**To analyze the time complexity of the Visibility Graph Algorithm:**

**1. Generating Visibility Nodes:**

- The algorithm iterates over the obstacles to generate visibility nodes.

- For each obstacle, it calculates the visibility nodes based on the obstacle's vertices.

- The time complexity of this step depends on the number of obstacles and the number of vertices in each obstacle.

- Let O(n) be the number of obstacles and m be the average number of vertices per obstacle.

- The time complexity for generating visibility nodes is O(n \* m^2), as it involves iterating over each pair of vertices to check visibility.

**2. Generating Road Map Information:**

- For each visibility node, the algorithm checks visibility to other nodes and stores valid connections.

- The time complexity of this step depends on the number of visibility nodes and the number of edges in the resulting graph.

- Let V be the number of visibility nodes and E be the number of edges in the graph.

- The time complexity for generating road map information depends on the implementation but is typically O(V^2) for a dense graph.

**Time complexity of visibility graph algo is : O(n \* m^2 + V^2)**

**3. Dijkstra's Search:**

- Once the visibility graph is constructed, Dijkstra's algorithm is used to find the shortest path.

- The time complexity of Dijkstra's algorithm depends on the implementation of the priority queue and the density of the graph.

- For a dense graph, the time complexity is typically O(V^2), and for a sparse graph, it is typically

O((V + E) \* log V).

In summary, the overall time complexity of the Visibility Graph Algorithm implemented in the provided code is dominated by the generation of visibility nodes and the subsequent construction of the visibility graph. Depending on the number of obstacles and the complexity of the environment, the time complexity can be expressed as **O(n \* m^2 + V^2) or O(n \* m^2 + (V + E) \* log V),** where n is the number of obstacles, m is the average number of vertices per obstacle, V is the number of visibility nodes, and E is the number of edges in the visibility graph.

Time Complexity Analysis:

1. Generating Visibility Nodes:

- Complexity: O(N)

- Explanation: This step involves iterating over each obstacle's vertices to generate nodes in the configuration space. The number of vertices in each obstacle determines the linear time complexity.

2. Generating Road Map Info:

- Complexity: O(N \* M)

- Explanation: In this step, connections between visibility nodes are determined based on obstacle intersection. For each visibility node, it checks intersection with every other node. Therefore, the time complexity is quadratic, where N is the number of nodes and M is the average number of intersections per node.

3. Dijkstra Search:

- Complexity: O((V + E) \* log(V))

- Explanation: Dijkstra's algorithm is used to find the shortest path between start and goal nodes. The time complexity depends on the number of vertices (V) and edges (E) in the graph. With proper implementation using a priority queue, the time complexity becomes logarithmic for each insertion and deletion operation, resulting in the overall time complexity.

Overall Time Complexity:

- O(N) + O(N \* M) + O((V + E) \* log(V))

Question 1: What is the time complexity of the provided code?

Answer:

The time complexity of the provided code depends on the specific algorithms and operations used within the code. Without specific details on the algorithms and their implementations, it's challenging to determine the exact time complexity. However, we can analyze the computational complexity of individual components or algorithms within the code to estimate an overall complexity. For instance, if we examine the Dijkstra's algorithm used for path planning in Code 1 or the visibility graph construction in Code 2, we can determine their respective time complexities and factor in any additional operations or computations to estimate the overall complexity.

Question 2: Compare the two codes and tell which one is efficient, and explain in detail why it is efficient.

Answer:

Both codes employ different approaches for path planning. Code 1 uses a grid-based approach, while Code 2 utilizes a visibility graph approach. In terms of efficiency, Code 2 (using the visibility graph) is generally more efficient for several reasons:

1. Grid-based approaches suffer from discretization issues, where the grid resolution directly affects path accuracy and computational complexity. In contrast, visibility graphs capture direct connections between points in the environment, providing a more accurate representation without discretization limitations.

2. Visibility graphs inherently consider obstacle geometries, leading to more natural and collision-free paths. Grid-based approaches often require additional processing, such as obstacle inflation (using parameters like `expand\_distance`), to ensure collision avoidance.

3. The visibility graph approach tends to produce smoother and more direct paths compared to grid-based approaches, resulting in more efficient path execution.

Question 3: Explain what is the grid-based approach in code 1.

Answer:

In Code 1, the grid-based approach involves discretizing the environment into a grid, where each cell represents a portion of the space. The presence of obstacles is typically represented by marking grid cells as occupied. Path planning involves navigating this grid using algorithms like A\* or Dijkstra's algorithm to find a path from the start to the goal while avoiding occupied cells. Parameters like `expand\_distance` are often used to artificially increase the size of obstacles to ensure safe passage for the robot.

Question 4: Why is the `expand\_distance` parameter present in code 1 but not in code 2?

Answer:

The `expand\_distance` parameter is present in Code 1 (grid-based approach) but not in Code 2 (visibility graph approach) because of the differences in how obstacles are handled in each approach. In the grid-based approach, obstacles are represented by occupied grid cells, and the `expand\_distance` parameter is used to artificially increase the size of obstacles to ensure safe passage for the robot. In contrast, the visibility graph approach directly considers obstacle geometries when constructing the graph, eliminating the need for artificial expansion.

Question 5: Explain why the `expand\_distance` parameter is not required in code 2.

Answer:

The `expand\_distance` parameter is not required in Code 2 (visibility graph approach) because the approach fundamentally differs from the grid-based approach used in Code 1. In the visibility graph approach, obstacles are considered directly during graph construction, ensuring that paths naturally avoid obstacles without the need for artificial expansion. The visibility graph captures direct connections between points in the environment, accounting for obstacle geometries inherently and providing collision-free paths without additional expansion.

Question 6: What if direct connections between points in the visibility graph have a width of the path lesser than the obstacles?

Answer:

If direct connections between points in the visibility graph have a width of the path lesser than the obstacles, it introduces potential collision risks. In such cases, additional considerations or adjustments may be needed in the path planning process to ensure safe and feasible paths for the robot. This could involve artificially expanding obstacles, adjusting the path generation algorithm, or accounting for dynamic obstacles or uncertainties in the environment. It's essential to ensure that paths provide adequate clearance for the robot to navigate without colliding with obstacles.

**Structure of Obstacle List:**

* The obstacle list is a Python list containing instances of the **ObstaclePolygon** class.
* Each element of the list represents a separate obstacle in the environment.

**Structure of ObstaclePolygon:**

* The **ObstaclePolygon** class represents a polygonal obstacle.
* It has an attribute **vertices**, which is a list of **vg.Point** objects.
* Each **vg.Point** object represents a vertex of the polygon.

**Representation of Vertices:**

* Each **vg.Point** object consists of **x** and **y** coordinates, representing the position of the vertex in the two-dimensional space.
* These coordinates define the location of the vertex within the environment.

For example, let's consider a simplified scenario with a single obstacle defined by a square. The obstacle list would contain one **ObstaclePolygon** object. This object would have four **vg.Point** objects in its **vertices** attribute, representing the four corners of the square.

Here's a simplified representation of the obstacle list structure:

pythonCopy code

obstacles = [ ObstaclePolygon([ vg.Point(x1, y1), vg.Point(x2, y2), vg.Point(x3, y3), vg.Point(x4, y4) ]),

# Additional ObstaclePolygon objects if there are more obstacles

]

Each **vg.Point** object within the **ObstaclePolygon** objects represents a vertex of the corresponding obstacle. These vertices collectively define the shape and size of the obstacle within the environment.

In summary, the structure of the obstacle list consists of **ObstaclePolygon** objects, each containing a list of **vg.Point** objects representing the vertices of the polygonal obstacles in the environment.

**STORAGE OF VISIBILITY GRAPH:**

the visibility graph is being constructed using the **pyvisgraph** library and stored internally within an instance of the **VisGraph** class. Let's discuss how the visibility graph is being stored:

1. **Construction of Visibility Graph**:
   * The visibility graph is constructed using the **build** method of the **VisGraph** class.
   * This method takes a list of polygons as input, where each polygon represents an obstacle in the environment.
   * Internally, the **build** method computes the visibility graph by determining visible edges between vertices of the polygons.
2. **Representation of Visibility Graph**:
   * The visibility graph is typically represented as a data structure that stores the vertices and edges.
   * In the context of **pyvisgraph**, the visibility graph is represented as a graph data structure.
   * Each vertex of the visibility graph corresponds to a point of visibility within the environment, i.e., a point that can "see" other points without intersecting any obstacles.
   * Edges between vertices represent visibility connections between points.
   * This graph structure allows efficient traversal and pathfinding algorithms to be applied to find shortest paths between given start and end points.
3. **Storage of Visibility Graph**:
   * Once constructed, the visibility graph is stored internally within the **VisGraph** object.
   * The specific implementation details of how the visibility graph is stored may vary based on the **pyvisgraph** library's internal representation.
   * However, it typically involves data structures such as adjacency lists or adjacency matrices to efficiently store the vertices and edges of the graph.
4. **Accessing Visibility Graph**:
   * The visibility graph can be accessed through the methods provided by the **VisGraph** class, such as **shortest\_path**.
   * These methods utilize the internally stored visibility graph to perform operations such as pathfinding.

In summary, the visibility graph is constructed and stored internally within an instance of the **VisGraph** class, allowing efficient computation of shortest paths and other operations related to visibility within the environment.

Let's break down how the graph is stored and provide an example:

**Structure of the Graph:**

* The graph is represented as a dictionary where keys are **Point** objects and values are sets containing **Edge** objects.
* Each **Edge** object connects two **Point** objects.
* The **Graph** class also maintains a separate set called **edges**, which contains all the edges in the graph.

**Example:**

Let's consider a simple example with two polygons:

polygon1 = [Point(0, 0), Point(2, 0), Point(2, 2), Point(0, 2)] polygon2 = [Point(1, 1), Point(3, 1), Point(3, 3), Point(1, 3)] graph = Graph([polygon1, polygon2])

In this example:

* **polygon1** represents a square with vertices at (0, 0), (2, 0), (2, 2), and (0, 2).
* **polygon2** represents another square with vertices at (1, 1), (3, 1), (3, 3), and (1, 3).

**Storage of the Graph:**

* The graph would store edges connecting the vertices of both polygons.
* Each vertex (**Point** object) is a key in the graph dictionary.
* The values in the graph dictionary are sets containing edges (**Edge** objects) incident on each vertex.
* The **edges** set contains all the edges in the graph.

**Example of Graph Storage:**

For the given example, the graph storage would look something like this:

{

Point(0.00, 0.00): {Edge(Point(0.00, 0.00), Point(2.00, 0.00)), Edge(Point(0.00, 0.00), Point(0.00, 2.00))},

Point(2.00, 0.00): {Edge(Point(2.00, 0.00), Point(0.00, 0.00)), Edge(Point(2.00, 0.00), Point(2.00, 2.00))},

Point(2.00, 2.00): {Edge(Point(2.00, 2.00), Point(2.00, 0.00)), Edge(Point(2.00, 2.00), Point(0.00, 2.00))},

Point(0.00, 2.00): {Edge(Point(0.00, 2.00), Point(2.00, 2.00)), Edge(Point(0.00, 2.00), Point(0.00, 0.00))},

Point(1.00, 1.00): {Edge(Point(1.00, 1.00), Point(3.00, 1.00)), Edge(Point(1.00, 1.00), Point(1.00, 3.00))},

Point(3.00, 1.00): {Edge(Point(3.00, 1.00), Point(1.00, 1.00)), Edge(Point(3.00, 1.00), Point(3.00, 3.00))},

Point(3.00, 3.00): {Edge(Point(3.00, 3.00), Point(3.00, 1.00)), Edge(Point(3.00, 3.00), Point(1.00, 3.00))},

Point(1.00, 3.00): {Edge(Point(1.00, 3.00), Point(3.00, 3.00)), Edge(Point(1.00, 3.00), Point(1.00, 1.00))}

}

This representation shows the vertices (**Point** objects) as keys and the edges incident on each vertex as sets of **Edge** objects. Each **Edge** object connects two **Point** objects.

In Python, the **Graph** class provided in the code snippet stores the graph in memory using a combination of dictionaries and sets. Let's break down how the graph is stored:

1. **Vertex Storage**:
   * Vertices are represented by instances of the **Point** class.
   * Each vertex is stored as a key in a dictionary called **graph**.
   * The **graph** dictionary maps each vertex to a set of incident edges.
   * Vertices are uniquely identified by their coordinates **(x, y)**.
2. **Edge Storage**:
   * Edges are represented by instances of the **Edge** class.
   * Each edge is stored as an element in the **edges** set.
   * An edge consists of two endpoints, which are instances of the **Point** class.
   * Edges are stored independently in the **edges** set and are not directly associated with vertices.
3. **Polygon Storage**:
   * Polygons are represented by sequences of **Point** objects.
   * Polygons are provided as input during the initialization of the **Graph** class.
   * Each polygon is traversed to extract edges connecting consecutive points.
   * Edges associated with each polygon are stored in a separate dictionary called **polygons**.

**Memory Representation:**

* The graph data structure is stored in memory using the following attributes of the **Graph** class:
  + **graph**: A dictionary where keys are **Point** objects representing vertices, and values are sets of **Edge** objects representing incident edges.
  + **edges**: A set containing all **Edge** objects in the graph.
  + **polygons**: A dictionary where keys are polygon IDs and values are sets of **Edge** objects representing edges of polygons.
* Each **Point** object consumes memory for storing its **x**, **y**, and **polygon\_id** attributes.
* Each **Edge** object consumes memory for storing references to its two endpoints.
* The overall memory usage depends on the number of vertices, edges, and polygons in the graph, as well as the memory overhead of Python objects.

In summary, the graph is stored in memory using dictionaries and sets to efficiently represent vertices, edges, and polygons, allowing for fast access and manipulation of the graph structure.

**SHORTEST PATH DATA STRUCTURE :**

The code you provided implements Dijkstra's algorithm for finding the shortest path in a graph. Let's break down the data structures used in this implementation:

1. **Priority Dictionary (priority\_dict)**:
   * The **priority\_dict** class is a subclass of the built-in Python **dict**.
   * It is used as a priority queue where keys represent items to be put into the queue, and values represent their respective priorities.
   * Keys in the dictionary represent vertices or items in the graph, and values represent their priorities (shortest distance from the source vertex).
   * This priority dictionary allows efficient updating of priorities, which is crucial for Dijkstra's algorithm.
   * Methods provided include **smallest()** (to retrieve the item with the smallest priority) and **pop\_smallest()** (to remove and return the item with the smallest priority).
2. **Dijkstra's Algorithm Functions (dijkstra and shortest\_path)**:
   * The **dijkstra** function computes the shortest distances and predecessors using Dijkstra's algorithm.
   * It takes the graph, origin vertex, destination vertex, and an optional additional graph structure (**add\_to\_visgraph**) as input.
   * The **shortest\_path** function retrieves the shortest path from the predecessors computed by Dijkstra's algorithm.
   * Both functions utilize the priority dictionary to efficiently handle vertex priorities during the algorithm's execution.
3. **Heap Operations**:
   * The **heapify**, **heappush**, and **heappop** functions are imported from the **heapq** module.
   * These functions are used internally within the **priority\_dict** class to maintain the heap property of the priority queue.

Overall, the primary data structure used in this implementation is the **priority\_dict**, which serves as a priority queue for efficiently implementing Dijkstra's algorithm. Additionally, heap operations from the **heapq** module are utilized to maintain the heap property of the priority queue.

Dijkstra's algorithm is indeed utilizing a priority queue, specifically implemented using the **priority\_dict** class. This class functions as a priority queue where the keys represent items to be placed in the queue, and their values represent their respective priorities.

Here's why the **priority\_dict** serves as a priority queue:

1. **Min Priority Access**: The **smallest()** method efficiently retrieves the item with the smallest priority from the priority queue.
2. **Removal of Minimum Priority Item**: The **pop\_smallest()** method removes and returns the item with the smallest priority from the priority queue, ensuring that the queue maintains its property.
3. **Efficient Priority Updates**: Upon updating the priority of an item in the graph, the **priority\_dict** automatically maintains the heap property, ensuring that the item with the smallest priority remains at the top of the queue.
4. **Heap Operations**: Internally, the **heapify**, **heappush**, and **heappop** functions from the **heapq** module are utilized by the **priority\_dict** class to manage the priority queue.

Therefore, the provided Dijkstra's algorithm implementation indeed works out of a priority queue, facilitated by the **priority\_dict** data structure.

**WEIGHT OF EDGES :**

In the context of Dijkstra's algorithm, "weight" typically refers to the numerical value assigned to edges in a weighted graph. These weights represent the cost or distance associated with traversing from one vertex to another along an edge.

In the provided implementation of Dijkstra's algorithm, the weight or distance between vertices is computed using the **edge\_distance()** function, which calculates the Euclidean distance between two points (vertices) in the graph. This distance is used as the weight of the edge between the two vertices.

Here's how the weight is used in the algorithm:

1. When computing the shortest path, the algorithm considers the cumulative weight of edges traversed from the source vertex to each of its neighbors. It selects the neighbor with the minimum total weight as the next vertex to visit.
2. The **priority\_dict** class maintains the priority queue based on the cumulative weights of vertices. The vertex with the smallest cumulative weight is given the highest priority in the queue.

Overall, in the context of this implementation, the weight represents the Euclidean distance between vertices and is crucial for determining the shortest path between two points in the graph.

**ADDING NEW OBSTACLE IN VISIBILITY GRAPH:**

In the provided code, the visibility graph is updated by adding new obstacles to the existing graph using the **update\_visibility\_graph** function. Here's how it works:

1. Initially, the visibility graph **g** is constructed using the **build** method of **VisGraph**, which creates a visibility graph based on the given obstacles.
2. Later, new obstacles are read from a file using the **read\_obstacles\_from\_file** function. These new obstacles are represented as lists of **(x, y)** coordinates.
3. The **update\_visibility\_graph** function takes the existing visibility graph **g** and the list of new obstacles as input arguments. It then iterates over each new obstacle and updates the visibility graph by adding the visibility edges associated with each new obstacle using the **update** method of **VisGraph**.
4. After updating the visibility graph with the new obstacles, the shortest path computation is recomputed using the updated graph to reflect the changes in the environment.
5. Finally, the updated obstacles and the new shortest path are displayed.

In summary, the visibility graph is updated by adding new edges to the existing graph rather than rebuilding the graph from scratch. This approach is more efficient as it avoids unnecessary recomputation of existing edges and takes advantage of the existing graph structure.