package com.twitter.ann.hnsw;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Objects;

import java.util.Optional;

import java.util.Random;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.atomic.AtomicReference;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReadWriteLock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

import java.util.function.Function;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.ImmutableList;

import org.apache.thrift.TException;

import com.twitter.ann.common.IndexOutputFile;

import com.twitter.ann.common.thriftjava.HnswInternalIndexMetadata;

import com.twitter.bijection.Injection;

import com.twitter.logging.Logger;

import com.twitter.mediaservices.commons.codec.ArrayByteBufferCodec;

import com.twitter.search.common.file.AbstractFile;

/\*\*

\* Typed multithreaded HNSW implementation supporting creation/querying of approximate nearest neighbour

\* Paper: https://arxiv.org/pdf/1603.09320.pdf

\* Multithreading impl based on NMSLIB version : https://github.com/nmslib/hnsw/blob/master/hnswlib/hnswalg.h

\*

\* @param <T> The type of items inserted / searched in the HNSW index.

\* @param <Q> The type of KNN query.

\*/

public class HnswIndex<T, Q> {

private static final Logger LOG = Logger.get(HnswIndex.class);

private static final String METADATA\_FILE\_NAME = "hnsw\_internal\_metadata";

private static final String GRAPH\_FILE\_NAME = "hnsw\_internal\_graph";

private static final int MAP\_SIZE\_FACTOR = 5;

private final DistanceFunction<T, T> distFnIndex;

private final DistanceFunction<Q, T> distFnQuery;

private final int efConstruction;

private final int maxM;

private final int maxM0;

private final double levelMultiplier;

private final AtomicReference<HnswMeta<T>> graphMeta = new AtomicReference<>();

private final Map<HnswNode<T>, ImmutableList<T>> graph;

// To take lock on vertex level

private final ConcurrentHashMap<T, ReadWriteLock> locks;

// To take lock on whole graph only if vertex addition is on layer above the current maxLevel

private final ReentrantLock globalLock;

private final Function<T, ReadWriteLock> lockProvider;

private final RandomProvider randomProvider;

// Probability of reevaluating connections of an element in the neighborhood during an update

// Can be used as a knob to adjust update\_speed/search\_speed tradeoff.

private final float updateNeighborProbability;

/\*\*

\* Creates instance of hnsw index.

\*

\* @param distFnIndex Any distance metric/non metric that specifies similarity between two items for indexing.

\* @param distFnQuery Any distance metric/non metric that specifies similarity between item for which nearest neighbours queried for and already indexed item.

\* @param efConstruction Provide speed vs index quality tradeoff, higher the value better the quality and higher the time to create index.

\* Valid range of efConstruction can be anywhere between 1 and tens of thousand. Typically, it should be set so that a search of M

\* neighbors with ef=efConstruction should end in recall>0.95.

\* @param maxM Maximum connections per layer except 0th level.

\* Optimal values between 5-48.

\* Smaller M generally produces better result for lower recalls and/ or lower dimensional data,

\* while bigger M is better for high recall and/ or high dimensional, data on the expense of more memory/disk usage

\* @param expectedElements Approximate number of elements to be indexed

\*/

protected HnswIndex(

DistanceFunction<T, T> distFnIndex,

DistanceFunction<Q, T> distFnQuery,

int efConstruction,

int maxM,

int expectedElements,

RandomProvider randomProvider

) {

this(distFnIndex,

distFnQuery,

efConstruction,

maxM,

expectedElements,

new HnswMeta<>(-1, Optional.empty()),

new ConcurrentHashMap<>(MAP\_SIZE\_FACTOR \* expectedElements),

randomProvider

);

}

private HnswIndex(

DistanceFunction<T, T> distFnIndex,

DistanceFunction<Q, T> distFnQuery,

int efConstruction,

int maxM,

int expectedElements,

HnswMeta<T> graphMeta,

Map<HnswNode<T>, ImmutableList<T>> graph,

RandomProvider randomProvider

) {

this.distFnIndex = distFnIndex;

this.distFnQuery = distFnQuery;

this.efConstruction = efConstruction;

this.maxM = maxM;

this.maxM0 = 2 \* maxM;

this.levelMultiplier = 1.0 / Math.log(1.0 \* maxM);

this.graphMeta.set(graphMeta);

this.graph = graph;

this.locks = new ConcurrentHashMap<>(MAP\_SIZE\_FACTOR \* expectedElements);

this.globalLock = new ReentrantLock();

this.lockProvider = key -> new ReentrantReadWriteLock();

this.randomProvider = randomProvider;

this.updateNeighborProbability = 1.0f;

}

/\*\*

\* wireConnectionForAllLayers finds connections for a new element and creates bi-direction links.

\* The method assumes using a reentrant lock to link list reads.

\*

\* @param entryPoint the global entry point

\* @param item the item for which the connections are found

\* @param itemLevel the level of the added item (maximum layer in which we wire the connections)

\* @param maxLayer the level of the entry point

\*/

private void wireConnectionForAllLayers(final T entryPoint, final T item, final int itemLevel,

final int maxLayer, final boolean isUpdate) {

T curObj = entryPoint;

if (itemLevel < maxLayer) {

curObj = bestEntryPointUntilLayer(curObj, item, maxLayer, itemLevel, distFnIndex);

}

for (int level = Math.min(itemLevel, maxLayer); level >= 0; level--) {

final DistancedItemQueue<T, T> candidates =

searchLayerForCandidates(item, curObj, efConstruction, level, distFnIndex, isUpdate);

curObj = mutuallyConnectNewElement(item, candidates, level, isUpdate);

}

}

/\*\*

\* Insert the item into HNSW index.

\*/

public void insert(final T item) throws IllegalDuplicateInsertException {

final Lock itemLock = locks.computeIfAbsent(item, lockProvider).writeLock();

itemLock.lock();

try {

final HnswMeta<T> metadata = graphMeta.get();

// If the graph already have the item, should not re-insert it again

// Need to check entry point in case we reinsert first item where is are no graph

// but only a entry point

if (graph.containsKey(HnswNode.from(0, item))

|| (metadata.getEntryPoint().isPresent()

&& Objects.equals(metadata.getEntryPoint().get(), item))) {

throw new IllegalDuplicateInsertException(

"Duplicate insertion is not supported: " + item);

}

final int curLevel = getRandomLevel();

Optional<T> entryPoint = metadata.getEntryPoint();

// The global lock prevents two threads from making changes to the entry point. This lock

// should get taken very infrequently. Something like log-base-levelMultiplier(num items)

// For a full explanation of locking see this document: http://go/hnsw-locking

int maxLevelCopy = metadata.getMaxLevel();

if (curLevel > maxLevelCopy) {

globalLock.lock();

// Re initialize the entryPoint and maxLevel in case these are changed by any other thread

// No need to check the condition again since,

// it is already checked at the end before updating entry point struct

// No need to unlock for optimization and keeping as is if condition fails since threads

// will not be entering this section a lot.

final HnswMeta<T> temp = graphMeta.get();

entryPoint = temp.getEntryPoint();

maxLevelCopy = temp.getMaxLevel();

}

if (entryPoint.isPresent()) {

wireConnectionForAllLayers(entryPoint.get(), item, curLevel, maxLevelCopy, false);

}

if (curLevel > maxLevelCopy) {

Preconditions.checkState(globalLock.isHeldByCurrentThread(),

"Global lock not held before updating entry point");

graphMeta.set(new HnswMeta<>(curLevel, Optional.of(item)));

}

} finally {

if (globalLock.isHeldByCurrentThread()) {

globalLock.unlock();

}

itemLock.unlock();

}

}

/\*\*

\* set connections of an element with synchronization

\* The only other place that should have the lock for writing is during

\* the element insertion

\*/

private void setConnectionList(final T item, int layer, List<T> connections) {

final Lock candidateLock = locks.computeIfAbsent(item, lockProvider).writeLock();

candidateLock.lock();

try {

graph.put(

HnswNode.from(layer, item),

ImmutableList.copyOf(connections)

);

} finally {

candidateLock.unlock();

}

}

/\*\*

\* Reinsert the item into HNSW index.

\* This method updates the links of an element assuming

\* the element's distance function is changed externally (e.g. by updating the features)

\*/

public void reInsert(final T item) {

final HnswMeta<T> metadata = graphMeta.get();

Optional<T> entryPoint = metadata.getEntryPoint();

Preconditions.checkState(entryPoint.isPresent(),

"Update cannot be performed if entry point is not present");

// This is a check for the single element case

if (entryPoint.get().equals(item) && graph.isEmpty()) {

return;

}

Preconditions.checkState(graph.containsKey(HnswNode.from(0, item)),

"Graph does not contain the item to be updated at level 0");

int curLevel = 0;

int maxLevelCopy = metadata.getMaxLevel();

for (int layer = maxLevelCopy; layer >= 0; layer--) {

if (graph.containsKey(HnswNode.from(layer, item))) {

curLevel = layer;

break;

}

}

// Updating the links of the elements from the 1-hop radius of the updated element

for (int layer = 0; layer <= curLevel; layer++) {

// Filling the element sets for candidates and updated elements

final HashSet<T> setCand = new HashSet<T>();

final HashSet<T> setNeigh = new HashSet<T>();

final List<T> listOneHop = getConnectionListForRead(item, layer);

if (listOneHop.isEmpty()) {

LOG.debug("No links for the updated element. Empty dataset?");

continue;

}

setCand.add(item);

for (T elOneHop : listOneHop) {

setCand.add(elOneHop);

if (randomProvider.get().nextFloat() > updateNeighborProbability) {

continue;

}

setNeigh.add(elOneHop);

final List<T> listTwoHop = getConnectionListForRead(elOneHop, layer);

if (listTwoHop.isEmpty()) {

LOG.debug("No links for the updated element. Empty dataset?");

}

for (T oneHopEl : listTwoHop) {

setCand.add(oneHopEl);

}

}

// No need to update the item itself, so remove it

setNeigh.remove(item);

// Updating the link lists of elements from setNeigh:

for (T neigh : setNeigh) {

final HashSet<T> setCopy = new HashSet<T>(setCand);

setCopy.remove(neigh);

int keepElementsNum = Math.min(efConstruction, setCopy.size());

final DistancedItemQueue<T, T> candidates = new DistancedItemQueue<>(

neigh,

ImmutableList.of(),

false,

distFnIndex

);

for (T cand : setCopy) {

final float distance = distFnIndex.distance(neigh, cand);

if (candidates.size() < keepElementsNum) {

candidates.enqueue(cand, distance);

} else {

if (distance < candidates.peek().getDistance()) {

candidates.dequeue();

candidates.enqueue(cand, distance);

}

}

}

final ImmutableList<T> neighbours = selectNearestNeighboursByHeuristic(

candidates,

layer == 0 ? maxM0 : maxM

);

final List<T> temp = getConnectionListForRead(neigh, layer);

if (temp.isEmpty()) {

LOG.debug("existing linkslist is empty. Corrupt index");

}

if (neighbours.isEmpty()) {

LOG.debug("predicted linkslist is empty. Corrupt index");

}

setConnectionList(neigh, layer, neighbours);

}

}

wireConnectionForAllLayers(metadata.getEntryPoint().get(), item, curLevel, maxLevelCopy, true);

}

/\*\*

\* This method can be used to get the graph statistics, specifically

\* it prints the histogram of inbound connections for each element.

\*/

private String getStats() {

int histogramMaxBins = 50;

int[] histogram = new int[histogramMaxBins];

HashMap<T, Integer> mmap = new HashMap<T, Integer>();

for (HnswNode<T> key : graph.keySet()) {

if (key.level == 0) {

List<T> linkList = getConnectionListForRead(key.item, key.level);

for (T node : linkList) {

int a = mmap.computeIfAbsent(node, k -> 0);

mmap.put(node, a + 1);

}

}

}

for (T key : mmap.keySet()) {

int ind = mmap.get(key) < histogramMaxBins - 1 ? mmap.get(key) : histogramMaxBins - 1;

histogram[ind]++;

}

int minNonZeroIndex;

for (minNonZeroIndex = histogramMaxBins - 1; minNonZeroIndex >= 0; minNonZeroIndex--) {

if (histogram[minNonZeroIndex] > 0) {

break;

}

}

String output = "";

for (int i = 0; i <= minNonZeroIndex; i++) {

output += "" + i + "\t" + histogram[i] / (0.01f \* mmap.keySet().size()) + "\n";

}

return output;

}

private int getRandomLevel() {

return (int) (-Math.log(randomProvider.get().nextDouble()) \* levelMultiplier);

}

/\*\*

\* Note that to avoid deadlocks it is important that this method is called after all the searches

\* of the graph have completed. If you take a lock on any items discovered in the graph after

\* this, you may get stuck waiting on a thread that is waiting for item to be fully inserted.

\* <p>

\* Note: when using concurrent writers we can miss connections that we would otherwise get.

\* This will reduce the recall.

\* <p>

\* For a full explanation of locking see this document: http://go/hnsw-locking

\* The method returns the closest nearest neighbor (can be used as an enter point)

\*/

private T mutuallyConnectNewElement(

final T item,

final DistancedItemQueue<T, T> candidates, // Max queue

final int level,

final boolean isUpdate

) {

// Using maxM here. Its implementation is ambiguous in HNSW paper,

// so using the way it is getting used in Hnsw lib.

final ImmutableList<T> neighbours = selectNearestNeighboursByHeuristic(candidates, maxM);

setConnectionList(item, level, neighbours);

final int M = level == 0 ? maxM0 : maxM;

for (T nn : neighbours) {

if (nn.equals(item)) {

continue;

}

final Lock curLock = locks.computeIfAbsent(nn, lockProvider).writeLock();

curLock.lock();

try {

final HnswNode<T> key = HnswNode.from(level, nn);

final ImmutableList<T> connections = graph.getOrDefault(key, ImmutableList.of());

final boolean isItemAlreadyPresent =

isUpdate && connections.indexOf(item) != -1 ? true : false;

// If `item` is already present in the neighboring connections,

// then no need to modify any connections or run the search heuristics.

if (isItemAlreadyPresent) {

continue;

}

final ImmutableList<T> updatedConnections;

if (connections.size() < M) {

final List<T> temp = new ArrayList<>(connections);

temp.add(item);

updatedConnections = ImmutableList.copyOf(temp.iterator());

} else {

// Max Queue

final DistancedItemQueue<T, T> queue = new DistancedItemQueue<>(

nn,

connections,

false,

distFnIndex

);

queue.enqueue(item);

updatedConnections = selectNearestNeighboursByHeuristic(queue, M);

}

if (updatedConnections.isEmpty()) {

LOG.debug("Internal error: predicted linkslist is empty");

}

graph.put(key, updatedConnections);

} finally {

curLock.unlock();

}

}

return neighbours.get(0);

}

/\*

\* bestEntryPointUntilLayer starts the graph search for item from the entry point

\* until the searches reaches the selectedLayer layer.

\* @return a point from selectedLayer layer, was the closest on the (selectedLayer+1) layer

\*/

private <K> T bestEntryPointUntilLayer(

final T entryPoint,

final K item,

int maxLayer,

int selectedLayer,

DistanceFunction<K, T> distFn

) {

T curObj = entryPoint;

if (selectedLayer < maxLayer) {

float curDist = distFn.distance(item, curObj);

for (int level = maxLayer; level > selectedLayer; level--) {

boolean changed = true;

while (changed) {

changed = false;

final List<T> list = getConnectionListForRead(curObj, level);

for (T nn : list) {

final float tempDist = distFn.distance(item, nn);

if (tempDist < curDist) {

curDist = tempDist;

curObj = nn;

changed = true;

}

}

}

}

}

return curObj;

}

@VisibleForTesting

protected ImmutableList<T> selectNearestNeighboursByHeuristic(

final DistancedItemQueue<T, T> candidates, // Max queue

final int maxConnections

) {

Preconditions.checkState(!candidates.isMinQueue(),

"candidates in selectNearestNeighboursByHeuristic should be a max queue");

final T baseElement = candidates.getOrigin();

if (candidates.size() <= maxConnections) {

List<T> list = candidates.toListWithItem();

list.remove(baseElement);

return ImmutableList.copyOf(list);

} else {

final List<T> resSet = new ArrayList<>(maxConnections);

// Min queue for closest elements first

final DistancedItemQueue<T, T> minQueue = candidates.reverse();

while (minQueue.nonEmpty()) {

if (resSet.size() >= maxConnections) {

break;

}

final DistancedItem<T> candidate = minQueue.dequeue();

// We do not want to creates loops:

// While heuristic is used only for creating the links

if (candidate.getItem().equals(baseElement)) {

continue;

}

boolean toInclude = true;

for (T e : resSet) {

// Do not include candidate if the distance from candidate to any of existing item in

// resSet is closer to the distance from the candidate to the item. By doing this, the

// connection of graph will be more diverse, and in case of highly clustered data set,

// connections will be made between clusters instead of all being in the same cluster.

final float dist = distFnIndex.distance(e, candidate.getItem());

if (dist < candidate.getDistance()) {

toInclude = false;

break;

}

}

if (toInclude) {

resSet.add(candidate.getItem());

}

}

return ImmutableList.copyOf(resSet);

}

}

/\*\*

\* Search the index for the neighbours.

\*

\* @param query Query

\* @param numOfNeighbours Number of neighbours to search for.

\* @param ef This param controls the accuracy of the search.

\* Bigger the ef better the accuracy on the expense of latency.

\* Keep it atleast number of neighbours to find.

\* @return Neighbours

\*/

public List<DistancedItem<T>> searchKnn(final Q query, final int numOfNeighbours, final int ef) {

final HnswMeta<T> metadata = graphMeta.get();

if (metadata.getEntryPoint().isPresent()) {

T entryPoint = bestEntryPointUntilLayer(metadata.getEntryPoint().get(),

query, metadata.getMaxLevel(), 0, distFnQuery);

// Get the actual neighbours from 0th layer

final List<DistancedItem<T>> neighbours =

searchLayerForCandidates(query, entryPoint, Math.max(ef, numOfNeighbours),

0, distFnQuery, false).dequeueAll();

Collections.reverse(neighbours);

return neighbours.size() > numOfNeighbours

? neighbours.subList(0, numOfNeighbours) : neighbours;

} else {

return Collections.emptyList();

}

}

// This method is currently not used

// It is needed for debugging purposes only

private void checkIntegrity(String message) {

final HnswMeta<T> metadata = graphMeta.get();

for (HnswNode<T> node : graph.keySet()) {

List<T> linkList = graph.get(node);

for (T el : linkList) {

if (el.equals(node.item)) {

LOG.debug(message);

throw new RuntimeException("integrity check failed");

}

}

}

}

private <K> DistancedItemQueue<K, T> searchLayerForCandidates(

final K item,

final T entryPoint,

final int ef,

final int level,

final DistanceFunction<K, T> distFn,

boolean isUpdate

) {

// Min queue

final DistancedItemQueue<K, T> cQueue = new DistancedItemQueue<>(

item,

Collections.singletonList(entryPoint),

true,

distFn

);

// Max Queue

final DistancedItemQueue<K, T> wQueue = cQueue.reverse();

final Set<T> visited = new HashSet<>();

float lowerBoundDistance = wQueue.peek().getDistance();

visited.add(entryPoint);

while (cQueue.nonEmpty()) {

final DistancedItem<T> candidate = cQueue.peek();

if (candidate.getDistance() > lowerBoundDistance) {

break;

}

cQueue.dequeue();

final List<T> list = getConnectionListForRead(candidate.getItem(), level);

for (T nn : list) {

if (!visited.contains(nn)) {

visited.add(nn);

final float distance = distFn.distance(item, nn);

if (wQueue.size() < ef || distance < wQueue.peek().getDistance()) {

cQueue.enqueue(nn, distance);

if (isUpdate && item.equals(nn)) {

continue;

}

wQueue.enqueue(nn, distance);

if (wQueue.size() > ef) {

wQueue.dequeue();

}

lowerBoundDistance = wQueue.peek().getDistance();

}

}

}

}

return wQueue;

}

/\*\*

\* Serialize hnsw index

\*/

public void toDirectory(IndexOutputFile indexOutputFile, Injection<T, byte[]> injection)

throws IOException, TException {

final int totalGraphEntries = HnswIndexIOUtil.saveHnswGraphEntries(

graph,

indexOutputFile.createFile(GRAPH\_FILE\_NAME).getOutputStream(),

injection);

HnswIndexIOUtil.saveMetadata(

graphMeta.get(),

efConstruction,

maxM,

totalGraphEntries,

injection,

indexOutputFile.createFile(METADATA\_FILE\_NAME).getOutputStream());

}

/\*\*

\* Load hnsw index

\*/

public static <T, Q> HnswIndex<T, Q> loadHnswIndex(

DistanceFunction<T, T> distFnIndex,

DistanceFunction<Q, T> distFnQuery,

AbstractFile directory,

Injection<T, byte[]> injection,

RandomProvider randomProvider) throws IOException, TException {

final AbstractFile graphFile = directory.getChild(GRAPH\_FILE\_NAME);

final AbstractFile metadataFile = directory.getChild(METADATA\_FILE\_NAME);

final HnswInternalIndexMetadata metadata = HnswIndexIOUtil.loadMetadata(metadataFile);

final Map<HnswNode<T>, ImmutableList<T>> graph =

HnswIndexIOUtil.loadHnswGraph(graphFile, injection, metadata.numElements);

final ByteBuffer entryPointBB = metadata.entryPoint;

final HnswMeta<T> graphMeta = new HnswMeta<>(

metadata.maxLevel,

entryPointBB == null ? Optional.empty()

: Optional.of(injection.invert(ArrayByteBufferCodec.decode(entryPointBB)).get())

);

return new HnswIndex<>(

distFnIndex,

distFnQuery,

metadata.efConstruction,

metadata.maxM,

metadata.numElements,

graphMeta,

graph,

randomProvider

);

}

private List<T> getConnectionListForRead(T node, int level) {

final Lock curLock = locks.computeIfAbsent(node, lockProvider).readLock();

curLock.lock();

final List<T> list;

try {

list = graph

.getOrDefault(HnswNode.from(level, node), ImmutableList.of());

} finally {

curLock.unlock();

}

return list;

}

@VisibleForTesting

AtomicReference<HnswMeta<T>> getGraphMeta() {

return graphMeta;

}

@VisibleForTesting

Map<T, ReadWriteLock> getLocks() {

return locks;

}

@VisibleForTesting

Map<HnswNode<T>, ImmutableList<T>> getGraph() {

return graph;

}

public interface RandomProvider {

/\*\*

\* RandomProvider interface made public for scala 2.12 compat

\*/

Random get();

}

}