Strongly Connected Components, Dijkstra

Lecture19
Topological Sort

\[ \text{TOPOLOGICALSORT}(G): \]
\[ \text{add vertex } s \]
\[ \text{for all vertices } v \neq s \]
\[ \text{add edge } s \rightarrow v \]
\[ \text{status}(v) \leftarrow \text{NEW} \]
\[ \text{TOPOSORTDFS}(s) \]
\[ \text{for } i \leftarrow 1 \text{ to } V \]
\[ S[i] \leftarrow \text{POP} \]
\[ \text{return } S[1..V] \]

\[ \text{TOPOSORTDFS}(v): \]
\[ \text{status}(v) \leftarrow \text{ACTIVE} \]
\[ \text{for each edge } v \rightarrow w \]
\[ \text{if } \text{status}(w) = \text{NEW} \]
\[ \text{PROCESSBACKWARDDFS}(w) \]
\[ \text{else if } \text{status}(w) = \text{ACTIVE} \]
\[ \text{fail gracefully} \]
\[ \text{status}(v) \leftarrow \text{DONE} \]
\[ \text{PUSH}(v) \]
\[ \text{return } \text{TRUE} \]

\[ i \quad j \]

\[ i < j \]
Strong Connectivity

In directed graph vertex $u$ can reach vertex $v$ iff there is a directed path from $u$ to $b$.

$\text{reach}(u) =$ set of vertices $u$ can reach

$u$ and $v$ are strongly connected if $u$ can reach $v$ and $v$ can reach $u$. 
Strong Connectivity, SCC

- Strong connectivity is an equivalence relation
- Equivalence classes are called strongly connected components
- If $G$ has a single strongly connected component: strongly connected
Strong Connectivity, SCC

- Strong connectivity is an equivalence relation
- Equivalence classes are called strongly connected components
- If G has a single strongly connected component: strongly connected
- When is G a DAG?
- No two vertices strongly connected
- Every SCC is a single vertex
Strong Connectivity, SCC

- How to compute SCC of vertex \( u \) in \( O(|V|+|E|) \) time?
  
  \( \text{DFS}(G,u) \) gives us \( \text{Reach}(u) \)
  
  \( \text{DFS}(G^{rev},u) \) gives us all the stuff that can reach \( u \)
  
  Take intersection of both for SCC
Strong Connectivity, SCC

- How to compute SCC of vertex $u$ in $O(|V|+|E|)$ time?

  $\text{DFS}(G,u)$ gives us $\text{Reach}(u)$
  $\text{DFS}(G^{\text{rev}},u)$ gives us all the stuff that can reach $u$
  Take intersection of both for SCC
Strong Connectivity, SCC

• How to compute SCC of vertex u in $O(|V|+|E|)$ time?
  
  $\text{DFS}(G, u)$ gives us $\text{Reach}(u)$
  $\text{DFS}(G^{\text{rev}}, u)$ gives us all the stuff that can reach u
  Take intersection of both for SCC
Strong Connectivity, SCC

- How to compute SCC of vertex $u$ in $O(|V|+|E|)$ time?
  - $\text{DFS}(G,u)$ gives us $\text{Reach}(u)$
  - $\text{DFS}(G^{\text{rev}},u)$ gives us all the stuff that can reach $u$
  - Take intersection of both for SCC
Strong Connectivity, SCC

- How to compute SCC of vertex $u$ in $O(|V|+|E|)$ time?
- Compute $\text{Reach}(u)$ with DFS on $G$ in $O(|V|+|E|)$
- Compute $\text{Reach}^{-1}(u) = \{v: u \text{ is in Reach}(v)\}$ with DFS on reverse graph $G^{rev}$ in $O(|V|+|E|)$
- SCC is the intersection of the two sets (mark vertices that have been visited on the first DFS).
- How to compute all SCC of a graph?
  - Naive: $O(|V||E|)$ time (for every vertex compute its component).
  - Can we do better?
  - Combine all the DFS into one.
For every directed graph $G$, $scc(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
For every directed graph $G$, $scc(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
For every directed graph $G$, $\text{scc}(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
For every directed graph $G$, $\text{scc}(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
For every directed graph G, $\text{scc}(G)$ is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges.

$scc(G)$ ALWAYS A DAG!
We want to find all SCC, namely compute $\text{scc}(G)$ graph in linear time.
What if I try to do it recursively?

Find a sink (or source) component of $\text{sc}(G)$, remove it and recurse.
Can compute all the SCC:

\[
\text{STRONGCOMPONENTS}(G): \\
\begin{align*}
  &count \leftarrow 0 \\
  &\text{while } G \text{ is non-empty} \\
  &\quad count \leftarrow count + 1 \\
  &\quad v \leftarrow \text{any vertex in a sink component of } G \\
  &\quad C \leftarrow \text{ONECOMPONENT}(v, count) \\
  &\quad \text{remove } C \text{ and incoming edges from } G
\end{align*}
\]

How to find a vertex in a sink component?
• What if I try to do it recursively?
• Find a sink (or source) component of scc(G), remove it and recurse.
• Last time for DAGS: first vertex DONE in DFS is a sink!
Finding SCC

• **Claim**: Last vertex DONE is in a source component of \( \text{scc}(G) \).

**DFS\(_\text{ALL}(G)\):**
- for all vertices \( v \)
  - unmark \( v \)
  - \( \text{clock} \leftarrow 0 \)
  - for all vertices \( v \)
    - if \( v \) is unmarked
      - \( \text{clock} \leftarrow \text{DFS}(v, \text{clock}) \)

**DFS\(_v, \text{clock}\):**
- mark \( v \)
- for each edge \( v \rightarrow w \)
  - if \( w \) is unmarked
    - \( \text{clock} \leftarrow \text{DFS}(w, \text{clock}) \)
- \( \text{clock} \leftarrow \text{clock} + 1 \)
- \( \text{finish}(v) \leftarrow \text{clock} \)
- return \( \text{clock} \)

Running time?
Do something for every SCC, will give us something quadratic on worst case (e.g. DAG)
But the vertices are in the correct order!
Finding SCC

• **Claim**: For any edge $v \rightarrow w$ in $G$, if $\text{finish}(v) < \text{finish}(w)$, then $v$ and $w$ are strongly connected in $G$. 
Finding SCC

- SCC in $O(|V|+|E|)$ time (just two DFS one in $G$ and one in reverse!)

\[
\text{KosarajuSharir}(G):
\]

\[
\langle \text{Phase 1: Push in finishing order} \rangle
\]

unmark all vertices
for all vertices $v$
  if $v$ is unmarked
    \[
    \text{clock} \leftarrow \text{RevPushDFS}(v)
    \]

\[
\langle \text{Phase 2: DFS in stack order} \rangle
\]

unmark all vertices
$\text{count} \leftarrow 0$
while the stack is non-empty
  $v \leftarrow \text{Pop}$
  if $v$ is unmarked
    $\text{count} \leftarrow \text{count} + 1$
    \[
    \text{LABELOneDFS}(v, \text{count})
    \]

\[
\text{RevPushDFS}(v):
\]

mark $v$
for each edge $v \rightarrow u$ in $\text{rev}(G)$
  if $u$ is unmarked
    \[
    \text{RevPushDFS}(u)
    \]

\[
\text{LABELOneDFS}(v, \text{count}):
\]

mark $v$
$\text{label}(v) \leftarrow \text{count}$
for each edge $v \rightarrow w$ in $G$
  if $w$ is unmarked
    \[
    \text{LABELOneDFS}(w, \text{count})
    \]
Single Source Shortest Paths
Shortest Paths

- Single source shortest path (one s, all t)
Shortest Paths

- Single source shortest path (one s, all t)
Shortest Paths

Input = directed graph (V,E) with lengths \( w(e) \) on edges

- all \( w(e) \geq 0 \)
- some \( w(e) < 0 \)

Dijkstra only (?!?) works for single source shortest paths when all weights non-negative (not really…)

- Single source shortest path (one s, all t)
- All pairs shortest path (all s, all t)
Can we allow arbitrary negative weights?
No shortest path!
Negative cycles are bad. Assume they don't exist
Shortest Path Trees

If shortest paths are unique they form a tree.

What if they are not unique?
There is a set of shortest paths from s to every vertex that defines a tree.
Every SSSP algorithm

Maintain at every vertex:
• \( \text{dist}(v) \) : the length of the tentative shortest path from \( s \) to \( v \) or \( \infty \) if there is no such path.
• \( \text{pred}(v) \) : the predecessor of \( v \) in the tentative shortest path from \( s \) to \( v \) or NULL if there is no such vertex.
• think of storing the \( \text{dist}(v) \) value on the node.

edge \( u \rightarrow v \) is tense if \( \text{dist}(v) > \text{dist}(u) + w(u \rightarrow v) \)

\[
\begin{align*}
\text{RELAX}(u \rightarrow v): \\
\text{dist}(v) &\leftarrow \text{dist}(u) + w(u \rightarrow v) \\
\text{pred}(v) &\leftarrow u
\end{align*}
\]
Every SSSP algorithm

**INITSSSP**\((s)\):

\[
\begin{align*}
\text{dist}(s) & \leftarrow 0 \\
\text{pred}(s) & \leftarrow \text{NULL} \\
\text{for all vertices } v \neq s & \\
\text{dist}(v) & \leftarrow \infty \\
\text{pred}(v) & \leftarrow \text{NULL}
\end{align*}
\]

If there are no tense edges then for every vertex \(v\), \(\text{dist}(v)\) is shortest path distance.

While some edges is tense, relax it

**RELAX**\((u \rightarrow v)\):

\[
\begin{align*}
\text{dist}(v) & \leftarrow \text{dist}(u) + w(u \rightarrow v) \\
\text{pred}(v) & \leftarrow u
\end{align*}
\]

edge \(u \rightarrow v\) is tense if \(\text{dist}(v) > \text{dist}(u)+w(u \rightarrow v)\)
Every SSSP algorithm

\[
\text{InitSSSP}(s):
\begin{align*}
& \text{dist}(s) \leftarrow 0 \\
& \text{pred}(s) \leftarrow \text{NULL} \\
& \text{for all vertices } v \neq s \\
& \quad \text{dist}(v) \leftarrow \infty \\
& \quad \text{pred}(v) \leftarrow \text{NULL}
\end{align*}
\]

While some edges is tense, relax it

makes no assumption on negative weights.
Does assume no negative cycle (how?).
Every SSSP algorithm

**InitSSSP(s):**

\[\begin{align*}
\text{dist}(s) &\leftarrow 0 \\
\text{pred}(s) &\leftarrow \text{NULL} \\
\text{for all vertices } v \neq s \\
\text{dist}(v) &\leftarrow \infty \\
\text{pred}(v) &\leftarrow \text{NULL}
\end{align*}\]

While some edges is tense, relax it

\[\begin{align*}
dist(s) &= 0 \\
dist(v) &= \infty \\
dist(u) &= \infty
\end{align*}\]
Every SSSP algorithm

**InitSSSP(s):**
- \( \text{dist}(s) \leftarrow 0 \)
- \( \text{pred}(s) \leftarrow \text{NULL} \)
- for all vertices \( v \neq s \)
  - \( \text{dist}(v) \leftarrow \infty \)
  - \( \text{pred}(v) \leftarrow \text{NULL} \)

While some edges is tense, relax it

- \( \text{dist}(s) = 0 \)
- \( \text{dist}(v) = 1 \)
- \( \text{dist}(u) = \infty \)
Every SSSP algorithm

\[
\text{InitSSSP}(s): \\
\quad \text{dist}(s) \leftarrow 0 \\
\quad \text{pred}(s) \leftarrow \text{null} \\
\quad \text{for all vertices } v \neq s \\
\quad \quad \text{dist}(v) \leftarrow \infty \\
\quad \quad \text{pred}(v) \leftarrow \text{null}
\]

While some edges is tense, relax it

\[
\begin{align*}
\text{dist}(s) &= 0 \\
\text{dist}(v) &= 1 \\
\text{dist}(u) &= 4
\end{align*}
\]
Every SSSP algorithm

**InitSSSP(s):**
- \( \text{dist}(s) \leftarrow 0 \)
- \( \text{pred}(s) \leftarrow \text{NULL} \)
- for all vertices \( v \neq s \)
  - \( \text{dist}(v) \leftarrow \infty \)
  - \( \text{pred}(v) \leftarrow \text{NULL} \)

While some edges is tense, relax it

- \( \text{dist}(s) = -1 \)
- \( \text{dist}(v) = 1 \)
- \( \text{dist}(u) = 4 \)
Every SSSP algorithm

\[\text{InitSSSP}(s):
\begin{align*}
\text{dist}(s) &\leftarrow 0 \\
\text{pred}(s) &\leftarrow \text{NULL} \\
\text{for all vertices } v \neq s &\text{ do:} \\
\text{dist}(v) &\leftarrow \infty \\
\text{pred}(v) &\leftarrow \text{NULL}
\end{align*}\]

While some edges is tense, relax it

\[\begin{align*}
\text{dist}(s) &= -1 \\
\text{dist}(v) &= 0 \\
\text{dist}(u) &= 4
\end{align*}\]
Every SSSP algorithm

```python
def InitSSSP(s):
    dist(s) = 0
    pred(s) = NULL
    for all vertices v != s:
        dist(v) = \infty
        pred(v) = NULL
```

While some edges is tense, relax it

Ford ('53)

runs into infinite loop

dist(s) = -1

dist(v) = 0

dist(u) = 3

Some edge always tense!
Every SSSP algorithm

**INITSSSP(s):**
- \( dist(s) \leftarrow 0 \)
- \( pred(s) \leftarrow \text{NULL} \)
- for all vertices \( v \neq s \)
  - \( dist(v) \leftarrow \infty \)
  - \( pred(v) \leftarrow \text{NULL} \)

**GENERICSSSP(s):**
- \( \text{INITSSSP}(s) \)
- put \( s \) in the bag
- while the bag is not empty
  - take \( u \) from the bag
  - for all edges \( u \rightarrow v \)
    - if \( u \rightarrow v \) is tense
      - \( \text{RELAX}(u \rightarrow v) \)
      - put \( v \) in the bag

Without specifying how to find tense edges, not an algorithm weird thing about it: a vertex might be put into bag multiple times
Every SSSP algorithm

**InitSSSP(s):**
- \( \text{dist}(s) \leftarrow 0 \)
- \( \text{pred}(s) \leftarrow \text{NULL} \)
- for all vertices \( v \neq s \)
  - \( \text{dist}(v) \leftarrow \infty \)
  - \( \text{pred}(v) \leftarrow \text{NULL} \)

**GenericSSSP(s):**
- \( \text{InitSSSP}(s) \)
- put \( s \) in the bag
- while the bag is not empty
  - take \( u \) from the bag
  - for all edges \( u \to v \)
    - if \( u \to v \) is tense
      - RELAX(\( u \to v \))
      - put \( v \) in the bag

What data structure? queue, stack? (both give correct algo, but maybe exp time)
Every SSSP algorithm

**InitSSSP(s):**
- \( \text{dist}(s) \leftarrow 0 \)
- \( \text{pred}(s) \leftarrow \text{NULL} \)
- for all vertices \( v \neq s \)
  - \( \text{dist}(v) \leftarrow \infty \)
  - \( \text{pred}(v) \leftarrow \text{NULL} \)

**GenericSSSP(s):**
- **InitSSSP(s)**
- put \( s \) in the bag
- while the bag is not empty
  - take \( u \) from the bag
  - for all edges \( u \rightarrow v \)
    - if \( u \rightarrow v \) is tense
      - \( \text{RELAX}(u \rightarrow v) \)
      - put \( v \) in the bag

Dijkstra: Priority Queue

increasing order of their shortest path distance.

Every vertex is visited exactly once, and when that happens the distance is correct.
Dijkstra

assume I have computed a partial shortest path tree

consider the edges from partial tree to all red vertices
what edge to choose in order to extend the tree?

Claim: this edge is in the tree
assume I have computed a partial shortest path tree

consider the edges from partial tree to all red vertices

what edge to choose in order to extend the tree?

Claim: this edge is in the tree
Dijkstra

assume I have computed a partial shortest path tree

Claim: this edge is in the tree
assume I have computed a partial shortest path tree

Claim: this edge is in the tree

If no negative weights, Dijkstra is greedy!
Dijkstra

a.k.a “Closest first search”

Algorithm:
if all \( w(e) \geq 0 \) then
each node leaves priority queue once
\( \leq 1 \) priority queue operation per edge
\( O(|E|\log V) \)

if there is \( w(e) < 0 \) then
\( O(2^{|V|}) \) time
Every SSSP algorithm

<table>
<thead>
<tr>
<th>InitSSSP(s):</th>
<th>GenericSSSP(s):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{dist}(s) \leftarrow 0 )</td>
<td>( \text{InitSSSP}(s) )</td>
</tr>
<tr>
<td>( \text{pred}(s) \leftarrow \text{Null} )</td>
<td>put ( s ) in the bag</td>
</tr>
<tr>
<td>for all vertices ( v \neq s )</td>
<td>while the bag is not empty</td>
</tr>
<tr>
<td>( \text{dist}(v) \leftarrow \infty )</td>
<td>take ( u ) from the bag</td>
</tr>
<tr>
<td>( \text{pred}(v) \leftarrow \text{Null} )</td>
<td>for all edges ( u \rightarrow v )</td>
</tr>
<tr>
<td></td>
<td>if ( u \rightarrow v ) is tense</td>
</tr>
<tr>
<td></td>
<td>RELAX(( u \rightarrow v ))</td>
</tr>
<tr>
<td></td>
<td>put ( v ) in the bag</td>
</tr>
</tbody>
</table>

Difference between Dijkstra and Generic?