Högskolan i Gävle

Mazerunner

Projektarbete

Sofia Ågren
Sofiaagren1@hotmail.com
Andreas Roghe
Andreas.roghe@gmail.com

2021-01-06

Kurs: Algoritmer och datastruktur

Lärare: Anders Jackson
Lärare/handledare: Hanna Holmgren

Innehållsförteckning

1 Inledning

Detta projekt gick ut på att skapa en slumpmässig labyrint och sedan skapa en algoritm som löser labyrinten, antingen via kortaste vägen eller enbart tills den hittar utgången. Labyrinten skall generera slumpmässigt olika labyrinter varje gång, storlek skall gå att välja av användaren. Labyrinten skall även sparas ner för att återanvändas. Två olika generatorer som skapar labyrinter skall skapas även två olika algoritmer för att lösa labyrinten.

1.1 Syfte

Syftet med detta projekt var att förstå att olika algoritmer har olika tidseffektivitet. Användandet av abstrakta datatyper och hur klasser kan separeras för att frikoppla dessa från varandra och göras generella för att kunna återanvändas.

1.2 Bakgrund

En algoritm är en detaljerad sekvens för att utföra en viss uppgift, en algoritm måste ha ett avslut annars är det ej en algoritm. Algoritmens tid för att lösa en uppgift beror på ordonaliteten.

1.2.1 Aldous-Border algoritm

Denna algoritm går ut på att slumpmässigt gå igenom labyrinten, den tar bort celler som inte har varit besökta förut. Denna algoritm är inte så effektiv.

1.2.2 Depth-First algoritm

Denna algoritm går ut på att traversera igenom ett träd, den börjar vid root-noden och går igenom alla grenar så långt det går sedan går den tillbaka.

2 Metod

För att skapa labyrinter och lösa dem behövs klasser som hanterar just detta. Genom att skapa en klass som genererar en labyrint och en som löser den. För att frikoppla dessa klasser ifrån varandra behövs abstrakta datatyper användas så som interface och abstrakta klasser. När detta görs blir det möjligt att skapa olika labyrinter med olika algoritmer och desamma med lösningar.

2.1 Interface

Interface skapades för att skapa en större abstraktionsnivå genom objekt-orienterad programmering.

2.1.1 Maze

Samtliga generatorer som bygger på den abstrakta klassen `AbstractGenerator`, klassas som en `Maze` i slutändan. Med de sätts ett villkor att `AbstractGenerator` måste implementera metoden `getMaze()`.

2.1.2 Solver

Detta interface har endast en uppgift och det är att anropa metoden `solveMaze()` som finns i `MouseAlgorithm` och `MazeSolver`.

2.2 Abstrakta generatorn

Denna klass är en abstrakt implementation av hur en generator för en labyrint skall se ut. Den implementerar interfacet `Maze`. I denna klass sätts orienteringen för vilka vägar som kan tas i form av norr, syd, öst och väst. En hållare för labyrinten i form av en dubbel array för cellerna skapas i denna klass. Även bredden och höjden på de olika labyrinterna som skall skapas.

Denna klass har en inre klass som representerar en cell eller en vägg för att hålla reda på nuvarande, föregående och nästa cell. Den håller även reda på om en cell har blivit besökt eller ej och sätter koordinater för dessa.

2.3 Algoritmer för skapandet av labyrinter

De två algoritmer som användes för att skapa labyrinter var Aldous-Border och Depth-First, dessa två är snarlika i koden men vid testning av skapandet av labyrinter märks skillnad i tidsåtgången.

2.3.1 Aldous-Border

Denna algoritm bygger på att besöka alla obesökta celler som ligger i datastrukturen Lista. Algoritmen är klar när samtliga celler i listan är besökt. När en cell är besökt markeras den som besökt och tas bort från listan. Eftersom algoritmen bygger på att slumpmässigt ta fram en ny riktning är villkoret att inte göra något med en nod som redan är besökt. Om så fallet att det händer, slumpas en ny riktning ut.

Ordonaliteten på denna algoritm är $O(n^2)$.

2.3.2 Depth-First

Klassen `DepthFirstSearch` ärver av `AbstractGenerator` för att kunna använda des metoder för att skapa labyrinter. Konstruktorn i denna klass tar emot storleken av den valda labyrinten som skall skapas och skickar vidare den till `AbstractGenerator` som sätter upp grunden för labyrinten genom att fylla matrisen med väggar och sätter ut de tänkta rummen.

Denna algoritm är en rekursiv algoritm. Metoden `depthAlgorithm()` tar en slumpmässig cell och väljer en slumpmässig riktning den skall gå, sätter cellen till besökt och anropar sig själv igen med en ny cell som den har går till, tills labyrinten är klar.

Ordonaliteten på denna algoritm är O(log(n)).

2.4 Algoritm för att lösa labyrinter

De algoritmer som användes för att lösa labyrinterna var Brute-Force och `MouseAlgorithm`. Brute force har ordonaliteten O(log(n)) medan `MouseAlgorithm` är oberäknelig och har $O(n^2)$ och O(log(n)) beroende på vilken algoritm som används för att skapa labyrinten.

2.4.1 MazeSolver

Denna klass håller reda på den aktuella informationen om en specifik väg i labyrinten. En inre klass som hanterar noder och skapar listor för dessa. `MazeSolver` implementerar interfacet `Solver`.

Denna klass använder sig av `Brute-Force` som kollar alla noder i labyrinten, den spara alla vägar till den sista noden med hjälp av pekare på föregående nod. Till slut jämförs samtliga vägar för att den nod som har den kortaste distansen ”vinner” och det blir den slutliga vägen.

2.4.2 MouseAlgorithm

Även denna klass är till för att lösa en labyrint genom att hitta ”osten”. Samtliga noder som ”Musen” besöker läggs till i Listan `path`. När väl ”Musen” har hittat osten skrivs samtliga noder ut som har besökts med en asterisk. Vid en korsning markerar musen vilka möjliga vägar som finns genom att lägga till dem i en separat lista. Sedan slumpar ”Musen” vilken väg utav de närliggande möjliga vägar den ska gå, den har då 1–3 möjliga vägar. Sedan fortsätter den till nästa korsning och upprepar valet av de möjliga vägarna. Kommer den till en ”dead end” vänder den sig och byter håll för att sedan leta efter nästa korsning. Även `MouseSolver` använder även interfacet `Solver`.

2.5 MazeRunner

`MazeRunner` var huvudklassen för programmet. Den skapar labyrinter med hjälp av `DepthFirst` och `AldousBorderAlgorithm`. Användaren får välja storlek och vilken algoritm som skall användas när labyrinten skapas och sedan vilken algoritm som skall användas för att lösa labyrinten.

`ChooseMaze`-metoden i denna klass frågar efter användares val av algoritm för att skapa labyrinten, storleken och vilket namn filen skall ha som skall spara labyrinten. Sedan anropas den önskade algoritm-klassen.

`ChooseSolver`-metoden låter användare välja vilken algoritm för att som skall användas för att lösa labyrinten.

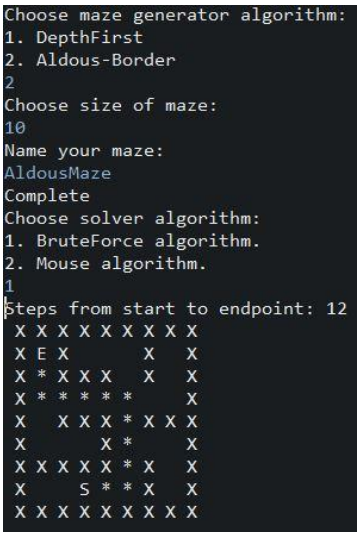
`PrintToFile`-metoden sparar filen med labyrinten med hjälp av `PrintWriter`. Och skriver även ut labyrinten och kortaste vägen på den valda labyrinten till konsolen för användaren att se.

2.6 JUnit-tester

Användandet utav JUnit-tester varit minimalt. Med filosofin för BlackBox-testning användes testerna enbart för att kontrollera om klassen fungerande som den skulle genom att skriva ut resultatet för antingen generatorn eller en lösnings-algoritm. Beroende på vad som skulle kontrolleras för tillfället.

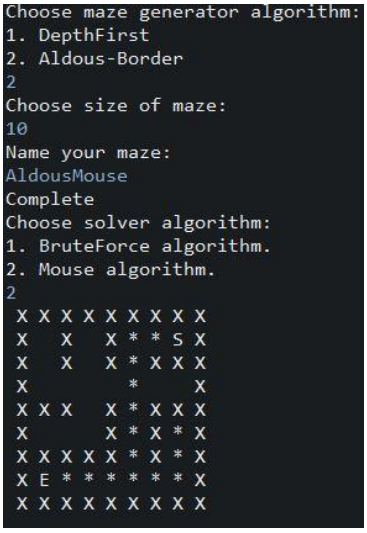
3 Resultat

3.1 Aldous – BruteForce



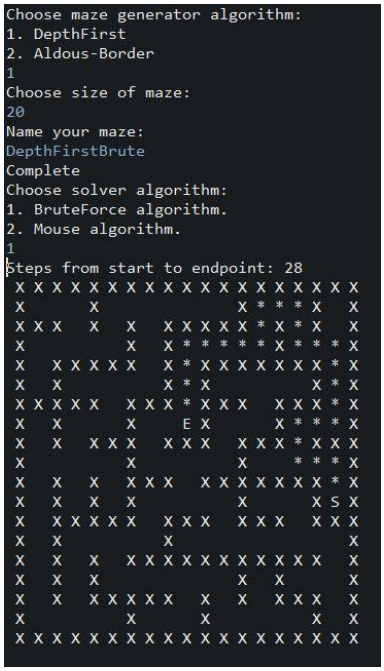
Figur 1 AldousBorder generator med en BruteForce lösare, storleken på labyrintent är 10x10

3.2 Aldous – MouseAlgorithm



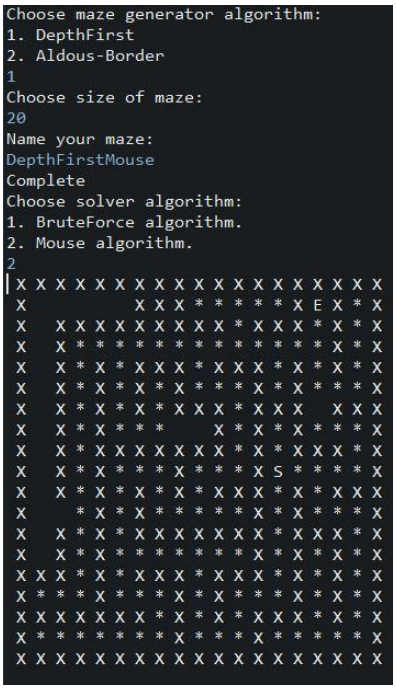
Figur 2 AldousBorder generator med en MouseAlgorithm lösare, storleken på labyrintent är 10x10

3.3 DepthFirst - BruteForce



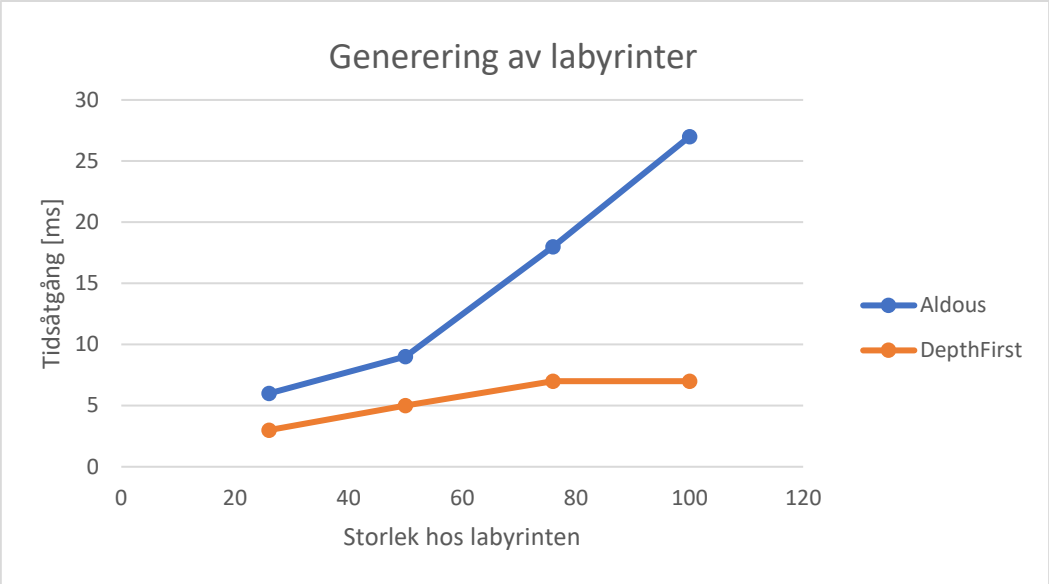
Figur 3 DepthFirst generator med en BruteForce lösare där labyrinten har en storlek om 20x20.

3.4 DepthFirst – MouseAlgorithm



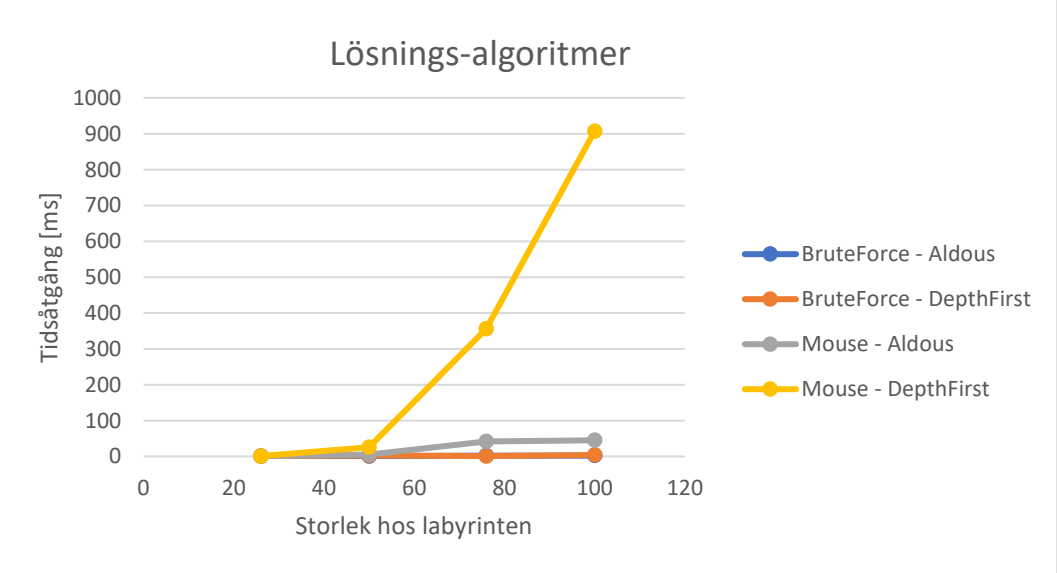
Figur 4 DepthFirst generator med en MouseAlgorithm lösare där labyrinten har en storlek om 20x20.

3.5 Tidsåtgången för generatorerna



Figur 5 Tidsåtgången hos respektive generatorer beroende på indatamängden

3.6 Tidsåtgången för Lösnings-algoritmerna



Figur 6 Tidsåtgången hos respektive lösnings-algoritm beroende på indatamängden

4 Diskussion

Valet av algoritm är viktigt om man ser till tidsåtgången, genom att skapa grafer kunde vi tydligt se att när en labyrint skapas så är Depth-First algoritmen betydligt snabbare än Aldous-Border.

Anledningen till att Depth-First är snabbare är för att den är en rekursiv algoritm och byggs upp som en trädstruktur. Aldous är långsammare eftersom vi går igenom en lista av obesökta noder, vilket gör att den får en högre ordonalitet.

När labyrinterna skall lösas såg vi att `MouseAlgorithm` är väldigt oberäknelig, detta är för att ”musen” kan gå på samma väg flera gånger innan ”rätt” riktning väljs. Vår implementation är kanske inte den bästa eftersom varje nod som besöks kan läggas i path-listan flera gånger, vilket resulterar i `OutOfMemory-Exception` om man har otur.

Brute Force är däremot väldigt effektiv eftersom det är en rekursiv funktion där varje närliggande cell besöks endast en gång tills samtliga noder är besökta. Vi valde att implementera en pekare till föregående nod för att olika vägar skall kunna jämföra distansen mellan start och slut, för att sedan välja kortaste vägen.

En implementation av den abstrakta klassen AbstractGenerator gjorde det enkelt att bygga grundlabyrinter för samtliga algoritmgeneratorer så att varje generator starta med samma förutsättning. Genom att sätta fasta ytterväggar och fylla hela labyrinten först för att sedan sätta varannan cell till möjliga vägar och övriga celler till potentiella rum. Eftersom den kolla varannan cell så spelar det ingen roll om det är en väg eller ett rum, den kommer att gräva en väg mellan start- och slut-mål ändå.

Vår DepthFirstSearch inspirerades av Migel Kano [1]. Vilket även gjorde att vi kunde förstå idén hur vi skulle bygga labyrinter från grunden och fick lättare att implementera övriga algoritmer.

Referenser

[1] ”migapro,” 23 11 2011. [Online]. Available: Depth-First Search, Maze Algorithm | Miguel Kano (migapro.com). [Använd 20 12 2020].