BFS/DFS runtime

Sunday, October 23, 2022 6:07 PM

```
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, |V| times through this while loop

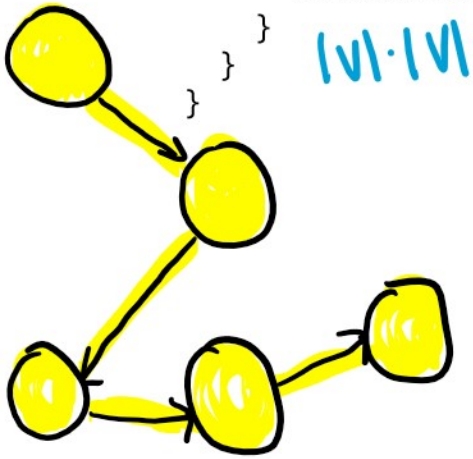
O(1)

O(|V|)

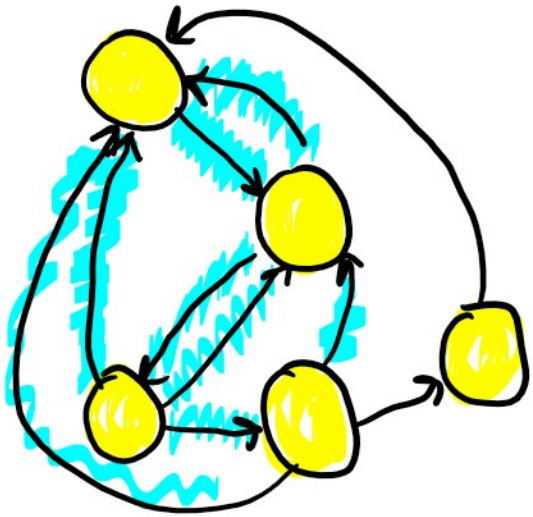
A vertex may have up to |V| - 1 neighbors

O(1)

$|V| \cdot |V| = O(|V|^2)$ <- initial estimate



BUT: vertices aren't the only thing that contributes to the runtime on graph algorithms. The two graphs shown here have the same number of vertices, but the one on the right has more edges and the algorithm will take longer to run. Hence, we measure the runtime of graphs 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 operation across all iterations of the while loop, they're going to contribute O(|V|) to the runtime

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

Final runtime: $O(|V|+|E|)$

Dijkstra runtime

Monday, October 24, 2022 1:06 PM

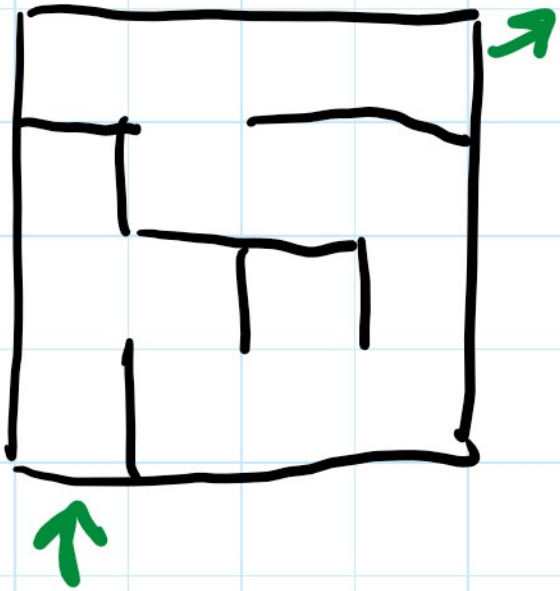
```
toCheckQueue = V (prioritized on routeDist)
cameFrom = empty map

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
```

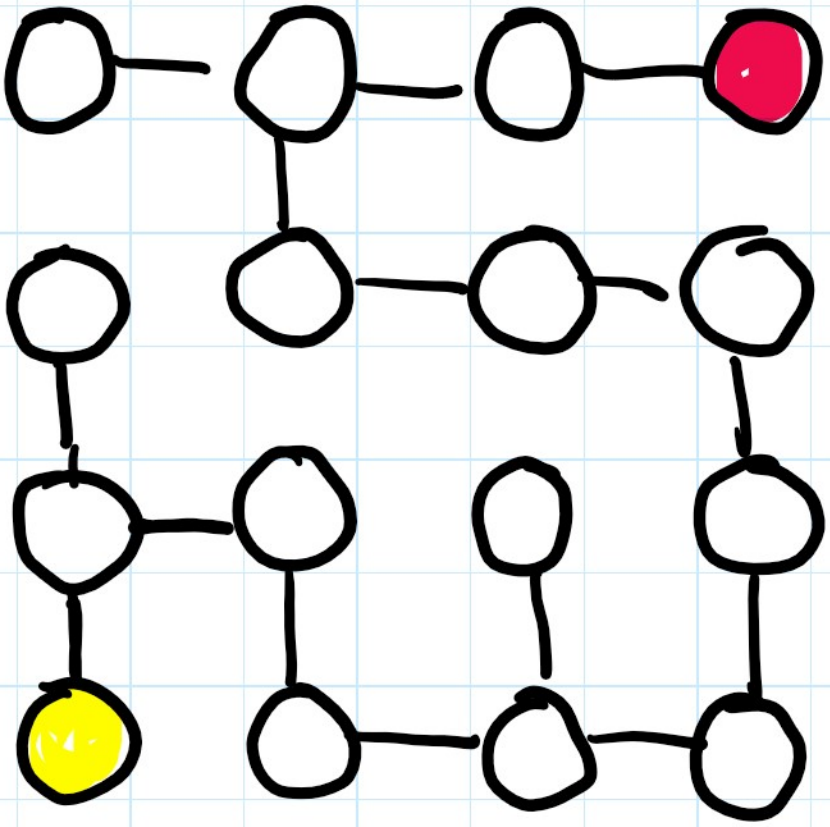
O(|V|) O(|V|) |V| times through this loop O(log|V|) O(1) O(1) O(1) O(log|V|) (with optimized priority queue implementation -- see notes) O(|V|)

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

Representing mazes as graphs

Sunday, October 23, 2022 4:54 PM

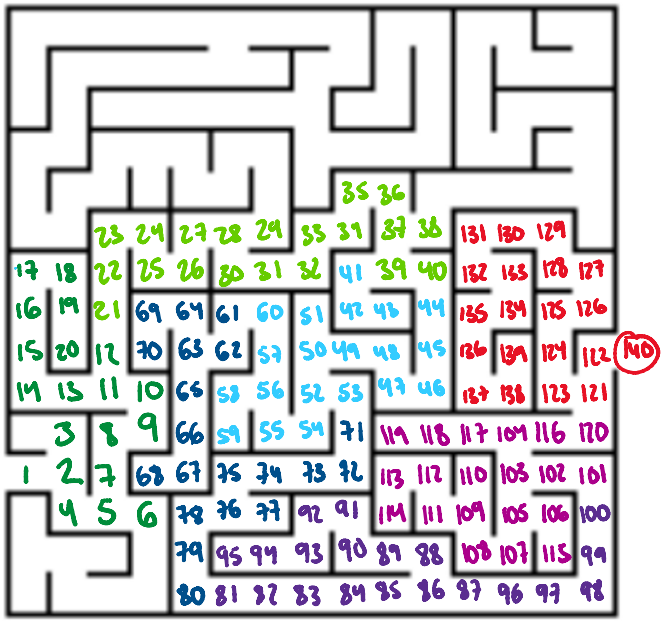




Solving the maze = finding route (DFS or BFS) from vertex that represents starting cell to vertex that represents ending cell

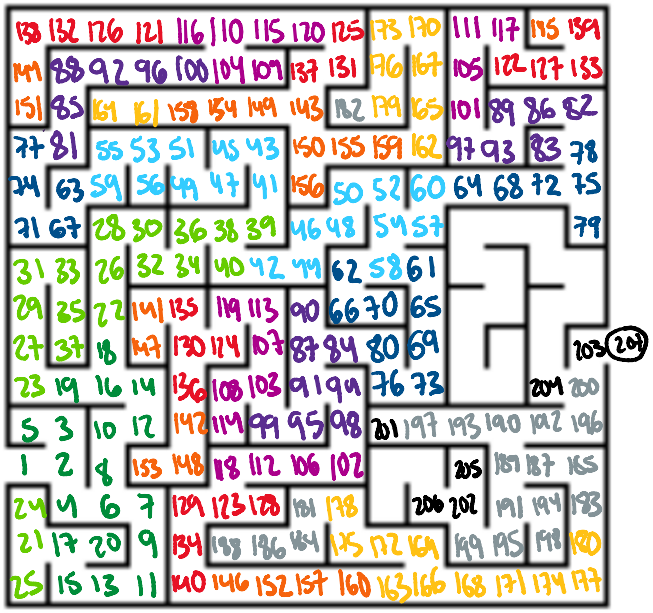
Bigger maze comparison

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.*