CSE 331
Software Design & Implementation

Spring 2021
Section 7 – Dijkstra's algorithm; Model-View-Controller, HW7
Administrivia

• HW6 due today
  – Use a DEBUG flag to dial down an expensive checkRep
  – Revise your ADT with any feedback from HW5-2

• HW7 due one week from today (Thursday).

• Any questions?
Agenda

• Overview of HW7 – “Pathfinder”

• Dijkstra’s algorithm

• Model-View-Controller (MVC) design

• The campus dataset
HW7 – Pathfinder

A program to find the shortest walking routes through campus ca. 2006
  – Network of walkways in campus constitutes a graph!

Homework progresses through 4 steps:
1. Modify your graph ADT to use generic types for node/edge labels
   a. Update HW5 to use the generic graph ADT
   b. Make sure all the HW5 tests pass!
   c. Update HW6 to use the generic graph ADT
   d. Make sure all the HW6 tests pass!

2. Implement Dijkstra’s algorithm
   – Starter code gives a path ADT to store search result:
     pathfinder.datastructures.Path

3. Run tests for your implementation of Dijkstra’s algorithm

4. Complete starter code for the Pathfinder application
Dijkstra’s algorithm

• Named for its inventor, Edsger Dijkstra (1930–2002)
  – Truly one of the “founders” of computer science
  – Just one of his many contributions

• Key idea: Proceed roughly like BFS, factoring in edge weights:
  – Track the path to each node with least-yet-seen cost
  – Shrink a set of pending nodes as they are visited

• A priority queue makes handling weights efficient and convenient
  – Helps track which node to process next

• **Note:** Dijkstra’s algorithm requires all edge weights be nonnegative
  – (Other graph search algorithms can handle negative weights – see Bellman-Ford algorithm)
Priority queue

• A queue-like ADT that reorders elements by associated *priority*
  – Whichever element has the *least* priority dequeues next (not FIFO)
  – Priority of an element traditionally given as a separate integer

• Java provides a standard implementation, `PriorityQueue<E>`
  – Implements the `Queue<E>` interface but has distinct semantics
  – Enqueue (add) with the `add` method
  – Dequeue (remove highest priority) with the `poll` method

• `PriorityQueue<E>` uses comparison order for priority order
  – Default: class `E` implements `Comparable<E>`
  – May configure otherwise with a `Comparator<E>`
Priority queue – example

```java
def PriorityQueue<Double>():
    q = new PriorityQueue<Double>();
    q.add(5.1);
    q.add(4.2);
    q.add(0.3);
    q.poll(); // 0.3
    q.add(0.8);
    q.poll(); // 0.8
    q.add(20.4);
    q.poll(); // 4.2
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>4.2</td>
<td>5.1</td>
</tr>
<tr>
<td>4.2</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>4.2</td>
<td>5.1</td>
</tr>
<tr>
<td>4.2</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>5.1</td>
<td>20.4</td>
</tr>
<tr>
<td>5.1</td>
<td>20.4</td>
<td></td>
</tr>
</tbody>
</table>
Finding the “shortest” path

• HW6 measured the “shortest” path by the number of its edges
  – So really, the path with the fewest edges (i.e., fewest hops)
  – Implemented by breadth-first search (BFS)
  – Edge labels totally irrelevant (aside from our tie-breaking rules)

• In HW7, edge labels are numbers, called weights
  – Labeled graphs like that are called weighted graphs
  – An edge’s weight is considered its cost (think time, distance, price, …)

• HW7 measured the “shortest” path by the total weight of its edges
  – So really, the path with the least cost
  – Find using Dijkstra’s algorithm
  – Edge weights crucially relevant
Dijkstra’s algorithm

• **Main idea:** Start at the source node and find the shortest path to all reachable nodes.
  – This will include the shortest path to your destination!

• What is the shortest path from A to C for the given graph using Dijkstra’s algorithm? Using BFS?

![Graph Diagram]
Dijkstra’s algorithm – pseudocode

active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active

<inv ???> What would be a good invariant for this loop?
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = (minDest, child):
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
    add minDest to finished
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A]</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UW CSE 331 Spring 2021
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, C]</td>
<td>1</td>
</tr>
<tr>
<td>[A, B]</td>
<td>2</td>
</tr>
<tr>
<td>[A, D]</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>≤ 1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>≤ 4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B]</td>
<td>2</td>
</tr>
<tr>
<td>[A, D]</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>≤2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>≤4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>≤ 4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>≤ 12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B]</td>
<td>2</td>
</tr>
<tr>
<td>[A, D]</td>
<td>4</td>
</tr>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

\[\begin{align*}
&[A, B] & 2 \\
&[A, D] & 4 \\
&[A, C, E] & 12
\end{align*}\]
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, D]</td>
<td>4</td>
</tr>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>≤4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, D]</td>
<td>4</td>
</tr>
<tr>
<td>[A, B, F]</td>
<td>4</td>
</tr>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>≤4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>≤12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>≤4</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B, F]</td>
<td>4</td>
</tr>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

node | finished | cost | prev |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

**Priority Queue**

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

 priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B, F, H]</td>
<td>7</td>
</tr>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>≤ 7</td>
<td>F</td>
</tr>
</tbody>
</table>

path cost

[A, B, F, H] 7
[A, C, E] 12
[A, B, E] 12
Dijkstra’s algorithm – paths from A

![Graph with nodes A, B, C, D, E, F, G, H and edges with weights.]

**Priority Queue**

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

**Table**

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B, F, H, G]</td>
<td>8</td>
</tr>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

node | finished | cost | prev |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>≤ 11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, C, E]</td>
<td>12</td>
</tr>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, B, E]</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm – paths from A

Now we know the cost and path to every single node by looking at the table!

<table>
<thead>
<tr>
<th>node</th>
<th>finished</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

priority queue

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now it’s your turn!
Dijkstra’s algorithm – pseudocode

active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active
<inv: All paths found so far are shortest paths>
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = ⟨minDest, child⟩:
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
            add minDest to finished
Dijkstra’s algorithm – pseudocode

active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active
<inv: All paths found so far are shortest paths>
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = (minDest, child):
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
    add minDest to finished
Dijkstra’s algorithm – pseudocode

active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active
<inv: All paths found so far are shortest paths>
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = (minDest, child):
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
    add minDest to finished

All nodes not reached yet are farther away than those reached so far
Dijkstra’s algorithm – pseudocode

active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active
<inv: All paths found so far are shortest paths>
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = (minDest, child):
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
    add minDest to finished
Dijkstra’s algorithm – pseudocode

active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active
<inv: All paths found so far are shortest paths & ... >
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = (minDest, child):
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
    add minDest to finished

Let’s take a moment to think what else is true here?
Dijkstra’s algorithm – pseudocode

**active** = priority queue of paths.

**finished** = empty set of nodes.

add a path from start to itself to active

<inv: All paths found so far are shortest paths & ... >

while active is non-empty:

    **minPath** = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = (minDest, child):
        if child is not in finished:
            newPath = minPath + e
            add newPath to active
    add minDest to finished

It follows from our updated invariant that this path is the shortest path (assuming node is not in finished)
Model-View-Controller

Model-View-Controller (MVC) is a ubiquitous design pattern:

- The **model** abstracts and represents the application’s data.
- The **view** provides a user interface to display the application data.
- The **controller** handles user input to affect the application.
Model-View-Controller: Example

- Accessing my Google Drive files through my laptop and my phone

<table>
<thead>
<tr>
<th>Laptop</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>View:</strong> The screen displays options for me to select files</td>
<td><strong>View:</strong> The screen displays the file</td>
</tr>
<tr>
<td><strong>Control:</strong> Get input selection from mouse/keyboard</td>
<td><strong>Control:</strong> Get input selection from touch sensor</td>
</tr>
<tr>
<td><strong>Control:</strong> Request the selected file from Google Drive</td>
<td><strong>Control:</strong> Request the selected file from Google Drive</td>
</tr>
<tr>
<td><strong>Model:</strong> Google Drive sends back the request file to my device</td>
<td><strong>Model:</strong> Google Drive sends back the request file to my device</td>
</tr>
<tr>
<td><strong>Control:</strong> Receive the file and pass it to View</td>
<td><strong>Control:</strong> Receive the file and pass it to View</td>
</tr>
</tbody>
</table>
HW 7 – Model-View-Controller

• HW7 is an MVC application, with much given as starter code.
  – View: pathfinder.textInterface.TextInterfaceView
  – Controller: pathfinder.textInterface.TextInterfaceController

• You will need to fill out the code in pathfinder.CampusMap.
  – Since your code implements the model functionality
HW7: text-based View-Controller

- **TextInterfaceView**
  - Displays output to users from the result received from **TextInterfaceController**.
  - Receives input from users.
    - Does not process anything; directly pass the input to the **TextInterfaceController** to process.

- **TextInterfaceController**
  - Process the passed input from the **TextInterfaceView**
    - Include talking to the **Model** (the graph & supporting code)
  - Give the processed result back to the **TextInterfaceView** to display to users.

* HW9 will be using the same **Model** but different and more sophisticated View and Controller
Campus dataset

- Two CSV files in src/main/resources/data:
  - campus_buildings.csv – building entrances on campus
  - campus_paths.csv – straight-line walkways on campus

- Exact points on campus identified with (x, y) coordinates
  - Pixels on a map of campus (campus_map.jpg, next to CSV files)
  - Position (0, 0), the origin, is the top left corner of the map

- Parser in starter code: pathfinder.parser.CampusPathsParser
  - CampusBuilding object for each entry of campus_buildings.csv
  - CampusPath object for each entry of campus_paths.csv
Campus dataset – coordinate plane
Campus dataset – sample

- **campus_buildings.CSV** has entries like the following:

<table>
<thead>
<tr>
<th>shortName</th>
<th>longName</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGR</td>
<td>By George,</td>
<td>1671.5499</td>
<td>1258.4333</td>
</tr>
<tr>
<td>MOR</td>
<td>Moore Hall,</td>
<td>2317.1749</td>
<td>1859.502</td>
</tr>
</tbody>
</table>

- **campus_paths.CSV** has entries like the following:

<table>
<thead>
<tr>
<th>x1</th>
<th>y1</th>
<th>x2</th>
<th>y2</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1810.0</td>
<td>431.5</td>
<td>1804.6429</td>
<td>437.92857</td>
<td>17.956615...</td>
</tr>
<tr>
<td>1810.0</td>
<td>431.5</td>
<td>1829.2857</td>
<td>409.35714</td>
<td>60.251364...</td>
</tr>
</tbody>
</table>

- See **campus_routes.jpg** for nice visual rendering of **campus_paths.csv**
Campus dataset – demo

- Your TA will open the starter files of HW 7.
Script testing in HW7

• Extends the test-script mechanism from HW5
  – Using numeric weights instead of string labels on edges
  – New command `FindPath` to find shortest path with Dijkstra’s algorithm
  – No command like `LoadGraph`

• Must write the test driver (`PathfinderTestDriver`) yourself
  – Feel free to copy pieces from `GraphTestDriver` in HW5

<table>
<thead>
<tr>
<th>Command (in <code>foo.test</code>)</th>
<th>Output (in <code>foo.expected</code>)</th>
</tr>
</thead>
</table>
| `FindPath graph node_1 node_n` | `path from node_1 to node_n:`
  `node_1 to node_2 with weight w_{1,2}`
  `node_2 to node_3 with weight w_{2,3}`
  ...
  `node_{n-1} to node_n with weight w_{n-1,n}`
  `total cost: w` |
| ... | ... |