Lecture 16 - Dijkstra's Algorithm

Eric A. Autry

Last time: DFS/BFS

This time: Dijkstra's Algorithm

Next time: More Shortest Paths

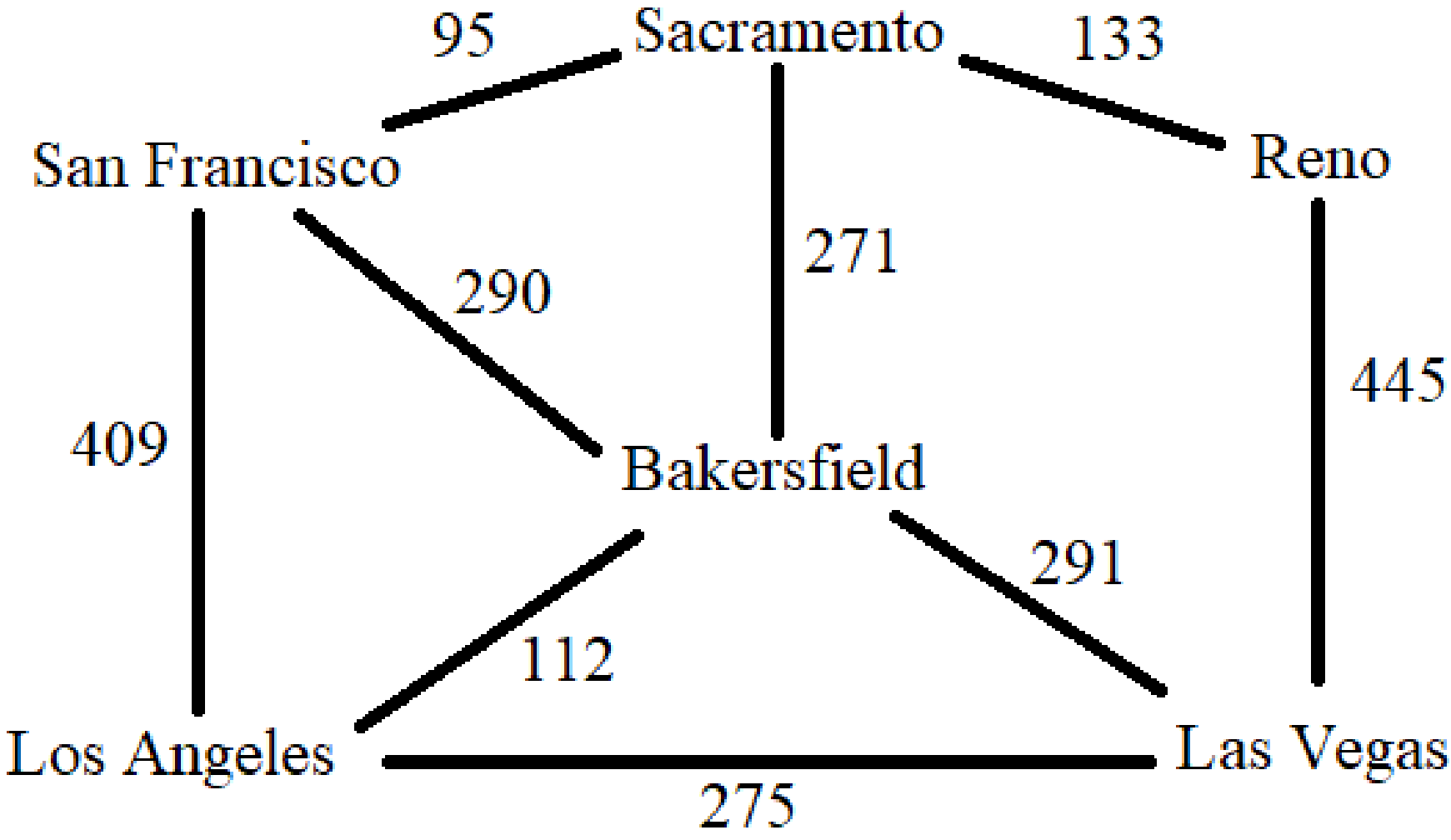
Project 1 due tomorrow, Friday the 2nd!

HW 6 due next Thursday, the 8th.

Project 2 will be assigned today and due Friday the 16th.

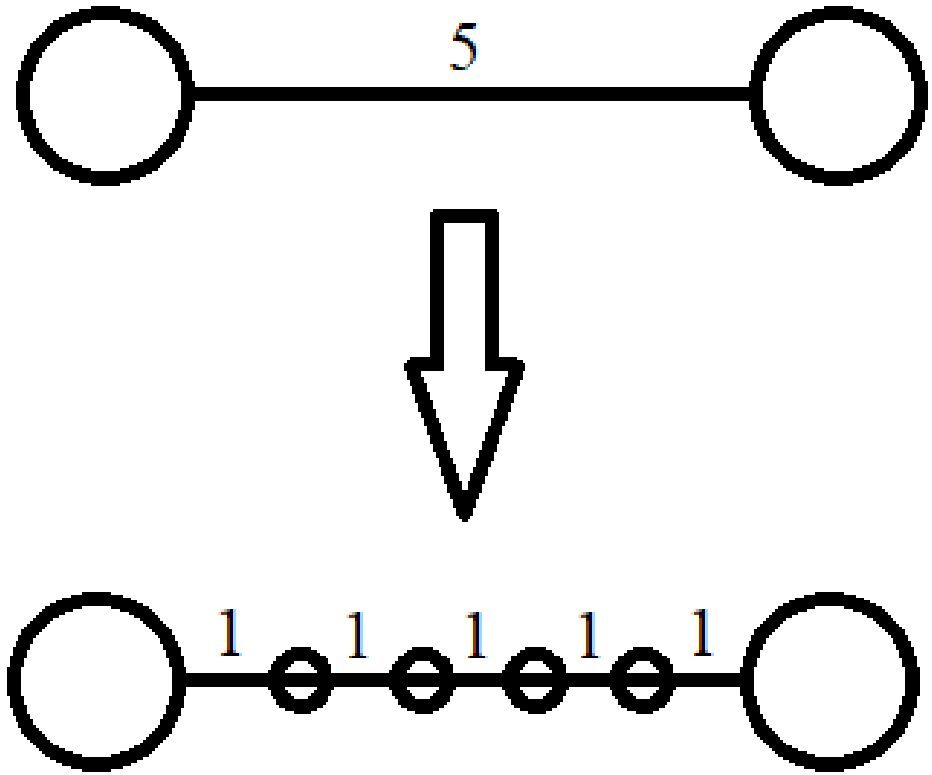
Weighted Graphs

What should we do with a weighted graph, if distances between vertices are not 1?



BFS in a Weighted Graph

Idea #1: an edge weighted as 5 is the same as a twig that is 5 edges long, so introduce dummy edges/vertices in an extended graph and run regular BFS!



Problem: introducing a lot of dummy edges/vertices slows down runtime (what if an edge had a very large weight?)

Weighted Graphs

Idea #2: use a stopwatch.

We will track when the BFS algorithm would have reached each vertex in the extended graph.

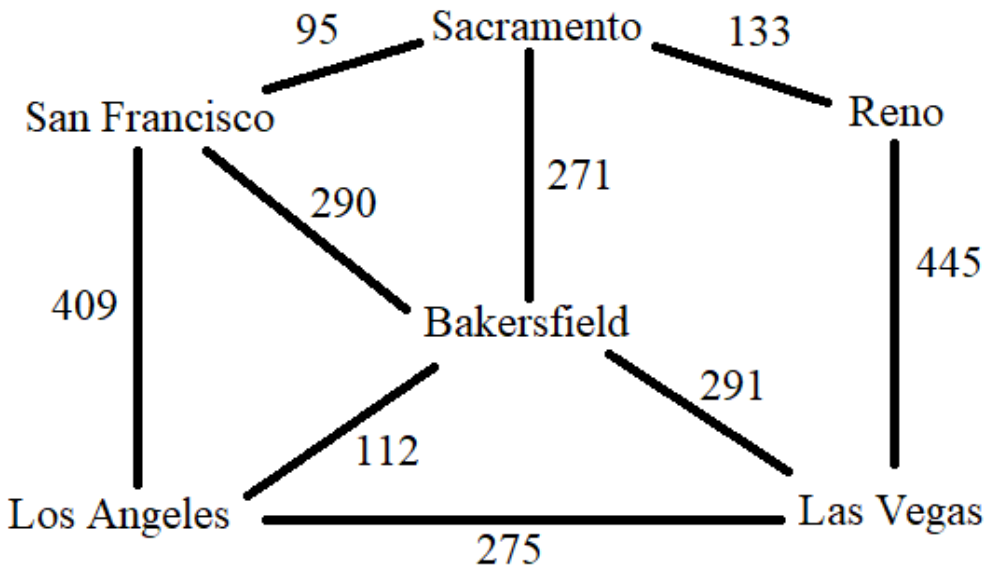
This gives us Dijkstra's algorithm.

Dijkstra's Algorithm

The main idea here is that we are going to make predictions about when BFS would have reached each city.

At first, we have only been in the start city, so our predictions are based only on the edges leaving the start.

When we reach a new city, we can update our predictions.



Might be helpful to imagine sending out racers along each path.

Dijkstra's Algorithm

Description of the algorithm:

- Set the prediction time for the start node at 0.
- While there are still cities left to visit:
  - Say that the next predicted city to visit is city u at time T.
  - This means that the city u will definitely be visited next with a distance of T from the start.
  - For each neighbor of u:
    - If there is no prediction for the neighbor, set its prediction to T plus the length of the edge from u.
    - If its prediction is larger than T plus the length of the edge from u, update the prediction.

How to keep track of the next predicted city?

It is the city with the minimum predicted arrival time. We will need a priority queue!

Priority Queue

Our priority queue will need the operations:

- `Insert`:
  add a new predicted arrival time to the queue
- `Decrease-key`:
  update a predicted arrival time to a smaller time
- `Delete-min`:
  find the minimum prediction and remove it from the queue
- `Make-queue`:
  build the queue originally

Dijkstra's Algorithm Implementation

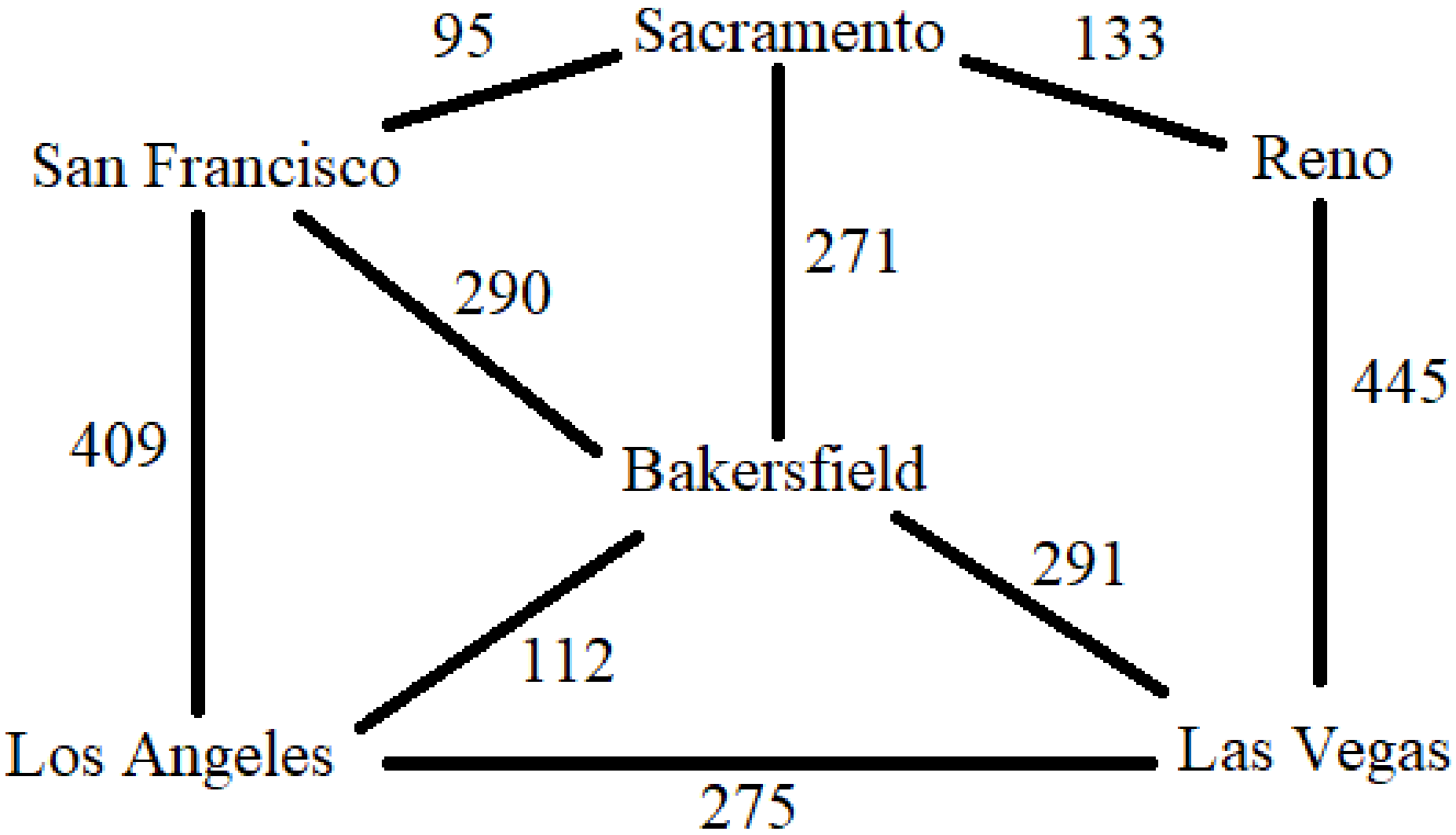
```
def dijkstra(graph, start):
    # Set initial dist and prev.
    for vertex in graph:
        vertex.dist = ∞
        vertex.prev = null
    start.dist = 0

    # Make the queue out of the distances.
    queue = makequeue(dists)

    # Loop while there are vertices left to visit.
    while not queue.isEmpty():
        # Find the next vertex to visit.
        u = deletemin(queue)

        # Check each neighbor of u.
        # Update predictions and previous vertex.
        for neighbor of u:
            if neighbor.dist > u.dist + length(u, neighbor):
                neighbor.dist = u.dist + length(u, neighbor)
                neighbor.prev = u
                decreasekey(queue, neighbor)
```

Dijkstra's Algorithm Example



Dijkstra's Algorithm Alternate Derivation

Idea: divide graph into known and unknown distances.

- Originally the start is known as distance 0 and all other vertices are unknown.

We will then grow the known region one vertex at a time.

- The known region is some subset of the vertices, R, that includes the start.
- The next vertex to visit is *the vertex outside of R that is the closest to the start*. Call this vertex v.

How do we know which v to pick?

Let u be the vertex in the shortest path from start to v that is right before v. Then the distance for u is known (otherwise it would have been the closest vertex outside of R).

So we can easily calculate the distance for all possible v, and then pick the closest of these.

Dijkstra's Algorithm Runtime

What is the runtime of Dijkstra's Algorithm?

It is structurally the same as BFS, but now we have a special priority queue instead of the constant time push and pop of the regular BFS queue.

Let's count the operations performed:

- `makequeue` requires at most $|V|$ `insert` operations.
- Each vertex gets removed from the queue when visited, for a total of $|V|$ `deletemin` operations.
- Potentially each edge in the graph could cause us to update the value of some vertex's distance, for a total of $|E|$ `decreasekey` operations.

So the runtime will depend on how we implement the priority queue.

Unsorted Array

Priority Queue Implementation #1:
Keep an unsorted array of predicted arrival times associated with each vertex.

- `Insert`: append a new value in $O(1)$ time.
- `Decrease-key`: update a single value in $O(1)$ time.
- `Delete-min`: scan through the array to find the minimum and remove it in $O(n)$ time. Array could have $|V|$ elements.

Total runtime:

$$|V| * \texttt{insert} + |E| * \texttt{decreasekey} + |V| * \texttt{deletemin}$$

$$\in O\left(|V| + |E| + |V|^2\right) \in O\left(|V|^2\right)$$

Binary Heap

A binary heap is a **complete** binary tree.

- Each level is filled left-to-right and kept full.

We will be dealing with a min-heap, so each key is always less than its children's keys.

So the root of the heap always contains the minimum.

Binary Heap

- `Insert`: place the new element at the very bottom of the heap, and bubble it up (swap with its parent if it is smaller than the parent). This takes $O(\log_2 n)$ time.
- `Decrease-key`: change the value of the given key, and bubble it up if necessary. This takes $O(\log_2 n)$ time.
- `Delete-min`: remove the root and replace it with the element at the bottom of the heap, then sift that element down (swap with its smallest child if the child is smaller). This takes $O(\log_2 n)$ time.

Note that the heap can be stored as an array, where each node is indexed by numbering left-to-right going down the levels.

A node at index i has parent at $\lfloor i/2 \rfloor$ and children at $2i$ and $2i + 1$.

Binary Heap

- `Insert`: place the new element at the very bottom of the heap, and bubble it up (swap with its parent if it is smaller than the parent). This takes $O(\log_2 n)$ time.
- `Decrease-key`: change the value of the given key, and bubble it up if necessary. This takes $O(\log_2 n)$ time.
- `Delete-min`: remove the root and replace it with the element at the bottom of the heap, then sift that element down (swap with its smallest child if the child is smaller). This takes $O(\log_2 n)$ time.

Total runtime:

$$|V| * \texttt{insert} + |E| * \texttt{decreasekey} + |V| * \texttt{deletemin}$$

$$\in O\left((|V| + |E|) \log |V|\right)$$

d-ary Heap

A d-ary Heap is a heap where each node has d children.

- A binary heap is special case where $d = 2$.
- The height of the heap is now $\log_d n = \log n / \log d$.
- Bubble up operations are faster by a factor of $\log d$ due to the shorter height.
- Sift down operations are slower at $O(d \log n / \log d)$ because we have to check d children for each sift.

Note that the d-ary heap can also be stored as an array, where each node is indexed by numbering left-to-right going down the levels.

Now a node at index i has a parent at $\lfloor (i-1)/d \rfloor$ and children at $(i-1)d + 2$ through $(i-1)d + d + 1$ (unless we hit end of heap first).

Fibonacci Heap

Fibonacci Heaps are very complicated heap structure and are hard to implement (so are often avoided).

- `Insert`: takes varying amount of time because every now and again we can have a very expensive insert.
  - But, it averages to $O(1)$ time per `insert`.
  - We call this **amortized** $O(1)$ time. We will talk about this more next week.
- `Decrease-key`: same as `insert`.
- `Delete-min`: $O(\log |V|)$ time

Dijkstra's Algorithm Runtime

Total runtime:

$$|V| * \texttt{insert} + |E| * \texttt{decreasekey} + |V| * \texttt{deletemin}$$

- Unsorted Array: $O\left(|V|^2\right)$
- Binary Heap: $O\left((|V| + |E|) \log |V|\right)$
- d-ary Heap: $O\left((|V| \cdot d + |E|) \frac{\log |V|}{\log d}\right)$
- Fibonacci Heap: $O(|V| \log |V| + |E|)$

Dijkstra's Algorithm Runtime

So the runtime will depend on whether the graph is **dense** (a lot of edges) or **sparse** (few edges).

Recall that if there are n vertices, then $|E| \in O(n^2)$ in the worst case and $|E| \in O(n)$ in the best case.

- If the graph is dense, the unsorted array is better with a runtime of $O(n^2)$.
- If the graph is sparse enough, the binary heap is better with a runtime of $O(n \log n)$. The binary heap is better if

$$|E| \log n < n^2 \quad \Rightarrow \quad |E| < n^2 / \log n.$$

Dijkstra's Algorithm Runtime

For the d-ary heap, the runtime depends on choice of d.

It is best is to pick $d \sim |E|/|V|$, which is the average degree of the graph.

- If the graph is very dense where $|E| \sim n^2$, then the runtime is $O(n^2)$, which is as good as the unsorted array (which was preferred for dense graphs).
- If the graph is very sparse where $|E| \sim n$, then the runtime is $O(n \log n)$, which is as good as the binary heap (which was preferred for sparse graphs).
- For graphs with an intermediate density where $|E| \sim n^{1+\alpha}$ $(0 < \alpha < 1)$, the runtime is $O(|E|) = O(n^{1+\alpha})$.

Course Feedback

I would like you all to take a course feedback survey at:

`http://duke.is/xdRkXT`

I would like this feedback to give me a better sense of where you all stand with the class, how you feel about the pacing and difficulty, and what your backgrounds are.

I will use this feedback to potentially adjust the pace of the lectures going forward.

This survey will be **completely anonymous**, so please feel free to share negative feedback along with any positive feedback you have.

Please complete the survey by **next Tuesday the 6th at 6 pm**. It should only take a couple minutes.