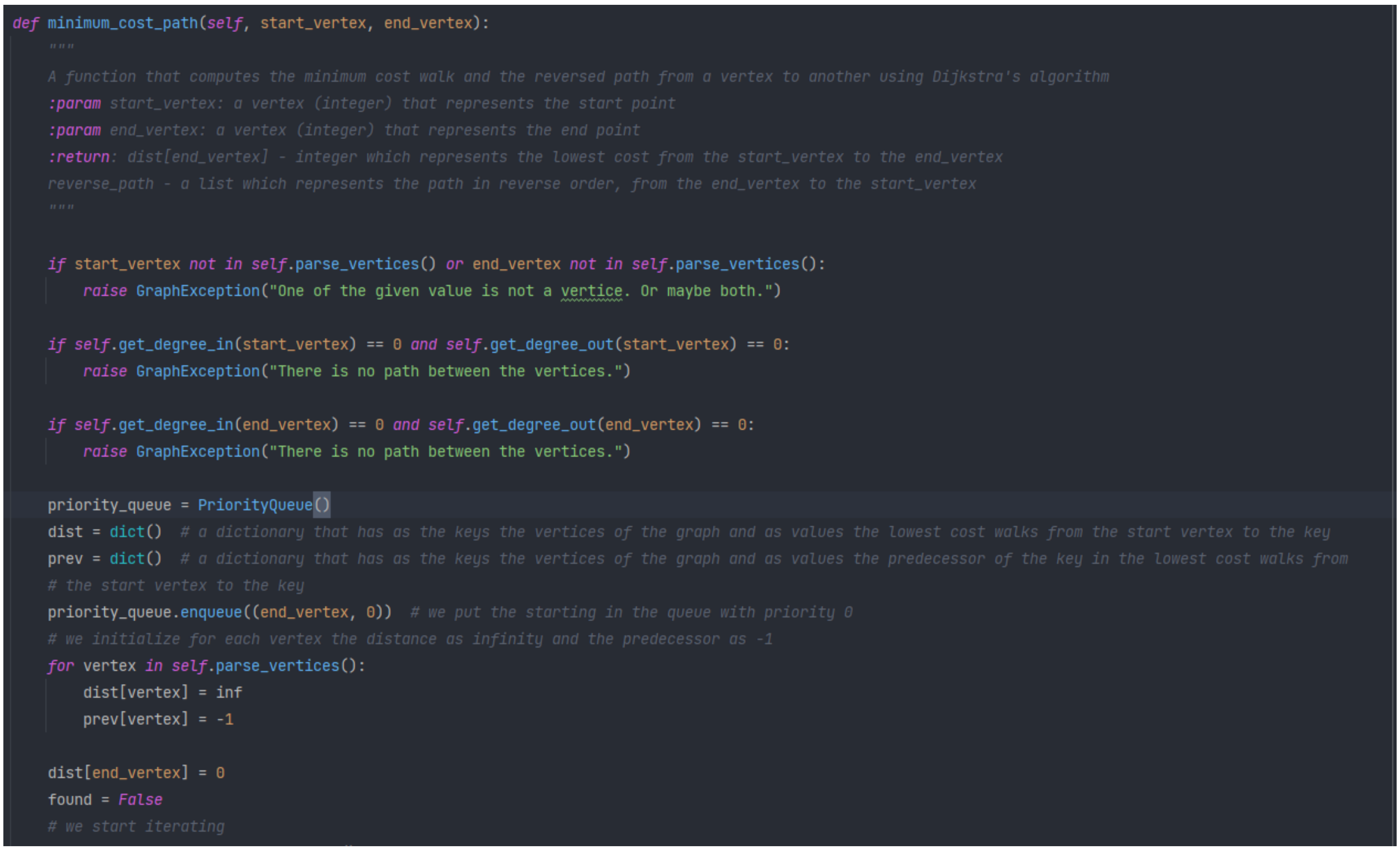
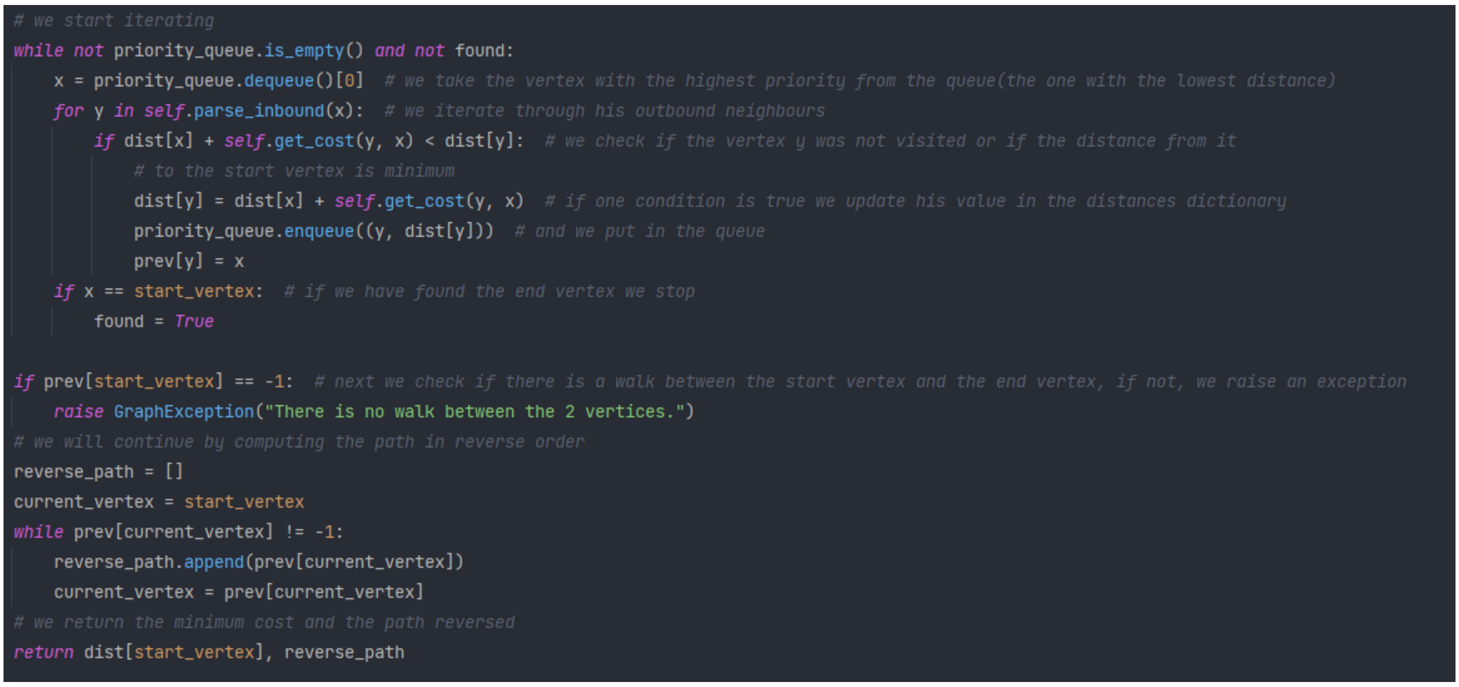
The Source Code of the Function

(with # comments)



```python
def minimum_cost_path(self, start_vertex, end_vertex):
    """
    A function that computes the minimum cost walk and the reversed path from a vertex to another using Dijkstra's algorithm
    :param start_vertex: a vertex (integer) that represents the start point
    :param end_vertex: a vertex (integer) that represents the end point
    :return: dist[end_vertex] - integer which represents the lowest cost from the start_vertex to the end_vertex
    reverse_path - a list which represents the path in reverse order, from the end_vertex to the start_vertex
    """

    if start_vertex not in self.parse_vertices() or end_vertex not in self.parse_vertices():
        raise GraphException("One of the given value is not a vertice. Or maybe both.")

    if self.get_degree_in(start_vertex) == 0 and self.get_degree_out(start_vertex) == 0:
        raise GraphException("There is no path between the vertices.")

    if self.get_degree_in(end_vertex) == 0 and self.get_degree_out(end_vertex) == 0:
        raise GraphException("There is no path between the vertices.")

    priority_queue = PriorityQueue()
    dist = dict()  # a dictionary that has as the keys the vertices of the graph and as values the lowest cost walks from the start vertex to the key
    prev = dict()  # a dictionary that has as the keys the vertices of the graph and as values the predecessor of the key in the lowest cost walks from
    # the start vertex to the key
    priority_queue.enqueue((end_vertex, 0))  # we put the starting in the queue with priority 0
    # we initialize for each vertex the distance as infinity and the predecessor as -1
    for vertex in self.parse_vertices():
        dist[vertex] = inf
        prev[vertex] = -1

    dist[end_vertex] = 0
    found = False
    # we start iterating
```



```python
# we start iterating
while not priority_queue.is_empty() and not found:
    x = priority_queue.dequeue()[0]  # we take the vertex with the highest priority from the queue(the one with the lowest distance)
    for y in self.parse_inbound(x):  # we iterate through his outbound neighbours
        if dist[x] + self.get_cost(y, x) < dist[y]:  # we check if the vertex y was not visited or if the distance from it
            # to the start vertex is minimum
            dist[y] = dist[x] + self.get_cost(y, x)  # if one condition is true we update his value in the distances dictionary
            priority_queue.enqueue((y, dist[y]))  # and we put in the queue
            prev[y] = x
    if x == start_vertex:  # if we have found the end vertex we stop
        found = True

if prev[start_vertex] == -1:  # next we check if there is a walk between the start vertex and the end vertex, if not, we raise an exception
    raise GraphException("There is no walk between the 2 vertices.")
# we will continue by computing the path in reverse order
reverse_path = []
current_vertex = start_vertex
while prev[current_vertex] != -1:
    reverse_path.append(prev[current_vertex])
    current_vertex = prev[current_vertex]
# we return the minimum cost and the path reversed
return dist[start_vertex], reverse_path
```