Mittuniversitetet
MID SWEDEN UNIVERSITY

Projekt - Labyrinter

Objektbaserad programmering i C++

Jimmy Åhlander*

6 maj 2021

Innehåll

*jimmy.ahlander@miun.se. Avdelningen för informationssystem och -teknologi (IST).
Baserat på projektet *Labyrintlaborerande* av Martin Kjellqvist.

1 Introduktion

De flesta har nog någon gång stött på en labyrint, om inte i den klassiska skepnaden av gröna häckar så kanske istället som en utmaning i en knep och knåp-bok. Labyrinter är intressanta ur ett datavetenskapligt perspektiv då de kan genereras (och lösas) med hjälp av ett flertal olika algoritmer, ofta med varierade slutresultat. Samtidigt kan labyrinter också representeras med datastrukturen träd, förutsatt att det inte finns några loopar, vilket öppnar upp flera möjligheter. I det här projektet kommer du implementera algoritmen *depth-first search* (DFS) för att generera och potentiellt även lösa tvådimensionella labyrinter.

2 Mål

Efter genomfört projekt ska du *självständigt* kunna lösa datatekniska problem genom att implementera algoritmer med hjälp av klasser, structar, datastrukturer och tillhörande metoder.

Följande lärandemål är kopplade till projektet. Du ska kunna:

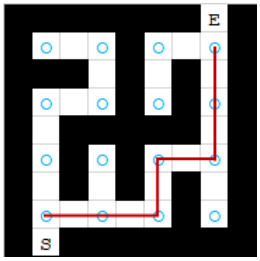
- specificera och implementera egna klasser.
- använda objekt av andra typer som datamedlemmar i en klass.

3 Teori

Labyrinter

En labyrint definieras inom ramen för detta projekt som en serie anslutningar mellan ett antal noder som saknar loopar. Vidare har en labyrint också en startpunkt och en slutpunkt längs med ytterväggarna. Avsaknaden av loopar innebär att det alltid kommer finnas en optimal väg från startpunkten till slutpunkten. Att *lösa en labyrint* syftar till att finna denna väg.

Det är enklast att visualisera labyrinten på ett tvådimensionellt plan där ett antal noder anslutits till varandra, se Figur 1.



Figur 1: En 9x9-labyrint där `S` är start och `E` end. De blåa cirklarna representerar labyrintens noder. Den röda linjen visar den optimala vägen från start till slut.

Det är generellt tacksamt om labyrinten har udda dimensioner som den i Figur 1. En 9x9-labyrint får då 16 noder. Det hjälper oss att undvika knöliga situationer där vi får för breda väggar eller isolerade noder som inte går att ansluta.

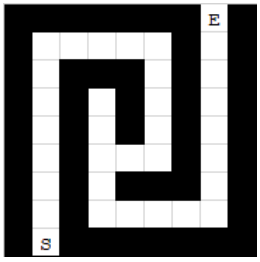
Varje nod kan ses som en struct med fyra anslutningar – en i vardera kardinal riktning. En fri passage representeras av en öppnad anslutning mellan två noder och en vägg representeras av en stängd anslutning mellan två noder. Utöver det tillkommer väggar vid alla positioner diagonalt mot noderna och mot ytterkanterna – de väggarna kommer alltid existera oavsett labyrintens skepnad. Start- och slutpunkterna ersätter en yttervägg vardera och måste angränsa någon nod.

Labyrintgenerering

Det finns många olika algoritmer för att generera labyrinter. I detta projekt kommer du tillämpa *depth-first search* (DFS) för ändamålet.

DFS nyttjar att vi kan se en labyrint som en trädstruktur där vi börjar på en rotnod som förgrenar sig utåt tills att vi når de yttersta noderna som kallas för löv. I grunden är nämligen DFS en algoritm som används just för att söka i träd. Krydda den med lite slump, låt den genomsöka *alla* noder och vipps så fungerar den alldeles utmärkt för att generera labyrinter också. Lite som en ambitiös men fullständigt mållös spårhund. Intressant nog kan vi även använda DFS för att lösa labyrinter. Att lösa en labyrint är samma sak som att söka efter en specifik nod i ett träd med tillägget att vi även sparar vägen vi går för att nå den noden.

DFS är alltså flexibel och förhållandevis simpel. Den har dock en för människor bländade nackdel, se Figur 2.



Figur 2: En labyrint genererad med DFS. Gradera svårigheten på en skala 1-10.

Labyrinterna blir inte särskilt svåra. Anledningen till det är att DFS bara förgrenar sig när den hamnar i en återvändsgränd. Detta beteende tenderar att skapa långa gångar med få förgreningar samtidigt som förgreningarna ofta når snabba och snöpliga slut. Knappast en brain buster.

I just vårt exempel hamnar DFS i en enda återvändsgränd precis efter att den svängt uppåt för första gången. Den backar då tillbaks samma väg som den kom

tills att den hittar en nod med en obesökt granne. Detta arbetssätt kallas för *backtracking*. Därefter är det en stillsam resa till mål.

DFS kan implementeras iterativt eller rekursivt och använder vanligtvis en stack för att spåra vägen den gått i ämne att kunna backtracka vid behov. En beskrivning för en iterativ version av algoritmen följer nedan:

```
- Markera alla noder som obesökta.
- Välj en startnod och låt den vara nuvarande nod N.
- Markera N som besökt.
- Lägg till N till en stack S.
- Så länge som S inte är tom:
  - Hämta en nod från S och låt den vara N.
  - Om N har obesökta grannar:
    - Lägg till N till S.
    - Välj slumpmässigt en obesökt granne G.
    - Öppna kopplingen mellan N och G.
    - Markera G som besökt.
    - Lägg till G till S.
```

Labyrintlösning

Att lösa en labyrint innebär att hitta en väg från startpunkten till slutpunkten. Även för detta ändamål kan vi använda DFS. Den stora skillnaden mot tidigare är att vi nu stannar när vi nått noden vid slutpunkten. Stacken kommer då innehålla vägen från start till mål vilket du får ut i omvänd ordning om du tömmer stacken. Se följande:

```
- Markera alla noder som obesökta.
- Välj en startnod och lägg den till en stack S.
- Så länge som S inte är tom:
  - Hämta en nod från S och låt den vara N.
  - Markera N som besökt.
  - Om N är slutnoden:
    - Lägg till N till S. Du är klar.
  - Om N har obesökta grannar:
    - Lägg till N till S.
    - Lägg till alla obesökta grannar av N till S.
- Om S är tom så finns ingen lösning.
```

4 Uppgift

Din uppgift är att självständigt implementera en labyrintgenerator som tillämpar DFS. Beroende på vilket betyg du siktar på ställs olika krav på din lösning, se nedan. Kraven är transitiva, undantaget de krav som ersätts av högre krav.

Betyg E

För betyg E ska din lösning inkludera:

1. Labyrinter[1] av storleken 11x11 genererade med DFS. Start- och slutpunkterna är fasta.
2. En struct för labyrintens noder.
3. En klass för labyrinten. Klassen ska inkludera en tvådimensionell container för att hålla alla noder. Klassen ska också ha metoder för att:
   - Generera en ny labyrint med DFS.
   - Rita labyrinten grafiskt på skärmen (med text i konsolen).
4. Ett gränssnitt för att interagera med användaren. Användaren ska kunna generera och visa nya labyrinter. Programmet får inte stängas av förrän användaren väljer att stänga av. Programmet får inte heller under några omständigheter krascha, inklusive vid felaktig indata från användaren. Rimliga felmeddelanden och god användbarhet krävs.
5. Structar, klasser, funktioner och metoder som följer designprincipen *separation of concerns* (SoC) i så stor utsträckning som möjligt.
6. Kommenterad kod, till en rimlig nivå. Förklara varför, inte vad.
7. En header i varje fil med åtminstone ditt namn, aktuellt datum och betyget du har för avsikt att nå.

Betyg C

För betyg C ska din lösning också inkludera:

8. En flexibel storlek för labyrinten, *inte nödvändigtvis kvadratisk*, som kan anges av användaren, men med ett defaultvärde 11x11 om användaren inte väljer något annat.[2]
9. Slumpade start- och slutpunkter, positionerade enligt tidigare angivna regler.
10. Metod(er) för att lösa en labyrint samt visualisera vägen grafiskt.

[1] Se definitionen i teorin. En labyrint får inte ha några loopar, isolerade noder eller dubbelbreda, ojämna väggar.

[2] Du bör endast acceptera udda dimensioner för att inte bryta tidigare krav.

Betyg A

För betyg A ska din lösning också inkludera:

11. En metod för att generera labyrinter med *breadth-first search* (BFS).[3] Användaren ska nu kunna välja vilken metod som används för att generera en labyrint.

12. Ett CLI. Användaren ska kunna mata in textfiler på stdin med färdiga labyrinter som du löser. Programmet anropas då genom:

    ```
    ./maze < maze.txt
    ```

    När programmet startas på detta sätt ska det omedelbart lösa den inmatade labyrinten, visa lösningen på skärmen och därefter stänga av programmet utan paus. Ingen interaktion med användaren sker. Om du löst detta på rätt sätt ska även följande vara möjligt:

    ```
    ./maze < maze.txt > solution.txt
    ```

    I textfilen tolkar du whitespaces som öppningar, undantaget radbrytningar, och non-whitespaces som väggar, undantaget `S` som är startpunkten och `E` som är slutpunkten. Dina utskrifter behöver inte använda samma tecken som den inlästa labyrinten. Kom ihåg felhantering och felmeddelanden i alla dess former. Ta inte för givet att textfilen är välstrukturerad eller ens lösbar.

    Startas programmet utan inmatad textfil så ska det förstås fortfarande fungera precis som vanligt.

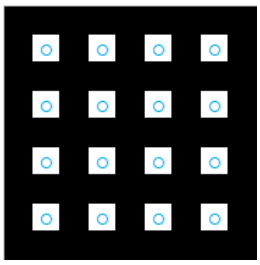
5 Genomförande

Du bör börja med att läsa på hur DFS fungerar. Detta kommer att avsevärt förenkla implementeringen då du mer naturligt kommer förstå vilka datastrukturer du behöver och vilka andra programmeringskonstrukter som kan komma till nytta. Ett tips är att rita en liten labyrintmall med endast 9 noder, ungefär som Figur 3 fast mindre. Följ därefter beskrivningen av DFS i Teori **steg för steg** och observera hur labyrinten utbreder sig. Det spelar ingen roll *hur* du representerar att en nod är besökt eller obesökt, bara du själv håller koll på något sätt. Indexera noderna 1, 2, 3, och så vidare för att enklare håll koll på vem som är vem i stacken, vem som är N och G, och så vidare. Om du upplever problem att följa algoritmen är det troligen för att du inte följer algoritmen stegvis, för att du försöker hålla allt i huvudet istället för att rita, eller för att du inte förstår hur exempelvis en stack fungerar. Läs då på om dessa koncept.

[3] BFS har inte beskrivits i instruktionerna utan du måste instudera detta själva. Skillnaden gentemot DFS är liten och du rekommenderas läsa på om detta redan vid projektets start för att undvika koduppprepning.

När du känner dig bekväm med algoritmen kan du gå vidare till att sätta upp dina klasser och structar. Du kommer behöva en struct för noderna med ett flertal attribut. Bland annat måste noderna på något sätt lagra vilka grannar som de anslutits med. Du kommer också behöva en klass för labyrinten som håller alla noderna i någon tvådimensionell datastruktur[4]. Labyrintklassen kommer också behöva ett antal metoder för att bland annat konstruera och generera en labyrint samt för att skriva ut labyrinten.

För att generera en labyrint börjar du enklast med en mall där alla kopplingar är stängda, se Figur 3. Därefter väljer du din startnod och tillämpar DFS för att generera labyrinten.



Figur 3: En mall som utgångspunkt för en labyrint. Start- och slutpunkterna kan vara förutbestämda redan nu eller bestämmas senare, beroende på implementation.

Från beskrivningen av DFS i Teori är det inte självklart hur du ska representera vilka noder som besökts. Du skulle kunna lösa detta med en separat lista eller kanske mer lämpligt som ett attribut i noden.

Se upp för att blanda ihop *representation* med *visualisering* av labyrinten. Representation är hur du lagrar labyrinten i programmet. Visualiseringen är istället hur labyrinten ser ut när den visas på skärmen, effektivt programmets grafik. Du kan välja vilka tecken du vill för att visualisera väggar respektive öppningar. Det är förstås möjligt att lagra alla dessa tecken som används i visualiseringen direkt i en tvådimensionell char-array och att strunta i noderna som koncept men du blandar då ihop visualiseringen med den underliggande representationen, vilket bryter mot SoC och ska undvikas. Fundera alltså bara på hur labyrinten ska se ut när du faktiskt skriver ut den och inte i någon annan funktion. Kika gärna på följande video som illustrerar skillnaden: `http://apachepersonal.miun.se/~jimahl/DT019G/anim/maze_gen.mp4`.

Vikta slumpen och var inte rädd för mindre tillägg och förändringar för att krydda labyrintgenereringen. Försök att göra labyrinten svårare!

Till skillnad från i laborationerna har du inte erhållit något interface för detta

[4]Som Lippman säger: *"use vector unless there is a good reason to prefer another container"*. [1, s. 327]

projekt. Ett mål med projektet är att du nu självständigt ska kunna konstruera ett rimligt interface för din klass som du senare implementerar.

FAQ

Q: *Bör jag spara nodernas position/koordinater i nodstructen eller labyrintklassen?*

> A: Vad anser du är rimligast? Att elementet vet vilken position den har i en behållare eller att behållaren vet vilka positioner dess element har?

Q: *Vilka tecken får användas för att visualisera väggarna, med mera?*

> A: Startpunkten ska vara `S` och slutpunkten `E`, i övrigt är det ditt val. Textfilerna som läses in (krav för betyg A) kan komma att använda olika tecken.

Q: *Kan jag se en 11x11-labyrint som bestående av 121 noder istället för 25?*

> A: Ja. Det är inte en ovanlig ansats, se t.ex. Computerphile: https://www.youtube.com/watch?v=rop0W4QDOUI. Det är ett tacksamt synsätt när labyrinter löses och samtidigt det mest uppenbara sättet att lagra dem. Det uppenbara är dock inte alltid bäst. Det kan avsevärt komplicera labyrintgenereringen då alla noder inte längre kan behandlas lika. Du uppmanas därför i första hand att inte arbeta så.

6 Examination

Projektet redovisas i första hand muntligt inför examinator vid ett projektredovisningstillfälle. Under redovisningen presenterar du vilket betyg du siktat på. Du visar därefter funktionaliteten utifrån de krav som ställts för det betyget. Efter det visar du din kod, förklarar hur du löst uppgiften och besvarar frågor relaterade till såväl koden som projektet i helhet.

Efter redovisningen lämnar du in projektet i en **zip-fil** bestående av ett antal headerfiler och cpp-filer. Kom ihåg att varje fil ska innehålla en header med åtminstone ditt namn, aktuellt datum och betyget du har för avsikt att nå.

Uppgiften ska genomföras självständigt. Du förväntas ange dina källor om du nyttjar invecklade kodsnuttar som du hittat online. Grundregeln är att om du vet vad något gör och varför så behövs ingen källa, om du vet vad men inte varför så ska du källhänvisa, och om du vet ingetdera så ska det inte med alls. Du får dock aldrig kopiera hela program eller omfattande funktioner. För

information om konsekvenserna för vilseledande vid examination, till exempel otillåtet samarbete, se kurswebbplatsen och inlämningslådorna.

6.1 Bedömning och återkoppling

Uppgiften bedöms med betygen `A-F`, `Fx`.

Betyget grundar sig i vilken ansats du haft för projektet. Krav finns beskrivna för betygen A, C och E. Utöver de uttryckta kraven för koden förväntas du även kunna presentera din kod muntligt och visa förståelse för hur du gått tillväga för att lösa uppgiften. Vid mindre anmärkningar kan du erhålla ett lägre betyg än det du siktat på. Vid större eller flera mindre anmärkningar kan du erhålla Fx. Du har då möjlighet att inom en vecka skriftligt komplettera uppgiften för att nå ett godkänt betyg, i normalfallet inte högre än E. Beroende på anmärkningarnas natur och omfattning kan du också erhålla F. Om du exempelvis inte kan förklara hur du löst uppgiften erhåller du alltid F. Du måste då redovisa projektet igen vid nästa tillfälle och lämna in på nytt.

Viss återkoppling erhåller du muntligt under redovisningen, **dock inte betyget**. Efter inlämning erhåller du skriftlig återkoppling, i normalfallet endast i form av betyg och en kort motivering.

Referenser

[1] S. B. Lippman, *C++ primer*, 5th ed. Upper Saddle River, N.J.: Addison-Wesley, 2013.