Q3: ELI5 of Dijkstra's (SSSP problem) and Prim's (MST problem) algorithms

> Both Dijkstra and Prim's algorithms are so fundamental in understanding the areas of greedy algorithms and Graph theory. Because of that, they are taught in literally every algorithms and graph theory class. So it is very important for you to have a complete and clear understanding of these algorithms. Also, understanding these two algorithms opens the door to solving other similar problems.
>
> There are many articles and references available out there for the topics. You may do your own research as much as you can but you should come out with your own words to explain how they work in detail along with a few slides to summarize them. Please be mindful of the spirit of ELI5 and try to prove it in simple English.

Explanation:

- The Graph Model

To understand the following algorithms, we must first understand what a Graph Model is.

- A **Graph Model** in computer science refers to an *abstract data structure* used to represent non-linear relationships between data items.
- A graph consists of a set of **vertices** (nodes containing data) and **edges** that connect the vertices.
  - Graphs use the following notation: `G = {Vertex,Edge}`
- There are different types of graphs:
  - Undirected Graph:
    - can move in any direction between vertices (nodes)
    - like a 2-way street
  - Directed Graph:
    - can only move in direction specified by arrows on edges
    - like a 1-way street
  - Weighted Graph:
    - can be either undirected or directed graphs

- have values (weights) associated with the edges
- similar to distances between places

> Dijkstra's Algorithm (SSSP Problem)

The Single Source Shortest Path (SSSP) Problem consists of finding the shortest path from a specific vertex to ***all*** vertices in the given graph.

- Properties:
  - Graph is directed & weighted
  - Graph only has positive weights

- The Greedy Algorithm:

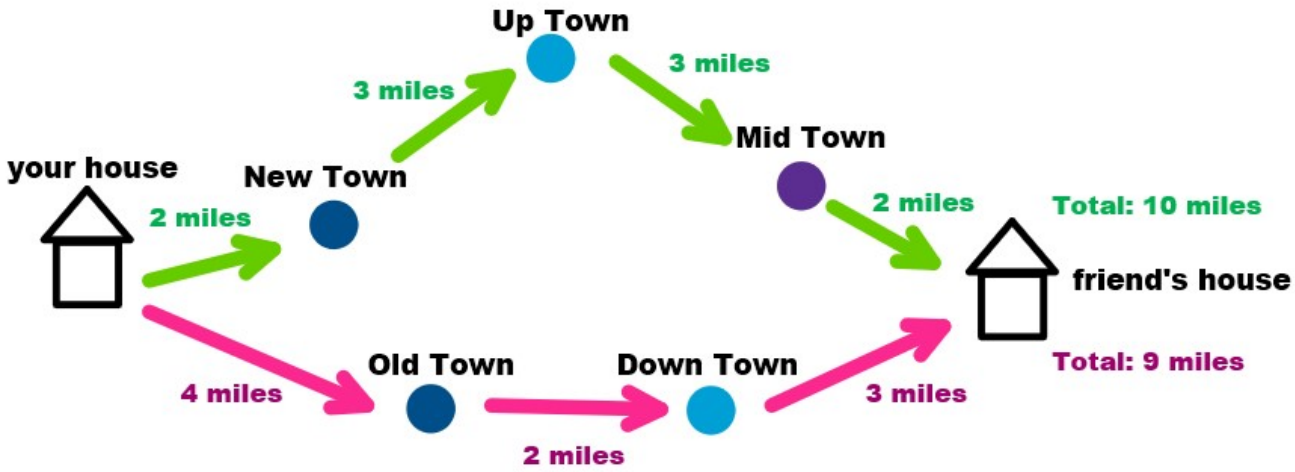
Dijkstra's algorithm is a way to solve this problem, by implementing a **greedy algorithm**. A greedy algorithm is one that builds a solution step by step, where at each step the "best" option for that step is chosen.

In the way we are approaching the SSSP problem using Dijkstra's greedy algorithm, we can think of it like this:

> You want to go to your friend's house in a different town.
>
> You don't know the directions, or have a map, so you ask someone for directions. They tell you that you can either go through **New Town** or **Old Town** to get to where your friend lives.
>
> You ask how far both of those are, **New Town** is **2 miles** away, while **Old Town** is **4 miles** away.
>
> Obviously, you don't want to walk that far, so you pick the shortest route, through **New Town**.
>
> Once you get there, you repeat the process and ask someone else where to go next. Then again, you choose the shortest path.
>
> You repeat these steps till you get to your friend's house. It took you ***10 miles*** to get there.

> While you think this is the right path to take, you didn't know that if you had gone to **Old Town** first, you would have gotten to your friend's house in ***9 miles*** instead.

Here is a simple diagram of your path (in green) and the shortest path (in pink):



This is basically the same idea Dijkstra's algorithm follows in order to solve the SSSP problem. However, it does one more thing that makes it beneficial to problem-solving: using *edge relaxation*.

Edge relaxation refers to updating the path based on the weight of the edges in a graph.

In our example, this would mean that instead of just choosing the shortest distance for our first town visited, we would also look at the following town and the one after that, to see if the path *really* is shorter, or just *seems* to be.

- Code for Dijkstra's Algorithm

```
def dijkstra(graph, s):
    visited = {s: 0}; path = {}
    nodes = set(graph.nodes)
    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None: min_node = node
                elif visited[node] < visited[min_node]: min_node = node
            if min_node is None: break
            nodes.remove(min_node)
            current_weight = visited[min_node]
            for v in graph.edges[min_node]:
                weight = current_weight + graph.distances[(min_node, v)]
                if v not in visited or weight < visited[v]:
```

```
                    visited[v] = weight
                    path[v] = min_node
    return visited, path
```

> Prim's Algorithm (MST Problem)

The Minimum Spanning Tree (MST) Problem consists of a **spanning tree** which is a form of a graph. A tree is simply a disjoint graph, and a spanning tree is a *subgraph* of that graph, which only includes the vertices (nodes containing data) of the graph.

Properties of spanning tree (subgraph):

- Contains all vertices from main graph
- All vertices are connected (there is a path of edges going to each vertex)
- There can't be any cycles (loops where vertices are connected in a circular fashion)

In the MST Problem, we are given an undirected & weighted graph. The goal is to achieve a *minimum* total weight while visiting every vertex in the graph.

Prim's algorithm utilizes a *greedy algorithm* just like Dijkstra's. But in Prim's algorithm, we use it to find the minimum spanning tree.

- Steps to Prim's Algorithm:

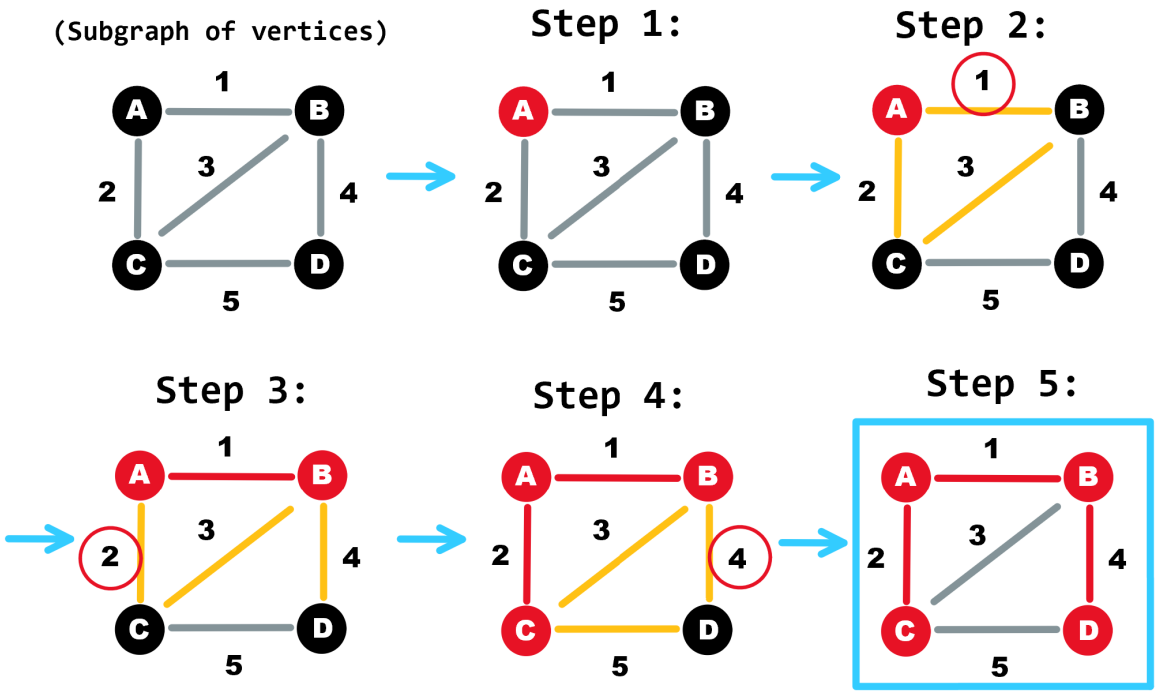
1. To build our tree, choose **any** vertex from the graph as a starting point
    - Here, we'll pick node `A` to begin
    - When a vertex is visited, it's added to the MST: `{A}`

2. From the starting node, look at all the **weights** of the edges connected to it
    - For `A` the edges have the values *1, 2, 3*
    - Pick the edge with the **smallest weight**
    - Here, that is the edge with weight `1` leading to node `B`
    - Add the new node to the MST: `{A, B}`

3. Now look at next set of edges that connect visited nodes to unvisited nodes
   - We want to pick only edges that lead to **unvisited nodes**
   - So the edge with value `1` is out
   - Choose the smallest weighted edge again
   - In this case, the value is `2` for the edge connecting `A` to `C`
   - Visit the new node, add it to the MST: `{A, B, C}`

4. Now we've reached the last set of edges, with values *3, 4, 5*
   - Usually we would pick the smallest weight, which is *3*
   - This is not the case here, since the edge with weight *3* would lead us back to an already visited node, `B`
   - The next lowest weight is *4*, so we choose this edge

5. Visit the final node and add it to the MST: `{A, B, C, D}`
   - The MST is complete!
   - All nodes have been visited only once, and the minimum weighted path was found using Prim's algorithm.
   - The minimum spanning tree has a total weight, or **cost** of *1 + 2 + 4 =* ***7***



- Code for Prim's Algorithm

```
def primMST(self): #self is a graph (G)
    key = [sys.maxsize] * self.V # set to infinity
    parent = [None] * self.V # Array to store
    key[0] = 0
    mstSet = [False] * self.V
    parent[0] = -1 # First node is always the root
    for cout in range(self.V):
        u = self.minKey(key, mstSet)
        mstSet[u] = True
        # self.graph[u][v] is an adjacent matrix
        for v in range(self.V):
            if mstSet[v] == False and self.graph[u][v] > 0 and key[v] >
self.graph[u][v]:
                key[v] = self.graph[u][v]
                parent[v] = u
```

References:

Graphs - Isaac Computer Science

Graphs: Single-Source-Shortest-Path

Greedy Algorithm - Programiz

Edge Relaxation in Dijkstra's Algorithm - Baeldung

Activity Selection Problem | Greedy Algo-1 - GeeksforGeeks

Minimum Spanning Tree Tutorials & Notes | Algorithms

Spanning Tree and Minimum Spanning Tree - Programiz