UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

Aplicații ale algoritmilor euristici în determinări de drumuri minime

COORDONATOR ȘTIINȚIFIC

Lect. Dr. Ruxandra Marinescu-Ghemeci

STUDENT

Adrian Munteanu

BUCUREȘTI

IUNIE 2015

Cuprins

Introducere

Eficiența și optimizarea algoritmilor reprezintă o direcție de cercetare continuă 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. În această lucrare sunt combinate elemente de teoria grafurilor, inteligență artificială, tehnologii web și nu în ultimul rând geometrie computațională.

Vom studia pe parcursul lucrării atât partea teoretică a algoritmilor cât și partea practică, explicând totodată unele îmbunătățiri implementate sau doar propuse. Am ales să folosesc tehnologia web disponibilă, întrucât aplicația devine ușor de accesat și utilizat. Astfel, voi prezenta detalii de implementare corelate cu pseudocodul din partea teoretică.

Această lucrare prezintă și un mod de lucru eficient, exemplificând utilitatea unor programe cât mai bune. Printre acestea voi prezenta un sistem de versionare a fișierelor, un editor și librării, toate contribuind la ușurința dezvoltării aplicației.

Algoritmii de căutare sunt importanți, întrucât rezolvă o problemă esențială pentru aplicații reale, și de aceea este necesară analiza lor pentru a le determina aplicabilitatea și eficiența. Astfel, voi compara aceste aspecte ale algoritmilor studiați și le voi ilustra în aplicație pentru a le înțelege cât mai ușor.

Capitolul I. Modelul matematic al problemei și soluțiilor

Problema studiată în această 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.

1.1. 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 $d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}$.

În figura 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)**.

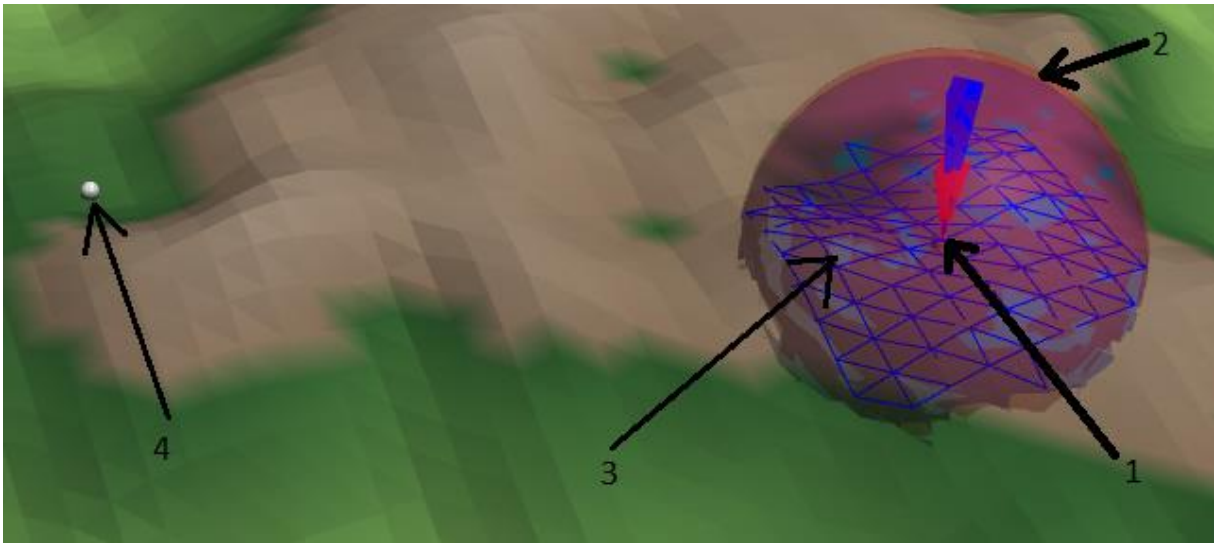


Figura 1. Ilustrare – Sursă prelucrare proprie

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 găsească cel mai scurt drum ținând cont de incompletitudinea informației.

Așa cum în realitate terenul se poate modifica, și astfel devenind inaccesibil sau, din contră, mai accesibil, vrem ca algoritmul folosit să țină cont de acestea și în simulația noastră.

1.2. Soluții

Pentru a rezolva problema celui mai scurt drum putem implementa diverşi 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 şi mai mult situația 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 $d(R, n) < viz$, 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 întâmpla 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 $h(x) \leq d(x, y) + h(y)$ pentru orice *x* și *y* noduri adiacente.

1.2.1. 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:

```
Procedura DrumInapoi:
Fie d = [] lista nodurilor drumului
Cât timp v ≠ null execută
        d.adauga_la_inceput(v)
        v ← v.pred
Sfârşit – cât timp
Returnează d
```

```
Fie G un graf
Fie v_start ∈ G nodul inițial și v_dest ∈ G nodul destinație
Fie Q ← ∅ o coadă
Q.adauga(v_start)
Marchează v_start ca vizitat
v_start.pred ← null
Cât timp Q ≠ ∅ execută
        v ← Q.scoate()
        Dacă v = v_dest atunci execută DrumInapoi și STOP
        Pentru toate muchiile de la v la w din G execută
                Dacă w nu este marcat ca vizitat atunci
                        Q.adauga(w)
                        Marchează w ca vizitat
                        Setează w.pred ← v
                Sfârșit – dacă
        Sfârșit – pentru
Sfârșit – cât timp
Returnează eroare (nu există drum)
```

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 v_{start}, ce devine astfel rădăcina arborelui de parcurgere. Notăm cu l numărul de vârfuri în care se poate ajunge din v_{start}. Definim pentru un vârf v următoarele:

$nivel[v]$ = nivelul lui v în arborele de parcurgere
$dist[v]$ = distanța de la v_{start} la v în graf
$poz[v]$ = numărul de ordine i al v, unde $1 \leq i \leq l$, și v este al i-lea vârf adăugat în coadă

Vrem să demonstrăm că $nivel[v] = dist[v]$ pentru orice v. Vom recurge la inducție după $poz[v]$, mai exact vom arăta pentru $i = 1 \ldots l$ că pentru v cu $poz[v] = i$ avem

(**i1**) $dist[v] = nivel[v]$ și

(**i2**) pentru orice vârf w, dacă $dist[w] < dist[v]$, atunci $poz[w] < poz[v]$

Cazul $i = 1$ este trivial, având evident v_{start} introdus primul în coadă, deci $nivel[v] = 0 = dist[v]$ și nu există noduri w cu $dist[w] < 0$.

Presupunem ipoteza de inducție adevărată pentru $k < i$ și demonstrăm pentru i.

Pentru a demonstra (**i1**), fie $poz[v] = i$ și v' părintele lui v în arborele de parcurgere. Presupunem prin reducere la absurd că există un drum $v_{start} \dashrightarrow w \rightarrow v$ de lungime $x < nivel[v]$. Avem deci

$$poz[v'] < poz[v], \qquad (1)$$

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

$$dist[w] < dist[v'], \quad (2)$$

deoarece din (1) putem aplica ipoteza de inducție (i1) la $poz[v']$, obținând astfel

$$dist[v'] = nivel[v'] = nivel[v] - 1 > x - 1 \geq dist[w'].$$

În plus, aplicând (1) în ipoteza de inducție (**i2**) pentru $poz[v']$, și având (2) rezultă că

$$poz[w] < poz[v'] \quad (3)$$

Dar, eliminându-l pe *w* din coadă și existând muchia $w \to v$, 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 $poz[w]$, care din (3) este strict mai mic decât $poz[v]$, și deci v' nu poate fii părintele lui *v*, având contradicție cu presupunerea.

Demonstrație pentru (**i2**): Fie $poz[v] = i$ și

$$dist[w] < dist[v]. \qquad (4)$$

Presupunem prin reducere la absurd că $poz[w] \geq poz[v]$. Din (4) avem că $w \neq v$ implică $poz[w] \neq poz[v]$, deci

$$poz[w] > poz[v]. \qquad (5)$$

Fie v' părintele lui *v* in arborele de parcurgere și fie $v_{start} \dashrightarrow w' \to w$ un drum minim de la v_{start} la w. Asta implică faptul că $v_{start} \dashrightarrow w'$ este un drum minim de la v_{start} la w'. Concluzionăm că

$$dist[w'] = dist[w] - 1. \quad (6)$$

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

$$poz[v'] < poz[v]. \qquad (7)$$

Din ipoteza (**i1**) în $poz[v]$ drumul $v_{start} \dashrightarrow v' \to v$ dat de parcurgere este drum minim, deci și $v_{start} \dashrightarrow v'$ este drum minim. Din aceasta rezultă că

$$dist[v'] = dist[v] - 1. \quad (8)$$

Din (4), (6) și (8) implică

$$dist[w'] < dist[v']. \qquad (9)$$

Aplicând ipoteza de inducție (**i2**) în $poz[v']$, care din (7) este mai mic decât $poz[v]$, și împreună cu (9), obținem

$$poz[w'] < poz[v']. \quad (10)$$

La momentul extragerii lui v' din coadă a fost deja adăugat *v* la un moment anterior, iar la momentul extragerii lui w' este adăugat *w* în coadă dacă nu era deja. Din (10) deducem ca w' fiind inserat înaintea lui v' 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 $dist[v]$ = distanța de la v_{start} la *v* în graf. Observăm că această distanță este pentru muchii cu cost = 1 chiar $nivel[v]$ = nivelul lui v în arborele de parcurgere. Încercăm deci să introducem explicit $nivel[v]$ în coadă odată cu *v* pentru a parcurge în ordinea costului. Acest algoritm este chiar algoritmul lui Dijkstra și îl vom studia în cele ce urmează.

1.2.2. Dijkstra

Algoritmul lui Dijkstra rezolvă problema găsirii celui mai scurt drum între două puncte într-un graf în care muchiile au asociate costuri. Alți algoritmi asemănători sunt prezentați pe larg în lucrarea Uninformed pathfinding: A new approach[13].

Acest algoritm pentru găsirea drumului de cost minim pornește cu presupunerea că nu există un drum minim către niciun nod de la v_{start}, 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 G un graf
Fie v_start ∈ G nodul inițial și v_dest ∈ G nodul destinație
Fie Q ← ∅ o coadă
v_start.dist ← 0
v_start.pred ← null
Pentru v ∈ G vârf
        Dacă v ≠ v_start atunci
                v.dist ← ∞
                v.pred ← null
        Sfârșit – dacă
        Q.adauga(v)
Sfârșit – pentru
```

```
Cât timp Q ≠ ∅ execută
        u ← Q.scoateMin()
        Pentru toate muchiile de la u la v din G execută
                alt ← u.dist + cost(u, v)
                Dacă alt < v.dist atunci
                        v.dist ← alt
                        v.pred ← u
                Sfârşit – dacă
        Sfârşit – pentru
Sfârşit – cât timp
Dacă v_dest.dist = ∞ atunci
        Returnează eroare (nu există drum)
Altfel
        Execută DrumInapoi
Sfârşit – dacă
```

Algoritmul lui Dijkstra găsește un drum de cost minim de la nodul de start la un nod oarecare în momentul în care îl scoate din coadă. Intuitiv, algoritmul extrăgând vârful cu cel mai mic cost asociat, dacă la momentul extragerii unui nod acesta nu are cel mai mic cost asociat, atunci ar fi existat un alt nod prin care să fi trecut un drum minim, nod care să nu fi fost explorat. Însă, acest nod din urmă fiind obligatoriu să aibă un cost mai mic decât cel inițial, ar fi fost deja extras înaintea primului. Demonstrație:

Definim costul drumului $p = [v_o, v_1, \dots, v_k]$ cu funcția $w(p) = \sum_{i=1}^{k} cost(v_{i-1}, v_i)$ și drumul minim între u și v cu funcția

$$\delta(u,v) = \begin{cases} \min\left\{w(p) : u \xrightarrow{p} v\right\}, dacă\ există\ un\ drum\ p\ între\ u\ şi\ v \\ \infty, altfel \end{cases}$$

Observăm că un drum *p* între *u* și *v* este minim dacă $w(p) = \delta(u, v)$.

Vom spune că un vârf este vizitat dacă nu se află în coadă și presupunem că *u* este primul vârf vizitat pentru care $u.dist \neq \delta(v_{start}, u)$ și observăm că:

- u nu poate fi v_{start}, deoarece $v_{start}.dist = 0$
- trebuie sa existe un drum de la v_{start} la u, pentru că *u* a fost vizitat
- din moment ce există cel puțin un drum, cel puțin unul este minim

Fie $v_{start} \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$ drumul minim de la v_{start} la *u*, unde *x* este vizitat și *y* este primul nod nevizitat. La momentul vizitării lui *x* aveam $x.dist = \delta(v_{start}, x)$, 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:

$$y.dist = \delta(v_{start}, y) \leq \delta(v_{start}, u) \leq u.dist$$

Și *y* și *u* erau nevizitate la momentul alegerii lui *u*, deci $u.dist \leq y.dist$, ceea ce rezultă că

$$y.dist = \delta(v_{start}, y) = \delta(v_{start}, u) = u.dist$$

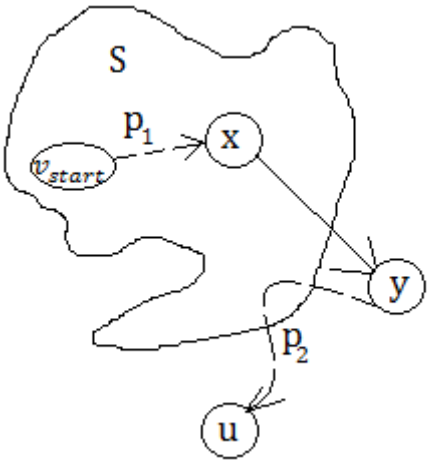


Figura 2. Ilustrare – Sursă prelucrare proprie

Deci $u.dist = \delta(v_{start}, u)$ contrazicând astfel presupunerea făcută, demonstrând astfel că pentru orice nod vizitat *u* a fost găsit drumul minim $u.dist$. 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 v_{dest} a fost vizitat, i.e. – a fost scos din coadă.

Algoritmul lui Dijkstra are o complexitate timp de $O(|E| + |V|^2)$ 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 $O(|E| + |V| \log |V|)$ fiind astfel un algoritm semnificativ mai eficient decât căutarea în lățime. Detalii pot fi găsite în cartea Introduction to Algorithms, Second Edition[2], iar pentru analiza cazurilor nefavorabile este utilă lucrarea Worst-Case Analysis of Heuristic Algorithms[6].

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.

1.2.3. 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 $f(n) = g(n) + h(n)$, unde $g(n)$ este costul de la vârful de start la *n* și $h(n)$ 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 $v.dist$ din algoritmul lui Dijkstra în $v.g$ pentru orice $v \in V$. 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 G un graf
Fie $v_{start} \in G$ nodul inițial și $v_{dest} \in G$ nodul destinație
Fie $Q \leftarrow \emptyset$ o coadă
$v_{start}.g \leftarrow 0$
$v_{start}.f \leftarrow h(v_{start})$
$v_{start}.pred \leftarrow null$
$Q.adauga(v_{start})$
Cât timp $Q \neq \emptyset$ execută
    $u \leftarrow Q.scoateMin()$
    Dacă $u = v_{dest}$ atunci execută *DrumInapoi* și STOP
    $u.vizitat \leftarrow Adevărat$
    Pentru toate muchiile de la u la v din G execută
        Dacă $v.vizitat = Adevărat$ atunci treci la următorul
        $alt \leftarrow u.g + cost(u, v)$
        Dacă $v \notin Q$ sau $alt < v.g$ atunci
            $v.g \leftarrow alt$
            $v.f \leftarrow v.g + h(v)$
            $v.pred \leftarrow u$
            Dacă $v \notin Q$ atunci $Q.adauga(v)$
        Sfârșit – dacă
    Sfârșit – pentru
Sfârșit – cât timp
Returnează eroare (nu există drum)

Propoziție: $g(n) = \delta(n)$ și algoritmului A* garantează găsirea unui drum minim dacă următoarele condiții sunt îndeplinite pentru *h*:

- h este optimistă, adică nu supraestimează niciodată drumul rămas de parcurs: $h(n) \leq \delta(n, v_{dest}), \forall n \in V$
- h este consistentă: $h(u) \leq cost(u, v) + h(v), \forall\, muchie\ (u, v) \in E$

Demonstrație

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

Fie $p = [v_1, v_2, \dots, v_k]$ un drum minim între v_1 și v_k și pentru orice *i* și *j* cu

$1 \leq i \leq j \leq k$, fie $p_{ij} = [v_i, v_{i+1}, \dots, v_j]$ sub-traseul din p de la vârful *i* la vârful *j*. Atunci drumul p_{ij} este un drum minim de la v_i la v_j. Demonstrație: descompunem drumul *p* în

$v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, și avem că $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$

În continuare, presupunem că există un drum p'_{ij} de la v_1 la v_k de cost $w(p'_{ij}) < w(p_{ij})$. Atunci $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ este un drum de la v_1 la v_k al cărui cost

$w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ este mai mic decât $w(p)$, ceea ce contrazice presupunerea că *p* este drum minim de la v_1 la v_k.

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 $f(v_{dest}) > \delta(v_{start}, v_{dest})$ și deci trebuie să existe un nod *n* care este neexplorat de algoritm și care face parte dintr-un drum minim. Adică $\exists p = [v_{start}, v_1, v_2, \dots, n, \dots, v_{dest}]$ drum minim, ceea ce implică $[v_{start}, v_1, v_2, \dots, n]$ drum minim de cost $\delta(v_{start}, n)$ datorită lemei demonstrate anterior, și avem că $f(n) \geq f(v_{dest})$, deoarece algoritmul nu s-ar fi oprit altfel.

De asemenea mai avem și $f(n) = g(n) + h(n) = \delta(v_{start}, n) + h(n)$ despre care știm că este mai mică sau egală cu $\delta(v_{start}, n) + \delta(n, v_{dest}) = \delta(v_{start}, v_{dest})$, deoarece h este admisibilă.

Rezultă astfel că $\delta(v_{start}, v_{dest}) \geq f(n) \geq f(v_{dest})$, ceea ce contrazice presupunerea făcută. Astfel că $\delta(v_{start}, v_{dest}) = f(v_{dest})$.

Pentru cazul de față în care $h(v) = d(v, v_{dest})$ știm că este optimistă, deoarece pentru orice muchie $(x, y) \in E$ avem $cost(x, y) = d(x, y)$ și pentru orice 3 puncte *a*, *b*, *c* din spațiul geometric euclidian avem că $d(a, b) + d(b, c) \geq d(a, c)$, astfel că dacă nu există muchie de la *v* la v_{dest} având costul $d(v, v_{dest})$ atunci orice drum minim *p* de la *v* la v_{dest} are costul total $w(p) \geq d(v, v_{dest})$. Tot de aici rezultă și faptul că *h* este consistentă. Mai multe informații despre consistența funcțiilor euristice se află în lucrarea Inconsistent heuristics in theory and practice[5].

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 $O(|E|)$, 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 fi 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.

1.2.4. 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 $\varepsilon > 1$ ce reprezintă înclinarea către nodurile mai apropiate de destinație.

Definim în continuare pentru ușurință $h_\varepsilon(v) = \boldsymbol{\varepsilon} h(v)$, astfel funcția pe baza căreia se face comparația nodurilor devine $f = g + \varepsilon h = g + h_\varepsilon$

Pseudocodul rămâne același de la algoritmul A*, cu mica modificare la calcularea funcției *f*: $v.f \leftarrow v.g + \boldsymbol{\varepsilon} h(v)$.

Din cauza ușoarei modificări a ordinii de explorare acest algoritm nu găsește soluția optimă. Se numește că euristica h_ε folosită este ε-consistentă dacă

$h_\varepsilon(v_{dest}) = 0$ și $h_\varepsilon(u) \leq \varepsilon cost(u, v) + h_\varepsilon(v)$, pentru orice u și v cu $(u, v) \in E$ muchie și $f(u) \leq f(v)$ și $u \neq v_{dest}$. Astfel, algoritmul A* ponderat este ε-sub-optim, adică

$w(p) \leq \varepsilon \delta(v_{start}, v_{dest})$, unde *p* este soluția găsită de algoritm. Cu alte cuvinte, este garantat că soluția găsită de A* ponderat va avea un cost de maxim ε ori mai mare decât costul soluției optime.

Știind că *h* este optimistă avem că $h(v) \leq \delta(v, v_{dest}), \forall v \in V$ ceea ce rezultă că și $\varepsilon h(v) \leq \varepsilon \delta(v, v_{dest})$. Adunând funcția *g*, rezultă că

$f(v) = g(v) + \varepsilon h(v) \leq g(v) + \varepsilon \delta(v, v_{dest}), \forall v \in V$

Din *h* este consistentă: $h(u) \leq cost(u, v) + h(v), \forall\, muchie\ (u, v) \in E$ rezultă că și $\varepsilon h(u) \leq \varepsilon cost(u, v) + \varepsilon h(v), \forall\, muchie\ (u, v) \in E$.

În figura alăturată[1] avem algoritmul A* original în partea stângă și algoritmul A* cu pondere $\varepsilon = 5$ 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 fără culoare sunt în coada de priorități. Derivații de la algoritmii Dijkstra, A* și A* ponderat sunt detaliate în lucrarea Incorporating weights into real-time heuristic search[16].

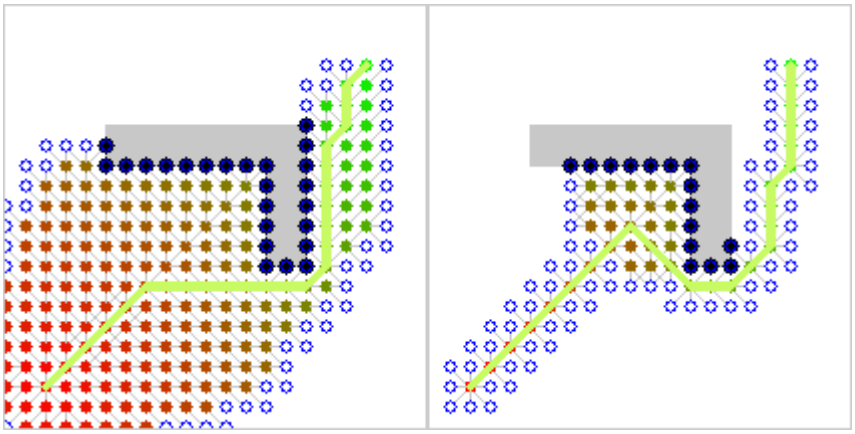


Figura 3 – sursa Wikipedia

[1] Sursa http://en.wikipedia.org/wiki/A*_search_algorithm

1.2.5. 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). Această metodă a fost descrisă în Fast Pathfinding via Symmetry Breaking[8], Online Graph Pruning for Pathfinding on Grid Maps[9] și Path planning with modified A star algorithm for a mobile robot[3].

După cum se poate observa în figura alăturată[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.

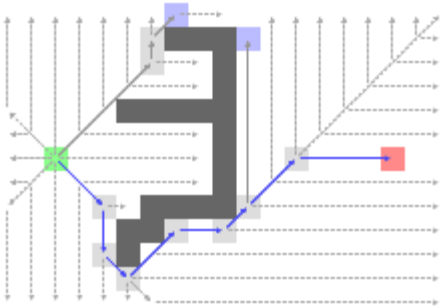


Figura 4. JPS - Sursa Witmer Nathan

Pseudocod:

Fie G un graf
Fie $v_{start} \in G$ nodul inițial și $v_{dest} \in G$ nodul destinație
Fie $Q \leftarrow \emptyset$ o coadă
$v_{start}.g \leftarrow 0$
$v_{start}.f \leftarrow h(v_{start})$
$v_{start}.pred \leftarrow null$
$Q.adauga(v_{start})$
Cât timp $Q \neq \emptyset$ execută
    $u \leftarrow Q.scoateMin()$
    Dacă $u = v_{dest}$ atunci execută *DrumInapoi* și STOP
    $u.vizitat \leftarrow Adevărat$
    Pentru toate muchiile de la u la v din G execută
        Dacă $v.vizitat = Adevărat$ atunci treci la următorul
        $n \leftarrow salt(u, directie(u, v), v_{start}, v_{dest})$
        $alt \leftarrow u.g + w(u, n)$

[2] Sursa http://zerowidth.com/2013/05/05/jump-point-search-explained.html

```
            Dacă n ∉ Q sau alt < n.g atunci
                n.g ← alt
                n.f ← n.g + h(n)
                n.pred ← u
                Dacă n ∉ Q atunci Q.adauga(n)
            Sfârşit – dacă
        Sfârşit – pentru
Sfârşit – cât timp

Returnează eroare (nu există drum)
Funcția directie(u, v) = (v_x − u_x, v_y − u_y)
Funcția pas(u, d) = v, unde v este vârful de la poziția (u_x + d_0, u_y + d_1)
```

Pentru a defini funcția *salt* vom introduce notația de *nod forțat*, folosind figura 5:

Aceste două cazuri sunt singurele făcând abstracție de simetrie, iar nodul forțat este marcat cu o bulină gri deschis, acesta fiind un nod obligatoriu pentru drumul curent, din cauza obstacolului marcat cu gri închis. Acest fapt se datorează imposibilității de a găsi un drum de cost mai mic folosind ceilalți vecini.

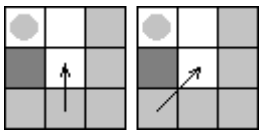


Figura 5. Nod forțat – sursa ilustrare proprie

```
Funcția salt(u, d, s, g) este
    n ← pas(u, d)
    Dacă n e obstacol sau în afara terenului atunci
        Returnează null
    Sfârşit – dacă
    Dacă n = g atunci
        Returnează n
    Sfârşit – dacă
    Dacă există n' în vecini(n) a.i. n' este forțat atunci
        Returnează n
    Sfârşit – dacă
    Dacă d este diagonală atunci
        Dacă salt(n, (0, d_1), s, g) ≠ null atunci
            Returnează n
        Sfârşit – dacă
        Dacă salt(n, (d_0, 0), s, g) ≠ null atunci
            Returnează n
        Sfârşit – dacă
    Returnează salt(n, d, s, g)
Sfârşit – funcție
```

Această metodă găsește soluția optimă reducând timpul de rulare cu până la un ordin de magnitudine. Deşi 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.

1.2.6. 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.

Vom defini în continuare elemente de bază pentru a descrie mai pe larg algoritmul.

Presupunând o matrice dreptunghiulară fiind terenul de deplasare al robotului, definim o celulă a matricei $A_{m\times n} = \{a_{ij}\}$, $a_{ij} = \{0, 1\}$, 1 fiind obstacol și inaccesibilă, 0 fiind traversabilă. Vom spune că celula a_{ij} se află în partea stângă (partea dreaptă) a altei celule a_{lk} dacă $j < k$ (respectiv $j > k$). Celula a_{ij} vom spune că este localizată mai jos (mai sus) decât celula a_{lk} dacă $i < l$ (respectiv $i > l$).

Vom înțelege prin traiectoria nulă a două celule diferite a_{ij} și a_{lk} o secvență de celule adiacente $tr(a_{ij}, a_{lk}) = \{a_{i_0 j_0}, a_{i_1 j_1}, \dots, a_{i_r j_s}\}$ și:

1. $a_{i_0 j_0} = a_{ij}, a_{i_r j_s} = a_{lk}$
2. Dacă a_{ij} nu este localizată în dreapta celulei a_{lk} atunci $a_{i_v j_w}$ nu se află în dreapta celulei $a_{i_{v+1} j_{w+1}}$, pentru $\forall 0 \le v < r, 0 \le w < s$
3. Analog pentru stânga: dacă a_{ij} nu este localizată în stânga celulei a_{lk} atunci $a_{i_v j_w}$ nu se află în stânga celulei $a_{i_{v+1} j_{w+1}}$, pentru $\forall 0 \le v < r, 0 \le w < s$
4. Dacă $k \neq j$, atunci $\forall a_{i_v j_w} \in tr(a_{ij}, a_{lk})$ satisface relația $i_v = \lfloor K j_w + B \rfloor, 0 \le v \le r, 0 \le w \le s$ unde $K = \frac{l-i}{k-j}$, $B = 1 - kK$
5. Dacă $k = j$, atunci $\forall a_{i_v j_w} \in tr(a_{ij}, a_{lk})$ satisface relația $j_w = j = k, i_v = i_{v-1} + \beta$, pentru $\forall 0 < v \le r, 0 \le w \le s$, unde $\beta = -1$ dacă a_{ij} se află mai sus de a_{lk}, $\beta = 1$, dacă a_{ij} se află mai jos de a_{lk}

Traiectoria nulă este, pe scurt, un segment compus din celule cu capetele în a_{ij} și a_{lk}. Observăm din acestea că traiectoria nulă are costul egal cu costul drumului determinat de aceasta și același cu euristica diagonală *h*. Putem, deci, să spunem că secțiunea de drum de la a_{ij} la a_{lk} este traversabilă dacă și numai dacă traiectoria nulă $tr(a_{ij}, a_{lk})$ conține doar celule traversabile. Aplicând recursiv această condiție, putem reduce problema găsirii unui drum între cele două puncte la problema găsirii unei secțiuni traversabile începând de la traiectoria nulă.

Algoritmul începe cu drumul parțial format de algoritmul lui Bresenham, ce determină discretizarea liniilor între două puncte, și menține o listă de drumuri candidate. La fiecare

iterație se face o selecție de celule a_{ij} și a_{lk} pentru care se determină traversabilitatea traiectoriei nulă aferente. Acest algoritm se oprește când găsește un drum candidat care este traversabil complet, adică toate selecțiile de celule a_{ij} și a_{lk} făcute sunt secțiuni traversabile și celulele a_{ij} și a_{lk} selectate au fost și v_{start} și v_{dest}.

Cazul în care traiectoria nulă nu este traversabilă îl rezolvăm prin căutarea unor rute ocolitoare. Astfel, vom găsi o celulă a_{ij} din drumul selectat ce se află cel mai aproape de obstacol, de la care drumul curent se bifurcă. Vom avea patru celule a, b, c, d ce delimitează obstacolul și care vor forma două rute ocolitoare. În funcție de direcția de deplasare vor fi formate fie secțiunle $\langle a, b\rangle$ și $\langle c, d\rangle$, fie $\langle a, d\rangle$ și $\langle b, c\rangle$.

În figura alăturată este ilustrat un prim pas al algoritmului, astfel că este detectat un obstacol și sunt obținute două rute ocolitoare. Pentru a determina cele 4 celule se poate folosi un simplu algoritm greedy care să găsească cel mai mic dreptunghi care încadrează obstacolul fără să îl intersecteze și este împărțită traiectoria drumului selectat în secțiuni, conform celulelor determinate. Presupunem în continuare prin simetrie că secțiunile luate în considerare sunt $\langle a, b\rangle$ și $\langle c, d\rangle$ și construim rutele ocolitoare:

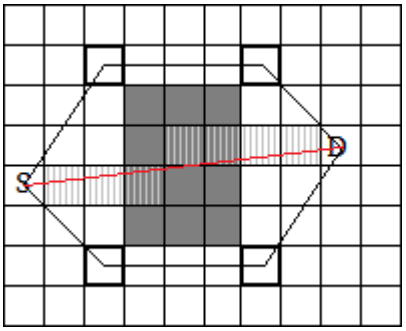


Figura 6. Rute ocolitoare – sursa prelucrare proprie

$PP_1 = \{v_{start}, \dots, a_{ij}, a, b, \dots, v_{dest}\}$ și $PP_2 = \{v_{start}, \dots, a_{ij}, c, d, \dots, v_{dest}\}$

Aceste două rute vor fi inserate în lista de candidați în locul drumului selectat inițial. Procedând în acest fel, algoritmul va menține întotdeauna o lista de drumuri ce au capetele în cele două noduri ale parcurgerii, mai mult decât atât, folosind un criteriu de sortare, acest algoritm va selecta drumul cel mai promițător din lista de candidați. La orice oprire a algoritmului, acesta are selectat cel mai promițător drum, și este chiar unul spre destinație.

Prezentarea acestui algoritm este făcută pentru mișcări într-un plan bidimensional, însă acesta poate fi extins folosind o matrice cu mai multe dimensiuni pentru a stoca terenul. În practică acest lucru nu este necesar, terenul pe care circulă un robot fiind deseori un plan cu denivelări, ce pot fi reprezentate printr-o matrice de înălțimi.

Eficiența algoritmului aduce și o problemă ce constă în faptul că deciziile pe care robotul le ia în privința traseului folosind acest algoritm trebuie validate astfel încât să nu îl direcționeze spre un obstacol, sau pe un drum fără întoarcere (dacă este cazul).

Conform unui studiu[19] acest algoritm găsește un drum apropiat de cel optim efectuând, în general, mai puțini pași decât A* ponderat chiar și cu ponderea 5, având o rație de eficiență de procesare / cost ce ajunge până la 1% din rația algoritmului A*. O altă lucrare pe același subiect este cea Near Optimal Hierarchical Path-finding[1].

1.2.7. 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 ușoară 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.

$$rhs(s) = \begin{cases} 0, & s = s_{start} \\ min_{s' \epsilon Pred(s)}(g(s') + c(s', s)), & altfel \end{cases}$$

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

Cheia pentru coada de priorități o formăm din două valori: $k(s) = [k_1(s); k_2(s)]$, unde $k_1(s) = \min\big(g(s), rhs(s)\big) + h(s, s_{dest})$ și $k_2(s) = \min(g(s), rhs(s))$, 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 v_{start}, 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 [min(g(u), rhs(u)) + h(u); min(g(u), rhs(u))]
Sfârșit – funcție
```

```
Funcția initializare este
        Fie U ← ∅ coada de priorități
        Pentru u ∈ V execută
                u.rhs ← u.g ← ∞
        Sfârșit – pentru
        v_start.rhs ← 0
        U.adauga(v_start, [h(v_start); 0])
Sfârșit – funcție
```

```
Funcția actualizareVarf(u) este
    Dacă u ≠ v_start atunci
        u.rhs ← min_{v∈Pred(u)}(g(v) + c(v,u))
    Sfârșit – dacă
    Dacă u ∈ U atunci
        U.sterge(u)
    Sfârșit – dacă
    Dacă u.g ≠ u.rhs atunci
        U.adauga(u, calculeazaCheie(u))
    Sfârșit – dacă
Sfârșit – funcție
```

```
Funcția calculeazaDrumMinim este
    Cât timp U.cheieMinima() < calculeazaCheie(v_dest) sau v_dest.rhs ≠ v_dest.g
execută
        u ← U.extrageMin()
        Dacă u.g > u.rhs atunci
            u.g ← u.rhs
            Pentru v ∈ succ(u) execută actualizareVarf(v)
        Altfel
            u.g ← ∞
            Pentru v ∈ succ(u) ∪ {u} execută actualizareVarf(v)
        Sfârșit – dacă
    Sfârșit – cât timp
Sfârșit – funcție
```

```
Funcția main este
    initializare()
    Cât timp v_start ≠ v_dest execută
        calculeazaDrumMinim()
        Deplasează robotul conform drumului găsit și așteaptă modificări de
costuri de muchii
        Pentru toate muchiile (u, v) cu cost modificat execută
            actualizareVarf(u)
            actualizareVarf(v)
        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.

Acest algoritm este prezentat mai în detaliu în lucrările „Lifelong Planning A*”[10] și „A Generalized Framework for Lifelong Planning A* Search”[11].

1.2.8. D*

Pornind de la algoritmul LPA* putem optimiza rularea lui inversând parcurgerea, și anume de la v_{dest} la v_{start}, de această dată v_{start} modificându-se odată cu poziția curentă a robotului. În acest fel algoritmul D* devine direcționat de scop, adică sosirea în nodul destinație v_{dest} cât mai repede. Detalii pot fi găsite în lucrarea Fast Replanning for Navigation in Unknown Terrain[12].

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:

- $h(v_{start}) = 0$
- $h(v) \leq h(u) + cost(u, v)$, pentru orice vârf $v \in V$ și $u \in Pred(v)$

Deoarece nodul de start va fi modificat, aceste condiții pentru *h* trebuie îndeplinite pentru orice $v_{start} \in V$ vârf.

Pseudocod:

```
Funcția calculeazaCheie(u) este
        Returnează [min(g(u), rhs(u)) + h(u); min(g(u), rhs(u))]
Sfârşit – funcție
```

```
Funcția initializare este
        Fie U ← ∅ coada de priorităţi
        Pentru u ∈ V execută
                u.rhs ← u.g ← ∞
        Sfârşit – pentru
        v_dest.rhs ← 0
        U.adauga(v_dest, calculeazaCheie(v_dest))
Sfârşit – funcție
```

```
Funcția actualizareVarf(u) este
    Dacă u ≠ v_dest atunci
        u.rhs ← min_{v∈Succ(u)}(g(v) + c(u, v))
    Sfârșit – dacă
    Dacă u ∈ U atunci
        U.sterge(u)
    Sfârșit – dacă
    Dacă u.g ≠ u.rhs atunci
        U.adauga(u, calculeazaCheie(u))
    Sfârșit – dacă
Sfârșit – funcție
```

```
Funcția calculeazaDrumMinim este
    Cât timp U.cheieMinima() < calculeazaCheie(v_dest) sau v_start.rhs ≠
v_start.g execută
        u ← U.extrageMin()
        Dacă u.g > u.rhs atunci
            u.g ← u.rhs
            Pentru v ∈ Pred(u) execută actualizareVarf(v)
        Altfel
            u.g ← ∞
            Pentru v ∈ Pred(u) ∪ {u} execută actualizareVarf(v)
        Sfârșit – dacă
    Sfârșit – cât timp
Sfârșit – funcție
```

```
Funcția main este
    initializare()
    calculeazaDrumMinim()
    Cât timp v_start ≠ v_dest execută
        v_start ← arg min_{v∈Succ(v_start)}(g(v) + c(v_start, v))
        Deplasează robotul la v_start și caută muchii cu costul modificat
        Pentru toate muchiile (u, v) cu cost modificat execută
            actualizareVarf(u)
            actualizareVarf(v)
        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.

Algoritmi asemănători sunt descriși în lucrarea „An incremental alghorithm for a generalization of the shorthest-path problem”[15], printre care se numără și dynamic SWSF.

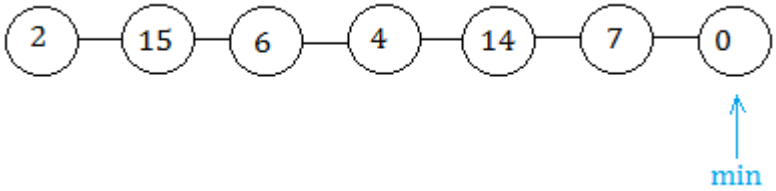
1.3. Structuri de date necesare

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 $O(n)$ la extragere.

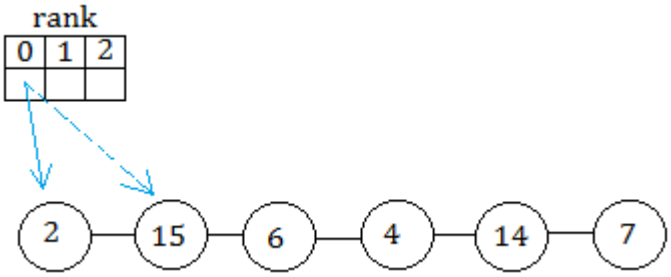
Printre metodele folosite de obicei se află Binary Heap, fiind un arbore binar complet, care satisface proprietatea ca orice nod să fie mai mic sau egal decât copiii lui. Complexitatea operațiilor este $O(\log n)$ pentru inserare, extragere de minim și modificare de prioritate.

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 A_i va avea maxim 2^{i-1} noduri dispuse în sub-arbori de adâncimi de la 0 la i-1, pentru orice $i = \overline{1,n}$. 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 – $O(1)$ – lăsând reordonarea nodurilor pentru alte operații. Această abordare nu influențează rezultatul, deoarece la extragerea minimului se consolidează structura, operația executându-se în timp amortizat $O(rank(H)), rank(H) \leq \log n$ reprezentând adâncimea unui arbore oarecare din listă.

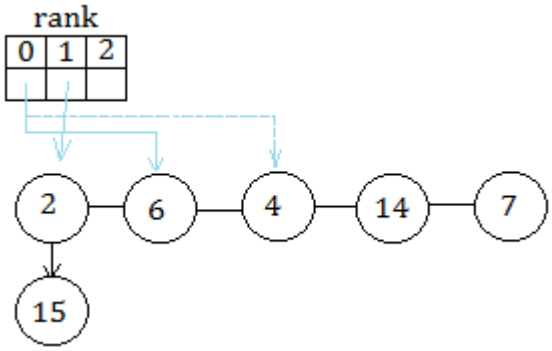
Pentru a înțelege mai bine această structură vom studia în continuare un exemplu de extragere a minimului. Întrucât operația de inserare nu face altceva decât să introducă valoarea în lista de rădăcini, presupunând că am inserat prioritățile 2, 15, 6, 4, 14, 7, 0, avem structura:



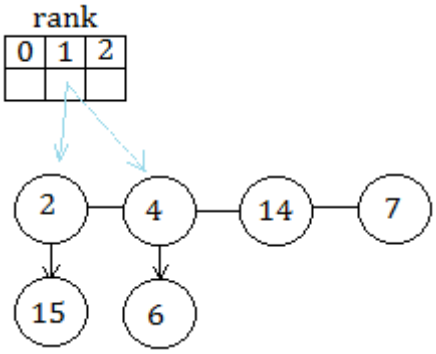
Apelând metoda de extragere de minim, activăm totodată și mecanismul de consolidare a structurii, care va menține un vector de referințe, ce vor fi legate la un nod rădăcină respectiv rangului corespunzător indexului vectorului. În figura alăturată avem ilustrat un conflict de adresare la rank, și îl vom rezolva prin mutarea nodului mai puțin prioritar (15) ca fiind copil al celui mai prioritar (2).

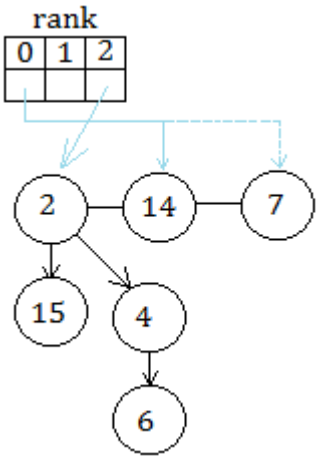


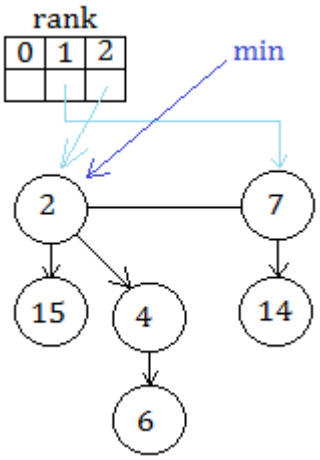
Astfel, am creat un arbore de rang = 1 (adâncime), însă există în continuare un conflict pentru rang = 0. Vom scoate nodul (6) din lista de rădăcini pentru a-l introduce în lista de copii a nodului (4).



În figurile următoare vedem pașii evidenți până când nu mai există două noduri rădăcină de rang egal.







Și în final setăm și referința către minim. Avem astfel două rădăcini cu adâncimea maximă 2 și cu maxim 2 copii.

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

```
Funcție inserare(nod, prioritate) este
        x ← (nod, prioritate)
        radacini.adauga(x)
        Dacă prioritate < m.prioritate atunci
                m ← x
        Sfârșit – dacă
Sfârșit – funcție
```

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

- $radacini$ o listă (dublu înlănțuită) ce conține rădăcinile arborilor ce se vor forma
- m (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 extrageMin este
    x ← m
    radacini.sterge(m)
    Pentru n ∈ x.copii execută
        radacini.adauga(n)
    Sfârșit – pentru
    m ← arg min_{n ∈ radacini}(n.prioritate)
    Fie k un vector de adâncimi
    Pentru n ∈ radacini execută
        i ← adancime(n)
        Dacă k_i = null atunci
            k_i ← n
            Treci la următorul
        Sfârșit – dacă
        Dacă k_i.prioritate < n.prioritate atunci
            k_i.copii.adauga(n)
            radacini.sterge(n)
        Altfel
            n.copii.adauga(k_i)
            radacini.sterge(k_i)
            k_i ← n
        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 dorim ca structura rezultată în urma consolidării să aibă 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 ștergere din coadă, pe care îl vom implementa folosind o metodă mai generală, și anume *scadereCheie* ce va permite modificarea cheii unui element la $-\infty$, 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 aibă 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 tăierea copiilor 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 $scadereCheie(\text{x}, pNoua)$ este
    $x.prioritate \leftarrow pNoua$
    Dacă $pNoua < m.prioritate$ atunci
        $m \leftarrow x$
    Sfârșit – dacă
    Dacă *x* nu are părinte atunci STOP
    Dacă $x.prioritate \geq x.parinte.prioritate$ atunci STOP
    $marcheaza(x.parinte)$
    $x.parinte.copii.scoate(x)$
    $x.parinte \leftarrow null$
    $radacini.adauga(x)$
Sfârșit – funcție
Funcție $marcheaza(nod)$ este
    Dacă $nod.parinte = null$ atunci STOP
    Dacă $nod.marcat = Adevărat$ atunci
        $nod.parinte.copii.scoate(nod)$
        $nod.parinte \leftarrow null$
        $radacini.adauga(nod)$
        $marcheaza(nod.parinte)$
    Sfârșit – dacă
    $nod.marcat \leftarrow Adevărat$
Sfârșit – funcție

Capitolul II. 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.

2.1. Librării și instrumente

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.

2.1.1. 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.

21.2. 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  = {};        var obj = new Set();          //se creează un obiect
obj[x] = true;        obj.add(x);                   //se setează o valoare
delete obj[x];        obj.delete(x);                //se șterge o valoare
if(x in obj){  … }    if(obj.has(x)){ … }           //se verifică existența unei valori
```

2.1.3. 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. Sunt explicate elemente de detaliu în lucrările 3D graphics on the web: A survey[4] și HTML5 And WebGL Fit Interactive Embedded Application[18] și în cartea WebGL Game Development[17].

2.1.4. Three.JS

Three.JS[3] 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, cuaternioane, 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 );
```

Am folosit astfel multe dintre elementele pe care le pune la dispoziție 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[4], cum ar fi generare de matrice simplex cu transformarea acesteia în geometrie.

---

[3] http://threejs.org/

[4] 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:

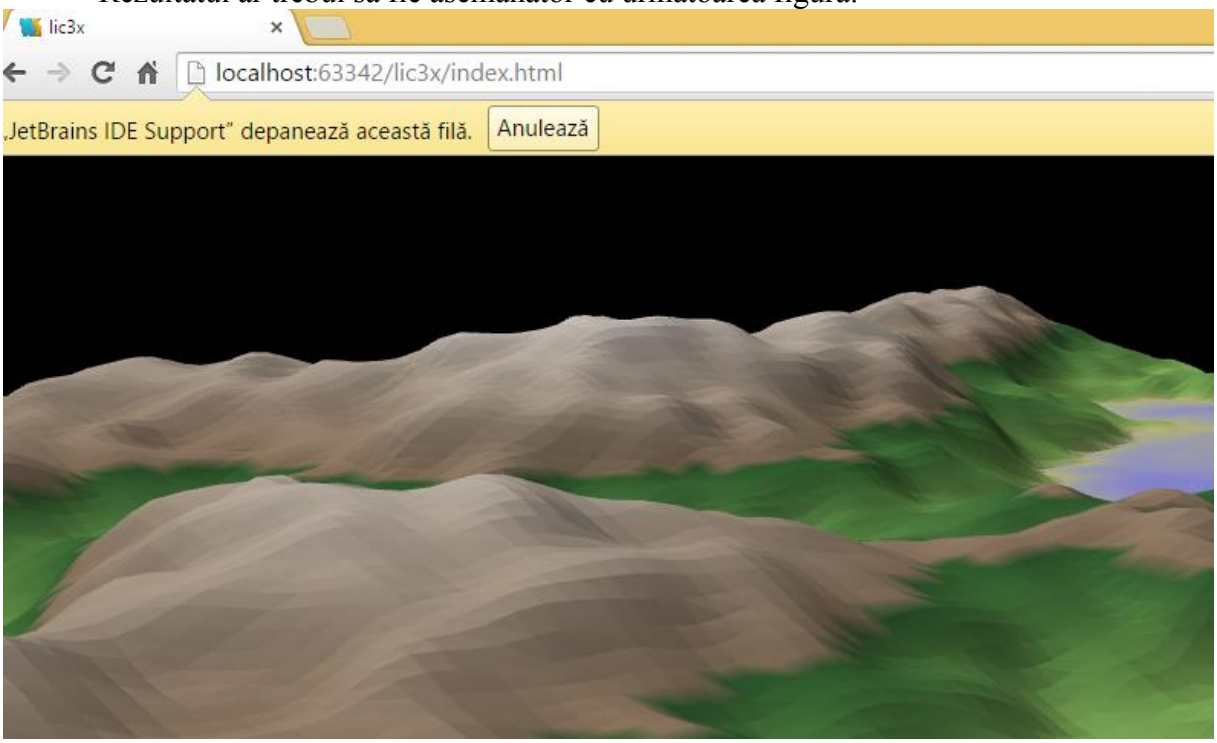


Figura 7. Sursa – prelucrare proprie

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.

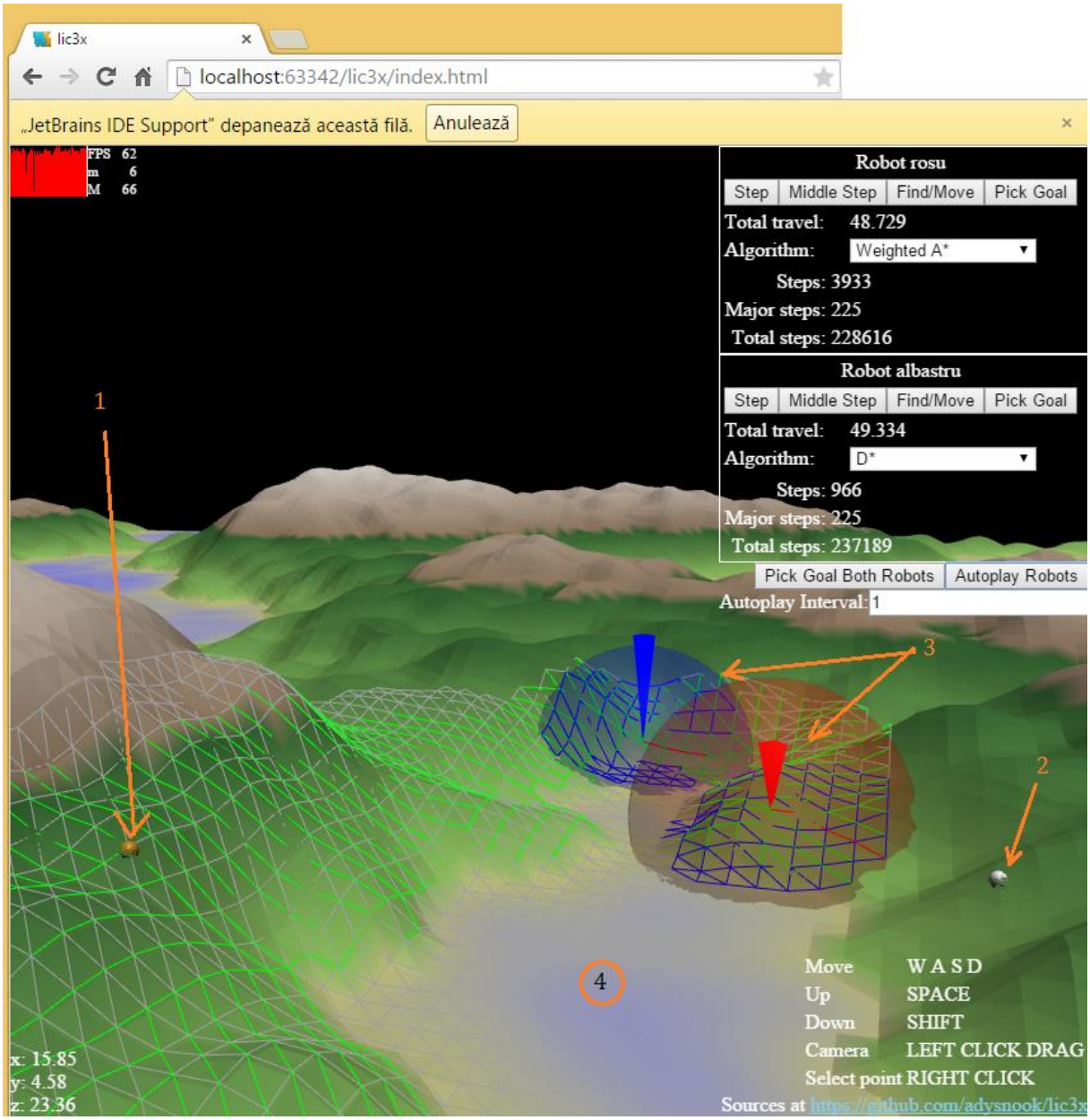


Figura 8. Ilustrarea aplicației – sursă prelucrare proprie

În figura 8 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 semnificând:

- 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 evidență 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.

2.1.5. 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.

2.1.6. jQuery

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[5] include, dar nu se rezumă la:

- Operații cu elementele din document, cum ar fi traversarea arborelui de elemente și accesarea 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.

[5] https://jquery.com/

2.1.7. JetBrains WebStorm

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. Procesul de dezvoltare poate fi simplificat folosind instrumente profesionale web, cum ar fi editorul WebStorm[6] 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 înțeleg care era cauza foarte ușor, punând unul sau mai multe breakpoint-uri în cod și analizând stările prin care trecea aplicația.

2.1.8. Git

Pe parcursul dezvoltării aplicației pot apărea probleme cum ar fi ștergerea unui cod corect sau a unor linii de cod corecte și modificarea eronată a unei funcționalități a aplicației, dar și regăsirea ultimei versiuni stabile a aplicației. 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[7], 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).

Mai mult decât atât, am folosit și GitHub[8], 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.

---

[6] https://www.jetbrains.com/webstorm/
[7] https://git-scm.com/
[8] https://github.com/

2.2. 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.

2.2.1. Mediul de operare

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 $+\infty$ 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 indicele 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 chiar dacă este încă experimentală.

2.2.2. Roboții

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 indicele 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;
    }
}
```

2.2.3. Algoritmul lui Dijkstra cu Fibonacci Heap

Algoritmul lui Dijkstra ce folosește o coadă de priorități, având la bază structura Fibonacci Heap, l-am implementat astfel:

Pentru starea 0 am executat inițializări:

```
case 0:
    this.nqs = [];
    this.queue = new PriorityQueue(this._compare);
    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;
    var stepCount = this.queue.insertUpdate(cq, 0);
    this.stepCount += stepCount; this.totalStepCount += stepCount;
    this.visited.add(cq.ki);
    this.state = 1;
    this.stepCount = 1;
    return true;
```

În acest fragment de cod se inițializează coada de priorități, se introduce nodul de start în coadă, se setează contori și alte variabile necesare, și se trece la starea următoare, ce reprezintă parcurgerea cozii.

În starea 1 se verifică dacă există elemente în coada de priorități. Dacă nu există, atunci se consideră ca nu există drum dacă cel mai bun găsit la momentul actual are costul infinit sau se trece la starea 3 pentru a determina drumul.

```
case 1:
    if (this.queue.length == 0) {
        this.state = 3;
        this.pathrev = this.bestF;
        if this.bestF.c == Number.POSITIVE_INFINITY) {
            console.info("Dijkstra Fibonacci: blocked!");
            return false; }
        return true; }
    var qp = this.queue.pop();
    this.cq = qp.value;
    this.stepCount += qp.stepCount; this.totalStepCount += qp.stepCount;
    this.iter = this.robot.known_v[this.cq.ki].neighbors.values();
    this.state = 2;
    return true;
```

Starea 2 este cea care parcurge vecinii nodului explorat și îi evaluează pentru a-i introduce în coadă.

```
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(typeof this.nqs[nki] == "undefined"){
            this.nqs[nki] = nq; }else{
            if(nq.c >= this.nqs[nki].c){ return true; }
            this.nqs[nki] = nq; }
        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.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);
var stepCount = this.queue.insertUpdate(nq, nq.c);
this.stepCount += stepCount; this.totalStepCount += stepCount;
if (this.robot.bridgeVertices.has(nki)) {
if (this.bestF == null || this.vertexH(ngi) + nq.c <
this.vertexH(this.robot.known_v[this.bestF.ki].g_v_idx) + this.bestF.c)
{ this.bestF = nq; } } } } }
    return true;
```

În această stare este verificată condiția de terminare, modificată astfel încât să țină cont de drumurile posibile, dar încă necunoscute din cauza vizibilității. Acest lucru este realizat cu ajutorul verificării: *costul până la vecinul explorat este mai mic sau egal decât valoarea celui mai bun nod vizibil parțial prin funcția* $f(n) = \delta(v_{start}, n) + h(n)$. Cu alte cuvine, dacă nu există nici un nod vizibil parțial care să aibă această valoare mai bună decât costul prin vecinul explorat, iar acest vecin este chiar v_{dest} atunci acest drum sigur este cel mai scurt. Condiția detaliată este în cod **if**(**this**.**goal** == ngi){ **if**(**this**.**bestF** == **null** || nq.**c** <= **this**.**robot**.**known_v** [**this**.**bestF**.**ki**].**h** + **this**.**bestF**.**c**)

Tot în starea 2 este introdus vecinul inspectat în coada de priorități, cu prioritatea fiind costul de a ajunge până la acesta. Dacă există deja, i.e. a fost găsit un drum la o explorare anterioară, atunci verificăm dacă drumul găsit prin nodul curent este mai bun, iar în caz contrar sărim peste acest vecin.

Starea 3 execută găsirea drumului folosind legăturile „înapoi" ale nodurilor, aceste legături fiind referință la nodul din care se ajunge pe un drum de cost minim. În starea 4 este mutat robotul și resetat algoritmul pentru următorul pas, întrucât se va modifica graful în urma schimbării vizibilității robotului. Aceste două stări sunt asemenea celor respective din implementarea algoritmului A* ponderat și vor fi detaliate în acel cadru.

2.2.4. Algoritmul A* ponderat

Pentru implementarea algoritmului A* ponderat am folosit parte din codul algoritmului lui Dijkstra, astfel că unele cazuri asemănătoare le-am tratat identic.

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 creează coada vidă, mulțimea de noduri vizitate, se adaugă nodul de start în coadă, se resetează numărul de pași și se trece la următoarea 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; }
    var minC = 0;
    for(var i=1; i<this.queue.length; ++i){
        ++this.stepCount;
        var ingi = this.robot.known_v[this.queue[i].ki].g_v_idx;
        var mngi = this.robot.known_v[this.queue[minC].ki].g_v_idx;
        if(this.queue[i].c + this.vertexH(ingi) *this.eps
<this.queue[minC].c + this.vertexH(mngi) *this.eps){
            minC = i; } }
    this.cq = this.queue[minC];
    this.queue.splice(minC, 1);
    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.

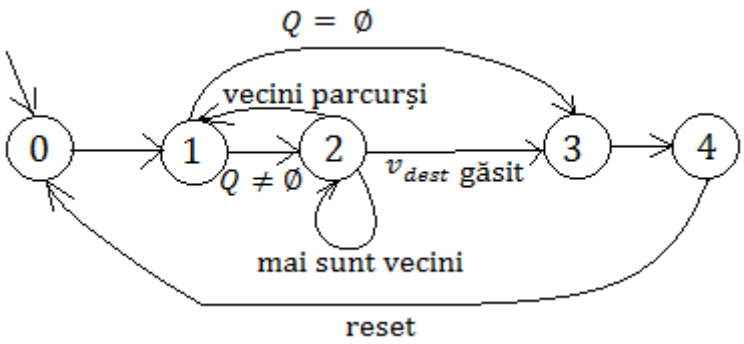


Figura 9. Diagrama automatului determinat – sursă prelucrare proprie

2.2.5. Algoritmul D*

Algoritmul D* l-am împărțit pe 5 cazuri, însă spre deosebire de algoritmii prezentați anterior, această implementare va executa căutarea într-o singură stare, celelalte fiind pentru procesarea vârfurilor modificate sau inițializare.

În prima stare (0), am apelat funcția de inițializare și de căutare a algoritmului, pe care le voi descrie ulterior.

```
case 0:
    this.s_last = this.s_start;
    this._init();
    this._computeShortestPath();
    this.state = 1; ++this.middleStepCount; ++this.majorStepCount;
    this._colorShorthesPath();
    return true;
```

```
case 1:
    var rStart = this.robot.known_v[this.s_start];
    if (this.s_start == this.s_goal) {
        console.log("D*: finished (goal)");
        this.state = -2;
        return false; }
    if (rStart.g == Number.POSITIVE_INFINITY) {
        console.info("D*: Blocked!"); }
    var nextV = null;
    var nextMin = null;
    for (sPrim of rStart.neighbors) {
        ++this.stepCount;
        ++this.totalStepCount;
        var nsPrim = this.robot.neighborV(sPrim, this.s_start);
        var rsPrim = this.robot.known_v[nsPrim];
        var nowMin = this._c(this.s_start, nsPrim) + rsPrim.g;
        if (nextMin == null || nowMin < nextMin) {
            nextV = nsPrim;
            nextMin = nowMin; } }
    this.updatedVertices = new Set();
    this.state = 2;
    this.s_start = nextV;
    this.robot.moveTo(nextV);
    this.stepCount = 1;
    ++this.middleStepCount; ++this.majorStepCount;
    return true;
```

În starea cu numărul 1 sunt efectuate validări de terminare, iar dacă nu este cazul de oprire este mutat robotul și pregătită următoarea stare. Pentru a determina nodul în care se va face mutarea robotului am implementat o parcurgere a vârfurilor vecine lui v_{start} și îl alegem pe cel cu cea mai mică valoare a drumului găsit prin acesta. Tot aici inițializez și mulțimea de vârfuri actualizate **this.updatedVertices** = **new** Set(); pentru a fi populată de către entitatea robot la modificările posibil apărute în urma mutării.

În starea ce urmează (2), se verifică existența nodurilor actualizate, iar dacă nu este cazul drumul rămâne cel deja determinat și starea curentă trece din nou la 1 pentru mutare. Altfel, este pregătită parcurgerea mulțimii de noduri actualizate pentru starea 3.

```
case 2:
    if (this.updatedVertices.size == 0) {
        ++this.middleStepCount;
        ++this.majorStepCount;
        this.state = 1;
        this._colorShorthesPath();
        return true;
    }
    this.k_m += this._h(this.s_last, this.s_start);
    this.s_last = this.s_start;
    this.iter = this.updatedVertices.values();
    this.state = 3;
    return true;
```

În starea 3 este apelată funcția de actualizare a nodului **this**._updateVertex(next.**value**) descrisă după starea 4.

```
case 3:
    var next = this.iter.next();
    if (next.done) {
        this.state = 4;
        ++this.middleStepCount;
    } else {
        this._updateVertex(next.value);
    }
    return true;
```

În final, starea 4 reprezintă recalcularea drumului după actualizarea nodurilor, în urma căreia trebuie sa rezulte un drum de cost minim date fiind informațiile cunoscute.

```
case 4:
    this._computeShortestPath();
    ++this.middleStepCount;
    ++this.majorStepCount;
    this.state = 1;
    this._colorShorthesPath();
    return true;
```

Cea mai importantă funcție a acestui algoritm, calcularea drumului minim, este prezentată în cele ce urmează.

```
_computeShortestPath: function () {
    var topKey = this.U.peek().priority;
    this._vertexCheckILF(this.s_start);
    var rStart = this.robot.known_v[this.s_start];
    var k_old; var u; var ru; var gu;
    while (!this.U.isEmpty() && (this._compare(topKey,
this._calcKey(this.s_start)) || rStart.rhs != rStart.g)) {
        ++this.stepCount; ++this.totalStepCount;
        k_old = topKey;
        var pop = this.U.pop();
        this.stepCount += pop.stepCount;
        this.totalStepCount += pop.stepCount;
        u = pop.value;
        this._vertexCheckILF(u); ru = this.robot.known_v[u];
        gu = ru.g_v_idx;
        if (this._compare(k_old, this._calcKey(u))) {
            var stepCount = this.U.insertUpdate(u, this._calcKey(u));
            this.stepCount += stepCount;
            this.totalStepCount += stepCount;
        } else if (ru.g > ru.rhs) {
            ru.g = ru.rhs;
            for (s1 of ru.neighbors) {
                ++this.stepCount;
                ++this.totalStepCount;
                var nki = this.robot.neighborV(s1, u);
                var ngi = this.robot.known_v[nki].g_v_idx;
                this._updateVertex(nki);
                this.robot.colorEdge(gu, ngi, 0x00FF00); }
            if (u == this.s_goal) {
                for (s1p of this.robot.bridgeVertices) {
                    ++this.stepCount;
                    ++this.totalStepCount;
                    this._updateVertex(s1p); } }
        } else {
            ru.g = Number.POSITIVE_INFINITY;
            for (s2 of ru.neighbors) {
                ++this.stepCount;
                ++this.totalStepCount;
                var nki = this.robot.neighborV(s2, u);
                var ngi = this.robot.known_v[nki].g_v_idx;
                this._updateVertex(nki);
                this.robot.colorEdge(gu, ngi, 0x00FF00);
            }
            if (u == this.s_goal) {
                for (s1p of this.robot.bridgeVertices) {
                    this._updateVertex(s1p);
                }
            }
            this._updateVertex(u); }
        if (!this.U.isEmpty()) {
            topKey = this.U.peek().priority; } } }
```

În această funcție este parcursă coada de priorități, iar pentru fiecare vârf este apelată funcția de actualizare dacă este cazul. Actualizarea trebuie rulată pentru vecinii nodului dacă acesta are cheia modificată, i.e. s-a modificat valoarea *g*, *rhs* sau *h* a nodului. Dacă valoarea *g* a nodului abia extras din coadă este mai mică decât valoarea *rhs* a sa atunci trebuie actualizat și acest vârf.

Parcurgerea se termină atunci când coada este goală, sau a fost extras vârful de start, algoritmul începând cu căutarea din nodul destinație și fiind primul inserat în coada de priorități.

O altă funcție importantă, cea de actualizare a unui vârf:

```
_updateVertex: function (u) {
    this._vertexCheckILF(u);
    var ru = this.robot.known_v[u];
    if (u != this.s_goal) {
        var minRHS = null;
        var nki, rki, nowRHS;
        for (sPrim of ru.neighbors) {
            ++this.stepCount; ++this.totalStepCount;
            nki = this.robot.neighborV(sPrim, u);
            this._vertexCheckILF(nki);
            rki = this.robot.known_v[nki];
            nowRHS = this._c(u, nki) + rki.g;
            if (minRHS == null || nowRHS < minRHS) {
                minRHS = nowRHS; }
        }
        if (this.robot.bridgeVertices.has(u)) {
            nowRHS = this._h(u, this.s_goal);
            if (minRHS == null || nowRHS < minRHS) {
                minRHS = nowRHS; } }
        ru.rhs = minRHS;
    }
    var stepCount = this.U.deleteNode(u);
    this.stepCount += stepCount;
    this.totalStepCount += stepCount;
    if (ru.g != ru.rhs) {
        stepCount = this.U.insertUpdate(u, this._calcKey(u));
        this.stepCount += stepCount;
        this.totalStepCount += stepCount; } }
```

În această funcție, pentru nodurile diferite de vârful destinație, sunt parcurși vecinii pentru a calcula valoarea *rhs* a nodului primit ca parametru. Pentru a adapta acest algoritm la problema noastră, foarte des neavând un drum între punctele căutării, am ținut cont și de mulțimea de noduri vizibile parțial, **this.robot.bridgeVertices**.

Un nod care are valoarea *rhs* diferită de valoarea *g* trebuie introdus în coada de priorități cu cheia actualizată unde este cazul. Acest fapt este datorat chiar definiției funcției *rhs*, care are la bază valoarea *g* a nodului.

2.3. Testare

Am testat compatibilitatea aplicației pe unele dintre cele mai populare browsere de internet la nivelul anului 2015: Google Chrome, Microsoft Internet Explorer, Mozilla Firefox, Apple Safari și Opera. Sistemul de operare folosit la testare a fost Microsoft Windows 8.1. Din cauză că browser-ul Safari este disponibil doar pentru dispozitivele Apple nu am testat compatibilitatea și cu acest browser.

Chrome versiunea 43, versiunea pentru arhitectura 64-bit, rulează aplicația fără probleme, dar consumul de memorie RAM este foarte mare. Intuitiv, acest consum de memorie pare să apară cu scopul optimizării performanței aplicației prin menținerea de date cache – i.e. date care nu sunt necesare, dar pot fi utile pentru a nu le mai recalcula sau reîncărca. Concret, consumul pe care l-am observat este în jur de 300 - 400 MegaBytes (MB) pentru utilizarea foarte ușoară: drumuri relativ scurte de maxim 40 de vârfuri, mulțime de vârfuri mică de maxim 500 și până la 5 drumuri executate. Utilizarea medie poate face ca aplicația să meargă deranjant de încet și să aibă consumul de memorie de până la 500 - 600 MB. Cauza îngreunării este clar numărul mare de muchii afișate și pași executați deoarece vârful destinație este de obicei în afara grafului cunoscut roboților, algoritmii necesitând inițial o parcurgere completă sau aproape completă în funcție de algoritm. Utilizarea intensă este impracticabilă întrucât memoria folosită ajunge chiar și la 1000 MB, fără a se opri aici, și procesarea unei mutări poate dura câteva zeci de minute pentru a ilustra executarea a 50-100 de milioane de pași. O astfel de rulare până la nodul destinație a durat în jur de 20 de ore, având peste 16.000 de noduri în memoria roboților și peste 40.000 de muchii, pentru un drum de peste 500 de noduri.

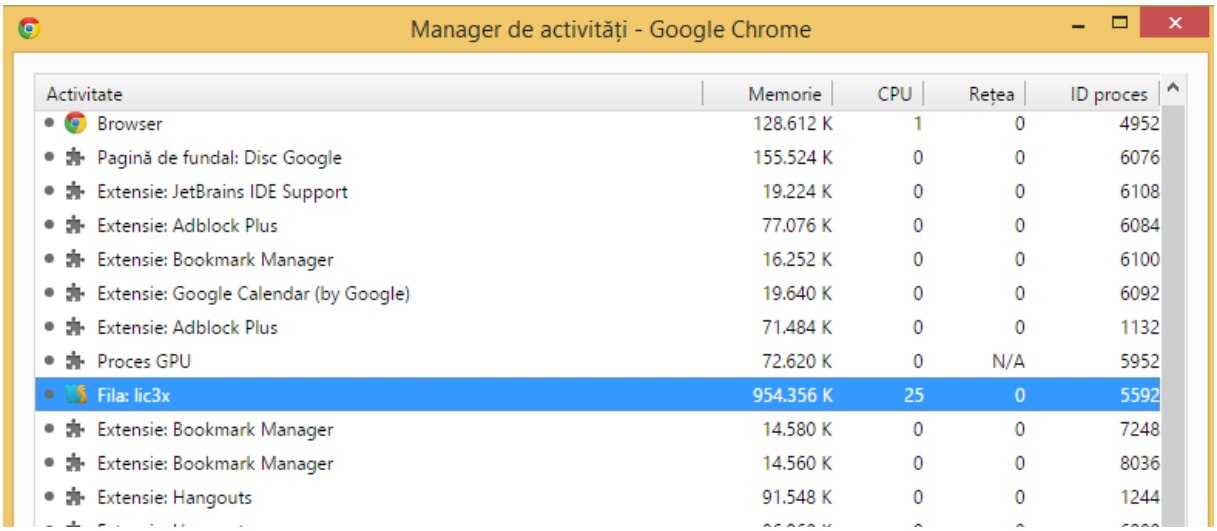


Figura 10. Memorie folosită de Chrome – sursa prelucrare proprie

Opera versiunea 29 rulează de asemenea aplicația fără probleme, însă cu o necesitate mai mică de memorie decât Chrome. La o utilizare ușoară acest browser folosește între 250 MB și 350 MB, însă la o utilizare medie spre intensă depășește inevitabil pragul de 500 MB ajungând chiar la 700 MB în cazuri extreme.

Firefox versiunea 38 încarcă pagina, însă nu o afișează corect. Această problemă poate să fie cauzată și de faptul că unele instrucțiuni folosite în aplicație nu sunt încă standardizate, fiind în forma „schiță de lucru” (din engleză working draft). Acest navigator afișează contextul 2D de lucru și elemente de interfață parțial și rulează codul sursă fără erori. Se poate observa acest lucru și din elementul din stânga sus, ce ilustrează grafic numărul de cadre afișate pe secundă, care este actualizat cu informația corectă.

Internet Explorer versiunea 11 nu are implementate noile instrucțiuni din ECMAScript 6 și drept urmare are erori la execuția scriptului.

Aplicația rulează chiar și pe smartphone. Pe dispozitive Android aplicația rulează bine în browser-ul Chrome pentru mobil, acest browser având în mare parte aceleași librării și motoare de rulare la bază. Totuși, aplicația nu poate fi utilizată pe telefon, deoarece necesită tastatură pentru a mișca poziția camerei, necesitând astfel o versiune specială pentru dispozitivele cu touch screen și fără tastatură.

Deși pe un computer de tip desktop aplicația poate fi folosită, aceasta nu are o interfață intuitivă, fiind creată mai mult cu scopul de a compara câțiva algoritmi din punct de vedere al performanței și optimalității. Instrucțiunile de folosire vor fi necesare astfel pentru majoritatea utilizatorilor.

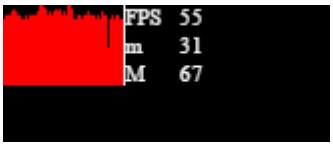


Figura 11. Cadre pe secunda – sursa prelucrare proprie

În figura alăturată este ilustrat grafic numărul de cadre pe secundă. Din câte se pare acesta este limitat la cât suportă monitorul, folosind probabil tehnica vsync – sincronizare verticală. Se poate observa numărul curent de cadre pe secundă, 55, numărul minim, 31, și numărul maxim, 67, precum și istoricul ultimelor 60 de secunde din grafic. Acest instrument este deseori util când sunt căutate probleme de performanță din cauza detaliilor excesive pe care placa video trebuie să le prelucreze. Astfel, am reușit să descopăr rapid problema cu prea multe linii afișate, menționată și soluționată în capitolul anterior. Am observat din acest grafic faptul că deși aplicația trebuia sa fie în repaus, roboții fiind opriți, când îndreptam camera spre zonele cu multe linii acest număr de cadre pe secunde scădea dramatic, ceea ce făcea utilizarea aplicației foarte incomodă.

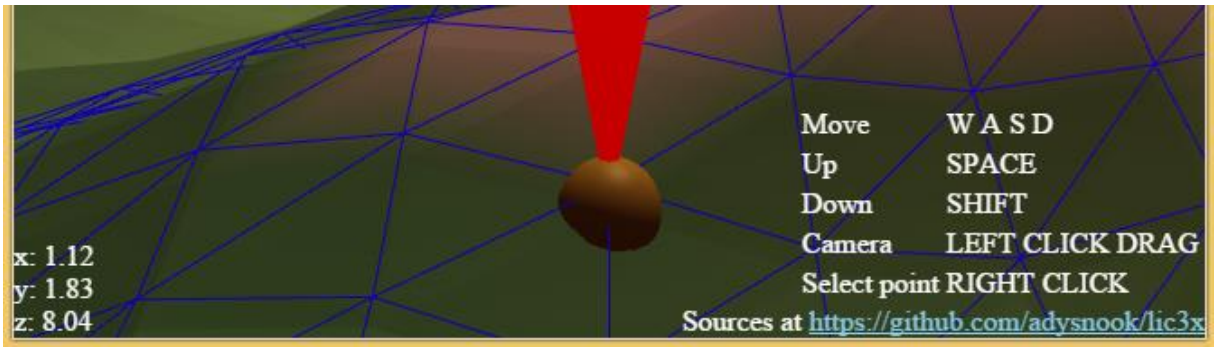


Figura 12. Informații suplimentare – sursa prelucrare proprie

În figura 12 sunt prezentate informații suplimentare. În partea stângă jos avem poziția curentă a camerei dinspre care este făcută afișarea. Această informație este utilă pentru dezvoltare, însă nu ajută la folosirea aplicației. În partea dreaptă sunt afișate câteva informații despre funcționalitatea aplicației, cum ar fi mișcarea camerei cu tastele W, A, S, D, SPACE și SHIFT și rotirea ei trăgând cu mouse-ul cu click stânga. Am pus și link către sursele aplicației pentru mai multe detalii.

Am afișat în partea dreaptă sus informații despre roboți și elemente de control ale acestora. Pentru o mai ușoară identificare, am colorat roboții în roșu și albastru. Butoanele Step, Middle Step și Find/Move controlează execuția algoritmului selectat prin executarea unui pas simplu, sau mai multora până la trecerea la un pas mijlociu sau mare. Un pas mijlociu reprezintă explorarea unui vârf în cazul algoritmului A* sau echivalentul în alte cazuri. Un pas mare este găsirea drumului sau mutarea robotului. Butoanele Pick Goal acționează începerea de alegere a vârfului destinație, necesitând alegerea efectivă prin click dreapta și finalizarea selecției acționând același buton ce nu va mai avea titlul Pick Goal ci Select Point. În afară de aceste butoane din contextul robotului am introdus și selecția nodului destinație pentru ambii roboți deodată și activarea sau dezactivarea unei funcții de autoplay. Această funcție acționează execuția unui pas mare pentru ambii roboți la fiecare cadru, dar la un interval de timp de cel puțin *x* milisecunde, unde *x* este numărul introdus în câmpul de dedesubtul butonului. Tot în dreapta sus pot fi observate informațiile despre robot și algoritm, observând în cazul de față aproape două milioane de pași până la terminarea primei căutări de drum. Observăm și drumul total parcurs aproape patru mii de unități, știind că o muchie are maxim 0,5. Ambii roboți au aceleași informații întrucât au plecat din același punct, au avut același vârf destinație, și au folosit același algoritm de căutare pentru drumurile precedente.

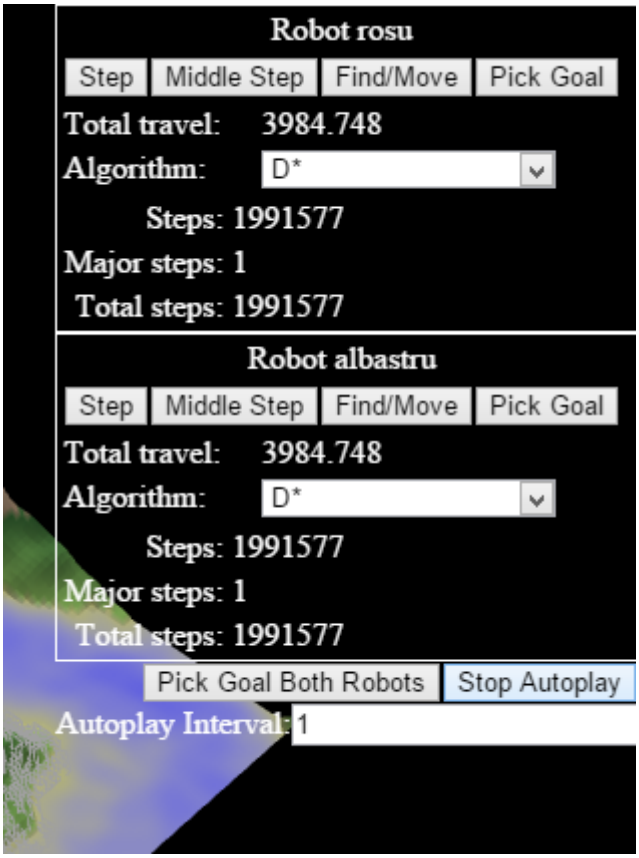


Figura 13. Interacțiune – sursa prelucrare proprie

2.4. Rezultate

După cum am arătat și teoretic eficiența și optimalitatea algoritmilor studiați este diferită de la o problemă la alta. Astfel, în practică avem următoarele rezultate:

- Dijkstra comparat cu A* ponderat pe un drum drept (fără obstacole) cu distanță mică (în jur de 50 de noduri) în teren necunoscut

	Drum parcurs	Mutări	Pași executați
Dijkstra	32,429	79	210.424

A* ponderat	32,460	79	219.349

- Dijkstra comparat cu A* ponderat pe un drum ce se află după un obstacol mediu (în jur de 50 de noduri) cu distanță medie (în jur de 200 de noduri) în teren necunoscut

	Drum parcurs	Mutări	Pași executați
Dijkstra	62,223	150	764.670
A* ponderat	62,577	150	666.271

- Dijkstra comparat cu A* ponderat pe un drum înconjurat din trei părți de obstacole (peninsulă) cu distanță mare (peste 500 de noduri) în teren necunoscut

	Drum parcurs	Mutări	Pași executați
Dijkstra	411,053	972	14.158.943
A* ponderat	438,197	1.039	13.496.452

- A* ponderat comparat cu D* pe un drum fără obstacole cu distanță mică în teren necunoscut

	Drum parcurs	Mutări	Pași executați
A* ponderat	21,924	52	112.904
D*	21,712	52	48.858

- A* ponderat comparat cu D* pe un drum cu obstacol de dimensiune medie și distanță medie în teren necunoscut

	Drum parcurs	Mutări	Pași executați
A* ponderat	82,504	193	1.085.437
D*	78,893	191	312.945

- A* ponderat comparat cu D* pe un drum înconjurat din 3 părți de obstacole, cu distanță mare în teren necunoscut

	Drum parcurs	Mutări	Pași executați
A* ponderat	928,357	2.059	54.114.822
D*	743,068	1.666	5.259.610

- A* ponderat comparat cu D* pe un drum foarte lung cu un obstacol foarte mare în teren necunoscut

	Drum parcurs	Mutări	Pași executați
A* ponderat	1.244,792	2.842	118.718.229
D*	744,944	1.668	4.842.293

- A* ponderat comparat cu D* pe un drum mic cu un obstacol de dimensiune medie în teren cunoscut mai mult de 90%

	Drum parcurs	Mutări	Pași executați
A* ponderat	36,199	87	675.679
D*	35,190	86	145.498

Algoritmul lui Dijkstra ce folosește o coadă de priorități bazată pe Fibonacci Heap găsește aceleași soluții ca algoritmul original, operația de căutare a minimului executându-se mai eficient în unele cazuri. Am observat că pentru seturi mici de date este mai ineficientă această abordare, însă pentru distanțe mari și operații multiple eficiența se îmbunătățește. Concret, pentru un drum compus din 175 de mutări (noduri), algoritmul cu Fibonacci Heap a rulat în aproximativ 19 milioane de pași, în timp ce algoritmul clasic a rulat în aproximativ 25 de milioane de pași.

Comparând implementarea pe bază de Fibonacci Heap cu alți algoritmi cum ar fi A* ponderat și D*, această diferență nu este notabilă și algoritmul rămâne inferior.

	Drum parcurs	Mutări	Pași executați
Dijkstra cu Fibonacci Heap	80,435	168	3.541.477
A* ponderat	84,593	175	1.206.569

	Drum parcurs	Mutări	Pași executați
Dijkstra cu Fibonacci Heap	106,233	204	5.954.883
D*	110,704	222	247.688

Concluzii și recomandări

După cum era de așteptat, în practică lucrurile diferă de teorie, astfel că am constatat că unii algoritmi studiați deși teoretic sunt mai eficienți sau găsesc soluții mai bune, în realitate această situație nu mai este regăsită.

Simplul fapt că robotul nu cunoaște drumul până la destinație introduce în problemă cazuri nerezolvate teoretic încă. Drept urmare, consider că acest subiect merită studiat intens, întrucât domeniile de aplicabilitate sunt foarte variate și devin tot mai interesante pentru umanitate. Printre aplicațiile reale ale problemei se numără navigarea roboților, automatizarea, și chiar și chirurgia robotizată sau mașina fără șofer. Pentru aceasta din urmă se observă interesul marilor producători de automobile cum ar fi Merceds-Benz, Toyota, Audi, Volvo, Peugeot și multe altele, chiar și Google. Este de așteptat ca printre avantaje să se numere reducerea de accidente, aglomerații și ambuteiaje, furturi și nerespectări ale legilor de circulație. Mai mult decât atât este posibil să fie crescute limitele maxime de viteză întrucât aceste vehicule ar putea procesa informația mai repede ca un om și chiar lua decizii mai bune.

Posibilele îmbunătățiri ale acestor operații de găsire a drumului minim pot fi efectuate în sisteme distribuite și precalculate, astfel încât necesitatea robotului care trebuie să se miște se transformă în necesitatea unei conexiuni fiabile la un astfel de sistem. Google Maps este un astfel de sistem prezent pe internet de mai mult de 10 ani, fiind lansat în 2005, și gratuit ceea ce împreună cu un sistem GPS este una dintre cele mai bune opțiuni pentru oricine are nevoie de ajutor în alegerea traseului. Din 2007 acest serviciu include și Google Traffic, ce afișează în timp real nivelul traficului pe străzi.

Schimbările în teren pot să apară oricând, fapt pentru care eu consider că este obligatoriu ca un robot să aibă implementat un sistem de căutare a drumului cel mai bun, astfel încât să poată lua decizii și off-line. În caz contrar există posibilitatea ca terenul să fie modificat într-un moment în care conexiunea robotului la sistemul dedicat pentru căutare să nu fie disponibilă și să intervină în felul acesta probleme.

Trebuie menționat faptul că tehnologia web este interesantă pentru foarte multă lume, astfel că evoluția acesteia este rapidă. Un exemplu concret este faptul că într-un timp relativ scurt de la lansarea WebGL au fost concepute proiecte foarte utile, unul dintre ele fiind „A web-based collaborative framework for facilitating decision making on a 3D design developing process”[14].

În concluzie, consider că a cunoaște diferențele dintre algoritmi este un lucru esențial în programarea roboților, fapt pentru care cred că este utilă această lucrare în alegerea și implementarea celui mai potrivit algoritm.

Bibliografie

1. Botea ,A. ,Müller, M, Schaeffer,J (2004). Near Optimal Hierarchical Path-finding. Journal of Game Development (Volumul 1),
2. Cormen,T.H.,(2001). Introduction to Algorithms, Second Edition
3. Ducho, F (2014). Path planning with modified A star algorithm for a mobile robot, Procedia Engineering, pp 59 – 69.
4. Evans, A., Romeo, M.,Bahrehmand, A.,(2014)-3D graphics on the web : A survey.Computers &Graphics41pp 43–61
5. Felner, A. Inconsistent heuristics in theory and practice Artificial Intelligence 175 (2011) PAG 1570–1603
6. Fisher, M.L(1980), Worst-Case Analysis of Heuristic Algorithms. Management Science, Vol. 26, No. 1 , pp. 1-17
7. Geisberger, R, (2008). Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. Institut für Theoretische Informatik Universität Karlsruhe (TH).
8. Harabor, D.,(2011). Fast Pathfinding via Symmetry Breaking
9. Harabor, D., Grastien, A.,(2011). Online Graph Pruning for Pathfinding on Grid Maps. National Conference on Artificial Intelligence (AAAI).
10. Koenig, S.; Likhachev, M.; Furcy, D. (2004.) Lifelong Planning A*. Artificial Intelligence 155(1-2) pp 93–146.
11. Koenig, S.; Likhachev, M (2005). A Generalized Framework for Lifelong Planning A* Search . American Association for Artificial Intelligence.
12. Koenig, S.; Likhachev, M.(2005). Fast Replanning for Navigation in Unknown Terrain. IEEE TRANSACTIONS ON ROBOTICS, VOL. 21, NO. 3.
13. Li Lim, K.(2015).Uninformed pathfinding: A new approach, Expert Systems with Applications 42 pp 2722–2730.
14. Nyamsurena,P. (2015). A web-based collaborative framework for facilitating decision making on a 3D design developing process. Journal of Computational Design and Engineering.
15. Ramalingam, G.(1992). An incremental alghorithm for a generalization of the shorthest-path problem.
16. Rivera, N.(2015). Incorporating weights into real-time heuristic search. Artificial Intelligence 225 pp 1–23.
17. Sumeet, A.(2014). WebGL Game Development
18. Wong, B., (2015). HTML5 And WebGL Fit Interactive Embedded Application. ProQuest
19. Yakovlev, K. S.(2010). HGA*, an Efficient Algorithm for Path Planning in a Plane. Scientific and Technical Information Processing, Vol. 37, No. 6, pp. 438–447.
20. ***http://www.ecma-international.org/ecma-262/5.1/
21. ***https://www.khronos.org/registry/webgl/specs/latest/2.0/
22. ***http://www.policyalmanac.org/games/aStarTutorial.htm
23. ***http://theory.stanford.edu/~amitp/GameProgramming/
24. ***http://en.wikipedia.org/wiki/A*_search_algorithm
25. ***http://en.wikipedia.org/wiki/A*_search_algorithm

26. ***http://en.wikipedia.org/wiki/D*
27. ***http://en.wikipedia.org/wiki/Jump_point_search
28. ***http://en.wikipedia.org/wiki/Pathfinding
29. ***http://en.wikipedia.org/wiki/Motion_planning
30. ***http://en.wikipedia.org/wiki/Shortest_path_problem
31. ***http://en.wikipedia.org/wiki/Incremental_heuristic_search
32. ***http://en.wikipedia.org/wiki/ECMAScript
33. ***http://en.wikipedia.org/wiki/JavaScript
34. ***http://www.w3.org/TR/html5/
35. ***http://zerowidth.com/2013/05/05/jump-point-search-explained.html