UNIVERSITATEA DIN BUCUREŞTI

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licenţă

PATHFINDING IN GAMES

Absolvent
Moraru Mihai-Liviu

Coordonator ştiinţific
Lect. Dr. Dumitran Marius

Bucureşti, septembrie 2023

Pathfinding in Games

Moraru Mihai-Liviu

Rezumat

În lumea curentă a tehnologiei, în care suntem bombardați constant de informații și așteptările pentru viitorii programatori și ingineri sunt în continuă creștere, importanța învățământului de calitate în domeniul informatic nu poate fi subestimată. Deși un mare număr de concepte sunt dificile de explicat pentru un cadru didactic și de înțeles pentru student, foarte multe sunt explicate prin metode rudimentare: desene, schițe, diagrame. Printre conceptele de bază ale informaticii se află graf-urile și algoritmii de parcurgere, fiind noțiuni esențiale pentru rezolvarea unei varietăți de probleme practice.

Această lucrare propune o aplicație ce poate facilita înțelegerea acestor concepte, fiind un suport vizual pentru o multitudine de algoritmi de parcurgere, exemplificând avantajele și dezavantajele fiecăruia și ajutând la ilustrarea conceptelor de nod și graf prin asocierea lor cu elemente clare din aplicație.

Abstract

In the current technological landscape, in which we are constantly bombarded with information and the expectations for future programmers and engineers are constantly increasing, the importance of quality education in the computer science field cannot be understated. Although a large number of concepts are difficult to explain for a teacher while also being understood by the student, many are explained through rudimentary methods: drawing, sketches, diagrams. Among the fundamental concepts in computer science, we find graphs and pathfinding algorithms, essential notions for solving a variety of practical problems.

This paper proposes an application that can facilitate the understanding of these concepts, providing a visual representation for a multitude of pathfinding algorithms, showing the advantages and disadvantages of each and helping illustrate the concepts of node and graph by associating them to actual elements from the application.

Cuprins

1. Introducere

1.1 Motivația și rolul lucrării

Motivația principală pentru această lucrare o reprezintă dificultatea pe care am întâlnit-o pe parcursul studiului în facultate în ceea ce privește asocierea conceptelor teoretice predate la cursurile din cadrul facultății cu utilizări practice și aplicații concrete. Personal, singurul mod în care pot procesa orice concept și definiție teoretică este dacă văd o implementare practică și relevantă ce le aplică. Din fericire, facultatea noastră pune un accent mare pe partea practică a informaticii și acest lucru mi-a facilitat studiul de-a lungul anilor, însă, multe din noțiuni erau explicate prin desene pe tablă și pseudocod, fiind necesar ca un profesor să caute și să direcționeze studenții ca mine către varii site-uri ce le prezintă într-un mod practic pentru a putea să ne facă să le înțelegem.

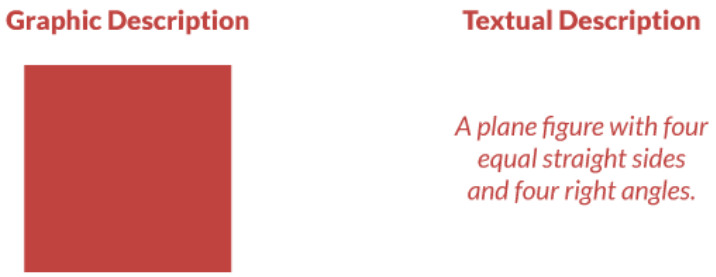


Fig. 1.1 *Exagerarea diferenței dintre descrieri teoretice și practice* [1]

Scopul acestei aplicații este de a prezenta câteva din conceptele de bază ale programării într-o formă ușoară de înțeles și familiară oricui: un labirint. Ideea principală este că anumiți studenți nu ar vedea inițial legătura dintre un graf cu noduri și un labirint, însă printr-o prezentare corectă aceștia ar putea face legătura între cele două concepte și ar putea observa graf-uri în multe alte implementări. De asemenea, aplicația ar ilustra mai mulți algoritmi de parcurgere și ar arăta întru-un mod interactiv diferența dintre ei la nivel de memorie utilizată, complexitate si viteză de execuție.

Speranța mea este că acest proiect poate fi folosit pentru a explica aceste concepte într-un mod mai ușor și poate chiar ar încuraja unii studenți să își creeze propriile implementări cu aceste noțiuni.

1.2 Argumentarea tehnologiilor folosite

Pentru implementarea aplicației pe care se bazează această lucrare am decis să folosesc Unity. Unity este un mediu de development perfect pentru aplicații vizuale, fiind folosit în extrem de multe experiențe interactive, de la jocuri video, la aplicații de simulare și chiar de realitate virtuală (VR). De asemenea, suita Unity este complet gratuită pentru utilizatori individuali și pentru aplicații non-profit [2]. În principal, preferința pentru Unity față de alte game engine-uri se bazează pe accesibilitatea sa cât și pe faptul că acesta are deja implementate o mulțime de funcționalități pentru jocuri și aplicații 2D.

Deoarece limbajul de programare principal pentru scripting în Unity este C#, IDE-ul pe care l-am folosit este Visual Studio.

1.3 Structura lucrării

Lucrarea în sine constă din 5 capitole, dacă excludem introducerea: Capitolul 2: Tehnologii, conține detalii legate de tehnologiile folosite de Unity și funcționalitățile ce au ajutat la crearea proiectului. Capitolul 3: Pathfinding Algorithms, este capitolul ce conține descrierea în detaliu a conceptelor și algoritmilor utilizați la nivel teoretic. Capitolul 4: Descrierea aplicației, prezintă implementarea, funcționalitățile aplicației cât și modul de utilizare. Capitolul 5: Comparații între algoritmi, constă într-o analiză generală a diferenței dintre algoritmi, rezultată în urma utilizării programului. Capitolul 6: Concluzie, reprezintă finalul lucrării și oferă o compilare a tuturor ideilor prezentate in lucrare și posibile viitoare funcționalități și dezvoltări ale acesteia.

2. Tehnologii

2.1 Unity Editor

Principalul sistem ce stă la baza unei aplicații în Unity o reprezintă abordarea pe scene pentru a crea un mediu interactiv. O scenă este o colecție ce conține o multitudine de elemente, de la camere, GameObject-uri, sisteme de iluminare și altele. Fiecare scenă marchează o interfață a aplicației, fiind folosite pentru a reprezenta diferite meniuri sau nivele dintr-un joc și pentru a permite tranziția cu ușurință între ele. Fiecare scenă este organizată sub formă de ierarhie, fiind posibil ca anumite obiecte să aibă alte obiecte subordonate, formând relații relevante pentru diferite transformări din scenă. [3]

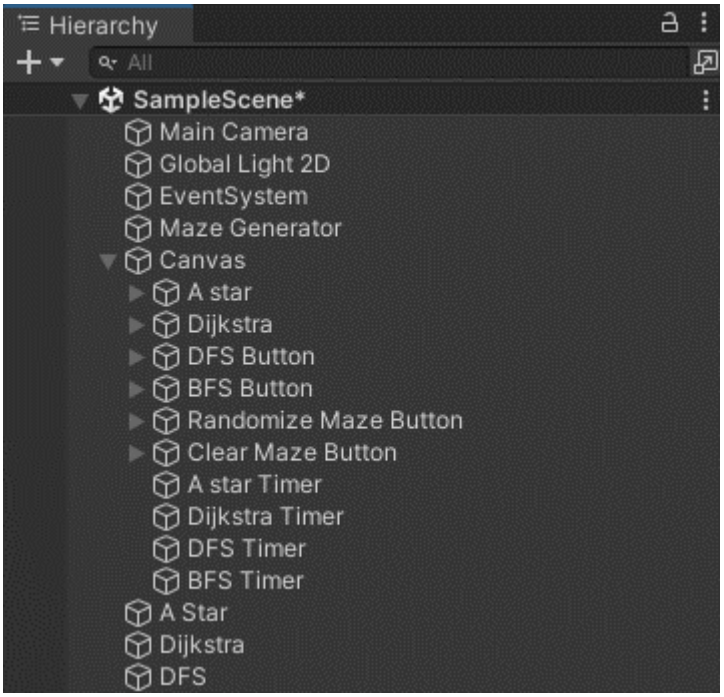


Fig. 2.1 *Exemplu de ierarhie într-o scenă*

Elementele fundamentale într-o scenă sunt GameObject-urile, structuri de bază cărora li se atribuie proprietăți. Toate obiectele dintr-o scenă sunt GameObject-uri, funcționalitatea și aspectul lor fiind determinate de componentele atașate. Putem considera un GameObject ca fiind o cutie care, atunci când este umplută cu diferite componente, își schimbă forma și rolul în aplicație. O componentă reprezintă o serie de funcții ce influențează comportamentul și proprietățile unui obiect din scenă. Componentele pot fi orice, de la funcționalități integrate în Unity precum poziția, materialul, culoarea sau chiar setările de rendering, la un script personalizat ce conferă mai multă libertate în modificarea obiectului. [4]

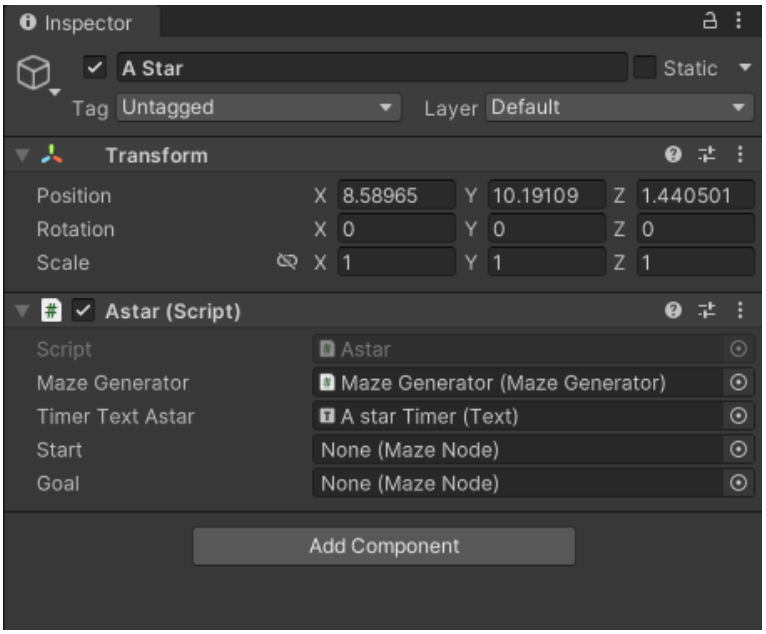


Fig. 2.2 *Un GameObject cu un script atașat vizualizat în Inspector*

Datorită structurii de tip ierarhie, Unity permite crearea de grupuri de GameObject-uri numite Prefab. Aceste pachete facilitează manipularea proprietăților tuturor elementelor subordonate și pot fi reprezentate ca structuri de date în codul unui script. De asemenea, acestea pot fi multiplicate cu ușurință, iar orice schimbare la nivelul Prefab-ului inițial se va propaga la toate celelalte copii ale sale.

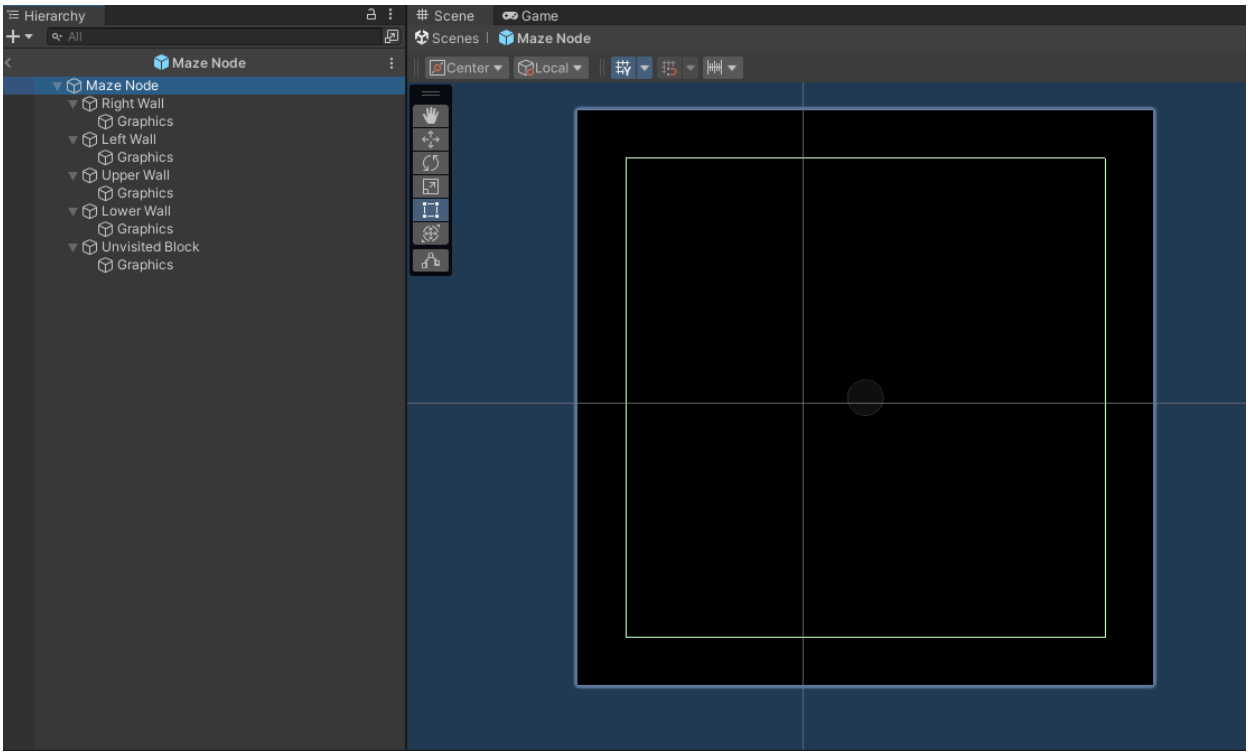


Fig. 2.3 *Exemplu de Prefab si componentele sale*

2.2 Scripting

Metoda cea mai folositoare pentru a adăuga funcționalitate obiectelor dintr-o scenă o reprezintă scripting-ul. Scripting-ul în contextul dezvoltării unei aplicații în Unity reprezintă scrierea de cod în fișiere numite Script-uri. Limbajul de programare principal folosit în programarea în Unity este C#, un limbaj modern, orientat obiect, creat de Microsoft. C# a fost creat cu intenția de a fi ușor de folosit, cu accent pe productivitate și crearea de software cu durată de viață lungă, fiind specific realizat pentru a permite portabilitatea cu ușurință între diferite platforme. [5] Această trăsătură a limbajului permite aplicațiilor create în Unity să fie transferate pe o multitudine de platforme, de la sisteme de operare pentru computere personale(Windows, MacOS), la console de jocuri (PlayStation, Xbox), platforme mobile (Android, iOS) și chiar direct în browsere prin WebGL. [6] Datorită accentului pe programarea în C#, framework-ul de bază în Unity îl reprezintă .NET, o platformă open-source ce găzduiește o gamă largă de librării.

În urmă creării unui script, acesta moștenește în mod implicit clasa MonoBehaviour ce conține un număr mare de utilități și permite atașarea scriptului la GameObject-uri. Una dintre funcționalitățile importante din această clasă implicită o reprezintă posibilitatea de a folosi Coroutine, metode ce permit scrierea de funcții asincron, de a întârzia și a organiza ordinea de execuție pentru diferite bucăți de cod. De asemenea, MonoBehaviour conține metode ce sunt capabile să execute cod pe baza activității curente din proiect. La crearea unui script nou, două din aceste metode sunt instanțiate: Start(), care este apelată la crearea obiectului și Update(), care este apelată la fiecare schimbare de frame (bazat pe FPS - frames per second). Alte metode relevante pentru proiect sunt StartCoroutine și StopCoroutine care, precum se poate deduce din numele lor, facilitează în executarea Coroutine-lor.

Precum a fost precizat mai devreme, putem declara un GameObject în interiorul script-ului pentru a crea o referință la acesta. Modul în care se stabilește legătura dintre câmpul declarat și obiect în sine este cu ajutorul Inspectorului din Unity. Dacă declararea obiectului este publică, sau privată și serializată prin adăugarea atributului "[SerializeField]", acesta va apărea în inspector sub componenta de script, de unde atribuirea se poate face printr-un simplu drag and drop.

```
Unity Script (1 asset reference) | 99+ references
public class MazeNode : MonoBehaviour
{
    [SerializeField]
    private GameObject _leftWall;
    6 references
    public bool hasLeftWall { get; private set; } = true;

    [SerializeField]
    private GameObject _rightWall;
    6 references
    public bool hasRightWall { get; private set; } = true;

    [SerializeField]
    private GameObject _upperWall;
    6 references
    public bool hasUpperWall { get; private set; } = true;

    [SerializeField]
    private GameObject _lowerWall;
    6 references
    public bool hasLowerWall { get; private set; } = true;

    [SerializeField]
    private GameObject _unvisitedBlock;
```

Fig. 2.3.2 *Definirea elementelor unui Prefab într-un script*

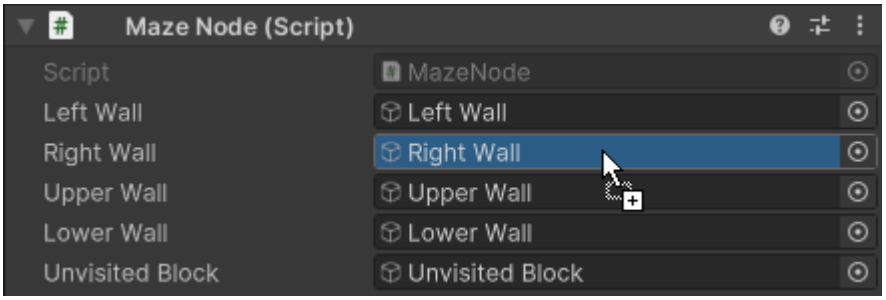


Fig. 2.3.1 *Atribuirea elementelor în Inspector*

Diferența principală dintre cele două metode o reprezintă faptul că un câmp public poate fi accesat de orice alt script, în timp ce unul privat, chiar dacă este serializat, poate fi accesat doar în același script, chiar dacă ambele modalități permit vizualizarea în Inspector. Există și un atribut numit “[HideInInspector]” care permite unui câmp public să nu poată fi accesat de Inspector, păstrându-și accesibilitatea pentru alte script-uri. Un câmp privat simplu nu poate fi accesat de Inspector.

3. Pathfinding Algorithms

3.1. Graf

Definirea concretă a conceptului de graf este dificilă datorită numărului mare de tipuri de grafuri folosite pentru exprimarea și rezolvarea a varii probleme în care sunt necesare. În principal mă voi axa pe definițiile prezentate în referința [7].

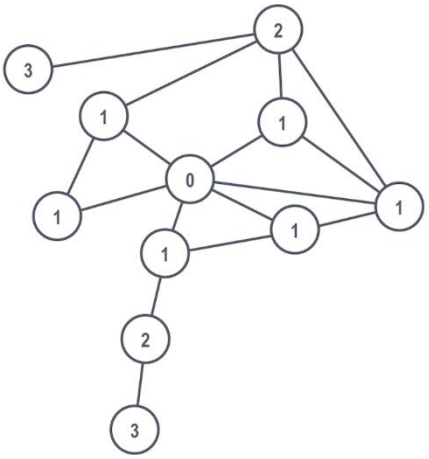


Fig. 3.1 *Exemplu de graf simplu neorientat* [8]

Un graf simplu neorientat este o structură de date fundamentală în informatică, fiind utilizată pentru a ilustra relații între diferite entități. Acesta este format din noduri(numite și vârfuri sau puncte) și muchii (sau linii). Pentru o definiție matematică:

Def. 1: Considerăm un graf simplu neorientat G ca fiind o pereche $G = (N, M)$ unde:

- N este o mulțime denumită nodurile lui G;

- $M \subseteq \{\{x, y\} | x, y \in N \text{ și } x \neq y\}$ o mulțime denumită muchiile lui G și reprezintă perechi neordonate de noduri.

Această definiție permite o singură conexiune între oricare două noduri și, din acest motiv, dacă permitem mai multe muchii între aceleași noduri obținem definiția:

Def. 2: Un multigraf neorientat G este definit ca fiind $G = (N, M, \Phi)$ unde:

- N este o mulțime denumită nodurile lui G;

- M este o mulțime denumită muchiile lui G;

- $\Phi: M \rightarrow \{\{x, y\} | x, y \in N \text{ și } x \neq y\}$ este o funcție de incidență care atribuie fiecărei muchii o pereche neordonată de noduri.

O observație importantă este că definițiile anterioare nu permit existența unei muchii între un nod și el însuși. Existența ciclurilor poate fi permisă în ambele definiții dacă eliminăm condiția că x și y să nu coincidă. Deja am întâlnit o primă ambiguitate și nici nu am ajuns la definirea muchiilor orientate și a grafurilor orientate ca rezultat. Putem, bine înțeles, să specificam dacă un graf definit anterior permite cicluri pentru a evita ambiguitatea.

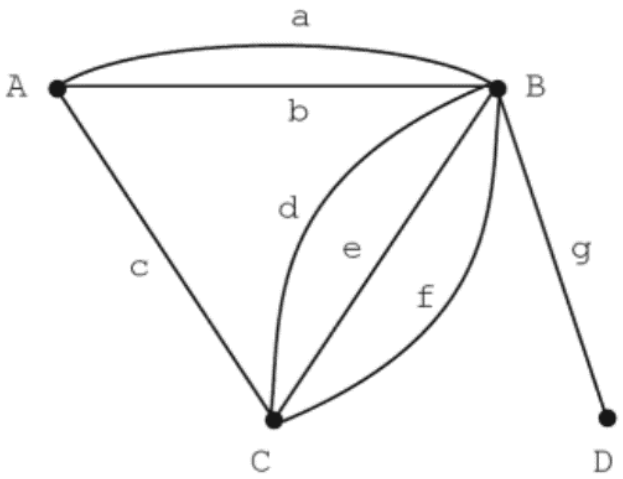


Fig. 3.2 *Exemplu de multigraf neorientat* [7]

Un graf orientat simplu are definiția asemănătoare cu cea a grafului simplu neorientat, însă se adaugă proprietatea că toate muchile au asociată o ordine între noduri care denotă o anumită orientare a muchilor. Ca rezultat, definiția matematică este:

Def. 3: Un graf orientat simplu G este reprezentat ca o pereche $G = (N, M)$ unde:

- N este o mulțime denumită nodurile lui G;

- $M \subseteq \{(x, y) | (x, y) \in N^2 \text{ și } x \neq y\}$ este o mulțime denumită muchiile lui G și reprezintă perechi ordonate de noduri.

Iar definiția unui multigraf orientat este:

Def. 4: Un multigraf orientat G este definit ca fiind $G = (N, M, \Phi)$ unde:

- N este o mulțime denumită nodurile lui G;

- M este o mulțime denumită muchiile lui G;

- $\Phi: M \rightarrow \{(x, y) | (x, y) \in N^2 \text{ și } x \neq y\}$ este o funcție de incidență care atribuie fiecărei muchii o pereche ordonată de noduri.

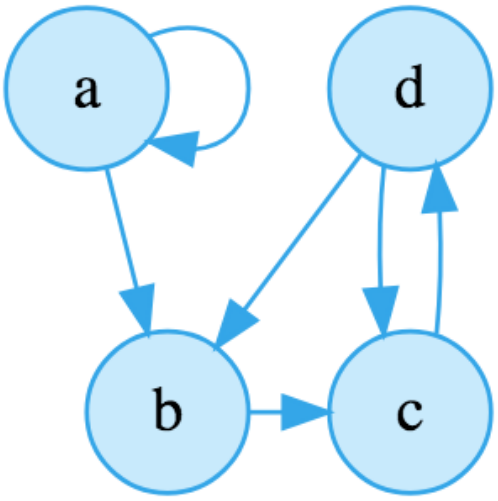


Fig. 3.3 *Exemplu de multigraf orientat cu cicluri* [9]

La aceste definiții se adaugă și grafurilor orientate posibilitatea existenței ciclurilor, în cazul în care s-ar elimina condiția $x \neq y$. De asemenea, definițiile anterioare se pot extinde dacă luăm în considerare adăugarea unui cost fiecărei muchii, definind astfel existența grafurilor ponderate, care ar necesita adăugarea în definiție a unei funcții de cost care atribuie o valoare fiecărei muchii.

În total ar fi necesare 16 definiții pentru a putea cuprinde toate formele unui graf, de la simplu la multigraf, la orientat și neorientat, la ponderat și neponderat, la posibilitatea că permite cicluri sau nu. Vreau să accentuez cât de complex și greu de definit este acest concept de bază și banal din informatică. Sper că acest fapt pune în evidență importanța explicării acestor principii într-un mod cât mai clar și simplu de înțeles, care să permită analiză personală a posibilelor ambiguități.

3.2 Breadth-First Search

Algoritmul BFS (Abreviat de la Breadth-First Search) reprezintă un algoritm de parcurgere utilizat pentru căutarea și explorarea unei rețele de tip graf sau arbore. Acesta se bazează pe explorarea vecinilor pe nivele, trecând la următorul nivel numai după ce a vizitat toți vecinii cei mai apropiați. Acesta este folosit în principal pentru a găsi distanță minimă între două noduri dintr-un graf.

Exemplu implementare algoritm:

- Alege nodul de start
- Creează o coadă și stochează nodurile ce vor fi vizitate
- Creează o structură de date care să stocheze nodurile deja vizitate
- Marchează nodul de start ca vizitat

- Cât timp coada nu este goală:
  - Scoate un nod din coadă (va fi nodul curent)
  - Procesează nodul curent
  - Pentru fiecare vecin al nodului curent:
    - Dacă vecinul nu a fost vizitat:
      - Marchează vecinul ca vizitat
      - Marchează nodul curent ca părinte al vecinului
      - Adaugă vecinul în coadă

Programul termină execuția dacă se golește întreaga coadă (au fost parcurse toate nodurile accesibile din nodul sursă)

BFS este de obicei implementat folosind o coadă deoarece acest lucru garantează faptul că nodurile sunt procesate în ordinea în care au fost adăugate. Algoritmul este garantat să găsească cel mai scurt drum între două noduri dintr-un graf neponderat. În cazul căutării celui mai scurt drum, procesarea nodului ar presupune verificarea dacă este nodul final și terminarea execuției programului în acest caz, iar drumul minim ar fi dat de ordinea părinților de la nodul scop la nodul de start.

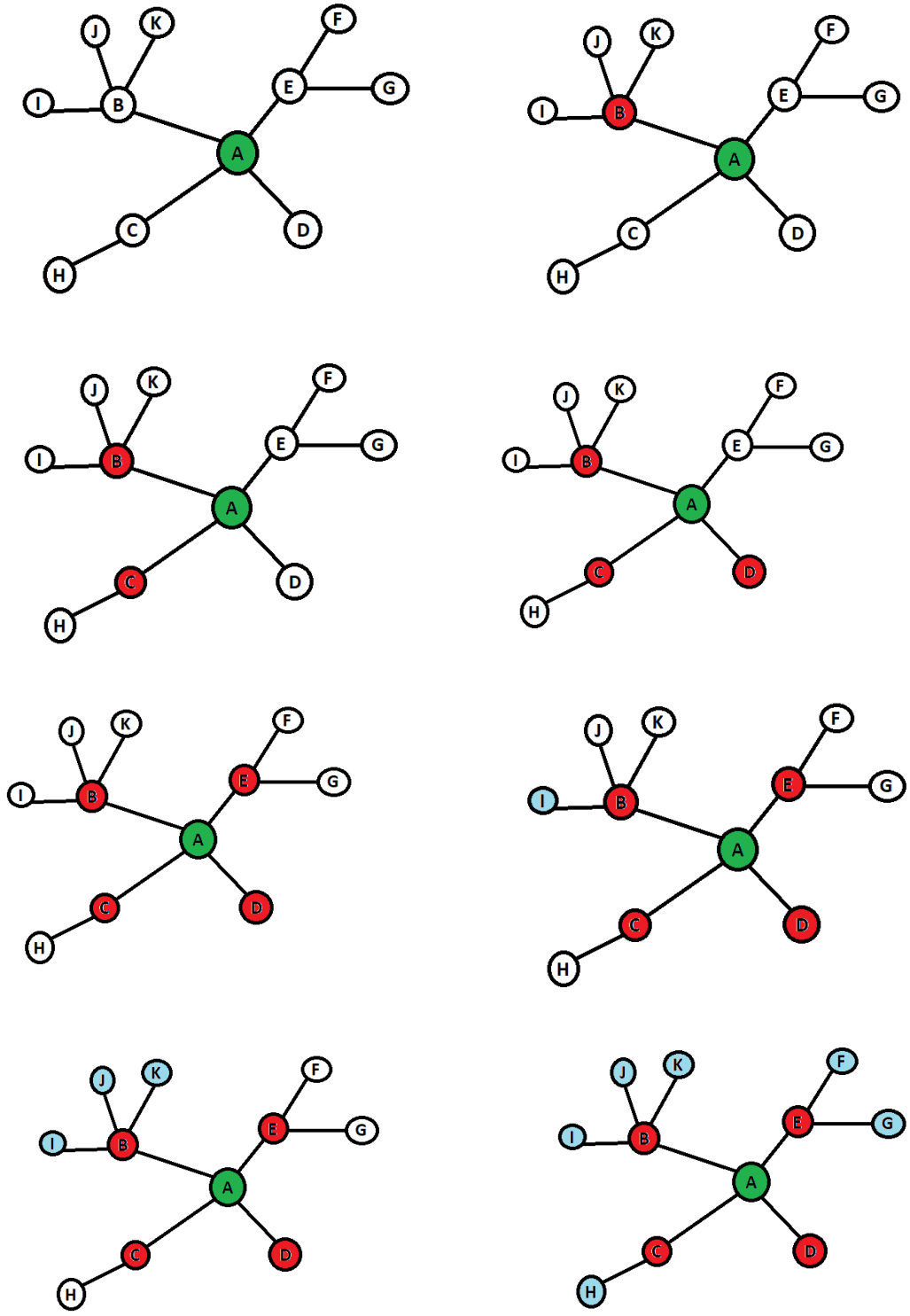


Fig. 3.4 *Exemplu de parcurgere BFS (ordinea de parcurgere ABCDEIJKHFG)*

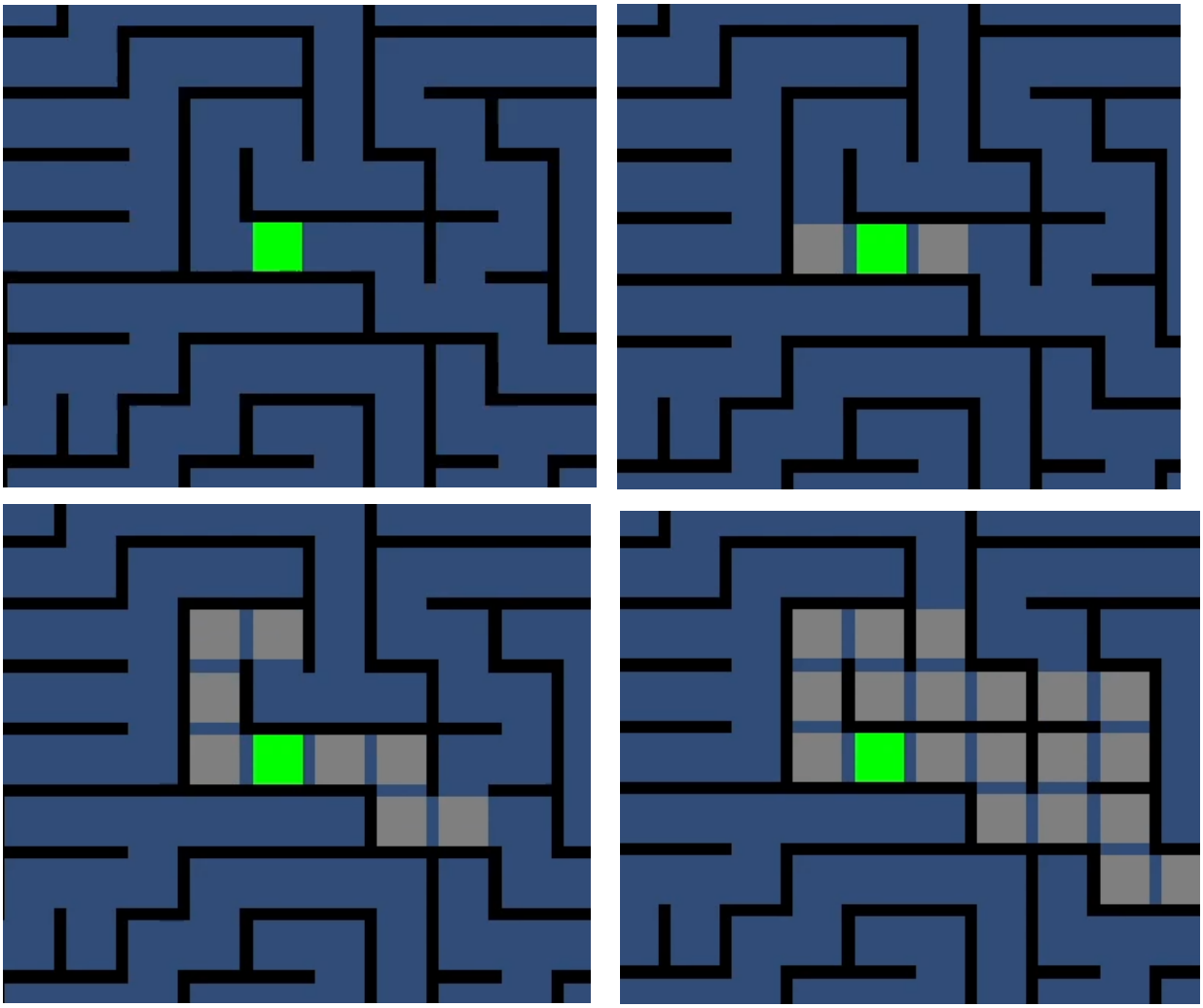


Fig. 3.5 *Comportamentul algoritmului BFS într-un labirint. Se observă faptul că algoritmul vizitează în același ritm toate direcțiile posibile.*

Complexitatea timp a algoritmului BFS este $O(n+m)$, unde n reprezintă numărul de noduri și m numărul de muchii, fiind necesar ca algoritmul să viziteze fiecare nod o singură dată și explorând vecinii determinați de muchii. Complexitatea spațiu este determinată de numărul de noduri maxim din coadă, fiind $O(n)$ în cel mai rău caz, în care întregul graf se află în coadă.

3.3 Depth-First Search

Algoritmul DFS(Depth-First Search) reprezintă, asemenea lui BFS, un algoritm de parcurgere utilizat pentru căutarea și explorarea unei rețele de tip graf sau arbore. Spre deosebire de BFS, care explorează nodurile pe nivele, DFS explorează un singur nod per nivel până când ajunge la un nod fără noi vecini și apoi face backtrack.

Exemplu implementare algoritm:

- Alege nodul de start
- Creează o stivă și stochează nodurile ce vor fi vizitate
- Creează o structură de date care să stocheze nodurile deja vizitate
- Marchează nodul de start ca vizitat și adaugă-l în stivă

- Cât timp stiva nu este goală:
    - Scoate un nod din stivă (va fi nodul curent)
    - Procesează nodul curent
    - Pentru fiecare vecin al nodului curent:
        - Dacă vecinul nu a fost vizitat:
            - Marchează vecinul ca vizitat
            - Marchează nodul curent ca părinte al vecinului
            - Adaugă vecinul în stivă

Programul finalizează execuția când stiva este goală (au fost parcurse toate nodurile accesibile din nodul sursă).

DFS, spre deosebire de BFS, nu garantează drumul cel mai scurt într-un graf neponderat deoarece acesta explorează mai întâi în adâncime, putând omite conexiuni directe între anumite noduri într-un multigraf. DFS poate determina drumul cel mai scurt într-un graf simplu neponderat.

Asemănător cu BFS, complexitatea timp este determinată de numărul de noduri și muchii, fiind $O(n + m)$. Complexitatea spațiu este determinată de numărul de nivele ale grafului, în cel mai rău caz (într-un graf ce reprezintă un lanț de noduri) aceasta ar fi $O(n)$.

În general, cei doi algoritmi sunt apropiați în ceea ce privește performanța în găsirea celui mai scurt drum între două noduri într-un graf simplu neponderat, BFS fiind mai rapid în cazuri în care nodul scop este aproape de nodul de start, iar DFS este avantajat dacă nodul scop este pe o ramură adâncă a grafului.

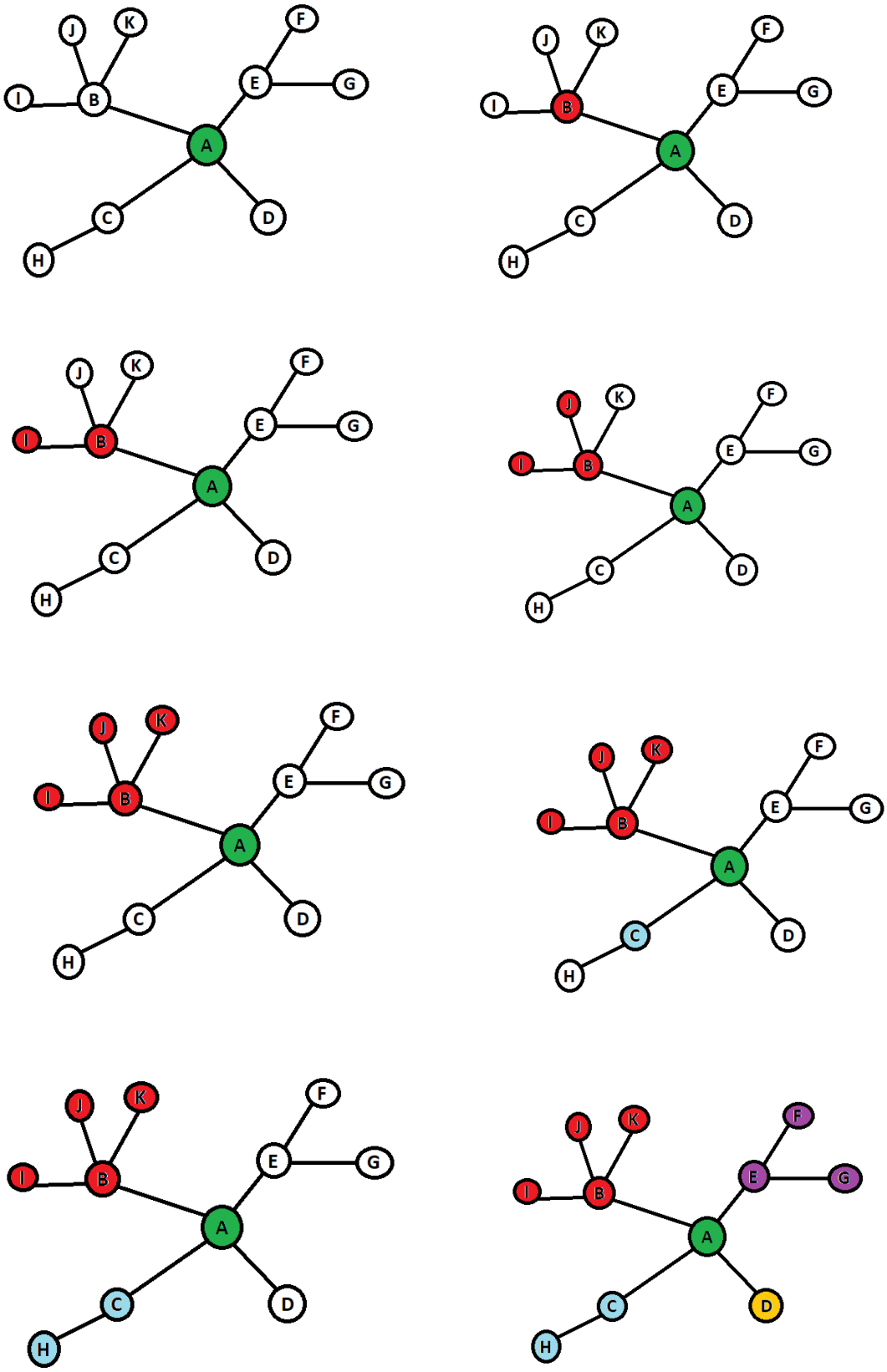


Fig. 3.6 *Exemplu de parcurgere DFS (ordinea de parcurgere ABIJKCHDEFG)*

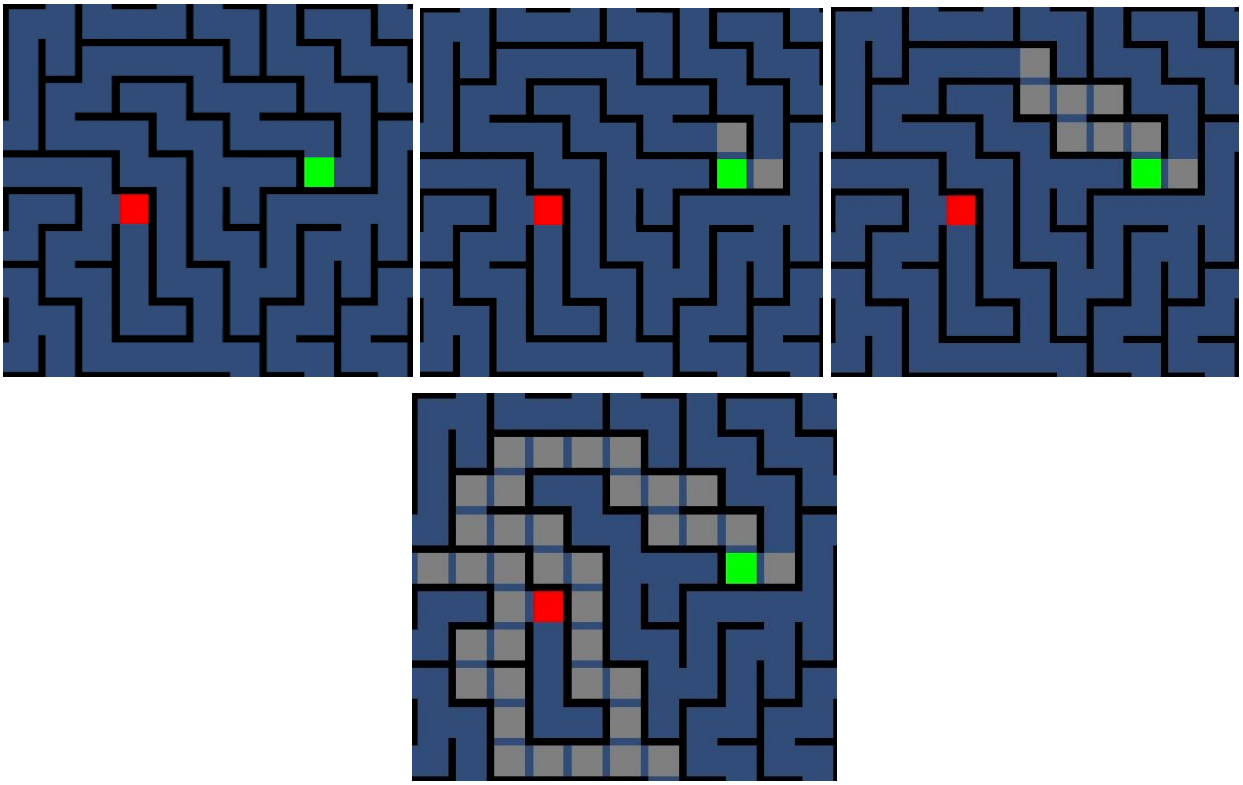


Fig. 3.7 *Comportamentul algoritmului DFS într-un labirint. Se observă tendința algoritmului de a vizita toate nodurile dintr-o ramură până la capăt înainte de a schimba ramura. Într-un labirint, acest lucru poate duce la posibilitatea nefericită a ignorării nodului final chiar și când se învecinează cu un nod deja vizitat.*

3.4 Algoritmul lui Dijkstra

Algoritmul lui Dijkstra reprezintă un algoritm de parcurgere utilizat pentru găsirea distanței minime între noduri într-un graf ponderat. Acesta funcționează prin atribuirea distanței minime curente fiecărui nod și actualizarea acestei distanțe dacă se găsește un drum mai scurt. Se diferențiază de algoritmii menționați până acum prin posibilitatea de a determina distanță minimă de la un nod sursă la toate celelalte noduri din graf.

Exemplu implementare algoritm:

- Alege nodul de start
- Creează un set de noduri nevizitate
- Creează un priority queue (min-heap) pentru a stoca noduri în funcție de distanță intermediară
- Atribuie distanța intermediară 0 nodului de start și infinit(un număr foarte mare) celorlalte noduri

- Cât timp există noduri nevizitate în priority queue:
    - Elimină nodul cu cea mai mică distanță intermediară din queue (va fi nodul curent)
    - Procesează nodul curent
    - Pentru fiecare vecin al nodului curent:
        - Calculează distanța intermediară a vecinului (suma dintre distanță nodului curent și valoarea muchiei dintre acesta și vecin)
        - Dacă distanța calculată este mai mică decât distanța intermediară a vecinului:
            -Actualizează distanța în queue

Algoritmul finalizează execuția când nodul scop este vizitat sau când priority queue-ul este gol, indicând că toate nodurile au fost vizitate.

Deși nu este prezentă în această implementare, se poate modifica algoritmul lui Dijkstra pentru a obține și cel mai scurt drum între noduri, nu doar distanță minimă între ele. De asemenea, deși Dijkstra a fost conceput pentru a fi utilizat în grafuri ponderate, putem să îl folosim în grafuri neponderate dacă atribuim un cost 1 pentru fiecare muchie.

Complexitatea timp a algoritmului lui Dijkstra este determinată de structura de date folosită pentru priority queue, folosirea unui heap binar sau a unui heap Fibonacci pentru extracția nodului cu cea mai mică distanță are o complexitate de O(log n). Recursivitatea principală iterează prin toate nodurile și relaxează toate muchiile o dată, ceea ce ne dă o complexitate de $O(m)$. Astfel, complexitatea timp va fi $O((n+m) * log\ n)$ pentru heap binar și $O(n + m * log\ n)$ pentru heap Fibonnaci. [10]

Complexitatea spațiu depinde și ea de structura de date folosită, fiind de obicei între $O(n)$ și $O(n+m)$ în funcție de implementare. În general, heap-urile Fibbonaci oferă complexități de timp mai bune, dar necesită spațiu mai mult, în timp ce heap-urile binare nu necesită la fel de mult spațiu și sunt mai încete.

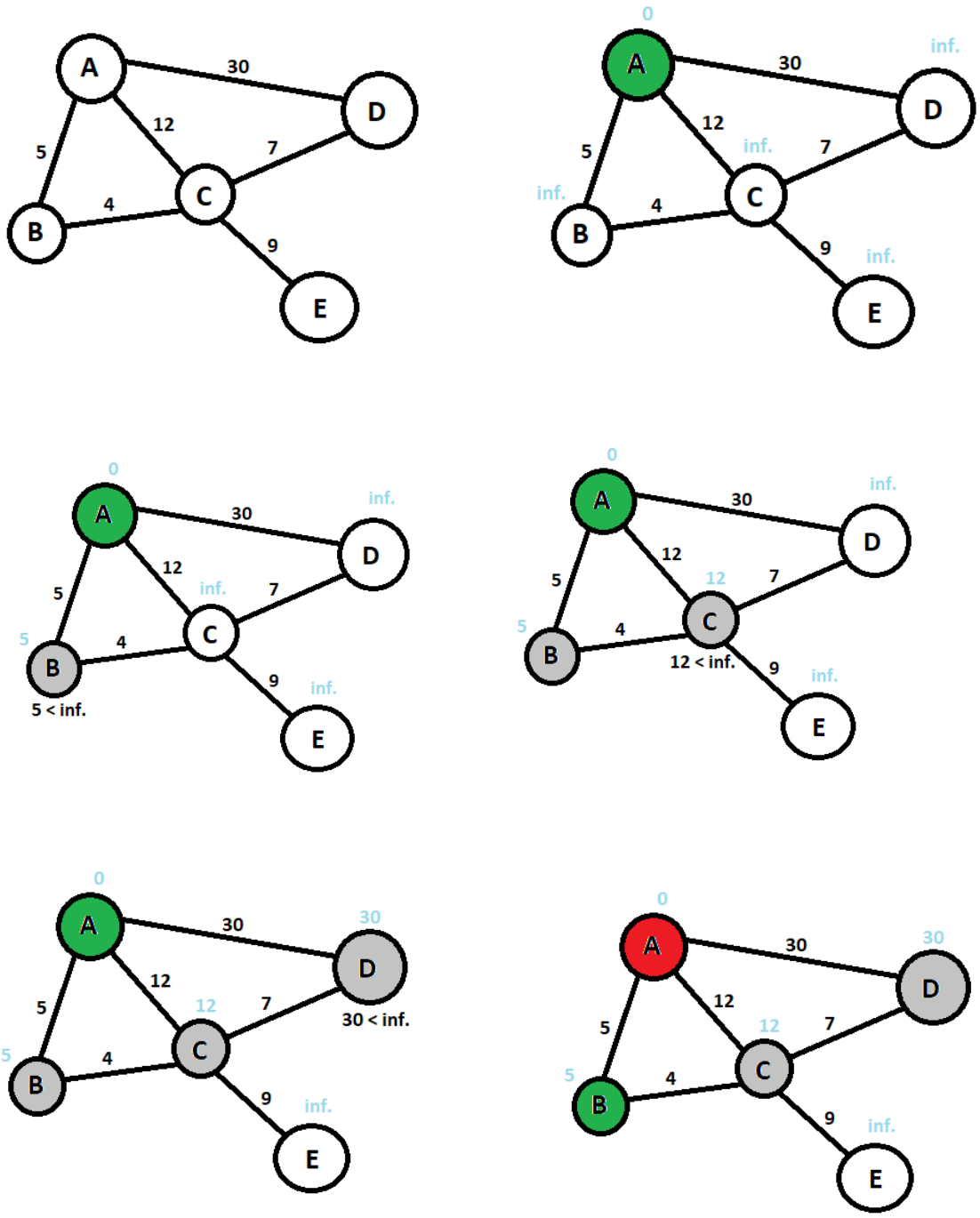


Fig. 3.8.1 *Exemplu de parcurgere Dijkstra (Partea 1). Alegem nodul A ca nod de start, marcăm costurile 0 în start și infinit în restul. Verificăm dacă folosirea muchiilor de la A la toți vecinii săi ar rezulta într-un cost mai mic decât cel curent (logic, orice număr real < infinit). După, marcăm nodul A ca fiind complet utilizat și considerăm următorul nod ca fiind principal.*

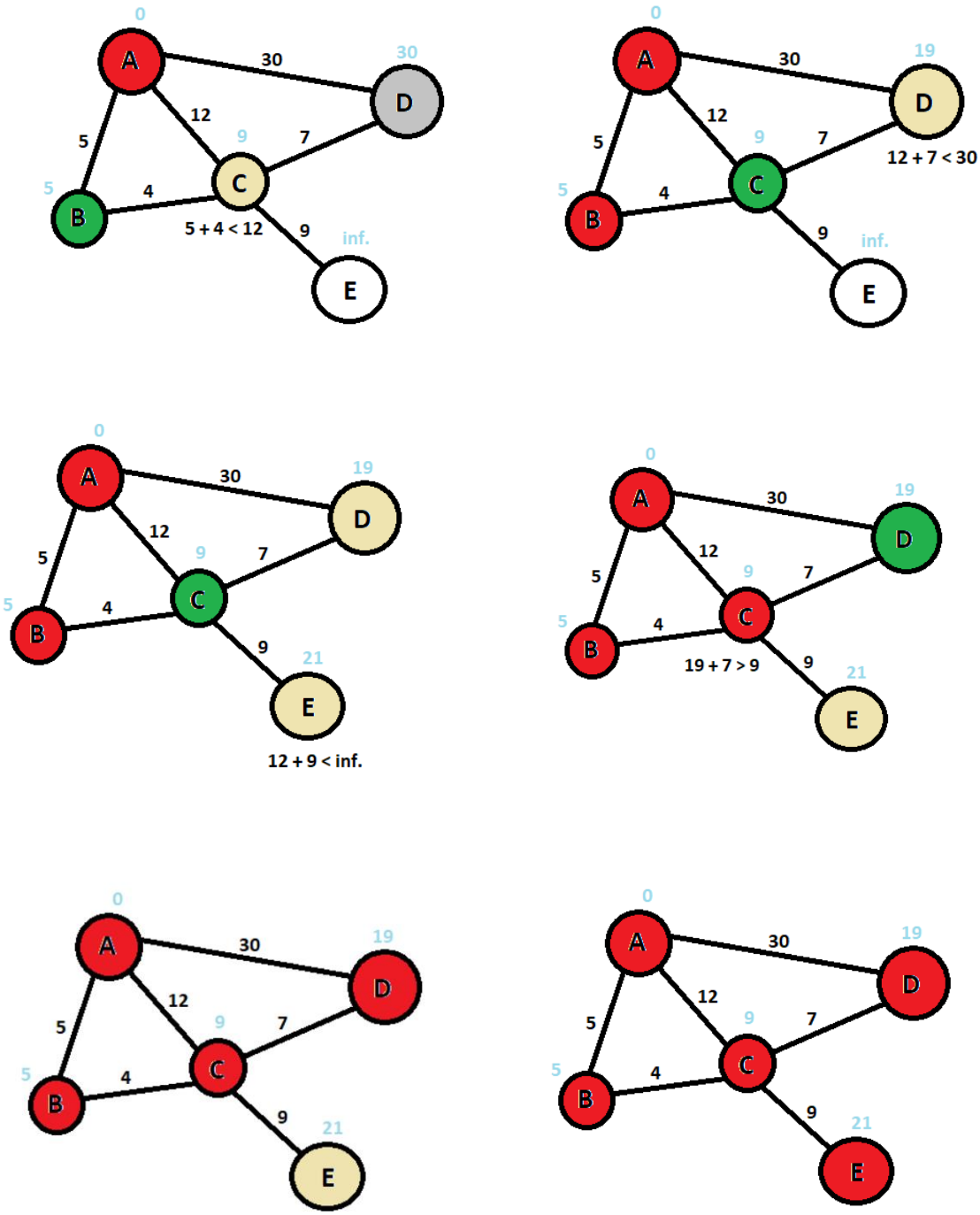


Fig. 3.8.2 *Exemplu de parcurgere Dijkstra (Partea 2). Verificăm pentru fiecare vecin al noului nod principal dacă muchia dintre ei ar duce la un cost mai mic și continuăm procesul până când toate muchiile dintre toate nodurile au fost evaluate. Valorile notate cu albastru deschis reprezintă distanța minimă de la A la respectivul nod.*

Ilustrarea procedeului utilizat de algoritmul lui Dijkstra este extrem de dificilă doar prin imagini și reprezintă încă o motivație pentru crearea unor implementări descriptive a acestuia.

3.5 A*

Algoritmul A* este un algoritm de parcurgere ce este utilizat pentru găsirea celui mai scurt drum între două noduri într-un graf ponderat. Acesta reprezintă o optimizare a algoritmului lui Dijkstra ce utilizează euristici pentru a influența algoritmul să se apropie de nodul scop la fiecare pas.

Exemplu implementare algoritm:

- Alege nodul de start
- Creează o listă (de obicei un priority queue) care să stocheze nodurile de evaluat
- Creează o listă pentru a stoca nodurile deja evaluate
- Atribuie un cost (notat "g") de 0 nodului de start
- Calculează un cost estimativ de la nodul de start la nodul scop

- Cât timp lista de evaluat nu e goală:
    - Elimina nodul cu cea mai mică valoare f = g + h din lista de evaluat (va fi nodul curent)
    - Dacă nodul curent este nodul scop, algoritmul finalizează execuția

    - Pentru fiecare vecin al nodului curent:
        - Calculează valoarea intermediară a lui g adăugând costul muchiei de la nodul curent la vecin
        - Dacă vecinul este în lista de noduri deja evaluate și valoarea intermediară a lui g este mai mare, treci peste
        - Dacă vecinul nu e în lista de noduri deja evaluate sau valoarea intermediară a lui g este mai mică, actualizează valoarea lui g și adaugă vecinul în lista de evaluat
        - Calculează valoarea euristică h de la vecin la scop

Algoritmul termină execuția dacă se ajunge la nodul scop sau dacă lista de evaluat devine goală, ceea ce înseamnă că nu există un drum valid.

Algoritmul A* folosește atât costul drumului "g" cât și un cost estimativ "h" care este obținut printr-o funcție euristică admisibilă și consistentă în algoritm și care permite prioritizarea nodurilor ce se apropie de nodul scop, spre deosebire de algoritmul lui Dijkstra care explorează toate nodurile în ordine până găsește nodul scop.

O euristică este admisibilă dacă valoarea lui h este mai mică sau egală cu costul real dintre nodul curent și nodul scop și consistentă dacă este satisfăcută ecuația $h(n) \leq c(n, n') + h(n')$ unde n' este un vecin al nodului n și $c(n, n')$ reprezintă costul muchiei dintre cele două noduri.

Complexitatea in timp a algoritmului A* depinde foarte mult de euristica utilizată, în cel mai bun caz fiind aproape de complexitatea algoritmului lui Dijkstra, însă complexitatea în spațiu va fi destul de des mai bună decât pentru Dijkstra (va tinde să nu exploreze noduri irelevante pentru a ajunge la nodul scop).

Una dintre euristicile importante care este implementată și în proiect o reprezintă distanță Manhattan, o sumă a distanțelor verticale și orizontale dinte nodul curent și nodul scop. Într-o configurație de tip grid, aceasta este admisibilă și consistentă, presupunând că deplasarea nu se poate face pe diagonală, ci doar în cele patru direcții cardinale.

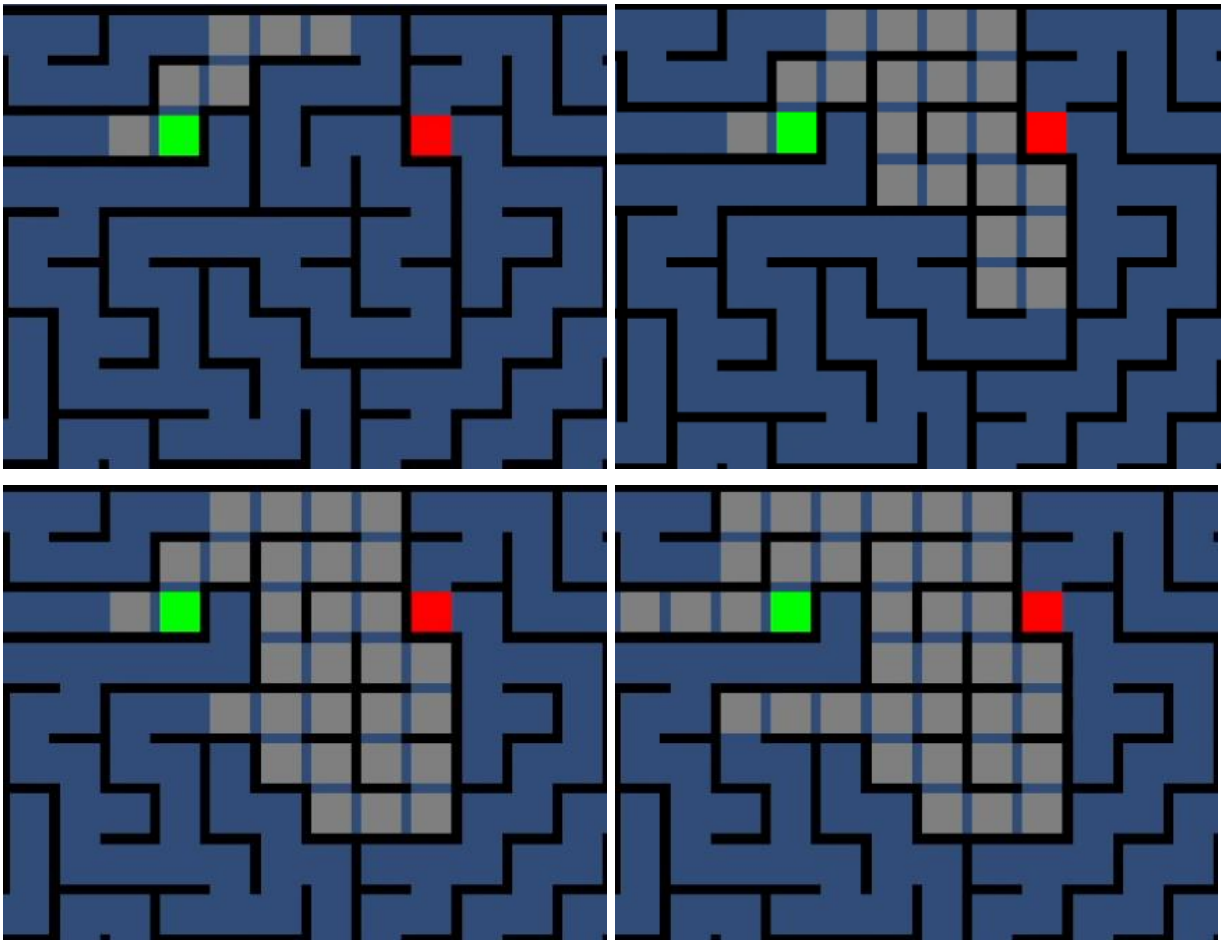


Fig. 3.9.1 *Traversarea labirintului cu A*. Folosind distanța Manhattan pentru euristică, algoritmul știe mereu unde se află nodul curent față de nodul final și explorează prioritar nodurile ce se apropie de el. În imaginea a patra se observă cum algoritmul începe să exploreze în sensul opus celui pe care l-a explorat până atunci deoarece drumul corect se îndepărtează de final și astfel nodurile din drumul greșit sunt mai aproape de nodul final.*

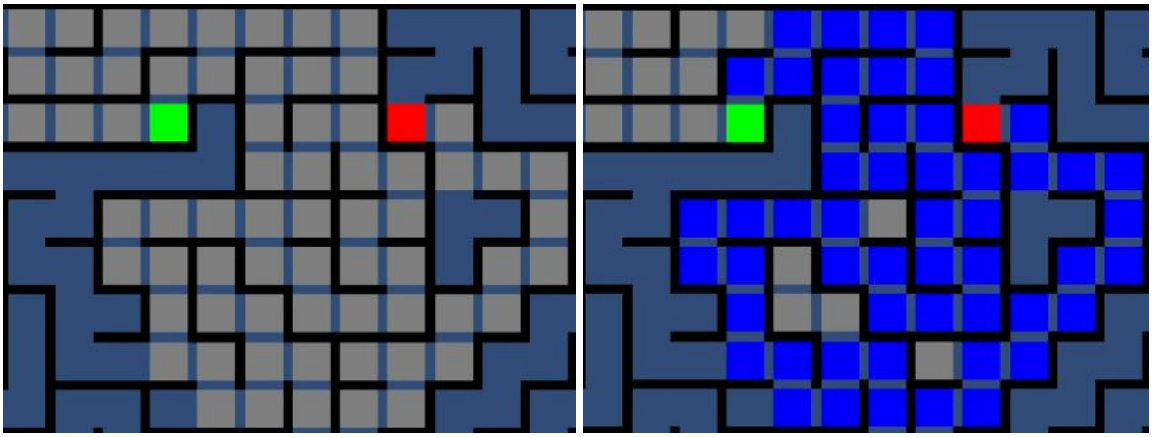


Fig. 3.9.2 *Finalul traversării labirintului cu A* și drumul rezultat.*

Euristicile tind să fie greu de exprimat în termeni generali deoarece acestea sunt definite în funcție de datele și scopul în care se utilizează A*. O euristică definită corect folosește cât mai multe din informațiile aplicației pentru a limita numărul de noduri explorate cât mai mult cu putință. Câteva exemple de informații și constrângeri ce sunt luate în considerare pentru crearea de euristici în aplicații reale sunt:

- **Condițiile de trafic** în aplicații de navigare. Acestea permit crearea unor euristici capabile să estimeze cea mai rapidă rută în funcție de diferite evenimente ce pot apărea pe drum (accidente, construcții, ambuteiaje). O euristică bine definită poate determina, cu un grad de precizie ridicat, cât de mult ar influența timpul de deplasare un eveniment în trafic și ar facilita la recomandarea unor rute alternative.
- **Variațiile de teren și altitudine** în implementări din robotică. O euristică poate ajuta la atribuirea unui cost concret pentru fiecare tip de teren, de la asfalt, pietriș, pământ etc., la diferențe de altitudine, unghiul unei pante și alte detalii relevante pentru un robot ce ar trebui, de exemplu, să se deplaseze între două locații și să minimizeze energia consumată pe traseul său. Bineînțeles că drumurile asfaltate și plate vor fi cele mai eficiente, dar posibilitatea de a decide în timp real ajustarea traiectoriei bazată pe informații obținute de senzori ar fi implementată foarte calitativ folosind o euristică potrivită. Acest tip de sistem ar putea fi implementat și în anumite jocuri video, pentru a permite unor inamici să traverseze un mediu divers pentru a ajunge la jucător într-un mod optim sau realist.
- **Condițiile de conexiune** în direcționarea pachetelor de date într-o rețea. Într-o rețea de calculatoare, factori precum bandwidth-ul, latency-ul și posibilele fluctuații în calitatea conexiunii pot influența negativ transmiterea de pachete. Crearea unei euristici potrivite pentru a clasifica și ajusta dinamic parametrii transferului de date permite menținerea unei comunicari fluide între calculatoare și servere.

A* este un algoritm extrem de folositor, cu aplicații într-o varietate de probleme și poate fi optimizat extrem de mult prin modificarea euristicilor ce determină dacă algoritmul se apropie sau se depărtează de scop. Acesta reprezintă modul principal de rezolvare a problemelor de Pathfinding în jocuri video, programarea sistemelor de navigare GPS pentru a găsi rute eficiente, în robotică pentru deplasarea și evitarea obstacolelor și chiar în diferite simulări, algoritmul putând fi utilizat pentru a simula modul cum se deplasează un grup de oameni într-un mediu îngust.

4. Descrierea aplicației

4.1 Conceptul programului

Ideea inițială pentru program a fost de a crea un labirint sub formă de matrice, cu fiecare element fiind ori un drum ori un zid, elementele de tip drum reprezentând noduri dintr-un graf simplu neponderat. In acest concept inițial se puteau atribui noduri de start si scop, iar în urma rulării programului s-ar încerca găsirea drumului cel mai scurt dintre cele două noduri folosind diferiți algoritmi. Deoarece labirintul are un singur drum corect , toți algoritmii utilizați ar obține același drum final, dar îmi doream în principal să se ilustreze logica parcurgerii din spatele fiecăruia. Din acest motiv, am dorit, de asemenea, să fie marcate toate celulele vizitate în urma rulării fiecăruia dintre algoritmii de parcurgere.

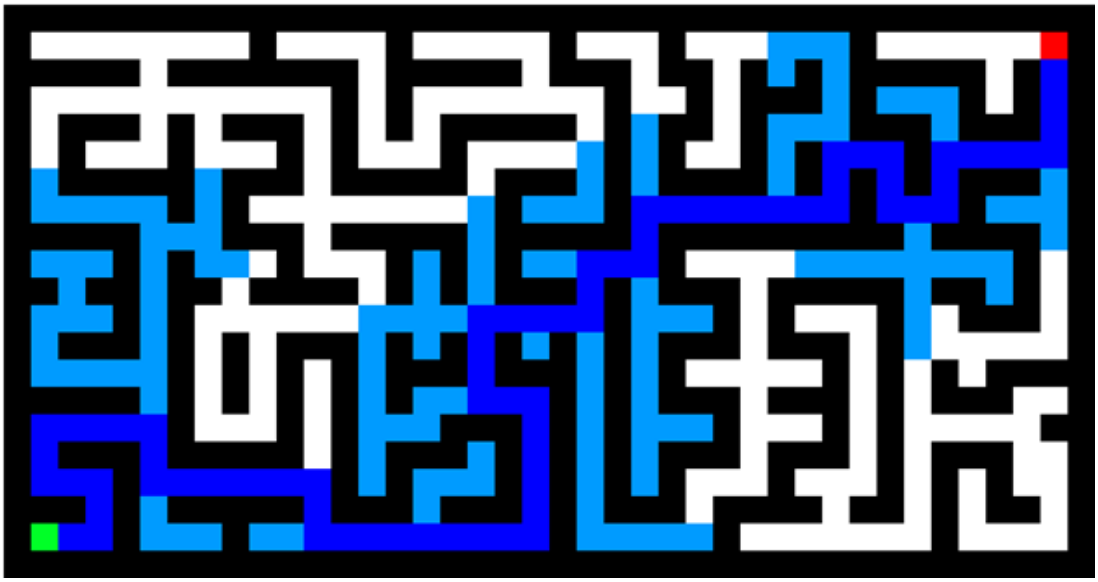


Fig. 4.1 *Conceptul inițial pentru program: Cele două puncte roșu și verde pentru reprezentarea nodurilor scop si start, ilustrarea nodurilor parcurse cu albastru deschis și drumul final cu albastru închis*

4.2 Implementare

În urma experimentării cu diferite metode de generare a unui labirint și explorării a diferite tehnologii ce ar fi putut să îmi faciliteze viziunea pentru aplicație, am decis să înlocuiesc ideea de a desemna anumite noduri din labirint ca fiind ziduri și am preferat să consider fiecare nod din matrice ca fiind parte din drum și fiecare nod ar reprezenta o cutie cu patru pereți. Pereții vor putea fi eliminați de un algoritm ce generează labirintul și lipsa de pereți intre două noduri ar semnifica faptul că cele două sunt vecine. Din fericire, acest lucru a fost foarte util de realizat cu ajutorul sistemului de Prefab din Unity.

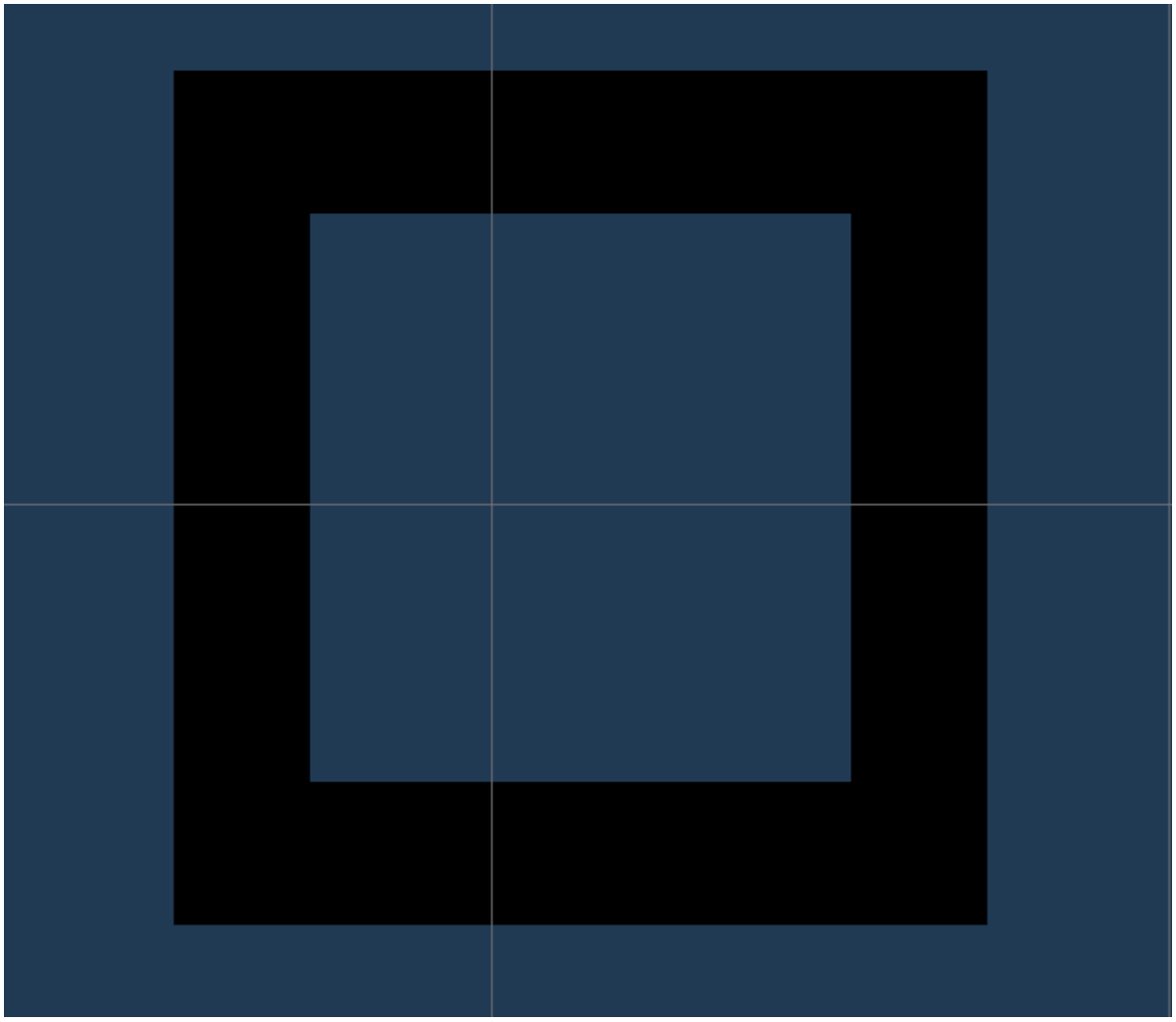


Fig. 4.2 *Prefab-ul de tip "Cutie" folosit pentru reprezentarea elementelor din labirint*

După crearea graficilor pentru o celulă, a fost necesară definirea lor ca o structură ce poate fi duplicată si modificată pentru a permite algoritmului de generare să le formeze într-un labirint competent. Precum am prezentat mai devreme în descrierea utilizării script-urilor ca și componente ce manipulează un GameObject, putem face legătura între aceste elemente și cod foarte ușor și putem adăuga noi funcții pentru a le putea modifica. Am legat astfel prefab-ul denumit "Maze Node" la un script denumit "MazeNode.cs" ce conține declarări ale tuturor obiectelor prefab-ului cât și funcții publice ce pot fi apelate de alte scripturi pentru a altera trăsăturile fiecărui element generat de program.

```
1 reference
public void SetAsStart()
{
    isStart = true;
    isEnd = false;
    _unvisitedBlock.SetActive(true);
    Transform graphics = _unvisitedBlock.transform.Find("Graphics");
    if (graphics != null)
    {
        SpriteRenderer spriteRenderer = graphics.GetComponent<SpriteRenderer>();

        if (spriteRenderer != null)
        {
            spriteRenderer.color = Color.green;
        }
    }
}
```

Fig. 4.3 *Funcție ce setează un nod ca fiind nodul de start și îi schimba culoarea si flag-urile*

Următorul pas l-a reprezentat crearea unui script pentru generarea labirintului. Am folosit pentru acest scop algoritmul de Recursive Backtracking, o variație a algoritmului DFS care alege următorul nod pentru vizitat în mod aleatoriu. Modul cum funcționează este următorul:

- Alege un nod pentru a începe (acesta va fi nodul curent)
- Îl marchează ca fiind vizitat
- Cât timp celula curentă are vecini nevizitați:
    - Alege aleatoriu unul dintre vecinii nevizitați,
    - Elimină pereții dintre nod si vecin
    - Apelează funcția recursiv pentru vecinul ales

Astfel, în momentul în care celula curentă nu are vecini, algoritmul revine la ultima celula ce avea vecini și continuă execuția până când nu mai există celule nevizitate.

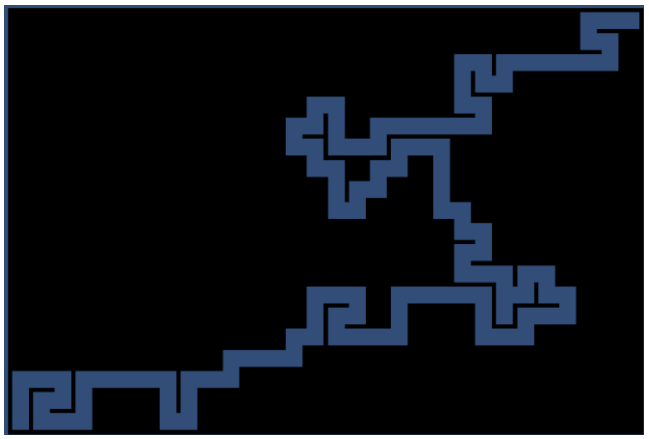


Fig. 4.4 *Scriptul de generare in acțiune*

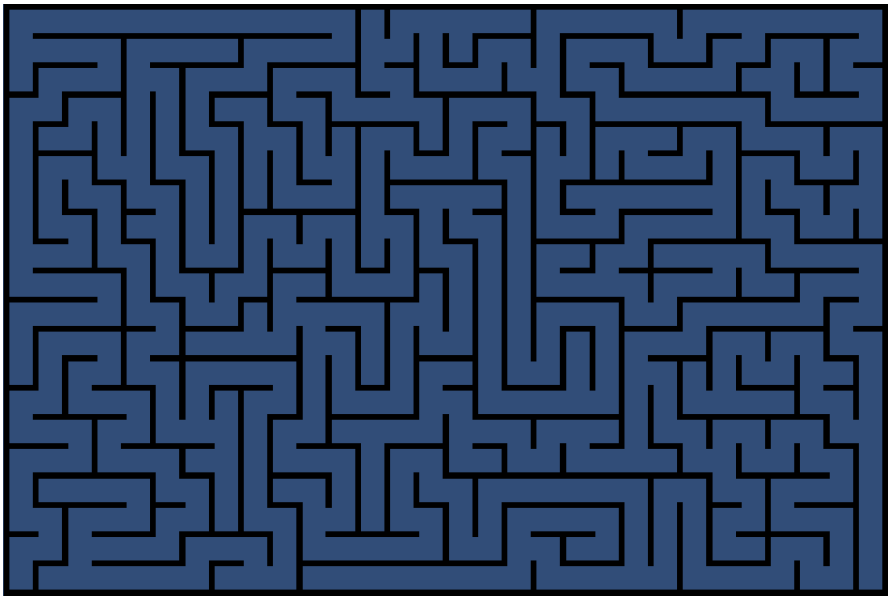
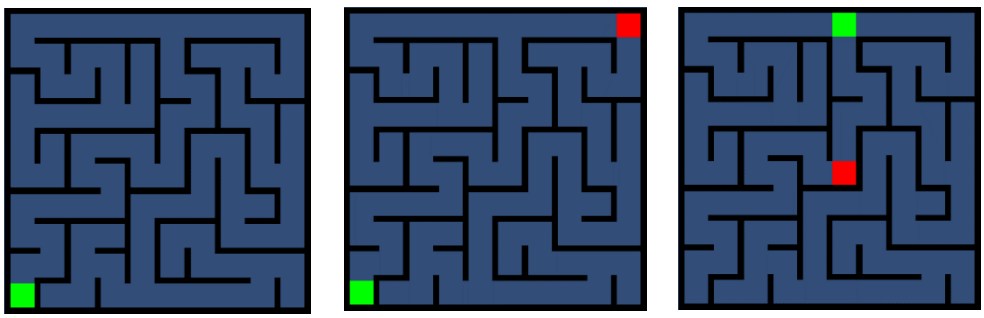


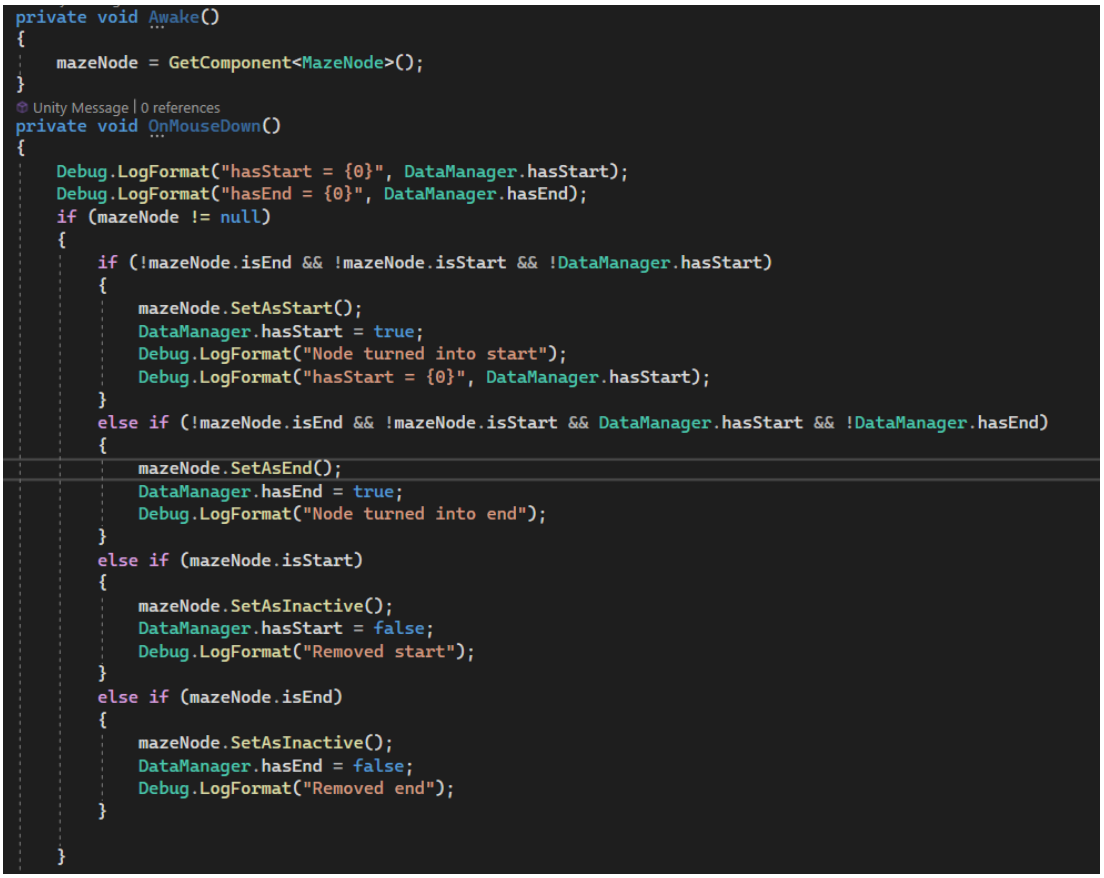
Fig. 4.5 *Labirintul finalizat*

Cu labirintul acum creat, pentru a permite algoritmilor de parcurgere să îl... parcurgă, a trebuit să implementez un script pentru a seta punctele de start si final. Deoarece una dintre motivațiile principale în crearea proiectului a fost accesibilitatea, am vrut să permit utilizatorului să poată apăsa direct pe o celulă pentru a o putea seta ca start, respectiv final. Am realizat această funcționalitate prin crearea unui script ce detectează GameObject-ul pe care utilizatorul apasă, apelează funcții din scriptul de MazeNode pentru a schimba aspectul și modifică într-un alt script un set de variabile globale care determină dacă există deja celule de start și final și in funcție de aceste variabile am format următoarele condiții:

- Dacă nu există nod start, nodul pe care s-a apăsat va fi nodul start
- Dacă există nod start, nu există nod final și nodul apăsat nu este nodul start, nodul pe care s-a apăsat va fi nodul final
- Dacă există nod start și nodul apăsat este nodul start, se elimină nodul start
- Dacă există nod final și nodul apăsat este nodul final, se elimină nodul final

În acest mod, se pot alege cu ușurință nodurile de start si final și se pot modifica la fel de ușor.





```
private void Awake()
{
    mazeNode = GetComponent<MazeNode>();
}
Unity Message | 0 references
private void OnMouseDown()
{
    Debug.LogFormat("hasStart = {0}", DataManager.hasStart);
    Debug.LogFormat("hasEnd = {0}", DataManager.hasEnd);
    if (mazeNode != null)
    {
        if (!mazeNode.isEnd && !mazeNode.isStart && !DataManager.hasStart)
        {
            mazeNode.SetAsStart();
            DataManager.hasStart = true;
            Debug.LogFormat("Node turned into start");
            Debug.LogFormat("hasStart = {0}", DataManager.hasStart);
        }
        else if (!mazeNode.isEnd && !mazeNode.isStart && DataManager.hasStart && !DataManager.hasEnd)
        {
            mazeNode.SetAsEnd();
            DataManager.hasEnd = true;
            Debug.LogFormat("Node turned into end");
        }
        else if (mazeNode.isStart)
        {
            mazeNode.SetAsInactive();
            DataManager.hasStart = false;
            Debug.LogFormat("Removed start");
        }
        else if (mazeNode.isEnd)
        {
            mazeNode.SetAsInactive();
            DataManager.hasEnd = false;
            Debug.LogFormat("Removed end");
        }

    }
```

Fig. 4.6 *Script-ul pentru alegerea startului si finalului*

Ajungem astfel la funcționalitatea principală a programului, algoritmii de parcurgere. Fiecare dintre algoritmi este dezvoltat într-un fișier de tip script, utilizând funcții comune care determină relațiile dintre vecini in funcție de pereții dezactivați de scriptul de generare a labirintului și funcții care, în urma finalizării execuției, permit afișarea în ordine a celulelor vizitate până la găsirea celulei de final. Afișarea acestor informații se face întârziat, pentru a permite utilizatorului să observe modul în care fiecare algoritm decide să aleagă următorul nod de parcurs. Acest comportament este realizat cu ajutorul Coroutine-lor menționate anterior ca fiind specifice clasei MonoBehaviour. Tot cu ajutorul acestor Coroutine putem să cronometrăm timpul de execuție al fiecărui algoritm cu scopul de a afișa această informație pentru a putea compara diferiții algoritmi.

Pentru executarea fiecărui algoritm în parte am decis să le atribui câte un buton, Unity având un GameObject deja configurat pentru acest scop, permițând rularea unei funcții publice dintr-un script în urma apăsării butonului prin funcționalitatea On Click().

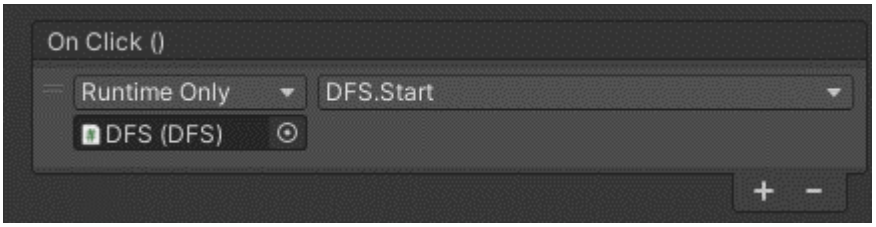


Fig. 4.7 *Componenta buton si opțiunea de a executa o funcție din script după apăsare*

De asemenea, Unity permite foarte ușor afișarea datelor din program prin obiecte de tip Text cărora le pot fi atribuite string-uri generate în interiorul programului. Acest fapt mi-a permis să afișez timpul cronometrat pentru fiecare algoritm în parte pentru a le putea compara și analiza în același labirint la nivel de runtime:

Am adăugat pe lângă aceste funcționalități și două butoane: unul pentru a curăța labirintul de reprezentările unui algoritm pentru a permite rularea consecutivă a tuturor algoritmilor și unul pentru a genera labirintul încă o dată, pentru a genera unul care ar putea evidenția variația unora dintre metodele de parcurgere la nivel de performanță când scenariul este nefavorabil.

Astfel, am implementat o aplicație ce poate facilita înțelegerea unor algoritmi de parcurgere, îi compară în mai multe moduri și permite unui student să tragă concluzii relevante legate de complexitățile lor raportate la timp și spațiu.

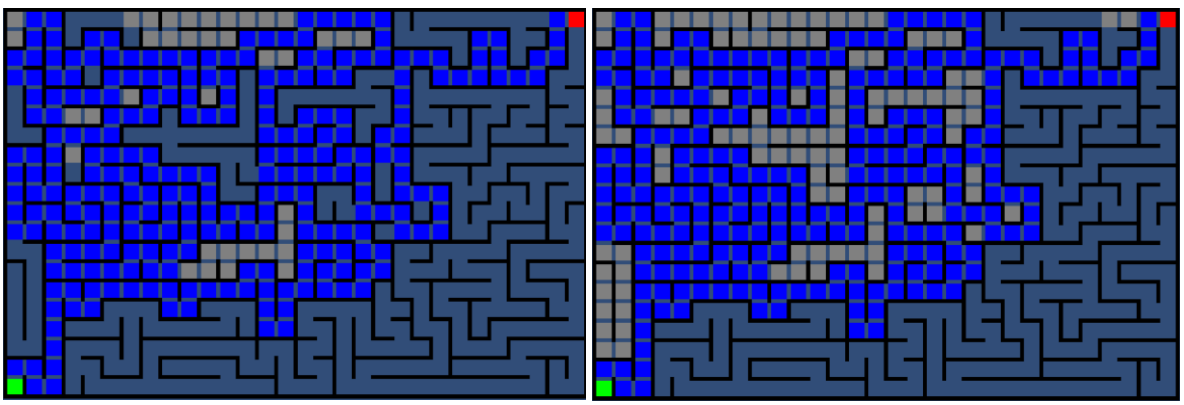


Fig. 4.8.1 *O rulare a algoritmilor A* (stânga), Dijkstra (dreapta);*

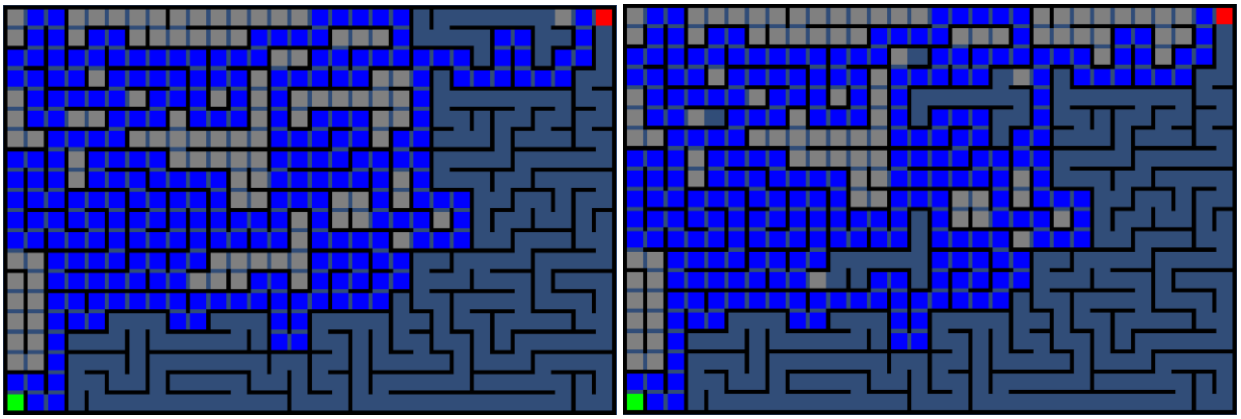


Fig. 4.8.2 *BFS (stânga) și DFS (dreapta).*

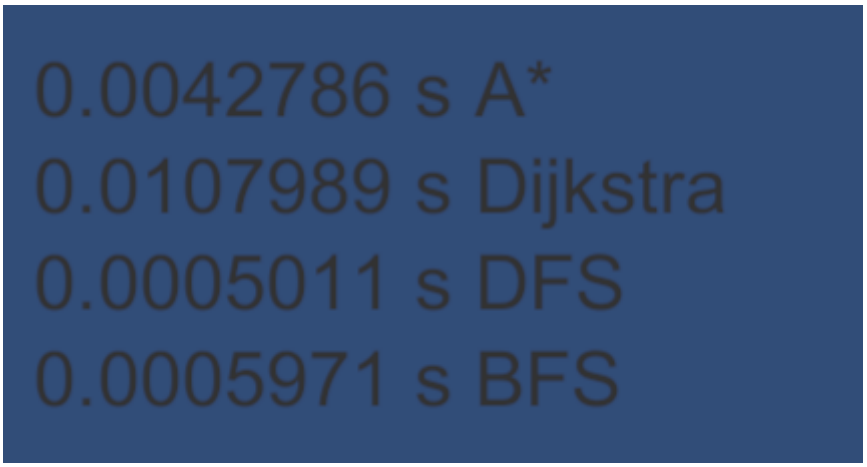


Fig. 4.8.3 *Timpii de rulare.*

5. Comparații între algoritmi

5.1 BFS vs. DFS

După implementarea și folosirea aplicației se poate observa că implementările din proiect ale BFS si DFS sunt apropiate la nivel de complexitate timp, diferența dintre cele două fiind bazată pe variațiile din generarea labirintului și poziționarea nodurilor de start si final. BFS preferă scenarii în care cele două noduri sunt apropiate, în timp ce DFS este mai avantajat de drumuri lungi între ele.

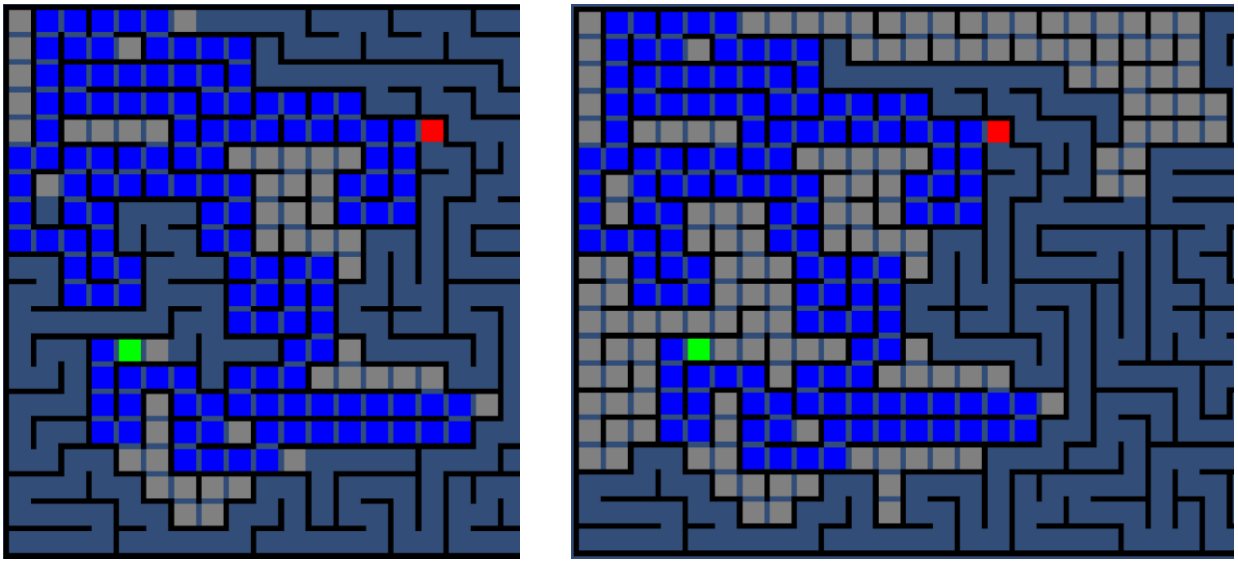


Fig. 5.1.1 *Parcurgeri DFS(stânga) respectiv BFS(dreapta). Se observă ușor faptul că BFS a vizitat mult mai multe celule în acest caz decât DFS.*

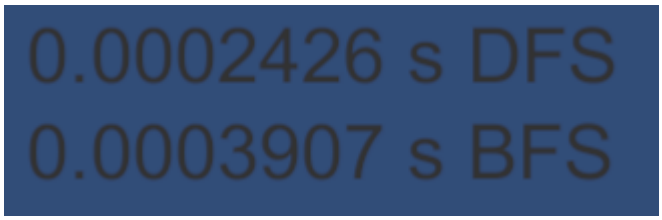


Fig. 5.1.2 *Iar acest lucru este reflectat de diferența în timpul de execuție.*

În cazul implementării curente a aplicației, labirintul are dimensiunile maxime de 20x30 pentru a putea fi prezentat pe ecran. Din acest motiv, în majoritatea testelor, algoritmii de BFS si DFS tind să fie cei mai rapizi la prima vedere. Cu toate acestea, este important să considerăm contextul fiecărei rulări a programului când vine vorba de analiza rezultatelor:

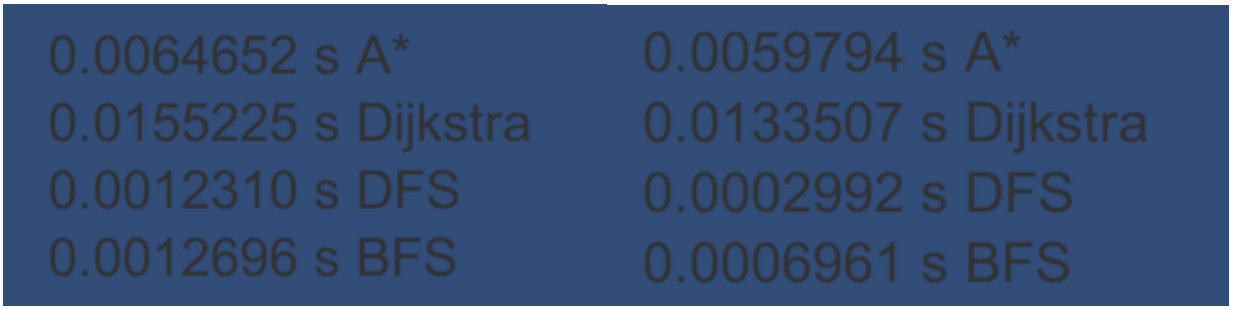


Fig. 5.2.1 *Rezultatele de mai sus par să susțină ideea că DFS este cel mai rapid dintre algoritmi*

Fig. 5.2.2 *În timp ce o altă execuție respinge această concluzie. Motivul?*

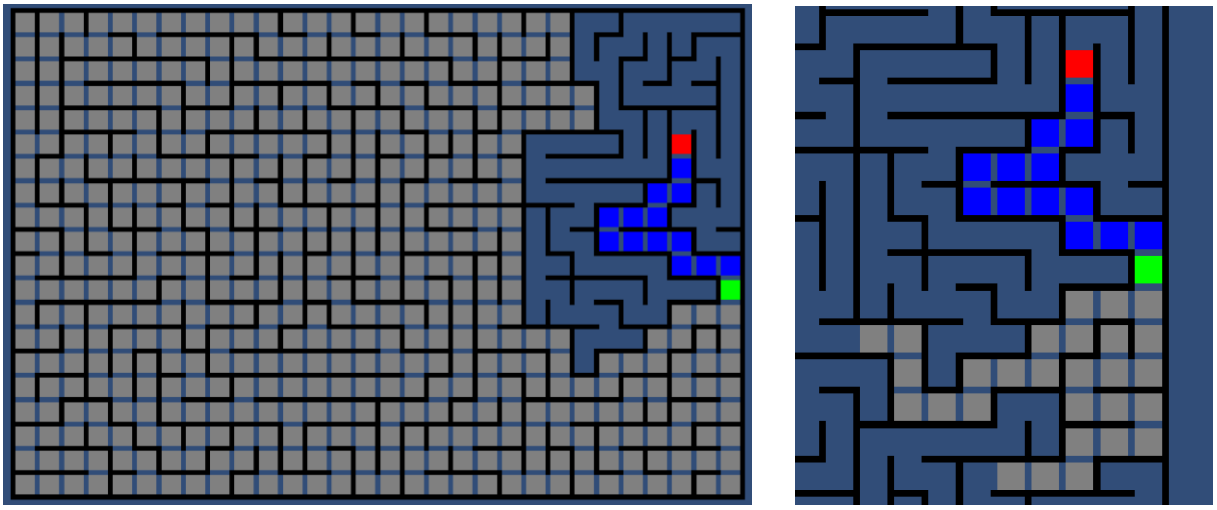


Fig. 5.2.3 *Plasarea celor două noduri foarte aproape și cu un drum drept între ele. Parcurgerea din stânga este cu DFS, iar cea din dreapta cu BFS. DFS nu poate prioritiza parcurgerea drumului corect, iar dacă drumul pe care îl parcurge primul se întâmplă să fie greșit, îl va parcurge până ajunge la capăt de drum pentru a începe explorarea în sensul opus.*

Din acest motiv, BFS tinde să fie mai folositor în această implementare deoarece, deși DFS poate fi mai eficient în anumite situații, acesta este constant la nivel de performanță și nu cade niciodată mult în urma celorlalți algoritmi datorită faptului că explorează toate direcțiile de deplasare în mod egal. DFS este eficient în cazuri favorabile și extrem de ineficient în cazuri nefavorabile.

Fig. 5.3.1 *În condițiile optime, până și DFS poate avea un rezultat cu mult superior restului*

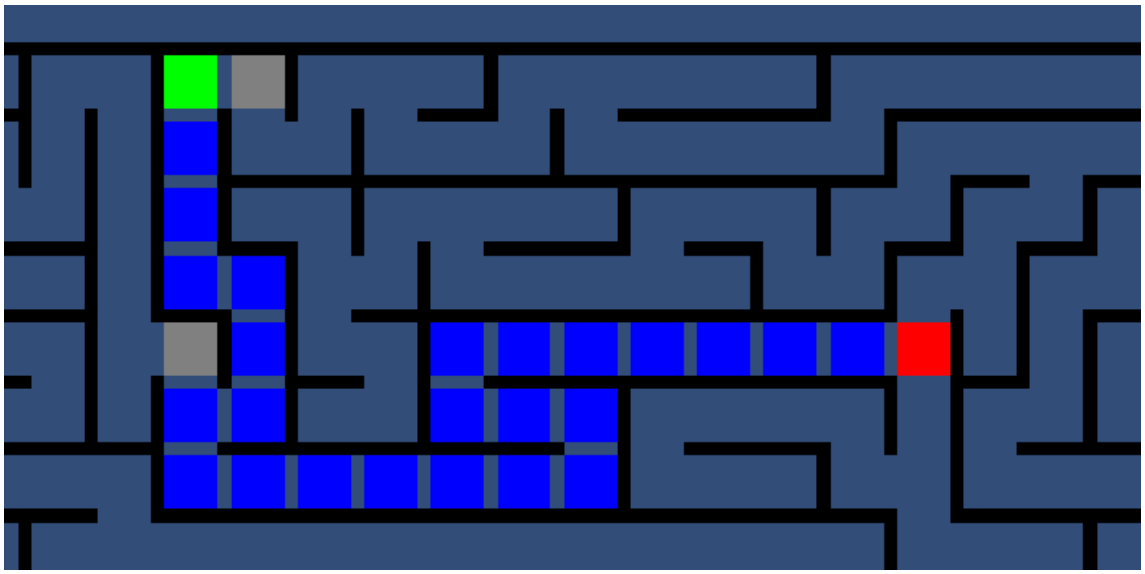


Fig. 5.3.2 *Parcurgerea DFS a cronometrării de mai sus. Primul drum pe care îl parcurge algoritmul se întâmplă să fie cel corect.*

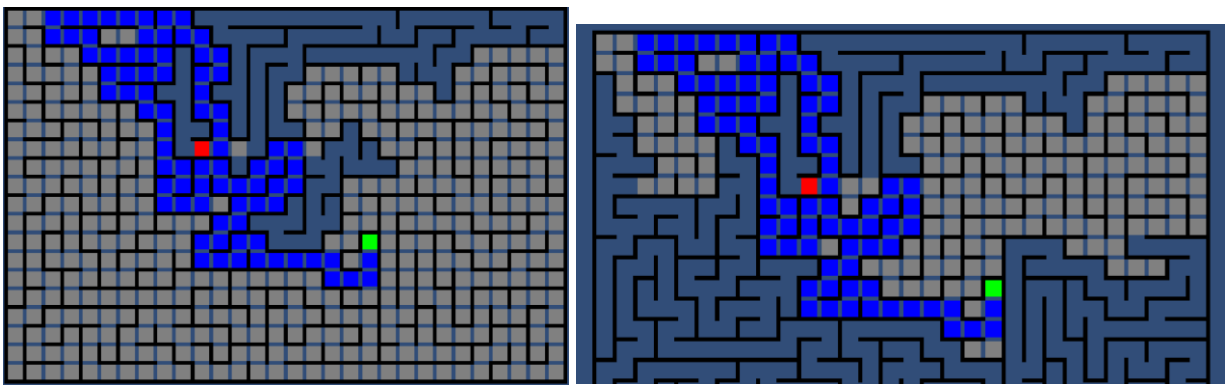


Fig. 5.4 *BFS(dreapta) este, mereu capabil să obțină un rezultat satisfăcător în comparație cu DFS(stânga) care depinde prea mult de poziția nodurilor de reper și forma labirintului.*

5.2 A* vs. restul

A* tinde să fie cel mai eficient din punct de vedere al memoriei (nu vizitează la fel de multe celule nefolositoare) și întrece toate celelalte implementări într-un labirint suficient de mare. Avantajul masiv pe care îl prezintă A* este faptul că euristica îi permite omiterea anumitor noduri care nu s-ar apropia de nodul final. Chiar dacă euristica consumă mai multe resurse, algoritmul este pe departe cel mai eficient la nivel de memorie utilizată.

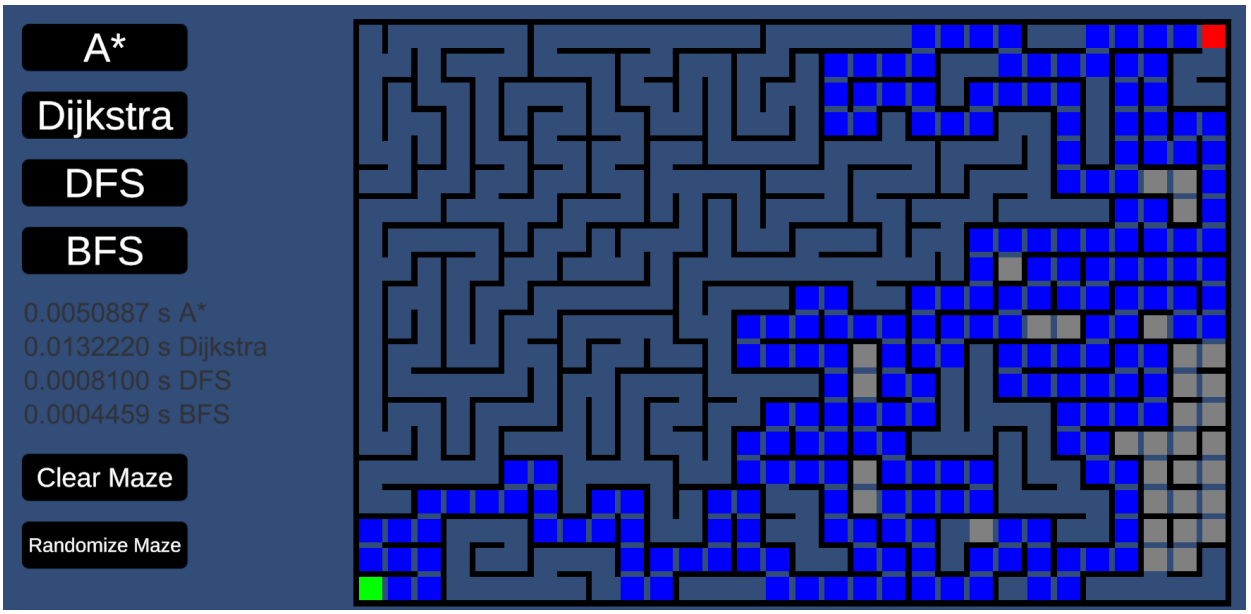


Fig. 5.5.1 *Un caz favorabil pentru A**

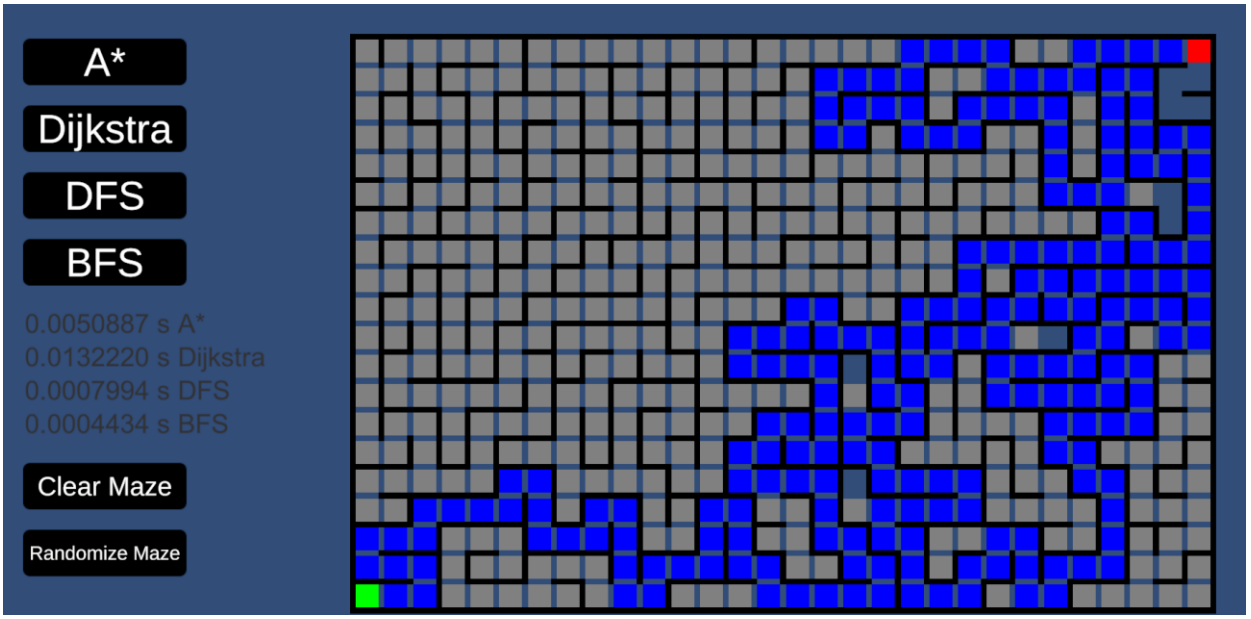


Fig. 5.5.2 *Dar foarte nefavorabil pentru DFS. Cu toate acestea, timpul de execuție pare să fie în favoarea algoritmului ce parcurge aproape tot labirintul.*

Diferența relativ mare dintre A*, algoritmul care pare să fie cel mai adept în ceea ce privește evitarea vizitării nodurilor inutile, și cele care parcurg labirintul folosind reguli simple poate fi explicată prin dimensiunea redusă a labirintului. Verificarea tuturor celulelor într-o configurație ce conține doar 600 este mult mai ușoară din punct de vedere computațional decât calcularea costurilor pentru fiecare nod in cazul lui A*. Cu toate acestea, într-un labirint destul de masiv este posibil ca omiterea unui număr mare de celule în parcurgere să fie mai atractivă ca timp de rulare.

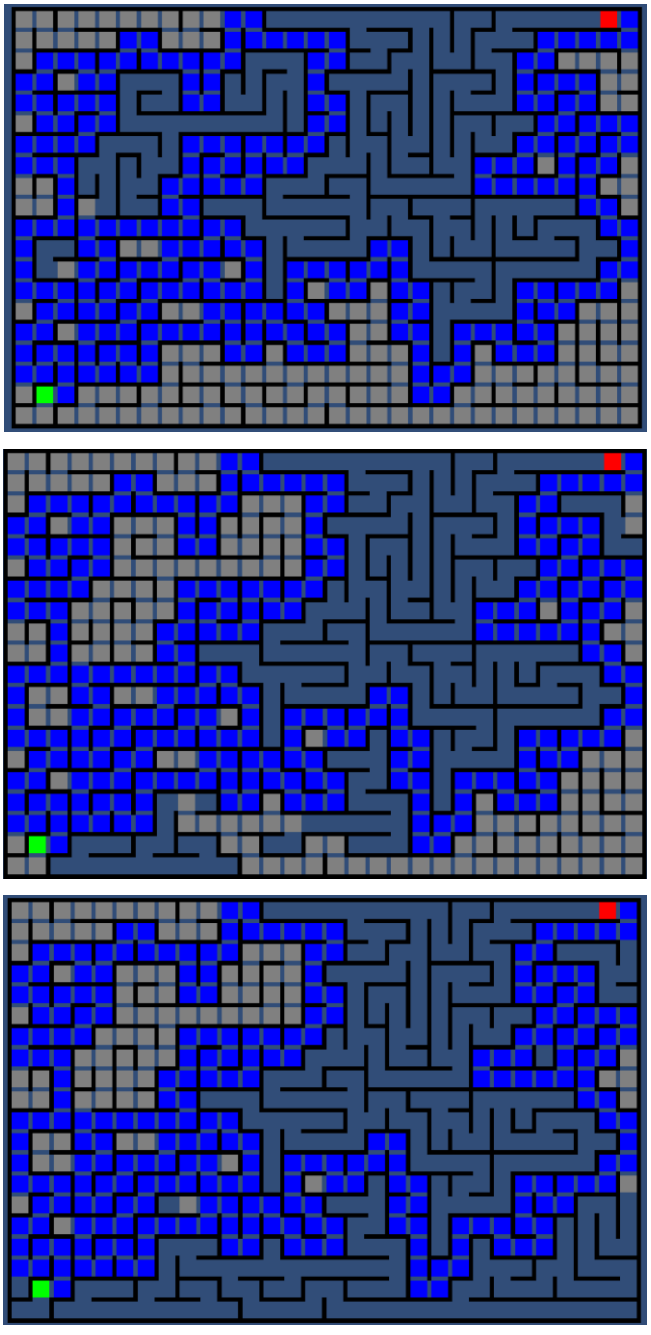


Fig. 5.6 *Parcurgeri DFS(a), BFS(b), A*(c). Se observă un număr similar de celule inutile vizitate în primele două parcurgeri, însă este evident că A* a vizitat cele mai puține noduri datorită euristicii cu distanța Manhattan.*

5.3 Dijkstra vs. el însuși

Dijkstra tinde să fie cel mai încet dintre cele 4 deoarece calcularea constantă a distanței de la nodul de start la celelalte noduri crește cerința computațională, însă rolul său în program este în principal de a exemplifica eficiența algoritmului A* cu euristica folosind distanța Manhattan. De asemenea, legat de algoritmul lui Dijkstra, trebuie să luam în considerare faptul că, deși este fără nevoie de argumentare cel mai slab la nivel de timp, acesta are o funcționalitate complexă pe care niciunul dintre ceilalți algoritmi prezentați nu o are: determinarea celui mai scurt drum de la nodul de start la toate celelalte noduri din labirint. Dacă aș fi implementat o metodă de a stoca permanent valorile obținute în urma rulării algoritmului, aș putea muta nodul de final în orice altă poziție a labirintului, iar Dijkstra ar trebui doar să acceseze datele legate de execuția inițială pentru a găsi drumul cel mai scurt, în timp ce restul algoritmilor ar trebui să ruleze în întregime pentru fiecare modificare a nodurilor de start și final. Într-un anumit sens, Dijkstra este într-o complet altă ligă față de celelalte implementări și regulile jocului prezent nu îl avantajează. Cu toate acestea, consider că este important să arăt că și un algoritm care nu este ideal pentru problemă o poate rezolva acceptabil.

```
private Dictionary<MazeNode, int> distance = new Dictionary<MazeNode, int>();
```

Fig. 5.7 *Distanța de la nodul de start la toate nodurile vizitate este mereu actualizată în scriptul "Dijkstra.cs" și algoritmul ar putea determina numărul de celule parcurse între start și oricare din ele, o funcționalitate irosită în acest program, dar care poate fi extrem de folositoare în alte aplicații.*

6. Concluzii

Deși proiectul prezentat nu este de o complexitate ridicată, vreau să subliniez că scopul principal a fost de a ilustra concepte de bază într-un mod ușor de analizat și înțeles pentru cineva care tocmai a început drumul în înțelegerea informaticii și este o utilitate la care mi-aș fi dorit să fi avut acces când mă aflam în acel moment. Nu am beneficiat de profesori calitativi în domeniul Informaticii de liceu, descoperindu-mi pasiunea pentru programare abia după ce am avut plăcerea să interacționez cu cadrele didactice din facultatea noastră. Din acest motiv consider că este de o importanță uriașă să oferim metode ce prezintă programarea într-un mod ușor de înțeles, pentru a încuraja și studenții ce preferă predarea vizuală și practică.

În viitor aș vrea să adaug mai multe metode de parcurgere și să ilustrez de ce unele euristici nu ar fi eficiente pentru A*. Aveam inițial ideea să adaug o alternativă în program pentru prezentarea labirintului sub forma unui tile-map în care fiecare tile are un tip de teren și deci un cost asociat, cu scopul de a compara cu mai multă variație unii din algoritmii ce pot lucra cu muchii ponderate.

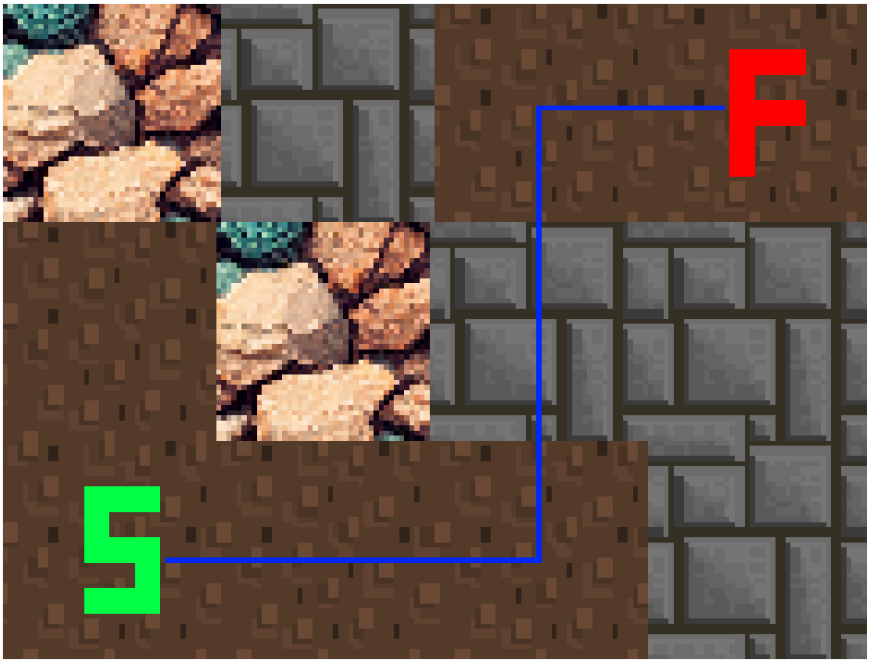


Fig. 6.1 *Prototip pentru programul menționat*

În exemplul de mai sus, toate celulele au 4 vecini și deplasarea nu mai este limitată de ziduri, ci numai de dificultatea de traversare a fiecărui tip de teren. Cazul acesta ar permite evidențierea utilității algoritmului lui Dijkstra, iar adăugarea de terenuri ce se consideră de cost negativ mi-ar permite să pun accentul pe proprietatea algoritmului Bellman-Ford de a interacționa cu muchii negative.

O altă funcționalitate pe care intenționez să o adaug o reprezintă posibilitatea de a schimba de la afișarea sub forma de grid a labirintului într-o formă de graf, pentru a ilustra cu mai mare ușurință corelarea dintre celula din labirint și nodul unui graf.

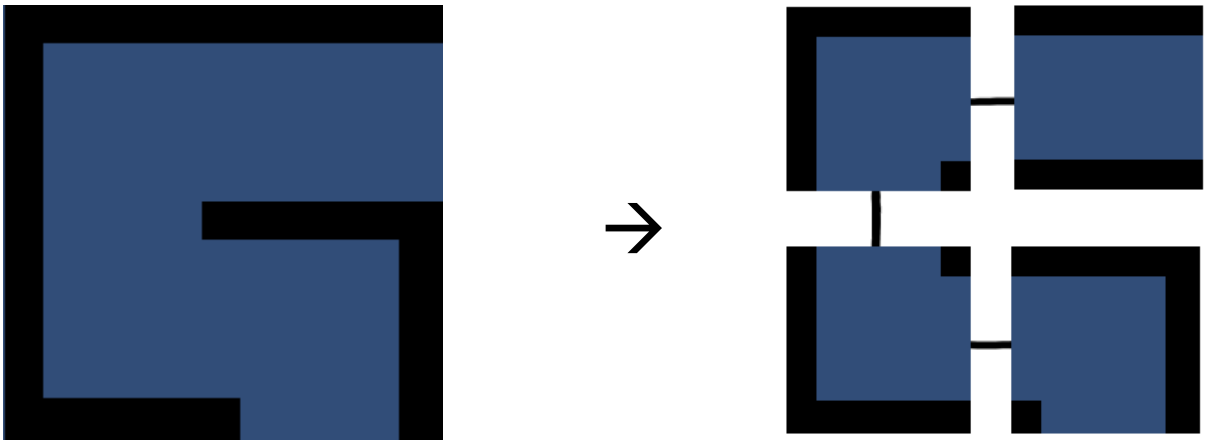


Fig. 6.2 *Sugestie de reprezentare sub formă de graf.*

Am considerat de asemenea ilustrarea unei concatenări (sau simplificări) a unor serii de celule ce reprezintă drumuri fără intersecții din labirint sub celule mai mici ce ar avea costul egal cu numărul de celule înlocuite. Ideea principală era să arăt că labirintul poate fi simplificat sub forma unui graf ponderat.

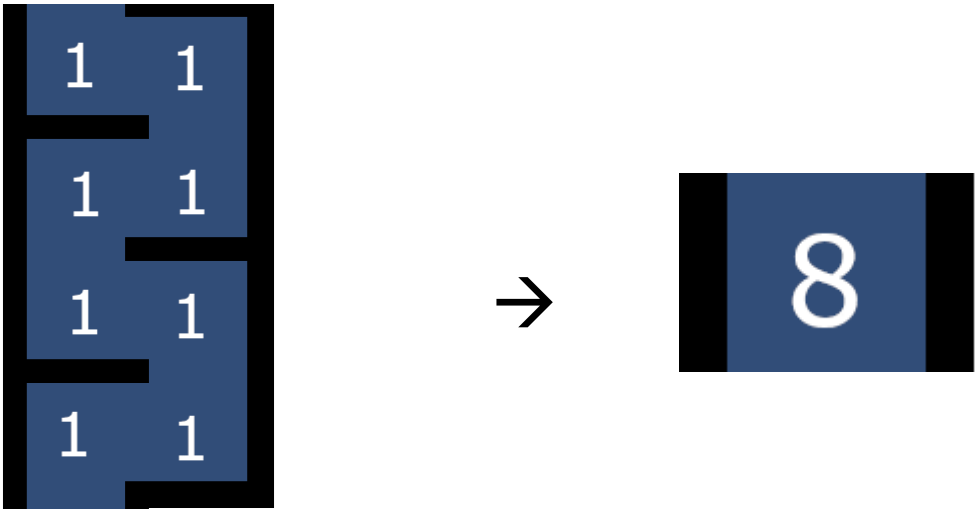


Fig. 6.3 *Ideea de concatenare a celulelor.*

În final, aș vrea să reușesc să creez o suită de aplicații open-source pentru a ilustra diferite concepte din programare și orice alte detalii pe care le-am considerat dificile de-a lungul anilor. Deși rolul lor principal ar fi facilitarea educării altor persoane, interacțiunea la un nivel profund cu aceste definiții îmi permite să îmi actualizez și să îmi îmbunătățesc propriile cunoștințe.

Bibliografie

[1] https://www.shiftelearning.com/blog/bid/350326/studies-confirm-the-power-of-visuals-in-elearning (ShiftLearning article ilustration) - accesat 29.08.2023

[2] https://unity.com/products/unity-personal (Unity website, pricing) - accesat 29.08.2023

[3] https://docs.unity3d.com/Manual/CreatingScenes.html (Unity Manual, Scenes) - accesat 30.08.2023

[4] https://docs.unity3d.com/Manual/GameObjects.html (Unity Manual, GameObjects) - accesat 30.08.2023

[5] https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ (C# Language Specification - Ecma International, Introduction) - accesat 30.08.2023

[6] https://support.unity.com/hc/en-us/articles/206336795-What-platforms-are-supported-by-Unity- - accesat 31.08.2023

[7] Bender, Edward A.; Williamson, S. Gill (2010) , “Lists, Decisions and Graphs” https://books.google.ro/books?id=vaXv_yhefG8C&redir_esc=y , pg 147-149 - accesat 30.08.2023

[8] https://neo4j.com/blog/graph-search-algorithm-basics/ - accesat 31.08.2023

[9] https://adrianmejia.com/data-structures-for-beginners-graphs-time-complexity-tutorial/ - accesat 31.08.2023

[10] Mehlhorn, Kurt; Sanders, Peter (2008). "Chapter 10. Shortest Paths" (PDF). Algorithms and Data Structures: The Basic Toolbox. Springer. doi:10.1007/978-3-540-77978-0. ISBN 978-3-540-77977-3. URL: https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/ShortestPaths.pdf - accesat 31.08.2023