Luleå tekniska universitet
Skellefteå

Laborationsrapport

Kurs: S0006D, Datorspels AI

Labb 2

Morgan Nyman

mornym-9@student.ltu.se

Källkod: github.com

15 februari 2021

Luleå tekniska universitet
Skellefteå

Problemspecifikation

Läs in tre kartor: Map1.txt, Map2.txt, Map3.txt och använd pathfinding algoritmer för att traversera från början till målet.

Användarmanual

Projektet finns på https://github.com/mogggen/AI_Lab_1

Det finns 5 färgade kvadrater:

Röd är startpunkten, S i textform.

Grönt är Målet och är i textform G.

Blå är en unwalkable, spelaren kan inte gena runt hörn som är blå eller traversera över dem, som representeras med ett X i textformat.

Grå är den traversering som utförs av respektive algoritm.

Vit/Oföränderlig walkable (, som antingen är 0 eller mellanslag i textformat.

I Lab 2.py finns en “” function. Ändra decimaltalet i inuti parenteserna i “time.sleep(0)” för att ändra flödet av tiden.

Agenterna ges ett startvärde för varje respektive behov där startvärdet är mellan 6-99 inklusivt. Detta ger de olika agenterna olika förutsättningar. målet är att hålla alla behov, också känt som states, mellan ett värde på större än 0 och mindre än 100 samtidigt för alla states och per agent.

Algoritmbeskrivning

Depth-first search går i en riktning tills det tar stopp, sen byter algoritmen riktning tills den antingen har klivit på alla noder i grafen eller hittat målet.

Breadth-first search besöker en nods alla grannar innan den går till en av grannarnas grannar och besöker alla kommande grannar till alla grannar är besökta eller målet är hittat. Den första stig som är funnen kommer att vara den kortaste eftersom algoritmen går åt alla håll samtidigt, i en cirkelform som växer runt start noden.

A* (A star) ger varje närmaste granne en vikt som består av antalet steg från start summerat med längden i kvadrat.
Egen Algoritm Använder sig av depth-first search men går endast vertikalt och horisontellt.

Systembeskrivning

Hela systemet är klasslöst och består av listor, tuples och dictionaries, och kombinationer av dessa. Lite som en serialisering.
Noderna lagras som en nyckel i ett (key, value)-pair. Där value är en lista med först nodens typ, som läses in från respektive karta där de följande tuples är grannarnas koordinater i grafen. I fallet för A* så genereras f-värdet från $f = g + h$ direkt i algoritmen så att komplexiteten räknas, eftersom ingen annan algoritm behöver väga sin graf.
T.ex:
noder = [{ (1, 1) : ‘S’, (1, 2), (2, 1), (2, 2) }, { (1, 2) : ‘0’, (1, 1), (2, 1), (2, 2), (2, 3), (1, 3) }]

Lösningens begränsningar

Astar är inte garanterad att hitta den kortaste vägen, för den ser bara en granne “in i framtiden åt gången” och kan inte planera en ännu bättre passage än vad den ser just i nästa fas.
DFS och BFS är Brute force bara och är ej vägda grafer.

Diskussion

Jag störde mig på hur lång tid det tog att rita ut kartorna, men det slutade med att jag lade ner för mycket tid på att optimera det, än vad jag hade tid med.
Jag hittade bra och konkreta exempel av algoritmernas principer som jag enkelt kunde implementera i min egen version.
Det var förbryllande att min variant av depth-first search som inte tillåter diagonala steg var snabbaste vid två av tre kartor på de fyra tusen testkörningar som jag gjorde, med en säker andraplats i den första kartan. Med en så liten ändring.

Testkörning

4000 iterationer	Map 1	Map 2	Map 3
dfs	0,063	0,132	7,776
bfs	1,713	0,443	14,606
A*	0,171	0,248	1,270
custom	0,084	0,210	0,314

Alla mått är i millisekunder (ms)