```
import heapq

# Define Contact class to store contact information
class Contact:
    def __init__(self, contact_id, start, end, sender, receiver, owlt):
        self.contact_id = contact_id
        self.start = start
        self.end = end
        self.sender = sender
        self.receiver = receiver
        self.owlt = owlt

try:
    with open("C:/Users/suman/Desktop/ContactList.txt", 'r') as file:
        contacts = []
        for line in file:
            fields = line.strip().split()
            contact = Contact(int(fields[0]), float(fields[1]), float(fields[2]),
int(fields[3]), int(fields[4]), float(fields[5]))
            contacts.append(contact)
except IOError:
    print("Error reading file. Please check the file path and try again.")
    exit()

# Define PriorityQueue class using heapq module
class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return not self.elements

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]

# Define Dijkstra's search algorithm
def dijkstra_search(graph, start, end):
    # Initialize variables
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {start: None}
    cost_so_far = {start: (0, None)}

    # Iterate until frontier is empty
    while not frontier.empty():
        current = frontier.get()
        if current == end:
            break
        for next_contact in graph[current]:
            # Skip if contact has already ended
            if next_contact.end <= cost_so_far[current][0]:
                continue
            # Calculate new cost of reaching the next contact
            new_cost = cost_so_far[current][0] + next_contact.owlt if
next_contact.start <= cost_so_far[current][0] else next_contact.start +
next_contact.owlt
```

```
            # Update cost and priority of reaching the next contact
            if next_contact.receiver not in cost_so_far or new_cost <
cost_so_far[next_contact.receiver][0]:
                cost_so_far[next_contact.receiver] = (new_cost,
next_contact.contact_id)
                priority = new_cost
                frontier.put(next_contact.receiver, priority)
                came_from[next_contact.receiver] = current

    # Construct and print optimal path
    current, path = end, [(cost_so_far[end][1], end)]
    while current != start:
        current = came_from[current]
        path.append((cost_so_far[current][1], current))
    path.reverse()

    print("The following is the Optimal Path:")
    for contact_id, receiver in path:
        print(f"Contact ID: {contact_id}\nNodes: {receiver}", end="")
        if receiver != end:
            print(" -> ", end="")
    print(f"\nBest Arrival Time: {cost_so_far[end][0]}")

# Create graph dictionary from contacts
graph = {contact.sender: [] for contact in contacts}
for contact in contacts:
    graph[contact.sender].append(contact)

# Run Dijkstra's search algorithm on graph
dijkstra_search(graph, 1, 12)
```