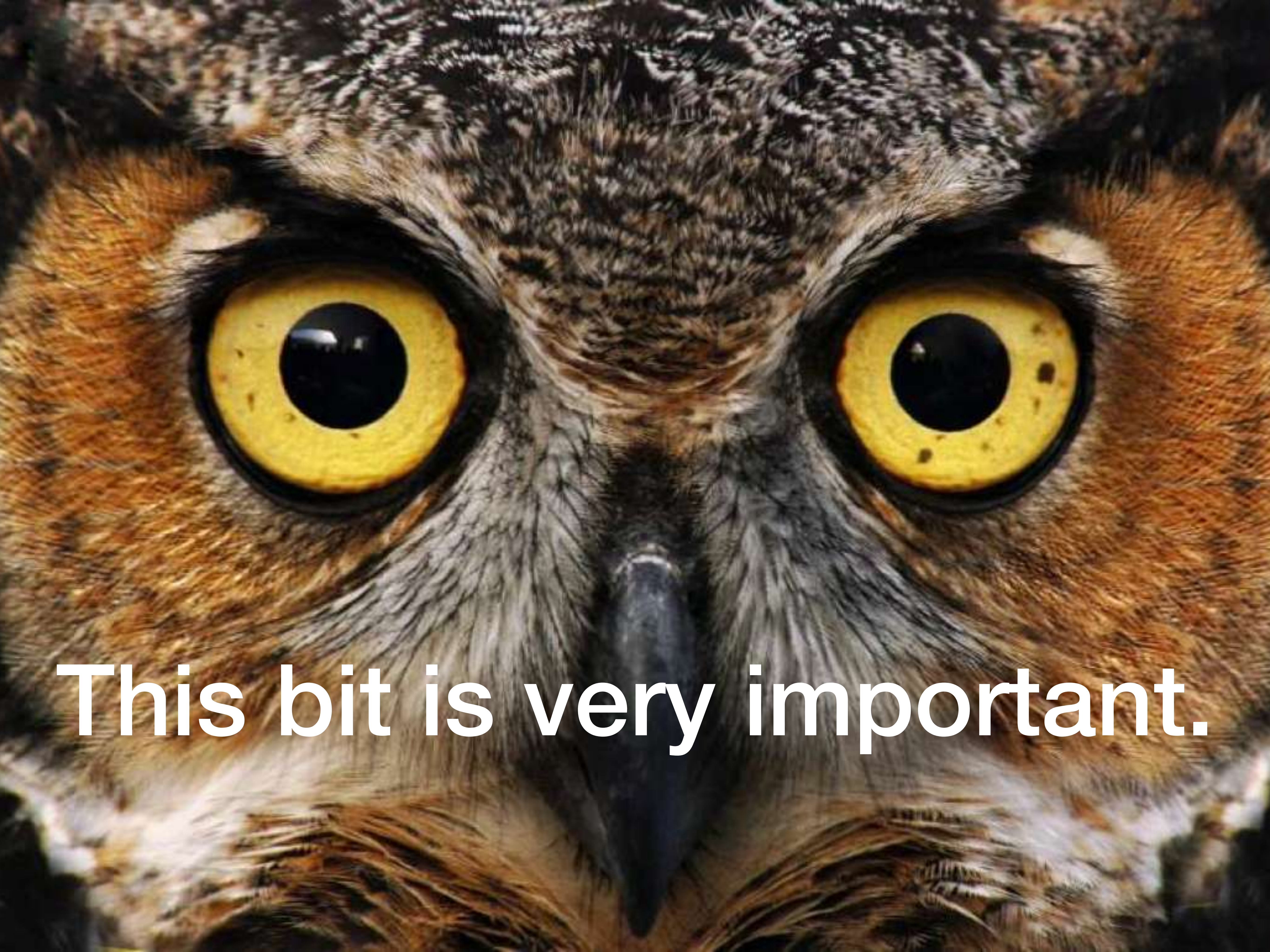
CSCI 241

Scott Wehrwein

Dijkstra's Algorithm:
Efficient Implementation

Goals

Be prepared to implement Dijkstra's algorithm efficiently.


This bit is very important.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
   }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. To store w.d and w.bp, we will use a HashMap<Node,PathData>
4. We don't need to explicitly store Settled or Unexplored sets:
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

1. **Store Frontier in a min-heap priority queue with d-values as priorities.**
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. To store w.d and w.bp, we will use a HashMap<Node,PathData>
4. We don't need to explicitly store Settled or Unexplored sets:
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. **To efficiently iterate over neighbors, use an adjacency list graph representation.**
3. To store w.d and w.bp, we will use a HashMap<Node,PathData>
4. We don't need to explicitly store Settled or Unexplored sets:
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
   }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. **To store w.d and w.bp, we will use a HashMap<Node,PathData>**
4. We don't need to explicitly store Settled or Unexplored sets: a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
  f = node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
     if (w not in S or F) {
          w.d =  f.d + weight(f, w);
          w.bp = f;
          add w to F;
     } else if (f.d+weight(f,w) < w.d) {
          w.d = f.d+weight(f,w);
          w.bp = f
     }
  }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. To store w.d and w.bp, we will use a HashMap<Node,PathData>
4. **We don't need to explicitly store Settled or Unexplored sets: a node is in S or F iff it is in the map.**

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

4. We don't need to explicitly store Settled or Unexplored sets: w is in S or F ⇔ it is in the map.

The only time we need to check membership in S is **here**.

If w is not in S or F,
it must be in Unexplored.

therefore,
we haven't found a path to it.

therefore,
it has no d or bp yet.

therefore,
it isn't in the map!