Facultatea de Matematică și Informatică,

Universitatea din București

*Lucrare de Licență*

Aplicații ale algoritmilor euristici în parcurgeri de drumuri minime

ÎNDRUMĂTOR ȘTIINȚIFIC

***Lect. Dr. Marinescu-Ghemeci Ruxandra***

ABSOLVENT

***Munteanu Adrian***

București

2015

Cuprins

[Introducere 3](#_Toc420441255)

[Capitolul I. HTML5, ECMAScript 6 și WebGL 4](#_Toc420441256)

[1.1. HTML5 4](#_Toc420441257)

[1.2. ECMAScript 6 4](#_Toc420441258)

[1.3. WebGL 4](#_Toc420441259)

[1.4. Three.JS 5](#_Toc420441260)

[1.5. Browserul Google Chrome 5](#_Toc420441261)

[Capitolul II. Modelul matematic al problemei și soluțiilor 6](#_Toc420441262)

[2.1. Ipoteza problemei 6](#_Toc420441263)

[2.2. Soluții 6](#_Toc420441264)

[2.2.1. Căutarea în lățime 7](#_Toc420441265)

[2.2.2 Dijkstra 10](#_Toc420441266)

[2.2.3. A\* 11](#_Toc420441267)

[2.2.4. A\* ponderat (Weighted A\*) 14](#_Toc420441268)

[2.2.5. Jump Point Search (JPS) 15](#_Toc420441269)

[2.2.6. A\* ierarhic (Hierarchical A\* / HGA\*) 16](#_Toc420441270)

[2.2.7. Lifelong planning A\* (LPA\*) 16](#_Toc420441271)

[2.2.8. D\* 18](#_Toc420441272)

[2.3. Structuri de date necesare 20](#_Toc420441273)

[Capitolul III. Aplicație 23](#_Toc420441274)

[3.1. Librării și instrumente 23](#_Toc420441275)

[3.2. Detalii de implementare 27](#_Toc420441276)

[3.3. Testare 30](#_Toc420441277)

[3.4. Rezultate 30](#_Toc420441278)

# Introducere

Eficiența și optimizarea algoritmilor reprezintă o continuă cercetare în domeniul informaticii. Aceste îmbunătățiri implementate atât hardware cât și software pot aduce o diferență majoră în domeniul roboticii.

Scopul acestei lucrări este de a expune vizual comparația diferiților algoritmi utilizați în programarea roboților pentru a se deplasa într-un teren parțial cunoscut.

# Capitolul I. HTML5, ECMAScript 6 și WebGL

Începând cu anul 2011, când a fost propusă o variantă finală pentru HTML5 de către grupul W3C, s-a putut observa o mișcare în vederea standardizării a tot mai multe funcționalități pentru navigatoare de internet. De asemenea, aceste noi funcționalități sunt însoțite de îmbunătățiri la nivelul browser script.

Tot în anul 2011 a fost lansată versiunea 5.1 a ECMAScript, venind ca o completare la HTML5. WebGL, sau Web Graphics Library, a fost de asemenea introdus, iar la câteva luni avea sa apară librăria Three.js pentru a facilita dezvoltarea aplicațiilor 3D care rulează direct în browser.

## 1.1. HTML5

Pachetul de îmbunătățiri cu care vine HTML5 conține noul element **canvas**. Acest element reprezintă o porțiune dreptunghiulară în pagina web unde poate se poate „desena”. Acest procedeu de a desena poate fi realizat folosind funcționalități noi de programare din JavaScript ce permit accesarea conținutului zonei de canvas și modificarea acestuia.

## 1.2. ECMAScript 6

ECMAScript este un limbaj de scripting standardizat și este folosit la scară largă pentru scripturi la nivel de client de aplicație. El stă la baza limbajului JavaScript, ceea ce face compatibilă utilizarea lui în diferite implementări de browser web.

Versiunea 6 a ECMAScript aduce ca elemente de noutate, printre altele, structuri noi de date, cum ar fi **Set** (mulțime de elemente) și **Map** (hartă cheie/valoare), și **iteratori** și **for…of**. Având aceste instrumente la dispoziție este mai ușor sa menținem, de exemplu, lista de vecini ai unui nod.

Exemplu:

|  |  |  |
| --- | --- | --- |
| ECMAScript 5 | ECMAScript 6 |  |
| var obj = {};  obj[x] = true;  delete obj[x];  if(x in obj){ … } | var obj = new Set();  obj.add(x);  obj.delete(x);  if(obj.has(x)){ … } | //se creaza un obiect  //se seteaza o valoare  //se sterge o valoare  //se verifica existenta unei valori |

## 1.3. WebGL

WebGL este o specificație de JavaScript menită să îmbunătățească experiența de navigare pe web venind cu grafică 3D și 2D la dispoziția browser-ului web fără a fi nevoie de vreun program adițional. Astfel, conținutul unui site web poate sa folosească placa video a calculatorului pentru cea mai bună performanță. Această librărie este bazată pe librăria multi-platformă OpenGL.

## 1.4. Three.JS

Three.JS este o librărie scrisă în JavaScript ce ajută la implementarea aplicațiilor 3D și oferă, dar nu se rezumă la următoarele facilități:

* Motoare de randare grafica: WebGL, <canvas>, <svg>, CSS3D, DOM, Software
* Scene: pentru a adăuga și elimina obiecte în timpul rulării
* Camere: de perspectivă și ortografică
* Animații
* Lumini: de ambient, direcționale, punctiforme; umbre
* Materiale: Lambert, Phong, cu texturi și umbrire netedă
* Obiecte: rețea, particule, sprites, lumini
* Geometrii: plan, cub, sferă, 3D text; modificatori: alungire, extrudare și tăiere
* Funcții matematice cum ar fi manipulări de matrice, cuaternioni, UV

Ilustrarea unei simulări 3D poate fi realizată direct pe orice navigator compatibil și poate fi la fel de performantă ca o aplicație ce rulează doar pe anumite sisteme de operare sau dispozitive. De exemplu, pentru a crea spațiul geometric necesar va fi nevoie doar de o geometrie de tip plan, având un material și opțional o textură. Având acces direct în JavaScript la aceste facilități putem apoi folosi evenimentele de mouse și tastatură pentru a manipula scena și obiectele din ea.

var geometry = new THREE.PlaneGeometry(width, height, widthSegments, heightSegments);

var material = new THREE.MeshBasicMaterial( {color: 0xffff00} );

var plane = new THREE.Mesh( geometry, material );

scene.add( plane );

## 1.5. Browserul Google Chrome

Pentru a putea beneficia de cele mai noi tehnologii în domeniul web se poate folosi browserul Google Chrome, ce are implementate multe dintre funcționalitățile încă în proces de standardizare. Acesta suportă HTML5 și WebGL și o bună parte din ECMAScript 6.

# Capitolul II. Modelul matematic al problemei și soluțiilor

Problema studiată în acestă aplicație este găsirea celui mai scurt drum între două puncte în teren parțial cunoscut. Această problemă este echivalentă în practică în domeniul roboticii cu problema automatizării deplasării unui robot spre un punct destinație.

## 2.1. Ipoteza problemei

Fie un robot cu instrumente pentru a primi următoarele date: poziția curentă, poziția destinație și mulțimea de muchii aflate în spațiul vizibil al robotului.

Definim în continuare harta pe care se află robotul ca o mulțime de puncte în spațiul geometric euclidian tridimensional, astfel un punct având cele 3 coordonate: x, ,y, z. Un **obstacol** este tradus ca o muchie de cost ∞.

Spunem că o muchie se află în spațiul vizibil al robotului dacă ambele capete se află la o distanță mai mică sau egală cu o **rază de viziune** fixată. Distanța dintre două puncte **p** și **q** este distanța euclidiană .

În figura *Fig.1* este marcată poziția curentă a doi roboți **(1)**, sfera de viziune **(2)**, mulțimea de muchii din spațiul vizibil **(3)**, marcate cu albastru, și poziția destinație **(4)**.

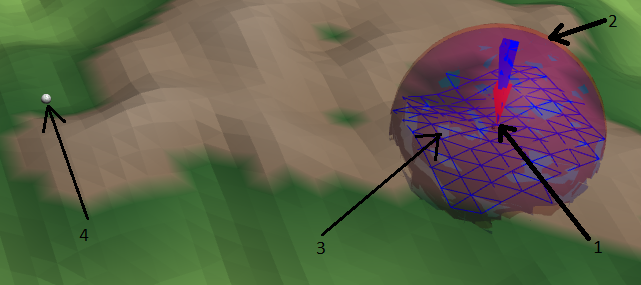


Figura 1. Ilustrare – Sursă prelucrare proprie

Se poate observa faptul că în afara sferei de viziune a roboților nu este marcată nicio muchie, deoarece terenul nu este cunoscut. Din această cauză algoritmii folosiți trebuie să gasească cel mai scurt drum ținând cont de incompletitudinea informației.

## 2.2. Soluții

Pentru a rezolva problema celui mai scurt drum putem implementa diverși algoritmi, cum ar fi căutarea în lățime, Dijkstra, A\*, D\* și altele. Însă pentru a putea aplica acești algoritmi este necesar să existe cel puțin un lanț între cele două puncte, ceea ce nu se întâmplă frecvent pentru configurația aleasă. De aceea, introducem noțiunea de **nod vizibil complet**, desemnând un nod pentru care robotul despre care vorbim nu cunoaște toți vecinii. Simplificăm și mai mult situația în continuare fixând lungimea maximă a unei muchii cu o valoare strict mai mică decât raza de viziune a robotului (roboților).

Combinând aceste modificări, observăm că un **nod vizibil complet** este unul care satisface condiția , unde *viz*este raza de viziune a robotului, *R* este poziția curentă a robotului și *n* este poziția nodului. Spunem astfel că un nod este **vizibil parțial** dacă nu este vizibil complet.

Astfel, pentru cazul în care nu există un drum în graful cunoscut al robotului este suficient să căutăm cel mai scurt drum spre nodul vizibil parțial cu cea mai mică **distanță estimată**. Pentru acest scop este suficient să menținem o listă ordonată a nodurilor vizibile parțial, realizabilă cu o coadă de priorități sau echivalent cu algoritmi de sortare. De asemenea, se poate întampla și ca „drumul cunoscut sa *nu* fie cel mai scurt”, existând posibilitatea ca unele noduri vizibile parțial să conducă pe un drum mai scurt.

Definim în continuare **distanța estimată** *h(x) = d(x, s)*, unde *s* este poziția destinație. Această distanță este astfel o euristică admisibilă, deoarece nu supraestimează costul și consistentă, pentru că îndeplinește condiția pentru orice *x* și *y* noduri adiacente.

### 2.2.1. Căutarea în lățime

Având în vedere faptul că robotul se află într-un teren necunoscut, această căutare va fi efectuată după fiecare actualizare a cunoștințelor robotului. Astfel, la primul traseu parcurs, algoritmul va căuta un drum minim înainte de fiecare mișcare.

Pseudocod:

Fie un graf

Fie nodul inițial și nodul destinație

Fie o coadă

Marchează ca vizitat

Cât timp execută

Dacă atunci execută *DrumInapoi* și STOP

Pentru toate muchiile de la la din execută

Dacă nu este marcat ca vizitat atunci

Marchează ca vizitat

Setează

Sfârșit – dacă

Sfârșit – pentru

Sfârșit – cât timp

Returnează eroare (nu există drum)

Procedura *DrumInapoi*:

Fie lista nodurilor drumului

Cât timp execută

Sfârșit – cât timp

Returnează

Această metodă garantează găsirea drumului cu cele mai puține noduri între cele două puncte, deoarece la fiecare pas explorează un nivel întreg de adâncime. Demonstrația se poate face prin inducție după cum urmează.

Notăm cu *n*  numărul de vârfuri în graful *G*. Căutarea începe cu vârful , ce devine astfel rădăcina arborelui de parcurgere. Notăm cu *l* numărul de vârfuri în care se poate ajunge din . Definim pentru un vârf următoarele:

= nivelul lui în arborele de parcurgere

= distanța de la la *v* în graf

= numărul de ordine *i* al *v*, unde , și *v* este al *i*-lea vârf adăugat în coadă

Vrem să demonstrăm că pentru orice *v*. Vom recurge la inducție după , mai exact vom arăta pentru că pentru *v* cu avem

(**i1**) și

(**i2**) pentru orice vârf *w*, dacă , atunci

Cazul *i* = 1 este trivial, având evident introdus primul în coadă, deci și nu există noduri *w* cu .

Presupunem ipoteza de inducție adevărată pentru și demonstrăm pentru *i*.

Pentru a demonstra (**i1**), fie și părintele lui *v* în arborele de parcurgere. Presupunem prin reducere la absurd că există un drum de lungime . Avem deci

, (1)

din moment ce *v* este adăugat în coadă când este scos, și

, (2)

deoarece din (1) putem aplica ipoteza de inducție (i1) la , obținând astfel

.

În plus avem că

, (3)

pentru folosind (1) putem aplica ipoteza de inducție (i2) în și avem (2).

Dar, scoțându-l pe *w* din coadă și existând muchia , atunci algoritmul l-ar fi marcat ca vizitat pe *v* la acel moment, dacă nu ar fi fost deja. Astfel, părintele lui *v* are numărul de ordine cel mult , care din (3) este strict mai mic decât , și deci nu poate fii părintele lui *v*, având contradicție cu presupunerea.

Demonstrație pentru (**i2**): Fie și

. (4)

Presupunem prin reducere la absurd că . Din (4) avem că implică , deci

. (5)

Fie părintele lui *v* in arborele de parcurgere și fie un drum minim de la la . Asta implică faptul că este un drum minim de la la . Concluzionăm că

. (6)

Ca în demonstrația propoziției (**i2**), avem

. (7)

Din ipoteza (**i1**) în drumul dat de parcurgere este drum minim, deci și este drum minim. Din aceasta rezultă că

. (8)

Din (4), (6) și (8) implică

. (9)

Aplicând ipoteza de inducție (**i2**) în , care din (7) este mai mic decât , și împreună cu (9), obținem

. (10)

La extragerea lui din coadă a fost adăugat *v*, iar la extragerea lui a fost adăugat w în coadă sau era deja. Din (10) deducem ca fiind inserat înaintea lui și *w* a fost inserat înaintea lui *v*, ceea ce contrazice presupunerea (5).

Această metodă, deși găsește drumul cu cele mai puține noduri între cele două puncte, nu ține cont de costul muchiilor, ceea ce nu rezolvă problema. Încercăm în continuare să rezolvăm această problemă pornind de la = distanța de la la *v* în graf. Observăm că această distanță este pentru muchii cu cost = 1 chiar = nivelul lui în arborele de parcurgere. Încercăm deci să introducem explicit în coadă odată cu *v* pentru a parcurge în ordinea costului. Acest algoritm este chiar algoritmul lui Dijkstra și il vom studia în cele ce urmează.

### 2.2.2 Dijkstra

Algoritmul lui Dijkstra pentru găsirea drumului de cost minim pornește cu presupunerea că nu există un drum minim către niciun nod de la , asociându-le astfel valoarea ∞ tuturor nodurilor. Apoi, introduce toate nodurile in coadă și, asemenea algoritmului anterior, parcurge nodurile din coadă și le procesează, însă în ordinea distanței asociate.

Pseudocod:

Fie un graf

Fie nodul inițial și nodul destinație

Fie o coadă

Pentru vârf

Dacă atunci

Sfârșit – dacă

Sfârșit – pentru

Cât timp execută

Pentru toate muchiile de la la din execută

Dacă atunci

Sfârșit – dacă

Sfârșit – pentru

Sfârșit – cât timp

Dacă atunci

Returnează eroare (nu există drum)

Altfel

Execută *DrumInapoi*

Sfârșit – dacă

Demonstrație:

Definim costul drumului cu funcția și drumul minim între u și v cu funcția

Observăm că un drum *p* între *u* și *v* este minim dacă .

Vom spune că un vârf este vizitat dacă nu se află în coadă și presupunem că *u* este primul vârf vizitat pentru care și observăm că:

* u nu poate fii , deoarece
* trebuie sa existe un drum de la la *u*, pentru că *u* a fost vizitat
* din moment ce există cel puțin un drum, cel puțin unul este minim

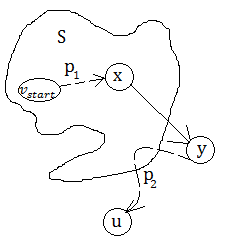
Fie drumul minim de la la *u*, unde *x* este vizitat și *y* este primul nod nevizitat. La momentul vizitării lui *x* aveam , deoarece am presupus ca *u* este primul pentru care nu este îndeplinită egalitatea. Totodată muchia (*x, y*) era relaxată – adică se poate găsi un drum mai scurt decât cel existent spre *y* trecând prin *x* - ceea ce rezultă în:

Figura 2. Ilustrare – Sursă prelucrare proprie

Și *y* și *u* erau nevizitate la momentul algerii lui *u*, deci , ceea ce rezultă că

Deci contrazicând astfel presupunerea făcută, demonstrând astfel că pentru orice nod vizitat *u* a fost găsit drumul minim . Observăm că acest algoritm determină cel mai scurt drum de la un nod inițial către toate celelalte, ceea ce este mai mult decât cerința problemei și conform acestei demonstrații putem deci să oprim algoritmul de îndată ce nodul a fost vizitat, i.e. – a fost scos din coadă.

Algoritmul lui Dijkstra are o complexitate timp de dacă implementarea cozii de prioritate este făcută cu o simplă listă înlănțuită sau un vector, necesitând astfel parcurgerea completă a cozii la fiecare extragere de minim. Însă, dacă folosim Fibonacci Heap descris la (2.3) putem obține o complexitate de fiind astfel un algoritm semnificativ mai eficient decât căutarea în lățime.

Pentru a îmbunătăți acest algoritm ne putem folosi de distanța estimată, introdusă anterior la (2.2.) pentru a alege vârful intermediar, de această dată însă în scopul explorării drumurilor celor mai promițătoare spre nodul destinație.

### 2.2.3. A\*

Deși este în continuare necesar să efectuăm o căutare completă la orice actualizare, de această dată algoritmul A\* va explora semnificativ mai puține noduri. Acest efect este dat de faptul că algoritmul A\* prioritizează nodurile care se „apropie” de soluție, în sensul că deși costul de a ajunge în aceste noduri este mai mare, distanța estimată este mai mică.

Definim astfel funcția de prioritizare , unde este costul de la vârful de start la *n* și este distanța estimată de la *n* la vârful destinație. Introducem astfel la nivel de nod informația funcției *f* adițional la distanța drumului minim ce trece prin vârful predecesor. Astfel, redenumim din algoritmul lui Dijkstra în pentru orice . Metoda *scoateMin* a cozii de prioritate va face astfel comparațiile pe baza valorii *f* a vârfurilor. Pentru a explora maxim o dată fiecare vârf vom introduce și un marcator *vizitat* la nivel de nod cu valoarea inițială *fals*.

Pseudocod:

Fie un graf

Fie nodul inițial și nodul destinație

Fie o coadă

Cât timp execută

Dacă atunci execută *DrumInapoi* și STOP

Pentru toate muchiile de la la din execută

Dacă atunci treci la următorul

Dacă sau atunci

Dacă atunci

Sfârșit – dacă

Sfârșit – pentru

Sfârșit – cât timp

Returnează eroare (nu există drum)

Observăm că și algoritmului A\* garantează găsirea unui drum minim dacă urmatoarele condiții sunt îndeplinite pentru *h*:

* *h* este optimistă, adică nu supraestimează niciodată drumul rămas de parcurs:
* *h* este consistentă:

Demonstrație

Lemă: Orice drum inclus într-un drum minim este de asemenea minim.

Fie un drum minim între și și pentru orice *i* și *j* cu

, fie sub-traseul din *p* de la vârful *i* la vârful *j*. Atunci drumul este un drum minim de la la . Demonstrație: descompunem drumul *p* în

, și avem că

În continuare, presupunem că există un drum de la la de cost . Atunci este un drum de la la al cărui cost

este mai mic decât , ceea ce contrazice presupunerea că *p* este drum minim de la la .

Folosind această lemă putem în continuare să demonstrăm faptul că algoritmul A\* găsește cel mai scurt drum, acesta construind drumuri minime asemenea lui Dijkstra, dar într-o ordine mai promițătoare.

Presupunem prin reducere la absurd că algoritmul găsește un drum care nu e optim, astfel că și deci trebuie să existe un nod *n* care este neexplorat de algoritm și care face parte dintr-un drum minim. Adică drum minim, ceea ce implică drum minim de cost datorită lemei demonstrate anterior, și avem că , deoarece algoritmul nu s-ar fi oprit altfel.

De asemenea mai avem și despre care știm că este mai mică sau egală cu , deoarece *h* este admisibilă.

Rezultă astfel că , ceea ce contrazice presupunerea făcută. Astfel că .

Pentru cazul de față în care știm că este optimistă, deoarece pentru orice muchie avem și pentru orice 3 puncte *a, b, c* din spațiul geometric euclidian avem că , astfel că dacă nu există muchie de la *v* la având costul atunci orice drum minim *p* de la *v* la are costul total . Tot de aici rezultă și faptul că *h* este consistentă.

Deoarece euristica *h* trebuie să fie optimistă pentru ca algoritmul A\* să determine soluția optimă, acest algoritm are și dezavantaje. Pentru cazul în care destinația se află „în spatele” unui obstacol, acest algoritm se aseamănă cu algoritmul lui Dijkstra, având nevoie de timp și memorie. Cazul cel mai nefavorabil dă algoritmului o complexitate , ceea ce în aplicații practice poate constitui o problemă, și sunt luate în calcul variante de compromis, cerința transformându-se în găsirea unui drum cât mai bun într-un timp fixat.

Algoritmul A\* poate fii modificat astfel încât să prioritizeze mai mult după distanța estimată a vârfurilor, însă nu fără să țină cont deloc de costul până în acel nod.

### 2.2.4. A\* ponderat (Weighted A\*)

A\* ponderat aduce o modificare minoră în schema algoritmului A\* introducând o relaxare a criteriului de admisibilitate. Funcția euristică *h* se va înmulți cu un număr fixat ce reprezintă înclinarea către nodurile mai apropiate de destinație.

Definim în continuare pentru ușurință , astfel funcția pe baza căreia se face comparația nodurilor devine

Pseudocodul rămâne același de la algoritmul A\*, cu mica modificare la calcularea funcției *f*: .

Din cauza ușoarei modifcări a ordinei de explorare acest algoritm nu găsește soluția optimă. Se numește că euristica folosită este -consistentă dacă

și , pentru orice *u* și *v* cu muchie și și . Astfel, algoritmul A\* ponderat este -sub-optim, adică

, unde *p* este soluția găsită de algoritm. Cu alte cuvinte, este garantat că soluția găsită de A\* ponderat va avea un cost de maxim ori mai mare decât costul soluției optime.

Știind că *h* este optimistă avem că ceea ce rezultă că și . Adunând funcția *g*, rezultă că

Din *h* este consistentă: rezultă că și .

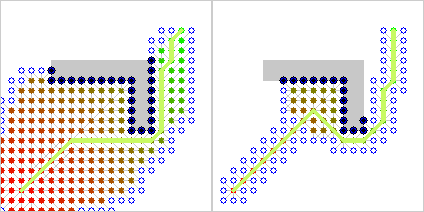
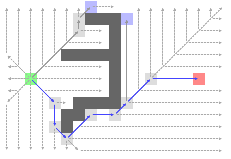
 În figura alăturată[[1]](#footnote-1) avem algoritmul A\* original în partea stângă și algoritmul A\* cu pondere în partea dreaptă. Observăm că parcurgerea cu pondere găsește mai rapid un drum între cele două puncte, fiind relativ scurt, dar nu optim. Sunt marcate cu roșu spre verde vârfurile explorate, culoarea reprezentând valoarea dată de funcția *f*. Vârfurile marcate cu albastru sunt inaccesibile, iar cele fară culoare sunt în coada de priorități.

Figura 3 – sursa Wikipedia

### 2.2.5. Jump Point Search (JPS)

Jump Point Search este o altă metodă de a optimiza algoritmul A\* mai ales pentru căutarea într-o matrice. Prin acest procedeu sunt eliminați vecinii candidați în care se poate ajunge din pasul anterior sau cu un cost echivalent, determinând astfel puncte prin care trebuie sa treacă drumul minim, zise puncte de salt (jump points).

 După cum se poate observa în figura alăturată[[2]](#footnote-2) acest algoritm elimină simetriile, considerând în cazul de față drumuri care sunt unice, abstracție făcând de un izomorfism pe baza punctului de plecare, punctului de sosire și eventual costului. Liniile punctate reprezintă evident drumuri excluse din parcurge datorită simetriei, liniile continue fiind singurele drumuri relevante.

Pseudocod:

Figura 4. Sursa Witmer Nathan

Fie un graf

Fie nodul inițial și nodul destinație

Fie o coadă

Cât timp execută

Dacă atunci execută *DrumInapoi* și STOP

Pentru toate muchiile de la la din execută

Dacă atunci treci la următorul

Dacă sau atunci

Dacă n atunci

Sfârșit – dacă

Sfârșit – pentru

Sfârșit – cât timp

Returnează eroare (nu există drum)

Funcția

Funcția , unde *v* este vârful de la poziția

Funcția este

Dacă *n* e obstacol sau în afara terenului atunci

Returnează null

Sfârșit – dacă

Dacă atunci

Returnează *n*

Sfârșit – dacă

Dacă există *n*' în vecini(n) a.i. *n*' este forțat atunci

Returnează n

Sfârșit – dacă

Dacă *d* este diagonală atunci

Dacă atunci

Returnează *n*

Sfârșit – dacă

Dacă atunci

Returnează *n*

Sfârșit – dacă

Returnează

Sfârșit – funcție

Această metodă găsește soluția optimă reducând timpul de rulare cu până la un ordin de magnitudine. Deși varianta prezentată este pentru rezolvarea problemei drumului minim într-o matrice, Jump Point Search poate fi extrapolată și pentru utilizarea în trei dimensiuni. Cu toate acestea însă, pentru un graf în care costurile muchiilor sunt foarte variate devine practic totuna cu algoritmul A\*, nefiind simetrii de eliminat.

### 2.2.6. A\* ierarhic (Hierarchical A\* / HGA\*)

Un alt algoritm pentru determinarea unui drum între două puncte este HGA\*. Acesta însă construiește o soluție aproximată pentru a reduce timpul de calcul. Idea de bază a algoritmului este găsirea de rute ocolitoare pentru obstacolele aflate pe traiectoria de mers, traiectoria inițială fiind între punctul de start și cel de sosire. Astfel, algoritmul caută cea mai apropiată cale liberă pornind de la intersecția cu primul obstacol și se construiesc recursiv astfel rute ocolitoare ce pot fi ordonate după cost.

### 2.2.7. Lifelong planning A\* (LPA\*)

Lifelong planning A\* sau A\* cu planificare de lungă durată este un algoritm bazat pe A\* care, după cum îi spune numele, caută drumul minim păstrând informații care să facă usoară transformarea drumului găsit într-unul minim după (re)actualizarea costurilor muchiilor. Astfel, algoritmul LPA\* introduce notația „right-hand side value” – valoarea optimă pentru un vârf.

unde *Pred(s)* este mulțimea vârfurilor predecesori lui *s*

Cheia pentru coada de priorități o formăm din două valori: , unde și , comparația cheilor facându-se în ordine lexicografică. Datorită alegerii cheii în acest fel, prioritatea nodurilor din coadă va fi dată de cel mai mic cost pentru a ajunge în nod și euristica din acel nod, spărgând egalitățile in favoarea costului cel mai mic spre un nod.

Algoritmul va inițializa valorile *g* și *rhs* cu ∞ pentru toate nodurile, excepție făcând , care va avea 0, fiind nodul de la care începe parcurgerea. Actualizarea costurilor muchiilor va presupune actualizarea valorii *rhs* a nodurilor afectate și reintroducerea acestora în coada de priorități cu noile chei.

Pseudocod:

Funcția *calculeazaCheie*(*u*) este

Returnează

Sfârșit – funcție

Funcția *initializare* este

Fie coada de priorități

Pentru execută

Sfârșit – pentru

U.adauga()

Sfârșit – funcție

Funcția *actualizareVarf*(*u*) este

Dacă atunci

Sfârșit – dacă

Dacă atunci

Sfârșit – dacă

Dacă atunci

Sfârșit – dacă

Sfârșit – funcție

Funcția *calculeazaDrumMinim* este

Cât timp sau execută

()

Dacă atunci

Pentru execută

Altfel

Pentru execută

Sfârșit – dacă

Sfârșit – cât timp

Sfârșit – funcție

Funcția *main* este

*initializare()*

Cât timp execută

*calculeazaDrumMinim*()

Deplasează robotul conform drumului găsit și așteaptă modificări de costuri de muchii

Pentru toate muchiile cu cost modificat execută

Sfârșit – pentru

Sfârșit – cât timp

Sfârșit – funcție

În pseudocodul de mai sus, funcția *main* este cea apelată de robot, iar robotul va efectua traseul dat de drumul găsit de algoritm. Când apar modificări la costuri de muchii, robotul apelează în continuare algoritmul pentru a replanifica traseul dacă este cazul. De menționat este și faptul că acest algoritm găsește drumul minim între vârful de start și vârful destinație la fiecare pas, ceea ce din cauza modificărilor apărute pe parcurs este posibil să nu mai fie soluție optimă între vârful curent în care se află robotul și vârful destinație.

### 2.2.8. D\*

Pornind de la algoritmul LPA\* putem optimiza rularea lui inversând parcurgerea, și anume de la la , de această dată modificându-se odată cu poziția curentă a robotului. În acest fel algoritmul D\* devine direcționat de scop, adică sosirea în nodul destinație cât mai repede.

Spunem că un vârf *s* este **consistent local** daca *g(s) = rhs(s)*, altfel este **inconsistent local**. Dacă toate vârfurile sunt consistente local atunci valoarea *g* a lor este egală cu distanța respectivă de la start, ceea ce permite găsirea drumului minim de la start la orice vârf. Totuși, vom folosi euristica *h* pentru a focusa căutarea spre destinație. Observăm că în coada de priorități va fi suficient să reținem vârfurile inconsistente local, deoarece sunt inițializate valorile *g* și *rhs* cu ∞ pentru toate vârfurile din graf, cu excepția celui de start, și sunt modificate aceste valori doar la analizarea vecinilor nodului explorat.

Parcurgerea „de la coadă la cap” necesită analizarea predecesorilor nodului explorat, și nu a succesorilor, așa cum se întâmpla în algoritmul LPA\*. Tot din această cauză euristica aleasă trebuie să fie modificată astfel încât să calculeze drumul estimat până la un anume nod, și astfel să fie consistentă înapoi. Noile condiții pentru euristica *h* sunt:

* , pentru orice vârf și

Deoarece nodul de start va fi modificat, aceste condiții pentru *h* trebuie îndeplinite pentru orice vârf.

Pseudocod:

Funcția *calculeazaCheie*(*u*) este

Returnează

Sfârșit – funcție

Funcția *initializare* este

Fie coada de priorități

Pentru execută

Sfârșit – pentru

Sfârșit – funcție

Funcția *actualizareVarf*(*u*) este

Dacă atunci

Sfârșit – dacă

Dacă atunci

Sfârșit – dacă

Dacă atunci

Sfârșit – dacă

Sfârșit – funcție

Funcția *calculeazaDrumMinim* este

Cât timp sau execută

()

Dacă atunci

Pentru execută

Altfel

Pentru execută

Sfârșit – dacă

Sfârșit – cât timp

Sfârșit – funcție

Funcția *main* este

*initializare()*

*calculeazaDrumMinim*()

Cât timp execută

Deplasează robotul la și caută muchii cu costul modificat

Pentru toate muchiile cu cost modificat execută

Sfârșit – pentru

Sfârșit – cât timp

Sfârșit – funcție

Principalul avantaj al acestui algoritm față de LPA\* îl constituie faptul că, deși drumul optim se poate modifica pe parcursul deplasării, vârful destinație rămâne același, iar găsirea unui drum de lungime minimă se transformă în găsirea nodului de la care drumul curent este minim și continuarea căutării de la cei mai buni candidați. Astfel, chiar dacă robotul se mișcă, schimbându-și vârful de start, drumul minim nu trebuie recalculat de fiecare dată complet.

## 2.3. Structuri de date necesare

Eficiența structurilor de date folosite stă la baza algoritmilor eficienți. În cazul de față coada de priorități este una dintre ele. O implementare simplistă și ineficientă este, de exemplu, utilizarea unui vector, iar la fiecare extragere de minim se va face o parcurgere. Această metodă are o complexitate de .

Printre metodele folosite de obicei se află Binary Heap, fiind un arbore binar complet care satisface proprietatea ca orice nod să fie mai mic sau egal decât copiii lui. Complexitatea operațiilor este pentru inserare, extragere de minim și modificare de prioritate.

Fibonacci Heap este o altă implementare a cozii de priorități, ce are la bază o listă de arbori. Această structură nu se consolidează la inserare, ci doar la extragere. Principiul este de a construi arbori heap de adâncime crescătoare, spre exemplu, pentru n noduri vor exista maxim log(n) arbori în listă, iar arborele va avea maxim noduri dispuse în sub-arbori de adâncimi de la 0 la i-1, pentru orice . Această structură este menținută recursiv. Operația de inserare va presupune simpla adăugare a nodului nou în lista de arbori, reprezentând o rădăcină. În acest caz operația este executată în timp constant – – lăsând reordonarea nodurilor pentru alte operații. Această abordare nu influențează rezultatul, deoarece la extragerea minimului se consolidează structura, operația executându-se în timp amortizat reprezentând adâncimea unui arbore oarecare din listă.

Concret, metoda de inserare în coada de priorități implementată cu Fibonacci Heap are următorul pseudocod:

Funcție este

Dacă atunci

Sfârșit – dacă

Sfârșit – funcție

Notăm în continuare la nivel de structură a cozii de prioritate:

* o listă (dublu înlănțuită) ce conține rădăcinile arborilor ce se vor forma
* (referință către) elementul din structură cu cea mai mică prioritate, despre care observăm că va fii întotdeauna o rădăcină de arbore heap

Descriem în continuare metoda de extragere minim, cea care și consolidează structura astfel încât următoarele operații să devină mai eficiente. Pseudocod:

Funcție este

Pentru execută

Sfârșit – pentru

Fie *k* un vector de adâncimi

Pentru execută

Dacă atunci

Treci la următorul

Sfârșit – dacă

Dacă atunci

Altfel

Sfârșit dacă

Sfârșit – pentru

Sfârșit – funcție

Astfel, pentru a rearanja elementele din structura este suficient să folosim un vector de adâncimi, adică elementul de la indexul *i* va avea adâncimea *i*, deoarece vrem ca structura rezultată în urma consolidării să aiba arbori de adâncimi diferite. Mai mult, folosind un astfel de vector, arborii uniți vor fi de adâncimi egale, ceea ce implică faptul că un arbore de adâncime *i* va avea cel mult un copil de adâncime *i*-1, astfel creându-se o structură simetrică copil-părinte, generând un cost de accesare logaritmic.

Pentru ca structura să fie completă este nevoie și de un mecanism de stergere din coadă, pe care îl vom implementa folosind o metodă mai generală, și anume *scadereCheie* ce va permite modificarea cheii unui element la , astfel devenind minimul pe care să îl putem extrage.

Operația de scădere de prioritate trebuie să taie nodul afectat dacă acesta invalidează regula de heap, adică să aiba cheie mai mare decât părintele său, astfel că după mai multe astfel de operații există posibilitatea ca structura să nu mai respecte regulile descrise inițial și din această cauză să afecteze performanța. Acest impact nefiind observat ca o greșeală, structura funcționând corect în continuare, este posibil să nu fie observat cu ușurință. Pentru a evita situația menționată, este de ajuns să limităm tăiera copiiilor unui părinte la maxim unu, la al doilea tăind și părintele, acesta devenind ineficient. Acest procedeu poate fi implementat cu un simplu marcator la nivel de nod. Acest marcator trebuie însă aplicat recursiv dacă părintele era deja marcat și s-a efectuat tăierea lui. Observăm că o rădăcină nu poate fi marcată din cauză că aceasta nu are un părinte de la care să fie decuplată. Pseudocod:

Funcție este

Dacă atunci

Sfârșit – dacă

Dacă *x* nu are părinte atunci STOP

Dacă atunci STOP

Sfârșit – funcție

Funcție este

Dacă atunci STOP

Dacă atunci

Sfârșit – dacă

Sfârșit – funcție

# Capitolul III. Aplicație

Aplicația are ca scop ilustrarea performanței și optimalității unora dintre algoritmii expuși în capitolul II. Alegerea acestora a fost bazată pe aplicabilitatea lor la problema studiată, întrucât nu avea sens să includem unii algoritmi care nici nu oferă performanță și nici nu găsesc drumul minim. Dezvoltarea și testarea aplicației au fost realizate cu ajutorul unora dintre cele mai noi și eficiente librării și instrumente, drept urmare efortul depus a constat în mare parte în implementarea propriu-zisă a logicii aplicației.

## 3.1. Librării și instrumente

O librărie foarte intens folosită în aplicațiile web este jQuery, aceasta completând limbajul de programare JavaScript, făcând mai ușoare cele mai multe dintre operațiile des utilizate cu browser-ul.

Lista de îmbunătățiri pe care le aduce jQuery include, dar nu se rezumă la:

* Operații cu elementele din document, cum ar fi traversarea arborelui de elemente și acesarea și modificarea multiplă a acestora
* Manipularea evenimentelor de browser
* Animații
* Ajax – un mecanism de comunicare cu alte resurse fără a necesita reîncărcări de pagină sau introducerea unor elemente de tip (i)frame
* Este compatibil cu majoritatea browser-elelor
* Este ușor extensibil

Această librărie este apelată din aplicație pentru a găsi și actualiza anumite elemente din interfață, cum ar fi modificatori de configurație, butoane și afișări de text ori de imagini. Mai multe detalii pot fii găsite pe website-ul oficial al librăriei: <https://jquery.com/>

Tot pentru a ușura folosirea limbajului JavaScript am folosit și librăria Three.JS, care se face mai mult decât utilă în implementarea aplicațiilor 3D. Detalii pot fi găsite la adresa oficială a librăriei: <http://threejs.org/>

Am folosit astfel multe dintre elementele cu care se laudă librăria fără nicio problemă, aceasta fiind foarte bine documentată și intuitivă. Printre funcționalitățile pe care le apelează aplicația se numără: motor de randare grafică, scenă, cameră, controale de mișcare, geometrii planare, sferice și de alte forme cu materialele de rigoare, lumini de diferite tipuri și nu în ultimul rând structuri și operații matematice indispensabile aplicațiilor grafice. La acestea se adaugă cele din extensia ThreeX, cum ar fi generare de matrice simplex cu transformarea acesteia în geometrie. Link: <http://www.threejsgames.com/extensions/>

Astfel, pentru a crea spațiul de afișare este de ajuns următorul cod.

**var *renderer*** = **new *THREE***.WebGLRenderer();

**var *scene*** = **new *THREE***.Scene();

**var *camera***=**new *THREE***.PerspectiveCamera(…);

document.**body**.appendChild(***renderer***.**domElement**);

În continuare construim terenul ce constă într-o geometrie plană:

**var *heightMap*** = ***THREEx***.**Terrain**.allocateHeightMap(***width***, ***depth***);

***THREEx***.**Terrain**.simplexHeightMap(***heightMap***);

**var *geometry*** = **new *THREE***.PlaneGeometry(100, 100, ***width*** - 1, ***depth*** - 1);

***THREEx***.**Terrain**.heightMapToPlaneGeometry(***heightMap***, ***geometry***);  
***THREEx***.**Terrain**.heightMapToVertexColor(***heightMap***, ***geometry***);

**var *material*** = **new *THREE***.MeshLambertMaterial({  
 **shading**: ***THREE***.**FlatShading**,  
 **vertexColors**: ***THREE***.**VertexColors**});  
**var *mesh*** = **new *THREE***.Mesh(***geometry***, ***material***);  
***scene***.add(***mesh***);

Pentru a putea distinge obiectele mai ușor adăugăm și iluminare medie:

**var** light = **new *THREE***.AmbientLight(0x202020);  
***scene***.add(light);light = **new *THREE***.DirectionalLight(**'white'**, 0.8);  
light.position.set(10, 40, 10);  
***scene***.add(light);

Rezultatul ar trebui să fie asemănător cu următoarea figură:

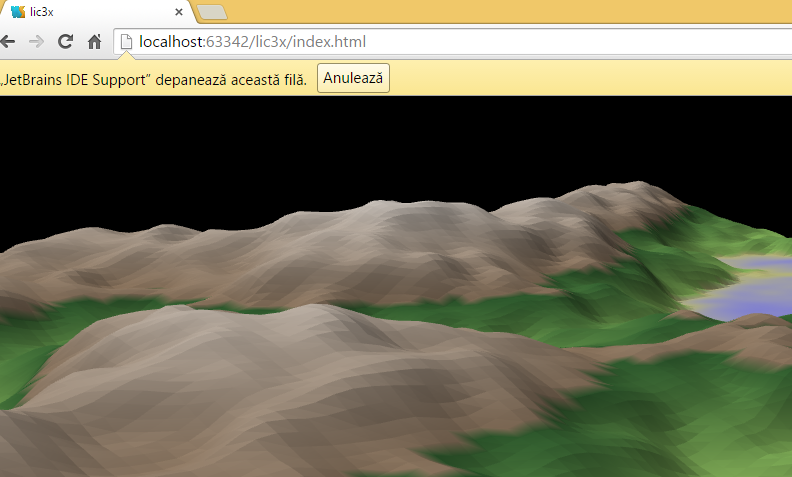


Figura 5. Sursa – prelucrare proprie

Adăugăm în spațiul geometric și două grupuri de obiecte ce vor reprezenta doi roboți cu sfera de viziune a lor și poziția curentă, afișarea muchiilor cunoscute din graf sub formă de linii, și puncte marcatoare de start și destinație. La interfața grafică adăugăm și posibilitatea de a modifica configurația rulării, precum și controlul acesteia și informații despre traseul parcurs de roboți.

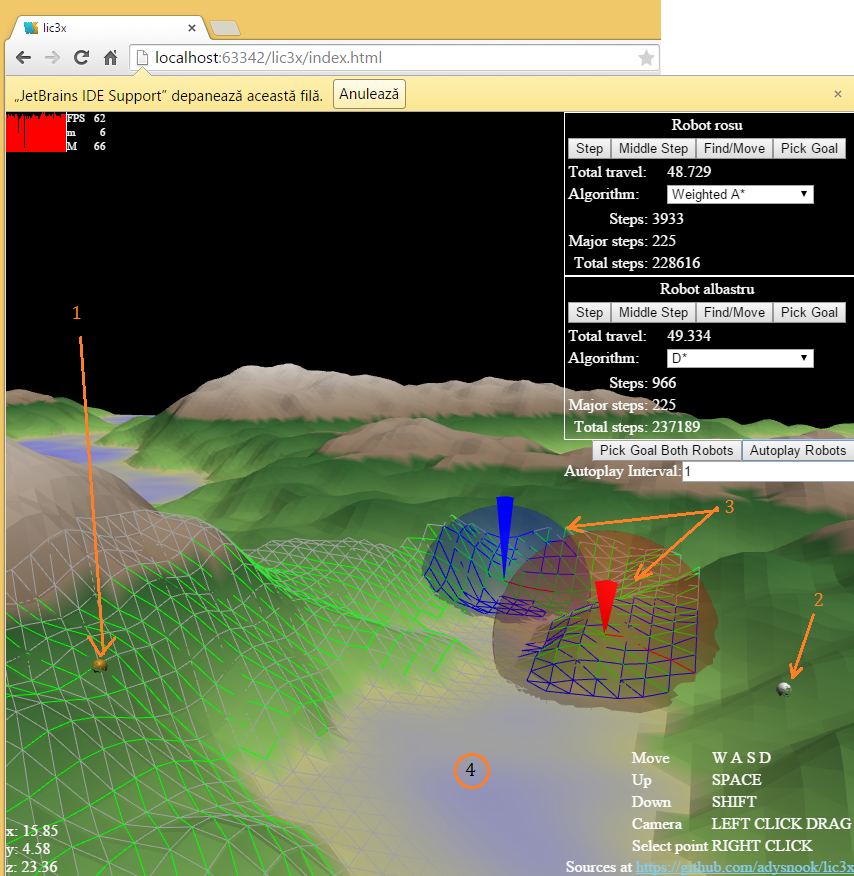


Figura 6. Ilustrarea aplicației – sursă prelucrare proprie

În figura 6 putem observa punctul de plecare marcat cu **1**, punctul de sosire **2**, roboții **3**, și un teren impasabil (un lac) **4**. Cu verde aprins, gri, albastru și roșu sunt marcate muchiile din grafurile cunoscute de către roboți, culorile seminificând:

* Gri = muchie din graf
* Verde = muchie parcursă de algoritm pentru a determina drumul minim
* Albastru = muchii aflate în sfera de viziune și implicit actualizate la mișcarea anterioară
* Roșu = drumul minim determinat de algoritm

Am afișat în partea dreaptă informații despre starea curentă a roboților pentru a scoate în evdiență anumite aspecte:

* Total travel = costul total al traseului parcurs de roboți
* Algorithm = algoritmul folosit pentru ghidare
* Steps = numărul de pași efectuați la ultima căutare
* Major steps = numărul de pași majori efectuați = 2 x numărul de mutări ale robotului
* Total steps = numărul total de pași efectuați de algoritm de la începutul rulării până în momentul curent

Pentru a optimiza rularea aplicației am grupat segmentele de linii pe culori, deoarece transferul între procesor (CPU) și placa video (GPU) se efectuează pentru fiecare geometrie. Astfel, reducând numărul de geometrii sunt decuplate procesarea de afișare, placa video menținându-și buffer-ul mai ușor. Gruparea se mai poate face și folosind alte criterii, cum ar fi factorul de modificare a culorii unei linii, zona în care se află, sau conform modelului geometric creat.

O aplicație web de acest gen poate fi scrisă in orice editor, tot ceea ce este nevoie pentru a rula fiind fișiere text ce conțin codul HTML și JavaScript. Dar de ce să nu ne facem viața mai ușoară folosind instrumente profesionale de dezvoltare web, cum ar fi editorul WebStorm de la JetBrains. Acest editor oferă următoarele facilități:

* Completare automată inteligentă de JavaScript
* Analiză pe calitatea codului în timp real
* Detectarea erorilor chiar de la scriere
* Sugestii de reparare rapidă a problemelor
* Găsirea definiției funcțiilor apelate
* Găsirea folosirilor unei funcții
* Formatare document
* Mutări de cod și „refactoring” inteligente
* Debugger
* Unit testing
* Versiuni multiple ale ECMAScript
* Recunoaște HTML5 și CSS3 și altele
* Integrare cu sisteme de versionare

La fiecare problemă pe care am avut-o pe parcursul dezvoltării aplicației am reușit să ințeleg care era cauza foarte ușor, punând unul sau mai multe breakpoint-uri în cod și analizând stările prin care trecea aplicația. Detalii despre WebStorm pot fi găsite pe pagina aplicației: <https://www.jetbrains.com/webstorm/>

Dacă am șters cod care era bun ce fac? sau dacă a apărut o problemă cu ce am scris azi, cum găsesc ușor toate modificările de la ultima versiune stabilă? pentru a rezolva aceste probleme cât mai inteligent au fost dezvoltate sistemele de versionare, cum ar fi Subversion, Mercurial, CVS, TFS, Git și altele.

Am ales să versionez fișierele folosind Git, deoarece este un sistem distribuit de versionare, adică nu necesită un loc (repository) central pentru a funcționa. Mai mult decât atât este open-source, funcționează pe orice platformă și are o comunitate mare. Poate funcționa cu mai multe repository la distanță (remote). Pagina oficiala git: <https://git-scm.com/>

Mai mult decât atât, am folosit și GitHub, o platformă online ce găzduiește repository de git și are o interfață foarte utilă. Acum am aplicația oriunde, de pe orice fel de dispozitiv, GitHub având direct pe website expuse fișierele. Website-ul GitHub: <https://github.com/>

## 3.2. Detalii de implementare

Algoritmii studiați funcționează folosind anumite structuri de date, cum ar fi grafurile stocate ca mulțime de noduri, mulțime de muchii și mulțime de vecini la nivel de vârf. Trecerea de la vârfurile și muchiile generate de către aplicația 3D presupune parcurgerea fețelor. Acestea sunt triunghiuri generate pornind de la vârfurile din geometrie.

Avem în continuare mai multe grafuri pe care trebuie sa le stocăm: unul pentru terenul real, și câte unu pentru fiecare robot, reprezentând cunoștințele acestora. Deoarece va fi util să identificăm rapid o muchie având capetele, vom stoca vecinii unui nod ca fiind muchiile adiacente acestuia, despre care știm că vor fii maxim 6.

Parcurgerea de generare a vecinilor este:

**function** *computeNeighbors*(geometry) {  
 **for** (**i** = 0; **i** < geometry.**vertices**.length; ++**i**) {  
 geometry.**vertices**[**i**].**neighbors** = **new** Set();  
 geometry.**vertices**[**i**].**faces** = **new** Set();  
 }  
 **for** (**i** = 0; **i** < geometry.**faces**.length; ++**i**) {  
 **var** face = geometry.**faces**[**i**];  
 geometry.**vertices**[face.**a**].**neighbors**.add(face.**b**);  
 geometry.**vertices**[face.**b**].**neighbors**.add(face.**a**);  
 geometry.**vertices**[face.**b**].**neighbors**.add(face.**c**);  
 geometry.**vertices**[face.**c**].**neighbors**.add(face.**b**);  
 geometry.**vertices**[face.**c**].**neighbors**.add(face.**a**);  
 geometry.**vertices**[face.**a**].**neighbors**.add(face.**c**);  
 geometry.**vertices**[face.**a**].**faces**.add(**i**);  
 geometry.**vertices**[face.**b**].**faces**.add(**i**);  
 geometry.**vertices**[face.**c**].**faces**.add(**i**);  
 }  
}

Pentru a defini obstacolele din teren am decis să setez o variabilă cu înălțimea de la care începe „nivelul mării” și am colorat tot ce este mai mic de atât cu albastru pentru o ușoară observare. Astfel, am implementat costul muchiilor ca o funcție ce întoarce dacă unul dintre capete este mai mic de acest nivel sau distanța euclidiană a celor două puncte tridimensionale altfel. Calculul distanței pentru puncte în trei dimensiuni este deja implementată în librăria Three.Js, și poate fi apelată ca metodă a unui obiect de tipul Vector3. Așadar avem următoarea funcție de calcul de cost:

**var *sea\_level*** = 0.41;

**function** *edgeCost*(a, b) {  
 **if** (***heightMap***[a % 256][**Math**.floor(a / 256)] <= ***sea\_level*** || ***heightMap***[b % 256][**Math**.floor(b / 256)] <= ***sea\_level***) {  
 **return** Number.**POSITIVE\_INFINITY**;  
 }  
 **var** u = ***geometry***.**vertices**[a];  
 **var** v = ***geometry***.**vertices**[b];  
 **return** u.distanceTo(v);  
}

În continuare, pentru a furniza roboților vârfurile și muchiile pe care le au aceștia în sfera de viziune definim următoarea funcție.

**var *vision\_range*** = 2;

**function** *computeVertexVision*(vertexidx) {  
 **var** vertex = ***geometry***.**vertices**[vertexidx];  
 **var** visited = **new** Set();  
 **var** queue = [vertexidx];  
 **var** edges = [];  
 **while** (queue.**length** > 0) {  
 **var** u = queue.pop();  
 **var** v = ***geometry***.**vertices**[u];  
 **for** (**i of** v.**neighbors**) {  
 **var** w = ***geometry***.**vertices**[**i**];  
 **var** d = vertex.distanceTo(w);  
 **if** (d <= ***vision\_range*** && !visited.has(**i**)) {  
 queue.push(**i**);  
 edges.push({**a**: u, **b**: **i**});  
 }  
 }  
 visited.add(u);  
 }  
 **return** {**vertices**: visited, **edges**: edges};  
}

Această funcție primește ca parametru indicele vârfului din graful terenului în care se află robotul și efectuează o parcurgere bazată pe căutarea în lățime. Variabila ***geometry*** conține geometria terenului și este declarată global pentru ușurința accesării.

Deși terenul poate avea foarte multe vârfuri, robotul nu va cunoaște foarte multe inițial. În cazul de față am ales o matrice de 256x256 de noduri, adică 65.536, ce vor genera în jur de 130.000 de fețe. Astfel, am ales să adaug un „index” pe indicile vârfului stocat la nivel de robot. O implementare simplă a acestui index este o hartă (map) de chei valori. Teoretic, această implementare poate avea o complexitate timp și spațiu liniară în dimensiunea valorilor, folosind funcție și tabel hash. Cum ECMAScript 6 are această structură implementată nativ, am folosit-o încrezător, chiar dacă este încă experimentală.

La nivel de Robot avem:

**var** *Robot* = **function** (start\_index) {

**…**

**this**.**known\_v** = [];**this**.**known\_e** = [];**this**.**g2k\_map** = **new** Map();

…

**known\_v** este lista de vârfuri cunoscute de robot și **known\_e** este lista de muchii cunoscute de acesta. Funcția următoare va fi folosită pentru a găsi sau introduce un vârf în lista **known\_v** având indicele global al acestuia, adică indicile din graful terenului.

global2knownVertex: **function** (globalVertexIndex) {  
 **var** k\_idx = **this**.**g2k\_map**.get(globalVertexIndex);  
 **if** (**typeof** k\_idx === **"undefined"**) {  
 k\_idx = **this**.**known\_v**.**length**;  
 **this**.**known\_v**.push({  
 **g\_v\_idx**: globalVertexIndex,  
 **g\_v\_i**: globalVertexIndex % 256,  
 **g\_v\_j**: **Math**.floor(globalVertexIndex / 256),  
 **neighbors**: **new** Set(),  
 **g\_props**: **new *THREE***.Vector3()  
 });  
 **this**.**g2k\_map**.set(globalVertexIndex, k\_idx);  
 }  
 **return** k\_idx;  
}

Funcția prin care robotul își actualizează aceste informații este:

updateVision: **function** () {  
 **var** gsi = **this**.**known\_v**[**this**.**c\_v\_k\_idx**].**g\_v\_idx**;  
 **var** gs = ***geometry***.**vertices**[gsi];  
 **var** vision = *computeVertexVision*(gsi);  
 **var** myarr = [];  
 **for** (v **of** vision.**vertices**) {  
 myarr.push({**g\_v**: v, **k\_v**: **this**.global2knownVertex(v)});  
 **this**.vertexUpdated(**this**.global2knownVertex(v));  
 }  
 **for** (**var** i = 0; i < **this**.**known\_e**.**length**; ++i) {  
 **var** ga = **this**.**known\_v**[**this**.**known\_e**[i].**a**].**g\_v\_idx**;  
 **var** gb = **this**.**known\_v**[**this**.**known\_e**[i].**b**].**g\_v\_idx**;**this**.colorEdge(ga, gb, 0xA0A0A0);  
 }  
 **for** (i = 0; i < vision.**edges**.length; ++i) {  
 ga = vision.**edges**[i].**a**;  
 gb = vision.**edges**[i].**b**;  
 **var** a = **this**.global2knownVertex(ga);  
 **var** b = **this**.global2knownVertex(gb);  
 **this**.addEdge(a, b, 0x0000FF);  
 }  
}

Am introdus tot aici și colorarea tuturor muchiilor cu gri (0xA0A0A0) și a celor vizibile cu albastru (0x0000FF). Tot aici apelez și o funcție a robotului ce verifică dacă un vârf cunoscut și-a modificat proprietățile, fiind unul nou sau actualizat, și în caz afirmativ să transmită informația mai departe algoritmului folosit pentru găsirea drumului minim. Algoritmul folosit poate fi schimbat fără a modifica robotul. Pentru a putea realiza acest lucru am abstractizat metodele pe care le expune un algoritm prin folosirea acelorași signaturi de metode. Robotul reține o referință către obiectul algoritmului, indiferent de care este acesta.

vertexUpdated: **function** (v) {  
 **var** kv = **this**.**known\_v**[v];  
 **var** kvp = kv.**g\_props**;  
 **var** gvp = ***geometry***.**vertices**[kv.**g\_v\_idx**];  
 **var** dist = **this**.\_h(**this**.**c\_v\_k\_idx**, v);  
 **var** isPartial = ***vision\_range*** - dist < 0.5;  
 **if**(kvp.**x** != gvp.x || kvp.**y** != gvp.y || kvp.**z** != gvp.z || (!isPartial && **this**.**bridgeVertices**.has(v))) {  
 kvp.**x** = gvp.x; kvp.**y** = gvp.y; kvp.**z** = gvp.z;  
 **if** (isPartial) {  
 **this**.**bridgeVertices**.add(v);  
 } **else** {  
 **this**.**bridgeVertices**.delete(v);  
 **this**.**notBridgeVertices**.add(v);  
 }  
 **if** (**this**.algorithm != **null**)  
 **this**.algorithm.vertexUpdated(v);  
 }  
}

Am implementat tot aici, la nivel de robot, și două mulțimi de vârfuri pentru a îmbunătăți rularea algoritmilor:

* **bridgeVertices**: mulțime de vârfuri care nu au fost explorate complet, i.e. nu au fost la o distanță minimă de robot cât să nu existe muchie cu celălalt capăt în afara sferei de viziune
* **notBridgeVertices**: mulțime de vârfuri care au fost explorate complet, și deci nu pot exista alte muchii adiacente în afară de cele știute

Procedând în acest fel reușim să reducem căutarea, știind că nodurile cu un cost egal cu ∞ și se află în mulțimea **notBridgeVertices** nu sunt interesante deloc, întrucât știm că sunt obstacole, adică nu este permis accesul robotului spre a fi mutat în aceste puncte. Totuși, deși avem această informație, terenul poate fi modificat în orice moment și nu putem spune că este blocat accesul spre destinație. Dacă graful terenul este foarte volatil, atunci problema devine una mai complexă, iar algoritmii studiați nu mai au aceleași proprietăți.

Pașii algoritmilor îi voi ilustra grafic, și pentru a putea realiza acest lucru am transformat algoritmii în „automate”, unde am împărțit structurile repetitive ale algoritmilor în stări, tranzițiile fiind date de terminarea acestora. Astfel, toți algoritmii au funcția step care efectuează un pas al algoritmului în cauză.

Tot în această funcție incrementez și numărul de pași efectuați atât în parcurgerea curentă pentru mutare, cât și per total, aceste variabile având același nume pentru abstractizarea accesului. Am decis ca funcția să și întoarcă o valoare booleană, care să indice faptul că algoritmul mai are sau nu pași de făcut până în vârful destinație. Valoarea returnată este utilă la detectarea cazului în care algoritmul determină că nu există drum sau la depanarea eventualelor probleme de implementare. Drept urmare structura acestei funcții este comună tuturor algoritmilor și arată în felul următor:

step: **function** () {  
 ++**this**.stepCount;  
 ++**this**.totalStepCount;  
 **switch** (**this**.**state**) {  
 **case** 0:  
**…**  
 **return true**;  
 **break**;  
 **case** 1:  
**…**  
 **return true**;  
 **break**;

…  
 **case** n:  
**…**  
 **break**;  
 **default**:  
 **this**.**state** = -1;  
 **console**.error(**"Weighted A\*: crashed"**);  
 **return false**;  
 **break**;  
 }  
}

Algoritmul A\* ponderat l-am implementat astfel:

Cazul 0 – inițializare:

**case** 0:  
 **this**.**queue** = [];  
 **this**.**visited** = **new** Set();  
 **var** cq = {**ki**: **this**.**robot**.**c\_v\_k\_idx**, **c**: 0, **prev**: **null**};  
 **this**.**robot**.**known\_v**[cq.**ki**].**h** = Number.**POSITIVE\_INFINITY**;  
 **this**.**bestF** = **null**;  
 **this**.**queue**.push(cq);  
 **this**.**visited**.add(cq.**ki**);  
 **this**.**state** = 1;  
 **this**.stepCount = 1;  
 **return true**;

Se crează coada vidă, mulțimea de noduri vizitate, se adaugă nodul de start în coadă, se resetează numărul de pași și se trece la umrătoarea stare.

**case** 1:  
 **if** (**this**.**queue**.**length** == 0) {  
 **this**.**state** = 3; **this**.**pathrev** = **this**.**bestF**;  
 **if** (**this**.**bestF**.**c** == Number.**POSITIVE\_INFINITY**) {  
 **console**.info(**"Weighted A\*: blocked!"**);  
 **return false**;  
 }  
 **return true**;  
 }  
 **this**.**cq** = **this**.**queue**.shift();  
 **this**.**iter** = **this**.**robot**.**known\_v**[**this**.**cq**.**ki**].**neighbors**.values();  
 **this**.**state** = 2;  
 **return true**;

Cazul 1 – parcurgerea cozii. Aici se verifică existența unui element în coadă și se pregătește parcurgerea listei de vecini pentru cazul 2. Se face și verificarea pentru cazul de blocaj tot aici.

**case** 2:  
 **var** next = **this**.**iter**.next();  
 **if** (next.done) {  
 **this**.**state** = 1;  
 ++**this**.middleStepCount;  
 } **else** {  
 **var** nki = **this**.**robot**.neighborV(next.**value**, **this**.**cq**.**ki**);  
 **var** ngi = **this**.**robot**.**known\_v**[nki].**g\_v\_idx**;  
 **var** cgi = **this**.**robot**.**known\_v**[**this**.**cq**.**ki**].**g\_v\_idx**;  
 **var** cost = *edgeCost*(cgi, ngi);  
 **var** nq = {**ki**: nki, **c**: **this**.**cq**.**c** + cost, **prev**: **this**.**cq**};  
 **if** (**this**.**robot**.**known\_v**[nki].**h** != Number.**POSITIVE\_INFINITY**) {  
 **var** h = Number.**POSITIVE\_INFINITY**;  
 **if** (***vision\_range*** - nq.**c** <= 0.5) {  
 h = **this**.vertexH(ngi);  
 }  
 **this**.**robot**.**known\_v**[nki].**h** = h;  
 }  
 **if** (!**this**.**visited**.has(nki) && nq.**c** != Number.**POSITIVE\_INFINITY**) {  
 **if** (**this**.**goal** == ngi) {  
 **if** (**this**.**bestF** == **null** || nq.**c** <= **this**.**robot**.**known\_v**[**this**.**bestF**.**ki**].**h**\***this**.**eps** + **this**.**bestF**.**c**){**this**.**state** = 3;  
 **this**.**pathrev** = nq;  
 }  
 } **else** {  
 **this**.**visited**.add(nq.**ki**);  
 **this**.**robot**.colorEdge(ngi, **this**.**robot**.**known\_v**[nq.**prev**.**ki**].**g\_v\_idx**, 0x00FF00);  
 **this**.**queue**.push(nq);**if** (**this**.**robot**.**bridgeVertices**.has(nki)) {  
**if** (**this**.**bestF** == **null** || **this**.vertexH(ngi)\***this**.**eps** + nq.**c** < **this**.vertexH(**this**.**robot**.**known\_v**[**this**.**bestF**.**ki**].**g\_v\_idx**)\***this**.**eps** + **this**.**bestF**.**c**) {**this**.**bestF** = nq;  
 }  
 }  
 }  
 }  
 }

Cazul 2 – se ia primul vecin din listă, se calculează costul până la el, se verifică dacă acesta este vizibil parțial sau complet, se verifică dacă este accesibil, i.e. nu are costul de a ajunge în el ∞, și se verifică dacă a mai fost vizitat deja.

Se verifică și dacă acest nod este cel destinație, iar în caz afirmativ am decis să opresc căutarea numai atunci când costul drumului până la acest nod este mai mic decât suma dintre costul și euristica nodului cel mai bun candidat la acel moment. Teoretic, această condiție este valoarea în funcția *f* este maxim cea mai bună valoare în *f* cunoscută. Dacă totuși nu există un vârf candidat la momentul găsirii vârfului destinație, atunci acesta este un drum minim. Acest algoritm fiind A\* ponderat am înmulțit valoarea funcției *h* cu **this**.**eps** fiind constanta de pondere ε.

Pentru cazul în care vecinul explorat în acest caz nu este cel destinație îl adăugăm în coadă, îl marcăm ca vizitat, și îl setăm ca și candidat pentru drumul minim în caz că nu există unul deja sau are valoarea în funcția *f* mai mică decât a celui existent.

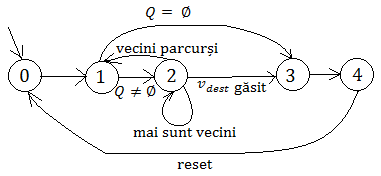
**case** 3:  
 **var** q, lq;  
 **for** (q = **this**.**pathrev**, lq = q; q.**prev** != **null**; lq = q, q = q.**prev**) {  
 **this**.**robot**.colorEdge(**this**.**robot**.**known\_v**[q.**ki**].**g\_v\_idx**, **this**.**robot**.**known\_v**[q.prev.**ki**].**g\_v\_idx**, 0xFF0000);  
 }  
 **this**.**move\_to** = lq.**ki**;  
 **this**.**state** = 4;  
 ++**this**.middleStepCount;  
 ++**this**.majorStepCount;

Cazul 3 – afișarea drumului minim și setarea vârfului în care va fi mutat robotul. În acest caz se poate ajunge din cazul 2 dacă am explorat nodul destinație printr-un drum minim relativ la candidatul curent, sau din cazul 1 dacă am terminat de parcurs toate nodurile, neavând un drum direct până la destinație prin graful cunoscut, dar avem încă vârfuri vizibile parțial.

În acest caz este implementată funcția *DrumInapoi* definită în 2.2.1, aceasta parcurgând referința **pathrev** din noduri pentru a găsi primul nod din drumul minim determinat. Am folosit variabila **this**.majorStepCount pentru a separa mutările roboților și determinările de drumuri.

**case** 4:  
 **this**.**robot**.moveTo(**this**.**move\_to**);  
 ++**this**.middleStepCount;  
 ++**this**.majorStepCount;  
 **if** (**this**.**robot**.**known\_v**[**this**.**move\_to**].**g\_v\_idx** == **this**.**goal**) {  
 **this**.**state** = -2;  
 **console**.info(**"Weighted A\*: finished (goal)"**);  
 **return false**;  
 }  
 **this**.**state** = 0;*//reset* **return true**;

Cazul 4 – mutarea robotului. În acest caz se execută simpla operațiune de a muta robotul în nodul determin de algoritm.

*Figura 7. Diagrama automatului determinat – sursă prelucrare propire*

## 3.3. Testare

Scenarii de testare, ușurința utilizării, resursele necesare pentru utilizare

## 3.4. Rezultate

Rulare – numar de pași efectuați, distanța parcursă, dimensiunea grafului cunoscut, timp și spațiu de rulare. Statistici – corelații, complexitate, aplicabilitate.

1. Sursa http://en.wikipedia.org/wiki/A\*\_search\_algorithm [↑](#footnote-ref-1)
2. Sursa http://zerowidth.com/2013/05/05/jump-point-search-explained.html [↑](#footnote-ref-2)