Data Structures and Algorithms

Shortest Path

Chris Brooks

Department of Computer Science
University of San Francisco
6 Given a directed weighted graph $G$ (all weights non-negative) and two vertices $x$ and $y$, find the least-cost path from $x$ to $y$ in $G$.
   △ Undirected graph is a special case of a directed graph, with symmetric edges.

6 Least-cost path may not be the path containing the fewest edges
   △ “shortest path” == “least cost path”
   △ “path containing fewest edges” = “path containing fewest edges”
Shortest path ≠ path containing fewest edges

Shortest Path from A to E?
20-2: Shortest Path Example

- Shortest path ≠ path containing fewest edges

- Shortest Path from A to E:
  - A, B, C, D, E
To find the shortest path from vertex $x$ to vertex $y$, we need (worst case) to find the shortest path from $x$ to all other vertices in the graph.

△ Why?
To find the shortest path from vertex $x$ to vertex $y$, we need (worst case) to find the shortest path from $x$ to all other vertices in the graph.

- To find the shortest path from $x$ to $y$, we need to find the shortest path from $x$ to all nodes on the path from $x$ to $y$.
- Worst case, all nodes will be on the path.
If all edges have unit weight ...
If all edges have unit weight,

We can use Breadth First Search to compute the shortest path

BFS Spanning Tree contains shortest path to each node in the graph

Need to do some more work to create & save BFS spanning tree

When edges have differing weights, this obviously will not work
20-7: *Single Source Shortest Path*

- Divide the vertices into two sets:
  - Vertices whose shortest path from the initial vertex is known
  - Vertices whose shortest path from the initial vertex is not known
- Initially, only the initial vertex is known
- Move vertices one at a time from the unknown set to the known set, until all vertices are known
20-8: Single Source Shortest Path

Start with the vertex A
Known vertices are circled in red

We can now extend the known set by 1 vertex
Why is it safe to add D, with cost 1?
Why is it safe to add D, with cost 1?
Could we do better with a more roundabout path?
Why is it safe to add D, with cost 1?

△ Could we do better with a more roundabout path?
△ No – to get to any other node will cost at least 1
△ No negative edge weights, can’t do better than 1
We can now add another vertex to our known list ...
How do we know that we could not get to B cheaper than by going through D?
How do we know that we could not get to B cheaper than by going through D?

- Costs 1 to get to D
- Costs at least 2 to get anywhere from D
  - Cost at least \((1+2 = 3)\) to get to B through D
20-16: Single Source Shortest Path

Next node we can add ...

- Node | Distance
--------|------
- A     | 0    
- B     | 2    
- C     | 1    
- D     |      
- E     |      
- F     |      
- G     |      

Department of Computer Science — University of San Francisco – p.18/83
(We also could have added E for this step)

Next vertex to add to Known ...
Cost to add F is 8 (through C)
Cost to add G is 5 (through D)
20-19: **Single Source Shortest Path**

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>

Last node ...
We now know the length of the shortest path from $A$ to all other vertices in the graph.
To solve the Shortest Path Problem, we can use Dijkstra's Algorithm as follows:

1. **Keep a table** that contains, for each vertex:
   - Is the distance to that vertex known?
   - What is the best distance we've found so far?

2. **Repeat:**
   - Pick the smallest unknown distance
   - Mark it as known
   - Update the distance of all unknown neighbors of that node

3. **Until all vertices are known**
20-22: Dijkstra’s Algorithm Example
**20-23: Dijkstra’s Algorithm Example**

A B C D E F

![Graph with distances and known nodes]

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>∞</td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>1</td>
</tr>
</tbody>
</table>
20-24: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>
20-25: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>true</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>true</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>
20-27: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>true</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>
20-28: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>true</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>true</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>
After Dijkstra’s algorithm is complete:

- We know the *length* of the shortest path
- We do not know *what* the shortest path is

How can we modify Dijkstra’s algorithm to compute the path?
After Dijkstra’s algorithm is complete:

- We know the *length* of the shortest path
- We do not know *what* the shortest path is

How can we modify Dijkstra’s algorithm to compute the path?

- Store not only the distance, but the immediate parent that led to this distance
20-31: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
20-32: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm Example

Node | Known | Dist | Path
--- | --- | --- | ---
A | true | 0 | |
B | false | 5 | A
C | true | 3 | A
D | false | 4 | C
E | false | ∞ | |
F | false | ∞ | |
G | false | ∞ | |
### Dijkstra’s Algorithm Example

#### Graph

- **Nodes:** A, B, C, D, E, F, G
- **Edges and Weights:**
  - A to B: 4
  - A to C: 5
  - A to D: 3
  - B to D: 2
  - B to E: 1
  - C to D: 3
  - D to E: 5
  - D to G: 1
  - E to D: 1
  - F to C: 2
  - F to D: 5

#### Table

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>true</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>7</td>
<td>D</td>
</tr>
</tbody>
</table>
## Dijkstra’s Algorithm Example

### Graph

![Graph Diagram]

### Table

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>true</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td>false</td>
<td>7</td>
<td>D</td>
</tr>
</tbody>
</table>
20-36: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>true</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>false</td>
<td>8</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>true</td>
<td>7</td>
<td>D</td>
</tr>
</tbody>
</table>
20-37: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>true</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>false</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>8</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>true</td>
<td>7</td>
<td>D</td>
</tr>
</tbody>
</table>
20-38: Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Known</th>
<th>Dist</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>true</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>true</td>
<td>9</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>true</td>
<td>8</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>true</td>
<td>7</td>
<td>D</td>
</tr>
</tbody>
</table>
20-39: Dijkstra’s Algorithm

Given the “path” field, we can construct the shortest path
- Work backward from the end of the path
- Follow the “path” pointers until the start node is reached
- We can use a sentinel value in the “path” field of the initial node, so we know when to stop
void Dijkstra(Edge G[], int s, tableEntry T[]) {
    int i, v;
    Edge e;
    for (i=0; i<G.length; i++) {
        T[i].distance = Integer.MAX_VALUE;
        T[i].path = -1;
        T[i].known = false;
    }
    T[s].distance = 0;
    for (i=0; i < G.length; i++) {
        v = minUnknownVertex(T);
        T[v].known = true;
        for (e = G[v]; e != null; e = e.next) {
            if (T[e.neighbor].distance >
                T[v].distance + e.cost) {
                T[e.neighbor].distance = T[v].distance + e.cost;
                T[e.neighbor].path = v;
            }
        }
    }
}
Calculating minimum distance unknown vertex:

```java
int minUnknownVertex(tableEntry T[]) {
    int i;
    int minVertex = -1;
    int minDistance = Integer.MAX_VALUE;
    for (i=0; i < T.length; i++) {
        if (!T[i].known && (T[i].distance < MinDistance)) {
            minVertex = i;
            minDistance = T[i].distance;
        }
    }
    return minVertex;
}
```
Time for initialization:

```java
for(i=0; i<G.length; i++) {
    T[i].distance = Integer.MAX_VALUE;
    T[i].path = -1;
    T[i].known = false;
}
T[s].distance = 0;
```
Time for initialization:

```java
for (i = 0; i < G.length; i++) {
    T[i].distance = Integer.MAX_VALUE;
    T[i].path = -1;
    T[i].known = false;
}
T[s].distance = 0;
```

$\Theta(V)$
Total time for all calls to minUnknownVertex, and setting $T[v].known = true$ (for all iterations of the loop)

```java
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);          // These two lines
    T[v].known = true;                // -------------------
    for (e = G[v]; e != null; e = e.next) {
        if (T[e.neighbor].distance >
            T[v].distance + e.cost) {
            T[e.neighbor].distance = T[v].distance + e.cost;
            T[e.neighbor].path = v;
        }
    }
}
```
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T); < These two lines
    T[v].known = true; < ---------------
    for (e = G[v]; e != null; e = e.next) {
        if (T[e.neighbor].distance >
            T[v].distance + e.cost) {
            T[e.neighbor].distance = T[v].distance + e.cost;
            T[e.neighbor].path = v;
        }
    }
}

$\Theta(V^2)$
Total # of times the if statement will be executed:

```java
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
    for (e = G[v]; e != null; e = e.next) {
        if (T[e.neighbor].distance >
            T[v].distance + e.cost) {
            T[e.neighbor].distance = T[v].distance + e.cost;
            T[e.neighbor].path = v;
        }
    }
}```
Total # of times the if statement will be executed:

```java
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
    for (e = G[v]; e != null; e = e.next) {
        if (T[e.neighbor].distance > T[v].distance + e.cost) {
            T[e.neighbor].distance = T[v].distance + e.cost;
            T[e.neighbor].path = v;
        }
    }
}
```
Total running time for all iterations of the inner for statement:

```java
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
    for (e = G[v]; e != null; e = e.next) {
        if (T[e.neighbor].distance > T[v].distance + e.cost) {
            T[e.neighbor].distance = T[v].distance + e.cost;
            T[e.neighbor].path = v;
        }
    }
}
```
Total running time for all iterations of the inner for statement:

\[
\text{for (i=0; i < G.length; i++)} \{
\hspace{1em} v = \text{minUnknownVertex}(T);
\hspace{1em} T[v].\text{known} = \text{true};
\hspace{1em} \text{for (e = G[v]; e != null; e = e.next) \{}
\hspace{2em} \text{if (T[e.neighbor].distance >}
\hspace{3em} T[v].distance + e.cost) \{
\hspace{4em} T[e.neighbor].distance = T[v].distance + e.cost;
\hspace{4em} T[e.neighbor].path = v;
\hspace{2em} \}
\hspace{1em} \}\}
\]

\[\Theta(V + E)\]

Why \(\Theta(V + E)\) and not just \(\Theta(E)\)?
Total running time:

- Time for initialization
- Time for executing all calls to $\text{minUnknownVertex}$
- Time for executing all distance / path updates

$$\text{Total running time} = \Theta(V + V^2 + (V + E)) = \Theta(V^2)$$
Can we do better than $\Theta(V^2)$?

For **dense** graphs, we can’t do better
- To ensure that the shortest path to all vertices is computed, need to look at all edges in the graph
- A dense graph can have $\Theta(V^2)$ edges

For **sparse** graphs, we can do better
- Where should we focus our attention?
Can we do better than $\Theta(V^2)$?

For dense graphs, we can’t do better
- To ensure that the shortest path to all vertices is computed, need to look at all edges in the graph
- A dense graph can have $\Theta(V^2)$ edges

For sparse graphs, we can do better
- Where should we focus our attention?
- Finding the unknown vertex with minimum cost!
To improve the running time of Dijkstra:
- Place all of the vertices on a min-heap
  - Key value for min-heap = distance of vertex from initial
- While min-heap is not empty:
  - Pop smallest value off min-heap
  - Update table

Problems with this method?
To improve the running time of Dijkstra:

- Place all of the vertices on a min-heap
  - Key value for min-heap = distance of vertex from initial
- While min-heap is not empty:
  - Pop smallest value off min-heap
  - Update table

Problems with this method?

- When we update the table, we need to rearrange the heap
Store vertices in heap

When we update the table, we need to rearrange the heap

Solution:
- When the cost of a vertex decreases, add a new copy to the heap
Create a new priority queue, add start node

While the queue is not empty:
   △ Remove the vertex $v$ with the smallest distance in the heap
   △ If $v$ is not known
      - Mark $v$ as known
      - For each neighbor $w$ of $v$
         • If $\text{distance}[w] > \text{distance}[v] + \text{cost}(v, w)$
         • Set $\text{distance}[w] = \text{distance}[v] + \text{cost}(v, w)$
         • Add $w$ to priority queue with priority $\text{distance}[w]$
6 Each vertex can be added to the heap once for each incoming edge

6 Size of the heap can then be up to $\Theta(E)$

   - $E$ inserts, on heap that can be up to size $E$
   - $E$ delete-mins, on heap that can be upto to size $E$

6 Total: $\Theta(E \lg E)$
Don’t use priority queue, running time is $\Theta(V^2)$

Do use a priority queue, running time is $\Theta(E \lg E)$

Which is better?
Don’t use priority queue, running time is $\Theta(V^2)$

Do use a priority queue, running time is $\Theta(E \lg E)$

Which is better?
- For dense graphs, ($E \in \Theta(V^2)$), $\Theta(V^2)$ is better
- For sparse graphs ($E \in \Theta(V)$), $\Theta(E \lg E)$ is better
20-60: All-Source Shortest Path

What if we want to find the shortest path from all vertices to all other vertices?
How can we do it?
What if we want to find the shortest path from all vertices to all other vertices?

How can we do it?

△ Run Dijkstra’s Algorithm \( V \) times

△ How long will this take?
What if we want to find the shortest path from all vertices to all other vertices?

How can we do it?

△ Run Dijkstra’s Algorithm \( V \) times
△ How long will this take?
△ \( \Theta(VE \lg E) \) (using priority queue)
  - for sparse graphs, \( \Theta(V^2 \lg V) \)
  - for dense graphs, \( \Theta(V^3 \lg V) \)
△ \( \Theta(V^3) \) (not using a priority queue)
20-63: Floyd’s Algorithm

- Alternate solution to all pairs shortest path
- Yields $\Theta(V^3)$ running time for all graphs
Vertices numbered from 0..n

$k$-path from vertex $v$ to vertex $u$ is a path whose intermediate vertices (other than $v$ and $u$) contain only vertices numbered $k$ or less

0-path is a direct link
Shortest 0-path from 0 to 4: 5
Shortest 1-path from 0 to 4: 4
Shortest 2-path from 0 to 4: 4
Shortest 3-path from 0 to 4: 3
20-66: *k*-path Examples

- Shortest 0-path from 0 to 2: 7
- Shortest 1-path from 0 to 2: 6
- Shortest 2-path from 0 to 2: 6
- Shortest 3-path from 0 to 2: 6
- Shortest 4-path from 0 to 2: 4
20-67: Floyd’s Algorithm

- Shortest \(n\)-path = Shortest path

- Shortest 0-path:
  - \(\infty\) if there is no direct link
  - Cost of the direct link, otherwise
Floyd’s Algorithm

6 Shortest $n$-path = Shortest path

6 Shortest 0-path:
- $\infty$ if there is no direct link
- Cost of the direct link, otherwise

6 If we could use the shortest $k$-path to find the shortest $(k + 1)$ path, we would be set
Floyd’s Algorithm

6 Shortest $k$-path from $v$ to $u$ either goes through vertex $k$, or it does not

6 If not:
   △ Shortest $k$-path = shortest $(k - 1)$-path

6 If so:
   △ Shortest $k$-path = shortest $k - 1$ path from $v$ to $k$, followed by the shortest $k - 1$ path from $k$ to $w$
20-70: Floyd’s Algorithm

If we had the shortest \( k \)-path for all pairs \((v, w)\), we could obtain the shortest \( k + 1 \)-path for all pairs

- For each pair \( v, w \), compare:
  - length of the \( k \)-path from \( v \) to \( w \)
  - length of the \( k \)-path from \( v \) to \( k \) appended to the \( k \)-path from \( k \) to \( w \)

- Set the \( k + 1 \) path from \( v \) to \( w \) to be the minimum of the two paths above
Let $D_k[v, w]$ be the length of the shortest $k$-path from $v$ to $w$.

$D_0[v, w] = \text{cost of arc from } v \text{ to } w$ ($\infty$ if no direct link)

$D_k[v, w] = \text{MIN}(D_{k-1}[v, w], D_{k-1}[v, k] + D_{k-1}[k, w])$

Create $D_0$, use $D_0$ to create $D_1$, use $D_1$ to create $D_2$, and so on – until we have $D_n$
20-72: Floyd’s Algorithm

6 Use a doubly-nested loop to create $D_k$ from $D_{k-1}$
   - Use the same array to store $D_{k-1}$ and $D_k$ – just overwrite with the new values
6 Embed this loop in a loop from 1..k
Floyd’s Algorithm

Floyd(Edge G[], int D[][]) {
    int i, j, k

    Initialize D, D[i][j] = cost from i to j

    for (k=0; k<G.length; k++)
        for (i=0; i<G.length; i++)
            for (j=0; j<G.length; j++)
                if ((D[i][k] != Integer.MAX_VALUE) &&
                    (D[k][j] != Integer.MAX_VALUE) &&
                    (D[i][j] > (D[i][k] + D[k][j])))
                    D[i][j] = D[i][k] + D[k][j]
}

20-73: Floyd’s Algorithm
20-74: Example
\begin{align*}
\text{k} &= 0
\end{align*}
20-76: Example

k = 1

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 &  &  &  &  & \\
1 &  & 6 &  &  & 2 \\
2 & 6 & 3 & 8 &  & \\
3 &  &  & 3 &  & 4 \\
4 &  &  &  & 4 & \\
5 & 1 & 2 & 8 & 4 & 4 \\
\end{array}
\]
20-77: Example

\[ k = 2 \]

\[
\begin{array}{c|ccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 &   &   &   &   &   \\
1 & 6 & 9 & 2 &   &   \\
2 & 6 & 3 & 8 &   &   \\
3 &   &   & 4 &   &   \\
4 & 9 & 3 & 4 &   &   \\
5 & 1 & 2 & 8 & 4 & 4 \\
\end{array}
\]
20-78: Example

```
\begin{array}{c}
\text{0} \\
\text{1} \\
\text{2} \\
\text{3} \\
\text{4} \\
\text{5} \\
\text{6} \\
\end{array}
```

```
$\begin{array}{cccccc}
\text{0} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} \\
\text{0} & & & & & \\
\text{1} & & & & & \\
\text{2} & & & & & \\
\text{3} & & & & & \\
\text{4} & & & & & \\
\text{5} & & & & & \\
\end{array}$
```

$k = 3$
Example

$\begin{array}{c}
\text{k = 4}
\end{array}$
20-80: Example

```
k = 5
```

```
0 1 2 3 4 5
0 3 8 3 5 1
1 6 6 6 2
2 8 6 11 3 7
3 6 11 8 4
4 5 6 3 8 4
5 1 2 7 4 4
```
20-81: Floyd’s Algorithm

- We’ve only calculated the distance of the shortest path, not the path itself.
- We can use a similar strategy to the PATH field for Dijkstra to store the path.
  - We will need a 2-D array to store the paths: $P[i][j] =$ last vertex on shortest path from $i$ to $j$. 

Department of Computer Science — University of San Francisco – p.83/83