Simulated annealing minimum spanning tree

```
import random

import math

from collections import defaultdict


class Graph:

    def __init__(self):

        self.edges = defaultdict(list)


    def add_edge(self, u, v, weight):

        self.edges[u].append((v, weight))

        self.edges[v].append((u, weight))  # Undirected graph


    def get_edges(self):

        return [(u, v, weight) for u in self.edges for v, weight in self.edges[u] if u < v]


def random_spanning_tree(graph):

    nodes = list(graph.edges.keys())

    random.shuffle(nodes)

    tree_edges = set()

    selected = {nodes[0]}


    while len(selected) < len(nodes):

        u = random.choice(list(selected))
```

```
        candidates = [(v, weight) for v, weight in graph.edges[u] if v not in selected]
        if candidates:
            v, weight = random.choice(candidates)
            tree_edges.add((u, v, weight))
            selected.add(v)

    return tree_edges

def energy(tree):
    return sum(weight for u, v, weight in tree)

def generate_neighbor(tree, graph):
    tree_list = list(tree)
    if len(tree_list) < 2:
        return tree

    # Select a random edge to remove
    u, v, weight = random.choice(tree_list)
    new_tree = tree - {(u, v, weight)}

    # Find a new edge to add
    candidates = [(x, w) for x, w in graph.edges[u] if (x, u, w) not in tree and (u, x, w) not in tree]
    if not candidates:
        # If no candidates are available, return the original tree
```

```
            return tree

        new_v, new_weight = random.choice(candidates)

        # Add the new edge and check for cycles
        new_tree.add((u, new_v, new_weight))

        # Ensure the new tree is valid (could add a check here if necessary)
        return new_tree

def simulated_annealing(graph):
    T = 1.0  # Initial temperature
    final_temperature = 0.001
    cooling_factor = 0.95
    current_solution = random_spanning_tree(graph)
    best_solution = current_solution

    while T > final_temperature:
        for _ in range(100):  # Number of iterations at current temperature
            neighbor = generate_neighbor(current_solution, graph)
            current_energy = energy(current_solution)
            neighbor_energy = energy(neighbor)

            if neighbor_energy < current_energy:
```

```
            current_solution = neighbor
        else:
            acceptance_probability = math.exp((current_energy - neighbor_energy) / T)
            if random.random() < acceptance_probability:
                current_solution = neighbor

        if energy(current_solution) < energy(best_solution):
            best_solution = current_solution

    T *= cooling_factor

  return best_solution

# Example usage:
if __name__ == "__main__":
  graph = Graph()
  edges = [(0, 1, 4), (0, 2, 1), (1, 2, 2), (1, 3, 5), (2, 3, 3)]
  for u, v, weight in edges:
    graph.add_edge(u, v, weight)

  mst = simulated_annealing(graph)
  print("Edges in the Minimum Spanning Tree:")
  for u, v, weight in mst:
    print(f"{u} -- {v} (weight: {weight})")
```

```
print("Total weight:", energy(mst))
```

```
Edges in the Minimum Spanning Tree:
2 -- 0 (weight: 1)
2 -- 3 (weight: 3)
2 -- 1 (weight: 2)
Total weight: 6
```