This code implements Dijkstra's algorithm for finding the shortest path from a single source node to all other nodes in a weighted graph. Here's an explanation of each part of the code:

1. `import heapq`: Imports the `heapq` module, which provides an implementation of the heap queue algorithm, needed for the priority queue used in Dijkstra's algorithm.

2. `def dijkstra(graph, start)`: Defines a function `dijkstra` that takes a graph and a start node as input and returns a dictionary of shortest distances from the start node to all other nodes in the graph.

3. `distances = {node: float('inf') for node in graph}`: Initializes a dictionary `distances` with all nodes in the graph mapped to infinity, indicating that the shortest distance to those nodes is currently unknown.

4. `distances[start] = 0`: Sets the shortest distance from the start node to itself to 0, as it is the starting point of the traversal.

5. `priority_queue = [(0, start)]`: Initializes a priority queue with a tuple containing the current distance (0 for the start node) and the start node itself.

6. `while priority_queue:`: Loops as long as there are nodes in the priority queue.

7. `current_distance, current_node = heapq.heappop(priority_queue)`: Pops the node with the smallest distance from the priority queue.

8. `if current_distance > distances[current_node]: continue`: Skips the current node if a shorter path to it has already been found.

9. `for neighbor, weight in graph[current_node].items()`: Iterates over the neighbors of the current node and their corresponding edge weights.

10. `distance = current_distance + weight`: Calculates the total distance to the neighbor node through the current node.

11. `if distance < distances[neighbor]:`: Updates the shortest distance to the neighbor node if the new path is shorter.

12. `distances[neighbor] = distance`: Updates the shortest distance to the neighbor node.

13. `heapq.heappush(priority_queue, (distance, neighbor))`: Pushes the neighbor node onto the priority queue with its updated distance.

14. `return distances`: Returns the dictionary of shortest distances from the start node to all other nodes in the graph.

15. `graph`: Defines an example graph represented as a dictionary of nodes mapped to their neighbors and edge weights.

16. `start_node = 'A'`: Specifies the start node for running Dijkstra's algorithm.

17. `shortest_distances = dijkstra(graph, start_node)`: Runs Dijkstra's algorithm on the example graph starting from the specified start node.

18. `print("Shortest distances from node", start_node + ":")`: Prints a message indicating the start node.

19. `for node, distance in shortest_distances.items():`: Iterates over the nodes and their corresponding shortest distances.

20. `print(node + ":", distance)`: Prints each node and its shortest distance from the start node.

Overall, this code efficiently finds the shortest path from a single source node to all other nodes in a graph using Dijkstra's algorithm.