1. `__initialize_matrices()`
    - Description: Initializes the matrices used in the matrix multiplication algorithm for the lowest cost walk.
    - Parameters: None
    - Returns: None
    - Usage: This method is called internally within the class to initialize the '__prev' and '__w' matrices based on the number of vertices in the graph. The '__prev' matrix is used to store the previous vertex in the shortest path, while the '__w' matrix represents the weight of the edges between vertices.
2. `__extend()`
    - Description: Extends the shortest paths using matrix multiplication.
    - Parameters: None
    - Returns: The updated '__d' matrix representing the shortest distances between vertices.
    - Usage: This method is called internally within the class to extend the shortest paths using the matrix multiplication algorithm. It iterates over the vertices and calculates the shortest distance between them using intermediate vertices. The '__d' matrix is updated with the shortest distances, and the '__prev' matrix is updated with the previous vertex in the shortest path.
3. `get_path(i, j)`
    - Description: Retrieves the path between two vertices in the graph.
    - Parameters:
        - i: The source vertex index.
        - j: The destination vertex index.
    - Returns: A list representing the path from vertex i to vertex j.
    - Usage: This method is used to retrieve the path between two vertices in the graph. It recursively calls itself to find the intermediate vertices in the path and concatenates the resulting paths to form the complete path from vertex i to vertex j.
4. `compute_costs_of_paths()`
    - Description: Computes the costs of paths between all pairs of vertices in the graph.
    - Parameters: None
    - Returns: None
    - Usage: This method initializes the matrices and then computes the costs of paths between all pairs of vertices in the graph. It uses the '__initialize_matrices()' method to initialize the necessary matrices and the '__extend()' method to compute the shortest distances. It also checks for negative cycles in the graph by verifying if any diagonal element in the '__d' matrix is negative.
5. `get_path_cost(i, j)`

- Description: Retrieves the cost of the shortest path between two vertices in the graph.
- Parameters:
    - i: The source vertex index.
    - j: The destination vertex index.
- Returns: The cost of the shortest path between vertex i and vertex j.
- Usage: This method is used to retrieve the cost of the shortest path between two vertices in the graph. It checks if the given vertices are valid, and then returns the corresponding cost from the '__d' matrix. If the vertices are not present in the graph, a ValueError is raised.

## Bonus:

**`min_cost_walks(start, end)`**

Calculates the number of distinct walks of minimum cost between the given start and end vertices in the graph.

- Parameters:
    - start: The starting vertex of the walk.
    - end: The ending vertex of the walk.
- Returns:
    - num_walks: The number of distinct walks of minimum cost between the start and end vertices.
- Description:
    - This method uses a modified breadth-first search (BFS) algorithm to calculate the number of distinct walks of minimum cost between the specified start and end vertices in the graph. It maintains two lists: `num_walks` to keep track of the number of walks for each vertex, and `min_cost` to store the minimum cost of reaching each vertex from the start vertex. The BFS traversal is performed starting from the start vertex, and for each visited neighbor vertex, the method checks if the cost of reaching that neighbor vertex can be minimized. If the cost can be minimized, the number of walks to the neighbor vertex is updated, and the neighbor vertex is added to the traversal queue. If the cost is equal to the current minimum cost, the number of walks is incremented to account for multiple paths with the same minimum cost. Finally, the method returns the number of distinct walks of minimum cost between the start and end vertices.