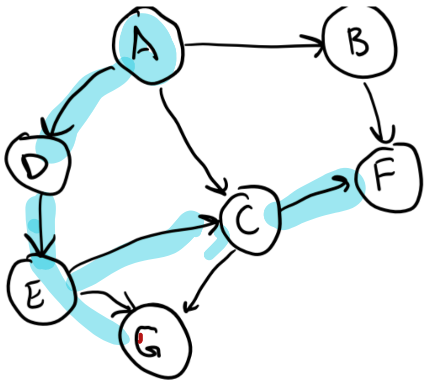
A → F ?



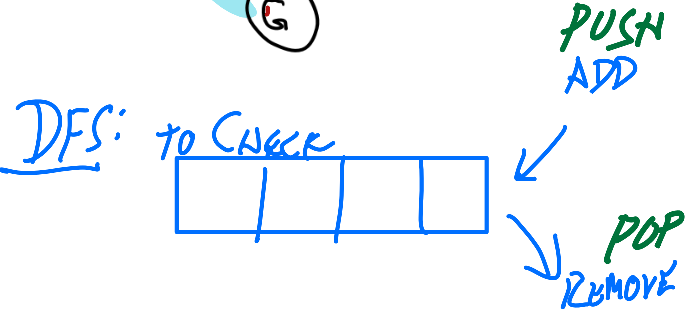
Visited A, B, C, D, E, G, F

TO CHECK

~~A~~ B
~~C~~ ~~F~~
~~D~~ ~~E~~ ~~G~~

F

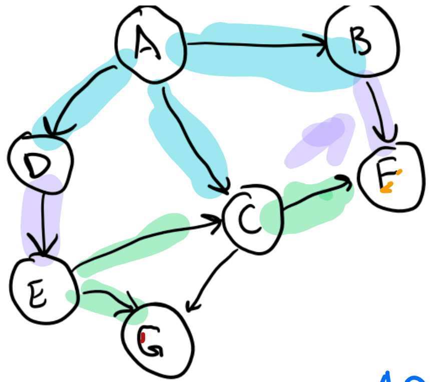
DFS:



DFS: Removing most recent item to have been added to list
This is called “last in/first out” (LIFO) order
=> This is a stack
(which happens to be implemented with a linked list)

DFS (DEPTH-FIRST SEARCH)

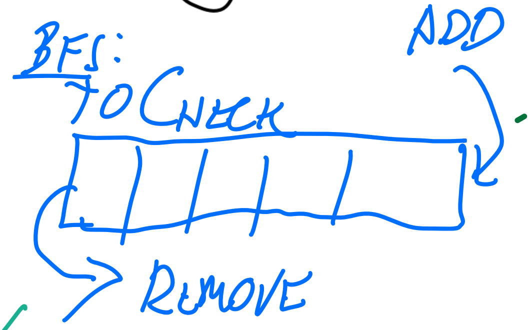
Visited
A, B, C, D, F, G, E



TO CHECK

~~A~~ ~~B~~
~~C~~
~~D~~
~~F~~
G
E

BFS:



BFS: Removing the least recent item to have been added to the list
This is called “first in/first out” (FIFO) order
=> This is a queue
(Which also happens to be implemented with a linked list)

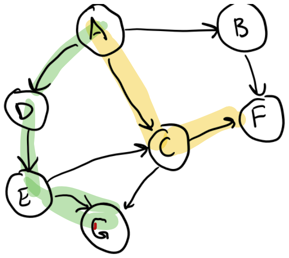
BFS (BREADTH-FIRST SEARCH)

What other info would you need to return THE PATH from A->F?? (Example: A->C->F)
The code we've seen so far (below), implements canReach(), which just tells us if a path exists, not what it is.

BFS/DFS peudocode

```
HashSet<Vertex> visited = new HashSet<Vertex>();
LinkedList<Vertex> toCheck = new LinkedList<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);
    }
  }
  return false;
}
```

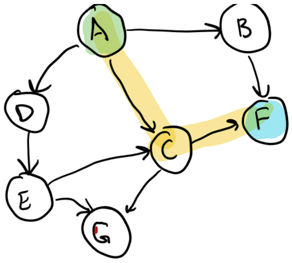


How would we implement this???

Starting point: *could* store the path each time we visit a node, but the paths could get really long => would need a lot of storage!

A A
D A→D
E A→D→E
G A→D→E→G

Instead: would like to track which node we "came from" when considering each node
=> Need some data structure to store this info, then can read it "backwards" (from end to start) to find the path



EG. FOR A→F:

STORE THIS

A → B
A → C
A → D
C → F

USE TO LOOK UP:

A → F

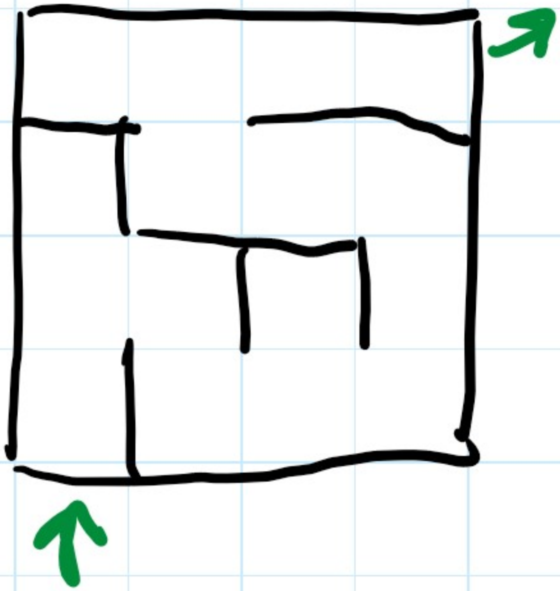
START

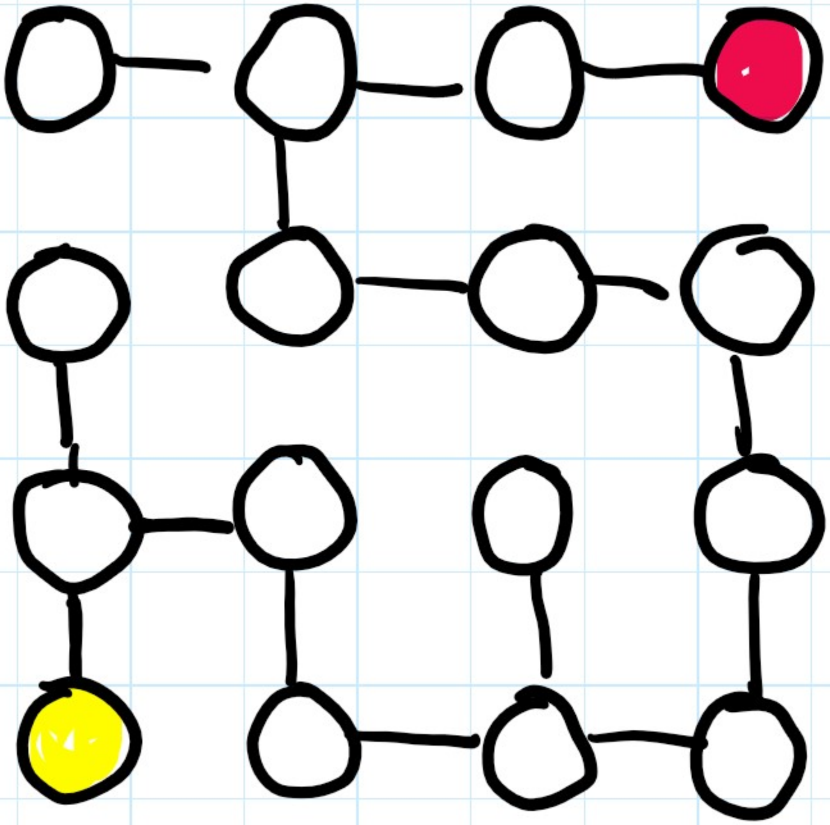
END

For more discussion on this, see the lecture recording

Representing mazes as graphs

Sunday, October 23, 2022 4:54 PM



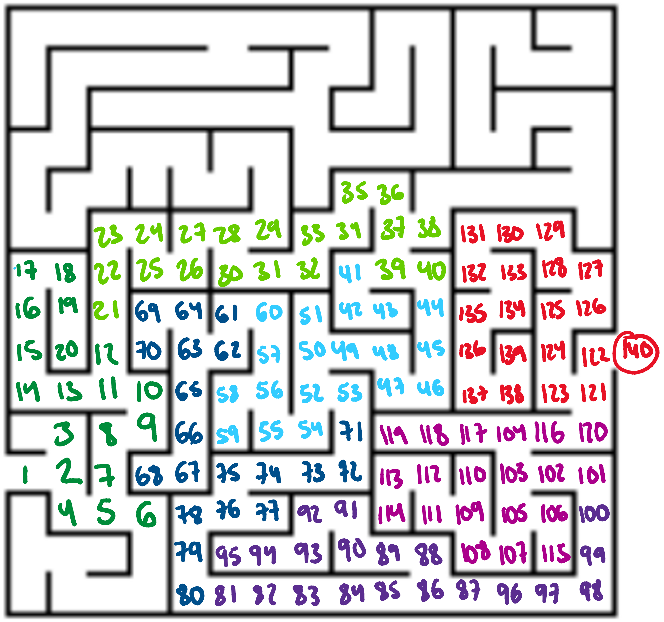


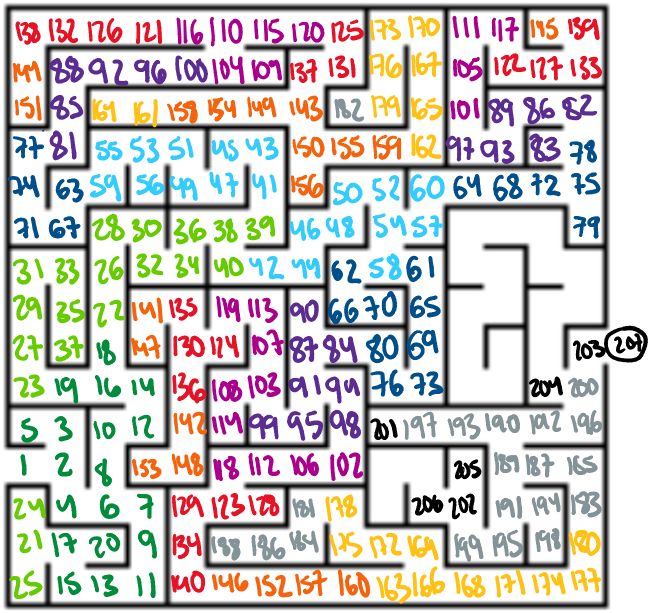
Solving the maze = finding route (DFS or BFS) from vertex that represents starting cell to vertex that represents ending cell

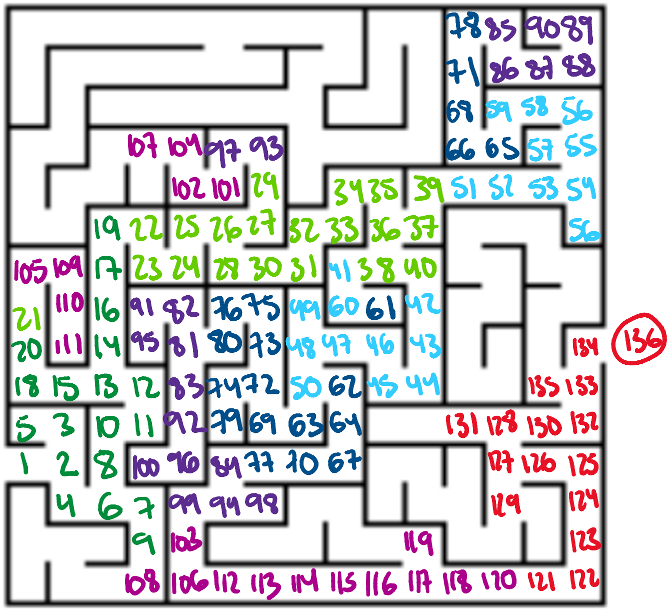
(Attribution: Pages 3-4 drawn by Milda from Fall 2022 version)

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.