The UsefulnessIndex Specification

Overview: The *UsefulnessIndex* maintains a collection of nodes, prioritizing them based on a calculated usefulness metric. It supports adding or updating nodes, retrieving nodes sorted by usefulness, and removing nodes. The class ensures that the number of stored nodes does not exceed a specified maximum capacity by evicting the least useful nodes when necessary.

Mathematical Definitions of Parameters and Sets

Maximum Capacity: $C_{max} \in N (Given\ constant,\ e.g.,\ C_{max} = 1000)$

Time: $T = \{0,1,2,3, \dots\} (Discrete\ time\ steps)$

Nodes: $N = \{n \mid n.Id \in \mathbb{Z},\ n.Label \in Strings\}$

State Variables: At any time $t \in T$, the state of the *UsefulnessIndex* is defined by:

Node Lookup Function: $NodeLookupt_t : \mathbb{Z} \to TrackedNode$

Maps Node Ids to their corresponding *TrackedNode* objects.

Sorted Usefulness Index: $SortedIndext_t : U_t \to 2^{TrackedNode}$

A mapping from usefulness scores to sets of *TrackedNode* objects with that score.

$Ut \subseteq [0,1]$ is the set of usefulness scores at time t.

Current Time: $CurrentTime = t$

TrackedNode

A *TrackedNode* is defined as a tuple:

$$TrackedNode = (n, UseCount, LastUsed)$$

where:

$n \in N$

$UseCount \in N$

$LastUsed \in T$

Usefulness Function: For a *TrackedNode* $\tau = (n, UseCount, LastUsed)\ at\ time\ t$:

Recency: $Recency_t(\tau) = max\ (0, t - LastUsed)$

Raw Usefulness: $RawUsefulness_t(\tau) = \frac{UseCount_t}{Recency_t(\tau)+1}$

Normalized Usefulness: $Usefulness_t(\tau) = \frac{RawUsefulness_t(\tau)}{RawUsefulness_t(\tau)+1} \in [0,1)$

Operations

1. **IncrementTime()**

Purpose: Advance the current time by 1.

Effect: $t \leftarrow t + 1$

2. AddOrUpdateNode(n)

Input: $n \in N$

Behaviour: **Check if Node Exists**:

$If\ n.Id \in dom(NodeLookupt)$:

$Retrieve\ \tau = NodeLookupt(n.Id)$.

$Update$:

$UseCount \leftarrow UseCount + 1$

$LastUsed \leftarrow t$

$Recalculate\ Usefulness$:

$Remove\ \tau\ from\ its\ current\ usefulness\ bucket\ in\ SortedIndext.$

$Compute\ new\ Usefulnesst(\tau).$

$Add\ \tau\ to\ the\ new\ bucket\ in\ SortedIndext.$

$Else$:

Capacity Check:

$If\ \mid NodeLookupt \mid \geq Cmax$:

Evict Nodes (See *EvictLowUsefulnessNodes*).

$Create\ new\ \tau = (n, 1, t).$

$Compute\ Usefulnesst(\tau).$

$Add\ \tau\ to$:

$NodeLookupt(n.Id) = \tau$

$Appropriate\ bucket\ in\ SortedIndext.$

3. EvictLowUsefulnessNodes()

Purpose: Ensure the capacity constraint is maintained by evicting least useful nodes.

Behaviour: $While\ \ |\ NodeLookupt\ | \geq Cmax$:

1. Identify the minimum usefulness score: $Umin = min\ \ Ut$
2. Retrieve the set of nodes with U_{min}: $[S_{min} = SortedIndex_t(U_{\min})]$
3. Remove one node τ from S_{min}:

$Remove\ \tau\ from\ S_{min}.$

$If\ S_{min}\ is\ empty\ after\ removal$

$Remove\ U_{min}\ from\ SortedIndex_t$

$Remove\ \tau\ from\ NodeLookup_t$

$NodeLookup_t \leftarrow NodeLookup_t \setminus \{n.Id\}$

$Break\ the\ loop\ to\ maintain\ capacity$

4. RemoveNode(nodeId)

Input: $nodeId \in \mathbb{Z}$

Behaviour:

$If\ nodeId \in dom(NodeLookup_t)$

$Retrieve\ \tau = NodeLookup_t(nodeId)$

$Compute\ U = Usefulness_t(\tau)$

$Remove\ \tau\ from\ SortedIndex_t(U)$

$If\ SortedIndex_t(U)\ is\ empty\ after\ removal$

$Remove\ U\ from\ SortedIndext$

$Remove\ nodeId\ from\ NodeLookup_t$

5. GetSortedNodes()

Output: A list of nodes $[n_1, n_2, \ldots, n_k]$ sorted in descending order of usefulness at time t.

Behaviour: For all $U \in U_t$ **descending order**:

$For\ each\ \tau \in SortedIndex_t(U)$:

$Add\ \tau.n\ to\ the\ output\ list.$

6. **DisplayState()**

Purpose: For debugging purposes; outputs the current state of the index.

Behaviour: $For\ all\ U \in U_t\ in\ descending\ order$:

$For\ each\ \tau \in SortedIndex_t(U)$:

$Display$:

$\tau.n.Label$

$\tau.UseCount$

$\tau.LastUsed$

$U\ (formatted\ as\ needed)$

Data Consistency and Constraints

Capacity Constraint: $S_{min} = SortedIndex_t(U_{min})$

Bidirectional Mapping: Every τ in $NodeLookup_t$ is in exactly one bucket in $SortedIndex_t$, corresponding to $Usefulness_t(\tau)$

No duplicate nodes exist in $NodeLookup_t$

Usefulness Score Range: $Usefulnesst(\tau) \in [0,1)$

Time Advancement: Time t advances only via the *IncrementTime()* method.

Error Handling: Methods handle cases where inputs do not correspond to existing nodes gracefully (e.g., removing a non-existent node does nothing).

Notes

Normalization of Usefulness: The normalization function:

$$f(x) = \frac{x}{x+1}$$

ensures that usefulness scores are normalized between 0 and 1, avoiding issues with unbounded values.

Recency Factor: The addition of 1 in $Recency + 1$ prevents division by zero.

Eviction Strategy: Nodes with the lowest usefulness are evicted first.

When multiple nodes share the same lowest usefulness score, nodes are evicted in an unspecified order from that bucket.

Use Count and Last Used: These attributes are updated to reflect node usage patterns, influencing the usefulness calculation.

Dependencies: **Node**:

Properties: $n.Id \in \mathbb{Z}\ and\ n.Label \in Strings$

TrackedNode: Represents nodes with usage tracking:

$$\tau = (n, UseCount, LastUsed)$$

This formal specification provides a mathematical foundation for the *UsefulnessIndex*, defining its behaviour, state, and operations precisely. It facilitates understanding, verification, and potential implementation in various contexts.