N-Gram Based Word Predictor

Today, we are presenting our DSA Project for the odd semester of academic year 2024-25, titled '**N-Gram Based Word Predictor'**. This project tries to build a word prediction system that efficiently predicts the next word based on user input using n-grams. We achieve so by :

- efficiently storing the data in **trie** and **B+Trees**.
- **Prefix matching** and **Levenshtein** distance.
- **Markov** assumption
- **Stupid backoff** technique
- Data structures used : **priority queues**, **stack**, **trie**, **b+trees**.

Structure used in the project

For B+Trees:

```
#define MAX_KEYS 15

#define MAX_LINE_LENGTH 300

typedef struct {

   char *ngram[3];

   int count[3];

   int top;

} bt_priority_q;


typedef struct BTreeNode {

   int isLeaf;

   int numKeys;

   char keys[MAX_KEYS][200];

   int counts[MAX_KEYS];

   struct BTreeNode *children[MAX_KEYS + 1];

   struct BTreeNode *next;

} BTreeNode;


typedef struct BPlusTree {
```

```
    BTreeNode *root;
    long long int totalNgramsCount;
} BPlusTree;
```

For trie data structures and string processing:

```
#define MAX_WORDS 3
#define MAX_EDIT_DISTANCE 0.25
#define MAX_TOKEN_LEN 100
#define DELIMS " .,/?;:{}[]~`!|$%&*()_-+=^\'\"\t\n"

typedef struct trie_node {
    struct trie_node *children[26];
    int count;
    bool isEndOfWord;
} trie_node;

typedef struct trie{
    trie_node * root;
    long long int total_unigram_count;
}trie;

typedef struct {
    char word[100];
    double prob;
    float distance;
} word_element;

typedef struct {
```

```
    word_element words_collection[MAX_WORDS];
    int size;
} priority_Q;

//the string processing part
typedef struct node{
    char token[MAX_TOKEN_LEN];
    struct node * next;
}node;

typedef struct{
    node * top;
    int size;
}stack;
```

Stupid backoff

- **Stupid Backoff technique** which is a **smoothing technique used in language models**, such as the **trigram model**, to handle cases where a higher-order n-gram is not found in the training data.

- The approach avoids assigning a zero probability to unseen n-grams by falling back to lower-order n-grams, such as bigrams or unigrams.

- We apply the **Markov Assumption** to predict the next word by considering only the most recent one or two words from the input.

$$S(w_i|w_{i-N+1:i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-N+1:i})}{\text{count}(w_{i-N+1:i-1})} & \text{if count}(w_{i-N+1:i}) > 0 \\ \lambda S(w_i|w_{i-N+2:i-1}) & \text{otherwise} \end{cases}$$

[Why B + trees for storing N - grams?]

- ***Future Scalability:***

- Ideal for managing large datasets.
- consistent retrieval time
- Highly scalable

- ***Consistent Performance:***

- All operations maintain logarithmic time complexity :

Time Complexity of B+ Tree Operations

Operation	Time Complexity	Explanation
Search	$O(\log n)$	The search time is logarithmic in the number of elements n due to the balanced tree structure.
Insertion	$O(\log n)$	Insertion involves searching for the correct position and then possibly rebalancing, both of which take logarithmic time.
Deletion	$O(\log n)$	Deletion requires searching for the element, removing it, and possibly rebalancing the tree, which takes logarithmic time.

- ***Compatibility with the stupid backoff technique:***

- Their **hierarchical structure**, which **aligns perfectly with the backoff mechanism**.

[Why Tries for prefix matching and Levenshtein Distance Algorithm ?]

Time Complexity of Trie Operations

Operation	Time Complexity	Explanation
Search	$O(k)$	k is the length of the word being searched. Each character is checked in sequence from root to leaf.
Insertion	$O(k)$	k is the length of the word being inserted. Nodes are added for characters not already in the trie.
Deletion	$O(k)$	k is the length of the word to delete. Each character is checked, and nodes are removed if they are no longer needed.
Prefix Matching	$O(k)$	k is the length of the prefix being matched. The search traverses nodes up to the length of the prefix.
Levenshtein Distance	$O(m \cdot n)$	m is the length of the input word and n is the length of the word in the trie being compared.
Memory Usage	$O(k \cdot n)$	k is the average word length, and n is the number of words stored in the trie, considering each unique character stored in the trie.

Efficient Prefix Matching:

- Tries are **specifically designed for efficient prefix matching**. .

Speed for Large Datasets:

- Tries **minimize redundant comparisons.**
- Improved speed and space used, especially **when the dataset has common prefixes**.

Comparative analysis : B + trees v/s Hash Tables for storing N grams.

1. **Time Complexity:**

- **Hash Tables:**
  - **Average-case: O(1)** for insertion, search, and deletion with a good hash function.
  - **Worst-case**: **O(n)** due to collisions
- **B+ Trees: Consistent O(logn)** time complexity for insertion, search, and deletion, ensuring stable performance even with large datasets.

2. **Scalability:**
   - **Hash Tables:**Best suited for small to medium datasets stored in memory. Performance suffers when the dataset grows or requires frequent resizing.
   - **B+ Trees:**Highly scalable for large datasets, with efficient disk-based operations and minimal memory overhead.

3. **Load Balance and Collisions:**
   - **Hash Tables:**Susceptible to collisions, especially as the load factor increases. Collisions are resolved through chaining or open addressing, both of which degrade performance.
   - **B+ Trees:**Always balanced, with no collision issues, ensuring consistent performance regardless of data distribution.

4. **Disk and Memory Usage:**
   - **Hash Tables:** Requires frequent resizing and additional memory for collision resolution.
   - **B+ Trees:** Their memory usage is proportional to the dataset size, making them efficient for large-scale storage.

5. **Use Case for N-grams:**
   - **Hash Tables:** Best for exact-match lookups in small datasets where hierarchical searches are not needed.
   - **B+ Trees:** Ideal for NLP tasks like "stupid backoff," where hierarchical backoff, range queries, and scalability are essential.

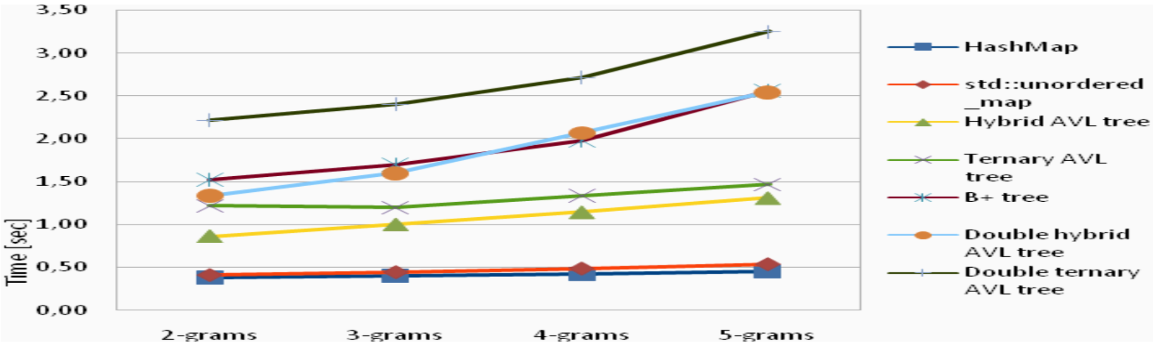

Fig. 2. Comparison of search time

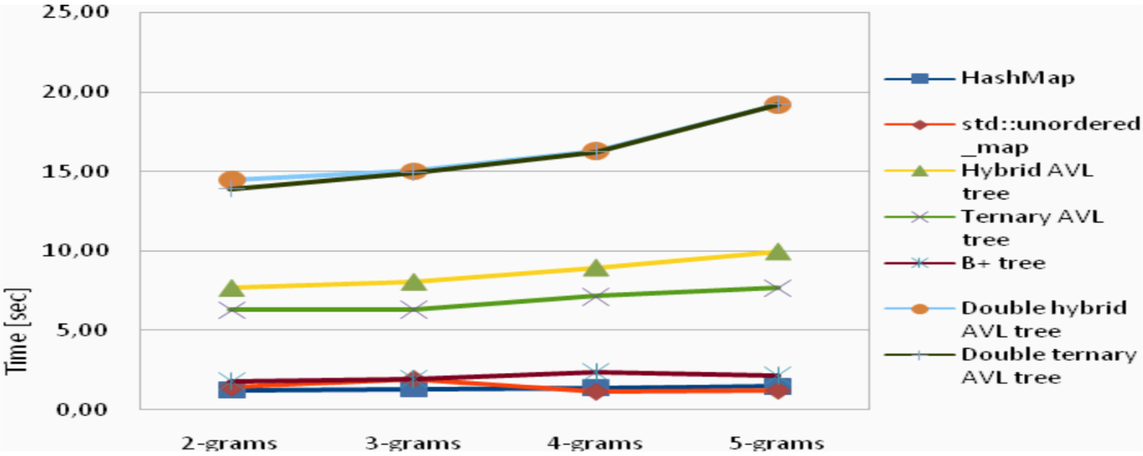

Fig. 1. Insert time comparison

<u>Presented by :</u> DISHA KADAM - 612303076, JYOTIKA SHARAN - 612303075 , HAKIMUDDIN SLATEWALA - 612303065

<u>References :</u>

1.Jurafsky, D., & Martin, J. H. (2024). **N-gram language models**. *In Speech and Language Processing (3rd ed.). Draft of August 20, 2024.*

2. **Efficient in-memory data structures for n-grams Indexing** *.Daniel Robenek, Jan Platoš, Václav Snášel Department of Computer Science, FEI, VSB – Technical University of Ostrava*

*3.*Natural Language Processing with Probabilistic Models
by ***DeepLearning.AI. Taught by:Younes Bensouda Mourri*** *and* ***Łukasz Kaiser****,*