Luleå tekniska universitet

Skellefteå

Laborationsrapport laboration 1b

S0006D

Laboration 1b

Pontus Stenlund

Ponste-5@student.ltu.se

Laboration_1b i SVN

2017-02-15

Innehåll

Pontus Stenlund, ponste-5@student.ltu.se

1. Problemspecifikation

Uppgiften gick ut på att implementera fyra stycken pathfinding algoritmer för att hitta en väg från start till mål i givna kartor.

Vi skulle även jämföra tiden mellan olika algoritmerna på de olika kartorna.

De algoritmer som vi skulle implementera är följande:

A* algoritmen.

Bredden först.

Djupet först.

En egen algoritm.

Jag anser att jag har uppnått lösningen för betyg 3.

2. Användarhandledning

Projektet finns på SVN i mappen som heter S0006D AI/Laboration_1b.

2.1 Köra lösningen

Det första som måste göra är att lägga sina kartor i D:. Annars kommer inte programmet att hitta filen och då fungerar ingenting.

För att köra min lösning så måste startar man stingray och sedan köra spelet med ”Test level” eller ”Run project”.

För att ladda de olika banorna så trycker man 1,2 eller 3 på tangentbordet.

För att köra A* algoritmen så trycker man ”E”.

För att köra Breadth-First Search så trycker man ”R”.

För att köra Depth-First Search så trycker man ”F”.

För att köra min egen algoritm så trycker man ”T”.

För att ladda om banan för att köra nästa algoritm så trycker man banans knapp och sedan algoritmen man vill köra. Detta reloadar banan och tar bort vägen man har fått från föregående test.

3. Algoritmbeskrivning

3.1 A*

Är väldigt lik Dijkstra's algoritm i och med att man vill hitta kortaste vägen mellan två punkter. Det gör vi genom att varje nod har värden i sig som beskriver kostnaden för att gå till den noden.

Heuristiska värdet (H-värde): Hur långt det är till målet i steg från given nod. I min implementation av A* så har jag använt Manhattan Distance för att räkna ut kostnaden.

G-värdet: Är summan av alla förflyttningar man har gjort hittills från startpunkten.

F-värdet: Är totala kostnaden av H+G.

Genom att beräkna alla kostnader och sedan välja den nod som har lägst F kostnad så kan vi få ut den kortaste vägen från start till mål.

Algoritmen använder sig av en öppen och stängd lista för att hålla reda på den kortaste vägen. Den öppna listan innehåller alla noder vi har utvärderat.

Den stängda listan innehåller alla noden vi har gått till.

1. Vi adderar startnoden till den öppna listan.
2. Vi repeterar följande:
3. Vi kollar i den öppna listan med noden med lägst F-kostnad och sätter den som aktuell nod.
4. Vi flyttar den till den stängda listan.
5. Kollar de åtta noder som ligger runt oss. Så länge noden går att gå på och inte redan finns i den stängda listan så lägger vi till den i den öppna listan och sätter aktuell nod som förälder.
   Om den redan finns i den öppna listan så tar vi och jämför G-kostnaden och om den är lägre så ändrar vi föräldern till aktuell nod och räknar om G- värde och F- värde.
6. Vi stannar när lägger till målnoden till den öppna listan eller om den öppna listan är tom. Om vägen är funnen så går vi tillbaka från målnoden och kollar föräldrarna till vi har nått startnoden. Då har vi funnit vår väg.

3.2 Breadth-First Search

Breadth-First Search går kortfattat ut på att från startnoden så besöker vi våra grannar och sedan våra grannars grannar o.sv. tills vi har nått målet.

För att förklara ytterligare så gör vi följande steg.

Vi skapar en lista som kommer att innehålla alla noder vi har besökt. Sedan lägger vi till startnoden i en kö. Sedan loopar vi följande.

1. Vi sätter vår aktuella nod från köns början.
2. Kollar ifall den noden är målnoden. Uppfyller den kravet så går vi baklänges tillbaka till start genom att titta på föräldrarna.
3. Annars så kollar vi på de närmsta noderna som vi kan besöka. Sedan lägger vi till dem ifall inte finns i besökslistan och till kön. Och vi sätter föräldern till aktuell nod.

3.3 Depth-First Search

Depth-First Search går ut på vi besöker en granne och dennes granne tills vi inte kan gå längre. Har vi inte nått målet så backar vi och väljer en ny granne och fortsätter tills vi har besökt alla grannar eller hittat målet.

För att implementera depth-first search så använder vi oss av en stack med besökta grannar.

1. Vi börjar med att lägga till start till stacken.
2. Medans stacken inte är tom så upprepar vi följande:
   a. Vi sätter aktuell nod från toppen av stacken och sedan tar bort den från stacken.
   b. Vi kollar om den leder någonvart. Om den har en eller flera grannar så besöker vi en av dom och upprepar proceduren. Annars går vi tillbaka tills vi hittar en granne som vi inte har besökt och går dit.

3.4 Contour Tracing

Min egna algoritm är contour tracing. Den fungerar ungefär som depth-first-search men den går alltid åt ett håll (i detta fall vänster) tills den inte kan gå längre. Då väljer den ett annat håll och går dit tills den kan gå åt vänster igen. Och sedan upprepas detta tills målet har hittats.

4. Systembeskrivning.

4.1 Project.lua

Är en fil som skapas automatiskt när man skapar projektet i Stingray.

I denna fil så har jag lagt till följande saker:

LoadMap(Path), laddar in kartan från den givna sökvägen. LoadMap anropar sedan ett plugin som är skrivet i C++. Detta plugin läser raderna från filen som anges och lägger till dem i ett lua table. Sedan går LoadMap igenom raderna och kollar på varje element i strängen och kollar ifall den sedan ska lägga till en vägg eller golv. Den sparar även positionerna på start och mål. Golven och väggarna läggs sedan till i separata tables för att kunna iterera dem senare.

DestroyMap(), går igenom listorna för golv och väggar för att sedan plocka bort dom från spelvärlden.

RemoveAgentsFromList(), tar bort de objekt som läggs till i världen när man har kört algoritmerna.

Alla algoritmers funktioner anropas från Project.update.

4.2 Pathfinding_a

ManhattanDistance(currentNode, goalNode), räknar ut antaget steg i x-led och y-led från currentNode till goalNode och returnerar antal steg.

DistanceBetweenNodes(NodeA, NodeB), räknar ut avståndet mellan två noder genom att använda pytagoras sats och returnerar längden.

NodeIsValid(neighbor), kollar ifall noden går att gå på. Returnerar ett booleskt värde.

LowestFScore(nodeList, Fscore), kollar i den angivna listan efter en nod med den lägsta F-kostnaden och returnerar den.

NodeIsTheSame(nodeA, nodeB), kollar ifall de är samma nod. Returnerar ett booleskt värde.

CheckNeighbors(currentNode, Nodes, closed), itererar den listan som innehåller alla noder och letar efter de intilliggande noderna som den sedan lägger till i den öppna listan om den inte redan finns. Om den inte finns i öppna listan så sätter vi G-värdet. Funktionen kollar också ifall det finns en vägg framför, bakom till höger och vänster. Om det finns det så ignorerar den alla noder som finns snett framför/bakom. Detta är till för att förhindra att man ”genar” runt hörn på väggar.

Den returnerar en lista med alla noder som finns runtom som går att gå på.

NotInNodeList(nodeList, currentNode), kollar så att noden man anger inte finns i den angivna listan. Returnerar ett booleskt värde.

RemoveNode(nodeList, currentNode), tar bort den angivna noden från nodeList.

UnravelPath(foundPath, currentNode), medans föräldern till currentNode inte är nil så lägger vi till den till foundPath och sätter om currentNode till föräldern.

Returnerar en lista med noder som ger oss vår väg till målet.

AStar(start, goal, nodes), startar algoritmen och sätter in noderna för den bästa vägen till klassens lista bestPath.

4.3 Bfs_pathfinding

TableContainsNode(nodeTable,node), kollar så att noden man anger inte finns i den angivna listan. Returnerar ett booleskt värde.

IsTheSameNode(nodeA, nodeB), kollar ifall de är samma nod. Returnerar ett booleskt värde.

CheckChildren(currentNode, nodeList, visitedList), kollar ifall noden har några barn och lägger till den i en lista och returnerar den.

FoundPath(nodePath, targetNode), går från targetNode till start genom att gå genom nodernas föräldrar tills den hamnar på start.. Innan den går till föräldern lägger den till den aktuella noden i en lista. När den har gått ut loopen så sätter den klassens lista pathFound till listan med vägen till start.

BreadthFirstSearch(graph, startNode, goalNode), startar sökningen till goalNode.

4.4 Dfs_pathfinding

TableContainsNode(nodeTable,node), kollar så att noden man anger inte finns i den angivna listan. Returnerar ett booleskt värde.

IsTheSameNode(nodeA, nodeB), kollar ifall de är samma nod. Returnerar ett booleskt värde.

CheckChildren(currentNode, nodeList, visitedList), kollar ifall noden har några barn och returnerar det barn som hittas.

FoundPath(nodePath, targetNode), går från targetNode till start genom att gå genom nodernas föräldrar tills den hamnar på start.. Innan den går till föräldern lägger den till den aktuella noden i en lista. När den har gått ut loopen så sätter den klassens lista pathFound till listan med vägen till start.

DepthFirstSearch(graph, startNode, goalNode), startar sökningen till goalNode.

4.5 Contour_tracing

CheckChildren(currentNode, nodeList), kollar ifall den kan har någon nod den kan gå till. Om den har det så returnerar funktionen den noden. Annars returneras föräldern till currentNode.

BackTrace(nodePath, targetNode), går från targetNode till start genom att gå genom nodernas föräldrar tills den hamnar på start.. Innan den går till föräldern lägger den till den aktuella noden i en lista. När den har gått ut loopen så sätter den klassens lista pathFound till listan med vägen till start.

CheckEnd(graph, goal), kollar hur många noder i kartan man kan gå på och returnerar talet.

ContourTracing(graph, startNode, goalNode), startar sökningen.

4.6 Node

SetPosition(xPos, yPos), sätter nodens position.

IsWalkable(val), sätter att noden går att gå på.

SetParent(parentNode), sätter nodens förälder.

SetFValue(), sätter nodens F-värde genom att addera nodens G och H värde.

4.7 Queue

Push(node), lägger node längst bak i listan och ökar storleken med ett.

Pop(), hämtar ut den nod som ligger först i listan, tar bort den, minskar storleken med ett och returnerar noden.

4.8. Stack

Push(node), lägger noden längst back i listan och ökar storleken med ett.

Pop(), hämtar ut den nod som ligger längst bak i listan, tar bort den, minskar storleken med ett och returnerar noden.

5. Lösningens begränsningar

5.1 Begräsningar

5.1.a A*

Den största begränsningen i min implementation av denna algoritm är att jag itererar igenom listan med alla noder varje gång jag ska hitta en ny nod att gå till. Detta medför att körtiden ökar något.

En annan begränsning är att när jag kollar efter grannar så gör jag upp väldigt många jämförelser vilket gör att körtiden ökar något.

5.1.b Breadth-First Search
En begränsning i denna algoritm är att jag inte väljer vilken granne jag ska gå till först på måfå. Detta kan i sin tur leda till att jag hela tiden kommer att välja den sämsta grannen att gå till. Detta beror mycket på hur kartan är uppbyggd.

Som i A* så blir det många iterationer genom listan med alla noder.

5.1.c Depth-First Search
Samma begränsningar som breadth-first search förutom att begränsningen blir värre i denna algoritm.

5.1.d Contour Tracing
Algoritmen med störst begränsning. Eftersom jag bara går till en nod så tar den en väldigt dålig väg, speciellt om kartan har större öppna ytor.

5.2 Hur kunde begränsningarna undvikas?

5.2.a A*

Genom att använda skapa en tvådimensionell lista så skulle jag kunna använda mig av listans index för att kolla grannarna. Detta gör så att jag inte behöver iterera listan.

5.2.b Breadth-First Search
Genom att ta välja vilken granne som jag besöker först på måfå så kan jag välja rätt granne direkt. Detta kan lösa problemet att den hela tiden tar fel granne på en viss karta.

5.2.c Depth-First Search
Samma lösning som i breadth-first search.

5.2.d Contour Tracing
Genom att använda mig av ett värde som sätts när jag besöker noden så skulle jag kunna jämföra det med grannens värde och om den har någon väg till den skulle jag bara kunna ändra den aktuella nodens förälder till den granne med lägre värde.

6. Diskussion

Jag har stött på ett par problem.

Det största problemet som har varit är att jag är inte van vid lua som språk. Detta gör att implementationerna har varit tidskrävande och omständliga.

Ett annat problem jag stött på är att stingray har lite speciell syntax för att skapa objekt och sätta ut dom i världen.

Och att försöka göra en objektorienterad lösning har varit ett litet problem genom att jag i början inte förstod att när jag skapar ett objekt och försöker skapa ett till objekt så fick jag bara pekaren till det första objektet.

Allt som allt tycker jag laborationen har varit väldigt rolig och givande. Den har gett bra insikt i hur man ska kunna hitta en väg till en given plats.

Jag har tagit information och inspiration från:

https://github.com/lattejed/a-star-lua/blob/master/a-star.lua

https://en.wikipedia.org/wiki/Depth-first_search

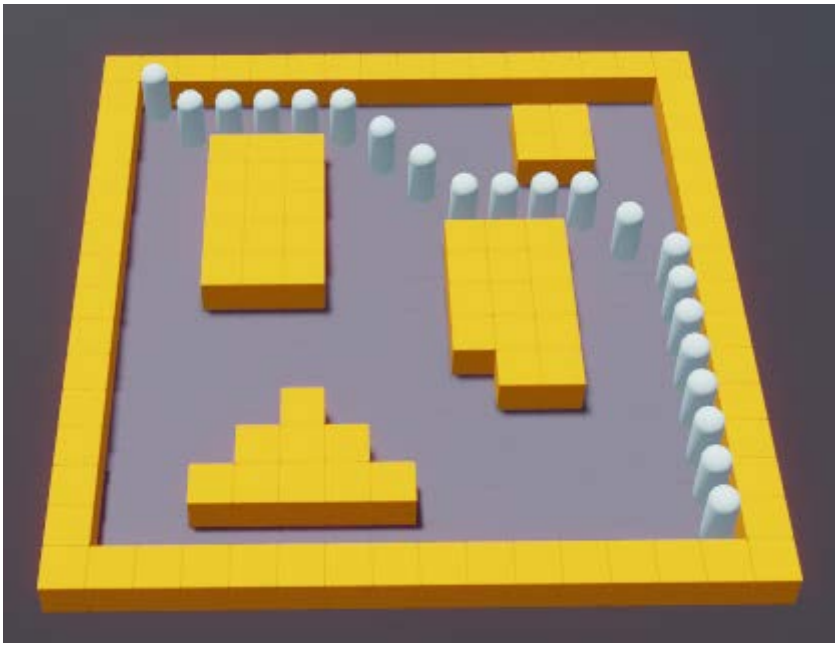
https://en.wikipedia.org/wiki/Breadth-first_search

Jag har även läst de dokument som Patrik har lagt ut på canvas för att hitta inspiration.

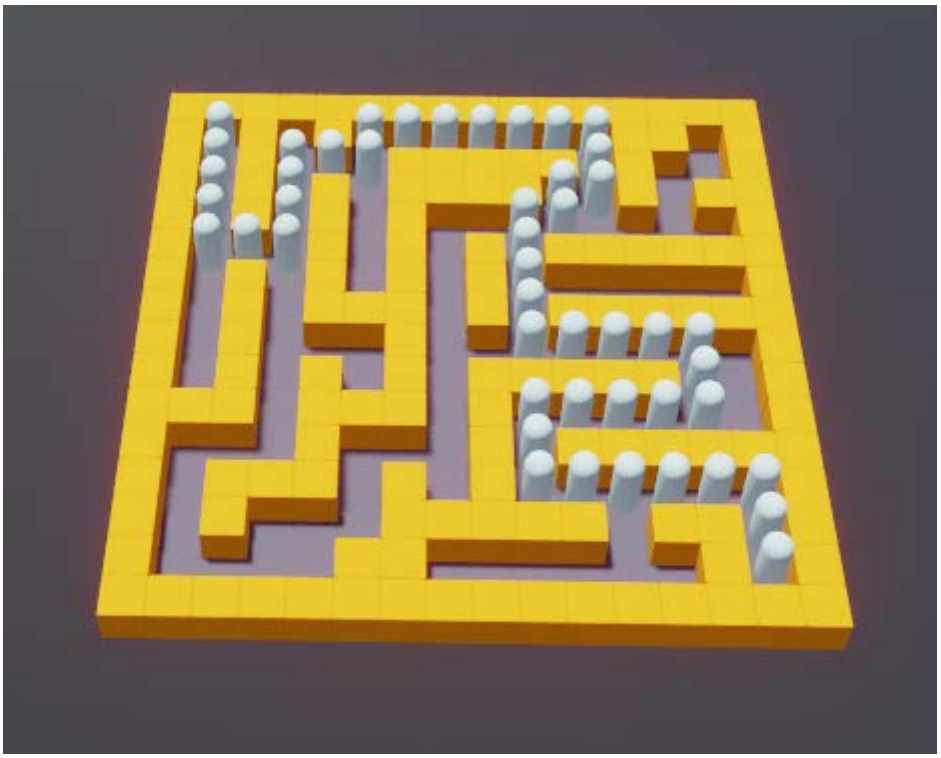
Jag vill tacka alla mina klasskamrater som har gett mig goda råd och för alla givande diskussioner som vi har haft angående labben.

7. Testkörningar

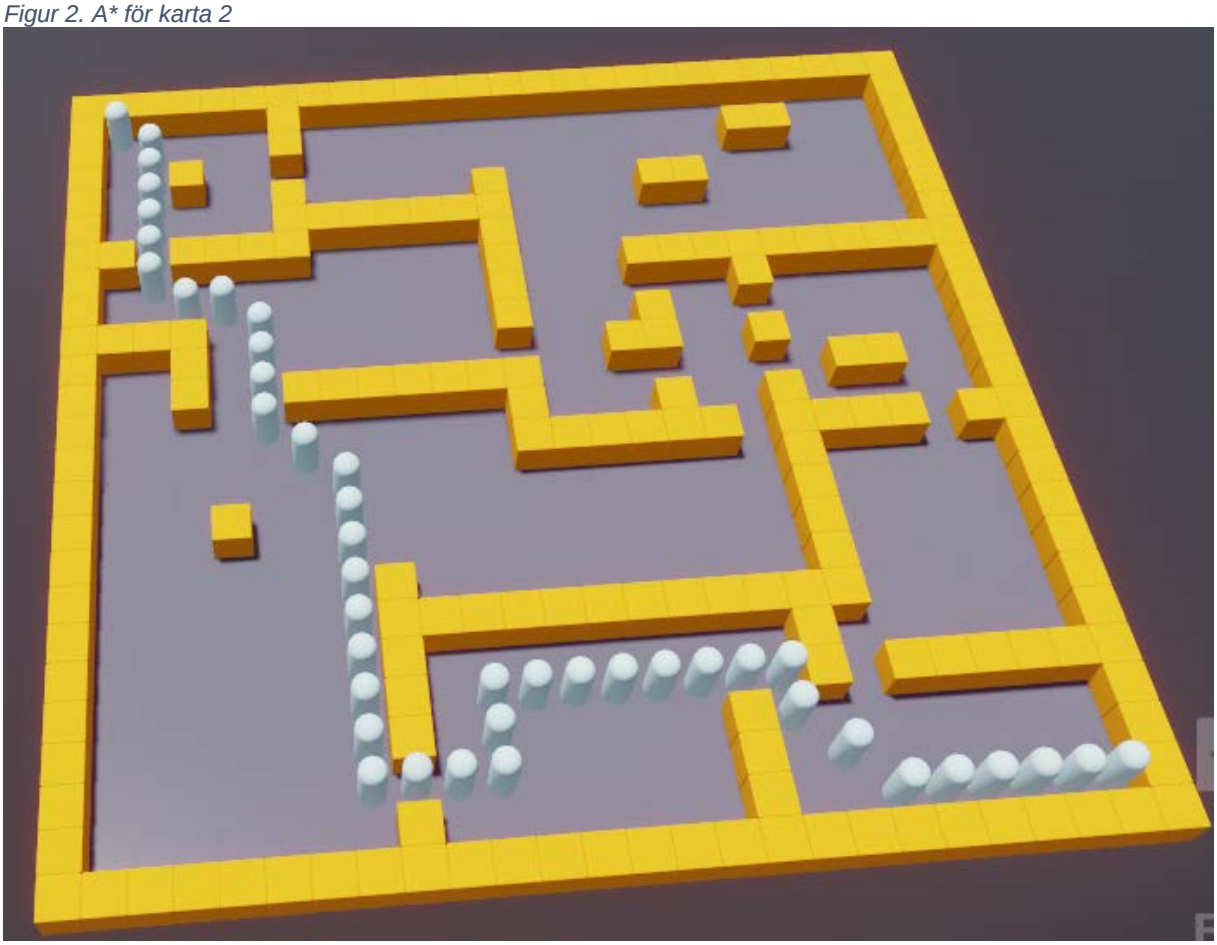
7.1 A*



Figur 1. A för karta 1*



Figur 2. A för karta 2*

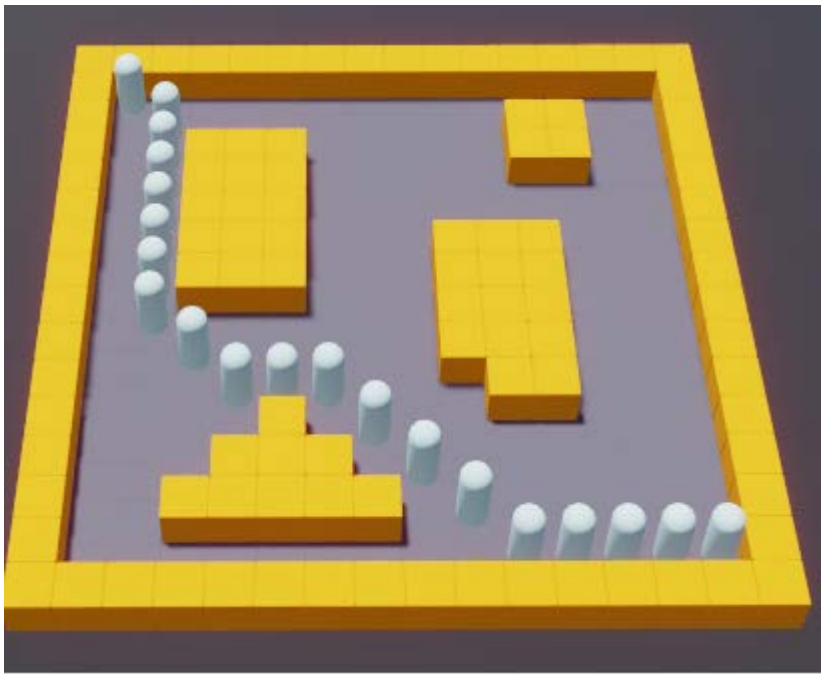


Figur 3. A för karta 3*

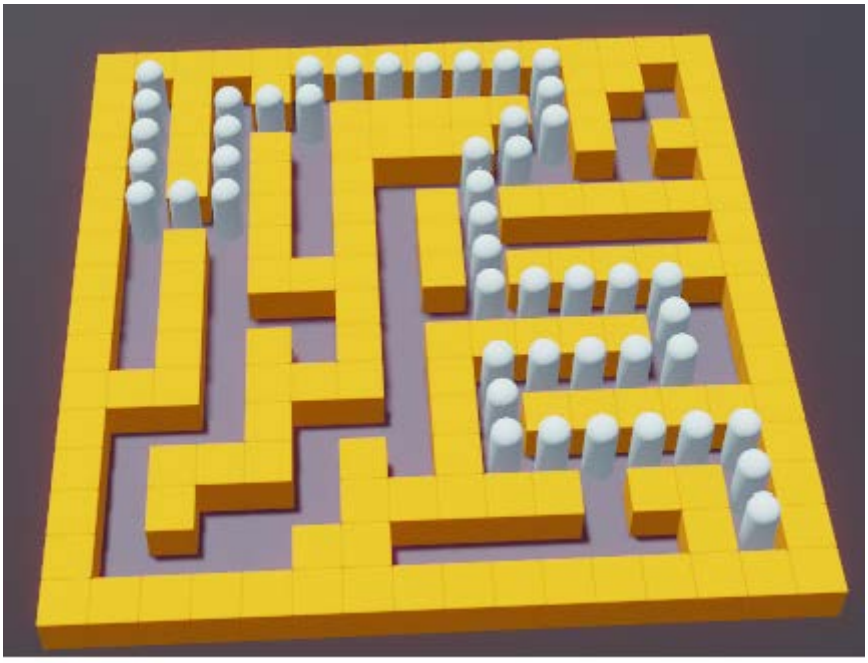
Tidtagning för A*.

Karta 1	0.0039999999999054 sekunder
Karta 2	0.02100000000064 sekunder
Karta 3	0.30400000000009 sekunder

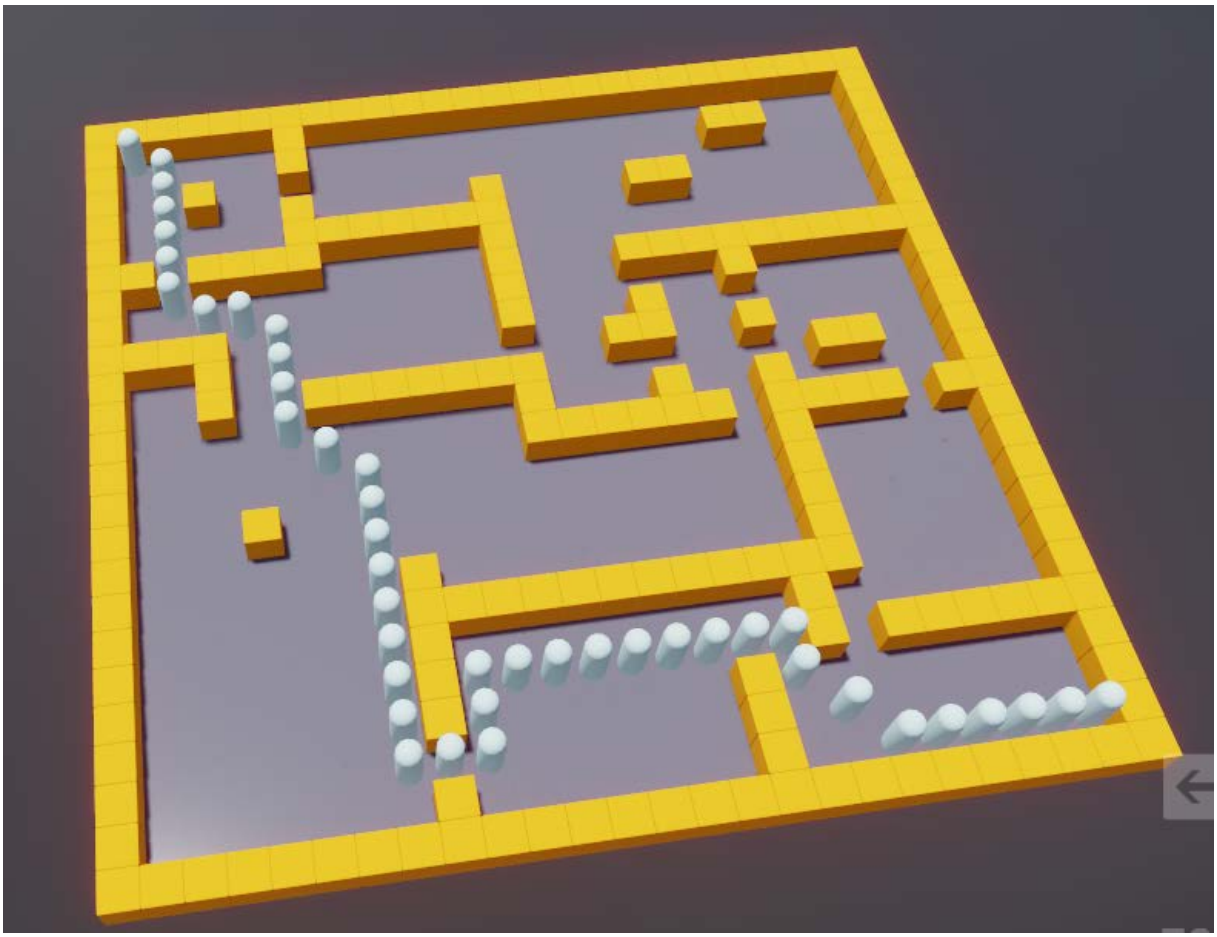
7.2 BFS



Figur 4. BFS för karta 1



Figur 5. BFS för karta 2

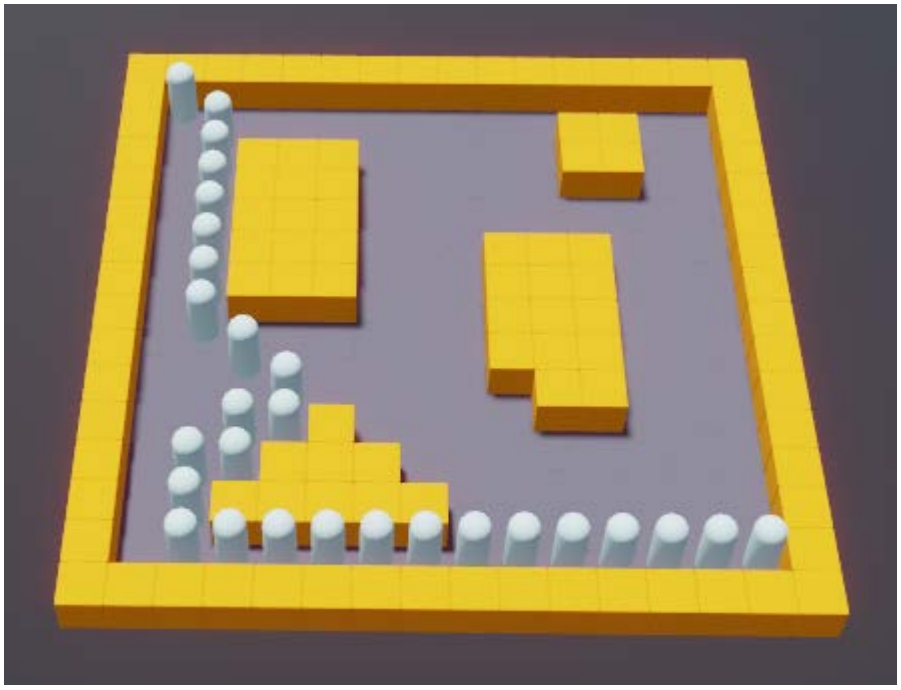


Figur 6. BFS för karta 3

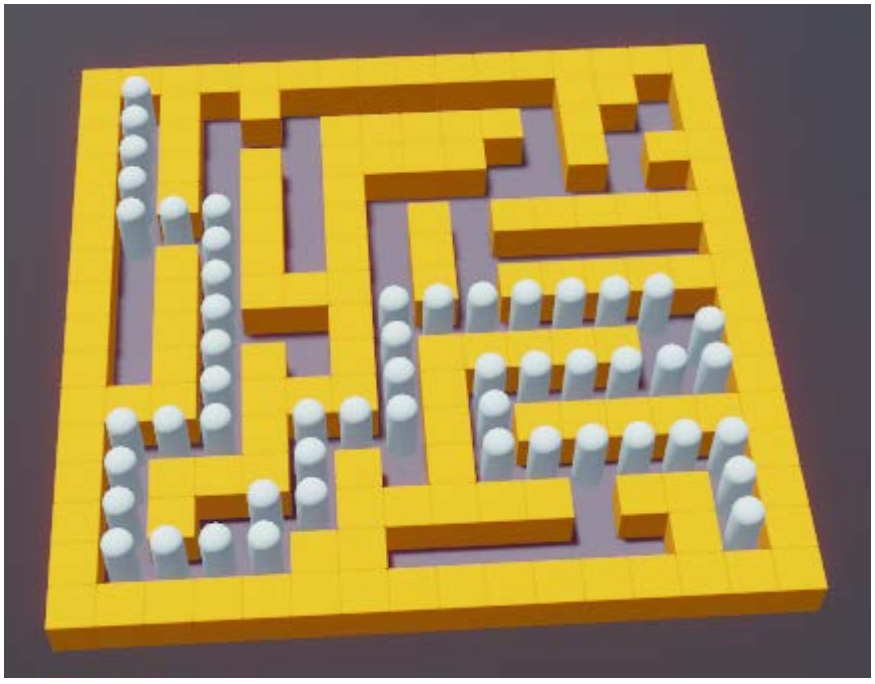
Tidtagning för BFS

Karta 1	0.024000000000342 sekunder
Karta 2	0.018000000000029 sekunder
Karta 3	0.17599999999948 sekunder

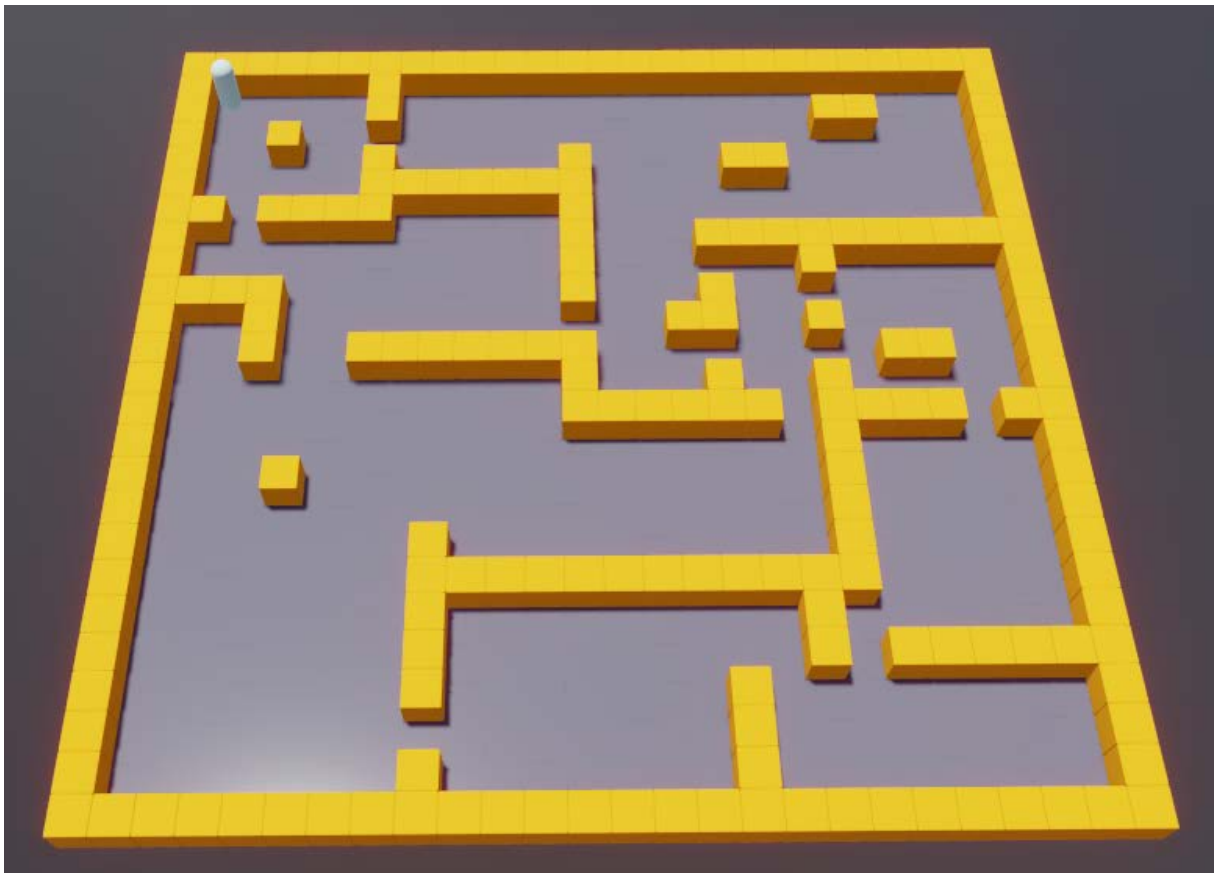
7.3 DFS



Figur 7. DFS för karta 1



Figur 8. DFS för karta 2

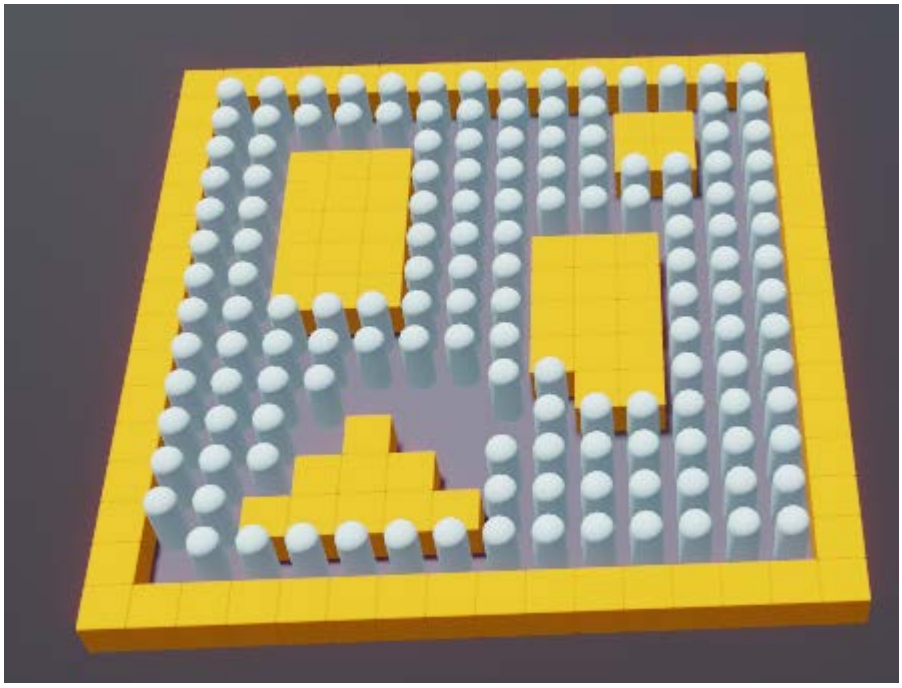


Figur 9. DFS för karta 3

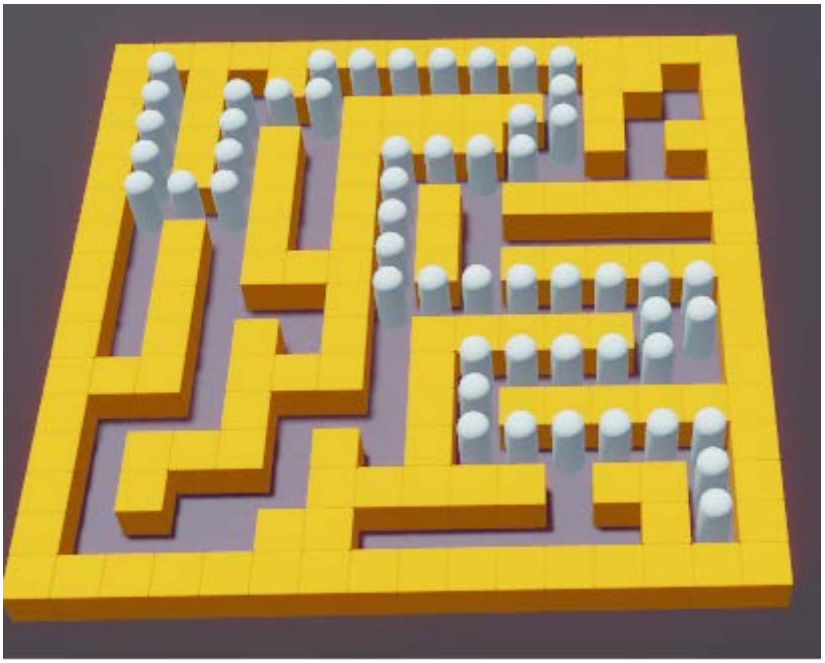
Tidtagning för DFG

Karta 1	0.0039999999999054 sekunder
Karta 2	0.010000000000218 sekunder
Karta 3	0.11799999999948 sekunder

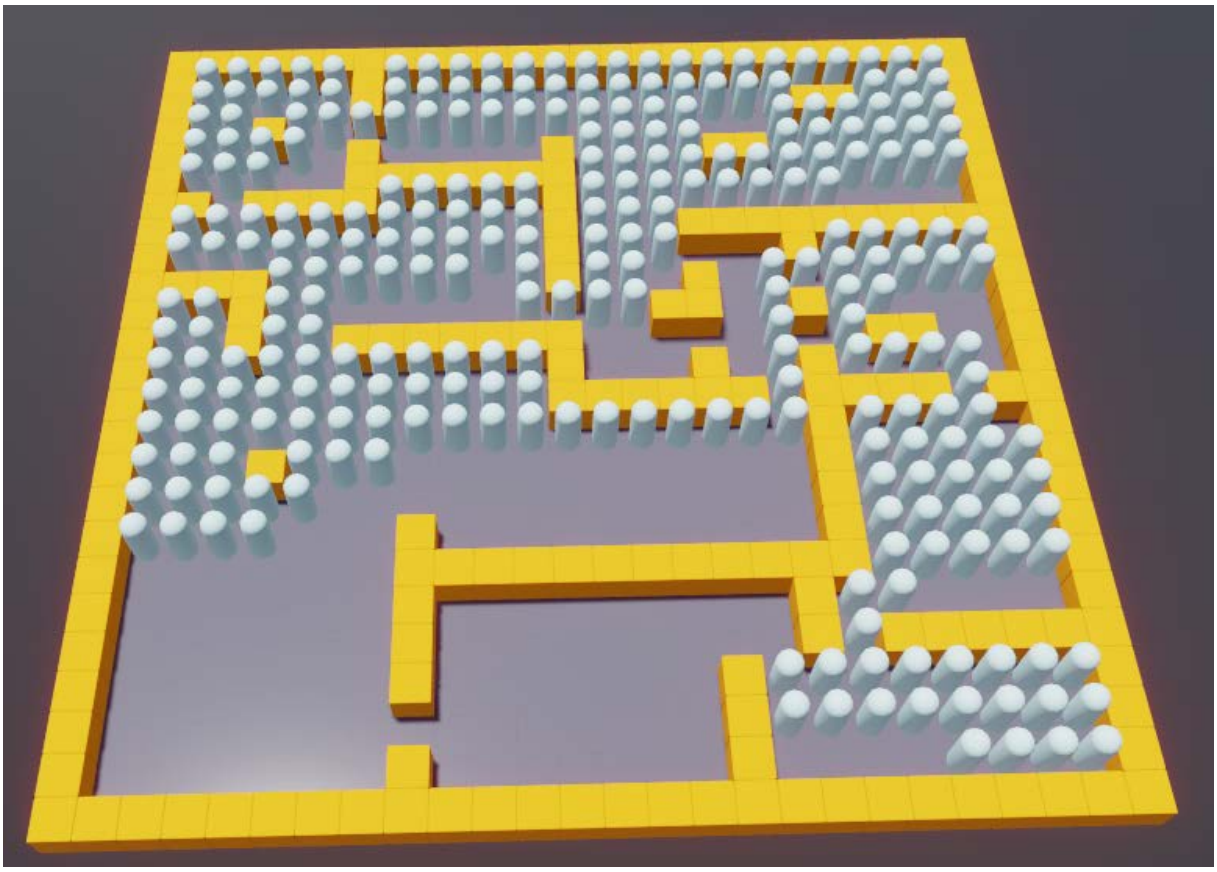
7.4 CT



Figur 10. CT för karta 1



Figur 11. CT för karta 2



Figur 12. CT för karta 3

Tidtagning för CT

Karta 1	0.00099999999929423 sekunder
Karta 2	0.0010000000002037 sekunder
Karta 3	0.0039999999999054 sekunder

7.5 Slutsats

Den slutsats som jag drar är att A* är mest effektiv att hitta den snabbaste vägen till givet mål.

Breadth-First Search ger också en ganska bra väg på kort tid.

Depth-First Search ger en väg men den är lite konstig men fördelen är att den tar lite tid.

Contour Tracing är den värsta av dom alla. Den är förvisso snabbast men den ger den dummaste vägen till målet.