Dijkstra’s shortest-path algorithm

Edsger Dijkstra, in an interview in 2010 (CACM):

… the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiance, and tired, we sat down on the cafe terrace to drink a cup of coffee, and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. [Took place in 1956]


Visit http://www.dijkstrascry.com for all sorts of information on Dijkstra and his contributions. As a historical record, this is a gold mine.

1968 NATO Conference on Software Engineering

- In Garmisch, Germany
- Academicians and industry people attended
- For first time, people admitted they did not know what they were doing when developing/testing software. Concepts, methodologies, tools were inadequate, missing
- The term software engineering was born at this conference.
- Get a good sense of the times by reading these reports!

A7. Implement shortest-path algorithm

Not due until after the prelim, but it’s in your best interest to do it before it’s on the prelim.

Last semester: Average time was 3.3 hours.

This semester, because of the tutorial and today’s lecture, you will be better prepared for the assignment.

We give you complete set of test cases and a GUI to play with. Efficiency and simplicity of code will be graded.

We demo it.
Dijkstra's shortest path algorithm

The \( n (> 0) \) nodes of a graph numbered \( 0..n-1 \).

Each edge has a positive weight.

\( \text{wgt}(v1, v2) \) is the weight of the edge from node \( v1 \) to \( v2 \).

Some node \( v \) be selected as the start node.

Calculate length of shortest path from \( v \) to each node.

Use an array \( d[0..n-1] \): for each node \( w \), store in \( d[w] \) the length of the shortest path from \( v \) to \( w \).

\[
\begin{align*}
  d[0] &= 2 \\
  d[1] &= 5 \\
  d[2] &= 6 \\
  d[3] &= 7 \\
  d[4] &= 0
\end{align*}
\]

Theorem about the invariant

1. For a Settled node \( s \), \( d[s] \) is length of shortest \( v \to s \) path.
2. For a Frontier node \( f \), at least one \( v \to f \) path contains only settled nodes (except perhaps for \( f \)) and \( d[f] \) is the length of the shortest such path.
3. All edges leaving \( S \) go to \( F \).

Case 1: \( v \) is in \( S \).
Case 2: \( v \) is in \( F \). Note that \( d[v] = 0 \); it has minimum \( d \) value.
1. For \( s \), \( d[s] \) is length of shortest \( v \rightarrow s \) path.
2. For \( f \), \( d[f] \) is length of shortest \( v \rightarrow f \) path using red nodes (except for \( f \)).
3. Edges leaving \( S \) go to \( F \).

**Theorem:** For a node \( f \) in \( F \) with min \( d \) value, \( d[f] \) is shortest path length

Loopy question 1:
How does the loop start? What is done to truthify the invariant?

Loopy question 2:
When does loop stop? When is array \( d \) completely calculated?

Loopy question 3:
Progress toward termination?

Loopy question 4:
Maintain invariant?

Algorithm is finished!
Extend algorithm to include the shortest path

Let’s extend the algorithm to calculate not only the length of the shortest path but the path itself.

\[
\begin{align*}
d[0] &= 2 \\
d[1] &= 5 \\
d[2] &= 6 \\
d[3] &= 7 \\
d[4] &= 0
\end{align*}
\]

Let’s extend the algorithm to calculate not only the length of the shortest path but the path itself.

We should store in \( v \) itself the shortest path from \( v \) to every node? Or do we need another data structure to record these paths?

Maintain backpointers

When \( w \) not in \( S \) or \( F \):

Getting first shortest path so far:

This is our final high-level algorithm. These issues and questions remain:

1. How do we implement \( F \)?
2. The nodes of the graph will be objects of class Node, not ints. How will we maintain the data in arrays \( d \) and \( bk \)?
3. How do we tell quickly whether \( w \) is in \( S \) or \( F \)?
4. How do we analyze execution time of the algorithm?

Maintain backpointers

When \( w \) not in \( S \) or \( F \):

Implement \( F \) as a min-heap of Nodes!

The priority of a node in \( F \) is its \( d \)-value —the shortest distance known from \( v \) to that node.
f = node in F with min d value;
while (F ≠ {}) {
  f = node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
    if (w not in S or F) {
      d[w] = d[f] + wgt(f, w);
      add w to F; bk[w] = f;
    } else if (d[f]+wgt (f,w) < d[w]) {
      d[w] = d[f] + wgt(f, w);
      bk[w] = f;
    }
  }
}
Directed graph
n nodes reachable from v
e edges leaving those n nodes
F ≠ {} is true n times

Example. How many times
does F ≠ {} evaluate to true?
Yes! n times, v is in F initially.
Every node is put into F once
and taken out once.

How many times does
w not in S or F
evaluate to true?
For each neighbor w of f
if (w not in S or F) {
  d[w]:= d[f] + wgt(w, f);
  add w to F; bk[w]:= f;
} else if (d[f]+wgt(f, w) < d[w]) {
  d[w]:= d[f] + wgt(f, w);
  bk[w]:= f;
}
public class SFdata {
  private node backPointer;
  private int distance; ...
}

Assume: directed graph, using adjacency list
n nodes reachable from v
e edges leaving those n nodes

To find an upper bound on time
complexity, multiply complexity of each part by the number of times its executed.
Then add them up.
S = ∅; F = {v}; d[v] = 0;

while (F ≠ ∅) {
    f = node in F with min d value;
    Remove f from F, add it to S;
    for each neighbor w of f {
        if (w not in S or F) {
            d[w] = d[f] + wgt(f, w);
            add w to F; bk[w] = f;
        } else if (d[f] + wgt(f, w) < d[w]) {
            d[w] = d[f] + wgt(f, w);
            bk[w] = f;
        }
    }
}

Dense graph, so e close to n^2: Line 10 gives \(O(n^2 \log n)\)

Sparse graph, so e close to n: Line 4 gives \(O(n \log n)\)