Large dictionaries

Paolo Boldi
DSI
LAW (Laboratory for Web Algorithmics)
Università degli Studi di Milan

Term map

- While building the inverted index we build a map from terms to numbers.
- The same kind of map is needed while building the graph: in that case it is a map from URLs to numbers:

`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

- The numbering used is arbitrary, but lexicographic numbering turns out to be convenient for many reasons...

Building the graph

- From the map, one can build a graph by scanning the pages (one at a time), parsing them and determining outgoing links (anchors).
- Given a link to, say, `http://foo/bar/index.html`, we just have to determine the *number* to which this URL corresponds.
- Keeping URLs in an array and doing a binary search is out of questions for reasons of time (1G nodes=30 comparisons) and space (1G nodes=300GB of data!).

The problem

- In other words, we want to represent a function like:

x	$f(x)$
`http://pippo.pluto/x/y`	0
`http://topolino.minnie/w.htm`	1
`http://topolino.minnie/z.htm`	2
...	...

- with a data structure that can compute $f(x)$ *quickly* given x.
- Construction time is not a problem (within reasonable limits)!
- We *don't need* the inverse function.

Short interlude: hash functions

- Given a universe Ω and an integer m, a *an m-bucket hash function for* Ω is a function $h : \Omega \to [m] = \{0, 1, \dots, m-1\}$
- It must be easy to compute and as "injective" as possible.
- In particular, we are interested in its behaviour on a specific set $S \subseteq \Omega$.
- Ideally, if $|S| \leq m$, we would like h to be injective on S. In such a case we say that *h is perfect for S*.
- If moreover $|S| = m$, we say that h is *minimal perfect*.
- Usually, obtaining a minimal perfect hash function is impossible because S is *unknown*. Not in our case, though...
- We will present a technique introduced by Majewski, Wormald, Havas and Czech.

How to make a perfect hash minimal

A general problem: you have a *perfect* hash $h : \Omega \to \{0, 1, \ldots, m-1\}$ (for $S \subseteq \Omega$) and you want to make it *minimal*, i.e., a map $g : \Omega \to \{0, 1, \ldots, n-1\}$.

- One way to do so is to use an extra vector $R[]$ of m bits where $R[i]$ is a 1 iff $h(x) = i$ for some $x \in S$.
- The vector contains exactly n ones (because h is injective).
- Then, you can define $g(x)$ as the number of 1's before $R[h(x)]$.
- Counting the number of 1's before a given position in a bitvector is called a *ranking problem*.
- It is possible to rank a bitvector of length m in constant time using only $o(m)$ extra bits.

Short interlude: string hashing

- As a concrete example: let $\Omega = \Sigma^{\leq w}$ (the set of all strings of length $\leq w$ on an alphabet Σ);
- Let m be an integer
- Let us draw w weights (at random) between 0 and $m - 1$:

3	12	7	41	33	...	5

- Given a string $x \in \Omega$

n	i	n	o

  we look at it as a sequence of w numbers (padding it with zeroes at the end):

110	105	110	111	0	...	0

- $h(x)$ is defined multiplying each character to the corresponding weight, summing up and taking the result modulo m: $(3 \times 110 + 12 \times 105 + \dots) \bmod m$.

The MWHC Algorithm: 1

- Take some $m \geq n$, and choose *uniformly at random* two hash functions h_1 and h_2 from strings to $\{0, 1, \ldots, m-1\}$
- For example

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32111
...	...	...	...

- Build a graph whose vertices are $\{0, 1, \ldots, m-1\}$ and with an edge for every string: the edge for string x connects $h_1(x)$ and $h_2(x)$.
- Special cases: degenerate arcs? coincident arcs? cyclic graph? We throw h_1 and h_2 away and generate another pair.
- **Theorem**: if m is large enough ($m \geq 1.75n$) with high probability (with an expected number of $e^{4/5} \approx 2$ attempts) we will get a graph satisfying the constraints.

The MWHC Algorithm: 2

- Let's go back to our example:

x	$f(x)$	$h_1(x)$	$h_2(x)$
`http://pippo.pluto/x/y`	0	231	3443
`http://topolino.minnie/w.htm`	1	32	5534
`http://topolino.minnie/z.htm`	2	231	32111
...	...	...	...

- We associate to every vertex a variable (the variable associated to 231 is x_{231}) and look at the graph as a system of modular equations:

$$\begin{aligned}(x_{231} + x_{3443}) \bmod m &= 0\\ (x_{32} + x_{5534}) \bmod m &= 1\\ (x_{231} + x_{32111}) \bmod m &= 2\end{aligned}$$

- **Theorem**: If the graph is acyclic, this system admits a solution (it can be found with a DFS).

The MWHC Algorithm: 3

- We store an array $x[]$ with the solution found: this array has the property that, for every string x:

$$(x[h_1(x)] + x[h_2(x)]) \bmod m = f(x).$$

- So to compute $f(x)$ we just need $x[]$ ($m \approx 2n$ integers, $2n \log n$ bits) and the two weight vectors for computing the two hash functions (the size of this is *independent* from n, it just depends on the string length)
- Remarks:
  - we are not storing the strings, so it will be IMPOSSIBLE to compute f^{-1}
  - if we try to compute $f(x)$ for some $x \notin S$ we will obtain *something*, although in theory this would be an "error": it makes sense to query f only with strings in S (*it is not a dictionary!*).
  - Observe that the MWHC construction gives much more than a simple minimal perfect hash: it is an *order preserving* one

The real MWHC algorithm

- The same idea can be applied to more than two hash functions, working with hypergraphs instead of graphs.
- The advantage is that we can get acyclicity with *less* vertices (in the case of graphs, we need $m \geq 1.75n$).
- It can be shown that the optimum is obtained with *three* hash functions, in which case we just need $m \geq 1.21n$.
- The same idea is actually more general: every function $h : S \subseteq \Omega \to \Psi$ can be represented using only $1.21n \log |\Psi|$ bits, with constant-time evaluation

Signing the map

- If you want to avoid that $f(x)$ returns a sensible result when $x \notin S$, you can *sign the map*
- Compute some signature (e.g., CRC) $\sigma(x)$ for every $x \in S$, and store them in an array $s[]$ ($\sigma(x)$ is stored in $s[f(x)]$)
- On input x, compute $f(x)$ and check if $s[f(x)] = \sigma(x)$:
  - if not, return -1 (certainly $x \notin S$)
  - otherwise, return $f(x)$.
- The latter can be a false positive, but with low probability (even zero probability, if $\sigma(-)$ is taken to be the identity).

A perfect hash

- If you just need a *perfect* hash...
- You proceed exactly like explained, and you get a system of equations, one per edge:

$$\begin{aligned}(x_{231} + x_{3443}) \bmod ??? &= ???\\ (x_{32} + x_{5534}) \bmod ??? &= ???\\ (x_{231} + x_{32111}) \bmod ??? &= ???\end{aligned}$$

- Each equation is of the form

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod ??? = ???$$

for some $w \in S$. Acyclicity guarantees that it is possible to *sort* these equations in some order so that every equation contains a variable that never appeared before.
- Let us call the corresponding subscript (which is either h_1 or h_2) the *hinge* for that equation.

A perfect hash (cont'd)

- Then, it is possible to write the system so that

$$(x_{h_1(w)} + x_{h_2(w)}) \bmod 2 = 0 \text{ or } 1$$

depending on whether the hinge is $h_1(w)$ or $h_2(w)$.

- The system has a solution (because of acyclicity). In other words, you can determine a vector $x[]$ (of bits) such that, for every $w \in S$,

$$1 + (x[h_1(w)] + x[h_2(w)]) \bmod 2$$

gives an index $j(w)$ such that th $h_{j(w)}(w)$ are all distinct.

- In other words, the function $g(w) := h_{j(w)}(w)$ is a perfect hash (not a minimal one).
- Just needs n bits (besides the weights for the two hashes!). Actually, $2n$ bits for the 3-hypergraph.

A minimal perfect hash

- Combining the construction explained (using n bits) with a bitvector of m bits for ranking, we obtain a minimal perfect hash.
- This structure uses $n + m + o(m) < 3n$ bits, plus the (constant) bits needed to store the hash functions h_1 and h_2.
- Note that it is minimal and perfect, but *not* order preserving.