COMPRESSED TRIES AND TOP-K STRING COMPLETION

Giuseppe Ottaviano

ISTI-CNR – Pisa

Joint work with Roberto Grossi and Paul Hsu

String set representation

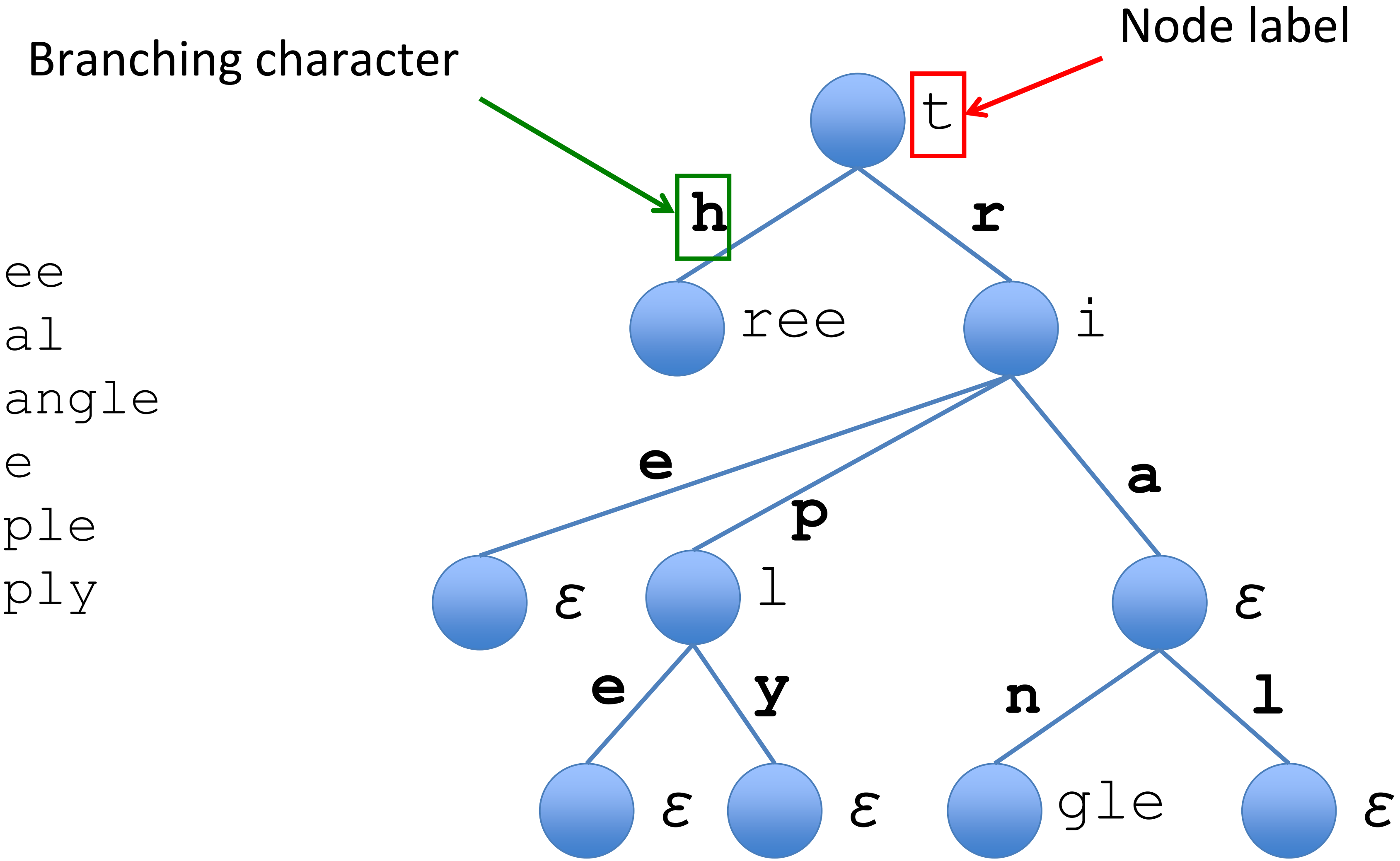
```
three
trial
triangle
trie
triple
triply
```

Represent a string set so that

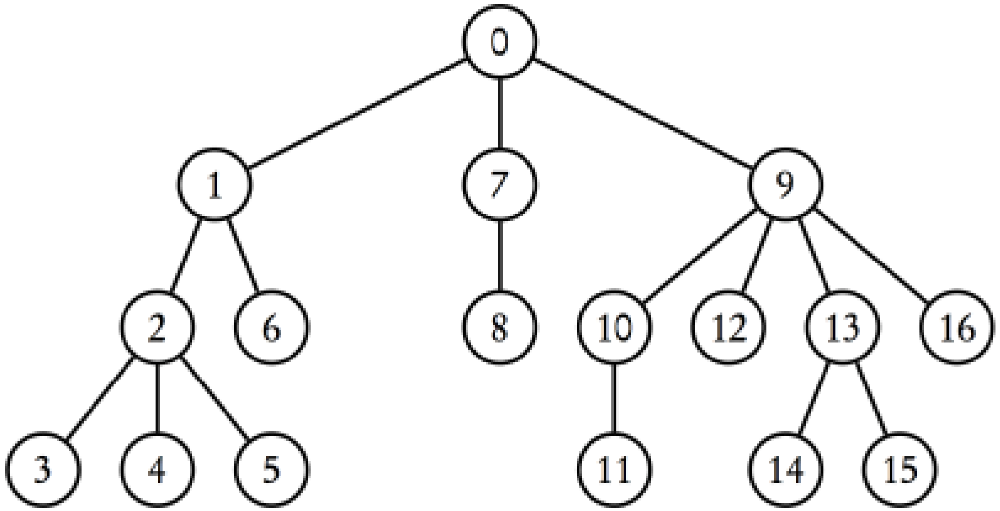
- Lookup and access operations are fast
- Space of the representation is small

Compacted tries

three
trial
triangle
trie
triple
triply



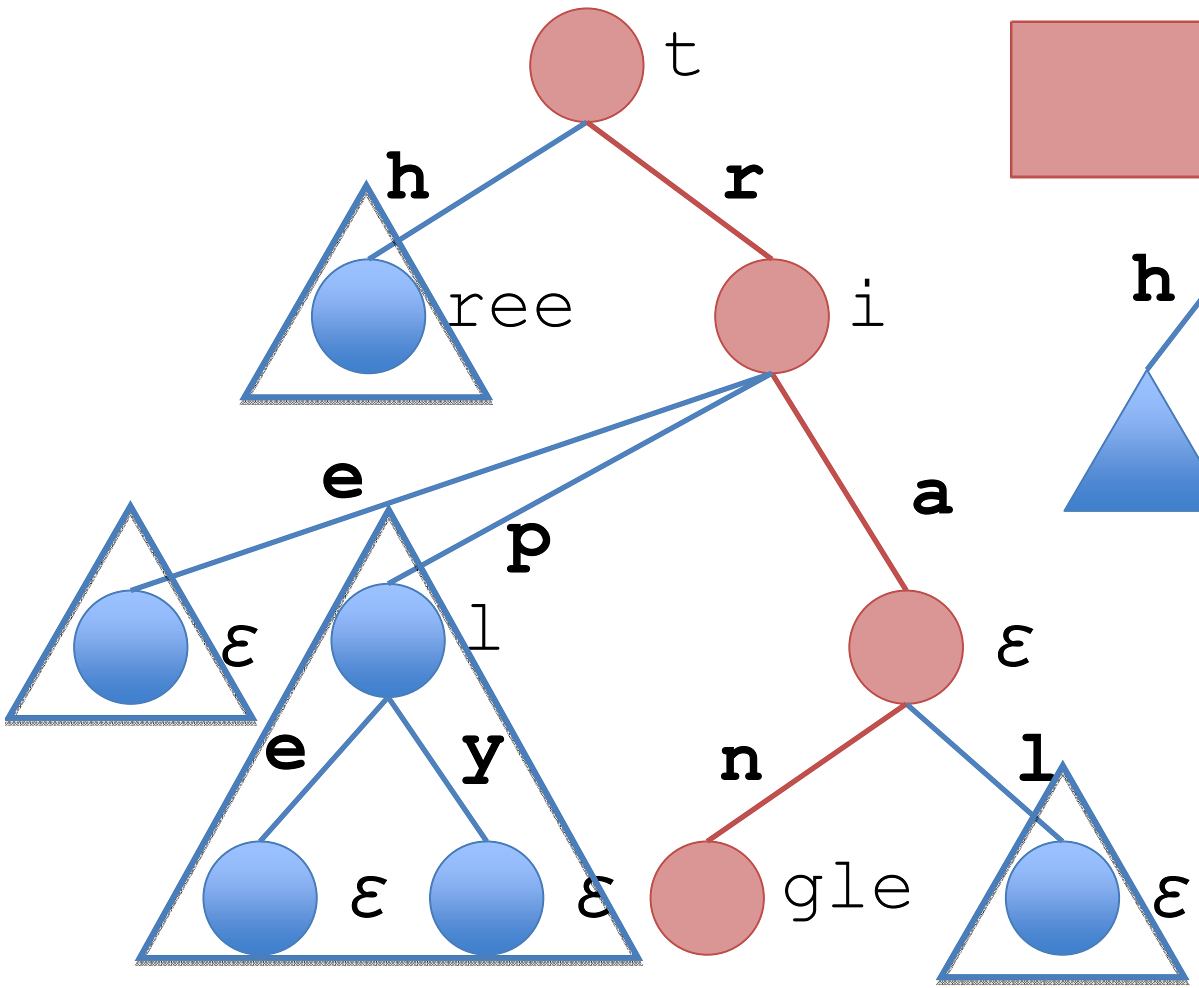
Succinct tree encoding

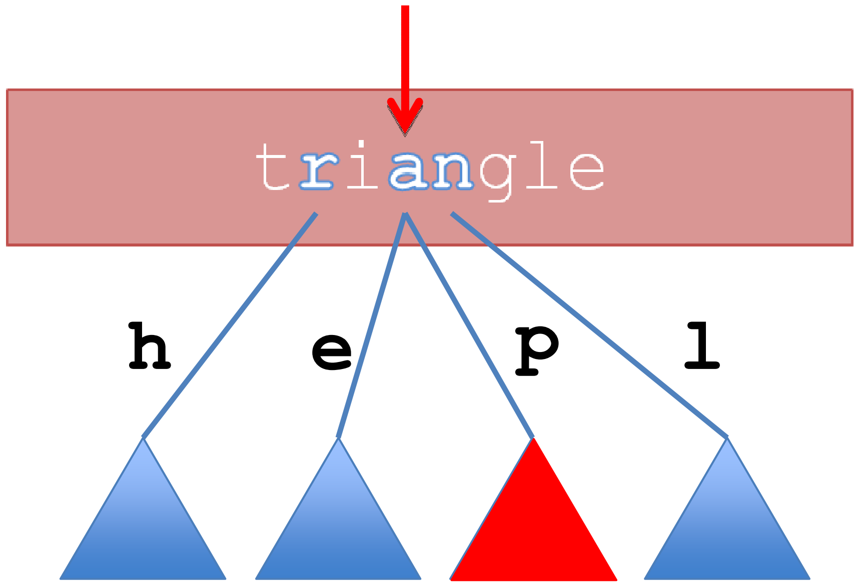


Succinct *trie* encoding?

- Good candidate for succinct tree encoding
  - Most space is taken by tree pointers
- But… tries can be tall, usually small fanout
- Navigation of succinct trees is “slow”
  - A few cache misses for each FirstChild
  - Even if no cache misses, constants hidden in O(1) are high
- Existing libraries using LOUDS tree encoding are indeed slow

Path decomposition





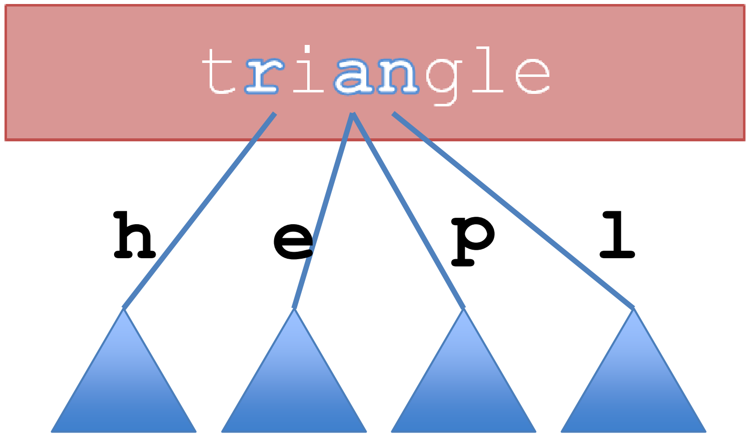
Recurse here with suffix `le`

Query: `triple`

Centroid path decomposition

- Starting from the root, recursively choose the node with most descendants
- Height of path decomposition tree O(log n) with this strategy

Succinct encoding

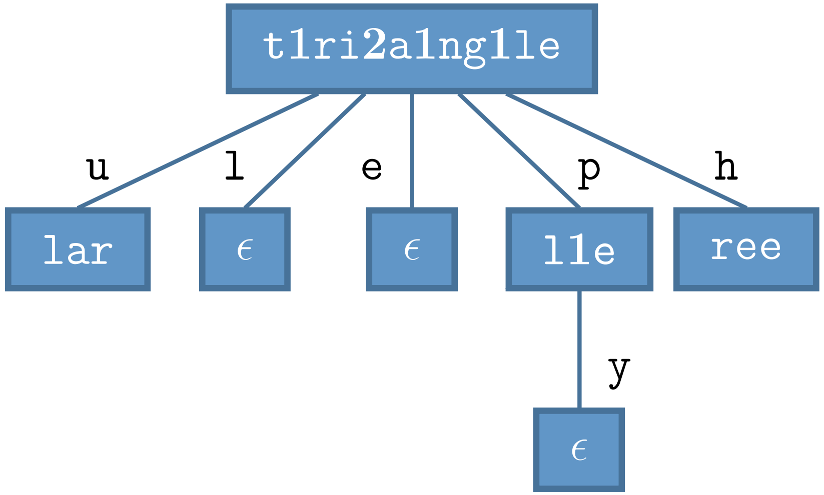



```
L : t1ri2a1ngle
BP:  (  (((    )
B :  h  epl
```

(spaces added for clarity)

- Node label written literally, interleaved with number of *other* branching characters at that point in array **L**
- Corresponding branching characters in array **B**
- Tree encoded with DFUDS in bitvector **BP**
  - Variant of Range Min-Max tree [ALENEX 10] to support Find{Close,Open}, more space-efficient (Range Min tree)

```
three
trial
triangle
triangular
trie
triple
triply
```



L `t1ri2a1ng1lelarl1eree`

BP `(((((())))()))`

B `hpeluy`

Compression of **L**

`...$...index.html$....html$....html$...index.html$`

...	
3	`index`
...	
5	`.html`
...	

Dictionary

`...$...35$...5$...5$...35$`

- Dictionary codewords shared among labels
- Codewords do not cross label boundaries (`$`)
- Use vbyte to compress the codeword ids

Experimental results (time)

- Experiments show gains in time comparable to the gains in height
- Confirm that bottleneck is traversal operations

	Web Queries	URLs	Synthetic
Hu-Tucker Front Coding	3.8	7.0	22.0
Lexicographic trie	3.5	5.5	119.8
Centroid trie	**2.4**	**3.4**	**5.1**

(microseconds, lower is better)

Code available at `https://github.com/ot/path_decomposed_tries`

Experimental results (space)

- For strings with many common prefixes, even non-compressed trie is space-efficient
- Labels compression considerably increases space-efficiency
- Decompression time overhead: ~10%

	Web Queries	URLs	Synthetic
Hu-Tucker Front Coding	40.9%	24.4%	19.1%
Centroid trie	55.6%	22.4%	17.9%
Centroid trie + compression	**31.5%**	**13.6%**	**0.4%**

(compression ratio, lower is better)

Code available at `https://github.com/ot/path_decomposed_tries`

Top-k string completion

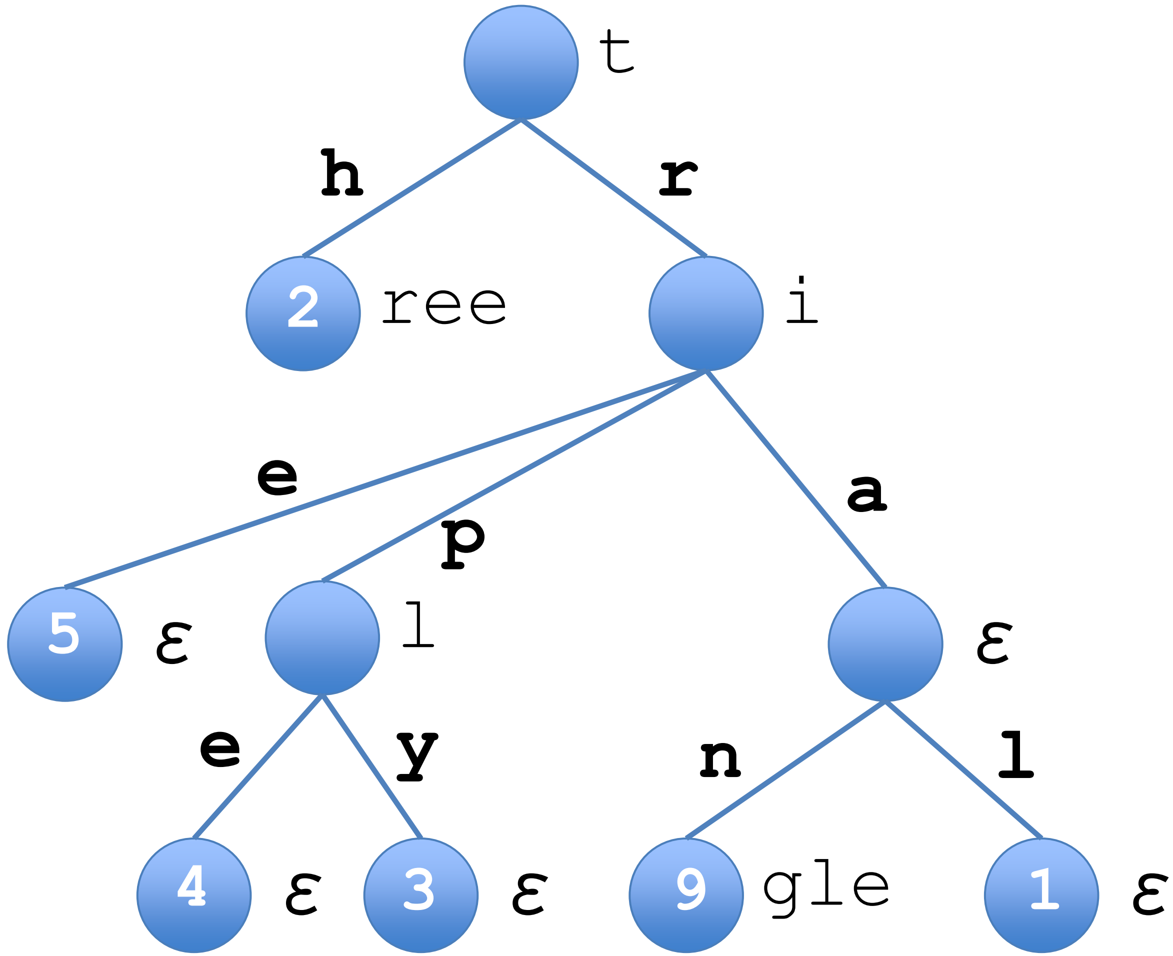
`three`	`2`
`trial`	`1`
`triangle`	`9`
`trie`	`5`
`triple`	`4`
`triply`	`3`

- Top-k Completion query
  - Given prefix p, return k strings prefixed by p with highest scores
- Example: p="tr", k=2
  - `(triangle, 9), (trie, 5)`

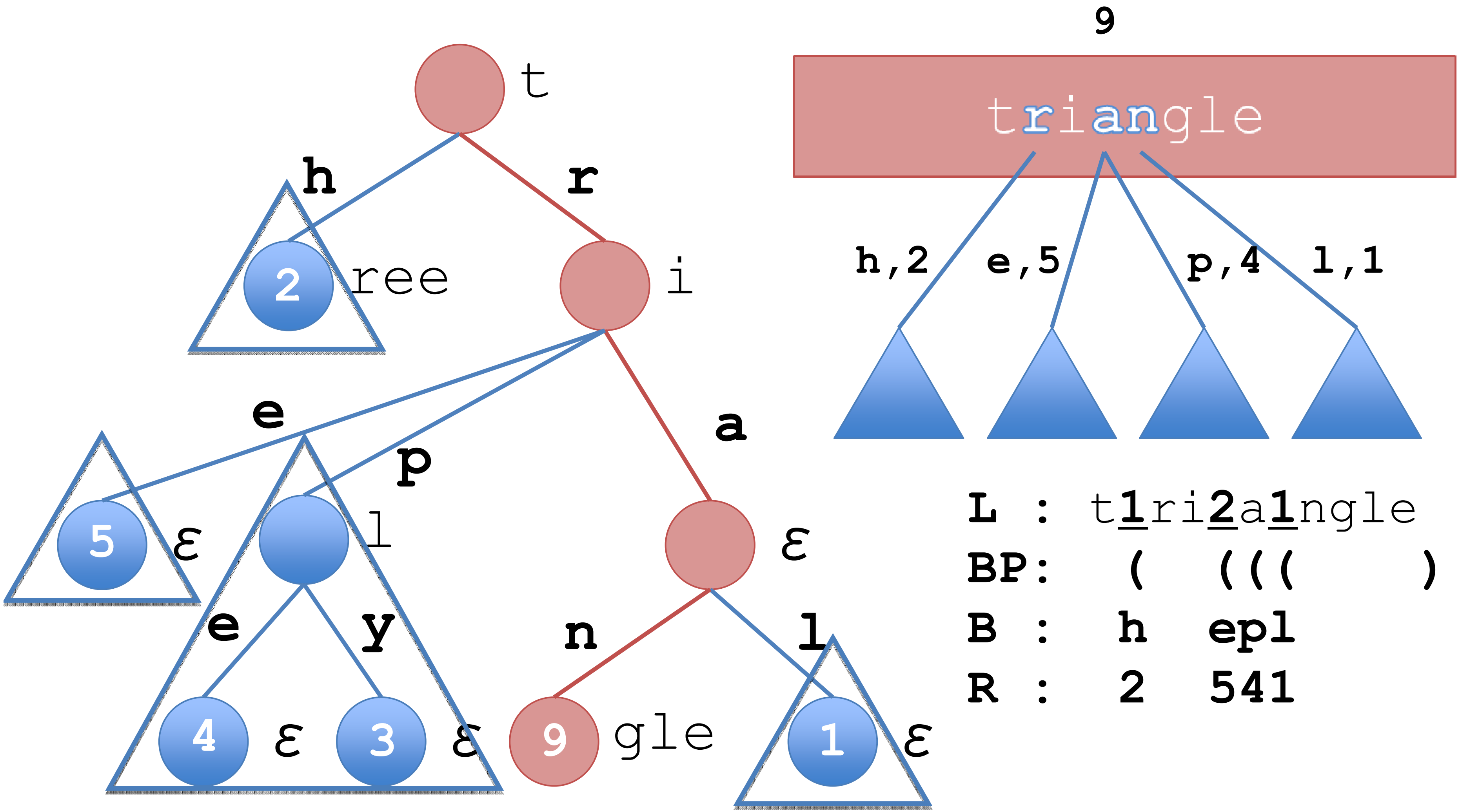
Motivation: query suggestion

(Scored) compacted tries

three
trial
triangle
trie
triple
triply

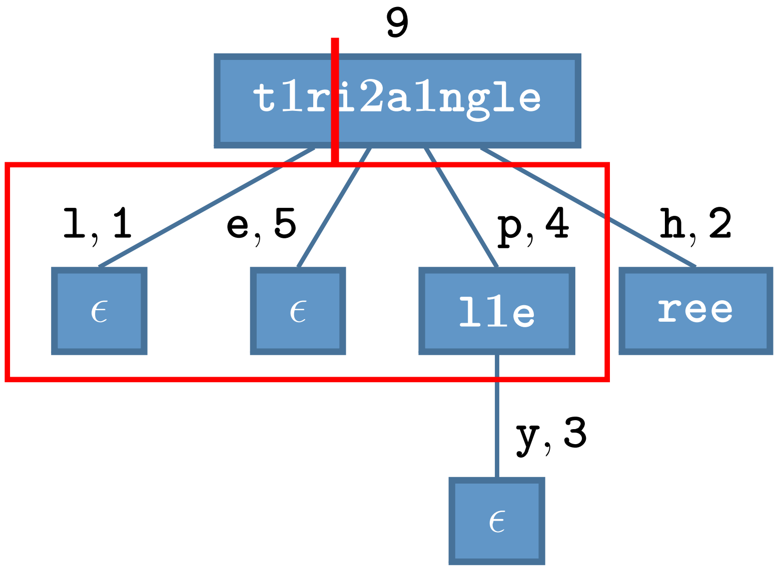


Max-score path decomposition



three	2
trial	1
triangle	9
trie	5
triple	4
triply	3

Complete **tr**



L **t1ri2a1nglel1eree**

BP ((((()))()))

B **hpely**

R **245139**

Score compression

...	3 5 1 2 3 0	0 1 2 4 1900 1	1 2 3 2 1 10000	...
	3 bits/value	11 bits/value	16 bits/value	

- *Packed-blocks array*
  - "Folklore" data structure, similar to many existing packed arrays
- Divide the array into fixed-size blocks
- Encode the values of each block with the same number of bits
- Store separately the block offsets

Score compression

. . .	3 5 1 2 3 0	0 1 2 4 1900 1	1 2 3 2 1 10000	. . .
	3 bits/value	11 bits/value	16 bits/value	

- Can be unlucky
  - Each block may contain a large value
- But scores are power-law distributed
- Also, tree-wise monotone sorting
- On average, 4 bits per score

Results

- Bing query histogram: 400M queries
- Raw data (TSV, decimal scores): **94G**
- Gzipped data: **23G**

- Score-decomposed trie: **24G**

Results

Dataset	Raw	gzip	SDT
AOL Queries	209.8	56.3	62.4
Bing Queries	235.6	57.9	61.2
URLs	228.7	54.7	58.6
Unigrams	114.3	44.2	39.8

bits per string-score pair

Performance

- About 10 **microseconds** for top-10 completions
  - Basically the same as retrieving 10 strings from an std::set (red-black tree)
- Why care? Network latency is in the **milliseconds**
- Important if we need to search **several prefixes** for each query
  - Example: approximate completion

Thanks for your attention!

Questions?