令和元年度卒業研究論文

URLの情報指向型クラシフィケーション

2020年2月7日(金)

指導教員 井上一成 教授

明石工業高等専門学校電気情報工学科

報告者 E1533 西 総一朗

目次

第 1 章　序論　1

1.1　TCP/IP の課題　1

1.2　情報指向ネットワーク　2

1.3　ICN の実用化に向けた課題　3

1.4　本研究の目的　3

1.5　本論文の構成　4

第 2 章　解析手法　5

2.1　URL の構造　5

2.2　ICN におけるコンテンツ名　6

2.3　性能の評価手順　7

2.3.1　解析データ　7

2.3.2　ハッシュアルゴリズム　9

2.3.3　ハッシュテーブル　10

2.4　解析プログラム　12

2.4.1　前処理　12

2.4.2　コンテンツ名への変換　12

2.4.3　ハッシュテーブルとポインタテーブル生成　12

2.4.4　衝突数の解析　12

第 3 章　性能評価　13

3.1　解析データの妥当性　13

3.2　eTLD の分布　14

3.3　ハッシュアルゴリズムの比較　14

3.4　eTLD ごとのハッシュ衝突率　16

3.5　eTLD に Root を加えた場合　17

3.6　まとめ　18

第 4 章　結論　21

参考文献　23

付録 A　作成したプログラム　24

A.1 重複を削除する 24
A.2 ランダムにサンプルを抽出 28
A.3 ICN-URL に変換する 32
A.4 ハッシュの計算 38
A.5 ハッシュの衝突数を数え上げる 43
A.6 ハッシュテーブルとポインタテーブルの作成 46

第 1 章

序論

1.1 TCP/IP の課題

1983 年から今日のインターネットと呼ばれているネットワークにおいて通信プロトコル TCP/IP がデファクトスタンダードとなった [1]．約 20 年前のインターネットのトラヒックや利用形態は現在とは大きく異なっている．1992 年の全世界のインターネットトラフィックは 1 日あたり約 100 GB であったが, その 10 年後の 2002 年には 1 秒あたり 100 GB に増え，2017 年には 1 秒あたり 45,000 GB 以上に到達した．また利用形態も 2017 年においてはトラヒックの 75% をビデオコンテンツが占めている．Cisco によると全世界のインターネットトラヒックは 2022 年には 150,700 GB/秒となりその 82% をビデオコンテンツが占めると予測されている [2]．

また，インターネットの使用目的も変遷している．当初はインターネットを高性能コンピュタあるいは高性能プリンタを利用するように，様々なリソースを遠隔から共有することが主な目的であった．現在は情報の共有，情報の取得といった情報のやり取りが中心となっている．それに伴って，通信形態も変化している．従来の TCP/IP はホスト中心の Host-to-Host の通信形態であり，IP プロトコルは位置情報であるネットワークアドレスを用いてホストアドレスを指定するというロケーション・オリエンテッド*1な通信であった．ところが，現在は情報をユーザに送るというインフォメーション・セントリック*2な通信形態に変わりつつある．

このように TCP/IP の通信形態と現在のインターネットに求められている通信形態との間の差が広がっている．そこで，情報の効率的な取得のために P2P*3や CDN*4などの新しいプロトコルが提案された．しかし，これらはロケーション・セントリックな TCP/IP ネットワーク上のプロトコルであるので本質的な解決ではない．本来，情報を取得するという行為に対して，ネットワークアドレスやホストアドレスなどを意識する必要はなく，もし近くにある通信機器が当該コンテンツ (情報)*5を持っておりそこから情報を取得できるなら，それはより効率的であり将来の通信量増大にも対応できると考えられる．そこで，情報を効率的に取得するために情報指向ネットワーク: Information-Centric-Network (ICN)[3] というプロトコル体系が提案された [4]．

---

*1 Location-oriented: 地理的指向な

*2 Information-Centric: 情報指向な

*3 Peer to Peer: インターネットにおいて一般的に用いられるクライアント・サーバ型モデルでは，データを保持・提供するサーバとそれに対してデータを要求・アクセスするクライアントという 2 つの立場が固定されているのに対して，各ピアに対して対等 にデータの提供及び要求・アクセスを行う自立分散型のネットワークモデル.

*4 Content Delivery Network: 頻繁に使われる Web サイトがあると一つのノード（サーバ）のみでは負荷が集中するためいくつかのノードにデータを分散しておき，各ユーザは分散したノードに接続して情報を取得するという方法.

*5 参考文献 [3] では情報 (Information) とコンテンツ (Contents) は同様の意味で用いられている．本稿でも同様の意味で用いる．

1.2 情報指向ネットワーク

情報指向ネットワーク (ICN) においてユーザはサーバの IP アドレスではなくコンテンツ名を指定してコンテンツ取得要求を行う．そのコンテンツ要求を受け取った近隣のルータやノードが当該コンテンツを保持していた場合，それらはユーザに対してそのコンテンツを直接転送する．ICN では情報を保持している者をパブリッシャ (Publisher)，情報の取得要求を出すものをサブスクライバ (Subscriber) と呼び，各コンテンツ (情報) に対してコンテンツ名 (名前) が対応付けられている．

ICN へのアプローチとして様々な研究がなされているが，現在最も多くの研究者により研究されている Named Data Networking (NDN)[5] 及びその前身である Content Centric Networking (CCN)[6] を代表的な ICN アーキテクチャとして述べる．CCN アーキテクチャはパロアルト研究所[*6]により研究されている ICN の先駆となった本格的なアーキテクチャである．また，US Future Internet Architecture プログラム[*7]によって資金提供された NDN プロジェクトは，CCN アーキテクチャをさらに発展させたものである．

■コンテンツ名　NDN におけるコンテンツ名の命名規則は階層構造になっており，現在のインターネットで流通している識別子である Uniform-Resource-Locator (URL) に似ている．たとえば，コンテンツ名は `/aueb.gr/ai/main.html` となる．ただし，コンテンツ名は必ずしも URL とは一致せず，最初のセクション[*8]は DNS 名または IP アドレスなどの形式である必要もない．つまり NDN では，各セクションについての具体的な規格は定義されていない．またコンテンツ取得要求において，要求されたコンテンツ名のプレフィックスの名前を持つ情報と一致すると見なされる．たとえば，`/aueb.gr/ai/main.html/_v1/_s1` は `/aueb.gr/ai/main.html` という名前のコンテンツと一致する．これは要求されたコンテンツの初版であり，コンテンツを分割したセグメントの最初のデータを表している．このデータを受信したあとに Subscriber は `/aueb.gr/ai/main.html/_v1/_s2` により次のセグメントを要求することや，新たなバージョンを要求することもできる．このようにコンテンツを分割して扱う際にはコンテンツ名にそのメタデータを付与することが可能である．

■名前解決とデータルーティング　NDN において，Subscriber はコンテンツを取得する際はコンテンツ取得要求である INTEREST パケットを発行して，Publisher からの DATA パケットの形式で到着するコンテンツを取得する．INTEREST/DATA パケットは，それぞれ要求/転送されるコンテンツのコンテンツ名を持つ．Fig. 1.1 に示すように，すべてのパケットは Content Router (CR) によってホップバイホップ (hop by hop) で転送される．各 CR には 3 つのデータ構造 (Forwarding Information Base (FIB), Pending Interest Table (PIT), Content Store (CS)) がある．FIB は，INTEREST パケットを適切なデータソースに転送するために使用するインターフェイスとコンテンツ名をマッピングする．PIT は，保留中の INTEREST パケットが到着した受信インターフェイス，つまり一致する DATA パケットが転送されてきたときに返送するインターフェースとコンテンツ名をマッピングすることで INTEREST パケットを追跡する．最後に，CS は CR を通過したコンテンツのローカルキャッシュとして機能する．

INTEREST パケットが到着すると，CR はコンテンツ名を抽出し要求されたプレフィックスと一致する名前を持つ CS のコンテンツを探す．CS でキャッシュが見つかった場合，すぐに DATA パケットとして受

[*6] Palo Alto Research Center (PARC)：アメリカ合衆国のカリフォルニア州パロアルトにある研究開発企業
[*7] NSF FUTURE INTERNET ARCHITECTURE PROJECT (http://www.nets-fia.net/)
[*8] ”/”で区切られた部分をセクションと呼ぶ．

信インターフェイスを介して返送され，INTEREST パケットは破棄される．それ以外の場合，CR はこの INTEREST パケットを転送するインターフェースを決定するために，FIB で最長プレフィックス検索を実行する．FIB でエントリが見つかった場合，CR は PIT に INTEREST パケットの受信インターフェイスとコンテンツ名を記録し，FIB が示す CR に INTEREST パケットを転送する．Fig. 1.1 では，Subscriber は `/aueb.gr/ai/new.htm` という名前の INTEREST パケットを送信する (矢印 1〜3)．PIT にコンテンツ名のエントリが既に含まれている場合，つまりこのコンテンツが既に要求されている場合，CR は受信インターフェイスをこの PIT エントリに追加し，INTEREST パケットを破棄する．

要求されたコンテンツ名に一致するコンテンツが Publisher または CS で見つかった場合，INTEREST パケットは破棄され，コンテンツは DATA パケットとして返送される．この DATA パケットは，PIT で維持されている状態に基づいてホップバイホップ方式で Subscriber に転送される．具体的には，CR は DATA パケットを受信すると，対応するコンテンツを CS に保存し，PIT で最長プレフィックス検索を実行して，DATA パケットに一致するエントリを見つける．PIT エントリに複数のインターフェイスがある場合，DATA パケットが複製され，マルチキャスト配信が実現される．最後に，CR は DATA パケットをこれらのインターフェイスに転送し，PIT からエントリを削除する (矢印 4〜6)．PIT に一致するエントリがない場合，CR は DATA パケットを重複データとして破棄する．NDN では，DATA パケットは INTEREST パケットによって PIT に残された経路に従うため，名前解決とデータルーティングは対称である [7]．

1.3 ICN の実用化に向けた課題

ICN における Contents Router (CR) は未だに研究段階にありソフトウェアとして参照実装[*9]はあるが，ハードウェアとして実装されたものはない．ICN の実用化を行うため，CR のハードウェアによる実装が不可欠である．ハードウェア実装に向けた課題として，TCP/IP における IP アドレスの代わりにコンテンツ名を用いて名前解決とルーティングを行うため CR での処理が複雑であり，各テーブルに必要な記憶容量も増加するという点が挙げられる．この問題を解決するためにコンテンツ名に対してハッシュ化[*10]を行うことによりルーティングに用いる識別子の圧縮などが研究されている [8][9]．しかし，これらは既存のハッシュアルゴリズムを用いているため，ハードウェアで実装するには複雑である．また，ICN におけるコンテンツ名のハッシュ化という用途では完全に衝突のないハッシュを用いるより，多少の衝突を許容しながらも高速に計算可能なハッシュアルゴリズムが求められる．この多少の衝突を許容するために各テーブルにおける新たなデータ構造による検索手法も求められる．

1.4 本研究の目的

本研究では，上記課題を解決するために新たなデータ構造による検索手法と高速で軽量なハッシュアルゴリズムを提案，検証することである．コンテンツ名はランダムな文字列ではなくある程度自然言語的な規則があるのでそれを用いることで軽量化を図る．

[*9] Cefore: `https://cefore.net/` NICT により開発された CCNx 準拠の TCP/IP 上で CCN パケットをシミュレートするソフトウェアルータ

[*10] 本稿ではあるデータを規則に則って処理したときに出力されるものをハッシュ，その規則をハッシュアルゴリズム，またこの処理を行うことをハッシュ化と呼ぶ．

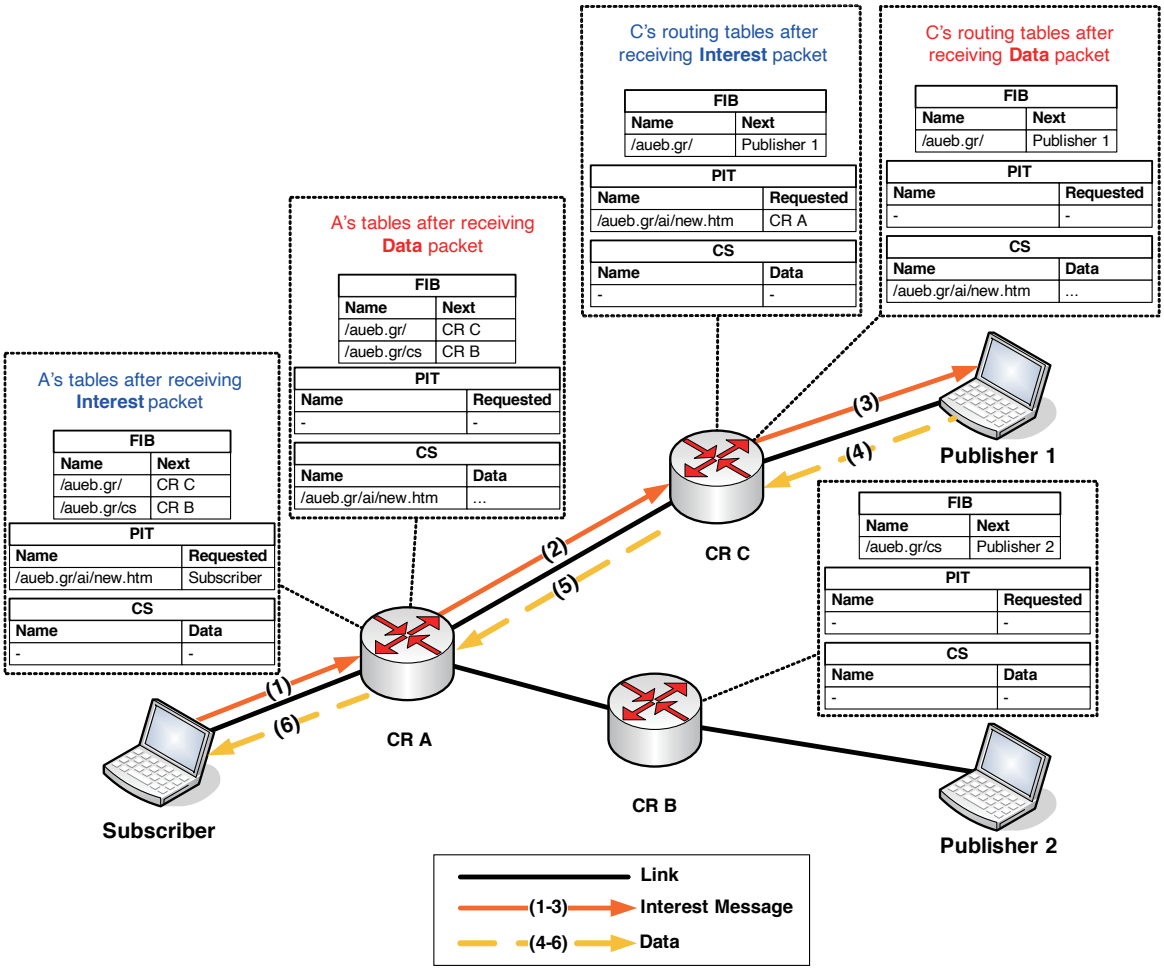

Fig. 1.1 The CCN/NDN architecture. CR stands for Content Router, FIB for Forwarding Information Base, PIT for Pending Interest Table, CS for Content Store (Excerpt from[7]).

1.5 本論文の構成

本論文は本章を含め 4 章で構成されている．第 2 章では，URL の階層構造と ICN のコンテンツ名の関係を述べたのちハッシュアルゴリズムの解析方法について説明する．第 3 章では，性能評価の結果を記述する．最後に，第 4 章において，本論文を通して得られた結果をまとめる．

第 2 章

解析手法

2.1 URL の構造

NDN のコンテンツ名は URL に似ているので URL の階層構造について調べる．Uniform-Resource-Locator(URL) は RFC1738[10] により規格化されており，リソースの位置を特定するために用いられる．HTTP URL の構造は

<scheme>://<host>:<port>/<path>?<searchpart>

である．また，Table 2.1 に各変数の説明を示す．

Table 2.1 HTTP URL variables description[10].

Variables	Description
<scheme>	http, https などのプロトコルを表す．
<host>	ネットワークホストのドメイン名，または IP アドレス．ドメイン名は”.”によって区切られるドメインラベルの連続．
<port>	接続先のポート番号．デフォルトのポート番号 (80, 443) 以外のポートを指定するときのみコロンで区切って続ける．
<path>	HTTP セレクタでリソースの位置を指定する．
<searchpart>	検索パラメータ．これが省略されたときは”?”も省略される．

URL は host 部によって階層構造に分けることができる．host 部はドメイン名または IP アドレスによって構成されているが，ここでは一般に多く使われているドメイン名のみに注目する．Domain Name System (DNS) は木構造でありドメイン名にはその階層構造が反映されている．前述のドメイン名とは実際には Fully Qualified Domain Name (FQDN)[11] のことであり，Fig. 2.1 のような構造である．FQDN は”.”で区切られた各ラベルの連続であり右端の”.”がルートとなる．ルートから順に各ラベルは Top-Level Domain (TLD), Second-Level Domain (SLD), 3rd-Level Domain (3rdLD), 4th-Level Domain (4thLD), ... のように名前がつけられている．また左端のラベルを Host Name と呼び，それ以外のラベルをまとめて Domain Name と呼ぶ．

Top-Level Domain (TLD) には country code TLD (ccTLD), generic TLD (gTLD), restricted generic TLD (grTLD), sponsored TLD (sTLD) などがありそれぞれ ICANN により管理されている [12]．それぞれ

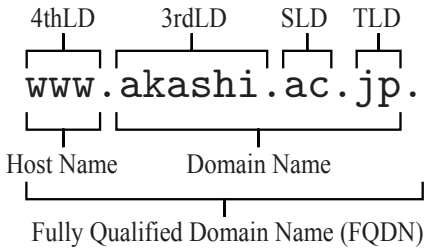

Fig. 2.1 Fully Qualified Domain Name (FQDN) structure. FQDN ends with a period. TLD stands for Top-Level Domain, SLD for Second-Level Domain, 3rdLD for 3rd-Level Domain, 4thLD for 4t-Level Domain.

Table 2.2 Examples of each TLDs.

ccTLD	gTLD	grTLD	sTLD	eTLD
.jp	.com	.biz	.aero	.co.jp
.uk	.info	.name	.asia	.ac.jp
.cn	.net	.pro	.cat	.akashi.hyogo.jp

の例を Table 2.2 に示す.

■Public Suffix (eTLD) 各ブラウザによるクッキーの適応可能範囲決定のために“1 つのサイト”の単位というものが求められた．それを受けて Public Suffix というものが提案された[*1]．これは effective TLD (eTLD) とも呼ばれ，TLD や `co.jp` などの実質的に TLD のように機能するドメインを指す．TLD 内のサブドメインの構造は TLD ごとに異なるため，機械的に eTLD を決定することはできない．そのため Public Suffix List (PSL)[13] として一覧表が管理されている.

2.2 ICN におけるコンテンツ名

ICN におけるコンテンツ名を本研究では

`icn:/<reTLD>/<Root>/<rHostName>/<Path>`

のように定義し，ICN-URL と呼ぶ．`reTLD` (reverse-eTLD) と `rHostName` (reverse-HostName) はそれぞれ `eTLD` と `HostName` を”.”を区切りとして逆順に配置したものである．すなわち，`eTLD` が `ab.cd.ef` なら `reTLD` は `ef.cd.ab` となる．Fig. 2.2 に具体例を示す.

[*1] 例えば，`example.co.jp` や `a.example.co.jp` の Public Suffix は `co.jp` で，`example.co.jp` がサイトの単位となる．もし，`example.co.jp` がドメインが `co.jp` や `jp` のクッキーを発行できてしまうと，異なるサイトであるはずの `test.co.jp` にも干渉できてしまう．これを防ぐためにクッキーの処理では PSL を参照するようになっている.

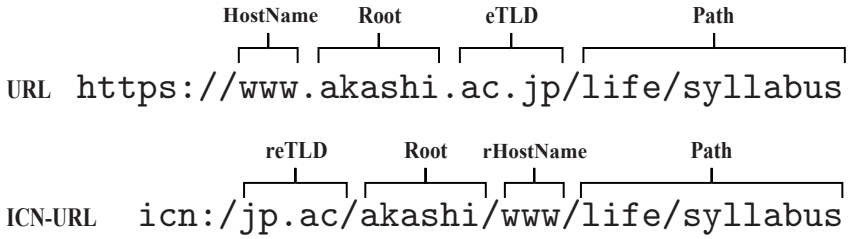

Fig. 2.2 Example of ICN-URL.

2.3 性能の評価手順

Fig. 1.1 のテーブルを Fig. 2.3 に再掲する．FIB, PIT, CS の 3 つのテーブルは Name をキーとしており，その分布を評価することでアルゴリズムの性能を評価する．評価のために Fig. 2.4 に示す手順を行った．入手可能な全 URL のリスト (All list) から 10MB 程度のサイズになるようにランダムに抽出したリスト (Sampled list) を作る．そのリストで URL の規格に合わないものを除外したのち，ICN-URL に変換し eTLD の頻度の多い順に並べる．順に ICN-URL からハッシュを計算してハッシュテーブルを作成し，ハッシュテーブルの各 eTLD の先頭アドレスを保持したポインタテーブルを作成する．作成したハッシュテーブルの中で同一のハッシュ値となっているものを衝突という．衝突がないほうが性能が良いと考えられるため，衝突数の分布を調べることにより性能を評価する．

2.3.1 解析データ

解析に用いるデータとして，The Content Name Collection[*2]で公開されている情報指向ネットワークのための膨大な URL のデータセットを用いる．そのデータセットの内の一つである `urls.txt` を使う．`urls.txt` は Fig. 2.5 のように改行で区別されている URL のリストとなっている．`urls.txt` の概要を Table 2.3 に示す．

Table 2.3 Dataset ”urls.txt” overview.

Dataset	Number of URLs	unique	File size
`urls.txt`	2,144,314,011	no	121 GB (130,782,049,461 Bytes)

`urls.txt` の前処理として以下の工程を行う．このデータには重複が含まれているので重複を削除する．これは 60GB ほどのサイズで約 8.8 億件の URL を含み，全 URL リストと呼ぶ．各 CR での実際的な URL は 10MB ほどであるという仮定の下，ランダムな 10MB を抽出し，解析データとした．ここには約 14 万件の URL が含まれる．この解析データ中の各 URL を ICN-URL に変換する．

[*2] `http://www.icn-names.net/` にて”The Content Name Collection”というバーゼル大学による情報指向ネットワークのためのデータセットが 2019 年 10 月まで公開されていたが，ドメインの有効期限切れのため現在は全く関係のない中国の会社によりドメインが取得されている．

FIB	
Name	**Next**
/aueb.gr/	CR C
/aueb.gr/cs	CR B

PIT	
Name	**Requested**
/aueb.gr/ai/new.htm	Subscriber

CS	
Name	**Data**
-	-

Fig. 2.3 Tables of FIB, PIT, CS.

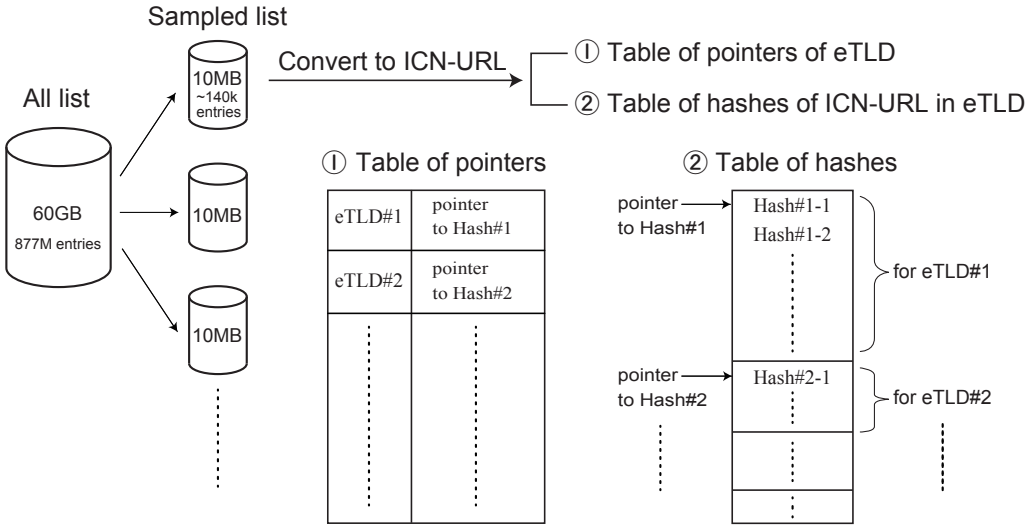

Fig. 2.4 Analysis procedure.

```
http://www.google.com
http://images.google.com/imgres
http://www.19lou.com
http://www.sfd.com
http://www.baidu.com
http://www.sina.com.cn
http://www.netvibes.com/
http://www.google.com/search
http://images.google.com/
http://wrestlingabrasil.blogspot.com/
...
...
...
```

Fig. 2.5 Samples 10 in the urls.txt.

2.3.2 ハッシュアルゴリズム

ICN-URL からハッシュ値を求めるアルゴリズムを述べる．まず，Fig. 2.6 に示すように ICN-URL を”/”で分割する．それぞれをセクション (section) と呼ぶ．また，自然言語には連続した特徴が存在し，3-gram を用いることで曖昧性を解消できるという先行研究 [14] に基づき 3 文字の抽出を行う．その際セクションの文字数が 3 文字未満の場合はセクションの文字数に応じて次のように 3 文字にする (パディング).

セクションが 1 文字のとき　ICN-URL の長さとスラッシュの数を掛けたものを uint16 型で 2 バイト付加する

セクションが 2 文字のとき　ICN-URL のスラッシュの数を byte 型として 1 バイト付加する

次に，各セクションから前 3 文字を抜き出して配列 heads とする．同様に後 3 文字を抜き出して配列 tails とするが，パティングが含まれているセクションはパディングと元の文字との順序を入れ替える．結果的に heads と tails のどちらかに含まれるセクションの数が 3 未満であればその URL は処理の対象外とする．

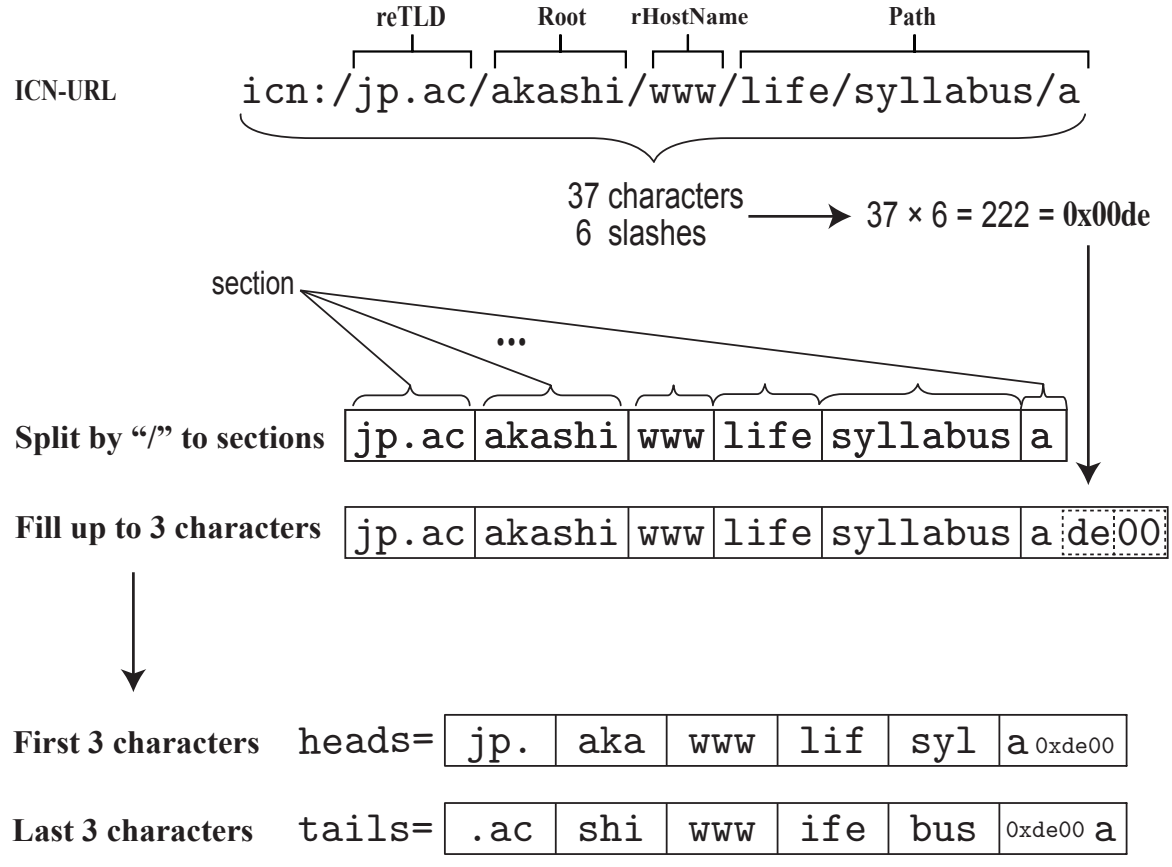

Fig. 2.6　Preparation to make hash.

先程作成した配列 heads と tails を元に 3 種類のハッシュアルゴリズムを考案した.

ハッシュアルゴリズム A

heads の先頭要素 3 バイト，tails の末尾要素 3 バイト，末尾から 3 つ目の要素 3 バイトの各 3 バイト，計 9 バイトをそれぞれのバイトごとに XOR を計算して 3 バイトにする．heads の先頭 1 文字と先程の 3 バイトを連結したものを 4 バイトのハッシュ値とする．具体例を Fig. 2.7 に示す．

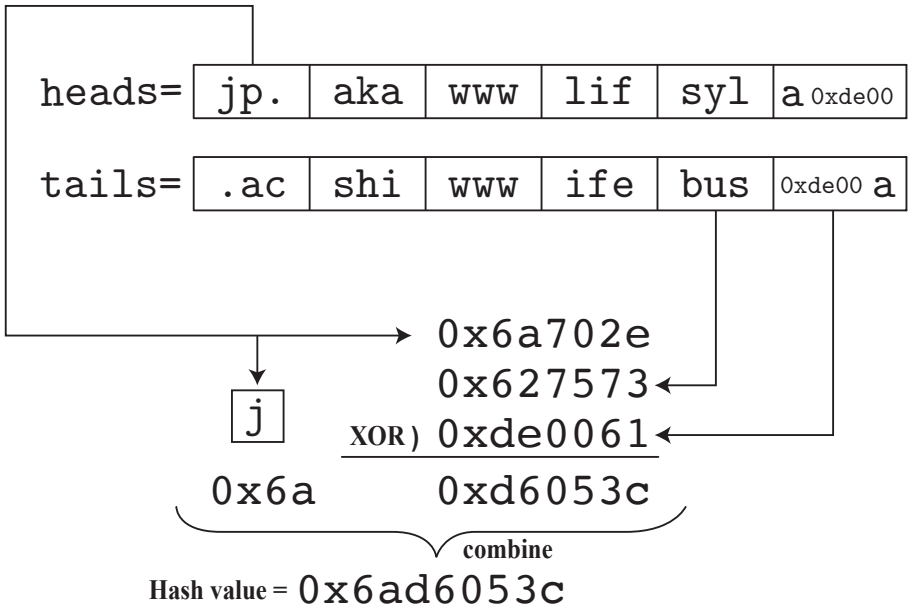

Fig. 2.7　Hash algorithm A.

ハッシュアルゴリズム B

heads の先頭要素 3 バイト，tails の末尾要素 3 バイトのそれぞれのバイトごとに XOR を計算し，それを連結して 2 バイトにする．heads の先頭 1 文字と先程の 2 バイトを連結したものを 3 バイトのハッシュ値とする．具体例を Fig. 2.8 に示す．

ハッシュアルゴリズム C

heads の先頭要素 3 バイト，tails の末尾要素 3 バイト，末尾から 2 つ目の要素 3 バイトのそれぞれのバイトごとに XOR を計算し，それを連結して 3 バイトにする．heads の先頭 1 文字と先程の 3 バイトを連結したものを 4 バイトのハッシュ値とする．具体例を Fig. 2.9 に示す．

2.3.3　ハッシュテーブル

約 10MB の ICN-URL に変換された解析データを reTLD（1 番目のセクション）の頻度の多い順に並べる．これは，eTLD の頻度の多い順と同じである．各行の ICN-URL に対して上記の 3 種類のハッシュアルゴリズムの内のどれかにより順にハッシュ値を求め，ハッシュテーブルを作成する．このハッシュテーブルには頻度順の同じ eTLD に対応するハッシュ値が連続して並んでいる．すなわち，eTLD ごとにグループ分けされたハッシュテーブルが得られる (Fig. 2.4 の②)．新たな eTLD のグループの出現する位置のアドレスをポイ

ンタと呼び，それと eTLD との対応関係をポインタテーブルと呼ぶ (Fig. 2.4 の①).

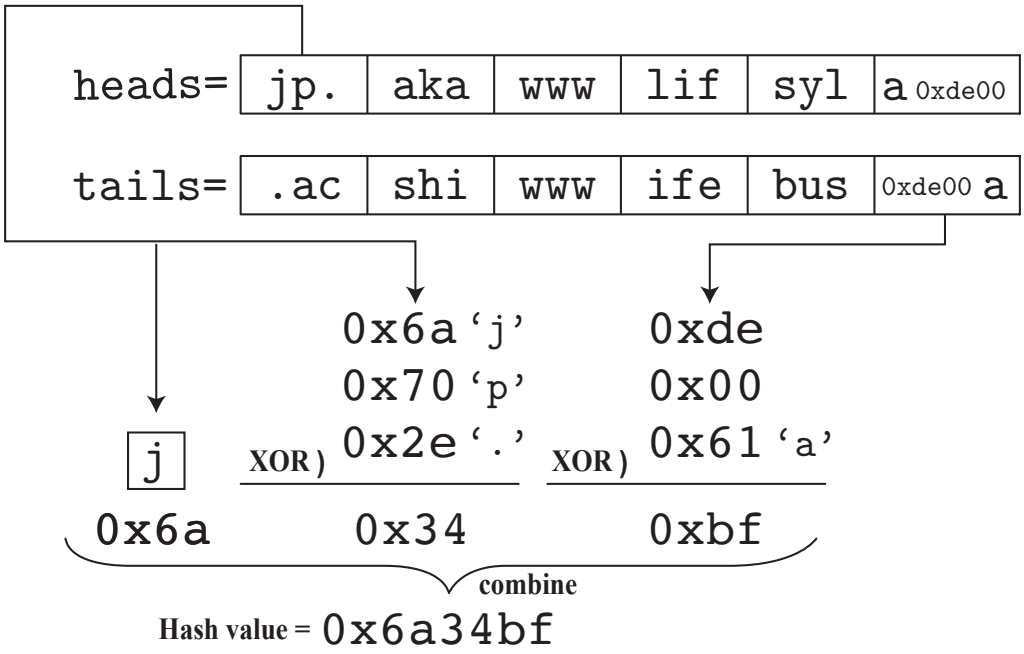

Fig. 2.8 Hash algorithm B.

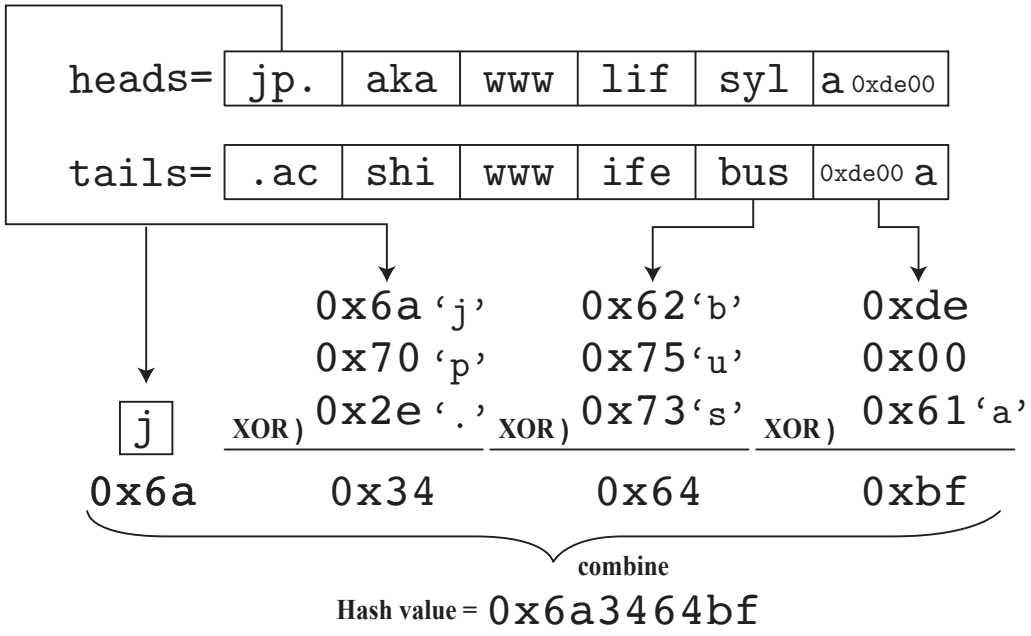

Fig. 2.9 Hash algorithm C.

2.4 解析プログラム

本研究を行うために Golang で作成したプログラムの概要を述べる．各処理の具体的なプログラムコードは付録に掲載する．

2.4.1 前処理

重複削除

入力されたデータの重複を削除する．今回は 121GB のデータなので linux の uniq コマンドでは時間がかかりすぎるため独自に実装した．文字数による分割統治法を用いて，分割したデータごとに並行処理を行うことで重複削除の高速化を図った．重複削除することで 60GB 程度にデータサイズが削減された．

サンプル抽出

重複のないデータから 10MB 分をランダムに抽出する機能を実装した．60GB のデータをメモリ上に展開することはできないので，行番号と先頭からのバイト数のテーブルを作成した．これをメモリ上に展開し，乱数により行番号を選択し，対応するファイル位置を求めファイルポインタを移動させてその行を選択しファイルに出力する．これを出力したファイルサイズが 10MB になるまで繰り返す．

2.4.2 コンテンツ名への変換

バッファに入るだけ読み込んでから並列プロセスに渡して以下の処理をする．1 行読んで URL の構文解析を行い URL の規格に沿っていないものを除外する．Public Suffix List とドメイン名を照合して eTLD と Root と HostName を抽出する．これを定義した ICN-URL の形式に合うように組み立てる．

2.4.3 ハッシュテーブルとポインタテーブル生成

バッファに入るだけ読み込んでから並列プロセスに渡して以下の処理をする．ICN-URL から 3 種類のいずれかのハッシュアルゴリズムによりハッシュ値を計算する．ハッシュ値を順に出力する際に，eTLD と出力バイト数の関係をポインタテーブルとして出力する．

2.4.4 衝突数の解析

作成したハッシュテーブルの中で同一のハッシュ値となっているものを数え上げ，その件数を出力する．また同一のハッシュ値の数がどのような分布となっているかを確認するために相補累積分布を出力する．

第 3 章

性能評価

ハッシュアルゴリズムの違いや，新たに提案したポインタを用いるデータ構造による衝突数の分布を調べることで性能を評価する．

3.1　解析データの妥当性

全 URL リストからランダムに 10MB 分を抽出した解析データが元の全 URL リストの分布と同じ特徴を持っているかを確かめる．それぞれの eTLD の頻度分布を Fig. 3.1 に示す．グラフ中の ALL URLs は全 URL リスト，Sampled URLs は解析データをそれぞれ示している．グラフから解析データは全 URL リストとほぼ同じ分布を示しており，全 URL リストの特徴を代表していると考えることができる．

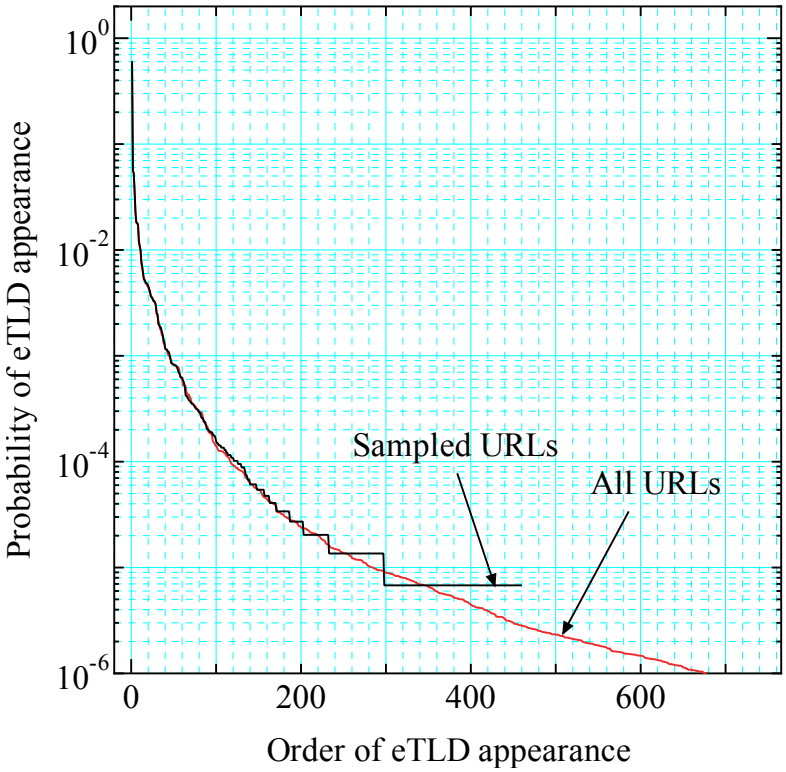

Fig. 3.1　Probability of eTLD appearance.

3.2 eTLD の分布

解析データの eTLD の分布を調べる．その分布を Fig. 3.2 に示す．Fig. 3.2 からわかるようにごく少数の eTLD に大半の URL が含まれている．そこで Fig. 3.3 と Table 3.1 に上位 10 件の詳細の分布を示す．解析データの URL の件数は 147,315 であり，eTLD の上位 10 件に含まれる URL の件数は 125,161 である．すなわち上位 10 件で 85% を占めている．この内 1 位の com で 60% を占めており，極端に偏っていることがわかる．これに対する対策が必要と予想される．

Table 3.1 Top 10 of eTLD, URL count and probability in the sampled URLs (147,315).

ID	eTLD	URL count	Probability[%]
0	com	89399	60.6856
1	net	8285	5.6240
2	me	7720	5.2405
3	org	5201	3.5305
4	de	3301	2.2408
5	blogspot.com	2667	1.8104
6	jp	2644	1.7948
7	co.uk	2576	1.7486
8	info	1783	1.2103
9	com.br	1585	1.0759

3.3 ハッシュアルゴリズムの比較

3 種類のハッシュアルゴリズム A–C を比較するために，解析データ用いてハッシュテーブルを作成した．そのハッシュテーブルの中で同じハッシュ値を持つ ICN-URL の数を数え上げて衝突数とする．そして同じ衝突数を持つ ICN-URL の合計数を求める．この合計数の累積の ICN-URL 総数に対する比率を算出する．これを累積分布と呼ぶ．累積分布と 1 との差の絶対値を相補累積分布 (Complementary Cumulative Distribution Function: CCDF) と呼ぶ．これを Fig. 3.4 に示す．このグラフではある横軸において値が小さい方がより衝突の割合が少ないことを示す．横軸は同じハッシュ値の数なので 1 は衝突していないことを意味し，2 以上は衝突していることを表す．したがって，ハッシュアルゴリズム A が最良であるとわかる．4 回程度の衝突ではハードウェアで処理可能だが，それ以上になるとソフトウェアで処理しなければならなくなるという仮定を行っている．そのため，ハッシュアルゴリズムの評価の基準として 4 回衝突のところで縦の補助線を引いてある．ハッシュアルゴリズム A では，4 回衝突のとき，CCDF$= 1.3 \times 10^{-2}$ であった．この確率が十分小さいかどうかは，今後実装するハードウェアの処理速度に依存する．

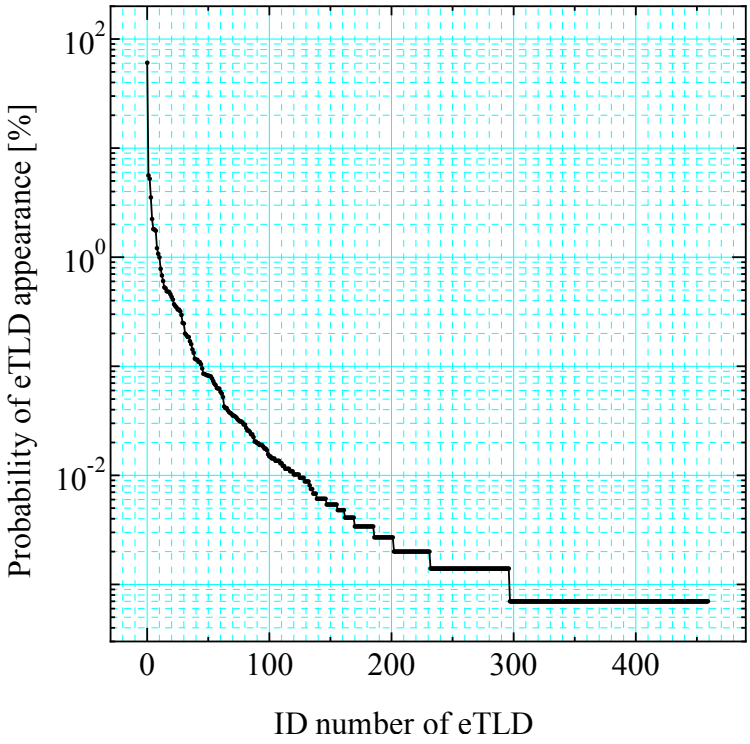

Fig. 3.2 Probability of eTLD appearance in the sampled URLs.

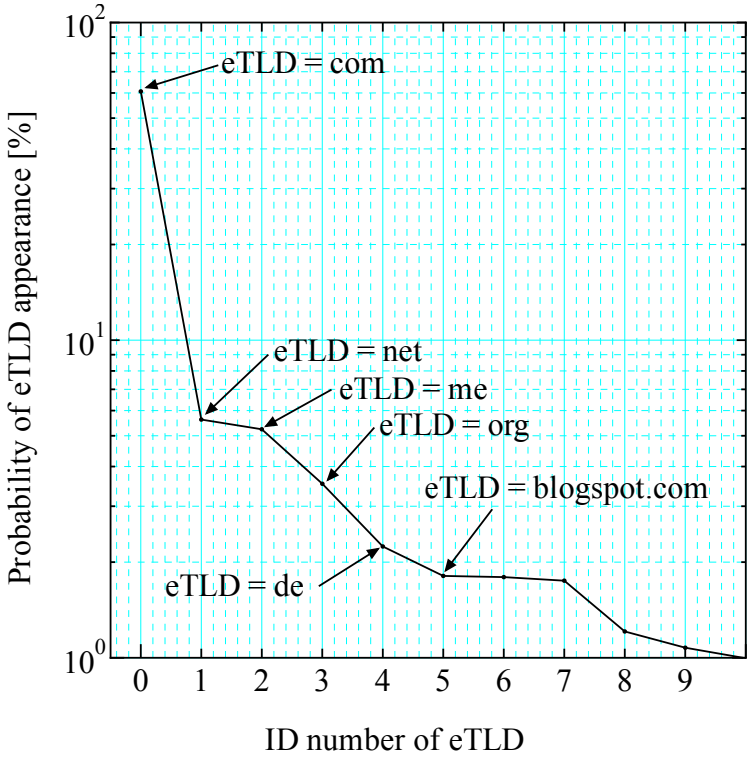

Fig. 3.3 Probability of eTLD appearance in the sampled URLs (top 10 eTLD).

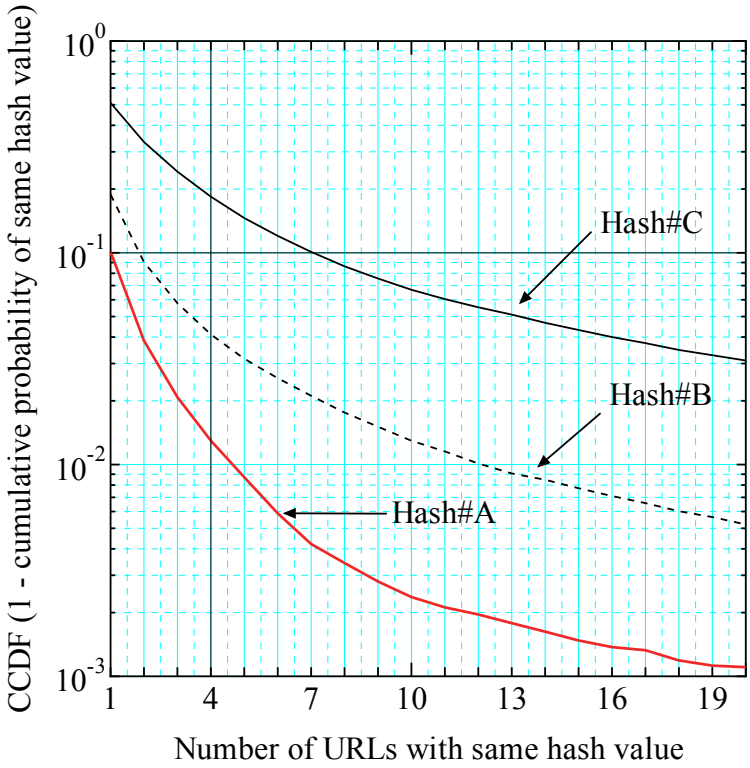

Fig. 3.4 Complementary cumulative distribution function of same hash value with each hash algorithm A – C.

3.4 eTLD ごとのハッシュ衝突率

Fig. 2.4 のポインタテーブルを用いることで，異なる eTLD は区別されるので 1 つの eTLD に対するハッシュ値の衝突だけが問題となる．したがって各 eTLD についてだけハッシュ値の衝突率を求めればよい．Fig. 3.5 に各 eTLD ごとのハッシュアルゴリズム A のときのハッシュ衝突率を示す．

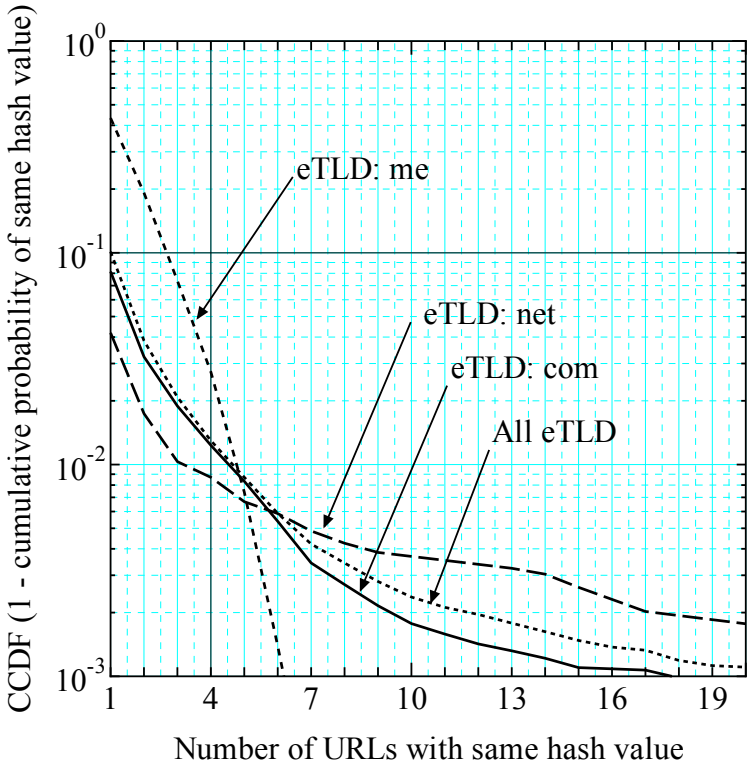

Fig. 3.5　Complementary cumulative distribution function of same hash value for each eTLD when hash algorith A.

Fig. 3.5 より eTLD が `me` のときだけ衝突しない確率が他と比べ極端に低くなっていることがわかる．これは次のように考えられる．eTLD が me では，`formspring.me` というサイトがデータの大半を占めており，`icn:/me/formspring/nandaghizzo/q/908453668` のような ICN-URL の形式である．このときハッシュアルゴリズム A を用いると毎回異なるパラメータとなるのは最後の 3 文字つまり 3 桁の数字である．これは投稿などの ID だと考えられるので連番である可能性が高い．このことから 1000 件に 1 回は衝突することになり，今回 `me` の件数は 7720 件であったため，最大でも 7 回の衝突のみとなり結果的に数件の衝突に集中した．他の eTLD については ALL eTLD と近い傾向を示した．

3.5　eTLD に Root を加えた場合

eTLD のみだと同じ eTLD をもつ URL の数が大きいので衝突が発生しやすくなると考えた．そこでこれを細分化するために次の 3 種の規則により Root を eTLD に追加する．

1. com のときの Root の出現回数が 10 の Root を eTLD に加える (Fig. 3.6, 3.7 の赤線).
2. com のときの Root の出現回数の多い順の上位 256 を eTLD に加える (Fig. 3.6 の青線).
3. 2. に加え net の上位 256 も加える (Fig. 3.6 の緑線).

Fig. 3.6 から加える Root 部の数が多いほどポインタの数が増加することがわかる．上記 1 の規則について詳しく見るために Fig. 3.7 と Table 3.2 に出現確率の多い上位 10 件のポインタを示す．グラフから `twitter.com` と `google.com` が上位に入っていることがわかる．そして，`com` が占める割合が 60% から 25%

に減ったことにより，衝突確率が下がることが期待できる．

Table 3.2　Top 10 of eTLD+Root, URL count and probability in the sampled URLs (147,315).

ID	eTLD+Root	URL count	Probability[%]
0	com	36423	24.7246
1	net	8285	5.624
2	me	7720	5.2405
3	org	5201	3.5305
4	twitter.com	4357	2.9576
5	de	3301	2.2408
6	google.com	3092	2.0989
7	blogspot.com	2667	1.8104
8	jp	2644	1.7948
9	co.uk	2576	1.7486

Fig 3.8 に eTDL に Root を加えたときと，加えてないときのハッシュ値の衝突率の比較を示す．グラフから `com`, `net` の衝突率が eTLD に Root を加えることで低減されていることがわかる．例えば `com` について，衝突しない確率が 92% から eTLD に Root を加えることで 96% に向上した．また，4 回以上衝突する確率 1.3% から 0.3% に減少した．すなわち 4 倍以上の向上がみられた．

3.6　まとめ

ハッシュアルゴリズム A のような 3 回の XOR を行うだけの軽量な計算負荷で求まる 4 バイトのハッシュ値でも，ポインタテーブルを併用することにより 5 回以上の衝突確率を 0.3% に抑えることができた．

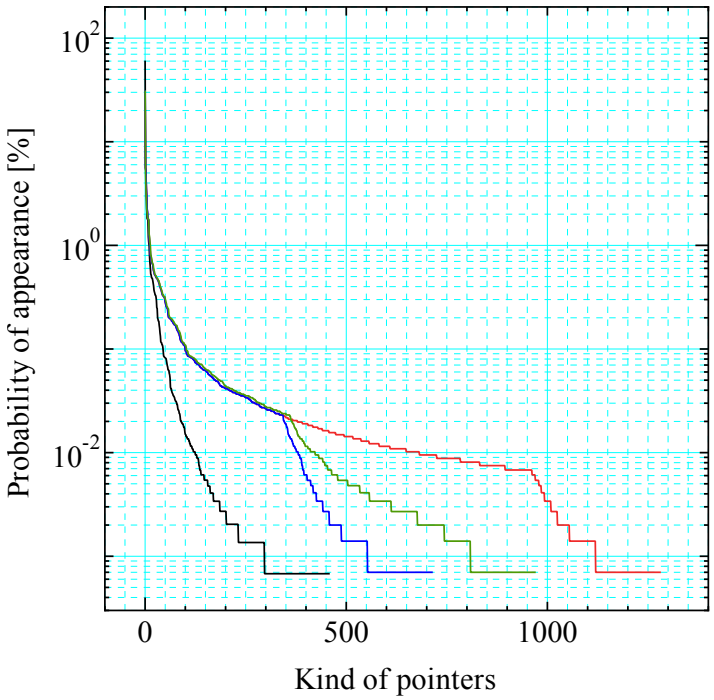

Fig. 3.6 Probability of appearance of eTLD or eTLD + Root. Black line shows only eTLD. Red line shows eTLD + Root when number of appearance of Root is more than 10 in eTLD of com. Blue and green lines show eTLD + Root when top 256 most frequently appearing Root in eTLD of com, com and net, respectively.

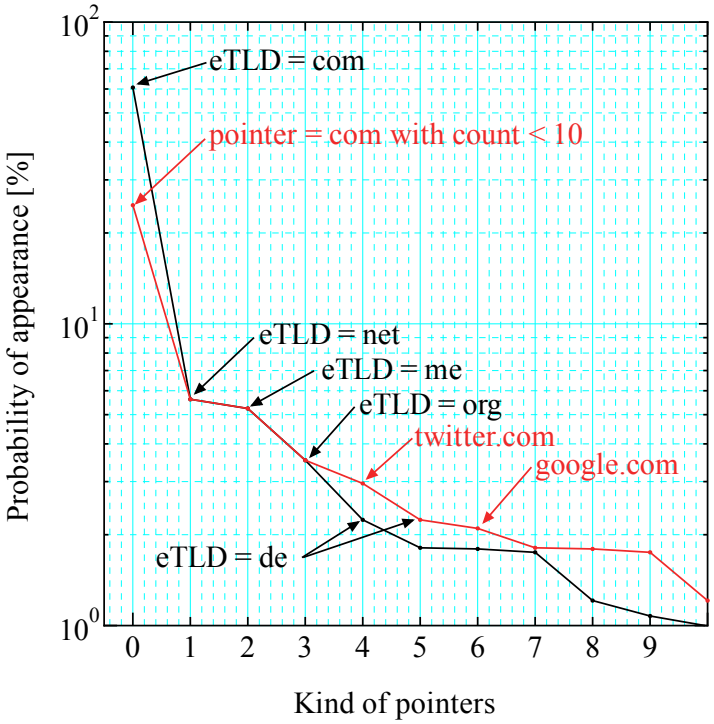

Fig. 3.7 Detail of Fig. 3.6 of red line and black line (top 10 of pointer).

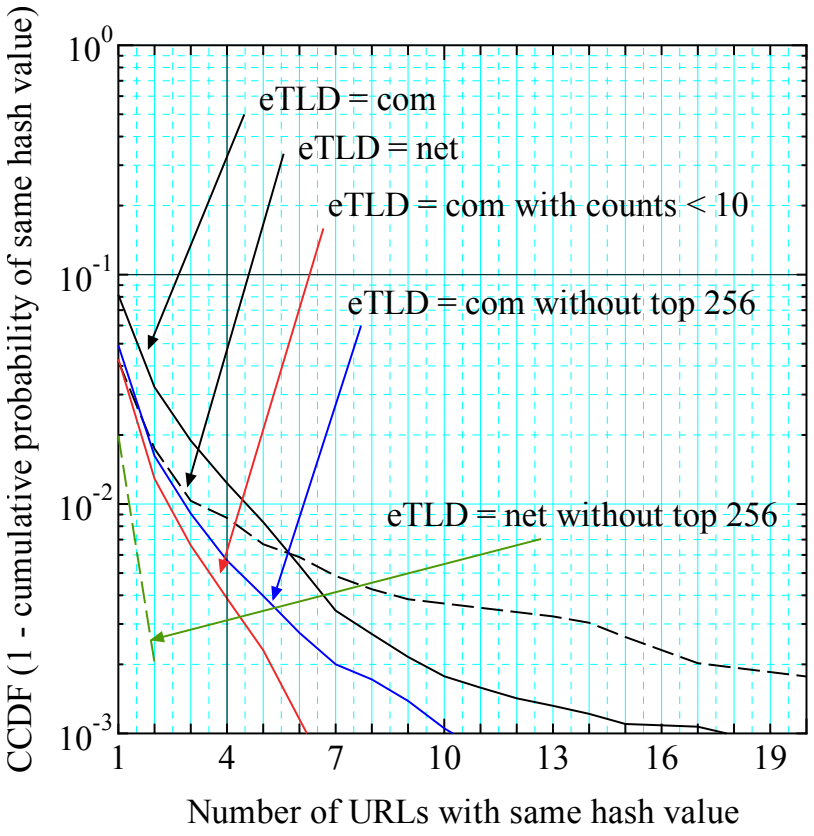

Fig. 3.8 Complementary cumulative distribution function of same hash value for each eTLD when hash algorith A.

第 4 章

結論

本論文の ICN のルーティングのための URL のクラシフィケーションついてシミュレーションを用いて解析検討した結果をまとめ，結論とする．

1. 3 通りのハッシュアルゴリズムを提案した．ICN-URL を”/”で分割し，3 バイトの文字列を切り出し，そのうちの 2–3 の文字列どうしの XOR 演算から 3 バイトから 4 バイトのハッシュ値を生成した．このハッシュ値と ICN-URL の対応をハッシュテーブルとして作成し，衝突確率を求めた．
2. 衝突数を軽減するためにポインタを併用したハッシュ作成アルゴリズムを提案した．URL の eTLD の分布を調べることにより，com が 60% 占めていて偏りが大きいことがわかった．そのため衝突率が大きくなっていた．
3. eTLD に Root を加えることで偏りが大きかった com の占める割合を 24% に下げることで 5 回以上衝突する確率を 0.3% に抑えることができた．
4. 上記ポインタとハッシュの併用を新たな検索手法として共同研究先に提案を行った．

今後の課題

1. ハッシュアルゴリズムが簡易的なものだったので，eTLD を Root に加えた際にハッシュ値への寄与はなかった．そこで，eTLD を Root に加えた際にハッシュ値が変化するようなハッシュアルゴリズムの改良をする．
2. 本研究では 4 回程度の衝突ではハードウェアで処理可能だが，それ以上になるとソフトウェアで処理しなければならなくなるという仮定のもと研究を行った．しかし，どの程度の衝突数であればハードウェアで処理可能であるかといったことは来年以降の課題とする．

謝辞

本研究を進めるにあたり，ご指導いただいた指導教員の井上一成教授に感謝いたします．
御助言を頂いた共同研究先の国立研究開発法人情報通信研究機構の大岡睦氏に感謝の意を表します．

参考文献

[1] David D. Clark et al. Barry M. Leiner, Vinton G. Cerf. Brief history of the internet. Internet Society, 1997.

[2] Cisco. Cisco visual networking index: Forecast and trends, 2017 - 2022. Cisco, 2019.

[3] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT ' 09, pp. 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[4] 朝枝仁, 松園和久. 情報指向ネットワーク技術におけるプロトタイプ実装と評価手法. コンピュータ ソフトウェア, Vol. 33, No. 3, pp. 3_3–3_15, 2016.

[5] NSF Named Data Networking project. [Online]. Available: `http://www.named-data.net/`.

[6] Content Centric Networking project. [Online]. Available: `http://www.ccnx.org/`.

[7] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos. A survey of information-centric networking research. *IEEE Communications Surveys Tutorials*, Vol. 16, No. 2, pp. 1024–1049, Second 2014.

[8] Lorenzo Saino, Ioannis Psaras, and George Pavlou. Hash-routing schemes for information centric networking. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, pp. 27–32, 2013.

[9] 顕士小松, 卓也朝香. コンテンツ指向ネットワークにおけるブルームフィルタを用いた経路情報管理方式 (ネットワークシステム). 電子情報通信学会技術研究報告 = IEICE technical report : 信学技報, Vol. 113, No. 35, pp. 13–18, may 2013.

[10] M. McCahill Xerox Corporation. Uniform resource locators (url), dec 1994. [Online]. Available: `https://tools.ietf.org/html/rfc1738`.

[11] Xylogics Xylogics. Internet users' glossary, aug 1996. [Online]. Available: `https://tools.ietf.org/html/rfc1983`.

[12] Internet Assigned Numbers Authority. Root zone database, feb 2020. [Online]. Available: `https://www.iana.org/domains/root/db`.

[13] Mozilla Foundation. Public suffix list, feb 2020. [Online]. Available: `https://publicsuffix.org/`.

[14] 小堀望. Iotlan と icn アーキテクチャの構築. 平成 30 年度卒業論文, 2019. Content Centric Networking project. [Online]. Available: http://www.ccnx.org/.

付録 A

作成したプログラム

作成したプログラムの一部を付録として記載する．
詳細は `https://gitlab.com/inue-lab/icn-hash` にて公開しているのでそちらを参照．

A.1　重複を削除する

文字列の長さごとに分割し，分割したそれぞれにおいて重複を削除する．

util/unique.go

```
// SplitByLength 文字列の長さごとに分割する
func SplitByLength(inputFilePath string, freeMem uint64) {
  processFilePath = inputFilePath
  scanner, inputFile := InputFile(processFilePath)
  defer inputFile.Close()

  os.RemoveAll(splitTmpDIr)
  os.MkdirAll(splitTmpDIr, os.ModePerm)

  log.Printf("Input file : %s", processFilePath)
  log.Printf("Output dir : %s", splitTmpDIr)

  bufferSize := freeMem / maxURLLength * 10
  log.Printf("Buffer Size : %v", humanize.Bytes(bufferSize))

  sbuffer := make(map[int]*strings.Builder, maxURLLength)
  start := time.Now()

  for scanner.Scan() {
    url := scanner.Text()
    url = RemoveMeta(url)
    if len(url) == 0 {
```

```
23          continue
24        }
25
26        if sb, ok := sbuffer[len(url)]; ok {
27          sb.WriteString(url)
28          sb.WriteString("\n")
29          processedURLCount++
30
31          if sb.Cap() > int(bufferSize) { // バッファーサイズを超えたらファイルに書き出す
32            writer, outFile := OutputFileAppend(splitTmpDIr + strconv.Itoa(len(url)) +
              ↪  ".txt")
33            writer.WriteString(sb.String())
34            writer.Flush()
35            outFile.Close()
36            sb.Reset()
37          }
38        } else {
39          sbuffer[len(url)] = &strings.Builder{}
40
41          sb := sbuffer[len(url)]
42          sb.Grow(IntermediateMapSize)
43          sb.WriteString(url)
44          sb.WriteString("\n")
45
46          processedURLCount++
47        }
48        inputURLCount++
49      }
50
51      inputFile.Close()
52
53      for length, sb := range sbuffer {
54        if sb.Len() != 0 {
55          writer, outFile := OutputFileAppend(splitTmpDIr + strconv.Itoa(length) +
            ↪  ".txt")
56          writer.WriteString(sb.String())
57          writer.Flush()
58          outFile.Close()
59          sb.Reset()
```

```
60      }
61    }
62
63    end := time.Now()
64
65    log.Printf("Total execution time : %v",
      ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
66    log.Printf("Number of input URLs : %v", humanize.Comma(int64(inputURLCount)))
67    log.Printf("Number of processed URLs : %v",
      ↪  humanize.Comma(int64(processedURLCount)))
68
69    inputURLCount, processedURLCount = 0, 0
70  }
71
72  // uniqueGoroutine Goroutine ごとに実行されて重複を取り除く
73  func uniqueGoroutine(urlCount, originalURLCount *int, inputFilePath string, writer
    ↪  *bufio.Writer, wg *sync.WaitGroup, mux *sync.Mutex) {
74    defer wg.Done()
75
76    scanner, inFile := InputFile(inputFilePath)
77    defer inFile.Close()
78
79    // 1st level uniq per goroutines
80    m := make(map[string]bool, IntermediateMapSize)
81    count := 0
82    for scanner.Scan() {
83      url := scanner.Text()
84      if len(url) == 0 {
85        continue
86      }
87      count++
88      if !m[url] {
89        m[url] = true
90      }
91    }
92
93    // 2nd level uniq with getting mutex
94    mux.Lock()
95    defer mux.Unlock()
```

```
96
97    (*originalURLCount) += count
98    for url := range m {
99      writer.WriteString(url)
100     writer.WriteString("\n")
101     (*urlCount)++
102   }
103   writer.Flush()
104 }
105
106 // Unique 重複をなくす
107 func Unique(maxWorkers int) {
108   files, err := ioutil.ReadDir(splitTmpDIr)
109   if err != nil {
110     Elog.Print(splitTmpDIr + " is not directory.")
111     os.Exit(1)
112   }
113
114   outputFilePath := GetFileName(processFilePath) + ".unique"
115   writer, outFile := OutputFile(outputFilePath)
116   defer outFile.Close()
117
118   log.Printf("Input dir : %s", splitTmpDIr)
119   log.Printf("Output file : %s", outputFilePath)
120
121   // sync primitives
122   wg := new(sync.WaitGroup)
123   mux := new(sync.Mutex)
124
125   start := time.Now()
126
127   fileList := []int{}
128   for _, file := range files {
129     name, _ := strconv.Atoi(GetFileName(file.Name()))
130     fileList = append(fileList, name)
131   }
132   sort.SliceStable(fileList, func(i, j int) bool {
133     return fileList[i] < fileList[j]
134   })
```

```
135
136   for _, file := range fileList {
137     inputFilePath := filepath.Join(splitTmpDIr, strconv.Itoa(file)+".txt")
138     wg.Add(1)
139     go uniqueGoroutine(&processedURLCount, &inputURLCount, inputFilePath, writer,
      ↪  wg, mux)
140
141     // wait if number of goroutines reach max workers for resource limitation
142     if runtime.NumGoroutine() >= maxWorkers {
143       wg.Wait()
144     }
145   }
146
147   wg.Wait()
148   writer.Flush()
149
150   os.RemoveAll(splitTmpDIr)
151
152   end := time.Now()
153   log.Printf("Total execution time : %v",
    ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
154   log.Printf("Number of input URLs : %v", humanize.Comma(int64(inputURLCount)))
155   log.Printf("Number of unique-URLs : %v",
    ↪  humanize.Comma(int64(processedURLCount)))
156 }
```

A.2　ランダムにサンプルを抽出

メルセンヌ・ツイスタにより生成した疑似乱数に対応する行を抽出する.

util/random.go

```
1 // CreatePointer 行数に対応するポインターを作る
2 func CreatePointer(inputFilePath string) {
3   processFilePath = inputFilePath
4   scanner, inputFile := InputFile(processFilePath)
5   defer inputFile.Close()
6
7   pointerFileName = GetFileName(processFilePath) + "-pointer.bin"
8   pointerFileWriter, pointerFile := OutputFile(pointerFileName)
9
```

```
10    log.Printf("Input file : %s", processFilePath)
11    log.Printf("Output pointer table file : %s", pointerFileName)
12
13    start := time.Now()
14
15    var readByte uint64
16    for scanner.Scan() {
17      buf := make([]byte, binary.MaxVarintLen64)
18      binary.LittleEndian.PutUint64(buf, readByte)
19      pointerFileWriter.Write(buf)
20      readByte += uint64(len(scanner.Bytes()))
21      readByte++ // for "\n"
22      inputURLCount++
23    }
24
25    buf := make([]byte, binary.MaxVarintLen64)
26    binary.LittleEndian.PutUint64(buf, readByte)
27    pointerFileWriter.Write(buf)
28
29    pointerFileWriter.Flush()
30    pointerFile.Close()
31    end := time.Now()
32
33    log.Printf("Total execution time : %v",
      ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
34    log.Printf("Number of input URLs : %v", humanize.Comma(int64(inputURLCount)))
35  }
36
37  // Random ファイルをランダムに分割する
38  func Random() {
39    scanner, inputFile := InputFile(pointerFileName)
40    log.Printf("Input pointer file : %s", pointerFileName)
41    log.Printf("Number of input URLs : %v", humanize.Comma(int64(inputURLCount)))
42
43    start := time.Now()
44    split := func(data []byte, atEOF bool) (advance int, token []byte, err error) {
45      if atEOF && len(data) == 0 {
46        return 0, nil, nil
47      }
```

```
48      return binary.MaxVarintLen64, data[0:binary.MaxVarintLen64], nil
49    }
50    scanner.Split(split)
51
52    pointerList := make([]uint64, inputURLCount+1)
53    i := 0
54    var pointer uint64
55    for scanner.Scan() {
56      bytes := scanner.Bytes()
57      pointer = binary.LittleEndian.Uint64(bytes)
58      pointerList[i] = pointer
59      i++
60    }
61
62    inputFile.Close()
63    randomFileCount, processedURLCount := 0, 0
64
65    outputDir := GetFileName(processFilePath) + "-random/"
66    os.RemoveAll(outputDir)
67    os.MkdirAll(outputDir, os.ModePerm)
68
69    log.Printf("Output dir : %s", outputDir)
70    log.Printf("Split Size : %v", humanize.Bytes(uint64(splitSize)))
71    writer, outFile := OutputFile(outputDir + strconv.Itoa(randomFileCount) + ".txt")
72
73    fp, err := os.Open(processFilePath)
74    if err != nil {
75      panic(err)
76    }
77    usedNum := make(map[int]bool, inputURLCount)
78
79    num := 0
80    var writeByteSize int64
81
82    seed, _ := crand.Int(crand.Reader, big.NewInt(math.MaxInt64))
83    rng := rand.New(mt19937.New()) // Mersenne twister
84    rng.Seed(seed.Int64())
85
86    for ; ; num = rng.Intn(inputURLCount) {
```

```
87      if usedNum[num] {
88        continue
89      }
90      usedNum[num] = true
91
92      startPointer := pointerList[num]
93      length := pointerList[num+1] - startPointer // include "\n"
94
95      fp.Seek(int64(startPointer), 0) // seek to pointer
96      buf := make([]byte, length)
97      _, err = io.ReadFull(fp, buf)
98      if err != nil {
99        panic(err)
100     }
101
102     processedURLCount++
103     writer.Write(buf)
104     writeByteSize += int64(length)
105
106     if writeByteSize >= int64(splitSize) {
107       writer.Flush()
108       outFile.Close()
109       randomFileCount++
110       if randomFileCount > 30 {
111         break
112       }
113       writer, outFile = OutputFile(outputDir + strconv.Itoa(randomFileCount) +
          ↪  ".txt")
114       writeByteSize = 0
115     }
116   }
117
118   writer.Flush()
119   outFile.Close()
120   fp.Close()
121   os.RemoveAll(pointerFileName)
122   end := time.Now()
123
```

```
124   log.Printf("Total execution time : %v",
      ↪ humanize.SI(float64(end.Sub(start).Seconds()), "s"))
125   log.Printf("Number of processed URLs : %v",
      ↪ humanize.Comma(int64(processedURLCount)))
126 }
```

A.3 ICN-URL に変換する

icn/convert.go

```
1  // convertGoroutine 各 Goroutine ごとに実行されて URL を icn に変換する
2  func convertGoroutine(mbuf *util.MovableBuffer, wg *sync.WaitGroup, mux
   ↪  *sync.Mutex) {
3    defer wg.Done()
4
5    var icn strings.Builder     // icn の URL のビルダー
6    var etldIcn strings.Builder // ポインタと eTLD とを分割して出力する際のビルダー
7
8    icn.Grow(util.IntermediateMapSize)
9    if separate { // ポインタと eTLD とを分割して出力するか
10     etldIcn.Grow(util.IntermediateMapSize)
11   }
12
13   m := make(map[int]int, util.IntermediateMapSize)
14   count := 0
15
16   for _, URL := range mbuf.Buf {
17     icnString, segment, isSpecified := generateICN(URL)
18     if segment < 0 {
19       continue
20     }
21
22     if separate && !isSpecified { // ポインタと eTLD とを分割して出力する際の eTLD 側
23       etldIcn.WriteString(icnString)
24     } else { // それ以外はこっち
25       icn.WriteString(icnString)
26     }
27
28     if v, ok := m[segment]; ok {
29       m[segment] = v + 1
```

```
    } else {
      m[segment] = 1
    }

    count++
  }

  mbuf.Move() // バッファを開放する

  // mutex をロックする
  mux.Lock()
  defer mux.Unlock()

  icnFileWriter.WriteString(icn.String())
  icnFileWriter.Flush()

  if separate { // ポインタと eTLD とを分割して出力する場合
    etldIcnFileWriter.WriteString(etldIcn.String())
    etldIcnFileWriter.Flush()
  }

  icnCount += count
  for k, v := range m {
    if rv, ok := numberOfSegmentMap[k]; ok {
      numberOfSegmentMap[k] = rv + v
    } else {
      numberOfSegmentMap[k] = v
    }
  }

}

// ConvertICN URL から ICN を生成する
func ConvertICN(uniqueURLFilePath string, specificTLDs []string, com, sep bool,
↪  bufSize, maxW int) string {
  separate, bufferSize, maxWorkers = sep, bufSize, maxW

  start := time.Now()

```

```
68    if com { // com の 10 件以上出現したものをポインタとする
69      icnFileName = util.GetFileName(uniqueURLFilePath) + "-icn-com10.txt"
70    } else if specificTLDs[0] != "" { // 上位 256 をポインタとする eTLD が指定されている場合
71      icnFileName = util.GetFileName(uniqueURLFilePath) + "-icn-" +
      ↪  strings.Join(specificTLDs, "-") + "256.txt"
72    } else { // eTLD のみをポインタとする場合
73      icnFileName = util.GetFileName(uniqueURLFilePath) + "-icn.txt"
74    }
75
76    graphFile := util.GetFileName(icnFileName) + "-segment.tsv"
77
78    if specificTLDs[0] != "" {
79      specifiedRoots = countSpecifiedRoots(uniqueURLFilePath, specificTLDs, com) // 指
      ↪  定した eTLD を条件に沿った Root を調べる
80    }
81
82    log.Printf("Input file : %s", uniqueURLFilePath)
83    scanner, inputFile := util.InputFile(uniqueURLFilePath)
84    defer inputFile.Close()
85
86    if separate { // ポインタと eTLD とを分割して出力する場合
87      etldIcnFileName = util.GetFileName(icnFileName) + "-etld.txt"
88      icnFileName = util.GetFileName(icnFileName) + "-pointer.txt"
89      etldIcnFileWriter, etldIcnFile = util.OutputFile(etldIcnFileName)
90      log.Printf("Output file : %s", etldIcnFileName)
91    }
92    defer etldIcnFile.Close()
93
94    log.Printf("Output file : %s", icnFileName)
95    icnFileWriter, icnFile = util.OutputFile(icnFileName)
96    defer icnFile.Close()
97
98    log.Printf("Output segment file : %s", graphFile)
99
100   // tld parser
101   var err error
102   extract, err = tldextract.New(util.Cache, false)
103   if err != nil {
104     util.Elog.Print(err)
```

```
105     os.Exit(1)
106   }
107
108   urlCount = util.Parallel(scanner, bufferSize, maxWorkers, convertGoroutine)
109
110   icnFileWriter.Flush()
111   if separate { // ポインタと eTLD とを分割して出力する場合
112     etldIcnFileWriter.Flush()
113   }
114
115   processedURLCount := 0
116   segment := []segmentStruct{}
117   for s, c := range numberOfSegmentMap {
118     segment = append(segment, segmentStruct{
119       segment: s,
120       count:   c,
121     })
122     processedURLCount += c
123   }
124   numberOfSegmentMap = map[int]int{}
125
126   sort.Slice(segment, func(i, j int) bool {
127     return segment[i].segment < segment[j].segment
128   })
129
130   writer, outFile := util.OutputFile(graphFile)
131   for _, v := range segment {
132     writer.WriteString(strconv.Itoa(v.segment))
133     writer.WriteString("\t")
134     writer.WriteString(strconv.Itoa(v.count))
135     writer.WriteString("\n")
136   }
137   writer.Flush()
138   outFile.Close()
139
140   os.Remove(util.Cache)
141
142   end := time.Now()
143
```

```
144   log.Printf("Total execution time : %v",
      ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
145   log.Printf("Number of input URLs : %v", humanize.Comma(int64(urlCount)))
146   log.Printf("Number of processed URLs : %v",
      ↪  humanize.Comma(int64(processedURLCount)))
147   log.Printf("Number of ICNs : %v", humanize.Comma(int64(icnCount)))
148
149   return icnFileName
150 }
```

icn/pointer.go

```
1  // countSpecifiedRoots 指定された eTLD での該当する Root を返す
2  func countSpecifiedRoots(inputFilePath string, specificTLDs []string, com bool)
   ↪  map[string][]string {
3    specifiedRoots := make(map[string][]string, len(specificTLDs))
4    for _, tld := range specificTLDs {
5      scanner, file := util.InputFile(domain.Count(inputFilePath, tld, "", false,
       ↪  false, bufferSize, maxWorkers))
6      count := 0
7      for scanner.Scan() {
8        count++
9        txt := strings.Split(scanner.Text(), "\t")
10       if com {
11         count, _ := strconv.Atoi(txt[0])
12         if count < 10 { // 10 件以上あるもののみ対象
13           break
14         }
15       } else {
16         if count > 256 { // 上位 256 件をポインタとする
17           break
18         }
19       }
20
21       specifiedRoots[tld] = append(specifiedRoots[tld], txt[2])
22     }
23     file.Close()
24   }
25
26   return specifiedRoots
27 }
```

```
28
29 // generateICN ICNを生成する
30 func generateICN(URL string) (string, int, bool) {
31   u, err := url.Parse(URL) // url parse
32   if err != nil {
33     return "", -1, false
34   }
35
36   domain := strings.ToLower(u.Hostname())
37   d := extract.Extract(domain) // domain parse
38   if d.Flag != 1 {
39     return "", -1, false
40   }
41
42   var icn strings.Builder
43   icn.WriteString("icn:/")
44
45   rtld := util.ReverseTLD(d.Tld)
46   icn.WriteString(rtld)
47
48   var isSpecified bool
49   if roots, ok := specifiedRoots[d.Tld]; ok && util.Contains(roots, d.Root) { // 指
     ↪ 定されている eTLD かつその Root が条件を満たしているか
50     isSpecified = true
51     icn.WriteString(".")
52   } else {
53     isSpecified = false
54     icn.WriteString("/")
55   }
56
57   icn.WriteString(d.Root) // u.Path is include "/"
58
59   var segment int
60   if d.Sub != "" {
61     var rsub string
62     icn.WriteString("/")
63     rsub, segment = util.ReverseSubDomain(d.Sub) // segment 数は eTLD と Root を除いたと
       ↪ きにスラッシュで区切られている個数
64     icn.WriteString(rsub)                          // u.Path is include "/"
```

```
65    }
66
67    if u.Path != "" {
68      path := url.QueryEscape(u.Path)                  // マルチバイト文字をエスケープする
69      path = strings.ReplaceAll(path, "%2F", "/")      // スラッシュは戻す
70      segment += len(strings.Split(u.Path, "/")) - 1 // u.Path is include "/"
71      icn.WriteString(path)
72    }
73
74    icn.WriteString("\n")
75
76    return icn.String(), segment, isSpecified
77  }
```

A.4　ハッシュの計算

コード中の hash5–7 がハッシュアルゴリズム A–C に対応する.

hash/algo.go

```
1   // hash5 Hash#5 のアルゴリズム
2   func hash5(heads, tails [][]byte) string {
3     byte1 := heads[0]
4     byte2 := tails[len(tails)-1]
5     byte3 := tails[len(tails)-2]
6     bytes1 := (byte1[0] ^ byte2[0]) ^ byte3[0]
7     bytes2 := (byte1[1] ^ byte2[1]) ^ byte3[1]
8     bytes3 := (byte1[2] ^ byte2[2]) ^ byte3[2]
9
10    hash := fmt.Sprintf("%02x%02x%02x%02x", byte1[0], bytes1, bytes2, bytes3)
11    return hash
12  }
13
14  // hash6 Hash#6 のアルゴリズム
15  func hash6(heads, tails [][]byte) string {
16    byte1 := heads[0]
17    byte2 := tails[len(tails)-1]
18    bytes1 := (byte1[0] ^ byte1[1]) ^ byte1[2]
19    bytes2 := (byte2[0] ^ byte2[1]) ^ byte2[2]
20
21    hash := fmt.Sprintf("%02x%02x%02x", byte1[0], bytes1, bytes2)
```

```
  return hash
}

// hash7 Hash#7のアルゴリズム
func hash7(heads, tails [][]byte) string {
  byte1 := heads[0]
  byte2 := tails[len(tails)-1]
  byte3 := tails[len(tails)-2]
  bytes1 := (byte1[0] ^ byte1[1]) ^ byte1[2]
  bytes2 := (byte2[0] ^ byte2[1]) ^ byte2[2]
  bytes3 := (byte3[0] ^ byte3[1]) ^ byte3[2]

  hash := fmt.Sprintf("%02x%02x%02x%02x", byte1[0], bytes1, bytes2, bytes3)
  return hash
}
```

hash/calc.go

```
// CalcHash ハッシュを計算する
func CalcHash(icn string, hashT string) (string, bool) {
  sicn := strings.SplitN(icn, "/", 3)
  if len(sicn) < 3 {
    return "-", false
  }
  heads, tails, ok := splitToArrayFillByte(sicn[0] + "/" + sicn[2])
  if !ok {
    return "-", false
  }

  var hash string
  switch hashT {
  case "hash5":
    hash = hash5(heads, tails)
  case "hash6":
    hash = hash6(heads, tails)
  case "hash7":
    hash = hash7(heads, tails)
  }

  return hash, true
}
```

```
24
25  // calcHashGoroutine Goroutine ごとに CalcHash を実行する
26  func calcHashGoroutine(mbuf *util.MovableBuffer, wg *sync.WaitGroup, mux
    ↪  *sync.Mutex) {
27    defer wg.Done()
28
29    var sbuffer strings.Builder
30    sbuffer.Grow(util.IntermediateMapSize)
31    count := 0
32
33    for _, icn := range mbuf.Buf {
34      hash, ok := CalcHash(icn, hashType)
35      if !ok {
36        continue
37      }
38      sbuffer.WriteString(hash)
39      sbuffer.WriteString("\t")
40      sbuffer.WriteString(icn)
41      sbuffer.WriteString("\n")
42
43      count++
44    }
45
46    mbuf.Move() // バッファを開放する
47
48    // mutex をロックする
49    mux.Lock()
50    defer mux.Unlock()
51
52    writer.WriteString(sbuffer.String())
53    hashCount += count
54    writer.Flush()
55  }
56
57  // Hash ハッシュを計算して出力する
58  func Hash(inputFilePath, hashT string, bufferSize, maxWorkers int) {
59    hashType = hashT
60
61    start := time.Now()
```

```
62
63    scanner, inputFile := util.InputFile(inputFilePath)
64    defer inputFile.Close()
65
66    outputFileName := util.GetFileName(inputFilePath) + "-" + hashType + ".txt"
67    writer, outputFile = util.OutputFile(outputFileName)
68    defer outputFile.Close()
69
70    log.Printf("Input file : %s", inputFilePath)
71    log.Printf("Output file : %s", outputFileName)
72    log.Printf("Buffer Size : %v", humanize.Bytes(uint64(bufferSize)))
73
74    hashCount = 0
75    icnCount = util.Parallel(scanner, bufferSize, maxWorkers, calcHashGoroutine)
76
77    writer.Flush()
78
79    end := time.Now()
80    log.Printf("Total execution time : %v",
      ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
81    log.Printf("Number of ICNs : %v", humanize.Comma(int64(icnCount)))
82    log.Printf("Number of Hashes : %v", humanize.Comma(int64(hashCount)))
83  }
```

hash/split.go

```
1  // splitToArrayFillByte ICN を各セクションの前後 3Byte の配列に分解する
2  func splitToArrayFillByte(icn string) ([][]byte, [][]byte, bool) {
3    idx, dx := 5, 0    // 現在の位置 (最初のセクション (icn:/) は無視する), 現在の位置から "/"
     ↪  までの距離
4    var heads [][]byte // 各セクションの最初の 3Byte
5    var tails [][]byte // 各セクションの最後の 3Byte
6
7    for {
8      if idx >= len(icn) { // 終端に達したとき
9        break
10     }
11
12     dx = strings.Index(icn[idx:], "/") // 現在の位置から "/" を検索
13     if dx == -1 {                      // 最後のセクション
14       dx = len(icn[idx:]) // 末尾までの距離
```

```
15      }
16      var head []byte
17      var tail []byte
18
19      switch dx {
20      case 0:
21        idx += dx + 1 // 次の"/"に進む
22        continue
23      case 1: // セクションが 1Byte の場合は 2Byte で ICN の長さとスラッシュの数を掛けたものを連
        ↪  結する
24        c := len(icn) * strings.Count(icn, "/")
25        buf := make([]byte, 2)
26        binary.LittleEndian.PutUint16(buf, uint16(c))
27        head = []byte(icn[idx : idx+1])
28        head = append(head, buf...)
29        tail = append(tail, buf...)
30        tail = append(tail, []byte(icn[idx:idx+1])...)
31      case 2: // セクションが 2Byte の場合は 1Byte でスラッシュの数を連結する
32        c := strings.Count(icn, "/")
33        head = []byte(icn[idx : idx+2])
34        head = append(head, byte(c))
35        tail = append(tail, byte(c))
36        tail = append(tail, []byte(icn[idx:idx+2])...)
37      default:
38        head = []byte(icn[idx : idx+3])
39        tail = []byte(icn[idx+dx-3 : idx+dx])
40      }
41
42      heads = append(heads, head)
43      tails = append(tails, tail)
44
45      idx += dx + 1 // 次の"/"に進む
46    }
47
48    if len(heads) < 3 { // カウントしたセクションが３列以下のとき
49      return [][]byte{}, [][]byte{}, false
50    }
51
52    return heads, tails, true
```

```
}
```

A.5　ハッシュの衝突数を数え上げる

ハッシュの衝突数を数え上げて CCDF を計算する.

hash/count.go

```
// countHash メインプロセス
func countHash(mbuf *util.MovableBuffer, wg *sync.WaitGroup, mux *sync.Mutex) {
  defer wg.Done()

  m := make(map[string]int, util.IntermediateMapSize)
  for _, k := range mbuf.Buf {
    key := strings.SplitN(k, "\t", 2)[0]
    if key == "-" {
      continue
    }

    if v, ok := m[key]; ok {
      m[key] = v + 1
    } else {
      m[key] = 1
    }
  }

  mbuf.Move() // バッファを開放する

  // mutex をロックする
  mux.Lock()
  defer mux.Unlock()

  for k, v := range m {
    if rv, ok := result[k]; ok {
      result[k] = rv + v
    } else {
      result[k] = v
    }
  }
}

```

```go
// Count ハッシュの衝突数を数える
func Count(inputFilePath string, bufferSize, maxWorkers int) {
  start := time.Now()

  scanner, inputFile := util.InputFile(inputFilePath)
  defer inputFile.Close()

  mode := false
  outputFileName := ""
  if strings.Contains(inputFilePath, "-count") {
    outputFileName = util.GetFileName(inputFilePath) + "-ccdf.tsv"
    mode = true
  } else {
    outputFileName = util.GetFileName(inputFilePath) + "-count.tsv"

  }

  writer, outputFile := util.OutputFile(outputFileName)
  defer outputFile.Close()

  log.Printf("Input file : %s", inputFilePath)
  log.Printf("Output file : %s", outputFileName)
  log.Printf("Buffer Size : %v", humanize.Bytes(uint64(bufferSize)))

  // resultMap
  result = make(map[string]int, bufferSize)

  icnCount = util.Parallel(scanner, bufferSize, maxWorkers, countHash)

  if mode {
    hash := []hashCountCountStruct{}

    for h, c := range result {
      h, _ := strconv.Atoi(h)
      hash = append(hash, hashCountCountStruct{
        count:     c,
        hashCount: h,
      })
    }
```

```
73
74      result = nil
75
76      sort.Slice(hash, func(i, j int) bool {
77        return hash[i].hashCount < hash[j].hashCount
78      })
79
80      hashCount = len(hash)
81
82      cdf := 0
83      for _, h := range hash {
84        cdf += h.count
85        writer.WriteString(strconv.Itoa(h.hashCount))
86        writer.WriteString("\t")
87        writer.WriteString(strconv.Itoa(h.count))
88        writer.WriteString("\t")
89        writer.WriteString(strconv.Itoa(cdf))
90        writer.WriteString("\t")
91        writer.WriteString(strconv.FormatFloat(float64((icnCount-cdf))/float64(icnCount),
          ↪  'E', 6, 64))
92        writer.WriteString("\n")
93      }
94    } else {
95      hash := []hashCountStruct{}
96
97      for h, c := range result {
98        hash = append(hash, hashCountStruct{
99          count: c,
100         hash:  h,
101       })
102     }
103
104     result = nil
105
106     sort.Slice(hash, func(i, j int) bool {
107       return hash[i].count > hash[j].count
108     })
109
110     hashCount = len(hash)
```

```
111
112     for _, h := range hash {
113       writer.WriteString(strconv.Itoa(h.count))
114       writer.WriteString("\t")
115       writer.WriteString(h.hash)
116       writer.WriteString("\n")
117     }
118   }
119
120   writer.Flush()
121   end := time.Now()
122
123   log.Printf("Total execution time : %v",
      ↪ humanize.SI(float64(end.Sub(start).Seconds()), "s"))
124   log.Printf("Number of ICNs : %v", humanize.Comma(int64(icnCount)))
125   log.Printf("Number of Hashes : %v", humanize.Comma(int64(hashCount)))
126 }
```

A.6 ハッシュテーブルとポインタテーブルの作成

table/split.go

```
1  // SplitByTLD eTLD ごとに分割する
2  func SplitByTLD(icnFileName, hashType string, bufferSize int) string {
3    start := time.Now()
4
5    scanner, inputFile := util.InputFile(icnFileName)
6    defer inputFile.Close()
7
8    outputDir := util.GetFileName(icnFileName) + "-" + hashType + "-tld/"
9    os.RemoveAll(outputDir)
10   os.MkdirAll(outputDir, os.ModePerm)
11
12   log.Printf("Input file : %s", icnFileName)
13   log.Printf("Output dir : %s", outputDir)
14   log.Printf("Buffer Size : %v", humanize.Bytes(uint64(bufferSize)))
15
16   originalICNCount, icnCount := 0, 0
17
18   sbuffer := make(map[string]*strings.Builder, 10000)
```

```
19
20    for scanner.Scan() {
21      icn := scanner.Text()
22
23      hashStr, _ := hash.CalcHash(icn, hashType)
24
25      tld := strings.SplitN(icn, "/", 3)[1] // icn:/tld/
26
27      if sb, ok := sbuffer[tld]; ok {
28        sb.WriteString(hashStr)
29        sb.WriteString("\t")
30        sb.WriteString(icn)
31        sb.WriteString("\n")
32        icnCount++
33
34        if sb.Cap() > bufferSize { // バッファーサイズを超えたらファイルに書き出す
35          writer, outFile := util.OutputFileAppend(outputDir + tld + ".tsv")
36          writer.WriteString(sb.String())
37          writer.Flush()
38          outFile.Close()
39          sb.Reset()
40        }
41
42      } else {
43        sbuffer[tld] = &strings.Builder{}
44
45        sb := sbuffer[tld]
46        sb.Grow(util.IntermediateMapSize)
47        sb.WriteString(hashStr)
48        sb.WriteString("\t")
49        sb.WriteString(icn)
50        sb.WriteString("\n")
51
52        icnCount++
53      }
54      originalICNCount++
55    }
56
57    inputFile.Close()
```

```
58
59    for tld, sb := range sbuffer {
60      if sb.Len() != 0 {
61        writer, outFile := util.OutputFileAppend(outputDir + tld + ".tsv")
62        writer.WriteString(sb.String())
63        writer.Flush()
64        outFile.Close()
65        sb.Reset()
66      }
67    }
68
69    end := time.Now()
70
71    log.Printf("Total execution time : %v",
      ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
72    log.Printf("Number of original URLs : %v",
      ↪  humanize.Comma(int64(originalICNCount)))
73    log.Printf("Number of processed URLs : %v", humanize.Comma(int64(icnCount)))
74
75    return outputDir
76  }
```

table/table.go

```
1   // GenerateTable テーブルを作成する
2   func GenerateTable(inputFilePath, domainFile, tldDir, hashType string) {
3     start := time.Now()
4
5     tldScanner, inputFile := util.InputFile(domainFile)
6     defer inputFile.Close()
7
8     hashTableFileName := util.GetFileName(inputFilePath) + "-" + hashType +
      ↪  "-hash_table.tsv"
9     hashTableWriter, hashTableFile := util.OutputFile(hashTableFileName)
10    defer hashTableFile.Close()
11
12    pointerTableFileName := util.GetFileName(inputFilePath) + "-" + hashType +
      ↪  "-pointer_table.tsv"
13    pointerTableWriter, pointerTableFile := util.OutputFile(pointerTableFileName)
14    defer pointerTableFile.Close()
15
```

```
16    log.Printf("Output hash table file : %s", hashTableFileName)
17    log.Printf("Output pointer table file : %s", pointerTableFileName)
18
19    icnCount, writeByteSize, hashCount := 0, 0, 0
20
21    pointerTableWriter.WriteString(hashTableFileName) // メタ情報出力
22    pointerTableWriter.WriteString("\t")
23    pointerTableWriter.WriteString(hashType)
24    pointerTableWriter.WriteString("\n")
25
26    for tldScanner.Scan() {
27      txt := tldScanner.Text()
28      tld := strings.SplitN(txt, "\t", 3)[2]
29      rtld := util.ReverseTLD(tld)
30
31      pointerTableWriter.WriteString(tld)
32      pointerTableWriter.WriteString("\t")
33      pointerTableWriter.WriteString(strconv.Itoa(writeByteSize))
34      pointerTableWriter.WriteString("\n")
35
36      icnScanner, icnInputFile := util.InputFile(filepath.Join(tldDir, rtld+".tsv"))
37      for icnScanner.Scan() {
38        icn := icnScanner.Text()
39        icnCount++
40        if strings.Split(icn, "\t")[0] != "-" {
41          b1, _ := hashTableWriter.WriteString(icn)
42          b2, _ := hashTableWriter.WriteString("\n")
43          hashCount++
44          writeByteSize += b1
45          writeByteSize += b2
46        }
47      }
48
49      hashTableWriter.Flush()
50      icnInputFile.Close()
51    }
52
53    pointerTableWriter.Flush()
54    end := time.Now()
```

```
55
56    log.Printf("Total execution time : %v",
      ↪  humanize.SI(float64(end.Sub(start).Seconds()), "s"))
57    log.Printf("Number of input ICNs : %v", humanize.Comma(int64(icnCount)))
58    log.Printf("Number of generated Tables : %v", humanize.Comma(int64(hashCount)))
59
60  }
```