### Screen Shots

**program\_a.py**

A black screen with white text

AI-generated content may be incorrect.

**program\_b.py**

A black background with white text

AI-generated content may be incorrect.

### Python Code

**program\_a.py**

import heapq

import timeit

def dijkstra(graph, start, end):

    queue = [(0, start)]

    distances = {node: float('infinity') for node in graph}

    distances[start] = 0

    previous\_nodes = {node: None for node in graph}

    while queue:

        current\_distance, current\_node = heapq.heappop(queue)

        if current\_distance > distances[current\_node]:

            continue

        for neighbor, weight in graph[current\_node]:

            distance = current\_distance + weight

            if distance < distances[neighbor]:

                distances[neighbor] = distance

                previous\_nodes[neighbor] = current\_node

                heapq.heappush(queue, (distance, neighbor))

    return distances, previous\_nodes

def find\_path(previous\_nodes, start, end):

    path = []

    current\_node = end

    while current\_node is not None:

        path.append(current\_node)

        current\_node = previous\_nodes[current\_node]

    path.reverse()

    return path if path[0] == start else None

def main():

    graph = {

        'A': [('B', 22), ('C', 9), ('D', 12)],

        'B': [('A', 22), ('C', 35), ('F', 36), ('H', 34)],

        'C': [('A', 9), ('B', 35), ('D', 4), ('E', 65), ('F', 42)],

        'D': [('A', 12), ('C', 4), ('E', 33), ('I', 30)],

        'E': [('C', 65), ('D', 33), ('F', 18), ('G', 23)],

        'F': [('B', 36), ('C', 42), ('E', 18), ('G', 39), ('H', 24)],

        'G': [('E', 23), ('F', 39), ('H', 25), ('I', 21)],

        'H': [('B', 34), ('F', 24), ('G', 25), ('I', 19)],

        'I': [('D', 30), ('G', 21), ('H', 19)]

    }

    distances, previous\_nodes = dijkstra(graph, 'A', 'I')

    shortest\_path = find\_path(previous\_nodes, 'A', 'I')

    print(f"Shortest path from A to I using Dijkstra's Algorithm: {shortest\_path}")

    print(f"Distance: {distances['I']}")

if \_\_name\_\_ == "\_\_main\_\_":

    time\_taken = timeit.timeit(main, number=1)

    print(f"Time taken: {time\_taken:.10f} seconds")

**program\_b.py**

import heapq

import timeit

def heuristic(node, end):

    # Simple heuristic: straight-line distance (Manhattan distance for simplicity)

    return abs(ord(node) - ord(end))

def a\_star(graph, start, end):

    queue = [(0, start)]

    g\_costs = {node: float('infinity') for node in graph}

    g\_costs[start] = 0

    f\_costs = {node: float('infinity') for node in graph}

    f\_costs[start] = heuristic(start, end)

    previous\_nodes = {node: None for node in graph}

    while queue:

        current\_f\_cost, current\_node = heapq.heappop(queue)

        if current\_node == end:

            return g\_costs, previous\_nodes

        for neighbor, weight in graph[current\_node]:

            tentative\_g\_cost = g\_costs[current\_node] + weight

            if tentative\_g\_cost < g\_costs[neighbor]:

                g\_costs[neighbor] = tentative\_g\_cost

                f\_costs[neighbor] = tentative\_g\_cost + heuristic(neighbor, end)

                previous\_nodes[neighbor] = current\_node

                heapq.heappush(queue, (f\_costs[neighbor], neighbor))

    return g\_costs, previous\_nodes

def find\_path(previous\_nodes, start, end):

    path = []

    current\_node = end

    while current\_node is not None:

        path.append(current\_node)

        current\_node = previous\_nodes[current\_node]

    path.reverse()

    return path if path[0] == start else None

def main():

    graph = {

        'A': [('B', 22), ('C', 9), ('D', 12)],

        'B': [('A', 22), ('C', 35), ('F', 36), ('H', 34)],

        'C': [('A', 9), ('B', 35), ('D', 4), ('E', 65), ('F', 42)],

        'D': [('A', 12), ('C', 4), ('E', 33), ('I', 30)],

        'E': [('C', 65), ('D', 33), ('F', 18), ('G', 23)],

        'F': [('B', 36), ('C', 42), ('E', 18), ('G', 39), ('H', 24)],

        'G': [('E', 23), ('F', 39), ('H', 25), ('I', 21)],

        'H': [('B', 34), ('F', 24), ('G', 25), ('I', 19)],

        'I': [('D', 30), ('G', 21), ('H', 19)]

    }

    g\_costs, previous\_nodes = a\_star(graph, 'A', 'I')

    shortest\_path = find\_path(previous\_nodes, 'A', 'I')

    print(f"Shortest path from A to I using A\* Algorithm: {shortest\_path}")

    print(f"Distance: {g\_costs['I']}")

if \_\_name\_\_ == "\_\_main\_\_":

    time\_taken = timeit.timeit(main, number=1)

    print(f"Time taken: {time\_taken:.10f} seconds")