Dijkstra's Algoritme

Johannes Jørgensen
S2o

2021 Febuar

Contents

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 i denne case med brug af Dijkstra's Algoritm finde sammenhængen mellem den matematiske og programmerings del af rekursion. Rekursion er en stor del af Pathfinding algorithms. Dijkstra's Algoritme kan hjælpe os til at finde den korteste vej fra punkt a til punkt b.

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

2.2.1 rekursionligning

Når der snakkes om rekursion i matematik, snakkes der oftest om en rekursion ligning. En rekursion ligning opfattes som en regel der beskriver, hvordan et tal eller element i en talfølge skal beregnes. Hvis vi ser på en simpel talfølge som: 0,1,2,3,4,5.... Denne talfølge har det næste element i talfølgen altid 1 større end den forrige. Denne talfølge kan dermed beregnes generelt med en rekursion ligning.

$$x_n = x_{n-1} + 1$$

Rekursion ligning beregner et vilkårligt element i talfølgen. Dette gøres ved at finde forrige værdi af n med x_{n-1} og derefter gøre x en større. Dette eksempelt var meget simpelt da hvad end man vælger n til at være blevet x det samme. Et lidt mere kompliceret eksempel af en rekursion ligning er:

$$x_n = 2x_{n-1}$$

Hvis vi sætter en begyndelsesbetingelse på x_0. Med en begyndelsesbetingelse kan der kun komme en talfølge som løsning, ud fra eksemplet er talfølgen:

n	x_n
0	1
1	2
2	4
3	8
4	16

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

Et objekt er en abstrakt enhed med specifikke karakteriseringer og opførelse. Et objekt kan være et variable, datastruktur, en funktion eller en kombination alle dem alle. Et praktiskt eksempl på dette kan være en bil som objekt. En bil har specifikke karakteriseringer såsom hjul, farve, producent osv. En bil har også opførsel som at starte motoren, accelerere bilen m.m.
Hvis vi tager samme praktiske eksempel på en bil som objekt, kan vi overføre det til kode med en "Klasse" som er en form for plan over alle klassens objekter

```
  class Car { // Blueprint a car
   int model;
   int year;
   string color;

    void drive() { // objekt
      // Get the car to drive
    }

    void reverse() { // objekt
      // Get the car to reverse
    }
}
```

Listing 2: Eksempel på et objekt

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 computer-programmer 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 3: 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 4: 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 5: 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.

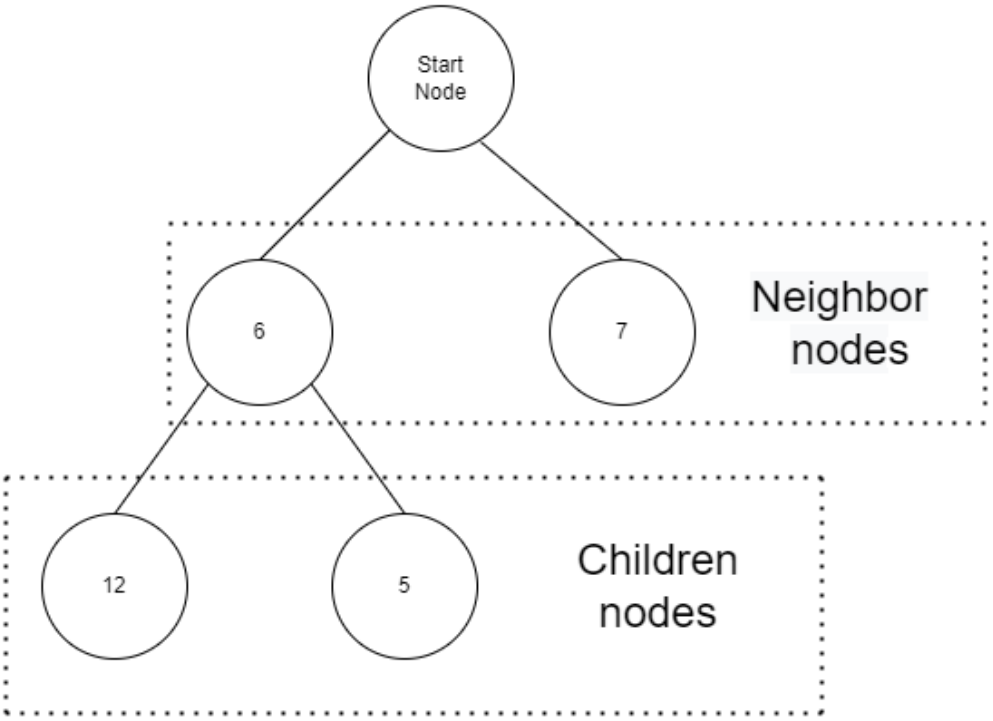

Figure 1: Illustration af et datatree med Nodes

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.

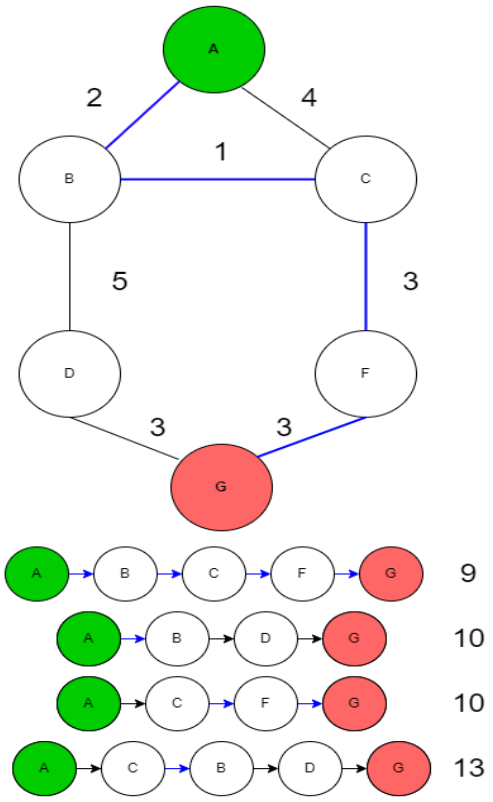

Figure 2: Eksempel på gennemgang af datatree af Dijkstra's Algorithm

2.3.8 Dijkstra's Algorithm Pseudocode

Nu når vi ved hvordan Dijkstra's Algorithm virker skal vi prøve så oversætte det til kode. Den bedste måde at generelt oversætte algoritmen kan vi lave en Pseudokode. En Pseudokode er en måde at skrive en algoritme eller en processen på en general programmerings måde uden at implementere direkte kode. Dette kan være hjælpe andre med generelt at forstå en algoritme kodemæssigt.

```
function Dijkstra(G, S); // G = Graf, S = Source (evt. startpunkt)
  for hver V i G // V = vej
    afstand[V] = infinity // set alle veje til infinity
    forrige[V] = null
    hvis V != S
      V += Q // Q = prioriterings queue (gem)
  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 6: Dijkstra's Algorithm Pseudocode oversat til dansk

3 Metode og Analyse

At bruge Dijkstra's algoritme til at finde den korteste var fra a til b, er på papir en effektiv måde at gøre det på. Men er det dog også sådan i praksis? Hvilke scenarier er Dijkstra's algoritme god imod andre pathfinding algoritmer såsom A* search algorithmen?

3.1 Dijkstra mod andre algoritmer

Dijkstra er en af de mest kendte algoritmer indenfor pathfinding. Der er andre pathfinding algorithms som bygger videre på Dijkstra's algoritme, som er bedre til at løse andre problemer end Dijkstra. En af efterkommerne af Dijkstra's algorithm er A* (udtalt A star). A* er en generelt mere effektiv algoritme i forhold til Dijkstra's, dette er fordi A* hænger tungt på "heuristic". Heuristic (eller heuristik på dansk) inde for programmering, er en designet teknik til at løse et problem hurtigere. Man kan se det som en form for systematisk søgning efter information eller afprøvning af de bedste muligheder.[1] Det der også gøre A* algoritmen hurtigere og bedre i de fleste tilfælde er at den er en informeret algoritme. Dette betyder at A* har mulighed for lidt mere infomation på grund af dens brug af Heuristics. Dijkstra's har ikke denne fordel, som A* har [2]. Dijkstra's er en form for et "brute force" på et problem, da den bare søger løs for slut noden. Ordet "brute force" betyder at algorithmen løser et problem den mest simple måde, dette er ofte mindre elegant og en langsom måde at løse problemer på. Dette eksempel viser et eksempel på Dijkstra's algoritme og A* search og hvordan de hver især løser en labyrint. [3]

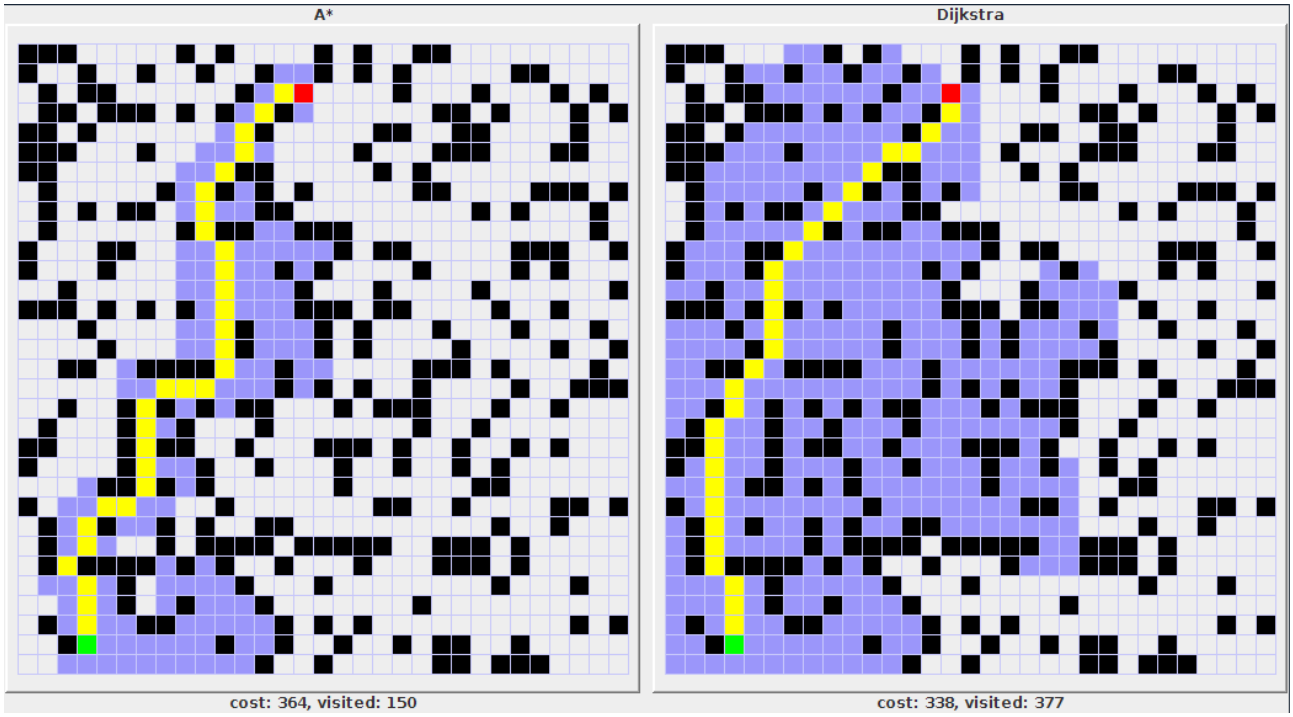

Figure 3: A* search vs. Dijkstra algoritme

Figur 3 viser et godt eksempel på hvordan A* har haft en bedre fornemmelse for hvor slut noden er på grund af dens brug af Heuristics. A* har meget færre besøgte nodes (blå celler i labyrinten) som har øget dens hastighed i at løse problemet. Hvor imod Dijkstra har over dobbelt så mange besøgte nodes som den skulle gå igennem.

[1] Heuristic Programming
[2] Dijkstra's Algorithm vs A* Algorithm
[3] Kevin Wang - Compare A* with Dijkstra algorithm

3.2 Gennemgang i min kode

Første step til at gennemgå funktionen Dijkstra's algoritme (7), er at den indtager 3 parameter. parametere er "grid", "startNode" og "Finishnode". Disse parameter bliver brugt til diverse ting i algorithmen. "grid" bruges til at indsamle alle "unvisitedNodes" nodes fra griddet. "StartNode" bruges til at definere hvor algoritmen skal starte, og selfølig "Finishnode" til at definere hvor vi skal slutte. Bemærk at algoritmen ved ikke på forhånd hvor "FinishNode" er, algoritmen skal blot bruge den til at se om det har fundet "FinishNode".

```
function dijkstra(grid , startNode , finishNode)
```

Algorithmen sortere så alle nodes i griddet ud fra deres afstand fra hinanden, til at finde de tætteste nodes på "startNode". Disse nodes bliver så defineret i variablet "closestNode".

```
sortNodesByDistance(unvisitedNodes);
const closestNode = unvisitedNodes.shift();
```

Nu begynder diverse betingelse at blive tjekket igennem for den "closestNode". Den "closestNode" bliver først tjekket for om den er en væg, hvis det er springer den node over. Derefter tjekket om vi sidder fast og der ikke er nogen løsning. Dette sker ved at se om alle resterende nodes vi kan få fat i, har deres oprindelig startværdi "Infinity".

```
if (closestNode.isWall) continue; // If we encounter a wall , we skip it.

// If the closest nodes distance = infinity, then we are trapped
// and should stop.
if (closestNode.distance === Infinity) return visitedNodesInOrder;
```

Når alle betingelserne er opfyldt sætter vi så det tætteste node "closestNode" til besøgt og tilføjer den til vores prioriterings kø.

```
// If node passed parameters, then visited
  closestNode.isVisited = true;
// Push node to priority queue
  visitedNodesInOrder.push(closestNode);
```

Hvis vi så er så heldige at finde vores mål "FinishNode" stopper vi så søgningen og returnere vores resultater. Ellers hvis vi ikke finder vores mål, opdatere vi bare vores nye grid med vores "closestNode" som besøgt.

```
// If reached finishNode, return visitedNodesInOrder for furthere use.
  if (closestNode === finishNode) return visitedNodesInOrder;
// Update grid with new visited nodes
  updateUnvisitedNeighbors(closestNode , grid);
```

3.2.1 Rekursion i min kode

Som nævnt 2.3.1 kan det rekursive kald i en funktion være direkte og indirekte. Det rekursive kald i mit tilfælde er indirekte og kan være svær at gennemskue.
Min kode for Dijkstra's Algoritme (7) er opbygget på, at den starter med indsamle alle nodes fra griddet. Alle nodes bliver opsamlet i et array "unvisitedNodes" fra ekstern funktion getAllNodes (12). Dermed er der allerede et loft på funktionen, dette loft vil så stoppe det rekursive kald, hvis der ikke er flere nodes tilbage i griddet:

```
const unvisitedNodes = getAllNodes(grid); // get nodes from grid
```

Dette er ligesom linje 2 i 2.3.1, her stopper det rekursive kalde også:

```
if (n == 1) return 1;
```

Det rekursive kald kommer via et while loop (linje 5 i 7). Dette while loop brydes kun, hvis alle "unvisitedNodes" har en længde (altså blevet søgt igennem).

```
while (!!unvisitedNodes.length) // caster unvisitedNodes til boolean
```

4 Konklution

5 Litteraturliste

6 Bilag

```
function dijkstra(grid, startNode, finishNode) {
  const visitedNodesInOrder = [];
  startNode.distance = 0;
  const unvisitedNodes = getAllCells(grid); // get cells from grid
  while (!!unvisitedNodes.length) {
    sortNodesByDistance(unvisitedNodes);
    const closestNode = unvisitedNodes.shift();

    if (closestNode != undefined) {
      // If we encounter a wall, we skip it.
      if (closestNode.isWall) continue;
      // If the closest nodes distance = infinity, then we are trapped
      // and should stop.
      if (closestNode.distance === Infinity) return visitedNodesInOrder;
      // If node passed parameters, then visited
      closestNode.isVisited = true;
      // Push node to priority queue
      visitedNodesInOrder.push(closestNode);
      // If reached finishNode, return visitedNodesInOrder for furthere use.
      if (closestNode === finishNode) return visitedNodesInOrder;
      // Update grid with new visited nodes
      updateUnvisitedNeighbors(closestNode, grid);
    } else console.log("error, closestNode returned 0");
  }
}
```

Listing 7: Kode for Dijkstra's Algoritme

```
function updateUnvisitedNeighbors(node, grid) {
const unvisitedNeighbors = getUnvisitedNeighbors(node, grid);

for (const neighbor of unvisitedNeighbors) {
  // set node from infinity to 1 (now visited).
  neighbor.distance = node.distance + 1;
  neighbor.previousNode = node; // set new node to previous node
  }
}
```

Listing 8: Kode for opdatering af unvisitedNodes

```
  function sortNodesByDistance(unvisitedNodes) {
  unvisitedNodes.sort(
    (nodeA, nodeB) => nodeA.distance - nodeB.distance
  );
}
```

Listing 9: Kode for finde tætteste nodes

```
  function getUnvisitedNeighbors(node, grid) {
    const neighbors = [];
    const { col, row } = node;
    if (row > 0) neighbors.push(grid[row - 1][col]);
    if (row < grid.length - 1) neighbors.push(grid[row + 1][col]);
    if (col > 0) neighbors.push(grid[row][col - 1]);
    if (col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
    return neighbors.filter((neighbor) => !neighbor.isVisited);
  }
```

Listing 10: Kode for at søge nye nodes

```
  function getNodesInShortestPathOrder(finishNode) {
  const NodesInShortestPathOrder = [];
  let currentNode = finishNode;
  while (currentNode !== null) {
    NodesInShortestPathOrder.unshift(currentNode); //shift back though
   finishNode
    currentNode = currentNode.previousNode;
  }
  console.log("Shorstes path length: ", NodesInShortestPathOrder.length);

  return NodesInShortestPathOrder;
}
```

Listing 11: Kode for finde korteste vej

```
function getAllNodes(grid) {
  const Nodes = [];
  for (const row of grid) {
    for (const node of row) {
      Nodes.push(node);
    }
  }
  return Nodes;
}
```

Listing 12: Kode for finde alle nodes i grid