Skip List and HNSW

Big Data Management and Governance

Prof. Giovanni Simonini, Giovanni Malaguti

University of Modena and Reggio Emilia

November 13, 2025

Details of the Lab (1)

- We will implement Skip List and HNSW data structures
- Simple implementations, no optimization and we won't cover all the details
- Goal: to obtain two new data structures that support creation, insert and some kind of search operations
- At home, attempt to implement other operations (e.g., key-value semantics, different distance functions, update, delete) and identify necessary code changes

Details of the Lab (2)

- Clone the repository `https://github.com/Stravanni/bdm.git`
- Inside the cloned repository, navigate to "bdm/lab/skiplist-hnsw"
- There are three files: skip_list.py, hnsw.py, and utils.py
- We will implement algorithms and data structures inside the first two files from scratch. In utils.py there are helper functions for visualization and experiments

Details of the Lab (3)

Open a shell and move to the lab folder. There create the virtual environment:

```
(skiplist-hnsw)$ python -m venv .venv
```

Activate the environment:

```
(skiplist-hnsw)$ source .venv/bin/activate
```

Install the required packages:

```
(skiplist-hnsw)$ pip install -r requirements.txt
```

Skip Lists

SkipList: Definition

- Linked List + Binary-search tree/sorted arrays
- Each node has 1+ layers, and each layer might be connected to different nodes
- Layers are assigned following a geometric distribution
- We'll use a simple implementation of Skip Lists, but you can extend it to include key-value pairs inside nodes

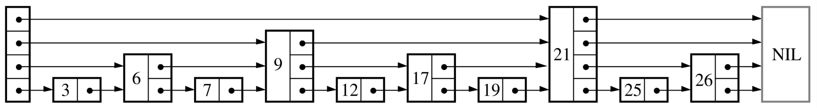

Figure 1: Example of Skip List (from [2])

Search

Algorithm 1 Search

Require: Skip list S with header $head$, value to search q
Ensure: Node with q if present, NIL otherwise

$current \leftarrow head$
$level \leftarrow$ max level of the skip list
for $i = level$ **to** 0 **do**
    while $current.forward[i] \neq NIL$
        and $current.forward[i].item < q$ **do**
        $current \leftarrow current.forward[i]$
    end while
end for
$current \leftarrow current.forward[0]$
if $current \neq NIL$ **and** $current.item = q$ **then**
    return $current$
else
    return NIL
end if

Random Level Generation

Algorithm 2 Random Level Generation

Require: Probability p, max level $maxLevel$
Ensure: Random level for the new node

1: $level \leftarrow 0$
2: **while** random() $< p$ **and** $level < maxLevel$ **do**
3:     $level \leftarrow level + 1$
4: **end while**
5: **return** $level$

Insertion

Algorithm 3 Insert

Require: Skip list S with header *head*, value q
Ensure: Skip list updated with the new node

```
update ← array of pointers with size maxLevel + 1
current ← head
level ← current max level
for i = level to 0 do
    while current.forward[i] ≠ NIL and current.forward[i].item < q do
        current ← current.forward[i]
    end while
    update[i] ← current
end for
current ← current.forward[0]
if current ≠ NIL and current.item = q then
    return
end if
newLevel ← RandomLevel()
newNode ← createNode(q, newLevel)
if newLevel > currentLevel then
    for i = currentLevel + 1 to newLevel + 1 do
        update[i] ← head
    end for
    currentLevel ← newLevel
end if
for i = 0 to newLevel + 1 do
    newNode.forward[i] ← update[i].forward[i]
    update[i].forward[i] ← newNode
end for
```

HNSW

From Skip Lists to HNSW

- HNSW[1] is a generalization of Skip List data structure
- Each node can have ≥ 1 neighbors at each level, not just one
- The neighbors are a proximity graph of the node

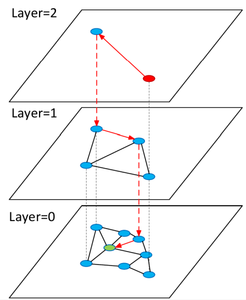

Figure 2: An HNSW graph

Search

Algorithm 4 Search

Require: multilayer graph *hnws*, query element q, number of nearest neighbors to return K, size of the dynamic candidate list *efSearch*

Ensure: K nearest neighbors to q

$W \leftarrow \emptyset$
$ep \leftarrow$ enter point for *hnsw*
$L \leftarrow$ level of ep
for $l_c \leftarrow L...0$ **do**
    $W \leftarrow$ **SearchLayer**$(q, ep, ef = 1, l_c)$
    $ep \leftarrow$ get nearest element from W to q
end for
$W \leftarrow \text{SearchLayer}(q, ep, ef = efSearch, l_c = 0)$
return **SelectNeighborsSimple**(q, W, K)

Search: how to search candidates on each layer?

Algorithm 5 SearchLayer

Require: query element q, enter point**s** ep, number of nearest neighbors to q to return ef, layer number l_c
Ensure: ef nearest neighbors to q

```
v ← ep // visited nodes. Note that the passed ep is a set of points, not a single node!
W ← ep // set of final results (updated during search)
C ← ep // set of candidate nodes for search steps
while |C| > 0 do
    c ← extract nearest element from C to q
    f ← get furthest element from W to q
    if distance(q, c) > distance(q, f) then
        break
    end if
    for each e ∈ neighbourhood(c) at layer l_c do
        if e ∉ v then
            v ← v ⋃ e
            f ← get furthest element from W to q
            if distance(q, e) < distance(q, f) or |W| < ef then
                C ← C ⋃ e
                W ← W ⋃ e
                if |W| > ef then
                    remove furthest element from W to q
                end if
            end if
        end if
    end for
end while
return W
```

Search: how to select neighbors (simple version)?

- This is the simplest version of a selection method for neighbors
- It may lead disconnected graphs, where few nodes are not properly linked to the others
- It's simpler and faster than other heuristics (perfect for us)

Algorithm 6 Select Neighbors Simple

Require: base element q, candidate elements C, number of neighbors to return K
Ensure: K nearest elements to q

1: **return** K nearest elements from C to q

Insert

Algorithm 7 Insert

Require: multilayer graph *hnsw*, new element q, number of connections per node M_{max}, size of dynamic candidate list *efConstruction*, normalization factor m_L

Ensure: update *hnsw* inserting new element q

$W \leftarrow \emptyset$ *// list for the currently found nearest neighbors*
ep $\leftarrow$ get enter point for *hnsw*
$L \leftarrow$ level of *ep* *// current top-layer for hnsw*
$l \leftarrow$ **RandomLevel**(m_L)
for $l_c \leftarrow L \ldots l + 1$ **do**
    $W \leftarrow$ **SearchLayer**(q, *ep*, $ef = 1$, l_c)
    ep $\leftarrow$ get the nearest element from W to q
end for
for $l_c \leftarrow \min(L, l) \ldots 0$ **do**
    $W \leftarrow$ **SearchLayer**(q, *ep*, *efConstruction*, l_c)
    neighbors $\leftarrow$ **SelectNeighborsSimple**(q, W, M)
    add *bidirectional* connections from *neighbors* to q at layer l_c
    for each $e \in$ *neighbors* **do**
        eConn $\leftarrow$ *neighbourhood*(e) at layer l_c
        if |*eConn*| ¿ M_{max} **then**
            eNewConn $\leftarrow$ SelectNeighborsSimple (e, *eConn*, M_{max})
            set *neighbourhood*(e) at layer l_c to *eNewCon*
        end if
    end for
    ep $\leftarrow W$
end for
if $l > L$ **then**
    set enter point for *hnsw* to q
end if

References

MALKOV, Y. A., AND YASHUNIN, D. A.
Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018.

PUGH, W.
Skip lists: a probabilistic alternative to balanced trees.
Commun. ACM 33, 6 (June 1990), 668–676.