Algorithm 1: INSERT

Input: multilayer graph *hnsw*, new element q, normalization factor for level generation mL

Output: update *hnsw* inserting element q without connections

$l \leftarrow \lfloor -\ln(\text{uniform}(0..1)) \cdot mL \rfloor$ // new element's level
$L \leftarrow$ top layer of *hnsw*
$ep \leftarrow$ enter point of *hnsw*
for $lc \leftarrow 0$ **to** l **do**
    add q to layer lc in *hnsw*
// case of the first element inserted
if *enter point of hnsw = -1* **then**
    set enter point of *hnsw* to q
    set top layer of *hnsw* to l
    return;
// when inserting an element with a higher maximum layer
if $l > L$ **then**
    set enter point of *hnsw* to q
    set top layer of *hnsw* to l

Algorithm 2: INSERT-LIST

Input: multilayer graph *hnsw*, list of new elements V, maximum number of connections for each element per layer *M_max*, RNN parameters $L, S, T1, T2$, fingerprints *fps*

Output: update *hnsw* inserting elements V

for *each element v in V* **do**
    insert(v) // insert the element without connections
for $lc \leftarrow$ *top layer* **to** *0* **do**
    ntotal $\leftarrow$ number of elements in layer lc
    // naive connection method
    if *ntotal* < 30 **then**
        connect each element to all others except itself
    else
        $rnn \leftarrow$ RNN-Descent$(L, S, T1, T2, fps)$
        copy the results from rnn to *hnsw* at the layer lc
        delete rnn

Algorithm 3: K-NN-SEARCH

Input: multilayer graph *hnsw*, query element q, number of nearest neighbors to return K, size of the dynamic candidate list ef in *hnsw*

Output: K nearest elements to q

```
W ← ∅ // set for the current nearest elements
ep ← get enter point for hnsw
L ← level of ep // top layer for hnsw
for l ← L to 1 do
    W ← SEARCH-LAYER(q, ep, ef, l)
    ep ← get nearest element from W to q
W ← SEARCH-LAYER(q, ep, ef, l = 0)
return K nearest elements from W to q
```

Algorithm 4: SEARCH-LAYER

Input: multilayer graph *hnsw*, query element q, enter points *ep*, number of nearest to q elements to return ef, layer number l

Output: ef closest neighbors to q in *hnsw*

```
v ← ep // set of visited elements
C ← ep // set of candidates
W ← ep // dynamic list of found nearest neighbors
while |C| > 0 do
    c ← extract nearest element from C to q
    f ← get furthest element from W to q
    if distance(c, q) > distance(f, q) then
        break // all elements in W are evaluated
    for each e in neighborhood(c) at layer l do
        // update C and W
        if e ∉ v then
            v ← v ∪ e
            f ← get furthest element from W to q
            if distance(e, q) < distance(f, q) or |W| < ef then
                C ← C ∪ e
                W ← W ∪ e
                if |W| > ef then
                    remove furthest element from W to q
return W
```

Algorithm 5: RNN-Descent(L, S, T1, T2, fps)

Input: $L, S, T1, T2 \in Z$, fingerprints *fps*

Output: graph $G = (V, E)$

$G \leftarrow$ random graph with *fps* and with a max of S neighbors per node
initialize all flags to "new"
for $t1 = 1, \ldots, T1$ **do**
    for $t2 = 1, \ldots, T2$ **do**
        `UpdateNeighbors`(G)
    if $t1 \neq T1$ **then**
        `AddReverseEdges`(G, L)
return G

Algorithm 6: UpdateNeighbors(G)

Input: graph $G = (V, E)$, vertex $u \in V$

for *all* $u \in V$ **do**
    $U \leftarrow \{v \mid (v, u) \in E\}$
    sort $v \in U$ in ascending order of distance(u, v)
    $U' \leftarrow \emptyset$
    for *all* $v \in U$ **do**
        $f \leftarrow$ true
        for *all* $w \in U'$ **do**
            if *both flags of* v *and* w *are "old"* **then**
                continue
            if $\mathit{distance}(u, w) \geq \mathit{distance}(u, v)$ **then**
                $f \leftarrow$ false
                $E \leftarrow E \setminus \{(u, v)\} \cup \{(u, w)\}$
                break
        if f **then**
            $U' \leftarrow U' \cup \{v\}$
    set the flag "old" for all vertices in U'

Algorithm 7: AddReverseEdges(G, L)

Input: graph $G = (V, E)$, $L \in Z$

$E \leftarrow E \cup \{(u, v) \mid (v, u) \in E\}$

set flags of new neighbors to "new"

for *all* $v \in V$ **do**

  $E_u \leftarrow \{(v, u) \mid (u, v) \in E\}$

  remove top-L shortest edges from E_u

  $E \leftarrow E \setminus E_u$

for *all* $v \in V$ **do**

  $E_u \leftarrow \{(u, v) \mid (u, v) \in E\}$

  remove top-L shortest edges from E_u

  $E \leftarrow E \setminus E_u$

User guide:

To run the program, we need to install the Pybind module (pip install pybind) and RDKit (pip install rdkit).

The parameters that can be modified are L (the maximum number of neighbors for each node), mL (the normalization factor for level generation; the optimum value is 1/log(L)), S (the initial number of neighbors per node after random generation), T1 (the number of iterations for the outer loop in the RNNDescent algorithm) and T2 (the number of iterations for the inner loop in the RNNDescent algorithm). After a few tests, we found that L=25, S=10, T1=5 and T2=5 give very good results in terms of precision, while maintaining a good insertion time. Increasing these parameters allows us to be more precise, but it increases insertion time.

point represents the point whose N nearest neighbors are sought.

num_threads represents the number of threads used during insertion.

output_file represents the file in which the fingerprints will be saved as a sequence of bits.