Pardis Sadatian Moghaddam-Assignment 2

ReadMe file

Trajectory.py:

In the beginning, we are building a class as a collection of time-geometry pairs. As mentioned in the assignment we consider it as the tgi = (ti, gi). ti is the temporal extent (a time point/timestamp or time interval) and gi is the spatial extent representing a geometry. The other class is TimeGeometryPair, we are going to initialize the ti and gi as the temporal_extent and spatial_extent. The last step is traj_id and tgpairs that we need to consider.

We need to check that if we implement our trajectory correctly so we give two test cases to our code:

In this code, we give t1 and t2 as the temporal_extent and g1 and g2 as the spatial_extent. The next step is to create a trajectory and put it in t1 and t2. As we define a method “str” that only returns the trajectory_id, here is the output we get from this part:

```
from intervaltree import Interval, IntervalTree #we use the intervaltree
    library
import math
#We are building a class as a collection of time-geometry-pairs
#We have ti and gi, ti is the temporal extent and gi is the spatial extent
class TimeGeometryPair:
      def __init__(self, temporal_extent, spatial_extent):
          self.temporal_extent = temporal_extent #ti
          self.spatial_extent = spatial_extent #gi
#Here we made a list of time_geometry_pair
class Trajectory:
  def __init__(self, traj_id, tgpairs):
    self.traj_id = traj_id
    self.tgpairs = tgpairs

  def __str__(self):
    return f"{self.traj_id}" # replace as needed
# t1 = Trajectory("John", None)
# print(t1)
tgpairs1 = TimeGeometryPair(temporal_extent=(1, 10), spatial_extent=(2, 3)),
tgpairs2 = TimeGeometryPair(temporal_extent=(5, 15), spatial_extent=(4, 5)),
tgpairs3 = TimeGeometryPair(temporal_extent=(20, 30), spatial_extent=(30, 30
    )),
tgpairs4 = TimeGeometryPair(temporal_extent=(40, 50), spatial_extent=(40, 40
    )),
t1 = Trajectory(traj_id="pardis_1", tgpairs=tgpairs1)
t2 = Trajectory(traj_id="pardis_2", tgpairs=tgpairs2)
print (t1)
print (t2)
```

This is what we get as an output from Trajactory.py. We give two test cases t1 and t2 and based on the return f""{self.traj_id}" we will get the id's of the trajectory as an output:

```
1  pardis_1
2  pardis_2
```

GIT.py:

As we mentioned in the assignment, we have these variables and we initiate these variables sf_xmin (minimum x-coordinate value for spatial framework), sf_xmax (maximum x-coordinate value for the spatial framework), sf_ymin (minimum y-coordinate value for the spatial framework), sf_ymax (maximum y-coordinator value for the spatial framework). And we have delta x and delta y

Based on these variables, we calculate nx and ny. Also, we need to make the grid. This grid consists of keys as the temporal extent and value and the trajectory id. So, we can implement this with a dictionary.

Queries:

1. **Insertion**: we need to insert a trajectory object to the index structure. For implementing this query, we are going to consider the start time and end time. We put this inside the dictionary as the start_time and end_time. We also need to check if the start_time and end_time not be in the dictionary. The next step is the trajectory_id. So we will add this id inside the dictionary as the value. The next step is to check the insert method with two test cases. We consider t1 and t2 as we mentioned before and we need to check if these two trajectories be in the dictionary as the key and value. Here is the output of the code:
   We have the dictionary with (t1,t2) key and ['pardis_1'], ['pardis_2'], ['pardis_3'], ['pardis_4'] as the values. I build these two test cases for this query to see the results in two different dictionaries:

Test Case 1:

```
#Insertion Query
#print test case 1
tgpairs1 = TimeGeometryPair(temporal_extent=(1, 10), spatial_extent=(2, 3)),
tgpairs2 = TimeGeometryPair(temporal_extent=(5, 15), spatial_extent=(4, 5)),
tgpairs3 = TimeGeometryPair(temporal_extent=(20, 30), spatial_extent=(30, 30
    )),
tgpairs4 = TimeGeometryPair(temporal_extent=(40, 50), spatial_extent=(40, 40
    )),


t1 = Trajectory(traj_id="pardis_1", tgpairs=tgpairs1)
t2 = Trajectory(traj_id="pardis_2", tgpairs=tgpairs2)
t3 = Trajectory(traj_id="pardis_3", tgpairs=tgpairs3)
t4 = Trajectory(traj_id="pardis_4", tgpairs=tgpairs4)
git_index1 = GIT(sf_xmin=0, sf_xmax=10, sf_ymin=0, sf_ymax=10, delta_x=10,
    delta_y=10)

git_index1.insert(t1)
git_index1.insert(t2)
git_index1.insert(t3)
git_index1.insert(t4)

#test case 1 for the insertion query
print ("test case 1 Insertion")
print(git_index1.grid)
```

Test Case 1 Output:

```
test case 1 Insertion
{(1, 10): ['pardis_1'], (5, 15): ['pardis_2'], (20, 30): ['pardis_3'], (40, 50
    ): ['pardis_4']}
```

Test Case 2:

```
#print test case 2
tgpairs1 = TimeGeometryPair(temporal_extent=(1, 2), spatial_extent=(10, 10)),
tgpairs2 = TimeGeometryPair(temporal_extent=(3, 14), spatial_extent=(20, 20)),
tgpairs3 = TimeGeometryPair(temporal_extent=(5, 6), spatial_extent=(30, 30)),
tgpairs4 = TimeGeometryPair(temporal_extent=(7, 8), spatial_extent=(40, 40)),


t1 = Trajectory(traj_id="pardis_1", tgpairs=tgpairs1)
t2 = Trajectory(traj_id="pardis_2", tgpairs=tgpairs2)
t3 = Trajectory(traj_id="pardis_3", tgpairs=tgpairs3)
t4 = Trajectory(traj_id="pardis_4", tgpairs=tgpairs4)
git_index2 = GIT(sf_xmin=1, sf_xmax=5, sf_ymin=1, sf_ymax=5, delta_x=20,
    delta_y=20)

git_index2.insert(t1)
git_index2.insert(t2)
git_index2.insert(t3)
git_index2.insert(t4)

#test case 2 for the insertion query
print ("test case 2 Insertion")
print(git_index2.grid)
```

Test Case 2 Output:

```
test case 2 Insertion
{(1, 2): ['pardis_1'], (3, 14): ['pardis_2'], (5, 6): ['pardis_3'], (7, 8):
    ['pardis_4']}
```

2. **Deletion by id:** in this query, we are planning to remove the trajectory by id. We consider a Boolean to check if the trajectory id exists inside the dictionary and if there exists, we delete the trajectory id. This is the value so we need to delete the value with the key and removed the whole trajectory from the dictionary.
Test Case 1: In this test case, we are deleting the trajectory that the id is pardis_1

```
#delete one of the trajectories by ID
#test case 1
print ("test case 1 Deletion")
deleted = git_index2.delete_by_id("pardis_1")

if deleted:
    print("Trajectory deleted successfully")
    print(git_index2.grid)
else:
    print("Trajectory not found")
```

Test Case 1 Output:

```
test case 1 Deletion
Trajectory deleted successfully
{(3, 14): ['pardis_2'], (5, 6): ['pardis_3'], (7, 8): ['pardis_4']}
```

Test Case 2: In this test case, we are deleting the trajectory that the id is pardis_5. This trajectory is not exist so we not getting any results:

```
#test case 2
print ("test case 2 Deletion")
deleted = git_index2.delete_by_id("pardis_5")
if deleted:
    print("Trajectory deleted successfully")
    print(git_index2.grid)
else:
    print("Trajectory not found")
```

Test Case 2 Output:

```
test case 2 Deletion
Trajectory not found
```

3. **Temporal Window:** we have two time points (t1,t2). We need to find all the trajectory id's that are overlap with this time range. Based on the test cases we give t1=25 and t2=45. Based on the t1, t2, t3, t4 that we give we have this output is pardis_3 and pardis_4. The second test case is the temporal range between 60 and 70. So, we are not going to get any results.

Test Case 1:

```
#Temporal_Window_Query
#test case 1
print ("test case 1 Temporal_Window_Query")
results_tWindow = git_index1.t_window(25, 45) #Shoud return pardis_3 and
    pardis_4

if results_tWindow:
    print("Temporal trajectory overlap find successfully")
    print(results_tWindow)
else:
    print("Temporal trajectory overlap")
```

Test Case 1 Output:

```
test case 1 Temporal_Window_Query
Temporal trajectory overlap find successfully
[['pardis_3'], ['pardis_4']]
```

Test Case 2:

```
#test case 2
print ("test case 2 Temporal_Window_Query")
results_tWindow = git_index1.t_window(60, 70) #Shoud return nothing

if results_tWindow:
    print("Temporal trajectory overlap find successfully")
    print(results_tWindow)
else:
    print("Temporal trajectory overlap not find")
```

Test Case 2 Output:

```
test case 2 Temporal_Window_Query
Temporal trajectory overlap not find
```

4. **Spatial Window query**: This part needed more programming rather other parts. At first, we needed to compare the spatial extents. Therefore, we need to store these spatial extents inside another dictionary and compare them with bounding box. Again, we give t1, t2, t3 and t4 to the trajectory and this time we build a dictionary that the key values are spatial_extent and the values are the trajectory id. After this, we are going to give x1 = 10, y1=10 and x2=30, y2=30. We have this computation based on the query that we have:

Test Case 1:

```
#We need to find the range of the grid cells that are intersects with
    the bounding box
start_x = math.floor((min(x1, x2) - self.sf_xmin) / self.delta_x)
#print (start_x) -> 2
end_x = math.ceil((max(x1, x2) - self.sf_xmin) / self.delta_x)
#print (end_x) -> 6
start_y = math.floor((min(y1, y2) - self.sf_ymin) / self.delta_y)
#print (start_y) -> 2
end_y = math.ceil((max(y1, y2) - self.sf_ymin) / self.delta_y)
#print (end_y) -> 6
```

Based on this computation we have this range of grid cells that overlap with bounding box from **2 to 6**.

Based on the dictionary that we made with spatial_extent, we should get an output with grid cell on the range that we provided:

We have (2,3): [‘pardis_1’], (4,5): [‘pardis_2’]. So here is the output that we get based on spatial_extent and we have this as the ids. This at first shows the dictionary with spatial_extent as the key value and we have values as the id. Then we find the overlapping spatial_extent that is (2,3) and (4,5).

What we get, as the result is the pardis_1 and pardis_2 that is the ids of trajectories.

```
#test case 1 for sp_window query
#This part is for building a trajectory dictionary for sp_window() query so we can compare it with cells that are overlapping we also give
#new tgpairs for our code
print ("test case 1 sp_window query")
tgpairs1 = TimeGeometryPair(temporal_extent=(1, 2), spatial_extent=(2, 3)),
tgpairs2 = TimeGeometryPair(temporal_extent=(3, 14), spatial_extent=(4, 5)),
tgpairs3 = TimeGeometryPair(temporal_extent=(5, 6), spatial_extent=(30, 30)),
tgpairs4 = TimeGeometryPair(temporal_extent=(7, 8), spatial_extent=(40, 40)),

t1 = Trajectory(traj_id="pardis_1", tgpairs=tgpairs1)
t2 = Trajectory(traj_id="pardis_2", tgpairs=tgpairs2)
t3 = Trajectory(traj_id="pardis_3", tgpairs=tgpairs3)
t4 = Trajectory(traj_id="pardis_4", tgpairs=tgpairs4)
git_index3 = GIT(sf_xmin=0, sf_xmax=10, sf_ymin=0, sf_ymax=10, delta_x=5, delta_y=5)

git_index3.insert_trajectory(t1)
git_index3.insert_trajectory(t2)
git_index3.insert_trajectory(t3)
git_index3.insert_trajectory(t4)

#check for the trajectory build successfully
print(git_index3.trajectory)

dictionary_spatial_extent = git_index3.trajectory

result_sp_window = git_index3.sp_window(10, 10, 30, 30, dictionary_spatial_extent)

print (result_sp_window)
```

Test Case 1 Output:

```
test case 1 sp_window query
{(2, 3): ['pardis_1'], (4, 5): ['pardis_2'], (30, 30): ['pardis_3'], (40, 40): ['pardis_4']}
[(2, 3), (4, 5)]
['pardis_1', 'pardis_2']
```

Test Case 2:

```
#test case 2 for sp_window query
#This part is for building a trajectory dictionary for sp_window() query so we can compare it with cells that are overlapping we also give
#new tgpairs for our code
print ("test case 2 sp_window query")
tgpairs1 = TimeGeometryPair(temporal_extent=(1, 2), spatial_extent=(2, 3)),
tgpairs2 = TimeGeometryPair(temporal_extent=(3, 14), spatial_extent=(4, 5)),
tgpairs3 = TimeGeometryPair(temporal_extent=(5, 6), spatial_extent=(11, 15)),
tgpairs4 = TimeGeometryPair(temporal_extent=(7, 8), spatial_extent=(6, 8)),
t1 = Trajectory(traj_id="pardis_1", tgpairs=tgpairs1)
t2 = Trajectory(traj_id="pardis_2", tgpairs=tgpairs2)
t3 = Trajectory(traj_id="pardis_3", tgpairs=tgpairs3)
t4 = Trajectory(traj_id="pardis_4", tgpairs=tgpairs4)

git_index3 = GIT(sf_xmin=0, sf_xmax=10, sf_ymin=0, sf_ymax=10, delta_x=2, delta_y=2)

git_index3.insert_trajectory(t1)
git_index3.insert_trajectory(t2)
git_index3.insert_trajectory(t3)
git_index3.insert_trajectory(t4)

#check for the trajectory build successfully
print(git_index3.trajectory)

dictionary_spatial_extent = git_index3.trajectory

result_sp_window = git_index3.sp_window(10, 10, 30, 30, dictionary_spatial_extent)

print (result_sp_window)
```

This is the second test case that we are using for this query is the one that we got the range **5 to 15**

```
        #We need to find the range of the grid cells that are intersects with the bounding box
        start_x = math.floor((min(x1, x2) - self.sf_xmin) / self.delta_x)
        #print (start_x) -> 5
        end_x = math.ceil((max(x1, x2) - self.sf_xmin) / self.delta_x)
        #print (end_x) -> 15
        start_y = math.floor((min(y1, y2) - self.sf_ymin) / self.delta_y)
        #print (start_y) -> 5
        end_y = math.ceil((max(y1, y2) - self.sf_ymin) / self.delta_y)
        #print (end_y) -> 15
```

Test Case 2 Output:

What we get, as the result is the pardis_3 and pardis_4 that is the ids of trajectories.

```
test case 2 sp_window query
{(2, 3): ['pardis_1'], (4, 5): ['pardis_2'], (11, 15): ['pardis_3'], (6, 8):
    ['pardis_4']}
[(11, 15), (6, 8)]
['pardis_3', 'pardis_4']
```

5. Spatio-Temporal Window query:

This query is made by the two queries t-window and spatial_window. We go through the temporal_query and sp_window_query steps and we got the same results. However, in this part of the code we have two dictionaries one of them the key value is temporal and in the other one, the key value is spatial_extent. However, the values that are identities are the same. Therefore, we have this:

Test Case 1 Output:

This is for test case 1. That we have two dictionaries the first dictionary is (t1, t2) = [trajectory_id] for temporal_extent as the key, and the second dictionary is (g1,g2) = [trajectory_id] for spatial_extent as the key.

The line after the first dictionary is the key value of the grid dictionary that the temporal_extents are the key values.

The line after the second dictionary is the key value of the grid dictionary that the spatial_extents are the key values.

The results [[‘pardis_1’], [‘pardis_2’],] are the times that overlap.

The result [[‘pardis_1’] is the bounding box that overlaps. The final result is the pardis_1 that we can see here.

```
test case 1 st_window query
{(0, 10): ['pardis_1'], (5, 15): ['pardis_2'], (20, 30): ['pardis_3'], (40, 50
    ): ['pardis_4']}
[(0, 10), (5, 15), (20, 30), (40, 50)]
{(1, 1): ['pardis_1'], (4, 5): ['pardis_2'], (30, 30): ['pardis_3'], (40, 40):
    ['pardis_4']}
[(1, 1), (4, 5), (30, 30), (40, 40)]
[['pardis_1'], ['pardis_2']]
['pardis_1']
```

Test Case 2 Output:

```
test case 2 st_window query
{(0, 10): ['pardis_1'], (5, 15): ['pardis_2'], (20, 30): ['pardis_3'], (40, 50
    ): ['pardis_4']}
[(0, 10), (5, 15), (20, 30), (40, 50)]
{(1, 1): ['pardis_1'], (4, 5): ['pardis_2'], (30, 30): ['pardis_3'], (40, 40):
    ['pardis_4']}
[(1, 1), (4, 5), (30, 30), (40, 40)]
[['pardis_3'], ['pardis_4']]
['pardis_3']
```