Rubric

Checking if neighbours exists for a vertex	**2**
Deadend condition handled	**2**
Finding max/min edge	**5**
Checking if enough points available for next jump	**2**
Updating variables for next iteration	**2**
Returning result	**2**
Total	**15**

Set A

For Adjacency List

```
def traverse_max_edge (adj_list, s, p):
    current_vertex = s
    points = p

    while True:
        neighbors = adj_list[current_vertex]
        if not neighbors:
            print(f"Reached a dead end")
            break

        max_edge = find_max_edge(neighbors)
        destination, weight = max_edge

        if weight > points:
            break

        points -= weight
        current_vertex = destination

    return current_vertex
```

```
def find_max_edge(neighbors):
    max_edge = None
    max_weight = float('-inf')

    for edge in neighbors:
        if edge[1] > max_weight:
            max_edge = edge
            max_weight = edge[1]

    return max_edge
```

For Adjacency Matrix

```
def traverse_max_edge (adj_matrix, s, p):
    current_vertex = s
    points = p
    vertices = len(adj_matrix)

    while True:
        neighbors = []
        for v in range(vertices):
            if adj_matrix[current_vertex][v] > 0:
                neighbors += [(v, adj_matrix[current_vertex][v])]
        if not neighbors:
            print(f"Reached a dead end")
            break

        max_edge = find_max_edge(neighbors)
        destination, weight = max_edge

        if weight > points:
            break

        points -= weight
        current_vertex = destination

    return current_vertex
```

```
def find_max_edge(neighbors):
    max_edge = None
    max_weight = float('-inf')

    for destination, weight in neighbors:
        if weight > max_weight:
            max_edge = (destination, weight)
            max_weight = weight

    return max_edge
```

Set B

For Adjacency List

```
def traverse_min_edge(adj_list, s, p):
    current_vertex = s
    points = p

    while True:
        neighbors = adj_list[current_vertex]
        if not neighbors:
            print(f"Reached a dead end")
            break

        min_edge = find_min_edge(neighbors)
        destination, weight = min_edge

        if weight > points:
            break

        points -= weight
        current_vertex = destination

    return current_vertex
```

```
def find_min_edge(neighbors):
    min_edge = None
    min_weight = float('inf')

    for edge in neighbors:
        if edge[1] < min_weight:
            min_edge = edge
            min_weight = edge[1]

    return min_edge
```

For Adjacency Matrix

```
def traverse_min_edge(adj_matrix, s, p):
    current_vertex = s
    points = p
    vertices = len(adj_matrix)

    while True:
        neighbors = []
        for v in range(vertices):
            if adj_matrix[current_vertex][v] > 0:
                neighbors += [(v, adj_matrix[current_vertex][v])]
        if not neighbors:
            print(f"Reached a dead end")
            break

        min_edge = find_min_edge(neighbors)
        destination, weight = min_edge

        if weight > points:
            break

        points -= weight
        current_vertex = destination

    return current_vertex
```

```
def find_min_edge(neighbors):
    min_edge = None
    min_weight = float('inf')

    for destination, weight in neighbors:
        if weight < min_weight:
            min_edge = (destination, weight)
            min_weight = weight

    return min_edge
```