OMMÄSTARPROV 1

BENEDITH MULONGO

[DATUM]

19940311-0050
Benedith.kth.se

Uppgift 1 Brädspelet No

Problembeskrivning

Vi har ett brädspel av namnet No som spelas på ett bräde med NxN rutor. Ett drag består av att en pjäs flyttas ett steg uppåt, ett steg nedåt, ett steg till vänster eller ett steg till höger givet att man få ej gå utanför brädets gränser.

Vid början av spelet, placerar en robot ut N stycken pjäser på spelplanen på slumpmässiga valda platser/rutor. Motspelaren väljer sedan vilken av de slumpade pjäserna som får flyttas först. För att göra ett smartare drag måste vi hitta en pjäs bland de N-1 resterande som ligger närmast startpjäsen.

Problemet som vår algoritm ska lösa är just att söka bland pjäserna vilken som ligger närmast startpjäsen.

Problemsparameter

För att lösa problemet så bör vi bestämma hur vi kommer att representera de olika parameter som vi ska ta in i vår algoritm, hur dessa parametrar ska behandlas och vilka eventuella utdata som algoritm kan generera. Detta är viktigt ty hur problemet representeras påverkar väldigt mycket huruvida en eventuell lösning kan godtas.

Vi ska här representera spelplan som en graf med hjälp av en matris. Varje ruta kommer att representeras av ”ett objekt” som kommer att har fyra metoder som ger höger, vänster, upp och ner. Utöver dessa metoder kommer vi att kunna givet en ruta finna dess position X och Y, avstånd från startpjäsen, om rutan är ockuperad av en pjäs, om den är besökt under körningen.

Spelplanen kommer representeras som en graf och kommer att består av en NxN matris av sådana ovanbeskrivna ”objekt” som vi kallar celler.

Spelplan -> cell [][] spelplan = new [N][N]

Startpjäsen -> cell_start

Startpjäsen kommer att beskrivas av vilken cell den ockuperar.

Så den publika API för vår algoritm kommer att representeras på det här sättet:

distance(Cell [][] spelplan, cell startpjäsen)

Problem förståelse och natur

Som vi har det beskrivit ovan kommer vi att representera problemet som ett grafproblem på så sätt att spelplanen representeras med en NxN matris av ”objekt” celler. Problemets egentliga natur är att hitta det kortaste avståndet givet en startposition. Så vi kan enkelt reducera det här problemet till det kortaste avståndsproblemet i det avseende kommer vi att använda en lite omformulerad variant av Dijkistra Algoritm.

Anta en 5x5 matris där 5 olika pjäser (P1,P2,…, P5) har slumpmässigt placeras på matrisen (spelplan). De givna parameter kan representeras på det här sättet:

		P4		
	P1			
			P3	
P2				
				P5

Anta vidare att startpjäsen är P3. Pjäsen P3 befinner sig på position (X,Y) = (2,3). I början av algoritmen sätter vi P3.distance = 0 och P3.visited = true. Sedan samlar vi i en datastruktur P3.up, P3.down, P3.left, P3.right. För att se till att vi ”ej går utanför planen” har vi en struktur som för varje pjäs säkerställer att när vi går upp så är getPosRow() - 1 > 0, när vi går ner är getPosRow() + 1 < N, när vi går vänster så bör getPosCol() - 1 > 0 och slutligen när vi går höger ber getPosCol() + 1 < N. gäller inte detta får vi ett ”nil” värde. För varje sådana samlade grannar till startpjäsen sätter vi deras avstånd till P3.distance + 1 och för varje av deras grannar sätter vi P3.distance + P3.neighbours + 1 och så vidare. Till slut får vi det här resultatet:

5	**4**	**P4 3**	**2**	**3**
4	**P1 3**	**2**	**1**	**2**
3	**2**	**1**	**P3 0**	**1**
P2 4	**3**	**2**	**1**	**2**
5	**4**	**3**	**2**	**P5 3**

För att veta vilken av de N-1 pjäserna som ligger närmaste det n: te pjäsen så initialiseras vi en variabel cell nearest_piece som har distance = infinity och visited = false och sedan har vi en graf med alla slumpvalda pjäser. Medan vi beräknar avstånden för hela spelplanen kan vi nu för varje beräknad ruta verifiera om den pjäsen ligger bland de slumpvalda pjäser om så är fallet uppdaterar vi variabeln nearest_piece tills vi hamnar till den pjäsen som både finns bland de slumpvalda pjäser och ligger så närmast så möjligt startpjäsen.

Algoritm konstruktion

Vår nuvarande API ser ut på det här sättet distance(Cell [][] spelplan, cell startpjäsen, Cell [] slumpvalda). Givet de parametrarna använder vi Dijkstra algoritm.

Korrekthetsresonemang

Korrekthet till den här algoritmen följer direkt från Dijkstra algoritms korrekthet [1]. Vid varje iteration håller vi en invariant nämligen att de nuvarande beräknade avstånden är de kortaste från startasvtåndet. Denna invariant hålls igenom hela beräkningen genom att dels initialisera alla avstånd till infinity (det möjliga maximala avstånd) och dess visisted variabel till false dels genom att initialisera startpjäsen till 0 som avstånd till sig själv och göra den besökt. För varje iteration utgår vi från starpjäsen och från startpjäsen initialiserar vi dess grannar med avstånd från ”infinity” till startpjäs.distance + 1. Detta förfarande repeteras för varje rutan i spelplan där varje rutor besöks högt en gång och den invarianten hålls under hela körningen. Därför kommer resultat att ge det kortaste avståndet från startpjäsen. På grund av symmetrin så är avstånd från startpjäsen till alla andra pjäser dessamman oavsett vilken ”väg” man tar. Därför är algoritmen en reducerad version av Dijsktra algoritm. Algoritmen avslutas när alla rutor är besökt och dess mängd är tom.

Pseudokod

[1] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Algorithm 1 Shortest distance from start s

```
1: procedure distance(Cell [ ] [ ] plan, Cell start, Cell [ ] slump)
2:     cell nearest ← new cell()
3:     cell [ ] frontier ← new cell()
4:     cell [ ] Link
5:     cell [ ] new_frontier
6:     plan[start.posRow][start.posCol].dist ← 0
7:     plan[start.posRow][start.posCol].visited ← true
8:     frontier.add(start)
9:     while !frontier.isEmpty() do
10:        new_frontier ← new cell()
11:        Link ← new cell()
12:        for each cell i in frontier do
13:            cell current ← frontier.get(i)
14:            Link.add(current.getDown(n))
15:            Link.add(current.getLeft(n))
16:            Link.add(current.getRight(n))
17:            Link.add(current.getUp(n))
18:            for each cell i in Link do
19:                if Link.get(j) ! = null then
20:                    if !plan[Link(j).posRow][Link(j).posCol].visited then
21:                        plan[Link(j).posRow][Link.(j).posCol].dist ← current.dist + 1
22:                        plan[Link(j).posRow][Link(j).posCol].visited ← true
23:                        new_frontier.add(plan[Link(j).posRow][Link(j).posCol])
24:                        if slump.contains(plan[Link(j).posRow][Link(j).posCol]) then
25:                            if nearest.dist > plan[Link(j).posRow][Link(j).posCol].dist then
26:                                nearest ← plan[Link(j).posRow][Link(j).posCol]
27:                            end if
28:                        end if
29:                    end if
30:                end if
31:            end for
32:        end for
33:        frontier ← new_frontier
34:    end while
35:    Return nearest
36: end procedure
```

Tidskomplexitet

Den övre gränsen för vår algoritm är $O(n^3)$

Motivation för tidskomplexitet

Vi kommer att itererar igenom NxN rutor för att beräkna ruta/pjäs relativa avstånd till startpjäsen. För varje sådana beräkningar verifierar i värsta fall i N steg om den nuvarande pjäsen finns bland de slumpvalda. Den totala komplexiteten är därför $O(n^3)$.

Det finns säkert ett enklare sätt att implementerar detta på $O(n^2)$ kanske ännu snabbare. Men den lösningen duger.