Aside: Lambda functions in Python

• Want to sort a list of tuples by value of third element
  ```python
  lst = [(1,2,3),(2,5,8),(1,1,1),(5,2,4),(6,1,2)]
  ```

• Method `sort` and function `sorted` have a key field
  - Can write a function that takes tuple as input, and returns 3rd element
  ```python
def return3rd(tupple):
    return tupple[2]

sorted(lst,key=return3rd)
  ```

• Lambda calculus is a mathematic notation
  - Allows you to specify function without giving it a name
  - Instead of: $f(x) = x^3$ You can write: $\lambda x : x^3$
  ```python
  sorted(lst,key=lambda x: x[2])
  ```
  ```python
  sorted(lst,key=lambda x: sum(x))
  ```
Min and Max

• Min and Max can both support the key argument

```python
lst = [(1,2,3),(2,5,8),(1,1,1),(5,2,4),(6,1,2)]
print(min(lst,key=lambda x: x[3]))
```
sum = 0
for i in range(1000000):
    sum += i

• What is `range` doing?
  + Is it creating a list that is 1,000,000 long?
Iterable

```python
sum = 0
for i in range(1000000):
    sum += i
```

• What is `range` doing?
  + Is it creating a list that is 1,000,000 long?
  + It is creating an iterator
    + Every pass through the loop, `range` computes the next value

• `for` can take any object that can be iterated through
  + list, dictionary, set, string (character at a time)
  + can take an iterator, like `range`

• More accurately, `for` takes an iterator
  + list, dictionary, set, string objects have iterator methods defined for them
    + `__iter__` and `__next__`

• `list` function can convert an iterator into a list
Map and Filter

• **map**(function, iterator)
  + applies function to each element of list, and returns a list of the return values from function
  + actually, it returns an iterator, so that it doesn’t have to build the list
  + function can be a lambda expression
    a = [(1,2,3),(2,5,8),(1,1,1),(5,2,4),(6,1,2)]
    b = map(lambda x: sum(x),a)
    print(list(b))

• **filter**(function, iterator)
  + applies function to each element returned by iterator
  + function returns true/false
  + filter just includes the true ones in its iterator
    a = [(1,2,3),(2,5,8),(1,1,1),(5,2,4),(6,1,2)]
    c = filter(lambda x: x[2] == max(x),a)
    print(list(c))

• Don’t overuse them! Sometimes, a for loop is a better option
Single-Source Shortest Paths (Chapter 24)

• Find shortest path
  - Useful for navigation
    + Each intersection, onramp, exit is a vertex
    + roads between are edges, which have a distance

• Weighted directed graph $G = (V, E)$ with $w : E \rightarrow \mathbb{R}$
  - Weight of a path $p = \langle v_0, v_1, ..., v_k \rangle$ is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

• Shortest-path weight $\delta(u, v)$ from $u$ to $v$
  - Similar to shortest path for breadth-first search (where weights are all 1)

$$\delta(u, v) = \begin{cases} 
\min \{ w(p) : u \xrightarrow{p} v \} & \text{if there is a path from } u \text{ to } v \\
\infty & \text{otherwise}
\end{cases}$$
Variants

• Single-source shortest paths
  - From a particular point to all other points

• Single-destination shortest paths
  - To a particular point from all other points
  - Just transpose the graph

• Single-pair shortest path
  - From $a$ to $b$
  - Seems like it should be easier than single-source
    + But all known algorithms have same worse-case asymptotic running time as best single-course algorithms

• All-pairs shortest paths
  - Known algorithms better than running single-source for each vertex
Optimal Substructure

**Lemma 24.1**
Given a weighted direct graph, if there is a shortest path from \( u \) to \( v \) and it goes through \( x \) and \( w \), the subpath from \( x \) to \( w \) is a shortest path from \( x \) to \( w \).
Overview

⇒ Negative Weights and Cycles
• General Reasoning about Shortest Paths
• Bellman-Ford
• Single-source Shortest Paths in DAG
• Dijkstra’s Algorithm
• Negative weight edges are not a problem by themselves
• But if there is a negative weight cycle
  - Each time through cycle gives a lower weight
  - If -ve cycle can occur from source to a node, shortest path is undefined
  - Weight is $-\infty$

• Example:
  source is $s$
Cycles

• Can shortest path contain a cycle?
  - Path not defined if it can contain a negative-weight cycle
  - Positive weight cycle?
  - 0 weight cycle?

• Can restrict ourselves to paths of at most $|V| - 1$ edges
Algorithms

• Dijkstra:
  - disallows -ve weights

• Bellman-Ford
  - allows -ve weights
  - only produces correct results if no -ve cycles
  - but reports if a -ve cycle was found
**Arbitrage (Question 24-3)**

*Arbitrage* is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \times 2 \times 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent.

Suppose that we are given $n$ currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i, j]$ units of currency $c_j$.

**a.** Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$  

Analyze the running time of your algorithm.
Overview

• Negative Weights and Cycles
⇒ General Reasoning about Shortest Paths
• Bellman-Ford
• Single-source Shortest Paths in DAG
• Dijkstra’s Algorithm
Representing Shortest Paths

- Similar to breadth-first search
  - Each vertex has a predecessor $v.\pi$: a vertex or nil
- Predecessor subgraph $G_\pi = (V_\pi, E_\pi)$
  - $E_\pi = \{(v.\pi, v) \mid v \in V \text{ and } v.\pi \neq \text{NIL}\}$
  - $V_\pi = \{v \mid v \in V \text{ and } v.\pi \neq \text{NIL}\} \cup \{s\}$
Relaxation

- Relaxation: a general strategy used by all single-source algorithms
- Strategy: start with an upper bound and keep revising it when you find a lower cost
  - \(v.d\) upper bound on the weight of a shortest path from source \(s\) to \(v\) (similar to BFS)
  - \(v.\pi\) vertex that leads to vertex \(v\) for \(v.d\)

\[
\text{INITIALIZE-SINGLE-SOURCE}(G, s)
\]

\[
\begin{align*}
1 & \quad \text{for each vertex } v \in G.V \\
2 & \quad v.d = \infty \\
3 & \quad v.\pi = \text{NIL} \\
4 & \quad s.d = 0
\end{align*}
\]
Updating

- If current best guess is that distance from $s$ to $v$ is 9 and there is an edge from $u$ to $v$ of weight 2, and $u.d$ is 5 then we can lower $v.d$ to 7, using vertex $u$ (so set $v.d = 7$)

\[ \text{RELAX}(u, v, w) \]

1. if $v.d > u.d + w(u, v)$
2. $v.d = u.d + w(u, v)$
3. $v.\pi = u$

- In single-source algorithms, $v.d$ only revised by relax function
Properties of Shortest Paths and Relaxation

Needed in proofs of algorithms. Remember $\delta(s, v)$ is truth

- **Triangle Inequality**
  - For any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + w(u, v)$

- **Upper-bound Property**
  - $v.d \geq \delta(s, v)$ for all $v \in V$. Once $v.d$ achieves $\delta(s, v)$, it remains there

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

- **Convergence Property**
  - If $s \leadsto u \rightarrow v$ is a shortest path in $G$ for some $u, v \in V$ and if $u.d = \delta(s, u)$ at any time prior to relaxing edge $(u, v)$ then $v.d = \delta(s, v)$ for all times afterward
• Path-relaxation Property
  - If \( p = \langle v_0, v_1, ..., v_k \rangle \) is a shortest path from \( s = v_0 \) to \( v_k \)
    And we relax the edges of \( p \) in the order \((v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)\)
    then \( v_k.d = \delta(s, v_k) \) after the last relaxation step
  - This holds even if other relaxation steps that occur
  - Remember, if there is a negative cycle, there is no shortest path

• Predecessor-subgraph property
  - Once \( v.d = \delta(s, v) \) for all \( v \in V \), predecessor subgraph is a shortest-paths tree rooted at \( s \)
  - i.e., even though we might not grow predecessor subgraph in an organized way, it will still be a shortest-paths tree
Housekeeping

• Arithmetic with infinity
  - Let $a$ be a real number (so $a \neq \infty$ nor $-\infty$)
  - $a + \infty = \infty + a = \infty$
  - $a - \infty = \infty - a = \infty$

• Graph stored in adjacency-list representation
  - Weights stored in adjacency-list
  - Once we know if edge exists, we know its weight in $O(1)$ time
Overview

• Negative Weights and Cycles
• General Reasoning about Shortest Paths
  ⇒ Bellman-Ford
• Single-source Shortest Paths in DAG
• Dijkstra’s Algorithm
Bellman-Ford Algorithm

- Path-relaxation Property
  If $p = \langle v_0, v_1, ..., v_k \rangle$ is a shortest path from $s = v_0$ to $v_k$, And we relax the edges of $p$ in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ then $v_k.d = \delta(s, v_k)$

- So
  Longest possible path from $s$ is ?? So, if we cycle through all edges ?? Will account for all possible paths

Bellman-Ford($G, w, s$)
1. **Initialize-Single-Source**($G, s$)
2. for $i = 1$ to $|G.V| - 1$
3. for each edge $(u, v) \in G.E$
4. \hspace{1cm} RELAX($u, v, w$)
5. for each edge $(u, v) \in G.E$
6. \hspace{1cm} if $v.d > u.d + w(u, v)$
7. \hspace{1cm} return FALSE
8. return TRUE
How it Works

• Progressively decreases the estimate $v.d$

• After $V-1$ rounds, each vertex will reach its minimum value

• Can work with negative edges
  - If it finds a negative cycle, returns False
  - Knows if there is a negative cycle, if $v.d$ can be further reduced

• Used in Reinforcement Learning
  - To update estimates of what sequence of actions to perform to finish a task
**Time Complexity**

**Bellman-Ford**\((G, w, s)\)

1. **Initialize-Single-Source**\((G, s)\)
2. \textbf{for} \(i = 1\) \textbf{to} \(|G.V| - 1\)
3. \textbf{for} each edge \((u, v) \in G.E\)
4. \textbf{Relax} \((u, v, w)\)
5. \textbf{for} each edge \((u, v) \in G.E\)
6. \textbf{if} \(v.d > u.d + w(u, v)\)
7. \textbf{return} FALSE
8. \textbf{return} TRUE
Time Complexity

Initialization takes $\Theta(V)$
Loop $\Theta(V)$
  Loop $\Theta(E)$
Final check $\Theta(E)$
Overall $\Theta(VE)$

\begin{verbatim}
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i = 1 to $|G.V| - 1$
3     for each edge $(u, v) \in G.E$
4         RELAX(u, v, w)
5     for each edge $(u, v) \in G.E$
6         if $v.d > u.d + w(u, v)$
7             return FALSE
8         return TRUE
\end{verbatim}

© P. Heeman, 2017
Example
Overview

• Negative Weights and Cycles
• General Reasoning about Shortest Paths
• Bellman-Ford

⇒ Single-source Shortest Paths in DAG
• Dijkstra’s Algorithm
Single-source Shortest Paths in DAG

• Bellman-Ford seems to be inefficient
  - Doing a lot of $\Theta(V)$ passes to examine each edge
  - Blindly examines all edges
  - Time $\Theta(VE)$

• What about for a DAG?

• Usefulness:
  - PERT Chart on project management
    + Edges are tasks, nodes are states (what is done/not done)
    + Weights are how long the task takes
    + Finding the critical path: longest path through a DAG
For a DAG can do this in $\Theta(V + E)$ time

- Can be negative edges, but no cycles, so no negative weight cycles
- Start with a topological sort of edges $(u, v)$ means $u$ precedes $v$
- This will allow path relaxation property to be more effective!

If $p = \langle v_0, v_1, ..., v_k \rangle$ is a shortest path from $s=v_0$ to $v_k$,
And we relax the edges of $p$ in the order $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$
then $v_k.d = \delta(s, v_k)$
**DAG-Shortest-Paths** \((G, w, s)\)

1. topologically sort the vertices of \(G\)
2. **INITIALIZE-SINGLE-SOURCE** \((G, s)\)
3. **for** each vertex \(u\), taken in topologically sorted order
   4. **for** each vertex \(v \in G.\text{Adj}[u]\)
   5. **RELAX** \((u, v, w)\)

• **Running Time?**
Proof of Correctness (Theorem 24.5)

• How to we phrase the theorem?

• Proof:
Proof of Correctness (Theorem 24.5)

• How to we phrase the theorem?
  If $G$ is a weighted DAG has vertex $s$ and Dag-Shortest-Path is run on $G, s$. Then $v.d = \delta(s, v)$ and predecessor subgraph $G_\pi$ is shortest paths tree

• Proof:
  **Part 1:** Show that $v.d = \delta(s, v)$ at termination
  If $v$ is not reachable from $s$, then $v.d = \delta(s, v) = \infty$ by no-path property
  If $v$ is reachable from $s$, there is a shortest path $p = \langle v_0, v_1, ..., v_k \rangle$ where $v_o = s$ and $v_k = v$
  Since vertices are processed in topical order, we relax the edges on $p$ in order $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$
  Path relaxation property implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 0, 1, ... k$

  **Part 2:** Predecessor subgraph
  By predecessor subgraph property, $G_\pi$ is a shortest paths tree
Summary So Far

• Weighted directed graph
• Relaxation procedure
  - Basis of all algorithms
• Bellman-Ford
  - Can work with graphs with cycles and negative edges
  - Can detect negative cycles
  - Time: $\Theta(VE)$
• DAG Shortest Path
  - Restricted to DAGS: much faster!
  - Time: $\Theta(V + E)$
Overview

• Negative Weights and Cycles
• General Reasoning about Shortest Paths
• Bellman-Ford
• Single-source Shortest Paths in DAG
  ⇒ Dijkstra’s Algorithm
Dijkstra’s Algorithm

• Weight directed graph with no negative edges
  - Can have 0 weight edges
  - Can have cycles, but no negative weight cycles

• Bread-first search
  - Orders vertices by distance from source
    + since vertices put into a queue
  - What if vertices in queue were ordered by $v.d$?
Dijkstra’s Algorithm

• Weight directed graph with no negative edges
  - Can have 0 weight edges
  - Can have cycles, but no negative weight cycles

• Bread-first search
  - Orders vertices by distance from source
    + since vertices put into a queue
  - What if vertices in queue were ordered by \( v.d \)?
    - Are we guaranteed that the top node in the queue will have \( v.d \) set properly?
Dijkstra($G, w, s$)

1. Initialize-Single-Source($G, s$)
2. $S = \emptyset$
3. $Q = G.V$
4. while $Q \neq \emptyset$
   5. $u = \text{Extract-Min}(Q)$
   6. $S = S \cup \{u\}$
   7. for each vertex $v \in G.Adj[u]$
      8. $\text{Relax}(u, v, w)$

• Where are keys of min priority queue being updated?
  - Lowered or increased?
• How is this like best-first? Where is the frontier?
• What is the role of $S$?
Proof of Correctness

Claim: when we take a vertex \( v \) out of min priority queue (and into \( S \)), it will have its final weight: \( \delta(s, v) \)

Proof by Contradiction:
Let \( u \) be first vertex taken out of min queue (and into \( S \)) in which \( u.d \geq \delta(s, u) \)
\( u \) cannot be \( s \) since \( s.d = 0 \) from initialize, and that is correct.
\( \delta(s, u) \neq \infty \) because no path property tells us that \( u.d \) would be \( \infty \)
Let \( p \) be a shortest path from \( s \) to \( u \).
Prior to adding \( u \) to \( S \), consider 1st vertex
in \( p \) not in \( S \), call it \( y \) (so \( u \) is picked before \( y \))
Let \( x \) be \( y \)’s predecessor in \( p \), so \( x \in S \)
\( x.d = \delta(s, x) \) since \( x \) added to \( S \) before \( u \)
By lemma 24.1, subpath of \( p \) to \( y \) is a shortest path.
When \( x \) is added to \( S \), edge \((x, y)\) is relaxed,
and so \( y.d \) is set to \( \delta(s, y) \) by convergence property
and it is set before \( y \) is taken out of the min queue. So \( y \) is not \( u \)
When $u$ is taken out of the min queue,
   $u$ has minimum weight in the queue

**But** $u$ is chosen over $y$ so $u.d \leq y.d$

But $y.d = \delta(s, y)$ when $x$ is added,
   and remains so, even when $u$ is added

$\delta(s, y) \leq \delta(s, u)$ since $y$ is on $u$’s shortest path
so $u.d \leq \delta(s, u)$

By the upper bound property, $u.d \geq \delta(s, u)$
So $u.d = \delta(s, u)$
Contradiction
Timing

\begin{algorithm}
\textbf{Dijkstra}(G, w, s)
\begin{algorithmic}
\State \textbf{Initialize-Single-Source}(G, s)
\State S = \emptyset
\State Q = G.V
\While{Q \neq \emptyset}
\State u = \textbf{Extract-Min}(Q)
\State S = S \cup \{u\}
\For{each vertex v \in G.Adj[u]}
\State \textbf{Relax}(u, v, w)
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}
Timing

Initialize

iterate through all of the vertices

\[
\text{DIJKSTRA}(G, w, s) \\
1 \text{ INITIALIZE-SINGLE-SOURCE}(G, s) \\
2 \quad S = \emptyset \\
3 \quad Q = G.V \\
4 \quad \textbf{while} Q \neq \emptyset \\
5 \quad \quad u = \text{EXTRACT-MIN}(Q) \\
6 \quad \quad S = S \cup \{u\} \\
7 \quad \textbf{for} \text{ each vertex } v \in G.Adj[u] \\
8 \quad \quad \text{RELAX}(u, v, w)
\]
Timing

Initialize
iterate through all of the vertices $\Theta(V)$

$\text{DIJKSTRA}(G, w, s)$
1 $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$
2 $S = \emptyset$
3 $Q = G.V$
4 while $Q \neq \emptyset$
5 $u = \text{EXTRACT-MIN}(Q)$
6 $S = S \cup \{u\}$
7 for each vertex $v \in G.Adj[u]$
8 $\text{RELAX}(u, v, w)$
Timing

Initialize
iterate through all of the vertices
$\Theta(V)$

Create min priority queue
almost all of the values are $\infty$

$\text{DIJKSTRA}(G, w, s)$

1 $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$
2 $S = \emptyset$
3 $Q = G.V$
4 while $Q \neq \emptyset$
5 \hspace{1em} $u = \text{EXTRACT-MIN}(Q)$
6 \hspace{1em} $S = S \cup \{u\}$
7 \hspace{1em} for each vertex $v \in G.Adj[u]$
8 \hspace{2em} $\text{RELAX}(u, v, w)$
Timing

Initialize
   iterate through all of the vertices
   Θ(V)
Create min priority queue
   almost all of the values are ∞
   Θ(V)

Dijkstra(G, w, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = G.V
4. while Q ≠ ∅
5.     u = Extract-Min(Q)
6.     S = S ∪ {u}
7. for each vertex v ∈ G.Adj[u]
8.     Relax(u, v, w)
Dijkstra\((G, w, s)\)

1. \textbf{INITIALIZE-SINGLE-SOURCE}(\(G, s\))
2. \(S = \emptyset\)
3. \(Q = G.V\)
4. \textbf{while} \(Q \neq \emptyset\)
5. \(u = \text{EXTRACT-MIN}(Q)\)
6. \(S = S \cup \{u\}\)
7. \textbf{for} each vertex \(v \in G.Adj[u]\)
8. \(\text{RELAX}(u, v, w)\)

Initialize
iterate through all of the vertices \(\Theta(V)\)

Create min priority queue
almost all of the values are \(\infty\) \(\Theta(V)\)

Loop through each vertex \(\Theta(V)\)
Extract min \(O(\log V)\)

For each edge (not actually nested in other loop) \(\Theta(E)\)
reorder the keys \(O(\log V)\)

\(O(V \log V) + O(E \log V)\)

If all vertices reachable from source, \(|E| \geq |V|\)
So \(O(E \log V)\)
Greedy Algorithm?
Summary

• All work on weighted directed graph
• Relaxation procedure: Basis of all algorithms
• Bellman-Ford
  + Can work with graphs with cycles and negative edges
  + Can detect negative cycles
  + Time: $\Theta(VE)$
• DAG Shortest Path
  + Restricted to DAGS: much faster!
  + Time: $\Theta(V + E)$
• Dijkstra’s algorithm
  + No negative edges, can have 0 weight edges
  + Can have cycles, but no negative weight cycles
  + Time: $\Theta(E \log V)$ (assuming all vertices are reachable)