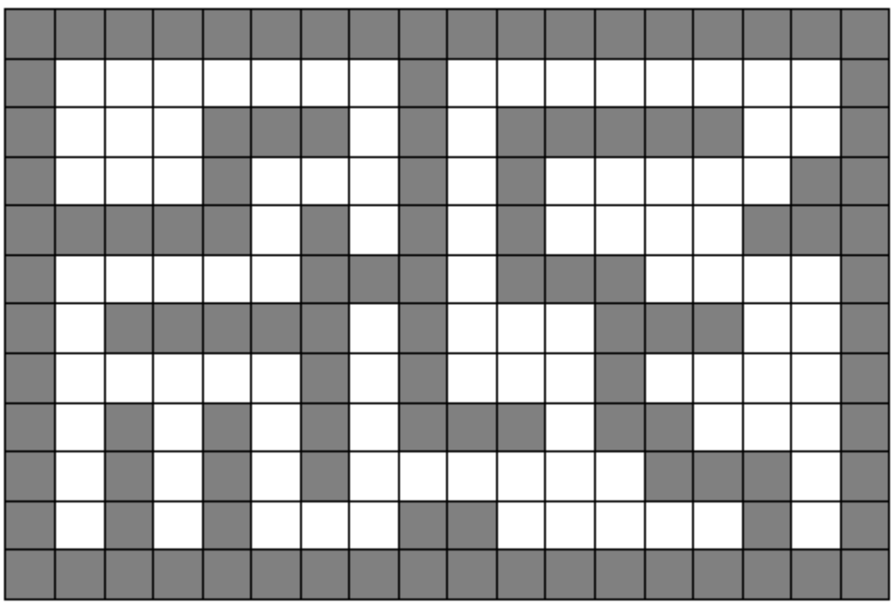
Planning (Chapter 2 from Lavalle book)



Abstraction of a planning problem

1. State space $\mathbf{s} \in \mathcal{S}$. For example, 2D coordinate of a grid $\mathbf{s} = (x, y)$.
2. Action space per state $\mathbf{u} \in \mathcal{U}(\mathbf{s})$. For example, up, down, left right movement can be encoded as $\mathcal{U}(\mathbf{s}_t) = \{(0, -1), (0, 1), (1, 0), (-1, 0)\}$.
3. State transition function $\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{u}_t)$. For example, the up-down-left-right action can be combined as addition to get the next state $\mathbf{s}_{t+1} = \mathbf{s}_t + \mathbf{u}_t$.
4. Initial State $\mathbf{s}_I \in \mathcal{S}$
5. Goal states $\mathbf{s}_G \subseteq \mathcal{S}$

A Graph

A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is defined by a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ such that each edge $e \in \mathcal{E}$ is formed by a pair of start and end vertices $e = (v_s, v_e), v_s \in \mathcal{V}, v_e \in \mathcal{V}$. The first vertex is called the start of the edge $v_s = \text{start}(e)$ and second vertex is called the end $v_e = \text{end}(e)$.

A discrete planning problem can be converted into a graph by definiting

1. Vertices as the state space $\mathcal{V} = \mathcal{S}$.
2. The action space at each state as the edges connected to that vertex/state, $\mathcal{U}(\mathbf{s}_t) = \{(\mathbf{s}_t, \mathbf{s}_j) \mid (\mathbf{s}_t, \mathbf{s}_j) \in \mathcal{E}\}$.
3. State transition function is the other end of th edge, $\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{u}_t) = \text{end}(\mathbf{u}_t)$, where $\mathbf{s}_t = \text{start}(\mathbf{u}_t)$.

Representations of Graphs

(Chapter 23 of Introduction to Algorithms by Carmen et al)

Undirected graph

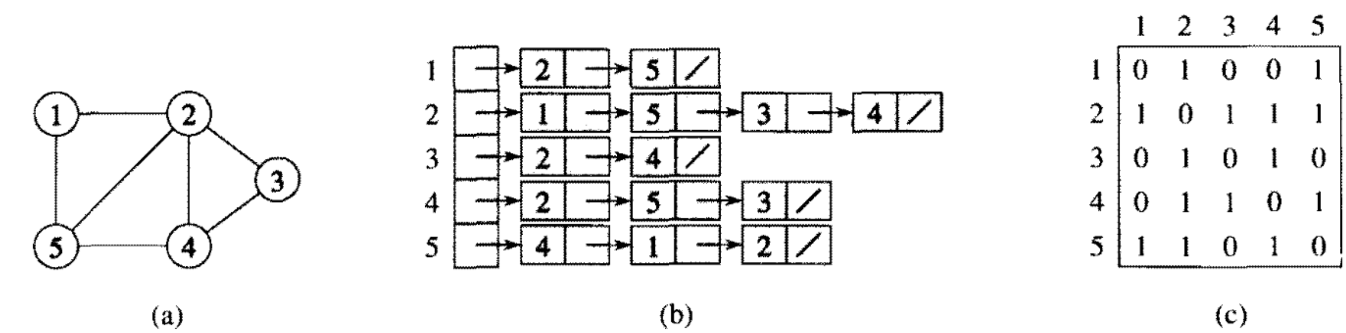


Figure 23.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

```
# Programmatically you can represent a adjacency list as python lists
# Python lists are not linked lists, they are arrays under the hood.
G_adjacency_list = {
    1 : [2, 5],
    2 : [1, 5, 3, 4],
    3 : [2, 4],
    4 : [2, 5, 3],
    5 : [4, 1, 2]
```

```
}

# Prefer to represent a matrix in python either as a list of lists or a numpy array
import numpy as np
G_adjacency_matrix = np.array([
    [0, 1, 0, 0, 1],
    [1, 0, 1, 1, 1],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 0, 1],
    [1, 1, 0, 1, 0]
])

# Edge list is another possible representation
G_edge_list = [
    (1, 2), (1, 5),
    (2, 1), (2, 5), (2, 3), (2, 4),
    (3, 2), (3, 4),
    (4, 2), (4, 5), (4, 3),
    (5, 4), (5, 1), (5, 2)
]
```

Directed graph representation

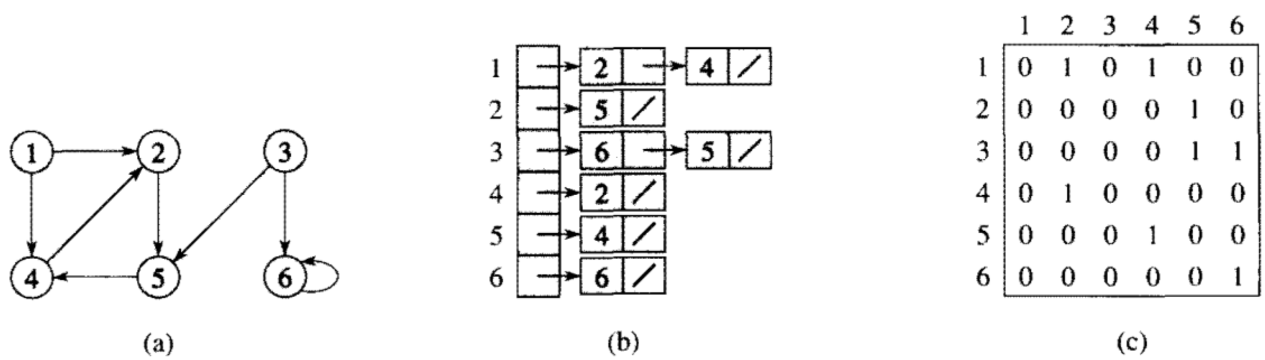


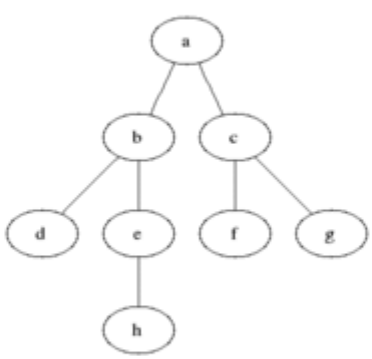
Figure 23.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

```
# Programmatically you can represent a adjacency list as python lists
# Python lists are not linked lists, they are arrays under the hood.
G_adjacency_list = {
    1 : [2, 4],
    2 : [5],
    3 : [6, 5],
    4 : [2],
    5 : [4],
    5 : [6]
}
```

```
# Prefer to represent a matrix in python either as a list of lists or a numpy array
import numpy as np
G_adjacency_matrix = np.array([
    [0, 1, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 1],
    [0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 1]
])

# Edge list is another possible representation
G_edge_list = [
    (1, 2), (1, 4),
    (2, 5),
    (3, 6), (3, 5),
    (4, 2),
    (5, 6)
]


# Exercise 1

# Write a function that converts a graph in adjacency list format to adjacency matr
def adjacency_list_to_matrix(G_adj_list):
    G_adj_mat = None # TODO: Write code to convert to adj_mat
    return G_adj_mat

def adjacency_matrix_to_list(G_adj_mat):
    G_adj_list = None # TODO: Write code to convert to adj_mat
    return G_adj_list

# Use the above graphs to test
print(adjacency_list_to_matrix(G_adjacency_list))
print(adjacency_matrix_to_list(G_adjacency_matrix))
```

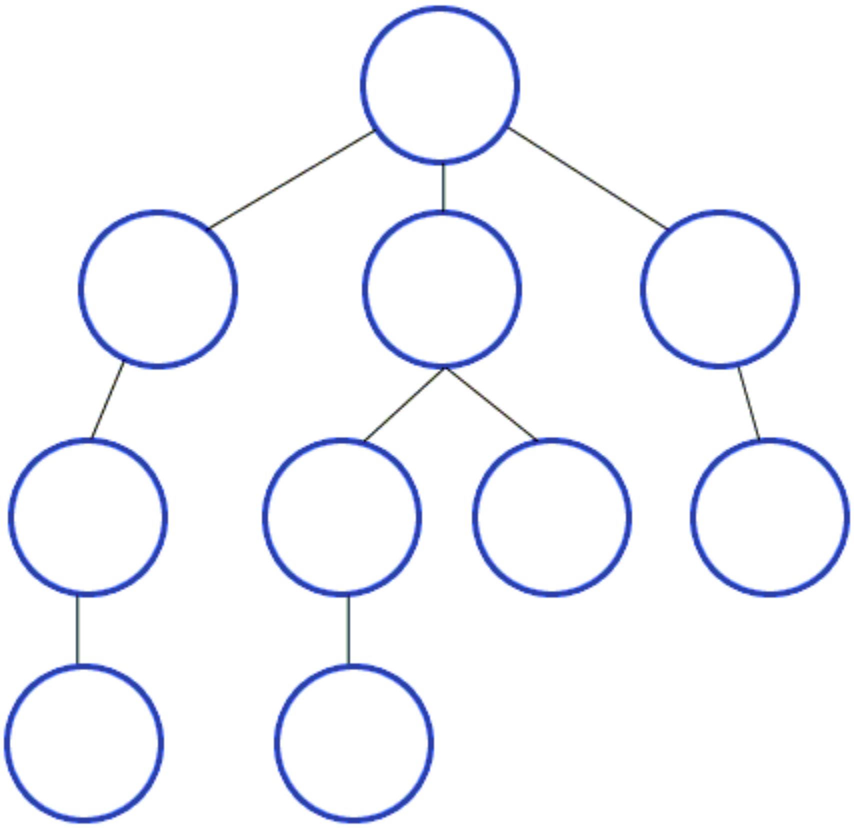
```
None
None
```

Graph Search algorithms

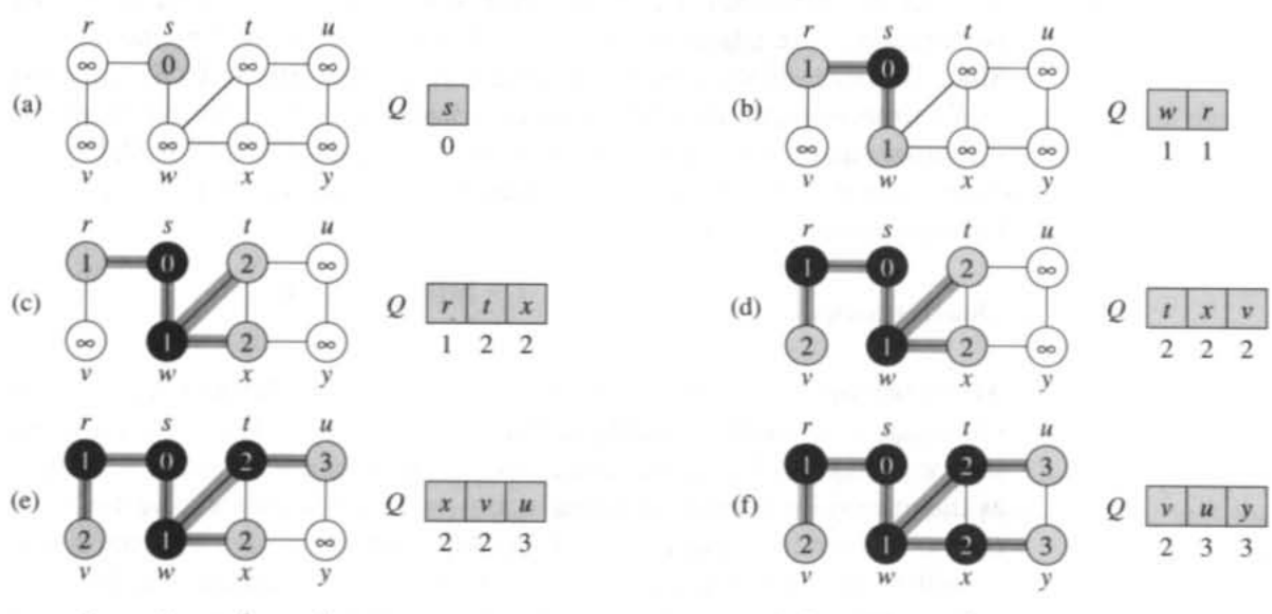


1. Breadth First Search

1. Breadth First Search
2. Depth First Search



Breadth first search (BFS)



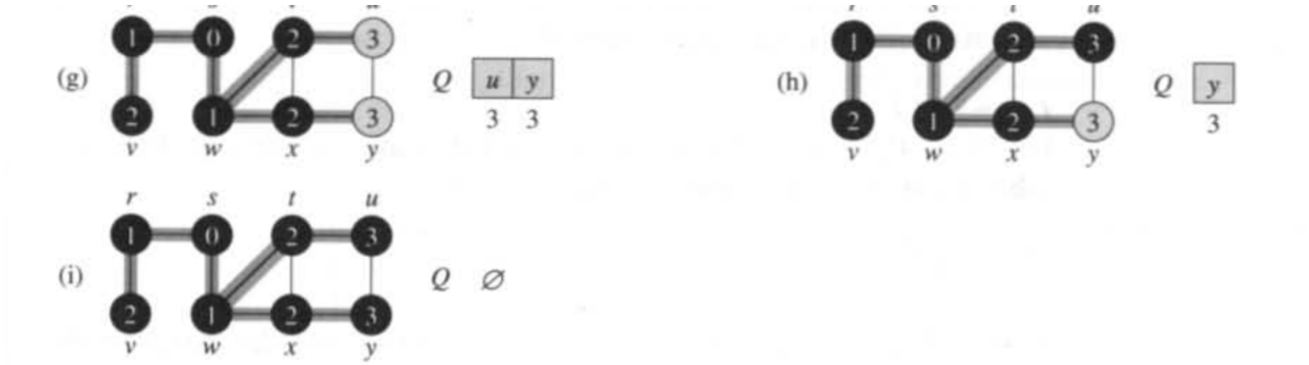


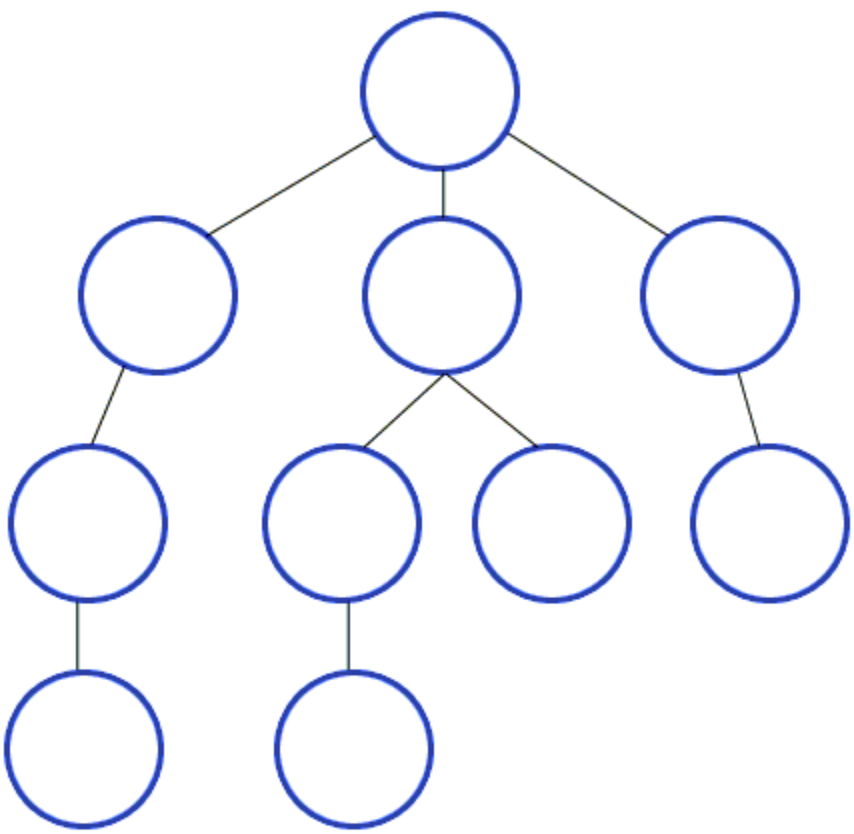
Figure 23.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 9–18. Vertex distances are shown next to vertices in the queue.

```
from queue import Queue, LifoQueue, PriorityQueue

graph = {
  's' : ['w', 'r'],
  'r' : ['v'],
  'w' : ['t', 'x'],
  'x' : ['y' ],
  't' : ['u'],
  'u' : ['y']
}

def bfs(graph, start, debug=False):
    seen = set() # Set for seen nodes (contains both frontier and dead states)
    # Frontier is the boundary between seen and unseen (Also called the alive state
    frontier = Queue() # Frontier of unvisited nodes as FIFO
    node2dist = {start : 0} # Keep track of distances
    search_order = []
    seen.add(start)
    frontier.put(start)


    i = 0 # step number
    while not frontier.empty():          # Creating loop to visit each node
        if debug: print("%d) Q = " % i, list(frontier.queue), end='; ')
        if debug: print("dists = " , [node2dist[n] for n in frontier.queue])
        m = frontier.get() # Get the oldest addition to frontier
        search_order.append(m)


        for neighbor in graph.get(m, []):
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(neighbor)
                node2dist[neighbor] = node2dist[m] + 1
```

```
            else:
                assert node2dist[neighbor] <= node2dist[m] + 1, 'this should not ha
                node2dist[neighbor] = min(node2dist[neighbor], node2dist[m] + 1)

        i += 1
    if debug: print("%d) Q = " % i, list(frontier.queue))
    return search_order, node2dist


print("Following is the Breadth-First Search order")
print(bfs(graph, 's', debug=True))    # function calling
```

```
Following is the Breadth-First Search order
0) Q =  ['s']; dists =  [0]
1) Q =  ['w', 'r']; dists =  [1, 1]
2) Q =  ['r', 't', 'x']; dists =  [1, 2, 2]
3) Q =  ['t', 'x', 'v']; dists =  [2, 2, 2]
4) Q =  ['x', 'v', 'u']; dists =  [2, 2, 3]
5) Q =  ['v', 'u', 'y']; dists =  [2, 3, 3]
6) Q =  ['u', 'y']; dists =  [3, 3]
7) Q =  ['y']; dists =  [3]
8) Q =  []
(['s', 'w', 'r', 't', 'x', 'v', 'u', 'y'], {'s': 0, 'w': 1, 'r': 1, 't': 2, ':
```

Depth first search



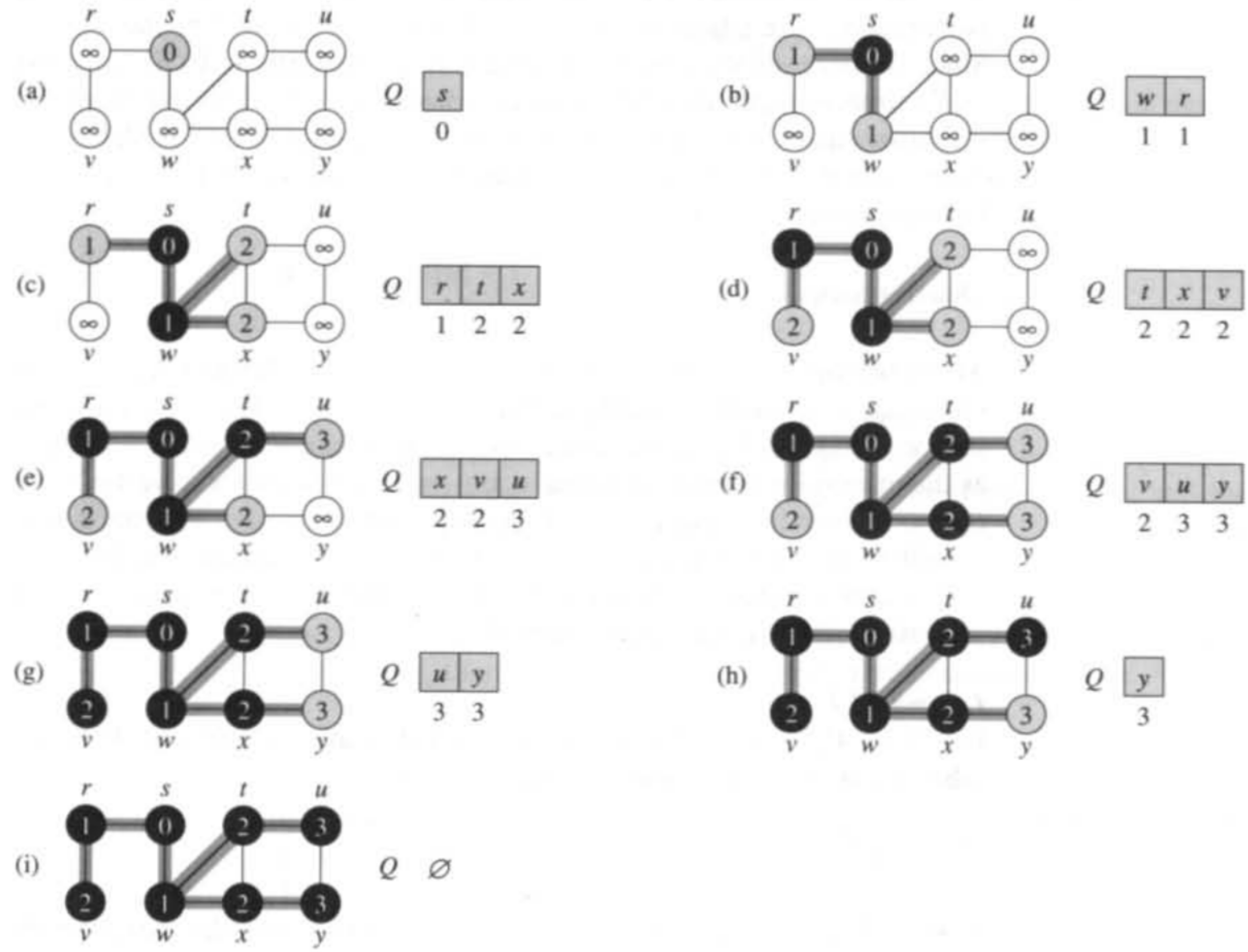


Figure 23.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 9–18. Vertex distances are shown next to vertices in the queue.

```
graph = {
  's' : ['w', 'r'],
  'r' : ['v'],
  'w' : ['t', 'x'],
  'x' : ['y' ],
  't' : ['u'],
  'u' : ['y']
}

def dfs(graph, start, debug=False):
    seen = set([start]) # List for seen nodes (contains both frontier and dead stat
    # Frontier is the boundary between seen and unseen (Also called the alive state
    frontier = LifoQueue() # Frontier of unvisited nodes as FIFO
    node2dist = {start : 0} # Keep track of distances
```

```
    search_order = [] # Keep track of search order
    frontier.put(start)


    i = 0 # step number
    while not frontier.empty():          # Creating loop to visit each node
        if debug: print("%d) Q = " % i, list(frontier.queue), end='; ')
        if debug: print("dists = " , [node2dist[n] for n in frontier.queue])
        m = frontier.get() # Get the oldest addition to frontier
        search_order.append(m)


        for neighbor in graph.get(m, []):
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(neighbor)
                node2dist[neighbor] = node2dist[m] + 1
            else:
                node2dist[neighbor] = min(node2dist[neighbor], node2dist[m] + 1)
        i += 1
    if debug: print("%d) Q = " % i, list(frontier.queue))
    return search_order, node2dist


# Driver Code
print("Following is the Depth-First Search path")
print(dfs(graph, 's', debug=True))    # function calling
```

```
Following is the Depth-First Search path
0) Q =  ['s']; dists =  [0]
1) Q =  ['w', 'r']; dists =  [1, 1]
2) Q =  ['w', 'v']; dists =  [1, 2]
3) Q =  ['w']; dists =  [1]
4) Q =  ['t', 'x']; dists =  [2, 2]
5) Q =  ['t', 'y']; dists =  [2, 3]
6) Q =  ['t']; dists =  [2]
7) Q =  ['u']; dists =  [3]
8) Q =  []
(['s', 'r', 'v', 'w', 'x', 'y', 't', 'u'], {'s': 0, 'w': 1, 'r': 1, 'v': 2, '
```

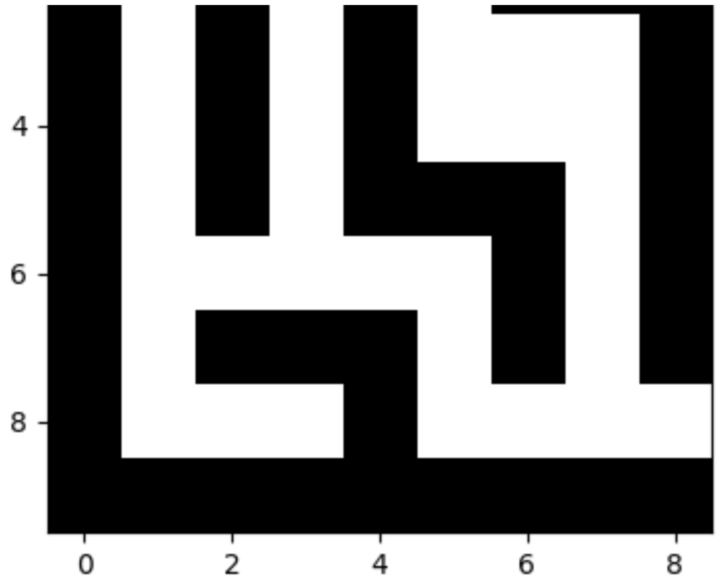
Converting a maze search to a graph search

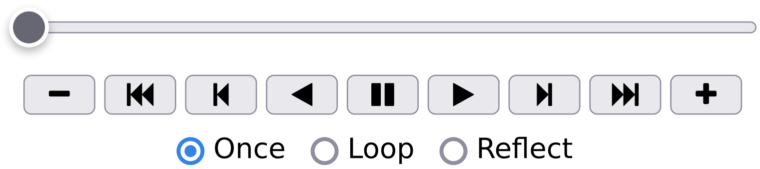
```
# Skip these utilities for the class

def batched(iterable, n):
    "Batch data into tuples of length n. The last batch may be shorter."
    # batched('ABCDEFG', 3) --> ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    it = iter(iterable)
```

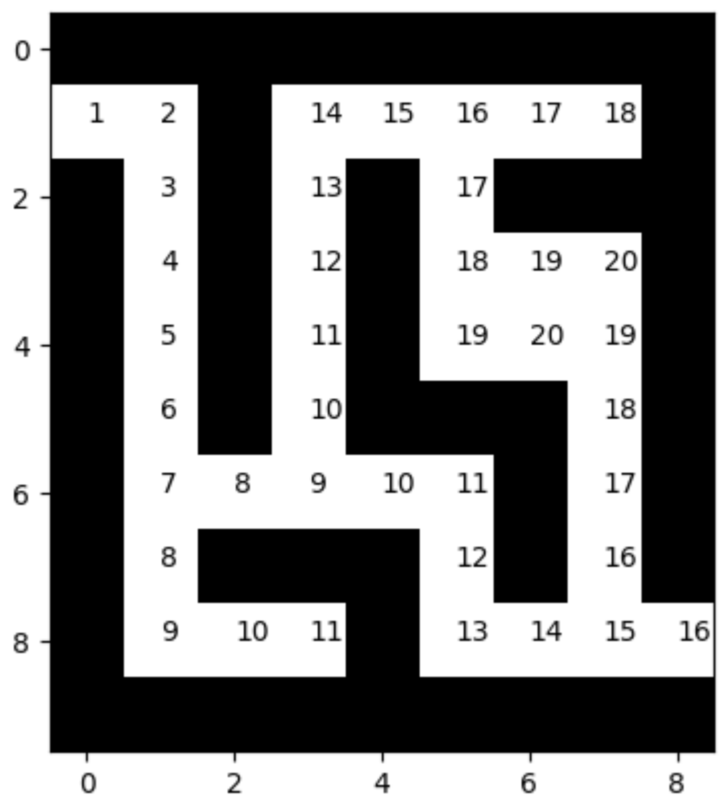
```
    while batch := tuple(islice(it, n)):
        yield batch

def draw_path(self, path, visited='*'):
    new_maze_lines = [list(l) for l in self.maze_lines]
    for (r, c) in path:
        new_maze_lines[r][c] = visited
        print('\n'.join([''.join(l) for l in new_maze_lines]))
        print('\n\n\n')

def init_plots(self, reinit=False):
    if self.fig is None or reinit:
        self.fig, self.ax = plt.subplots()

def plot_maze(self):
    self.init_plots()
    replace = { ' ' : 1, '+': 0}
    maze_mat = np.array([[replace[c] for c in line]
                          for line in self.maze_lines])
    return [self.ax.imshow(maze_mat, cmap='gray')]

def plot_step(self, i_node):
    i, (r, c) = i_node
    return [self.ax.text(c, r, '%d' % (i+1))]

def plot_path(self, path):
    self.plot_maze()
    return [self.plot_step((i, (r,c)))
            for i, (r, c) in enumerate(path)]

def animate_search_path(maze, search_path, node2dist):
    maze.init_plots()
    return animation.FuncAnimation(maze.fig, maze.plot_step, frames=[(node2dist[n],
                                                                      for n in sear
                                   init_func=maze.plot_maze, blit=True, repeat=False
```

```
import matplotlib.pyplot as plt
import numpy as np
maze_str = \
"""
+++++++++
  +     +
+ + + +++
+ + +   +
+ + +   +
+ + +++ +
+     + +
+ +++ + +
+   +
+++++++++
```

```
..........
"""

class Maze:
    def __init__(self, maze_str, freepath=' '):
        self.maze_lines = [l for l in maze_str.split("\n")
                           if len(l)]
        self.FREEPATH = freepath
        self.fig = None

    def get(self, node, default):
        (r, c) = node
        m_row = self.maze_lines[r]
        nbrs = []
        if c-1 >= 0 and m_row[c-1] == self.FREEPATH:
            nbrs.append((r, c-1))
        if c+1 < len(m_row) and m_row[c+1] == self.FREEPATH:
            nbrs.append((r, c+1))
        if r-1 >= 0 and self.maze_lines[r-1][c] == self.FREEPATH:
            nbrs.append((r-1, c))
        if r+1 < len(self.maze_lines) and self.maze_lines[r+1][c] == self.FREEPATH:
            nbrs.append((r+1, c))
        return nbrs if len(nbrs) else default
    init_plots = init_plots
    plot_maze = plot_maze
    plot_step = plot_step
    plot_path = plot_path
    animate_search_path = animate_search_path


import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib as mpl
%matplotlib inline
mpl.rc('animation', html='jshtml')


maze = Maze(maze_str)
search_path, node2dist = bfs(maze, (1, 0)) # prints the order of search all the sea
maze.plot_maze()

maze.animate_search_path(search_path, node2dist)
```

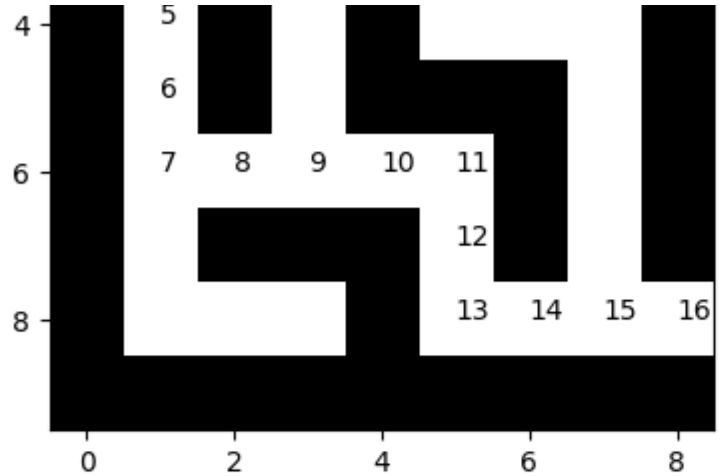







```
def bfs_path(graph, start, goal):
    """
    Returns success and node2parent

    success: True if goal is found otherwise False
    node2parent: A dictionary that contains the nearest parent for node
```

```
    """
    seen = [start] # List for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = Queue() # Frontier of unvisited nodes as FIFO
    node2parent = dict() # Keep track of nearest parent for each node (requires noc
    frontier.put(start)

    while not frontier.empty():          # Creating loop to visit each node
        m = frontier.get() # Get the oldest addition to frontier
        if m == goal:
            return True, node2parent

        for neighbor in graph.get(m, []):
            if neighbor not in seen:
                seen.append(neighbor)
                frontier.put(neighbor)
                node2parent[neighbor] = m
    return False, []


def backtrace_path(node2parent, start, goal):
    c = goal
    r_path = [c]
    parent = node2parent.get(c, None)
    while parent != start:
        r_path.append(parent)
        c = parent
        parent = node2parent.get(c, None) # Keep getting the parent until you reach
        #print(parent)
    r_path.append(start)
    return reversed(r_path) # Reverses the path

maze = Maze(maze_str)
start = (1, 0)
goal = (8, 8)
success, node2parent = bfs_path(maze, (1, 0), (8, 8))
path = backtrace_path(node2parent, (1, 0), (8, 8))
#print(list(path))
maze.plot_path(path) # Draws all the searched nodes
plt.show()
#node2parent
```



Dijkstra algorithm

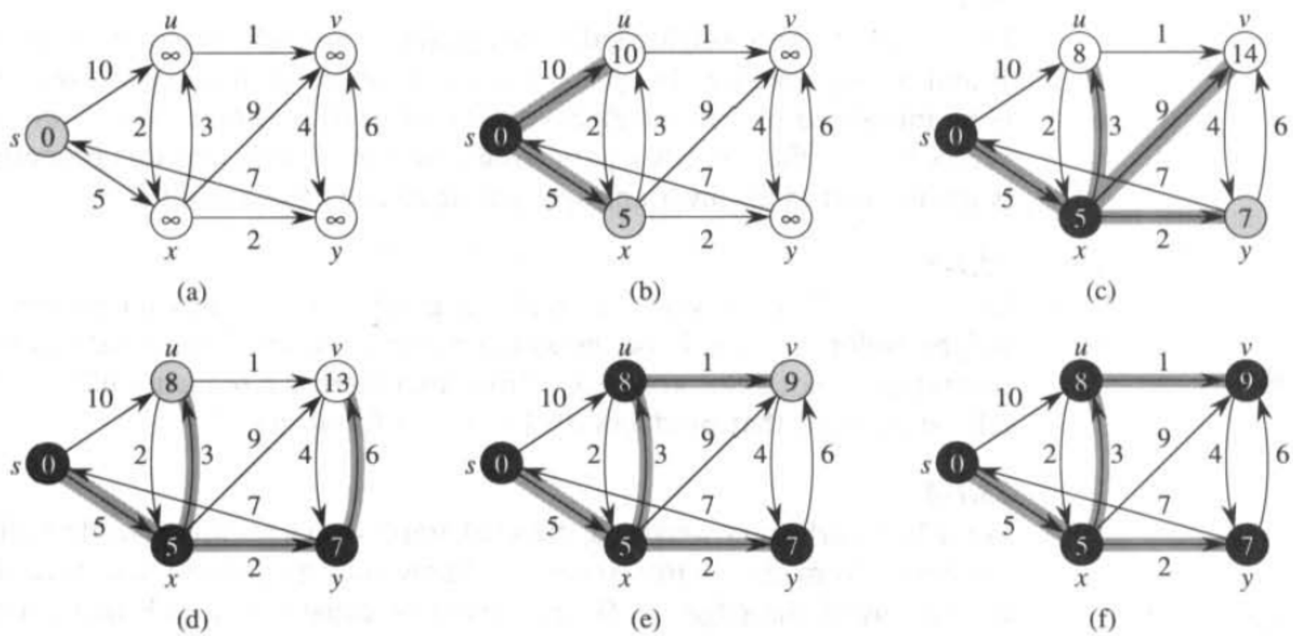


Figure 25.5 The execution of Dijkstra's algorithm. The source is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. Black vertices are in the set S, and white vertices are in the priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d and π values shown in part (f) are the final values.

PriorityQueue

PriorityQueue returns the smallest (or the largest) item in the queue faster than other data structures

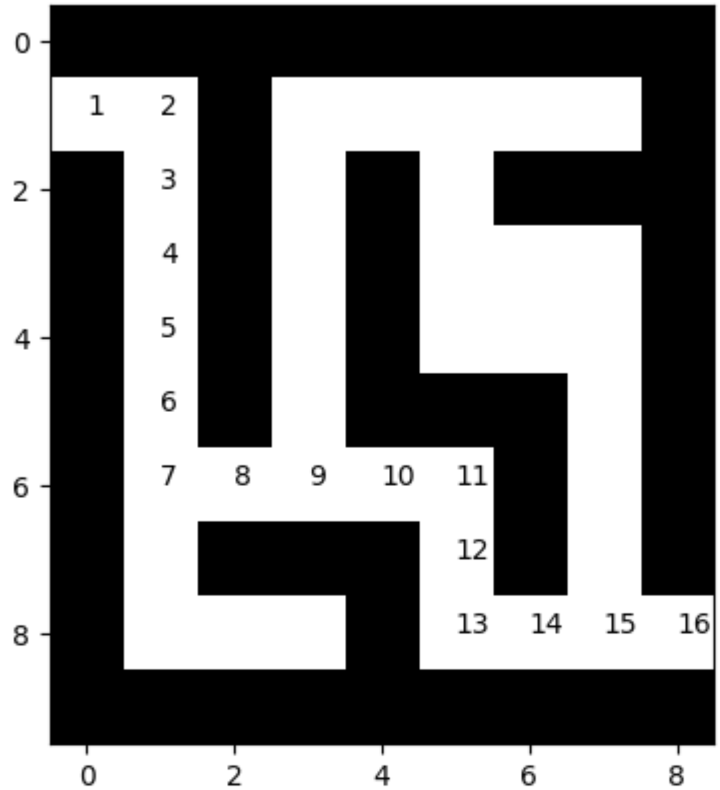
```
#from queue import PriorityQueue
from hw2_solution import PriorityQueueUpdatable
from dataclasses import dataclass, field
from typing import Any


# https://docs.python.org/3/library/queue.html#queue.PriorityQueue
@dataclass(order=True)
class PItem:
    dist: int
    node: Any=field(compare=False)

    # Make the PItem hashable
    # https://docs.python.org/3/glossary.html#term-hashable
    def __hash__(self):
        return hash(self.node)

graph = {
    's' : [('x', 5), ('u', 10)],
    'u' : [('v', 1), ('x', 2)],
    'x' : [('u', 3), ('v', 9), ('y', 2)],
    'y' : [('v', 6), ('s', 7)],
    'v' : [('y', 4)]
}


def dijkstra(graph, start, goal, debug=False):
    """
    edgecost: cost of traversing each edge

    Returns success and node2parent

    success: True if goal is found otherwise False
    node2parent: A dictionary that contains the nearest parent for node
    """
    seen = set([start]) # Set for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = PriorityQueueUpdatable() # Frontier of unvisited nodes as a Priority
    node2parent = {start : None} # Keep track of nearest parent for each node (requ
    node2dist = {start: 0} # Keep track of cost to arrive at each node
    search_order = []
    frontier.put(PItem(0, start))
    i = 0
    while not frontier.empty():          # Creating loop to visit each node
        dist_m = frontier.get() # Get the smallest addition to the frontier
        if debug: print("%d) Q = " % i, list(frontier.queue), end='; ')
        if debug: print("dists = " , [node2dist[n.node] for n in frontier.queue])
        m = dist_m.node
        m_dist = node2dist[m]
```

```
        m_dist = node2dist[m]
        search_order.append(m)
        if goal is not None and m == goal:
            return True, search_order, node2parent, node2dist

        for neighbor, edge_cost in graph.get(m, []):
            old_dist = node2dist.get(neighbor, float("inf"))
            new_dist = edge_cost + m_dist
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(PItem(new_dist, neighbor))
                node2parent[neighbor] = m
                node2dist[neighbor] = new_dist
            elif new_dist < old_dist:
                node2parent[neighbor] = m
                node2dist[neighbor] = new_dist
                # ideally you would update the dist of this item in the priority qu
                # as well. But python priority queue does not support fast updates
                old_item = PItem(old_dist, neighbor)
                if old_item in frontier:
                    frontier.replace(old_item, PItem(new_dist, neighbor))
        i += 1
    if goal is not None:
        return False, [], {}, node2dist
    else:
        return True, search_order, node2parent, node2dist


success, search_path, node2parent, node2dist = dijkstra(graph, 's', None, debug=Tru
print(success, node2parent, node2dist)
```

```
0) Q =  []; dists =  []
1) Q =  [PItem(dist=10, node='u')]; dists =  [10]
2) Q =  [PItem(dist=8, node='u'), PItem(dist=14, node='v')]; dists =  [8, 14]
3) Q =  [PItem(dist=13, node='v')]; dists =  [13]
4) Q =  []; dists =  []
True {'s': None, 'x': 's', 'u': 'x', 'v': 'u', 'y': 'x'} {'s': 0, 'x': 5, 'u'
```

```
import itertools

class MazeD(Maze):
    def get(self, node, default):
        nbrs = Maze.get(self, node, default)
        return zip(nbrs, itertools.repeat(1))

maze = MazeD(maze_str)
success, search_path, node2parent, node2dist = dijkstra(maze, (1, 0), (8, 8))
print(success, node2parent)
if success:
    path = backtrace_path(node2parent, (1, 0), (8, 8))
    maze.plot_path(path) # Draws all the searched nodes
```

```
True {(1, 0): None, (1, 1): (1, 0), (2, 1): (1, 1), (3, 1): (2, 1), (4, 1): (
```

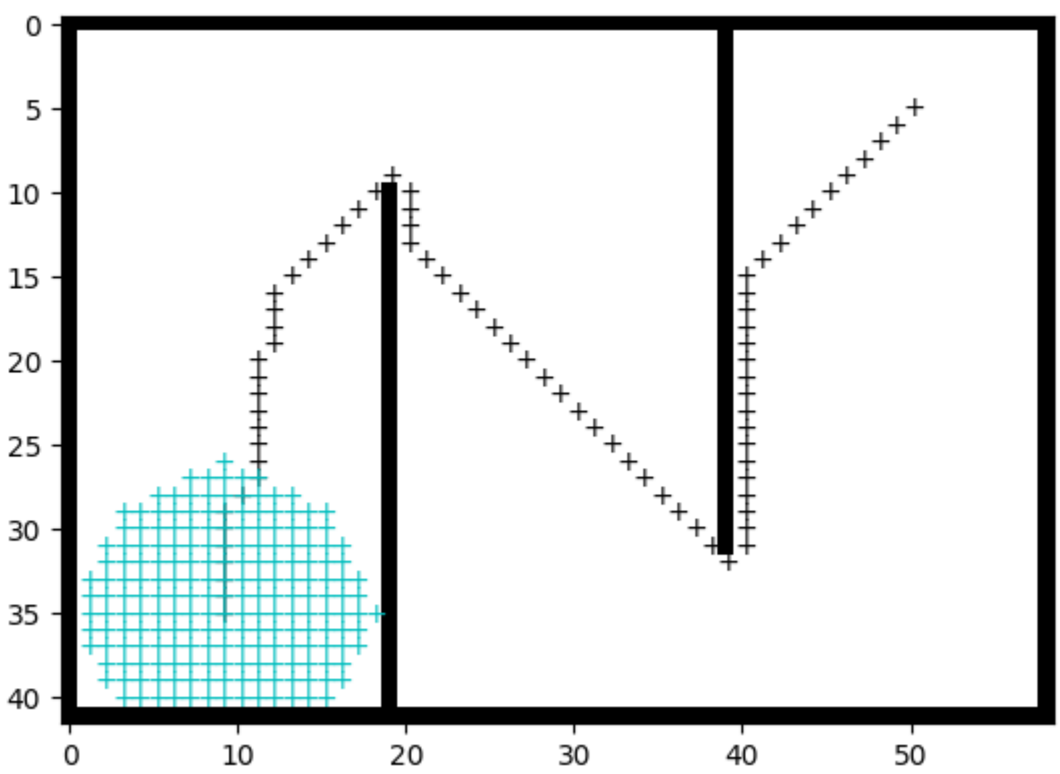


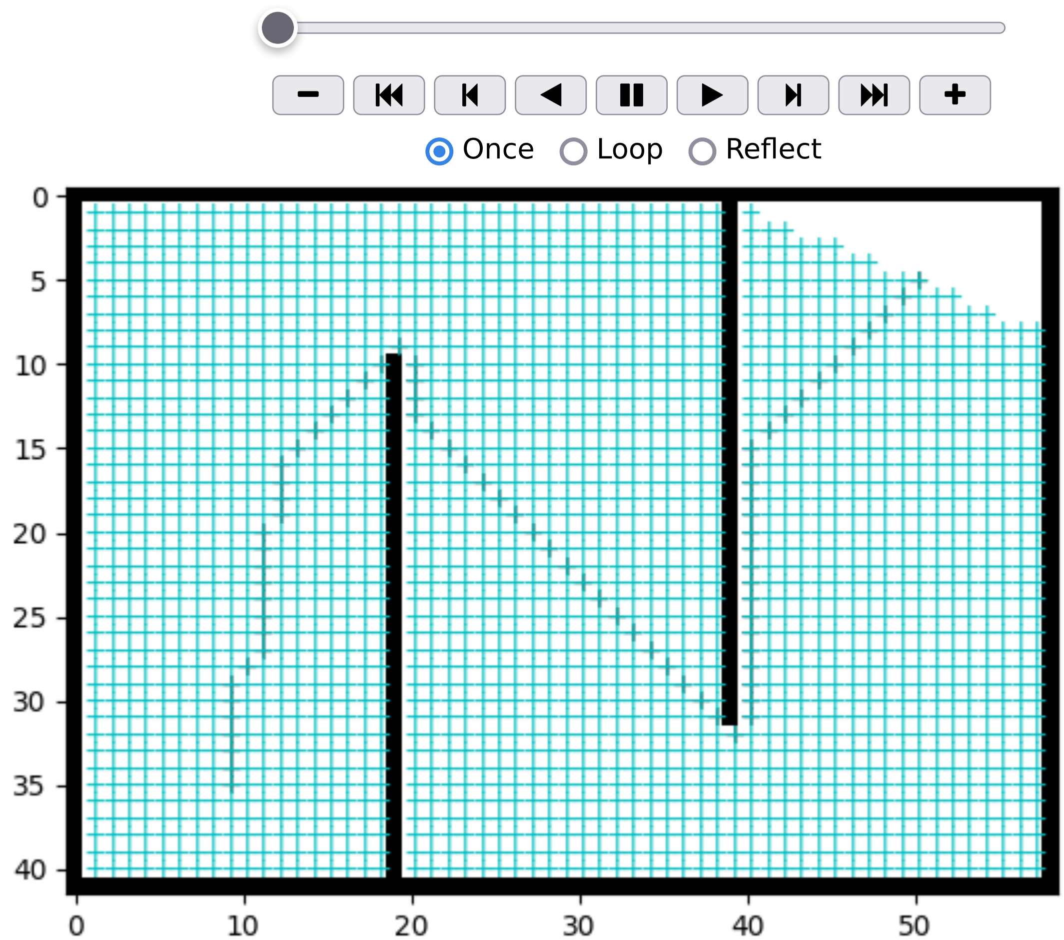

```
maze_str = \
"""
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                                       +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
+                  +                    +                  +
```

```
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
"""
start_pos, goal_pos = (35, 9), (5, 50)


import math
from itertools import islice

class Maze8(MazeD):
    def get(self, node, default):
        (r, c) = node
        rmax = len(self.maze_lines)
        cmax = len(self.maze_lines[0])
        m_row = self.maze_lines[r]
        possible_nbrs = [
            ((r, c-1), 1),
            ((r, c+1), 1),
            ((r-1, c), 1),
            ((r+1, c), 1),
            ((r-1, c-1), math.sqrt(2)),
            ((r-1, c+1), math.sqrt(2)),
            ((r+1, c-1), math.sqrt(2)),
            ((r+1, c+1), math.sqrt(2))
        ]
        free_nbrs = []
        for (ri, ci), dist in possible_nbrs:
            if (ri >= 0 and ci >= 0 and ri < rmax and ci < cmax
                    and self.maze_lines[ri][ci] == self.FREEPATH):
                free_nbrs.append(((ri, ci), dist))
        return free_nbrs if len(free_nbrs) else default

    def _plot_path(self, path, char='+', color='c'):
```

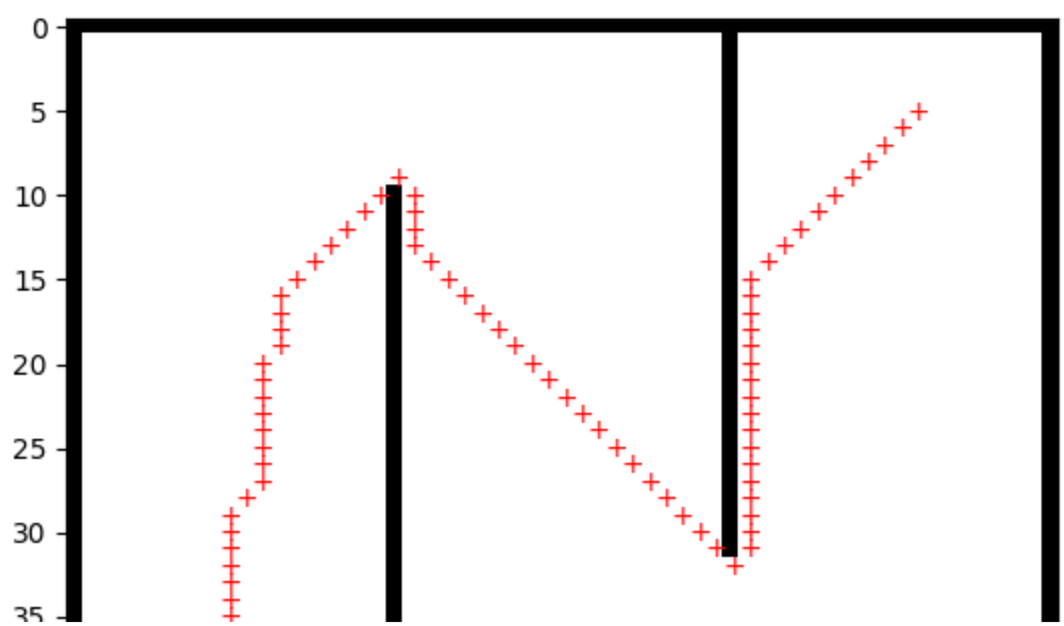
```
        return [self.ax.text(c-0.5, r+0.5, char, color=color)
               for (r, c) in path]

    def plot_path(self, path, **kw):
        self.plot_maze()
        return self._plot_path(path, **kw)

    def animate(self, path, batch_size=200):
        self.init_plots()
        anim = animation.FuncAnimation(self.fig, self._plot_path,
                                        frames=batched(search_path, batch_size),
                                       init_func=self.plot_maze, blit=True, repeat=F
        return anim


maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = dijkstra(maze, start_pos, goal_pos)
#print(success, search_path)
assert success
anim = maze.animate(search_path)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim
```

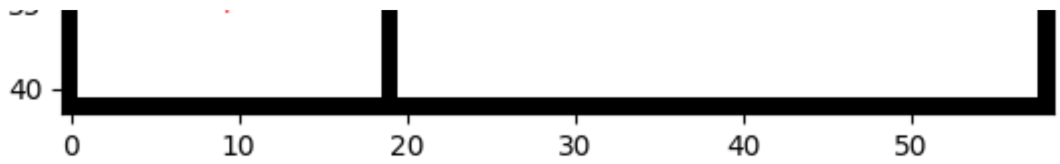
```
/tmp/ipykernel_240277/955263672.py:37: UserWarning: frames=<generator object l
  anim = animation.FuncAnimation(self.fig, self._plot_path,
```





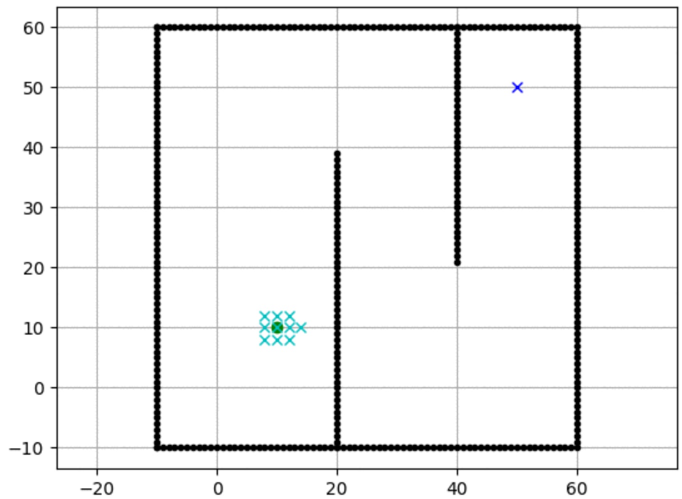

```
path = backtrace_path(node2parent, (35, 9), (5, 50))
maze.init_plots(reinit=True)
maze.plot_path(path, color='r') # Draws the traced shortest path
plt.show()
```

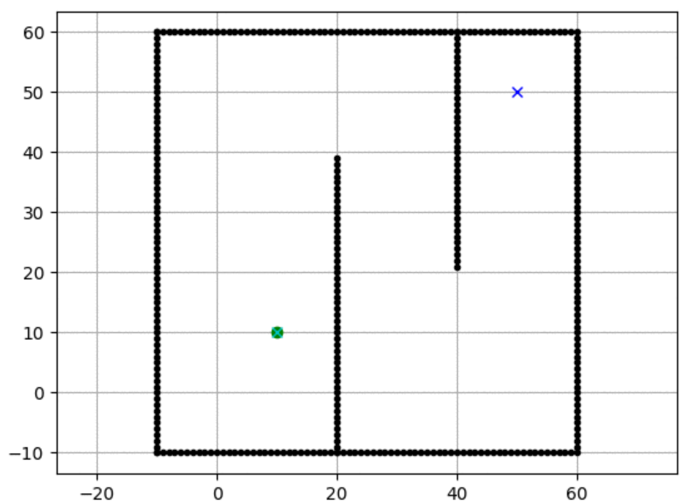




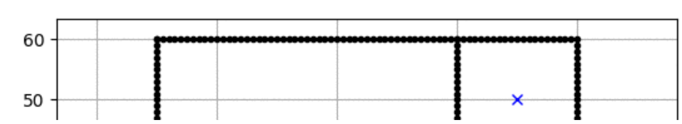
Search order in BFS vs DFS vs Dijkstra

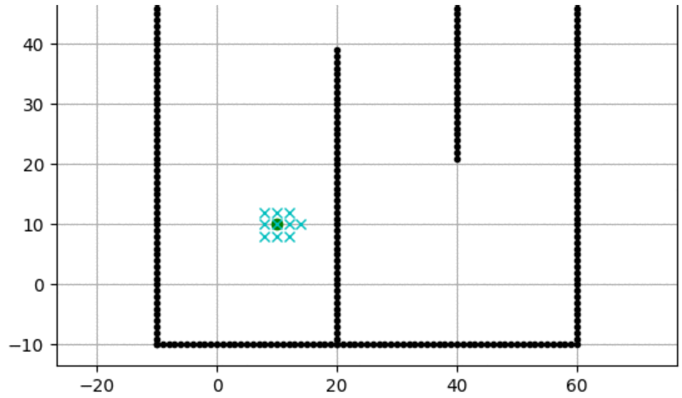
Breadth first search vs Depth first search

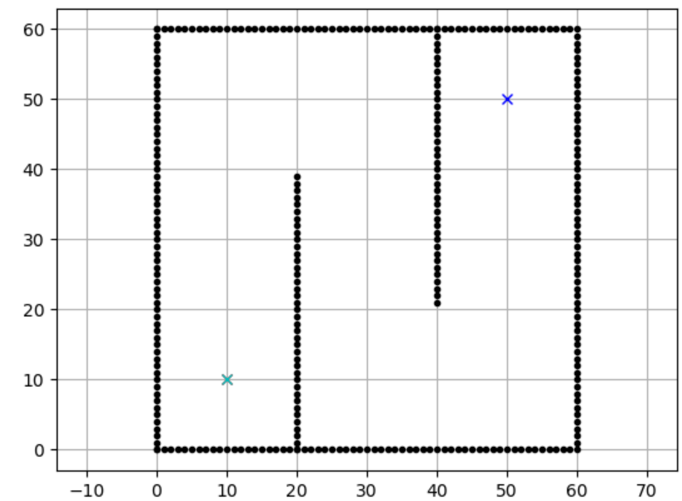




Breadth first search vs Dijkstra







Computational complexity of BFS

```
# Write down the computational complexity of each line in big-O notation O()
# Assume the graph has |V| nodes and |E| edges
def bfs_barebones(graph, start):
    seen = {start} # Set for seen nodes (contains both frontier and dead states) #
    # Frontier is the boundary between seen and unseen (Also called the alive state
    frontier = Queue() # Frontier of unvisited nodes as FIFO # O(1)
    frontier.put(start) # O(1)

    while not frontier.empty(): # Creating loop to visit each node # O(|V|)
        m = frontier.get() # Get the oldest addition to frontier # O(|V| * 1)

        for neighbor in graph.get(m, []): # O(|V| * |E|/|V|) = O(|E|)
            if neighbor not in seen: # O(|E| * 1)
                seen.add(neighbor) # O(|E| * 1)
                frontier.put(neighbor) # O(|E| * 1)
# The computational complexity of BFS is O(|E|). Some books write it as O(|V| + |E|
# where O(|V|) is the cost of initializing states of different nodes
```

Computational complexity of Dijkstra

```
# Write down the computational complexity of each line in big-O notation O()
# Assume the graph has |V| nodes and |E| edges
def dijkstra_barebones(graph, start):
    seen = {start} # Set for seen nodes (contains both frontier and dead states) #
    # Frontier is the boundary between seen and unseen (Also called the alive state
    frontier = PriorityQueue() # Frontier of unvisited nodes as PriorityQueue # O(1
    frontier.put(PItem(0, start)) # O(1)
    node2dist = {start: 0} # Keep track of cost to arrive at each node # O(1)

    while not frontier.empty(): # Creating loop to visit each node
        dist_and_node = frontier.get() # Get the smallest dist node # O(|V| * log(|
        m_dist = dist_and_node.dist
        m = dist_and_node.node

        for neighbor, edge_dist in graph.get(m, []): # O(|V| * |E|/|V|) = O(|E|)
            if neighbor not in seen: # O(|E| * 1)
                seen.add(neighbor) # O(|E| * 1)
                frontier.put(neighbor) # # O(|E| * log(1)) # for fibonacci heap
                node2dist[neighbor] = m_dist + edge_dist # O(1)
            elif node2dist[neighbor] > m_dist + edge_dist: # O(1)
                node2dist[neighbor] = m_dist + edge_dist # O(1)

# The computational complexity of Dijkstra is O(|V|log(|V|) + |E|) when implementec
# using a Fibonacci heap based PriorityQueue
```

PriorityQueue (Heaps Chapter 7 of Carmen's intro to algorithms)

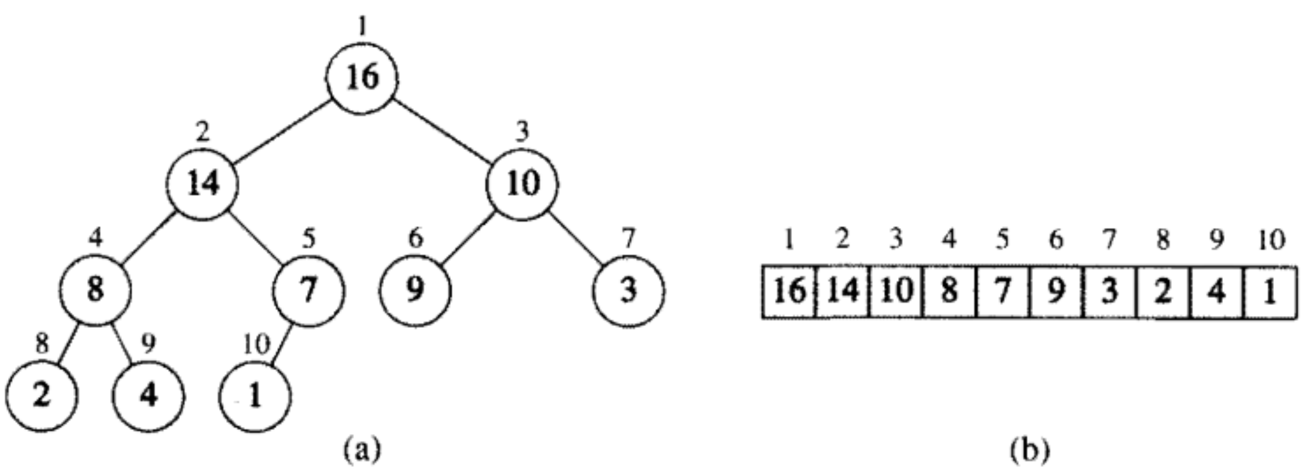


Figure 7.1 A heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

Heap property

1. $H[\text{Parent(i)}] \geq H[i]$
2. Parent(i) = ceil(i/2)
3. LeftChild(i) = 2i
4. RightRight(i) = 2i+1

Heapify

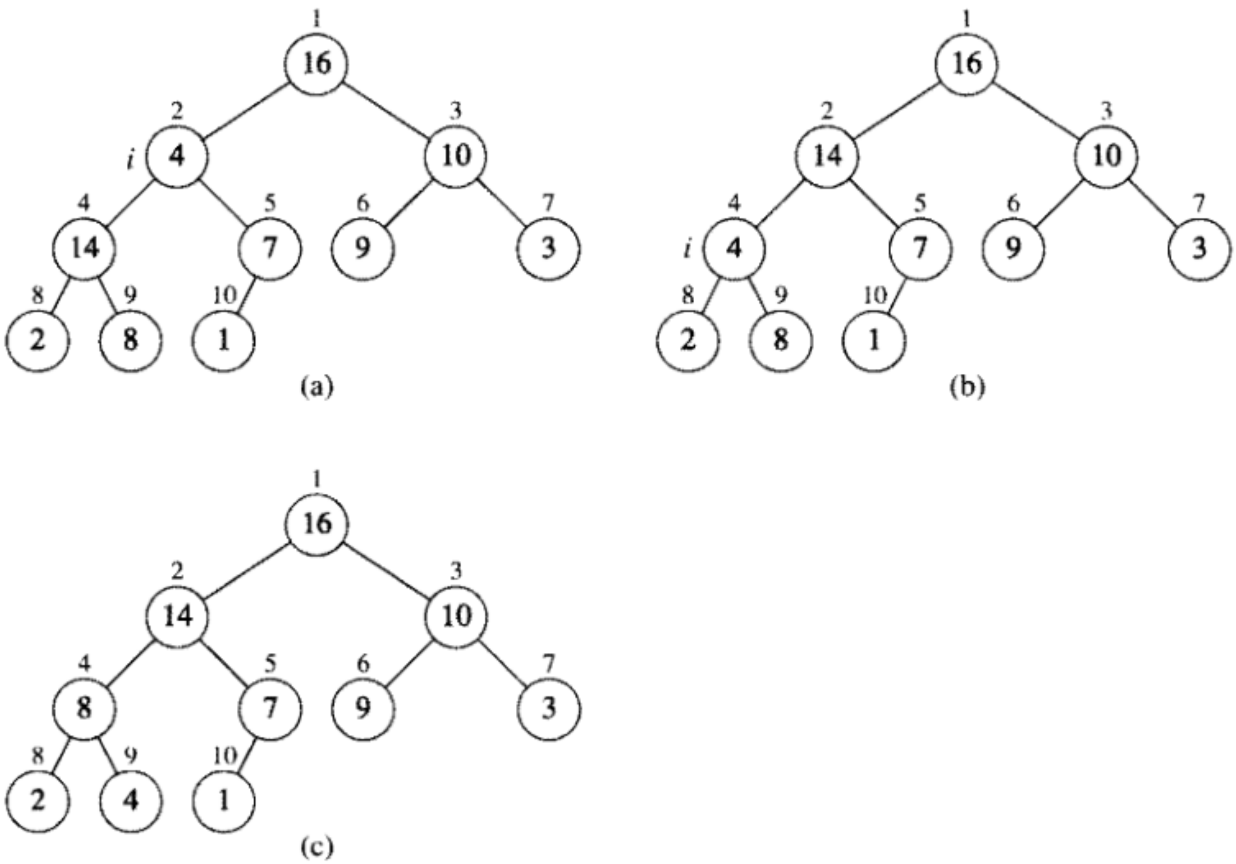


Figure 7.2 The action of HEAPIFY$(A, 2)$, where *heap-size*$[A] = 10$. **(a)** The initial configuration of the heap, with $A[2]$ at node $i = 2$ violating the heap property since it is not larger than both children. The heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the heap property for node 4. The recursive call HEAPIFY$(A, 4)$ now sets $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call HEAPIFY$(A, 9)$ yields no further change to the data structure.

Heapify pseudocode

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        HEAPIFY(A, largest)
```

Heap Insert

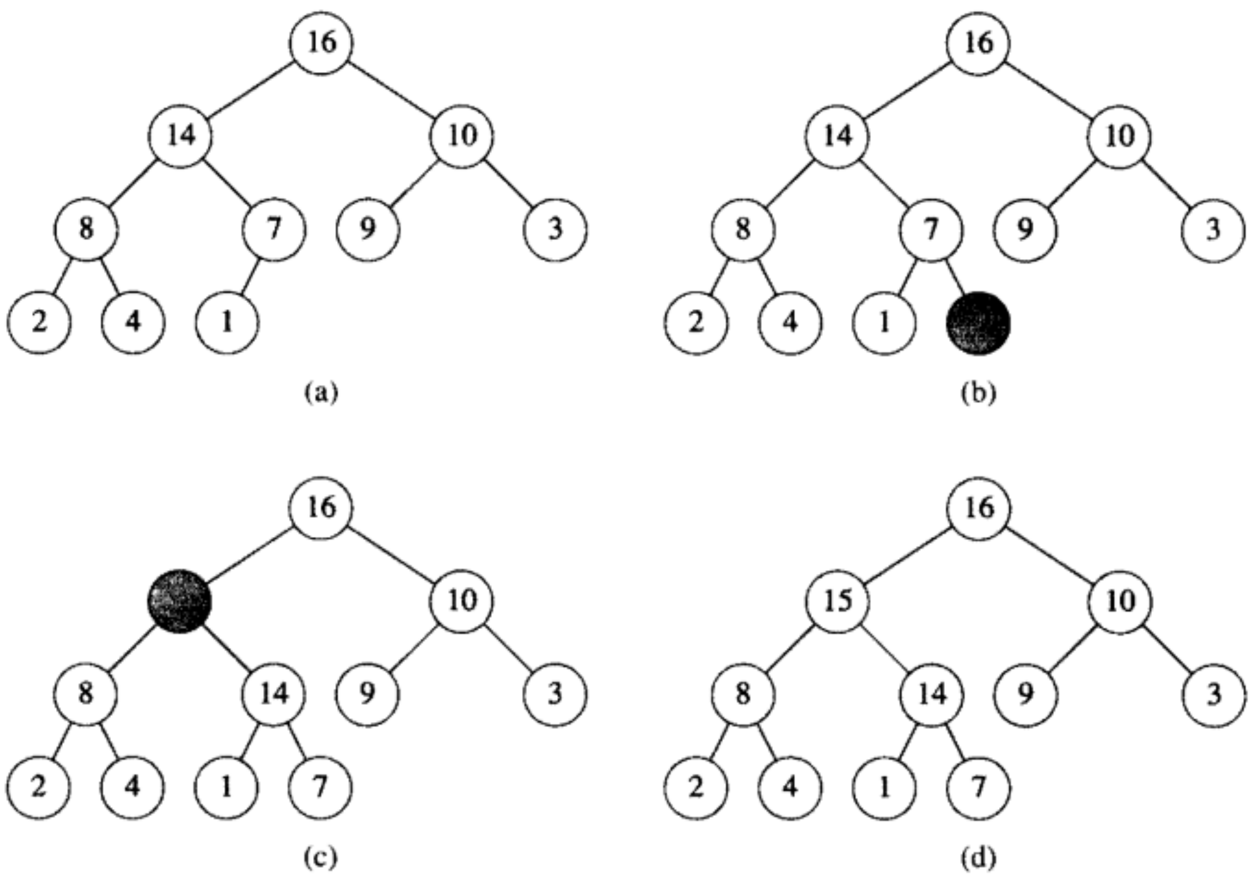


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

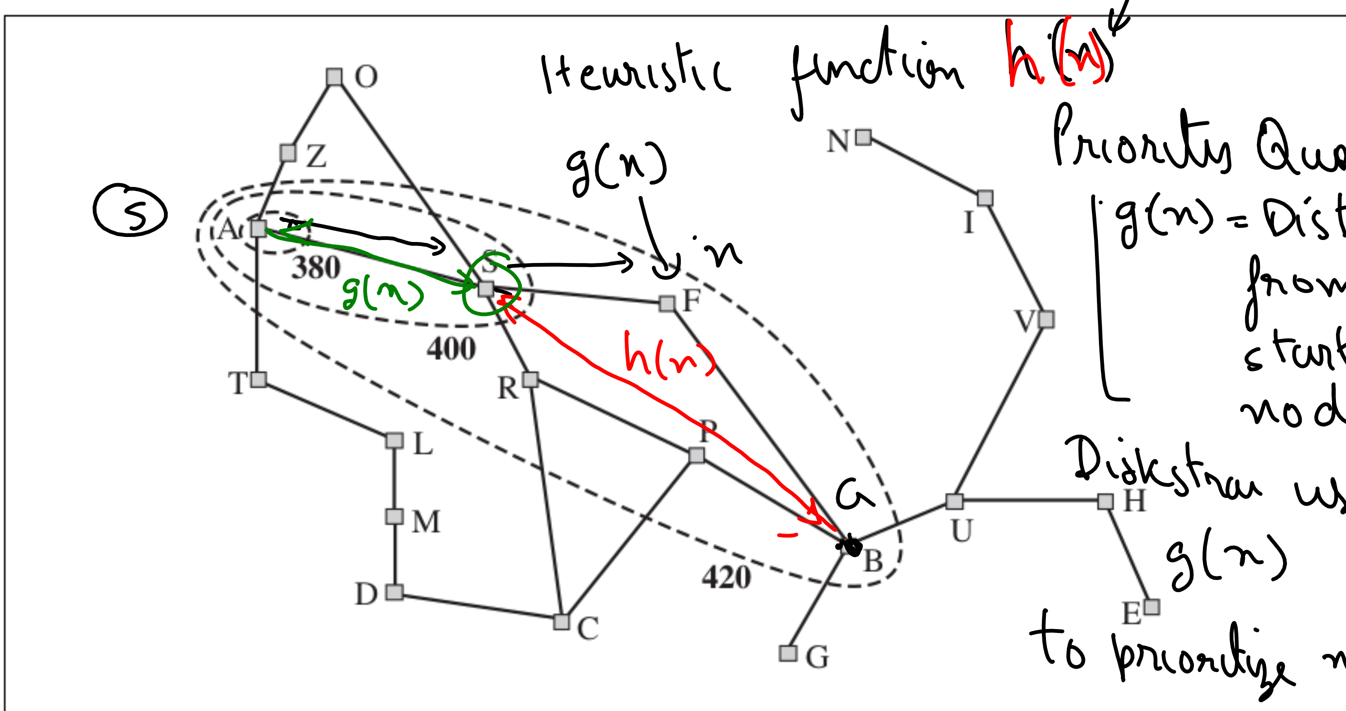
Heap runtimes

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[9]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[9][10]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[a]	$\Theta(\log n)$	$O(\log n)$
Skew binomial[11]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$[b]
Pairing[12]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$o(\log n)$[a][c]	$\Theta(1)$
Rank-pairing[15]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Fibonacci[9][2]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Strict Fibonacci[16]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Brodal[17][d]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap[19]	$O(\log n)$	$O(\log n)$[a]	$O(\log n)$[a]	$\Theta(1)$	?

A-star (A*) algorithm

(Required reading: 3.5.2 of Russel and Norving: Artificial Intelligence)

(optimistic best case distance to goal)



Heuristic function $h(n)$

$g(n)$ n

S

$g(n)$

$h(n)$

G

Priority Queue
$g(n)$ = Dist from start node

Dijkstra uses $g(n)$ to prioritize node

Astar uses $f(n) = \underline{g(n)} + h(n)$ to prioritize the Queue

```
from hw2_solution import PriorityQueueUpdatable
import sys
```

```
def astar(graph, heuristic_dist_fn, start, goal, debug=False, debugf=sys.stdout):
    """
    edgecost: cost of traversing each edge

    Returns success and node2parent

    success: True if goal is found otherwise False
    node2parent: A dictionary that contains the nearest parent for node
    """
    seen = set([start]) # Set for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = PriorityQueueUpdatable() # Frontier of unvisited nodes as a Priority
    node2parent = {start : None} # Keep track of nearest parent for each node (requ
    hfn = heuristic_dist_fn # make the name shorter
    node2dist = {start: 0  } # Keep track of cost to arrive at each node
    search_order = []
    frontier.put(PItem(0 + hfn(start, goal), start)) #   <------------- Different f

    if debug: debugf.write("goal = "  + str(goal) + '\n')
    i = 0
    while not frontier.empty():          # Creating loop to visit each node
        dist_m = frontier.get() # Get the smallest addition to the frontier
        if debug: debugf.write("%d) Q = " % i + str(list(frontier.queue)) + '\n')
        if debug: debugf.write("%d) node = " % i + str(dist_m) + '\n')
        #if debug: print("dists = " , [node2dist[n.node] for n in frontier.queue])
        m = dist_m.node
        m_dist = node2dist[m]
        search_order.append(m)
        if goal is not None and m == goal:
            return True, search_order, node2parent, node2dist

        for neighbor, edge_cost in graph.get(m, []):
            old_dist = node2dist.get(neighbor, float("inf"))
            new_dist = edge_cost +  m_dist
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(PItem(new_dist +  hfn(neighbor, goal), neighbor)) # <-
                node2parent[neighbor] = m
                node2dist[neighbor] = new_dist
            elif new_dist < old_dist:
                node2parent[neighbor] = m
                node2dist[neighbor] = new_dist
                # ideally you would update the dist of this item in the priority qu
                # as well. But python priority queue does not support fast updates
                # ------------- Different from dijkstra --------------------
                old_item = PItem(old_dist + hfn(neighbor, goal), neighbor)
                if old_item in frontier:
                    frontier.replace(
                        old_item,
                        PItem(new_dist + hfn(neighbor, goal), neighbor))
        i += 1
```

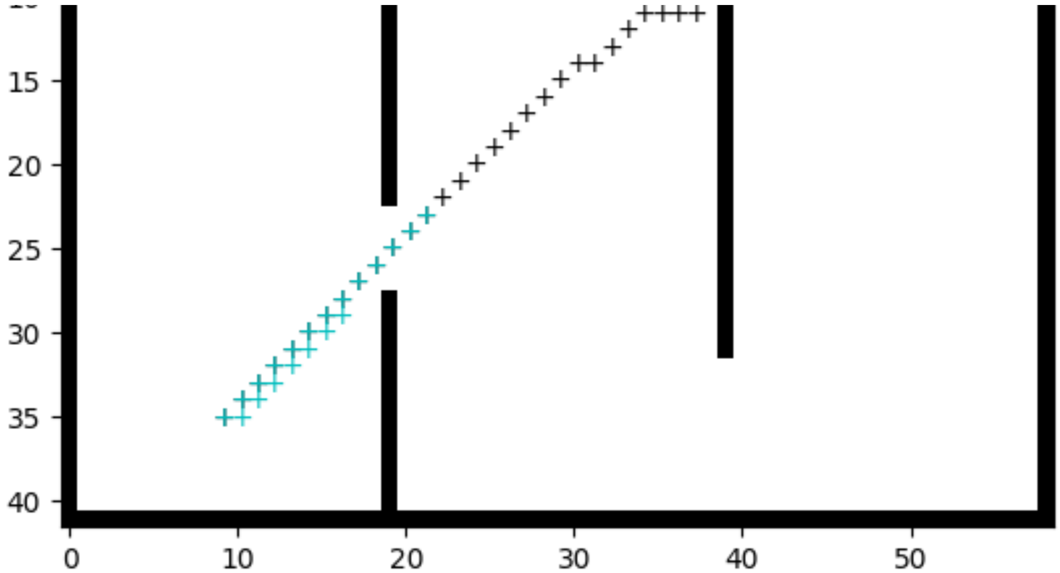
```
    if goal is not None:
        return False, [], {}, node2dist
    else:
        return True, search_order, node2parent, node2dist
```

```
import math
from functools import partial

def euclidean_heurist_dist(node, goal, scale=1):
    x_n, y_n = node
    x_g, y_g = goal
    return scale*math.sqrt((x_n-x_g)**2 + (y_n - y_g)**2)


maze = Maze8(maze_str)
debugf=open('log.txt', 'w')
success, search_path, node2parent, node2dist = astar(
    maze, partial(euclidean_heurist_dist, scale=1),
    start_pos, goal_pos, debug=True, debugf=debugf)
debugf.close()

#print(success, search_path)
assert success
anim = maze.animate(search_path)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim
```

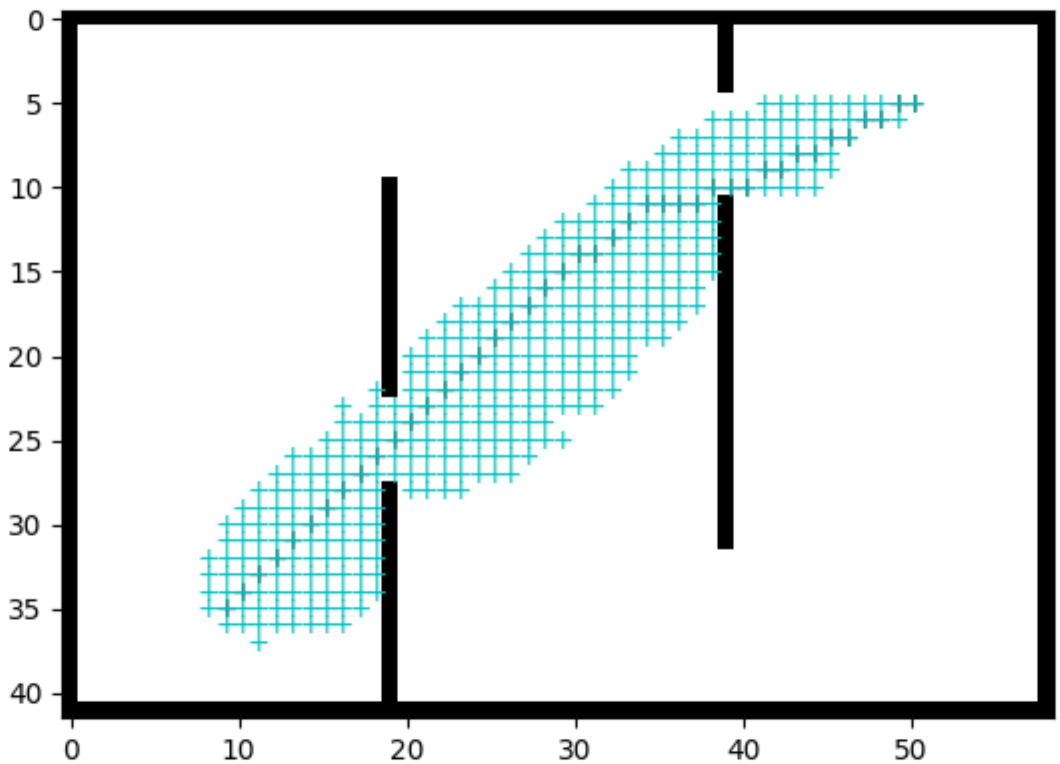
```
maze_str = \
"""
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                  +                                      +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
```

```
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
"""

start_pos, goal_pos = (35, 9), (5, 50)


maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = astar(
    maze, partial(euclidean_heurist_dist, scale=1),
    start_pos, goal_pos)
#print(success, search_path)
assert success
anim = maze.animate(search_path, batch_size=20)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim
```

```
/tmp/ipykernel_240277/955263672.py:37: UserWarning: frames=<generator object l
  anim = animation.FuncAnimation(self.fig, self._plot_path,
```



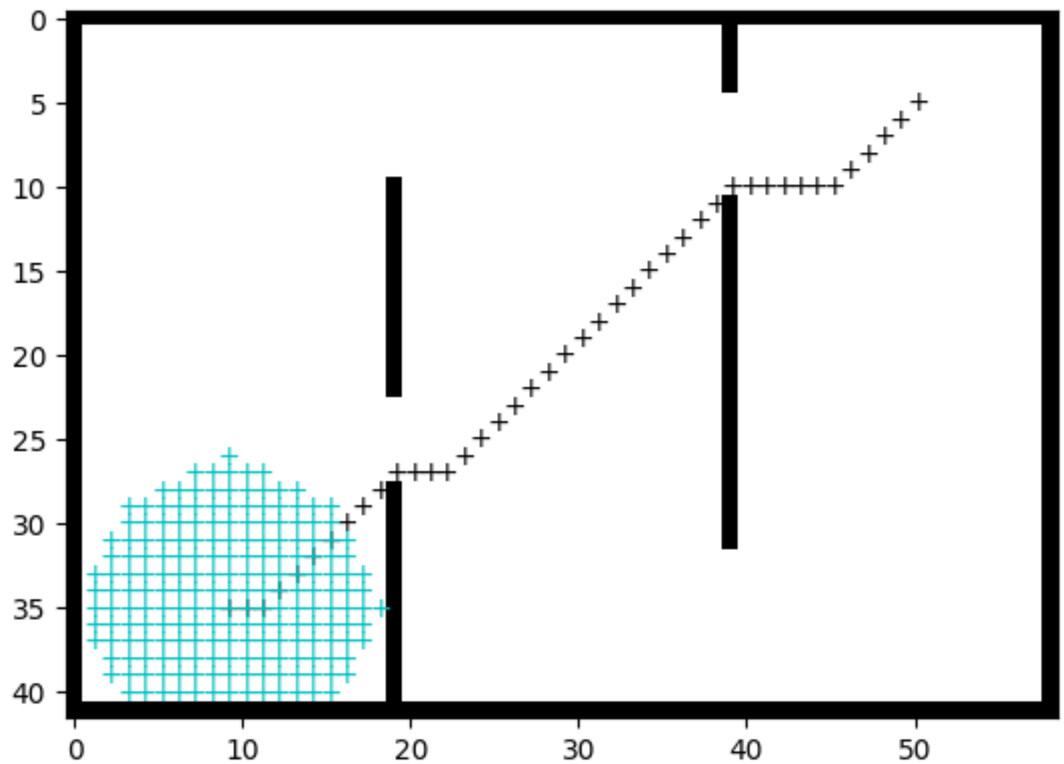
Once Loop Reflect



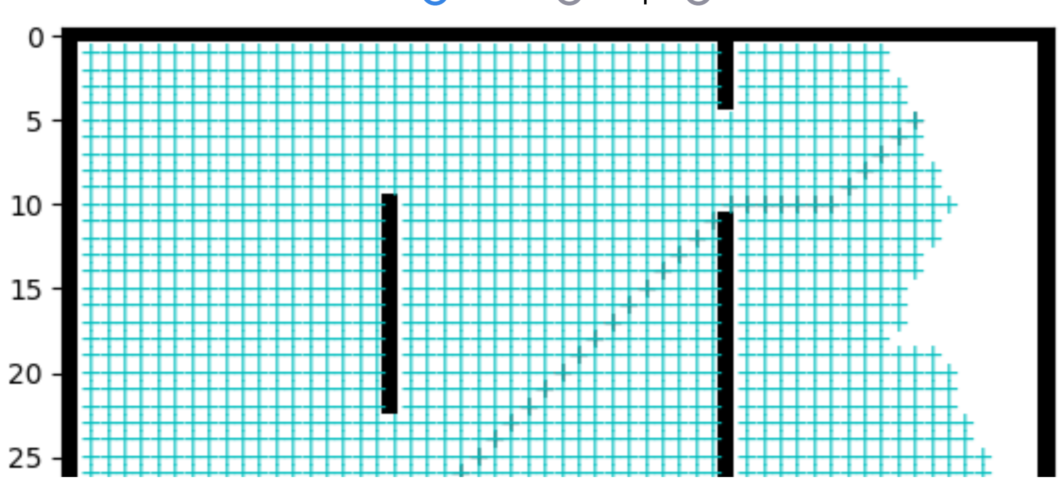

```
maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = astar(
    maze, partial(euclidean_heurist_dist, scale=0),
    start_pos, goal_pos)
#print(success, search_path)
```

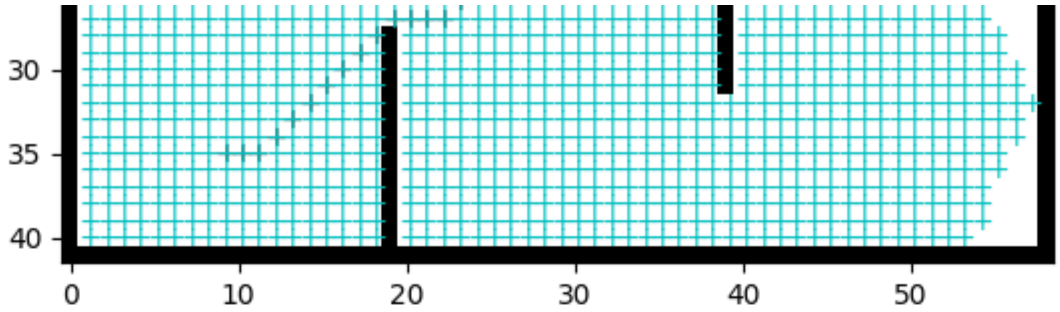
```
assert success
anim = maze.animate(search_path)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim
```

```
/tmp/ipykernel_240277/955263672.py:37: UserWarning: frames=<generator object l
  anim = animation.FuncAnimation(self.fig, self._plot_path,
```



Once Loop Reflect

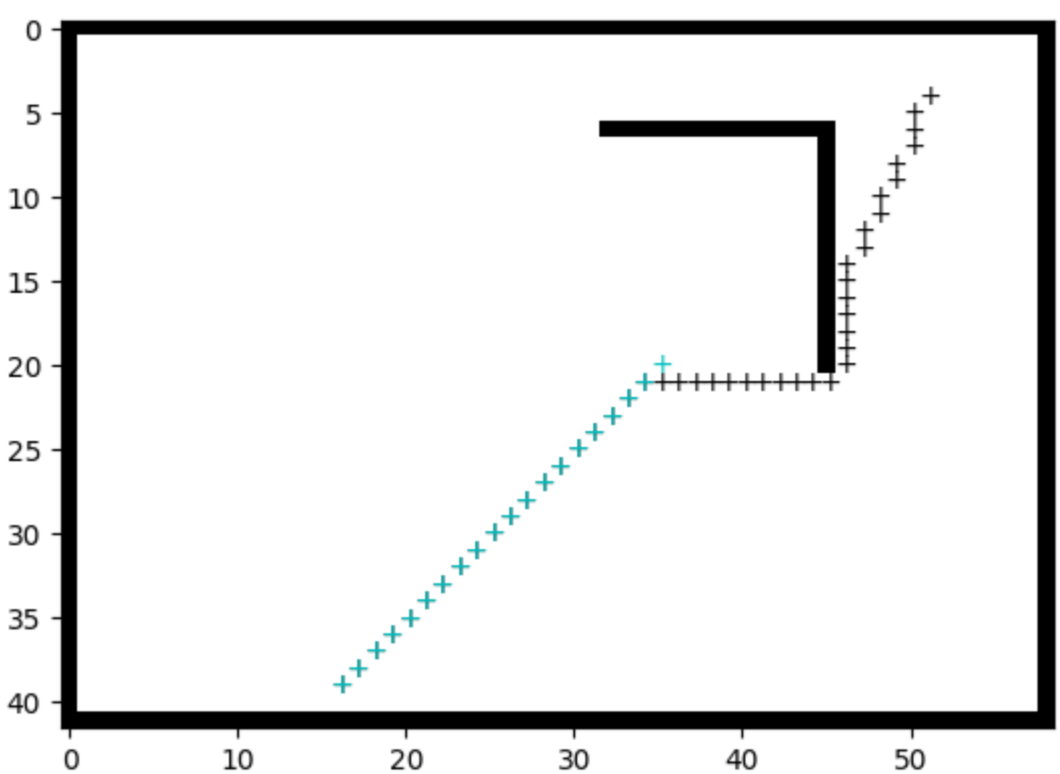




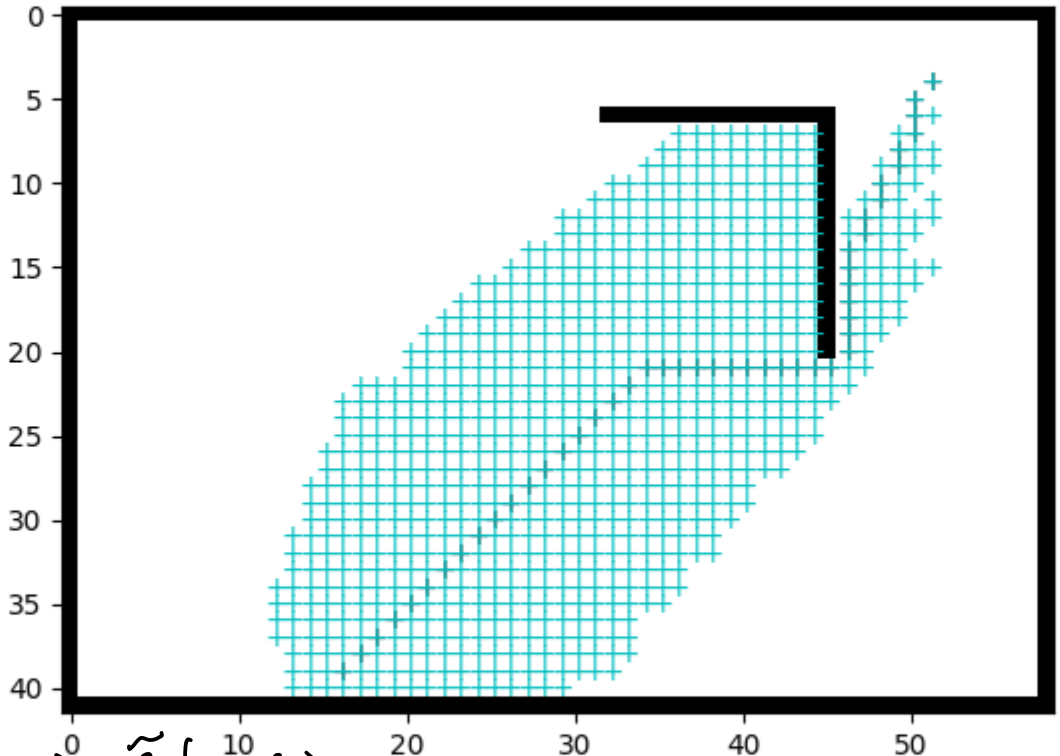

```
maze_str = \
"""
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                               ++++++++++++++            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                            +            +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+                                                         +
```

```
+                                                         +
+                                                         +
+                                                         +
+                                                         +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
"""
goal_pos = (9-5, 46+5)
start_pos = (9+30, 46-30)


maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = astar(
    maze, partial(euclidean_heurist_dist, scale=1),
    start_pos, goal_pos)
#print(success, search_path)
assert success
anim = maze.animate(search_path, batch_size=20)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim
```

```
/tmp/ipykernel_240277/955263672.py:37: UserWarning: frames=<generator object l
  anim = animation.FuncAnimation(self.fig, self._plot_path,
```



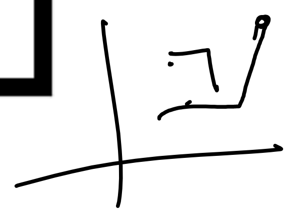
Once Loop Reflect



$c(s,n) + \tilde{c}(n, G)$

$f(n) = \underline{g(n)} + \underbrace{h(n)}$

$h(n) < c(n, G)$



Admisibility and Consistency of heuristic function

Tree search

$h(n) < C(n, G)$

1. An admissible heuristic is one that never overestimates the cost to reach the goal.

means its always optimistic

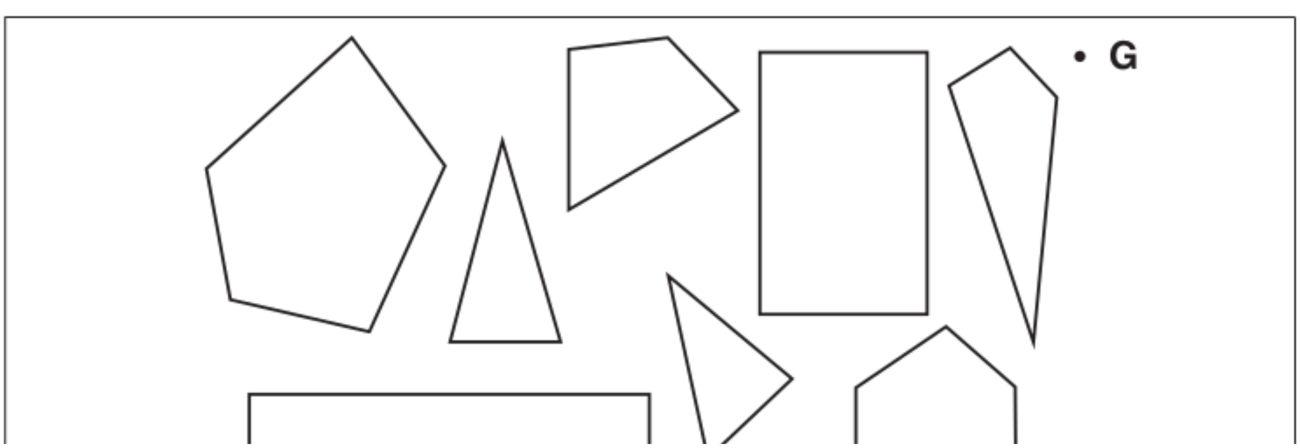
2. A heuristic h(n) is consistent if it satisfies the triangle inequality:

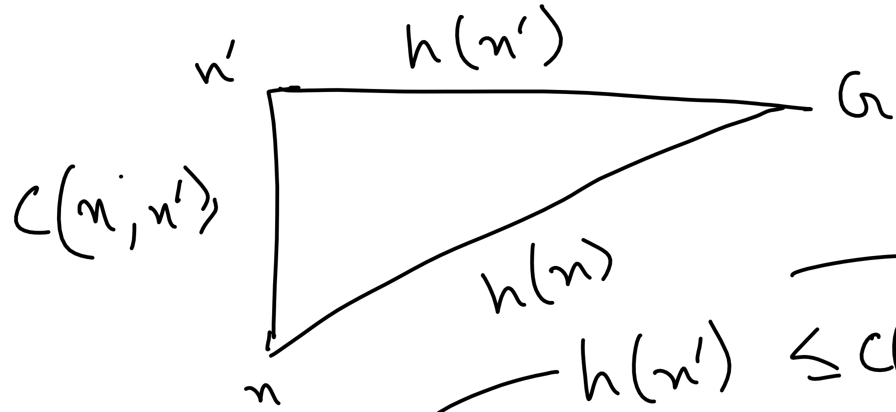
$h(\mathbf{s}_t) \leq c(\mathbf{s}_t, \mathbf{s}_{t+1}) + h(\mathbf{s}_{t+1}).$

Consistency ⇒ Admissible

Graph

Other ways of converting a maze into a graph

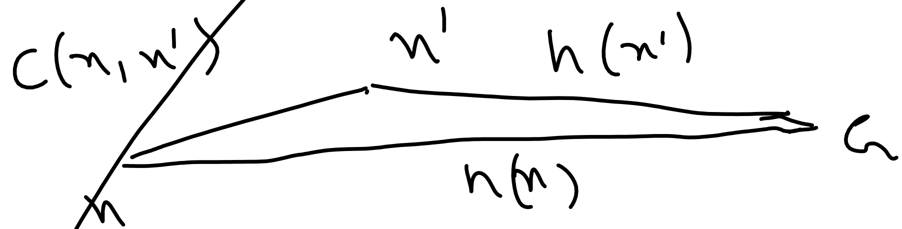




consisten heuristic h(n)

$$h(n') \le c(n,n') + h(n)$$

triangle inequality



$$h(n) \approx c(n,n') + h(n')$$

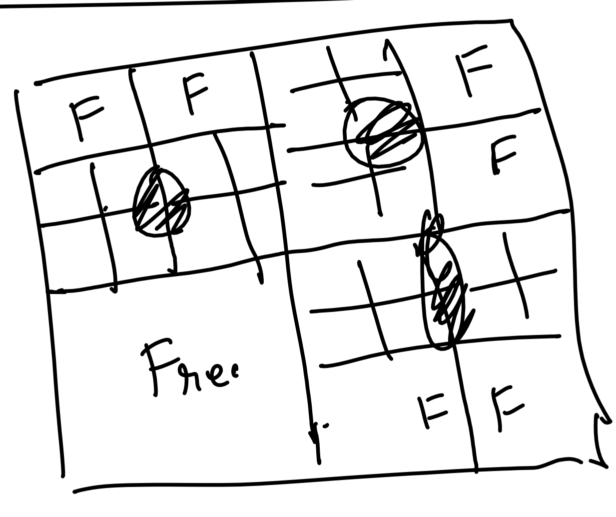
substitute $n = G$

$$h(n') \le c(n',G) + \underbrace{h(G)}_{=0}$$

$$\underbrace{h(n') \le c(n',G)}_{\text{admissible}}$$

optimistic

Quad tree



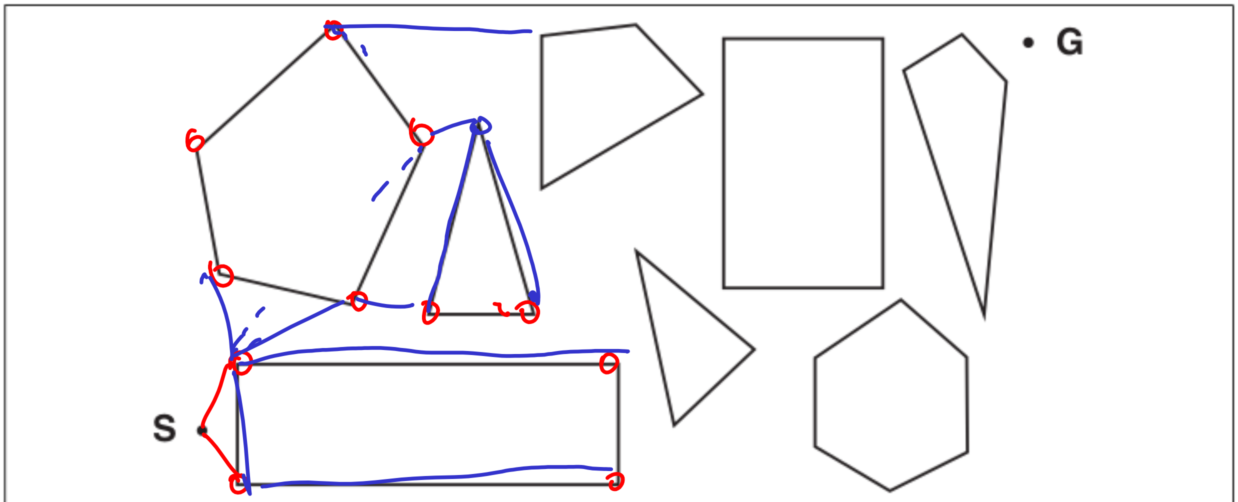


Figure 3.31 A scene with polygonal obstacles. S and G are the start and goal states.

Figure 3.31 A scene with polygonal obstacles. S and G are the start and goal states.

Rapidly exploring random trees

Uniformly random distribution