12 - Dijkstra’s Algorithm

Joseph Afework
CS 241

Dept. of Computer Science
California Polytechnic State University, Pomona, CA

Agenda

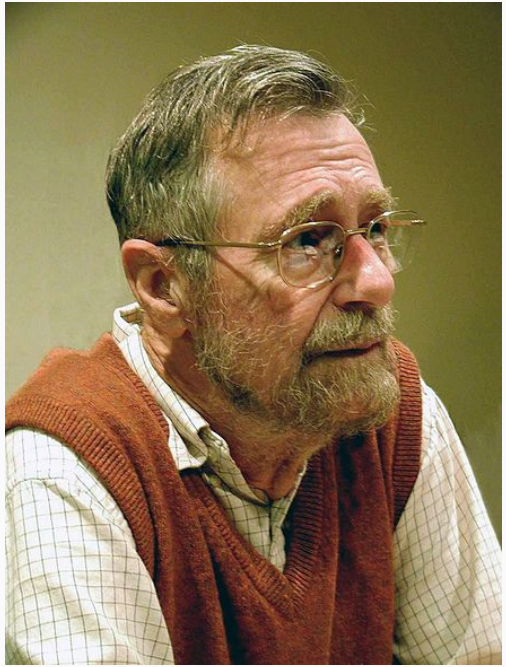
- Single-Source Shortest Path
- Dijkstra’s Algorithm
- Example
- Analysis
- Applications

Reading Assignment

- Read Chapter 28 - Graphs
  - Chapter 23 (Read about: **Dijkstra's Algorithm**)

Edsger Dijkstra

- **Edsger Dijkstra:** Dutch computer scientist
- In 1959 is a graph search algorithm that solves the **single-source shortest path problem for a graph** with non-negative edge path costs, producing a **shortest path tree**.
- This algorithm is often used in network routing protocols.



Dijkstra’s Algorithm

Observations

- Solves the single-source shortest path problem for a **graph with non-negative edge path costs**
- If all edge weights are non-negative, all shortest-path weights **must exist**.
- Generally uses a **greedy approach** to solving the problem.

Greedy Approach

“A **greedy algorithm** is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.” - wikipedia (https://en.wikipedia.org/wiki/Greedy_algorithm)

- Make the best decision now in hopes of finding the overall best solution later.
- For many other problems, greedy algorithms fail to produce the optimal solution (can yield suboptimal results)

Dijkstra’s Algorithm Overview

1. For a given **source vertex** in the graph, the algorithm finds the path with the **lowest cost between that vertex and every other vertex.**
    - Ex. Shortest path from vertex A to all other vertices in the graph

2. It can also be used for finding costs of shortest paths from a **single vertex** to a **single destination vertex** by stopping the algorithm once the shortest path to the destination vertex has been found.
    - Ex. Shortest path from vertex A to vertex B

Dijkstra’s Algorithm Overview

Idea: Greedy.

1. Maintain a set *S* of vertices whose shortest path distance from *s* are known.
2. At each step add to *S* the vertex *v* in (*V* - *S*) whose distance estimate from *s* is **minimal**.
3. Update the distance estimates of vertices adjacent to *v*.

Dijkstra’s Algorithm

Rules

- Let's call the node we are starting with an **initial node**.
- Let a **distance of a node** be the distance from the **initial node** to it.

Process: Dijkstra's algorithm will assign some initial distance values and will try to improve them step-by-step.

Dijkstra’s Algorithm Contd.

1. Assign to every node a distance value.
    a. Set it to **zero** for our initial node and to **infinity** for all other nodes.
2. Mark all nodes as unvisited. Set the initial node as the current node.
3. For the current node, consider all its unvisited neighbors and calculate their distances (from the initial node).
    a. Ex. If the current node (A) has a distance of 6, and an edge connecting it with another node (B) is 2, the distance to B through A will be 6 + 2 = 8.
    b. If this distance is less than the previously recorded distance (infinity in the beginning, zero for the initial node), overwrite the distance. **(Relaxation step.)**
4. Once done considering all neighbors of the current node, **mark it as visited**.
    a. A visited node will not be checked ever again; its distance recorded now is final and minimal.
5. Set the unvisited node with the smallest distance (from the initial node) as the next "current node" and continue from **step 3**. **(Greedy property.)**

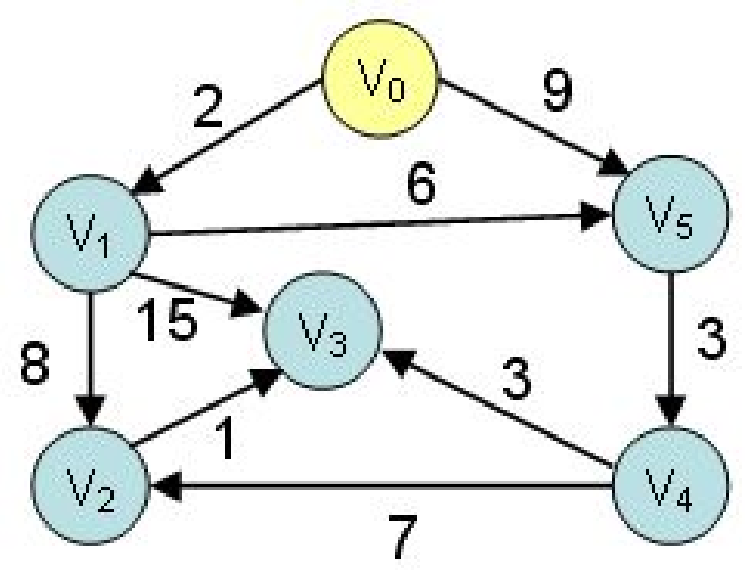
Pseudocode

```
1  function Dijkstra(Graph, source):
2      for each vertex v in Graph: // Initializations
3          dist[v] := infinity // Unknown distance function from source to v
4          previous[v] := undefined // Previous node in optimal path from source
5      dist[source] := 0 // Distance from source to source
6      Q := the set of all nodes in Graph // All nodes in the graph are unoptimized - thus are in Q (priority queue)
7      while Q is not empty: // The main loop
8          u := vertex in Q with smallest dist[]
9          if dist[u] = infinity:
10             break // all remaining vertices are inaccessible
11         remove u from Q
12         for each neighbor v of u: // where v has not yet been removed from Q
13             alt := dist[u] + dist_between(u, v)
14             if alt < dist[v]
15                 dist[v] := alt
16                 previous[v] := u
17     return previous[]
```

Pseudocode Contd.

- *u* is the vertex in *Q* with the smallest distance
- dist_between(*u*, *v*) returns the length between the two neighbor nodes *u* and *v*
- *alt* is the length of the path from the *source* node to *v* if it were to go through *u*
- *previous* array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the *source*

Example



Analysis

Worst Case: Generally ... $O(V^2)$

- By picking more optimized data structures, it can be further reduced...
- Overall time = O (V * (Time_to_extract_min + E * Time_to_decrease_dist))

Applications

- If the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. **(Project 3)**.
- The shortest path first is widely used in network routing protocols (sending packets across the shortest path etc).

Resources

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

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

https://www.youtube.com/watch?v=8Ls1RqHCOPw&feature=youtu.be