CSE 373: Data Structures & Algorithms
Graph Traversals / Topological Sort

Riley Porter
Winter 2017
Course Logistics

• HW4 out $\Rightarrow$ graphs!

• Midterms back in section tomorrow. Regrade policy on the website.
Graphs Review from last time

What is some of the terminology for graphs and what do those terms mean?

– vertices and edges
– directed / undirected
– in-degree and out-degree
– connected and fully connected
– weighted / unweighted
– acyclic
– DAG: Directed Acyclic Graph
Graphs Applications Review

For each of the following examples:

– what are the vertices and what are the edges?
– would you use directed edges? Would they have self-edges?
– Are there 0-degree nodes? Is it strongly or weakly connected?
– Does it have weights? Do negative weights make sense?
– Does it have cycles? Is it a DAG?

• Web pages with links
• Facebook friends
• Methods in a program that call each other
• Road maps (e.g., Google maps)
• Airline routes
• Family trees
• Course pre-requisites
• Political donations to candidates
Graph Traversals
Graph Traversals

For an arbitrary graph and a starting node \( v \), find all nodes \textit{reachable} from \( v \) (i.e., there exists a path from \( v \))

- Possibly “do something” for each node
- Examples: print to output, set a field, etc.

• \textbf{Also solves}: Is an undirected graph connected?
• \textbf{Related but different problem}: Is a directed graph strongly connected?

\textbf{Basic idea of traversal}:

- Keep following nodes
- But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once
Abstract Idea in Pseudocode

```java
void traverseGraph(Node start) {
    Set pending = emptySet()
    pending.add(start)
    mark start as visited
    while (pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
            if (u is not marked visited) {
                mark u
                pending.add(u)
            }
    }
}
```
Running Time and Options

• Assuming add and remove for pending set are $O(1)$, entire traversal is $O(|E|)$ using an adjacency list representation

• The order we traverse depends entirely on add and remove
  – Popular choice: a stack “depth-first graph search” $\rightarrow$ DFS
  – Popular choice: a queue “breadth-first graph search” $\rightarrow$ BFS

• **DFS and BFS** are “big ideas” in computer science
  – Depth: recursively explore one part before going back to the other parts not yet explored
  – Breadth: explore areas closer to the start node first

Cool visualization: [http://visualgo.net/dfsbfs.html](http://visualgo.net/dfsbfs.html)
Example: trees

- A tree is a graph and make DFS and BFS are easier to “see”

```plaintext
DFS(Node start) {
    mark and process start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```

- A, B, D, E, C, F, G, H
- Exactly what we called a “pre-order traversal” for trees
  - The marking is because we support arbitrary graphs and we want to process each node exactly once
Example: trees

DFS2(Node start) {
    initialize stack s to hold start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}

• A, C, F, H, G, B, E, D
• A different but perfectly fine depth traversal
Example: trees

A, B, C, D, E, F, G, H

A “level-order” traversal

BFS(Node start) {
    initialize queue q to hold start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process"
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}"
Comparison

• Breadth-first always finds shortest length paths, i.e., “optimal solutions”
  – Better for “what is the shortest path from \( x \) to \( y \)”

• But depth-first can use less space in finding a path
  – If longest path in the graph is \( p \) and highest out-degree is \( d \) then DFS stack never has more than \( d \times p \) elements
  – But a queue for BFS may hold \( O(|V|) \) nodes

• A third approach (useful in Artificial Intelligence)
  – Iterative deepening (IDFS):
    • Try DFS but disallow recursion more than \( K \) levels deep
    • If that fails, increment \( K \) and start the entire search over
  – Like BFS, finds shortest paths. Like DFS, less space.
Saving the Path

• Our graph traversals can answer the reachability question:
  – “Is there a path from node x to node y?”

• But what if we want to actually output the path?
  – Like getting driving directions rather than just knowing it’s possible to get there!

• How to do it:
  – Instead of just “marking” a node, store the previous node along the path (when processing \texttt{u} causes us to add \texttt{v} to the search, set \texttt{v.path} field to be \texttt{u})
  – When you reach the goal, follow \texttt{path} fields back to where you started (and then reverse the answer)
  – If just wanted path \textit{length}, could put the integer distance at each node instead
Example using BFS

What is a path from Seattle to Atlanta

– Remember marked nodes are not re-enqueued
– Note shortest paths may not be unique
Topological Sort
**Problem**: Given a DAG $G = (V, E)$, output all vertices in an order such that no vertex appears before another vertex that has an edge to it.

**Example input:**

```
CSE 142 -> CSE 143 -> CSE 374
  |                     |
  |                      |
MATH 126               CSE 373
```

**One example output:**

```
126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415
```
• Why do we perform topological sorts only on DAGs?
  – Because a cycle means there is no correct answer

• Is there always a unique answer?
  – No, there can be 1 or more answers; depends on the graph
  – Graph with 5 topological orders:

• Do some DAGs have exactly 1 answer?
  – Yes, including all lists

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it
Uses of Topological Sort

• Figuring out how to graduate

• Computing an order in which to recompute cells in a spreadsheet

• Determining an order to compile files using a Makefile

• In general, taking a dependency graph and finding an order of execution

• ...

A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   – Think "write in a field in the vertex"
   – Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $u$ adjacent to $v$ (i.e. $u$ such that $(v,u)$ in $E$), decrement the in-degree of $u$
Example

Output:

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed?  

In-degree: 0 0 2 1 1 1 1 1 1 3
Example

Output:
126

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed?: x
In-degree: 0 0 2 1 1 1 1 1 1 3
1

CSE 142 → CSE 143 → CSE 373
CSE 374
CSE 410
CSE 413
CSE 415
CSE 417
XYZ

MATH 126
Example

Output:

126
142

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x

In-degree: 0 0 2 1 1 1 1 1 1 1 3

CSE 142 → CSE 143 → CSE 373 → XYZ

CSE 374 → CSE 410

CSE 413 → CSE 415 → CSE 417

MATH 126
Example

Output:

126
142
143

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x
In-degree: 0 0 2 1 1 1 1 1 1 3

CSE373: Data Structures & Algorithms
Example

Output:
126
142
143
374

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x
In-degree: 0 0 2 1 1 1 1 1 1 3

CSE373: Data Structures & Algorithms
Example

Output:

126
142
143
374
373

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x x x x

In-degree: 0 0 2 1 1 1 1 1 1 3

1 0 0 0 0 0 0 0 2

0

CSE142 → CSE143 → CSE373 → CSE374 → XYZ

MATH126 → CSE142

CSE373 → CSE410

CSE373 → CSE413

CSE373 → CSE415

CSE373 → CSE417

CSE374 → XYZ

CSE410 → XYZ

CSE413 → XYZ

CSE415 → XYZ

CSE417 → XYZ

XYZ
Example

Output:

126
142
143
374
373
410
413
415
417
XYZ

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x x x x x

In-degree: 0 0 2 1 1 1 1 1 1 3

1 0 0 0 0 0 0 0 2 0
Example

Output:

126
142
143
374
373
410
413
415
417
XYZ

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x x x x x
In-degree: 0 0 2 1 1 1 1 1 1 3
           1 0 0 0 0 0 0 0 2
           0 1

Example

Output:
126
142
143
374
373
417
410
413

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x x x x x x
In-degree: 0 0 2 1 1 1 1 1 1 3
1 0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0 2
0 1 0 0 0 0 0 0 2

X

Example

Output:
126
142
143
374
373
410
417
413
XYZ

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed?  x  x  x  x  x  x  x  x  x  x
In-degree: 0 0 2 1 1 1 1 1 1 3
           1 0 0 0 0 0 0 0 2 1
           0 1 0
CSE373: Data Structures & Algorithms
Example

Output:
126
142
143
374
373
410
413
415
417
XYZ

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x x x x x x x x x x

In-degree: 0 0 2 1 1 1 1 1 1 3

1 0 0 0 0 0 0 0 2

0

CSE373: Data Structures & Algorithms
• Needed a vertex with in-degree 0 to start
  – Will always have at least 1 because no cycles

• Ties among vertices with in-degrees of 0 can be broken arbitrarily
  – Can be more than one correct answer, by definition, depending on the graph
Psuedocode Example

```java
labelEachVertexWithItsInDegree();
for (ctr = 0; ctr < numVertices; ctr++) {
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```

• What is the worst-case running time?
Pseudocode Example

```plaintext
labelEachVertexWithItsInDegree();
for (ctr = 0; ctr < numVertices; ctr++) {
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```

- What is the worst-case running time?
  - Initialization $O(|V|+|E|)$ (assuming adjacency list)
  - Outer loop: runs $|V|$ times
  - `findNewVertex`: $O(|V|)$
  - Sum of all decrements for the whole algorithm assuming adjacency list: $O(|E|)$ (each edge is removed once)
  - So total is $O(|V|^2)$ – not good for a sparse graph!
A better idea

The trick is to avoid searching for a zero-degree node every time!
  – Keep the “pending” zero-degree nodes in a list, stack, queue, bag, table, or something
  – Order we process them affects output but not correctness or efficiency provided add/remove are both $O(1)$

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) $v = \text{dequeue}()$
   b) Output $v$ and remove it from the graph
   c) For each vertex $u$ adjacent to $v$ (i.e. $u$ such that $(v,u)$ in $E$), decrement the in-degree of $u$, if new degree is 0, enqueue it
Pseudocode Example 2

```java
labelAllAndEnqueueZeros();
while queue not empty {
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if (w.indegree==0)
            enqueue(v);
    }
}
```

• What is the worst-case running time?
Pseudocode Example 2

```plaintext
labelAllAndEnqueueZeros();
while queue not empty {
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if (w.indegree==0)
            enqueue(v);
    }
}
```

- What is the worst-case running time?
  - Initialization: $O(|V|+|E|)$ (assuming adjacency list)
  - Sum of all enqueues and dequeues: $O(|V|)$
  - Sum of all decrements: $O(|E|)$ (assuming adjacency list)
  - So total is $O(|E| + |V|)$ – much better for sparse graph!
Shortest Cost Path
Single source shortest paths

• Done: BFS to find the minimum path length from \( v \) to \( u \) in \( O(|E|+|V|) \)

• Actually, can find the minimum path length from \( v \) to every node
  – Still \( O(|E|+|V|) \)
  – No faster way for a “distinguished” destination in the worst-case

• Now: Weighted graphs

  Given a weighted graph and node \( v \),
  find the minimum-cost path from \( v \) to every node

• As before, asymptotically no harder than for one destination
• Unlike before, BFS will not work -> only looks at path length.
Shortest Path: Applications

• Driving directions

• Cheap flight itineraries

• Network routing

• Critical paths in project management
Not as easy

Why BFS won’t work: Shortest path may not have the fewest edges
  – Annoying when this happens with costs of flights

We will assume there are no negative weights
  • *Problem is ill-defined* if there are negative-cost cycles
  • *Today’s algorithm is wrong* if edges can be negative
    – There are other, slower (but not terrible) algorithms
Dijkstra

• Algorithm named after its inventor Edsger Dijkstra (1930-2002)
  – Truly one of the “founders” of computer science; this is just one of his many contributions

  – My favorite Dijkstra quote: “computer science is no more about computers than astronomy is about telescopes”
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
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 $w$,
      \[
      \begin{align*}
      c_1 &= v\.cost + w // cost of best path through $v$ to $u$ \\
      c_2 &= u\.cost // cost of best path to $u$ previously known \\
      \text{if}(c_1 < c_2) &\{ // if the path through $v$ is better \\
      &\quad u\.cost = c_1 \\
      &\quad u\.path = v // for computing actual paths \\
      \}
      \end{align*}
      \]
Important 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
**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>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>??</td>
<td></td>
<td></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

<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>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>≤ 1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>≤ 4</td>
<td>A</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

<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>≤ 2</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>≤ 4</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>≤ 12</td>
<td></td>
<td>C</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, B
Example #1

Order Added to Known Set:

A, C, B, D
Example #1

Order Added to Known Set:

A, C, B, D, F

<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>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>≤ 7</td>
<td>F</td>
</tr>
</tbody>
</table>
Example #1

Order Added to Known Set:

A, C, B, D, F, H
Example #1

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

<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>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Example #1

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

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

<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>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

Order Added to Known Set:
A, C, B, D, F, H, G, E
Example #2

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>yes</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>no</td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>no</td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>no</td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>no</td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>no</td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>no</td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:

A

<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></td>
</tr>
<tr>
<td>C</td>
<td>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>≤ 1</td>
<td>A</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>
</tbody>
</table>
Example #2

Order Added to Known Set:

A, D

<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>≤ 6</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>≤ 2</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>≤ 2</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>≤ 7</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td>≤ 6</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:

A, D, C

<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>≤ 6</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤ 4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

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

vertex | known? | cost | path
--- | --- | --- | ---
A | Y | 0 | 
B | | ≤ 3 | E
C | Y | 2 | A
D | Y | 1 | A
E | Y | 2 | D
F | | ≤ 4 | C
G | | ≤ 6 | D
Example #2

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

<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>Y</td>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤ 4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:

A, D, C, E, B, F

<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>Y</td>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

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

<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>Y</td>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #3

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

Is this expensive?
Example #3

How will the best-cost-so-far for Y proceed? 90, 81, 72, 63, 54, ...

Is this expensive? No, each edge is processed only once
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, irrevocably 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.

CSE373: Data Structures & Algorithms
Naïve 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?
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?
  - A priority queue holding all unknown nodes, sorted by cost
  - But must support `decreaseKey` operation
    - Must maintain a reference from each node to its current position in the priority queue
    - Conceptually simple, but can be a pain to code up
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=infinity, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!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

```pseudocode
dijkstra(Graph G, Node start) {
    for each node: x.cost=\infty, x.known=false
    start.cost = 0
    build-heap with all nodes
    while (heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if (!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|)