Raport tehnic

Algoritm de căutare euristică (A^*)

Student: Cocei Janina Constantina
Specializarea Calculatoare in limba romana
Anul 2, Semestrul al II-lea
20 Mai 2023

Contents

1 *Enuntul problemei*

Să presupunem că doi prieteni locuiesc în diferite oraşe pe o hartă. La fiecare pas, putem muta simultan fiecare prieten într-un oraş vecin de pe hartă. Timpul necesar pentru a trece de la oraşul i la oraşul vecin *j* este egal cu lungimea drumului $d(i, j)$ dintre cele 2 oraşe, dar la fiecare pas prietenul care ajunge primul trebuie să aştepte până când şi celălalt prieten ajunge (şi îl sună pe primul pe telefonul său mobil) înainte ca următoarea tură să poată începe. Vrem ca cei doi prieteni să se întâlnească cât mai repede posibil.

a. Scrieţi o formulare detaliată pentru această problemă de căutare.

b. Identificaţi un algoritm de căutare pentru această sarcină şi explicaţi alegerea dumneavoastră.

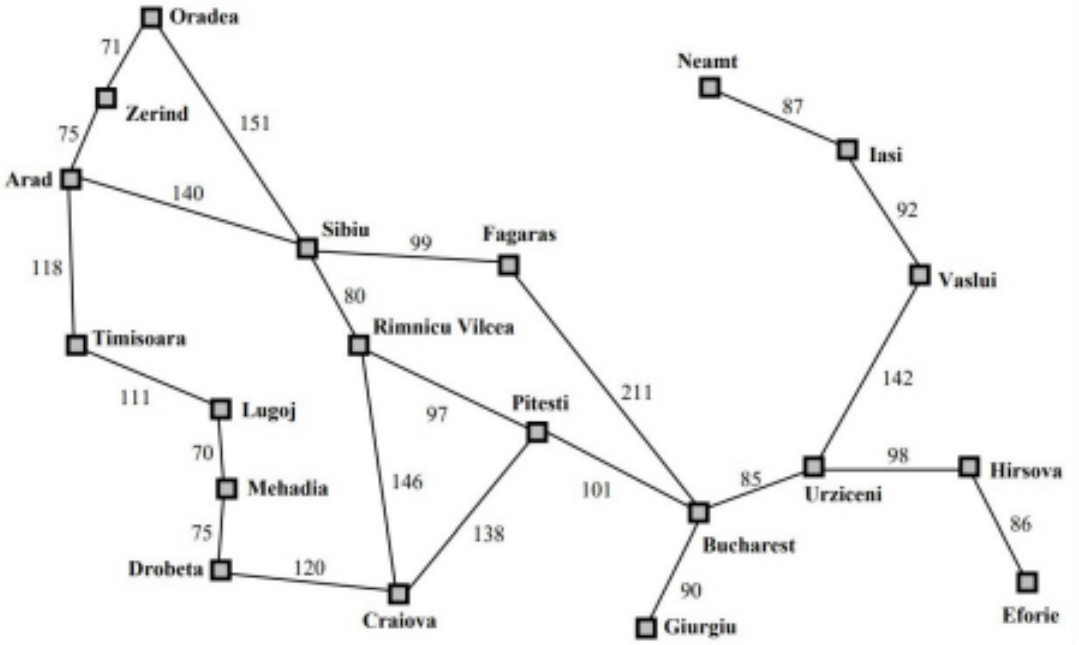

Figura 1: Romania Map

2 *Algoritm*

A*(*sursa, destinatie, graf*)

1. *lista_deschisa* = { *sursa* }
2. *lista_inchisa* = {}
3. **pentru fiecare** nod in graf **executa**
4. *calculeazaHCost* (*nod*)
5. *parinte* (*sursa*) = *null*
6. *g_cost*(sursa) = 0
7. *f_cost*(sursa) = *h*_cost (sursa)
8. **atata timp cat** *lista_deschisa* nu este goala **executa**
9. *curent* = *nodul din lista deschisa cu f_cost minim*
10. daca curent = destinatie executa
11. break
12. scoate curent din lista_deschisa
13. adauga curent in lista_inchisa
14. pentru fiecare vecin al nodului curent executa
15. *noulGCost*=*g_cost*(current) + *d*(current,vecin)
16. *noulFCost*=*h_cost*(vecin) + *noulGCost*
17. daca *vecin* $\in$ *lista_inchisa* si *noulFCost* $\geq$ *f_cost*(*vecin*)
18. continua
19. altfel daca *vecin* $\notin$ *lista_deschisa* sau *noulFCost* $\leq$ *f_cost*(*vecin*)
20. parinte(vecin) = curent
21. g_cost(vecin)=noulGCost
22. f_cost(vecin)=noulFCost
23. daca *vecin* $\in$ *lista_deschisa*
24. scoate vecin din lista_deschisa
25. adauga vecin in lista_deschisa

La începutul algoritmului considerăm 2 liste. O listă care conţine mulţimea nodurilor ce pot fi explorate (lista deschisă) şi o listă care ţine cont de nodurile care au fost deja vizitate (lista închisă). In lista deschisă introducem prima dată nodul sursă. După acest lucru, cu ajutorul funcţiei euristice ataşăm fiecărui nod un cost aproximativ al căii până la nodul destinaţie.

Acest lucru l-am realizat cu ajutorul algoritmului Bellman Ford aplicat asupra nodului destinaţie. In urma execuţiei algoritmului, în fiecare nod se va reţine distanţa minimă de la nodul destinaţie către sine. Am considerat funcţia euristică ca fiind 3/4 din valoarea obţinută în urma aplicării algoritmului Bellman Ford. In continuare iniţializez părintele nodului sursă ca fiind null iar f_cost ca fiind valoarea aproximativă a drumului minim până la destinaţie(h_cost). Atâta timp cât lista deschisă conţine noduri, vom extrage de fiecare dată nodul cu f_{-cost} minim. In cazul în care acesta este chiar nodul destinaţie, algoritmul se încheie. Altfel, scoatem nodul din lista deschisă şi îl marcăm ca vizitat, prin adăugarea acestuia în lista închisă. Parcurgem vecinii nodului curent şi pentru fiecare calculăm noulGCost ca fiind costul g al nodului curent plus valoarea muchiei care leagă nodurile. noulFCost va fi suma dintre noulGCost şi costul h al nodului vecin. In cazul în care nodul vecin se află în lista închisă şi noulFCost este mai mare sau egal decât costul f al nodului vecin nu facem nimic. Dacă nodul vecin nu se află în lista deschisă sau dacă noulFCost, calculat anterior, este mai mic decât costul f al nodului vecin atunci părintele nodului vecin va fi nodul curent şi actualizăm costul g si f al nodului vecin cu noulGCost şi noulFCost. Dacă nodul vecin se află în lista deschisă atunci îl scoatem pentru a evita adăugarea acestuia de 2 ori.

3 *Date experimentale*

Pentru a genera datele de intrare m-am folosit de clasa RandomMapGenerator. Deoarece fiecare nod trebuie să aibă un nume diferit, am utilizat librăria JavaFaker cu ajutorul căreia pot să apelez o funcţie pentru a-mi genera un nume aleator pentru oraşe. In această clasă, mai am 2 variabile numberOfCities şi numberO fRoads pe care le iniţializez în urma citrii valorilor din fişierele de input. Clasa conţine getRandomInteger BetweenRange(doublemin, doublemax), o functie care returnează un număr întreg aleator din intervalul [min,max]. Funcția principală din această clasă care realizează crearea în mod aleatoriu a hărţii cu oraşe este mapGenerator. După ce se execută, această funcţie returnează o listă cu nodurile care alcătuiesc harta şi în mod implicit şi drumurile dintre acestea. Prima dată în această funcţie creez un HashMap în care stochez oraşe al căror nume îl generez cu ajutorul functiei faker.address().city(). In cazul în care această funcţie generează un nume care se găseste deja în HashMap, în interiorul unui while se va genera un alt nume până când acesta nu se află in HashMap. Pentru a avea posibilitatea de a accesa noduri de pe poziţii alese in mod aleator, voi crea un ArrayList în care voi pune nodurile din HashMap, atribuind variabilei name din Node, valoarea cheii corespunzătoare nodului din HashMap. Prima dată, pentru a mă asigura că din orice nod pot ajunge în toate celelalte, am construit un drum între nodul 0 şi 1,1 şi 2, ..., numberOfCities - 1 şi numberOfCities. Deci numberOfCities -1 muchii sunt rezervate pentru a mă asigura că graful este conectat. In continuare, cu restul muchiilor generez aleator poziţii din lista de noduri până obţin 2 noduri diferite. Dacă există o muchie între acestea nu fac nimic, iar dacă nu există creez una. Pentru a ataşa un cost muchiilor, mă folosesc de functia getRandomIntegerBetweenRange(doublemin, doublemax). Pentru cele 10 teste am considerat costul muchiilor ca fiind în intervalul [50,500]. La final returnez lista de noduri care conţine implicit şi drumurile
.

4 *Proiectarea aplicației experimentale*

4.1 *Structura de nivel înalt a aplicației*

- Node.java
- Edge.java
- RandomMapGenerator.java
- BellmanFord.java
- AStar.java
- CalculateTime.java
- Main.java

4.2 *Descrierea mulțimii datelor de intrare*

Se citeşte dintr-un fişier numărul de noduri şi numărul de muchii din graful neorientat, iar apoi iniţializez variabilele numberOfCities şi numberOfRoads, din clasa RandomMapGenerator, cu valorile citite din fişier şi generez harta cu oraşe.

4.3 *Descrierea ieșirilor / rezultatelor*

Datele de ieşire din fişier menţionează numărul de oraşe şi de drumuri de pe hartă, poziţile celor 2 prieteni, nodurile care au fost vizitate şi numărul acestora în procesul de a determina cel mai scurt drum pe care să se întâlnească cei 2 prieteni, calea finală cu cost minim pe care să se întâlnească în cel mai scurt timp posibil, oraşul în care se vor întâlni şi timpii de execuţie ai algoritmului A^* şi a întregului program.

4.4 *Modulele aplicației*

Aplicaţia are următoarele module:

- 10 fişiere de tip text din care se citesc numărul de noduri şi de muchii care intervin în construirea în mod aleator a oraşului. Fiecare fişier contribuie la obţinerea unui test unic.
- 10 fişiere de tip text în care sunt scrise datele de iesire mentionate în subsecţiunea de mai sus.
- Node.java în care se definesc atributele unui nod(oraş) din graf(hartă). Fiecare nod conţine o listă de drumuri (cu tot cu costul lor) către oraşele vecine cu care se conectează (aceste drumuri se adaugă cu ajutorul functiei addNeighbour (Node neighbour, int cost) care construieşte un drum între nodul curent şi cel dat ca parametru funcţiei), un părinte în cazul în care nodul intervine în calea finală de cost minim pe care cei 2 prieteni se întâlnesc în cel mai scurt timp, variabila cost care în urma execuţiei algoritmului BellmanFord este actualizată şi va reprezenta distanţa minimă până la nodul pentru care s-a aplicat algoritmul, variabila *g_cost* care contorizează costul de până acum pentru a ajunge la destinaţie, variabila f_cost care reprezintă costul estimat total al căii prin nodul curent şi variabila $h_$cost care reprezintă o estimare a costului drumului de la nodul curent până la destinaţie. Această variabilă este iniţializată cu ajutorul funcţiei setHCost(), prin intermediul căreia am considerat ca $h_$cost să fie cost $-$ cost /4.

- Edge.java este clasa care defineşte atributele unei muchii din graf. Aceasta are un cost, şi o variabilă target, care reprezintă al doilea nod cu care se leagă primul atunci când se apelează funcţia addNeighbour .
- RandomMapGenerator.java este clasa pe care am descris-o în secţiunea 3 "Date experimentale".
- BellmanFord.java reprezintă clasa care execută algoritmul BellmanFord pentru a afla distanţa minimă a tuturor nodurilor până la un nod selectat. Această distanţă o voi folosi în funcţia euristică pentru a determina variabila h_cost.
- AStar.java este clasa care execută algoritmul A* pentru a determina drumul pe care cei 2 prieteni se vor întâlni în cel mai scurt timp.
- CalculateTime.java este clasa care se foloseşte de drumul aflat de A* pentru a determina oraşul în care se vor întâlni şi timpul scurs până se va intâmpla acest lucru. Timpul se calculează adunând, la fiecare tură, la timpul total, maximul dintre distanţa parcursă de primul prieten până la un oraş vecin şi distanţa parcursă de cel de-al doilea prieten. - Application.java este clasa în care testez toate celelalte funcţii. Aici realizez citirea din fişierele de input, generez în mod aleator harta cu oraşele, aplic BellmanFord pe nodul desemnat de una din poziţiile celor 2 prieteni, aplic A* având ca nod sursă şi destinaţie, poziţiile celor 2 prieteni, după care apelez functiia calculateTime pentru a determina timpul în care se vor întâlni.

4.5 *Functiile aplicației*

Aplicaţia are următoarele funcţii:

- bellmanFord(Node source, List < Node > nodeList) reprezintă funcţia care implementează algoritmul Bellman Ford. Prima dată considerăm costul nodului sursă ca fiind 0 pentru ca algoritmul să înceapă de la acesta. Considerăm o buclă for cu nodeList.size() - 1 iteraţii deoarece ne propunem să relaxăm muchia dintr-o cale cu o singură muchie, după muchiile dintr-o cale cu 2 muchii, până când ajungem la căi cu nodeList.size() - 1 muchii. nodeList.size() - 1 reprezintă lungimea maximă a căii minime dintr-un graf, dacă acesta nu are cicluri (în acest caz nu are). După trebuie să parcurgem lista cu noduri din graf, iar pentru fiecare nod să parcurgem lista cu muchii. Dacă costul ataşat nodului curent plus distanţa de la nodul curent la vecin este mai mic decât costul ataşat nodului vecin, atunci "relaxăm" muchia şi ataşăm noul cost nodului vecin.
- AStarSearch(Node source, Node destination, List < Node > nodeList, BufferedWriter writer) reprezintă funcţia care implementează algoritmul A*. Prima dată pentru fiecare nod de pe hartă, apelăm funcţia setHCost() pentru a iniţializa $h_$cost ca fiind 3/4 din cost al fiecărui nod care a fost determinat în urma algoritmului Bellman Ford. După aceea, iniţializăm părintele nodului sursă ca fiind null, g_cost al nodului sursă ca fiind 0 şi $f_$cost ca fiind egal cu $h_$cost deoarece $g_$cost este 0 . Considerăm un HashSet în care ţinem cont de nodurile vizitate, un PriorityQueue din care extragem de fiecare dată nodul cu $f_$cost minim şi o variabilă found care devine true atunci când nodul extras din coada cu priorităţi coincide cu nodul destinaţie. Prima dată introducem în coada cu priorităţi nodul sursă. Atâta timp cât coada cu priorităţi mai are elemente şi found = false extragem nodul cu $f_$cost minim din PriorityQueue şi îl adăugăm în lista cu noduri vizitate. De fiecare dată când extragem un nod din coada cu priorităţi îi scriem numele în fişier pentru a vizualiza la final ce oraşe au fost parcurse. Dacă nodul extras este chiar nodul destinaţie atunci variabila found devine true. Pentru fiecare nod extras

parcurgem lista sa de muchii. Calculăm un nou cost g ca fiind costul g al nodului curent plus distanţa între nodul curent şi nodul vecin şi un nou cost f ca fiind costul h al nodului vecin plus acest nou cost g. Dacă lista nodurilor vizitate conţine nodul vecin şi noul cost f este mai mare sau egal decât costul f al nodului vecin atunci nu facem nimic. Altfel dacă coada cu priorităţi nu conţine nodul vecin sau noul cost f este mai mic decât costul f al nodului vecin, atunci nodul curent devine părintele nodului vecin şi actualizăm costul g şi f al nodului vecin cu cele calculate anterior. În cazul în care coada cu priorităţi conţine acest nod vecin, atunci îl eliminăm din coadă pentru a evita existenţa sa de 2 ori. Apoi, indiferent dacă acesta era sau nu, adăugăm nodul în coadă. La final, pentru a reconstitui calea pornim de la nodul destinaţie şi mergând din părinte în părinte ajungem la nodul sursă şi determinăm calea pe care cei 2 prieteni se vor întâlni în cel mai scurt timp. Funcţia returnează această cale care va fi analizată de următoarea funcţie pe care o voi prezenta.

- În funcţia main din clasa Main se testează funcţiile discutate anterior de 10 ori, deoarece avem 10 fişiere de input. De fiecare dată se citeşte din fişierul de input numărul de noduri şi de muchii şi se iniţializează variabilele numberOfCities şi numberOf Roads din RandomMapGenerator. Se scrie în fişierul de output aceste 2 valori precum şi poziţiile generate în mod aleator ale celor 2 prieteni. Functiei bellmanFord oferim ca parametru oraşul în care se află cel de-al doilea prieten. După ce algoritmul A* găseşte calea pe care cei 2 prieteni se vor întâlni în cel mai scurt timp, o scriem în fişierul de output. După ce se execută funcţia calculateTime se va scrie oraşul în care se vor întâlni şi timpul.

5 *Rezultate și Concluzii*

a. Această problemă de căutare are ca şi scop să aşezăm cei 2 prieteni a şi b în acelaşi oraş i, în cel mai scurt timp T posibil. Considerăm distanţa în linie dreaptă între două oraşe i şi j ca fiind dist (i, j). Starea iniţială a problemei este reprezentată de perechea (i, j), unde i reprezintă oraşul în care se află primul prieten, iar j orasul în care se află cel de-al doilea prieten. Starea finală a problemei este determinată de perechea (i, i), ceea ce sugerează faptul că cei 2 prieteni s-au întâlnit în acelaşi oraş, într-un timp T care trebuie să fie minim. Acţiunile sunt determinate de mişcările celor 2 prieteni în unul din oraşele vecine celui în care se află. De exemplu, dacă primul se deplasează în i' şi al doilea in j', vom adăuga la timpul T, $\max(\text{dist}(i, i'), \text{dist}(j, j'))$. Spaţiul stărilor este determinat de mulţimea tuturor perechilor de oraşe (i, j). i şi j pot să reprezinte acelaşi oraş în situaţia în care cei 2 prieteni s-au întâlnit.

b. Am ales algoritmul A^* deoarece este un algoritm euristic de explorare a grafului (în această situatie o hartă cu oraşe generată în mod aleator). Deoarece sunt oferite ca şi date de intrare poziţiile celor 2 prieteni (oraşele în care aceştia se află), putem considera că aceste poziţii reprezintă o sursă şi o destinaţie pentru care aplicăm algoritmul A*. Acesta va afla calea de cost minim pe care cei doi prieteni se vor întâlni în cel mai scurt timp. După ce am aflat calea, cei 2 prieteni se deplasează simultan, unul spre celălalt. În tabelul următor este pusă în evidenţă evoluţia din perspectiva timpului de execuţie atât a algoritmului A* cât şi a întregului program pe seturi de date de dimensiuni din ce în ce mai mari. Se observă că timpul de execuţie al algoritmului A* este extrem de mic. Acest lucru se datorează funcţiei euristice de aproximare a distanţei minime a fiecărui nod până la nodul destinaţie, astfel încât se evită parcurgerea anumitor căi pe care ne îndepărtăm de destinaţia noastră.

Nr	Numărul de noduri	Numărul de muchii	A^*	Intreg programul
1	250	2500	0.002 s	0.22 s
2	750	4000	0 s	0.18 s
3	1000	5000	0 s	0.089 s
4	1500	7500	0.001 s	0.219 s
5	2000	10000	0 s	0.435 s
6	2500	12500	0.001 s	0.490 s
7	3000	15000	0.001 s	0.554 s
8	4000	20000	0.001 s	1.471 s
9	5000	25000	0.002 s	2.057 s
10	10000	30000	0.001 s	4.553 s

6 *References*

[1] A_AEX project site, http://latex-project.org/
[2] https://github.com/DiUS/java-faker
[3] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach. Prentice Hall, 3rd Edition, 2010, pp. 96-101.
[4] https://en.wikipedia.org/wiki/A*_search_algorithm
[5] https://en.wikipedia.org/wiki/Bellman–Ford_algorithm
[6] https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes
[7] https://dzone.com/articles/from-dijkstra-to-a-star-a-part-2-the-a-star-a-algo
[8] https://www.baeldung.com/java-faker