Review

Dr. John McDonough

Spoken Language Systems
Saarland University

July 20, 2009
In this lecture, we will review the graph algorithms. We will learn that the algorithms for solving such problems are somewhat more complex than the BFS and DFS discussed in prior lectures. Such problems are still tractable, however, and, for a graph $G = (V, E)$, can be solved in $O(V + E)$ time, provided there are no negative weights.

An important technique for solving such problems is that of relaxation.

**Coverage:** Cormen, Leiserson, and Rivest (1990), Chapter 24.
An ordered pair is denoted as \((a, b)\). The ordered pair \((a, b)\) is *not* the same as \((b, a)\).

The Cartesian product of \(A \times B\) of two sets is the set \(\{(a, b) : a \in A \text{ and } b \in B\}\).

A binary relation \(R\) on two sets \(A\) and \(B\) is a subset of the Cartesian product.

For \((a, b) \in R\) we typically write \(aRb\).

That \(R\) is a binary relation on \(A\) implies \(R\) is a subset of \(A \times A\).

**Example:** “Less than” is a binary relation on the natural numbers given by \(\{(a, b) : a, b \in \mathbb{N} \text{ and } a < b\}\).
A total or linear order $R$ on a set $A$ is a relation whereby for all $a, b \in A$, either $aRb$ or $bRa$.

In other words, every pairing of elements from $A$ can be related by $R$.

For example, $\leq$ is a linear order on the set of natural numbers.

The function “is not a descendant of” is not a linear order on the set of human beings, as there are pairs of men neither of whom is descended from the other.
A binary relation $R \subseteq A \times A$ is reflexive if $aRa$ for all $a \in R$. For example, $=$ and $\leq$ are reflexive, but $<$ is not.

A relation $R$ is symmetric if $aRb$ implies $bRa$ for all $a, b \in A$. For example, $=$ is symmetric, but $<$ and $\leq$ are not.

A relation is transitive if $aRb$ and $bRc$ imply $aRc$. The relations $<$, $\leq$, and $=$ are transitive, but the relation $R = \{(a, b) : a, b \in bN \text{ and } a = b - 1\}$ is not.

A relation that is reflexive, symmetric, and transitive is an equivalence relation. For example, $=$ is an equivalence relation on $\mathbb{N}$, but “$<$” is not.
The sets $A$ and $B$ are disjoint if they have no common elements such that $A \cap B = \emptyset$.

The sets of even and odd natural numbers are disjoint.

A set $S = \{S_i\}$ of nonempty subsets forms a partition of $cS$ if and only if
- the subsets are pairwise disjoint such that $S_i \cap S_j = \emptyset$ for all $i \neq j$,
- the union of all $S_i$ is $S$,

$$S = \bigcup_{s_i \in S} S_i.$$ 

The sets of even and odd natural numbers form a partition of $\mathbb{N}$.
Equivalence Classes

- If $R$ is an equivalence relation on the set $A$, then for $a \in A$, the equivalence class of $a$ is the set $[a] = \{ b \in A : aRb \}$.

- In other words, the equivalence class of $a$ is the set of all elements equivalent to $a$.

**Theorem:** The equivalence classes of any equivalence relation $R$ on a set $A$ form a partition of $A$. Any partition of $A$ determines an equivalence relation on $A$ for which the sets in the partition are the equivalence classes.
Definition: Shortest Path

- Consider a weighted, directed graph $G = (V, E)$ with a set of nodes or vertices $V$, and a set of edges $E$.
- There is also a weight function $w : E \rightarrow \mathbb{R}$ mapping edges to real-valued weights.
- The weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of the constituent edges:
  \[ w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i). \]
- The shortest-path weight from $u$ to $v$ is defined as
  \[ \delta(u, v) \triangleq \begin{cases} \min \{w(p) : u \xrightarrow{p} v\}, & \text{if there is a path from } u \text{ to } v, \\ \infty, & \text{otherwise}. \end{cases} \]
- The shortest-path from vertex $u$ to vertex $v$ is then defined as any path with weight $w(p) = \delta(u, v)$.
Algorithms for determining the shortest path through a graph typically exploit the fact that a given shortest path must contain other shortest paths within it.

This optimality is characterized more precisely in the following lemma.

**Lemma: (Subpaths of shortest paths are shortest paths)**

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$, let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex $v_1$ to vertex $v_k$, and for any $i$ and $j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of $p$ from vertex $v_i$ to vertex $v_j$. Then, $p_{ij}$ is a shortest path from $v_i$ to $v_j$. 
Proof of Optimal Substructure of a Shortest Path

- Decompose path $p$ as $v_1 \overset{p_{1i}}{\rightarrow} v_i \overset{p_{ij}}{\rightarrow} v_j \overset{p_{jk}}{\rightarrow} v_k$, such that $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.
- Now assume that there exists a path $p'_{ij}$ from $v_i$ to $v_j$ with weight $w(p'_{ij}) < w(p_{ij})$.
- Then $v_1 \overset{p_{1i}}{\rightarrow} v_i \overset{p'_{ij}}{\rightarrow} v_j \overset{p_{jk}}{\rightarrow} v_k$ is a path from $v_1$ to $v_k$ whose weight $w(p) = w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$.
- This contradicts the assumption that $p$ is the shortest path from $v_1$ to $v_k$. 
The graphs described in this lecture have real-valued weights on their edges.

The shortest path between $v_0$ and $v_k$ in a graph with only positive weights cannot contain any cycles.

Let $p = \langle v_0, v_2, \ldots, v_k \rangle$ denote the shortest path between $v_0$ and $v_k$.

Let $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ denote a cycle with positive weights such that $v_i = v_j$ and $w(c) > 0$.

This implies the path $p' = \langle v_0, v_2, \ldots, v_i, v_{j+1}, \ldots, v_k \rangle$ has weight $w(p') = w(p) - w(c) < w(p)$, which contradicts the assumption that $p$ is the shortest path from $v_0$ to $v_k$. 
The representation for shortest paths is similar to that previously used for BFS trees.

For a graph $G = (V, E)$, we store for each vertex $v \in V$ a predecessor $\pi[v]$ which is either another vertex or NULL.

Given a vertex for which $\pi[v] \neq$ NULL, the procedure `Print-Path(G, s, v)` can be used to print the shortest path from $s$ to $v$.

Question: Is the path correctly printed?

```python
00 def Print-Path(G, s, v):
01     print(v)
02     pred = \pi[v]
03     while not pred == NULL:
04         print(pred)
05         pred = \pi[pred]
```
Upon termination, a shortest path algorithm will have set the predecessor $\pi[v]$ for each $v \in V$ such that it points towards the prior vertex on the shortest path from $s$ to $v$.

Note that $\pi[v]$ will *not* necessarily point to the predecessor of $v$ on the shortest path from $s$ to $v$ while the algorithm is still running.

Let us define the *predecessor subgraph* $G_\pi(V_\pi, E_\pi)$ as that graph induced by the back pointers $\pi$ of each vertex.

Let us define the set $V_\pi \triangleq \{v \in V : \pi[v] \neq \text{NULL}\} \cup \{s\}$.

The directed edge set $E_\pi$ is the set of edges induced by the $\pi$ values for vertices in $V_\pi$:

$$E_\pi = \{(\pi[v], v) \in E : v \in V - \{s\}\}.$$
A shortest-paths tree rooted at $s$ is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$ such that

- $V'$ is the set of vertices reachable from $s \in G$,
- $G'$ forms a rooted tree with root $s$, and
- for all $v \in V$, the unique simple path from $s$ to $v$ in $G'$ is a shortest path from $s$ to $v$ in $G$. 
Initialization

- During the execution of a shortest-paths algorithm, we maintain for each $v \in V$ an attribute $d[v]$ which is the current estimate of the shortest path distance.
- The attributes $\pi[v]$ and $d[v]$ are initialized as in the algorithm shown below.
- After initialization, $\pi[v] = \text{NULL}$ for all $v \in V$, $d[s] = 0$ and $d[v] = \infty$ for $v \in V - \{s\}$.

```
00 def Initialize-Single-Source(G, s):
01     for v ∈ G:
02         d[v] ← \infty
03         \pi[v] ← NULL
04     d[s] ← 0
```
Relaxation

The process of *relaxing* an edge $u \rightarrow v$ means testing whether the distance from $s$ to $v$ can be reduced by traveling over $u$.

This process is illustrated in the pseudocode given below.

The relaxation procedure may decrease the value of the shortest path estimate $d[v]$ and update $\pi[v]$.

The estimate $d[v]$ can never increase during relaxation, only remain the same or decrease.

```python
def Relax(u, v, w):
    if $d[v] > d[u] + w(u, v)$:
        $d[v] \leftarrow d[u] + w(u, v)$
        $\pi[v] \leftarrow u$
```
Properties of Shortest Paths and Relaxation

1. **Triangle inequality:** For any edge $u \rightarrow v \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

2. **Upper bound property:** It holds that $d[v] \geq \delta(s, v)$ for all $v \in V$, and once $d[v] = \delta(s, v)$, the value of $d[v]$ is never again altered.

3. **No-path property:** If there is no path from $s$ to $v$, then we always have $d[v] = \delta(s, v) = \infty$.

4. **Convergence property:** If $s \leadsto u \rightarrow v$ is the shortest path in $G$ for some $u, v \in V$, and if $d[u] = \delta(s, u)$ at any time prior to relaxing $u \rightarrow v$, then $d[v] = \delta(s, v)$ at all times afterward.

5. **Path relaxation propery:** If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the shortest path from $s = v_0$ to $v_k$, and the edges of $p$ are relaxed in the order $v_0 \rightarrow v_1, v_1 \rightarrow v_2, \ldots, v_{k-1} \rightarrow v_k$, then $d[v_k] = \delta(s, v_k)$.

6. **Predecessor-subgraph property:** Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at $s$. 
Shortest distances are always well defined in dags (directed acyclic graphs), as no negative weight cycles can exist even if there are negative weights on some edges.

For a dag $G = (V, E)$, the shortest paths to all nodes can be found in $O(V + E)$ time.

First the vertices must be topologically sorted.

Thereafter the edges from each node can be relaxed, where the vertices are taken in topological order.

```python
def DAG-Shortest-Paths(G, w, s):
    sorted = Topo-Sort(G)
    Initialize-Single-Source(G, s)
    for u in sorted:
        for v in Adj[u]:
            Relax(u, v, w)
```
Run Time Analysis

- The topological sort of $G$ can be performed in $O(V + E)$ time.
- Thereafter, every vertex must be iterated over in the `for` loop of Line 03.
- The edges in the adjacency list of each vertex $v$ are examined exactly once.
- Hence, the total time spent on the inner `for` loop of Lines 04-05 is $O(V + E)$. 
Dijkstra’s Algorithm

- Dijkstra’s algorithm solves the single-source shortest paths algorithm on a weighted, directed graph \( G = (V, E) \), provided that \( w(u, v) \geq 0 \) for each edge \( u \rightarrow v \in E \).

- The set \( S \) contains vertices whose shortest path distances have already been determined, and \( Q \) is a priority queue.

- The algorithm repeatedly selects the vertex \( u \in V - S \) with the minimum shortest path estimate, whose edges are then relaxed.

```python
def Dijkstra(G, w, s):
    Initialize-Single-Source(G, s)
    S ← ∅
    Q ← V[G]
    while Q ≠ ∅:
        u ← Extract-Min(Q)
        S ← S ∪ {u}
        for u ∈ Adj[u]:
            Relax(u, v, w)
```
Correctness of Dijkstra’s Algorithm

Theorem (correctness of Dijkstra’s algorithm): Dijkstra’s algorithm, when run on a weighted, directed graph $G = (V, E)$ with a non-negative weight function $w : e \in E \rightarrow \mathbb{R}$ and source $s$, terminates with $d[u] = \delta(s, u)$ for all $v \in V$.

Proof: We use the following loop invariant:
At the start of each iteration of the `while` loop of Lines 04–08, $d[v] = \delta(s, v)$ for each vertex $s \in S$. 
Proof of Correctness of Dijkstra’s Algorithm

Assume that $u \in V$ is the first vertex added to $S$ such that $d[u] \neq \delta(s, u)$.

Let us examine the situation of the while loop when $u$ is added to $S$.

Prior to adding $u$ to $S$, there is a path $p$ connected a vertex in $S$, namely $S$, to a vertex in $V - S$, namely $u$.

Let $y$ be the first vertex on this path such that $y \in V - S$, and let $x$ be the predecessor of $y$.

The existence of such a $y \neq u$ implies $d[u] \leq d[y]$, as otherwise $y$ would have been chosen for insertion into $S$ ahead of $u$. 

As shown in the figure, the path $p$ can be decomposed as $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$.

Either subpath $p_1$ or $p_2$ can have no edges.

Firstly, $d[y] = \delta(s, y)$ when $u$ is added to $S$.

This follows from the fact that $d[x] = \delta(s, x)$ when $u$ is added to $S$.

As the edge $x \leadsto y$ was relaxed at the time that $x$ was added to $S$, the claim follows from the convergence property.
Because $y$ occurs before $u$ on the shortest path from $s$ to $u$, and all edges have nonnegative weights, we have $\delta(s, y) \leq \delta(s, u)$ and hence

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]. \tag{1}$$

But both $y$ and $u$ were in $V - S$ when $u$ was chosen for insertion in $S$, hence $d[u] \leq d[y]$.

Hence, both inequalities in (1) are actually equalities, such that

$$d[y] = \delta(s, y) = \delta(s, u) = d[u]. \tag{2}$$

Therefore, $d[u] = \delta(s, u)$, which contradicts our choice of $u$.

We conclude, $d[u] = \delta(s, u)$ when $u$ is inserted in $S$, and this equality was maintained at all later times.
Bellmann-Ford Algorithm

- The *Bellmann-Ford algorithm* determines the shortest path from the source $s$ to each $v \in V$ for a graph $G = (V, E)$ with real-valued weights, which may be negative.

- The algorithm assigns each vertex $v \in V$ its correct shortest path weight, provided there are no cycles with negative weights.

- The algorithm then returns true iff there are no negative weight cycles.

```python
def Bellmann_Ford(G, w, s):
    for i ← 1 to |V[G]| − 1:
        for u → v ∈ E[G]:
            Relax(u, v, w)
    for u → v ∈ E[G]:
        if d[v] > d[u] + w(u, v):
            return False
    return True
```
Lemma (correctness of the Bellmann-Ford algorithm): Let \( G = (V, E) \) be a weighted, directed graph with a source \( s \) and weight function \( w : E \to \mathbb{R} \), and assume that \( G \) contains no cycles with negative weights that are reachable from \( s \). Then, after the \(|V| - 1\) iterations of the \texttt{for} loop in Lines 01–03, it must hold that \( d[v] = \delta(s, v) \) for all vertices \( v \in V \) that are reachable from \( s \).
Proof of Correctness of Bellmann-Ford Algorithm

Consider any vertex \( v \) that is reachable from \( s \), and let 
\[ p = \langle v_0, v_1, \ldots, v_k \rangle, \]
where \( v_0 = s \), and \( v_k = v \), be an acyclic shortest path from \( s \) to \( v \).

Path \( p \) has at most \( |V| - 1 \) edges.

Each of the iterations of the \( \text{for} \) loop in Lines 01–03 relaxes all edges \( e \in E \).

Among the edges relaxed in the \( i \)-th iteration for all \( i = 1, 2, \ldots, k \) is \( v_{i-1} \rightarrow v_i \).

Therefore, by the path-relaxation property it follows

\[
d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v).
\]
Corollary: Let $G = (V, E)$ be a weighted, directed graph with source vertex $s$ and weight function $w : E \to \mathbb{R}$. Then for each vertex $v \in V$, there is a path from $s$ to $v$ iff Bellmann–Ford terminates with $d[v] < \infty$ when it is run on $G$. 
In this lecture, we discussed algorithms for determining the shortest path through a weighted graph. Such problems are still tractable, however, and, for a graph $G = (V, E)$, can be solved in $O(V, E)$ time.

An important technique for solving such problems is that of relaxation, whereby the length of the shortest path is successively approximated.

We considered two single-source shortest path algorithms:

- Dijkstra’s algorithm, which determines the shortest path from the source $s$ to each $v \in V$ for a graph $G = (V, E)$ with positive-valued weights.
- The Bellmann-Ford algorithm, which determines the shortest path from the source $s$ to each $v \in V$ for a graph $G = (V, E)$ with real-valued weights, which may be negative.