Tries

Data Structures and Algorithms for Computational Linguistics III
(ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2020/21

version: 090cabc @2021-02-03

Tries

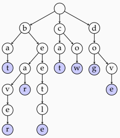
- A *trie* (or *prefix tree*) is a tree-based data structure, particularly used for fast pattern matching
- Common applications include
  - Information retrieval: indexing large collections of texts based on keyword sequences
  - Storing lexicons and implementing 'autocomplete'
  - As a replacement for hash tables
- A type of tries, *suffix trees*, are particularly useful for solving a number of questions about strings efficiently

---

Tries - or 'standard' tries

definition

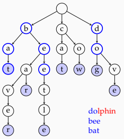
- A trie is a tree representation of a set of strings
- Each node is associated with a character
- Tracing paths from root to the leaf nodes produce each strings
- Shared prefixes in a trie is represented in common branches
- None of the string can be a prefix of another



Searching in tries

definition

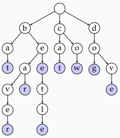
- Start from the root, jump to node with current character
- Fail:
  - If there is no character to follow
  - Input ends in a non-leaf node
- Accept if we are at a leaf node at the end of the input



Tries

with prefix conflicts

- To prevent that no string is a prefix of another, a common trick is append a special end-of-string symbol
- Another approach is to mark the nodes that correspond to ends of strings



Inserting, deleting and complexity

- Search in a trie is clearly linear in the size of the string being searched
- There is a factor coming from the alphabet size q, but this can be reduced to $O(\log q)$ with binary search, or $O(1)$ if a method allowing direct addressing is used
- Both insertion and deletion starts with a lookup, and possibly inserts new nodes or deletes them
- All operations are similarly $O(n)$ (without the effect of the alphabet size)

Properties of tries

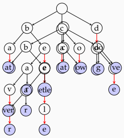
- Internal nodes may have as many children as the number of symbols in the alphabet
  - in practice this will be much smaller on average
  - average degree of nodes also goes down as the depth increase (longer prefixes are less likely)
- The height of the trie is the length of the longest string
- Number of leaves are equal to the number of strings
- In the worst case, the number of nodes is the total length of all strings

---

Compressed tries

analysis

- In typical use, tries are sparse, resulting long chains
- Tries can be compressed by replacing 'redundant' nodes with nodes labeled with substrings rather than characters
- Compressing tries saves space, and may also speed up some operations



Suffix tries or trees

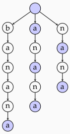
- *Suffix tries* (or suffix trees) are tries that include all suffixes of a string
- Suffix tries allow fast retrieval of any substring: substring search on a suffix trie is linear
- They are used extensively in information retrieval
- They can also be adapted for wild card search and approximate approximate search

---

Suffix tries

example: a suffix trie for banana

- If the search ends in a leaf node, the pattern is a suffix of the string
- If there is a path from root following until the end of the string, the pattern is in the string
- Suffix tries can also be compressed like the regular tries



All suffixes:

```
      #
     a#
    na#
   ana#
  nana#
 anana#
banana#
```

Properties of suffix tries

- Standard suffix tries use $O(n^2)$ space, compression reduces space requirement to $O(n)$
- Space complexity can be reduced by keeping indexes to the string rather than the string itself in the (compressed) trie nodes
- Iterative insertion of suffixes result in a quadratic ($O(qn^2)$) construction time complexity
- There are linear time algorithms for constructing suffix tries
- Generalized suffix tries allow storing multiple strings (documents) in a single suffix trie (each string gets a special end-of-string marker)

Summary

- Tries are useful tree-based data structures
- Their applications include set or map implementations, storing dictionaries, and information retrieval
- Reading suggestion: Goodrich, Tamassia, and Goldwasser (2013, chapter 13)

Next:

- Regular languages and finite state automata
- Suggested reading: Jurafsky and Martin (2009, chapter 2)

Acknowledgments, credits, references

- Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN: 9781118476734.
- Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second edition. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.

blank

blank

blank

blank