

**Universitatea Tehnică a Moldovei**

**Sistem de navigare prin labirint utilizând algoritmi de căutare**

**mAZE NAVIGATOR System using search algorithms**

|  |  |
| --- | --- |
| **Studenții:** | **gr. TI-231,**  **Bujac Ion**  **Lungu Sorina Bortă Victor** |
| **Coordonator:** | **Coșeru Cătălin,**  **asist.univ.** |

**Chişinău, 2024**

**CUPRINS**

[ABREVIERI ȘI DEFINIȚII 3](#_Toc183786628)

[INTRODUCERE 4](#_Toc183786629)

[1 REPREZENTAREA PROBLEMEI 5](#_Toc183786630)

[1.1 Definirea datelor problemei şi stabilirea relaţiei între date şi soluţia problemei 6](#_Toc183786631)

[2 ALGORITMII UTILIZAȚI 7](#_Toc183786632)

[2.1 Algoritmul lui Prim aleatoriu 7](#_Toc183786633)

[2.2 Algoritmul BFS 8](#_Toc183786634)

[2.3 Algoritmul Dijkstra 9](#_Toc183786635)

[2.4 Algoritmul A\* 10](#_Toc183786636)

[2.5 Argumentarea alegerii algoritmilor 11](#_Toc183786637)

[3 IMPLEMENTAREA JOCULUI 12](#_Toc183786638)

[3.1 Implementarea algoritmilor în limbajul de programare C# 12](#_Toc183786639)

[3.1.1 Implementarea algoritmului lui Prim 12](#_Toc183786640)

[3.1.2 Implementarea algoritmului BFS 13](#_Toc183786641)

[3.1.3 Implementarea algoritmului Dijkstra 15](#_Toc183786642)

[3.1.4 Implementarea algoritmului A\* 17](#_Toc183786643)

[3.2 Analiza algoritmilor 18](#_Toc183786644)

[3.3 Implementarea interfeței grafice. 21](#_Toc183786645)

[3.3 Interacțiunea cu utilizatorul 23](#_Toc183786646)

[CONCLUZII 27](#_Toc183786647)

[BIBLIOGRAFIE 28](#_Toc183786648)

[ANEXA A 29](#_Toc183786649)

# **ABREVIERI ȘI DEFINIȚII**

**BFS** – Breadth First Search este un algoritm utilizat pentru căutarea unui nod ce îndeplinește o anumită condiție într-un arbore sau graf.

**Checkpoin**t - un punct intermediar într-un labirint ce trebuie găsit înainte de a ajunge la destinație.

**Framework** - o platformă software formată dintr-un set de biblioteci și instrumente standardizate, utilizată pentru dezvoltarea și integrarea componentelor unui proiect software.

**Software** - un sistem de programe pentru calculatoare, care include proceduri de aplicare, fie preinstalat, fie creat sau achiziționat ulterior.

**Backtracking** - un algoritm general pentru găsirea tuturor soluțiilor unei probleme prin construirea incrementală a soluțiilor-candidat și abandonarea celor care nu pot deveni valide.

**FIFO** – abreviere pentru First In, First Out, o metodă de gestionare a structurilor de date, unde primul element introdus este primul procesat.

**Tehnică greedy** – algoritmi simpli folosiți la probleme de optimizare, cum ar fi să gasească cea mai bună ordine de executare a unor lucrari pe calculator, să gasească cel mai scurt drum într-un graf.

**Shakey** - primul robot mobil capabil să perceapă și să analizeze mediul înconjurător.

**.Net Framework -** un framework software dezvoltat de Microsoft, ce oferă un mediu de rulare și un set de biblioteci pentru dezvoltarea aplicațiilor pe sistemele de operare Windows.

**Windows Forms -** o bibliotecă de clase grafică (GUI) gratuită și open-source, inclusă ca parte a Microsoft .NET, .NET Framework sau Mono, care oferă o platformă pentru dezvoltarea aplicațiilor client pentru desktopuri, laptopuri și tablete.

**Desktop** - imagine care apare la deschiderea calculatorului și pe care se pot vedea simbolurile programelor ce pot fi activate.

**SortedSet -** structură de date utilizată pentru a stoca elemente unice într-o ordine sortată, definită printr-un comparator personalizat; în acest context, funcționează similar unei cozi de priorități.

**RGB -** un model aditiv de culoare, în care culorile albastru, roșu și verde sunt amestecate în diferite moduri pentru a produce o gamă largă de culori. O reprezentare a mixării aditive a culorilor.

**Drag and drop -** o practică răspândită, în care utilizatorii selectează, mai întâi, fie o porțiune de text, fie unul sau mai multe elemente și îl mută pe ecran într-un loc diferit sau pe un obiect virtual , plasându-l acolo.

**Design –** procesul de planificare și crearea aspectului vizual și funcțional al aplicației sau sistemului, inclusiv structurarea interfeței utilizatorului și definirea fluxului de interacțiune.

**GUI** - Graphical User Interface, o metodă de interacțiune cu un computer sau o aplicație, folosind imagini, icoane și butoane, în loc de comenzi text.

# **INTRODUCERE**

În termeni generali un algoritm reprezintă o metodă de rezolvare pas cu pas a problemelor. O problemă se consideră a fi constituită din date de intrare, și un enunț care specifică relația existentă între datele de intrare și soluția problemei. În cadrul algoritmului sunt descrise prelucrările necesare pentru a obține soluția problemei pornind de la datele de intrare.

Algoritmii au devenit tot mai buni, fiind capabili să proceseze cantități uriașe de informații. De exemplu, în medicină, algoritmii pot analiza rapid date complicate și pot ajuta doctorii să pună diagnostice mai precise. În finanțe, îi folosim pentru a prezice cum va evolua piața, iar în industrie, ajută la automatizarea proceselor și la reducerea greșelilor.

Un lucru interesant la algoritmi este că pot să învețe și să se adapteze cu ajutorul învățării automate. Cu cât primesc mai multe date, cu atât devin mai buni la găsirea soluțiilor, inclusiv la probleme neașteptate. În viitor, algoritmii vor face parte din și mai multe aspecte ale vieții umane, ajutând la crearea unor soluții mai eficiente pentru energie, transport și protecția mediului, schimbând modul în care sunt abordate problemele globale.

Proiectul urmărește să creeze un joc care să ajute utilizatorii să parcurgă labirinturi folosind algoritmi speciali de căutare. Acești algoritmi sunt foarte importanți pentru a rezolva probleme complicate, cum ar fi găsirea celor mai scurte drumuri.

Jocul va analiza cât de bine funcționează diferiți algoritmi, având o interfață grafică simplă, unde utilizatorii pot vedea modul de lucru al acestora. Unul dintre scopurile proiectului este să ajute oamenii să înțeleagă mai bine cum funcționează algoritmii și să ofere o experiență practică și educativă.

Indiferent de complexitatea unei aplicații informatice, la bazele ei stau algoritmi destinați rezolvării problemelor fundamentale ale aplicației. Oricât de sofisticată ar fi tehnologia software utilizată (interfețe, structurare globală a aplicației) eficiența aplicației este în mod esențial determinată de eficiența algoritmilor implicați.

Algoritmul este considerat corect dacă prin aplicarea lui asupra datelor problemei conduce la soluția acestuia și eficient dacă prin execuția acestuia rezultatul se obține într-un interval de timp rezonabil.

Proiectul are ca obiectiv principal cercetarea algoritmilor de căutare utilizați în navigarea prin labirint și analiza soluțiilor obținute. În cadrul acestuia, se urmărește proiectarea unei interfețe grafice cu posibilitatea de explorare a labirinturilor. Un alt obiectiv important este implementarea și compararea mai multor algoritmi de căutare pe diverse tipuri de labirinturi, oferind utilizatorilor posibilitatea de a vizualiza procesul de navigare. Performanța algoritmilor va fi testată și evaluată în funcție de timpul de execuție și numărul de pași pentru a găsi soluția.

# **1 REPREZENTAREA PROBLEMEI**

Navigarea prin labirinturi reprezintă o problemă care se regăsește în multe domenii, de la planificarea traseelor pentru vehicule moderne, la optimizarea rețelelor complexe. Complexitatea labirinturilor necesită algoritmi inteligenți care să identifice cea mai scurtă și scurtă cale de la un punct de start la destinație.

Proiectul dat se concentrează pe rezolvarea acestei probleme prin utilizarea algoritmilor BFS, Dijkstra și A\*. Aplicația dezvoltată permite observarea modului în care acești algoritmi explorează și rezolvă labirintul, oferind o posibilitate pentru analizarea și compararea performanței lor în diferite situații.

Problema este reprezentată prin implementarea unor checkpoint-uri (puncte de control) puse la dispoziția algoritmilor pentru a le colecta.

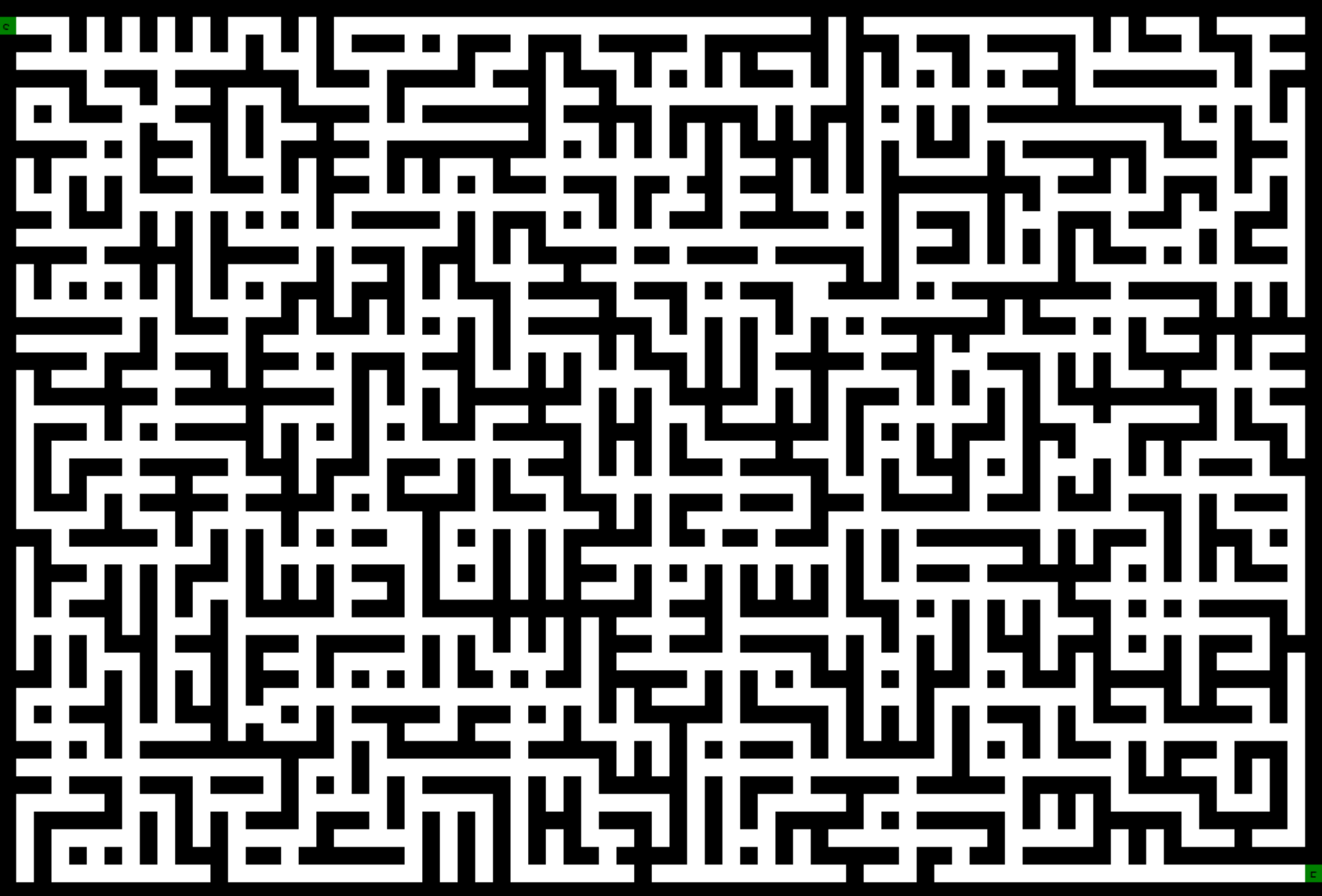
## **1.1 Definirea datelor problemei şi stabilirea relaţiei între date şi soluţia problemei**

Prelucrările efectuate în cadrul unui algoritm se efectuează asupra unor date. Acestea sunt entități purtătoare de informație (relevantă pentru problema de rezolvat).

Găsirea celui mai scurt drum printr-un labirint este o sarcină într-un spațiu structurat, unde fiecare celulă a labirintului poate fi considerată un nod al unui graf, iar legăturile dintre celule reprezintă muchiile care permit trecerea de la o celulă la alta. În această problemă, labirintul este reprezentat de o matrice bidimensională, unde fiecare element poate fi o cale liberă sau un obstacol, limitând astfel mișcările posibile.

Datele de intrare în această problemă includ dimensiunile labirintului, structura sa – definită prin obstacole și căi libere, pozițiile de start și destinație și punctele de control. Scopul este de a găsi un traseu minim de la punctul de start la destinație, evitând obstacolele și micșorând distanța sau numărul de pași necesari pentru a ajunge la destinație. Legătura dintre date și soluție constă în faptul că algoritmii de căutare explorează nodurile în dependență de modul de funcționare a fiecărui algoritm, analizând traseele posibile și identificând calea cea mai scurtă.

Soluția propusă va fi implementată în limbajul C#, utilizând framework-uri necesare pentru dezvoltarea interfeței grafice. Utilizatorul va putea interacționa cu labirintul prin intermediul unei interfețe grafice, reprezentată în Figura 1.1, care va permite vizualizarea labirintului, cât și urmărirea în timp real a parcurgerii acestuia de către algoritmii de căutare. De asemenea, utilizatorul va avea posibilitatea de a genera labirinturi noi.



**Figura 1.1 - Interfața grafică a labirintului**

Astfel, problema se reduce la găsirea celei mai eficiente căi de ieșire din labirint, prin intermediul algoritmului ales și ajungerea la punctul de ieșire.

# **2 ALGORITMII UTILIZAȚI**

Pentru generarea labirintului s-a utilizat o versiune a algoritmului lui Prim, care extinde labirintul treptat. Generarea începe dintr-un punct de start, iar la fiecare pas se adaugă un perete care creează o legătură nouă între celule, păstrând structura unui labirint unic, fără cicluri.

Pentru găsirea celui mai scurt drum în acest labirint, au fost aplicați trei algoritmi: BFS, A\* și Dijkstra, fiecare având caracteristici diferite.

BFS explorează toate drumurile posibile într-un mod diferit, verificând fiecare nivel înainte de a avansa. Este simplu de implementat și garantează găsirea celui mai scurt drum.

A\* utilizează o funcție de euristică pentru a ghida căutarea spre destinație. Acestă funcție reduce timpul de explorare și este eficient în labirinturi mari și complexe.

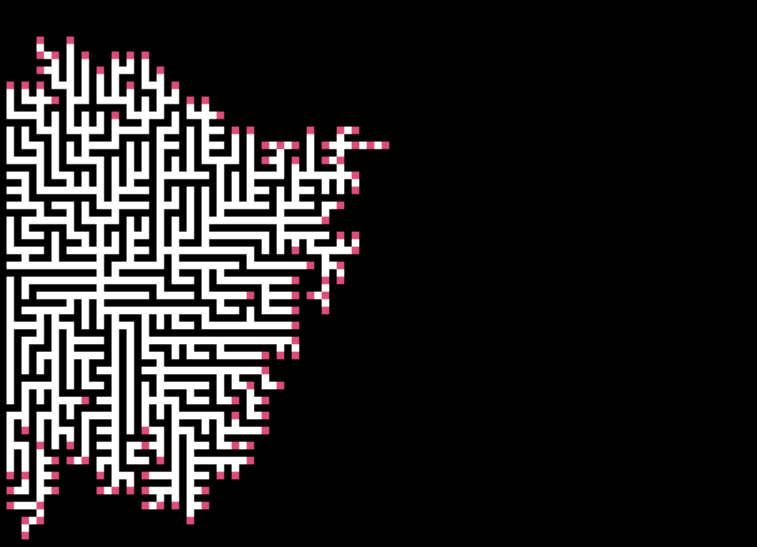
Dijkstra analizează toate drumurile posibile, dar ia în calcul costurile exacte pentru fiecare pas, fără a încerca să prezică direcția. Este ideal atunci când drumurile au costuri diferite.

Fiecare algoritm are punctele sale forte: BFS este simplu și direct, A\* este rapid, iar Dijkstra este precis pentru situațiile în care drumurile au costuri diferite.

## **2.1 Algoritmul lui Prim aleatoriu**

Algoritmul lui Prim este un algoritm lacom care este folosit pentru a găsi arborele de acoperire minim dintr-un grafic. Algoritmul lui Prim găsește submulțimea de muchii care include fiecare vârf al graficului astfel încât suma greutăților muchiilor să poată fi minimizată. Logica acestui algoritm poate fi utilizată de asemenea și pentru a genera aleatoriu labirinturi. Ideea generală este de a construi un graf cu vârfuri pentru fiecare celulă din labirint, fiecare celulă fiind conectată la patru vecini: sus, jos, stânga și dreapta.

În Figura 2.1 se poate observa cum algoritmul funcționează. Nodurile active, care sunt afișate cu roz, sunt de obicei spre capetele labirintului. Acest lucru se datorează faptului că, deși există noduri interioare, dacă în orice moment acele noduri sunt selectate din setul de noduri deschise și se descoperă că nu mai poate fi formată o ramură din acest nod, ele sunt ulterior eliminate din acel set și marcate cu alb. Deci, un nod interior marcat cu roz va deveni de obicei alb relativ repede.[1]



**Figura 2.1 – Modul de funționare al algoritmului lui Prim**

Modul de lucru este diferit față de backtracking, pentru că nu urmează mereu o cale fixă până la capăt înainte de a reveni. În schimb, când ajunge într-un punct unde există mai multe opțiuni (un nod cu ramificații), algoritmul nu alege doar continuarea pe ramura curentă. El selectează aleatoriu un alt nod, dintr-un set deschis care include toate nodurile explorate până acum, dar care mai au drumuri neexplorate.

Algoritmul explorează labirintul într-un mod mai variat, fără să fie strict liniar. Practic, sare între puncte diferite ale labirintului în loc să avanseze secvențial, ceea ce îl face să exploreze căile într-o ordine haotică.

## **2.2 Algoritmul BFS**

Parcurgerea în lățime (Breadth First Search) este un algoritm folosit pentru a vizita și explora toate nodurile dintr-un graf, inclusiv matrice bidimensionale 2D, într-o mișcare de tip nivel cu nivel. Memoria suplimentară, de obicei o coadă, este necesară pentru a ține evidența nodurilor secundare care au fost întâlnite, dar nu au fost încă explorate.

În cazul unei matrice 2D, BFS ia în considerare toate căile pornind de la sursă și se deplasează înainte cu o unitate în toate acele căi în același timp, ceea ce se asigură că prima dată când destinația este vizitată, aceasta este calea cea mai scurtă.

Pentru a aplica cu succes algoritmul BFS la o matrice 2D, este esențial să se înțeleagă bine ce reprezintă o matrice bidimensională. În informatică, o rețea, o hartă sau un labirint pot fi reprezentate ca o matrice 2D, o structură de date asemănătoare unei matrice cu rânduri și coloane. Cunoașterea modului în care datele sunt indexate și accesate pe rânduri și coloane este crucială pentru o implementare eficientă.[2]

Algoritmul BFS urmează o serie de pași pentru a explora toate nodurile accesibile dintr-un graf sau o matrice. Procesul începe prin inițializarea unei cozi ,,FIFO” și a unei structuri, cum ar fi un vector pentru a marca nodurile sau celulele deja vizitate.

Primul pas este adăugarea în coadă a celulei de start, care este și marcată ca vizitată. Urmează un ciclu principal, care continuă până când coada devine goală. La fiecare iterație, se extrage o celulă din coadă, aceasta devenind nodul curent. Vecinii nodului curent – fie că sunt în sus, jos, stânga sau dreapta – sunt verificați. Pentru fiecare vecin care nu a fost vizitat încă, se realizează două acțiuni: vecinul este adăugat în coadă și este marcat ca vizitat pentru a evita procesarea sa ulterioară.

Acest proces se repetă până când nodul destinație a fost vizitat, garantând astfel o explorare completă a spațiului, începând de la celula de start. Acest mod de parcurgere este ideal pentru probleme precum găsirea drumului minim într-o matrice sau explorarea labirinturilor și garantează găsirea celui mai scurt drum.

Complexitatea timpului poate fi exprimată ca O(V\*E), deoarece fiecare vârf și fiecare muchie vor fi explorate în cel mai rău caz. V este numărul de vârfuri și E este numărul de muchii din graf. De menționat că O(E) poate varia între O(1) și O(V2) în funcție de cât de rar este graful de intrare. Când numărul de vârfuri din graf este cunoscut din timp complexitatea spațiului poate fi exprimată ca O(V\*E).[3]

Algoritmul BFS este un instrument puternic pentru navigarea și analizarea matricelor 2D, contribuind la succesul numeroaselor aplicații din lumea reală, datorită adaptabilității sale și caracteristicilor limbajului C#.

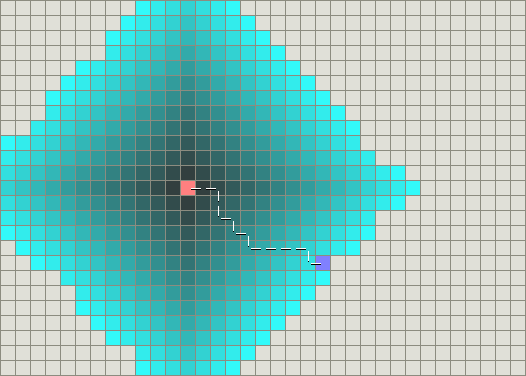
## **2.3 Algoritmul Dijkstra**

Algoritmul Dijkstra funcționează vizitând nodurile dintr-un graf, pornind de la punctul de start. Apoi examinează în mod repetat nodul cel mai apropiat care nu a fost încă vizitat și adaugă nodurile sale vecine în setul celor ce urmează să fie analizate. Se extinde treptat din punctul de pornire până ajunge la destinație. Algoritmul Dijkstra garantează găsirea celui mai scurt drum între punctul de start și destinație, atâta timp cât niciuna dintre muchii nu are cost negativ. Algoritmul folosește o tehnică greedy, în sensul că la fiecare pas găsim soluția imediat următoare optimă, sperând că soluția finală va fi cea mai bună pentru întreaga problemă.

Aplicații ale algoritmului Dijkstra:

* drumul cel mai scurt;
* aplicații de rețele sociale;
* locațiile pe hartă.

În Figura 2.2, pătratul roz reprezintă punctul de pornire, pătratul albastru este destinația, iar zonele turcoaz indică regiunile pe care algoritmul lui Dijkstra le-a explorat. Zonele de turcoaz mai deschis sunt cele mai îndepărtate de punctul de start și, prin urmare, formează "frontiera" explorării.[4]



**Figura 2.2 – Modul de funcționare a algoritmului Dijkstra**

Algoritmul Dijkstra folosește o structură de date pentru stocarea soluțiilor parțiale sortate în funcție de distanța de la început. Algoritmul original folosește o coadă de prioritate minimă și rulează în timp O((V+E)\*log V) unde V este numărul de noduri și E este numărul de muchii). Fiecare adăugare sau eliminare a unui element din coada de prioritate se realizează în O(log V) timp, de aici și complexitatea obținută. Complexitatea spațială a algoritmului este O(V).

## **2.4 Algoritmul A\***

Algoritmul A\* a fost creat ca parte a proiectului Shakey, care avea ca scop construirea unui robot mobil care să-și poată planifica propriile acțiuni. Nils Nilsson a propus inițial utilizarea algoritmului de traversare a grafului pentru planificarea traseului lui Shakey. Acest algoritm este ghidat de o funcție euristică h(n), distanța estimată de la nodul n la nodul țintă: ignoră complet g(n), distanța de la nodul de început la n. A\* a fost conceput inițial pentru a găsi căi cu cel mai mic cost atunci când costul unei căi este suma costurilor sale, dar s-a demonstrat că A\* poate fi folosit pentru a găsi căi optime pentru orice problemă care satisface condițiile unei algebre de cost.[5]

Algoritmul A\* este o variantă a algoritmului Dijkstra, folosit pentru a găsi cel mai scurt drum între două puncte, dar cu o îmbunătățire: utilizează o funcție euristică pentru a ghida căutarea către destinație. Această funcție estimează distanța rămasă de la un nod la destinație, permițând ca expansiunea să fie direcționată eficient. Dacă euristica nu supraestimează costul, algoritmul A\* garantează că va găsi drumul optim. O euristică admisibilă și consistentă asigură rezultatele corecte și rapide.

În Dijkstra, prioritatea unui nod este doar costul minim cunoscut pentru a ajunge la el, în timp ce în A\*, prioritatea include acest cost și o estimare a distanței rămase până la destinație, astfel diferența este că A\* adaugă euristica la coada de priorități.

Această abordare permite economisirea timpului explorând mai eficient căile către destinație și reducând timpul petrecut pe direcții nepotrivite.

Pentru a îmbunătăți eficiența algoritmului A\*, putem defini o funcție euristică care estimează cât de aproape suntem de obiectiv. În grafurile ponderate pe muchii, unde nodurile reprezintă locații fizice, iar ponderea muchiilor reprezintă distanțele între locații, euristica poate fi distanța euclidiană între două puncte, (x1,y1) și (x2,y2), conform formulei (2.1).

(2.1)

Aceasta oferă o estimare a costului minim necesar pentru a ajunge de la un nod la destinație.[6]

Complexitatea temporală depinde de euristică .În cel mai rău caz complexitate spațială este O(bd) = O(V), unde d reprezintă cea mai scurtă cale, iar b reprezintă factorul de ramificare, deoarece stochează toate nodurile generate în memorie. La fel ca în cazul lui Dijkstra fiecare adăugare sau eliminare a unui element din coada de prioritate se realizează în O(log V) timp, astfel complexitatea temporală este O((V+E)\*log V).

În figura 2.3 este ilustrată modul de funcționare a algoritmului A\* , utilizând o funcție euristică, pentru a găsi traseul cel mai scurt de la un punct sursă la un punct destinație.



**Figura 2.3 – Ilustrație a algoritmului A\***

## **2.5 Argumentarea alegerii algoritmilor**

Alegerea algoritmilor potriviți reprezintă un punct-cheie în rezolvarea unei probleme. În funcție de tipul probleme este necesar un algoritm specific. În cazul problemei navigării prin labirint, algoritmii selectați – BFS, A\* și Dijkstra pot fi considerați o soluție potrivită. Fiecare algoritm are particularități diferite, cu toate acestea toți duc la același rezultat: găsirea drumului minim.

Algoritmul BFS străbate graful în cel mai scurt timp posibil, nefiiind niciodată prins într-o buclă infinită. Acesta este un algoritm simplu și eficient pentru găsirea celui mai scurt drum într-un graf neponderat, cum ar fi un labirint în care fiecare mișcare are același cost (ex. de la o celulă la una vecină). Garantează găsirea celui mai scurt drum dacă există unul, deoarece explorează toate nodurile la o anumită distanță înainte de a trece la următoarea distanță. Totodată acest algoritm este ușor de implementat fiind potrivit pentru probleme precum navigarea într-o matrice bidimensională.

Algoritmul Dijkstra este util pentru găsirea drumului minim într-un graf ponderat, spre deosebire de BFS, Dijkstra utilizează o coadă de prioritate care reduce timpul de execuție a algoritmului. Dezavantajul algoritmului este că acesta nu poate funcționa cu ponderi negative. Algoritmul prioritizează explorarea nodurilor cu cel mai mic cost total acumulat, asigurând că soluția găsită este optimă. Într-un labirint, acest lucru poate însemna găsirea unui drum mai lung ca distanță, dar mai ieftin ca și cost total.

Algoritmul A\* spre deosebire de algoritmul BFS, care explorează uniform în toate direcțiile, folosește o funcție euristică pentru a prioritiza/favoriza nodurile care par mai aproape de destinație. Acest lucru reduce foarte mult numărul de noduri vizitate, ceea ce face algoritmul mai rapid în practică. Labirinturile pot fi reprezentate ca grafuri cu noduri și muchii, iar A\* lucrează foarte bine cu astfel de structuri și poate fi aplicat în labirinturi de dimensiuni mari.

Utilizarea acestor algoritmi permite o abordare diferită a problemei, dar soluția obținută fiind aceeași pentru toți algoritmii.

# **3 IMPLEMENTAREA JOCULUI**

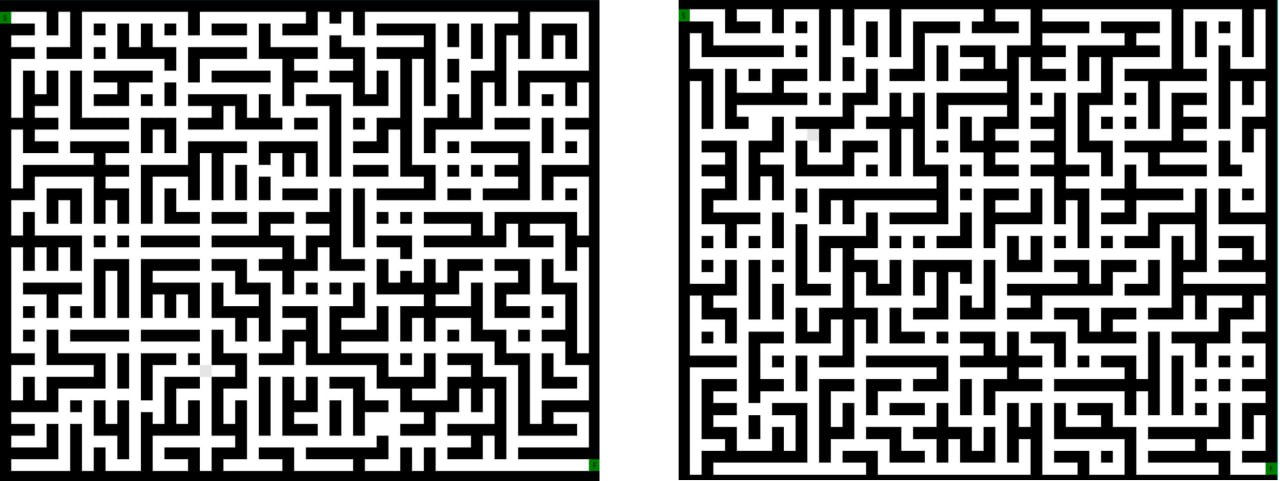
Maze Navigator este un joc unde scopul principal este colectarea punctelor de control cât mai

rapid posibil. Jocul dat a fost scris in limbajul C# utilizând Windows Forms. Windows Forms este o bibliotecă de clase GUI care este inclusă în .Net Framework. Scopul său principal este de a oferi o interfață mai ușoară pentru dezvoltarea aplicațiilor pentru desktop, tabletă etc. Aplicațiile care sunt dezvoltate utilizând Windows Forms sunt cunoscute sub numele de Aplicații Windows Forms care rulează pe computerul desktop.

## **3.1 Implementarea algoritmilor în limbajul de programare C#**

## **3.1.1 Implementarea algoritmului lui Prim**

Pentru realizarea jocului, la generarea labirintului s-a utilizat algoritmul lui Prim aleatoriu. În figura 3.1 sunt reprezentate două modele de labirinturi generate.



**Figura 3.1 – Labirinturi generate de algoritmul lui Prim aleatoriu**

Implementarea algoritmului de generare a labirintului a fost realizată urmând o serie de pași. Labirintul este reprezentat sub forma unui tablou bidimensional de celule, fiecare celulă având una dintre cele două stări: traseu sau obstacol. Procesul începe cu un tablou complet plin de obstacole, fără trasee accesibile. Mai jos este redat implementarea acestui algoritm la nivel de cod.

public void GenerateMaze()

{

int i = random.Next(1, rows - 1);

int j = random.Next(1, rows - 1);

Cell randomCell = new Cell(i, j);

SetCellAsPassage(randomCell);

FindAndSaveCellFrontiers(randomCell);

while (frontiers.Count > 0)

{

int randomFrontier = random.Next(0, frontiers.Count);

Cell cell = frontiers[randomFrontier];

SetCellAsPassage(cell);

FindAndSaveCellFrontiers(cell);

MakePassge(cell);

}

RemoveWall();

}

Se începe prin alegerea aleatorie a unei celule din tablou, care este setată ca traseu. Ulterior, se creează o listă de frontiere, unde frontiera este definită ca o celulă aflată la două unități de distanță de o celulă traseu, care este obstacol și se află în limitele tabloului.

Cât timp lista de frontiere nu este goală, se selectează aleatoriu o celulă din această listă, se marchează ca traseu și se generează noi frontiere pe baza acesteia. Noile frontiere valide sunt adăugate în listă, iar celula procesată este eliminată din listă.

Acest proces continuă până când toate celulele din lista de frontiere au fost procesate, rezultând astfel un labirint generat aleatoriu.

## **3.1.2 Implementarea algoritmului BFS**

Implementarea acestui algoritm este una destul de simplă. Mai jos este arătat algoritmul la nivel de cod.

public static int BFSearch(int[,] mat, BFSPoint src, BFSPoint dest)

{

int ROW = mat.GetLength(0);

int COL = mat.GetLength(1);

parent = new BFSPoint[ROW, COL];

if (mat[src.x, src.y] != 1 || mat[dest.x, dest.y] != 1)

return -1;

bool[,] visited = new bool[ROW, COL];

visited[src.x, src.y] = true;

Queue<queueNode> q = new Queue<queueNode>();

queueNode s = new queueNode(src, 0);

q.Enqueue(s);

while (q.Count != 0)

{

queueNode curr = q.Peek();

BFSPoint pt = curr.pt;

if (pt.x == dest.x && pt.y == dest.y)

{

GetPath(parent, src, dest);

return curr.dist;

}

q.Dequeue();

for (int i = 0; i < 8; i++)

{

int row = pt.x + rowNum[i];

int col = pt.y + colNum[i];

if (isValid(row, col, ROW, COL) && mat[row, col] == 1 && !visited[row, col])

{

visited[row, col] = true;

queueNode Adjcell = new queueNode(new BFSPoint(row, col), curr.dist + 1);

parent[row, col] = new BFSPoint(pt.x, pt.y);

q.Enqueue(Adjcell);

}}

}

Algoritmul pornește de la celula sursă, utilizând metoda BFS pentru parcurgere. Este inițializată o coadă care stochează coordonatele celulelor din matrice, având ca prim element celula sursă. În același timp, se creează o matrice de tip bool numită visited, de aceeași dimensiune, în care toate elementele sunt inițializate cu valoarea fals. Funcționarea algoritmului este următoarea: cât timp coada nu este goală, se extrage celula din fața cozii și se verifică dacă coordonatele acesteia corespund destinației. În cazul în care destinația este atinsă, algoritmul se oprește și returnează rezultatul. În caz contrar, se analizează cele patru celule adiacente ale celulei curente. Dacă o celulă adiacentă are valoarea 1 și nu a fost vizitată, aceasta este adăugată în coadă și marcată ca vizitată. Procesul continuă până când destinația este găsită sau toate celulele accesibile din matrice au fost explorate.

Modul de explorare standard al algoritmului BFS este în patru direcții: sus, jos, stânga și dreapta. La nivel de cod, direcțiile posibile de explorare ale algoritmului sunt reprezentate mai jos, sub formă de 2 tablouri unidimensionale.

static int[] rowNum = { -1, 0, 0, 1};

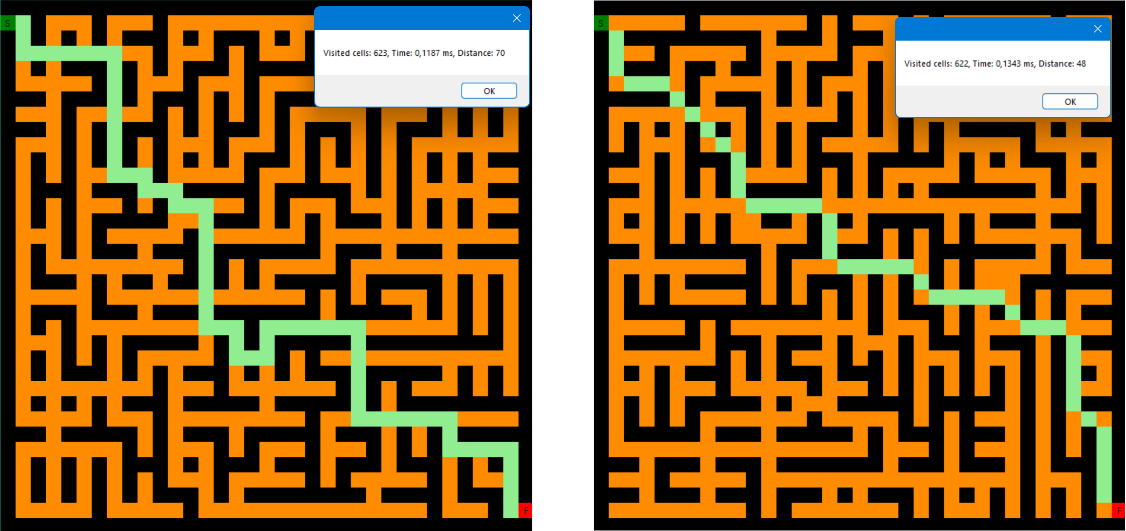
static int[] colNum = { 0, -1, 1, 0};

Fiecare pereche de elemente din tablouri reprezintă direcția care urmează sa fie explorată. Ținând cont că algortimul A\* explorează în opt direcții, traseul obținut este întotdeauna mai scurt. Astfel BFS a fost adaptat pentru opt direcții, codul modificat fiind redat mai jos.

static int[] rowNum = { -1, 0, 0, 1, -1, -1, 1, 1};

static int[] colNum = { 0, -1, 1, 0, -1, 1, -1, 1};

Modificarea constă în adăugarea de noi direcții suplimentare care reprezintă mișcarea pe diagonale. În figura 3.2 sunt reprezentate două rezultate în urma rulării algoritmului BFS.



**Figura 3.2 – Rezultatele obținute în urma algoritmului BFS**

Primul labirint este algoritmul BFS standard, explorarea fiind doar în patru direcții, iar al doilea labirint este algoritmul BFS modificat ulterior să exploreze în 8 direcții. Celule de culoare verde reprezinătă traseul minim de la punctul de start până la punctul de final. Celulele de culoare oranj sunt celulele care au fost vizitate de către algoritm pe parcursul eplorării. De asemenea, în colțul din dreapta sus se poate observa un mesaj afișat la finalul execuției care conține datele despre rezultatul obținut și anume: numărul total de celule vizitate, timpul de execuție cât și distanța drumului minim găsit.

## **3.1.3 Implementarea algoritmului Dijkstra**

Algoritmul Dijkstra după cum a fost menționat anterior, este utilizat în special pentru găsirea drumului minim în grafurile ponderate, unde distanțele dintre muchii sunt diferite. În cazul dat, labirintul reprezintă un graf neponderat, unde lungimea muchiilor este aceeași pentru toate egală cu 1. Adaptând la această situație, algoritmul este asemănător algoritmului BFS, diferența fiind utilizarea unei cozi de prioritate. În versiunea limbajului de programare utilizat coada de prioritate nu este disponibilă, de aceea s-a utilizat o structură de date numită SortedSet, care funcționează la fel ca o coadă de prioritate. La nivel de cod aceasta arată în modul următor.

SortedSet<(int, DijPair)> openList = new SortedSet<(int, DijPair)>(Comparer<(int, DijPair)>.Create((a, b) =>

{

int result = a.Item1.CompareTo(b.Item1);

if (result == 0)

return a.Item2.Row != b.Item2.Row ? a.Item2.Row.CompareTo(b.Item2.Row) : a.Item2.Col.CompareTo(b.Item2.Col);

return result;

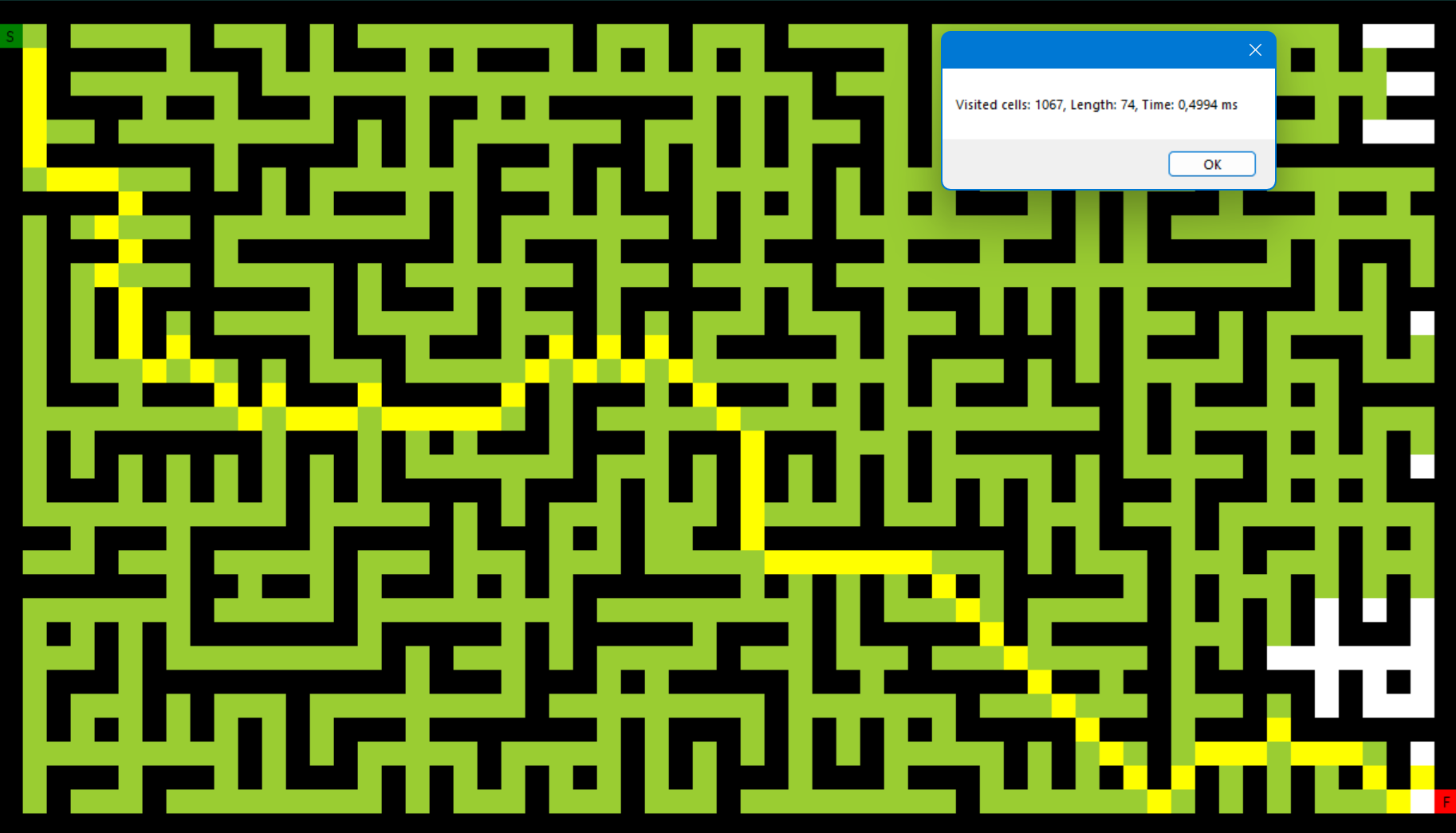
}));

Structura SortedSet<(int, DijPair)> este utilizată pentru a stoca o colecție de elemente care sunt sortate automat conform unui criteriu specific. Fiecare element din colecție este o pereche formată dintr-un număr întreg și un obiect de tip DijPair. Ordonarea elementelor din această colecție este definită de un comparator, care decide ordinea.

Comparatorul începe prin a compara primul element al fiecărei perechi, adică valoarea de tip întreg. Aceasta este considerată prioritatea elementului, astfel încât elementele cu o valoare mai mică sunt considerate mai importante și sunt plasate înaintea celor cu valori mai mari. În cazul în care două elemente au aceeași valoare, comparatorul trece la compararea celui de-al doilea element al perechii, obiectul DijPair.

Obiectul DijPair are doi parametri: Row (rândul) și Col (coloana), care reprezintă coordonatele celulei. Comparatorul verifică mai întâi parametrul Row. Dacă aceasta este diferită între cele două perechi comparate, sortarea se realizează pe baza valorii Row. În cazul în care valorile Row sunt egale, comparatorul verifică parametrul Col.

Când un nou element este adăugat în SortedSet, acesta este sortat imediat după valoarea lui. În figura 3.3 este reprezentat unul dintre rezultatele algoritmului Dijkstra, unde celulele de culoare galbenă reprezintă drumul minim găsit, celulele de culoare verde-pal reprezintă celulele vizitate. În imagine se pot observa și celule de culoare albă, care sunt nevizitate.



**Figura 3.3 – Rezultatul obținut în urma aloritmului Dijkstra**

Ca și în cazul algoritmului BFS, inițial algoritmul Dijkstra parcurge în patru direcții. Astfel, acesta a fost adaptat să parcurgă labirintul în 8 direcții, oferind o cale mult mai scurtă și mai precisă.

## **3.1.4 Implementarea algoritmului A\***

Algoritmul A\* este o metodă foarte bună pentru găsirea drumului minim într-un graf. A\* este o îmbunătățire a algoritmilor Dijkstra și BFS. Acesta folosește o funcție euristică, care estimează distanța dintre dintre celula curentă și celula destinație. În limbajul de programare această funcție arată în modul următor:

public static double CalculateHValue(int row, int col, Pair dest)

{

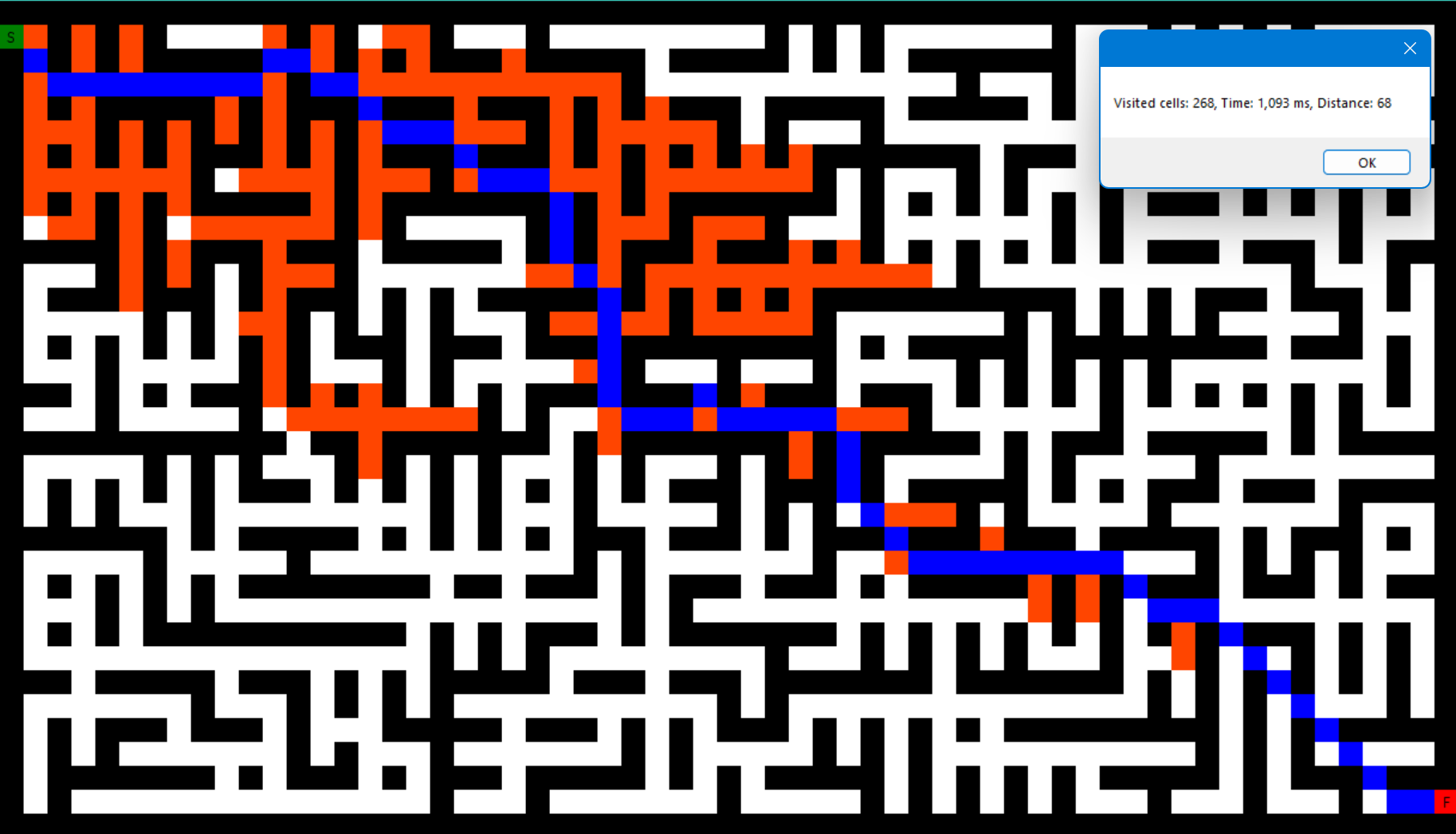
return Math.Sqrt(Math.Pow(row - dest.first, 2) + Math.Pow(col - dest.second, 2));

}

Funcția CalculateHValue este utilizată pentru a calcula valoarea euristică h(n). Aceasta folosește distanța euclidiană pentru a estima cât de departe este un nod curent de destinație. Funcția primește trei parametri: coordonatele row și col ale nodului curent, și un obiect Pair care reprezintă coordonatele destinației, cu valorile sale accesibile prin dest.first (pentru rând) și dest.second (pentru coloană). În interiorul funcției, distanța euclidiană este calculată utilizând formula (2.1).

Mai exact, funcția calculează diferențele dintre rândurile și coloanele celor două puncte, le ridică la pătrat pentru a elimina semnele negative, apoi adună aceste pătrate. În final, suma obținută este trecută prin funcția Math.Sqrt pentru a calcula rădăcina pătrată, obținând astfel distanța euclidiană dintre cele două puncte. Rezultatul este returnat ca un număr de tip double.

Codul complet al acestui algoritm se regăsește în Anexa A. În continuare, în figura 3.4, se poate observa un rezultat obținut în urma algoritmului A\*.

  
**Figura 3.4 – Rezultatul obținut în urma aloritmului A\***

Din figură de mai sus, se vede că numărul celulelor roșii, care semnifică celulele vizitate, este mai mic comparativ cu ceilalți algoritmi. Acest lucru se datoreză funcției sale euristice.

## **3.2 Analiza algoritmilor**

Având implementați toți trei algoritmi, a fost făcută o serie de teste pentru a analiza performanța acestora. Analiza constă în generarea a diferitor labirinturi cu diferite dimensiuni, iar compararea algoritmilor are loc după timpul de execuție, numărul de celule vizitate și distanța drumului minim găsit. În continuare este prezentat rezultatul testării în funcție de numărul de celule vizitate în tabelul 3.1.

**Tabelul 3.1 – Numărul de celule vizitate**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Dimensiunea** | **15x35** | **25x40** | **35x50** | **50x75** | **75x100** |
| **BFS** | 224 | 517 | 786 | 1995 | 3745 |
| **Dijkstra** | 223 | 517 | 786 | 1992 | 3719 |
| **A\*** | 142 | 258 | 128 | 540 | 731 |

Conform datelor din tabelul 3.1, rezultă că BFS și Dijkstra vizitează un număr similar de celule pentru fiecare dimensiune, datorită comportamentului lor similar. Totodată A\* este mult mai eficient, vizitând mai puține celule, datorită funcției sale euristice. Se poate de observat că diferența dintre A\* și ceilalți algoritmi devine mai evidentă pe dimensiuni mai mari, unde A\* vizitează cu mult mai puține celule (731 față de 3745 pentru 75x100). Astfel din punct de vedere al numărului de celule vizitate, A\* este algoritmul cu un rezultat mai favorabil.

În cele ce urmează, în tabelul 3.2 sunt afișate datele privind analiza algoritmilor în funcție de timpul de execuție.

**Tabelul 3.2 – Timpul de execuție în milisecunde al algoritmilor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Dimensiunea** | **15x35** | **25x40** | **35x50** | **50x75** | **75x100** |
| **BFS** | 0,0532 | 0,1112 | 0,2075 | 0,5043 | 0,934 |
| **Dijkstra** | 0,1471 | 0,2909 | 0,3398 | 1,6503 | 2,344 |
| **A\*** | 0,2587 | 0,2625 | 0,2711 | 0,8794 | 1,3199 |

Pe baza tabelului de mai sus, s-a construit graficul din figura 3.5 pentru o analiză mai clară asupra eficienței algoritmilor.

**Figura 3.5 – Timpul de executie în formă grafică**

Analizând datele obținute, se observă că BFS are cel mai scurt timp de execuție pentru toate dimensiunile, deoarece nu calculează costuri suplimentare. Dijkstra este mai lent, deoarece calculează costurile pentru fiecare nod, ceea ce crește timpul său de excuție, mai ales pe dimensiuni mai mari. În schimb, A\* este mai eficient decât Dijkstra deoarece utilizează o funcție euristică, care reduce numărul de noduri vizitate.

Deși acesta vizitează mai puține noduri decât BFS, are un timp de execuție mai mare datorită calculelor suplimentare și utilizării unei cozi de prioritate, care de asemenea consumă timp. Astfel, privind timpul de execuție, algoritmul BFS este cel mai favorabil pentru cazul dat.

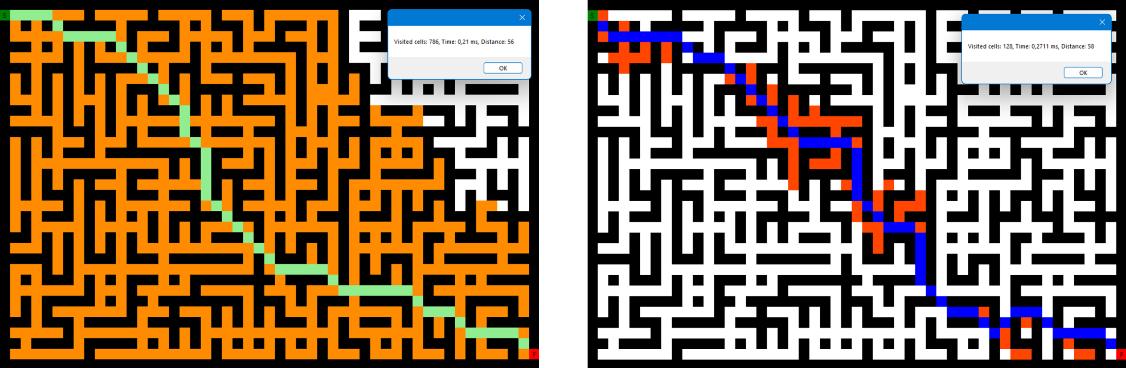
Pentru a vedea cât de corect funcționează algoritmii, se vor analiza rezultatele distanțelor drumului minim pentru fiecare caz în parte, prezentat în tabelul 3.3.

**Tabelul 3.3 – Distanța drumului minim**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Dimensiunea** | **15x35** | **25x40** | **35x50** | **50x75** | **75x100** |
| **BFS** | 32 | 52 | 56 | 94 | 132 |
| **Dijkstra** | 32 | 52 | 56 | 94 | 132 |
| **A\*** | 34 | 52 | 58 | 94 | 132 |

Pe baza rezultatelor din tabelul 3.3, se poate observa că BFS și Dijkstra oferă aceleași rezultate pentru distanța drumului minim în toate cazurile, garantând astfel mereu rezultatul corect. A\* în schimb, în două din cinci cazuri, a oferit rezultate diferite și anume o distanță mai mare a drumului minim, ceea ce poate fi cauzat de funcția euristică a acestuia, care nu garantează mereu găsirea drumului minim.

De asemenea pentru cazul al treilea, în figura 3.6 sunt atașate drumurile minime găsite de către algoritmii BFS și A\*.



**Figura 3.6 – Diferența dintre drumurile minime găsite de algoritmii BFS și A\***

În imaginea din stânga este prezentat labirintul care utilizează algoritmul BFS, traseul găsit fiind cel mai scurt, cu distanța de 56. În schimb, în imaginea din dreapta este ilustrat labirintul rezolvat folosind algoritmul A\*, traseul fiind mai lung în comparație cu BFS, având distanța de 58.

Diferența pare a fi mica, fiind de doar două unități. Totuși, pe parcursul testării algoritmilor, au fost încercate mai multe cazuri, selectând de mai multe ori punctele de start și final manual, două celule la dorință din labirint, pentru a putea analiza comportamentul algoritmilor în diferite situații.

Prin urmare, rezulatele de asemenea au fost uneori diferite, cu diferența dintre drumurile minime găsite fiind mult mai mare. Unul dintre aceste rezultate avea o diferență de distanțe a drumurilor egală cu 12 unități. Drumul rezultat de algoritmii BFS și Dijskstra aveau aceeași distanță egală cu 72. Algoritmul A\* însă, a rezultat un drum cu o distanță egală cu 84. Acest caz este afișat mai jos, în figura 3.7.



**Figura 3.7 - Caz obținut în urma testării algoritmilor**

Așadar, în figura 3.7 se poate observa clar diferența majoră dintre drumurile găsite de algoritmul BFS, imaginea din stânga, și algortimul A\*, imaginea din dreapta. Deși algoritmul BFS a explorat mai multe celule, drumul rezultat este mult mai mic, fapt ce îl face mai eficient.

Prin urmare, analizând rezultatele obținute în urma testării algoritmilor, fiecare algoritm are propriile sale avantaje și dezavantaje. Dijkstra garantează întotdeauna găsirea drumului minim în labirint, totuși, comparativ cu BFS este mai lent deoarece efectuează calcule suplimentare. Deși este inferior algoritmului BFS în cazul labirintului, Dijkstra este un algoritm foarte bun, și este o alegere perfectă pentru grafurile ponderate cu valori pozitive.

Algoritmul BFS, la fel ca și Dijsktra, vizitează un număr mare de celule comparativ cu algoritmul A\*, totuși a avut cel mai scurt timp de execuție pentru toate dimensiunile labirinturilor. Chiar dacă A\* vizitează mai puține celule, calculele adiționale, euristica și utilizarea unei cozi de prioritate, îl fac mai lent decât BFS. De asemenea, cel mai important, BFS oferă garanția că drumul minim găsit de acesta este mereu cel mai scurt, spre deosebire de A\*, care poate găsi drumuri mai lungi datorită funcției sale euristice.

Aceste caracteristici demonstrează faptul că în cazul dat, unde labirintul reprezintă un graf neponderat, algoritmul BFS este alegerea cea mai bună, fapt pentru care v-a fi utilizat în scopul final al jocului.

## **3.3 Implementarea interfeței grafice.**

Jocul dat a fost scris în limbajul de programare C#. Pentru crearea interfeței grafice a fost utilizat Windows Forms. Windows Forms este un framework UI special pentru crearea aplicațiilor Windows. Utilizând compilatorul Visual Studio, acest framework oferă o posibilitate simplă de a crea un design vizual mult mai ușor de implementat, deoarece include un set predefinit de instrumente care pot fi accesate prin intermediul unui meniu numit “toolbox”. Astfel, este ușor de utilizat de către dezvoltatori, mai ales pentru începători, fapt ce a determinat alegerea noastră.

Interfața grafică a jocului a fost creată atât la nivel de cod cât și în secțiunea Design cu instrumente grafice de care dispune Windows Forms. Câteva din instrumentele date, care au fost folosite și în cadrul acestui proiect sunt:

* button;
* label;
* panel;
* tablelayoutpanel;
* textbox.

Utilizând secțiunea Design, aceste instrumente pot fi folosite prin intermediul „drag and drop”, fiind plasate pe suprafața de design, care reprezintă și interfața grafică. Editarea acestora este posibilă direct din meniul disponibil în Visual Studio sau la nivel de cod.

La nivel de cod, inițializarea câtorva din caracteristicile unui instrument arată în felul următor.

this.button1.BackColor = System.Drawing.Color.FromArgb(((int)(((byte)(69)))), ((int)(((byte)(69)))), ((int)(((byte)(69)))));

this.button1.FlatStyle = System.Windows.Forms.FlatStyle.Popup;

this.button1.Font = new System.Drawing.Font("Microsoft Sans Serif", 18F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(204)));

this.button1.ForeColor = System.Drawing.Color.White;

this.button1.Location = new System.Drawing.Point(62, 478);

this.button1.Name = "button1";

this.button1.Size = new System.Drawing.Size(382, 57);

this.button1.Text = "Generate";

this.button1.Click += new System.EventHandler(this.button1\_Click);

Ce reprezintă totuși aceste caracteristici? Unele dintre ele determină aspectul butonului. Backcolor definește culoarea de fundal a butonului. Aceasta poate fi aleasă utilizând un cod RGB sau din culorile disponibile din program. FlatStyle reprezintă stilul butonului, cum ar fi o formă a marginilor modificată sau un aspect simplu. Text reprezintă caracterele afișate pe buton, acestea în mare parte semnificând funcția butonului, adică acțiunea pe care o va realiza. Font are ca scop stabilirea dimensiunii și stilului textului afișat, iar ForeColor reprezintă culoarea acestuia. Location indică poziția butonului pe suprafață, coordonatele inițiale fiind din colțul stânga sus. Size definește dimensiunile butonului, lățimea, inălțimea și totodată spațiul ocupat pe suprafață. Click reprezintă acțiunea ce urmează după apăsarea butonului, indicând ce cod va fi executat făcând un simplu click pe acesta.

Interfața jocului constă din două instrumente panel, unul reprezintă meniul cu butoane, iar al doilea instrument reprezintă secțiunea pe care se generează și se afișează labirintul. Labirintul a fost creat prin intermediul instrumentului tablelayoutpanel. Acesta constă dintr-o grilă, formată din butoane, fiecare buton reprezentând o celulă din labirint. Inițial, instrumentul tablelayoutpanel nu conține niciun element, iar mai apoi, toate modificările asupra acestuia, adică adăugarea celulelor sub formă de butoane, au loc dinamic la nivel de cod.

După ce utilizatorul introduce dimensiunea labirintului, înălțimea și lățimea, are loc modificare grilei printr-o funcție asincronă, care a fost realizată în felul următor.

private async Task DrawMazeFromButtonsOnUI()

{

await Task.Run(() =>

{

buttons = new Button[rows, columns];

this.tableLayoutPanel1.Invoke(new Action(() =>

{

ResizeGrid();

for (int row = 0; row < rows; row++)

{

this.tableLayoutPanel1.RowStyles.Add(new RowStyle(SizeType.Absolute, buttonSize));

}

for (int col = 0; col < columns; col++)

{

this.tableLayoutPanel1.ColumnStyles.Add(new ColumnStyle(SizeType.Absolute, buttonSize));

}

}));

for (int row = 0; row < rows; row++)

{

for (int col = 0; col < columns; col++)

{

buttons[row, col] = new Button

{

BackColor = primsGenerator.maze[row, col] != 0 ? Color.White : Color.Black,

Dock = DockStyle.Fill,

FlatStyle = FlatStyle.Flat,

Margin = new Padding(0),

};

buttons[row, col].FlatAppearance.BorderSize = 0;

this.tableLayoutPanel1.Invoke(new Action(() => this.tableLayoutPanel1.Controls.Add(buttons[row, col], col, row)));

}

}

});

}

Funcția DrawMazeFromButtonsOnUI se ocupă de desenarea labirintului pe interfața utilizatorului. Aceasta este o funcție ascincronă pentru a evita blocarea interfeței grafice principale. Inițial se inițializează o matrice bidimensională de butoane, numită buttons, ce corespunde dimensiunilor introduse de către utilizator. Apoi, utilizând funcția invoke, se asigură că modificările ce urmează vor fi aplicate asupra interfeței grafice principale.

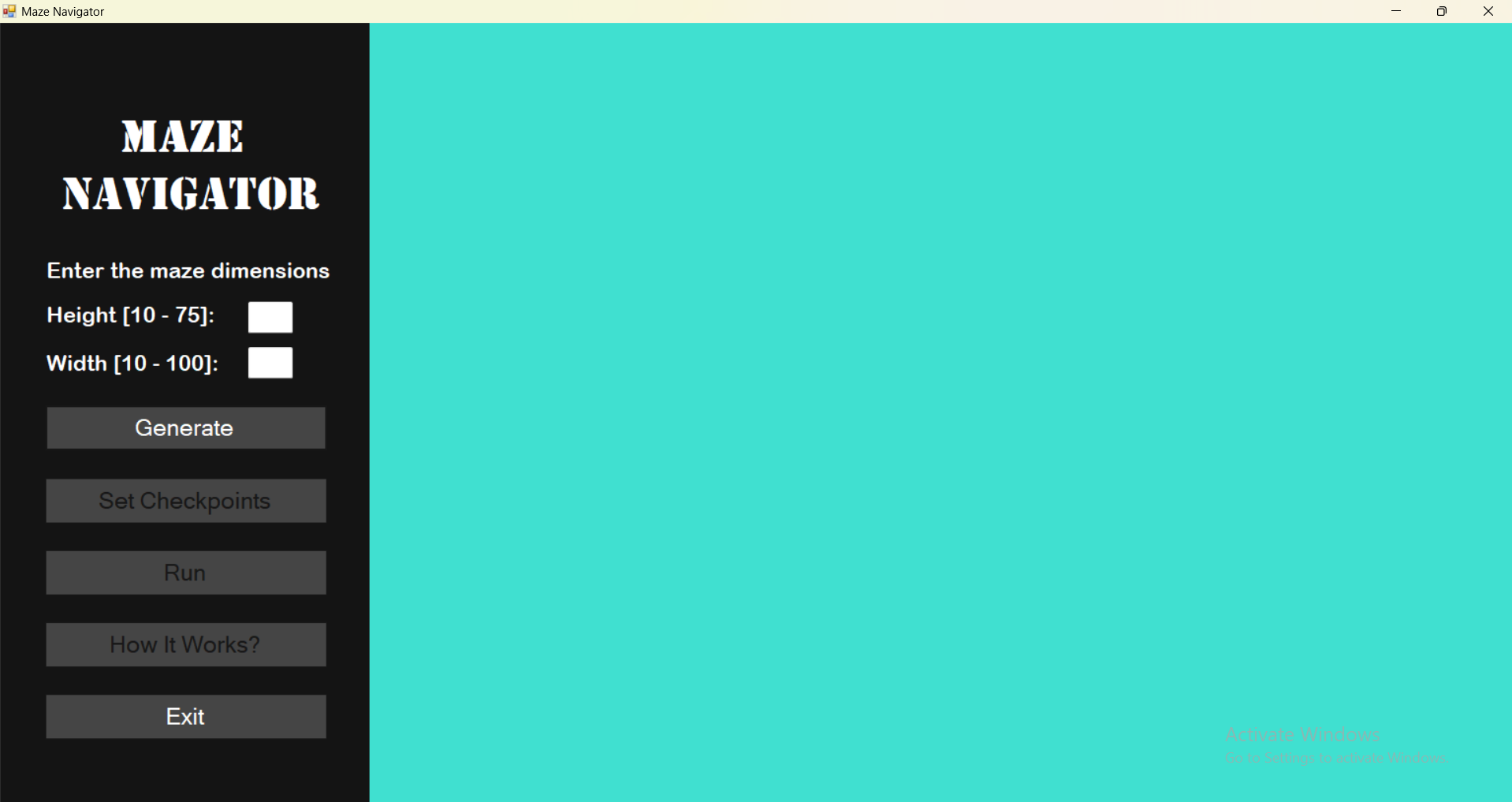
Se apelează întâi funcția ResizeGrid, în interiorul căreia, dacă pe interfață există deja un labirint, acesta se șterge, se redimensionează grila și se poziționează în centrul ferestrei. Urmează două metode for în care se parcurg rândurile și coloanele grilei și se setează dimensiunea fiecărei celule comform valorii variabilei buttonSize, care semnifică numărul de pixeli.

Ulterior, pentru fiecare celulă din labirint se creează un buton și se setează proprietățile acestuia. Dacă celula din matricea generată primsGenerator.maze are valoarea 0, adică este un perete, butonul primește culoarea neagră, dacă însă are valoarea 0, adică este o cale liberă, butonul primește culoarea albă. Se setează apoi încă trei proprietăți pentru buton care determină aspectul acestuia. Proprietatea DockStyle.Fill va modifica butonul ca acesta sa ocupe tot spațiul disponibil alocat. FlatStyle.Flat setează aspectul butonului în așa mod, încât acesta să fie simplu, fără umbre sau marginile ridicate. A treia proprietate Margin = new Padding(0)elimină spațiul dintre butoane, acesta fiind setat la 0 pixeli.

Fiecare buton este adăugat în tableLayoutPanel la poziția corespunzătoare. Funcția afișează vizual labirintul complet pe interfața utilizatorului, unde fiecare celulă este reprezentată de un buton personalizat.

## **3.3 Interacțiunea cu utilizatorul**

La rularea programului, apare fereastra de bază a jocului. În partea stângă a ferestrei se află un meniu, care conține titlul jocului și butoanele care execută anumite acțiuni, iar pe cealaltă parte a suprafaței din ferestrei, momentan, spațiul este gol, acolo urmând să se afișeze labirintul generat. În figura 3.8 este afișată această fereastră.

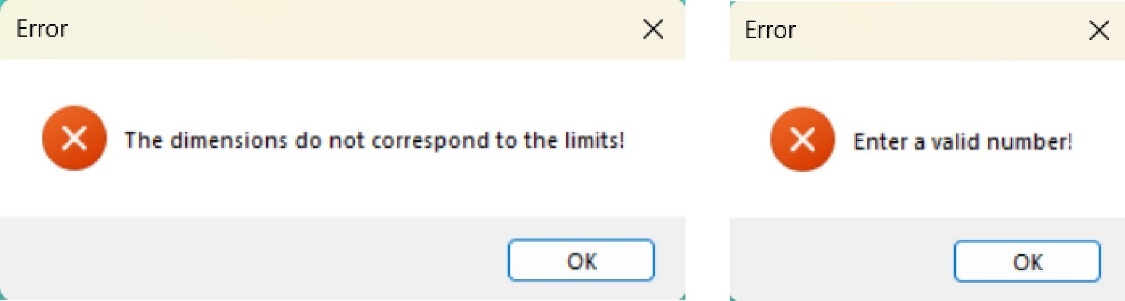


**Figura 3.8 – Meniul de bază a jocului**

Respectiv, în imagine se poate observa denumirea jocului urmată de instrucțiuni pentru introducerea dimensiunilor. Primele câmpuri care trebuie completate sunt cele cu dimensiunea labirintului. Limitele dimensiunilor sunt între 10 și 75 de celule pentru înălțime și între 10 și 100 de celule pentru lățime. Mai jos utilizatorul are la dispoziție cinci butoane:

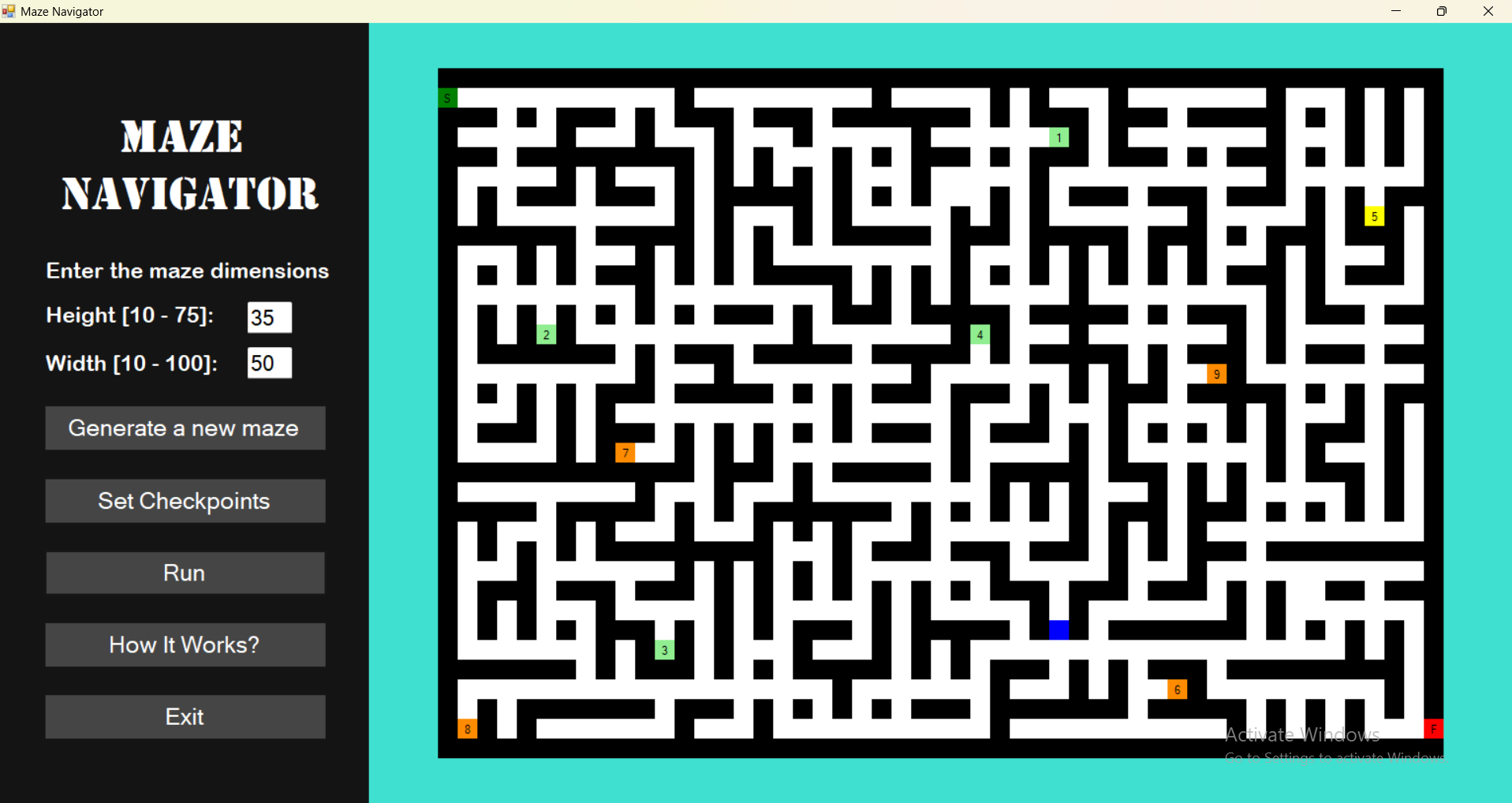
* butonul „Generate”;
* butonul „Set Checkpoints”;
* butonul „Run”;
* butonul „How It Works?”;
* butonul „Exit”.

Momentan utilizatorul poate folosi doar butoanele „Generate” și „Exit”, celelalte butoane fiind indisponibile. Butoanele respective vor fi disponibile abia după ce va fi generat labirintul deoarece acțiunile următoare pot fi realizate doar având un labirint la dispoziție. Astfel, utilizatorul introduce dimensiunile dorite și folosește butonul „Generate” pentru a genera labirintul. În cazul în care câmpurile vor fi completate cu alte dimensiuni decât cele acceptate sau cu alte caractere, la ecran vor fi afișate mesaje de eroare, reprezentate în figura 3.9.



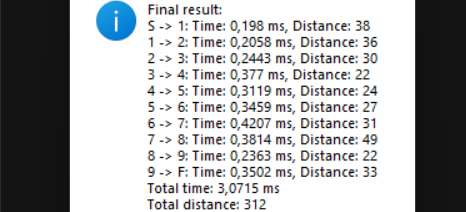
**Figura 3.9 – Mesaje de eroare în cazul completării incorecte a câmpurilor**

După generarea labirintului, restul butoanelor devin disponibile și utilizatorul poate executa următoarele acțiuni. Prin intermediul butonului „Set Checkpoints” utilizatorul poate plasa la dorință maxim 10 puncte de control pentru ca programul, cu ajutorul algoritmului BFS să înceapă navigarea prin labirint și să colecteze aceste puncte sau poate începe navigarea utilizând butonul „Run” fără a mai adăuga puncte de control adiționale, algoritmul căutând direct ieșirea din labirint. În figura 3.10 este afișat un labirint în timpul navigării conținând puncte de control adiționale.



**Figura 3.10 – Labirintul setat cu puncte de control**

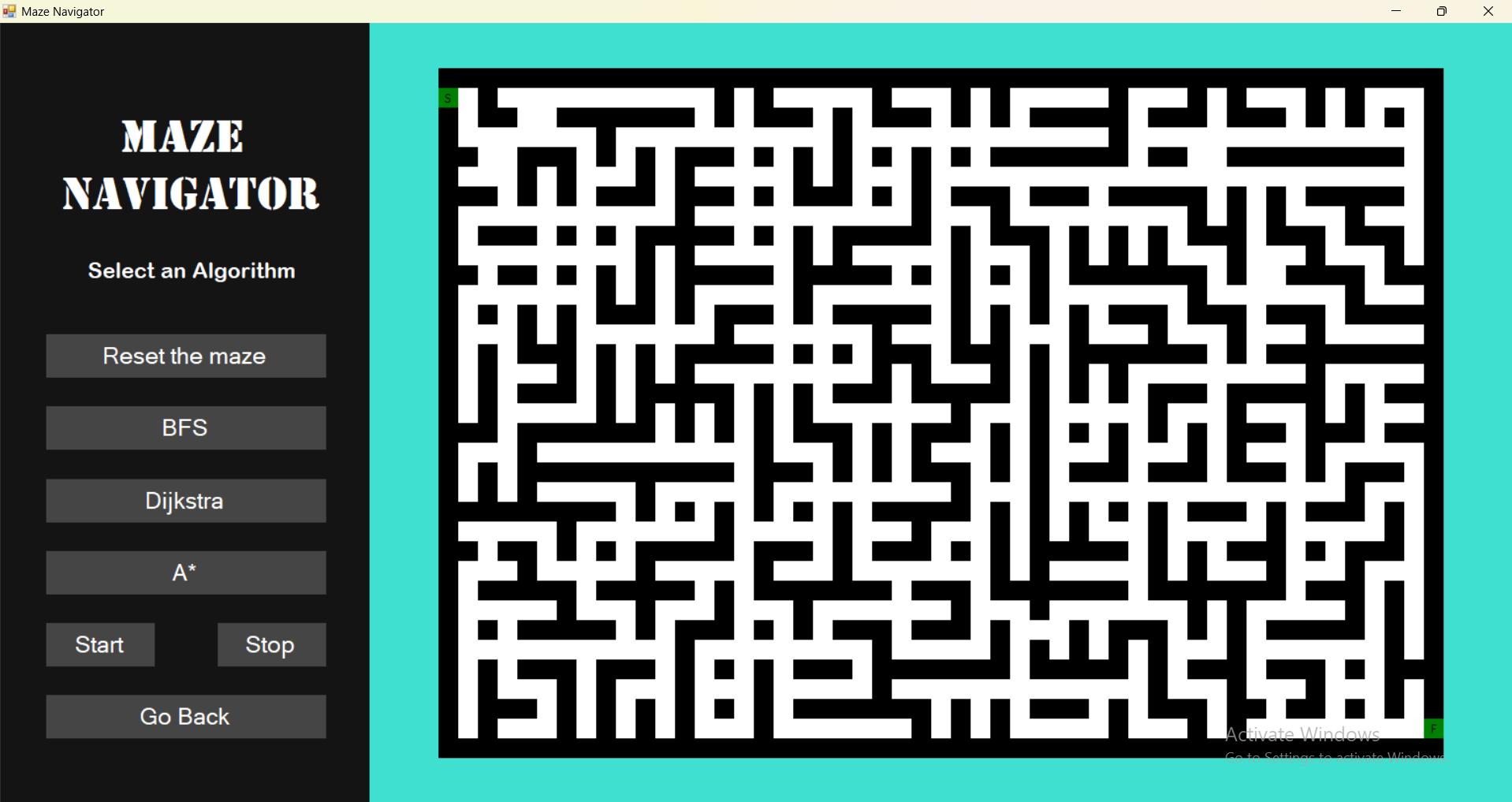
Punctul de start este afișat cu culoare verde și notat cu litera „S”, iar punctul de final este afișat cu culoare roșie și notat cu litera „F”. Poziția utilizatorului in labirint este reprezentată cu culoare albastră. Punctele de control sunt afișate cu trei culori diferite. Punctele de culoare verde sunt punctele deja colectate, punctul de culoare galbenă reprezintă punctul ce urmează a fi colectat, iar punctele de culoare oranj reprezintă punctele necolectate. Fiecare punct este notat cu cifra corespunzătoare ordinii în care acesta trebuie colectat. După ultimul punct de control, algoritmul se va îndrepta spre ieșirea din labirint. La finalul execuției, pe ecran va fi afișat rezultatul navigării, reprezentat în figura 3.11.



**Figura 3.11 – Rezultatul obținut în urma navigării**

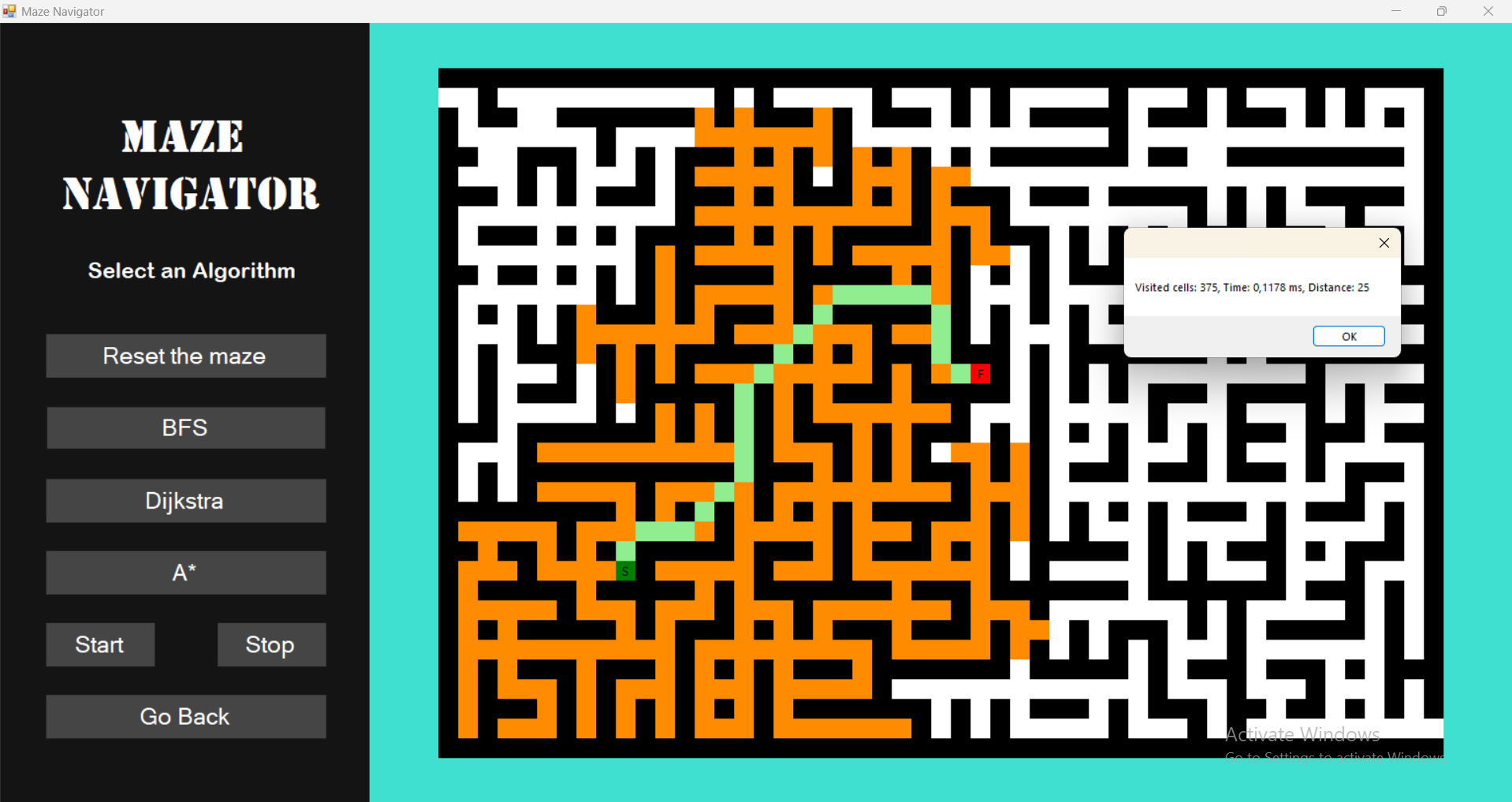
Rezultatul navigării conține timpul și distanța obținută între fiecare punct, iar mai jos, timpul și distanța totală de la start la final.

Meniul principal mai conține un buton numit „How It Works?”. Făcând click pe acest buton, utilizatorul va fi redirecționat într-un nou meniu în care va avea posibilitatea de a vedea modul de funcționare a fiecărui algoritm. În figura 3.12 este reprezentat cum arată meniul secundar.



**Figura 3.12 – Meniul secundar al jocului**

Din figura 3.12 se pot observa noile butoane de care dispune meniul secundar al jocului. După cum a fost menționat anterior, jocul dispune de posibilitatea de a vedea și analiza modul de funcționare al fiecărui algoritm. Prin selectarea unuia dintre cele trei butoane „BFS”, „Dijkstra” și „A\*”, programul va începe căutarea drumului minim utilizând algoritmul corespunzător butonului ales. De asemenea utilizatorul are posibilitatea de a selecta punctul de start și punctul de final la dorință prin intermediul butoanelor „Start” și „Stop”. În figura 3.13 este reprezentată o ilustrare a algoritmului BFS în care punctele de start și final au fost selectate manual.



**Figura 3.13 – Modul de funcționare al algoritmului BFS**

De asemenea, la sfarșitul execuției, este afișat și un mesaj cu rezultatul acesteia, care conține numărul de celule vizitate de către algoritm, timpul de execuție și distanța drumului minim găsit.

Pentru a selecta un nou algoritm sau pentru a reveni la meniul principal, utilizatorul trebuie mai întâi să șteargă progresul curent. Acest lucru se realizează utilizând butonul “Reset the maze” care resetează labirintul la starea sa inițială. Pentru a reveni înapoi la meniul principal, se apasă butonul “Go Back”.

# **CONCLUZII**

Jocul dezvoltat în acest proiect de an a avut ca scop principal crearea unei aplicații care permite navigarea eficientă prin labirinturi utilizând algoritmi de căutare, analizând eficiența și performanța algoritmilor. Prin realizarea acestui proiect s-a urmărit rezolvarea unei probleme întâlnite foarte des în diverse situații din viață, ce ține de găsirea și parcurgerea celui mai scurt drum, în special a unui labirint.

Algoritmii BFS, A\* și Dijkstra au fost implementați în scopul analizei și comparației performanței acestora în obținerea drumului minim, precum și în scopul înțelegerii importanței utilizării algoritmilor în rezolvarea problemelor.

Programul este ușor de înțeles și accesat datorită interfeței grafice simple, iar prin intermediul butoanelor utilizatorul poate interacționa cu meniul jocului. Datorită faptului că limbajul de programare C# permite realizarea atât a algoritmilor și a interfeței, legătura dintre ele a fost făcută destul de ușor, iar labirintul funcționează atât logic, cât și grafic.

În urma implementării și analizei celor trei algoritmi, s-a constatat că fiecare dintre algoritmi are atât avantaje, cât și dezavantaje în funcție de dimensiunea labirintului și modul de funcționare a acestora. Algoritmul BFS s-a dovedit a fi cel mai bun și cel mai rapid în ceea ce privește timpul de execuție, deoarece spre deosebire de Dijkstra acesta nu calculează costuri suplimentare. Atât BFS, cât și Dijkstra au oferit aceleași rezultate în ceea ce priveste distanța drumului minim.

În schimb, deși A\* este mai bun din punct de vedere al numărului de celule vizitate, totuși are un timp de execuție mai ridicat din cauza utilizării unei funcții euristice. Totodată, calculul funcției euristice face ca în unele cazuri drumul obținul de A\* să nu fie cel minim.

Obiectivele proiectului au fost atinse, întrucât au fost folosiți patru algoritmi, dintre care trei au fost nemijlocit pentru soluționarea problemei labirintului, iar algoritmul lui Prim a fost utilizat în scopul de a genera și crea labirintul propiu-zis. De asemenea, jocul oferă posibilitatea de a vizualiza în timp real parcurgerea labirintului de către algoritmi și drumul minim generat.

Prin intermediul checkpoint-urilor s-a reușit verificarea funcționării algoritmilor și în cazuri mai complexe, de unde s-a putut observa că acestea nu au împiedicat la găsirea rezultatelor corecte de către algoritmi.

În urma realizării acestui proiect, a fost demonstrat modul în care algoritmii de căutare pot fi utilizați pentru a rezolva probleme din viața reală. Jocul oferă o metodă interactivă de a înțelege cum funcționează acești algoritmi și ajută utilizatorii să analizeze și să compare logica acestora.

Jocul Maze Navigator este o metodă bună prin care se poate observa cum pot fi aplicați acești algoritmi în diverse domenii, cum ar fi labirinturile, hărțile sau aplicațiile de transport.

# **BIBLIOGRAFIE**

[1] Maze Generation Algorithms - An Exploration. The Randomized Prim's Algorithm [online]. Disponibil: <https://professor-l.github.io/mazes/> [citat 23 noiembrie 2024]

[2] TutorialsPoint. Breadth First Traversal (BFS) on a 2D Array using Java [online]. Disponibil: <https://www.tutorialspoint.com/breadth-first-traversal-bfs-on-a-2d-array-using-java> [citat 25 octombrie 2024].

[3] Breadth-first search [online]. Disponibil: <https://en.wikipedia.org/wiki/Breadth-first_search> [citat 23 noiembrie 2024]

[4] Amit, Patel. A Algorithm Comparison\*. Game Programming, Stanford University [online]. Disponibil: <https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html> [citat 25 octombrie 2024]

[5] A\* search algorithm [online]. Disponibil: [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm) [citat 25 noiembrie 2024]

[6] Emory University. A\* Algorithm. The Department of Mathematics and Computer Science, Oxford College [online]. Disponibil: <https://mathcenter.oxford.emory.edu/site/cs171/theAStarAlgorithm/> [citat 25 octombrie 2024]

[7] Codul curent este încărcat pe contul de Git-hub: <https://github.com/Victor45/Apa_Proiect>

# **ANEXA A**

**Algoritmul A\***

public static void A\_Star(int[,] grid, Pair src, Pair dest)

{

Aststopwatch.Reset();

Aststopwatch.Start();

Final.Clear();

AStVisited.Clear();

int ROW = grid.GetLength(0);

int COL = grid.GetLength(1);

if (!IsValid(src.first, src.second, ROW, COL) || !IsValid(dest.first, dest.second, ROW, COL))

{

return;

}

if (!IsUnBlocked(grid, src.first, src.second) || !IsUnBlocked(grid, dest.first, dest.second))

{

return;

}

if (src.first == dest.first && src.second == dest.second)

{

return;

}

bool[,] closedList = new bool[ROW, COL];

Cell[,] cellDetails = new Cell[ROW, COL];

for (int i = 0; i < ROW; i++)

{

for (int j = 0; j < COL; j++)

{

cellDetails[i, j].f = double.MaxValue;

cellDetails[i, j].g = double.MaxValue;

cellDetails[i, j].h = double.MaxValue;

cellDetails[i, j].parent\_i = -1;

cellDetails[i, j].parent\_j = -1;

}

}

int x = src.first, y = src.second;

cellDetails[x, y].f = 0.0;

cellDetails[x, y].g = 0.0;

cellDetails[x, y].h = 0.0;

cellDetails[x, y].parent\_i = x;

cellDetails[x, y].parent\_j = y;

SortedSet<(double, Pair)> openList = new SortedSet<(double, Pair)>(Comparer<(double, Pair)>.Create((a, b) => a.Item1.CompareTo(b.Item1)));

openList.Add((0.0, new Pair(x, y)));

bool foundDest = false;

while (openList.Count > 0)

{

(double f, Pair pair) p = openList.Min;

openList.Remove(p);

x = p.pair.first;

y = p.pair.second;

Pair visited;

visited.first = x;

visited.second = y;

closedList[x, y] = true;

AStVisited.Add(visited);

for (int i = -1; i <= 1; i++)

{

for (int j = -1; j <= 1; j++)

{

if (i == 0 && j == 0)

continue;

int newX = x + i;

int newY = y + j;

if (IsValid(newX, newY, ROW, COL))

{

if (IsDestination(newX, newY, dest))

{

cellDetails[newX, newY].parent\_i = x;

cellDetails[newX, newY].parent\_j = y;

AStdistance = cellDetails[x, y].g + 1.0;

TracePath(cellDetails, dest);

foundDest = true;

Aststopwatch.Stop();

return;

}

if (!closedList[newX, newY] && IsUnBlocked(grid, newX, newY))

{

double gNew = cellDetails[x, y].g + 1.0;

double hNew = CalculateHValue(newX, newY, dest);

double fNew = gNew + hNew;

if (cellDetails[newX, newY].f == double.MaxValue || cellDetails[newX, newY].f >= fNew)

{

openList.Add((fNew, new Pair(newX, newY)));

cellDetails[newX, newY].f = fNew;

cellDetails[newX, newY].g = gNew;

cellDetails[newX, newY].h = hNew;

cellDetails[newX, newY].parent\_i = x;

cellDetails[newX, newY].parent\_j = y;

}

}

}

}

}

}

}