package com.twitter.ann.hnsw;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.Iterator;

import java.util.List;

import java.util.PriorityQueue;

/\*\*

\* Container for items with their distance.

\*

\* @param <U> Type of origin/reference element.

\* @param <T> Type of element that the queue will hold

\*/

public class DistancedItemQueue<U, T> implements Iterable<DistancedItem<T>> {

private final U origin;

private final DistanceFunction<U, T> distFn;

private final PriorityQueue<DistancedItem<T>> queue;

private final boolean minQueue;

/\*\*

\* Creates ontainer for items with their distances.

\*

\* @param origin Origin (reference) point

\* @param initial Initial list of elements to add in the structure

\* @param minQueue True for min queue, False for max queue

\* @param distFn Distance function

\*/

public DistancedItemQueue(

U origin,

List<T> initial,

boolean minQueue,

DistanceFunction<U, T> distFn

) {

this.origin = origin;

this.distFn = distFn;

this.minQueue = minQueue;

final Comparator<DistancedItem<T>> cmp;

if (minQueue) {

cmp = (o1, o2) -> Float.compare(o1.getDistance(), o2.getDistance());

} else {

cmp = (o1, o2) -> Float.compare(o2.getDistance(), o1.getDistance());

}

this.queue = new PriorityQueue<>(cmp);

enqueueAll(initial);

new DistancedItemQueue<>(origin, distFn, queue, minQueue);

}

private DistancedItemQueue(

U origin,

DistanceFunction<U, T> distFn,

PriorityQueue<DistancedItem<T>> queue,

boolean minQueue

) {

this.origin = origin;

this.distFn = distFn;

this.queue = queue;

this.minQueue = minQueue;

}

/\*\*

\* Enqueues all the items into the queue.

\*/

public void enqueueAll(List<T> list) {

for (T t : list) {

enqueue(t);

}

}

/\*\*

\* Return if queue is non empty or not

\*

\* @return true if queue is not empty else false

\*/

public boolean nonEmpty() {

return !queue.isEmpty();

}

/\*\*

\* Return root of the queue

\*

\* @return root of the queue i.e min/max element depending upon min-max queue

\*/

public DistancedItem<T> peek() {

return queue.peek();

}

/\*\*

\* Dequeue root of the queue.

\*

\* @return remove and return root of the queue i.e min/max element depending upon min-max queue

\*/

public DistancedItem<T> dequeue() {

return queue.poll();

}

/\*\*

\* Dequeue all the elements from queueu with ordering mantained

\*

\* @return remove all the elements in the order of the queue i.e min/max queue.

\*/

public List<DistancedItem<T>> dequeueAll() {

final List<DistancedItem<T>> list = new ArrayList<>(queue.size());

while (!queue.isEmpty()) {

list.add(queue.poll());

}

return list;

}

/\*\*

\* Convert queue to list

\*

\* @return list of elements of queue with distance and without any specific ordering

\*/

public List<DistancedItem<T>> toList() {

return new ArrayList<>(queue);

}

/\*\*

\* Convert queue to list

\*

\* @return list of elements of queue without any specific ordering

\*/

List<T> toListWithItem() {

List<T> list = new ArrayList<>(queue.size());

Iterator<DistancedItem<T>> itr = iterator();

while (itr.hasNext()) {

list.add(itr.next().getItem());

}

return list;

}

/\*\*

\* Enqueue an item into the queue

\*/

public void enqueue(T item) {

queue.add(new DistancedItem<>(item, distFn.distance(origin, item)));

}

/\*\*

\* Enqueue an item into the queue with its distance.

\*/

public void enqueue(T item, float distance) {

queue.add(new DistancedItem<>(item, distance));

}

/\*\*

\* Size

\*

\* @return size of the queue

\*/

public int size() {

return queue.size();

}

/\*\*

\* Is Min queue

\*

\* @return true if min queue else false

\*/

public boolean isMinQueue() {

return minQueue;

}

/\*\*

\* Returns origin (base element) of the queue

\*

\* @return origin of the queue

\*/

public U getOrigin() {

return origin;

}

/\*\*

\* Return a new queue with ordering reversed.

\*/

public DistancedItemQueue<U, T> reverse() {

final PriorityQueue<DistancedItem<T>> rqueue =

new PriorityQueue<>(queue.comparator().reversed());

if (queue.isEmpty()) {

return new DistancedItemQueue<>(origin, distFn, rqueue, !isMinQueue());

}

final Iterator<DistancedItem<T>> itr = iterator();

while (itr.hasNext()) {

rqueue.add(itr.next());

}

return new DistancedItemQueue<>(origin, distFn, rqueue, !isMinQueue());

}

@Override

public Iterator<DistancedItem<T>> iterator() {

return queue.iterator();

}

}