DFS RUNTIME

At worst, |V| times through the while loop to check each vertex

```
while (! toCheck.isEmpty()) {
  Vertex<T> checkingVertex = toCheck.removeLast(); // removeFirst() for BFS
  if (dest.equals(checkingVertex)) {
    return true;
  }
  for (Vertex<T> neighbor : checkingVertex.getOutgoing()) {
    if (!visited.contains(neighbor)) {
      visited.add(neighbor);
      toCheck.addLast(neighbor);
    }
  }
}
```

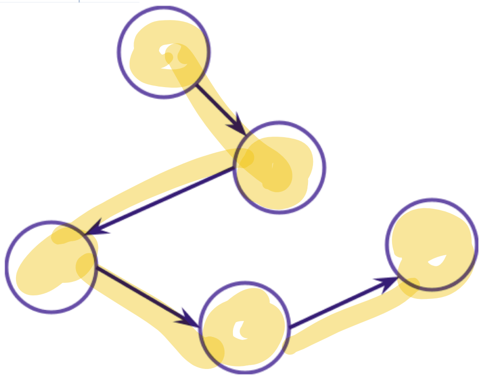
$O(1)$

$O(|V|)$?

A vertex might have |V| -1 neighbors

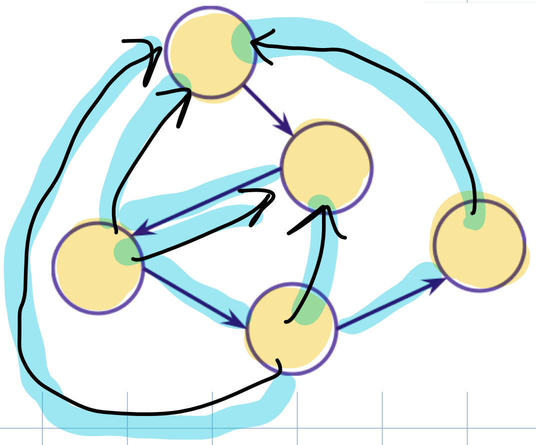
$O(1)$

INITIAL ESTIMATE: $|V| \cdot |V| = O(|V|^2)$



HOWEVER, vertices aren't the only thing that contributes to runtime in graph algorithms. The two groups here have the same number of vertices, but the one on the right has more edges, so it will take longer to run. Therefore it's important to measure the runtime of graph algorithms according to both:

- |V|: The size of the set of vertices
- |E|: The size of the set of edges



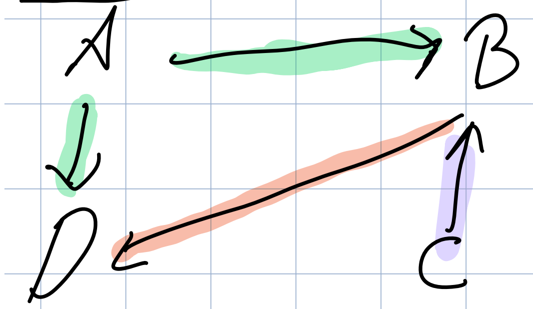
```
while (! toCheck.isEmpty()) {
  Vertex<T> checkingVertex = toCheck.removeLast(); // removeFirst() for BFS
  if (dest.equals(checkingVertex)) {
    return true;
  }
  for (Vertex<T> neighbor : checkingVertex.getOutgoing()) {
    if (!visited.contains(neighbor)) {
      visited.add(neighbor);
      toCheck.addLast(neighbor);
    }
  }
}
```

If we consider these operations across all iterations of the while loop, they'll contribute O(|V|) to the runtime

Q: What runtime will these lines (highlighted in blue) contribute across all iterations of the loop?
=> O(|E|)

Why? Each time we visit a vertex (while loop), we check the edges corresponding to that vertex's neighbors only. Across all iterations of the loop, this means that we check each edge exactly once. For an example, take a look at the shading in this graph where we've written out the set of vertices and edges and try to match it to the code.

EXAMPLE



VERTICES: A, B, C, D

EDGES: { A → B
A → D
B → D
C → B }

Therefore, final runtime is the combination of these contributions.

Why isn't it O(|V| * |E|)? Even though the for loop is nested inside the while loop, it runs on a different subset of E each time (ie, the neighbors of checkingVertex). In total, we end up checking each edge exactly once, so the total runtime is time to loop over each vertex (|V|) plus the time to loop over each edge (|E|).
Take a look at the graph on this page and the typed notes for more details.

FINAL RUNTIME

$O(|V| + |E|)$

Runtime for Dijkstra

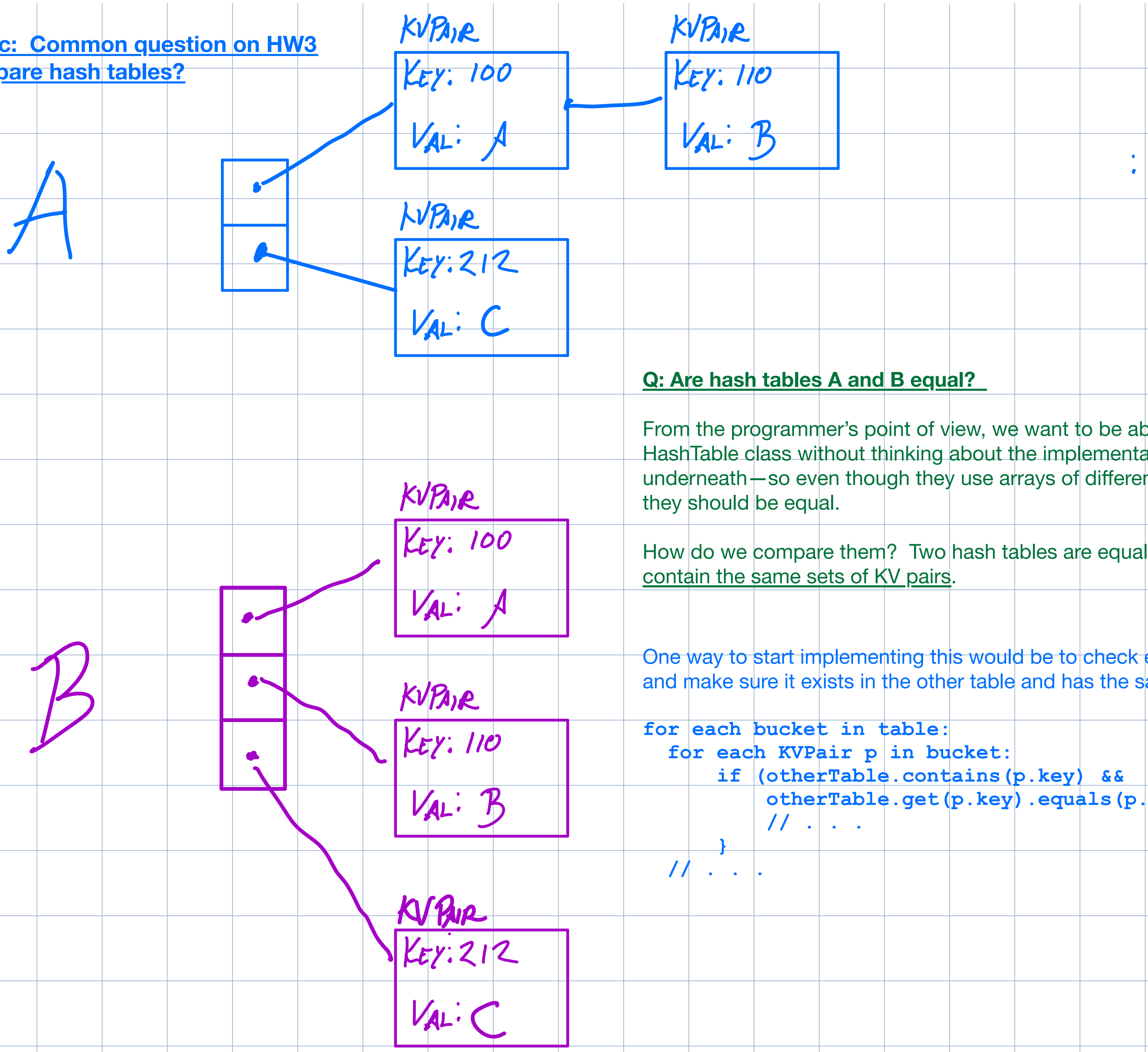
```
toCheckQueue = V (prioritized on routeDist)        ] O(|V|)
cameFrom = empty map

for v in V:                                        } O(|V|)
  v.routeDist = inf
source.routeDist = 0

while toCheckQueue is not empty:                   <- As before, this loop runs |V| times
  checkingV = toCheckQueue.removeMin()             <- Remove from ideal priority queue:  O(log(|v|)
  for neighbor in checkingV's neighbors:
    if checkingV.routeDist + cost(checkingV, neighbor) < neighbor.routeDist:   O(1)
      neighbor.routeDist = checkingV.routeDist + cost(checkingV, neighbor)     O(1)
      cameFrom.add(neighbor -> checkingV)          O(1)
      toCheckQueue.decreaseValue(neighbor)         <- decreaseValue has O(log(|V|)

backtrack from dest to source through cameFrom     <- O(|V|)
```

(with optimized priority queue implementation—see notes for details)

$$O(\cancel{|V|} + |V|\log(|V|) + |E|\log(|V|))$$

$$\approx O((|V|+|E|)\cdot\log(|V|))$$

Another topic: Common question on HW3
How to compare hash tables?



Q: Are hash tables A and B equal?

From the programmer's point of view, we want to be able to the HashTable class without thinking about the implementation underneath—so even though they use arrays of different sizes, they should be equal.

How do we compare them? Two hash tables are equal if they contain the same sets of KV pairs.

One way to start implementing this would be to check each KVPair and make sure it exists in the other table and has the same value

```
for each bucket in table:
  for each KVPair p in bucket:
      if (otherTable.contains(p.key) &&
          otherTable.get(p.key).equals(p.value)){
          // . . .
      }
  // . . .
```