Dijkstra’s Algorithm

- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a “best distance so far”
  - A priority queue will turn out to be useful for efficiency
- An example of a greedy algorithm
  - A series of steps
  - At each one the locally optimal choice is made

Dijkstra’s Algorithm: Idea

- Initially, start node has cost 0 and all other nodes have cost $\infty$
- At each step:
  - Pick closest unknown vertex $v$
  - Add it to the “cloud” of known vertices
  - Update distances for nodes with edges from $v$
- That’s it! (But we need to prove it produces correct answers)

The Algorithm

1. For each node $v$, set $v\.cost = \infty$ and $v\.known = false$
2. Set source\.cost = 0
3. While there are unknown nodes in the graph
   a) Select the unknown node $v$ with lowest cost
   b) Mark $v$ as known
   c) For each edge $(v,u)$ with weight $w$,
      $c1 = v\.cost + w // cost of best path through $v$ to $u$
      $c2 = u\.cost // cost of best path to $u$ previously known$
      if($c1 < c2$) {
          $u\.cost = c1 // if the path through $v$ is better$
          $u\.path = v // for computing actual paths$

Example #1

Order Added to Known Set:

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example #1

Order Added to Known Set:
A, C

vertex known? cost path
A  Y  0
B  ≤ 2  A
C  Y  1  A
D  ≤ 4  A
E  ≤ 12  C
F  ?
G  ?
H  ??

Example #1

Order Added to Known Set:
A, C, B

vertex known? cost path
A  Y  0
B  Y  2  A
C  Y  1  A
D  ≤ 4  A
E  ≤ 12  C
F  ≤ 4  B
G  ?
H  ??

Example #1

Order Added to Known Set:
A, C, B, D

vertex known? cost path
A  Y  0
B  Y  2  A
C  Y  1  A
D  ≤ 4  A
E  ≤ 12  C
F  ≤ 4  B
G  ?
H  ??

Example #1

Order Added to Known Set:
A, C, B, D, F

vertex known? cost path
A  Y  0
B  Y  2  A
C  Y  1  A
D  ≤ 4  A
E  ≤ 12  C
F  ≤ 4  B
G  ?
H  ≤ 7  F

Example #1

Order Added to Known Set:
A, C, B, D, F, H

vertex known? cost path
A  Y  0
B  Y  2  A
C  Y  1  A
D  ≤ 4  A
E  ≤ 12  C
F  ≤ 4  B
G  ≤ 8  H
H  Y  7  F
Features

- When a vertex is marked known, the cost of the shortest path to that node is known
  - The path is also known by following back-pointers
- While a vertex is still not known, another shorter path to it might still be found

Note: The "Order Added to Known Set" is not important
- A detail about how the algorithm works (client doesn’t care)
- Not used by the algorithm (implementation doesn’t care)
- It is sorted by path-cost, resolving ties in some way
  - Helps give intuition of why the algorithm works

Interpreting the Results

- Now that we’re done, how do we get the path from, say, A to E?

Stopping Short

- How would this have worked differently if we were only interested in:
  - The path from A to G?
  - The path from A to E?
Example #2

Order Added to Known Set:
A, D

Order Added to Known Set:
A, D, C

Order Added to Known Set:
A, D, C, E

Order Added to Known Set:
A, D, C, E, B

Order Added to Known Set:
A, D, C, E, B, F

Order Added to Known Set:
A, D, C, E, B, F, G
Example #3

How will the best-cost-so-far for Y proceed?
Is this expensive?

A Greedy Algorithm

• Dijkstra’s algorithm
  – For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges
  – An example of a greedy algorithm:
    – At each step, always does what seems best at that step
      • A locally optimal step, not necessarily globally optimal
    – Once a vertex is known, it is not revisited
      • Turns out to be globally optimal

Where are We?

• Had a problem: Compute shortest paths in a weighted graph with no negative weights
• Learned an algorithm: Dijkstra’s algorithm
• What should we do after learning an algorithm?
  – Prove it is correct
    • Not obvious!
    • We will sketch the key ideas
  – Analyze its efficiency
    • Will do better by using a data structure we learned earlier!

Correctness: Intuition

Rough intuition:

All the “known” vertices have the correct shortest path
  – True initially: shortest path to start node has cost 0
  – If it stays true every time we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need: When we mark a vertex “known” we won’t discover a shorter path later!
  – This holds only because Dijkstra’s algorithm picks the node with the next shortest path-so-far
  – The proof is by contradiction…
Correctness: The Cloud (Rough Sketch)

Suppose \( v \) is the next node to be marked known ("added to the cloud")
- The best-known path to \( v \) must have only nodes "in the cloud"
- Else we would have picked a node closer to the cloud than \( v \)
- Suppose the actual shortest path to \( v \) is different
  - It won’t use only cloud nodes, or we would know about it
  - So it must use non-cloud nodes. Let \( w \) be the first non-cloud node on this path. The part of the path up to \( w \) is already known and must be shorter than the best-known path to \( v \). So \( v \) would not have been picked. Contradiction.

Efficiency, first approach
Use pseudocode to determine asymptotic run-time
- Notice each edge is processed only once

\[
\text{Dijkstra}(V, E, v_{\text{Start}}):
\begin{align*}
\text{for } v \in V: & \quad v.\text{cost}=\infty; v.\text{known}=\text{False} \\
v_{\text{Start}}.\text{cost} = 0
\end{align*}
\]
\[
\text{while not all nodes are known:}
\begin{align*}
b & = \text{find unknown node with smallest cost} \\
b.\text{known} = \text{True} \\
\text{for edge } (b, a) \in E: & \quad \text{if not } a.\text{known:}
\begin{align*}
\text{if } b.\text{cost} + \text{weight}(b, a) < a.\text{cost:} & \quad a.\text{cost} = b.\text{cost} + \text{weight}(b, a) \\
a.\text{path} = b
\end{align*}
\end{align*}
\]
Improving asymptotic running time

- So far: $O(|V|^2)$
- We had a similar “problem” with topological sort being $O(|V|^2)$ due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges
- Solution?

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

Dijkstra(V, E, vStart):
  for v in V:
    v.cost = infinity; v.known = False
  vStart.cost = 0
  build-heap with all vertices
  while heap is not empty:
    b = deleteMin()
    b.known = True
    for edge = (b,a) in E:
      if not a.known:
        if b.cost + weight((b,a)) < a.cost:
          decreaseKey(a, “new cost – old cost”)
          a.path = b

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

Dijkstra(V, E, vStart):
  for v in V:
    v.cost = infinity; v.known = False
  vStart.cost = 0
  build-heap with all vertices
  while heap is not empty:
    b = deleteMin()
    b.known = True
    for edge = (b,a) in E:
      if not a.known:
        if b.cost + weight((b,a)) < a.cost:
          decreaseKey(a, “new cost – old cost”)
          a.path = b

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

Dijkstra(V, E, vStart):
  for v in V:
    v.cost = infinity; v.known = False
  vStart.cost = 0
  build-heap with all vertices
  while heap is not empty:
    b = deleteMin()
    b.known = True
    for edge = (b,a) in E:
      if not a.known:
        if b.cost + weight((b,a)) < a.cost:
          decreaseKey(a, “new cost – old cost”)
          a.path = b

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

Dijkstra(V, E, vStart):
  for v in V:
    v.cost = infinity; v.known = False
  vStart.cost = 0
  build-heap with all vertices
  while heap is not empty:
    b = deleteMin()
    b.known = True
    for edge = (b,a) in E:
      if not a.known:
        if b.cost + weight((b,a)) < a.cost:
          decreaseKey(a, “new cost – old cost”)
          a.path = b
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

Dijkstra(V, E, vStart):
  for v in V:
    v.cost = infinity; v.known = False
  vStart.cost = 0
  build-heap with all vertices
  while heap is not empty:
    b = deleteMin()
    b.known = True
    for edge = (b,a) in E:
      if not a.known:
        if b.cost + weight((b,a)) < a.cost:
          decreaseKey(a, "new cost - old cost")
          a.path = b

O(|V|)
O(|V| log |V|)
O(|E| log |V|)
O(|V| log |V| + |E| log |V|)

Dense vs. sparse again

- First approach: O(|V|^2)
- Second approach: O(|V| log |V| + |E| log |V|)
- So which is better?
  - Sparse: O(|V| log |V| + |E| log |V|) (if |E| > |V|, then O(|E| log |V|))
  - Dense: O(|V|^2)
- But, remember these are worst-case and asymptotic
  - Priority queue might have slightly worse constant factors
  - On the other hand, for “normal graphs”, we might call decreaseKey rarely (or not percolate far), making |E| log |V| more like |E|