```
import heapq

class Node:
    def __init__(self, name, parent=None, g=0, h=0):
        self.name = name
        self.parent = parent
        self.g = g  # cost from start to node
        self.h = h  # heuristic (estimated cost to goal)
        self.f = g + h  # total cost

    def __lt__(self, other):  # for priority queue
        return self.f < other.f

def astar_search(graph, heuristics, start, goal):
    open_list = []
    closed_set = set()

    start_node = Node(start, None, 0, heuristics[start])
    heapq.heappush(open_list, start_node)

    while open_list:
        current = heapq.heappop(open_list)

        if current.name == goal:
            path = []
            while current:
                path.append(current.name)
                current = current.parent
            return path[::-1]  # reverse path

        closed_set.add(current.name)

        for neighbor, cost in graph[current.name].items():
            if neighbor in closed_set:
                continue

            g = current.g + cost
            h = heuristics[neighbor]
            neighbor_node = Node(neighbor, current, g, h)

            # check if a better path exists
            if any(open_node.name == neighbor and open_node.f <=
neighbor_node.f for open_node in open_list):
                continue

            heapq.heappush(open_list, neighbor_node)
```

```
    return None

# Example Graph (with distances)
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'D': 3, 'E': 1},
    'C': {'A': 3, 'F': 5},
    'D': {'B': 3, 'G': 2},
    'E': {'B': 1, 'G': 1},
    'F': {'C': 5, 'G': 2},
    'G': {'D': 2, 'E': 1, 'F': 2}
}

# Heuristic values (straight-line estimates to goal 'G')
heuristics = {
    'A': 7,
    'B': 6,
    'C': 5,
    'D': 4,
    'E': 2,
    'F': 3,
    'G': 0
}

# Run A* Search
start, goal = 'A', 'G'
path = astar_search(graph, heuristics, start, goal)
print(f"Shortest path from {start} to {goal}: {path}")
```