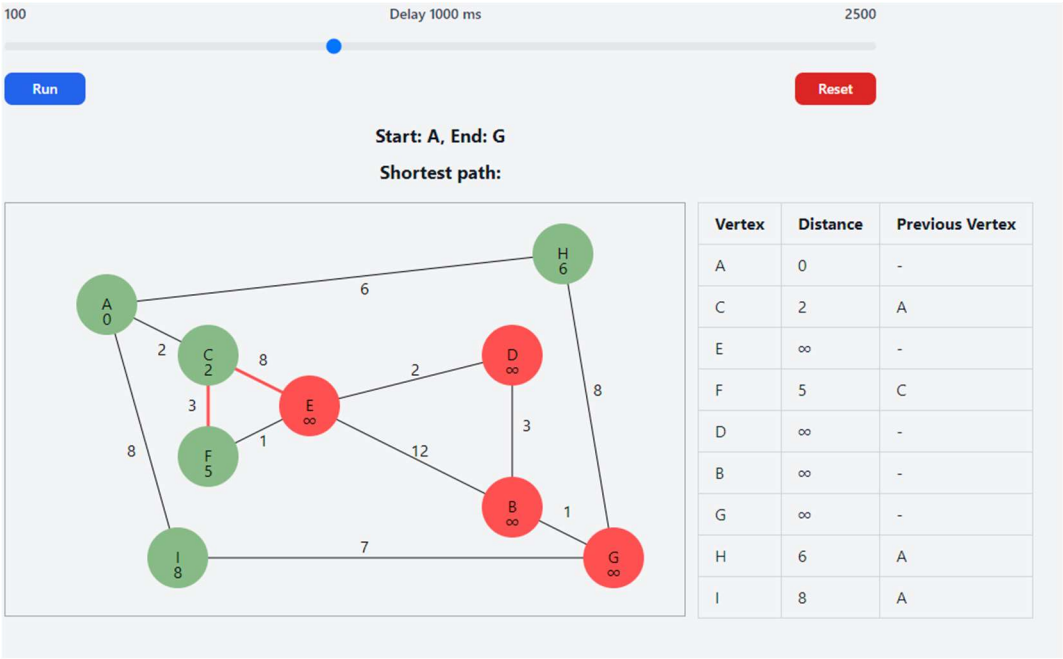
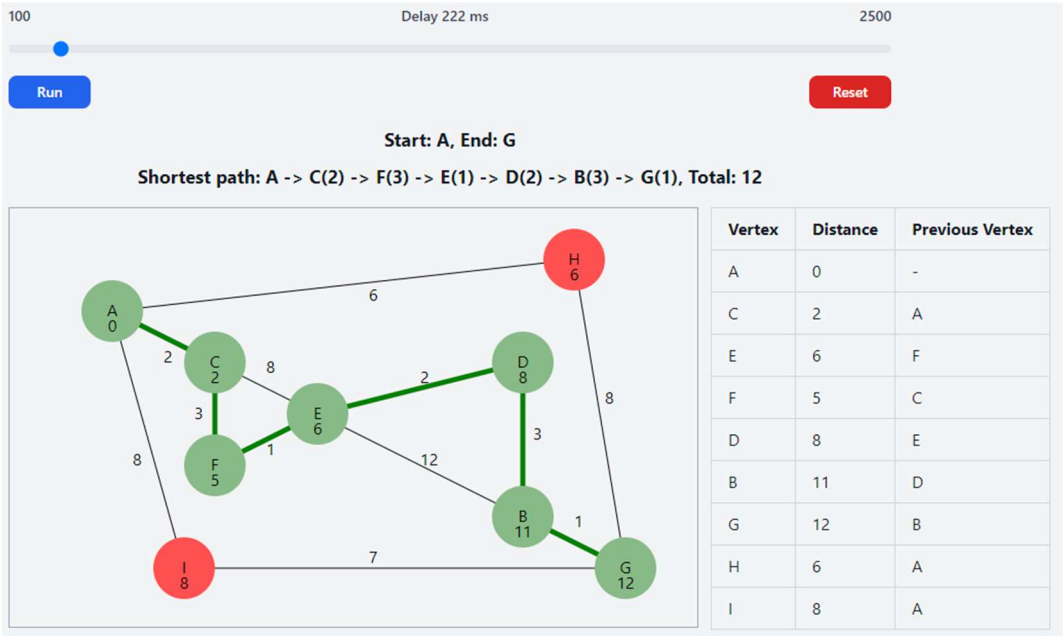
DIJKSTRA'S ALGORITME VISUALISERING



Vertex	Distance	Previous Vertex
A	0	-
C	2	A
E	∞	-
F	5	C
D	∞	-
B	∞	-
G	∞	-
H	6	A
I	8	A



Vertex	Distance	Previous Vertex
A	0	-
C	2	A
E	6	F
F	5	C
D	8	E
B	11	D
G	12	B
H	6	A
I	8	A

Navn: Frederik Hallengreen Hansen

Deployet udgave https://dijkstra.fhallengreen.com/

Github: https://github.com/FHallengreen/Dijkstra-algo

Beskrivelse

Dijkstras Algoritme er en metode, der bruges til at finde den korteste rute mellem to punkter i en vægtet graf. Dette projekt visualiserer algoritmens proces, hvor det viser, hvordan den iterativt vælger de nærmeste noder og opdaterer ruter for at finde den mest effektive vej til slutpunktet.

Algoritmer og Datastrukturer

Projektet implementerer Dijkstras Algoritme ved hjælp af en prioritetskø til at sikre, at noder med den mindste afstand behandles først. Grafen er repræsenteret som en adjacency list, hvilket gør det nemt at iterere over nabo-noder og opdatere deres afstande.

Prioritetskø

En prioritetskø er en datastruktur, der altid tillader adgang til det element med den højeste prioritet (i dette tilfælde, den korteste afstand). I min implementation bruger jeg en simpel array-baseret prioritetskø, der sorterer elementerne efter deres prioritet.

Kodebeskrivelse

Koden er struktureret således, at der er en hovedfil, script.js, som håndterer visualiseringen og algoritmens logik. Prioritetskøen og grafen er placeret i model.js, mens view.js indeholder funktioner til at opdatere UI.

I Dijkstra's algoritme bliver alle noder initialiseret med en afstand sat til uendelig, undtagen startnoden, som sættes til 0. Algoritmen bruger en prioritetskø til at holde styr på, hvilke noder der skal behandles baseret på deres aktuelle korteste afstand. En mængde (Set) kaldet visited bruges til at holde styr på de noder, der allerede er blevet behandlet.

Algoritmen kører, så længe prioritetskøen ikke er tom. Den node med den mindste afstand tages ud af køen. Hvis denne node er slutnoden, eller dens afstand er uendelig, stoppes algoritmen. For hver nabo til den aktuelt behandlede node, hvis naboen ikke allerede er besøgt, beregnes den nye potentielle afstand til naboen. Hvis den nye afstand er kortere end den kendte afstand, opdateres afstanden og forrige node, og naboen tilføjes til prioritetskøen med sin nye afstand. Visualiseringen opdateres løbende for at vise fremdriften i algoritmen.

Når Dijkstra's algoritme har kørt færdig, bliver funktionen findShortestPath kaldt for at rekonstruere den korteste vej fra startnoden til slutnoden. Dette gøres ved at følge forrige node fra slutnoden tilbage til startnoden. Hver node tilføjes til en sti-array, som derefter vendes om for at få den korrekte rækkefølge fra start til slut. Kanten mellem hvert par af noder i stien markeres for at vise den fundne korteste vej. Noder, der ikke er en del af stien, farves røde.

Nedenstående ses screenshot af psuedo kode fra Wikipedia og til højre min implementering af Dijstra’s algoritme.

```
function Dijkstra(Graph, source):
    create vertex priority queue Q

    dist[source] ← 0
    Q.add_with_priority(source, 0)

    for each vertex v in Graph.Vertices:
        if v ≠ source
            prev[v] ← UNDEFINED
            dist[v] ← INFINITY
            Q.add_with_priority(v, INFINITY)


    while Q is not empty:
        u ← Q.extract_min()
        for each neighbor v of u:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                prev[v] ← u
                dist[v] ← alt
                Q.decrease_priority(v, alt)

    return dist, prev
```

```
async function dijkstra(start, end) {
    const distances = {};
    const previous = {};
    const pq = new PriorityQueue();
    let visited = new Set();

    Object.keys(graph).forEach(vertex => {
        distances[vertex] = Infinity;
        previous[vertex] = null;
    });

    distances[start] = 0;
    pq.enqueue(start, 0);

    while (!pq.isEmpty()) {
        const { vertex: closestVertex } = pq.dequeue();

        if (closestVertex === end) break;

        if (distances[closestVertex] === Infinity) break;

        clearHighlightedEdges();

        visited.add(closestVertex);
        await updateVertexDistance(closestVertex, distances[closestVertex],
            previous[closestVertex], delay);

        for (let neighbor in graph[closestVertex]) {
            if (!visited.has(neighbor)) {
                highlightNeighbor(closestVertex, neighbor);
                await sleep(delay);
                let newDistance = distances[closestVertex] + graph[closestVertex][neighbor];
                if (newDistance < distances[neighbor]) {
                    distances[neighbor] = newDistance;
                    previous[neighbor] = closestVertex;
                    pq.enqueue(neighbor, newDistance);
                    await updateVertexDistance(neighbor, newDistance, closestVertex, delay);
                }
            }
        }
    }
    await findShortestPath(end, previous, start);
}
```

Tidskompleksitet og Optimeringer

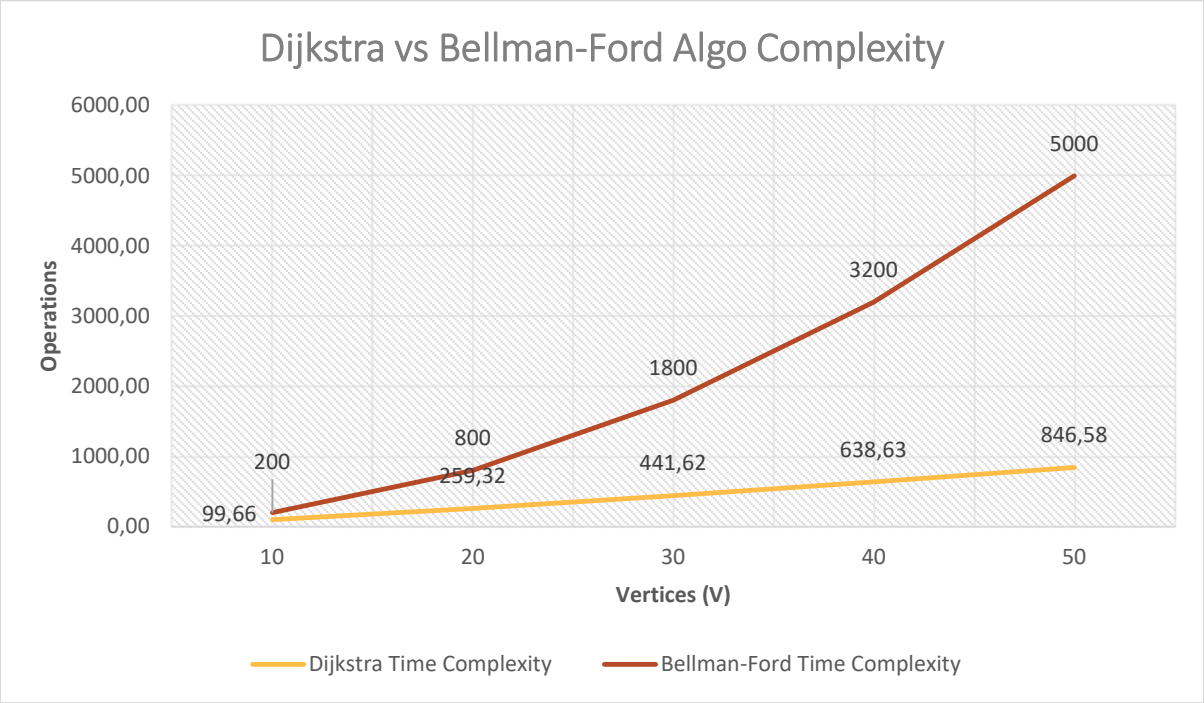
Dijkstra's algoritme har en tidskompleksitet på O((V + E) log V), hvor V er antallet er noder (vertices), og E er antallet af kanter (edges). Dette skyldes at hver node tilføjes til prioritetskøen én gang, og hver opdatering i prioritetskøen har en tidskompleksitet på O(log V). Funktionen til at finde den korteste vej fra slutnoden til startnoden har en tidskompleksitet på O(V), da hver node besøges én gang.

O-tiden kan forbedres yderligere ved at bruge en mere effektiv prioritets kø, som f.eks. binær heap eller Fibonacci heap i stedet for at sortere noderne ved hver iteration. Med en binær heap kan sorteringen og fjernelsen af nærmeste noder gøres i O(log V) tid.

Et alternativ til Dijkstra's algoritme er Bellman-Ford algoritmen. Bellman-Ford kan håndtere grafer med negative edges, hvilket Dijkstra's algoritme ikke kan. Bellman-Ford har dog en højere tidskompleksitet på O(V * E), hvilket gør den mindre effektiv for grafer med mange kanter sammenlignet med Dijkstra's algoritme.

Dijkstra's algoritme blev valgt i dette projekt, fordi den garanterer at finde den korteste vej i grafer med ikke-negative vægte, og fordi den generelt er hurtigere for grafer uden negative vægte.

Nedenfor har jeg lavet en graf der viser tidskompleksiteten for de to algoritmer.



Inspiration og kilder

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

https://www.youtube.com/watch?v=EFg3u_E6eHU

https://www.youtube.com/watch?v=pVfj6mxhdMw

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/

(Grokking Algorithms, 2016, pp. 116-139)