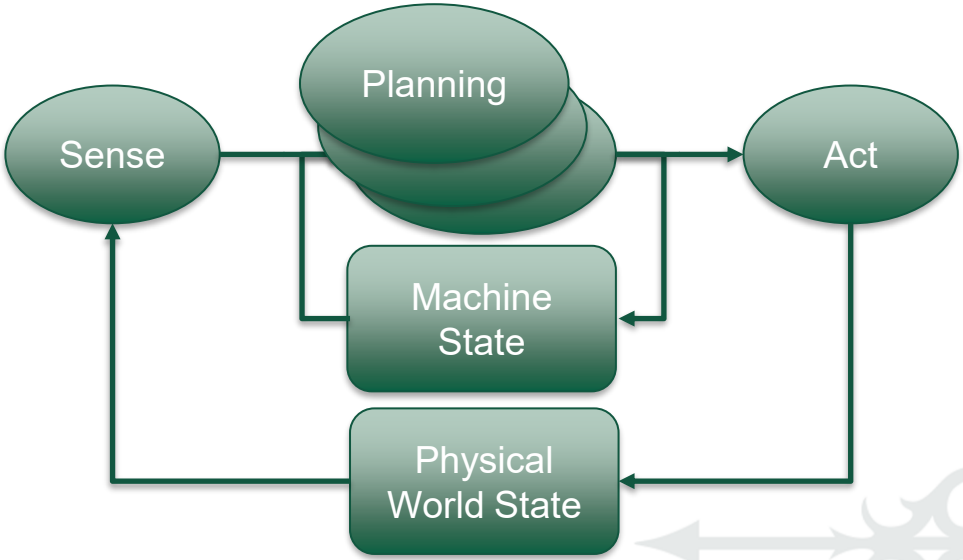
Graph-Based Planning

CSCI 420-04 Robotics

WILLIAM & MARY

CHARTERED 1693

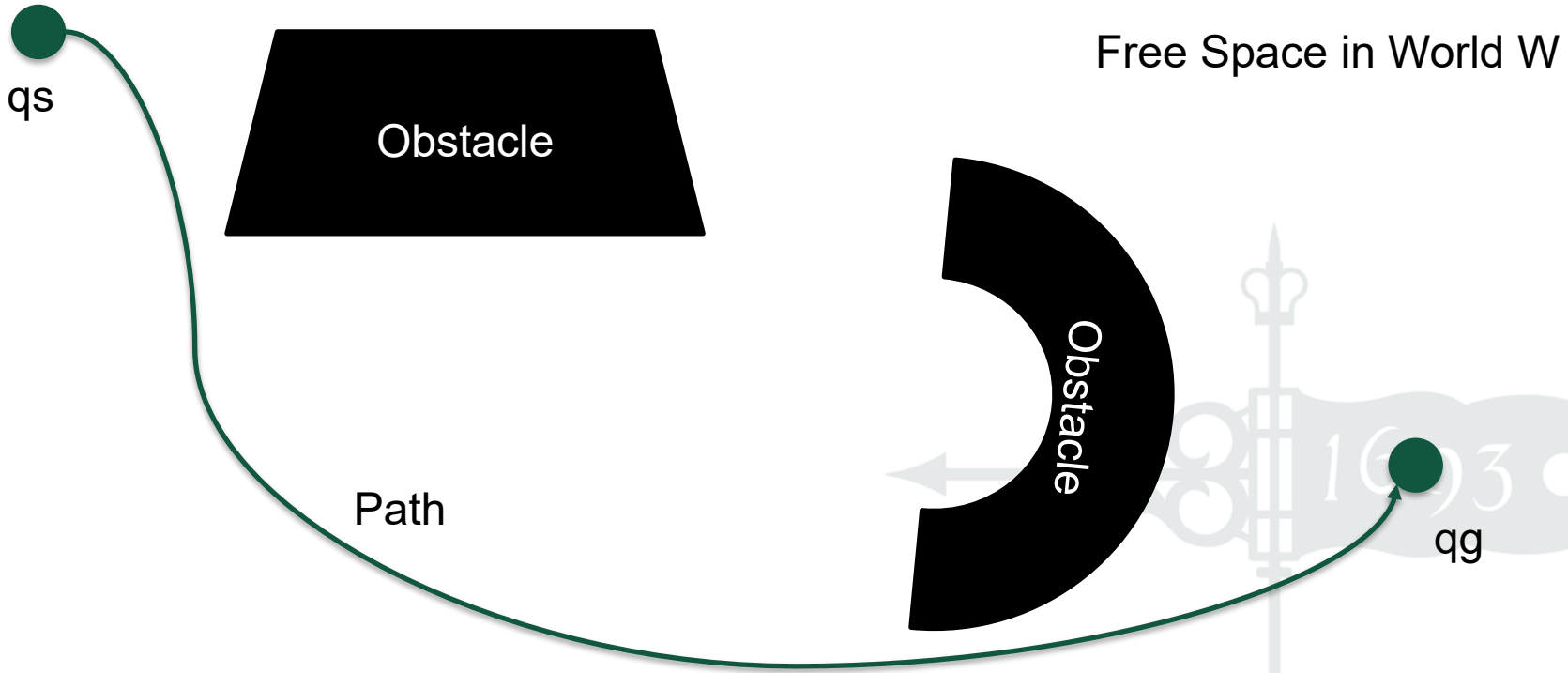
How do we choose the action?



2D Motion Planning

- Given:
  - World Space W
  - Obstacle Areas O
  - Robot State R
  - Start qs
  - End qg
- Find:
  - Path from qs to qg
  - Inside of W
  - Avoiding O
  - (with efficiency)

Motion Planning



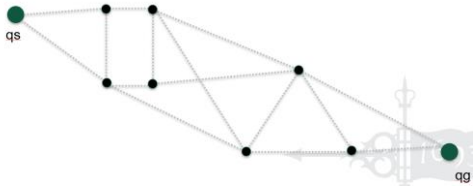
Model Planning Families

- Reactive
  - Bug family
  - Dynamic Window
- Model-based
  - Visibility
  - Grid
  - Probabilistic

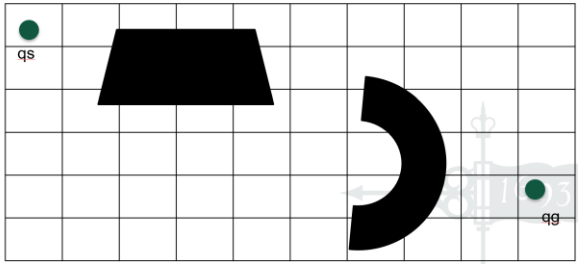
The right choice depends on the assumptions about sensor types and the available world models

Model-Based Approach: Graph

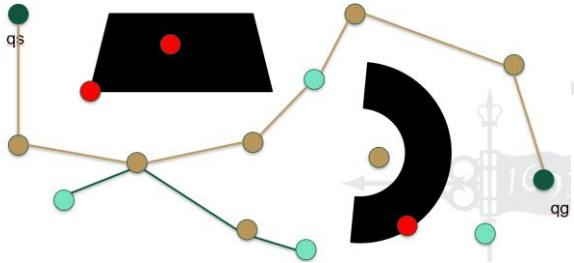
Path Planning: Visibility



Path Planning: Grid



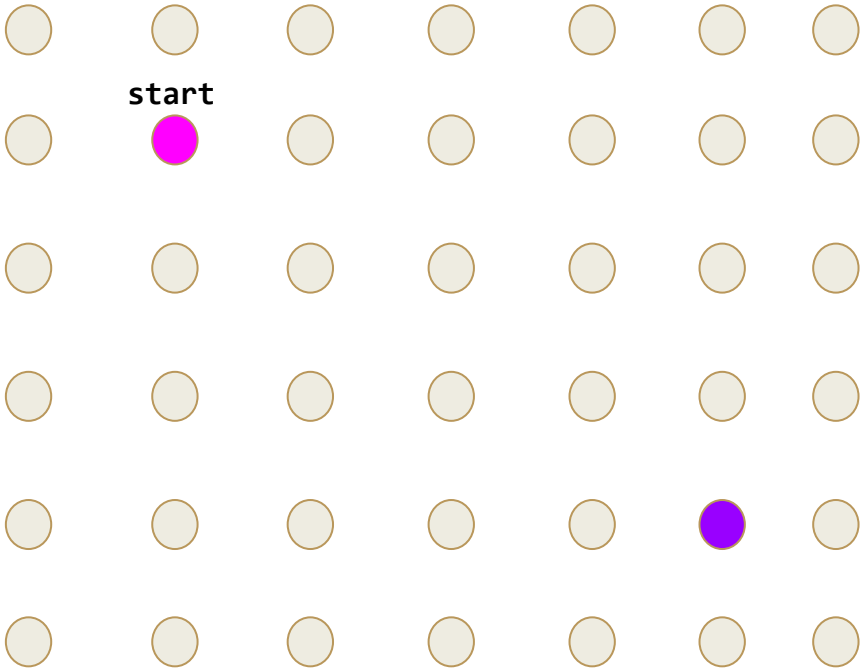
Path Planning: Probabilistic



Finding Graph Paths

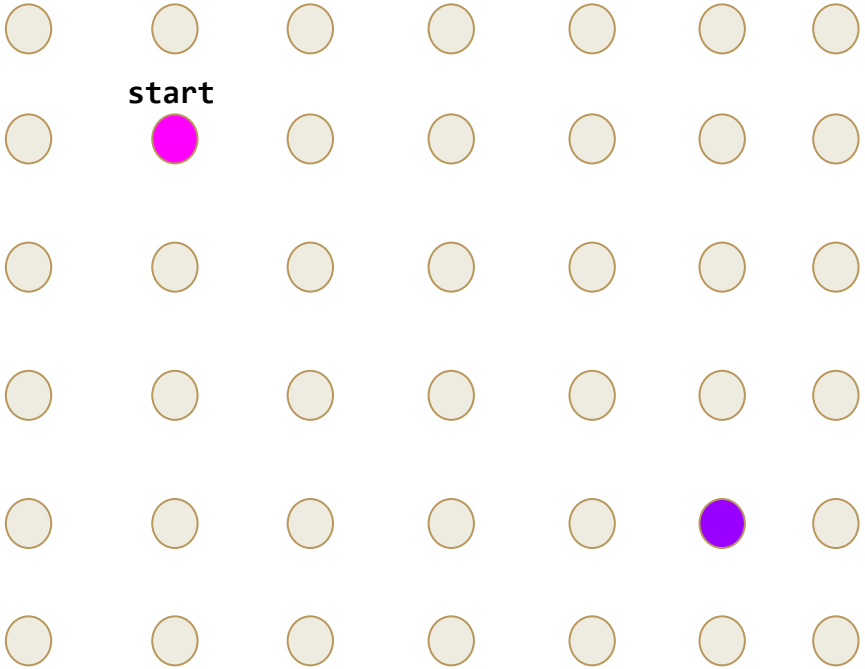
- General Algorithms
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
- Informed
  - “Heuristic” to guide the search

Finding the Goal: BFS



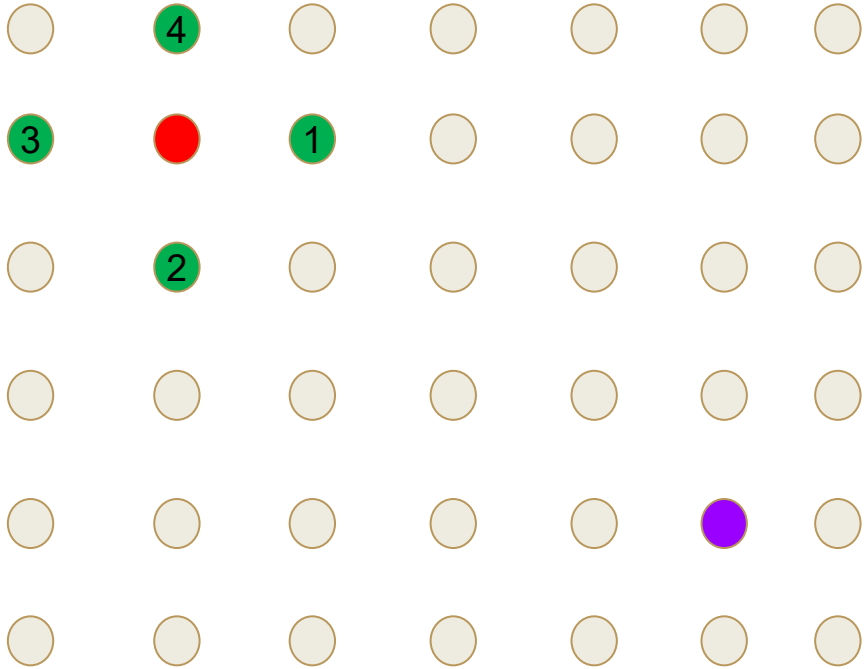

```
frontier = Queue()
frontier.push(start)
```

Finding the Goal: BFS



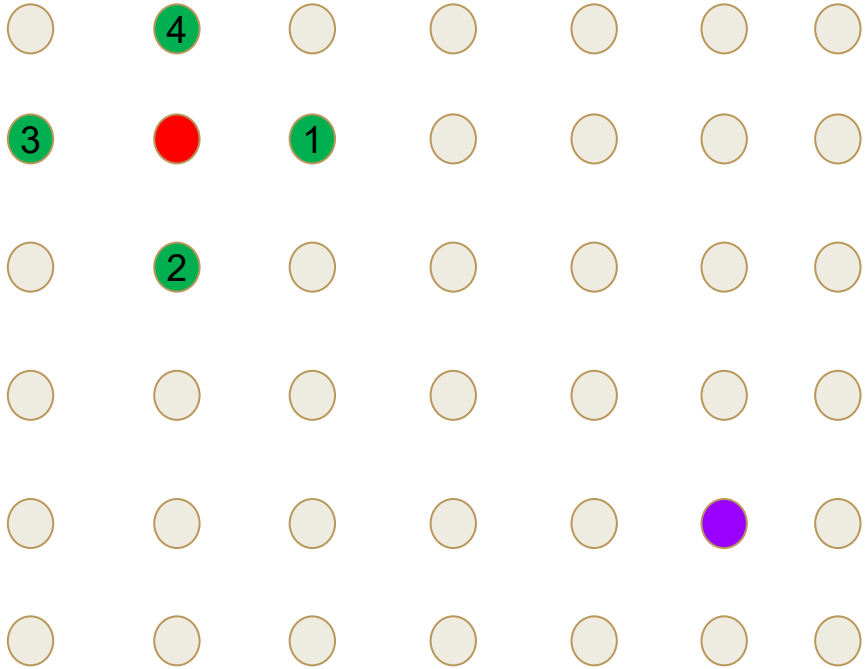

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
```

Finding the Goal: BFS



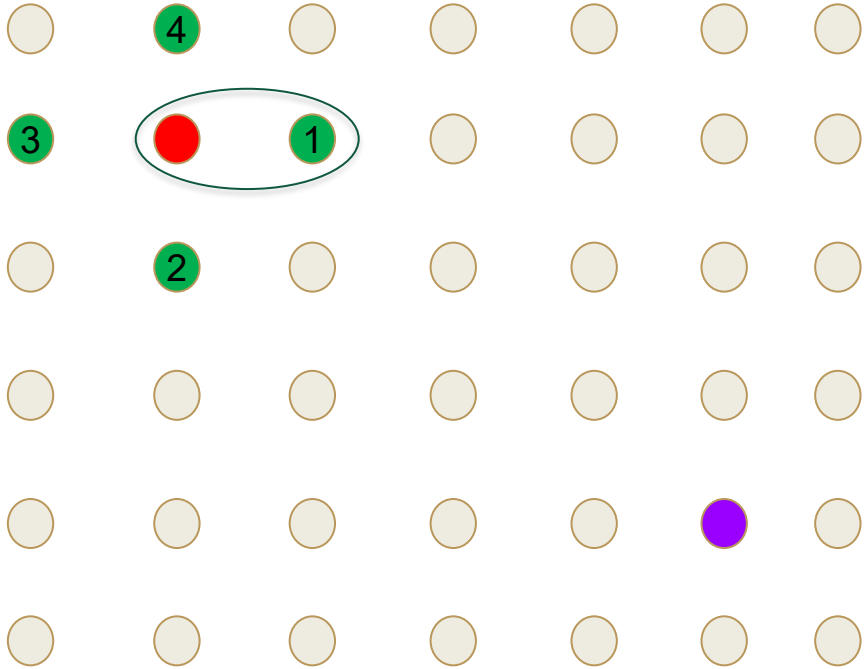

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
```

Finding the Goal: BFS



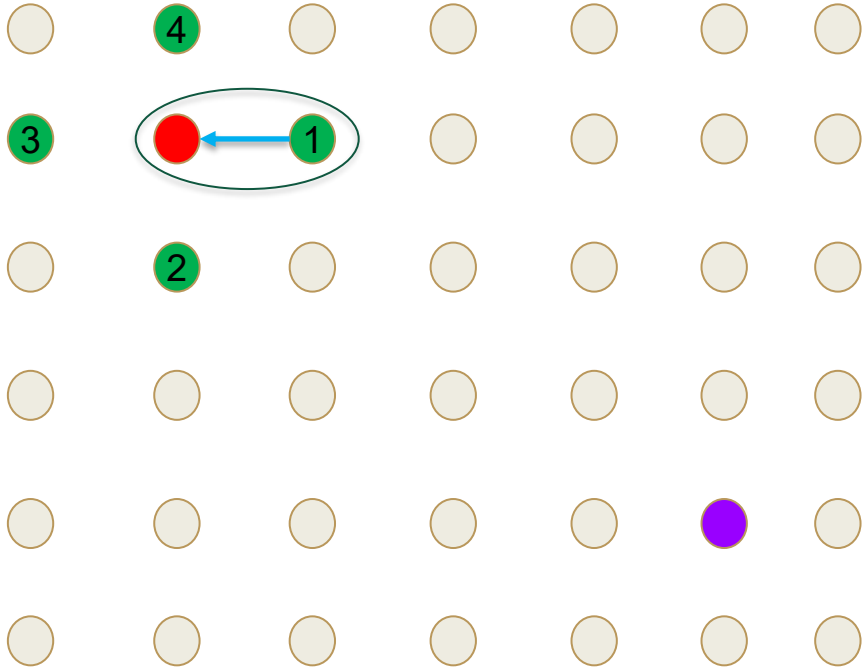

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
```

Finding the Goal: BFS



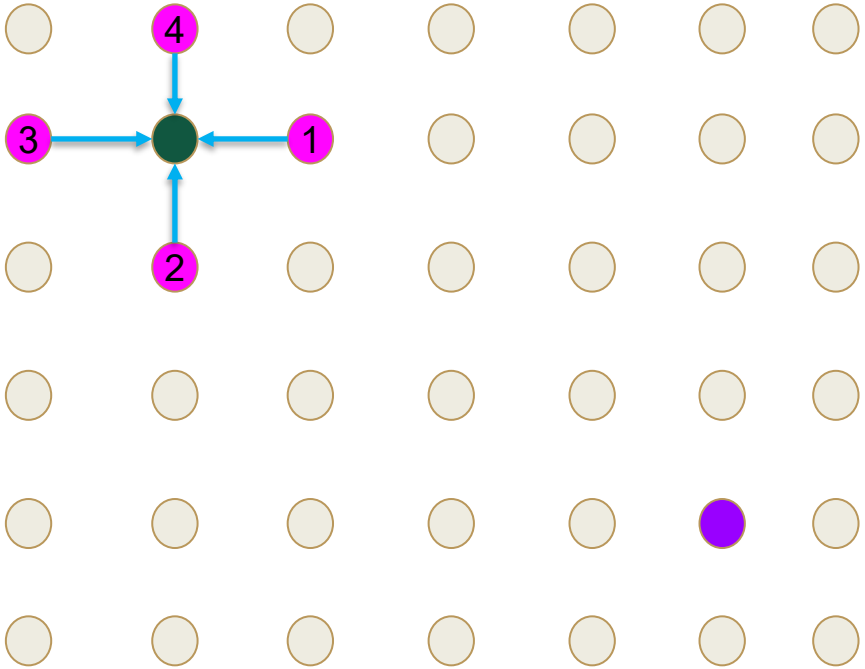

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
```

Finding the Goal: BFS



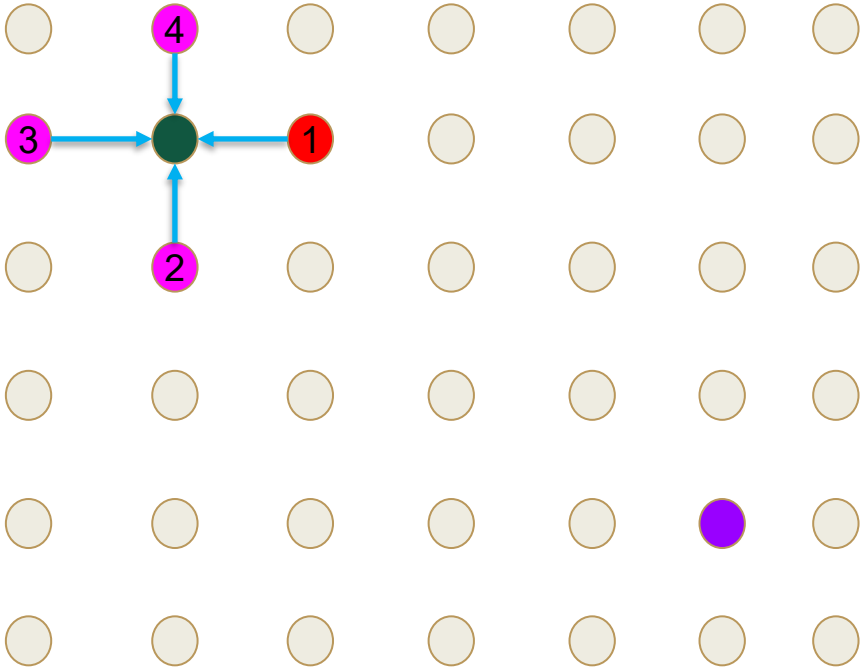

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



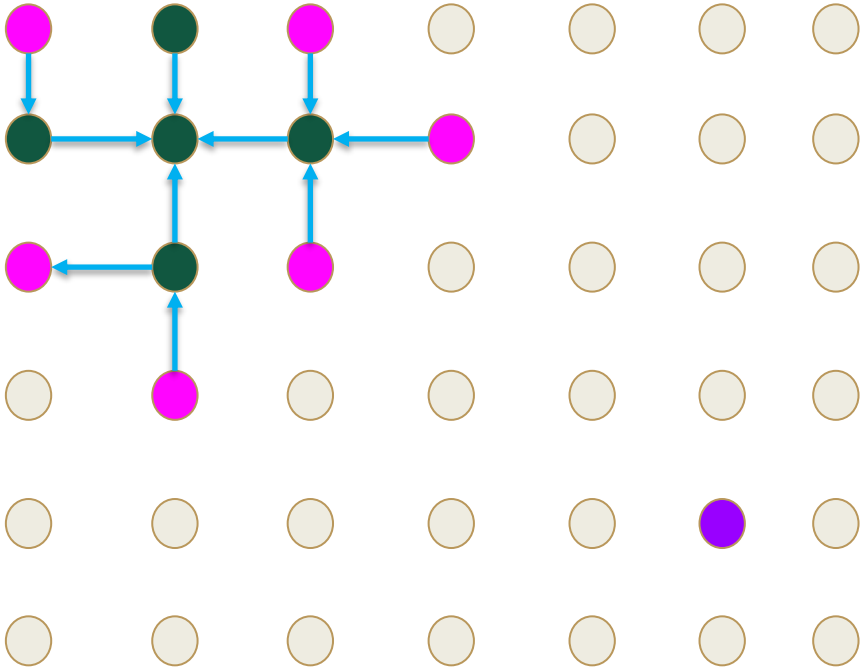

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



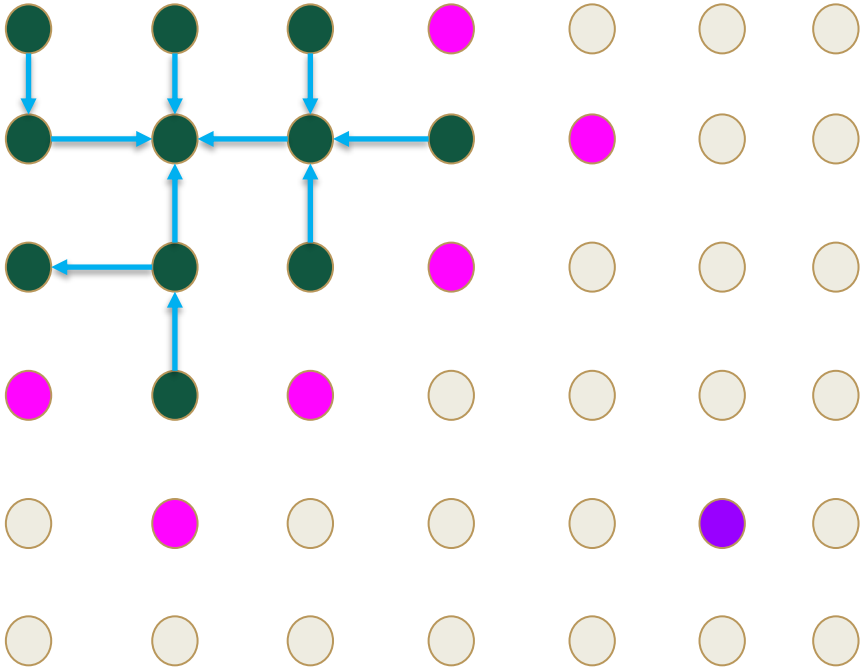

```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



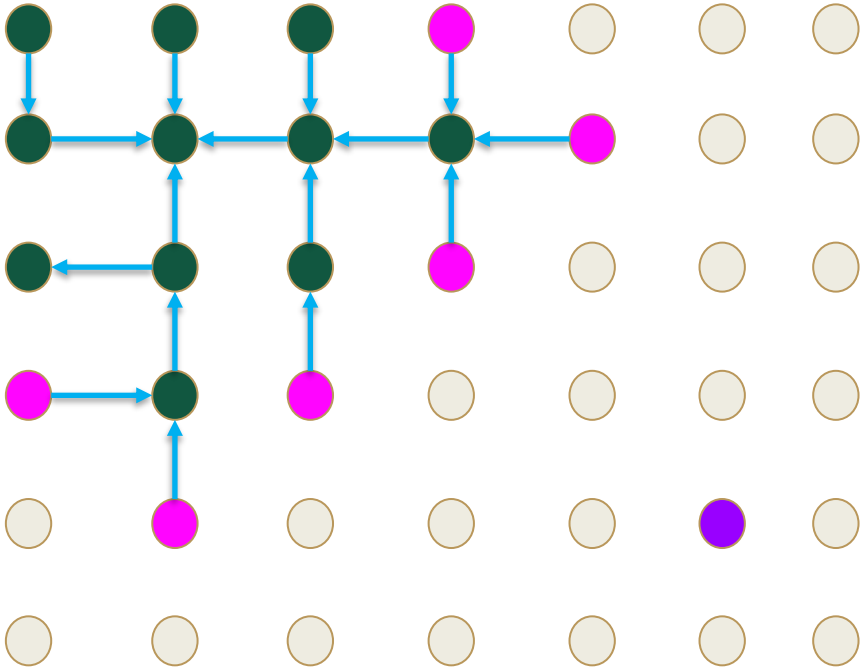
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



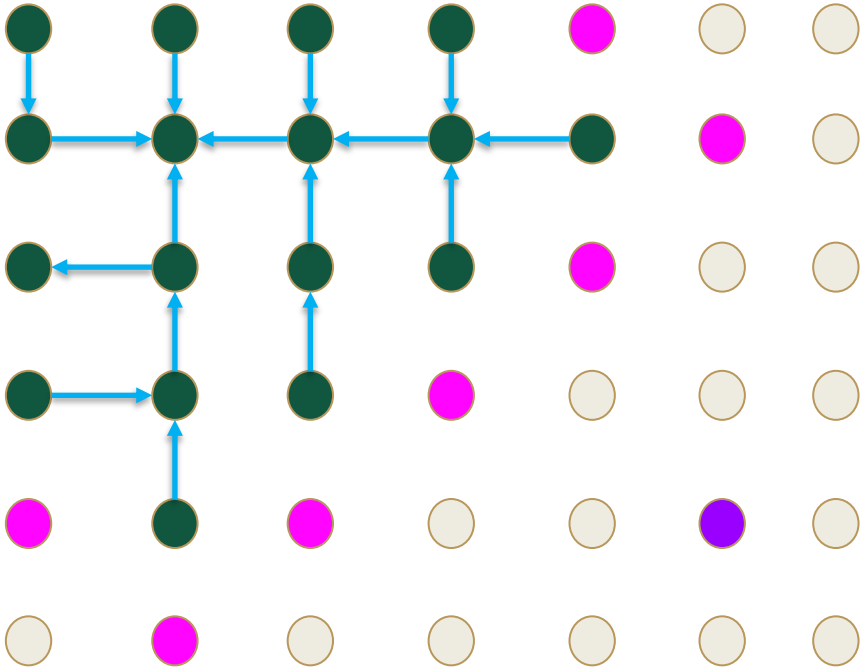
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



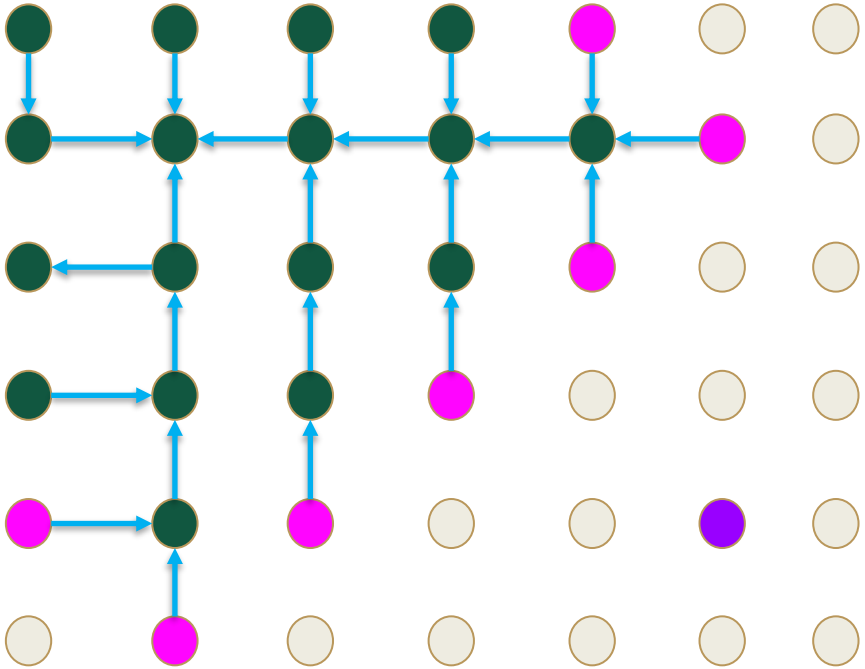
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



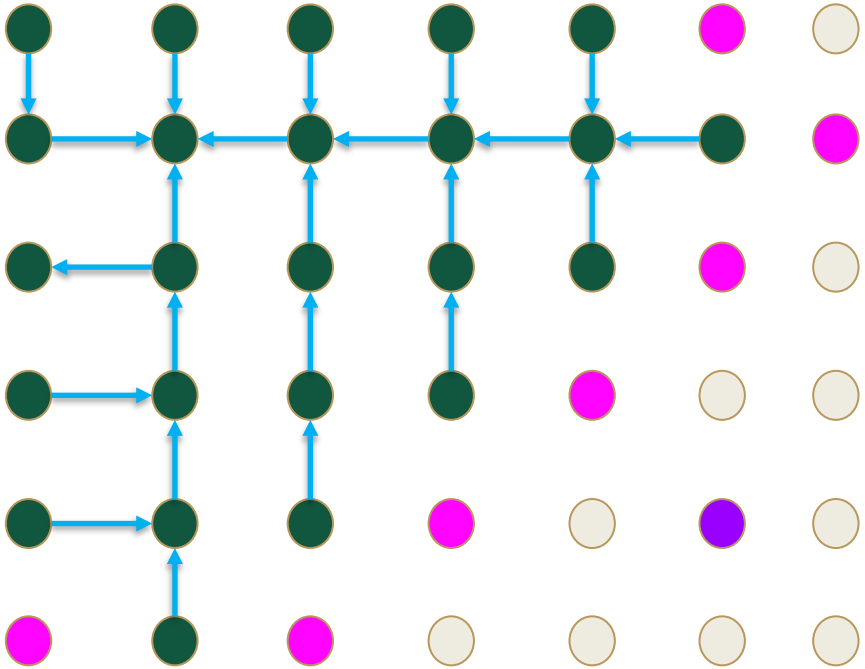
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



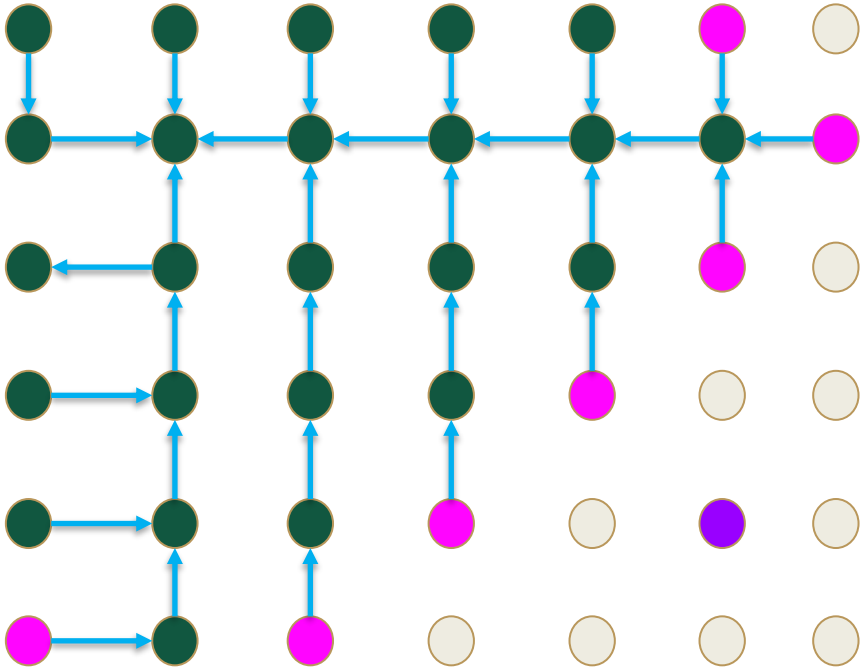
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



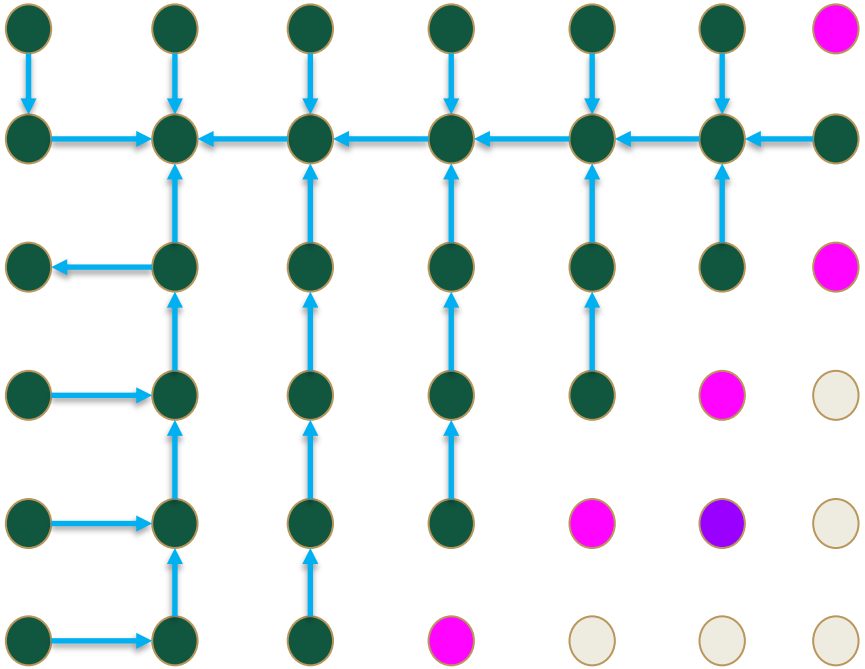
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



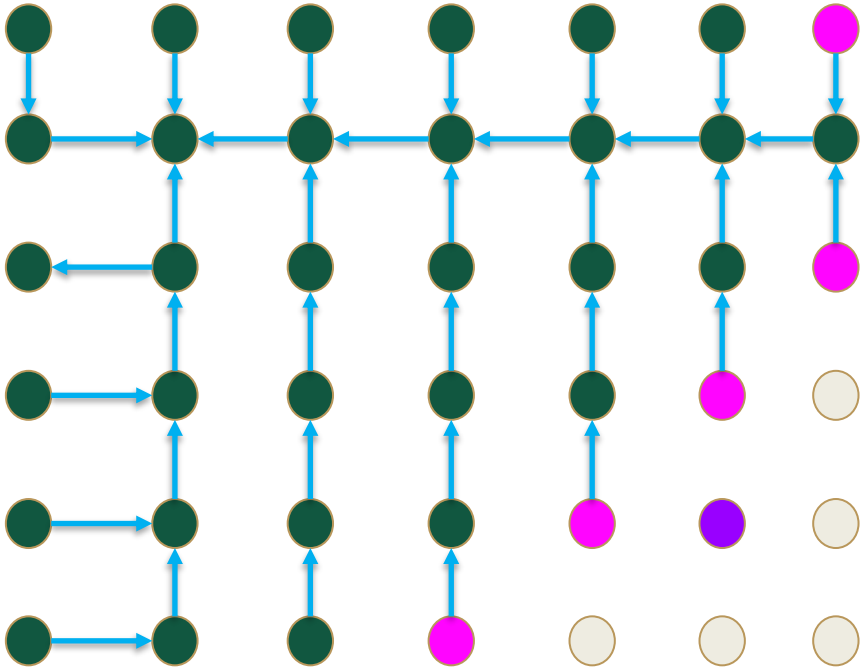
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



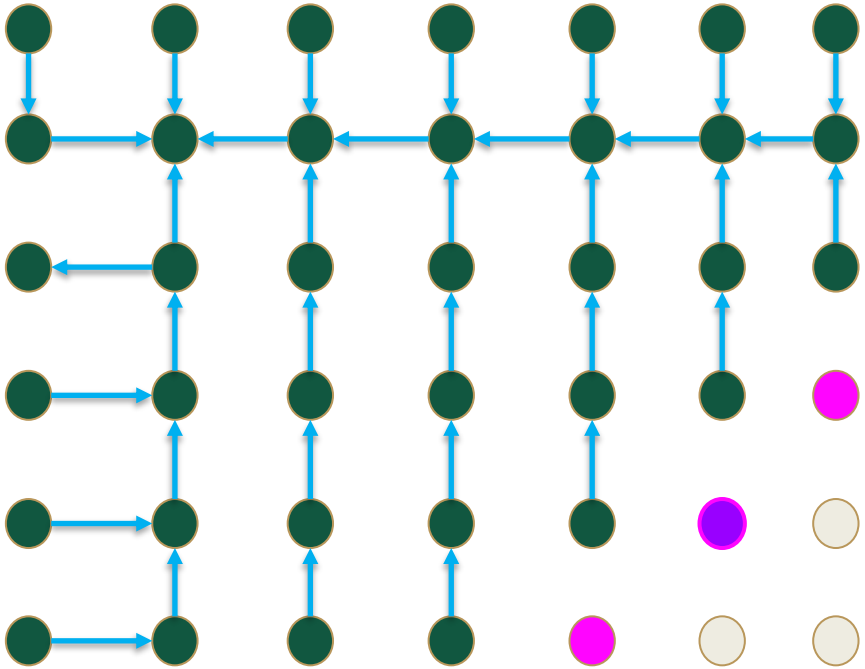
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



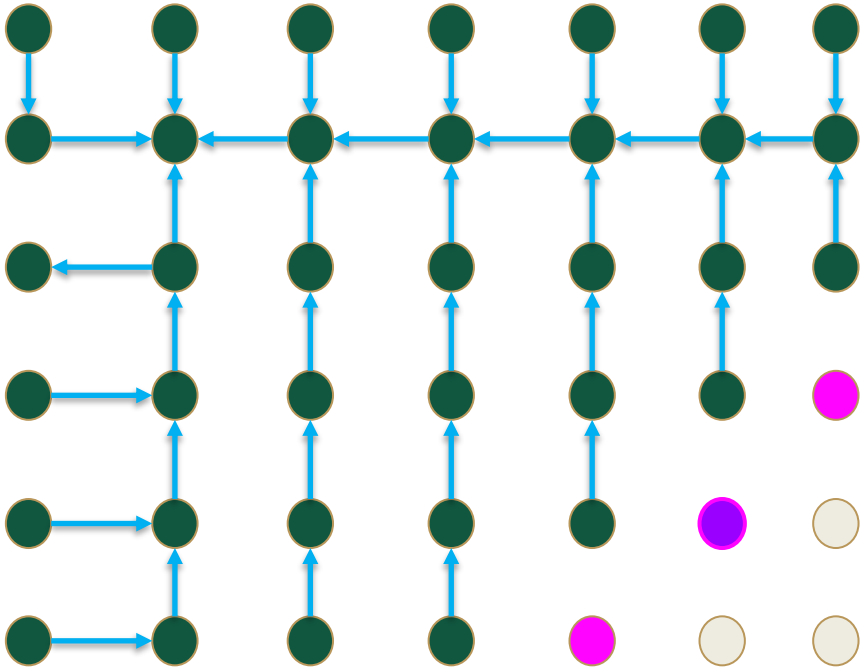
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



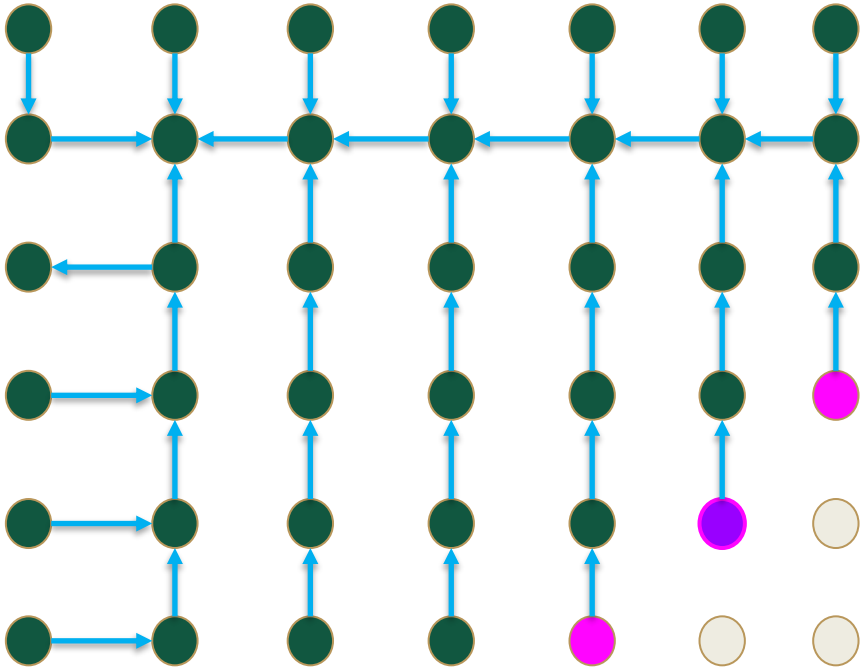
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



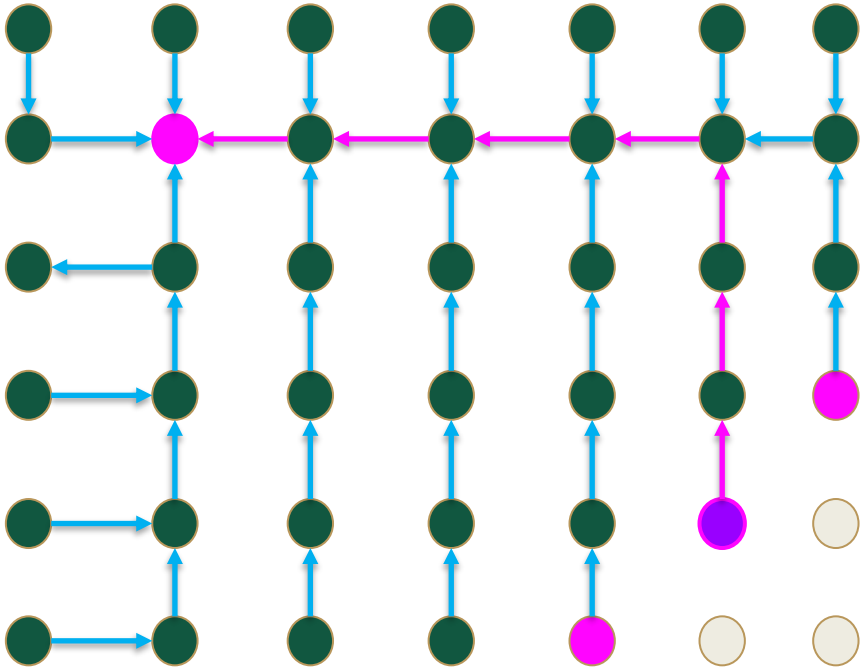
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



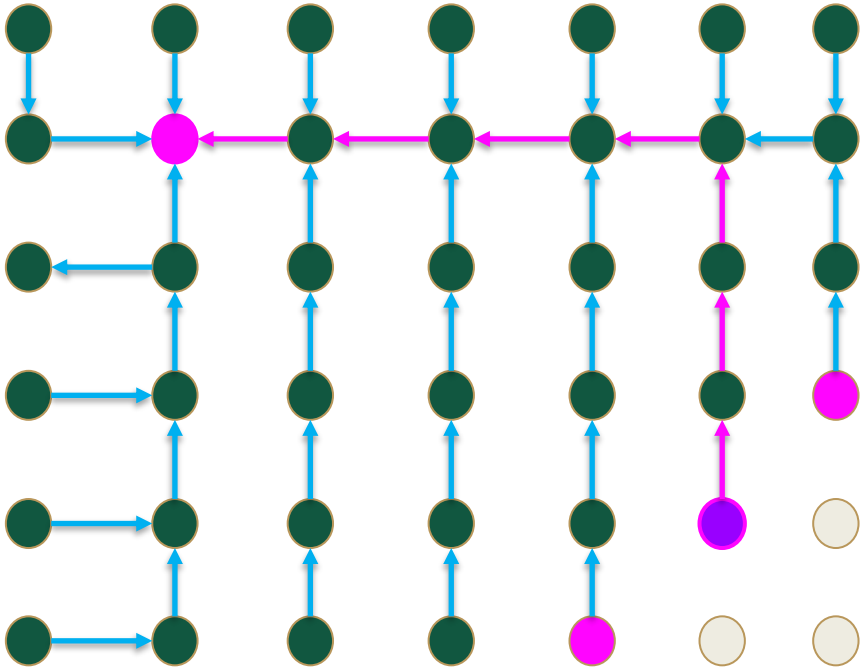
```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.get()
  if cur == goal:
    break
  for next in cur.neighbors():
    if next not in came_from:
      frontier.put(next)
      came_from[next] = cur
```

Finding the Goal: BFS



```
frontier = Queue()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  …
path = []
while cur != start
  path.append(cur)
  cur = came_from[cur]
path.append(start)
return path.reverse()
```

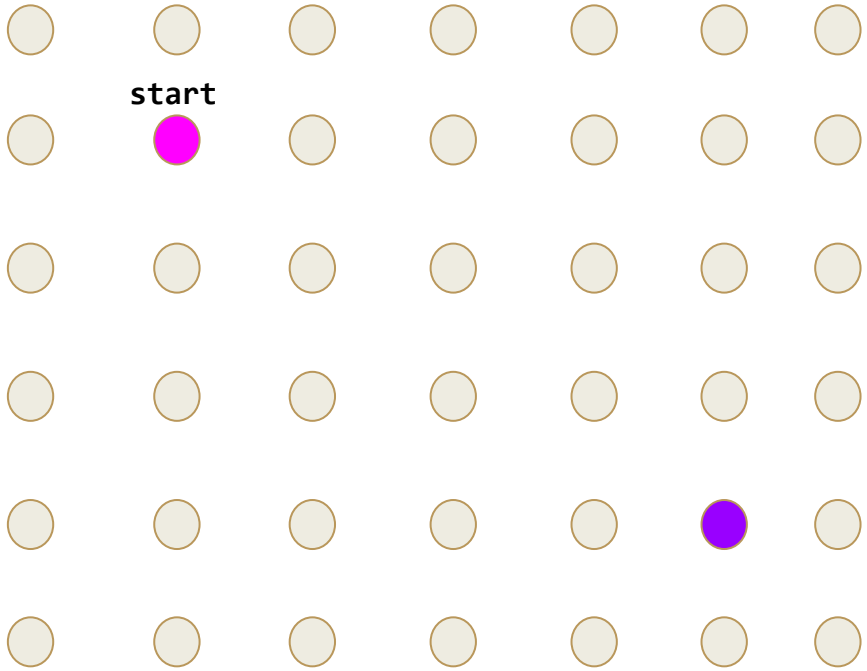
Finding the Goal: BFS



```
frontier = Queue()
frontier.push(start)
came_from = {}
while cur != start
  path.append(cur)
  cur = came_from[cur]
path.append(start)
return path.reverse()
```

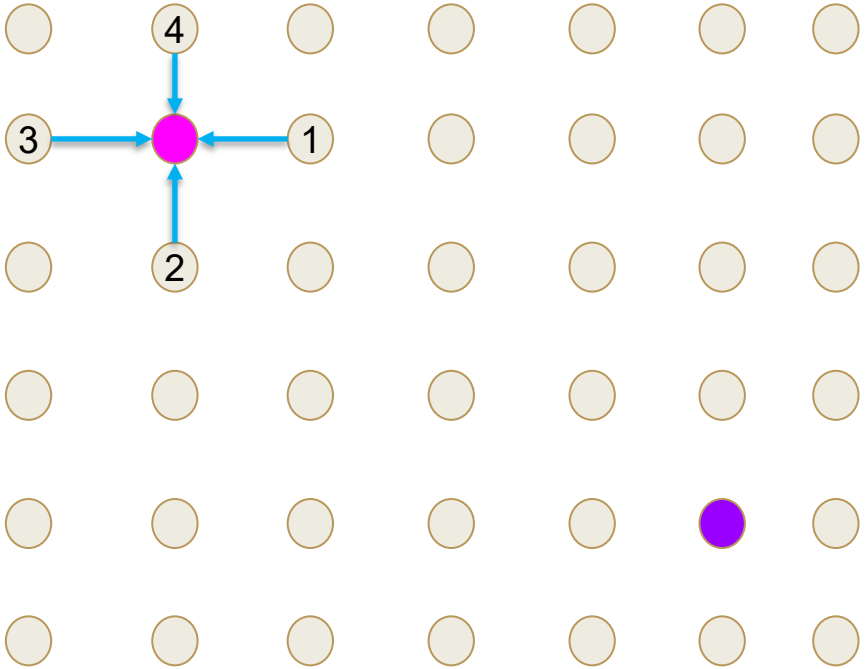
BFS will find a path, but will explore many nodes first

Finding the Goal: DFS



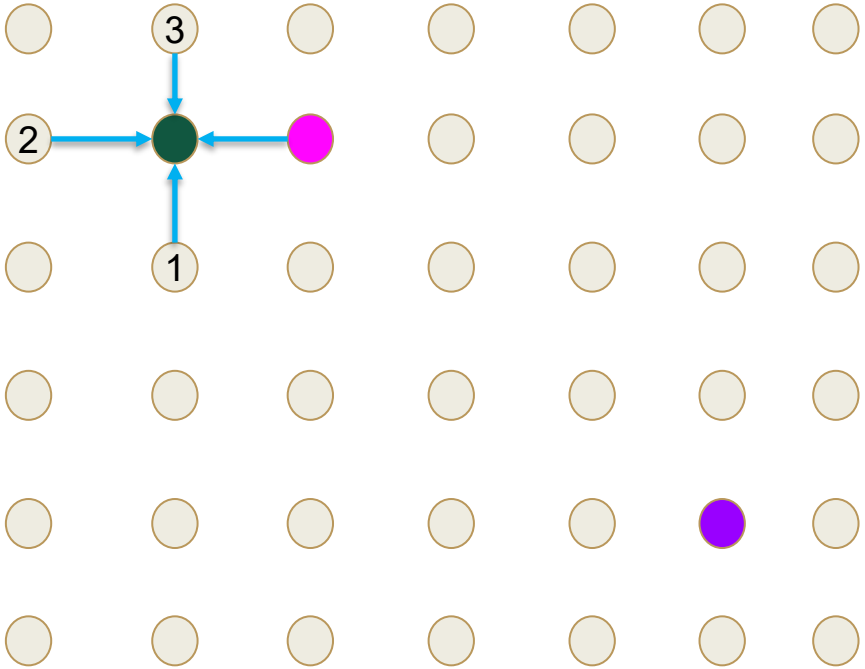

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



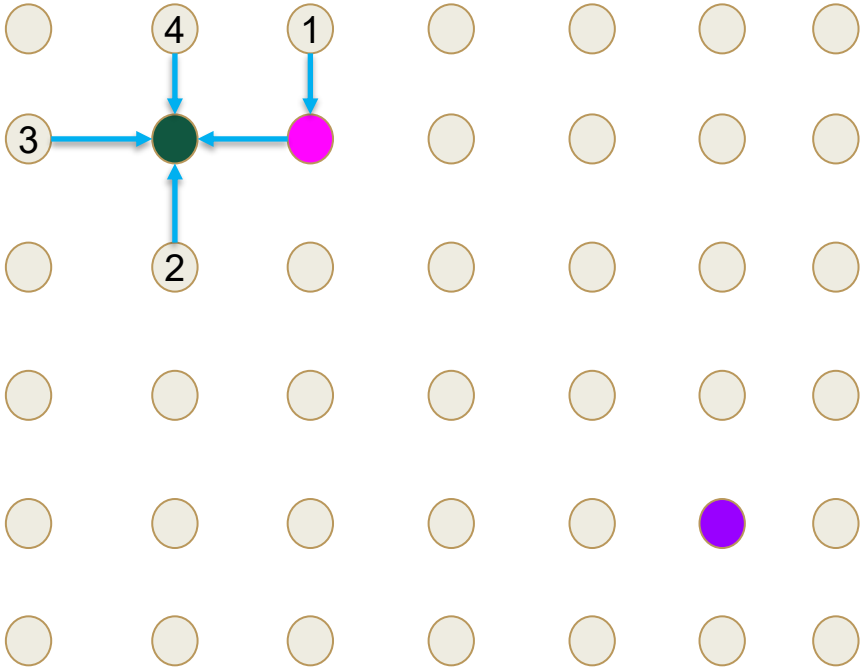

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



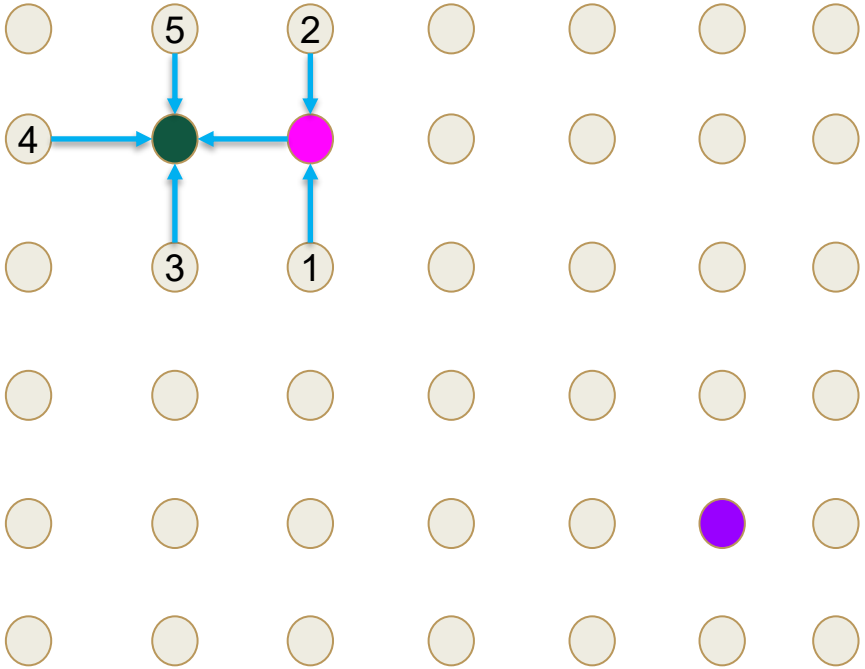

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



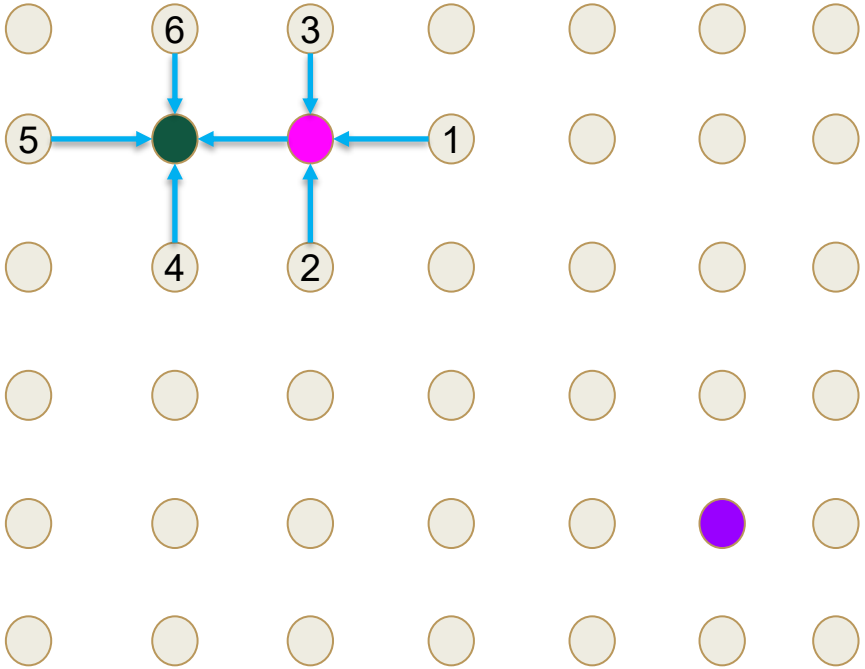

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



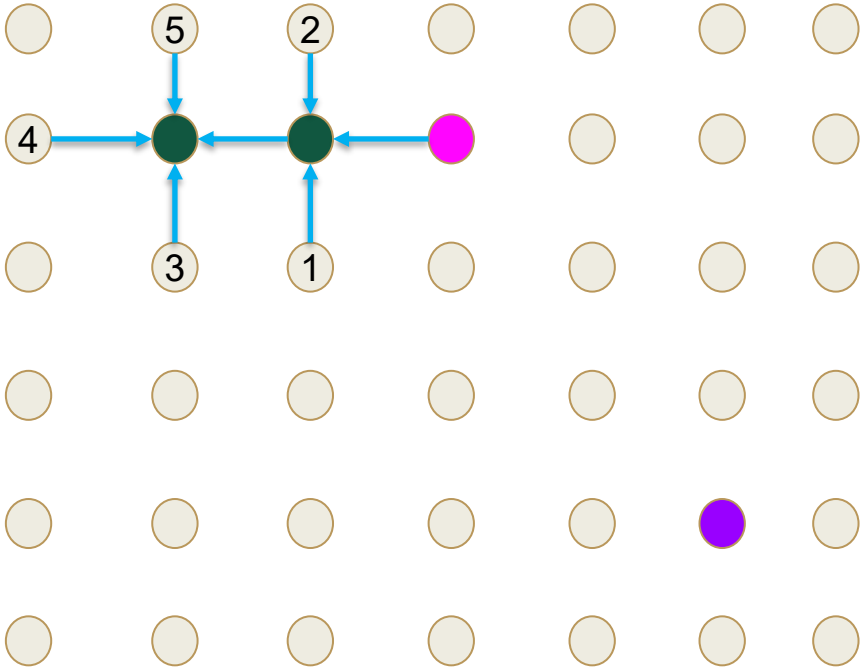

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



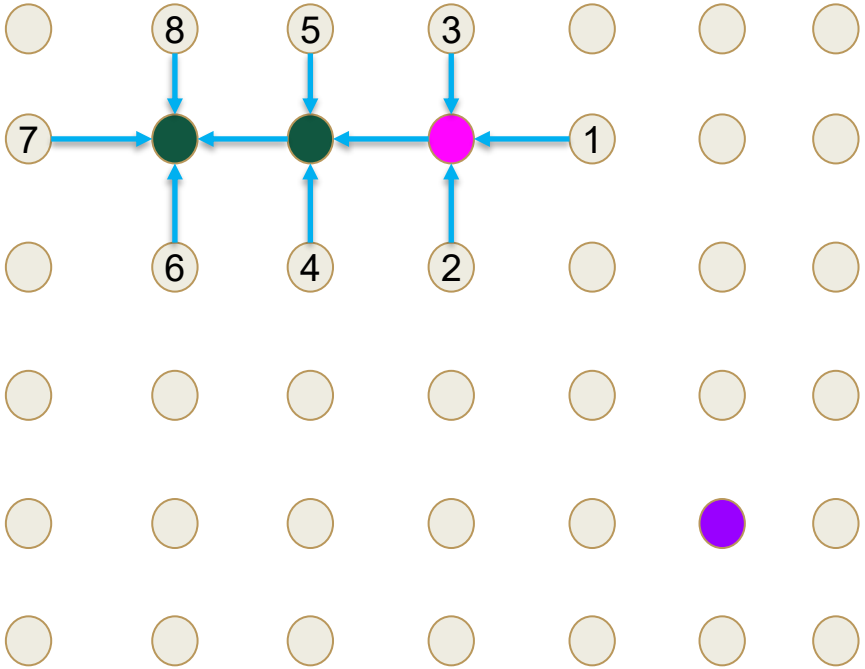

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



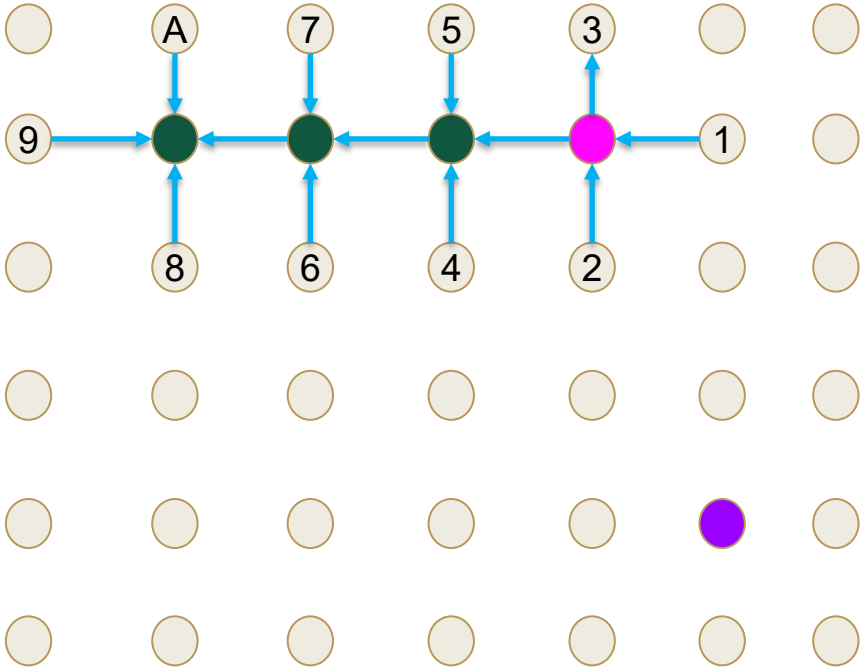

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



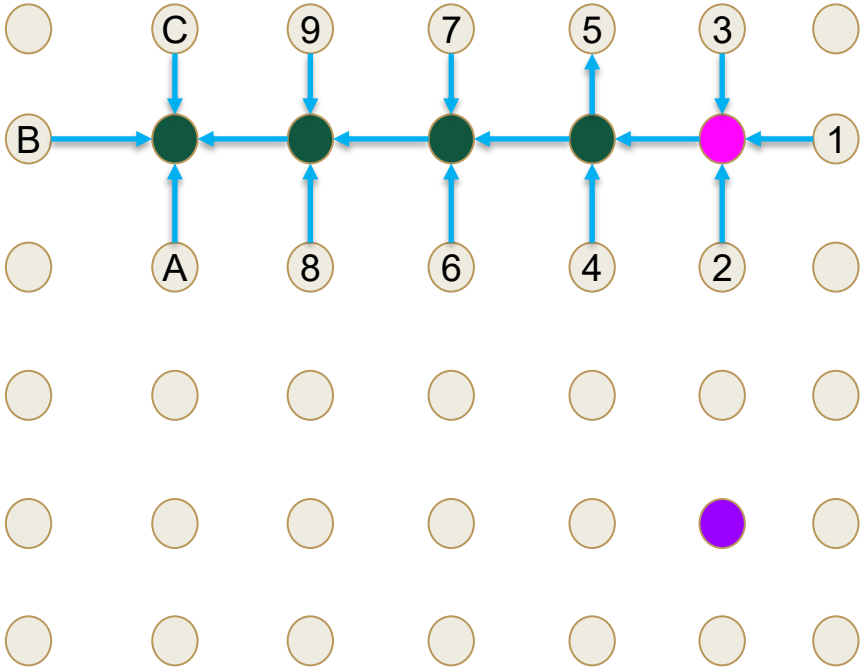

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



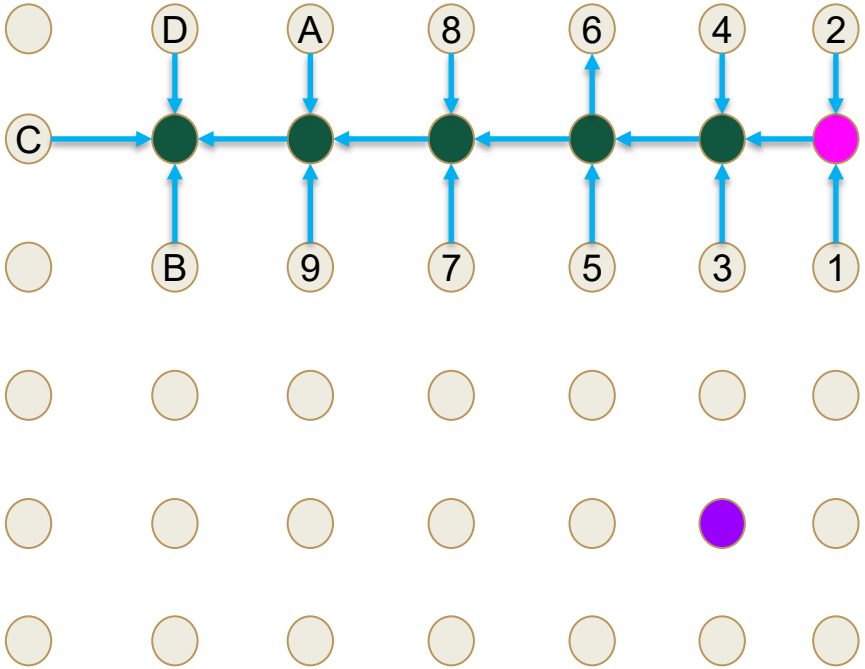

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



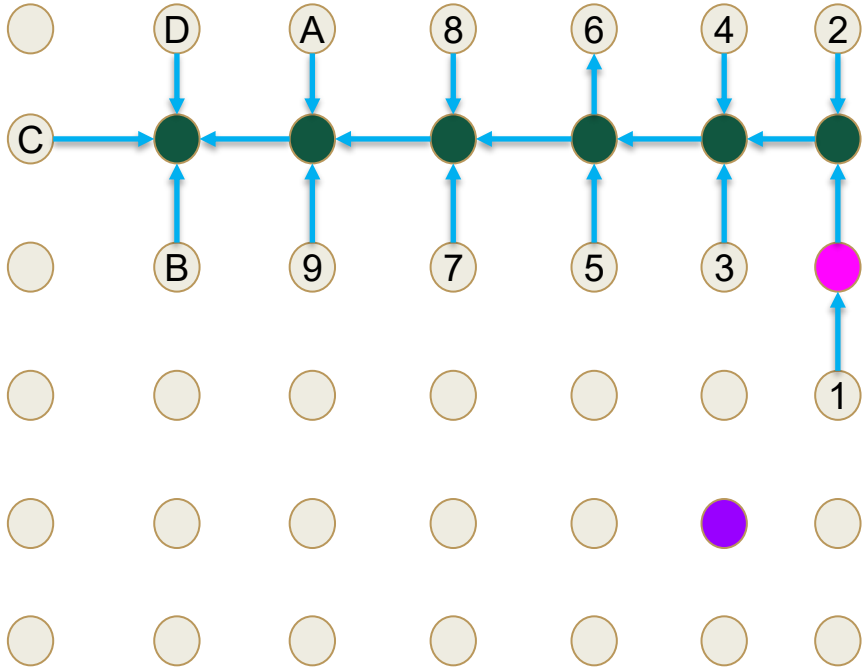

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



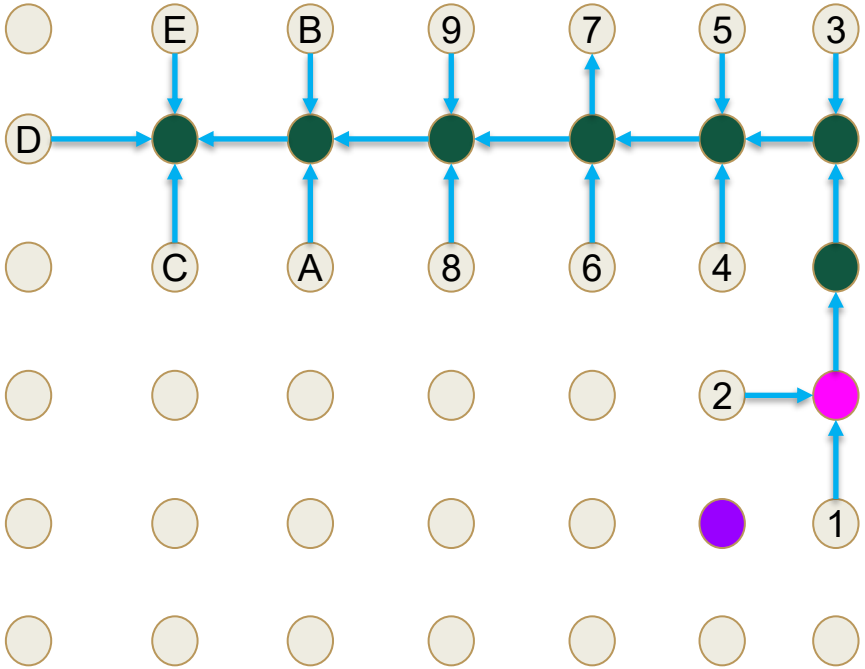

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



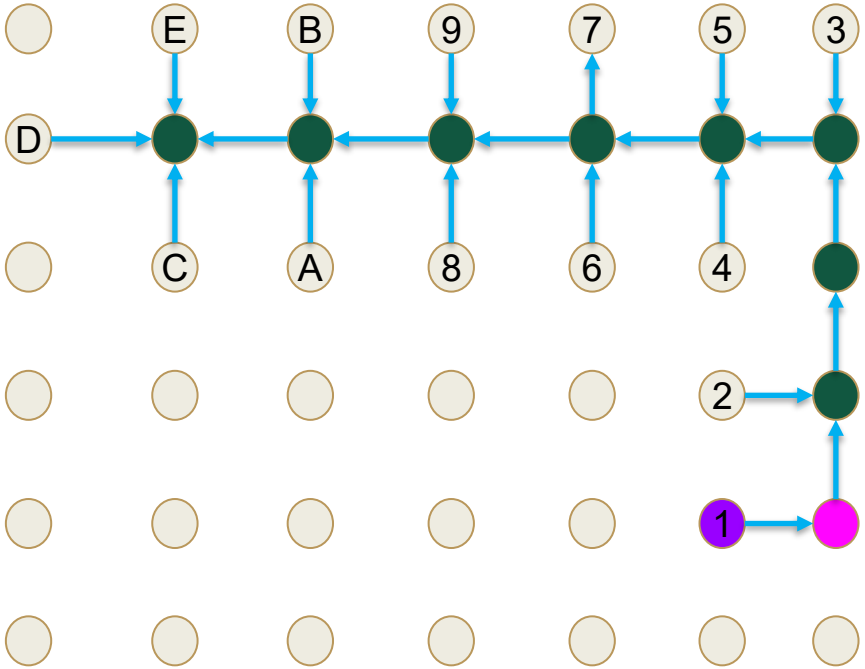

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS



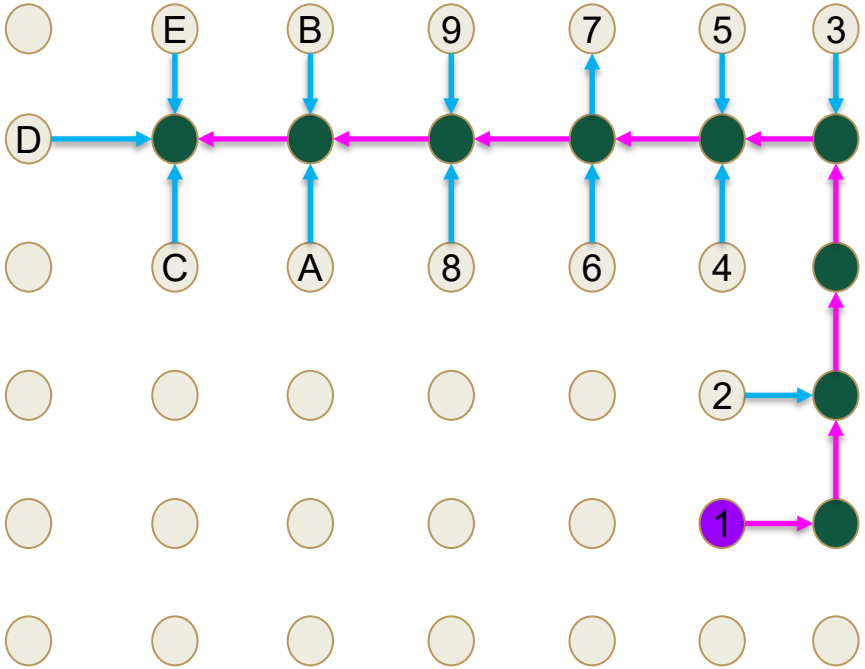

```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS




```
frontier = Stack()
frontier.push(start)
came_from = {}
came_from[start] = None
while not frontier.empty():
  cur = frontier.pop()
  for next in cur.neighbors():
    if next not in came_from:
      came_from[next] = cur
      frontier.push(next)
      if next == goal:
        # build path & return
```

Finding the Goal: DFS




```
frontier = Stack()
frontier.push(start)
came_from = {}
                                    ():
    if next not in came_from:
        came_from[next] = cur
        frontier.push(next)
        if next == goal:
            # build path & return
```

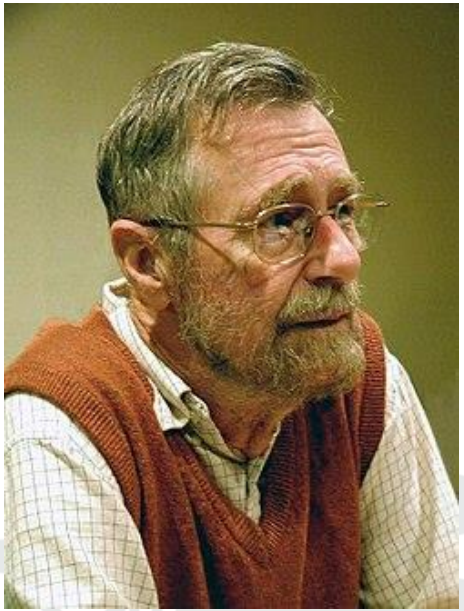
DFS explored 9 nodes and found path of length 9.
BFS explored 38 nodes and found path of length 7.

Heuristics

- Don’t simply walk through the neighbors
- Use additional information:
  - Estimated distance to goal
  - Known cost of current edges

Heuristics: Dijkstra's Algorithm

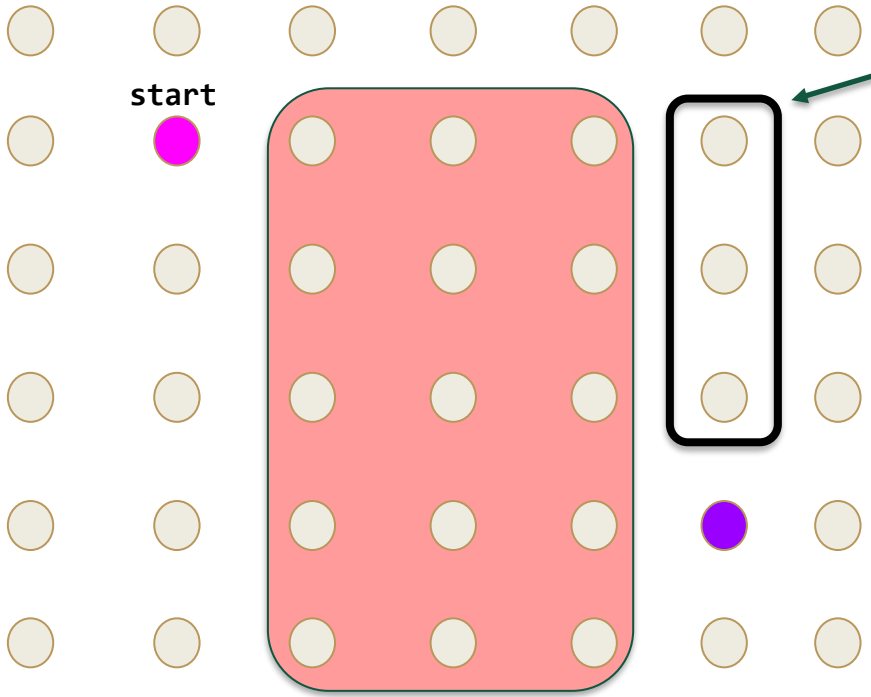
- Each edge has a cost
- Use the cost to explore
- Find the most efficient path
- *To all other nodes*



Dijkstra in 2002

My "great-grand advisor"

Heuristics: Dijkstra's Algorithm

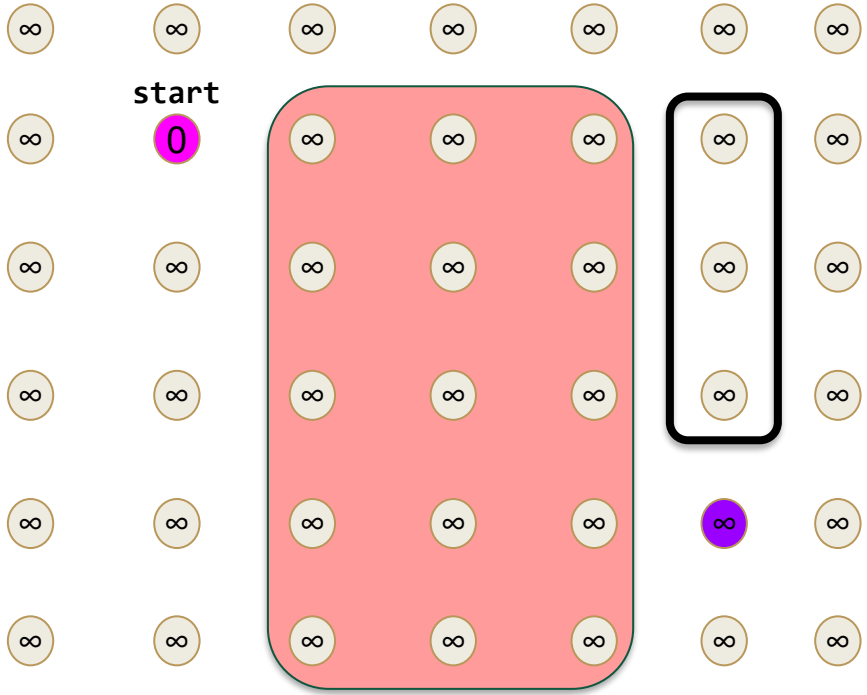


Impassable obstacle

Could add cost for other factors:
- Diagonal are more expensive
- Going next to an obstacle is risky

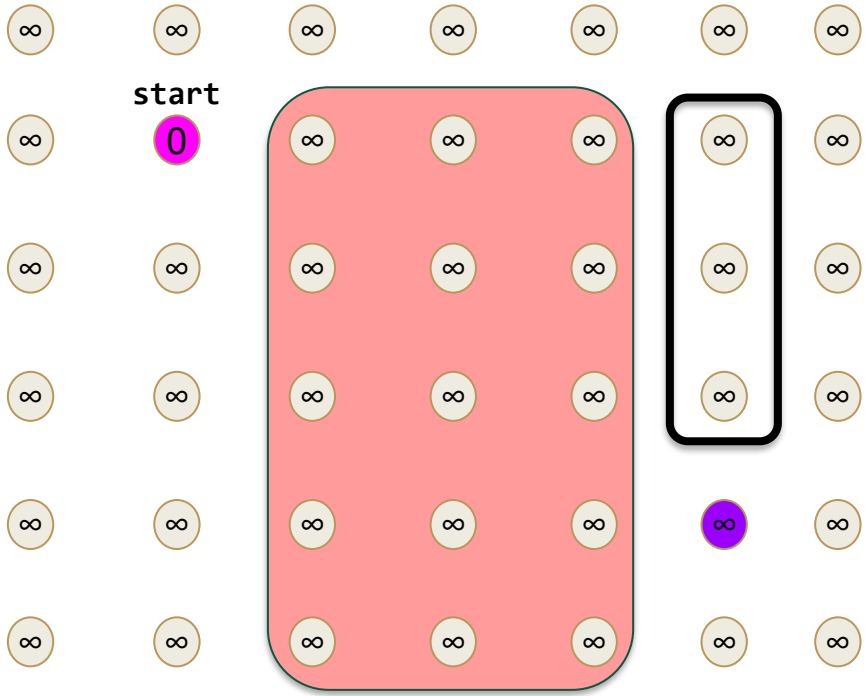
Expensive zone: 10x penalty to enter the zone (e.g., toll)

Heuristics: Dijkstra's Algorithm



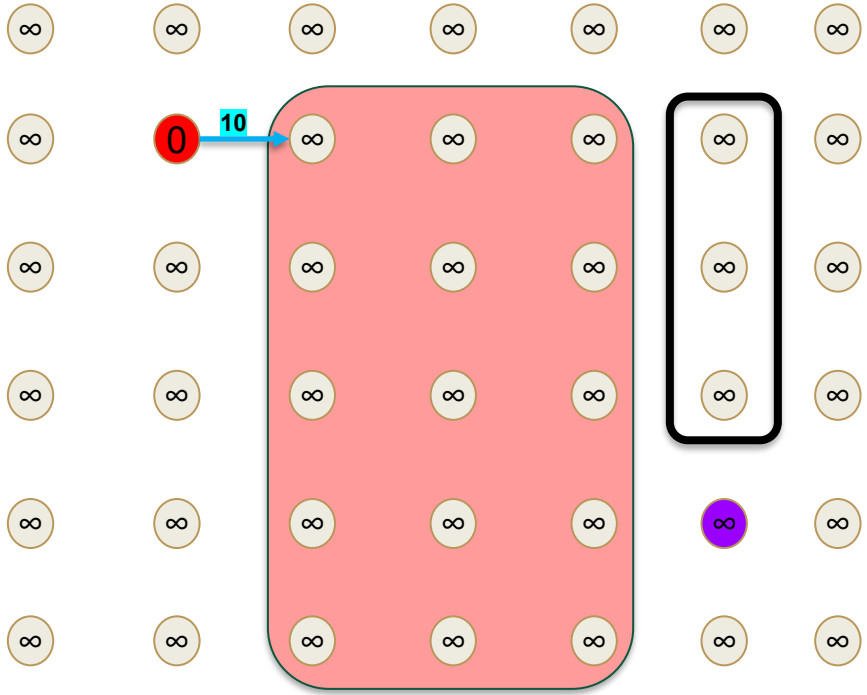

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
```

Heuristics: Dijkstra's Algorithm



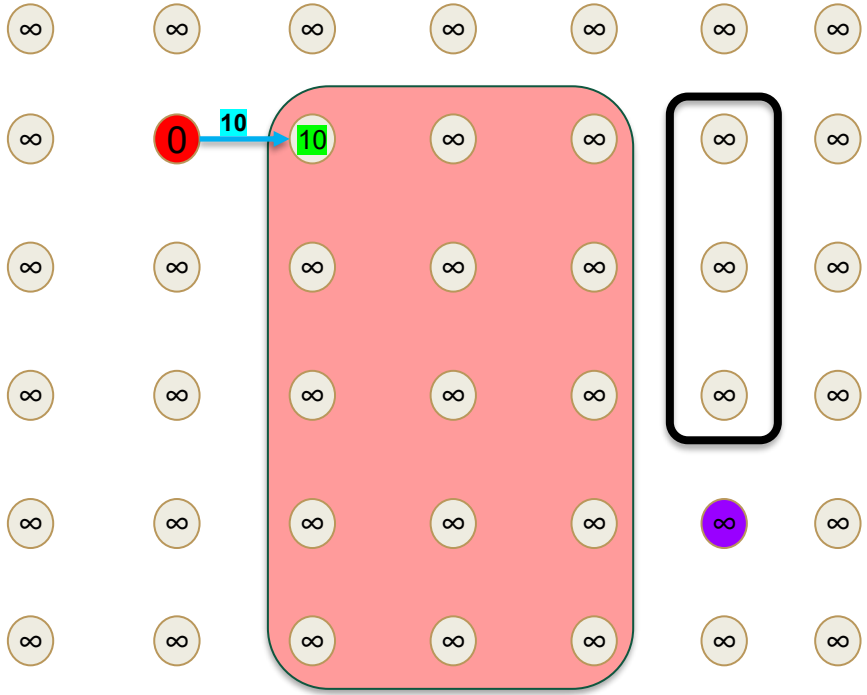

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
```

Heuristics: Dijkstra's Algorithm



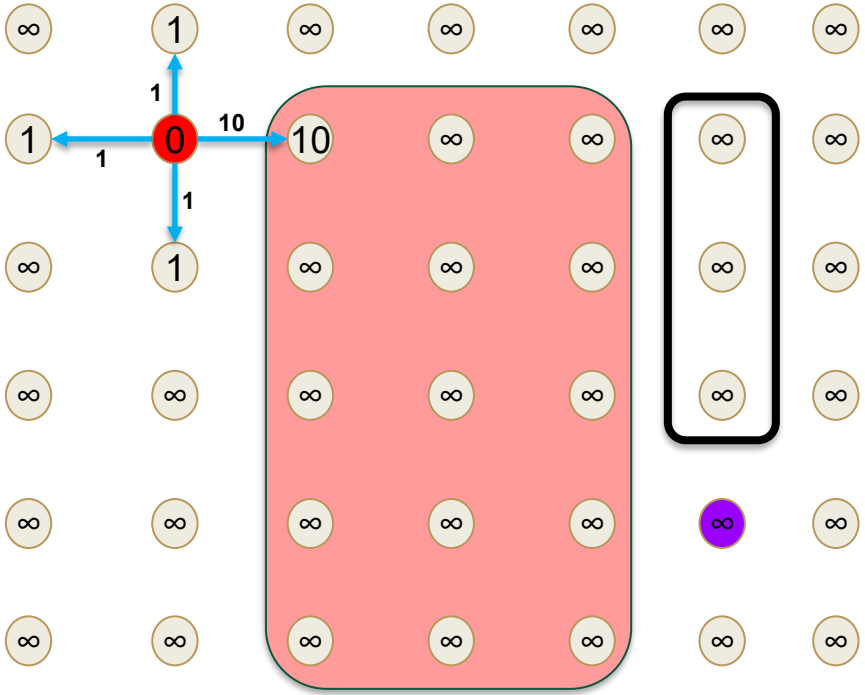

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



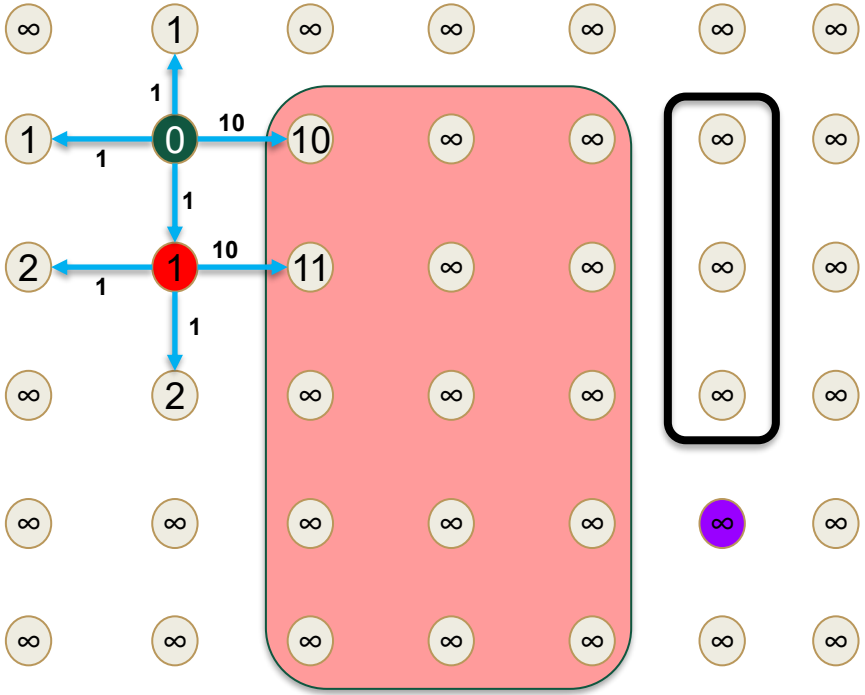

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



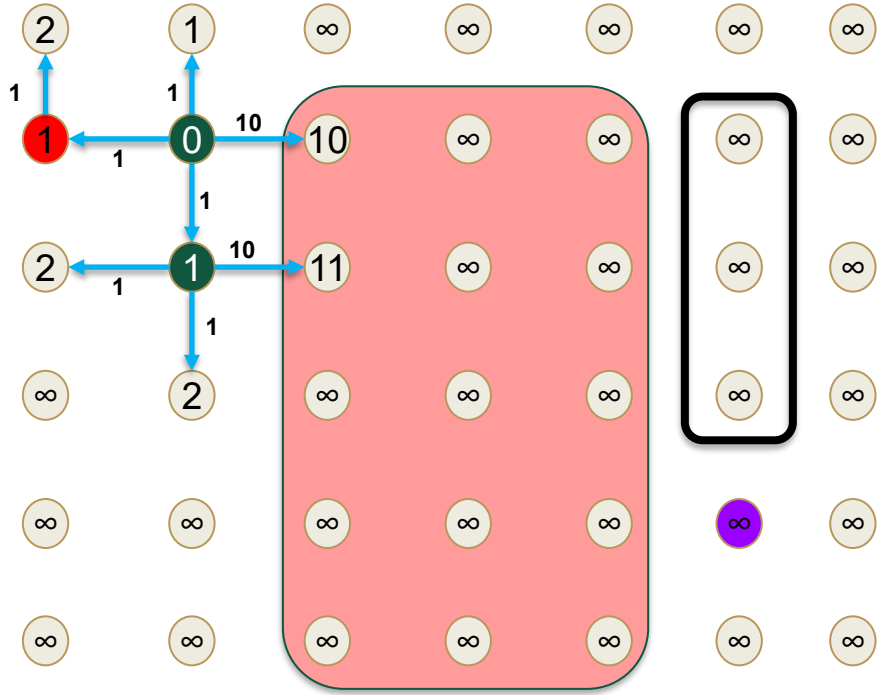

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



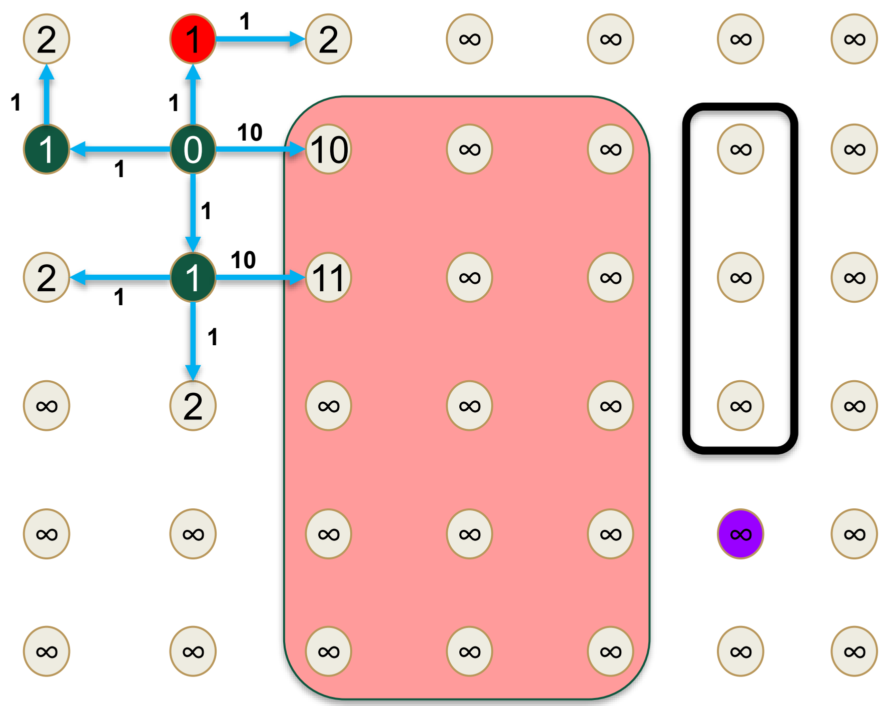

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



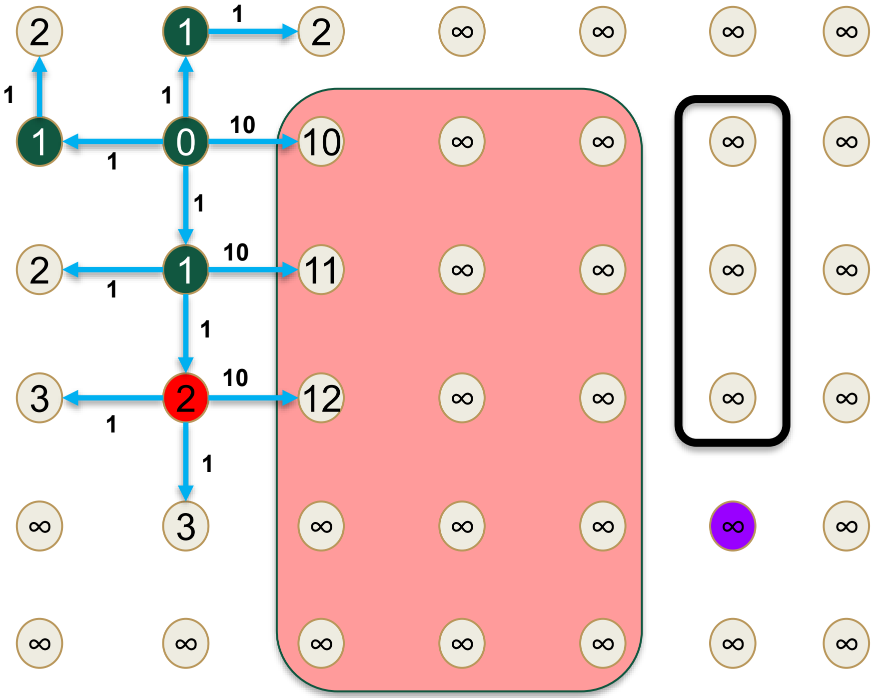

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



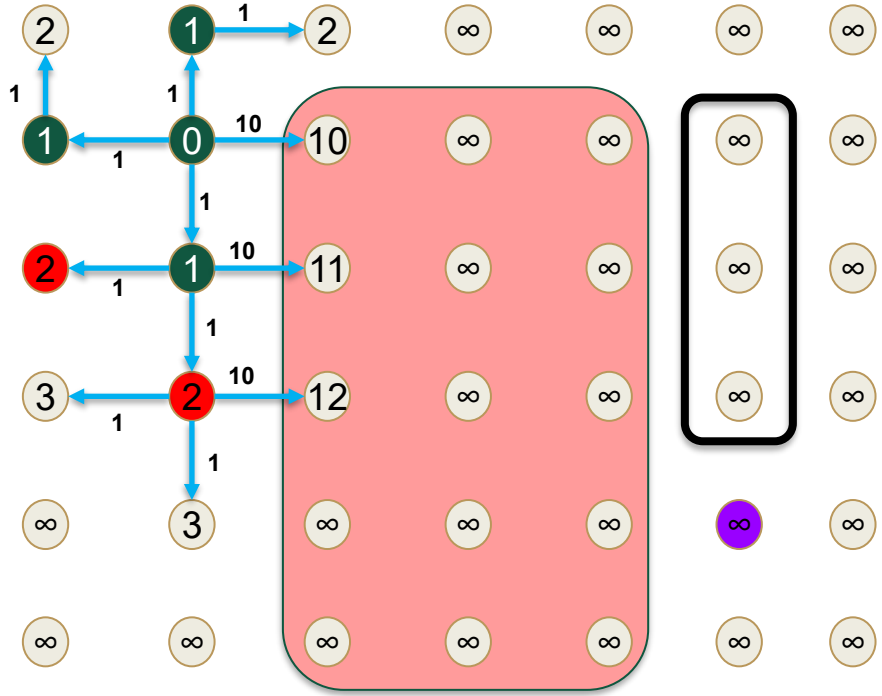
```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



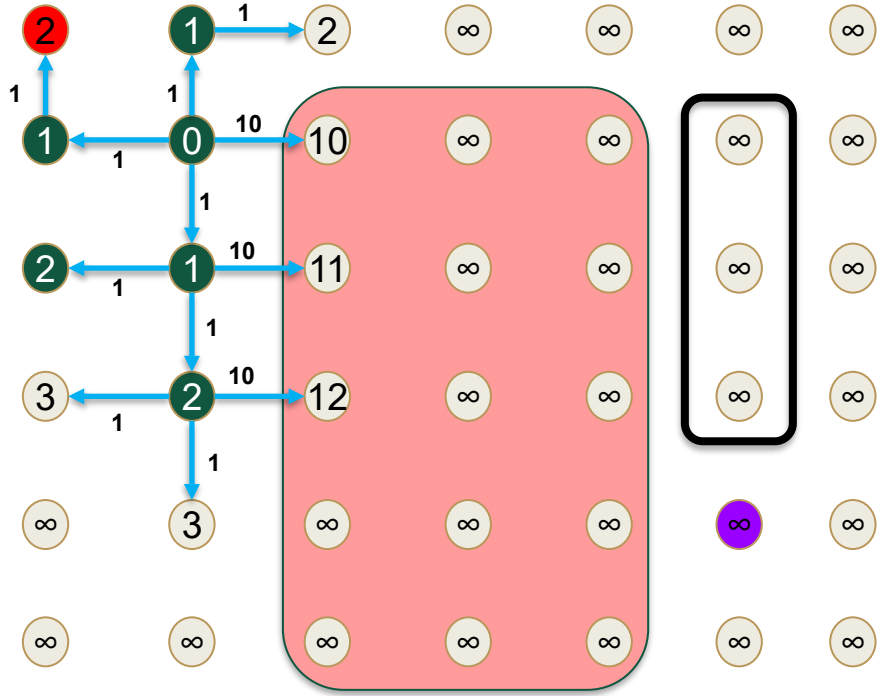

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



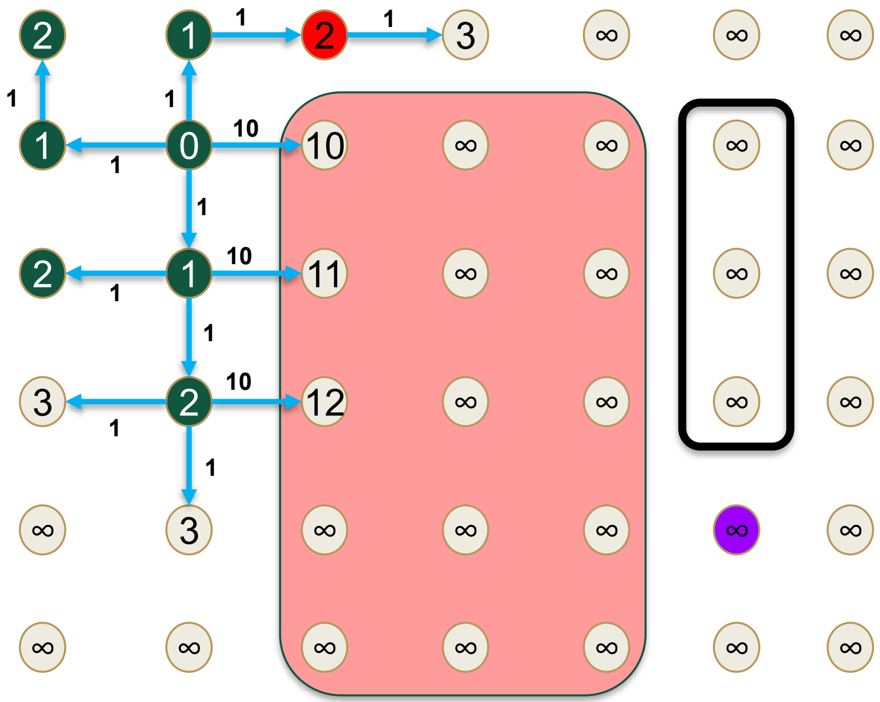

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



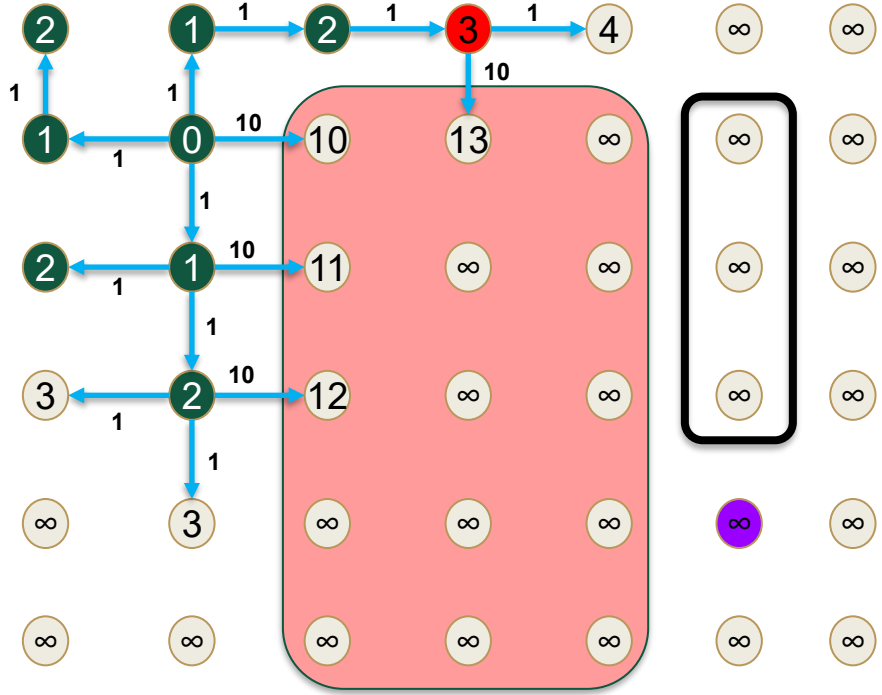

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



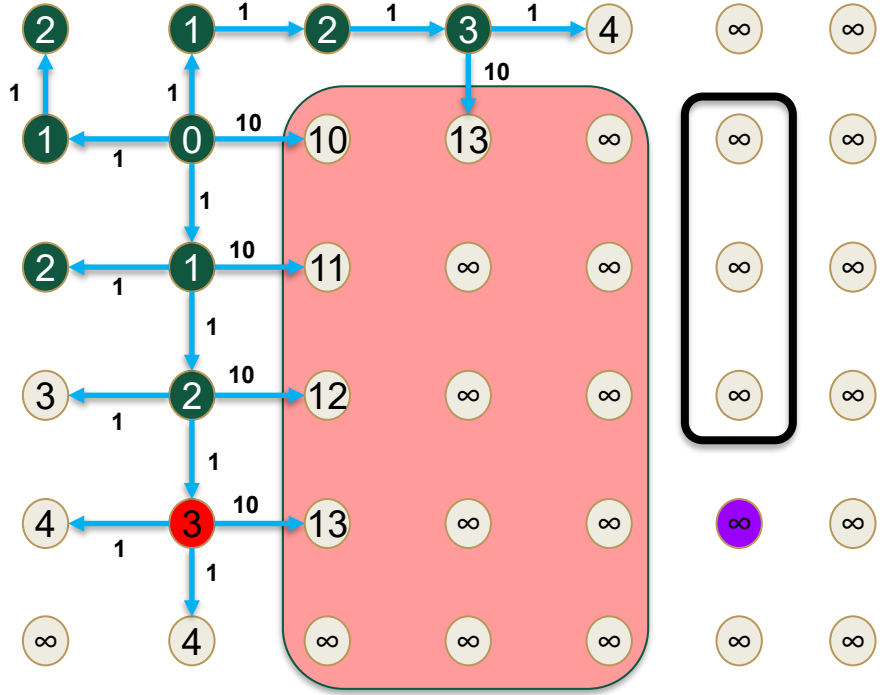
```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



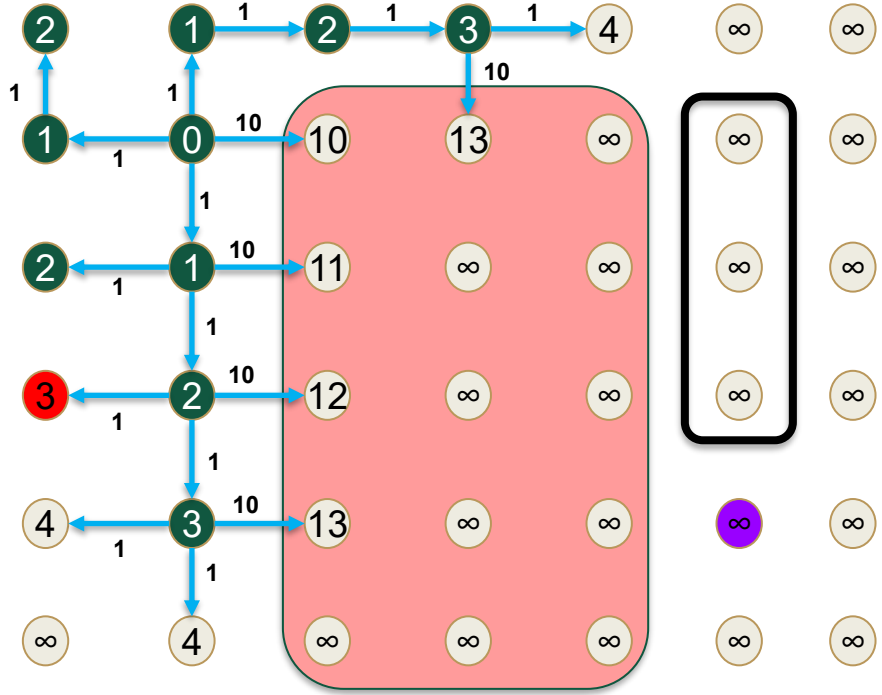

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



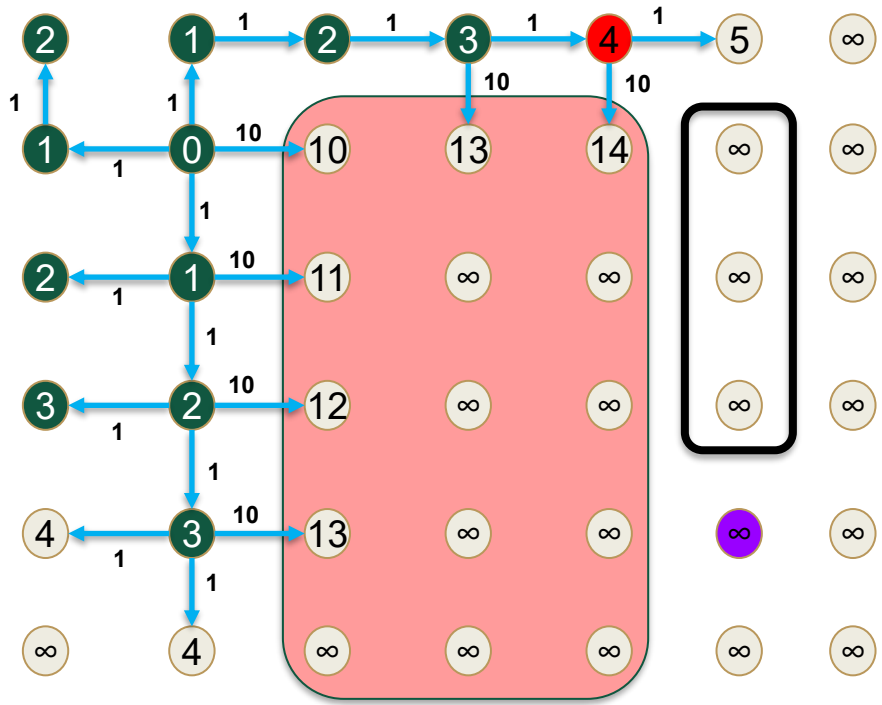

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



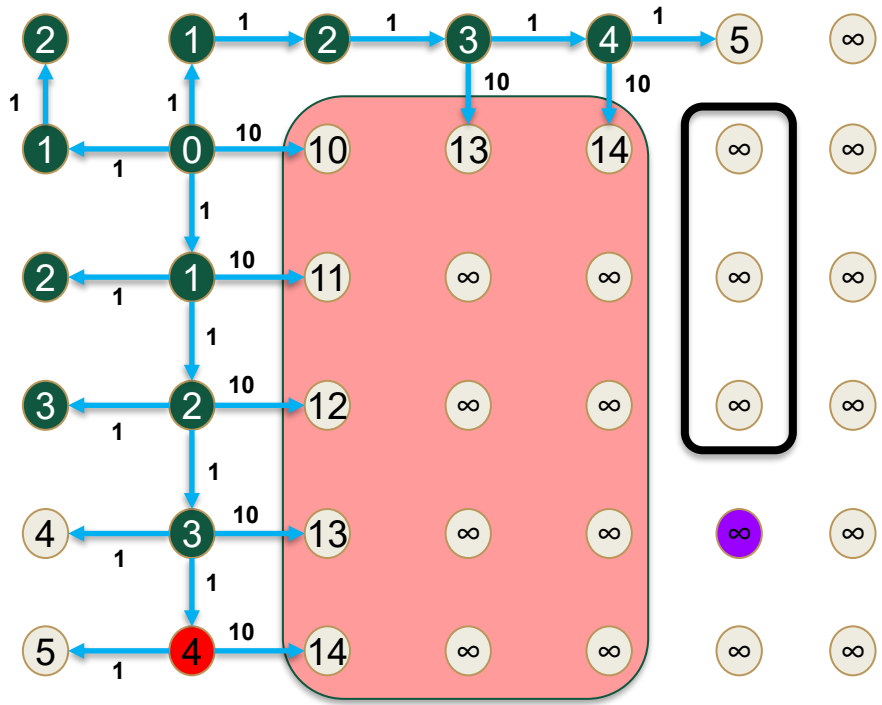

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



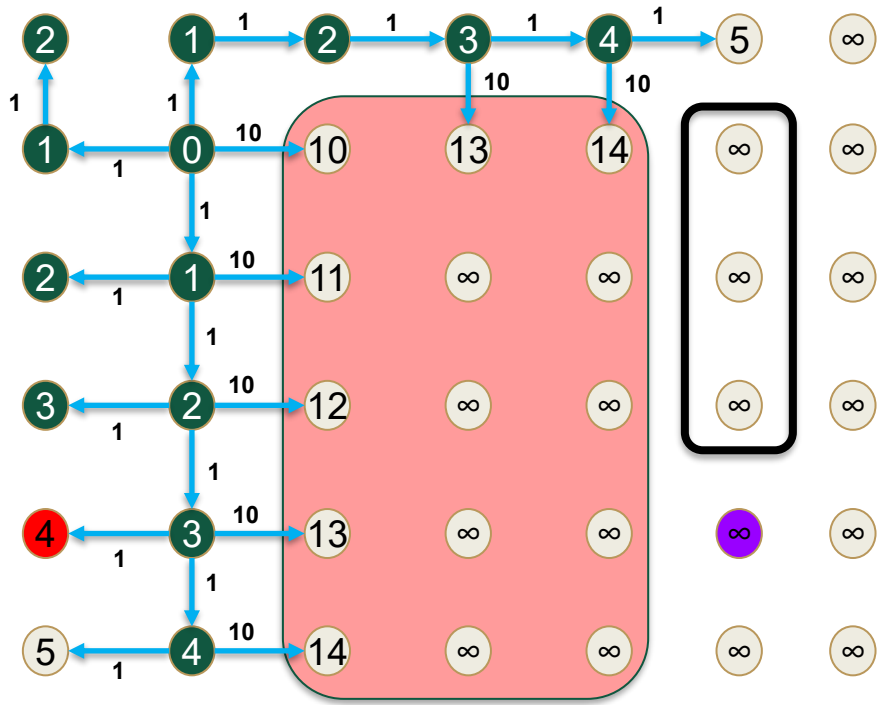

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



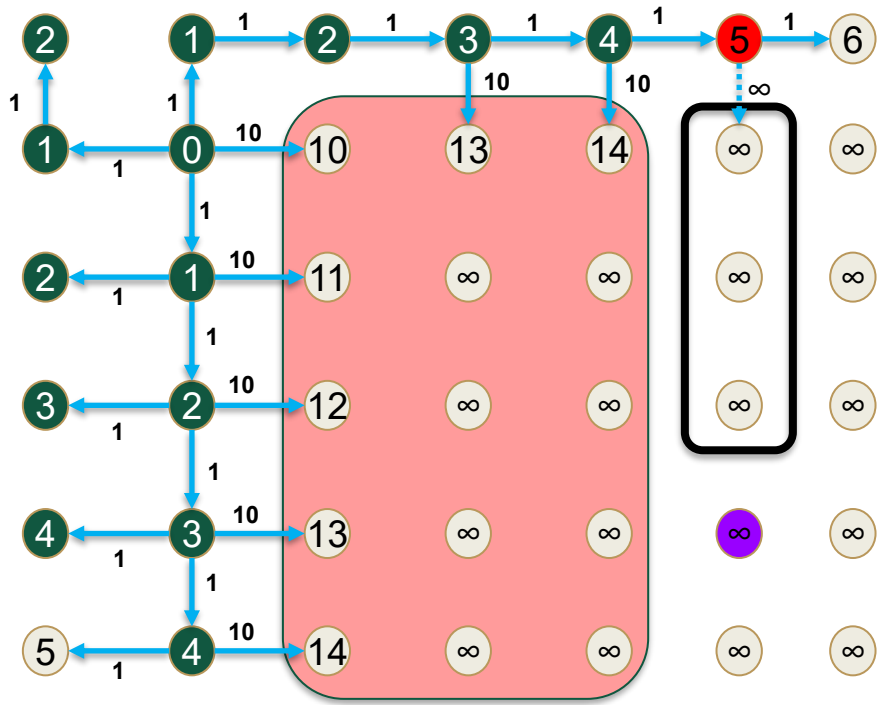

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



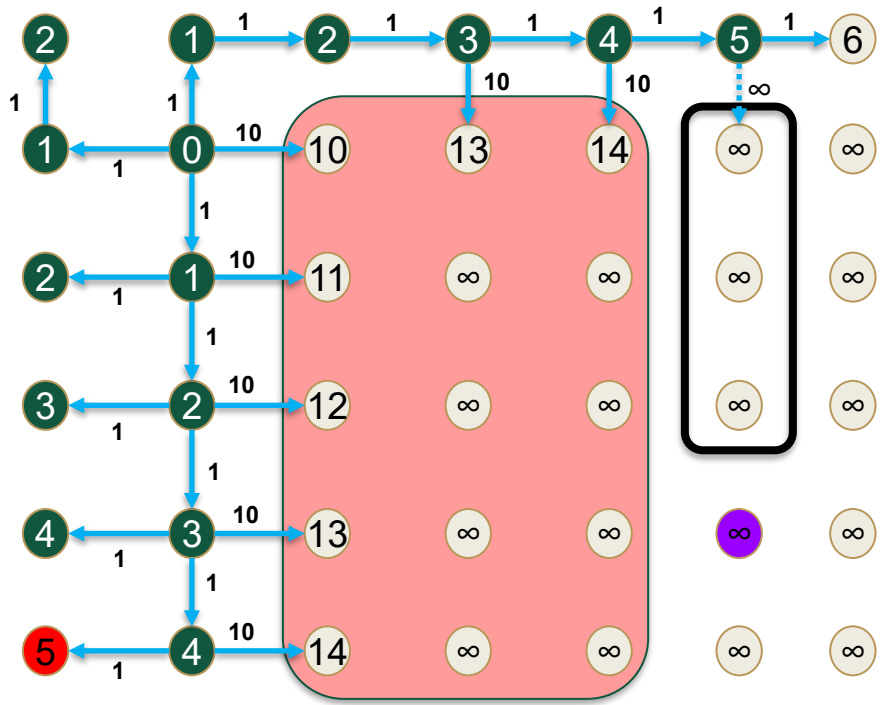

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



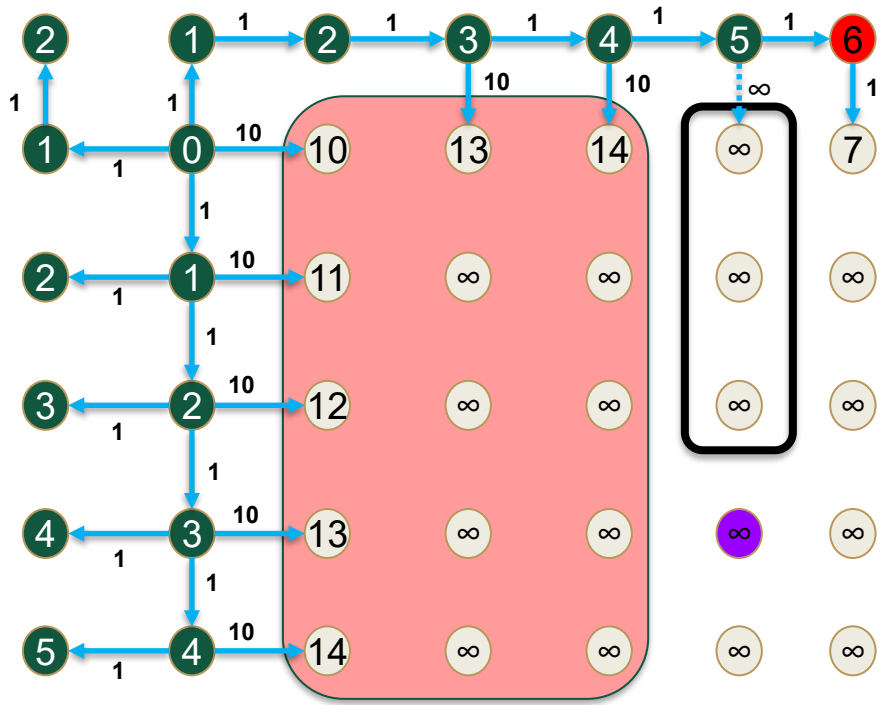

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



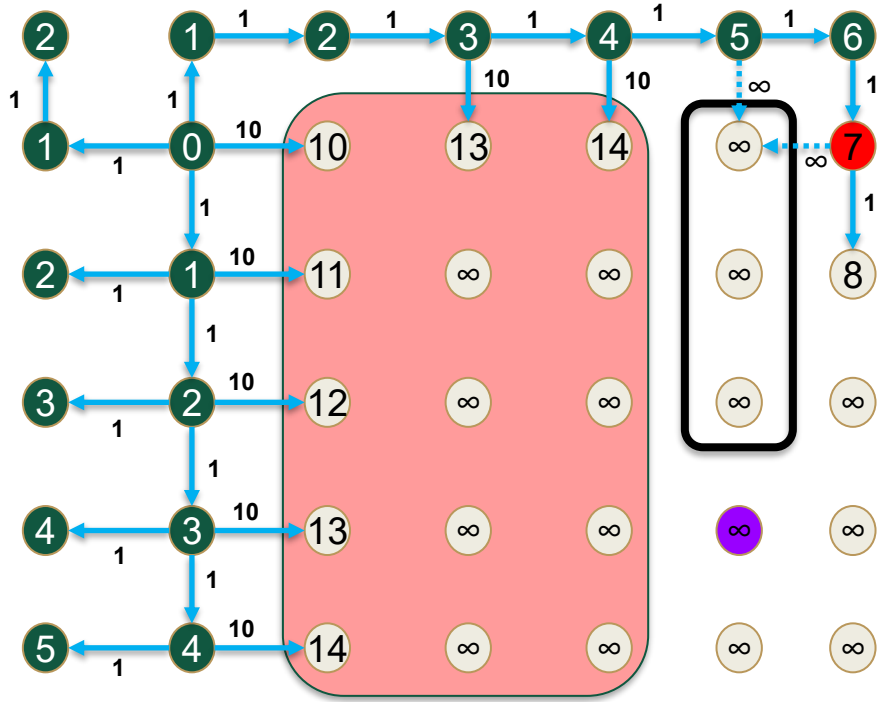

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



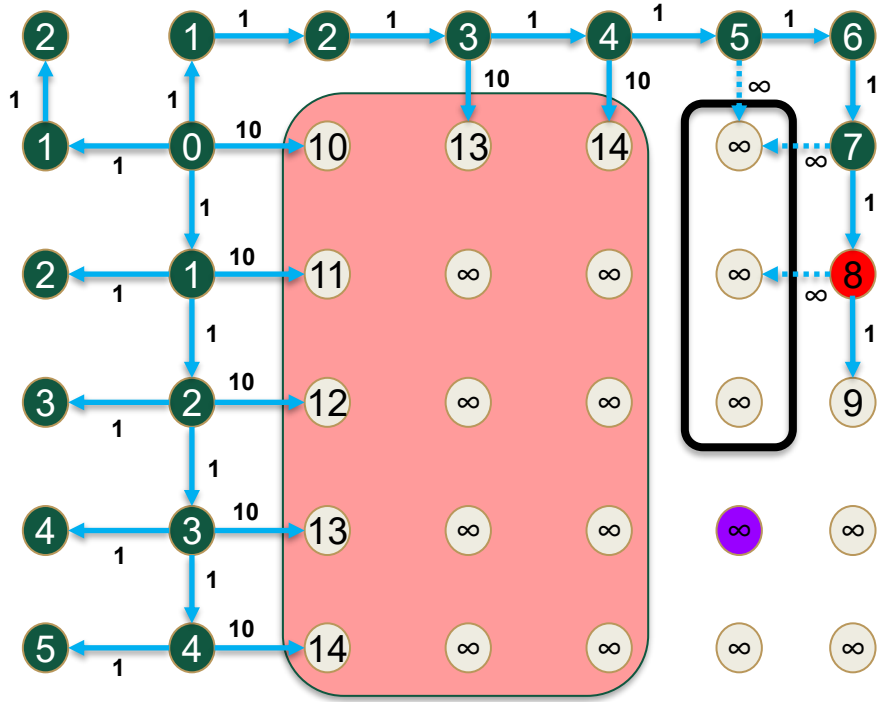

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



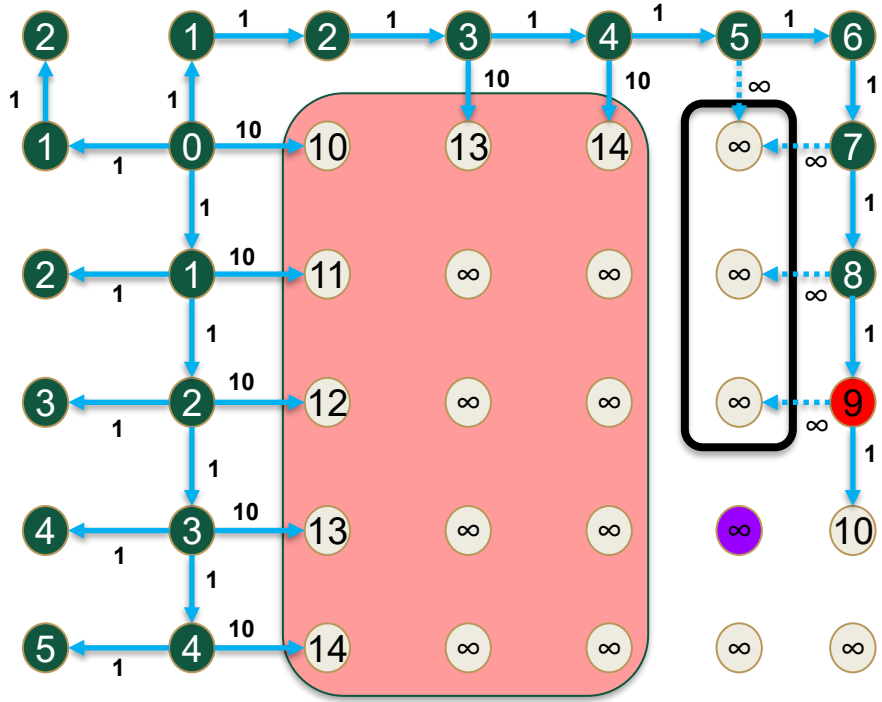

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



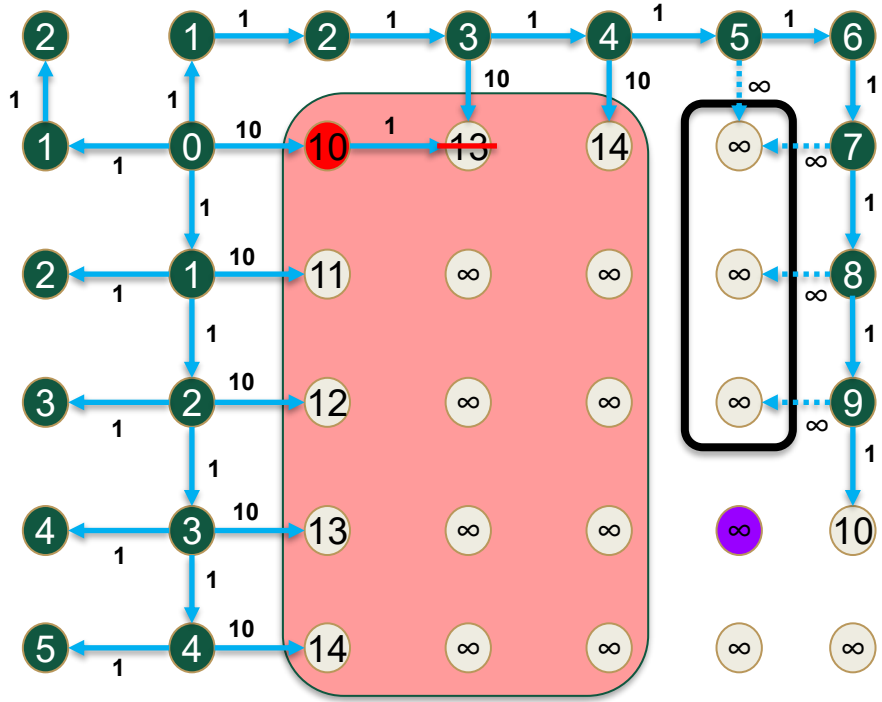

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



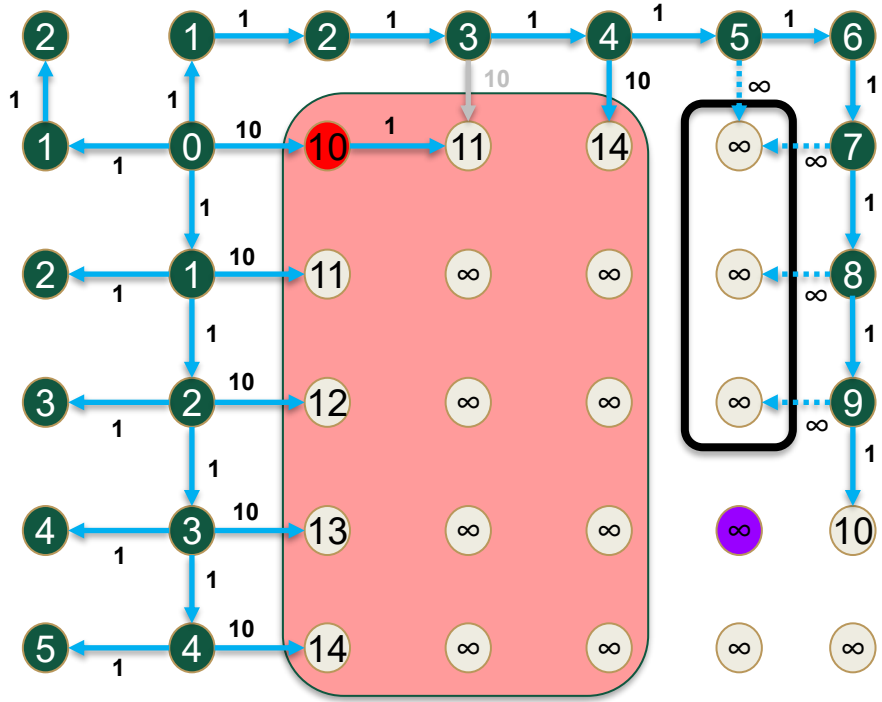

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



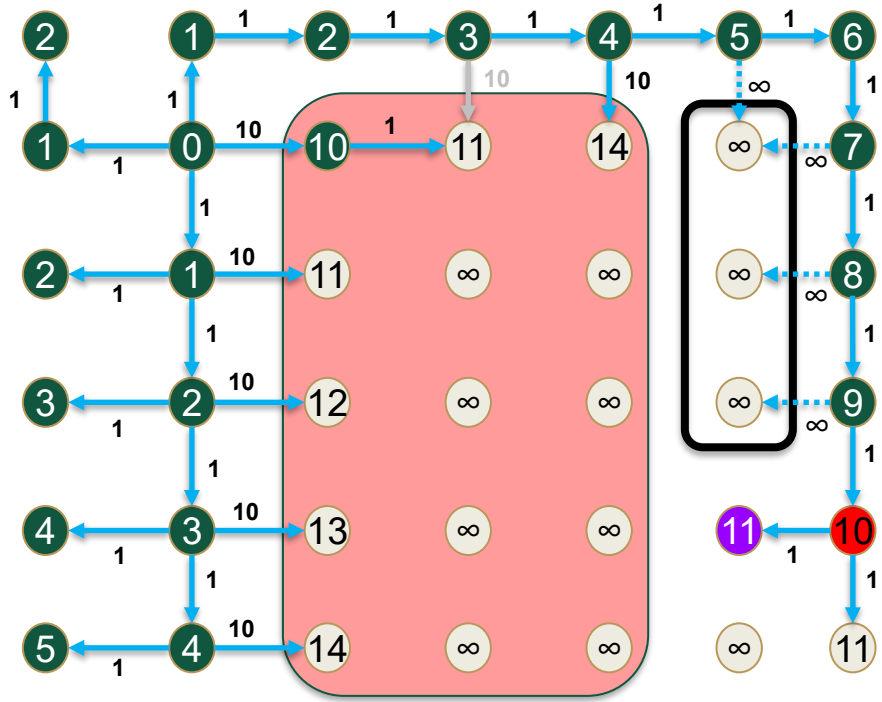

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



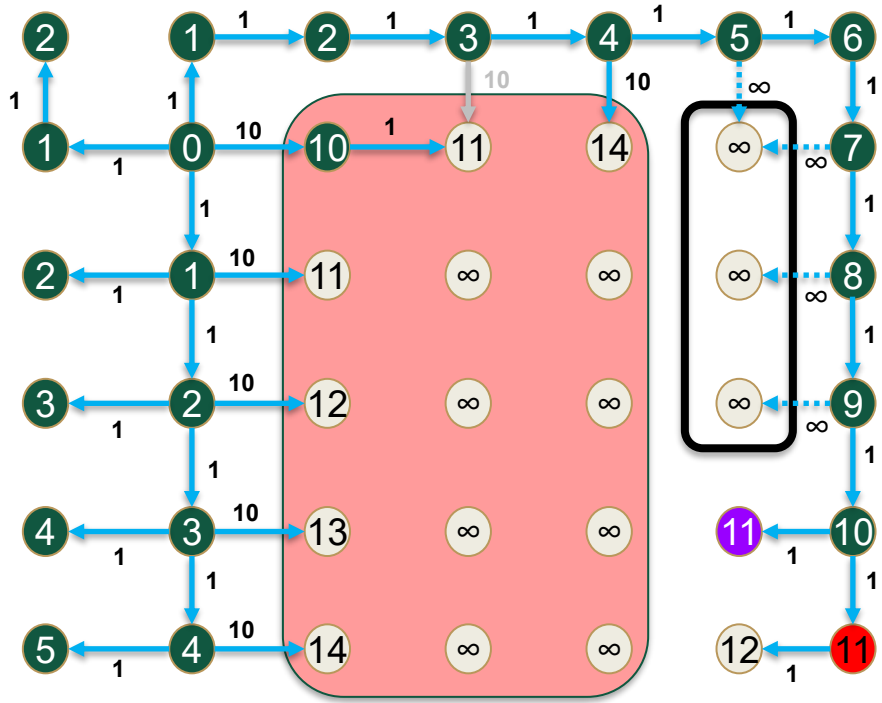

```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm



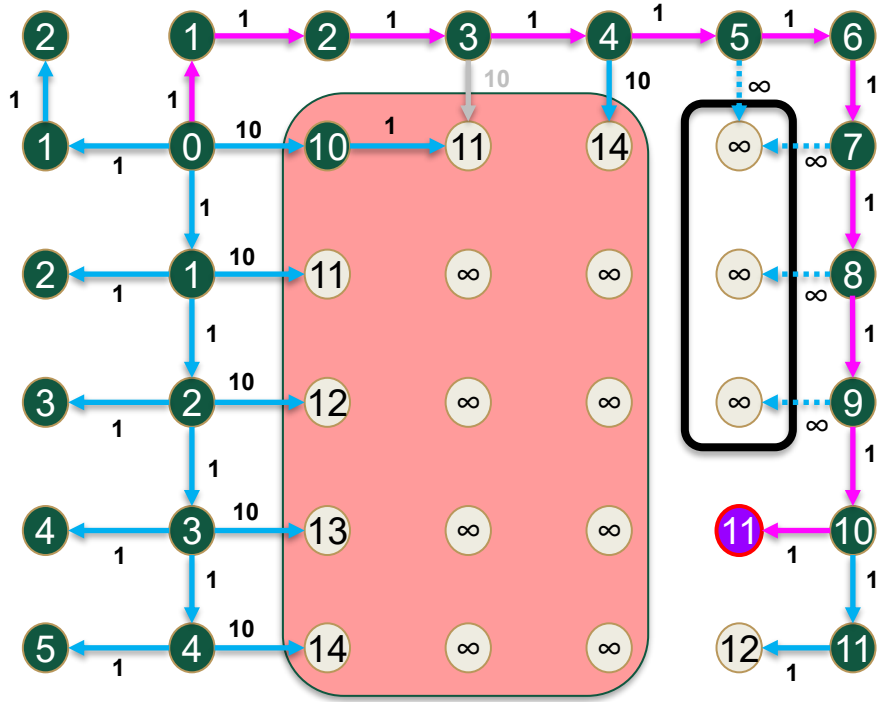
```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Heuristics: Dijkstra's Algorithm




```
cost = defaultdict(lambda x: inf)
cost[start] = 0
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

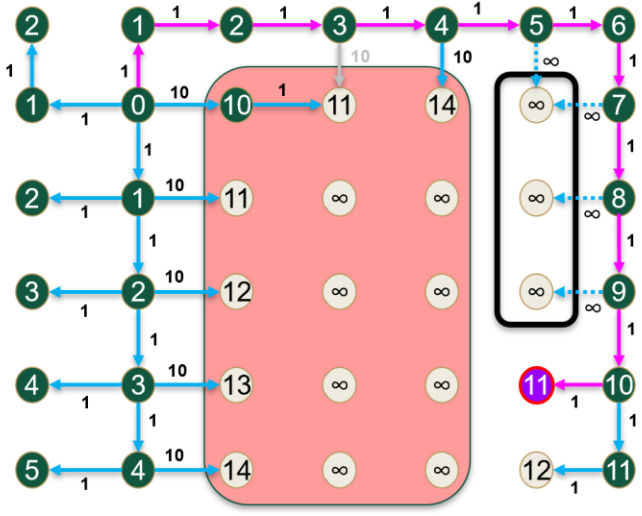
Heuristics: Dijkstra's Algorithm

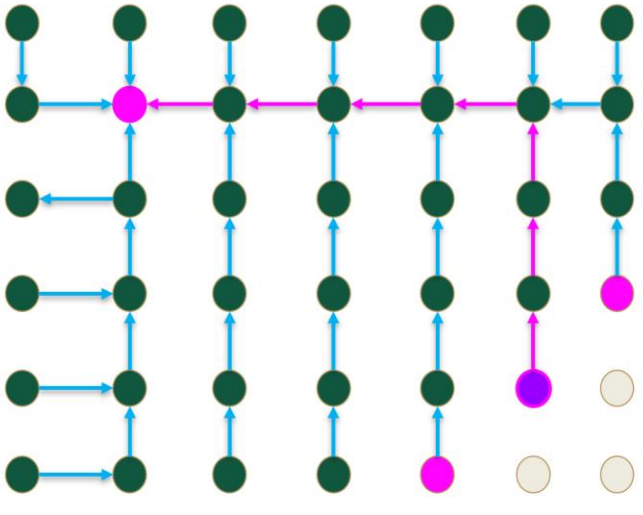


```
# …
while not frontier.empty():
  cur = frontier.pop()  # min dist
  if cur == goal:
    break  # build path and return
  for next, dist in cur.neighbors():
    new_dist = cost[cur] + dist
    if new_dist < cost[next]
      cost[next] = new_dist
      prev[next] = cur
      frontier.put(next, new_dist)
```

Dijkstra vs BFS

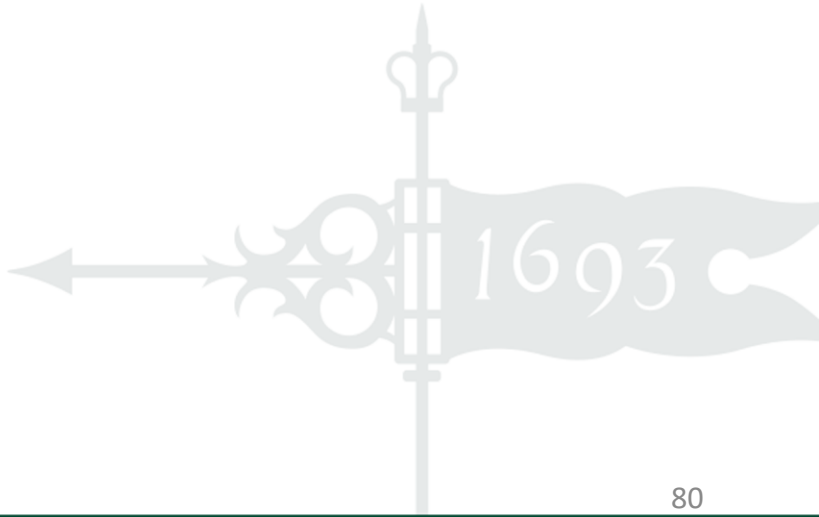
- Will find shortest path, but expensive
- Dijkstra handles path costs



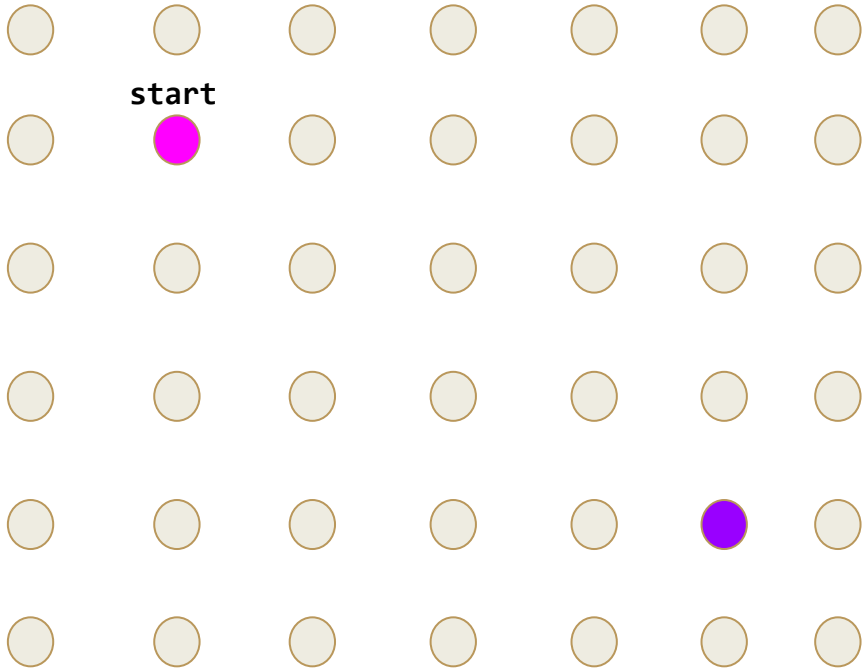


Greedy Heuristic

- Estimate distance to goal (e.g. Euclidean)
- Always go toward the goal

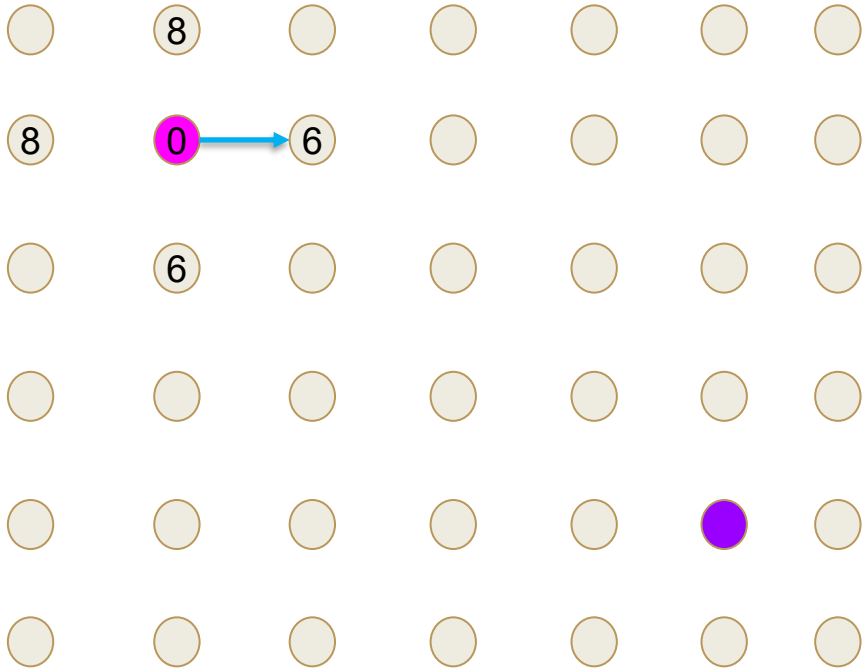


Greedy Heuristic



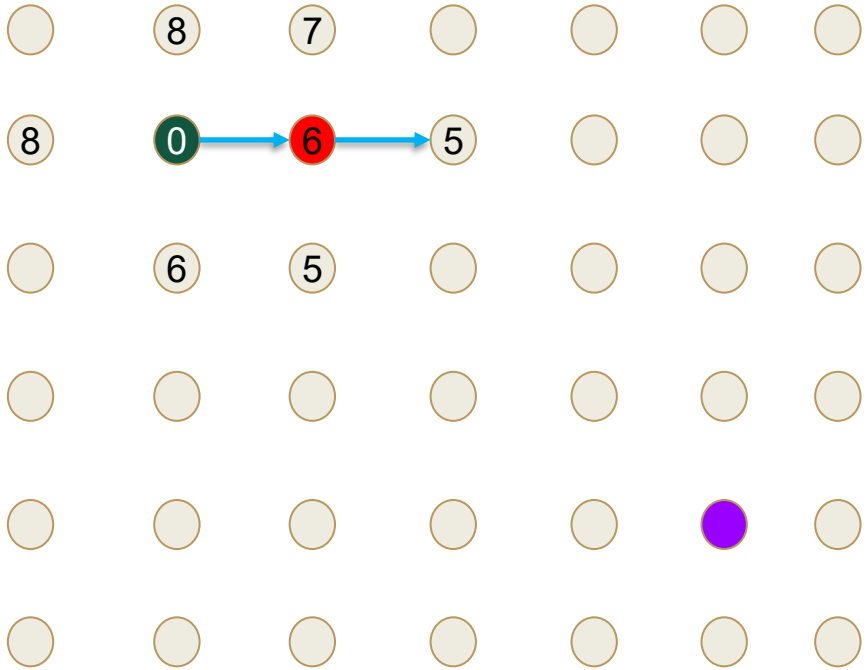

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



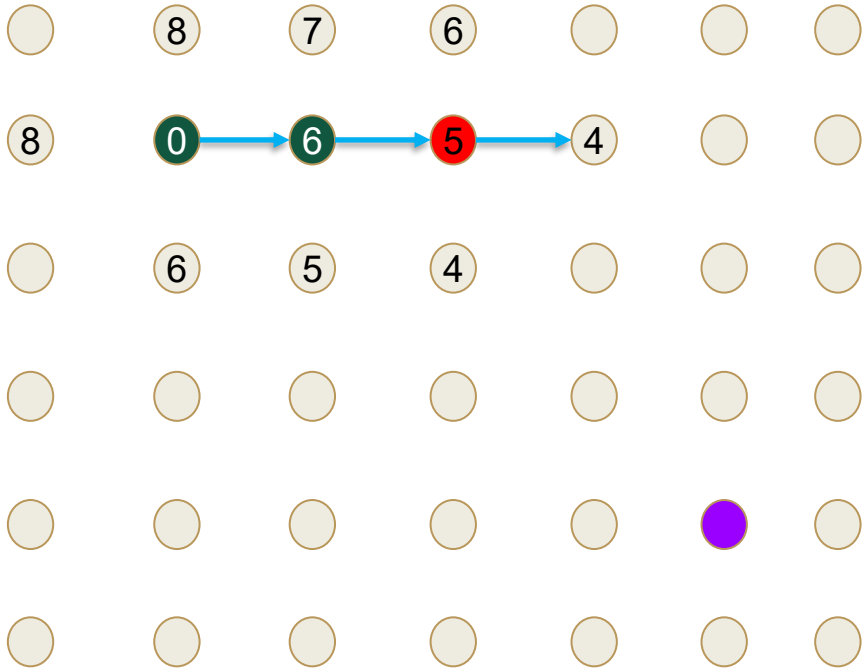

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



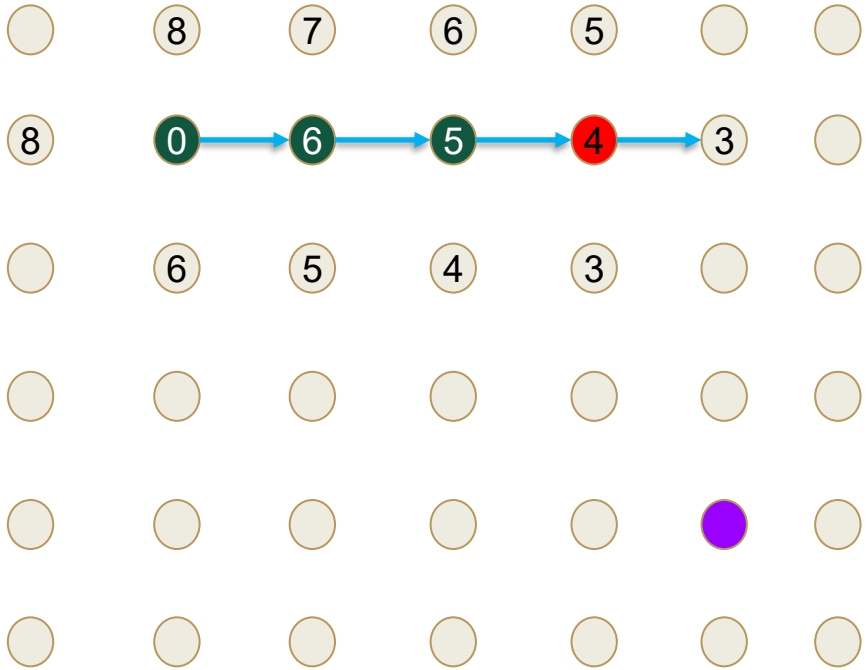

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



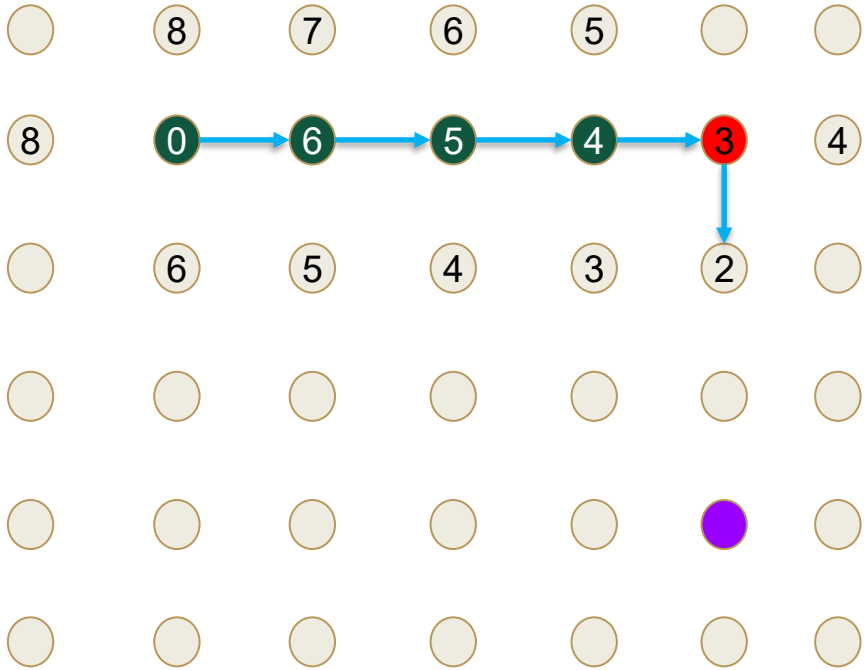

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



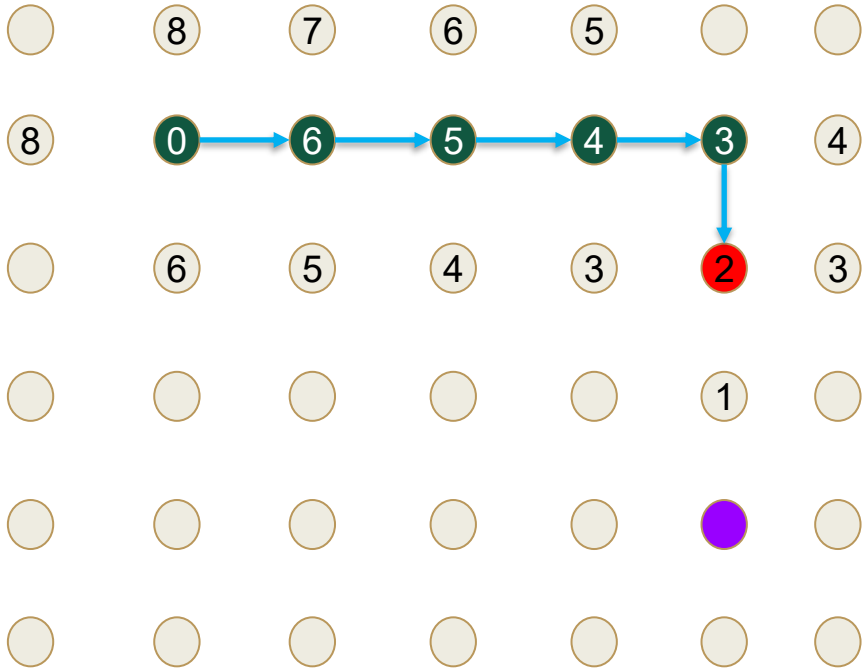

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



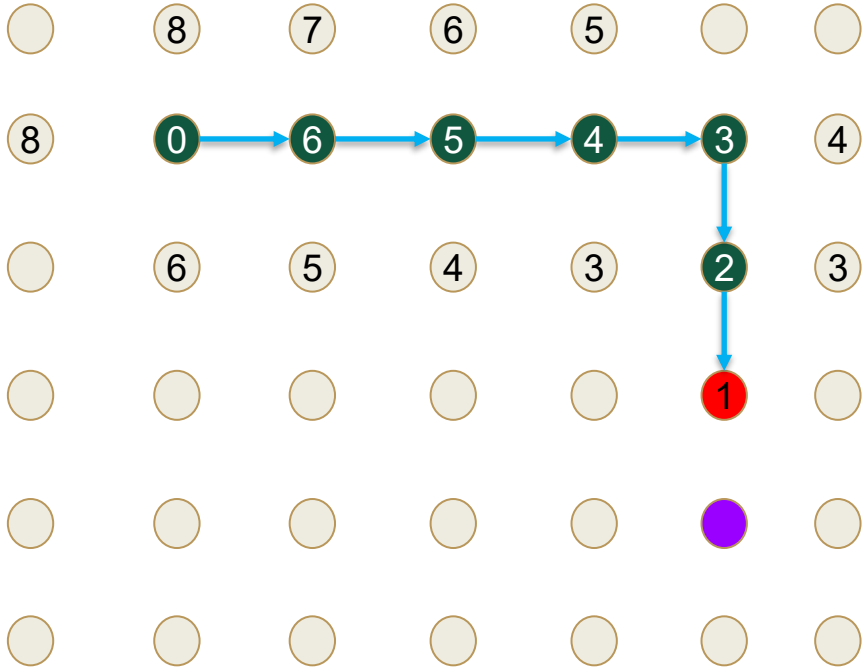

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



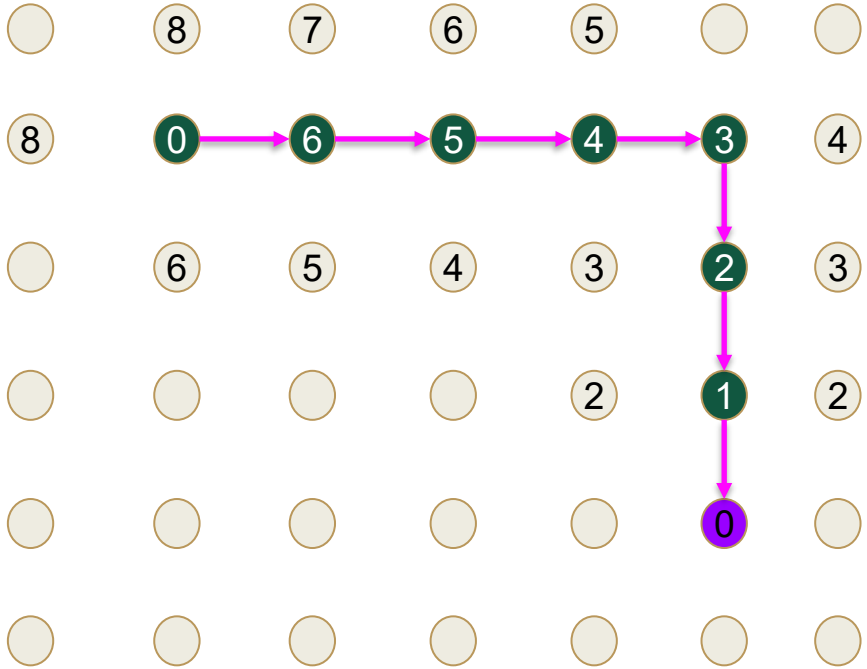

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic



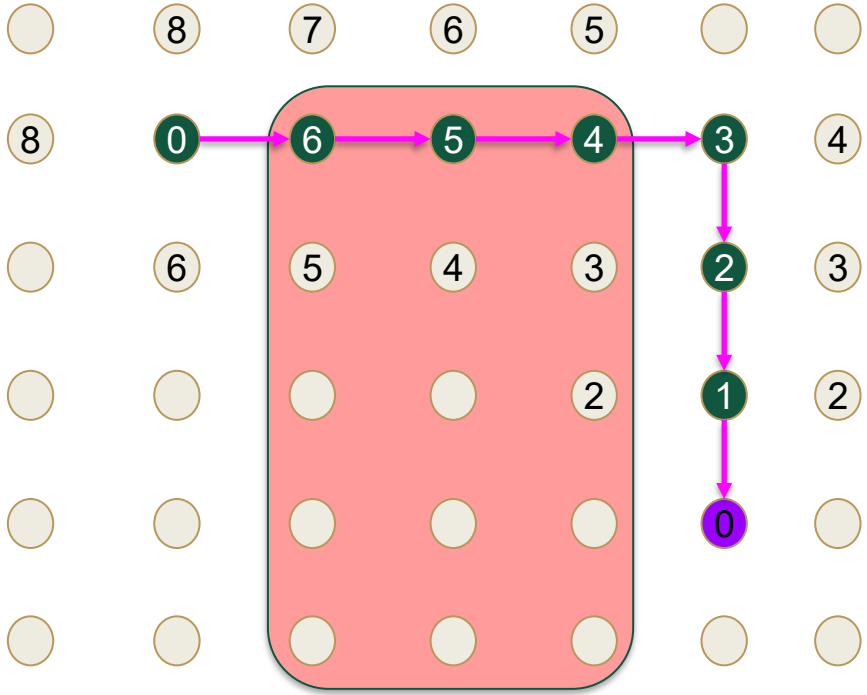

```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Greedy Heuristic




```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

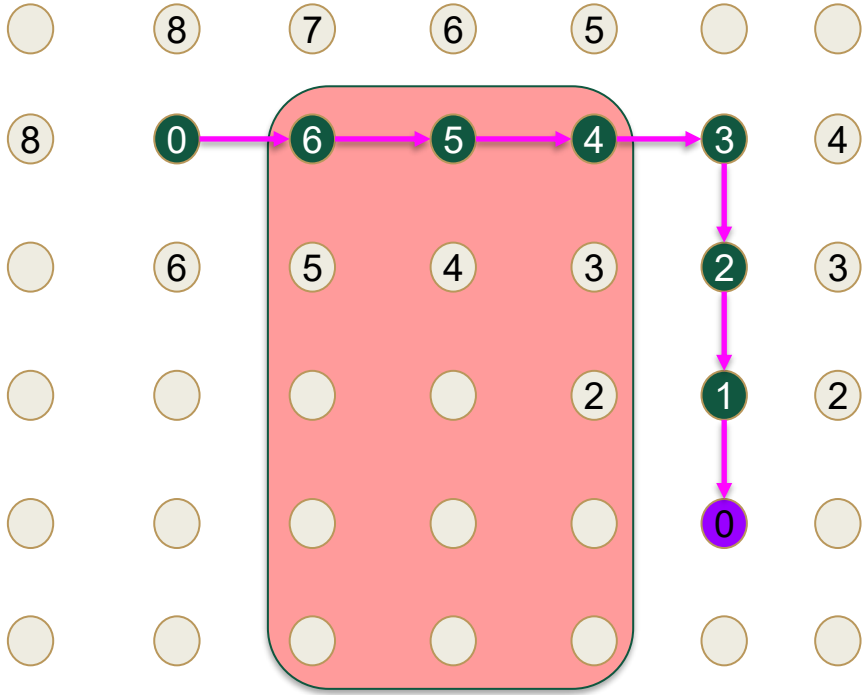
Greedy Heuristic




```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

Does anything change with the previous obstacle?

Greedy Heuristic

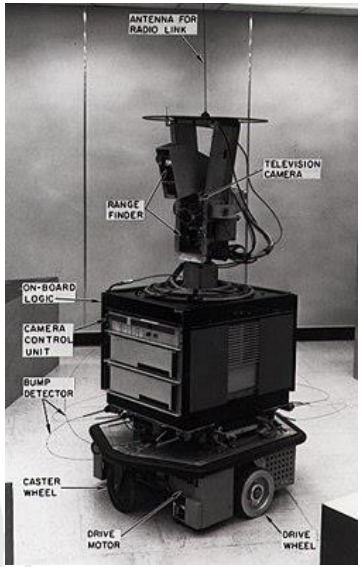



```
prev = {}
frontier = MinQueue()
frontier.put(start, 0)
while not frontier.empty():
  cur = frontier.pop()  # min dist
  for next, dist in cur.neighbors():
    if next not in prev:
      prev[next] = cur
      est = dist(goal, next)
      frontier.put(next, est)
```

No guarantees of cost!

Best of both: A*

- Use both:
  - Distance from home (Dijkstra's)
  - Distance from goal (greedy)
- Compared to Dijkstra's
  - Generally fewer nodes ("Informed Dijkstra's")
  - Still optimal path



Originally built for robots!

When to recalculate paths?

- The world changes – your path should too!
  - Every *n* steps (space or time)
  - When world change detected
  - When a landmark is reached
  - When lost
  - When possible (extra time or CPU)

What to recalculate?

- Full path
  - Expensive, guarantees still apply
- Partial path
  - Cheaper, fewer guarantees
  - Splice together with previous path

Path planning in ROS

- Packages with implementations
  - nav2_smac_planner
- Message Types:
  - MapMetaData
  - OccupancyGrid
  - Path

Occupancy Grid

nav_msgs/msg/OccupancyGrid Message

File: nav_msgs/msg/OccupancyGrid.msg

Raw Message Definition

```
# This represents a 2-D grid map

std_msgs/Header header

# MetaData for the map

MapMetaData info

# The map data, in row-major order, starting with (0,0).

# Cell (1, 0) will be listed second, representing the next cell in the x direction.

# Cell (0, 1) will be at the index equal to info.width, followed by (1, 1).

# The values inside are application dependent, but frequently,

# 0 represents unoccupied, 1 represents definitely occupied, and

# -1 represents unknown.

int8[] data
```

Contains width/height

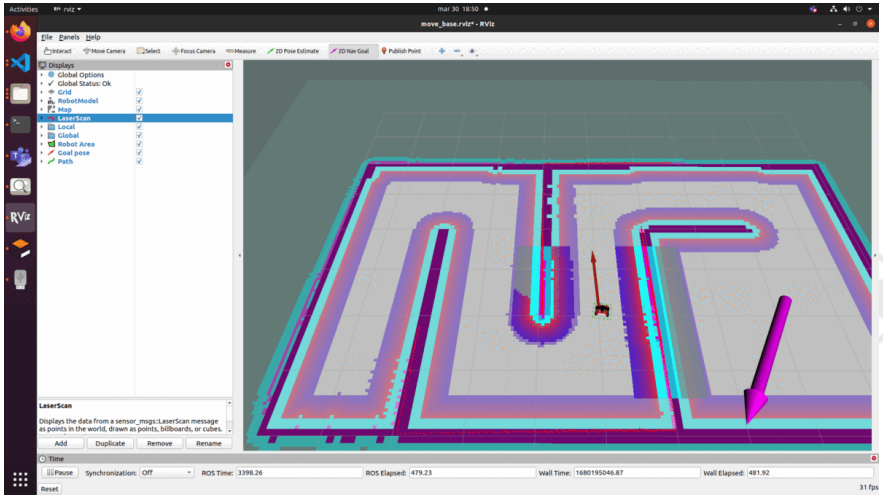
width

height

0	1	2	3	…	width
width+1	width+2				
…					

Occupancy Grid

- Binary – obstacle is or isn't there
- Confidence – 0-100% sure of obstacle



Husarion ROS tutorials

Path Planning

- Reactive:
  - Fast, local, no guarantees
- Model
  - Longer, global, can be optimal
  - Graph search
- ROS support!