Chapter 3

Breadth First Search, Dijkstra’s Algorithm for Shortest Paths

3.1 Breadth First Search

3.1.0.1 Breadth First Search (BFS)

Overview

1. BFS is obtained from BasicSearch by processing edges using a data structure called a queue.

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

As such...

(A) DFS good for exploring graph structure
(B) BFS good for exploring distances

3.1.0.2 Queue Data Structure

Queues

A queue is a list of elements which supports the operations:
(A) enqueue: Adds an element to the end of the list
(B) 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.
3.1.0.3 BFS Algorithm

Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

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

Proposition 3.1.1 BFS($s$) runs in $O(n + m)$ time.

3.1.0.4 BFS: An Example in Undirected Graphs


BFS tree is the set of black edges.
3.1.0.5 BFS: An Example in Directed Graphs

3.1.0.6 BFS with Distance

```
BFS(s)
Mark all vertices as unvisited and for each v set dist(v) = ∞
Initialize search tree T to be empty
Mark vertex s as visited and set dist(s) = 0
set Q to be the empty queue
enq(s)
while Q is nonempty do
    u = deq(Q)
    for each vertex v ∈ Adj(u) do
        if v is not visited do
            add edge (u, v) to T
            Mark v as visited, enq(v)
and set dist(v) = dist(u) + 1
```

3.1.0.7 Properties of BFS: Undirected Graphs

**Proposition 3.1.2** The following properties hold upon termination of BFS(s)

1. The search tree contains exactly the set of vertices in the connected component of s.
2. If dist(u) < dist(v) then u is visited before v.
3. For every vertex u, dist(u) is indeed the length of shortest path from s to u.
4. If u, v are in connected component of s and e = {u, v} is an edge of G, then either e is an edge in the search tree, or |dist(u) − dist(v)| ≤ 1.

**Proof**: Exercise.
3.1.0.8 Properties of BFS: Directed Graphs

Proposition 3.1.3 The following properties hold upon termination of BFS(s):

1. The search tree contains exactly the set of vertices reachable from s
2. If dist(u) < dist(v) then u is visited before v
3. For every vertex u, dist(u) is indeed the length of shortest path from s to u
4. If u is reachable from s and e = (u, v) is an edge of G, then either e is an edge in the search tree, or dist(v) − dist(u) ≤ 1. Not necessarily the case that dist(u) − dist(v) ≤ 1.

Proof: Exercise.

3.1.0.9 BFS with Layers

\textbf{BFSLayers}(s):
Mark all vertices as unvisited and initialize T to be empty
Mark s as visited and set $L_0 = \{s\}$
\begin{algorithmic}
  \State \textbf{while} \hspace{1em} $L_i$ is not empty \hspace{1em} \textbf{do}
  \State \hspace{1em} \textbf{initialize} $L_{i+1}$ to be an empty list
  \State \hspace{1em} \textbf{for} \hspace{1em} each $u$ in $L_i$ \hspace{1em} \textbf{do}
  \State \hspace{2em} \textbf{for} \hspace{1em} each edge $(u, v) \in \text{Adj}(u)$ \hspace{1em} \textbf{do}
  \State \hspace{3em} \textbf{if} \hspace{1em} $v$ is not visited
  \State \hspace{4em} mark $v$ as visited
  \State \hspace{4em} add $(u, v)$ to tree $T$
  \State \hspace{4em} add $v$ to $L_{i+1}$
  \State \hspace{1em} $i = i + 1$
\end{algorithmic}

Running time: $O(n + m)$

3.1.0.10 Example

3.1.0.11 BFS with Layers: Properties

Proposition 3.1.4 The following properties hold on termination of BFSLayers(s).

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

### 3.1.1 BFS with Layers: Properties

#### 3.1.1.1 For directed graphs

**Proposition 3.1.5** The following properties hold on termination of \( \text{BFS Layers}(s) \), if \( G \) is directed.

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

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

### 3.2 Bipartite Graphs and an application of BFS

#### 3.2.0.2 Bipartite Graphs

**Definition 3.2.1 (Bipartite Graph)** Undirected graph \( G = (V, E) \) is a **bipartite graph** if \( V \) can be partitioned into \( X \) and \( Y \) s.t. all edges in \( E \) are between \( X \) and \( Y \).

![Bipartite Graph](image)

#### 3.2.0.3 Bipartite Graph Characterization

**Question**

When is a graph bipartite?

**Proposition 3.2.2** Every tree is a bipartite graph.

**Proof**: Root tree \( T \) at some node \( r \). Let \( L_i \) be all nodes at level \( i \), that is, \( L_i \) is all nodes at distance \( i \) from root \( r \). Now define \( X \) to be all nodes at even levels and \( Y \) to be all nodes at odd level. Only edges in \( T \) are between levels.

**Proposition 3.2.3** An odd length cycle is not bipartite.
3.2.0.4 Odd Cycles are not Bipartite

Proposition 3.2.4 An odd length cycle is not bipartite.

Proof: Let \( C = u_1, u_2, \ldots, u_{2k+1}, u_1 \) be an odd cycle. Suppose \( C \) is a bipartite graph and let \( X, Y \) be the partition. Without loss of generality \( u_1 \in X \). Implies \( u_2 \in Y \). Implies \( u_3 \in X \). Inductively, \( u_i \in X \) if \( i \) is odd \( u_i \in Y \) if \( i \) is even. But \( \{u_1, u_{2k+1}\} \) is an edge and both belong to \( X \!

3.2.0.5 Subgraphs

Definition 3.2.5 Given a graph \( G = (V, E) \) a subgraph of \( G \) is another graph \( H = (V', E') \) where \( V' \subseteq V \) and \( E' \subseteq E \).

Proposition 3.2.6 If \( G \) is bipartite then any subgraph \( H \) of \( G \) is also bipartite.

Proposition 3.2.7 A graph \( G \) is not bipartite if \( G \) has an odd cycle \( C \) as a subgraph.

Proof: If \( G \) is bipartite then since \( C \) is a subgraph, \( C \) is also bipartite (by above proposition). However, \( C \) is not bipartite!

3.2.0.6 Bipartite Graph Characterization

Theorem 3.2.8 A graph \( G \) is bipartite if and only if it has no odd length cycle as subgraph.

Proof: Only If: \( G \) has an odd cycle implies \( G \) is not bipartite.
If: \( G \) has no odd length cycle. Assume without loss of generality that \( G \) is connected.
(A) Pick \( u \) arbitrarily and do \( \text{BFS}(u) \)
(B) \( X = \bigcup_{i \text{ is even}} L_i \) and \( Y = \bigcup_{i \text{ is odd}} L_i \)
(C) Claim: \( X \) and \( Y \) is a valid partition if \( G \) has no odd length cycle.

3.2.0.7 Proof of Claim

Claim 3.2.9 In \( \text{BFS}(u) \) if \( a, b \in L_i \) and \( (a, b) \) is an edge then there is an odd length cycle containing \( (a, b) \).

Proof: Let \( v \) be least common ancestor of \( a, b \) in \( \text{BFS} \) tree \( T \).
\( v \) is in some level \( j < i \) (could be \( u \) itself).
Path from \( v \leadsto a \) in \( T \) is of length \( j - i \).
Path from \( v \leadsto b \) in \( T \) is of length \( j - i \).
These two paths plus \( (a, b) \) forms an odd cycle of length \( 2(j - i) + 1 \).
3.2.0.8 Another tidbit

Corollary 3.2.10 There is an \( O(n + m) \) time algorithm to check if \( G \) is bipartite and output an odd cycle if it is not.

3.3 Shortest Paths and Dijkstra’s Algorithm

3.3.0.9 Shortest Path Problems

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.

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

Many applications!

3.3.1 Single-Source Shortest Paths:

3.3.1.1 Non-Negative Edge Lengths

Single-Source Shortest Path Problems

(A) 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.
(B) Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
(C) Given node \( s \) find shortest path from \( s \) to all other nodes.

(A) Restrict attention to directed graphs
(B) Undirected graph problem can be reduced to directed graph problem - how?
   (A) 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 \( G' \).
   (B) set \( \ell(u, v) = \ell(v, u) = \ell(\{u, v\}) \)
   (C) Exercise: show reduction works

3.3.1.2 Single-Source Shortest Paths via BFS

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

Special case: Suppose \( \ell(e) \) is an integer for all \( e \)?
Can we use \( \text{BFS} \)? 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.

### 3.3.1.3 Towards an algorithm

Why does BFS work?

BFS(s) explores nodes in increasing distance from \( s \)

**Lemma 3.3.1** 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 \( s = v_0 \to v_1 \to v_2 \to \ldots \to v_k \) is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):

(A) \( s = v_0 \to v_1 \to v_2 \to \ldots \to v_i \) is a shortest path from \( s \) to \( v_i \)

(B) \( \text{dist}(s, v_i) \leq \text{dist}(s, v_k) \).

**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 \( s = v_0 \to v_1 \to \ldots \to v_i \). Then \( P' \) concatenated with \( v_i \to v_{i+1} \ldots \to v_k \) contains a strictly shorter path to \( v_k \) than \( s = v_0 \to v_1 \ldots \to v_k \).

### 3.3.1.4 A proof by picture

![Shortest path from v0 to v6](image1)

![Shorter path from v0 to v4](image2)

### 3.3.1.5 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 \( S = \emptyset \),

\[
\text{for } i = 1 \text{ to } |V| \text{ do}
\]

(* Invariant: \( S \) contains the \( i - 1 \) closest nodes to \( s \) *)
Among nodes in \( V \setminus S \), find the node \( v \) that is the \( i \)th closest to \( s \)
Update \( \text{dist}(s; v) \)
\( S = S \cup \{v\} \)

How can we implement the step in the for loop?

3.3.1.6 Finding the \( i \)th closest node

(A) \( S \) contains the \( i - 1 \) closest nodes to \( s \)
(B) Want to find the \( i \)th closest node from \( V \setminus S \).

What do we know about the \( i \)th closest node?

Claim 3.3.2 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 \( S \).

Proof: If \( P \) had an intermediate node \( u \) not in \( S \) then \( u \) will be closer to \( s \) than \( v \). Implies \( v \) is not the \( i \)th closest node to \( s \) - recall that \( S \) already has the \( i - 1 \) closest nodes.

3.3.2 Finding the \( i \)th closest node repeatedly

3.3.2.1 An example
3.3.2.2 Finding the $i$th closest node

Corollary 3.3.3 The $i$th closest node is adjacent to $S$.

3.3.2.3 Finding the $i$th closest node

(A) $S$ contains the $i-1$ closest nodes to $s$

(B) Want to find the $i$th closest node from $V - S$.

(A) For each $u \in V - S$ let $P(s, u, S)$ be a shortest path from $s$ to $u$ using only nodes in $S$ as intermediate vertices.

(B) Let $d'(s, u)$ be the length of $P(s, u, S)$

Observations: for each $u \in V - S$,

(A) $\text{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths

(B) $d'(s, u) = \min_{a \in S}(\text{dist}(s, a) + \ell(a, u))$ - Why?

Lemma 3.3.4 If $v$ is the $i$th closest node to $s$, then $d'(s, v) = \text{dist}(s, v)$.

3.3.2.4 Finding the $i$th closest node

Lemma 3.3.5 Given:

(A) $S$: Set of $i-1$ closest nodes to $s$.

(B) $d'(s, u) = \min_{x \in S}(\text{dist}(s, x) + \ell(x, u))$

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 $S$ as intermediate nodes (see previous claim). Therefore $d'(s, v) = \text{dist}(s, v)$.

3.3.2.5 Finding the $i$th closest node

Lemma 3.3.6 If $v$ is an $i$th closest node to $s$, then $d'(s, v) = \text{dist}(s, v)$.

Corollary 3.3.7 The $i$th closest node to $s$ is the node $v \in V - S$ such that $d'(s, v) = \min_{u \in V - S}d'(s, u)$.
Proof: For every node \( u \in V - S \), \( \text{dist}(s, u) \leq d'(s, u) \) 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 \( u \in V - S \).

3.3.2.6 Algorithm

```
Initialize for each node \( v \): \( \text{dist}(s, v) = \infty \)
Initialize \( S = \emptyset \), \( d'(s, s) = 0 \)
for \( i = 1 \) to \( |V| \) do
    (* Invariant: \( S \) contains the \( i-1 \) closest nodes to \( s \) *)
    (* Invariant: \( d'(s, u) \) is shortest path distance from \( u \) to \( s 
    \) using only \( S \) as intermediate nodes*)
    Let \( v \) be such that \( d'(s, v) = \min_{u \in V - S} d'(s, u) \)
    \( \text{dist}(s, v) = d'(s, v) \)
    \( S = S \cup \{v\} \)
    for each node \( u \) in \( V \setminus S \) do
        \( d'(s, u) \leftarrow \min_{a \in S} \left( \text{dist}(s, a) + \ell(a, u) \right) \)
```

Correctness: By induction on \( i \) using previous lemmas.

Running time: \( O(n \cdot (n + m)) \) time.

(A) \( n \) outer iterations. In each iteration, \( d'(s, u) \) for each \( u \) by scanning all edges out of nodes in \( S \); \( O(m + n) \) time/iteration.

3.3.2.7 Example

3.3.2.8 Improved Algorithm

(A) Main work is to compute the \( d'(s, u) \) values in each iteration

(B) \( d'(s, u) \) changes from iteration \( i \) to \( i + 1 \) only because of the node \( v \) that is added to \( S \) in iteration \( i \).
Initialize for each node $v$, $\text{dist}(s, v) = d'(s, v) = \infty$
Initialize $S = \emptyset$, $d'(s, s) = 0$

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

Running time: $O(m + n^2)$ time.

(A) $n$ outer iterations and in each iteration following steps
(B) updating $d'(s, u)$ after $v$ added takes $O(\deg(v))$ time so total work is $O(m)$ since a node enters $S$ only once
(C) Finding $v$ from $d'(s, u)$ values is $O(n)$ time

3.3.2.9 Dijkstra’s Algorithm

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

Priority Queues to maintain $\text{dist}$ values for faster running time

(A) Using heaps and standard priority queues: $O((m + n) \log n)$
(B) Using Fibonacci heaps: $O(m + n \log n)$.

3.3.3 Priority Queues

3.3.3.1 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:

(A) makePQ: create an empty queue.
(B) findMin: find the minimum key in $S$.
(C) extractMin: Remove $v \in S$ with smallest key and return it.
(D) insert$(v, k(v))$: Add new element $v$ with key $k(v)$ to $S$.
(E) delete$(v)$: Remove element $v$ from $S$.
(F) 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.

3.3.3.2 Dijkstra's Algorithm using Priority Queues

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

Priority Queue operations:
(A) $O(n)$ insert operations
(B) $O(n)$ extractMin operations
(C) $O(m)$ decreaseKey operations

3.3.3.3 Implementing Priority Queues via Heaps

Using Heaps
Store elements in a heap based on the key value
(A) All operations can be done in $O(\log n)$ time

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

3.3.3.4 Priority Queues: Fibonacci Heaps/Relaxed Heaps

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

(A) 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.

(B) 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!)
3.3.3.5 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 $u \neq s$ do
    insert(Q, (u, $\infty$))
    prev(u) = null

S = \emptyset
for $i = 1$ to $|V|$ do
    $(v, dist(s, v)) = extractMin(Q)$
    $S = S \cup \{v\}$
    for each $u$ in Adj(v) do
        if $dist(s, v) + \ell(v, u) < dist(s, u)$ then
            decreaseKey(Q, (u, dist(s, v) + $\ell(v, u)$))
            prev(u) = v
```

3.3.3.6 Shortest Path Tree

**Lemma 3.3.8** The edge set $(u, prev(u))$ 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:** [Proof Sketch.]
(A) The edge set $\{(u, prev(u)) \mid u \in V\}$ induces a directed in-tree rooted at $s$ (Why?)
(B) Use induction on $|S|$ to argue that the tree is a shortest path tree for nodes in $V$.

3.3.3.7 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 $s$?

(A) In undirected graphs shortest path from $s$ to $u$ is a shortest path from $u$ to $s$ so there is no need to distinguish.

(B) In directed graphs, use Dijkstra’s algorithm in $G^{rev}$!