SHORTEST PATH ALGORITHM

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 then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. [Took place in 1956]

Visit http://www.dijkstracy.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.
- The NATO Software Engineering Conferences: http://homepages.ccs.dur.ac.uk/~brian.randell/NATO/index.html
Get a good sense of the times by reading these reports!

A7. Implement shortest-path algorithm

Last semester: mean time: 3.7 hrs, median time: 4.0 hrs.
We give you complete set of test cases and a GUI to play with.
Efficiency and simplicity of code will be graded.
Read pinned Assignment A7 note carefully:
2. Important! Grading guidelines.
We demo it.

A6 due Thursday FRIDAY. Late deadline still Sunday
Working with a partner? Group before submitting!!

We will talk about prelim 2 (24 April) on Thursday.
1968 NATO Conference on Software Engineering, Garmisch, Germany

10/04/2018

Dijkstra’s shortest path algorithm

The n (> 0) nodes of a graph numbered 0..n-1.
Each edge has a positive weight.
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.

\[
d[0] = 2 \\
d[1] = 5 \\
d[2] = 6 \\
d[3] = 7 \\
d[4] = 0
\]

Frontier F

Settled S

Far off

The loop invariant

1. For a Settled node s, a shortest path from v to s contains only settled nodes and d[s] is length of shortest v \rightarrow s path.
2. For a Frontier node f, at least one v \rightarrow 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.

Theorem about the invariant

1. For a Settled node s, d[s] is length of shortest v \rightarrow s path.
2. For a Frontier node f, d[f] is length of shortest v \rightarrow f path using only Settled nodes (except for f).
3. All edges leaving S go to F.

Theorem. For a node f in F with minimum d value (over nodes in F), d[f] is the length of a shortest path from v to f.

Case 1: v is in S.
Case 2: v is in F. Note that d[v] is 0; it has minimum d value.

What does the theorem tell us about this frontier set?

(Cortland, 20 miles) (Dryden, 11 miles)
(Enfield, 10 miles) (Tburg, 15 miles)

Answer: The shortest path from the start node to Enfield has length 10 miles.

Note: the following answer is incorrect because we haven’t said a word about the algorithm! We are just investigating properties of the invariant:

Enfield can be moved to the settled set.
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?

**Algorithm**

\[
S = \{ \}; \quad F = \{ v \}; \quad d[v] = 0; \\
while ( F \neq \{ \}) 
\{
    f = \text{node in } F \text{ with min } d \text{ value; Remove } f \text{ from } F, \text{ add it to } S; \\
    \text{for each neighbor } w \text{ of } f 
    \{
        \text{if } (w \text{ not in } S \text{ or } F) 
        \{
            d[w] = d[f] + \text{wgt}(f, w); \\
            \text{add } w \text{ to } F; \\
        \} \text{ else } 
        \{
            \text{if } (d[f] + \text{wgt}(f, w) < d[w]) 
            \{
                d[w] = d[f] + \text{wgt}(f, w); \\
            \}
        \}
    \}
}\]

**Loopy question 3:** Progress toward termination?

**Loopy question 4:** Maintain invariant?

**Algorithm is finished!**

\( F \neq \{ \} \)
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.

\[
d[0] = 2 \\
d[1] = 5 \\
d[2] = 6 \\
d[3] = 7 \\
d[4] = 0
\]

For each node, maintain the backpointer on the shortest path to that node.

Shortest path to 0 is \(v \rightarrow 0\). Node 0 backpointer is 4.
Shortest path to 1 is \(v \rightarrow 0 \rightarrow 1\). Node 1 backpointer is 0.
Shortest path to 2 is \(v \rightarrow 0 \rightarrow 2\). Node 2 backpointer is 0.
Shortest path to 3 is \(v \rightarrow 0 \rightarrow 2 \rightarrow 1\). Node 3 backpointer is 2.

\(b[k][w]\) is \(w\)'s backpointer
\[
\begin{align*}
b[0] & = 2 & b[0][0] & = 4 \\
b[1] & = 5 & b[1][1] & = 0 \\
b[4] & = 0 & b[4][4] & = \text{none}
\end{align*}
\]

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 \(b[k]\)?
3. How do we tell quickly whether \(w\) is in \(S\) or \(F\)?
4. How do we analyze execution time of the algorithm?

S F Far off
\[
\begin{align*}
S & = \{ \} ; F & = \{v\} ; d[v] & = 0 \\
while & (F \neq \{\}) \{ \\
f & = \text{node in } F \text{ with min } d \text{ value; Remove } f \text{ from } F, \text{ add it to } S; \\
for & \text{ each neighbor } w \text{ of } f \{ \\
& \text{if } (w \text{ not in } S \text{ or } F) \{ \\
& \text{d[w] = d[f] + wgt(f, w); add } w \text{ to } F; b[k][w] = f; \\
& \text{else if } (d[f] + wgt(f, w) < d[w]) \{ \\
& \text{d[w] = d[f] + wgt(f, w); } \\
& b[k][w] = f; \\
& \}
& \}
\}
\}
\]

Use a min-heap, with the priorities being the distances!

Distances ---priorities--- will change. That's why we need updatePriority in Heap.java
S  F  Far off
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;
    }
  }
}
For what nodes do we need a distance and a backpointer?

S  F  Far off
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;
    }
  }
}
For every node in S or F we need both its d-value and its backpointer (null for v)

F implemented as a heap of Nodes. What data structure do we use to maintain an SF object for each node in S and F?

S  F  Far off
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;
    }
  }
}
For every node in S or F we need both its d-value and its backpointer (null for v)

public class SF {
  private int distance;
  private node backPtr;
  ...
}

F implemented as a min-heap: data replaces S, d, b

public class SFinfo {
  private int distance;
  private node backPtr;
}

Assume: n nodes reachable from v e edges leaving those n nodes

Investigate execution time. Important: understand algorithm well enough to easily determine the total number of times each part is executed/evaluated
while 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;
  }
}
}

HashMap<Node, SF> data

public class SF {

  private Node backPtr;
  private int distance;

  public SF(Node v) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public SF(Node v, HashMap<Node, SF> data) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public void add(Node v) {
    F.add(v);
    d[v] = 0;
    S.add(v);
  }

  public void remove(Node v) {
    F.remove(v);
    S.remove(v);
  }

  public int getDistance(Node v) {
    return d[v];
  }

  public Node getBackPtr(Node v) {
    return bk[v];
  }

  public void reset() {
    S.clear();
    F.clear();
    d.clear();
    bk.clear();
  }

}

Directed graph
n nodes reachable from v
e edges leaving the n nodes

while 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;
  }
}
}

HashMap<Node, SF> data

public class SF {

  private Node backPtr;
  private int distance;

  public SF(Node v) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public SF(Node v, HashMap<Node, SF> data) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public void add(Node v) {
    F.add(v);
    d[v] = 0;
    S.add(v);
  }

  public void remove(Node v) {
    F.remove(v);
    S.remove(v);
  }

  public int getDistance(Node v) {
    return d[v];
  }

  public Node getBackPtr(Node v) {
    return bk[v];
  }

  public void reset() {
    S.clear();
    F.clear();
    d.clear();
    bk.clear();
  }

}

Directed graph
n nodes reachable from v
e edges leaving the n nodes

while 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;
  }
}
}

HashMap<Node, SF> data

public class SF {

  private Node backPtr;
  private int distance;

  public SF(Node v) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public SF(Node v, HashMap<Node, SF> data) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public void add(Node v) {
    F.add(v);
    d[v] = 0;
    S.add(v);
  }

  public void remove(Node v) {
    F.remove(v);
    S.remove(v);
  }

  public int getDistance(Node v) {
    return d[v];
  }

  public Node getBackPtr(Node v) {
    return bk[v];
  }

  public void reset() {
    S.clear();
    F.clear();
    d.clear();
    bk.clear();
  }

}

Directed graph
n nodes reachable from v
e edges leaving the n nodes

while 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;
  }
}
}

HashMap<Node, SF> data

public class SF {

  private Node backPtr;
  private int distance;

  public SF(Node v) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public SF(Node v, HashMap<Node, SF> data) {
    d[v] = 0;
    S = new HashSet<Node>();
    F = new HashSet<Node>();
  }

  public void add(Node v) {
    F.add(v);
    d[v] = 0;
    S.add(v);
  }

  public void remove(Node v) {
    F.remove(v);
    S.remove(v);
  }

  public int getDistance(Node v) {
    return d[v];
  }

  public Node getBackPtr(Node v) {
    return bk[v];
  }

  public void reset() {
    S.clear();
    F.clear();
    d.clear();
    bk.clear();
  }

}
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 the n nodes
Expected-case analysis

1 x O(1)                              1
O(n)                              2
O(n)                              3
O(n log n)                    4
O(e)                              5
O(e)                             6
O(n)                             7
O(n log n)                    8
O((e–n) log n).      9
O(e–n)                  10

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)