Algoritmica grafurilor

Curs 6 Grafuri ponderate, Algoritmul lui Dijkstra, Algoritmul lui Floyd-Warshall

Algoritmul lui Dijkstra

- Este utilizat pentru rezolvarea problemei P2 enuntate in cursul anterior(gasirea drumurilor de lungime minima de la un nod sursa s la toate nodurile unui graf orientat
- Functioneaza doar pentru grafurile cu ponderi pozitive ($\forall\ x \in V, w(x)>0$
- porneste de la un nod sursa si analizeaza graful pentru a gasi cel mai scurt drum de la nodul sursa la toate celelalte noduri ale grafului.
- algoritmul pastreza *urma* distantei minime curente de la nodul sursa la fiecare nod al grafului si actualizeaza aceasta valoare daca gaseste o cale mai scurta
- Odata ce algoritmul a gasit calea cea mai scurta intre nodul sursa si un anumit nod acest nod este marcat ca si vizitat si adaugat in cale
- procesul continua recursiv pana cand toate nodurile au fost adaugate in cale. In acest mod am determinat un drum care conecteaza nodul sursa de celelalte noduri prin cel mai scurt drum posibil

Istoria algoritmului

Algoritmul a fost creat de catre **Edsger Wybe Dijkstra,** un cercetator olandez care in 1959 a publicat un articol de 3 pagini, "A note on two problems in connexion with graphs" in care isi prezinta noul sau algoritm.

Intr-un interviu din 2001 dr, Dijkstra explica cum si de ce a proiectat acest algoritm:

What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later. The publication is still quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame. — As quoted in the article Edsger W. Dijkstra from An interview with Edsger W. Dijkstra.

Algoritmul lui Dijkstra- Pseudocod

1. Se creaza o multime sptSet (shortest path tree set) care retine varfurile incluse in sptSet, a caror distanta minima de la nodul sursa a fost calculata ; initial multimea este vida.
2. Se creaza un vector *dist* cu distantele de la nodul sursa la celelalte noduri. Acestea se initializeaza cu infinit in afara de distanta pana la nodul sursa care este 0
3. Pana cand *sptSet* nu include toate varfurile
   a. se alege un varf *u* care nu este in sptSet si are valoarea minima in vectorul de distante
   b. se include in sptSet
   c. se actualizeaza valorile distantelor pentru toti adiacentii v a lui u . Pentru fiecare varf adiacent v daca suma dintre dist(u) si ponderea muchiei u-v este mai mica decat dist(v) atunci dist[v]=dist[u]+w(u-v)

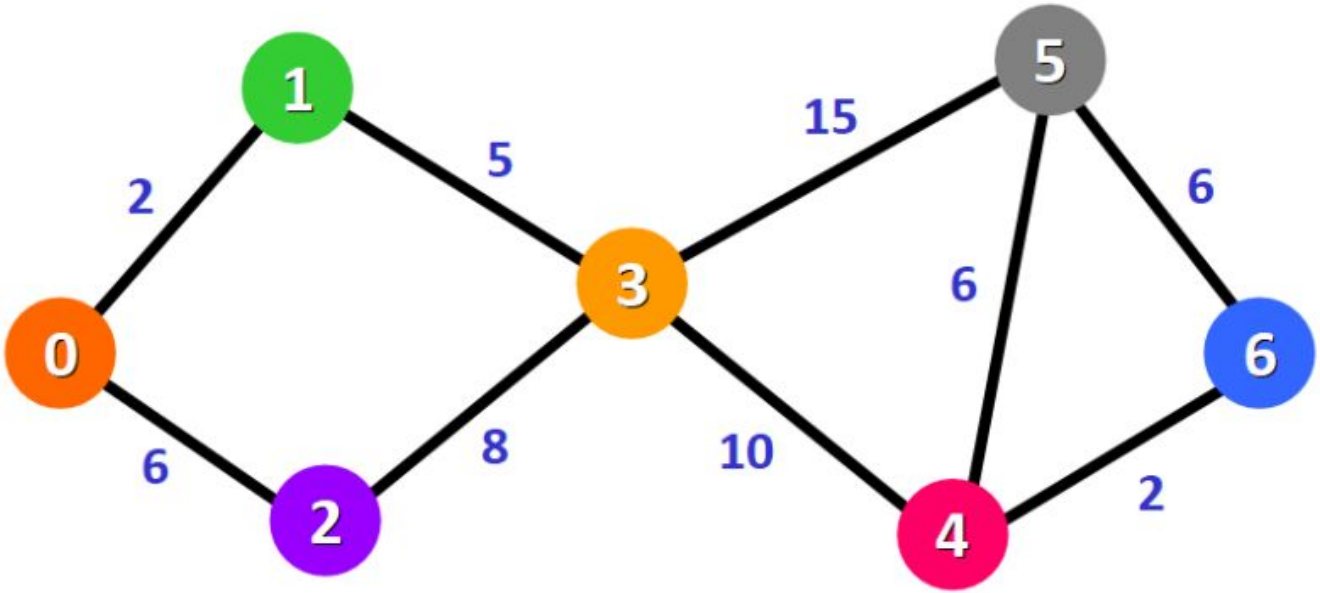
```
function Dijkstra(Graph, source):
       dist[source]  := 0                     // Distance from source to source is set to 0
       for each vertex v in Graph:            // Initializations
           if v ≠ source
               dist[v]  := infinity           // Unknown distance function from source to each node set to infin
           add v to Q                         // All nodes initially in Q

      while Q is not empty:                   // The main loop
          v := vertex in Q with min dist[v]   // In the first run-through, this vertex is the source node
          remove v from Q

          for each neighbor u of v:           // where neighbor u has not yet been removed from Q.
              alt := dist[v] + length(v, u)
              if alt < dist[u]:               // A shorter path to u has been found
                  dist[u]  := alt             // Update distance of u

      return dist[]
  end function
```

Exemplu

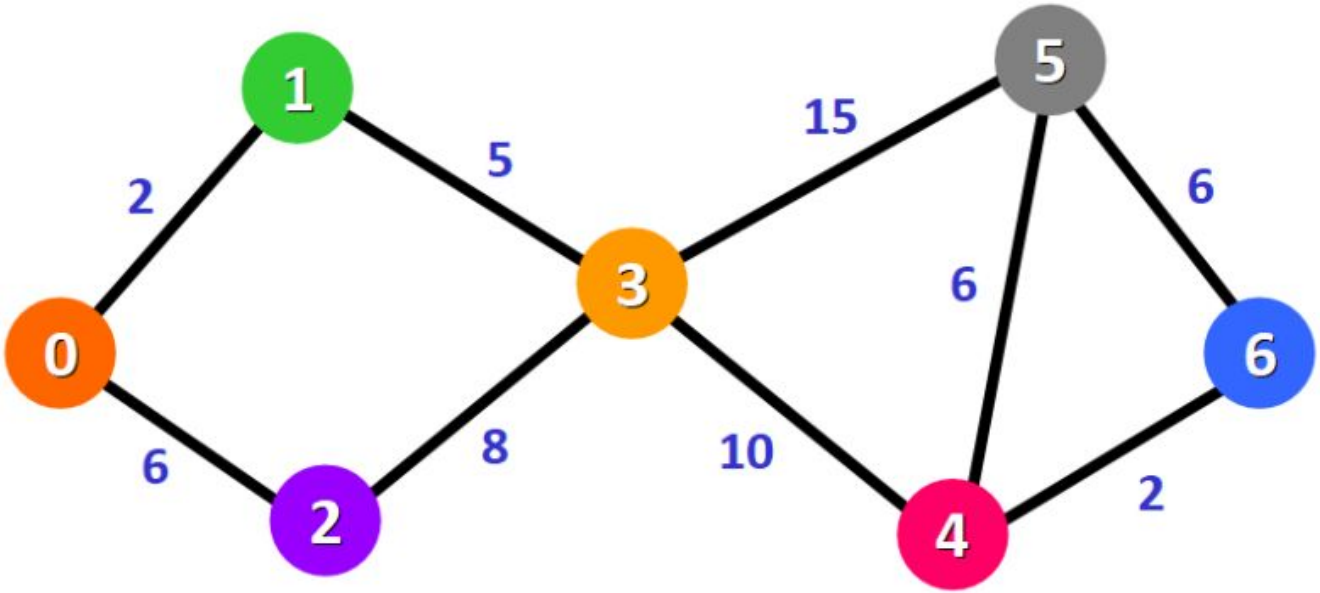


Distance:

0: 0
1: ∞
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, 1, 2, 3, 4, 5, 6}

Exemplu

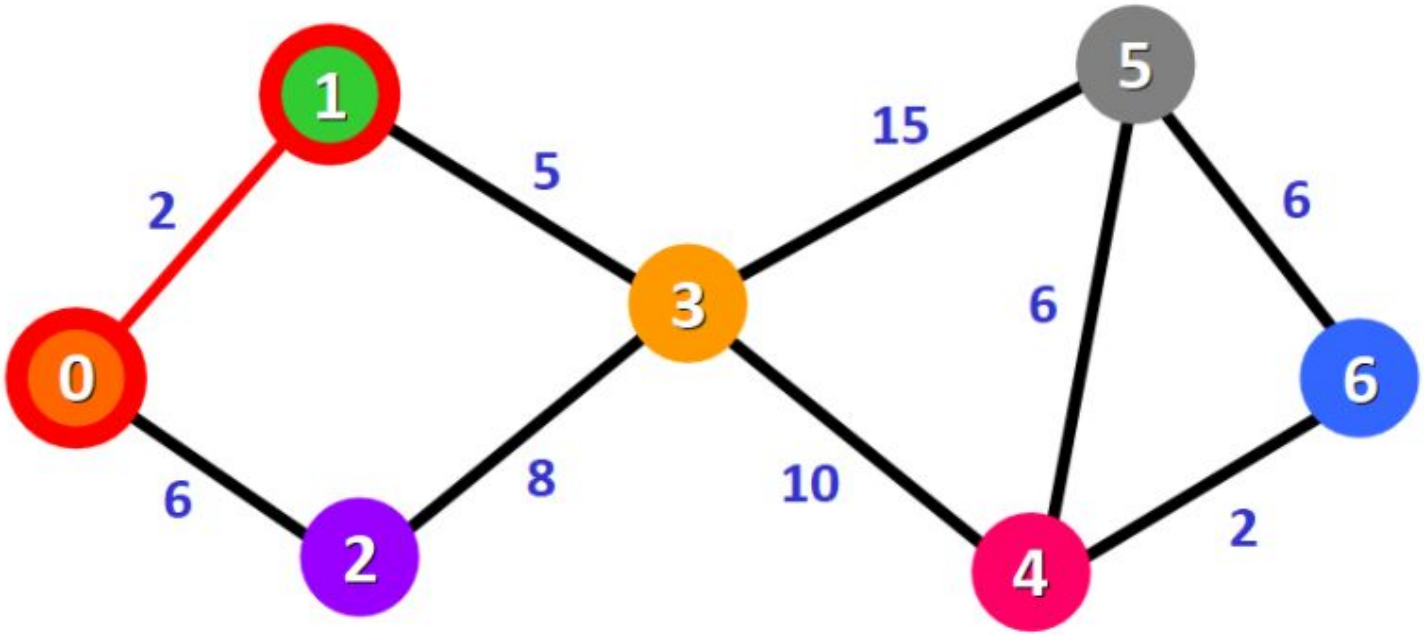


Distance

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ∞
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, 1, 2, 3, 4, 5, 6}

Exemplu

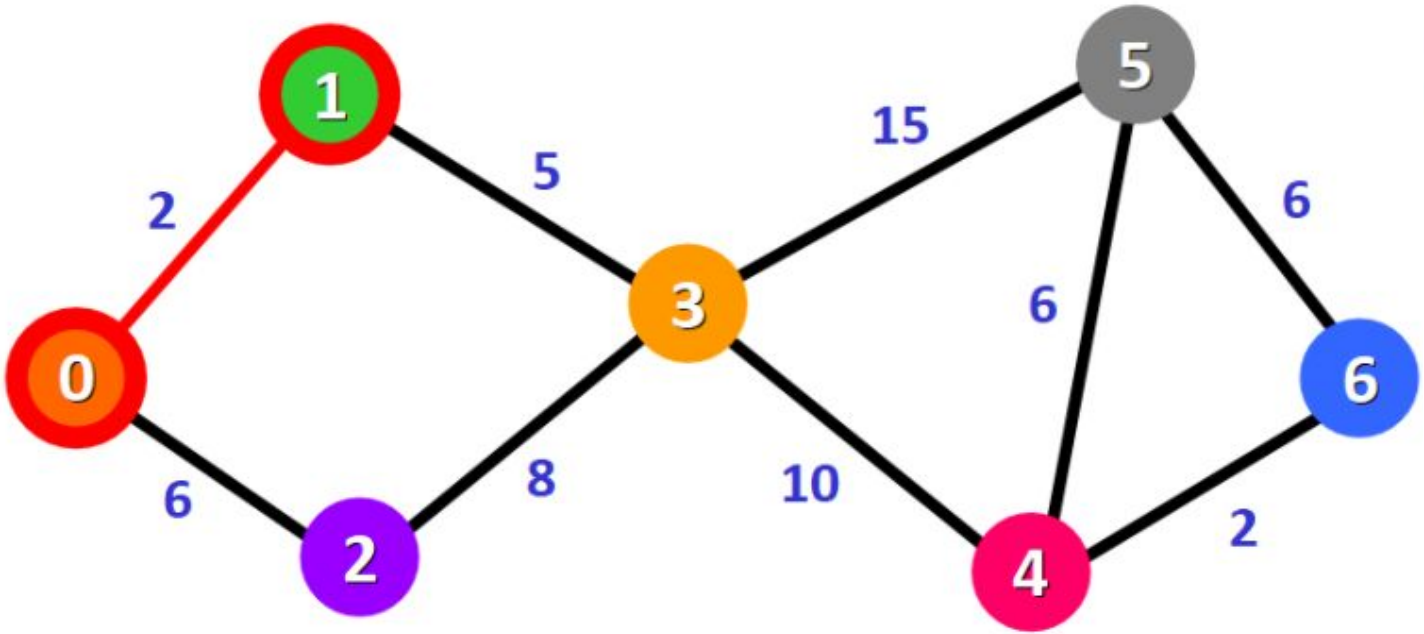


Distance

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ∞
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, 2, 3, 4, 5, 6}

Exemplu

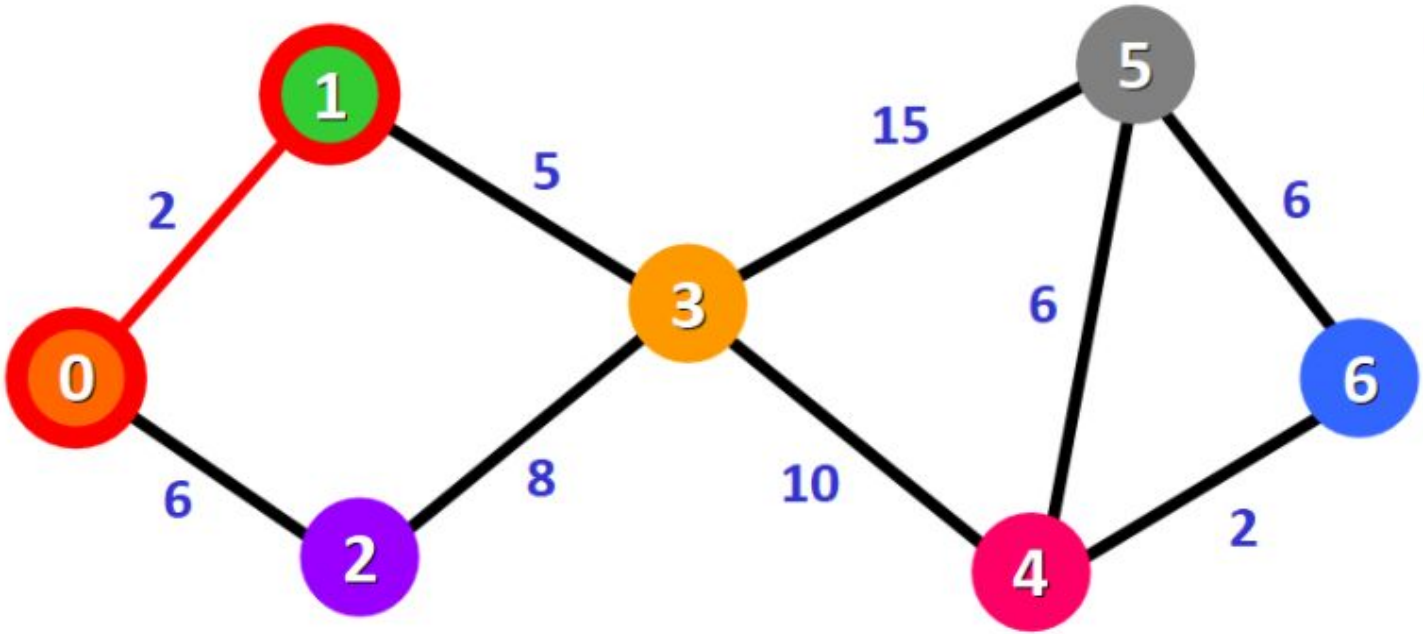


Distance:

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, 2, 3, 4, 5, 6}

Exemplu

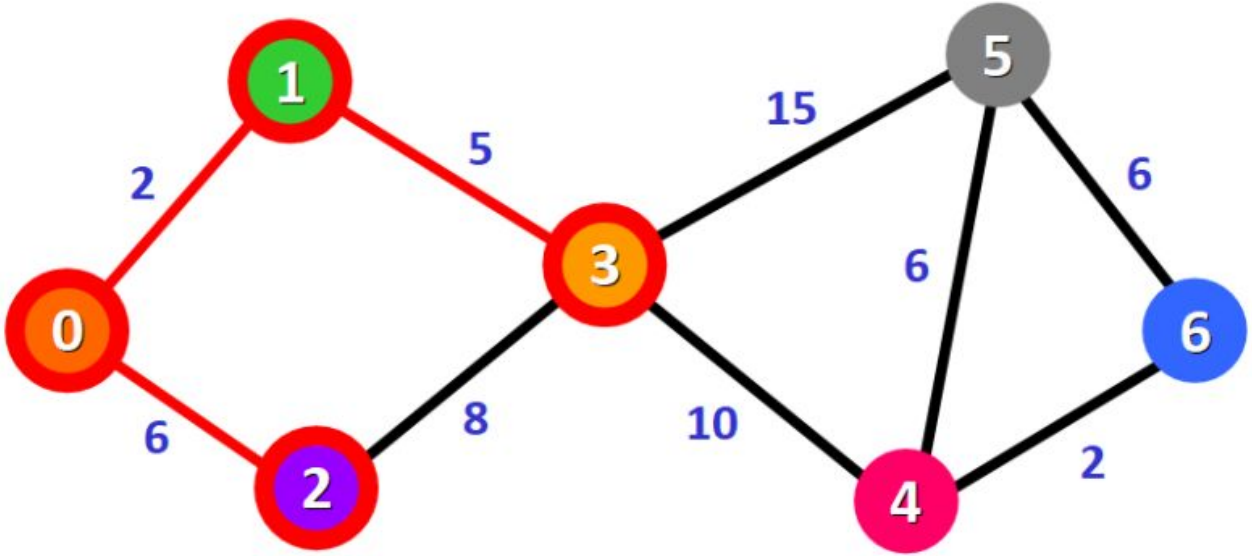


Distance:

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, 2, 3, 4, 5, 6}

Exemplu

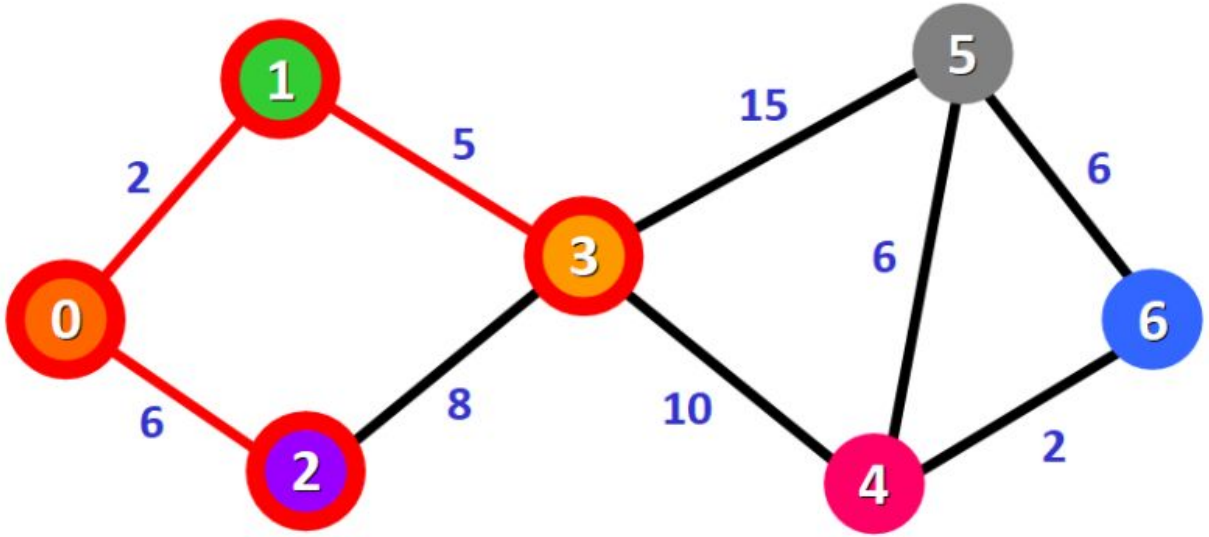


Distance:

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, 2, 3, 4, 5, 6}

Exemplu



Distance:

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ~~∞~~ 7
4: ~~∞~~ 17 from (2 + 5 +
5: ~~∞~~ 22 from (2 + 5 +
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, ~~2~~, ~~3~~, 4, 5, 6}

Algoritmul lui Dikstra - Vizualizare

https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

Cai cu lungime ponderata minima intre toate nodurile unui graf

Fie G un graf ponderat cu *n* noduri și *m* muchii Se cere:

- pentru toți x, y ∈ V să se găsesca cai $\pi_{x,y}$ cu $length_w(\pi_{x,y}) = \delta_w(x, y)$.

Observatii:

- Această problemă se poate rezolva rulând de n ori unul din algoritmii deja prezentați, câte o dată pentru fiecare nod x ∈ V(G) ca sursă.
- Complexitate temporala:
  - $O(n^4)$ dacă se foloseste alg. lui Bellman-Ford pentru cazul general, când putem avea muchii cu ponderi negative.
  - $O(n^3)$ daca se foloseste alg. lui Dijkstra pentru cazul particular, când w(e) > 0 pentru toate muchiile e ∈ E.
- Vom prezenta o metodă nouă: Algoritmul lui Floyd-Warshall: Complexitate temporală: $O(n^3)$ pentru cazul când putem avea muchii cu ponderi negative, dar fără cicluri cu lungime ponderată negativă.

Algoritmul A* Intuitie

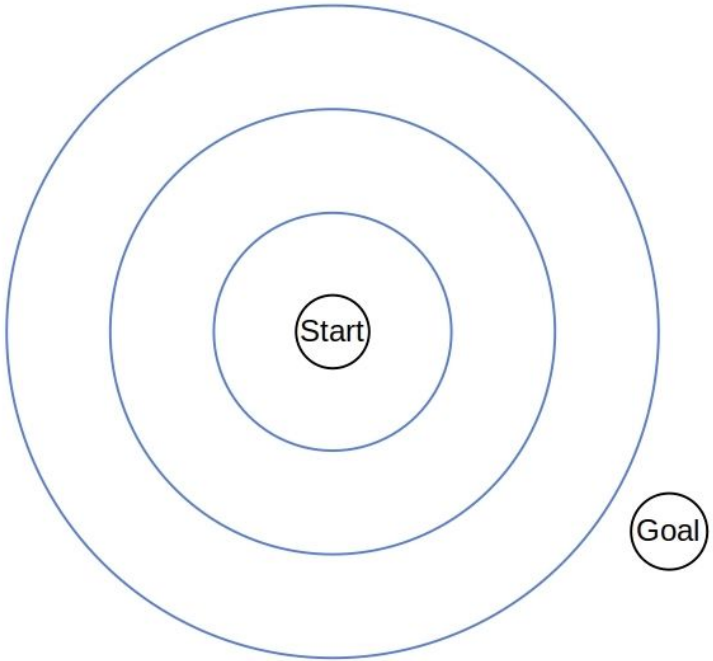
Să luăm în considerare un exemplu motivant. Suntem la o intersecție A și am dori să mergem la o intersecție B care, din fericire, știm că se află la nord de noi. În acest caz, intersecțiile sunt vârfurile unui grafic, iar drumurile sunt arcurile.

Dacă efectuați o primă căutare, așa cum este ilustrat de algoritmul lui Dijkstra , vom căuta fiecare punct cu o rază circulară fixă, treptat vom extinde acest cerc pentru a căuta intersecțiile cele mai îndepărtate de punctul nostru de plecare. Aceasta poate fi o strategie eficientă dacă nu știți unde este destinația, așa cum fac poliția în căutarea unui criminal ascuns.

Cu toate acestea, veți pierde timpul dacă aveți mai multe informații. O strategie mai bună este explorarea intersecțiilor situate la nord de prima, deoarece acestea vor fi probabil vârfurile cele mai apropiate de B. Cu toate acestea, trebuie remarcat faptul că drumurile ar putea fi închise obligându-ne să mergem spre sud pentru a ajunge la destinație cu o cale modelată. de C. Deci, dacă drumurile o permit, vom merge și vom explora intersecțiile din ce în ce mai aproape de intersecția obiectivului B. Vom avea nevoie de o retragere ocazională, dar intuitiv aceasta este o strategie care are șanse mari de găsind rapid obiectivul. În plus, se poate dovedi că această strategie va găsi, în orice caz, cea mai bună cale posibilă, adică soluția optimă, la fel ca și căutarea în primul rând. Aceasta este esența cercetării A *.

Cu toate acestea, nu există nicio garanție că A * va funcționa mai bine decât algoritmii simpli de căutare. Într-un mediu foarte complicat, singura modalitate de a ajunge la destinație ar putea fi să ne îndreptăm spre sud și apoi să ne plimbăm în jurul lui. În aceste cazuri, testarea nodurilor cele mai apropiate de destinația noastră ar putea fi o pierdere de timp.

Modul de cautare al algoritmului Dijkstra



Algoritmul A* - Vedere de ansamblu

- Se spune că este admisibil un algoritm de căutare care garantează întotdeauna găsirea celei mai scurte căi către un obiectiv. Dacă A * folosește o euristică, atunci distanța (sau, în general, costul) față de obiectiv nu trebuie niciodată supraestimată, deci se poate verifica dacă A * va fi admisibilă. O euristică care face căutarea A * admisibilă se numește euristică admisibilă .
- Dacă estimarea returnează pur și simplu zero, ceea ce nu va fi niciodată o supraestimare, atunci A * va efectua algoritmul lui Dijkstra și va găsi în continuare o soluție optimă, deși nu rapid. Cea mai bună euristică posibilă, deși nu este de obicei practic să o calculăm, este distanța minimă efectivă până la obiectiv. Un exemplu de euristică practică acceptabilă este distanța în care cioara zboară de la destinație pe o hartă.
- Se poate verifica că A * nu ia în considerare mai multe noduri decât orice alt algoritm de căutare fezabil, cu excepția cazului în care algoritmul alternativ are o estimare euristică mai precisă. În acest sens, A * este algoritmul cel mai eficient din punct de vedere al calculului, care garantează căutarea celei mai scurte căi.

Algoritmul A* descriere

- A * începe de la nodul selectat. Un cost de intrare este definit pentru fiecare nod (de obicei zero pentru nodul inițial). A * evaluează apoi distanța de la meta-nod de la cel curent. Această estimare și costul formează împreună euristica care va fi atribuită căii care trece prin acest nod. Nodul este apoi adăugat la o listă , numită adesea „deschis”.
- Algoritmul elimină apoi primul nod din listă (deoarece va avea cea mai mică valoare a funcției euristice). Dacă lista este goală, nu vor exista căi de la nodul de pornire la meta nod și algoritmul se va opri. Dacă nodul este meta nodul, A * reconstruiește și transmite calea obținută și se oprește. Această reconstrucție a căii pornind de la cele mai apropiate noduri înseamnă că nu este necesară memorarea căii în fiecare nod.
- Dacă nodul nu este meta nodul, vor fi create noi noduri pentru toate nodurile vecine admisibile; cum se face acest lucru depinde de problemă. Pentru fiecare nod ulterior A * calculează „costul” intrării în nod și îl salvează cu nodul. Acest cost este calculat din suma cumulativă a greutăților stocate în strămoși, plus costul operațiunii pentru a ajunge la acest nou nod.
- Algoritmul gestionează, de asemenea, o listă „închisă”, o listă de noduri care au fost deja verificate. Dacă un nou nod generat este deja în listă cu un cost egal sau mai mic, nu va exista nicio investigație viitoare a acelui nod sau a căii sale asociate. Dacă un nod din lista închis este același cu unul nou, dar a fost stocat la un cost mai mare, atunci acesta va fi eliminat din lista închisă, iar procesul va continua începând de la noul nod.
- Apoi, o estimare a distanței de la noul nod la obiectiv este adăugată la cost pentru a-și forma valoarea euristică. Acest nod este apoi adăugat la lista „deschisă”, cu excepția cazului în care există un nod identic cu valoare euristică mai mică sau egală.
- Algoritmul va fi adoptat la fiecare nod vecin, după care nodul original este preluat din listă și plasat în lista „închis”. Următorul nod va fi obținut din lista deschisă și odată cu acesta se va repeta procesul descris.

A* - Pseudocod

```
funcția A * ( start , obiectiv )
    closedset: = mulțimea vidă% Setul de noduri deja evaluate.
    cu carcasa: = set care conține inițial nod% Setul de noduri tentative de a fi evaluate.
    g_score [ start ] : = 0 % Distanța de la început de -a lungul căii optime .
    came_from: = vida map% Harta nodurilor navigau.
    h_score [ start ] : = heuristic_estimate_of_distance ( start , goal )
    f score [ start ] : = h_score [ start ] % Distanța totală estimată de la început până la
obiectiv până la y .
    în timp ce deschiderea nu este goală
        x: = nodul în care are valoarea cu carcasa cea mai mică f_score []
        dacă x = obiectiv
            returnează cale reconstituire ( venit_de , gol )
        eliminați x din deschidere
        adăugați x la setul închis
        y foreach în neighbor_nodes (x)
            dacă y în closetset
                continua
            tentativ g score : = g score [ x ] + dist between ( x , y )
```

A* - Pseudocod continuare

```
în cazul în care nu y în cu carcasa
                adăuga□i y la deschidere


                tentativ_este_mai bun : = adevărat
            elseif tentativ_g_score < g_score [ y ]
                tentativ_este_mai bun : = adevărat
            altceva
                tentativ_este_mai bun : = false
            dacă tentativ_este_mai bun = adevărat
                venit_de [ y ] : = x
                g_score [ y ] : = tentativ_g_score
                h_score [ y ] : = heuristic_estimate_of_distance ( y , goal )
                f_score [ y ] : = g_score [ y ] + h_score [ y ]
    e□ecul de întoarcere
```

A* vs. Dijkstra

https://giphy.com/gifs/xIxxOOMKIiVRmgcKrF?utm_source=iframe&utm_medium=embed&utm_campaign=Embeds&utm_term=https%3A%2F%2Fcdn.embedly.com%2F

Algoritmul lui Floyd-Warshall

Structuri de date necesare :

- Două tablouri n × n, astfel încât, pentru orice x, y ∈ V:
  - 1d[x][y]: o margine superioară a lui δ_w (x, y).
  - P[x][y] ∈ {null} ∪ V.

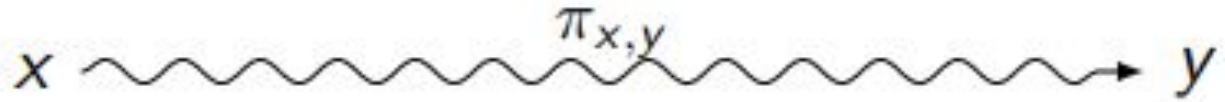
La terminarea algoritmului, valorile lui P[][] ¸si d[][] au proprietățile următoare:

- d[x][y] = δ_w (x, y).
- Dacă x≠yși există un drum cu lungime ponderată minimă de la x la y atunci P[x][y] este predecesorul nodului y pe o cale π_{xy} cu lungime ponderată minimă.

Algoritmul lui Floyd-Warshall

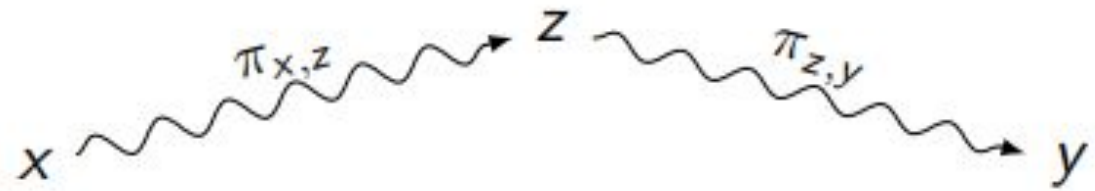
Dacă x, y, z ∈ V atunci orice drum $\pi_{x,y}$ cu lungime ponderată minimă de la x la y are una din următoarele 2 forme:

1. z nu este nod intermediar al drumului $\pi_{x,y}$



2. z este nod intermediar al drumurilor $\pi_{x,z}$ și $\pi_{z,}$

y



⇒ putem defini o metod˘a recursiv˘a de calcul al elementelor tablourilor P[][] ¸si d[][].

- Fie [x1, x2, . . . , xn] o enumerare fixata a nodurilor din V(G). pentru 0 ≤ k ≤ n definim:
  - d[k][i][j] este cea mai mică lungime ponderată a unei căi de la x_i la x_j care trece doar prin noduri intermediare din mulțimea {x1, . . . , xk }. Dacă o astfel de cale nu există, atunci d[k][i][j] = +∞.
  - P[k][i][j] este null dac[i = j sau d[k][i][j] = +∞. în caz contrar, P[k][i][j] este predecesorul nodului xj pe un drum cu lungime ponderată minimă de la x_i la x_j care trece doar prin noduri intermediare din mulțimea {x1, . . . , xk }.

Algoritmul lui Floyd-Warshall

Rezultă că, pentru toți i, j ∈ {1, 2, . . . , n} avem:

$$\begin{aligned} \mathtt{d}[0][i][j] &= w(x_i, x_j), \\ \mathtt{P}[0][i][j] &= \begin{cases} \mathtt{null} & \text{dacă } i = j \text{ sau } w(x_i, x_j) = +\infty, \\ x_i & \text{în caz contrar} \end{cases} \end{aligned}$$

iar dacă 1 ≤ k ≤ n atunci

$$\begin{aligned} \mathtt{d}[0][i][j] &= \min(\mathtt{d}[k-1][i][j], \mathtt{d}[k-1][i][k] + \mathtt{d}[k-1][k][j]), \\ \mathtt{P}[k][i][j] &= \begin{cases} \mathtt{P}[k-1][i][j] & \text{dacă } \mathtt{d}[k-1][i][j] = \mathtt{d}[k][i][j], \\ \mathtt{P}[k-1][k][j] & \text{în caz contrar.} \end{cases} \end{aligned}$$

Algoritmul lui Floyd-Warshall

Rezultă că, pentru toți i, j ∈ {1, 2, . . . , n} avem:

$$\begin{aligned} \mathtt{d}[0][i][j] &= w(x_i, x_j), \\ \mathtt{P}[0][i][j] &= \begin{cases} \mathtt{null} & \text{dacă } i = j \text{ sau } w(x_i, x_j) = +\infty, \\ x_i & \text{în caz contrar} \end{cases} \end{aligned}$$

iar dacă 1 ≤ k ≤ n atunci

$$\begin{aligned} \mathtt{d}[0][i][j] &= \min(\mathtt{d}[k-1][i][j], \mathtt{d}[k-1][i][k] + \mathtt{d}[k-1][k][j]), \\ \mathtt{P}[k][i][j] &= \begin{cases} \mathtt{P}[k-1][i][j] & \text{dacă } \mathtt{d}[k-1][i][j] = \mathtt{d}[k][i][j], \\ \mathtt{P}[k-1][k][j] & \text{în caz contrar.} \end{cases} \end{aligned}$$

Algoritmul lui Floyd-Warshall

Deoarece nodurile intermediare ale oricărei căi sunt ăn mulțimea {x1, x2, . . . , xn}, putem defini:

$d[x_i][x_j] = d[n][i][j]$ ¸si $P[x_i][x_j] = P[n][i][j]$.

Algoritmul lui Floyd-Warshall

Analiza complexitatii temporale:

1. Inițializarea valorilor lui d[0][i][j] ¸si P[0][i][j] pentru $1 \leq i, j \leq n$ durează $O(n^2)$.
2. Calculul valorilor lui d[k][i][j] ¸si P[k][i][j] din valorile pentru $k - 1$ durează $O(n\ )$.
3. Acest calcul trebuie repetat pentru k de la 1 la n

   $\Rightarrow$ complexitatea temporal $n \cdot O(n^2\ ) = O(n^3\ )$.

Algoritmul lui Floyd-Warshall

Exemplu animat:

https://www.cs.usfca.edu/~galles/visualization/Floyd.html