**Iesirea din labirint folosind algoritmi de inteligenta artificial**

Proiectul consta in determinarea ieșirii din labirint folosind algoritmi de inteligență artificială.

Un algoritm de rezolvare a labirintului este o metodă automată de rezolvare a unui labirint. Algoritmii mouse-ului aleatoriu, adepților de perete, Pledge și Trémaux sunt proiectați pentru a fi utilizați în interiorul labirintului de către un călător fără cunoștințe anterioare despre labirint, în timp ce algoritmii de umplere a fundului și calea cea mai scurtă sunt proiectați pentru a fi utilizați de o persoană sau program de calculator care poate vedea întregul labirint simultan.

Labirinturile care nu conțin bucle sunt cunoscute ca „pur și simplu conectate” sau „perfecte” și sunt echivalente cu un arbore în teoria grafurilor. Algoritmii de rezolvare a labirintului sunt strâns legați de teoria grafurilor. Intuitiv, dacă cineva trage și întinde potecile în labirint în mod corespunzător, rezultatul ar putea fi făcut să semene cu un arbore.

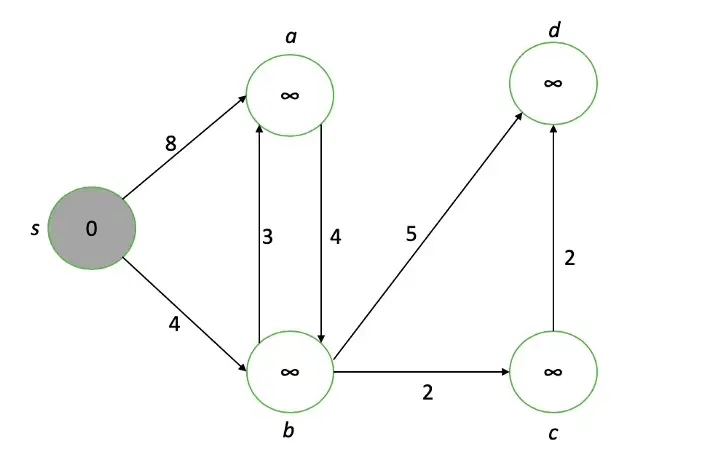
Când un labirint are mai multe soluții, rezolvatorul poate dori să găsească calea cea mai scurtă de la început până la sfârșit. Există mai mulți algoritmi pentru a găsi cele mai scurte căi, majoritatea provenind din teoria grafurilor. Un astfel de algoritm găsește calea cea mai scurtă prin implementarea unei căutări pe lățime (BFS), în timp ce un altul, algoritmul A\*, folosește o tehnică euristică. Algoritmul de căutare pe lățimea întâi folosește o coadă pentru a vizita celulele în ordine crescătoare a distanțelor de la început până când se ajunge la final. Fiecare celulă vizitată trebuie să țină evidența distanței sale de la început sau care celulă adiacentă mai aproape de început a determinat să fie adăugată la coadă. Când este găsită locația de final, urmați calea celulelor înapoi până la început, care este calea cea mai scurtă. Căutarea pe lățimea întâi în forma sa cea mai simplă are limitări, cum ar fi găsirea celei mai scurte căi în graficele ponderate.

**Alti algoritmi de solutionare**

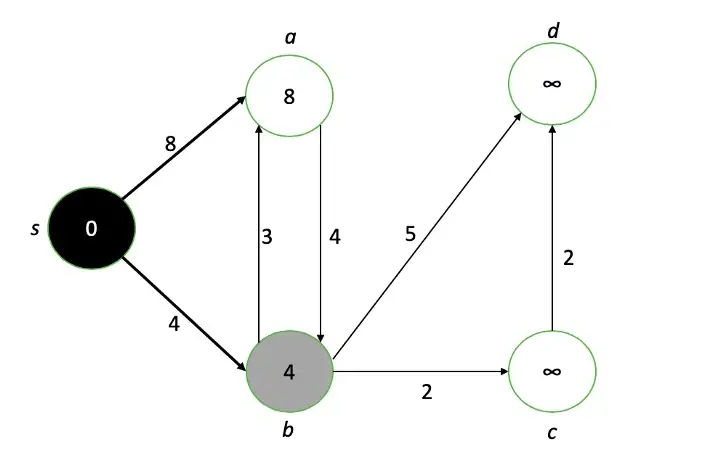
**Algoritmul lui Dijkstra**

Algoritmul lui Dijkstra este unul dintre cei mai populari algoritmi de bază de teorie a grafurilor. Este folosit pentru a găsi calea cea mai scurtă între noduri pe un grafic direcționat. Începem cu un nod sursă și cu lungimi de margine cunoscute între noduri.

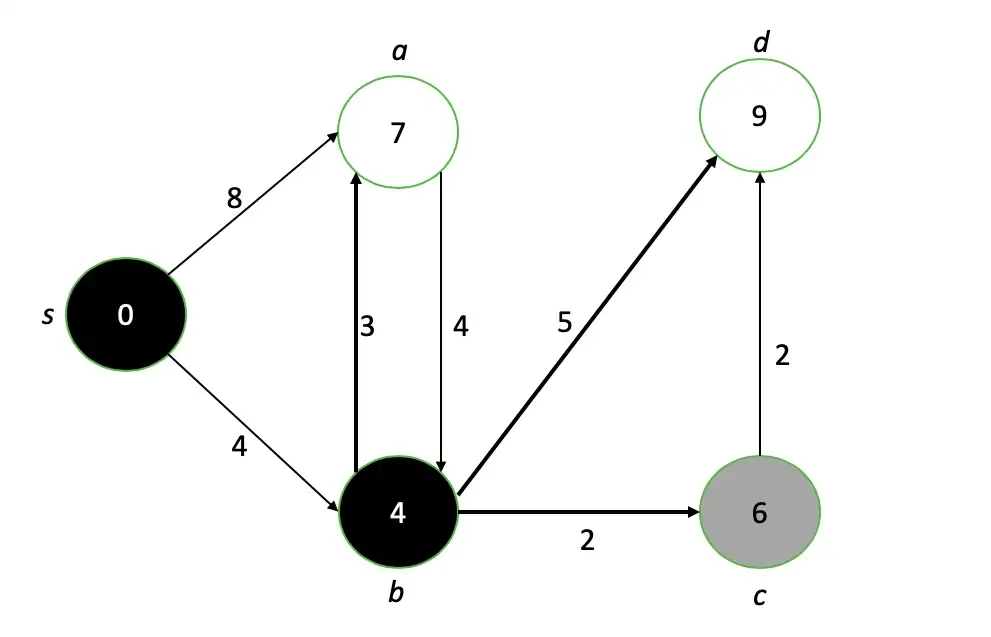
Mai întâi atribuim o valoare a distanței de la sursă tuturor nodurilor. Nodul *s* primește o valoare 0 deoarece este sursa; restul primesc valori de ∞ pentru a începe.



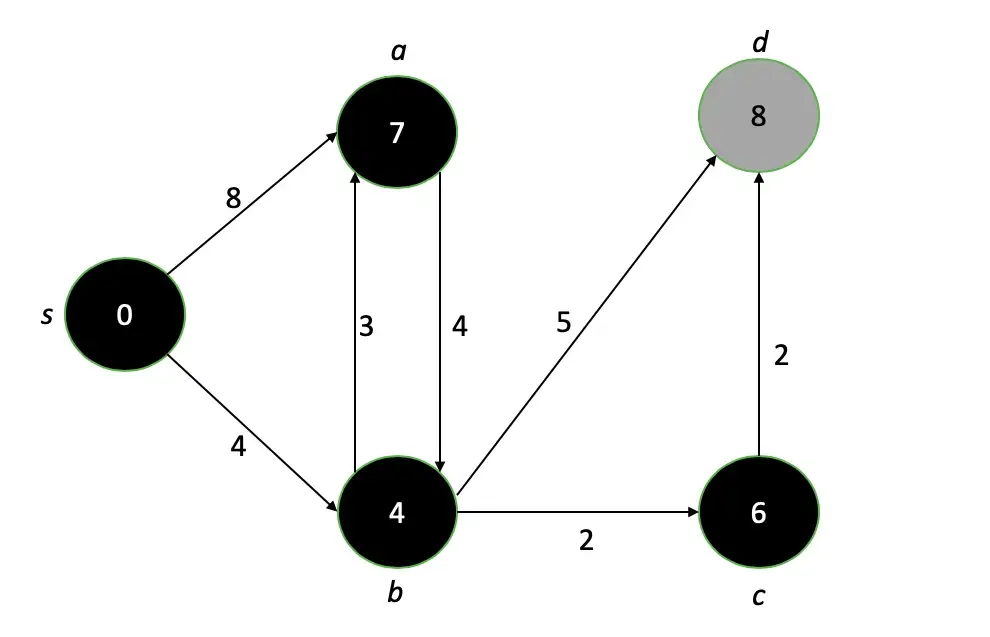
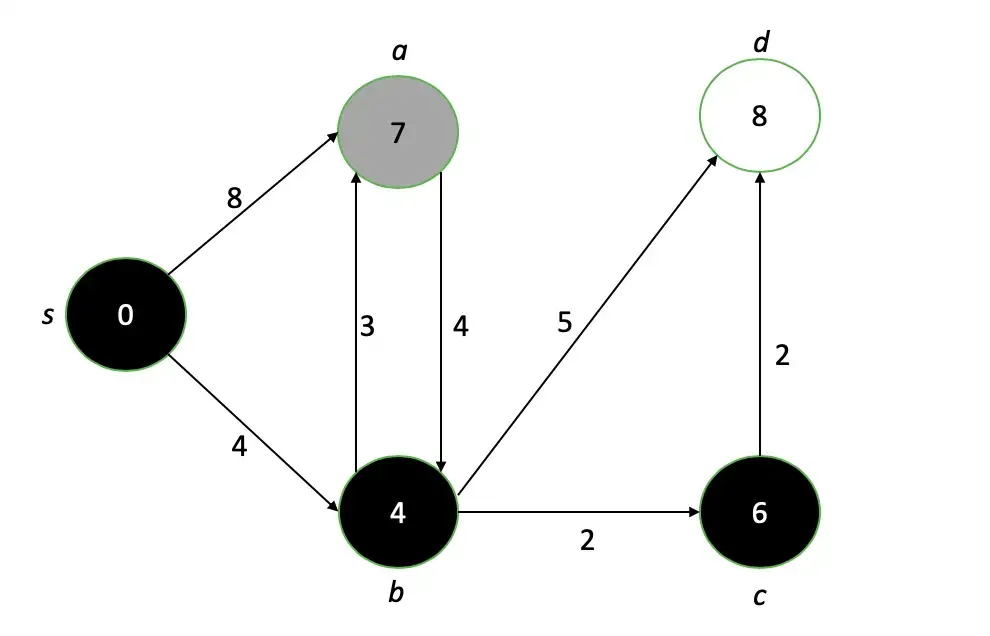
Nodul nostru de interes este nodul neprocesat cu cea mai mică valoare (afișat cu gri), care este *s*. În primul rând, „relaxăm” fiecare vârf adiacent nodului nostru de interes, actualizându-le valorile la minimul valorii lor curente sau valoarea nodului de interes plus lungimea muchiei de legătură.



Nodul *s* este acum finalizat (negru) iar vecinii săi *a* și *b* au preluat noi valori. Noul nod de interes este b, așa că repetăm ​​procesul de „relaxare” a nodurilor adiacente ale lui *b* și finalizarea valorii celei mai scurte căi a lui *b*.



După ce parcurgem fiecare nod, ajungem în cele din urmă cu un grafic care arată cea mai scurtă lungime a căii de la sursă la fiecare nod.



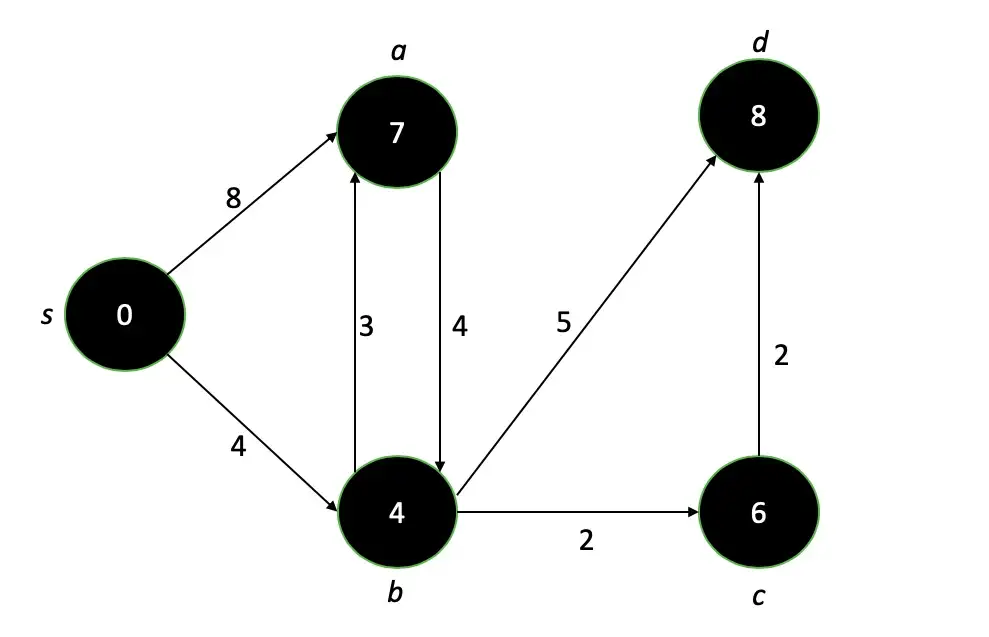
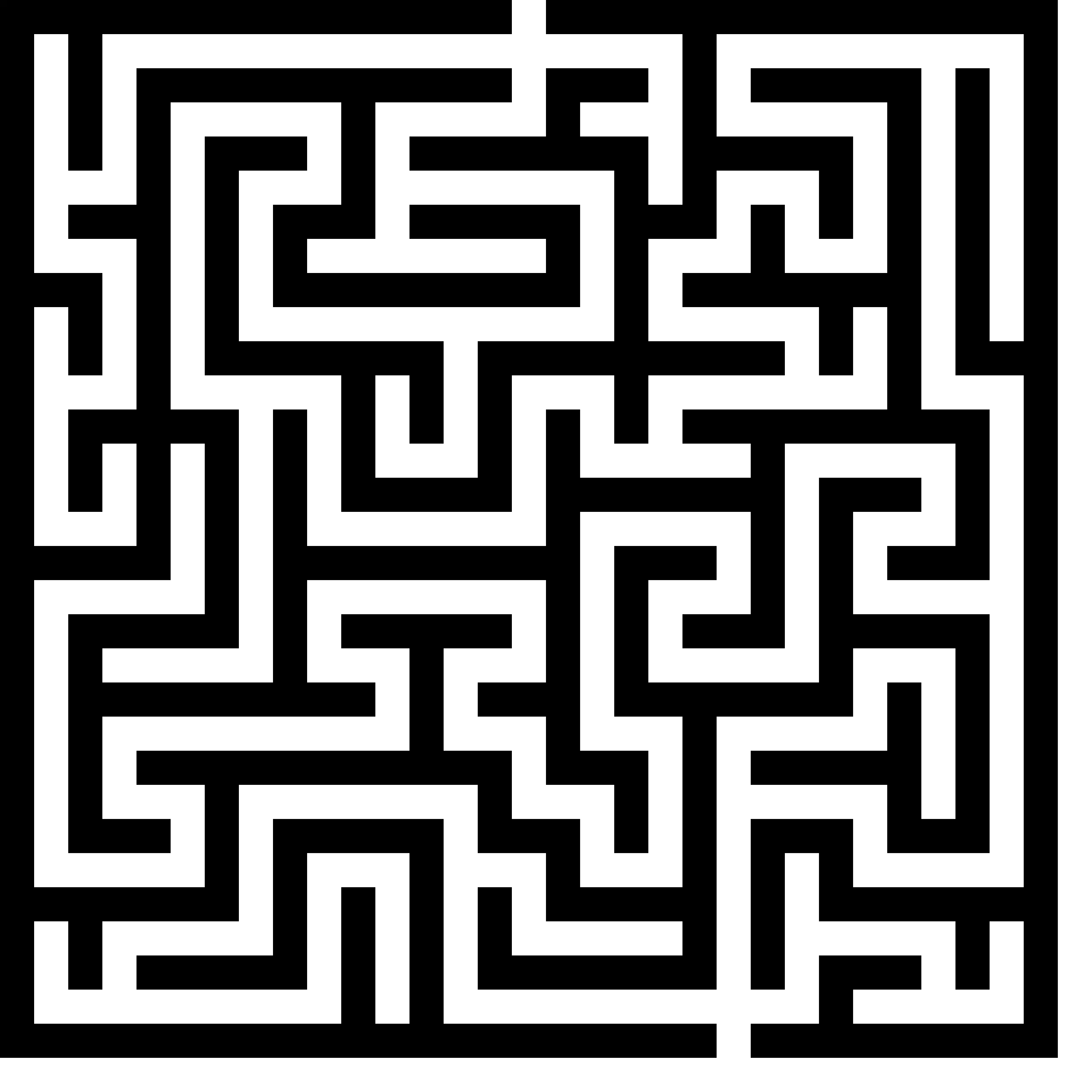


Diagrama noastră finală după rularea algoritmului lui Dijkstra. Numerele din fiecare nod reprezintă cea mai scurtă distanță posibilă de la nodul sursă

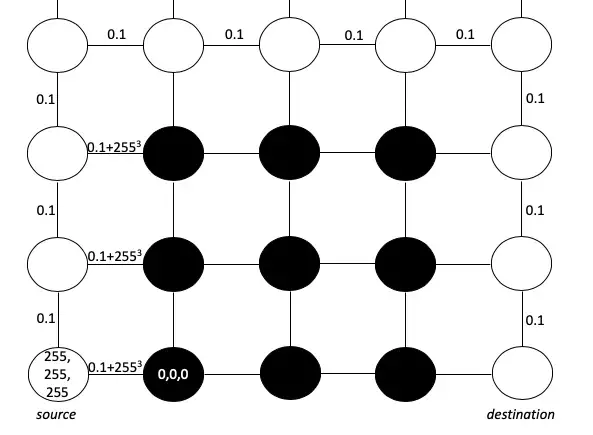
**Conceptualizarea imaginilor labirintului**

****

Ne putem gândi la o imagine ca la o matrice de pixeli. Fiecare pixel (de dragul simplității) are o valoare RGB de 0,0,0 (negru) sau 255,255,255 (alb). Scopul nostru este să creăm o cale cea mai scurtă care să înceapă în alb și să nu treacă în limitele negre. Pentru a reprezenta acest obiectiv, putem trata fiecare pixel ca pe un nod și putem desena margini între pixelii vecini cu lungimi de margine pe baza diferențelor de valori RGB. Vom folosi formula euclidiană a distanței pătrate și vom adăuga 0,1 pentru a ne asigura că nu există lungimi de cale cu distanța 0 (o cerință pentru algoritmul lui Dijkstra):

*distance* = 0.1 + (*R2* – *R1*)2 + (*G2* – *G1*)2 + (*B2* – *B1*)2

Această formulă face ca distanța de traversare a graniței labirintului să fie prohibitiv de mare. După cum putem vedea, cea mai scurtă cale de la sursă la destinație va fi în mod clar *în jurul* barierei, nu prin ea.



Lungimea traseului sunt afişate folosind formula noastră de distanţă de culoare la pătrat euclidianposibilă de la nodul sursă

**Algoritmi folositi**

**Algoritmul Breadth-First Search**

Breadth-First Search este un algoritm „orb”. Se numește „orb” deoarece acestui algoritm nu îi pasă de costul dintre vârfurile din grafic. Algoritmul pornește de la un nod rădăcină (care este starea inițială a problemei) și explorează toate nodurile de la nivelul actual înainte de a trece la nodurile de la nivelul următor. Dacă algoritmul găsește o soluție, o returnează și oprește căutarea, în caz contrar extinde nodul și continuă procesul de căutare. Breadth-First Search este „completă”, ceea ce înseamnă că algoritmul returnează întotdeauna o soluție dacă există. Mai precis, algoritmul returnează soluția care este cea mai apropiată de rădăcină, așa că pentru problemele în care tranziția de la un nod la nodurile sale secundare costă unul, algoritmul BFS returnează cea mai bună soluție. În plus, pentru a explora nodurile nivel cu nivel, folosește o structură de date în coadă, astfel încât noi noduri sunt adăugate la sfârșitul cozii, iar nodurile sunt eliminate de la începutul cozii. Pseudocodul algoritmului BFS este următorul.

Add start in both Frontier and Explored:

Repeat until Goal is reached or Frontier is Empty:

currCell = Frontier.pop(0) = [noduri explorate]

for each direction(ESNW):

childCell=Next Possible Cell

if childCell = Next Possible Cell

otherwise -> Append/Push childCell to both Explored & Frontier

Din cele de mai sus, este evident că pentru a rezolva o problemă folosind un algoritm de căutare precum BFS, trebuie mai întâi să modelăm problema sub formă de grafic și apoi să definim starea inițială a problemei (nodul inițial). După aceea, trebuie să găsim regulile care vor fi urmate pentru extinderea nodurilor (instanțe ale problemei). Aceste reguli sunt determinate de problema în sine. Ultimul lucru pe care trebuie să-l facem este să definim nodul țintă sau un mecanism, astfel încât algoritmul să fie capabil să recunoască nodul țintă.

**Algoritmul A\***

Algoritmul de căutare A-Star (A\*) este un algoritm inteligent pentru a rezolva o problemă de grafic. Spre deosebire de Depth First Search (DFS) și Breadth First Search (BFS), A\* este un algoritm de căutare informat, ceea ce înseamnă că ia în considerare poziția/locația obiectivului în timpul căutării acestuia și, prin urmare, caută destul de multe noduri pentru ajunge la scop.

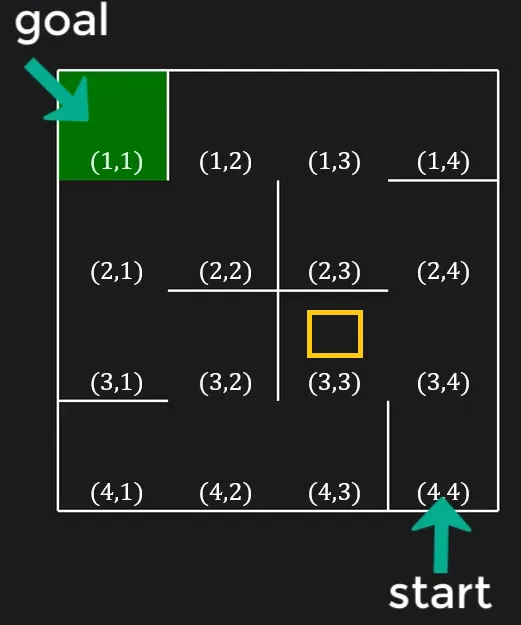
Modul în care funcționează A\* este că atribuie un cost fiecărei celule din labirint, iar algoritmul selectează calea cu cost minim. Costul unei celule (n) are două părți și este definit ca:

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

Unde f(n) este costul total pentru a ajunge la celula n și g(n) și h(n) sunt definite ca:

g(n) → Este costul real pentru a ajunge la celula n de la celula de pornire.

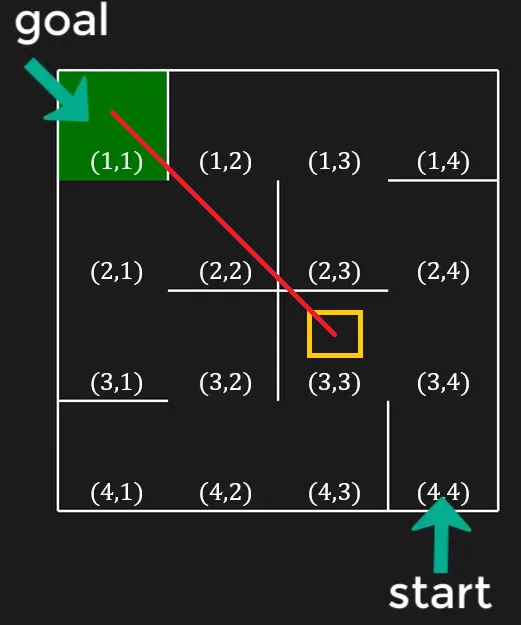
h(n) → Este costul euristic pentru a ajunge la celula obiectiv din celula n. Este costul estimat pentru a ajunge la celula obiectiv din celula n.



Costul g(n) pentru celula (3,3) este 2 deoarece de la celula de start putem ajunge la celula (3,3) în 2 pași. Atunci h(n) este costul estimat pentru a ajunge la celula obiectiv (1,1) din celula (3,3). Nu știm costul pentru a ajunge la celula obiectiv, așa că vom estima pur și simplu asta. Pot exista două funcții pentru a estima acest cost:

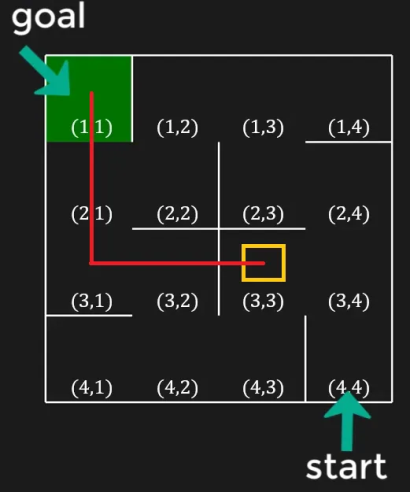
**1- Distanța euclidiană:**

Va fi distanța liniară dintre o celulă și celula obiectiv, așa cum se arată aici:



**2- Manhattan Distanța:**

A doua opțiune poate fi Distanța Manhattan dintre o celulă și celula obiectiv, care este distanța orizontală plus verticală dintre cele două celule.



Costul euristic este doar estimarea costului, iar selecția corectă a funcției euristice este un parametru cheie al algoritmului A\*. Vom folosi Distanța Manhattan ca Funcție Euristică.

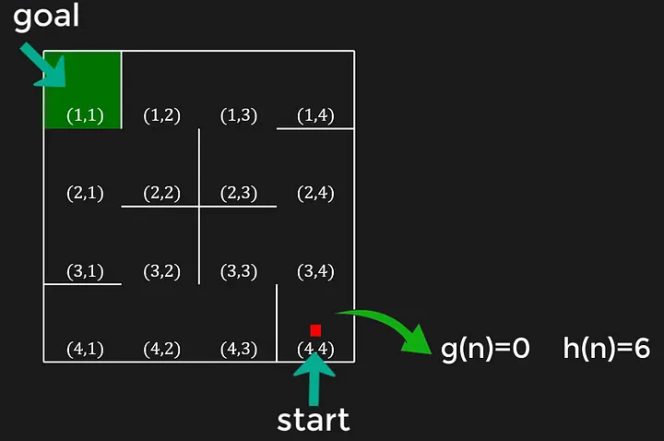
Distanța Manhattan dintre celula (3,3) și celula obiectiv este 4 și, prin urmare, costul total al celulei (3,3) este:

f(n)=g(n)+h(n)=2+4=6

Includerea costului euristic în algoritmul A\* este cea care îl face eficient în comparație cu BFS sau DFS, deoarece algoritmul selectează celulele cu cost minim (cost real + cost estimat) și, prin urmare, se va apropia rapid de obiectiv.

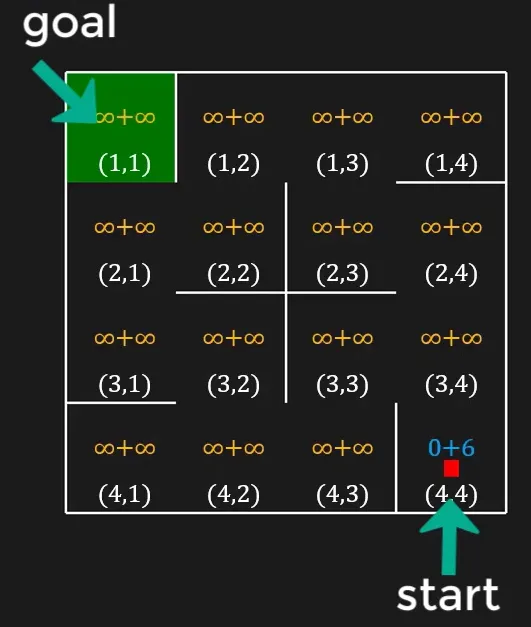
Acum să vedem cum se va extinde algoritmul A\* de la celula de pornire până când ajunge la celula obiectiv.

Aceasta este poziția de pornire a Labirintului, iar pătratul roșu arată celula curentă în care ne aflăm în acest moment, care este celula de pornire.



Pornind de la celula de pornire, g(n) celulei de pornire este 0 deoarece costul de a ajunge la celula de pornire din celula de pornire este evident zero. Și h(n) al celulei de început este 6, care este distanța Manhattan dintre celula de început și celula obiectiv.

Pentru alte celule nu știm costurile; atât g(n) cât și h(n) și, prin urmare, le vom presupune ca fiind infinit. Aceasta este valoarea costului întregului labirint, unde costul fiecărei celule este afișat ca două valori g(n)+h(n):



Acum vom explora celulele vecine ale celulei curente. Există doar o celulă vecină (3,4) a celulei curente și vom calcula costul acestei celule. g(n) al celulei (3,4) va fi 1, deoarece avem nevoie de un pas pentru a ajunge la celula (3,4) din celula de pornire și h(n) este 5.

Suntem încă în celula de start. Acum algoritmul A\* va selecta celula cu costul minim, pentru următoarea mișcare care pentru acest caz este celula (3,4) și deci se va muta acolo.

Există 3 vecini ai celulei (3,4) și costul acestora va fi calculate.

Noul cost al celulelor (4,4) este 8, ceea ce este mai mare decât vechiul cost de 6 și, prin urmare, nu îl vom actualiza.

Înseamnă pur și simplu că nu vrem mișcarea de la celulă (4,4) la celulă (3,4) și apoi de la celula (3,4) înapoi la celulă (4,4). Costul celorlalte două celule, (2,4) și (3,3) este mai bun decât infinitul și, prin urmare, le vom actualiza costul și nu vom actualiza costul celulei (4,4).

Suntem pe celula (3,4) și următoarea celulă va fi aleasă ca fiind cea care are costul minim. Nu vom lua în considerare celula (4,4) deoarece a fost deja vizitată și costul ei nu a fost actualizat după aceea. Aceasta este indicată ca o bară de culoare galbenă în interiorul celulei.

Dintre celelalte celule, cele două celule, (3,3) și (2,4) au cel mai mic cost de 6. Acum pe care să o selectați? În astfel de scenarii, ar trebui să preferăm să alegem celula cu un cost euristic mai mic, deoarece indică faptul că celula este mai aproape de celula obiectiv. În cazul de față, ambele celule au același cost euristic de 4 și, prin urmare, putem alege oricare dintre cele două și să presupunem că alegem celula (2,4). Ne vom muta acolo și aceasta va fi starea actualizată a Labirintului.

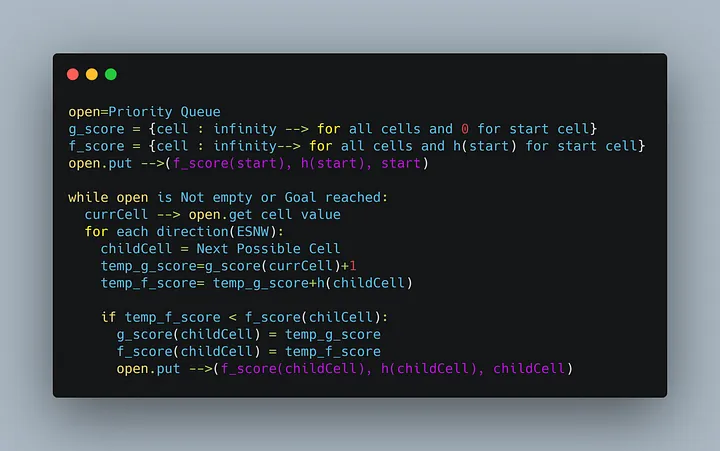
Procesul de calcul al costului celulelor vecine și apoi de alegere a celulei cu cost minim va continua până ajungem la celula obiectiv.

Odată ce ajungem la celula obiectiv, există o modalitate prin care putem selecta doar calea evidențiată, care este cea mai scurtă cale de la celula de pornire la celula obiectiv.

Acum să vedem care poate fi pseudocodul căutării A\*.

Deoarece trebuie să alegem celula cu cost minim, vom folosi Coada de prioritate a structurii de date pentru a implementa algoritmul A\*. Spre deosebire de Queue care funcționează pe principiul FIFO (First In First Out), elementele dintr-o Priority Queue sunt scoase pe baza priorității. Prioritatea poate fi valoarea elementului (cel mai mare sau cel mai mic). În Python, avem Coada de prioritate disponibilă în modulul Coadă de așteptare și prioritatea este cea mai mică valoare și, prin urmare, este cea mai potrivită pentru implementarea A\*.

Deci acesta este pseudocodul cautarii A\*:



Punctul important de remarcat în pseudocodul de mai sus este modul în care stocăm informațiile despre cost și celula în Coada de prioritate. Este stocat ca tuplu (f\_score(start), h(start), start). Tuplurile sunt comparate pe baza primului element din interiorul lor și, dacă acesta este același, comparația se face pe baza elementului următor și așa mai departe. Prin urmare, prima valoare stocată este costul celulei și a doua este costul euristic al celulei, astfel încât, dacă costul a două sau mai multe celule este același, comparația se va face pe costul euristic. A treia este valoarea celulei în sine.