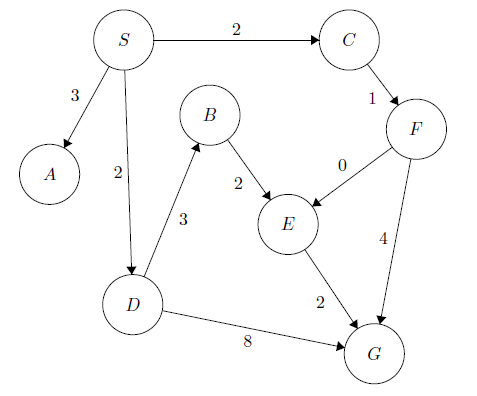
## Artificial Intelligence Lab 4

**Exercise 4.1.**

Using Breadth First Search (BFS) algorithm, write out the order in which nodes are added to the explored set, with **start state S** and **goal state G**. Break ties in alphabetical order. Additionally, what is the path returned by each algorithm? What is the total cost of each path?

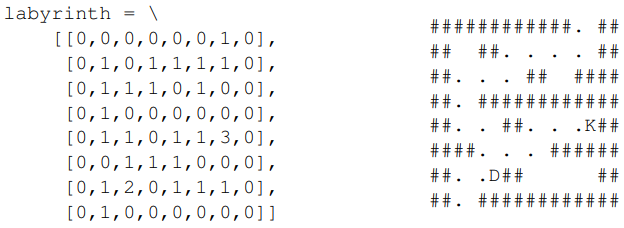


|  |
| --- |
| from collections import deque  def bfs(graph, start, goal):      # Create a queue for BFS and add the start state      queue = deque([start])        # Create a dictionary to keep track of the parent of each state      parent = {start: None}        # Create a set to keep track of the explored states      explored = set([start])        # Run BFS      while queue:          # Get the next state from the queue          curr\_state = queue.popleft()            # Check if we have reached the goal state          if curr\_state == goal:              # Construct the path from the goal state to the start state              path = []              total\_cost = 0              while curr\_state != start:                  path.append(curr\_state)                  total\_cost += graph[curr\_state][parent[curr\_state]]                  curr\_state = parent[curr\_state]              path.append(start)                # Reverse the path to get it in the correct order              path.reverse()                # Return the path and the total cost              return path, total\_cost            # Add the neighbors of the current state to the queue          for neighbor in graph[curr\_state]:              if neighbor not in explored:                  queue.append(neighbor)                  explored.add(neighbor)                  parent[neighbor] = curr\_state        # If we get here, then there is no path from the start state to the goal state      return None, None |

|  |
| --- |
| graph = {      'A': {'B': 4, 'C': 3, 'D': 6},      'B': {'A': 4, 'E': 1, 'F': 3},      'C': {'A': 3, 'G': 2},      'D': {'A': 6, 'H': 2, 'I': 7},      'E': {'B': 1, 'F': 3},      'F': {'B': 3, 'E': 3, 'G': 1},      'G': {'C': 2, 'F': 1},      'H': {'D': 2},      'I': {'D': 7}  }  start = 'A'  goal = 'G'  path, cost = bfs(graph, start, goal)  if path is None:      print("No path found!")  else:      print("Shortest path:", path)      print("Total cost:", cost) |

**Exercise 4.2.**

In Figure 1 you will find a matrix (list of lists) that represents a simple labyrinth. 0 represents a wall and 1 represents a passage. 2 represents a door that requires a key to be opened and 3 represents a supply of keys.



1. Write a function that takes a labyrinth matrix like the one shown above as argument and draws it using the characters ## for walls, two spaces for empty passages, D for doors and K for keys. This function should also accept a list of (x,y) tuples that represent positions visited by a robot in the labyrinth, you should print the character . at these points (note that you should print either a whitespace, D or K in addition to the dot).

|  |
| --- |
| def draw\_labyrinth(labyrinth, visited=[]):      for i in range(len(labyrinth)):          row = ''          for j in range(len(labyrinth[0])):              if labyrinth[i][j] == 0:                  row += '##'              elif labyrinth[i][j] == 1:                  if (i,j) in visited:                      row += '. '                  else:                      row += '  '              elif labyrinth[i][j] == 2:                  if (i,j) in visited:                      row += 'D.'                  else:                      row += 'D '              elif labyrinth[i][j] == 3:                  if (i,j) in visited:                      row += 'K.'                  else:                      row += 'K '          print(row) |

1. Write a function that takes a labyrinth matrix like the one above and the x and y coordinates for a place in the labyrinth as arguments, and returns a list of the coordinates (tuples of x and y) of all adjacent places (vertically and horizontally) that are passages (i.e. have value 1). Note that the function must handle all input coordinates in a consistent way, and not access memory outside the size of the labyrinth array under any circumstance.

|  |
| --- |
| def get\_adjacent\_passages(labyrinth, x, y):      adjacent = []      if x > 0 and labyrinth[x-1][y] == 1:          adjacent.append((x-1, y))      if x < len(labyrinth) - 1 and labyrinth[x+1][y] == 1:          adjacent.append((x+1, y))      if y > 0 and labyrinth[x][y-1] == 1:          adjacent.append((x, y-1))      if y < len(labyrinth[0]) - 1 and labyrinth[x][y+1] == 1:          adjacent.append((x, y+1))      return adjacent |

1. Write a recursive function that finds a way through the labyrinth. The entrance is at (6,0) the exit at (1,15) for the big labyrinth. The function should effectively perform a depth-first search. The base case is obviously that the goal is reached, and should return the path leading there. Therefore, you need to keep track of the path traversed so far. In the recursive case, you need to try all possible ways you can take the next step, except for those cases when you return to a state previously visited.

|  |
| --- |
| def find\_path(labyrinth, start, goal, visited=None):      if visited is None:          visited = set()      x, y = start      if (x, y) in visited:          return None      if x < 0 or y < 0 or x >= len(labyrinth) or y >= len(labyrinth[0]):          return None      if labyrinth[x][y] == 0:          return None      if (x, y) == goal:          return [goal]      visited.add((x, y))      for next\_pos in get\_adjacent\_passages(labyrinth, x, y):          path = find\_path(labyrinth, next\_pos, goal, visited)          if path is not None:              return [(x, y)] + path      return None |

|  |
| --- |
| # Example small labyrinth  small\_labyrinth = [[0, 0, 0, 0, 0],                     [0, 1, 1, 1, 0],                     [0, 1, 0, 1, 0],                     [0, 1, 1, 1, 0],                     [0, 0, 0, 0, 0]]  # Call the functions  visited = [(1, 1), (2, 1), (2, 2), (2, 3), (3, 3)]  draw\_labyrinth(small\_labyrinth, visited)  adjacent\_passages = get\_adjacent\_passages(small\_labyrinth, 2, 2)  print(adjacent\_passages)  path = find\_path(small\_labyrinth, (1, 1), (3, 3))  print(path) |

**Exercise 4.3.**

The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. Implement and solve the problem optimally using following search algorithm. Is it a good idea to check for repeated states?

1. BFS
2. DFS

|  |
| --- |
| from collections import deque  def missionaries\_and\_cannibals\_bfs():      start\_state = (3, 3, 0)  # starting state      goal\_state = (0, 0, 1)   # goal state      # Define valid moves      def valid\_moves(state):          m, c, b = state          moves = []          if b == 0:  # boat is on the starting bank              if m >= 1:                  moves.append((m-1, c, 1))              if m >= 2:                  moves.append((m-2, c, 1))              if c >= 1:                  moves.append((m, c-1, 1))              if c >= 2:                  moves.append((m, c-2, 1))              if m >= 1 and c >= 1:                  moves.append((m-1, c-1, 1))          else:      # boat is on the destination bank              if 3-m >= 1:                  moves.append((m+1, c, 0))              if 3-m >= 2:                  moves.append((m+2, c, 0))              if 3-c >= 1:                  moves.append((m, c+1, 0))              if 3-c >= 2:                  moves.append((m, c+2, 0))              if 3-m >= 1 and 3-c >= 1:                  moves.append((m+1, c+1, 0))          return moves      # Breadth-First Search algorithm      visited = set()      queue = deque([(start\_state, [])])  # Queue of (state, path) tuples      while queue:          state, path = queue.popleft()          if state == goal\_state:              return path + [state]          if state in visited:              continue          visited.add(state)          for move in valid\_moves(state):              if move not in visited:                  queue.append((move, path + [state]))      return None  # Solution not found |

|  |
| --- |
| path = missionaries\_and\_cannibals\_bfs()  if path:      print("Solution found!")      for state in path:          print(state)  else:      print("Solution not found.") |