Shortest Paths in Graphs

CS16: Introduction to Data Structures & Algorithms
Spring 2019
Outline

- Shortest Paths
- Breadth First Search
- Dijkstra’s Algorithm
What is a Shortest Path?

- Given weighted graph $G$ (weights on edges)...
- ...what is shortest path from node $u$ to $v$?

Applications
- Google maps
- Routing packets on the Internet
- Social networks
Single Source Shortest Paths (SSSP)

- Given a graph and a source node
  - find the shortest paths to all other nodes
Simpler Problem: Unit Edges

- Let's start with simpler problem
- On graph where every edge has unit cost
Simpler Problem: Unit Edges

- What is shortest path from A to each node?
  - B: [A, B]
  - D: [A, B, D] or [A, C, D]
  - C: [A, C]
  - E: [A, B, E] or [A, C, E]
Simpler Problem: Unit Edges

- Is there an algorithm we’ve already seen that solves problem?
  - Hint: yes!
- What graph traversals have we learned?
Breadth-First Search

- Use BFS to find shortest path from A to E.
- Consider all steps of adding/removing nodes from queue ...
- ...and updating each node's 'previous' pointer.

![Diagram of graph with nodes A, B, C, D, E and edges labeled with 1]
Breadth-First Search

- Use BFS to find shortest path from A to E.
- Consider all steps of adding/removing nodes from queue ...
- ...and updating each node's 'previous' pointer.

Activity #1

1 min
Breadth-First Search

- Use BFS to find shortest path from A to E.
- Consider all steps of adding/removing nodes from queue …
- …and updating each node's 'previous' pointer.
Breadth First Search

- BFS always reaches target node in fewest steps
- Let’s look at path from A to E
Breadth First Search Simulation

- **Strategy**
  - BFS uses queue to store nodes to visit
  - Enqueue start node
  - Decorate nodes w/ previous pointers to keep track of path

![Queue Diagram](image)
Breadth First Search Simulation

- Dequeue A
- Decorate its neighbors w/ “prev: A”
- Enqueue them
Breadth First Search Simulation

- Dequeue B and repeat...
- ...but ignoring nodes that have been decorated

<table>
<thead>
<tr>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

prev: A

prev: B

prev: A

prev: B
Breadth First Search Simulation

- Dequeueing C and D has no effect...
- ...since their neighbors have been decorated
Breadth First Search Simulation

- When we dequeue E...
- ...we traverse the prev pointers to return paths
  - shortest path to E: [A, B, E]
Non-Unit Edge Weights

- What if edge weights are not 1?
- More complicated
Shortest Path

- Fill in missing spaces using graph below
- Use A as source vertex

Activity #2

2 min
Shortest Path

- Fill in missing spaces using graph below
- Use A as source vertex

Activity #2
Shortest Path

- Fill in missing spaces using graph below
- Use A as source vertex

Activity #2

1 min
Shortest Path

- Fill in missing spaces using graph below
- Use A as source vertex
Non-unit Edge Weights

<table>
<thead>
<tr>
<th>Goal Node</th>
<th>Shortest Path</th>
<th>Shortest Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>[A, C, B]</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>[A, C]</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>[A, C, B, D]</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>[A, C, B, E]</td>
<td>6</td>
</tr>
</tbody>
</table>

![Graph with labeled nodes and edges with weights]
Shortest Path Application

- Road trip
- Alina, Maggie, Prakrit & Stephanie want to get from PVD to SF…
- …following limited set of highways
- Cities are nodes and highways are edges
- Get to SF using shortest path
Our Graph
Our Graph

What is the cost of this path?
Our Graph

What is the cost of this path? Is there a shorter path?
What is the cost of this path? Is there a shorter path?
Shortest Path

- Why does BFS work with unit edges?
  - Nodes visited in order of total distance from source
- We need way to do the same even when edges have distinct weights!
- How can we do this?
  - Hint: we’ll use a data structure we’ve already seen
Shortest Path

- Use a priority queue!
  - where priorities are total distances from source
  - By visiting nodes in order returned by `removeMin()`...
  - ...you visit nodes in order of how far they are from source

- You guarantee shortest path to node because...
  - ...you don’t explore a node until all nodes closer to source have already been explored
Dijkstra’s Algorithm

- The algorithm is as follows:
  - Decorate source with distance 0 & all other nodes with \(\infty\)
  - Add all nodes to priority queue w/ distance as priority
  - While the priority queue isn’t empty
    - Remove node from queue with minimal priority
    - Update distances of the removed node’s neighbors if distances decreased

- When algorithm terminates, every node is decorated with minimal cost from source
Dijkstra’s Algorithm Example

- **Step 1**
  - Label source with dist. 0
  - Label other vertices with dist. $\infty$
  - Add all nodes to $Q$

- **Step 2**
  - Remove node with min. priority from $Q$ (S in this example).
  - Calculate dist. from source to removed node’s neighbors…
  - …by adding adjacent edge weights to S’s dist.
Dijkstra’s Algorithm Example

- **Step 3**
  - While \( Q \) isn’t empty,
    - repeat previous step
    - removing \( A \) this time
  - Priorities of nodes in \( Q \) may have to be updated
    - ex: \( B \)’s priority

- **Step 4**
  - Repeat again by removing vertex \( B \)
  - Update distances that are shorter using this path than before
    - ex: \( C \) now has a distance 6 not 10
Dijkstra’s Algorithm Example

- Step 5
  - Repeat
    - this time removing C

- Step 6
  - After removing D...
  - …every node has been visited...
  - …and decorated w/ shortest dist. to source
Dijkstra’s Algorithm Example

› Previous example decorated nodes with shortest distance but did not “create” paths

› How could you enhance algorithm to return the shortest path to a particular node?
  › Previous pointers!

› Let’s do another example
  › but this time without explanation
  › try to explain what the algorithm is doing at each step
Dijkstra's Example

A

B

C

D

E

\[ \begin{array}{|c|c|c|c|c|c|} 
\hline
A & B & C & D & E \\
\hline
0 & \infty & \infty & \infty & \infty \\
\hline
\end{array} \]
Dijkstra's Example

![Graph]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>∞</td>
</tr>
</tbody>
</table>
Dijkstra’s Example

A
---
B
---
C
---
D
---
E

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Dijkstra’s Example

A  B  C  D  E
0  3  2  5  6
Dijkstra’s Example

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```
Simulate Dijkstra’s

Activity #3 2 min
Simulate Dijkstra’s

Activity #3

2 min
Simulate Dijkstra’s

1 min

Activity #3
Simulate Dijkstra's Activity #3
Dijkstra’s Algorithm

- Dijkstra’s algorithm is an example of a class of algorithms we previously mentioned.
- Since it uses a priority queue,
  - at each step of iteration...
  - …we consider next closest node given the information we have.
- What algorithm paradigm does this fall under?
function **dijkstra**(*G*, *s*):

  // Input: graph *G* with vertices *V*, and source *s*
  // Output: Nothing
  // Purpose: Decorate nodes with shortest distance from *s*

  for *v* in *V*:
    *v*.dist = infinity  // Initialize distance decorations
    *v*.prev = null     // Initialize previous pointers to null

  *s*.dist = 0         // Set distance to start to 0

  *PQ* = PriorityQueue(*V*)  // Use *v*.dist as priorities

  while *PQ* not empty:
    *u* = *PQ*.removeMin()
    for all edges (*u*, *v*):
      if *u*.dist + cost(*u*, *v*) < *v*.dist:  // cost() is weight
        *v*.dist = *u*.dist + cost(*u*,*v*)  // Replace as necessary
        *v*.prev = *u*                     // Maintain pointers for path
        *PQ*.decreaseKey(*v*, *v*.dist)
function dijkstra(G, s):
    for v in V: // 1. O(__)
        v.dist = infinity
        v.prev = null
    s.dist = 0

    PQ = PriorityQueue(V) // 2. O(__)
    while PQ not empty: // 3. O(__)
        u = PQ.removeMin() // 4. O(__)
        for all edges (u, v): // 5. O(__)
            if v.dist > u.dist + cost(u, v):
                v.dist = u.dist + cost(u,v)
                v.prev = u
                PQ.decreaseKey(v, v.dist) // 6. O(__)

    Total: 7. O(__)
function `dijkstra(G, s)`:
  for v in V:  // 1. O(__)
    v.dist = infinity
    v.prev = null
  s.dist = 0

  PQ = PriorityQueue(V)  // 2. O(__)
  while PQ not empty:  // 3. O(__)
    u = PQ.removeMin()  // 4. O(__)
    for all edges (u, v):  // 5. O(__)
      if v.dist > u.dist + cost(u, v):
        v.dist = u.dist + cost(u, v)
        v.prev = u
        PQ.decreaseKey(v, v.dist)  // 6. O(__)

Total: 7. O(__)

Activity #3

2 min
function `dijkstra`(G, s):
  for v in V: // 1. O(__)
    v.dist = infinity
    v.prev = null
  s.dist = 0

  PQ = PriorityQueue(V) // 2. O(__)
  while PQ not empty: // 3. O(__)
    u = PQ.removeMin() // 4. O(__)
    for all edges (u, v): // 5. O(__)
      if v.dist > u.dist + cost(u, v):
        v.dist = u.dist + cost(u, v)
        v.prev = u
        PQ.decreaseKey(v, v.dist) // 6. O(__)

Total: 7. O(__)

Activity #3
function dijkstra(G, s):
    for v in V:  // 1. O(__)
        v.dist = infinity
        v.prev = null
    s.dist = 0

    PQ = PriorityQueue(V)  // 2. O(__)
    while PQ not empty:  // 3. O(__)
        u = PQ.removeMin()  // 4. O(__)
        for all edges (u, v):  // 5. O(__)
            if v.dist > u.dist + cost(u, v):
                v.dist = u.dist + cost(u, v)
                v.prev = u
                PQ.decreaseKey(v, v.dist)  // 6. O(__)

Total: 7. O(__)
function \texttt{dijkstra}(G, s):
  for \(v\) in \(V\):
    \(v\).dist = infinity
    \(v\).prev = null
  \(s\).dist = 0

\(PQ = \text{PriorityQueue}(V)\)

while \(PQ\) not empty:
  \(u = PQ.removeMin()\)
  for all edges \((u, v)\):
    if \(v\).dist > \(u\).dist + cost(u, v):
      \(v\).dist = \(u\).dist + cost(u, v)
      \(v\).prev = u
  \(PQ.decreaseKey(v, v\).dist\)
Dijkstra Runtime

- Depends on priority queue implementation
- If PQ implemented with Array or Linked List
  - `insert()` is $O(1)$
  - `removeMin()` is $O(|V|)$
    - you have to scan to find min-priority element
  - `decreaseKey()` is $O(1)$
    - you already have node when you change its key
function **dijkstra**(G, s):
    for v in V:
        v.dist = infinity
        v.prev = null
    s.dist = 0

    PQ = PriorityQueue(V)
    while PQ not empty:
        u = PQ.removeMin()
        for all edges (u, v):
            if v.dist > u.dist + cost(u, v):
                v.dist = u.dist + cost(u,v)
                v.prev = u
                PQ.decreaseKey(v, v.dist)
If PQ implemented with Array or Linked List

\[ O(|V| + |V| + |V|^2 + |E|) = O(|V|^2 + |E|) \]

\[ = O(|V|^2) \]

since \(|E| \leq |V|^2\)
Dijkstra Runtime w/ Heap

- If PQ implemented with Heap
  - `insert()` is $O(\log |V|)$
    - you may need to upheap
  - `removeMin()` is $O(\log |V|)$
    - you may need to downheap
  - `decreaseKey()` is $O(\log |V|)$
    - assume we have dictionary that maps vertex to heap entry in $O(\log |V|)$ time (so no need to scan heap to find entry)
    - you may need to upheap after decreasing the key
Dijkstra Runtime w/ Heap

```python
function dijkstra(G, s):
    for v in V:
        v.dist = infinity
        v.prev = null
    s.dist = 0
    PQ = PriorityQueue(V)
    while PQ not empty:
        u = PQ.removeMin()
        for all edges (u, v):
            if v.dist > u.dist + cost(u, v):
                v.dist = u.dist + cost(u,v)
                v.prev = u
                PQ.decreaseKey(v, v.dist)
```

- \(O(|V|)\)
- \(O(|V| \log |V|)\)
- \(O(|E|)\)
- Total \(O(|V| \log |V|)\)
Dijkstra Runtime w/ Heap

- If PQ implemented with Heap

\[ O(|V| + |V| \log |V| + |V| \log |V| + |E| \log |V|) \]

\[ = O(|V| + |V| \log |V| + |E| \log |V|) \]

\[ = O\left( (|V| + |E|) \cdot \log |V| \right) \]

- Note

  - though the \( O(|E|) \) loop is nested in the \( O(|V|) \) loop
  - we visit each edge at most twice rather than \( |V| \) times
  - That’s why while loop is \( O\left( (V \log |V|) + (|E| \log |V|) \right) \)
Dijkstra’s on Graph with Negative Edges

Activity #5

1 min
Dijkstra’s on Graph with Negative Edges

Start

A

B

2

C

-7

D

5

End

Activity #5

1 min
Dijkstra’s on Graph with Negative Edges

Activity #5

Start

A

B

C

D

End

0 min
Dijkstra isn’t perfect!

- We can find shortest path on weighted graph in 
  - $O((|V| + |E|) \times \log|V|)$
  - or can we...
- Dijkstra fails with negative edge weights
- Returns $[A, C, D]$ when it should return $[A, B, C, D]$
Negative Edge Weights

- Negative edge weights are a problem for Dijkstra.
- But negative cycles are even worse!
  - because there is no true shortest path!
Bellman-Ford Algorithm

- Algorithm that handles graphs with negative edge weights
- Similar to Dijkstra’s but more robust
  - Returns same output as Dijkstra’s for any graph with only positive edge weights (but runs slower)
  - Returns correct shortest paths for graphs with negative edge weights