CSE 331
Software Design & Implementation

Autumn 2022

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

Administrivia

- HW6 due today
  - Revise your ADT with any feedback from HW5
  - Use a **`DEBUG`** flag to dial down an expensive **`checkRep`**
  - Set it to **`false`** when you submit!

- HW7 due one week from today (Thursday)
  - Assignment posted on web now, starter code pushed

- IntelliJ: Get the Ultimate Edition if you haven't already
  - We will start moving onto React next week. You will be at a big disadvantage if you are still using the Community Edition

- Any questions?

Agenda

- Overview of HW7
- Dijkstra’s algorithm
- Model-View-Controller (MVC) design
- The campus dataset

HW7 – Overview

- HW7 includes 2 folders:
  - **`hw-tasks/`**
  - **`hw-pathfinder/`**

- When done, attach the tag **`hw7-final`**
  - Reminder: commit/push everything, and **then** create/push the tag in a **<u>separate transaction</u>**!
  - Remember to check **Repository > Graph** on GitLab to verify that your tag is on the correct commit!

HW7 – Tasks

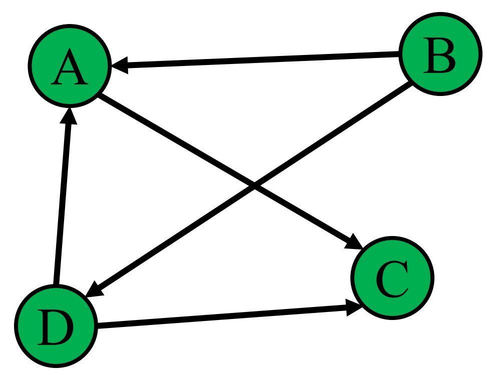
- You will first need to make your graph class **generic** to take other types for node and edge labels that are not Strings.
  a. Update HW5/6 to use the generic graph ADT
  b. Make sure all the HW5/6 tests pass!

- You will need to implement some of **`TaskSorter`**
  – Tasks can be dependent on other tasks (i.e. one needs to be completed before the other)
  – What's a natural way to represent this? A graph!
  – Given a set of tasks and dependencies, can we find an ordering of tasks that satisfies the dependencies?
    - This algorithm is already written for you (we suggest you take a look)

HW7 – Tasks

- Tasks are nodes, dependencies are edges
- Let's take a look at a visual:
  - If X -> Y, task X must be done before task Y.
  - What order can we complete these tasks in?

B -> D -> A -> C



HW7 – Pathfinder

Next part: a program to find the shortest walking routes through campus

- Network of walkways in campus constitutes a graph!

Pathfinder progresses through 3 steps:

1. Implement Dijkstra's algorithm
   - Starter code gives a path ADT to store search result: **`pathfinder.datastructures.Path`**

2. Run tests for your implementation of Dijkstra's algorithm

3. 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: find shortest path based on numeric 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 value 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 **`remove`** 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

Code			
`q = new PriorityQueue<Double>();`			
`q.add(5.1);`	5.1		
`q.add(4.2);`	4.2	5.1	
`q.add(0.3);`	0.3	4.2	5.1
`q.remove(); // 0.3`	4.2	5.1	
`q.add(0.8);`	0.8	4.2	5.1
`q.remove(); // 0.8`	4.2	5.1	
`q.add(20.4);`	4.2	5.1	20.4
`q.remove(); // 4.2`	5.1	20.4	

Finding the "shortest" path

- 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 measures the "shortest" path by the <u>total weight</u> of its edges
  - So really, the path with the <u>least cost</u>
  - Find using *Dijkstra's algorithm*
  - Edge weights crucially relevant

- There are other definitions of "shortest" path that we will not consider

Aside: **`break`** vs. **`continue`**

- **`break`** exits the loop, while **`continue`** skips the rest of this iteration

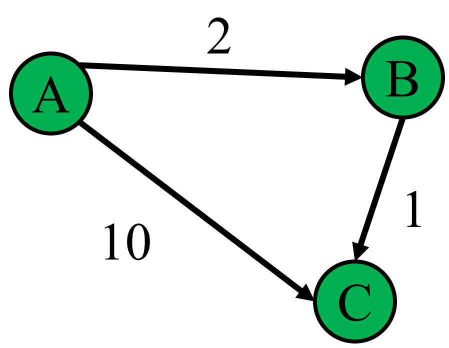
```
for (int i = 0; i < 5; i++) {
  if (i == 3) { break; }
  System.out.println(i + " ");
}
// out: 0 1 2

for (int i = 0; i < 5; i++) {
  if (i == 3) { continue; }
  System.out.println(i + " ");
}
// out: 0 1 2 4
```

Dijkstra’s algorithm

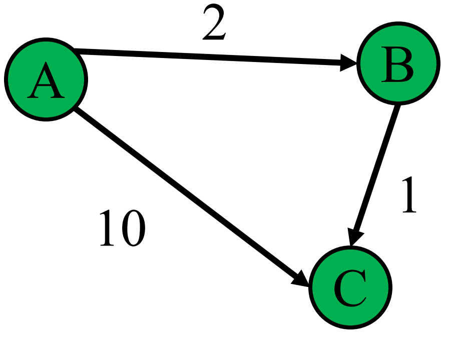
- **Main idea:** Start at the source node and find the shortest path to all reachable nodes.
  - This will include the shortest path to your destination!

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

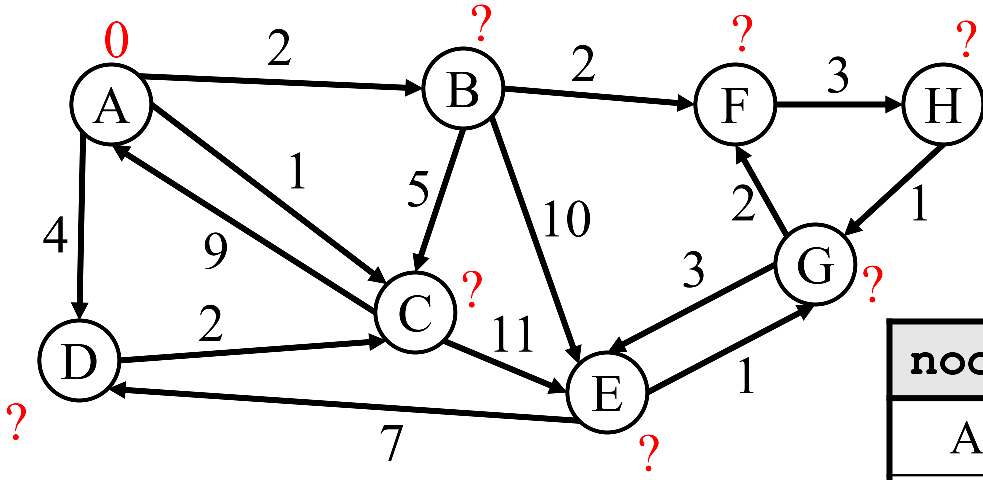


Dijkstra's algorithm – pseudocode

```
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

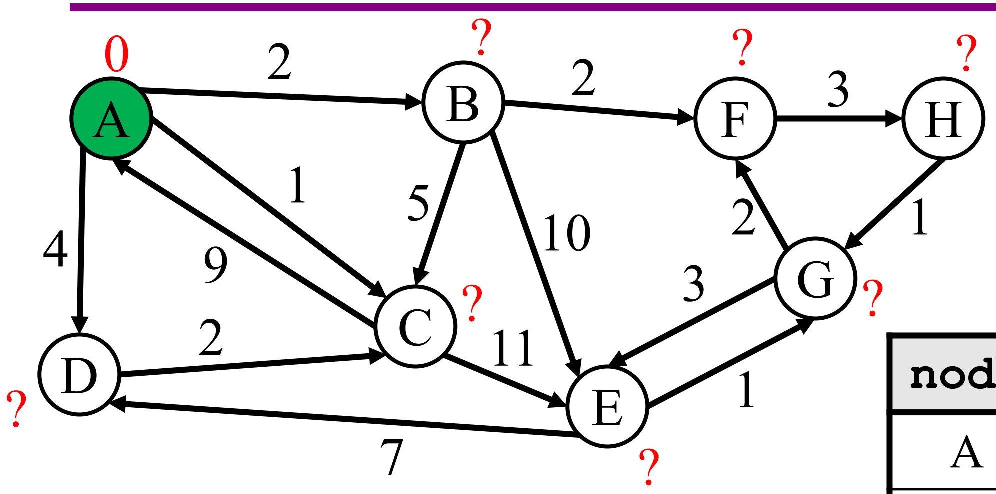


priority queue

path	cost
[A]	0

node	finished	cost	prev
A		0	-
B			
C			
D			
E			
F			
G			
H			

Dijkstra's algorithm – paths from A

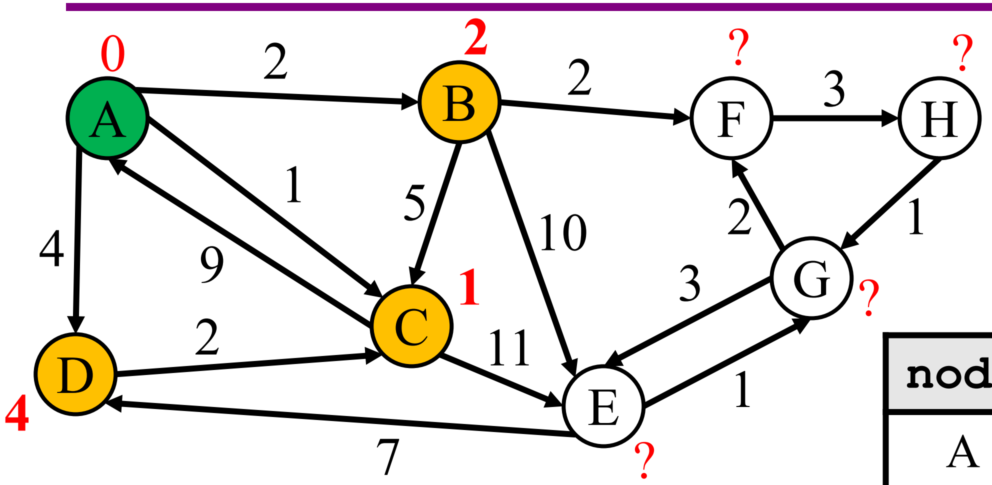


priority queue

path	cost

node	finished	cost	prev
A	Y	0	-
B			
C			
D			
E			
F			
G			
H			

Dijkstra's algorithm – paths from A

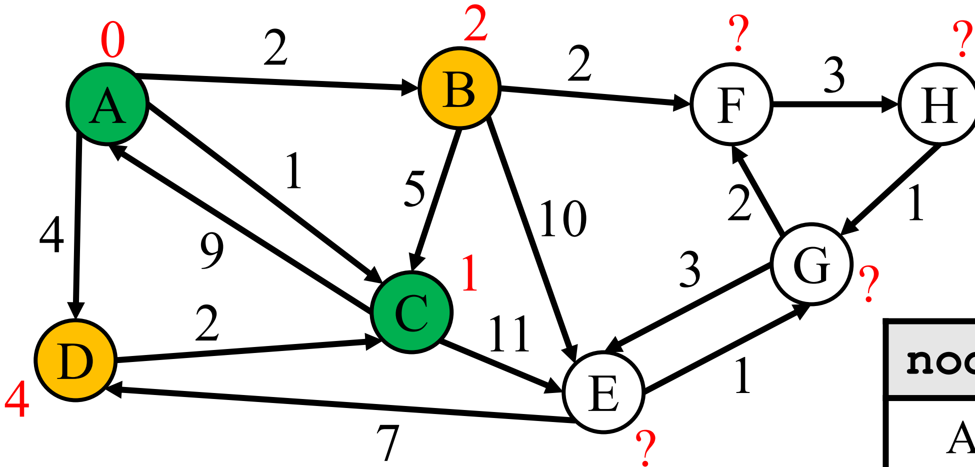


priority queue

path	cost
[A, C]	1
[A, B]	2
[A, D]	4

node	finished	cost	prev
A	Y	0	-
B		≤ **2**	**A**
C		≤ **1**	**A**
D		≤ **4**	**A**
E			
F			
G			
H			

Dijkstra’s algorithm – paths from A

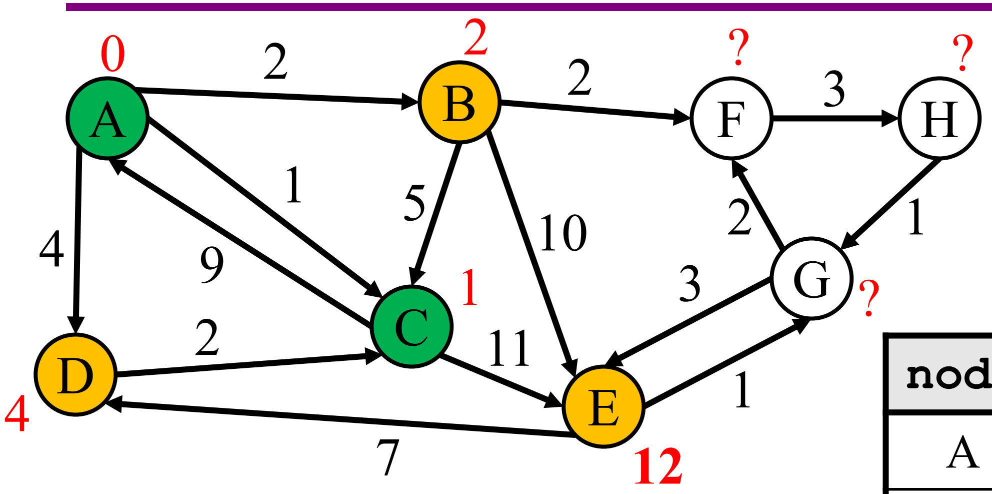


priority queue

path	cost
[A, B]	2
[A, D]	4

node	finished	cost	prev
A	Y	0	-
B		≤ 2	A
C	Y	**1**	A
D		≤ 4	A
E			
F			
G			
H			

Dijkstra's algorithm – paths from A

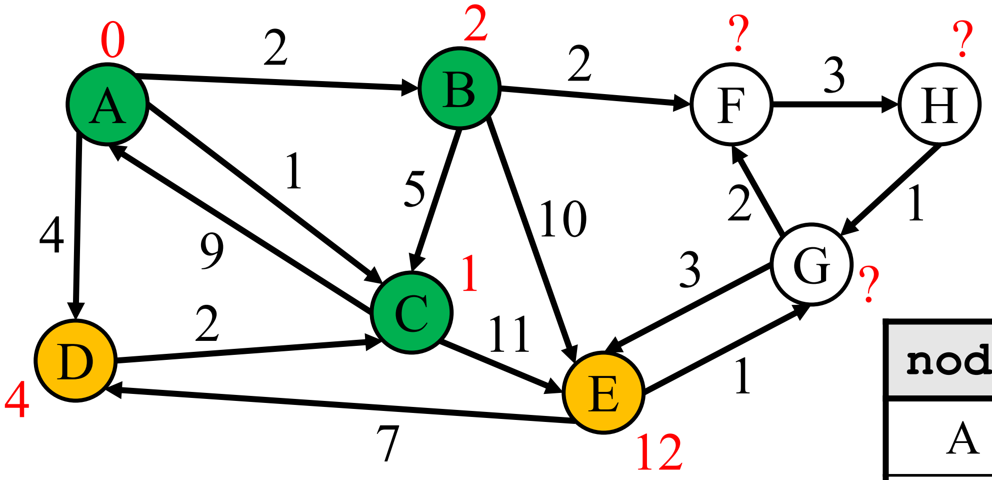


priority queue

path	cost
[A, B]	2
[A, D]	4
[A, C, E]	12

node	finished	cost	prev
A	Y	0	-
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		**≤ 12**	**C**
F			
G			
H			

Dijkstra's algorithm – paths from A

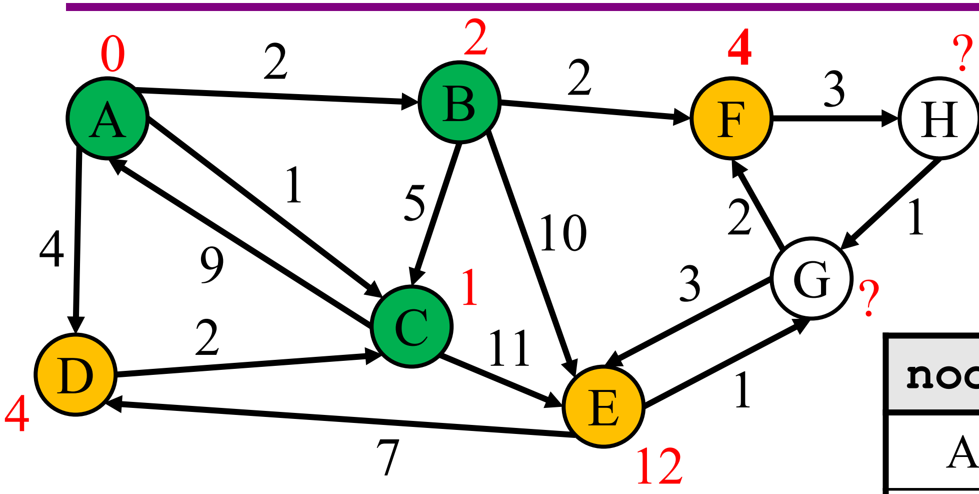


priority queue

path	cost
[A, D]	4
[A, C, E]	12

node	finished	cost	prev
A	Y	0	-
B	**Y**	**2**	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F			
G			
H			

Dijkstra's algorithm – paths from A

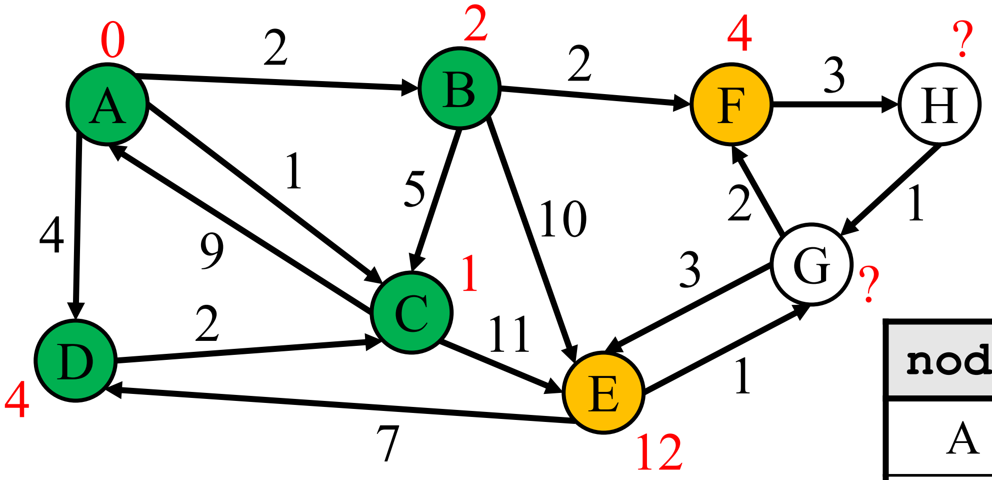


priority queue

path	cost
[A, D]	4
[A, B, F]	4
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		**≤ 4**	**B**
G			
H			

Dijkstra's algorithm – paths from A

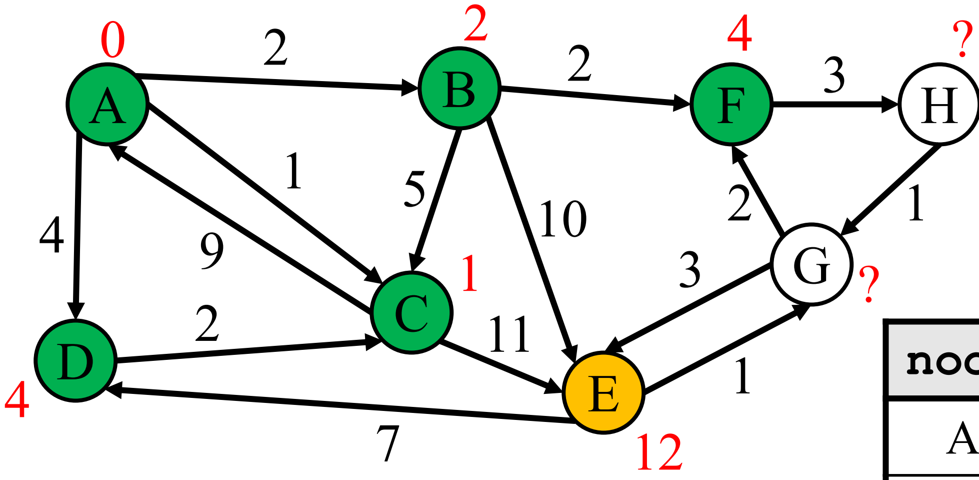


priority queue

path	cost
[A, B, F]	4
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	**4**	A
E		≤ 12	C
F		≤ 4	B
G			
H			

Dijkstra's algorithm – paths from A

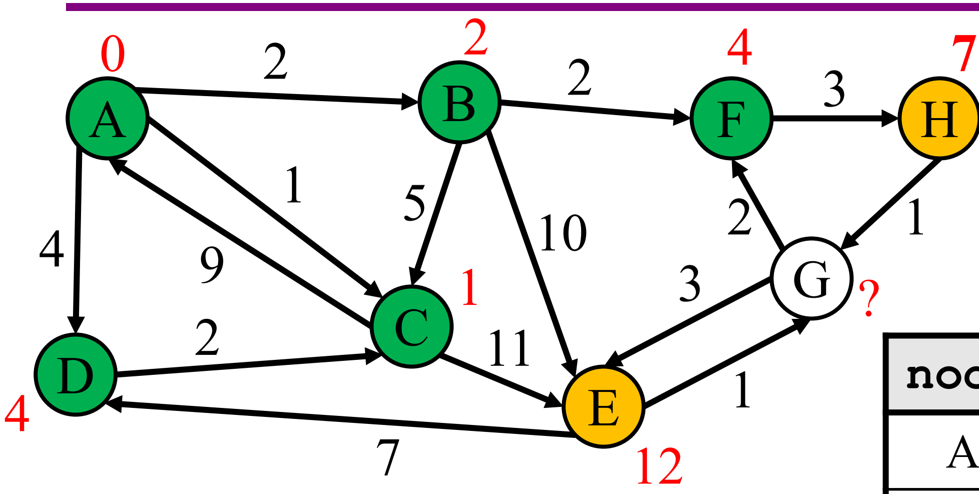


priority queue

path	cost
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	**Y**	**4**	B
G			
H			

Dijkstra’s algorithm – paths from A

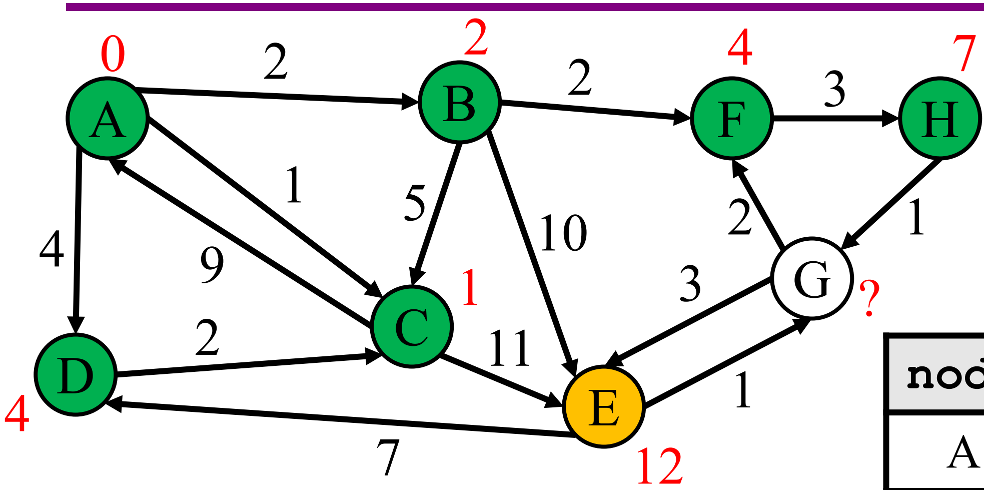


priority queue

path	cost
[A, B, F, H]	7
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G			
H		**≤ 7**	**F**

Dijkstra's algorithm – paths from A

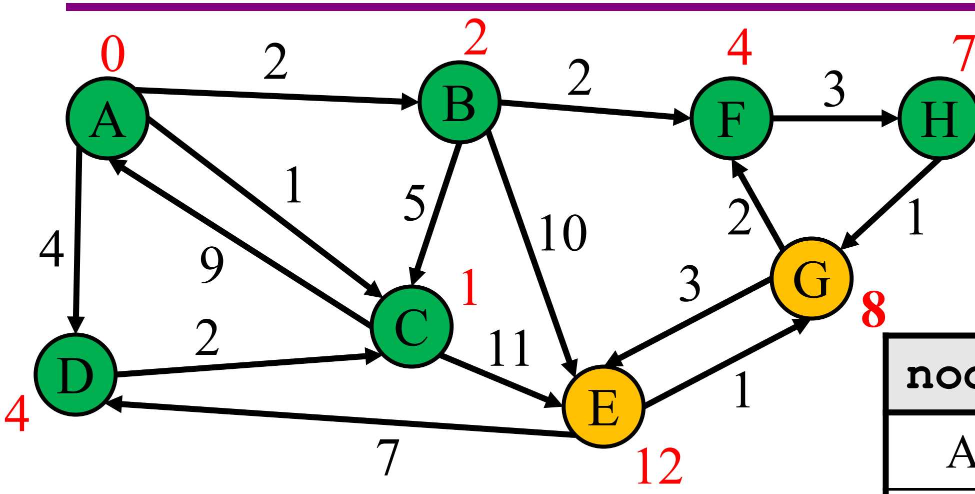


priority queue

path	cost
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G			
H	**Y**	**7**	F

Dijkstra's algorithm – paths from A

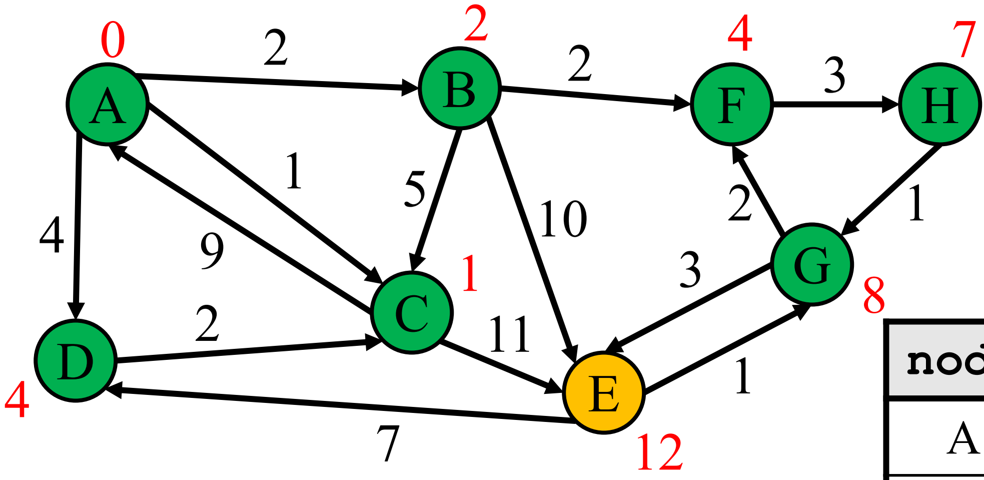


priority queue

path	cost
[A, B, F, H, G]	8
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq$ 12	C
F	Y	4	B
G		**$\leq$ 8**	**H**
H	Y	7	F

Dijkstra's algorithm – paths from A

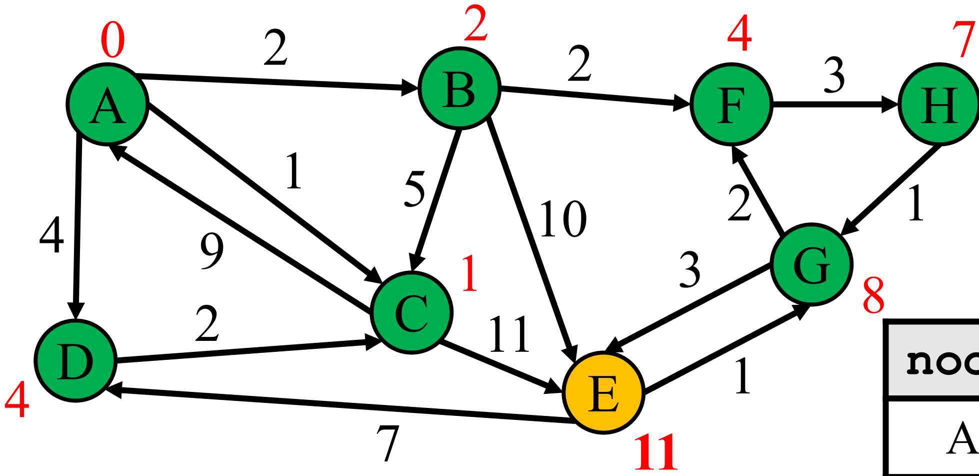


priority queue

path	cost
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G	**Y**	**8**	H
H	Y	7	F

Dijkstra's algorithm – paths from A

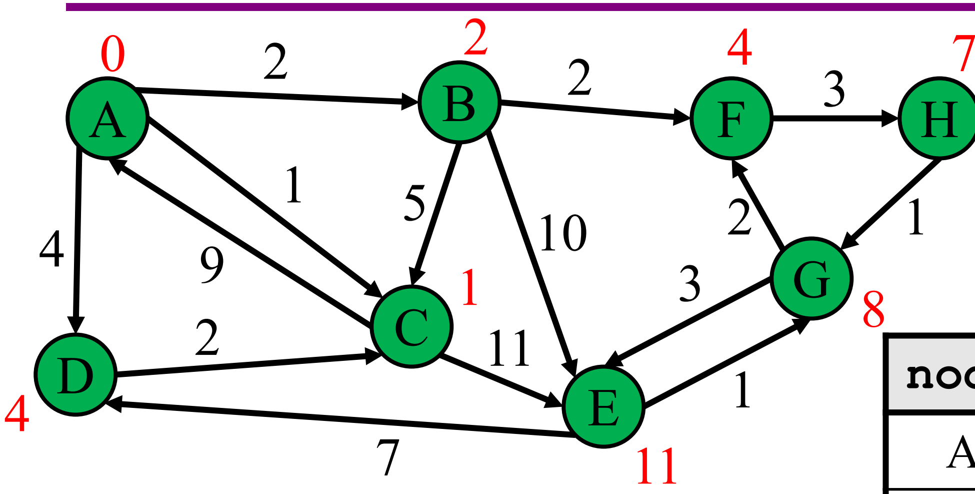


priority queue

path	cost
[A, B, F, H, G, E]	11
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		**≤ 11**	**G**
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's algorithm – paths from A

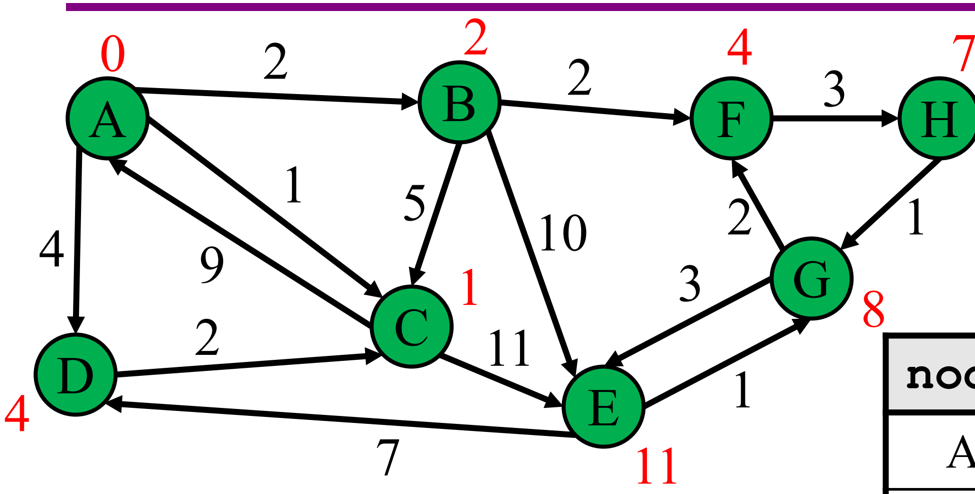


priority queue

path	cost
[A, C, E]	12
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	**Y**	**11**	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's algorithm – paths from A

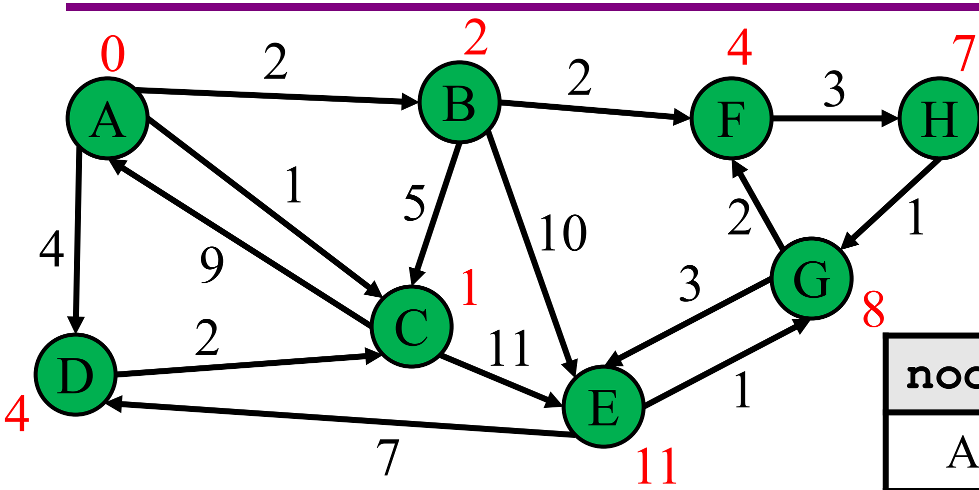


priority queue

path	cost
[A, B, E]	12

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	**Y**	**11**	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra’s algorithm – paths from A



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

priority queue

path	cost

node	finished	cost	prev
A	Y	0	-
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	**Y**	**11**	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's algorithm - Worksheet

- 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
```

What else?

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
```

All nodes not reached yet are farther away than those reached so far

The queue contains all paths formed by adding 1 more edge to a node we already reached.

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

- Model-View-Controller (MVC) is a ubiquitous design pattern:
  - The **model** abstracts + 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

Laptop	Phone
View: The screen displays options for me to select files	
Control: Get input selection from mouse/keyboard	**Control**: Get input selection from touch sensor
Control: Request the selected file from Google Drive	
Model: Google Drive sends back the request file to my device	
Control: Receive the file and pass it to View	
View: The screen displays the file	

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

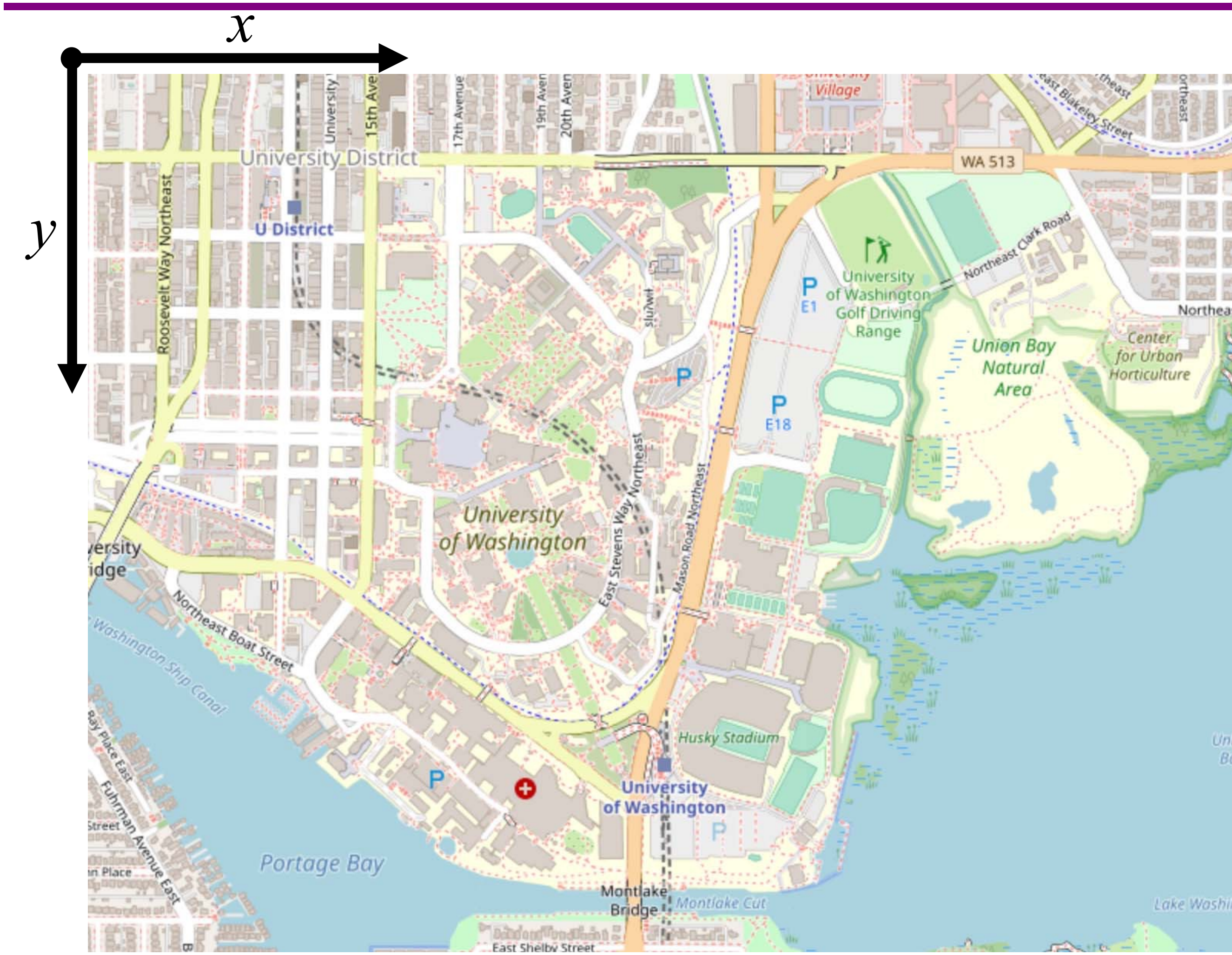
Campus dataset

- 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_map.jpg`

Campus dataset – sample

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

shortName	*longName*	*x*	*y*
`BGR,`	`By George,`	`1671.5499,`	`1258.4333`
`MOR,`	`Moore Hall,`	`2317.1749,`	`1859.502`

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

x1	*y1*	*x2*	*y2*	*distance*
`1810.0,`	`431.5,`	`1804.6429,`	`437.92857,`	`17.956615…`
`1810.0,`	`431.5,`	`1829.2857,`	`409.35714,`	`60.251364…`

- 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 HW5/6
  - Using numeric weights instead of string labels on edges
  - New command **`FindPath`** to find shortest path with Dijkstra's algorithm
- Must write the test driver (**`PathfinderTestDriver`**) yourself
  - Feel free to copy pieces from **`GraphTestDriver`** in HW5/6

Command (in *foo*.test)	Output (in *foo*.expected)
`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
. . .	. . .