Breadth First Search (BFS)

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

Queues

- **queue**: 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.
**BFS Algorithm**

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

- **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 $v \in \text{Adj}(u)$ do
      - If $v$ is not visited then
        - Add edge $(u, v)$ to $T$
        - Mark $v$ as visited and $\text{enq}(v)$

**Proposition**

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

---

**BFS: An Example in Undirected Graphs**

BFS tree is the set of black edges.

1. $[1]$
2. $[2,3]$
3. $[3,4,5]$
4. $[4,5,7,8]$
5. $[5,7,8]$
6. $[7,8,6]$
7. $[8,6]$
8. $[6]$
9. [ ]

---

**BFS: An Example in Directed Graphs**

**BFS with Distance**

- **BFS($s$)**
  - Mark all vertices as unvisited and for each $v$ set $\text{dist}(v) = \infty$
  - Initialize search tree $T$ to be empty
  - Mark vertex $s$ as visited and set $\text{dist}(s) = 0$
  - Set $Q$ to be the empty queue
  - $\text{enq}(s)$
  - While $Q$ is nonempty do
    - $u = \text{deq}(Q)$
    - For each vertex $v \in \text{Adj}(u)$ do
      - If $v$ is not visited do
        - Add edge $(u, v)$ to $T$
        - Mark $v$ as visited, $\text{enq}(v)$
        - Set $\text{dist}(v) = \text{dist}(u) + 1$
**Properties of BFS: Undirected Graphs**

**Proposition**

The following properties hold upon termination of $\text{BFS}(s)$

1. $V(\text{BFS tree comp.}) = \{s\}$, set vertices in connected component $s$.
2. If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$.
3. $\forall u \in V$, $\text{dist}(u)$ = the length of shortest path from $s$ to $u$.
4. If $u, v$ are connected component of $s$, and $e = uv$ is an edge of $G$, then $e \in \text{BFS tree}$, or $|\text{dist}(u) - \text{dist}(v)| \leq 1$.

**Proof.**

Exercise. □

---

**Properties of BFS: Directed Graphs**

**Proposition**

The following properties hold upon termination of $T \leftarrow \text{BFS}(s)$:

1. For search tree $T$. $V(T) = \{s\}$, set vertices reachable from $s$
2. If $\text{dist}(u) < \text{dist}(v)$ then $u$ is visited before $v$
3. $\forall u \in V(T)$: $\text{dist}(u)$ = length of shortest path from $s$ to $u$
4. If $u$ is reachable from $s$, $e = (u \rightarrow v) \in E(G)$. Then either (i) $e$ is an edge in the search tree, or (ii) $\text{dist}(v) - \text{dist}(u) \leq 1$.

Not necessarily the case that $\text{dist}(u) - \text{dist}(v) \leq 1$.

**Proof.**

Exercise. □

---

**BFS with Layers**

**BFS-Layers**$(s)$:
- 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 $(u, v) \in \text{Adj}(u)$ 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**

![Example Diagram]
BFS with Layers: Properties

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

1. \( \text{BFSLayers}(s) \) 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 \( e = uv \) is one of three types:
   1. tree edge between two consecutive layers
   2. non-tree forward/backward edge between two consecutive layers
   3. non-tree cross-edge with both \( u, v \) in same layer
4. \( \iff \) Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.

Example: Tree/cross/forward (backward) edges

Part II

Bipartite Graphs and an application of BFS
Bipartite Graphs

Definition (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 $Y$.

Odd Cycles are not Bipartite

Proposition
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 and $u_i \in Y$ if $i$ is even. But $\{u_1, u_{2k+1}\}$ is an edge and both belong to $X$.

Subgraphs

Definition
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
An undirected 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!
Bipartite Graph Characterization

**Theorem**
An undirected graph $G$ is bipartite $\iff$ 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.

1. Pick $u$ arbitrarily and do BFS($u$)
2. $X = \bigcup i$ is even $L_i$ and $Y = \bigcup i$ is odd $L_i$
3. **Claim:** $X$ and $Y$ is a valid partition if $G$ has no odd length cycle.

Proof of Claim

**Claim**
In BFS($u$) if $a, b \in L_i$ and $ab \in E(G)$ then there is an odd length cycle containing $ab$.

**Proof.**

Let $v$ be least common ancestor of $a, b$ in 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 $2(j - i) + 1$.

Another tidbit

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

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 $e = (u \rightarrow v)$, $\ell(e) = \ell(u \rightarrow v)$ 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: Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths. For edge $e = (u \rightarrow v)$, $\ell(e) = \ell(u \rightarrow v)$ is its length.
2. Given nodes $s, t$ find shortest path from $s$ to $t$.
3. 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 $(u \rightarrow v)$ and $(v, u)$ in $G'$.
- Set $\ell(u \rightarrow v) = \ell(v, u) = \ell(\{u, v\})$
- Exercise: show reduction works

Special case: All edge lengths are 1.

- Run 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.
Towards an algorithm

Why does BFS work?

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

**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

\[
\begin{align*}
\text{dist}(s, v_0) & \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \\
\text{dist}(s, v_k)
\end{align*}
\]

is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):

1. \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) is a shortest path from \( s \) to \( v_i \)
2. \( \text{dist}(s, v_i) \leq \text{dist}(s, v_k) \).

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

**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 \( S = \emptyset \)
- For \( i = 1 \) to \(|V|\) 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?
Finding the $i$th closest node repeatedly

An example

\[ S \]\n
\begin{align*}
\text{Finding the } i\text{th closest node} & \quad \text{Finding the } i\text{th closest node} \\
\text{Finding the } i\text{th closest node} & \quad \text{Finding the } i\text{th closest node} \\
\text{Finding the } i\text{th closest node} & \quad \text{Finding the } i\text{th closest node}
\end{align*}

- $S$ contains the $i-1$ closest nodes to $s$
- Want to find the $i$th closest node from $V - S$.
- 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.
- Let $d'(s, u)$ be the length of $P(s, u, S)$
- Observations: for each $u \in V - S$,
  - $d'(s, u) \leq d'(s, u)$ since we are constraining the paths
  - $d'(s, u) = \min_{a \in S} (\text{dist}(s, a) + \ell(a, u))$ - Why?

**Lemma**

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

- **Corollary**

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

- **Lemma**

Given:

- $S$: Set of $i-1$ closest nodes to $s$.
- $d'(s, u) = \min_{x \in S} (\text{dist}(s, x) + \ell(x, u))$

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 $d'(s, v) = \text{dist}(s, v)$. 

- **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 $d'(s, v) = \text{dist}(s, v)$. 

- **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 $d'(s, v) = \text{dist}(s, v)$. 

- **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 $d'(s, v) = \text{dist}(s, v)$.
Finding the $i$th closest node

**Lemma**

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

**Corollary**

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 $u \in V - S$, $dist(s, u) \leq d'(s, u)$ and for the $i$th closest node $v$, $dist(s, v) = d'(s, v)$. Moreover, $dist(s, u) \geq dist(s, v)$ for each $u \in V - S$.

---

Candidate algorithm for shortest path

Initialize for each node $v$: $dist(s, v) = \infty$

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 $d'(s, v) = \min_{u \in V - S} d'(s, u)$

$dist(v, v) = d'(s, v)$

$S = S \cup \{v\}$

for each node $u$ in $V \setminus S$ do

$d'(s, u) \leftarrow \min_{u \in S}\left(dist(s, u) + \ell(a, u)\right)$

**Correctness:** By induction on $i$ using previous lemmas.

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

- $n$ 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.

---

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 $S$ in iteration $i$.

Initialize for each node $v$, $dist(s, v) = d'(s, v) = \infty$

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 $d'(s, v) = \min_{u \in V - S} d'(s, u)$

$dist(v, v) = d'(s, v)$

$S = S \cup \{v\}$

Update $d'(s, u)$ for each $u$ in $V - S$ as follows:

$d'(s, u) = \min\left(d'(s, u), \; dist(s, v) + \ell(v, u)\right)$

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

- $n$ outer iterations and in each iteration following steps
- updating $d'(s, u)$ after $v$ added takes $O(\deg(v))$ time so total

Finding $v$ from $d'(s, u)$ values is $O(n)$ time
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 \(v\)

Initialize for each node \(v\), \(\text{dist}(s, v) = \infty\)
Initialize \(S = \{\}\), \(\text{dist}(s, s) = 0\)

for \(i = 1\) to \(|V|\) do
  Let \(v\) be such that \(\text{dist}(s, v) = \min_{u \in V - S} \text{dist}(s, u)\)
  \(S = S \cup \{v\}\)
  for each \(u \in \text{Adj}(v)\) do
    \(\text{dist}(s, u) = \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u))\)

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: \(O(m + n \log n)\)

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:

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

All operations can be performed in \(O(\log n)\) time.
\text{decreaseKey} is implemented via \text{delete} and \text{insert}.

Example: Dijkstra algorithm in action

Dijkstra’s Algorithm using Priority Queues

\[ Q \leftarrow \text{makePQ}() \]
\[ \text{insert}(Q, (s, 0)) \]
for each node \(u \neq s\) do
  \[ \text{insert}(Q, (u, \infty)) \]
\[ S \leftarrow \emptyset \]
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
    \[ \text{decreaseKey}(Q, (u, \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u)))) \]

Priority Queue operations:

- \(O(n)\) \text{insert} operations
- \(O(n)\) \text{extractMin} operations
- \(O(m)\) \text{decreaseKey} operations
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.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps
1. \textbf{extractMin}, \textbf{delete} in \( O(\log n) \) time.
2. \textbf{insert} in \( O(1) \) amortized time.
3. \textbf{decreaseKey} in \( O(1) \) amortized time: \( \ell \) decreaseKey operations for \( \ell \geq n \) take together \( O(\ell) \) time.

Relaxed Heaps: \textbf{decreaseKey} in \( O(1) \) worst case time but at the expense of \textbf{meld} (not necessary for Dijkstra’s algorithm)

1. 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.
2. 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 \( V \).

\textbf{Question:} How do we find the paths themselves?

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

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 \( \{(u, \text{prev}(u)) \mid u \in V\} \) induces a directed in-tree rooted at \( s \) (Why?)
2. Use induction on \( |S| \) 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 \( G^{\text{rev}} \)