**Task 01:**

We used the "relaxed Dijkstra" version of Dijkstra's algorithm to discover the shortest path in the first task from a specified source node (1) to a target node(50) in a graph. Let's step-by-step dissect the methodology employed in the code:

**Initialization:**

* **distances:** This dictionary is used to keep track of the minimum distance from the source node to each node in the graph. It initializes all distances to float('inf') except for the source node, which is set to 0. This is because the distance from the source node to itself is zero.
* **predecessors:** This dictionary is used to keep track of the previous node in the shortest path to each node. It initializes all predecessors to None.
* **queue:** This is a priority queue (min-heap) that stores nodes to be explored. Each element in the queue is a tuple (current\_distance, current\_node), where current\_distance is the minimum distance from the source to current\_node.

**Exploring Nodes:**

* In each iteration of the loop, the node with the smallest current distance (i.e., the node with the minimum current\_distance) is removed from the queue.
* If the removed node is the target node, the algorithm has found the shortest path. It reconstructs the path from the target node to the source node by following the predecessors dictionary in reverse order and returns the path.
* If the current\_distance for the removed node is greater than the stored distance in the distances dictionary, it means that a shorter path to that node has already been found, so this iteration is skipped.

**Path Reconstruction:**

If the target node is reached, the code reconstructs the shortest path from the target node to the source node by following the predecessors dictionary in reverse order. The path is stored in the path list.

**Termination:**

If the loop completes without reaching the target node, it means that there is no path from the source node to the target node, and an empty list is returned.

In summary, this relaxed Dijkstra's algorithm explores nodes in the graph while maintaining and updating the minimum distances to each node from the source. When it reaches the target node, it reconstructs the shortest path, if one exists. This algorithm is particularly useful for finding the shortest path between two nodes in a weighted graph.

**Task 02:**

In task 2 we defines a function called ucs\_with\_energy which stands for "Uniform Cost Search with Energy." This function is used to find the shortest path from a source node to a target node in a graph while taking into account an energy budget constraint. It is essentially a modification of Dijkstra's algorithm, where the cost of edges includes an energy cost, and the algorithm ensures that the accumulated energy cost along the path does not exceed the given energy budget.

Here's a step-by-step explanation of the code:

1. **Initialize data structures:**

* **distances:** A dictionary that stores the shortest distance from the source node to each node in the graph. All distances are initialized to positive infinity (float('inf')) except for the source node, which is set to 0.
* **predecessors:** A dictionary that keeps track of the predecessor node for each node in the graph. It is used to reconstruct the path later.
* **queue:** A priority queue (implemented as a list) to keep track of nodes that need to be explored. Each element in the queue is a tuple with three values: the current distance from the source node, the current node, and the current energy cost.

1. While the queue is not empty, the algorithm continues to explore nodes.
2. Pop the node with the smallest total distance (including energy cost) from the priority queue. This is done using heapq.heappop. The current node, its distance, and energy cost are extracted.
3. Check if the current node is the target node. If so, a path has been found. In this case, the function reconstructs the path by backtracking through the predecessors dictionary and returns the path as a list of nodes in the correct order.
4. If the current node is not the target, the algorithm continues.
5. Check if the current distance is greater than the previously recorded distance to the current node or if the current energy cost exceeds the given energy budget. If either of these conditions is met, skip the current node and move on to the next one.
6. Iterate through the neighbors of the current node (retrieved from the graph data structure). For each neighbor, calculate the total distance and energy cost if you were to reach that neighbor via the current node.
7. If the new distance is shorter than the previously recorded distance to the neighbor and the new energy cost is within the budget, update the distances and predecessors dictionaries, and push the neighbor into the priority queue with its new distance and energy cost.
8. If the queue becomes empty and no path is found, return an empty list to indicate that there is no path from the source to the target within the energy budget constraint.

In summary, this function performs a modified Dijkstra's algorithm to find the shortest path from the source to the target in a graph while considering energy constraints for edges. It returns the path if found, or an empty list if no path within the energy budget is possible.

**Task 03:**

In task 03 we define a function called **astar\_with\_energy** which stands for "A\* Search with Energy." This function is used to find the shortest path from a source node to a target node in a graph while considering an energy budget constraint. It is a modification of the A\* search algorithm, where the cost of edges includes an energy cost, and an admissible heuristic function is used to estimate the remaining energy required to reach the target node.

Here's a step-by-step explanation of the code:

1. **Initialize data structures:**

* **distances:** A dictionary that stores the estimated total cost from the source node to each node in the graph. All costs are initialized to positive infinity (float('inf')) except for the source node, which is set to 0.
* **predecessors:** A dictionary that keeps track of the predecessor node for each node in the graph. It is used to reconstruct the path later.
* **queue**: A priority queue (implemented as a list) to keep track of nodes that need to be explored. Each element in the queue is a tuple with three values: the current estimated total cost, the current node, and the current energy cost.

1. While the queue is not empty, the algorithm continues to explore nodes.
2. Pop the node with the smallest estimated total cost (including energy cost and the heuristic estimate) from the priority queue using heapq.heappop. The current node, its estimated total cost, and energy cost are extracted.
3. Check if the current node is the target node. If so, a path has been found. In this case, the function reconstructs the path by backtracking through the predecessors dictionary and returns the path as a list of nodes in the correct order.
4. If the current distance is greater than the previously recorded estimated total cost to the current node or if the current energy cost exceeds the given energy budget, skip the current node and move on to the next one.
5. Iterate through the neighbors of the current node (retrieved from the graph data structure). For each neighbor, calculate the energy cost to reach that neighbor and check if it exceeds the energy budget. If it does, skip the neighbor.
6. Calculate the new estimated total cost to reach the neighbor by adding the current energy cost and the estimated remaining energy required to reach the target. The estimate\_remaining\_energy function provides the heuristic estimate for the remaining energy.
7. If the new estimated total cost is lower than the previously recorded cost for the neighbor, update the distances and predecessors dictionaries and push the neighbor into the priority queue with its new estimated total cost and energy cost.
8. If the queue becomes empty and no path is found, return an empty list to indicate that there is no path from the source to the target within the energy budget constraint.

The **estimate\_remaining\_energy** function is a key component of the algorithm, providing an admissible heuristic estimate for the remaining energy required to reach the target node. In the provided code, a simple heuristic is used, which calculates the straight-line Euclidean distance between nodes, assuming no energy costs. You can modify this heuristic function based on your specific problem and data.

In summary, this function performs a modified A\* search to find the shortest path from the source to the target in a graph while considering energy constraints for edges and using an admissible heuristic estimate for the remaining energy required to reach the target. It returns the path if found, or an empty list if no path within the energy budget is possible.