Breadth-first Search and Shortest Paths in Graphs

CS16: Introduction to Data Structures & Algorithms
Summer 2021
BFT and DFT

- Remember BFT and DFT on trees?
- We can also do them on graphs
  - a tree is just a special kind of graph
  - often used to find certain values in graphs
Breadth First Traversal: Tree vs. Graph

function `treeBFT(root)`:
//Input: Root node of tree
//Output: Nothing
Q = new Queue()
Q.enqueue(root)
while Q is not empty:
    node = Q.dequeue()
doSOMETHING(node)
enqueue node’s children

doSOMETHING() could print, add to list, decorate node etc…

function `graphBFT(start)`:
//Input: start vertex
//Output: Nothing
Q = new Queue()
start.visited = true
Q.enqueue(start)
while Q is not empty:
    node = Q.dequeue()
doSOMETHING(node)
for neighbor in adj nodes:
    if not neighbor.visited:
        neighbor.visited = true
        Q.enqueue(neighbor)

Mark nodes as visited otherwise you will loop forever!
Applications: Flight Paths Exist

- Given undirected graph with airports & flights
  - is it possible to fly from one airport to another?

- Strategy
  - use breadth first search starting at first node
  - and determine if ending airport is ever visited
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?
Applications: Flight Paths Exist

- Is there flight from SFO to PVD?

- Yes! but how do we do it with code?
function **pathExists**(from, to):
    // Input: from: vertex, to: vertex
    // Output: true if path exists, false otherwise
    Q = new Queue()
    from.visited = true
    Q.enqueue(from)
    while Q is not empty:
        airport = Q.dequeue()
        if airport == to:
            return true
        for neighbor in airport’s adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                Q.enqueue(neighbor)
    return false
Applications: Flight Layovers

- Given undirected graph with airports & flights
  - decorate vertices w/ least number of stops from a given source
  - if no way to get to an airport decorate w/ $\infty$

Strategy

- decorate each node w/ initial ‘stop value’ of $\infty$
- use breadth first traversal to decorate each node...
- …w/ ‘stop value’ of one greater than its previous value
Flight Layovers Pseudo-Code

function numStops(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest number of
    //         layovers from source.

    for every node in G:
        node.stops = infinity

    Q = new Queue()
    source.stops = 0
    source.visited = true
    Q.enqueue(source)

    while Q is not empty:
        airport = Q.dequeue()
        for neighbor in airport’s adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                neighbor.stops = airport.stops + 1
                Q.enqueue(neighbor)
Flight Layovers Example
Flight Layovers Example
What if we want a path?

- numStops gives us the distance
- Want to know how to get from (e.g.) HNL to LGA
- Strategy: at each node we reach, record the node we used to get there
function `numStops(G, source):`

// Input: G: graph, source: vertex
// Output: Nothing
// Purpose: decorate each vertex with the lowest number of layovers from source.

for every node in G:
    node.stops = infinity
    node.previous = null

Q = new Queue()
source.stops = 0
source.visited = true
Q.enqueue(source)

while Q is not empty:
    airport = Q.dequeue()
    for neighbor in airport’s adjacent nodes:
        if not neighbor.visited:
            neighbor.visited = true
            neighbor.stops = airport.stops + 1
            neighbor.previous = airport
            Q.enqueue(neighbor)
Flight paths

Infinity

BTV

∞

JFK

∞

HNL

0

SFO

2 (LAX)

LAX

1 (HNL)

ORD

2 (LAX)

DFW

2 (LAX)

LGA

3 (DFW)

MIA

3 (DFW)

PVD

3 (ORD)
Single Source Shortest Paths

- SSSP problem: find shortest paths to all other nodes in a graph from a particular starting node
- Graph can be directed or undirected (we’ll present on undirected graphs)
- Edges can have weights
Train trip!
Train trip!

What’s the trip between PVD->SF that makes fewest stops?
Train trip!

What’s the trip between PVD->SF that makes fewest stops?
Train trip!

What’s the cheapest trip?

Start

PVD

CLE

NYC

PHL

DC

ATL

STL

CHI

35

20

15

20

15

15

10

10

10

10

10

10

10

10

End

SF

LA

PHX
Train trip!

What’s the cheapest trip?

BFS ignores edge weights!
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!
Can we modify BFS?

```plaintext
function distance(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest cost of
    //         a path from the source.

    for every node in G:
        node.stops = infinity
        node.previous = null

    Q = new Queue()
    source.stops = 0
    source.visited = true
    Q.enqueue(source)

    while Q is not empty:
        airport = Q.dequeue()
        for neighbor in airport’s adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                neighbor.stops = airport.stops + 1
                neighbor.previous = airport
                Q.enqueue(neighbor)
```

---

**Can we modify BFS?**

```plaintext
function distance(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest cost of
    //         a path from the source.

    for every node in G:
        node.stops = infinity
        node.previous = null

    Q = new Queue()
    source.stops = 0
    source.visited = true
    Q.enqueue(source)

    while Q is not empty:
        airport = Q.dequeue()
        for neighbor in airport’s adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                neighbor.stops = airport.stops + 1
                neighbor.previous = airport
                Q.enqueue(neighbor)
```
function distance(G, source):
   //Input: G: graph, source: vertex
   //Output: Nothing
   //Purpose: decorate each vertex with the lowest cost of
   //         a path from the source.

   for every node in G:
      node.distance = infinity
      node.previous = null

   Q = new Queue()
   source.distance = 0
   source.visited = true
   Q.enqueue(source)

   while Q is not empty:
      node = Q.dequeue()
      for neighbor in nodes’s adjacent nodes:
         if not neighbor.visited:
            neighbor.visited = true
            neighbor.distance = node.distance + 1
            neighbor.previous = node
            Q.enqueue(neighbor)
Can we modify BFS?

```plaintext
function distance(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest cost of
          //a path from the source.

    for every node in G:
        node.distance = infinity
        node.previous = null

    Q = new Queue()
    source.distance = 0
    source.visited = true
    Q.enqueue(source)
    while Q is not empty:
        node = Q.dequeue()
        for neighbor in node's adjacent nodes:
            if node.distance + cost(node, neighbor) < neighbor.distance:
                neighbor.visited = true
                neighbor.distance = node.distance + 1
                neighbor.previous = node
                Q.enqueue(neighbor)
```
Can we modify BFS?

```plaintext
def distance(G, source):
    //Input: G: graph, source: vertex
    //Output: Nothing
    //Purpose: decorate each vertex with the lowest cost of
    //         a path from the source.

    for every node in G:
        node.distance = infinity
        node.previous = null

    Q = new Queue()
    source.distance = 0
    source.visited = true
    Q.enqueue(source)
    while Q is not empty:
        node = Q.dequeue()
        for neighbor in node's adjacent nodes:
            if node.distance + cost(node, neighbor) < neighbor.distance:
                neighbor.distance = node.distance + cost(node, neighbor)
                neighbor.previous = node
                somehow add neighbor to Q at the right place
```
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 ∞
  - 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 distance 0
- Label other vertices with distance \( \infty \)
- Add all nodes to \( Q \)

Step 2
- Remove node with minimum priority from \( Q \) (\( S \) in this example).
- Calculate distance from source to removed node's neighbors...
- ...by adding adjacent edge weights to \( S \)'s distance.
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 Example 2

A

B

C

D

E

\[ \begin{array}{ccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
\end{array} \]
Dijkstra’s Example

![Graph Diagram]

<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>2</td>
<td>∞</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
Dijkstra’s Algorithm

- Comes up with an *optimal* solution
  - *shortest* path to each node
- Like many optimization algorithms, uses dynamic programming
  - overlapping subproblems (distances to nodes)
  - solved in a particular order (closest first)
- Dijkstra’s is *greedy*
  - at each step, considers next closest node
  - Greedy algorithms not always optimal, usually fast
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):    // each edge coming out of u
        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)
Dijkstra Runtime w/ Heap

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

function `dijkstra(G, s):`

```python
def 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 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 problem for Dijkstra
- But negative cycles are even worse!
  - because there is no true shortest path!
Bellman-Ford Algorithm

- Algorithm that handles graphs w/ neg. edge weights
- Similar to Dijkstra’s but more robust
  - Returns same output as Dijkstra’s for any graph w/ only positive edge weights (but runs slower)
  - Returns correct shortest paths for graphs w/ neg. edge weights
- How: not greedy!