## AI\_LAB\_3

Kola Deepti (120cs0151)

## IDDFS

def uniform\_cost\_search(goal, start):

    global graph, cost

    answer = []

    # priority queue

    queue = []

    for i in range(len(goal)):

        answer.append(10\*\*8)

    queue.append([0, [start]])

    # map to store visited node

    visited = {}

    # count

    count = 0

    # while the queue is not empty

    while len(queue) > 0:

        queue.sort()

        p = queue.pop(0)

        # get the original cost and path

        p\_cost = p[0]

        p\_path = p[1]

        # check if the last node in the path is part of the goal list

        if p\_path[-1] in goal:

            # get the position

            index = goal.index(p\_path[-1])

            # if a new goal is reached

            if answer[index] == 10\*\*8:

                count += 1

            # if the cost is less

            if answer[index] > p\_cost:

                answer[index] = p\_cost

            if count == len(goal):

                return answer, p\_path  # Return both answer and the path

        # check for the non-visited nodes which are adjacent to the present node

        if p\_path[-1] not in visited:

            for i in range(len(graph[p\_path[-1]])):

                new\_cost = p\_cost + cost[(p\_path[-1], graph[p\_path[-1]][i])]

                new\_path = p\_path + [graph[p\_path[-1]][i]]

                queue.append([new\_cost, new\_path])

        # mark as visited

        visited[p\_path[-1]] = 1

    return answer, None  # If goal is not reached, return None for the path

# main function

if \_\_name\_\_ == '\_\_main\_\_':

    graph,cost = [[] for i in range(8)],{}

    # add edge

    graph[0].append(1)

    graph[0].append(3)

    graph[3].append(1)

    graph[3].append(6)

    graph[3].append(4)

    graph[1].append(6)

    graph[4].append(2)

    graph[4].append(5)

    graph[2].append(1)

    graph[5].append(2)

    graph[5].append(6)

    graph[6].append(4)

    # add the cost

    cost[(0, 1)] = 2

    cost[(0, 3)] = 5

    cost[(1, 6)] = 1

    cost[(3, 1)] = 5

    cost[(3, 6)] = 6

    cost[(3, 4)] = 2

    cost[(2, 1)] = 4

    cost[(4, 2)] = 4

    cost[(4, 5)] = 3

    cost[(5, 2)] = 6

    cost[(5, 6)] = 3

    cost[(6, 4)] = 7

    #  # add edge

    # graph[0].append(1)

    # graph[0].append(3)

    # graph[3].append(4)

    # graph[3].append(6)

    # graph[1].append(2)

    # graph[1].append(4)

    # graph[4].append(5)

    # graph[4].append(6)

    # graph[2].append(5)

    # graph[5].append(6)

    # graph[5].append(7)

    # graph[7].append(6)

    # # add the cost

    # cost[(0, 1)] = 5

    # cost[(0, 3)] = 6

    # cost[(3, 4)] = 5

    # cost[(3, 6)] = 5

    # cost[(1, 2)] = 1

    # cost[(1, 4)] = 2

    # cost[(4, 5)] = 4

    # cost[(4, 6)] = 3

    # cost[(2, 5)] = 3

    # cost[(5, 6)] = 7

    # cost[(5, 7)] = 1

    # cost[(7, 6)] = 1

    goal = []

    # set the goal

    # there can be multiple goal states

    goal.append(6)

    # get the answer and path

    answer, path = uniform\_cost\_search(goal, 0)

    if path:

        print("Minimum cost from 0 to 6 is =", answer[0])

        print("Path:", path)

    else:

        print("Goal not reached.")

Output:



## Uniform cost search

def uniform\_cost\_search(goal, start):

    global graph, cost

    answer = []

    # priority queue

    queue = []

    for i in range(len(goal)):

        answer.append(10\*\*8)

    queue.append([0, [start]])

    # map to store visited node

    visited = {}

    # count

    count = 0

    # while the queue is not empty

    while len(queue) > 0:

        queue.sort()

        p = queue.pop(0)

        # get the original cost and path

        p\_cost = p[0]

        p\_path = p[1]

        # check if the last node in the path is part of the goal list

        if p\_path[-1] in goal:

            # get the position

            index = goal.index(p\_path[-1])

            # if a new goal is reached

            if answer[index] == 10\*\*8:

                count += 1

            # if the cost is less

            if answer[index] > p\_cost:

                answer[index] = p\_cost

            if count == len(goal):

                return answer, p\_path  # Return both answer and the path

        # check for the non-visited nodes which are adjacent to the present node

        if p\_path[-1] not in visited:

            for i in range(len(graph[p\_path[-1]])):

                new\_cost = p\_cost + cost[(p\_path[-1], graph[p\_path[-1]][i])]

                new\_path = p\_path + [graph[p\_path[-1]][i]]

                queue.append([new\_cost, new\_path])

        # mark as visited

        visited[p\_path[-1]] = 1

    return answer, None  # If goal is not reached, return None for the path

# main function

if \_\_name\_\_ == '\_\_main\_\_':

    graph,cost = [[] for i in range(8)],{}

    # add edge

    graph[0].append(1)

    graph[0].append(3)

    graph[3].append(1)

    graph[3].append(6)

    graph[3].append(4)

    graph[1].append(6)

    graph[4].append(2)

    graph[4].append(5)

    graph[2].append(1)

    graph[5].append(2)

    graph[5].append(6)

    graph[6].append(4)

    # add the cost

    cost[(0, 1)] = 2

    cost[(0, 3)] = 5

    cost[(1, 6)] = 1

    cost[(3, 1)] = 5

    cost[(3, 6)] = 6

    cost[(3, 4)] = 2

    cost[(2, 1)] = 4

    cost[(4, 2)] = 4

    cost[(4, 5)] = 3

    cost[(5, 2)] = 6

    cost[(5, 6)] = 3

    cost[(6, 4)] = 7

    #  # add edge

    # graph[0].append(1)

    # graph[0].append(3)

    # graph[3].append(4)

    # graph[3].append(6)

    # graph[1].append(2)

    # graph[1].append(4)

    # graph[4].append(5)

    # graph[4].append(6)

    # graph[2].append(5)

    # graph[5].append(6)

    # graph[5].append(7)

    # graph[7].append(6)

    # # add the cost

    # cost[(0, 1)] = 5

    # cost[(0, 3)] = 6

    # cost[(3, 4)] = 5

    # cost[(3, 6)] = 5

    # cost[(1, 2)] = 1

    # cost[(1, 4)] = 2

    # cost[(4, 5)] = 4

    # cost[(4, 6)] = 3

    # cost[(2, 5)] = 3

    # cost[(5, 6)] = 7

    # cost[(5, 7)] = 1

    # cost[(7, 6)] = 1

    goal = []

    # set the goal

    # there can be multiple goal states

    goal.append(6)

    # get the answer and path

    answer, path = uniform\_cost\_search(goal, 0)

    if path:

        print("Minimum cost from 0 to 6 is =", answer[0])

        print("Path:", path)

    else:

        print("Goal not reached.")

Output  
