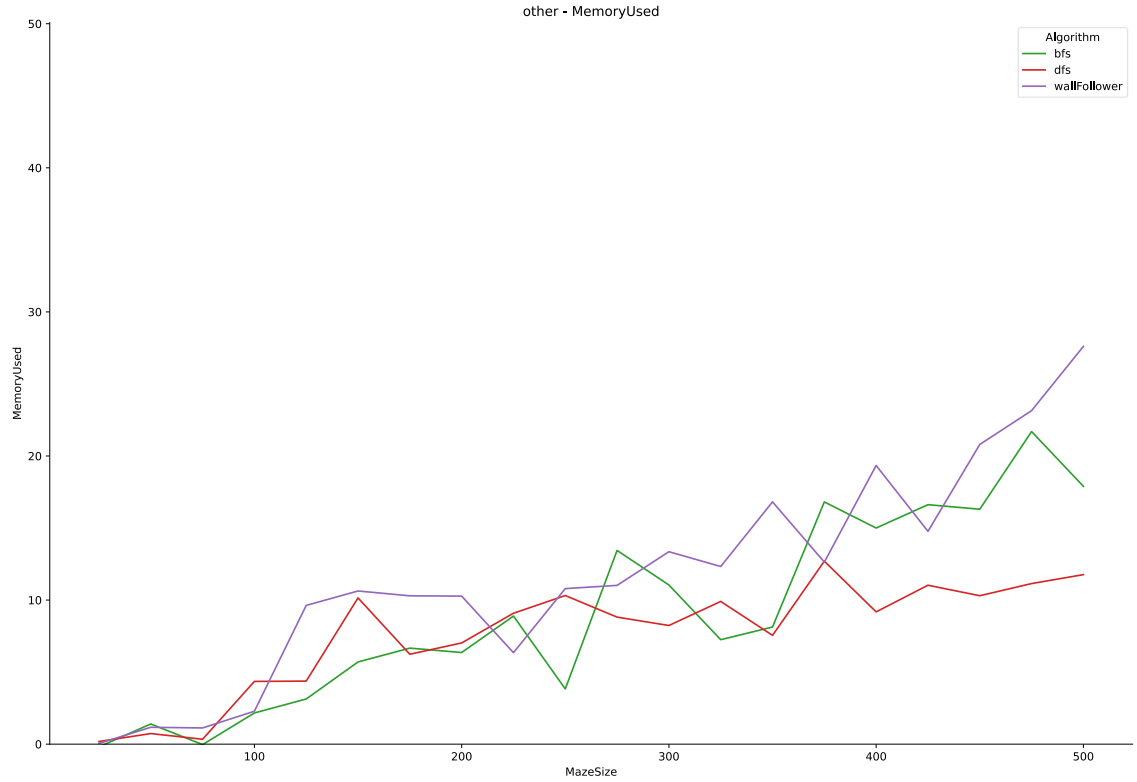
Slika 12: graf časa v milisekundah v odvisnosti od velikosti labirintov za DIjkstrov algoritem in algoritem A (vir: Lovro Hauptman)*



Slika 13: graf časa v milisekundah v odvisnosti od velikosti labirintov za algoritme BFS, DFS in sledilca steni (vir: Lovro Hauptman)



Slika 14: graf porabe pomnilnika v MB v odvisnosti od velikosti labirintov za Dijkstrov algoritem in algoritem A (vir: Lovro Hauptman)*



Slika 15: graf porabe pomnilnika v odvisnosti od velikosti labirintov za algoritme BFS, DFS in sledilca steni (vir: Lovro Hauptman)

Kot je razvidno iz grafov, rezultati sledijo približno enakemu trendu kot pri labirintu 100x100. Pri porabi pomnilnika je opazno nihanje. To se zgodi, ker JavaScript ne ponuja nobenega načina, da bi izmerili, koliko pomnilnika posamezen del programa porablja. Način, ki sem ga uporabil za merjenje pomnilnika ni ravno natančen, a ponuja vpogled v približno porabo pomnilnika med izvajanjem programa.

5. Zaključek

Po temeljiti analizi rezultatov izbranih algoritmov za iskanje poti lahko opazimo, da ima vsak izmed algoritmov svoje prednosti in slabosti, ki so odvisne od njihovih specifičnih pogojev uporabe. Dijkstrov algoritem se izkaže za najpočasnejšega in najbolj pomnilniško potratnega, vendar je učinkovit pri iskanju najkrajše poti pri uteženih grafih, kar pa moj primer labirinta ni. Algoritem A* se po mojih pričakovanjih izkaže za hitrejšega in učinkovitejšega kot Dijkstrov algoritem, kljub temu, da je A* prav tako narejen za iskanje najkrajše poti na uteženih grafih. To ga naredi primernega za številne aplikacije. BFS tako kot Dijkstrov algoritem in algoritem A* zanesljivo vedno najde najkrajšo pot skozi labirint ter pri tem porabi najmanj pomnilnika od vseh algoritmov, ki sem jih izvajal. BFS je tudi narejen za neutežene grafe, zato bi lahko rekli, da ima pri reševanju labirintov prednost domačega terena. Podobno kot BFS, se je tudi DFS izkazal pri hitrosti rešitve problema, vendar pa za razliko od prejšnjih treh algoritmov, DFS ne vrne vedno najkrajše možne poti skozi labirint, vendar le eno izmed možnosti. DFS ima še dodatno prednost – labirinti, ki sem jih testiral, so bili narejeni s pomočjo variacije DFS-ja. To je DFS-ju omogočalo biti tako hiter, kot je bil. Algoritem sledilca steni, ki je najbolj laičen izmed vseh algoritmov, ki sem jih primerjal, se je obnesel presenetljivo dobro. Ravno zaradi njegove preprostosti sem se ga odločil implementirati, saj me je zanimalo, če lahko s preprosto strategijo reši labirint tako hitro ali celo hitreje kot algoritmi, ki so specializirani za iskanje poti. Ta me je po hitrosti zelo presenetil, a je bila kvaliteta njegovih rešitev mnogo slabša v primerjavi z ostalimi. Vse implementacije algoritmov in programi, ki sem jih pri izdelavi seminarske naloge napisal so javno dostopni na povezavah:

- https://github.com/TheCowPlays/pathfinding_algorithms_test_runner in
- https://github.com/TheCowPlays/pathfinding_algorithms_visualizer.

Če povzamem, ni nujno, da je najbolj sofisticiran algoritem najboljši, kot je dokazal BFS, ki je po mojem mnenju zmagovalec mojega testiranja. Sam sem namreč pričakoval, da bo zmagal A*, saj sem o njem že veliko slišal. Skozi to seminarsko nalogo sem se veliko naučil. Spoznal sem orodje ReactJS in se soočil s številnimi izzivi, ki so se pojavili med razvojem aplikacije, nekateri tudi zaradi omenjenega orodja. Delo na tej nalogi mi je omogočilo pridobiti boljše razumevanje delovanja algoritmov za iskanje poti. Prepričan sem, da bo to znanje koristno za moje bodoče projekte na področju računalništva.

6. Viri in literatura

- Sodelavci Stack Overflow. 2019. Free Algorithms Book. Dostopno na URL: https://books.goalkicker.com/AlgorithmsBook/ (Uporabljeno 17. 12. 2023).
- Amit Patel. 2024. Amit's A* Pages. Dostopno na URL: https://theory.stanford.edu/~amitp/GameProgramming/ (Uporabljeno 18. 12. 2023).
- Jamis Buck. 2011. Buckblog: Maze Generation: Growing Tree algorithm. Dostopno na URL: https://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm (Uporabljeno 18. 12. 2023).
- Meta Platforms, Inc. 2024. Getting Started – React. Dostopno na URL: https://legacy.reactjs.org/docs/getting-started.html (Uporabljeno 11. 12. 2023).
- Sodelavci Wikipedie. 2024. Wikipedia. Dostopno na URL: https://www.wikipedia.org/ (Uporabljeno 17. 2. 2024).
- Daniel Monzonís Laparra, Pathfinding Algorithms in Graphs and Applications, Barcelona, 2019.

Izjava o avtorstvu

Izjavljam, da je seminarska naloga v celoti moje avtorsko delo, ki sem ga izdelal samostojno s pomočjo navedene literature in pod vodstvom mentorja.

April 2024 Lovro Hauptman, G 4. B