Maze Solving

CMPUT 355: Games, Puzzles, and Algorithms

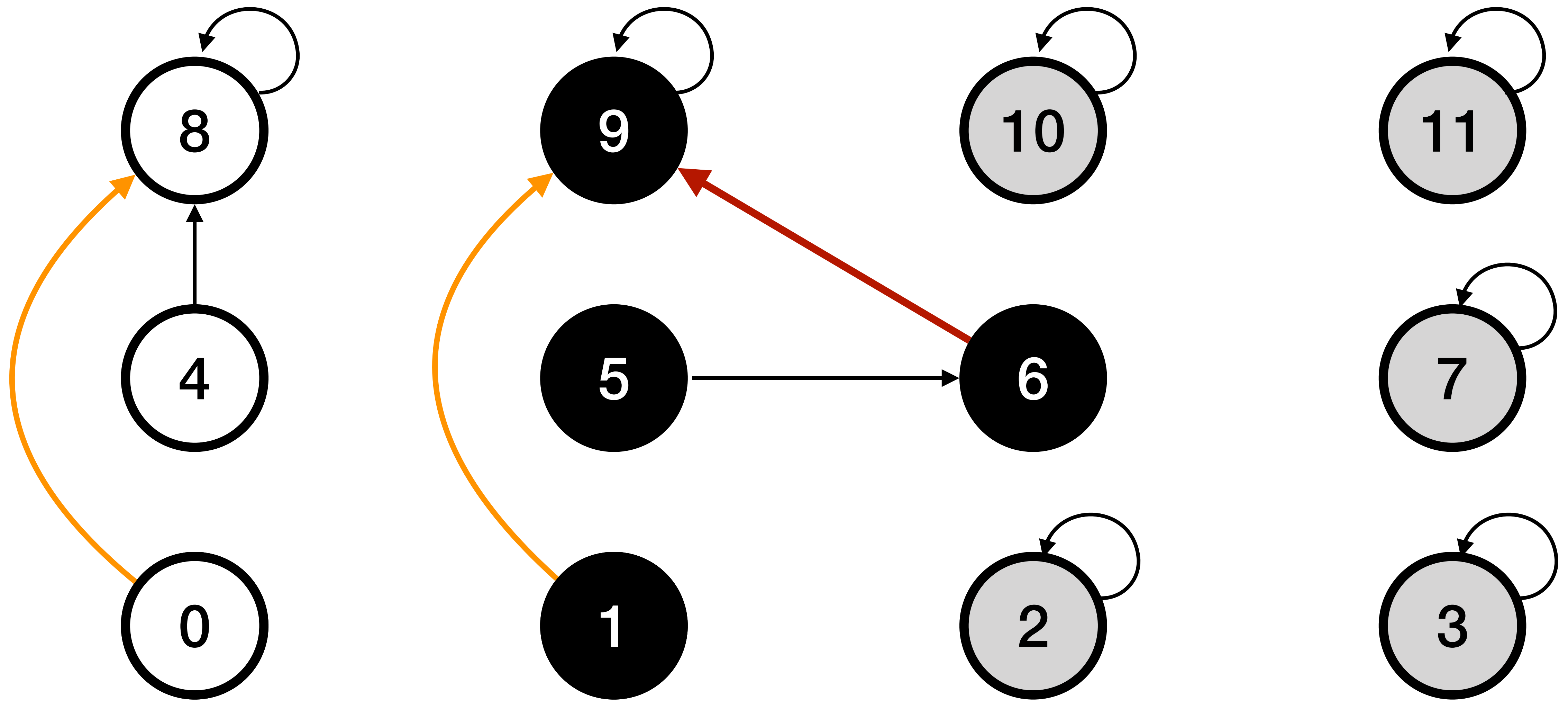
Lecture Outline

1. Logistics & Recap
2. Maze puzzle
3. Depth-first and breadth-first search

Logistics

- **Practice quiz questions:** Posted last Friday
  - Answers released tomorrow evening (Tue Jan 20)
- **Quiz 1:** This Friday, **Jan 23**
  - In-class, full 50 minutes
  - No need to email if you have to miss it; up to 3 replaced by final exam automatically
  - Questions will be very similar to practice questions

Recap: Union-Find Operations in Go



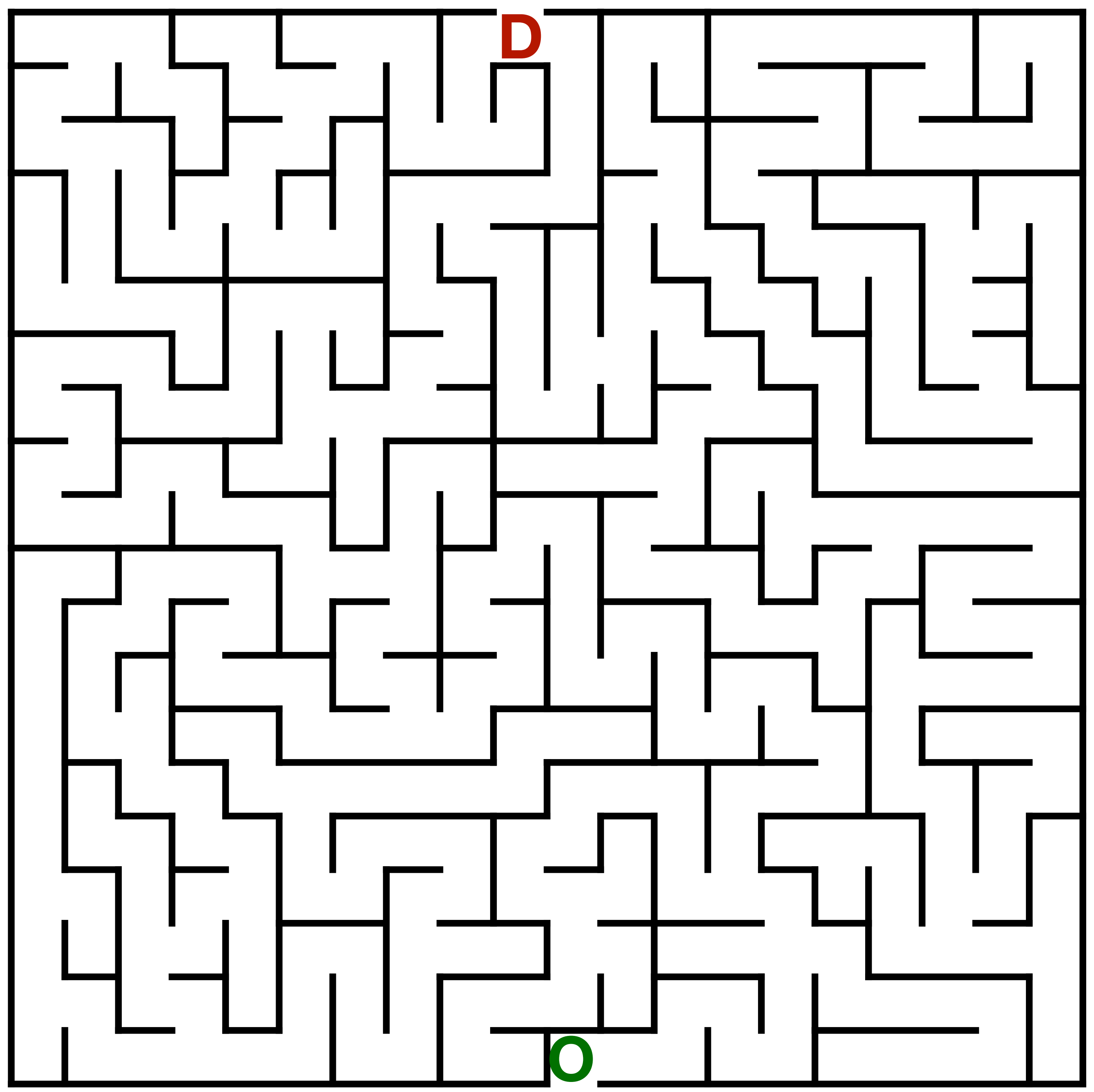
After 5 is placed, we check neighbours 6, 9, 4, 1 for merge operations

1. Join 5's block (i.e., 5) to o 6's block, so it points to 6's block's root (i.e., 6)
2. Next, join 5's block (i.e., 6) and 9's block (i.e., 9).
3. Without the union-by-rank optimization, that could mean *either* 6 points to 9 or 9 points to 6; I chose to show 6 points to 9.
4. But with the union-by-rank optimization, we would have to make 9 point to 6, because it has strictly lower rank than 6.

- Black stone on 6
- White stone on 8
- Black stone on 9
- White stone on 4
- Black stone on 5
- White stone on 0
- Black stone on 1
    - 0's block's liberties become empty

Maze Solving

- A maze is a grid of positions
  - One is the start position, one is the goal position
  - A maze is **solved** by a path from start to goal
- A cell's **neighbours** are the cells immediately above, below, left, or right of the cell *that are not blocked by a wall*
- Try to solve this maze!
  - **Question:** What algorithm did you follow?



Algorithm: Random Walk

Simplest idea that could possibly work:

- Start at the origin
- Move to random neighbour
- Keep going until you reach the destination

```
current := origin
while current != destination:
    current := random neighbour of current
    print current
```

Questions:

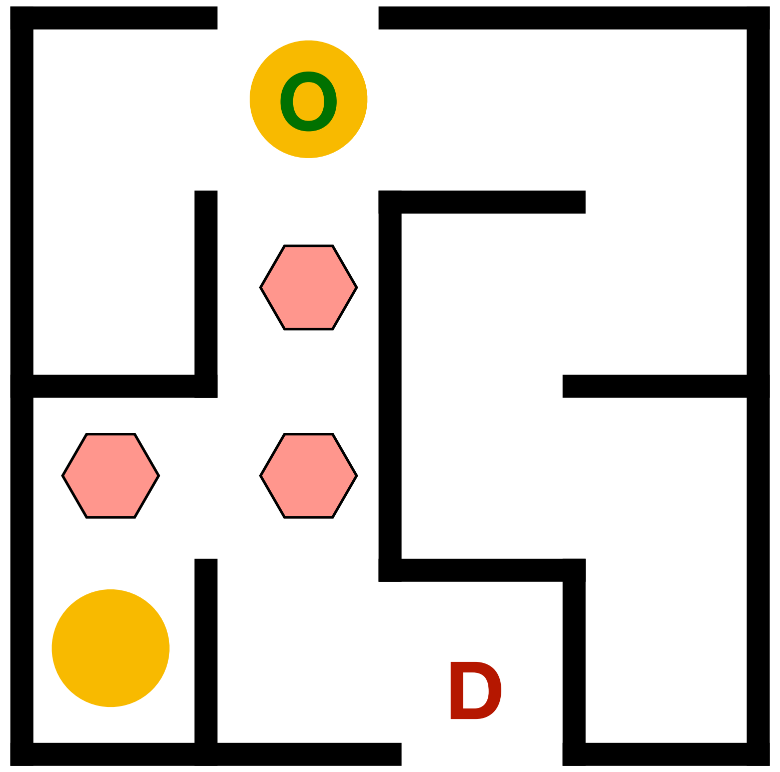
1. Is this algorithm **guaranteed** to find a path to the destination? Why or why not?
   - What is the probability of finding a t-length solution in exactly t steps?
   - What is the probability of looping forever (**never** finding the solution)?
2. Is this algorithm **time efficient**?
   - Pick arbitrary $n > t$. What is (a lower bound on) the probability of finding a solution in exactly n steps?
3. Is this algorithm **space efficient**?
4. What is an **easy improvement**?

Algorithm: Random Walk of No Return

What if we never returned to a previous position?

```
current := origin
visited := {origin}
while current != destination:
    next := random neighbour of current
    if next not in visited:
        current := next
        add current to visited
        print current
```

Question: Is this algorithm **guaranteed** to find a path to the destination? Why or why not?



Algorithm: No Return and Retry

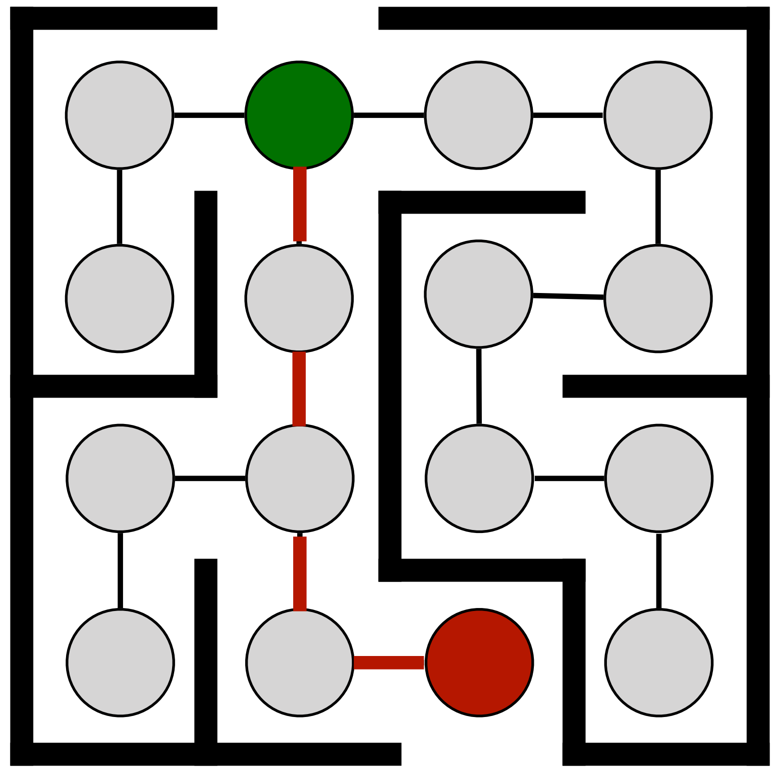
- Need to guarantee that every path is **eventually** explored (**why?**)
- Recursive search starting from a random neighbour
- If that doesn't work, do a recursive search from a different random neighbour!
- Implemented in maze/rmaze-class.py
- **Demo!**

```
def rwander(self,psn): # recursive wander
  here_ch = self.char_at(psn)
  assert(here_ch == empty_ch or here_ch == origin_ch)
  if here_ch == empty_ch:
    self.mark_location(psn, current_ch)
    self.showpretty() # print maze, so we can watch the traversal
    self.mark_location(psn, seen_ch)
  for shift in nbr_offsets:
    # to move from a grid-point to a neighboring grid-point,
    #    add the associated shift value to the row and column values
    #    e.g. adding shift (0, -1) to point (3, 4) moves to point (3,3)
    new_psn = psn[0]+shift[0], psn[1]+shift[1]
    new_ch = self.char_at(new_psn) # examine new_psn
    if new_ch == dest_ch:
      self.showpretty() # print maze, so we can watch the traversal
      return new_psn
    if new_ch == empty_ch:
      rec = self.rwander(new_psn)    # recursively traverse from new_psn
      if rec is not None:            # did recursive call find exit?
        return rec                   # yes? rwander(self,psn) terminates
```

Maze Solving as Graph Search

We can represent our problem as a **graph search**

- Every **cell** is a **node** in the graph
- Draw an **edge** between every pair of **neighbours**
- A **solution** is a **path** from the origin node to the destination node



Depth-First and Breadth-First Search

- The "No Return and Retry" algorithm is a special case of depth-first search
- Generic search algorithm:

```
fringe = { origin }
seen = { origin }
while fringe not empty:
    cur := remove node from fringe
    for each neighbour n of cur:
        if n is destination:  return n
        if n not in seen:
            add n to seen
            add n to fringe
```

Questions:

1. Is this algorithm **guaranteed** to find a path to the destination? Why or why not?
2. What is this algorithm's **worst-case time complexity**?
3. What **data structure** should we use for the fringe for **depth-first search**?
4. What **data structure** should we use for the fringe for **breadth-first search**?

- **Depth-first search** if we visit most recent neighbours first
- **Breadth-first search** if we visit all of one node's neighbours before moving on to others

DFS/BFS implementation: **maze/maze.py**

```
def __init__(self):
  self.lines = []
  for line in stdin:
    self.lines.append(line.strip('\n'))
  self.rows, self.cols = len(self.lines), len(self.lines[0])
  for j in range(1,self.rows-1):
    assert (self.cols == len(self.lines[j])) # each maze line has same len
    for line in self.lines:
      assert((line[0]==wall_ch and (line[self.cols-1]==wall_ch)))
      # top and bottom of maze must be solid wall
    for j in self.lines[0]: assert(j == wall_ch) # left wall not solid
    for j in self.lines[self.rows-1]: assert(j == wall_ch) # rt wall not solid
```

```
nbr_offsets = [(0,-1), (0,1), (-1,0), (1,0)]
  # python has row index, then column index, so neighbors processed
  #   in order left, right, up, down
```

```
def wander(self):
  psn = self.find_start()
  fringe = deque()
  fringe.append(psn)
  while len(fringe) > 0:
    # comment out one of these two lines
    #psn = fringe.pop() # pop from end of list, LIFO, stack, so DFS
    psn = fringe.popleft() # pop from front, FIFO, queue, so BFS
    if self.char_at(psn) != orgn_ch:
      self.mark_location(psn,done_ch)
      self.showpretty()
    for shift in nbr_offsets:
      new_psn = psn[0]+shift[0], psn[1]+shift[1]
      new_ch = self.char_at(new_psn)
      if new_ch == dest_ch: return new_psn
      elif new_ch == empt_ch:
        fringe.append(new_psn)  # append to end of list
        self.mark_location(new_psn,seen_ch)
        self.showpretty()
```

Questions:

1. Why do we not have a "seen" variable in this implementation?
2. Why aren't we doing any bounds checking for **new_psn**?

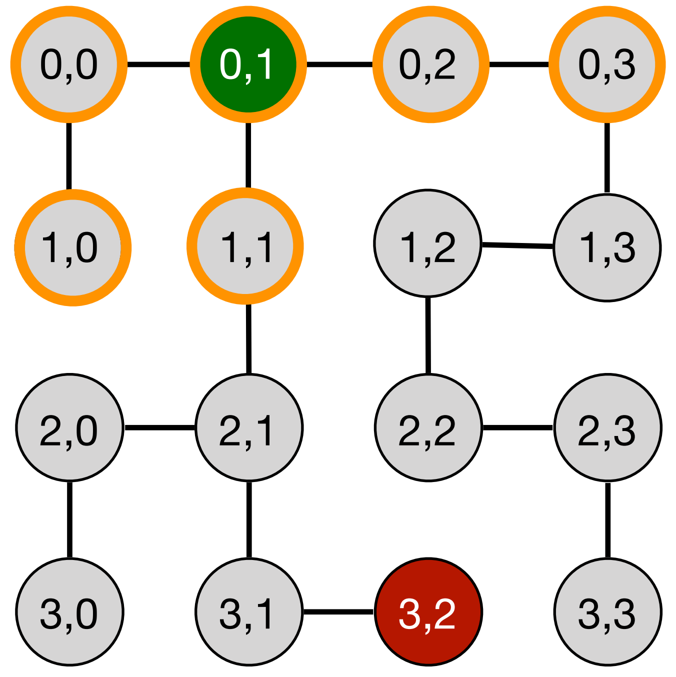
Example trace

```
nbr_offsets = [(0,-1), (0,1), (-1,0), (1,0)]
   # python has row index, then column index, so neighbors processed
   #   in order left, right, up, down
```

```
def wander(self):
  psn = self.find_start()
  fringe = deque()
  fringe.append(psn)
  while len(fringe) > 0:
    # comment out one of these two lines
    #psn = fringe.pop() # pop from end of list, LIFO, stack, so DFS
    psn = fringe.popleft() # pop from front, FIFO, queue, so BFS
    if self.char_at(psn) != orgn_ch:
      self.mark_location(psn,done_ch)
      self.showpretty()
    for shift in nbr_offsets:
      new_psn = psn[0]+shift[0], psn[1]+shift[1]
      new_ch = self.char_at(new_psn)
      if new_ch == dest_ch: return new_psn
      elif new_ch == empt_ch:
        fringe.append(new_psn)  # append to end of list
        self.mark_location(new_psn,seen_ch)
        self.showpretty()
```

Initially: add (0,1) to fringe — fringe = [(0,1)]

- Remove (0,1) — fringe = []
  - add (0,0) — fringe = [(0,0)]
  - add (0,2) — fringe = [(0,0), (0,2)]
  - add(1,1) — fringe = [(0,0), (0,2), (1,1)]
- remove (0,0) — fringe = [(0,2), (1,1)]
  - don't add (0,1) (**why?**) — fringe = [(0,2), (1,1)]
  - add (1,0) — fringe = [(0,2), (1,1), (1,0)]
- remove (0,2) — fringe = [(1,1), (1,0)]
  - don't add (0,1) — fringe = [(1,1), (1,0)]
  - add (0,3) — fringe = [(1,1), (1,0),(0,3)]
- ...



Summary

- **Maze puzzles:** Find a path from origin cell to a destination cell
- Completely random exploration is guaranteed to find it eventually
    - ...but can be arbitrarily slow
- Can straightforwardly represent as a graph
- **Depth-first search** and **breadth-first search** systematically search the graph
    - guaranteed to find the destination
    - might have to search **entire graph**
    - will never have to search **more** than the entire graph