Dijkstra's Algoritme

Johannes Jørgensen
S2o

2021 Febuar

Contents

1 Introduktion **3**

2 Teori **3**

2.1 Hvad er rekursion? 3
2.2 Rekursion i Matematik 3
2.3 Programmering 3
2.3.1 Rekursion i programmering 3
2.3.2 Variabler 4
2.3.3 Objekter 4
2.3.4 Arrays 4
2.3.5 Løkker 4
2.3.6 Betinget udførelse (if statement) 5
2.3.7 Dijkstra's algoritme 5
2.3.8 Dijkstra's Algorithm Pseudocode 6

1 Introduktion

Dijkstra's Algoritme er en kendt algoritmen indenfor pathfinding. Formål er at finde den korteste vej fra punkt a til punkt b. Dette er en større del af almindelige menneskers liv end man tænker. Hvad er den hurtigste vej til skole eller arbejde? Er det via motorvejen som måske har vejarbejde? Vil det så være hurtigere at tag vejen igennem byen? Disse spørgsmål kan man ofte få hurtigt svar på med ens GPS, som har implementeret diverse pathfinding algoritmer og er tilkoblet internettet med de seneste nyheder om vejarbejde, kø osv.
Jeg vil med brug af Dijkstra's Algoritme finde sammenhængen mellem den matematiske del af rekursion som og et rekursion-kald i programmering.

2 Teori

2.1 Hvad er rekursion?

Ordet rekursion er en betegnelse for noget som refererer til sig selv. Når noget refererer sig selv betyder det at noget behøver information eller data fra sit forrige jeg. Rekursion er oftest betegnet som en rekursionligning i matematik og en som en funktion der kalder sig selv i programmering.

2.2 Rekursion i Matematik

2.3 Programmering

2.3.1 Rekursion i programmering

Rekursion i programmering er en funktion som har et rekursivts kald, alså funktion kalder sig selv. Dette kald kan være meget direkte som i eksempel 1 (Simpel rekursions funktion), eller indirekte hvor det ikke er lige så overskuligt at se et rekursivt kald.

```
1 function fractorial(n) {
2   if (n == 1) return 1;
3   else return n*fractorial(n-1)
4 } // if n = 4, then exprected output: 24
```

Listing 1: Simpel rekursions funktion

Funktionen i eksempel 1 indtag en værdi n som parameter. Hvis denne parameteren n er 1 stopper det rekursive kald (se linje 2.). Dette betyder også at dette eksempel har et meget direkte grænse hvor n ikke kan komme under 1. Linje 3 beskriver så hvordan n skal reagere hvis dens værdi ikke er 1. Hvis n ikke er 1, men eksempelvis 3 skal den gange nuværende n med forrige n. I forrige n af 3 er 2, derfor skal 2 ganges med dens forrige n og således.

$$n(1) = 1 \Leftrightarrow 1$$

$$n(2) = 2 * 1 \Leftrightarrow 2$$

$$n(3) = 3 * 2 \Leftrightarrow 6$$

$$n(4) = 4 * 6 \Leftrightarrow \mathbf{24}$$

2.3.2 Variabler

Et variabel er en pladsholder for et stykke data. Typisk set bruger man variabler i brug når man skal gemme et stykke data som man skal bruge forskellige steder i programmet. Disse variabler kan ændres under programmets køretid afhængig i hvordan variables data bliver manipuleret. Variablers data kan eventuelt ændres under programmets køretid, hvis man ønsker. Lidt mere teknisk indeholder et variable særlige set af bits eller af variablenes datatype. Et variable har en datatype som identificere hvilke slags data variabelt kan indeholde. Diverse programmeringssprog bruger dynamiske datatyper, hvor variabler kan indeholde forskellige datatyper. Her et en tabel med forskellige datatyper.

Datatype	Skrivemåder	Eksempel på data	Kommentar
Integer	int	...-3, -2, -1, 0, 1, 2,... 1000	Kun heltal (både negativ og positiv)
Floating Point	float	..., -3.25, -2.11,... 100,12	Decimaltal (både negativ og positiv)
String	str eller text	"hello world", "This is a string"	Et række af karakterer oftest tekst.
Array	[data1,data2,...]	int val = [-2,-1,0,1,5]	Indeholder en række data under et navn
Character	char	@, s, ¤, k...	Kun et symbol
Boolean	bool	True (1) eller False (0)	Enten sandt eller falsk
Constant	const	"Hello", -3, 22, [1, 2, 3]	Dynamisk datatype, men konstant værdi

2.3.3 Objekter

2.3.4 Arrays

Et array er en datastruktur, der indeholder en gruppe af elementer. Disse elementer er typisk alle af samme datatype, såsom et integer eller en string. Arrays bruges typisk set i computerprogrammer til at organisere data, så et relateret sæt værdier nemt kan sorteres eller søges.

```
// Array with constant values of datatype String
const cars = ["Audi", "Volvo", "BMW"];
cars[0] = "Audi" // Index 0 of array
```

Listing 2: Eksempel på et array

2.3.5 Løkker

En løkke er et stykke kode som vil fortsætte med at køre indtil en given betingelse er opfyldt. der er to slags løkker, "while" - løkker og "for"-løkker. "While" løkker kører ind til betingelsen i parentesen bliver opfyldt. Mens "for" løkker lavet selv et variabel, som i det simpleste tilfælde vil tælle op eller ned fra og køre det antal gange ifølge betingelse. Løkker er meget brugbare siden det stopper en fra at skulle skrive mange linjer af den samme kode hvis man skal køre noget flere gange.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
} // exprected output: 0,1,2,3,4
```

Listing 3: Eksempel på et for løkke

2.3.6 Betinget udførelse (if statement)

Betinget udførelse eller "if statement", er et stykke kode som checker om en betingelse opfyldt nogle specifikke kriterier. Hvis de betingelser bliver opfyldt vil koden inde i dette if statement blive kørt. Med et if statement kan man give den flere kriterier ved at lave en "else if" eller "else" statement efter den første stykke kode. "else if" er et til if statement med en betingelse før den bliver kørt, samtidig med at dette stykke kode kun bliver kørt hvis det første if statement ikke opfylder sin betingelse. Et "else" statement er bare et stykke kode som vil blive kørt hvis koden ikke opfyldte sin betingelse.

```
let val = 5;

if(val == 1){
  console.log("The value is now 1!");
}else console.log("The value is not 1"); // exprected execution of statement
```

Listing 4: Eksempel på betinget udførelse

2.3.7 Dijkstra's algoritme

Dijkstra's algoritme består af Breadth-First Search Algorithm (BFS) med et lille twist. BFS er designet til at søge igennem et data tree eller en graf struktur. BFS algoritmen søger først alle "neighbors" til et givne "node" før den søge i naboernes "children". Twistet i Dijkstra's algoritme er dog Dijkstra gøre brug af en prioriterings kø af "nodes", for at prioritere hvilke naboer Dijkstra's skal søge igennem først.
"Node" indenfor programmering er datapunkter i et data tree eller en anden form for datastruktur. "Neighbors" eller naboer er nodes som har direkte kommunikation for den første node. "Children" er så tredje leds nodes til første node.

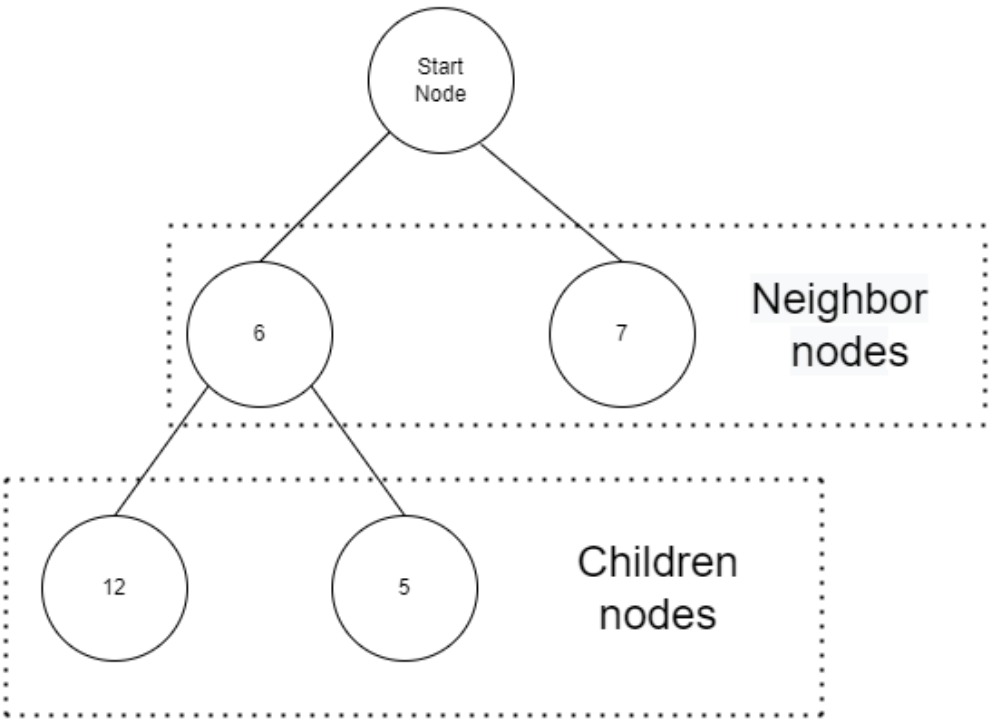

Figure 1: Illustration af et datatree med Nodes

Dijkstra's algoritme er designet til at finde den korteste vej mellem 2 notes. Algorithmen søger og finder diverse veje med den korteste vej og beregner den korteste vej gennem hver node som ikke er søgt igennem. Derefter opdatere den nabo nodernes med den korteste vej imens den holder øje med hvilke nodes algoritmen allerede har søgt igennem. Vejen til hver node har en "vægt". En vægt betyder sværhedsgraden for at kommer til valgte node. Et praktisk eksempel på dette kunne være transport. En højere vægt værdi kan være på grund af eventuelt vejarbejde, kø osv. Figur 2 har også lignende vægte bare data raleterende.
Eksemplet her viser en datatree med følgende nodes A, B, C, D, F, G. Algoritmen skal finde den korteste vej fra node $A \rightarrow G$. Algorithmen vil først søge igennem nabo nodes til vores start node A. Vejen fra node $A \rightarrow B$ koster 2 og vejen fra $A \rightarrow C$ koster 4. Derfor indtil videre er den korteste vej B. Så søger algorithmen fra node $B \rightarrow C$ og D, vejen til C koster nu kun 3 ud fra forrige $A \rightarrow C$. Så søges vejen fra $C \rightarrow F$, og den er kortere end $B \rightarrow D$ som kostet 5. Til sidst søges den korteste vej fra $F \rightarrow G$ og $D \rightarrow G$, som vejen $F \rightarrow G$ er stadig den korteste vej. Dermed bliver den korteste vej $A \rightarrow B \rightarrow C \rightarrow F \rightarrow G$ med total vægtkost på 9.

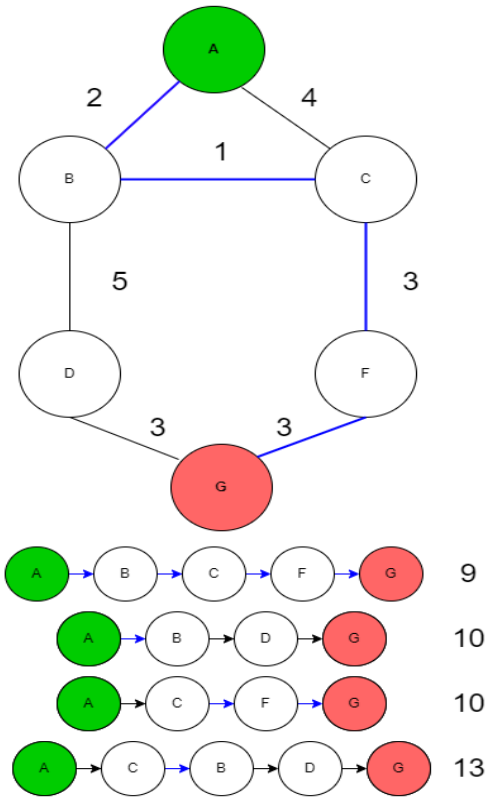

Figure 2: Eksempel på gennemgang af datatree af Dijkstra's Algorithm

2.3.8 Dijkstra's Algorithm Pseudocode

```
function Dijkstra(G, S){ // G = Graf, S = Source
  for hver V i Graf // V = vej
    afstand[V] = infinity // set alle veje til infinity
    forrige[V] = null
    hvis V != Start
      V += Q // Q = prioriterings queue
  afstand[S] // afstand fra node til node

  imens Q er tom
      U // U = min value fra Q
      for hver nabo V af U som stadig i Q
        midlertidigAfstand  // afstabd[U] + G.Sider[U, V]
        hvis midlertidigAfstand < afstand[V]
          afstand[V] // midlertidigAfstand
          forrige[V] // U

  return afstand[], forrige[]
}
```

Listing 5: Dijkstra's Algorithm Pseudocode