Prompt1:  
Provide a python implementation of a breadth-first search algorithm that takes in a nx.graph() called “graph”, a starting city called “start”, and a destination city called “goal”. Then finds the path from the start to goal within the provided graph.

Output:

Here is a Python implementation of the **Breadth-First Search (BFS)** algorithm to find the shortest path between a start city and a goal city in a networkx graph:

import networkx as nx

from collections import deque

def bfs\_shortest\_path(graph: nx.Graph, start: str, goal: str):

"""

Performs BFS on the given graph to find the shortest path from start to goal.

:param graph: A networkx Graph

:param start: Starting city (node)

:param goal: Destination city (node)

:return: List representing the shortest path from start to goal, or None if no path exists.

"""

if start not in graph or goal not in graph:

return None # Return None if either node is not in the graph

queue = deque([[start]]) # Initialize queue with the start node as a path

visited = set() # Track visited nodes

while queue:

path = queue.popleft() # Get the first path from the queue

node = path[-1] # Get the last node in the path

if node == goal:

return path # Return path if goal is reached

if node not in visited:

visited.add(node)

for neighbor in graph.neighbors(node):

new\_path = list(path) # Copy current path

new\_path.append(neighbor) # Append neighbor to new path

queue.append(new\_path) # Enqueue new path

return None # Return None if no path exists

# Example Usage

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

G = nx.Graph()

G.add\_edges\_from([

("A", "B"), ("A", "C"), ("B", "D"), ("C", "D"), ("C", "E"), ("D", "E"), ("E", "F")

])

start\_city = "A"

goal\_city = "F"

path = bfs\_shortest\_path(G, start\_city, goal\_city)

print("Shortest path:", path)

Prompt 2:  
Provide a python implementation of a A\* search algorithm that takes in a nx.graph() called “graph”, a starting city called “start”, and a destination city called “goal”, and distance from goal. Then finds the path from the start to goal within the provided graph.  
  
import networkx as nx

import heapq

Output:

Here is a *Python implementation of the A (A-Star) search algorithm*\* using networkx to find the optimal path from a start city to a goal city. The algorithm utilizes a heuristic function (straight-line distance) to guide the search.

def a\_star\_search(graph: nx.Graph, start: str, goal: str, heuristic: dict):

"""

Performs A\* search on the given graph to find the shortest path from start to goal.

:param graph: A networkx Graph with weighted edges

:param start: Starting city (node)

:param goal: Destination city (node)

:param heuristic: A dictionary containing estimated distance from each node to the goal

:return: List representing the shortest path from start to goal, or None if no path exists.

"""

if start not in graph or goal not in graph:

return None # Return None if either node is not in the graph

# Priority queue (min-heap) for A\* search, stores (cost, current\_node, path)

priority\_queue = [(0 + heuristic[start], 0, start, [start])]

visited = set()

while priority\_queue:

\_, cost, node, path = heapq.heappop(priority\_queue) # Get node with lowest estimated cost

if node == goal:

return path # Return path if goal is reached

if node not in visited:

visited.add(node)

for neighbor in graph.neighbors(node):

edge\_weight = graph[node][neighbor].get('weight', 1) # Default weight is 1 if not specified

new\_cost = cost + edge\_weight # Actual cost from start to this neighbor

# Calculate A\* priority: f(n) = g(n) + h(n)

priority = new\_cost + heuristic.get(neighbor, float('inf'))

heapq.heappush(priority\_queue, (priority, new\_cost, neighbor, path + [neighbor]))

return None # Return None if no path exists

# Example Usage

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

G = nx.Graph()

G.add\_weighted\_edges\_from([

("A", "B", 4), ("A", "C", 2), ("B", "D", 5), ("C", "D", 8),

("C", "E", 10), ("D", "E", 2), ("E", "F", 3)

])

# Heuristic function (straight-line estimated distance to goal)

heuristic = {

"A": 10, "B": 8, "C": 5, "D": 3, "E": 2, "F": 0 # Lower values closer to goal

}

start\_city = "A"

goal\_city = "F"

path = a\_star\_search(G, start\_city, goal\_city, heuristic)

print("Optimal path found:", path)