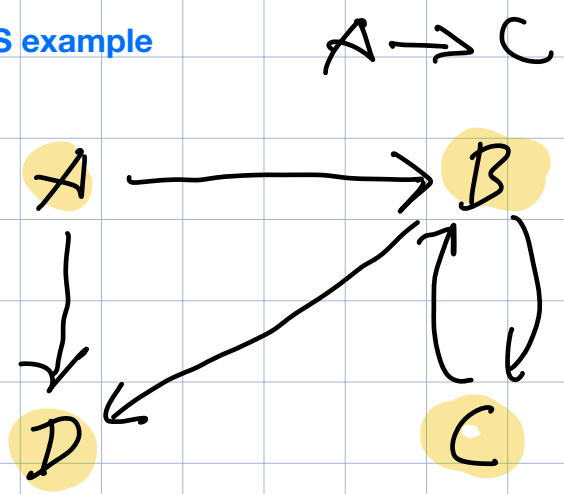
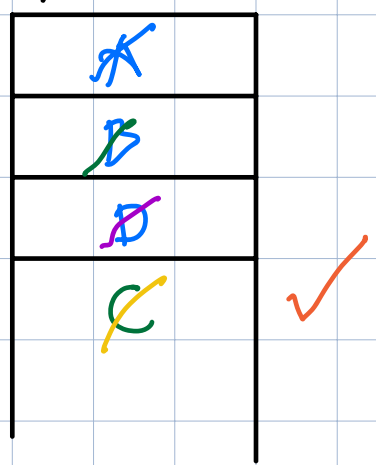
DFS example



TO CHECK:



```
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);
    }
  }
}
```

VISITED: A B D C

DFS Runtime

```
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);
    }
  }
}
```

At worst, runs |v| times through this while loop.

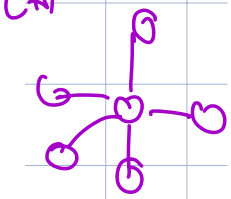
WHY NOT?

PICK O(1)

LOOP THROUGH ALL EDGES

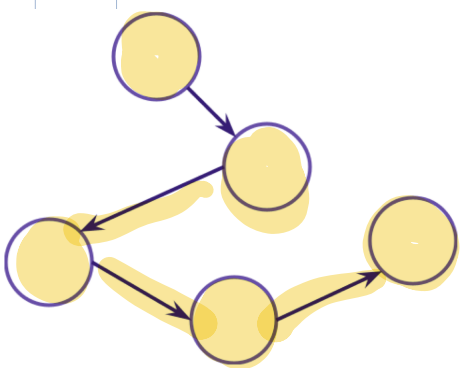
O(1)

ONE CHANGE



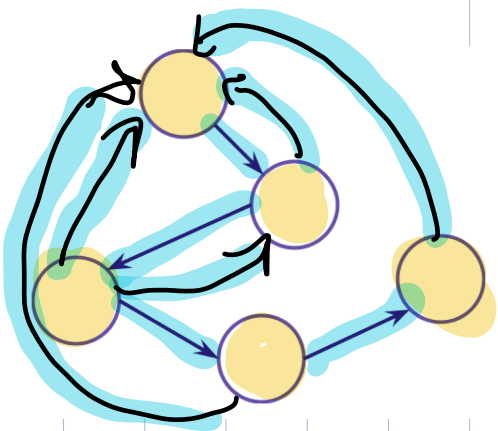
Look at neighbors
In the worst case, how many neighbors might a node have?
O(|V|)???

Let's look a bit deeper—what are we measuring runtime on, exactly?



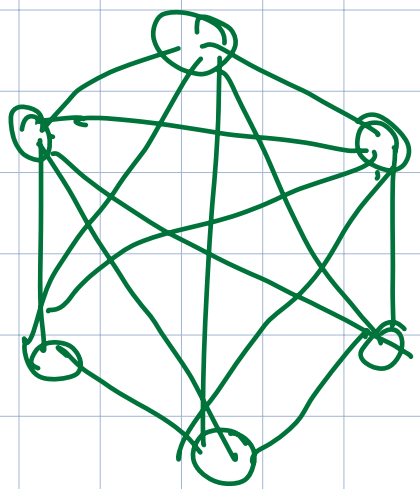
|V| (the size of the set of vertices)
|E| (the size of the set of edges)

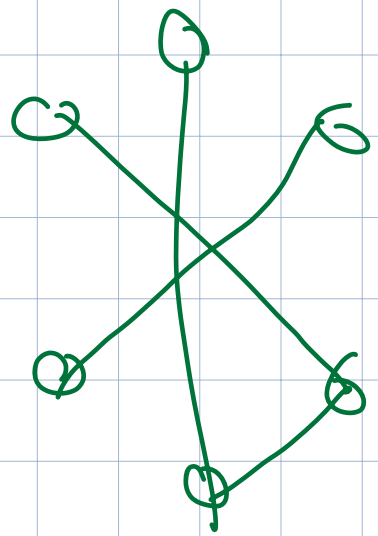
WHAT WOULD THE WORST CASE BE?



=> Even though both graphs have the same number of vertices, we're certainly doing more work in this one because it has more edges

=> If we follow our original rules, we'd get (O|V|^2)
but it's actually a bit different than that, because we change the set of nodes we're iterating over





There are dense graphs and sparse graphs. Some algorithms work well on sparse graphs, and some work well on densely connected graphs.

⇒ BOTH |V| AND |E| AFFECT RUNTIME!

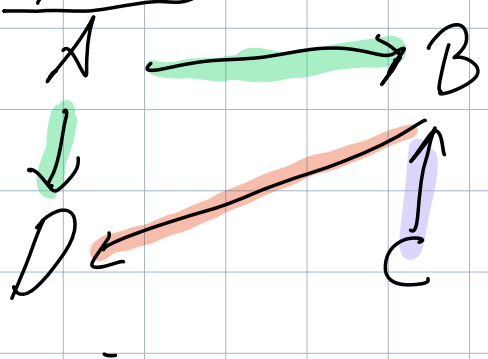
```
while (! toCheck.isEmpty()) {
  Vertex<T> checkingVertex = toCheck.removeLast(); // removeFirst() for BFS
  if (dest.equals(checkingVertex)) {
   return true;
  }
  for (Vertex<T> neighbor : checkingVertex.getOutgoing()) {
    if (!visited.contains(neighbor)) {
      visited.add(neighbor);
     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

for v in V:
  v.routeDist = inf
source.routeDist = 0

while toCheckQueue is not empty:
  checkingV = toCheckQueue.removeMin()
  for neighbor in checkingV's neighbors:
    if checkingV.routeDist + cost(checkingV, neighbor) < neighbor.routeDist:
      neighbor.routeDist = checkingV.routeDist + cost(checkingV, neighbor)
      cameFrom.add(neighbor -> checkingV)
      toCheckQueue.decreaseValue(neighbor)

backtrack from dest to source through cameFrom
```

$O(|V|)$

$O(|V|)$

As before, this loop runs |V| times

Remove from ideal priority queue: O(log(|v|)

$O(1)$

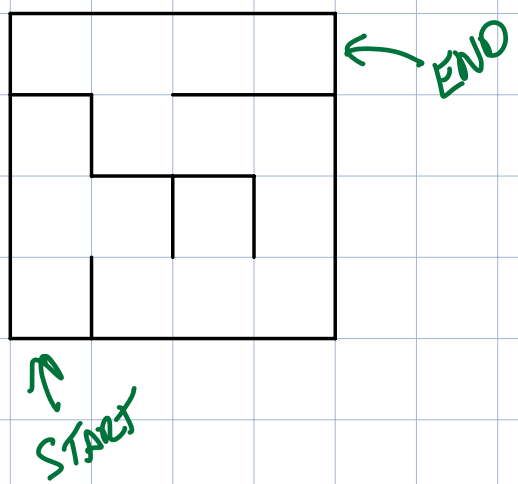
$O(1)$

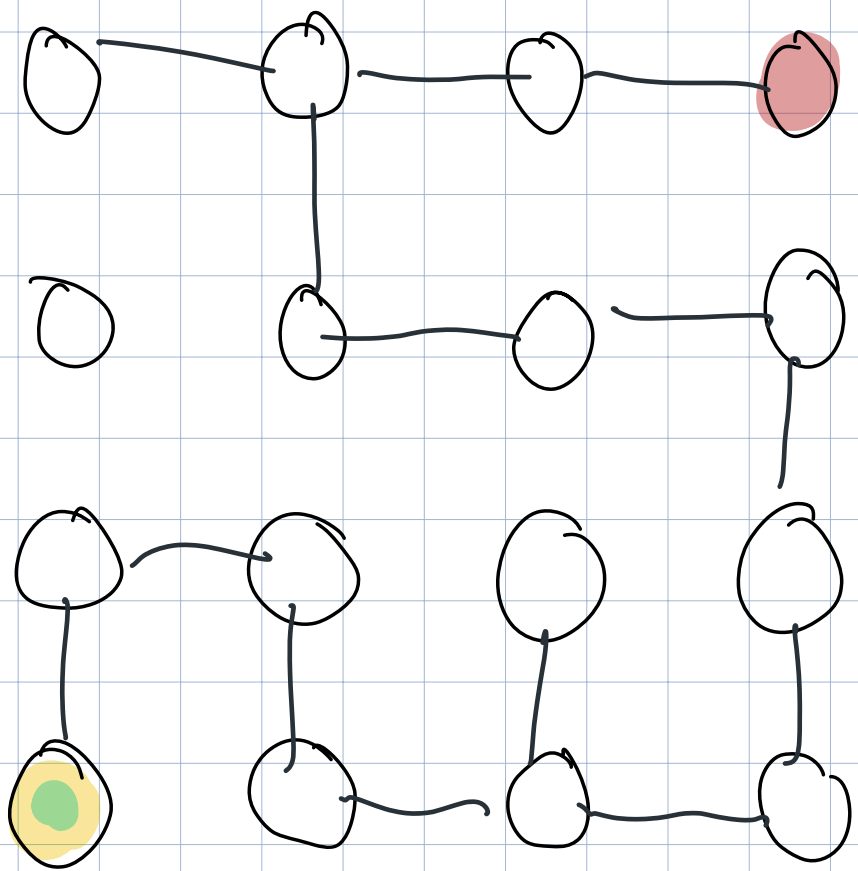
$O(1)$

decreaseValue has O(log(|V|)
(with optimized priority queue implementation—see notes for details)

$O(|V|)$

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



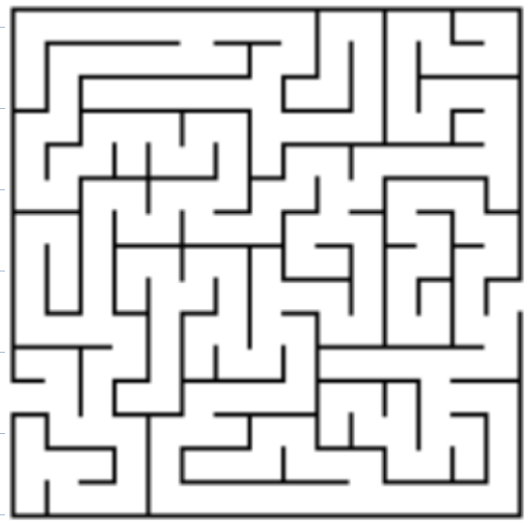


BFS, DFS are search algorithms: “can I find a path from source to dest”
Dijkstra is a “shortest path algorithm”: how to find cheapest-cost path from source to dest

DFS
(STACK)

Follow one path until you hit a dead end

BFS
(QUEUE)

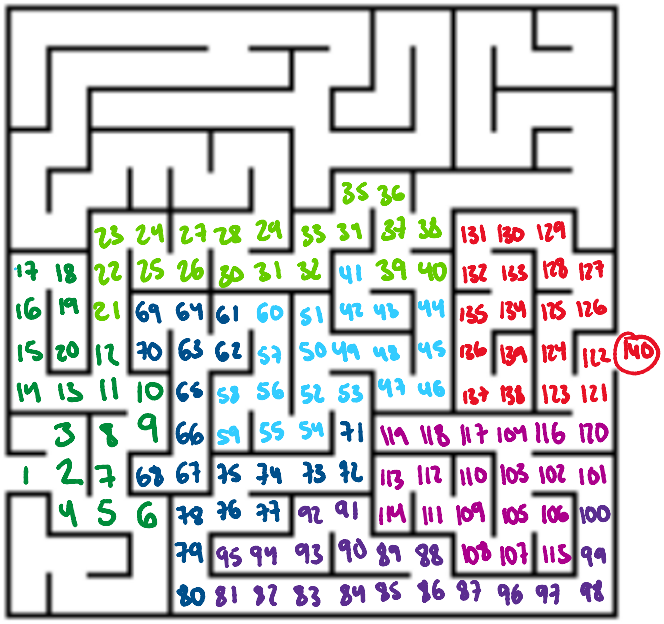


A*
(PRIORITY QUEUE)

[(Attribution: Pages 4-5 drawn by Milda from Fall 2022 version)]

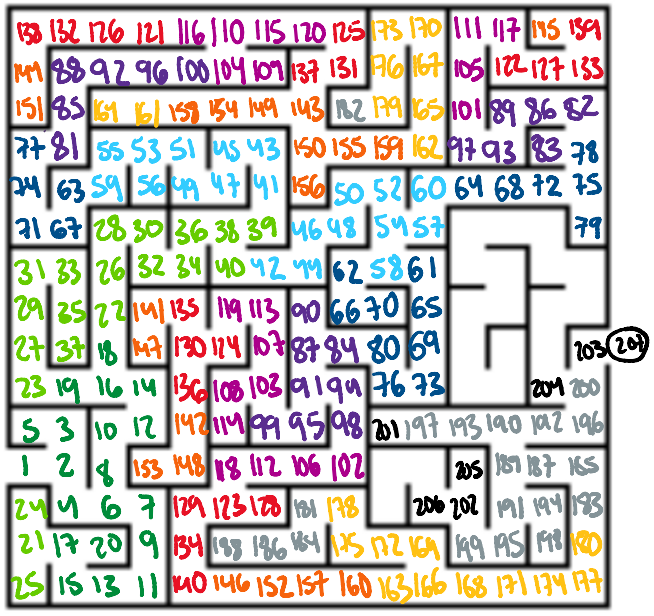
Bigger maze comparison

Monday, October 24, 2022 1:02 PM



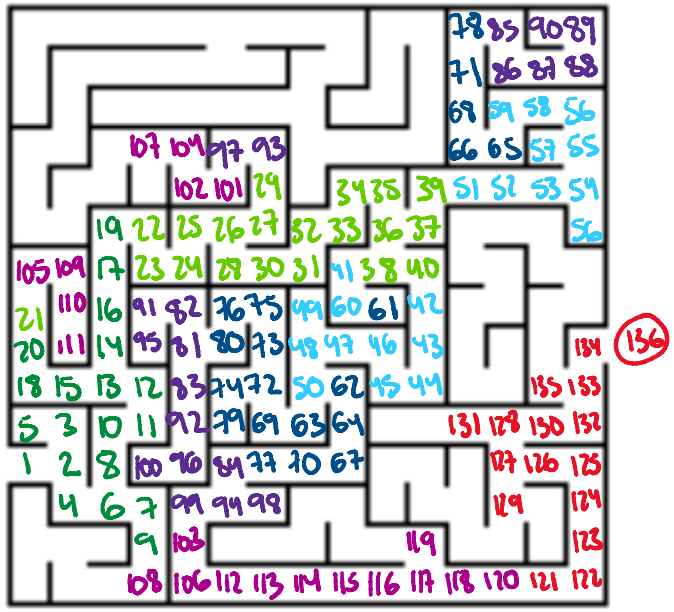
DFS (stack)

Will go down a path until it reaches a dead end and then search from last-seen branching-off point



BFS (queue)

Will "fan out" from the beginning of the maze (tracking many routes at once)



A* (priority queue)

Prioritizes based on distance to the end -- turns out to be fastest for most mazes

*A note on how these mazes were labeled: the number represents the timestep when that cell was *added* to the toCheck stack/queue/priority queue. Neighbors are checked in the order right, up, left, down (a different ordering can result in different numberings/traversals for the mazes). For A*, Manhattan distance is used and ties are broken by considering the cell that was added to the PQ earlier (has a lower timestep number). Colors change every 20 steps.*

Could we use Dijktra's algorithm to search the maze? BFS/DFS/A* are search algorithms (goal: find path to destination), whereas Dijkstra shortest path algorithm (ie, find shortest path to **any** node from source)—these are different types of algorithms and best-suited for different use cases! We'll talk about the runtime for BFS/DFS/Dijkstra in the next class.