A Nifty Large-Scale Text Search Algorithm

Viet Anh Trinh

Artificial Intelligence Class Presentation
Mentors: Sos Agaian

September 16, 2019

Outline

Problem

- A real world application:
  - We have a list of medical conditions
  - Requirement: finding out which medical theses discuss these conditions.
- Problem:
  - We have a group of search phrases , denoted P
  - Requirement: find which phrases in P are in a large body of text

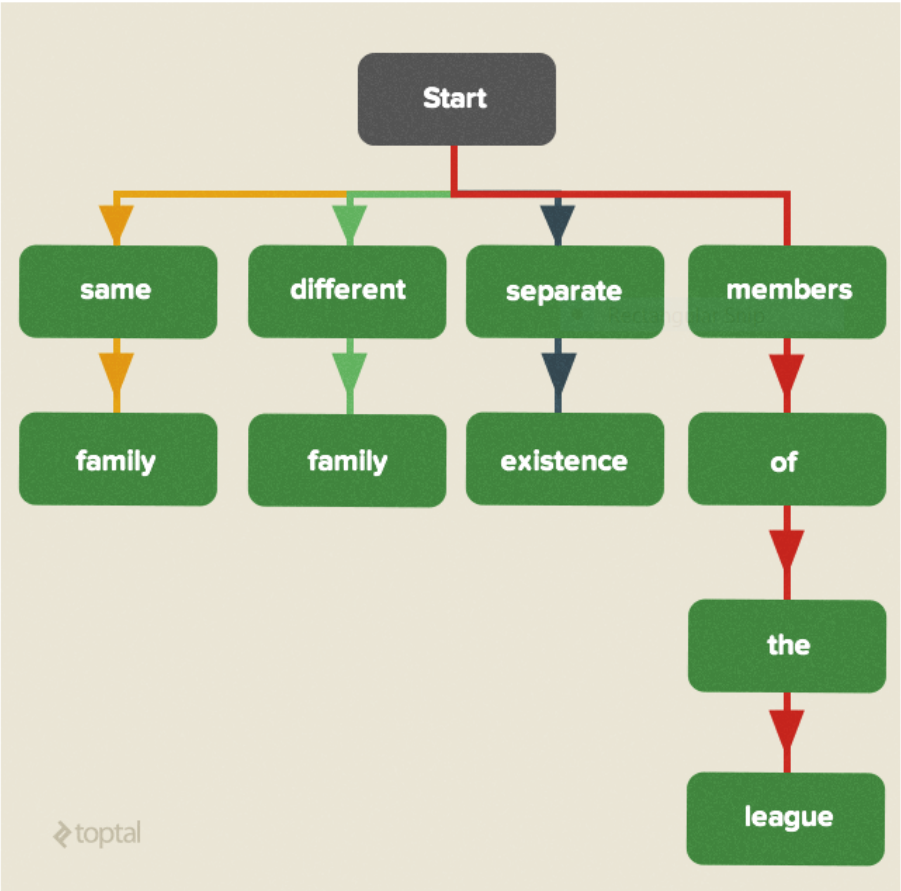
Algorithm Discussion

- There are m phrases, and text has N words
- A naive algorithm:
  - Search through the text each phrase, one by one
  - Search for the text m times $\rightarrow$ complexity $O(m * N)$
  - Example: Search 280K search phrases (23.5MB), and a text size of 1.5MB take 74 minutes on Intel i7 4700HQ, 16GB RAM

  Trie search algorithm
- Trie algorithm:
  - Complexity $O(N)$
  - Example: Trie search takes 2.5 seconds for building the trie and 0.3 seconds for the search on the same dataset, same machine.
- Link: https://www.toptal.com/algorithms/needle-in-a-haystack-a-nifty-large-scale-text-search-algorithmfootnote2

Buiding the trie

- Example: P =(“same family”, “different family””, separate existence”, “members of the league”)
- Text = The European languages are members of the same family. Their separate existence is a myth.



Trie Search

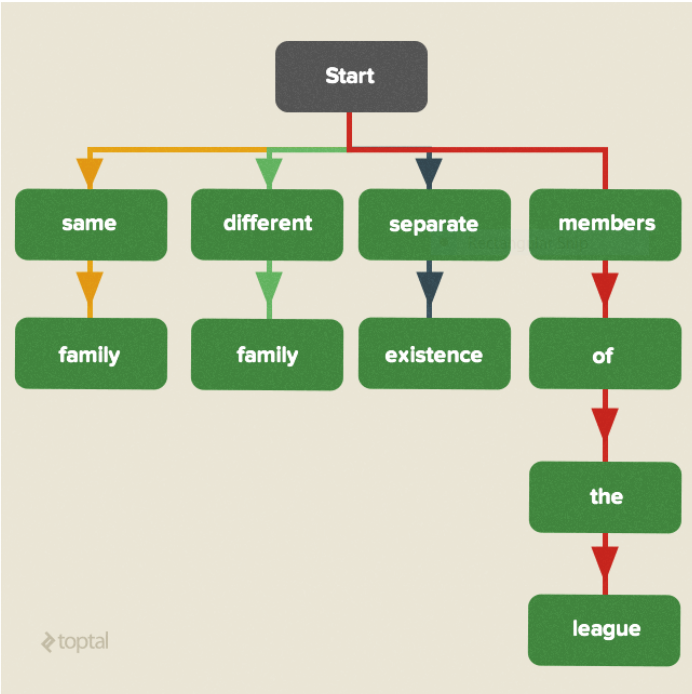


Figure: Trie seach

Step	Trie Indicator	Text Indicator	Match?	Trie Action	Text Action
0	*start*	The	-	Move to *start*	Move to next
1	*start*	European	-	Move to *start*	Move to next
2	*start*	languages	-	Move to *start*	Move to next
3	*start*	are	-	Move to *start*	Move to next
4	*start*	members	members	Move to members	Move to next
5	members	of	of	Move to of	Move to next
6	of	the	the	Move to the	Move to next
7	the	same	-	Move to *start*	-
8	*start*	same	same	Move to same	Move to next
9	same	family	family	Move to *start*	Move to next
10	*start*	their	-	Move to *start*	Move to next
11	*start*	separate	separate	Move to separate	Move to next
12	separate	existence	existence	Move to *start*	Move to next
13	*start*	is	-	Move to *start*	Move to next
14	*start*	a	-	Move to *start*	Move to next
15	*start*	myth	-	Move to *start*	Move to next

Figure: Trie

Algorithm

- Two pointers, one at the "start" node of the trie, another at the first word of the text.
- Two pointers move along together, word by word
- Text pointer simply move forward, except when a partial match is found.
  - i.e. after a series of matches a non-match is encountered before the branch ends. In this case the text indicator is not moved forward, since the last word could be the beginning of a new branch.
- Tree pointer traverses depth-wise
- The tree pointer return to start in 2 cases:
  - It reaches the end of a branch
  - It encouters a non-matching word

Code

```
void findPhrases(ref Node root, String textBody)
{
    // a pointer to traverse the trie without damaging
    // the original reference
    Node node = root;

    // a list of found ids
    List<int> foundPhrases = new List<int>();

    // break text body into words
    String[] words = textBody.Split ();

    // starting traversal at trie root and first
    // word in text body
```

Code

```
for (int i = 0; i < words.Length;)
{
    // if current node has current word as a child
    // move both node and words pointer forward
    if (node.Children.ContainsKey(words[i]))
    {
        // move trie pointer forward
        node = node.Children[words[i]];
        // move words pointer forward
        ++i;
    }
    else
    {
        // current node doesn't have current word in its children
        // If there is a phrase Id, then the previous sequence of
        //   words matched a phrase, add Id to found list
```

Code

```
            if (node.PhraseId != -1)                    foundPhrases.Add(node.P
            if (node == root)
            {
                // if trie pointer is already at root, increment
                // words pointer
                ++i;
            }
            else
            {
                // if not, leave words pointer at current word
                // and return trie pointer to root
                node = root;
            }
        }
    }
    // one case remains, word pointer as reached the end and the loop
    //  is over but the trie pointer is pointing to a phrase Id
    if (node.PhraseId != -1)
        foundPhrases.Add(node.PhraseId);
}
```

Thank you