```
import math
from queue import PriorityQueue

# Function to calculate heuristic (Euclidean distance)
def heuristic(node, goal):
    x1, y1 = coords[node]
    x2, y2 = coords[goal]
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

# State class to represent each node state
class State:
    def __init__(self, nid, parent, g, f):
        self.nid = nid      # Node ID
        self.parent = parent  # Parent state
        self.g = g          # Actual cost to reach this state
        self.f = f          # Total cost (g + heuristic)

    def __lt__(self, other):
        return self.f < other.f  # Compare based on f value

    def __eq__(self, other):
```

```
        return self.nid == other.nid and self.g == other.g and self.f
== other.f

# A* Search Implementation
def a_star_search(start, goal):
    pq = PriorityQueue()
    start_state = State(start, None, 0, heuristic(start, goal))
    pq.put(start_state)

    visited = set()

    while not pq.empty():
        curr_state = pq.get()  # Get state with smallest f

        # If goal is reached, reconstruct path
        if curr_state.nid == goal:
            path = [ ]
            total_cost = curr_state.g
            while curr_state:
                path.append(curr_state.nid)
                curr_state = curr_state.parent
```

```
            return path[::-1], total_cost  # Path is reversed

        # Mark current node as visited
        if curr_state.nid in visited:
            continue
        visited.add(curr_state.nid)

        # Explore neighbors
        for neighbor, cost in adjlist[curr_state.nid]:
            if neighbor in visited:
                continue

            g = curr_state.g + cost
            h = heuristic(neighbor, goal)
            f = g + h

            new_state = State(neighbor, curr_state, g, f)
            pq.put(new_state)

    return None, float("inf")  # If no path is found
```

```python
# Parse the input graph
coords = {}  # Node ID -> Coordinates
adjlist = {}  # Node ID -> List of adjacent nodes with costs
with open('input.txt', 'r') as f:
    V = int(f.readline())  # Number of vertices
    for i in range(V):
        nid, x, y = f.readline().split()
        coords[nid] = (int(x), int(y))
        adjlist[nid] = [ ]  # Initialize adjacency list for each node

    E = int(f.readline())  # Number of edges
    for i in range(E):
        n1, n2, c = f.readline().split()
        adjlist[n1].append((n2, int(c)))

    startnid = f.readline().strip()  # Start node
    goalnid = f.readline().strip()   # Goal node

# Run A* search
solution_path, solution_cost = a_star_search(startnid, goalnid)
```

```
# Output the result
if solution_path:
    print("Solution path:", " -> ".join(solution_path))
    print("Solution cost:", solution_cost)
else:
    print("No path found")
```

```
6
S 6 0
A 6 0
B 1 0
C 2 0
D 1 0
G 0 0
9
S A 1
S C 2
S D 4
A B 2
B A 2
B G 1
C S 2
C G 4
D G 4
S
G
```