

Implementing the Aho-Corasick Automata for Phonetic Search

Ondřej Sýkora

February 28, 2010

Introduction

What it is all about

- ▶ Phonetic search
 - ▶ Searching for words by their pronunciation, not by exact spelling.
 - ▶ Supported languages: Arabic and German
 - ▶ *Soundex approach* – “hashing” the words, comparing hash codes.
 - ▶ Hash code = initial letter + three numbers based on the following letters.
 - ▶ Easy to implement, fast algorithm, but too many false positives.
 - ▶ *CZfind* – phonetic search based on transcription rules and a modified Aho-Corasick automaton.
 - ▶ Good precision, but resource-intensive.

Introduction

What it is all about

- ▶ Aho-Corasick automata
 - ▶ Alphabet Σ
 - ▶ Set of search phrases \mathcal{S}
 - ▶ States, transition rules, backwards edges, ...

- ▶ ... do I really have to explain?

Introduction

Challenges of phonetic search

- ▶ Complexity of phonetic search. . .
 - ▶ Each “phonetic” group of characters is matched by all possible transcriptions

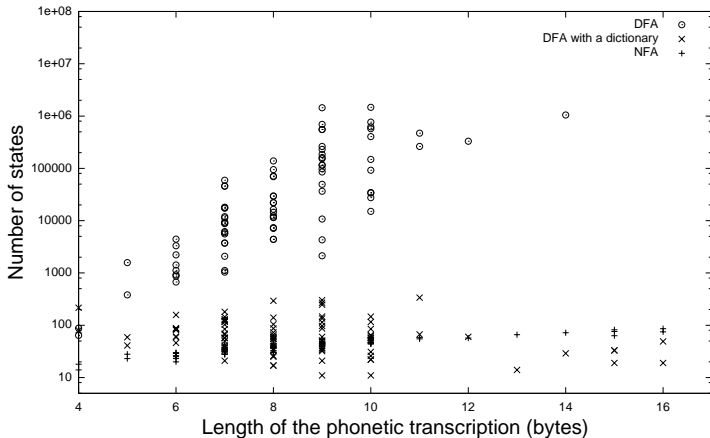
Word	Transcriptions
kála	33
bilmáji	944
kysatun	8547
kalratun	33973
machatatun	721709
tazkyratun	1082157
alkuránulkarýmu	22154993
aliskandrarájatu	430638209

- ▶ The transcription rules are non-deterministic in their nature.
 - ▶ Exponential growth of the number of transcriptions!

Introduction

Challenges of Phonetic Search

- ▶ Exponential growth of the number states of the automata.



Introduction

How to deal with the complexity

- ▶ How to reduce the complexity?
 - ▶ Using special hardware?
 - ▶ Using “more effective” representations?
 - ▶ Avoiding pointless work?

 - ▶ Well, let's see. . .

Using Special Hardware

Motivation

- ▶ Always question your beliefs. . .
 - ▶ Are the universal computers the best possible hardware for string matching?
 - ▶ Can a more flexible architecture support it better?
 - ▶ Field Programmable Gate Arrays (FPGAs) have been successfully used for string matching in networking and bioinformatics (!), how would they support phonetic search?
- ▶ Why consider a different architecture (FPGA)?
 - ▶ The new architecture might be slightly faster (in this task).
 - ▶ The transcription rules are naturally non-deterministic.
 - ▶ The current hardware is strictly deterministic.
 - ▶ Or, at least, that's what we want it to be.

Using Special Hardware

How could the FPGAs help?

- ▶ Idea: The deterministic automata seem to fit nicely
 - ▶ Very easy implementation, fits the hardware
 - ▶ State register for the current state
 - ▶ Transition rules hardcoded
 - ▶ The automata are built outside the FPGA!
- ▶ Idea: Hide the non-determinism in parallelism
 - ▶ Massively parallel computation
 - ▶ Resource intensive (in a different way)
 - ▶ Difficult interoperation with current computers
 - ▶ Obtain the data, obtain the configuration, report results. . .
 - ▶ Used in places, where this does not matter (i.e. network hardware, specialized “computing systems”).

Using Special Hardware

Practical results

- ▶ In case of deterministic, strictly serial calculations, universal CPUs are a better choice.
 - ▶ Faster, cheaper, better tools, easier to develop for, ...
 - ▶ Unfortunately, this is the case of deterministic automata.
- ▶ The FPGAs are more limited in terms of capacity
 - ▶ Compiling larger automata is often impossible

Transcription	Failure rate
≤ 6	0 %
7	63 %
8	88 %
9	89 %
≥ 10	100 %

Using Special Hardware

Practical results

- ▶ Non-deterministic search automata
 - ▶ It is possible to implement non-deterministic regular automata effectively (if they are not too large).
 - ▶ Non-determinism replaced by keeping track of all possible active states.
 - ▶ Performs a single step in constant time
 - ▶ Parallel updates of the activity of states
 - ▶ In terms of time to process a single state can outperform current CPUs significantly
 - ▶ However, the initialization and startup times are in case of FPGAs even more significantly higher.

Using Special Hardware

Conclusions

- ▶ Implementing traditional search algorithms is not effective
 - ▶ Extreme numbers of states
 - ▶ More effective implementation on a CPU
- ▶ Other approaches to string matching might prove more efficient
 - ▶ In case of phonetic search, the non-deterministic automata are a more natural choice
 - ▶ ... but this was the topic of my last presentation...

- ▶ This time, let's stay with the Aho-Corasick automata

Bit-Split Architecture

Motivation

- ▶ Tan, Sherwood, 2006
- ▶ For larger sets of search phrases, the Aho-Corasick automata grow rapidly
 - ▶ Fast implementations are memory-ineffective, and vice versa
- ▶ The most problematic part are the transition tables
 - ▶ Most of the transitions lead to the initial state
 - ▶ in case of phonetic search it is cca 99 %
 - ▶ Memory effective implementation of the tables leads to “slow” lookup
 - ▶ Search trees, hash tables, handling special cases, ...
 - ▶ Implementation with fast lookup is memory ineffective
 - ▶ Requires storing the transitions for all characters in Σ

Bit-Split Architecture

The general idea

- ▶ Consider a binary alphabet. . .
 - ▶ There are at most two transitions from each state
- ▶ Consider binary encoding of the characters in Σ
 - ▶ . . . while keeping the input stream character (byte) oriented
 - ▶ Each bit position forms a separate binary stream.
 - ▶ Transform the original search automaton into a set of automata, one for each binary stream
 - ▶ Run these (possibly smaller) automata in parallel, re-create the matches of the original automaton from the matches of the binary automata
 - ▶ All binary automata must report a match for the original automaton to report a match

Bit-Split Architecture

Splitting the automata

- ▶ Start by creating the original Aho-Corasick automaton A
- ▶ Each state in the automaton A represents a certain string w
 - ▶ A bit position “selects” a binary string from w
- ▶ Create automaton for bit position i
 - ▶ Create the initial state $0_i = \{0\}$
 - ▶ Use the “subset construction”, similar to conversion of non-deterministic regular automata to deterministic to convert the other states
 - ▶ The states of the “binary automaton” are sets of states of the original automaton
 - ▶ The transitions are defined natively by the subset construction
 - ▶ A state of the binary automaton is marked as final iff any of the states of the original automaton that the binary states contains is final

Bit-Split Architecture

Splitting the automata

- ▶ A simple example
 - ▶ A simple alphabet
 - a 01100001
 - b 01100010
 - ▶ Simple search phrases
 - ▶ abba
 - ▶ baba

Bit-Split Architecture

Remarks on effectivity

- ▶ Instead of a single automaton, we got eight, do they have less states in total?
 - ▶ Most certainly no
- ▶ Do at least all of them have they have less states than the original automaton?
 - ▶ Most probably yes, but not necessarily
- ▶ In the “non-deterministic” subset construction, the number of states increases exponentially, is this not the case?
 - ▶ No, the number of states of the binary automaton cannot be greater than the number of states of the original automaton
- ▶ Did we save anything, if not the states?
 - ▶ Yes, the state transition tables are slightly smaller.

Bit-Split Architecture

Finding a match

- ▶ How do we find a match?
 - ▶ The automata for all bit positions must report a match
- ▶ But let's consider more ASCII characters
 - ' 01100000
 - c 01100011
- ▶ ...and the word: 'cba'
- ▶ Getting a match from all of the binary automata is a necessary condition, but not a sufficient condition!
 - ▶ Report a match only if all of the binary automata report a match, and the intersection of their current states is not empty
 - ▶ Can be done "easily" by keeping a vector of matching words for each state ("partial match vector")

Bit-Split Architecture

Conclusions

- ▶ Partial match vectors are ineffective, either ineffective with respect to speed, or memory-ineffective
 - ▶ Binary vectors versus lists
- ▶ With smaller search phrase sets, the Bit-Split architecture can lead to a more space effective implementation (on an FPGA), but it does not avoid the exponential growth
- ▶ For larger search phrase sets, the partial match vectors will quickly become unsustainable

On-demand Aho-Corasick Automata

Motivation

- ▶ In phonetic search, only a small portion of the states is ever visited.
 - ▶ How many times the states are visited (when processing a real 250MB text sample)?
 - ▶ More than 200×10^6 – single state (initial state)
 - ▶ More than 10×10^6 – two states
 - ▶ More than 5×10^6 – three states
 - ▶ More than 10^6 – ten states
 - ▶ More than 1000 – 61 states
 - ▶ At least once – 1089 states
 - ▶ Never visited – 261575 states
- ▶ It doesn't make sense to even consider those 261000 states

On-demand Aho-Corasick Automata

General idea

- ▶ Idea: start with an empty automaton, add states as necessary

On-demand Aho-Corasick Automata

Explained in detail

- ▶ ... on the blackboard. . .

Conclusions

- ▶ Thank you for your attention!
- ▶ Questions? Comments?