

Algoritm de căutare euristică (A^*)
grupa CR 2.2B, an 2, al 2-lea semestru, Calculatoare Română

Niculescu Marius-Andrei

May 30, 2021

Contents

1	Enunțul problemei	3
2	Algoritm	3
3	Date experimentale	4
4	Proiectarea aplicației experimentale	5
4.1	Structura de nivel înalt a aplicației	5
4.2	Descrierea mulțimii datelor de intrare	5
4.3	Descrierea ieșirilor / rezultatelor	5
4.4	Modulele aplicației	6
4.5	Funcțiile aplicației	7
5	Rezultate și Concluzii	8

1 Enunțul problemei

Să presupunem că doi prieteni locuiesc în diferite orașe pe o hartă. La fiecare pas, putem muta simultan fiecare prieten într-un oraș vecin de pe hartă. Timpul necesar pentru a trece de la orașul i la orașul vecin j este egal cu lungimea drumului $d(i, j)$ dintre cele 2 orașe, dar la fiecare pas prietenul care ajunge primul trebuie să aștepte până când și celălalt prieten ajunge (și 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. Scrieți o formulare detaliată pentru această problemă de căutare.
- b. Identificați un algoritm de căutare pentru această sarcină și explicați alegerea dumneavoastră.

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. $noulGCost = g_cost(curent) + d(curent, vecin)$
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 $noulFCost < f_cost(vecin)$
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 conține mulțimea nodurilor ce pot fi explorate (lista deschisă) și o listă care ține cont de nodurile care au fost deja vizitate (lista închisă). În lista deschisă introducem prima dată nodul sursă. După acest lucru, cu ajutorul funcției euristice atașăm fiecărui nod un cost aproximativ al căii până la nodul destinație. Acest lucru l-am realizat cu ajutorul algoritmului Bellman Ford aplicat asupra nodului destinație. În urma execuției algoritmului, în fiecare nod se va reține distanța minimă de la nodul destinație către sine. Am considerat funcția euristică ca fiind $3/4$ din valoarea obținută în urma aplicării algoritmului Bellman Ford. În continuare inițializez părintele nodului sursă ca fiind null iar f_cost ca fiind valoarea aproximativă a drumului minim până la destinație (h_cost). Atâta timp cât lista deschisă conține noduri, vom extrage de fiecare dată nodul cu f_cost minim. În cazul în care acesta este chiar nodul destinație, algoritmul se încheie. Altfel, scoatem nodul din lista deschisă și îl marcăm ca vizitat, prin adăugarea acestuia în lista închisă. Parcurgem vecinii nodului curent și pentru fiecare calculăm $noulGCost$ ca fiind costul g al nodului curent plus valoarea muchiei care leagă nodurile. $noulFCost$ va fi suma dintre $noulGCost$ și costul h al nodului vecin. În cazul în care nodul vecin se află în lista închisă și $noulFCost$ 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 și actualizăm costul g și f al nodului vecin cu $noulGCost$ și $noulFCost$. 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 funcție pentru a-mi genera un nume aleator pentru orașe. În această clasă, mai am 2 variabile *numberOfCities* și *numberOfRoads* pe care le inițializez în urma citirii valorilor din fișierele de input. Clasa conține *getRandomIntegerBetweenRange(doublemin, doublemax)*, o funcție care returnează un număr întreg aleator din intervalul $[min, max]$. Funcția principală din această clasă care realizează crearea în mod aleatoriu a hărții cu orașe este *mapGenerator*. După ce se execută, această funcție returnează o listă cu nodurile care alcătuiesc harta și în mod implicit și drumurile dintre acestea. Prima dată în această funcție creez un HashMap în care stochez orașe al căror nume îl generez cu ajutorul funcției *faker.address().city()*. În cazul în care această funcție generează un nume care se găsește deja în HashMap, în interiorul unui while se va genera un alt nume până când acesta nu se află în HashMap. Pentru a avea posibilitatea de a accesa noduri de pe poziții alese în 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 și 1, 1 și 2, ..., *numberOfCities* - 1 și *numberOfCities*. Deci *numberOfCities* - 1 muchii sunt rezervate pentru a mă asigura că graful este conectat. În continuare, cu restul muchiilor generez aleator poziții din lista de noduri până obțin 2 noduri diferite. Dacă există o muchie între acestea nu fac nimic, iar dacă nu există creez una. Pentru a atașa un cost muchiilor, mă folosesc de funcția *getRandomIntegerBetweenRange(doublemin, doublemax)*. Pentru cele 10 teste am considerat costul muchiilor ca fiind în intervalul [50,500]. La final returnez lista de noduri care conține implicit și 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
- Application.java

4.2 Descrierea mulțimii datelor de intrare

Se citește dintr-un fișier numărul de noduri și numărul de muchii din graful neorientat, iar apoi inițializez variabilele *numberOfCities* și *numberOfRoads*, din clasa *RandomMapGenerator*, cu valorile citite din fișier și generez harta cu orașe.

4.3 Descrierea ieșirilor / rezultatelor

Datele de ieșire din fișier menționează numărul de orașe și de drumuri de pe hartă, pozițiile celor 2 prieteni, nodurile care au fost vizitate și 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, orașul în care se vor întâlni și timpii de execuție ai algoritmului A* și a întregului program.

4.4 Modulele aplicației

Aplicația are următoarele module:

- 10 fișiere de tip text din care se citesc numărul de noduri și de muchii care intervin în construirea în mod aleator a orașului. Fiecare fișier contribuie la obținerea unui test unic.
- 10 fișiere de tip text în care sunt scrise datele de ieșire menționate în subsecțiunea de mai sus.
- Node.java în care se definesc atributele unui nod(oraș) din graf(hartă). Fiecare nod conține o listă de drumuri (cu tot cu costul lor) către orașele vecine cu care se conectează (aceste drumuri se adaugă cu ajutorul funcției *addNeighbour(Node neighbour, int cost)* care construiește un drum între nodul curent și cel dat ca parametru funcției) , 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 execuției algoritmului BellmanFord este actualizată și va reprezenta distanța minimă până la nodul pentru care s-a aplicat algoritmul, variabila *g_cost* care contorizează costul de până acum pentru a ajunge la destinație, variabila *f_cost* care reprezintă costul estimat total al căii prin nodul curent și variabila *h_cost* care reprezintă o estimare a costului drumului de la nodul curent până la destinație. Această variabilă este inițializată cu ajutorul funcției *setHCost()*, prin intermediul căreia am considerat ca *h_cost* să fie $cost - cost/4$.
- Edge.java este clasa care definește atributele unei muchii din graf. Aceasta are un cost, și o variabilă *target*, care reprezintă al doilea nod cu care se leagă primul atunci când se apelează funcția *addNeighbour*.
- RandomMapGenerator.java este clasa pe care am descris-o în secțiunea 3 "Date experimentale".
- BellmanFord.java reprezintă clasa care execută algoritmul BellmanFord pentru a afla distanța minimă a tuturor nodurilor până la un nod selectat. Această distanță o voi folosi în funcția 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 folosește de drumul aflat de A* pentru a determina orașul în care se vor întâlni și timpul scurs până se va întâmpla acest lucru. Timpul se calculează adunând, la fiecare tură, la timpul total, maximul dintre distanța parcursă de primul prieten până la un oraș vecin și distanța parcursă de cel de-al doilea prieten.

- `Application.java` este clasa în care testez toate celelalte funcții. Aici realizez citirea din fișierele de input, generez în mod aleator harta cu orașele, aplic `BellmanFord` pe nodul desemnat de una din pozițiile celor 2 prieteni, aplic `A*` având ca nod sursă și destinație, pozițiile celor 2 prieteni, după care apelez funcția `calculateTime` pentru a determina timpul în care se vor întâlni.

4.5 Funcțiile aplicației

Aplicația are următoarele funcții:

- `bellmanFord(Node source, List < Node > nodeList)` reprezintă funcția 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` iterații 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 atașat nodului curent plus distanța de la nodul curent la vecin este mai mic decât costul atașat nodului vecin, atunci "relaxăm" muchia și atașăm noul cost nodului vecin.
- `ASearch(Node source, Node destination, List < Node > nodeList, BufferedWriter writer)` reprezintă funcția care implementează algoritmul `A*`. Prima dată pentru fiecare nod de pe hartă, apelăm funcția `setHCost()` pentru a inițializa `h_cost` ca fiind $3/4$ din `cost` al fiecărui nod care a fost determinat în urma algoritmului Bellman Ford. După aceea, inițializăm părintele nodului sursă ca fiind null, `g_cost` al nodului sursă ca fiind 0 și `f_cost` ca fiind egal cu `h_cost` deoarece `g_cost` este 0. Considerăm un `HashSet` în care ținem cont de nodurile vizitate, un `PriorityQueue` din care extragem de fiecare dată nodul cu `f_cost` minim și o variabilă `found` care devine `true` atunci când nodul extras din coada cu priorități coincide cu nodul destinație. Prima dată introducem în coada cu priorități nodul sursă. Atâta timp cât coada cu priorități mai are elemente și `found = false` extragem nodul cu `f_cost` minim din `PriorityQueue` și îl adăugăm în lista cu noduri vizitate. De fiecare dată când extragem un nod din coada cu priorități îi scriem numele în fișier pentru a vizualiza la final ce orașe au fost parcurse. Dacă nodul extras este chiar nodul destinație 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 distanța între nodul curent și nodul vecin și un nou cost `f` ca fiind costul `h` al nodului vecin plus acest nou cost `g`. Dacă lista nodurilor vizitate conține nodul vecin și noul cost `f` este mai mare sau egal decât costul `f` al nodului vecin atunci nu facem nimic. Altfel dacă coada cu priorități nu conține nodul vecin sau noul cost `f` este mai mic decât costul

f al nodului vecin, atunci nodul curent devine părintele nodului vecin și actualizăm costul g și f al nodului vecin cu cele calculate anterior. În cazul în care coada cu priorități conține acest nod vecin, atunci îl eliminăm din coadă pentru a evita existența 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 destinație și mergând din părinte în părinte ajungem la nodul sursă și determinăm calea pe care cei 2 prieteni se vor întâlni în cel mai scurt timp. Funcția returnează această cale care va fi analizată de următoarea funcție pe care o voi prezenta.

- În funcția main din clasa Application se testează funcțiile discutate anterior de 10 ori, deoarece avem 10 fișiere de input. De fiecare dată se citește din fișierul de input numărul de noduri și de muchii și se inițializează variabilele *numberOfCities* și *numberOfRoads* din RandomMapGenerator. Se scrie în fișierul de output aceste 2 valori precum și pozițiile generate în mod aleator ale celor 2 prieteni. Funcției *bellmanFord* oferim ca parametru orașul în care se află cel de-al doilea prieten. După ce algoritmul A* găsește calea pe care cei 2 prieteni se vor întâlni în cel mai scurt timp, o scriem în fișierul de output. După ce se execută funcția *calculateTime* se va scrie orașul în care se vor întâlni și timpul.

5 Rezultate și Concluzii

- Această problemă de căutare are ca și scop să așezăm cei 2 prieteni a și b în același oraș i, în cel mai scurt timp T posibil. Considerăm distanța în linie dreaptă între două orașe i și j ca fiind $\text{dist}(i, j)$. Starea inițială a problemei este reprezentată de perechea (i, j), unde i reprezintă orașul în care se află primul prieten, iar j orașul 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 același oraș, într-un timp T care trebuie să fie minim. Acțiunile sunt determinate de mișcările celor 2 prieteni în unul din orașele vecine celui în care se află. De exemplu, dacă primul se deplasează în i' și al doilea în j', vom adăuga la timpul T, $\max(\text{dist}(i, i'), \text{dist}(j, j'))$. Spațiul stărilor este determinat de mulțimea tuturor perechilor de orașe (i, j). i și j pot să reprezinte același oraș în situația în care cei 2 prieteni s-au întâlnit.
- Am ales algoritmul A* deoarece este un algoritm euristic de explorare a grafului (în această situație o hartă cu orașe generată în mod aleator). Deoarece sunt oferite ca și date de intrare pozițiile celor 2 prieteni (orașele în care aceștia se află), putem considera că aceste poziții reprezintă o sursă și o destinație 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 evidență evoluția din perspectiva timpului de execuție atât a algoritmului A* cât și a întregului program pe seturi de date de dimensiuni din ce în ce mai mari. Se observă că timpul de execuție al algoritmului A* este extrem de mic. Acest lucru se datorează funcției euristice de aproximare a distanței minime a fiecărui nod până la nodul destinație, astfel încât se evită parcurgerea anumitor căi pe care ne îndepărtăm de destinația noastră.

Nr	Numărul de noduri	Numărul de muchii	A*	Întreg programul
1	500	2500	0.002 s	0.48 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

References

- [1] L^AT_EX project site, <http://latex-project.org/>
- [2] https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes
- [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://www.baeldung.com/java-faker>
- [6] <https://github.com/DiUS/java-faker>
- [7] <https://dzone.com/articles/from-dijkstra-to-a-star-a-part-2-the-a-star-a-algo>
- [8] https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm