# The Webgraph Framework I: Compression Techniques

Edward Park

# WebGraph

- The Web graph is gigantic
  - >3 billion nodes, >50 billion arcs at time of publication
- How do we compress the Web?

- Have to deal with both the web and its transpose
  - Transpose = graph with the same nodes, but direction of all arcs are reversed
  - Useful in several ranking algorithms

# Properties of the Web graph

- Locality
  - Most links direct to another page in the same host
  - The source URL and target URL are close together lexicographically
- Similarity
  - Pages close to each other have many common successors
  - Many links are copied from one page to another in the same host

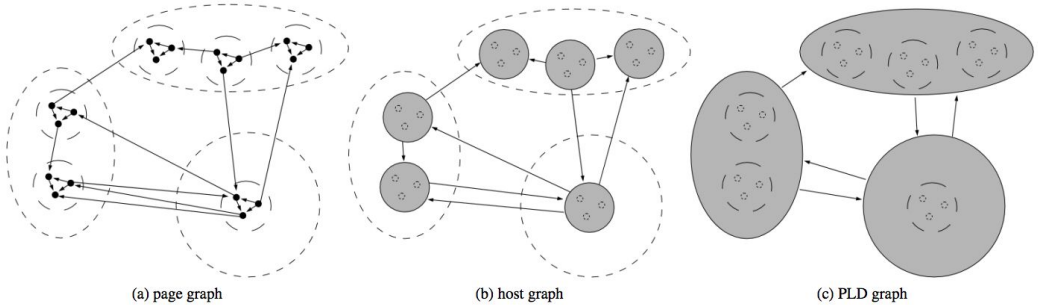
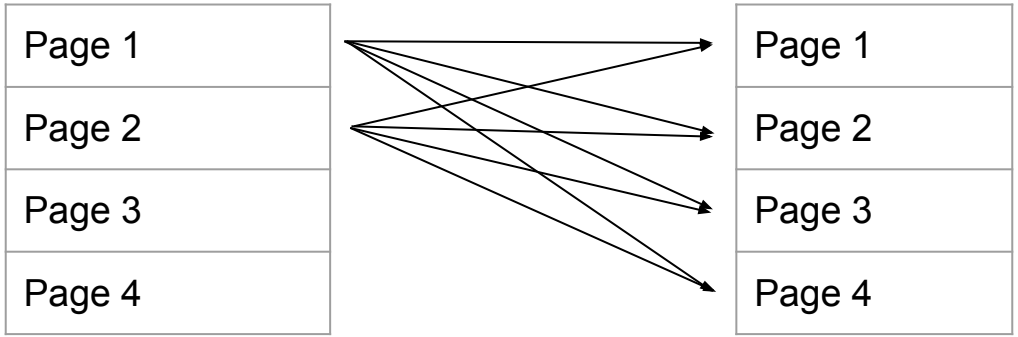



Figure 1: Different aggregation levels of the graph

# Properties of the Web graph

- Similarity is very concentrated
  - Either two lists have nothing in common, or they share large parts of their successor lists
- Consecutivity is common
  - Many links within a page are consecutive (with respect to lexicographic order)
  - Most pages contain sets of navigational links that point to a fixed level of the hierarchy
  - In the tranposed graph, important pages (ie home page) are pointed to by most pages
- Consecutivity is the dual of distance-one similarity
  - If two consecutive pages have very similar successor lists, then the tranposed Web graph has large intervals



# Gaps in Increasing Sequences of Successors

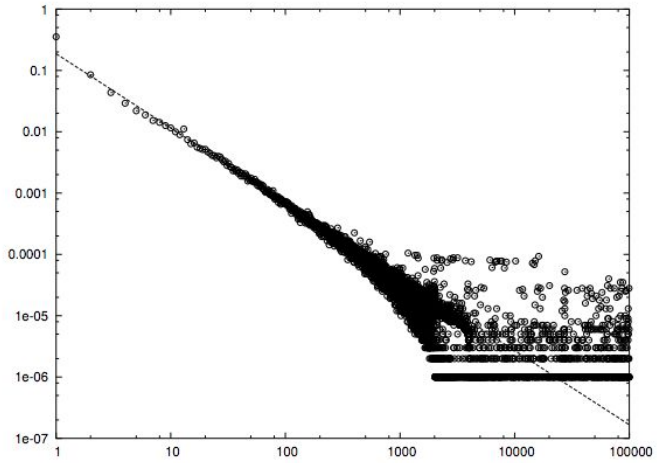



**Figure 1: Distribution of gaps in a 18.5 Mpages snapshot of the .uk domain. The scale is logarithmic on both axes, and the line displays a power law with exponent 1.21.**

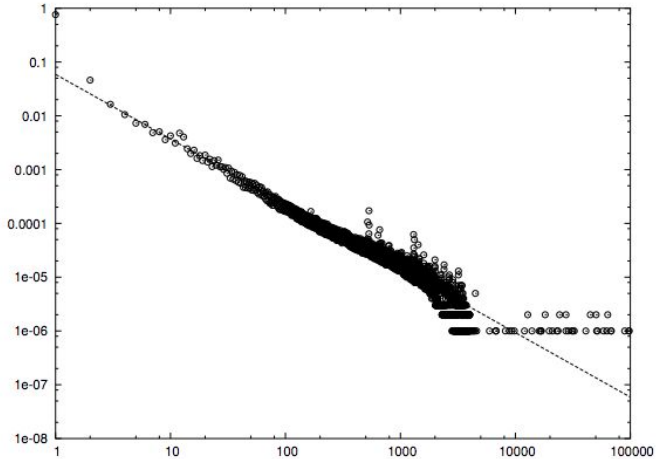



**Figure 2: Distribution of gaps in the transpose of a 18.5 Mpages snapshot of the .uk domain. The scale is logarithmic on both axes, and the line displays a power law with exponent 1.20 (modulo a scaling factor).**

# Compression, Part I

AKA, an exercise in abusing every conceivable compression tactic you can think of

- Naive representation - adjacency lists
- Using gaps - instead of storing the successors, store the differences between adjacent successors
  - The first element might be negative; to avoid this, use the map

$$\nu(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ 2|x| - 1 & \text{if } x < 0. \end{cases}$$

| Node | Outdegree | Successors |
|---|---|---|
| ... | ... | ... |
| 15 | 11 | 13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034 |
| 16 | 10 | 15, 16, 17, 22, 23, 24, 315, 316, 317, 3041 |
| 17 | 0 | |
| 18 | 5 | 13, 15, 16, 17, 50 |
| ... | ... | ... |

**Table 1: Naive representation using outdegrees and adjacency lists.**

| Node | Outdegree | Successors |
|---|---|---|
| ... | ... | ... |
| 15 | 11 | 3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718 |
| 16 | 10 | 1, 0, 0, 4, 0, 0, 290, 0, 0, 2723 |
| 17 | 0 | |
| 18 | 5 | 9, 1, 0, 0, 32 |
| ... | ... | ... |

**Table 2: Representation using gaps.**

# Compression, Part II

- Reference compression - instead of showing the adjacency list S(x) directly, say it is a modified version of some previous list S(y) [the reference list]
  - Uses a sequence of bits to indicate which successors in S(y) are also in S(x) or not
  - Also contains a list of extra nodes for any pages in S(x) that are not in S(y)
- The reference number r = x - y
  - Assume there is a fixed window size W, and r is chosen as the value between 0 and W that gives the best compression

| Node | Outdegree | Successors |
|---|---|---|
| ... | ... | ... |
| 15 | 11 | 13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034 |
| 16 | 10 | 15, 16, 17, 22, 23, 24, 315, 316, 317, 3041 |
| 17 | 0 | |
| 18 | 5 | 13, 15, 16, 17, 50 |
| ... | ... | ... |

**Table 1: Naive representation using outdegrees and adjacency lists.**

| Node | Outd. | Ref. | Copy list | Extra nodes |
|---|---|---|---|---|
| ... | ... | ... | ... | ... |
| 15 | 11 | 0 | | 13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034 |
| 16 | 10 | 1 | 01110011010 | 22, 316, 317, 3041 |
| 17 | 0 | | | |
| 18 | 5 | 3 | 11110000000 | 50 |
| ... | ... | ... | ... | ... |

**Table 3: Representation using copy lists.**

# Compression, Part III

- Differential compression - the copy list is seen as an alternating sequence of 1- and 0-blocks, where we specify the length of each block
  - Preceded by a block count telling the number of blocks
  - First block is a 1-block (so say first block has length 0 if we start with 0-block)
  - All block lengths are decremented by 1 (except the first block)
  - Omit the last block length because its value can be inferred
- This allows us to code a link in less than one bit!

| Node | Outd. | Ref. | Copy list | Extra nodes |
|---|---|---|---|---|
| ... | ... | ... | ... | ... |
| 15 | 11 | 0 | | 13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034 |
| 16 | 10 | 1 | 01110011010 | 22, 316, 317, 3041 |
| 17 | 0 | | | |
| 18 | 5 | 3 | 11110000000 | 50 |
| ... | ... | ... | ... | ... |

**Table 3: Representation using copy lists.**

| Node | Outd. | Ref. | # blocks | Copy blocks | Extra nodes |
|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... |
| 15 | 11 | 0 | | | 13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034 |
| 16 | 10 | 1 | 7 | 0, 0, 2, 1, 1, 0, 0 | 22, 316, 317, 3041 |
| 17 | 0 | | | | |
| 18 | 5 | 3 | 1 | 4 | 50 |
| ... | ... | ... | ... | ... | ... |

**Table 4: Representation using copy blocks.**

# Compression, Part IV

- Intervals - each list of extra nodes compressed as a list of integer intervals + a list of residuals
  - Only care about an interval if its length is above a threshold $L_{min}$
  - Each interval represented by its left extreme and its length
  - Left extremes compressed using difference b/w left extreme + previous right extreme minus 2
  - Interval lengths decremented by threshold $L_{min}$

| Node | Outd. | Ref. | # blocks | Copy blocks | Extra nodes |
|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... |
| 15 | 11 | 0 | | | 13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034 |
| 16 | 10 | 1 | 7 | 0, 0, 2, 1, 1, 0, 0 | 22, 316, 317, 3041 |
| 17 | 0 | | | | |
| 18 | 5 | 3 | 1 | 4 | 50 |
| ... | ... | ... | ... | ... | ... |

| Node | Outd. | Ref. | # blocks | Copy blocks | # intervals | Left extremes | Length | Residuals |
|---|---|---|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 15 | 11 | 0 | | | 2 | 0, 2 | 3, 0 | 5, 189, 111, 718 |
| 16 | 10 | 1 | 7 | 0, 0, 2, 1, 1, 0, 0 | 1 | 600 | 0 | 12, 3018 |
| 17 | 0 | | | | | | | |
| 18 | 5 | 3 | 1 | 4 | 0 | | | 50 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

# Compression, Part V

$$d\left[\overbrace{r\left[b\ B_1 \cdots B_b\ \right]_{r>0}}^{W>0}\ \left[\overbrace{i\ E_1 L_1 \cdots E_i L_i}^{L_{\min}<\infty}\ R_1\ \cdots\ R_k\right]_{\beta<d}\right]_{d>0}$$

| Datum | Meaning | Notes | Represented as… |
|---|---|---|---|
| $d$ | Outdegree | $d \geq 0$ | |
| $r$ | Reference number | $0 \leq r \leq W$ | |
| $b$ | Block count | $b \geq 0$ | |
| $B_1, \ldots, B_b$ | Blocks | $B_1 \geq 0, B_2, \ldots, B_b > 0$ | $B_1, B_2 - 1 \ldots, B_b - 1$ |
| $i$ | Interval count | $i \geq 0$ | |
| $E_1, \ldots, E_i$ | Left extremes | $E_{k+1} \geq E_k + L_k + 1$ | $v(E_1 - x), E_2 - E_1 - L_1 - 1, \ldots, E_i - E_{i-1} - L_{i-1} - 1$ |
| $L_1, \ldots, L_i$ | Interval lengths | $L_1, \ldots, L_i \geq L_{\min}$ | $L_1 - L_{\min}, \ldots, L_i - L_{\min}$ |
| $R_1, \ldots, R_k$ | Residuals | $0 \leq R_1 < R_2 < \cdots < R_k$ | $v(R_1 - x), R_2 - R_1 - 1, \ldots, R_k - R_{k-1} - 1$ |

**Table 6: Data describing the adjacency list of node $x$.**

- Note this is self-delimiting (except list of residuals)
- Each of these ideas abusing similarity + consecutivity

# Reference Counting

- Recall node x will refer to some node y in the past window size W
  - To access the adjacency list of node x, we have to decompress the adjacency list of node y
- Sequential accesses = everything fine
- Random accesses = problem
  - X -> y -> z -> ...
  - Leads to a reference chain - could be arbitrarily long, leading to massive slowdown
- Put a limit to the length of reference chains - R, the max reference count
  - When looking for references, consider all nodes x-1, ..., x-W that do not produce reference chains longer than R
  - Larger R = better compression, longer random access times

# Reference Counting

| 18.5 Mpages, 300 Mlinks from .uk | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $R$ | Average reference chain | | | Bits/node | | | Bits/link | | |
| | $W=1$ | $W=3$ | $W=7$ | $W=1$ | $W=3$ | $W=7$ | $W=1$ | $W=3$ | $W=7$ |
| $\infty$ | 171.45 | 198.68 | 195.98 | 44.22 | 38.28 | 35.81 | 2.75 | 2.38 | 2.22 |
| 3 | 1.04 | 1.41 | 1.70 | 62.31 | 52.37 | 48.30 | 3.87 | 3.25 | 3.00 |
| 1 | 0.36 | 0.55 | 0.64 | 81.24 | 62.96 | 55.69 | 5.05 | 3.91 | 3.46 |
| Tranpose | | | | | | | | | |
| $\infty$ | 18.50 | 25.34 | 26.61 | 36.23 | 33.48 | 31.88 | 2.25 | 2.08 | 1.98 |
| 3 | 0.69 | 1.01 | 1.23 | 37.68 | 35.09 | 33.81 | 2.34 | 2.18 | 2.10 |
| 1 | 0.27 | 0.43 | 0.51 | 39.83 | 36.97 | 35.69 | 2.47 | 2.30 | 2.22 |
| 118 Mpages, 1 Glinks from WebBase | | | | | | | | | |
| $R$ | Average reference chain | | | Bits/node | | | Bits/link | | |
| | $W=1$ | $W=3$ | $W=7$ | $W=1$ | $W=3$ | $W=7$ | $W=1$ | $W=3$ | $W=7$ |
| $\infty$ | 85.27 | 118.56 | 119.65 | 30.99 | 27.79 | 26.57 | 3.59 | 3.22 | 3.08 |
| 3 | 0.79 | 1.10 | 1.32 | 38.46 | 33.86 | 32.29 | 4.46 | 3.92 | 3.74 |
| 1 | 0.28 | 0.43 | 0.51 | 46.63 | 38.80 | 36.02 | 5.40 | 4.49 | 4.17 |
| Tranpose | | | | | | | | | |
| $\infty$ | 27.49 | 30.69 | 31.60 | 27.86 | 25.97 | 24.96 | 3.23 | 3.01 | 2.89 |
| 3 | 0.76 | 1.09 | 1.31 | 29.20 | 27.40 | 26.75 | 3.38 | 3.17 | 3.10 |
| 1 | 0.29 | 0.46 | 0.54 | 31.09 | 29.00 | 28.35 | 3.60 | 3.36 | 3.28 |

**Table 7: Experimental data about reference chains with $L_{\min} = 3$ and using $\zeta_3$ for residuals. The .uk data were gathered using UbiCrawler; the WebBase data refer to the 1/2001 general crawl.**

# Offset Array

- Need to keep an auxiliary vector of offsets
  - Offsets expressed as bit-displacements for flexibility and scalability
- Problem: We have a limited amount of central memory, can't store the entire offset array when doing random accesses
- Solution: Load the offset array partially - only keep the offsets of nodes J, 2J, 3J, ... for a parameter J [the jump]
  - J not a compression parameter, only fixed when reading the graph into memory
  - Larger J = smaller memory usage, longer random access times
- Subproblem: skipping over an adjacency list is nontrivial
  - Solution: Load adjacency lists into memory as blocks of J lists each. Store the J outdegrees at the beginning of each block.

# Lazy Iteration

- Problem: Computing referenced lists is pretty expensive
- Solution: Don’t do it unless you need to!
- WebGraph enumerates successors using lazy iterators
  - Each time an iterator is required to produce a new successor, check whether it can do it using local data (intervals + residuals) - if not, then pass the request to the iterator of the reference node
- No list is ever expanded into memory
  - The only state kept by the recursive stack is intervals + blocks
- Drastically improves performance
  - R = ∞ is fastest sequential access time - main cost is memory access + higher compression speeds it up

# Lazy Iteration

| 18.5 Mpages, 300 Mlinks from .uk | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $R$ | Graph size (MiB) | Offset-array size (MiB) | | | | Link access time (ns) | | | |
| | | seq. | $J=1$ | $J=2$ | $J=4$ | seq. | $J=1$ | $J=2$ | $J=4$ |
| $\infty$ | 79.0 | | | | | 198 | 31 237 | 35 752 | 43 699 |
| 3 | 106.6 | – | 141.3 | 70.7 | 35.3 | 206 | 611 | 753 | 886 |
| 1 | 122.9 | | | | | 233 | 442 | 491 | 605 |
| Transpose | | | | | | | | | |
| $\infty$ | 70.3 | | | | | 150 | 2 382 | 2 873 | 2 961 |
| 3 | 74.6 | – | 141.3 | 70.7 | 35.3 | 171 | 342 | 424 | 516 |
| 1 | 78.8 | | | | | 183 | 234 | 312 | 374 |

**Table 8: Experimental data about access time, obtained on 512 MiB 2.4 GHz Pentium for a 18.5 Mpages snapshot of the .uk domain.**

# Conclusion

- The Web graph has high locality, similarity, and consecutivity
  - Abuse these to have nice compression schemes
- Limit the size of the reference chain + do lazy iteration to speed up performance
- WebGraph achieves compression ratios about 2x the best results in the LINK database