UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Ago Allikmaa

Using URL Templates to Find Hidden Entity Pages

Bachelor's Thesis (6 ECTS)

Supervisor: Peep Küngas

Tartu 2016

Using URL Templates to Find Hidden Entity Pages

Abstract:

This thesis describes a method for finding hidden entity pages based on a list of URLs visited by a web crawler. The described method creates a list of URL templates based on the input URLs and predicts new possible entity page addresses based on those. In the initial template generation phase, templates are generated by detecting numeric path elements and treating other elements as static texts. To generate only one template for one set of entities, they are deduplicated in the unused path element detection phase by merging together templates that represent the same set of entities via an alternative path, which is achieved by comparing the contents of the pages they represent. The templates are split to have only one changing variable which is the numeric entity identifier, known as its index. New URLs are generated from the gaps of values in the entity index for a template.

Keywords:

Hidden web, entity page, URL normalization, URL deduplication

CERCS: P175 - Informatics, systems theory

Peidetud veebilehtede leidmine aadressimallide abil

Lühikokkuvõte:

See lõputöö kirjeldab meetodit peidetud objektilehtede leidmiseks kasutades selleks veebiroomaja poolt leitud aadresside nimekirja. Aadresside põhjal leitakse URLide mallid, mille põhjal genereeritakse uusi aadresse. Selleks, et üks objektileht erinevate mallide hulka ei satuks, tuleb malle agregeerida nii, et ühte objektiseeriat esitavad mallid oleksid üheks malliks kokku pandud. Mallide agregeerimiseks tuvastatakse aadressidest osasid, mis ei mõjuta olulisel määral lehe sisu. Iga mall peab viitama ühele objektilehtede seeriale, milles objekti identifikaatoriks on arv. Selleks peab lõplik mall koosnema ühest numbrilisest muutujast ning ülejäänud osadele peab andma kindla väärtuse. Mallidest genereeritakse uusi aadresse kasutades numbriliste väärtuste hulgast puuduvaid arve, mis jäävad suurima ja väikseima teadaoleva objekti identifikaatori vahele.

Võtmesõnad:

Sügav veeb, URL-ide normaliseerimine, URL-ide deduplikatsioon

CERCS: P175 - Informaatika, süsteemiteooria

Table of contents

1. Introduction

Web crawling is a common method of extracting information from web sites either for archiving, indexing for future searches or looking for specific information on the web. The most common way we interact with the results of web crawling is by search engines, which are used by many people on a daily basis [1].

Most of the current web crawlers only traverse the web by following links on already visited sites. This poses a limitation that only web pages that have been linked on an already visited site are reachable.

Web pages not accessible to a web crawler are often collectively referred to as the deep web [2]. While not reachable for a web crawler, these pages might be accessible to a user visiting the web site. This is in cases where the user can reach the page by either interacting with the web site in some way that either loads a web page dynamically or by entering data into a form and finding results that were not directly linked on the site individually. In this case, the user is expected to use the forms and the operator of the site did not see a reason to create a link-based index of the site content.

There has been much work done on crawling web sites based on forms [3][4][5][6]. A common approach is to detect the topic of the web site and generate a list of relevant words that are likely to yield results on submitting the form. It may also be done the other way around by crawling for pages on a specific topic and filtering out forms that are relevant to it [3]. Some form-based methods for finding hidden pages are designed specifically for finding entity pages [4].

An entity page is a web page that focuses on a single entity. That entity may be an article, a movie, a person or any other type of entity. Many web pages might appear to display a lot of different content on every page even when most of the individual pages are actually entity pages.

Sometimes there are other reasons why a specific web page might not be reachable. This includes the specific page being outdated, designed for manually sharing or soft-deleted, while still being accessible. In these cases, a form-based approach may not work either due to these pages not being included in results or when there is no form targeting this type of page at all.

The objective of this thesis is to develop a method to find hidden web pages by generating entity page URLs directly instead of a more common approach of generating queries based on forms.

This work describes an approach that uses URL templates generated from a list of URLs produced by a web crawler instead of making use of web forms. URLs may contain any number of path elements or parameters that may make it difficult to predict new working URLs based on existing ones. For that reason, this work focuses only on entity pages.

A URL of an entity page often contains some unique identifier of the entity. Two common methods for generating an identifier for a new entity is an incrementing index and random text. Random text is useless for the purpose of predicting new entity identifiers, but a set of identifiers generated using an incrementing index actually gives a lot of information, such as the relative age of an entity compared to another one or detecting entities missing from the set.

The approach described in this work focuses on trying to predict new entity page URLs by producing URL templates which contain only one variable, which is a numeric identifier of an entity.

For this approach, different templates should either refer to a different set of entities which are not generated from the same incrementing index, or display different type of information about the same entity. Otherwise this method would produce new URLs for entity pages that were actually present in the pages of another URL template representing the same set of entities.

While search engine optimization techniques have been helpful to ordinary web crawlers and also made URLs more human-readable, it has also caused many web sites to use an URL system where there are additional varying path elements in the URL.

To get URL templates of this quality, naively detected templates must be merged together if they are detected to actually represent the same type of information for the same set of entities. This is done in the method described in this thesis in the prefix and suffix detection phase and unused path element detection stage. As URLs themselves might not contain enough information to detect this, the content of the actual pages behind the URLs is compared.

2. Related Work

There are two common ways to find previously unindexed deep web pages. One of them is analysing URLs to produce templates for generating new URLs for either search queries or directly for entity pages. The other is detecting search forms on pages to generate queries that will produce lists of pages. This work uses the former approach, but there are certain similarities between the two approaches which makes them both worth looking at in this context.

Barbosa and Freire [3] developed a crawler algorithm that is designed to find web forms matching a specific topic. Its main focus is on finding as many forms as possible, and accurately filtering out forms that are either not search forms or not relevant for the user-defined topic. The components designed for efficiently finding many forms are called the link classifier and frontier manager. Link classifier determines how likely a link is to lead to a search form in a few steps. Frontier manager collects tracked links and maintains a priority between them, where the one most likely to produce quality forms is retrieved first from it. To improve the quality of detected forms, they have components called the page classifier and searchable form classifier. The page classifier detects if the given page is relevant to the user-defined topic, so that links and forms would not be harvested from unrelated pages. The searchable form classifier excludes all forms that it detects are not search forms, but for example login, email, posting or subscription forms. This also allows them to leave sites that do not produce productive results, to maximize efficiency over time. Another distinguishing factor of their crawler is that the link classifier is adaptive and it updates the keywords it uses to detect useful pages based on the rewards from already visited pages.

The main difference from the current work is that they operate with search forms and do not proceed to use them to find data using the forms they discover. In the current work, the role of the form is filled by URL templates. In case the URL parameters were the result of form fields, an URL template can be considered to be a result of reverse engineering a form based on links. There is a major difference in the focus of the work as their algorithm is designed to limit the results to a specific topic. This is not in the scope of the current work, but it could certainly be extended to fit additional purposes if the hidden pages found via URL templates were limited to a specific topic.

Both URL template generation and search form query generation can be used for accessing lists of entity pages - the former for simpler search forms and the latter for search forms with several different fields. Since many sites have very simple search forms to make it more convenient for the customers, the former approach is sufficient in many cases.

He et al. [4] describe an algorithm that utilizes URL templates to perform a deep crawl of entity pages by generating new URLs that lead to additional entity pages not found by directly following links. The described method of URL template generation is designed to detect templates for search queries that could be used to do semantic searches by using query keywords that are likely to produce many results on a specific site. The generated URLs are then crawled and resulting pages with empty results are filtered out. Links to entity pages are extracted from the crawled pages of search results. The combined links from different search results are deduplicated, both to produce a more representative list of

entity pages and to minimize the duplicate work when crawling the produced entity pages on the second iteration of crawling.

The steps used in this work are similar to the ones used in the current work, but they are used for different purposes. URL templates and generated URLs are used for search queries or category listings that produce lists of entities rather than for specific entity pages. In the current work, URL template generation is for the entity page URLs and therefore empty page filtering and URL deduplication is a part of it to generate better templates. While the methods used are similar and the objective is the same, the strategy for finding entity pages is completely different from the current work. The main difference is that while in this work the entity pages are found by generating pages that list them, in the current work the URL templates are generated for the entity pages themselves and are used to generate URLs for directly accessing new entity pages.

Agarwal et al. [7] describe a method of deduplicating URLs by generating a set of templates and detecting templates that reference the same pages in a different way. The initial step used to generate the rules requires tokenization of the URL. In addition to detecting tokens separated by the standard path or parameter delimiters, a method called deep tokenization is used. This is implemented by extracting multiple tokens from a single standard path element by detecting delimiters in the URL. These delimiters are found by analysing repeating substrings in those path elements between different URLs and considering those to be separate parameters. To reduce the number of URLs that need to be processed, they are ranked based on how close they are to a normalized format and then a certain number of URLs from the top of the ranking is selected for analysis. For this ranking, shorter URLs and URLs with less parameters are preferred, as additional parameters to the same URL often do not provide pages with new content. As the last step, these rules are generalized by grouping them with the key being the rule with the most varying elements of the URL replaced with wildcards.

The normalization method used has several similarities to the current work - tokenization, removal of superfluous path elements and producing final URL templates that are in a format which allows to generate new URLs based on that. The main difference in the current work is that removal of path elements or replacing varying values with a constant is achieved by verifying that they produce the same entity as a result. Another big difference is that the tokenization does not go as deep and only detects an integer either in the beginning or in the end of a path element. The authors of the work have also mentioned the possibility of using the results to generate new URLs.

Manica et al. [8] describe a method for discovering all entity pages of the same type from a site given an input URL to a specific entity page. Their approach considers a website to be a tree of nested entities where the children pages of one entity page belong to the same type. Starting from the input URL, they find its parent entity page which also acts as an index for the entity pages for the level of the given URL. Then they repeat this process until they have reached the root of the entire web site. To detect if a page is at the same level as another page, they compare the DOM trees of those pages. Based on this they can categorize pages into entity pages themselves and pagination pages to get a list of entity pages one level below. This allows them to find all the descendant pages of an arbitrary ancestor of the input page.

The approach used for detecting pages of the same level is similar to the approach used in the current work to detect if a page still presents the same entity after a certain path element was removed. Their Structurally Similar URLS and Pages (SSUP) method checks that the changes between an entity page and its potential sibling page is contained within

the DOM elements dedicated to an entity of that level. Detecting if the presented entity is still the same is an inverse of this, as it verifies that the changes did not affect the entity content itself, but only some elements outside of it.

Blanco et al. [9] describe a method for extracting the content of all structurally similar pages on a web page by taking one sample page as an input. The main focus of their work is on their Page Finder module, which locates all the pages that are structurally similar. It works by first locating link collections on the pages of the web site. One link collection is defined by its location in the DOM path of the web page. All links that are in the same collection are candidates for being structurally similar. The method of finding links used in their work is similar to the parallel paths method described by Weninger et al [10]. It finds the pages to check by following links from the original input page and then visiting links further from only the pages that have a link back to an already visited page. To actually measure the similarity of pages, they compare the distance between their schemas by using the result of Jaccard similarity coefficient with a predefined threshold.

The method described by Blanco et al. [9] compares pages by calculating a numeric value representing the similarity between the pages. In the current work, the methods used for detecting if a page is of the same type is instead done by finding the elements that differ between two pages that are known to be of the same type and expecting the elements where the differences start from to be the same between any pair of pages of the same type. This is a less dynamic approach and can be easily broken by only very little variations in the structure of the pages, whereas the calculated difference is less sensitive to small differences.

3. Method

This method consists of two main parts. The first step is URL template generation which takes a list of URLs as input and outputs templates for entity page URLs with one open variable for the numeric entity identifier, along with the set of known values for that variable.

The second step is generating new URLs based on the templates and visiting them in an order that will maximize the ratio of found working URLs and all generated URLs. This will produce a list of addresses of the new entity pages.

The application requires a list of URLs as input. Since it is highly likely that the effectiveness of this approach varies greatly between different sites, this method was designed to work on the URLs of one domain at a time.

Listing 1. Sample input and output for the entire method.

```
Sample input:
http://example.com/shop/cheap-items/yellow-keychain-1
http://example.com/shop/cheap-items/blue-carpet-2
http://example.com/shop/cheap-items/glowing-onions-4
http://example.com/shop/expensive-items/fancy-car-6
http://example.com/forum?topic=1&page=1
http://example.com/forum?topic=1&page=2
http://example.com/forum?topic=3&page=1
http://example.com/forum?page=2&topic=3

Sample output:
http://example.com/shop/cheap-items/yellow-keychain-3
http://example.com/shop/cheap-items/yellow-keychain-5
http://example.com/forum?page=1&topic=2
http://example.com/forum?page=2&topic=2
```

3.1 URL Template Generation

First, the URLs are grouped under different templates, where a template is identified by the URL by replacing path elements or parameters only containing a number with a constant.

Listing 2. Sample input and output for template generation.

```
Sample input:
http://example.com/shop/cheap-items/yellow-keychain-1
http://example.com/shop/cheap-items/blue-carpet-2
http://example.com/shop/cheap-items/glowing-onions-4
http://example.com/shop/expensive-items/fancy-car-6
http://example.com/forum?topic=1&page=1
http://example.com/forum?topic=1&page=2
http://example.com/forum?topic=3&page=1
http://example.com/forum?page=2&topic=3

Sample output:
http://example.com/shop/{T}/{T}{N}
http://example.com/forum?page=1&topic={N}
http://example.com/forum?page=2&topic={N}
```

Due to the prevalence of various search engine optimization techniques used by web sites, the URLs may contain several parts that are not actually used to find the entity that the page represents. There may be added text path elements before the identifier representing the category of it or its name after it as a separate path element or parameter. The entity identifier may also be prefixed or suffixed by some human-readable name for the entity.

The result of template generation should treat a path element that does not affect the served entity as a constant which can later be replaced with any of the values it had in the input data. These parts of the URL will be referred to as unused.

To optimize this, the initially generated templates are partitioned into groups in which different templates have the potential to be merged into one if they are detected to actually be equal if the differing elements in them are unused. After this, the resulting groups can be analysed independently.

The path elements in a group which consist of text prefixed or suffixed by a number are then checked. If the text part in them does not affect what the web page finds, then those templates are extracted from the group into a separate group where the text path element has been split into an unused text part and a number.

In the resulting groups, path elements are checked, starting from the end. At each step, the templates are grouped in a way where one group of templates could be merged if it was sure that the checked path element was unused. Then it is checked if that element really has no effect on the entity that is displayed by the web page and if it doesn't, those templates can be merged.

Last two steps have to actually check if two URLs represent the same entity or whether an URL represents any entity at all. For that, a DOM tree based page comparison method is used.

The templates at this point may contain multiple numeric parameters. This means they are not yet actually entity page URL templates. As the next step, the templates which contain

multiple numeric parameters are split into multiple templates so that the resulting ones would only have one.

Listing 3. Code for URL template generation.

```
def findUrlTemplates(urls):
   initialTemplates = createInitialTemplates(urls)
   templateGroups = createTemplateGroups(initialTemplates)
   templateGroups = prefixSuffixNumericParametersSplit(templateGroups)
   templateGroups = filterTemplateGroups(templateGroups, numericParamFilter)
   templateGroups = mergeByUnusedTextElements(templateGroups)
   return createFinalTemplates(templateGroups)
```

The resulting templates are subjected to the unused element detection once more with only GET parameters checked this time, to remove any previously numeric parameters that were marked as static text in the last step.

At this stage, the templates are ready to be used for generating new URLs. Templates with a too small set of values for the numeric parameter are first filtered out as they are the least likely to be useful for generating new URLs.

The templates produced by this are intended to serve only the purpose of generating new URLs. As such, the ones where landing on an entity page requires knowing a text identifier or the parameters of multiple values, such as both the index and the name, are excluded during the steps of template generation.

3.1.1 Initial Template Generation

Each URL is parsed and split into separate path elements, with each of them either being text, number or separator. Whether the element was in GET parameters or not is also stored as some later steps are only performed on path elements not part of the GET parameters.

Listing 4. Sample input and output for initial template generation stage.

```
Sample input:
http://example.com/shop/cheap-items/yellow-keychain-1
http://example.com/shop/cheap-items/blue-carpet-2
http://example.com/shop/cheap-items/glowing-onions-4
http://example.com/shop/expensive-items/fancy-car-6
http://example.com/forum?topic=1&page=1
http://example.com/forum?topic=1&page=2
http://example.com/forum?topic=3&page=1
http://example.com/forum?page=2&topic=3

Sample output:
http://example.com/shop/cheap-items/yellow-keychain-1
http://example.com/shop/cheap-items/blue-carpet-2
http://example.com/shop/cheap-items/glowing-onions-4
http://example.com/shop/expensive-items/fancy-car-6
http://example.com/forum?page={N}&topic={N}
```

The GET parameter elements are reordered alphabetically by name so that they could later be compared by index rather than name. This allows the name part to be considered a static part of the URL and GET parameter values will not need as much special treatment in some of the later steps.

Listing 5. Code for initial template generation stage.

```
def buildElements(url):
   domain, locationText, paramsText = splitUrl(url)
   elements = [{type: 'DOMAIN', get: False, values: [domain]}]
   possibleNumericParameter = False

   for location in locationText.split('/'):
      elements.add({type: 'SEPARATOR', get: False, values: ['/']})
      type = 'NUM_VALUE' if isNumber(location) else 'TEXT_STATIC'
      elements.append({type: type, get: False, values: [location]})
      possibleNumericParameter |= startsOrEndsWithDigit(location)

   params = paramsText.split('&').sort()

   for param in params:
      separator = '?' if param is params[0] else '&'
      elements.append({type: 'SEPARATOR', get: True, values: [separator]})

      key, value = param.split('=')
      elements.append({type: 'KEY', get: True, values: [key]})

      if value is not None:
         elements.append({type: 'SEPARATOR', get: True, values: ['=']})
         type = 'NUM_VALUE' if isNumber(value) else 'TEXT_STATIC'
         elements.append({type: type, get: True, values: [value]})
         possibleNumericParameter |= startsOrEndsWithDigit(value)

   return elements if possibleNumericParameter else None
```

```
def mergeTemplates(target, merged):
   target.count += merged.count
   for i in range(len(target.elements)):
      if target.elements[i].type in ['NUM_VALUE', 'TEXT_VALUE']:
         target.elements[i].values += merged.elements[i].values

def createInitialTemplates(urls):
   templates = {}

   for url in urls:
      elements = buildElements(url)

      if url:
         key = generateKey(elements, templateKeyPart)
         template = {key: key, elements: elements, count: 1}

         if template.key in templates:
            mergeTemplates(templates[key], template)
         else:
            templates[key] = template
```

A key is generated for each URL, which is the path elements concatenated with numeric elements replaced by a fixed constant. All URLs where the key is equal are grouped into one URL template. A list of all values is stored for each path element that has been replaced by a constant in the template key.

Templates which contain no unused text parts or where the single numeric parameter is not prefixed nor suffixed by text, the templates generated here are final. In any other cases, these templates may be merged or split in any of the next steps.

3.1.2 Template Grouping

Since there can be some unused path elements in the URL template keys, it is useful to first separate the templates into groups where each template has a chance to be merged with other templates in the same group.

Listing 6. Sample input and output for template grouping stage.

```
Sample input:
http://example.com/shop/cheap-items/yellow-keychain-1
http://example.com/shop/cheap-items/blue-carpet-2
http://example.com/shop/cheap-items/glowing-onions-4
http://example.com/shop/expensive-items/fancy-car-6
http://example.com/forum?page={N}&topic={N}

Sample output:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/cheap-items/yellow-keychain-1
  http://example.com/shop/cheap-items/blue-carpet-2
  http://example.com/shop/cheap-items/glowing-onions-4
  http://example.com/shop/expensive-items/fancy-car-6
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}
```

Two templates are considered for merging if they have the same number of path elements and each numeric path element is at the same place as in the other template. This means that templates with different number of parameters will not be merged.

Listing 7. Code for template grouping stage.

```
def groupKeyPart(element):
   return '{Y}' if element.type == 'TEXT_STATIC' else templateKeyPart(element)

def createTemplateGroups(templates):
   groups = {}

   for key, template in templates:
      groupKey = generateKey(elements, groupKeyPart)
      group = groups[groupKey]

      if group is None:
         group = {key: groupKey, templates: {}}

      group.templates[key] = template

   return groups
```

The reason for this restriction is that the existence or non-existence of a parameter cannot significantly increase the amount of unnecessary work done in later steps. At most, there might be matching templates in two different groups due to that. However, this kind of grouping greatly reduces the amount of total checks needed in the later steps as the number of templates compared to each other is a lot lower, which increases efficiency.

3.1.3 Detecting Suffixed or Prefixed Numbers

When generating the initial templates, a path element was considered to be numeric if it consisted entirely of digits. This step also detects cases where a path element is a number that is either a prefix or a suffix to an unused text part.

Listing 8. Sample input and output for suffix or prefix detection stage.

```
Sample input:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/cheap-items/yellow-keychain-1
  http://example.com/shop/cheap-items/blue-carpet-2
  http://example.com/shop/cheap-items/glowing-onions-4
  http://example.com/shop/expensive-items/fancy-car-6
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}

Sample output:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/cheap-items/{T}{N}
  http://example.com/shop/expensive-items/{T}{N}
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}
```

URLs containing these combined text and number elements are very common on sites that have made their URLs to look more human-readable. Usually the extra text part is generated from the name or description of an entity. As such, it is not vital for the web site for finding the entity.

There are three ways that web sites can handle the text part in these combined elements. One common way is that when the unused part is not the same as expected, the web site redirects to the URL with the correct text part. This makes it very obvious that the text part can be removed and the correct page will always be reached by only having the identifier as the web site will redirect to the appropriate URL anyway.

Some sites may expect the text part to be correct and will not show the page otherwise. In this case, merging the templates with different text parts on that path element is not useful since even though it is possible to generate new numbers for the identifier part, it is not possible to accurately predict the text part generated from the entity name or description.

The third common handling of those suffixes or prefixes is that it is completely ignored and changing that part to any random text will produce an identical web page. However, verifying that the new page is the same as before might sometimes be tricky, because web pages may have dynamically generated parts that change even when the displayed entity remains the same. The DOM structure based page comparison method described in later chapters is used in this case.

For each non-GET text path element, starting from the last, the following actions are taken on a group of URL templates. If that path element is prefixed or suffixed by a number in a template, a new template category key is generated where both the text part and number part of the path element being looked at is replaced by fixed constants. This may result in several templates being in the same category.

If the text part of the templates in the same category is verified to be unused, those templates can be merged together. Since one path element has now been split into two, the number of path elements has increased and the template is moved to a new group which is added to the list of groups that have yet to be processed in this step.

Listing 9. Code for suffix or prefix detection stage.

```
def applySplitMergePerTextGroup(groups, group, merge, groupsByText):
   for byText in groupsByText:
      if len(groupByText.templates) >= MERGE_MINIMUM_VALUES:
         createSplitTemplate(groups, merge.type, merge.index, byText.text,
merge.templates)
         removeFromGroup(group, merge.templates)

def applySplitMergeToAll(groups, group, merge):
   createSplitTemplate(groups, merge.type, merge.index, None, merge.templates)
   removeFromGroup(group, merge.templates)

def canSplitMergeAll(groupsByText):
   for i in range(1, min(MERGE_VERIFY_DEPTH, len(groupsByText))):
      if not canSplitMerge(groupsByText[0], groupsByText[i]):
         return False
   return True

def verifyAndApplyPrefixSuffixMerge(groups, group, merge):
   if len(merge.numbers) < len(merge.templates):
      return
   elif merge.numbers < MERGE_MINIMUM_VALUES:
      return

   groupsByText = getByTextGrouping(merge)

   if canSplitMergeAll(groupsByText):
      applySplitMergeToAll(groups, group, merge)
   else:
      applySplitMergePerTextGroup(groups, group, merge, groupsByText)

def prefixSuffixProcessGroupElement(groups, group, elementIndex):
   groupSplit = {index: elementIndex, mergePairs: {}}

   for template in group.templates:
      element = template.elements[elementIndex]
      if element.type != 'TEXT_STATIC' or element.get:
         return

      for splitType in ['PREFIX', 'SUFFIX']:
         parts = splitWithType(splitType, element.values[0])
         if parts:
            mergeKey = generateMergeKey(template.elements, elementIndex)
            pair = findOrCreateMergeFromGroupSplit(groupSplit, mergeKey)
            pair[splitType].numberSet.append(parts.number)
            pair[splitType].templates.append({template: template, parts: parts})

   for pair in groupSplit.merges:
      if len(pair['PREFIX'].numbers) >= len(pair['SUFFIX'].numbers):
          verifyAndApplyPrefixSuffixMerge(groups, group, pair['PREFIX'])
      else:
         verifyAndApplyPrefixSuffixMerge(groups, group, pair['SUFFIX'])

def prefixSuffixProcessGroup(groups, group):
   for elementIndex in range(getGroupPathElementCount(group) - 1, 1, -1):
      prefixSuffixProcessGroupElement(groups, group, elementIndex)

def prefixSuffixNumericParametersSplit(groups):
   finalGroups = {}

   while len(group) > 0:
      key, group = groups.popitem()
      prefixSuffixProcessGroup(groups, group)
      finalGroups[key] = group

   return finalGroups
```

One possible problem for this approach is when there is an additional full path element that is always different for every entity. This would cause every category of URL templates to contain only one template and therefore no merging would be done. However, as this does not seem to be a common case, these cases are ignored for the sake of the simplicity of this algorithm.

3.1.4 Detecting Unused Path Elements

As another method of search engine optimization, many web sites add extra elements to the URL that are designed only to make the address more human-readable and give more context to search engines.

These extra elements are often a category or type of an entity, which may differ even for entities whose numeric identifiers have been generated from the same increasing index. As such, it would be ideal to merge them into one template.

Merging those templates is only useful if the values of the extra unused elements are interchangeable in the URLs, otherwise many different values for the extra text element would have to be tried for every generated URL for that template, which would greatly increase the number of broken URLs generated later.

Listing 10. Sample input and output for unused path element detection stage.

```
Sample input:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/cheap-items/{T}{N}
  http://example.com/shop/expensive-items/{T}{N}
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}

Sample output:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/{T}/{T}{N}
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}
```

The steps followed to find which categories can be merged together are similar to the ones used in the previous chapter. For each path element, starting from the last one, the templates are put into a category, where the key is the template key for the case where that path element was unused, which is just that path element replaced by a fixed constant.

If there are multiple URLs per template in the templates being merged, then that path element is considered to be a category name and three URLs are being used for detecting if it can be removed. Two from one template with different entity indices and then one of those with the currently checked path element replaced with the value from another template that is a candidate for merging.

The category merge can only be performed if the third URL does not respond with a non-success code and comparing the first and second produces differences that are not present in the comparison result between the first and third.

If there is only one URL per template in the templates being merged, then that path element is unique to one URL and therefore unique to one entity. In that case, it is not possible to take multiple known entity URLs without first assuming the merge is possible, which is what needs to be checked in the first place.

In this case, two URLs are compared. First is the URL from one template and the second is the same as the first one with the element being considered from merging replaced with a value from another template. As with category, the result is obvious in case of redirections or error codes. Otherwise, there may be no differences other than the random ones occurring on every reload and the ones caused by the current page URL used in the source.

Listing 11. Code for unused element detection stage.

```
def verifyCategoryMerge(merge):
   templates = sorted(merge.templates, key=lambda x: -len(x.count))

   for i in range(1, min(MERGE_VERIFY_DEPTH, len(merge.templates))):
      if not canCategoryMerge(merge.templates[0], merge.templates[i]):
         return False
   return True

def verifyNameMerge(merge):
   for i in range(1, min(MERGE_VERIFY_DEPTH, len(merge.templates))):
      if not canNameMerge(merge.templates[0], merge.templates[i]):
         return False
   return True

def applyUnusedMerge(group, merge):
   first = merge.templates[0]

   newElements = first.elements
   values = first.elements[merge.index].values[0] * first.count
   changedElement = {type: 'TEXT_VALUE', get: False, values: values}
   newElements[merge.index] = changedElement

   templateKey = generateKey(newElements, templateKeyPart)
   newTemplate = {key: templateKey, elements: newElements, count: first.count}

   for i in range(1, len(merge.templates)):
      template = merge.templates[i]
      mergeTemplates(newTemplate, template)

      value = template.elements[merge.index].values[0]
      changedElement.values += [value] * (template.count - 1)

      group.templates.remove(template.key)

   group.templates.put(newTemplate.key, newTemplate)

def verifyAndApplyMerge(group, merge):
   if len(merge.templates) < 2:
      return
   elif bestUniqueValueRatio(merge.templates) < 0.95:
      return

   totalCount = getTotalUrlCount(merge.templates)
   result = False

   if len(merge.templates) >= 0.95 * totalCount:
      if len(merge.templates) >= MERGE_MINIMUM_VALUES:
         result = verifyNameMerge(merge)
   else:
      result = verifyCategoryMerge(merge)

   if result:
      applyUnusedMerge(group, merge)

def generateMergeKey(elements, elementIndex):
   parts = [templateKeyPart(element) for element in elements]
   parts[elementIndex] = '{T}'
   return ''.join(parts)

def mergeUnusedProcessGroupElement(group, elementIndex):
   groupMerge = {index: elementIndex, merges: {}}

   for template in group.templates:
      element = template.elements[elementIndex]
      if element.type != 'TEXT_STATIC' or element.get:
```

```
            return

        mergeKey = generateMergeKey(template.elements, elementIndex)
        merge = findOrCreateMergeInMergeGroup(groupMerge, mergeKey)
        merge.templates.append(template)

    for merge in groupSplit.merges:
        verifyAndApplyMerge(group, merge)

def mergeUnusedProcessGroup(group):
    for elementIndex in range(getGroupPathElementCount(group) - 1, 1, -1):
        mergeUnusedProcessGroupElement(group, elementIndex)

def mergeByUnusedTextElements(groups):
    for key, group in groups:
        mergeUnusedProcessGroup(group)

    return groups
```

After this step, the templates in one group of templates should either represent entities which have identifiers from different series, or display different type of data for an entity. As there are no additional numeric path elements generated from here on, all templates without a numeric element are filtered out as they cannot be useful for generating URLs.

3.1.5 Finalizing Templates

For generating new URLs from templates, the templates must be in the format where there is one numeric variable and the rest of the template is static. For that, all previously detected unused text parts in a template must be replaced by one random value from the set of values seen for that element.

That still leaves the case where there are multiple numeric parameters in a template. In those cases, the template has to be split in a way where the new templates have only one remaining variable.

To split the template, one numeric variable is selected to be the entity identifier and the rest will be considered to be static parts of the template.

Listing 12. Sample input and output for template finalization stage.

```
Sample input:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/{T}/{T}{N}
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}

Sample output:
http://example.com/shop/{T}/{T}{N}
http://example.com/forum?page=1&topic={N}
http://example.com/forum?page=2&topic={N}
```

It is very likely that the numeric variable that refers to an entity has a larger set of different values than any other, so the one selected to be the entity identifier is the one which has the largest set of unique values.

Listing 13. Code for template finalization stage.

```
def finalizeTemplate(template):
   numIndex = None
   highestUniqueValueCount = 0
   index = 0
   numericElementCount = 0
   finalTemplates = {}

   for element in template.elements:
      if element.type == 'NUM_VALUE':
         uniqueValueCount = len(set(element.values))
         if uniqueValueCount > highestUniqueValueCount:
            numIndex = index
            highestUniqueValueCount = uniqueValueCount
         numericElementCount += 1
      index += 1

   numElement = template.elements[numIndex]

   for i in range(template.count):
      singleNumberKey = generateSingleNumberKey(template.elements, i, numIndex)
      finalTemplate = finalTemplates[singleNumberKey]
      if finalTemplate:
         finalTemplate.elements[numIndex].values.append(numElement.values[i])
         finalTemplate.count += 1
      else:
         finalTemplate = buildSingleNumberElements(elements, i, numIndex)
         finalTemplates[singleNumberKey] = finalTemplate

   for key, template in finalTemplates:
      for i in range(len(template.elements)):
```

```
        template = template.elements[i]
        if element.type == 'NUM_VALUE':
           template.finalValues = sorted([int(x) for x in element.values])
           template.urlWithPlaceholder = nthUrlWithPlaceholder(template, 0, i)
           template.knownValue = int(element.values[0])


def createFinalTemplates(groups):
   finalTemplates = []

   for groupKey, group in groups:
      for key, template in group.templates:
         finalTemplates += finalizeTemplate(template)
```

For every template, each set of values, which correspond to one URL in the original data set, is then applied to the template key so that all the numeric parameters that are not the one chosen to be the entity index are replaced with the value itself as static text. The value set is added to its new template identified by the produced template key.

3.2 Comparing Pages

Several different steps in template generation require the ability to tell if a page contains the same entity content as another page or if a page contains any entity at all.

To detect if a page is empty, the simplest way is when the web server responds with the appropriate error code. It could also respond with a redirect to an error page, which can also be easily detected.

Another way to detect if another URL gives the same content as the first one is if the other URL redirects to the first one. This means that some parts of the URL might not be correct for the specific page, but the server fixes the URL for you by redirecting to the correct page.

When comparing, an URL for a page which is supposed to be empty is also generated, by setting the numeric value under inspection to a very high value that is unlikely to correspond to any actual entity. If the empty page results in a redirect or error code, then it means that every URL generated from the same template that does not do the same is a valid entity page.

Some sites do not serve proper error codes when given an invalid entity index. In this case it is necessary to look into the page content to detect whether the checked URL gives an empty page or the same entity as another URL. The approach used for that is based on comparing the DOM tree.

When comparing two pages, the result of the comparison is a list of changes. Each change itself is a list of numbers which indicates an index within the parent element, which essentially is a path to the element that differs.

An element is considered to be different from its corresponding element on the other page if its type, tag or attributes differ or the children elements do not match in the same order, where every corresponding element must have the same type and tag.

Some parts in the page content are excluded from comparison since they might be automatically generated from the current URL, in which case they would always be different for different URLs. These exclusions include the contents of meta and title elements and attributes containing URLs. In case of text elements, if the text contains a direct URL, that text is marked as equal for the same reason.

Every time when a page is compared to another one, first the initial page is loaded twice to record the differences that occur due to randomly generated content on the page. These will be excluded from any differences produced later to reduce the number of false negatives. Excluding a difference from a set of differences means removing all differences for which it is a prefix or an exact match.

The prefix and suffix detector requires knowledge about whether an URL leads to an empty page. If none of the response code based solutions work, the contents of three pages are compared: original URL, empty page URL and the alternative URL which would point to original page if it does not lead to an empty page. If all the differences between original and alternative URL were removed by excluding the random differences, but there are differences between empty page and original page, then the alternative URL is considered to be valid.

For removing text path elements which are considered to be entity name, there is less information to use. Original page is considered not to be empty and if the original and alternative URLs match, the path element is marked as unused.

When removing path elements acting as categories, two original URLs with the same category element value are used and one alternative URL for one of those with the category path element replaced with one from another template. When page content comparison is necessary, alternative URL is valid when it does not differ from the original of that same entity, but the original is different from the result of the other entity page.

Validating generated URLs uses similar comparison to prefix and suffix detector, except the generated URL is also valid if it redirects to any URL that matches the template it was generated from and instead of comparing an alternative URL of the original URL, the generated URL is for a different entity. Due to that, the other entity URL is not considered to be valid if it matches either the first entity page or the empty page exactly.

3.3 Using URL templates

At this point, there is a list of URL templates with one numeric variable and a list of values, also known as entity identifiers, seen for that variable. For every template, the set of known entity identifiers is traversed in order to detect all the gaps in values. A gap is a range of values from A to B where none of the values between A and B exist in the set, but A-1 and B+1 do.

Listing 14. Sample input and output for URL generation.

```
Sample output:
Group http://example.com/{Y}/{Y}/{Y}
  http://example.com/shop/{T}/{T}{N}
Group http://example.com/{T}?page={N}&topic={N}
  http://example.com/forum?topic={N}&page={N}

Sample output:
http://example.com/shop/{T}/{T}{N}
http://example.com/forum?page=1&topic={N}
http://example.com/forum?page=2&topic={N}
```

For each gap from A to B, a certain number of continuous values (X) starting from A and ending with B are checked. Then the rest of the gap not covered by that is sampled at a certain interval (Y). All the generated values are used with the URL template to generate new URLs. The new URLs are checked to see if they point to an entity page. In case the ratio between the number of working URLs and generated URLs exceeds a threshold, the value of X is increased, Y is decreased and the process is repeated. The threshold is reached when the ratio between the URLs that were detected not to be empty versus all tried URLs is above a predefined value.

Once a fixed limit of generated URLs, the total success ratio is checked after every template is processed and the whole operation is halted when the ratio is too low. This guarantees that the application does not send too many queries to the web site without actually getting any results.

Listing 15. Code for URL generation.

```
def recordGeneratedUrl(aggregate, result, generatedUrl):
   if generatedUrl:
      aggregate.validUrls.append(generatedUrl)
      result.validurls.append(generatedUrl)
   aggregate.totalQueries += 1
   result.totalQueries += 1
```

```
def checkIndices(aggregate, result, ids, template):
   idList = list(ids)
   random.shuffle(idList)

   originalUrl = template.urlWithPlaceholder.replace('{N}', template.knownValue)
   emptyUrl = template.urlWithPlaceholder.replace('{N}', VERY_HIGH_VALUE)

   for index in idList:
      generatedUrl = template.urlWithPlaceholder.replace('{N}', str(index))
      if not compareOtherEntityUrl(originalUrl, emptyUrl, generatedUrl):
         generatedUrl = None
      recordGeneratedUrl(aggregate, result, generatedUrl):

def generateFromTemplate(aggregate, template):
   result = {template: template, validUrls: [], totalQueries: 0}
   edgeX = INITIAL_X
   intervalY = INITIAL_Y
   processedIds = set()

   while intervalY >= MINIMUM_Y and edgeX <= MAXIMUM_X:
      currentIds = set()
      expectedValue = template.finalValues[0]

      for value in template.finalValues:
         if expectedValue != value:
            addGapIndices(currentIds, expectedValue, value, edgeX, intervalY)
         expectedValue = value + 1

      currentIds.difference_update(processedIds)

      if not checkIndices(aggregate, result, currentIds, template):
         return

      checkedIds.update(currentIds)

      intervalY /= 2
      edgeX *= 2


def generateNewUrls(templates):
   aggregate = {validUrls: [], totalQueries: 0}

   for template in templates:
      generateFromTemplate(aggregate, template)
      if isFailedCondition(aggregate, FAILED_MINIMUM, FAILED_RATIO):
         break

   return aggregateResults
```

4. Evaluation

The data used for evaluating this comes from crawling websites with Estonian top level domain crawled in July-August 2015. Since the crawler was working for over a year before that, this only represents the partial results of the crawling. As such, the results are not comparable to use cases where the data set contains all the links reachable via the internet. Instead it shows its efficiency in cases where the results are partial and contain few links from other websites, which might be the case for focused crawlers.

The raw data is in the format of metadata archives generated by Heritrix [11] crawler. These files contain the list of all addresses that were visited along with response code and mime type of the result. The test application first filtered that archive for URLs that returned success code and HTML output. The list of URLs that passed the filter were put into separate files based on the domain name.

The total number of domains visited in the data set was around 17000. For over 10000, the file containing the list of the URLs for that domain was less than a kilobyte, indicating a very low number of results.

The test set used for evaluating the results of the application was chosen as 50 domains where the size of the URL list was between 850 and 1150 kilobytes. Since the length of URLs differs per site, the number of URLs for these domains was between 6000 and 21000. The data set size was selected to be small enough to verify the results manually and to see meaningful patterns in the per domain results without having to first aggregate the results of different domains.

The application used for testing the method is written in Java. It uses the Apache HttpComponents [12] library to perform HTTP queries and jsoup [13] library to get the DOM representation of HTML pages used for comparing the content of different pages. Additional used libraries include Apache Commons utilities and logging libraries. The application is built using a Gradle [14] script.

It is a command line application which is started without any parameters and then commands can be entered. One of the commands is named “help” and prints out the format of other commands. Other commands include one for splitting warc files into domain-named text files containing the URLs from that domain, and a command that performs template generation and URL generation for a specified domain-named file containing URLs.

4.1 Template generation precision

The application was configured to filter out templates that match too few URLs. In this specific case the threshold was set at 100. This value was selected because after lowering it any further, the additional templates that were detected were of low quality on average, which is demonstrated in the next chapter.

For 29 of the domains being analysed, no templates were detected that passed the threshold of 100 matching URLs. The list of templates for the other domains were manually inspected to detect whether they are not entity page templates, are too specific templates or in other cases relevant templates.

Table 1. Per domain template precision.

TOTAL	NOT ENTITY	SPECIFIC	RELEVANT	RATIO
11	1	3	7	64%
12	0	4	8	67%
3	2	0	1	33%
2	0	1	1	50%
17	0	8	9	53%
22	0	18	4	18%
96	94	1	1	1%
13	3	6	4	31%
5	0	0	5	100%
8	3	1	4	50%
2	0	0	2	100%
8	0	6	2	25%
1	0	0	1	100%
16	0	12	4	25%
8	1	3	4	50%
7	3	0	4	43%
8	4	1	3	38%
3	3	0	0	0%
2	0	0	2	100%

1	0	0	1	100%
6	0	4	2	33%
TOTAL				
251	114	68	69	51%

In table 1, total number shows the number of templates that were detected and which passed the filters. These were split into three different categories.

If the variable number in the template is not an entity identifier, then the template is not considered to be an entity page template. One case that caused these templates to be detected is when the number was related to pagination, such as a page number or an offset in some list of results. This was the case for the single instance where non-entity templates heavily outnumbered other templates.

Cases where a page number was present in the template, but as static text, are considered to be relevant templates as they are likely to give different content from each other, while still related to the same entity which that list being paginated is tied to. Similar case applies to templates which have the language specified in them. From indexing perspective, finding the content for every language would be important and therefore they are all relevant results.

Another class of templates that are not considered to be entity page templates are those where the modified number in the URL is actually an entity index, but the page it returns does not display any information about that entity. In this data set, there are some cases where the entity index in the URL was used as an indicator where the user will be redirected once a certain action is performed by the user.

The detection of this kind of templates is caused by the page contents of templates that do not go through any merge or split process not being checked at all prior to URL generation. When there are no other numeric parameters, this only causes that template to be a false positive. However, if the number of different values for that parameter exceeds the one for another entity index that is actually used in the URL, it will cause that other template not to be detected.

Another type of non-relevant templates is the case where the same template is duplicated by another template that differs by parameters that do not affect the content displayed about that entity. This was often caused not by the difference in values of those parameters, but whether that parameter existed or not. Since the grouping method used for templates never compares templates that have a different set of parameters, the method does not manage to merge these templates into one. This causes false positives that most likely will cause duplicate URLs found in the URL generation process.

After excluding these two types of non-relevant templates, the other templates can be considered to be relevant. The average ratio of relevant templates per domain on this data set is approximately 51%, while the median is 50%. This ratio is also known as the precision of the method.

When looking at all domains together, the average ratio is just 28%. This indicates that a high number of detected templates means a higher accuracy. This makes sense as usually a website should contain a fixed number of entity page templates and if the number of de-

tected templates is high, which means that the template itself contains some fixed variable that may actually have a large number of different values.

4.2 Template generation recall

The ratio of relevant templates gives an insight into the false positives. Another important thing to look at is false negatives. To reliably tell which templates the method should have detected requires manual inspection of a large number of URLs in the original data set. It would be best to first try to get an idea of what might cause false negatives by looking at the statistical data of what happened to the templates inside the different stages of the method for every domain where no templates were detected.

In table 2, each stage represents a point after which the number of templates may have changed. Step INIT is the initial template generation, which merges URLs to a templates only if a path element is wholly a number. Step PREFIX is merging templates considering by detecting path elements that are a number concatenated to text that does not affect page content. Step FILTER filters out all templates which have no numeric element in the path. Step UNUSED removes unused path elements that are not GET parameters. Step NUM splits templates in a way that only one numeric path element may remain and others are considered to be static text. Step UNUSED2 removes unused path elements that are in GET parameters. For all of these, after the final filter of requiring 100 different URLs per template the template count reached 0.

Table 2. Per domain template counts per step.

URLS	INIT	PREFIX	FILTER	UNUSED	NUM	UNUSED2
9405	1719	1719	903	903	212	212
6298	6156	6156	77	77	0	0
11961	4121	4121	1089	1089	72	72
7461	410	410	5	5	0	0
11527	1079	1079	510	510	389	389
11234	381	381	272	272	1250	1250
8887	6711	6711	515	515	0	0
24791	24775	24775	3	3	1	1
10152	9003	8974	2	1	1	1
15236	9333	9248	357	357	12	12
8831	95	95	95	95	483	477
11684	18	18	17	17	0	0

4204	42	42	40	40	1428	1409
8730	496	487	366	366	379	379
10403	4291	4291	966	966	714	714
7973	623	623	81	81	9	9
9377	577	577	577	577	1123	1123
9292	1677	1621	1061	1061	208	208
7240	27	27	8	8	1	1
22063	45	45	0	0	0	0
24509	33	33	33	33	306	306
6525	6502	6502	16	16	0	0
9093	2153	2153	247	247	32	32
11146	485	485	12	12	0	0
11747	4017	4017	2715	2715	251	251
9431	7612	7612	14	14	0	0
16668	15860	15860	67	67	22	22
8933	664	664	4	4	19	19
13957	2918	2918	166	166	16	11
TOTAL						
328728	111823	111644	10218	10217	6928	6898

To make sense of this, each different pattern of changes should be looked at. Since the interesting part is which step might possibly be responsible for false negatives, different points where the template count reaches zero are worth looking at.

The first place where this has happened in this data set is the numeric parameter filter (FILTER). Based on the URLs in the domain where this happened, it appears that the site concatenated different parameter values together into one GET parameter key, not even a value. Since it is an uncommon approach to have variable GET parameter key names, this method was never intended to cover that use case even though it is a possible cause of false negatives.

The next stage (NUM) where the template count reached zero for some domains is splitting the templates to get the numeric variable count to one. The reason why this step can reduce the number of templates is that there is a minimum number of URLs that must

match any template that is created in this step and that value was set to 10 in this case. Removing that restriction did not prove useful as all these domains still produced zero templates even without this restriction. What all of these domains have in common is that the list of URLs does not contain almost any numeric identifiers and those which were detected in the previous steps just happened to be text parameters with a numeric value. These cases are not false negatives as no useful templates for number-based URL generation can be created.

Another case may be that for these domains, the threshold of 100 URLs per template is just too high. When the threshold was reduced to 50, 4 of these 29 domains produced at least one template. 2 of those contained only a long list of non-entity templates. Considering that the set of 21 domains contained only one similar case, then these 2 out of 4 may hint that the threshold and prevalence of such cases is related. Considering that reducing the threshold by half increased the number of domains with at least one relevant template by only 4% of the whole data set, it appears that reducing it does not significantly improve the results.

After looking manually over the URL lists for all of the 29 domains, only 7 appeared to have an entity index in a noticeable number of URLs. One of those is the one where GET parameter key was used and 4 were the ones which produced templates with the threshold set to 50. That leaves only 2 other domains with false negatives. Those 2 domains had a lot more than 100 different URLs per template, so catching those would require improving the method rather than decreasing the threshold.

Overall this means that the data set had 27 domains for which valid entity page templates exist and for 20 of the domains, at least one of those templates was detected, which means that at the domain level, the recall ratio is 74%.

4.3 URL generation

The URL generation part was configured to only visit a maximum of 200 URLs per template in random order and maximum of 10 templates per domain, ordered by the number of matched URLs in decreasing order. The limit of URLs per domain should not have a significant effect of the ratio of retrieved URLs per all attempted URLs since the selected subset to visit is random. The statistical effect of processing a limited number of templates based on matched URL count was not measured.

In order to evaluate this step independently, only the templates that were considered to be relevant were included when evaluating the URL generation. For non-entity pages it is not possible to validate whether the empty page detection works correctly as often no values for those templates even produce empty pages. For templates that are too specific, their results duplicate the results of the less specific template.

Some of the relevant templates were not tested because they were not part of the top 10 templates by URL count. Some others were not tested because there were no URLs to generate since there were no gaps in their value sets.

The empty page detection does not work correctly for some templates, in which case it either detects every page to be empty, not empty, or the classification does not match the actual result in some cases. The URL generation for a template is only considered valid if it accurately detects whether the page is empty (error page) or presents an entity.

Validating the accuracy of empty page detection was done by manually visiting around 10 URLs that were detected to be empty and 10 that were detected to be valid. In case the visual result matched the classification made by the application on every manually visited URL, the URL generation for that template was considered valid. This might allow for the rare cases where there were very few wrong classifications and they were not present in the sample that was manually checked.

For some templates, there were zero valid generated URLs. The cause of this is most likely that the set of input URLs contained every valid page for a template, which means that the crawler already visited all valid pages.

Table 3. Per domain URL generation results.

DOMAIN	RELEVANT	TESTED	VALID	ZERO	RATIO
FOORUM.KIPPER.EE	7	6	6	1	74%
NISSAN.WIRUAUTO.EE	8	6	1	0	90%
POOD.SEEGI.EE	1	1	1	1	0%
SHOP.PILLIPOOD.EE	1	1	1	1	0%
TARKINVESTOR.EE	9	6	5	0	42%
KOOLITUSED.EE	4	2	2	0	19%
KORGESSAARE.EE	1	1	1	0	93%
LANDROVERCLUB.EE	4	3	3	1	53%
MESINDUS.EE	5	5	5	2	61%
MIMIKO.EE	4	4	1	1	0%
MU.EE	2	2	1	0	5%
NEOSTORE.EE	2	2	2	2	0%
OKILVES.EE	1	1	0	0	-
ROPK.EE	4	4	2	0	3%
SHOOTING.EE	4	4	4	1	58%
SMART24.EE	3	3	3	1	7%
TANKLAVARUSTUS.EE	3	3	3	3	0%
TTU.EE	2	2	1	1	0%
VANARAAMAT.EE	1	1	1	1	0%

WESTSTAR.EE	2	2	2	0	43%
TOTAL	68	59	45	16	27%

The results shown in table 3 show that in cases where empty page detection worked correctly, new valid URLs were found for most domains. There were 7 domains where empty page detection worked correctly for at least one template, but no valid new URLs were found. For 12 domains, valid results were found. Which means that new URLs were discovered for 60% of the domains which had relevant templates.

For 7 domains, there was at least one template where empty page validation did not work correctly. In one case, this was caused by the web page having changed its entity page address and redirected to the new place. Redirecting to an address that has a different template is considered as redirecting to an error page. For other cases, the page content comparison did not work correctly and needs improvement to work with more different web sites.

Overall, the results indicate that new working URLs were successfully generated for 24% of the domains that were tested.

5. Conclusions

This thesis describes a method of finding entity pages using a list of URLs visited by a web crawler. The pages found using this method are either not reachable via links or not present in the data set due to the data intentionally being partial.

The method works by first generating a list of URL templates which contain a pattern with one open numeric variable and a list of entity indices that were used for that variable in the input URLs. Entity index can either be in the value of a GET parameter, an URL path element separated by a slash from other elements or a suffix or prefix of a path element that contains text that is not actually required by the web site. URLs where other parameters are different may be merged into one template in case it is detected that the differing elements do not affect the entity content of the page.

Based on experiments on a limited data set, the average precision of detected templates is around 50% and the recall based on the number of domains where templates were expected is around 75%. There is a lot of room for improvement, but this already works for many web sites.

URLs are generated by inserting values that are within the minimum and maximum known entity indices for a template, but are not present in the set of indices from input URLs for that template. In the experiment on a data set of 20 domains with relevant templates, new valid URLs were found on 12 of them.

Generating URLs heavily relies on empty page detection. The empty page detection method described in this thesis has some limitations, due to which it may give all false positives or all false negatives for some templates. This is one of the main bottlenecks of this method and if replaced with a more reliable method of empty page detection could significantly improve the overall reliability of this method.

In general, this method provides useful results for most sites it was tested on and as such would already be useful, especially in cases of limited data, such as combined with focused crawlers or used with partial data sets.

6. References

[1] "Search and email still top the list of most popular online activities". http://www.pewinternet.org/2011/08/09/search-and-email-still-top-the-list-of-most-popular-online-activities/ (accessed 11.05.2016)

[2] Michael Bergman, "The Deep Web: Surfacing Hidden Value". http://quod.lib.umich.edu/j/jep/3336451.0007.104?view=text;rgn=main (accessed 11.05.2016)

[3] Barbosa, Luciano, and Juliana Freire. "An adaptive crawler for locating hidden-web entry points." *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007.

[4] He, Yeye, et al. "Crawling deep web entity pages." *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 2013.

[5] Raghavan, Sriram, and Hector Garcia-Molina. "Crawling the hidden web." (2000).

[6] Álvarez, Manuel, et al. "Crawling the content hidden behind web forms." *ICCSA (2)*. 2007.

[7] Agarwal, Amit, et al. "URL normalization for de-duplication of web pages." *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009.

[8] Manica, Edimar, Renata Galante, and Carina F. Dorneles. "SSUP–A URL-Based Method to Entity-Page Discovery." *Web Engineering*. Springer International Publishing, 2014. 254-271.

[9] Blanco, Lorenzo, Valter Crescenzi, and Paolo Merialdo. "Efficiently Locating Collections of Web Pages to Wrap." *WEBIST*. 2005.

[10] Fumarola, Tim Weninger Fabio, et al. "Growing Parallel Paths for Entity-Page Discovery." (2011).

[11] Heritrix archival crawler. https://webarchive.jira.com/wiki/display/Heritrix (accessed 11.05.2016)

[12] Apache HttpComponents. https://hc.apache.org/ (accessed 11.05.2016)

[13] jsoup: Java HTML Parser. https://jsoup.org/ (accessed 11.05.2016)

[14] Gradle – Modern Open-Source Enterprise Build Automation. http://gradle.org/ (accessed 11.05.2016)

Appendices

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Ago Allikmaa**,
(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive license) to:
Using URL Templates to Find Hidden Entity Pages,
(*title of thesis*)

supervised by Peep Küngas,
(*supervisor's name*)

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright.
2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **11/05/2016**