Kim Arvola Bjelkesten

Rapport

En rapport på en teknik för skapandet av en labyrint gjord av celler och vägar mellan celler.

Som början vill jag introducera ett exempel på hur en labyrint skulle kunna se ut, ta i akt att alla labyrinter presenterade i denna rapport kommer ha endast två vägar ut ur labyrinten. I bilden under så finns en stängd labyrint med storleken av 9 x 9 enligt vidden x höjden är mängden celler. Med stängd menas att labyrinten inte har en väg mellan sina två öppningar, markerade med ”1” och ”2”. Notera även väggarna mellan cellerna. Om det finns en väg mellan ”1” och ”2” så blir det en Simpel labyrint (se Illustration 2). Om alla celler kan nås från alla andra celler så är labyrinten en Komplett labyrint (se Illustration 3).

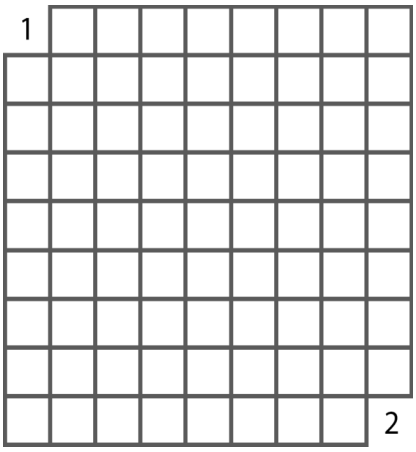

Illustration 1: Stängd.

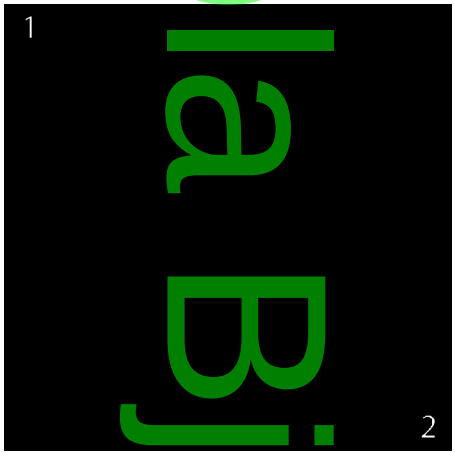

Illustration 2: Simpel.

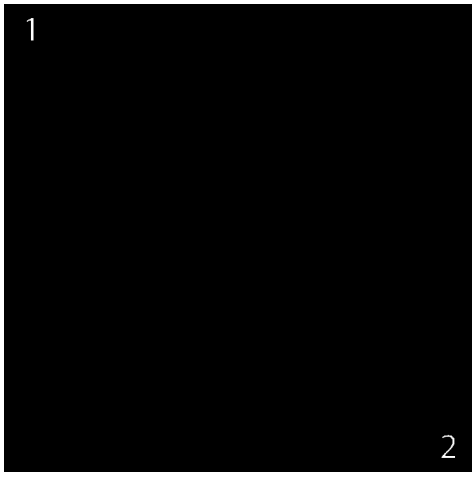

Illustration 3: Komplett.

Så det är grunden för en labyrint.

Ett sätt att representera en labyrint är att se det som en (i detta fallet oriktad) graf där noderna representeras av cellerna i labyrinten, i så fall skulle man kunna skapa en komplett labyrint genom att bygga det minsta uppspännande träd (Minimal Spanning Tree eller MST på engelska) för grafen.

För en labyrint som sedd ovan så är avståndet mellan noderna en konstant för direkt kontakt och -1 till alla andra.

Två av de vanligaste algoritmerna för att göra ett MST är Prim’s algoritm och Kruskal’s algoritm. Kruskal’s algoritm är grunden för den teknik som jag presenterar i denna rapport. Kruskal’s algoritm är en girig algoritm vilket betyder att den delar upp problemet i delar och löser fram den lokalt optimala lösningen. Kruskal’s algoritm består av att markera den minsta vägen mellan något par av noder så länge som de inte redan har en väg eller stig mellan sig. Jag kommer implementera noderna som disjunkta mängder vilket betyder att algoritmen hittar den kortaste vägen mellan två mängder och lägger ihop dessa i en delad delmängd (Union på mängderna) om de inte redan är i

Kim Arvola Bjelkesten

samma delmängd. Detta fortsätter tills det bara finns en enda delmängd kvar. Operationerna som används med disjunkta mängder är ”Union” vilket kopplar två noder genom att sätta en till roten för den andra. ”Find set” vilket tar reda på roten till noden operationen utförs på. ”Link” som utför ”Find set” på båda noderna och sedan gör ”Union” på resultatet av de två ”Find set” operatorerna. Anledningen att jag använder mig av disjunkta mängder är för att minimera den kostnad det tar att räkna ut om två noder redan kan nås via vägar genom andra noder. Disjunkta mängder räknar ut detta genom att kolla om de två noderna redan är i samma delmängd. Denna teknik om att skapa delmängder för alla noder och sedan länka dem tillsammans har dock en nackdel i att när du länkar samman delmängderna så bildar du ett träd. I och med att trädet får fler noder så växer trädet i höjd vilket ger tidskostnaden för att kolla delmängden. Så för att minska höjden på trädet, och därigenom kostnaden, använder jag mig av två tekniker. ”Union by rank” och ”Path compression”.

Den första tekniken ”Union by rank” gör att varje delmängd har en ”rank” vilket är höjden på delmängden och ändrar så att ”Union” kopplar den mindre delmängden in i den större, vilket inte leder till någon ändring av mängden med den högsta ranken. Om delmängderna har samma rank, exempelvis ”1” som ensam nod, så kopplas den första delmängden in i den andra delmängden vars rank därigenom ökas med ett. Detta leder till höjden blir till max lg(mänged av noder).

Bilden nedan visar hur tekniken bidrar till att hålla höjden nere.

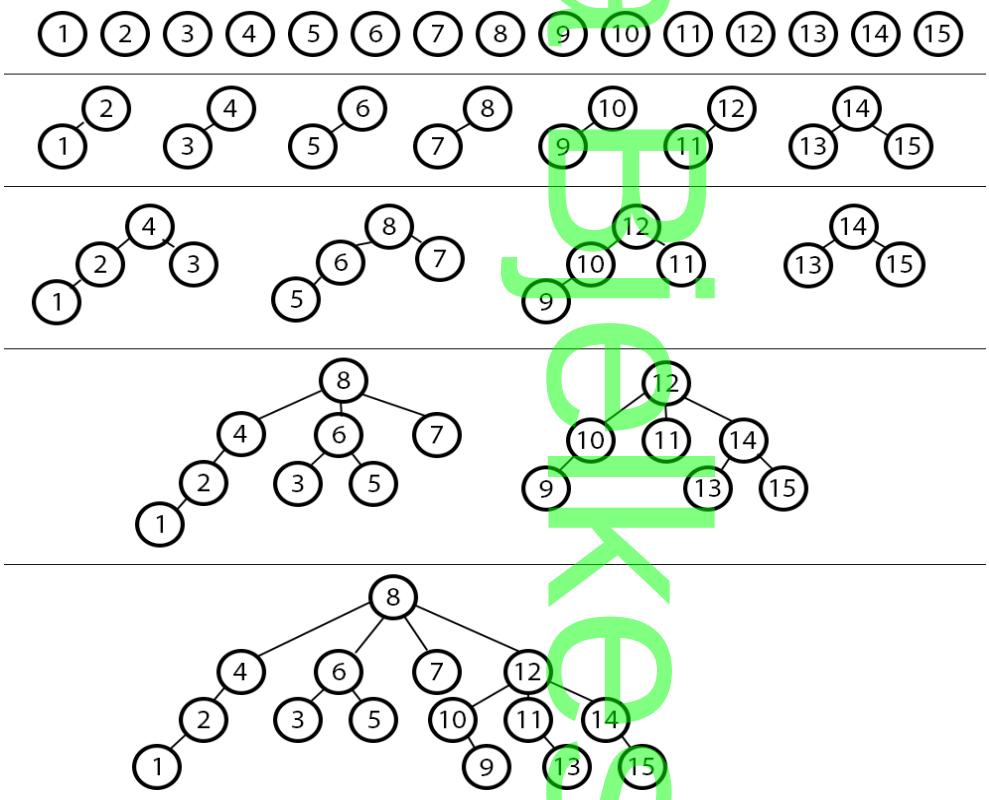

Illustration 4: "Join by rank"

”Path Compression” gör så att när ”Find set” operationen används på en nod så ändras föräldern till resultatet av ”Find set”, samma händer för alla noder mellan den och roten. Så om ”Path

Kim Arvola Bjelkesten

Compression” skulle användas på noden ”1” i den understa delen av bilden ovan så skulle både ”1” och ”2” ligga på samma nivå som ”4”, ”6”, ”7” och ”12”, med ”8” som förälder.

Tekniken jag tog fram bygger som sagt på Kruskal’s algoritm och använder sig av ”Join by Rank” och ”Path Compression” som optimering. Tekniken fungerar genom att definiera alla cellerna i labyrinten och sedan alla direkta vägar mellan cellerna; vilket i denna labyrint alltid är 1. Som grund för algoritmen så sparar jag dessa vägar sammanlagt med de associerade noderna i en lista. Algoritmen tar sedan ut en väg (och därigenom noderna som den går mellan) ur listan och kollar om noderna är del av samma delmängd. Om noderna inte var del av samma delmängd så utför jag operationen ”Link” på noderna. Om noderna var del av samma delmängd så sparar jag positionen för vägen så jag kan rendera en vägg mellan noderna/cellerna i labyrintens grafiska utritning. Detta fortsätter tills det enbart finns en delmängd kvar vilket betyder att algoritmen har skapat ett uppspännande träd som täcker varje nod/cell. Vilket betyder att det är en komplett labyrint. Så tekniken är väldigt simpel, och labyrintens egenskaper är beroende på hur man sorterar vägarna i listan. I specialfallet då labyrintens alla vägar är samma konstanta värde så finns det inte mycket att göra. Min implementation mixar vägarna slumpmässigt så att labyrinten själv blir slumpmässig. Men om vissa av vägarna är av annan längd eller har flera egenskaper så kunde listan av vägar sorteras efter dessa egenskaper, för en slumpmässig labyrint så applicerar man någon brus teknik för att göra små lokala ändringar i listan likt bilden nedan.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	1	3	6	4	5	8	7	9	12	10	11	13	16	15	14

Illustration 5: Lokala ändringar av lista inom 3 element.

Tester med tid i sekunder, 4 körningar för varje storlek.

Mängd Noder.	Skapandet av Noder, Vägar och i sekunder.	Skapandet av labyrintens ”Spanning Tree” i sekunder.
10 * 10	0.19 s, 0.279 s, 0.229 s, 0.253 s	0.00128 s, 0.00125 s, 0.00052 s, 0.0005 s
20 * 20	0.198 s, 0.24 s, 0.29 s, 0.26 s	0.0022 s, 0.002 s, 0.003 s, 0.0023 s
40 * 40	0.2 s, 0.27 s, 0.29 s, 0.25 s	0.012 s, 0.0095 s, 0.0095 s, 0.0105 s
100 * 100	0.29 s, 0.24 s, 0.23 s, 0.24 s	0.068 s, 0.069 s, 0.07 s, 0.077 s
200 * 200	0.3 s, 0.33 s, 0.34 s, 0.376 s	0.29 s, 0.29 s, 0.29 s, 0.28 s
400 * 400	0.56 s, 0.56 s, 0.56 s, 0.56 s	1.15 s, 1.14 s, 1.13 s, 1.16 s
800 * 800	1.6 s, 1.67 s, 1.7 s, 1.72 s	4.8 s, 4.8 s, 4.96 s, 4.8 s
1200 * 1200	3.4 s, 3.5 s, 3.6 s, 3.6 s	11.5 s, 10.9 s, 10.8 s, 11.0 s
2000 * 2000	9.1 s, 10 s, 9.57 s, 9.38 s	31.3 s, 33.9 s, 31.5 s, 31.3 s
3000 * 3000	21 s, 21 s, 21 s, 21 s	73.4 s, 72.9 s, 73.6 s, 72 s

Kim Arvola Bjelkesten

Kim Arvola Bjelkesten

Som slutord skulle jag vela säga att denna teknik inte är optimal för en helt randomiserad labyrint, tekniken används med fördel om användaren vill skapa en labyrint där avstånden mellan cellerna i labyrinten inte är av samma konstant.

Kim Arvola Bjelkesten