Breadth First Search, Dijkstra’s Algorithm for Shortest Paths

Lecture 17
Tuesday, October 31, 2017

Overview

(A) **BFS** is obtained from **BasicSearch** by processing edges using a **queue** data structure.

(B) It processes the vertices in the graph in the order of their shortest distance from the vertex **s** (the start vertex).

As such...

- **DFS** good for exploring graph structure
- **BFS** good for exploring **distances**

Queue Data Structure

A **queue** is a list of elements which supports the operations:

- **enqueue**: Adds an element to the end of the list
- **dequeue**: Removes an element from the front of the list

Elements are extracted in **first-in first-out (FIFO)** order, i.e., elements are picked in the order in which they were inserted.
Algorithm
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

\[
\text{BFS}(s) =
\begin{aligned}
&\text{Mark all vertices as unvisited} \\
&\text{Initialize search tree } T \text{ to be empty} \\
&\text{Mark vertex } s \text{ as visited} \\
&\text{set } Q \text{ to be the empty queue} \\
&\text{enqueue}(Q, s) \\
&\text{while } Q \text{ is nonempty do} \\
&\quad u = \text{dequeue}(Q) \\
&\quad \text{for each vertex } v \in \text{Adj}(u) \\
&\quad \quad \text{if } v \text{ is not visited then} \\
&\quad \quad \quad \text{add edge } (u, v) \text{ to } T \\
&\quad \quad \text{Mark } v \text{ as visited and } \text{enqueue}(v)
\end{aligned}
\]

Proposition
\[
\text{BFS}(s) \text{ runs in } O(n + m) \text{ time.}
\]
Properties of BFS: Undirected Graphs

Theorem

The following properties hold upon termination of BFS(s):

(A) The search tree contains exactly the set of vertices in the connected component of s.

(B) If dist(u) < dist(v) then u is visited before v.

(C) For every vertex u, dist(u) is the length of a shortest path (in terms of number of edges) from s to u.

(D) If u, v are in connected component of s and e = \{u, v\} is an edge of G, then |dist(u) − dist(v)| ≤ 1.

Properties of BFS: Directed Graphs

Theorem

The following properties hold upon termination of BFS(s):

(A) The search tree contains exactly the set of vertices reachable from s.

(B) If dist(u) < dist(v) then u is visited before v.

(C) For every vertex u, dist(u) is indeed the length of shortest path from s to u.

(D) If u is reachable from s and e = (u, v) is an edge of G, then dist(v) − dist(u) ≤ 1.

Not necessarily the case that dist(u) − dist(v) ≤ 1.

BFS with Layers

BFSLayers(s):

Mark all vertices as unvisited and initialize T to be empty
Mark s as visited and set L_0 = \{s\}
i = 0
while L_i is not empty do
initialize L_{i+1} to be an empty list
for each u in L_i do
for each edge (u, v) ∈ Adj(u) do
if v is not visited
mark v as visited
add (u, v) to tree T
add v to L_{i+1}
i = i + 1

Running time: O(n + m)

Example
with Layers: Properties

**Proposition**
The following properties hold on termination of \texttt{BFSLayers}(s).

1. \texttt{BFSLayers}(s) outputs a BFS tree.
2. \( L_i \) is the set of vertices at distance exactly \( i \) from \( s \).
3. If \( G \) is undirected, each edge \( e = \{u, v\} \) is one of three types:
   - a tree edge between two consecutive layers.
   - a non-tree forward/backward edge between two consecutive layers.
   - a non-tree cross-edge with both \( u, v \) in same layer.
4. \( \implies \) Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.

with Layers: Properties

For directed graphs

**Proposition**
The following properties hold on termination of \texttt{BFSLayers}(s), if \( G \) is directed.

For each edge \( e = (u, v) \) is one of four types:

1. a tree edge between consecutive layers, \( u \in L_i, v \in L_{i+1} \) for some \( i \geq 0 \)
2. a non-tree forward edge between consecutive layers
3. a non-tree backward edge
4. a cross-edge with both \( u, v \) in same layer

**Part II**

Shortest Paths and Dijkstra’s Algorithm
Shortest Path Problems

**Input** A (undirected or directed) graph \( G = (V, E) \) with edge lengths (or costs). For edge \( e = (u, v) \), \( \ell(e) = \ell(u,v) \) is its length.

- Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
- Given node \( s \) find shortest path from \( s \) to all other nodes.
- Find shortest paths for all pairs of nodes.

Many applications!

---

**Single-Source Shortest Paths via BFS**

- Special case: All edge lengths are 1.
  - Run \( \text{BFS}(s) \) to get shortest path distances from \( s \) to all other nodes.
  - \( O(m + n) \) time algorithm.

- Special case: Suppose \( \ell(e) \) is an integer for all \( e \)? Can we use BFS? Reduce to unit edge-length problem by placing \( \ell(e) - 1 \) dummy nodes on \( e \).

- Let \( L = \max_e \ell(e) \). New graph has \( O(mL) \) edges and \( O(mL + n) \) nodes. BFS takes \( O(mL + n) \) time. Not efficient if \( L \) is large.

---

**Non-Negative Edge Lengths**

- Single-Source Shortest Path Problems
  - Input: A (undirected or directed) graph \( G = (V, E) \) with non-negative edge lengths. For edge \( e = (u, v) \), \( \ell(e) = \ell(u,v) \) is its length.
  - Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
  - Given node \( s \) find shortest path from \( s \) to all other nodes.
  - Restrict attention to directed graphs
  - Undirected graph problem can be reduced to directed graph problem - how?
    - Given undirected graph \( G \), create a new directed graph \( G' \) by replacing each edge \( \{u, v\} \) in \( G \) by \( (u, v) \) and \( (v, u) \) in \( G' \).
    - set \( \ell(u, v) = \ell(v, u) = \ell(\{u, v\}) \)
    - Exercise: show reduction works. Relies on non-negativity!

---

**Towards an algorithm**

Why does BFS work?

**Lemma**

Let \( G \) be a directed graph with non-negative edge lengths. Let \( \text{dist}(s,v) \) denote the shortest path length from \( s \) to \( v \).

If \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \) shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):

- \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) is shortest path from \( s \) to \( v_i \)
- \( \text{dist}(s,v_i) \leq \text{dist}(s,v_k) \). Relies on non-neg edge lengths.

**Proof.**

Suppose not. Then for some \( i < k \) there is a path \( P' \) from \( s \) to \( v_i \) of length strictly less than that of \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i \). Then \( P' \) concatenated with \( v_i \rightarrow v_{i+1} \ldots \rightarrow v_k \) contains a strictly shorter path to \( v_k \) than \( s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k \). For the second part, observe that edge lengths are non-negative.
A proof by picture

Shortest path from $v_0$ to $v_6$

A Basic Strategy

Explore vertices in increasing order of distance from $s$:
(For simplicity assume that nodes are at different distances from $s$ and that no edge has zero length)

Initialize for each node $v$, $\text{dist}(s, v) = \infty$
Initialize $X = \{s\}$.
for $i = 2$ to $|V|$ do
  (* Invariant: $X$ contains the $i - 1$ closest nodes to $s$ *)
  Among nodes in $V - X$, find the node $v$ that is the $i^{th}$ closest to $s$
  Update $\text{dist}(s, v)$
  $X = X \cup \{v\}$

How can we implement the step in the for loop?

Finding the $i$th closest node repeatedly

An example

Finding the $i$th closest node

- $X$ contains the $i - 1$ closest nodes to $s$
- Want to find the $i$th closest node from $V - X$

What do we know about the $i$th closest node?

**Claim**

Let $P$ be a shortest path from $s$ to $v$ where $v$ is the $i$th closest node. Then, all intermediate nodes in $P$ belong to $X$.

**Proof.**

If $P$ had an intermediate node $u$ not in $X$ then $u$ will be closer to $s$ than $v$. Implies $v$ is not the $i$'th closest node to $s$ - recall that $X$ already has the $i - 1$ closest nodes.
**Finding the $i$th closest node**

- **X**: Set of $i - 1$ closest nodes to $s$.
- $d'(s, u) = \min_{t \in X} (\text{dist}(s, t) + \ell(t, u))$

If $v$ is an $i$th closest node to $s$, then $d'(s, v) = \text{dist}(s, v)$.

**Proof.**

Let $v$ be the $i$th closest node to $s$. Then there is a shortest path $P$ from $s$ to $v$ that contains only nodes in $X$ as intermediate nodes (see previous claim). Therefore $d'(s, v) = \text{dist}(s, v)$.

**Corollary**

The $i$th closest node is adjacent to $X$.

---

**Finding the $i$th closest node**

- $X$ contains the $i - 1$ closest nodes to $s$.
- Want to find the $i$th closest node from $V - X$.
- For each $u \in V - X$ let $P(s, u, X)$ be a shortest path from $s$ to $u$ using only nodes in $X$ as intermediate vertices.
- Let $d'(s, u)$ be the length of $P(s, u, X)$

Observations: for each $u \in V - X$,
- $\text{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths
- $d'(s, u) = \min_{t \in X} (\text{dist}(s, t) + \ell(t, u))$ - Why?

**Lemma**

If $v$ is the $i$th closest node to $s$, then $d'(s, v) = \text{dist}(s, v)$.

**Proof.**

For every node $u \in V - X$, $\text{dist}(s, u) \leq d'(s, u)$ and for the $i$th closest node $v$, $\text{dist}(s, v) = d'(s, v)$. Moreover, $\text{dist}(s, u) \geq \text{dist}(s, v)$ for each $u \in V - X$. 

---

**Finding the $i$th closest node**

- **X**: Set of $i - 1$ closest nodes to $s$.
- $d'(s, u) = \min_{t \in X} (\text{dist}(s, t) + \ell(t, u))$

If $v$ is an $i$th closest node to $s$, then $d'(s, v) = \text{dist}(s, v)$.

**Proof.**

Let $v$ be the $i$th closest node to $s$. Then there is a shortest path $P$ from $s$ to $v$ that contains only nodes in $X$ as intermediate nodes (see previous claim). Therefore $d'(s, v) = \text{dist}(s, v)$.

**Corollary**

The $i$th closest node to $s$ is the node $v \in V - X$ such that $d'(s, v) = \min_{u \in V - X} d'(s, u)$.

**Proof.**

For every node $u \in V - X$, $\text{dist}(s, u) \leq d'(s, u)$ and for the $i$th closest node $v$, $\text{dist}(s, v) = d'(s, v)$. Moreover, $\text{dist}(s, u) \geq \text{dist}(s, v)$ for each $u \in V - X$. 

Algorithm

Initialize for each node $v$: $\text{dist}(s, v) = \infty$
Initialize $X = \emptyset$, $d'(s, s) = 0$
for $i = 1$ to $|V|$ do
  // $X$ contains the $i - 1$ closest nodes to $s$, and the values of $d'(s, u)$ are current
  Initialize for each node $v$, $\text{dist}(s, v) = d'(s, v) = \infty$
  Initialize $X = \emptyset$, $d'(s, s) = 0$
  for $i = 1$ to $|V|$ do
    // $X$ contains the $i - 1$ closest nodes to $s$, and the values of $d'(s, u)$ are current
    Let $v$ be node realizing $d'(s, v) = \min_{u \in X} d'(s, u)$
    $\text{dist}(s, v) = d'(s, v)$
    $X = X \cup \{v\}$
    Update $d'(s, u)$ for each $u$ in $V - X$ as follows:
    $d'(s, u) = \min\{d'(s, u), \text{dist}(s, v) + \ell(v, u)\}$

Correctness: By induction on $i$ using previous lemmas.
Running time: $O(n \cdot (n + m))$ time.

Improved Algorithm

- Main work is to compute the $d'(s, u)$ values in each iteration
- $d'(s, u)$ changes from iteration $i$ to $i + 1$ only because of the node $v$ that is added to $X$ in iteration $i$

Dijkstra’s Algorithm

- eliminate $d'(s, u)$ and let $\text{dist}(s, u)$ maintain it
- update $\text{dist}$ values after adding $v$ by scanning edges out of $v$

Priority Queues to maintain $\text{dist}$ values for faster running time
- Using heaps and standard priority queues: $O((m + n) \log n)$
- Using Fibonacci heaps: $O(m + n \log n)$.

Example: Dijkstra algorithm in action
**Priority Queues**

Data structure to store a set $S$ of $n$ elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:

- **makePQ**: create an empty queue.
- **findMin**: find the minimum key in $S$.
- **extractMin**: Remove $v \in S$ with smallest key and return it.
- **insert**($v$, $k(v)$): Add new element $v$ with key $k(v)$ to $S$.
- **delete**($v$): Remove element $v$ from $S$.
- **decreaseKey**($v$, $k'(v)$): decrease key of $v$ from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
- **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.

decreaseKey is implemented via delete and insert.

**Implementing Priority Queues via Heaps**

**Using Heaps**

Store elements in a heap based on the key value

- All operations can be done in $O(\log n)$ time

Dijkstra’s algorithm can be implemented in $O((n+m) \log n)$ time.

**Dijkstra’s Algorithm using Priority Queues**

```plaintext
Q ← makePQ()
insert(Q, (s, 0))
for each node $u \neq s$ do
    insert(Q, (u, \infty))
X ← \emptyset
for $i = 1$ to $|V|$ do
    $(v, \text{dist}(s, v)) = \text{extractMin}(Q)$
    $X = X \cup \{v\}$
    for each $u$ in Adj($v$) do
        decreaseKey(Q, (u, \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u))))
```

Priority Queue operations:

- $O(n)$ insert operations
- $O(n)$ extractMin operations
- $O(m)$ decreaseKey operations

**Priority Queues: Fibonacci Heaps/Relaxed Heaps**

**Fibonacci Heaps**

- **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
- **decreaseKey** in $O(1)$ amortized time: $\ell$ decreaseKey operations for $\ell \geq n$ take together $O(\ell)$ time
- Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra’s algorithm)

**Dijkstra’s algorithm can be implemented in** $O(n \log n + m)$ **time. If** $m = \Omega(n \log n)$ **, running time is linear in input size.**

**Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice.** Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)
Shortest Path Tree

Dijkstra’s algorithm finds the shortest path distances from \( s \) to \( V \).

**Question:** How do we find the paths themselves?

---

```plaintext
Q = makePQ()
insert(Q, (s, 0))
prev(s) ← null
for each node \( u ≠ s \) do
    insert(Q, (u, ∞))
    prev(u) ← null
X = ∅
for \( i = 1 \) to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {v}
    for each \( u \) in Adj(v) do
        if \( \text{dist}(s, v) + ℓ(v, u) < \text{dist}(s, u) \) then
            decreaseKey(Q, (u, dist(s, v) + ℓ(v, u)))
            prev(u) = v
```

---

**Lemma**

The edge set \((u, \text{prev}(u))\) is the reverse of a shortest path tree rooted at \( s \). For each \( u \), the reverse of the path from \( u \) to \( s \) in the tree is a shortest path from \( s \) to \( u \).

**Proof Sketch.**

1. The edge set \( \{(u, \text{prev}(u)) \mid u ∈ V\} \) induces a directed in-tree rooted at \( s \) (Why?)
2. Use induction on |\( X \)| to argue that the tree is a shortest path tree for nodes in \( V \).

---

Shortest paths to \( s \)

Dijkstra’s algorithm gives shortest paths from \( s \) to all nodes in \( V \). How do we find shortest paths from all of \( V \) to \( s \)?

- In undirected graphs shortest path from \( s \) to \( u \) is a shortest path from \( u \) to \( s \) so there is no need to distinguish.
- In directed graphs, use Dijkstra’s algorithm in \( G^{rev} \).

---

Shortest paths between sets of nodes

Suppose we are given \( S ⊂ V \) and \( T ⊂ V \). Want to find shortest path from \( S \) to \( T \) defined as:

\[
\text{dist}(S, T) = \min_{s ∈ S, t ∈ T} \text{dist}(s, t)
\]

How do we find \( \text{dist}(S, T) \)?
Example Problem

You want to go from your house to a friend’s house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the “shortest” trip if you include this stop?

Given $G = (V, E)$ and edge lengths $\ell(e), e \in E$. Want to go from $s$ to $t$. A subset $X \subset V$ that corresponds to stores. Want to find $\min_{x \in X} d(s, x) + d(x, t)$.

**Basic solution:** Compute for each $x \in X$, $d(s, x)$ and $d(x, t)$ and take minimum. $2|X|$ shortest path computations. $O(|X|(m + n \log n))$.

**Better solution:** Compute shortest path distances from $s$ to every node $v \in V$ with one Dijkstra. Compute from every node $v \in V$ shortest path distance to $t$ with one Dijkstra.