Reinforcement Learning and the Web Crawling Problem

Lim Jia Yee

November 11, 2016

Contents

Abstract

Web crawlers browse the World Wide Web to provide users with the most up-to-date data. However, due to the ease of user content creation, the World Wide Web is an arguably infinite information space. As such, executing classical search algorithms such as breadth-first search and depth-first search for web crawling becomes less practical. One solution to this impractical computing cost would be distributed web crawling [1], which transfers the cost to hardware. This paper offers an alternative to this hardware-intensive solution, implementing reinforcement learning in web crawlers.

1 Introduction

1.1 Motivation

Under limited computational resources, such as time and space, it is not practical to crawl the web without ultimately deciding whether a web page deserves to be parsed, as parsing drains resources during crawling, while decision making can be a largely offline process.

Machine learning has been a key solution to many problems involving data and trends, some of which include classification, prediction, and even decision-making. This paper aims to explore the possibility of enhancing web crawling through data-backed decisions in machine learning.

1.2 Relevance

As users become increasingly connected, the World Wide Web expands exponentially due to an explosion of user content. Web crawling algorithms must be consistently maintained and enhanced wherever possible in order to fulfil the requirements of the users despite the burgeoning web. Some of the most critical requirements include:

1. Fast crawling

2. Relevant results

3. Widespread search

This paper aims to propose an implementation a single-process web crawler which meets the requirements above. The focus will be on decision-making, as this paper proposes to optimise web crawlers through planning and informed decision-making. Deciding beforehand whether a web page should even be parsed will save considerable resources as long as the resources needed to make a decision is less than the resources needed for parsing. As such, the web crawling problem can be summarised into the following problem statement:

How can a web crawling algorithm remain optimal as the World Wide Web expands?

1.3 Expected Input and Output

An end user will pass in a seed URL and a list of search words and/or phrases. A default seed URL will be used if not specified. The expected output will be a list of relevant URLs and/or content. A URL is deemed as relevant if its corresponding content contains keywords associated with the search words or phrases in meaning (e.g. banana and yellow fruit).

The user may also configure other components of the program, such as the decision boundaries for the decision-making and relevance-evaluation component.

1.4 Challenges

There are numerous technical components to an effective crawler. For crawling to be fast, optimised exploration and parsing programs are required. However, being able to decide what should not be parsed in the first place based on shorter text (i.e. URL itself) will certainly speed up the crawling process. As such, the potential technical challenges to such smart crawling include:

1. Optimising the general crawling algorithm (i.e. graph exploration algorithm).
2. Deciding if a web page should be visited and parsed based on the URL.
   (a) Assembling features from which the relevance of the content can be deduced from the URL.
3. Deciding if a web page is relevant when parsing it.
   (a) Formulating the relevance score for content to confirm (1).
   (b) Setting the decision boundary or threshold between relevant and irrelevant content.
4. Formulating the reward for parsing relevant content **and** not parsing irrelevant content.
5. Optimising the parsing performance.
6. Determining the seed (i.e. start point) web pages.
7. Parallelising the algorithm for scalability.

1.5 Structure

The structure of this paper is as follows:

1. Decomposing web crawling into a graph exploration and decision making problem.
2. Pre-analysis of the current graph exploration algorithms.
3. Reinforcement learning (RL) and its possible role in web crawling.
4. Pre-analysis of the effectiveness and feasibility of RL in web crawling.
5. Proposed implementation.
6. Experimental results and analysis.

2 Problem Decomposition

2.1 Graph Exploration

2.1.1 Graph

The graph is a graph of the World Wide Web and its **web pages**. The graph is a *directed, unweighted, and unknown* graph. The graph is directed because a web page may have a link to another page, but that page may not necessarily have a link back to the page which the click came from, unweighted because there is initially no insight about web pages with specific characteristics, and unknown because these web pages are dynamic.

2.1.2 Vertices and Edges

The *vertices* are the web pages, and the *edges* are the universal resource locators (URL) which lead users to a specific web page (a vertex) [2].

2.1.3 Other Information

The graph may be subjected to changes over time as users are free to create, update, or delete their content, so there is no database of all vertices and edges.

2.2 Decision Making

The cycle of creating requests to a host, and parsing the corresponding web page is a time-consuming process. Deciding whether to parse certain web content based on a smaller set of data, such as URL, will considerably save computational resources, as irrelevant web content can simply be skipped and not parsed.

3 (Discussion) Solution Part I: Graph Exploration

As graph exploration is a key component to web crawling, some of the existing graph exploration algorithms will be discussed and analysed here.

3.1 Depth-First Search (DFS)

DFS is likely to create continuous requests to the same host as it searches through each host exhaustively [3] before proceeding with the next same-level neighbour. This not only leads to the exploration of a narrow scope of hosts, but also the unnecessarily exhaustive exploration of these hosts.

Furthermore, the relevance of a set of web page(s) or a website can often be determined within a minimal page depth [4].

3.2 Breadth-First Search (BFS)

On the other hand, while BFS is less likely to encounter the above problems, it is likely incur impractical memory costs as the frontier may expand faster than dequeueing. The space complexity of these algorithms is influenced by the size of the queue or stack containing the unvisited vertices.

3.3 Iterative Deepening Depth-First Search (IDDFS)

IDDFS sets an incrementing depth level for each iteration of DFS, and the maximum depth level could either be the distance of the graph or a user-specified number.

For example, the n^{th} iteration of IDDFS is a DFS which treats all vertices which are n edges away from the seed as terminal vertices (regardless of whether the out-degree of these vertices) and there will no searching beyond these "terminal vertices".

Hence, IDDFS is a configurable program which can be configured to explore the hosts adequately rather than exhaustively while keeping memory usage in check.

However, the current IDDFS algorithm will lead to n visits to the vertices nearer to the seed for n iterations. This means that a data structure to track references to parent vertices is needed, and this may slow down the exploration.

3.4 Beam Search

Beam search is a graph search algorithm modified from BFS. The unvisited vertices are ranked and thus dequeued in the order of how close these vertices are in reaching the search target based on predefined heuristics and predictions.

Therefore, the effectiveness of beam search is mainly dependent on the accuracy of the heuristics in judging the vertices. The lower-ranked vertices will be pruned and there will no further exploration on these vertices.

Hence, there will always be a fixed (user-specified) number of unvisited vertices after every iteration, and this bounds the space complexity to just the maximum out-degree of any single vertex instead of an accumulation of all unvisited vertices.

3.5 Complexity Analysis

Time As the graph of the World Wide Web is unknown, both BFS and DFS will have a general runtime of O(E), as each vertex is explored once. E is the number of links explored. Checking if a particular vertex was visited is in O(1), as all incoming edges will be the same (disregarding short links), and these edges can be stored in a hash table.

For IDDFS, the general runtime would be O(E) for the same reasons above, but there will be additional overhead in maintaining a linked data structure which both BFS and DFS do not need, as they never needed to return to the seed vertex.

For beam search, the general runtime would be O($E\Delta GlogN$), where N is the (user-specified) size of the "frontier". ΔG is the maximum out-degree of a vertex in the graph. A fixed-size min-heap and max-heap with the same vertices are to be maintained.

The unvisited vertices will be added to the min-heap to track which vertices are to be removed from the max-heap. After any addition, the vertex at the root will be removed from the heaps. The max-heap will synchronise with the min-heap and produce a heap with the most "promising" vertex at the root. These heap operations result in the runtime above as a single heap operation costs O($logN$).

Space The average to worst space complexity of BFS is O(w), where w is the maximum *"width"* of the graph, i.e. maximum number of vertices of the same distance away from the start vertex [5]. For a tree, the *"width"* of a graph is:

$Width = (Branch\ Factor)^{Height}$

The above definition may be extended to fit a general undirected and unweighted graph.

On the other hand, the worst space complexity of DFS is O(h), where h is the *"height"* of the graph (maximum distance of the graph). And since the World Wide Web is more sparse than it is dense across hosts [6]:

$height < width \Rightarrow$ DFS has a lower space complexity.

For IDDFS, the space complexity increases with the search depth, and is thus O(h). However, there may be additional overhead in storing a linked structure compared to the other algorithms, which can use arrays.

For beam search, the worst space complexity is O(N), where N is the size of the heap.

3.6 Insights and Decisions

The analyses of existing graph exploration algorithms have raised much thought about the nature of the World Wide Web. They are:

1. Exhaustive crawling of a narrow scope of hosts sacrifices the diversity of the search under limited resources.
2. To convenience users, most information should be available in the shallower parts of the website.
3. Based on (2), websites can become uninformative in the deeper parts of the website.
4. DFS may be modified such that it does not traverse to vertices without any outgoing edges to unvisited vertices, but abandon search where appropriate, as can be seen in the example of IDDFS.

Based on the above insights, this paper will be using a modified DFS as the graph exploration solution in order to reduce the potentially massive space consumption and remain a diverse exploration. This modified DFS algorithm restricts the number of consecutive addition of edges from vertices belonging to the same host (Section 6.2.2 Modified DFS).

4 (Discussion) Solution Part II: Reinforcement Learning (RL)

4.1 Introduction

RL is a category of machine learning, where the software (the "agent") learns from its environment instead of an existing database. The general scenario consists of an agent in a specific state. This agent makes observations, performs actions, and is rewarded or punished when the outcome is reviewed [7]. The software aims to have its cumulative rewards converge towards a global maximum eventually, although the global maximum may fluctuate from time to time due to changes in the environment. Conversely, the software may aim to have its cumulative punishment converge towards a global minimum.

The following subsection will define the Markov decision process (MDP), a mathematical framework which models the components of decision making and outcomes [8] and the relevant formulae which will be used in this paper to conduct RL.

4.2 Markov Decision Process

The MDP is a tuple in the form of $(S, A, P_{sa}, \gamma, R)$ [8] where:

S* is a set of *states.

A* is a set of *actions. The action would be the act of either skipping a URL (denoted by *False*) or parsing the corresponding web page (denoted by *True*).

P_{sa} are the state transition probabilities. They are the probability distributions of S given some state and action. E.g. $P_{sa}(x)$ or $T(s, a, x)$ where $s, x \in S$ is the probability of $s \Rightarrow x$ with action a.

In this paper, the state transition probabilities will be derived via analysis and not modelling based on sample trials. There are 2^4 simplified states (True or False for each of the features).

R is the reward function. The function input is a state, s, and the function output is the reward for getting to that state. In this paper, the reward will be in the range [0, 1].

γ is the discount factor. The discount factor makes up the coefficients of the reward output given by the formula below:

$$\boldsymbol{\gamma^0 R(s_0, a_0) + \gamma^1 R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \gamma^3 R(s_3, a_3) + ...}$$

where $s_{x+1} = P_{s_x a_x}$ *(i.e. probability of 1)*

Since the discount factor is in the range $[0, 1)$, having higher powers of γ as the coefficients of the later rewards ensures:

1. Precedence of immediate rewards over delayed rewards [9] as the possibility of the agent terminating before such rewards are received should be taken into account.

2. Convergence of the reward output so that rewards for different sequences of actions remain comparable.

4.3 Solutions for Finite MDPs

The previous section defines the basic components in MDP and their exact formulations in the context of this paper. This section will discuss how MDPs can be solved and optimise decision making by maximising the final reward value through a specific sequence of actions.

Other Definitions

Policy (π) is a function which maps a state to an action (i.e. $a = \pi(s)$).

Value function (V^{π}) for the policy π is the expected sum of discounted rewards from a starting state $s \in S$. Recall $a = \pi(s)$. The value function is given by the formula below:

$$V^{\pi}(s) = E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...] \text{ where } s_0 \Rightarrow s_1 \Rightarrow s_2 \Rightarrow ...$$

State Utility/True Value ($U(s)$) is the immediate reward for the specified state, plus the expected discounted reward if the agent acted optimally from that point on. The state utility is independent from the policy π.

Policy Iteration Algorithm There are two classical algorithms which can solve finite-state MDPs. In this paper, the policy iteration algorithm will be implemented [10]. The policy iteration algorithm is as follows :

1. Create a random policy by selecting a random action for each state.
2. Repeat:
    (a) Compute the true value of each state based on the current policy (i.e. $V^{\pi}(s) = \sum P_{sa}(x)(R(s) + \gamma V^{\pi}(x))$ *where* $s, x \in S$).
    (b) Update the policy to fit the optimal state-action pair.
    (c) If the policy remains the same from the previous iteration, terminate and return the policy.

5 Proposed Solution

The proposed solution is a full search solution (from crawling to decision-making based on past parsing results) which aims to meet the following requirements:

1. Memory-efficient crawling of URLs.
2. Decide whether to parse a particular web page based on past experience.
3. Judge the relevance of content.

5.1 Web Crawling

Modified DFS

1. Push the seed(s) to a stack.
2. Repeat steps (3) to (9) until either:
   (a) Stack is empty, or
   (b) Maximum number (user-specified) of iterations is reached.
3. Pop a URL from the stack.
4. Parse the corresponding web page for topics.
5. For each topic:
   (a) Get the similarity score between the topic and each search word.
   (b) If the score is sufficient to be deemed as “relevant”, then record the tuple (score, search, topic, url).
6. Add this URL to a set of visited URLs.
7. If this domain has been crawled for the past 3 most recent iterations, then skip steps (8) and (9), and return to step (3). Otherwise, continue with step (8).
8. Parse the web page for URLs.
9. Push URLs that are not in the set of visited URLs.

5.2 Learning and Related Formulae

5.2.1 States

The proposed set of states S are defined as simplified representations of the real-valued features below:

1. $\boldsymbol{N_{pp}}$ $[0, \infty)$
   Total number of web pages belonging to the *previously processed* host name and *parsed.*

2. $\boldsymbol{N_{sp}}$ $[0, \infty)$
   Total number of web pages with the *same* host name and *parsed.*

3. $\boldsymbol{N_{ss}}$ $[0, \infty)$
   Total number of web pages with the *same* host name and *skipped.*

4. $\boldsymbol{R_{prev}}$ $[0, 1] \Rightarrow [0, \infty)$
   Cumulative *relevance* score for content belonging to the *previously processed* host name.

5. $\boldsymbol{R_{same}}$ $[0, 1] \Rightarrow [0, \infty)$
   Cumulative *relevance* score for content with the *same* host name.

6. $\boldsymbol{R_{url}}$ [0, 1]
   Relevance of the URL to *any* of the predefined search words, in the form of a list for the set of features, F, i.e. $[F_0, F_1, F_2, F_3...]$.

From the above information, the mean average for (4) and (5) can be derived readily. Note there can be infinite number of such states due to the infinite cardinality of real numbers. Hence, the above values will be processed to produce simplified representations so that there will be a finite number of states.

The mapping from specific ranges of states to their categorical representation, which will make up the *state*, is as follows:

1.
$$\textit{Host Parse-Skip Ratio} = \left\{ \begin{array}{ll} True, & \text{if } N_{sp} = 0 \\ & \text{or } (N_{ss} > 0 \text{ and } \frac{N_{sp}}{N_{ss}} < x) \\ False, & \text{otherwise} \end{array} \right\} \textit{where } x = \textit{1}$$

(*x can be any number greater than or equal to 1*)
This feature encourages the decision to parse a particular web page if:

(a) No web page belonging to the same host was parsed before
(b) **Or** web pages belonging to the same host name were mostly skipped.

This feature discourages the decision to parse a particular web page if:

(a) Web pages belonging to the same host were parsed before
(b) **And** there are more parsed web pages to skipped web pages for this host.

2.
$$\textit{Mean Avg. Relevance (Previous Host)} = \left\{ \begin{array}{ll} True, & \text{if } N_{pp} > 0 \text{ and } \frac{R_{prev}}{T_{prev}} \geq x \\ False, & \text{otherwise} \end{array} \right\} \textit{where } x = \frac{2}{3}$$

It is assumed that if the previous host was deemed relevant, then its outgoing links (one of which is the current host) are also relevant. As such, (2) is a feature of the state. This feature encourages the decision to parse a particular web page if:

(a) Web pages belonging to this previous host were parsed and deemed relevant

This feature discourages the decision to parse a particular web page if:

(a) Web pages belonging to the previous host were not parsed before
(b) **Or** the content of those web pages were parsed before but deemed as irrelevant.

3.
$$\textit{Mean Avg. Relevance (Current Host)} = \left\{ \begin{array}{ll} True, & \text{if } R_{same} = 0 \\ & \text{or } (N_{sp} > 0 \text{ and } \frac{R_{same}}{T_{same}} \geq x) \\ False, & \text{otherwise} \end{array} \right\} \textit{where } x = \frac{2}{3}$$

This feature encourages the decision to parse a particular web page if:

   (a) No web page belonging to the same host was parsed before
   (b) **Or** web pages belonging to the same host name were parsed and deemed relevant.

   This feature discourages the decision to parse a particular web page if:

   (a) Web pages belonging to the same host were parsed before
   (b) **And** the content of those web pages were deemed as irrelevant.

4.
$$\mathit{Predicted\ Relevance\ via\ URL} = \left\{ \begin{array}{ll} True, & \text{if } relevance \geq x \\ \\ False, & \text{otherwise} \end{array} \right\} \mathit{where}\ x = \frac{2}{3}$$

   ($x = \frac{2}{3}$ *was decided via observation; "relevance" is defined in Section 5.2.2 Relevance Score*)
   This feature encourages the decision to parse a particular web page if:

   (a) URL is adequately similar (and thus relevant) to *any* of the predefined search words

   . This feature discourages the decision to parse a particular web page if:

   (a) URL is inadequately similar (and thus irrelevant) to *all* of the predefined search words.

5.2.2 Relevance Score

Relevance scores in the range $[0, 1]$ may be calculated based on parsed web content and which can then be associated with a standard set of features in the corresponding URL. These scores play a part in deciding (in the future) if URLs with certain features should have their corresponding contents parsed.

This increases the efficiency of the web crawler while meeting the requirements for a search, especially under time and space constraints, as unnecessary cycles of request and parse are avoided for potentially irrelevant content.

For the set of N_K number of keywords in a web page, K, and set of N_S number of predefined search words, S, the overall relevance score has the following formula:

1. $relevance = \frac{1}{N_K} \sum_{i=1}^{N_K} max(similarity(K_i, S))$
2. Where $similarity(K_x, S) =$

   $$[\ similarity(K_x, S_0),\quad similarity(K_x, S_1),\quad similarity(K_x, S_2),\quad similarity(K_x, S_3),\quad ...\ ]$$

3. And

   $$similarity(K_x, S_y) = \left\{ \begin{array}{ll} 0, & \text{if } similarity(K_x, S_y) < z \\ similarity(K_x, S_y), & \text{otherwise} \end{array} \right\} where\ z = \frac{2}{3}$$

5.2.3 Reward Function

R(s) = Number of simplified features in the state which are True

5.2.4 Features of a URL

Some examples of URLs which could affect the decision of whether to parse the corresponding web page are [11]:

1. URLs which are way too long $\Rightarrow$ Deeply nested and irrelevant.
2. URLs which indicate non-textual file types $\Rightarrow$ gif, png, mp3, wmv, ...
3. URLs with search queries $\Rightarrow$ Potentially an advertisement URL.
4. URLs with gibberish host names $\Rightarrow$ Potentially a non-legitimate source.
5. URLs with gibberish relative links $\Rightarrow$ Potentially irrelevant.

5.3 Parsing

A corpus can be given a score by extracting its keywords and calculating their similarity to the user-specified search words. The following describes a keyword extraction algorithm, followed by a word/phrase similarity algorithm via the representation of words as vectors.

RAKE (Rapid Automatic Keyword Extraction)[12]

1. Download the text.
2. Remove punctuation and special characters.
3. Remove stop words.
4. The set of words separated by stop words are the candidate keywords or key phrases.
5. Find the degree of each candidate. The degree of a candidate is the number of times the candidate is used by other candidates.
6. Count the frequency of each candidate (with stemming).
7. Evaluate the candidate scores (for being a keyword or key phrase) via the formula $Score = \frac{Degree}{Frequency}$.

After RAKE, the following steps are proposed to measure relevance:

word2vec (Vector Representations of Words) [13]

1. Train a word2vec model.
2. Process the similarity scores between the keywords and the set of user-specified search words (phrases are permissible).

5.4 Effectiveness

The effectiveness of the proposed solution will be determined by the below criteria, which are numerical values related to the requirements in Section 1 Introduction:

1. Number of URLs parsed.
2. Number of web pages parsed.
3. Cumulative average relevance score of parsed content.

6 Proposed Implementation

6.1 Software Setup

1. Anaconda (`https://www.continuum.io/`) (Python 3.5)
2. Jupyter (`http://jupyter.org/`)
3. Requests (`http://docs.python-requests.org/en/master/`)
4. Beautiful Soup 4 (`https://www.crummy.com/software/BeautifulSoup`)
5. RAKE (`https://github.com/aneesha/RAKE`)

(1) and (2) can be downloaded from the corresponding websites, (3) and (4) can be installed via *pip install* or *conda install*, and (5) and (6) can be cloned into local machine from *GitHub*.

6.2 Code Snippets

6.2.1 BFS

From crawl.py

```
def bfsCrawlWrite(frontier, iterations):
    frontier = deque(frontier)
    relevant, threshold = [], 2/3
    indent1 = 0, '␣' * 2
    indent2, indent3 = indent1 * 2, indent1 * 3
    indent4 = indent2 * 2
    i = 0
    with open('similarity.json', 'w') as f:
        f.write('[\n%s[\n%s""' % (indent1, indent2))
        while frontier and i < iterations:
            f.write(',\n%s{\n' % (indent2))
            url = frontier.popleft()
            resp = requests.get(url)
            topics = parse.parseKeywords(resp)
            f.write('%s"":␣""' % (indent3))
            for topic in topics:
                relevant = False
                stringBuilder = []
                stringBuilder.append(',\n%s"%s":␣[\n' % (indent3, topic))
                for search in searches:
                    try:
                        score = similarity.getSim(search, topic)
                        if score >= threshold:
                            relevant = True
                            stringBuilder.append('%s"(%s,␣%s)",\n' % (indent4, \
                                search, score))
                    except:
                        continue
                if relevant:
                    stringBuilder[-1] = stringBuilder[-1].rstrip()[:-1]
                    stringBuilder.append('\n')
                    f.write(''.join(stringBuilder))
                    f.write('%s]' % (indent3))
            frontier.extend(deque(parse.parseLinks(resp)))
            f.write('\n%s}\n%s]' % (indent2, indent1))
            if frontier and i < iterations:
                f.write(',')
            else:
                break
            i += 1
        f.write('\n]')
```

6.2.2 Modified DFS

From crawl.py

```
def dfsCrawl(stack, iterations):
    stack = deque(stack)
    lowLim, uppLim = 3/5, 4/5
    depthMax, lastHost = 3, ""
    depthLeft = depthMax
    i, numVisited, urlVisited = 0, 0, set()
    while stack and i < iterations:
        url = stack.pop()
        urlVisited.add(url)
        resp = requests.get(url)
        topics = parse.parseKeywords(resp)
        for topic in topics:
            for search in searches:
                try:
                    score = similarity.getSim(search, topic)
                    if lowLim < score < highLim:
                        print(str((score, search, topic, url)))
                except:
                    continue
        host = parse.getDomain(url)
        if host == lastHost:
            if depthLeft:
                depthLeft -= 1
                stack.extend(deque(parse.parseLinks(resp)))
            else:
                depthLeft = depthMax
        else:
            depthLeft = depthMax
            stack.extend(deque(parse.parseLinks(resp) - urlVisited))
        numVisited += 1
        i += 1
    return (len(stack), numVisited)
```

6.2.3 RL

From reinforce.py

```
import mapping
from learn import MarkovAgent

agent = MarkovAgent(mapping.transitionRewardMap)
agent.learn()

policy = agent.policy

print(policy)
print(list(filter(lambda x: x, policy.values())))
```

6.2.4 Parser for URLs

From parse.py

```
def parseLinks(resp):
    domain = getDomain(resp.url)
    soup = BeautifulSoup(resp.text, "lxml", from_encoding = resp.encoding)
    edges = set()

    for link in soup.find_all('a', href = True):
        u = link['href']
        if u and u[0] not in ['#', '?']:
            if u[0] == '/':
                u = domain + u
            edges.add(u)

    return edges
```

6.2.5 Parser for Keywords

From parse.py

```
def parseKeywords(resp): return replaceSpace(parseRake(parseText(resp)))

# Adapted from Bochi, J. & Tugrul, K. (2016)
def parseText(resp):
    html = resp.text
    soup = BeautifulSoup(html, 'lxml')
    resultSetText = soup.findAll(text = True)
    tags = ['style', 'script', '[document]', 'head', 'title']

    def isVisible(doc):
        string = str(doc)
        return len(doc) >= 3 and \
            not (doc.parent.name in tags
                or re.match('<!--.*-->', string))

    def stripString(navigableString):
        stripped = navigableString.strip()
        encoded = stripped.encode('utf-8')
        string = str(encoded)
        finalString = ""
        for s in re.findall("'([^']*)'", string):
            if isShort(s):
                finalString = randomHash
            else:
                finalString += s
        return finalString

    def isReadable(string):
        return string != randomHash \
            and not any(special in string for special in specialString)

    def isShort(string): return len(string) < 3

    visible = filter(isVisible, resultSetText)
    randomHash = str(random.getrandbits(128))
    strippedStrings = map(stripString, visible)
    listText = filter(isReadable, strippedStrings)
    return ', '.join(listText)

def parseRake(string): return rakeInstance.run(string)

def replaceSpace(rakeOutput):
    return map(lambda w: w[0].split(' '), rakeOutput)
```

7 Experiments

7.1 Environment

The experiments were conducted on a personal computer with the following specifications:

1. 2nd generation processor
2. 4GB RAM
3. Windows 10 Education
4. Python 3.5
5. Geany IDE
6. Jupyter Notebook

7.2 Test Set

7.2.1 Seed URLs

The seed URLs were first crawled from `http://www.alexa.com/topsites/countries/US` as the parser can only process English. It is also desirable for seed URLs to have a considerable number of outgoing links.

For the experiments in this section, Stack Exchange was the seed for the web crawling algorithm, and Wikipedia was the seed for the parsing algorithm.

7.2.2 Search Words

This paper assumes that a web page with content related to popular search words is a relevant web page.

A list of popular search words (as declared by the source) were crawled from `http://www.pagetraffic.com/blog/most-popular-keywords-on-search-engines`.

7.3 Results

7.3.1 Runtime Analysis for Modified DFS

From memory.py

```
Iterations: 5

Size of Queue: 1205
Number of Visited: 5
Time Taken: 65.640972

Size of Stack: 923
Number of Visited: 5
Time Taken: 47.394564

Iterations: 50

Size of Queue: 9522
Number of Visited: 50
Time Taken: 535.173888

Size of Stack: 6096
Number of Visited: 50
Time Taken: 411.113517

Iterations: 500

Size of Queue: 68615
Number of Visited: 500
Time Taken: 4459.348122

Size of Stack: 26864
Number of Visited: 500
Time Taken: 1881.018762
```

The modified DFS has a consistently lower time and space complexity than BFS, as the modified DFS does not always push to the stack, as explained in Section 3.6 Insights and Decisions and implemented in Section 6.2.2 Modified DFS. Measurements should be read as approximate and relative indications rather than absolute values as the computer may have varying loads from other processes throughout the execution.

Additional Note The BFS in Section 6.2.1 is different from the BFS used to compare against DFS here. The BFS used for comparison can be found in Lines 69 to 90 of *crawl.py*.

7.3.2 RL

The states, actions, and rewards defined in Section 5.2 Learning and Other Formulae were passed into Epstein's library for reinforcement learning. The policy that was computed encourages the agent to do the "False" action in every state. This means the crawler will always never parse any web page.

From reinforce.py

```
{
    '(False,␣True,␣True,␣False)': False,
    '(True,␣False,␣False,␣True)': False,
    '(False,␣True,␣True,␣True)': False,
    '(True,␣False,␣True,␣False)': False,
    '(False,␣True,␣False,␣True)': False,
    '(True,␣True,␣True,␣False)': False,
    '(False,␣False,␣True,␣True)': False,
    '(True,␣True,␣True,␣True)': False,
    '(False,␣False,␣False,␣True)': False,
    '(False,␣False,␣False,␣False)': False,
    '(False,␣False,␣True,␣False)': False,
    '(False,␣True,␣False,␣False)': False,
    '(True,␣True,␣False,␣False)': False,
    '(True,␣True,␣False,␣True)': False,
    '(True,␣False,␣True,␣True)': False,
    '(True,␣False,␣False,␣False)': False
}
```

Increasing the penalty (by decreasing the reward) of choosing the "False" action and increasing γ to 1 did not lead to any changes in the policy.

7.3.3 Output from Section 6.2.1

The outermost keys are keywords found in the web page, and the inner keys are the popular search words which are deemed similar to the keywords in the web page; the values are the similarity scores.

From SampleSimilarity.json

```
[
  [
    "",
    {
      "": "",
      "['william', 'jennings', 'bryan']": [
        "(['jennifer', 'aniston'], 0.715375311301)",
        "(['jessica', 'simpson'], 0.707228508771)",
        "(['emma', 'watson'], 0.75140799662)",
        "(['justin', 'bieber'], 0.688868523925)",
        "(['lisa', 'ann'], 0.703440565176)",
        "(['brooke', 'burke'], 0.759845816284)",
        "(['casey', 'anthony'], 0.778917461542)",
        "(['amanda', 'knox'], 0.731735006866)",
        "(['alexis', 'texas'], 0.71553821859)",
        "(['jenny', 'mccarthy'], 0.709436693434)",
        "(['pamela', 'anderson'], 0.730096377937)",
        "(['jennifer', 'walcott'], 0.765803294461)",
        "(['sarah', 'palin'], 0.705588316234)"
      ],
      "['free', 'travel', 'guide']": [
        "(['travel'], 0.701035129669)"
      ],
      "['free', 'textbooks']": [
        "(['used', 'textbooks'], 0.777232581809)"
      ],
      "['free', 'silver']": [
        "(['free'], 0.674772488876)"
      ],
      "['car', 'crash']": [
        "(['car'], 0.75632134745)"
      ],
      "['classical', 'music']": [
        "(['music'], 0.866416812575)"
      ],
...
```

8 Analysis

8.1 Graph Exploration/Web Crawling

In Section 7.3.1 Runtime Analysis for Modified DFS, modified DFS has shown to significantly reduce the execution time and memory usage for the same number of iterations and URLs visited, compared to BFS.

This comparison may also be extended to the original DFS, as the original DFS will always push to the stack and increase memory usage. Pushing to the stack also costs a significant amount of time, because every URL has to be checked against the set of visited URLs to prevent duplicate requests.

8.2 RL

While the states and actions were well-defined in Section 5.2 Learning and Related Formulae, the actual state transition probabilities are unknown and can have a large variance due to the diverse nature of the World Wide Web. In addition, the generalisation of states might not have expressed the actual state well and might have led to a less meaningful representation instead.

Furthermore, the value function is dependent on a sequence of actions. This increases the weight of the seed URL, thereby rewarding more on the choice of seed URL, and on the immediate discoveries after the seed URL, than on a per-relevant-discovery basis. In Section 7.3.2, γ was increased to 1 so that all rewards at different points of time have the same weight. However, the policy remained the same.

8.3 Parsing

RAKE has been successful in extracting important words, although some noise, such as titles, footers, and other extra content remained.

However, such filtering and stemming may not be sufficient when these keywords may become too general and matching them to their specific counterparts in the search words list may become less meaningful.

In a search, users strive to be more specific so that the search engine will yield specific results. As such, a keyword should only be deemed as relevant if it has a greater than or equal level of specificity compared to the search words.

On the other hand, in the word2vec model, keywords such as "music" still had a high similarity to the search words "classical music" (see Section 7.3.3 Output from Section 6.2.1). This could be due to word2vec measuring the absolute distance between the vector representations of the words.

Moreover, the threshold of $\frac{2}{3}$ has no upper bound. Exact words will have the maximum similarity score of 1. This means that the threshold is encouraging an overfit of similarity, since exact words are much more likely to be deemed as relevant, compared to words which are different in form but related in meaning. Hence, to prevent such bias, Section 6.2.2 Modified DFS implements arbitrary bounds on the threshold (i.e. $\frac{3}{5} < \textit{relevant} < \frac{4}{5}$).

9 Future Work

Despite unfavourable results, the proposed solutions and implementations in this paper have led to other new possibilities in solving the web crawling problem.

9.1 Refining RL

The proposed usage of RL in this paper may not be suitable due to the abundance of unknown variables. However, a model-free and unbiased RL technique (Q-learning) may have the potential to enhance this decision-making process as it allows infinite state spaces and unknown state transition probabilities.

9.2 Learning from something else (URL)

The URL is the most easily attainable information about a web page without visiting it. Section 5.2.4 Features of a URL could be expanded to further influence the decision of whether to parse a particular web page.

Moreover, since the URLs are the edges in the graph of the World Wide Web, the strength of the classification of these URLs could be the heuristic in beam search, discussed in Section 3.4 Beam Search. This means that the more relevant content are likely to be parsed first, assuming relevant URL $\Rightarrow$ relevant content.

9.3 Learning from others

Insights from expert knowledge or other web crawlers may be added to the web crawler's knowledge base and help the crawler make informed choices. Preprocessing decisions based on existing data may also lead to more efficient runtime as there will be less online learning. Some of such insights can be in the form of whitelists, blacklists, or other naming conventions in URL or page titles.

9.4 Refining Parser

The current parser parses for all HTML text, and this may lead to noise in the content, such as titles, footers, and other extra content. The parser could be more efficient by avoiding information enclosed in certain HTML tags, such as <title>, <footer>, and so on.

9.5 Parallelism

It is observed that decision making and parsing can be separate processes, as the former writes to a data structure containing to a list of URLs to have their content parsed, and the latter can read from this list to get the next web page to parse. Separating such processes and parallelising them will speed up the execution significantly due to less frequent context switching. This allows the web crawler to take on a larger load.

10 Conclusion

There are numerous components to consider when creating or optimising a web crawler. This paper has only discussed the major components of the crawling process, while proposing an improvement by deciding whether some web pages should be downloaded and parsed, without having parsed them before. Beyond crawling, other components which should be considered include storage of parsed material, results, and retrieval (i.e. indexing).

References

[1] Wikipedia, *Distributed Web Crawling - Wikipedia, the free encyclopedia*, 2016, May 29.

[2] Deo N. & Gupta P., *World Wide Web: A Graph-Theoretic Perspective*, 2016, July 2.

[3] Pandit, D., *BFS or DFS for a Web Crawler? - Stack Overflow*, 2013, February 18.

[4] Kulkarni, M., & Patil, S., *Which Traversal is Better for Web Crawling? DFS or BFS, and Why? - Quora*, March 26.

[5] Andersson G., *Why does BFS use more memory than DFS? - Stack Overflow*, 2014, May 5.

[6] Kumar et al., Section 3.4, *The Web as a Graph*, 2000.

[7] Blum A., *Reinforcement Learning and Markov Decision Processes (MDPs)*, 2014, March 27.

[8] Ng A., Part XIII, *Reinforcement Learning and Control*, 2016, August.

[9] Murphy K., *A brief introduction to reinforcement learning*, 1998.

[10] Frazzoli E., Lecture 23: Markov Decision Processes - Policy Iteration *Principles of Autonomy and Decision Making*, 2010, December 1.

[11] Kan M., Nguyen T., *Fast web page classification using URL features* , 2005.

[12] Atli A., *What are the best keyword extraction algorithms? - Quora*, 2015, January 31.

[13] Mikolov et al., *Distributed Representations of Words and Phrases and their Compositionality*, 2013.