CSCI B505 Spring 20: Extra Programming assignment 5 (10 + 10 + 10 + 10 points)

Due online: Saturday, May 2, 11:59pm EST

Submit your work online using Canvas. You can use Python or Java. Ask AI's if you have any questions. Please use files graphs_and_hashtables.py or GraphsAndHashtables.java.

Grading

This is is a bonus programming assignment. It's pretty big and can give you up to 40 points (twice as much as other programming assignments). The grading is the following:

- 11 points for HashTable implementation.
- 9 points for graph implementation.
- 10 points for minimum spanning forest implementation.
- 10 points for shortest path implementation.

For each part, 1 point is for tests. For first 2 parts, 0.5 points are for time complexity.

Since part 2 depends on part 1, and parts 3 and 4 depend on part 2, you are allowed to do the following:

- If you have problems with part 1, you can use built-in HashTables for part 2.
- If you have problems with part 2, you can use adjacency list representation for parts 3 and 4.

Of course, you'll lose points for parts which you didn't implement.

What to submit

You should submit a single source file. Please follow good coding practices: use indentation, write comments, etc. The code should include the following:

- Implementation of methods as described below.
- Running time for all methods.
- An exhaustive set of tests (using unittest for Python and JUnit for Java).

Motivation

We want to implement an **undirected weighted** graph, which supports the following operations:

- Create edge (u, v) with weight $w \geq 0$.
- Delete edge (u, v).
- Check that edge (u, v) exists.
- Create vertex u in a graph.
- Delete vertex u. The vertex should disappear from the graph. All edges adjacent to this vertex must be deleted.
- Find $deg(u)$.
- For vertex u, create a list of its neighbors.
- **(10 points)** Build a minimum spanning forest of the graph.
- **(10 points)** Finding a shortest path from vertex u to vertex v.

Moreover, we want to have optimal memory complexity and optimal running time for all operations. For adjacency list and list of edges representations, edge deletion is inefficient. For adjacency matrix representation, memory (and some operations) is not optimal.

Main idea

The asymptotically optimal representation has a design which is similar to one of an adjacency list. However, instead of list of lists, we store a graph as a HashTable of HashTables.

- For each node u, let $\{(u, v_i)\}_i$ be its adjacent edges with corresponding weights $\{w_i\}_i$. If we would use an adjacency list, we could store such edges as a list of pairs $[(v_1, w_1), \ldots, (v_k, w_k)]$. However, some list operations are slow (e.g. search and removal), so we instead store them in HashTable which contains mappings $v_i \to w_i$. We denote such HashTable as $\mathcal{N}(u)$.
- We store a mapping from u to $\mathcal{N}(u)$. We denote such HashTable as $\mathcal{G}$.

For example, we can store graph with vertices 1, 7, 10, 20 and edges $(1, 7)$ with weight 2 and $(7, 10)$ with weight 3 in the following way:

```
{
    1  -> { 7 -> 2 },
    7  -> { 1 -> 2, 10 -> 3},
    10 -> { 7 -> 2 },
    20 -> {}
},
```

where $\{a \to b, c \to d\}$ denotes a HashTable where a is mapped to b and c is mapped to d.

Implementation

HashTable

(10 points) HashTables should be implemented by hand in the following way:

- $hash(u) = u$.
- If m is a capacity of a HashTable, then index of u in the HashTable should be calculated as $|hash(u)|\%m$.
- Collisions should be resolved using chaining, as described in lectures. For chaining, please Python's list or Java's ArrayList.
- When the number of elements in a HashTable increases, its performance plummets. To avoid it, when a HashTable becomes half-full (i.e. the number of elements is 1/2 of HashTable capacity), perform the following operation:
  - Create a new HashTable, with the capacity (the number of different indices) twice as large as previous HashTable capacity.
  - For all mappings $k \to v$ in the old HashTable, create mapping $k \to v$ in the new HashTable.
  - Replace the old HashTable with the new one.

  Since you have a class HashTable, this operation can be performed inside the class itself, without being visible to outside code. Since a HashTable is internally represented as an array, you can create a new array, fill at as described and replace the old array with it.

  You can implement HashTable without this operation (-3 points). In this case, create HashTable of a reasonable constant size.

HashTable should have the following methods:

- $double\_capacity()$. Doubles the capacity, as described above.
- $set(k, v)$ (or $\_\_setitem\_\_(k, v)$). Maps key k to value v. If key k already exists, its value is changed. **If the number of elements becomes at least** $1/2$ **of the capacity,** $double\_capacity()$ **should be called**.
- $get(k)$ (or $\_\_getitem\_\_(k)$). Returns the value associated to the key. Returns null (or None) if the key doesn't exist.
- $delete(k)$. Deletes key k. If the key doesn't exists, throws an exception.
- $keys()$. Returns a list which contains all existing keys. Create an empty list, iterate over HashTable and add all keys into the list.
- $size()$. Returns the number of entries. Must take $O(1)$ time. The simplest way to do this is to maintain a field $\_size$ and update it when the number of items changes.

Graph

We represent nodes as integers. However, they are not necessarily integers from 0 to $n - 1$: they can be arbitrary integers. For example, we can have a graph with three nodes: 1, -10 and 135376314.

Graph operations should be implemented in the following way:

- $create\_edge(u, v, w)$: create edge (u, v) with weight $w \geq 0$. In $\mathcal{N}(u)$, create a mapping $v \to w$. In $\mathcal{N}(v)$, create a mapping $u \to w$. If an edge already exists, throw an exception. If u or v don't exist, throw an exception. If $w < 0$, throw an exception.

- *delete_edge*(u, v). Remove key v from $\mathcal{N}(u)$. Remove key u from $\mathcal{N}(u)$. If edge (u, v) doesn't exist, throw an exception
- *has_edge*(u, v): check that edge (u, v) exists. Check that $\mathcal{N}(u)$ has key v. If u or v don't exists, throw an exception.
- *create_vertex*(u). Create a mapping $u \to \mathcal{N}(u)$, where $\mathcal{N}(u)$ is an empty HashTable. If u already exists in the graph, throw an exception.
- *delete_vertex*(u). For all neighbors v of u, remove key u from $\mathcal{N}(v)$. Then remove u from $\mathcal{G}$. Throw an exception if u doesn't exist.
- *deg*(u). Return size of $\mathcal{N}(u)$. If u doesn't exist, throw an exception.
- *neighbors*(u). Just use $\mathcal{N}(u).keys()$. If u doesn't exist, throw an exception.
- *mst*(): Build a minimum spanning forest of the graph. Use any algorithm from lectures. You should return a list of edges which form a spanning forest. If there are multiple solutions, return any of them.
- *shortest_path*(u, v). Use Dijkstra's algorithm from lectures. Time complexity must be $O((n+m) \log n)$. You should return the path itself as a list: $[u, v_1, \ldots, v_k, v]$. If u and v are not connected, return an empty list. If u or v don't exist, throw an exception.

It's up to you what types of exceptions to throw; select the one which reasonably describes the problem.

Running times

In a comment in the beginning of your source file, show worst-case time complexity for all methods (you don't have to prove it). For each operation, please specify a tightest possible running time in terms of n, m and degree of vertices passed as parameters of the operation.

Tests (4 points)

Please check correctness of your implementation with an exhaustive set of tests (using unittest for Python and JUnit for Java). Your tests should also cover cases when an operation is invalid (e.g. when attempting to delete an edge which doesn't exist); for this you can check `https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises` for Python and `https://stackoverflow.com/a/156528/12463032` (you'll likely have JUnit 4.12) for Java.