Comparison of Graph Traversal Algorithms in Web Crawling

Anwesa Basu, Lakshmi Posni, Swapnendu Majumdar, Zakar Handricken

(CS5800 Final Project Fall 2022 Semester)

https://www.youtube.com/watch?v=jpqgqRooPTg
CS5800 Final Project Presentation.pptx
https://github.com/SwapnenduM/crawlingAlgorithms

Introduction

In this paper, we present our results on the comparative analysis of graph traversal algorithms and their performance in web crawlers under different scenarios. To do our analysis, we implemented breadth and depth-focused graph traversal algorithms such as Breadth First Search (BFS), Depth First Search (DFS), Depth Limited Search (DLS), Iterative Deepening Search (IDS), and Greedy Search. With that, we aimed to answer the following questions:

- What algorithms are suitable for implementation in web crawlers?
- What are the drawbacks and benefits of these algorithms in web crawlers?
- Under different scenarios, which of these algorithms is most performant?
- Can we modify these algorithms to improve their use in web crawlers?

Background Context

Anwesa Basu: First I want to explain what web crawlers are and then I will discuss why I chose this topic by elaborating on the importance of web crawlers. Lastly, I will describe the scope of implementing web crawlers in our project with the help of several graph traversal algorithms. So, a web crawler is a computer program that is used to search and index the content of websites available over the internet. Mainly they are operated by a search engine. Normally search engines will have their algorithms to search relevant content from the data collected by the web crawlers. Web crawlers play a very important role to pull a piece of targeted specific information from various websites. Instead of a manual web search which could be more time-consuming and erroneous, it is always wiser to use an automated tool such as a custom crawler which can save manpower and can bring out the most relevant content in less time. Web crawlers can also help identify navigational errors and blocks present in a website. It also helps businesses to take important decisions to stay relevant in a competitive market by gathering data. We can implement web crawlers by implementing several graph traversal algorithms such as BFS, DFS, etc. With the help of this project, I can explore several graph traversal algorithms and can compare their performances in terms of efficiency. I can learn their use cases based on real-life hands-on project scenarios. I will also be able to understand their advantages and drawbacks etc. This research may help me unleash some new findings or observations about these concepts which I might not have been able to gather otherwise from a theoretical perspective.

Lakshmi Posni: A web crawler is used to search and automatically index website contents and other information. For example, Google has a web crawler called Googlebot that specifically mobile and desktop crawling, but they also have other crawlers like Googlebot images and Googlebot videos. The main purpose of these bots is to learn about different web pages on the internet, they specifically work towards analyzing this data. Web crawlers are very important applications of graph algorithms like BFS and DFS. The idea of these web crawlers and how they are related to graphs is that the internet can be repented by a directed graph. So the vertices of a graphs can be things like domains, URLs, and websites and the edges of a graph can be things like connections. So how it usually works is by parsing the raw HTML of a website and look for other URL, and if there is a URL then add it to a queue or a stack depending which graph algorithm is being used. So the topic we specifically chose is to test out which Algorithm is the most efficient algorithm for web crawlers. This is quite important to learn and experiment on, because web crawlers are very important when it comes to analyzing data on the internet, and this type of data is quite important and there's usually a lot of it. So knowing the most efficient algorithm for web crawlers can help in the future efficiency of these crawlers.

Swapnendu Majumdar: Web crawling, as the name suggests, involves the tedious task of crawling through web pages and indexing individual links encountered in order to help track down specific links as and when required, and thereby help optimizing search engines. Seems fairly basic to want to utilize an automation tool for the task mentioned above and thereby help reduce human effort in a task which can be considered somewhat redundant as well. The tools developed for said purpose are known as web crawlers, and fairly popular today. However, as the sheer volume of content available on the internet keeps on increasing, we must lay emphasis to the efficiency of the algorithms which can be used by these tools, since any improvement in performance might help improve the efficiency of the algorithm, and thereby the tool. In this project, we will try to perform a comparison between some of the common algorithms available today which lie within the purview of our course, and try to improve our understanding and knowledge of these algorithms by analyzing the same. Some of the common algorithms which I personally am excited about, and hope to learn more about through this project are - Breadth First Search, Depth First Search, Iterative Deepening Search, Uniform Cost Search and A* Search algorithms.

Zakar Handricken: Web crawlers are important because they index links to websites on the World Wide Web (WWW) across the Internet. Understandably, one representation of the WWW is a directed graph with nodes as websites and edges as urls. The size of the WWW is vast compared to its early days on ARPANET so it is important to implement efficient graph traversal algorithms in web crawlers that work at scale. That importance motivates me to implement such algorithms and analyze their performance in web crawlers. Note that we will not implement graph traversal algorithms comparable to those used in web crawlers by companies like Google. That said, it is an exciting opportunity to explore this subject because graph traversal algorithms are applicable beyond web crawlers, but anywhere there are graphs to be traversed.

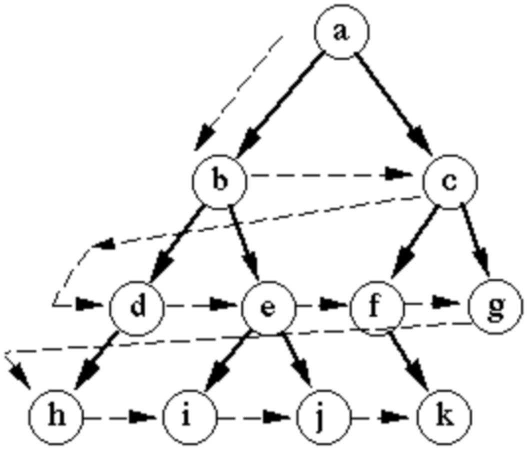
Analysis

Graph Traversal Algorithms

BFS and Web Crawler

The Breadth First Search or BFS algorithm is used to search a tree or a graph data structure for a node that meets a given criteria. It usually begins at the root of the tree or graph and looks at all the nodes at the current depth level before moving on to nodes at the next depth level. Many problems can be solved using breadth first search, such as finding the shortest path, implementation of web crawlers etc.

Path of Traversal in Breadth First Search:



BFS Advantages and Disadvantages

A few advantages of the Breadth First Search algorithm are:

- There always has to be a solution in BFS.
- A BFS approach will never get trapped by visiting unwanted nodes..
- The BFS approach also finds the shortest goal in the least amount of time.

A few disadvantaged of the Breadth First Search algorithm are:

- There can be memory constraint problems, since it has to store all the nodes in the current level before moving on to the next level.
- If the solution is far away in the data structure then it can consume more time to find the solution.

Time Complexity

O(V+E), V is for Vertices and E is for Edges.

Implementation Description

In our implementation, we have leveraged BFS to implement a web crawler. We have passed a seed Url as our root node and have performed BFS by traversing all its child URLs level wise. We have also passed a target URL to check if it can be found or not. We provided a maximum depth up to which the algorithm will perform its search. For this implementation we have used data structures such as HashMap, Queue etc. We have also printed the memory usage and total running time of this entire operation.

Code

For implementation please refer to Breadth First Search in Appendix: Code

Output

```
............................................................
Current Depth: 0 / 2
Queued Urls #: 1
1. https://basuanw.sites.northeastern.edu/
............................................................
Url to crawl: https://basuanw.sites.northeastern.edu/
............................................................
Current Depth: 1 / 2
Queued Urls #: 8
1. https://basuanw.sites.northeastern.edu
…
8. https://brand.northeastern.edu
............................................................
Url to crawl: https://basuanw.sites.northeastern.edu
............................................................
…
............................................................
Url to crawl: https://brand.northeastern.edu
............................................................
Total Urls Queued #: 33
 1. https://www.linkedin.com
…
33. https://bbpress.org
............................................................
Algorithm: BFS - Max Depth: 2 - Time: 10135 ms - Url Found: true
```

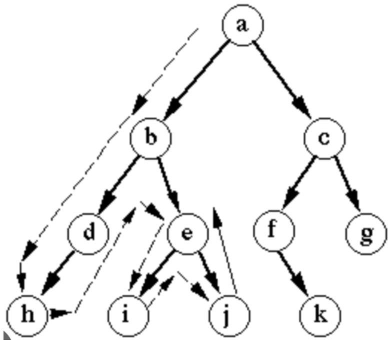
Drawbacks

If an URL resides in a more depth then it will take more time for BFS, which can be solved by DFS.

DFS and Web Crawler

The Depth First Search or DFS algorithm is also a searching algorithm to search a graph or a tree data structure. The DFS algorithm usually starts at the root node and goes as deep down as it can on the given branch. Then it backtracks until it lands on a path that has not been explored previously, and then continues to explore that path. This algorithm continues until the entire graph or tree data structure has been explored. Depth First Search is usually used in analyzing problems such as mapping routes, scheduling, finding spanning trees, and just like BFS it is used to find the shortest path and in the Ford-Fulkerson algorithm

Example of How DFS Searches:



DFS Advantages and Disadvantages

A few advantages of the Depth First Search algorithm are:

- The DFS approach is advantageous when finding the element that is farther away from the root node
- It requires less memory since it is linear with respect to nodes

Time Complexity

O(V+E), V is for Verticis and E is for Edges.

Implementation Description

In this scenario we have maintained the main operations same as BFS, but only changed the algorithm from BFS to DFS. We have used a Stack data Structure to implement the same.

Code

For implementation please refer to Depth First Search in Appendix: Code

Output

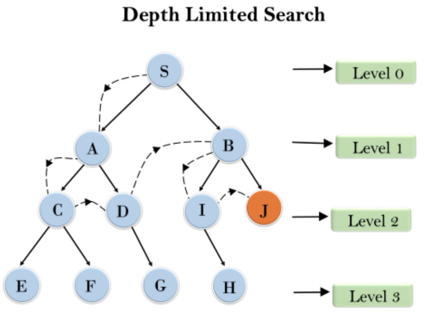
```
Total urls stacked: 9
.....................................................................
...........
Total urls stacked at Iteration 0 #: 1
1. https://basuanw.sites.northeastern.edu/
.....................................................................
...........
Total urls stacked at Iteration 1 #: 8
1. https://brand.northeastern.edu
…
8. https://api.w.org
.....................................................................
...........
Total urls stacked at Iteration 2 #: 0
............................................................
Algorithm: DFS - Max Depth: 2 - Time: 585 ms - Url Found: true
```

Drawbacks

It will perform badly if the target URLS resides in the upper level but in the furthest branch and also it may go to infinite loop if no depth limit is mentioned. It can also not find a target URL as well.

DLS and Web Crawler

Depth Limited Search is an uninformed search algorithm, similar to the Depth First Search Algorithm(DFS). Depth Limited Search is very similar to DFS, but unlike DFS it has a predetermined depth limit. Nodes that are at this depth limit are usually nodes with no children (they are leaf nodes). So just like DFS, DLS also starts at the root node and follows each branch to its deepest node. But the problem with DFS is that it could lead to an infinite loop. By including a specific depth, like the depth limit, the DLS algorithm could eliminate the issue of DFS getting stuck in an infinite loop.



For this graph the depth limit = 2

DLS Advantages and Disadvantages

A few advantages of the Depth Limited Search algorithm are:

- DLS is more efficient than DFS, it uses less memory and time
- If a solution exists to a problem then DLS guarantees that it will be found in a finite amount of time
- DLS has applications in graph theory that are highly comparable to DFS

A few disadvantaged of the Depth Limited Search algorithm are:

- For DLS to work there must be a depth limit
- If the goal node does not exist within the specified depth, then it will not be discovered

Time Complexity

O(b^l), b is known as the branching factor (number of children at each node and l is the given depth limit.

Implementation Description:

We have previously implemented DFS. But we found a drawback. In order to rectify that we have implemented DLS where we have added a maximum depth up to which the operation will continue. For this implementation we have kept the DFS operation the same, but with each URL we have associated a level with them by storing them in a HashMap as a key and value pair. By doing this, we can easily find the level of each URL and can make sure that the traversal is limited to only a certain maximum level and if somehow that exceeds in one branch then it can backtrack other nodes of lower levels in a different branch.

Code

For implementation please refer to Depth Limited Search in Appendix: Code

Output

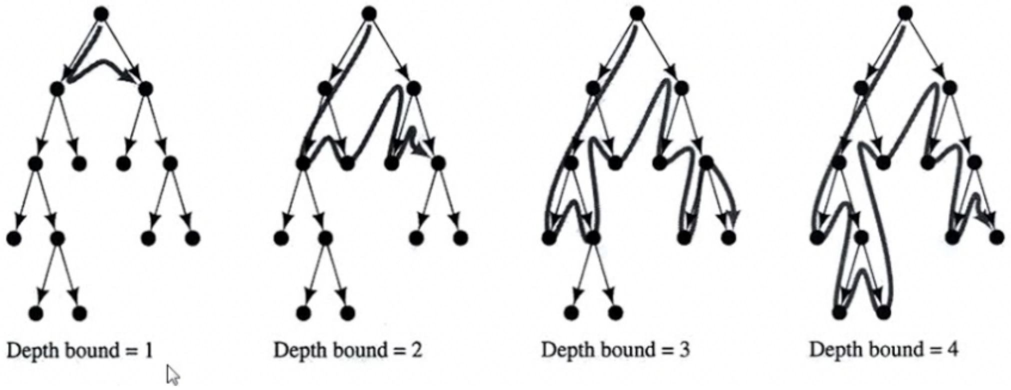
```
............................................................
Url to crawl: https://basuanw.sites.northeastern.edu/
 - Fetched Url: https://basuanw.sites.northeastern.edu
…
 - Fetched Url: https://brand.northeastern.edu
............................................................
Url to crawl: https://brand.northeastern.edu
............................................................
Total Urls Stacked #: 9
1. https://brand.northeastern.edu
…
9. https://api.w.org
............................................................
Algorithm: DLS - Max Depth: 2 - Time: 172 ms - Url Found: true
```

Drawbacks

It takes more time as If the target node does not exist inside the chosen depth limit, the algorithm will be forced to iterate again, increasing the execution time.

IDS and Web Crawler

Iterative Deepening Search is an iterative graph searching strategy. This search takes advantage of the completeness of the Breadth First Search strategy, and at the same time uses much less memory in each iteration like the Depth First Search approach. This desired completeness is achieved by enforcing a depth limit on DFS, which eliminates the possibility of getting stuck in an infinite search. IDS searches each branch of a node from left to right until the required depth is reached. Once it reaches the required depth, then IDS goes back to the root node and explores a different branch similar to DFS. Although IDS repeatedly goes over the same node again and again the cost is still kept to a bare minimum, because most nodes in a tree are at the bottom levels which are only visited once or twice and the upper-level nodes do not make up the majority of the nodes in the tree. A good time to use IDS is when the depth is unknown and the problem has a requirement to search a very large space. It can also be used as a slower substitute to BFS if there is a memory or space constraint.



IDS Advantages and Disadvantages

A few advantages of the Iterative Deepening Search algorithm are:

- If the solution exists then IDS can find it
- If the solutions are found at lesser depth then the algorithm is a lot more time-efficient
- Space-efficient search such as DFS can be used at each phase

A few disadvantaged of the Iterative Deepening Search algorithm are:

- Time it takes to reach the goal node is exponential
- If the BFS algorithm ever fails on a problem then the IDS algorithm would fail too

Time Complexity

O(b^k), b is the branching factor and k is where the first solution is at depth k.

Implementation Description

For this we have called the DLS method in each iteration and have passed that iteration value(using a while loop) as a maximum level to DLS.

Code

For implementation please refer to Iterative Depth Search in Appendix: Code

Output

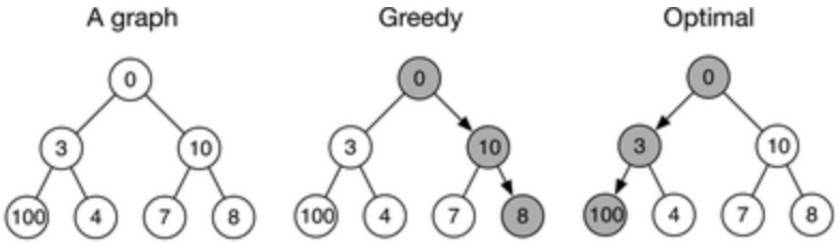
```
............................................................
Total Urls Stacked #: 1
1. https://basuanw.sites.northeastern.edu/
............................................................
Url to crawl: https://basuanw.sites.northeastern.edu/
 - Fetched Url: https://basuanw.sites.northeastern.edu
…
 - Fetched Url: https://brand.northeastern.edu
............................................................
Total Urls Stacked #: 9
1. https://brand.northeastern.edu
…
9. https://api.w.org
............................................................
Algorithm: IDS - Max Depth: 2 - Time: 85 ms - Url Found: true
```

Drawbacks

There is time and calculations wasted at each depth.

Greedy and Web Crawler

A Greedy algorithm is an approach to solving a problem by selecting the best option available at that moment, this option does not worry about whether the current result is the overall optimal result. So the algorithm never reverses an earlier option even if it is the wrong option. The greedy algorithm works in a top-down approach. A problem can be solved using the greedy approach if the optimal solution for the problem can be found by choosing the best option at each step without reconsidering any previous steps once the solution has been chosen and also if the overall solution corresponds to the optimal solutions of the subproblems.



Greedy Advantages and Disadvantages

A few advantages of the Greedy algorithm are:

- Greedy approach is easy to implement
- Greedy typically has less time complexity
- Performs better than other algorithms (not in all cases)

A few disadvantaged of the Greedy algorithm are:

- The local optimal solution may not always be globally optimal

Time Complexity

O(n log n)

Implementation Description

In this case we have chosen a greedy approach based on a heuristic value. Our heuristic was determined by calculating the loading time of each website and by choosing the least among them and continue like that. So, our greedy algorithm will first prioritise the URL with the least loading time and then will traverse to the second URL with the second least loading time and so on. For this implementation we have written another separate function to calculate our heuristic and then another function for the traversal. In this function we have used BFS but with a priority queue where we have added our priority as the heuristic value.

Code

For implementation please refer to Greedy Search in Appendix: Code

Output

```
Calculating heuristic
heuristic calculation completed in 0 ms
............................................................
Url to crawl: https://basuanw.sites.northeastern.edu/
............................................................
Queued Urls at depth 1 #: 8
1. https://basuanw.sites.northeastern.edu
…
8. https://fonts.gstatic.com
............................................................
Url to crawl: https://basuanw.sites.northeastern.edu
............................................................
Url to crawl: https://brand.northeastern.edu
............................................................
Total Urls Queued #: 9
1. https://brand.northeastern.edu
…
9. https://api.w.org
............................................................
Algorithm: Greedy - Max Depth: 2 - Time: 746 ms - Url Found: true
```

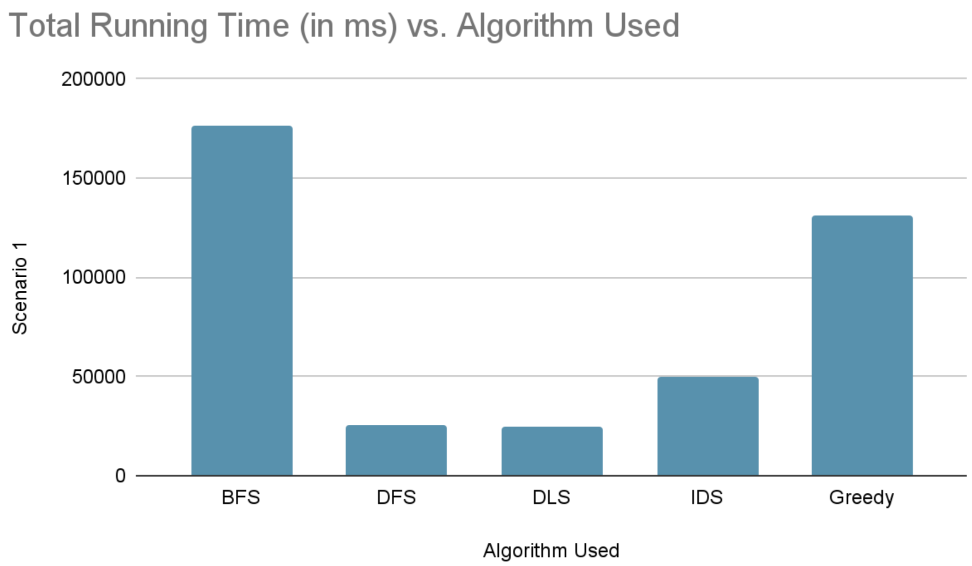
Drawbacks

Greedy may or may not always give the optimal results.

Observations

Scenario 1

Scenario 1	**Algorithm**	**Time (ms)**	**URLs Crawled**	**Found Target URL**
The most efficient Algorithm when the target URL is at a deeper-level of the graph	**BFS**	176826	1100	yes
	DFS	25144	294	yes
	DLS	24990	367	yes
	IDS	49853	363	yes
	Greedy	131572	950	yes
Which is the most efficient?	**DLS**			

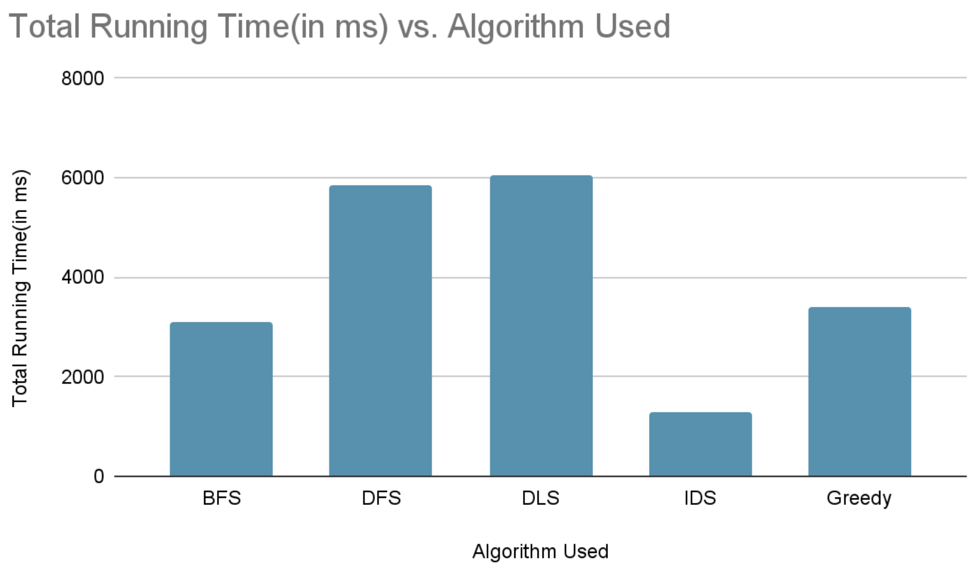


Observations

1. IDS may take more time than DLS if the depth of the goal node is deeper as there is another .
2. BFS will take the most time in this scenario as it performs level order traversal.
3. DLS will perform the best in this scenario as it will search depth wise and will find the goal url faster as it resides in deeper level.

Scenario 2

Scenario 2	Algorithm	Time (ms)	URLs Crawled	Found Target URL
The most efficient Algorithm when the target URL is in the upper-level of the graph	**BFS**	3090	33	Yes
	DFS	5857	66	yes
	DLS	6036	81	yes
	IDS	1286	9	yes
	Greedy	3393	15	yes
Which is the most efficient?	**IDS**			

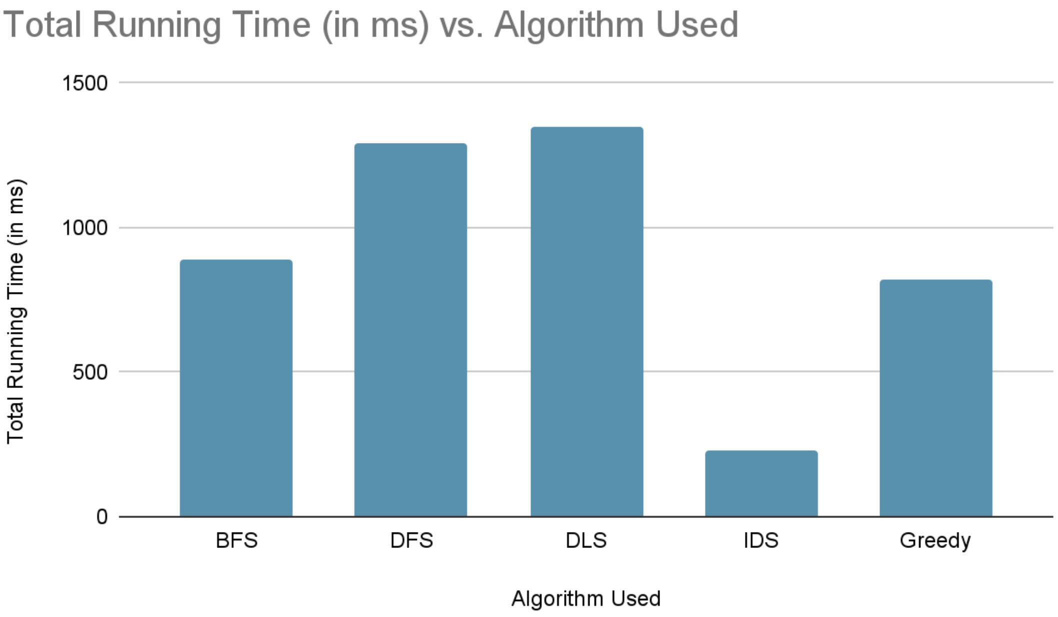


Observations

1. BFS will perform better than DFS or DLS as the target url can be found in the upper level itself without going deep and as BFS performs level wise traversal it will find it earlier.
2. IDS performs the best in this scenario.
3. Greedy may or may not always give the optimal result.

Scenario 3

Question	Algorithm	Time (ms)	URLs Crawled	Found Target URL
The most efficient Algorithm where the targetURL is in upper level and on a sparse tree(Target node lies on later branches)	**BFS**	890	19	yes
	DFS	1289	83	no
	DLS	1346	87	yes
	IDS	230	9	yes
	Greedy	820	12	yes
Which is the most efficient?	**IDS**			



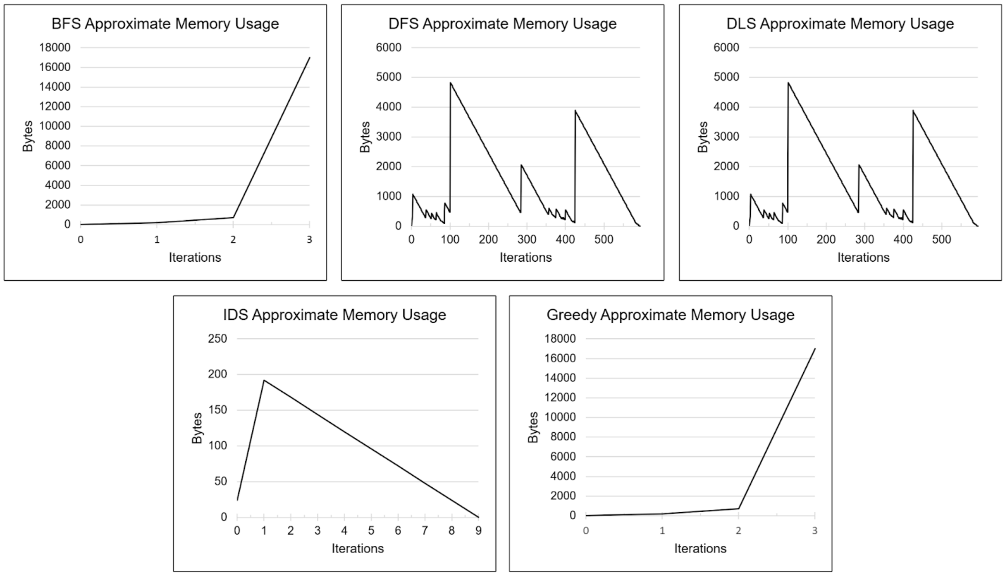
Observations

1. In this case BFS performs much better than DLS as it works best in the cases where the target URL will reside on the upper level but on a furthest branch.
2. Greedy performs really well in this scenario and can be utilised.
3. IDS performs the best.
4. DFS performs the worst and the target URL could not be found.

Scenario 4

Question	Algorithm	Max Depth	Time (ms)	URLs Crawled
Which is the most efficient algorithm based on approximate memory usage?	**BFS**	3	16258	748
	DFS	3	17808	562
	DLS	3	19839	562
	IDS	3	14723	562
	Greedy*	2	157793	748*
Which is the most efficient?	**IDS, in the worst-case, when exploring the entire searchable space.**			

*Greedy Search was run at a max depth of 2 due to time-constraints. Since it uses BFS an assumption is made that it will visit the same number of URLs and have similar memory usage.

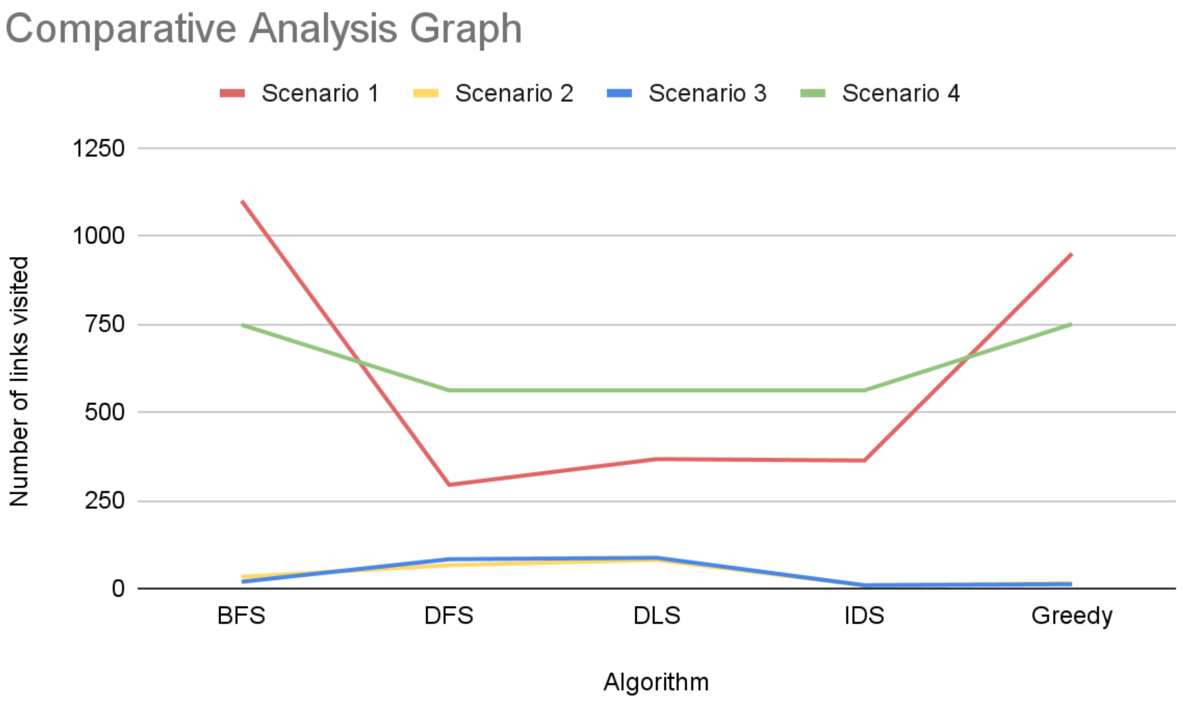


Scenario 4: Approximate Memory Usage per Algorithm Overtime

Observations

1. Depth-focused algorithms outperform breadth-focused algorithms in memory usage.
2. Memory usage by algorithms DFS, DLS, and IDS is stable and has a shark fin pattern.
3. Memory usage by algorithms BFS and Greedy appears to be exponential.
4. A solution that solves the issue of exceeding memory usage for both types of algorithms is offloading results into a database on disk and accessing them as needed.

Comparative Analysis Graph



Conclusion

Anwesa Basu:-
We have implemented several graph traversal algorithms to create an efficient web crawler. We have seen the drawbacks and advantages of those algorithms and have tried to find a better one by rectifying those drawbacks. We have performed a comparative analysis based on their run time performances and memory optimisation. We have tried to find the most efficient algorithm based on a particular scenario.From this analysis we can conclude that if we have to perform web crawling and our target url resides in the upper level, then we can use BFS or IDS as they can give us the most efficient results. Similarly, for any other scenarios where we have to find the target URL in a more deeper level, then DLS or IDS can give us an efficient solution. In terms of memory usage also IDS performs the best for this cases. But, since in most of the real life scenarios Web crawler performs crawling or web scrapping mostly in upper levels, so in those cases BFS/ IDS outperforms others.

Lakshmi Posni: So as said in my introduction web crawlers are something that are very useful in the wide web. It stores all the relevant data that the user requires based on their search. So for a crawler the search algorithm that is being used is quite important. And from our analysis it can be seen that based on the type of scenario that is being presented to the crawler the search algorithm can alter by providing a better time complexity than others. From the different scenarios presented it is noticeable that if the target URL is at a deeper level then DFS would perform better than the other algorithms, but for the other scenarios such as memory usage and if the URL is in the upper-level then IDS performs better.

Swapnendu Majumdar: While we observed and implemented the different algorithms to traverse through the URLs, we noticed different performance outcomes based on different scenarios in which some algorithms outdid others in one case, but fell behind on the others in other cases. These observations were fascinating and helped us understand each of the said algorithms slightly better.

Zakar Handricken: Through this paper we implemented a number of different graph traversal algorithms as mentioned above. We observed the disadvantages and advantages of breadth and depth focused algorithms and their impact on web crawlers in terms of time and memory performance. We identified algorithms that outperform others and also considered ways to improve both types of algorithms when it comes to memory usage.

References

- https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/#:~:text=Time%20complexity%20You%20have%202,O(N%20*%20logN).
- https://www.keycdn.com/blog/web-crawlers
- https://en.wikipedia.org/wiki/Web_crawler
- https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/
- https://tutorialedge.net/artificial-intelligence/depth-limited-search-in-java/
- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

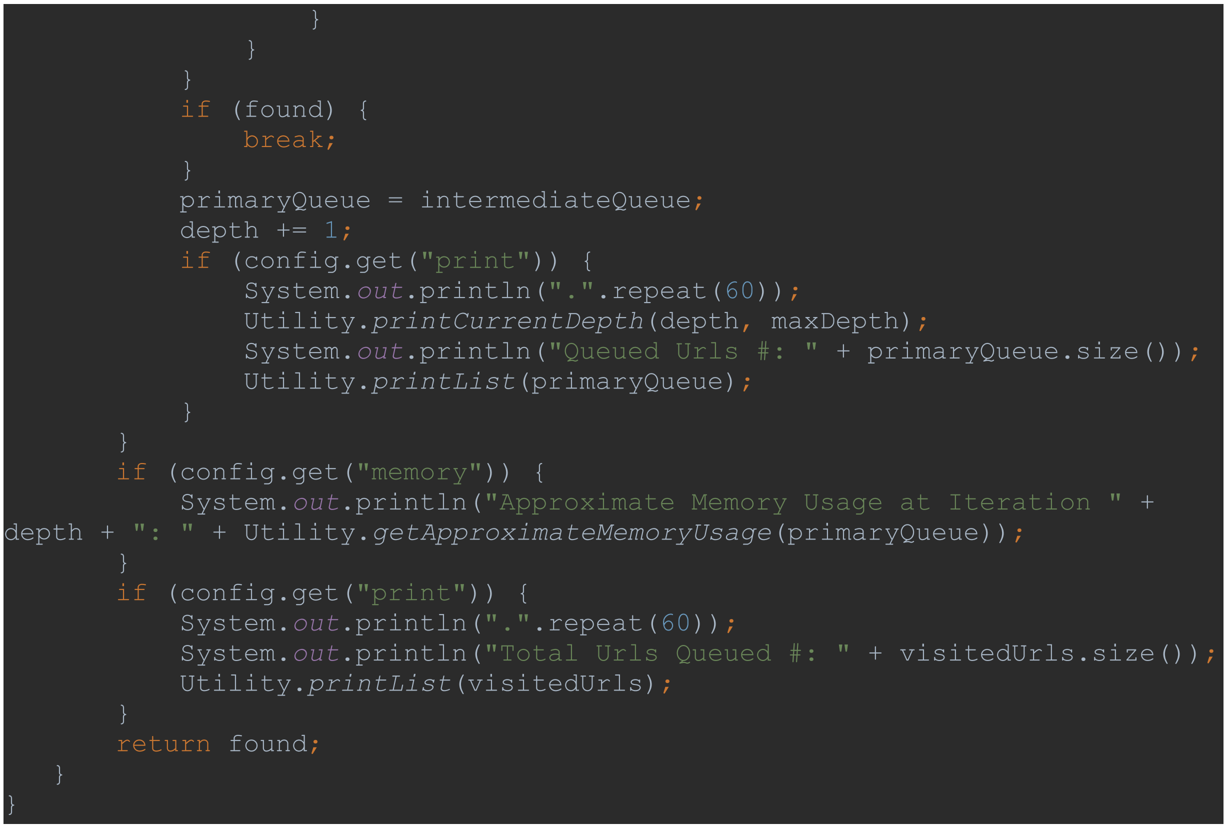
Appendix: Code

Code Repository

https://github.com/SwapnenduM/crawlingAlgorithms

Breadth First Search

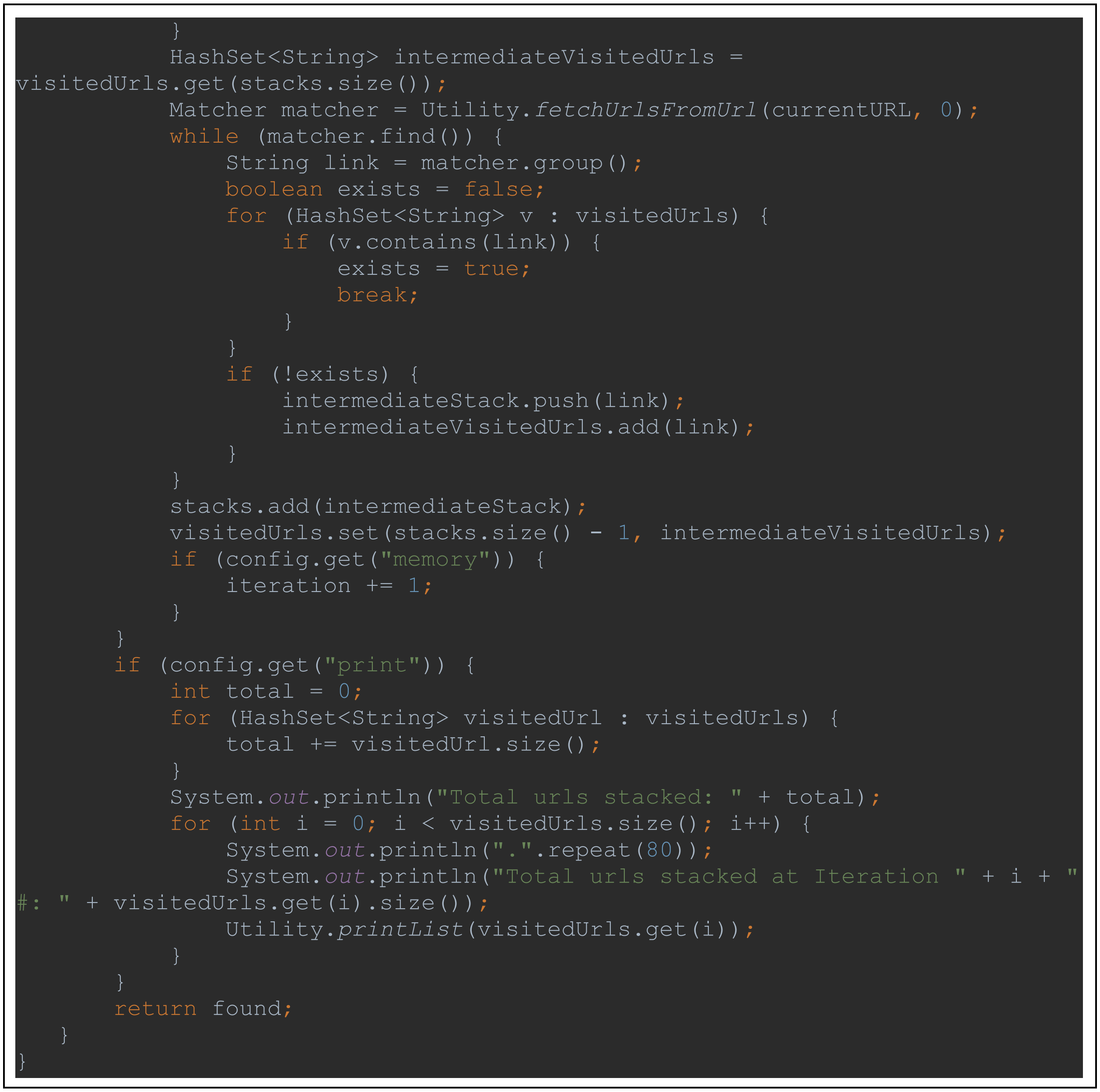
```
package GraphAlgorithms;

import java.util.*;
import java.util.regex.Matcher;

public class BFS {
    public static boolean crawl(String seedURL, int maxDepth, String
targetURL, Map<String, Boolean> config) {
        Queue<String> primaryQueue = new LinkedList<>();
        HashSet<String> visitedUrls = new HashSet<>();

        primaryQueue.add(seedURL);
        visitedUrls.add(seedURL);

        int depth = 0;
        if (config.get("print")) {
            System.out.println(".".repeat(60));
            Utility.printCurrentDepth(depth, maxDepth);
            System.out.println("Queued Urls #: " + primaryQueue.size());
            Utility.printList(primaryQueue);
        }
        if (config.get("memory")) {
            System.out.println(".".repeat(60));
        }
        boolean found = false;
        while (depth < maxDepth) {
            if (config.get("memory")) {
                System.out.println("Approximate Memory Usage at Iteration " +
depth + ": " + Utility.getApproximateMemoryUsage(primaryQueue));
            }
            Queue<String> intermediateQueue = new LinkedList<>();
            while (!primaryQueue.isEmpty()) {
                String currentURL = primaryQueue.poll();
                if (config.get("print")) {
                    System.out.println(".".repeat(60));
                    System.out.println("Url to crawl: " + currentURL);
                }
                found = Objects.equals(currentURL, targetURL);
                if (found) {
                    break;
                }
                Matcher matcher = Utility.fetchUrlsFromUrl(currentURL, 0);
                while (matcher.find()) {
                    String link = matcher.group();
                    if (!visitedUrls.contains(link)) {
                        intermediateQueue.add(link);
                        visitedUrls.add(link);
```



```
                    }
                }
            }
            if (found) {
                break;
            }
            primaryQueue = intermediateQueue;
            depth += 1;
            if (config.get("print")) {
                System.out.println(".".repeat(60));
                Utility.printCurrentDepth(depth, maxDepth);
                System.out.println("Queued Urls #: " + primaryQueue.size());
                Utility.printList(primaryQueue);
            }
        }
        if (config.get("memory")) {
            System.out.println("Approximate Memory Usage at Iteration " +
depth + ": " + Utility.getApproximateMemoryUsage(primaryQueue));
        }
        if (config.get("print")) {
            System.out.println(".".repeat(60));
            System.out.println("Total Urls Queued #: " + visitedUrls.size());
            Utility.printList(visitedUrls);
        }
        return found;
    }
}
```

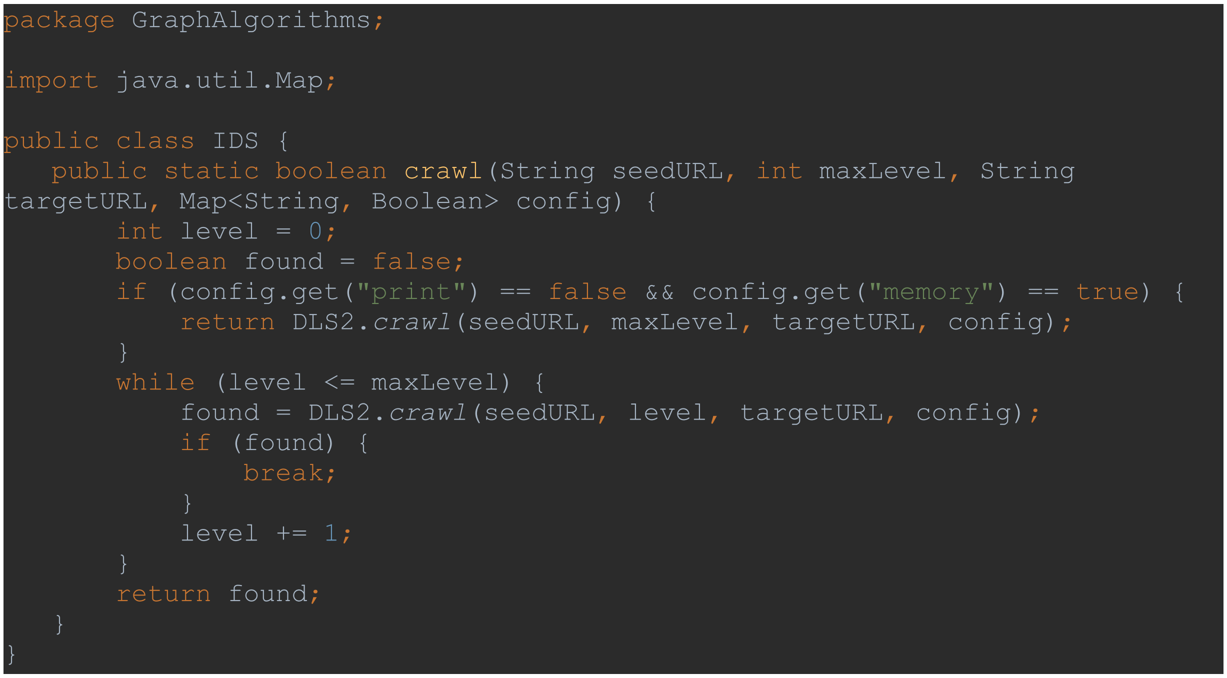
Depth First Search

```
package GraphAlgorithms;

import java.util.*;
import java.util.regex.Matcher;

public class DFS {
   public static boolean crawl(String seedURL, int maxDepth, String
targetURL, Map<String, Boolean> config) {
       List<Stack<String>> stacks = new ArrayList<>();
       List<HashSet<String>> visitedUrls = new ArrayList<>();

       Stack<String> _st = new Stack<>();
       _st.push(seedURL);
       stacks.add(_st);

       HashSet<String> _hs;
       for (int i = 0; i < maxDepth + 1; i++) {
           _hs = new HashSet<>();
           visitedUrls.add(_hs);
       }

       _hs = visitedUrls.get(0);
       _hs.add(seedURL);
       visitedUrls.set(0, _hs);

       if (config.get("memory")) {
           System.out.println(".".repeat(60));
       }
       int iteration = 0;
       boolean found = false;
       while (stacks.size() > 0) {
           if (config.get("memory")) {
               System.out.println("Approximate Memory Usage at Iteration " +
iteration + ": " + Utility.getApproximateMemoryUsage(stacks));
           }
           Stack<String> stack = stacks.get(stacks.size() - 1);
           if (stack.size() == 0) {
               if (config.get("memory")) {
                   iteration += 1;
               }
               stacks.remove(stacks.size() - 1);
               continue;
           }
           String currentURL = stack.pop();
           found = Objects.equals(currentURL, targetURL);
           if (found) {
               break;
           }
           Stack<String> intermediateStack = new Stack<>();
           if (stacks.size() - 1 >= maxDepth) {
               if (config.get("memory")) {
                   iteration += 1;
               }
               continue;
```



```
            }
            HashSet<String> intermediateVisitedUrls =
visitedUrls.get(stacks.size());
            Matcher matcher = Utility.fetchUrlsFromUrl(currentURL, 0);
            while (matcher.find()) {
                String link = matcher.group();
                boolean exists = false;
                for (HashSet<String> v : visitedUrls) {
                    if (v.contains(link)) {
                        exists = true;
                        break;
                    }
                }
                if (!exists) {
                    intermediateStack.push(link);
                    intermediateVisitedUrls.add(link);
                }
            }
            stacks.add(intermediateStack);
            visitedUrls.set(stacks.size() - 1, intermediateVisitedUrls);
            if (config.get("memory")) {
                iteration += 1;
            }
        }
        if (config.get("print")) {
            int total = 0;
            for (HashSet<String> visitedUrl : visitedUrls) {
                total += visitedUrl.size();
            }
            System.out.println("Total urls stacked: " + total);
            for (int i = 0; i < visitedUrls.size(); i++) {
                System.out.println(".".repeat(80));
                System.out.println("Total urls stacked at Iteration " + i + "
#: " + visitedUrls.get(i).size());
                Utility.printList(visitedUrls.get(i));
            }
        }
        return found;
    }
}
```

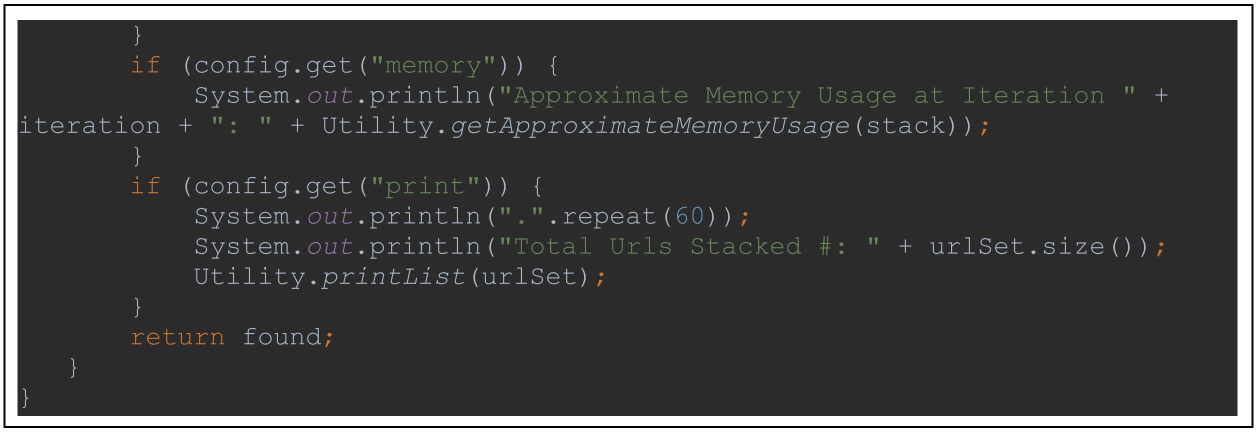
Iterative Depth Search



```
package GraphAlgorithms;

import java.util.Map;

public class IDS {
   public static boolean crawl(String seedURL, int maxLevel, String
targetURL, Map<String, Boolean> config) {
       int level = 0;
       boolean found = false;
       if (config.get("print") == false && config.get("memory") == true) {
           return DLS2.crawl(seedURL, maxLevel, targetURL, config);
       }
       while (level <= maxLevel) {
           found = DLS2.crawl(seedURL, level, targetURL, config);
           if (found) {
               break;
           }
           level += 1;
       }
       return found;
   }
}
```

Depth Limited Search

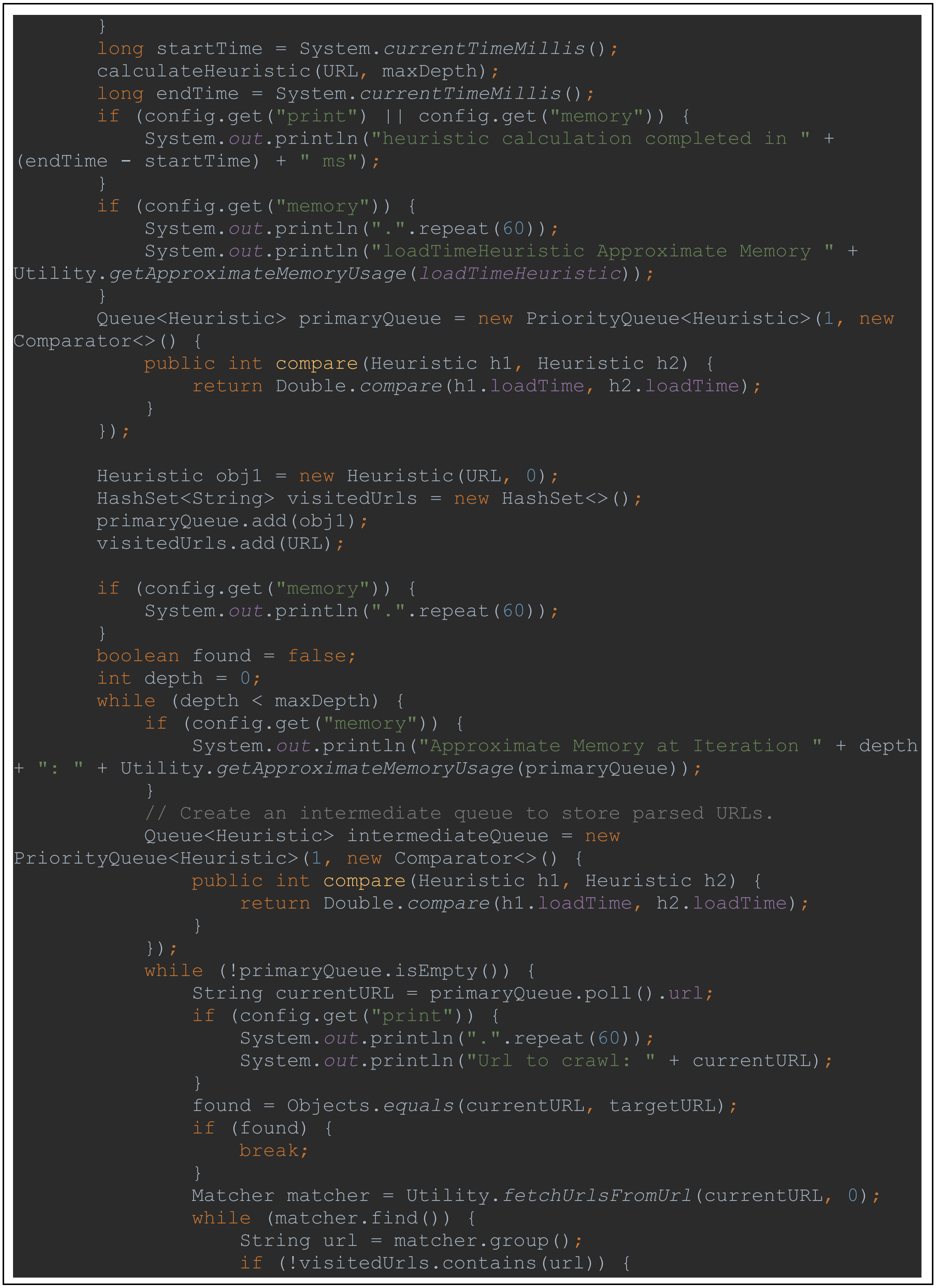
```java
package GraphAlgorithms;

import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.Stack;
import java.util.regex.Matcher;

public class DLS2 {
   public static boolean crawl(String seedURL, int maxLevel, String
targetURL, Map<String, Boolean> config) {
       Stack<String> stack = new Stack<>();
       Map<String, Integer> urlSet = new HashMap<>();

       stack.add(seedURL);
       urlSet.put(seedURL, 1);

       if (config.get("memory")) {
           System.out.println(".".repeat(60));
       }
       int iteration = 0;
       boolean found = false;
       while (!stack.isEmpty()) {
           if (config.get("memory")) {
               System.out.println("Approximate Memory Usage at Iteration " +
iteration + ": " + Utility.getApproximateMemoryUsage(stack));
           }
           String currentURL = stack.pop();
           if (config.get("print")) {
               if (urlSet.get(currentURL) <= maxLevel) {
                   System.out.println(".".repeat(60));
                   System.out.println("Url to crawl: " + currentURL);
               }
           }
           found = Objects.equals(currentURL, targetURL);
           if (found) {
               break;
           }
           if (urlSet.get(currentURL) <= maxLevel) {
               Matcher matcher = Utility.fetchUrlsFromUrl(currentURL, 0);
               while (matcher.find()) {
                   String url = matcher.group();
                   if (!urlSet.containsKey(url)) {
                       if (config.get("print")) {
                           System.out.println(" - Fetched Url: " + url);
                       }
                       urlSet.put(url, urlSet.get(currentURL) + 1);
                       stack.add(url);
                   }
               }
           }
           if (config.get("memory")) {
               iteration += 1;
           }
```



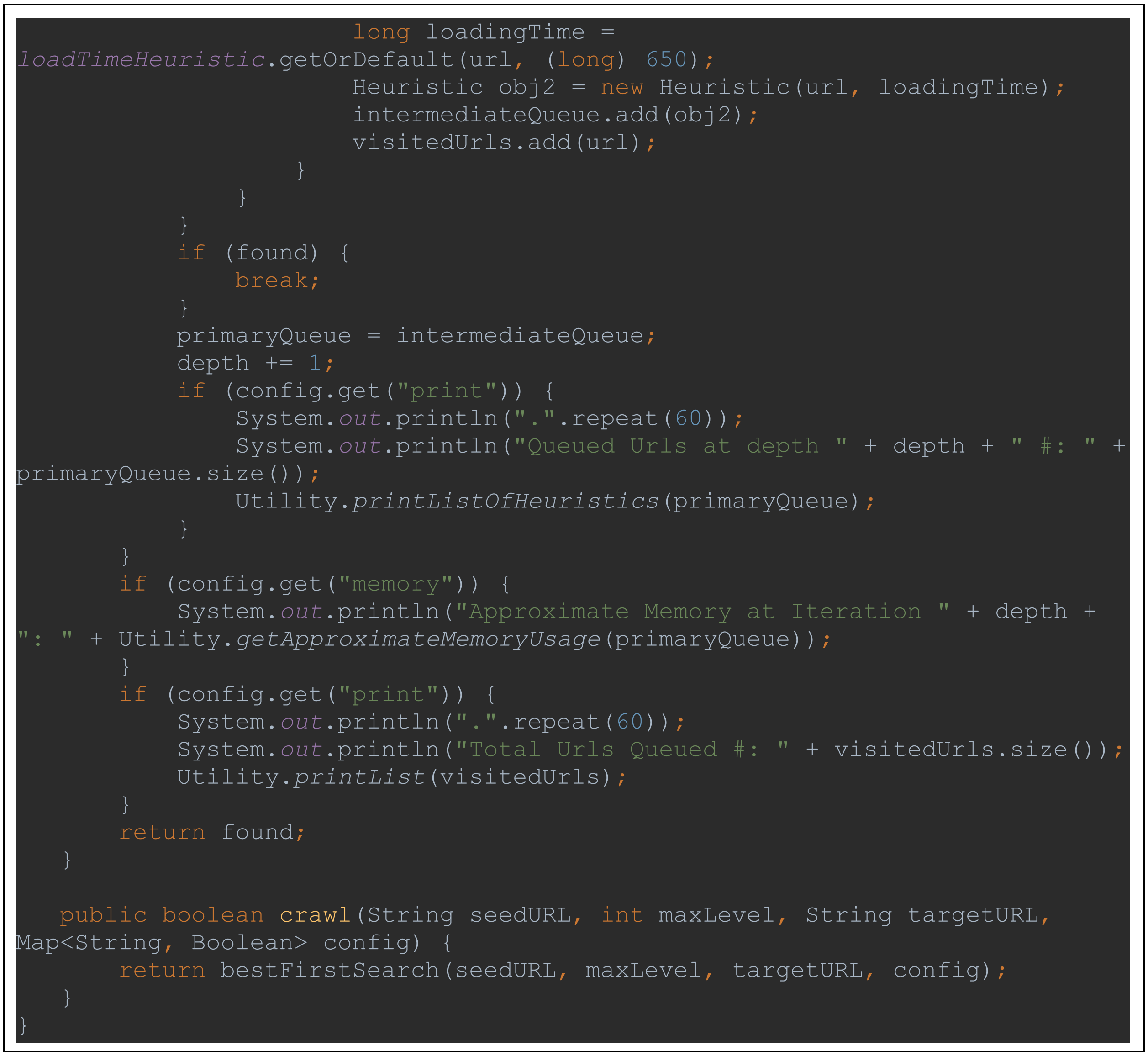
```
        }
        if (config.get("memory")) {
            System.out.println("Approximate Memory Usage at Iteration " +
iteration + ": " + Utility.getApproximateMemoryUsage(stack));
        }
        if (config.get("print")) {
            System.out.println(".".repeat(60));
            System.out.println("Total Urls Stacked #: " + urlSet.size());
            Utility.printList(urlSet);
        }
        return found;
    }
}
```

Greedy Search

```java
package GraphAlgorithms;

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.*;
import java.util.regex.Matcher;

public class Greedy {
    public static Map<String, Long> loadTimeHeuristic = new HashMap<>();

    private void calculateHeuristic(String seedURL, int maxLevel) {
        Queue<String> primaryQueue = new LinkedList<>();
        primaryQueue.add(seedURL);
        long time = 0;
        loadTimeHeuristic.put(seedURL, time);

        int level = 0;
        while (level < maxLevel) {
            Queue<String> intermediateQueue = new LinkedList<>();
            while (!primaryQueue.isEmpty()) {
                String currentURL = primaryQueue.poll();
                Matcher matcher = Utility.fetchUrlsFromUrl(currentURL, 1);
                while (matcher.find()) {
                    String link = matcher.group();
                    try {
                        if (!loadTimeHeuristic.containsKey(link)) {
                            HttpURLConnection connection = null;
                            URL url = new URL(link);
                            long start = System.currentTimeMillis();
                            connection = (HttpURLConnection)
url.openConnection();
                            connection.getInputStream();
                            long finish = System.currentTimeMillis();
                            long totalTime = finish - start;
                            loadTimeHeuristic.put(link, totalTime);
                            intermediateQueue.add(link);
                        }
                    } catch (IOException e) {
                        // TODO: REMOVE. Suppressed for debugging purposes.
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
            primaryQueue = intermediateQueue;
            level += 1;
        }
    }

    private boolean bestFirstSearch(String URL, int maxDepth, String
targetURL, Map<String, Boolean> config) {
        if (config.get("print") || config.get("memory")) {
            System.out.println("Calculating heuristic");
```



```
        }
        long startTime = System.currentTimeMillis();
        calculateHeuristic(URL, maxDepth);
        long endTime = System.currentTimeMillis();
        if (config.get("print") || config.get("memory")) {
            System.out.println("heuristic calculation completed in " +
(endTime - startTime) + " ms");
        }
        if (config.get("memory")) {
            System.out.println(".".repeat(60));
            System.out.println("loadTimeHeuristic Approximate Memory " +
Utility.getApproximateMemoryUsage(loadTimeHeuristic));
        }
        Queue<Heuristic> primaryQueue = new PriorityQueue<Heuristic>(1, new
Comparator<>() {
            public int compare(Heuristic h1, Heuristic h2) {
                return Double.compare(h1.loadTime, h2.loadTime);
            }
        });

        Heuristic obj1 = new Heuristic(URL, 0);
        HashSet<String> visitedUrls = new HashSet<>();
        primaryQueue.add(obj1);
        visitedUrls.add(URL);

        if (config.get("memory")) {
            System.out.println(".".repeat(60));
        }
        boolean found = false;
        int depth = 0;
        while (depth < maxDepth) {
            if (config.get("memory")) {
                System.out.println("Approximate Memory at Iteration " + depth
+ ": " + Utility.getApproximateMemoryUsage(primaryQueue));
            }
            // Create an intermediate queue to store parsed URLs.
            Queue<Heuristic> intermediateQueue = new
PriorityQueue<Heuristic>(1, new Comparator<>() {
                public int compare(Heuristic h1, Heuristic h2) {
                    return Double.compare(h1.loadTime, h2.loadTime);
                }
            });
            while (!primaryQueue.isEmpty()) {
                String currentURL = primaryQueue.poll().url;
                if (config.get("print")) {
                    System.out.println(".".repeat(60));
                    System.out.println("Url to crawl: " + currentURL);
                }
                found = Objects.equals(currentURL, targetURL);
                if (found) {
                    break;
                }
                Matcher matcher = Utility.fetchUrlsFromUrl(currentURL, 0);
                while (matcher.find()) {
                    String url = matcher.group();
                    if (!visitedUrls.contains(url)) {
```



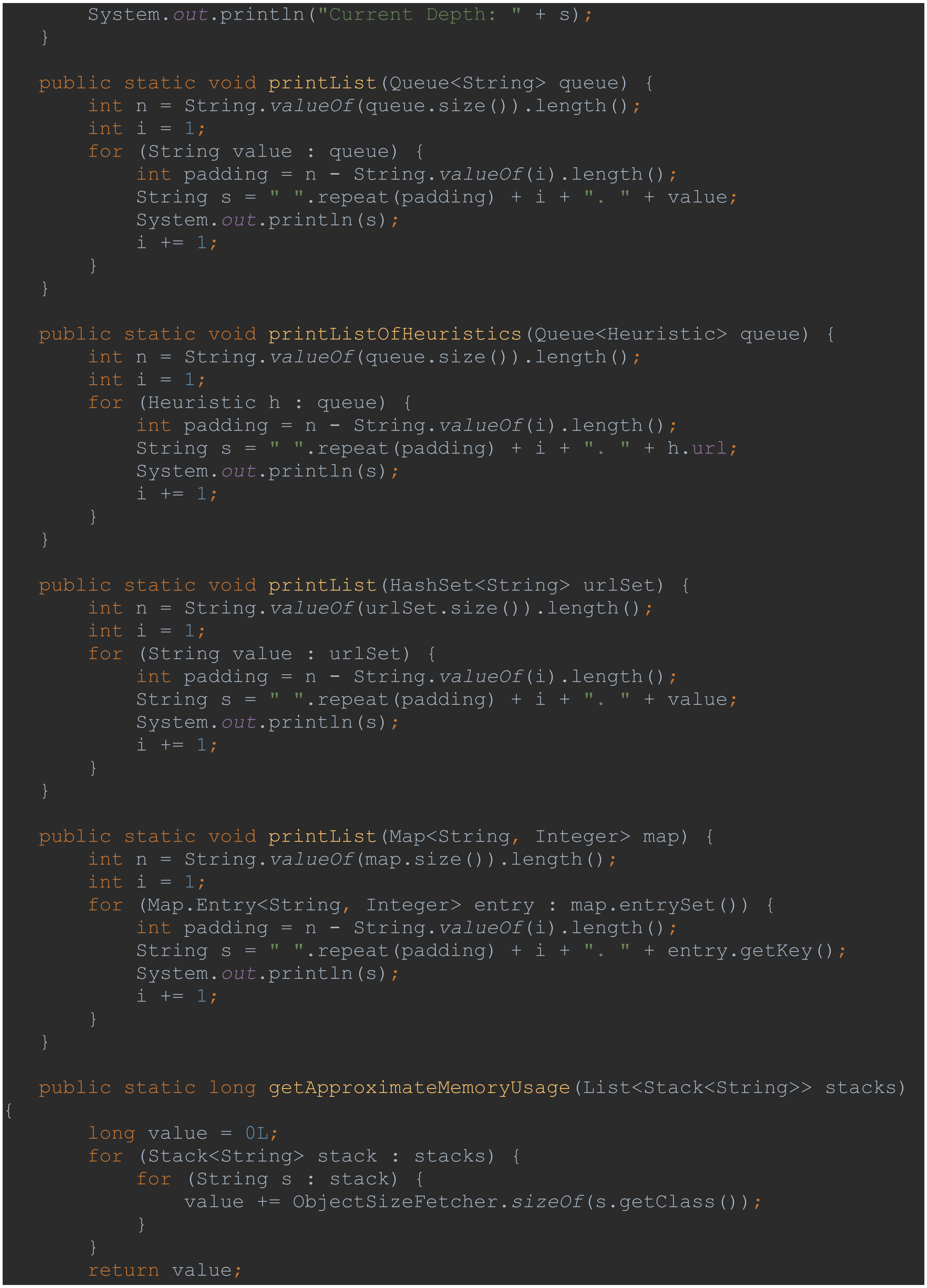
```
                       long loadingTime =
loadTimeHeuristic.getOrDefault(url, (long) 650);
                       Heuristic obj2 = new Heuristic(url, loadingTime);
                       intermediateQueue.add(obj2);
                       visitedUrls.add(url);
                   }
               }
           }
           if (found) {
               break;
           }
           primaryQueue = intermediateQueue;
           depth += 1;
           if (config.get("print")) {
               System.out.println(".".repeat(60));
               System.out.println("Queued Urls at depth " + depth + " #: " +
primaryQueue.size());
               Utility.printListOfHeuristics(primaryQueue);
           }
       }
       if (config.get("memory")) {
           System.out.println("Approximate Memory at Iteration " + depth +
": " + Utility.getApproximateMemoryUsage(primaryQueue));
       }
       if (config.get("print")) {
           System.out.println(".".repeat(60));
           System.out.println("Total Urls Queued #: " + visitedUrls.size());
           Utility.printList(visitedUrls);
       }
       return found;
   }

   public boolean crawl(String seedURL, int maxLevel, String targetURL,
Map<String, Boolean> config) {
       return bestFirstSearch(seedURL, maxLevel, targetURL, config);
   }
}
```

Object Size Fetcher

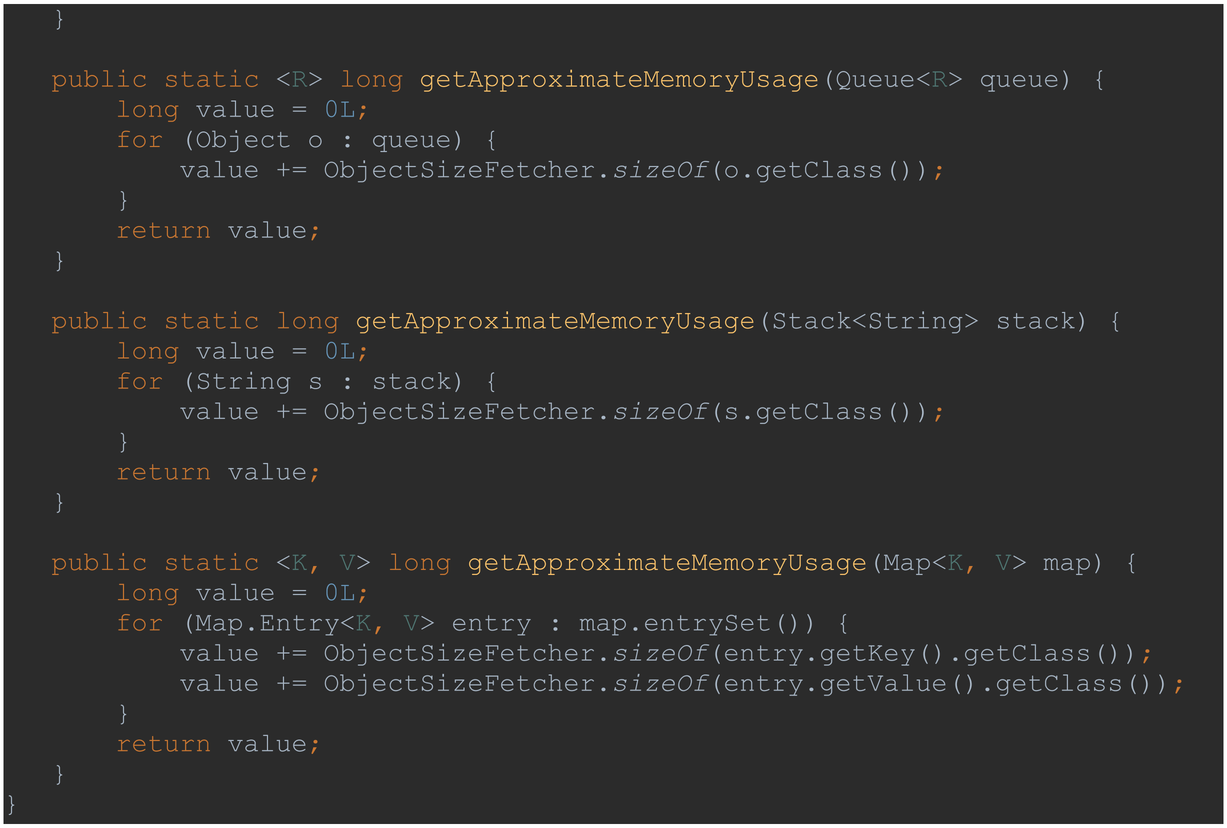
```
package GraphAlgorithms;

import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

public class ObjectSizeFetcher {

   private static final int NR_BITS =
Integer.valueOf(System.getProperty("sun.arch.data.model"));
   private static final int BYTE = 8;
   private static final int WORD = NR_BITS / BYTE;
   private static final int HEADER_SIZE = 8;

   public static int sizeOf(Class<?> object) {
       int result = 0;

       while (object != null) {
           Field[] fields = object.getDeclaredFields();
           for (int i = 0; i < fields.length; i++) {
               if (!Modifier.isStatic(fields[i].getModifiers())) {
                   if (fields[i].getType().isPrimitive()) {
                       Class<?> primitiveClass = fields[i].getType();
                       if (primitiveClass == boolean.class || primitiveClass
== byte.class) {
                           result += 1;
                       } else if (primitiveClass == short.class) {
                           result += 2;
                       } else if (primitiveClass == int.class ||
primitiveClass == float.class) {
                           result += 4;
                       } else if (primitiveClass == double.class ||
primitiveClass == long.class) {
                           result += 8;
                       }
                   } else {
                       // assume compressed references.
                       result += 4;
                   }
               }
           }

           object = object.getSuperclass();

           // round up to the nearest WORD length.
           if ((result % WORD) != 0) {
               result += WORD - (result % WORD);
           }
       }

       result += HEADER_SIZE;

       return result;
   }
}
```

Utility

```
package GraphAlgorithms;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class Utility {

    public static String urlRegexVersion0 = "https://(\\w+\\.)*(\\w+)";
    public static String urlRegexVersion1 =
"https?:\\/\\/(www\\.)?[-a-zA-Z0-9@:%._\\+~#=]{1,256}\\.[a-zA-Z0-9()]{1,6}\\
b([-a-zA-Z0-9()@:%_\\+.~#?&//=]*)";

    public static String fetchUrlContent(String url) {
        String content = "";
        try {
            URL _url = new URL(url);
            BufferedReader input = new BufferedReader(new
InputStreamReader(_url.openStream()));
            content = input.lines().collect(Collectors.joining());
            input.close();
        } catch (IOException e) {
            // TODO: REMOVE. Suppressed for debugging purposes.
        } catch (Exception e) {
            e.printStackTrace();
        }
        return content;
    }

    public static Matcher fetchUrlsFromUrl(String url, int version) {
        String content = Utility.fetchUrlContent(url);
        String regex = Utility.urlRegexVersion0;
        if (version == 1) {
            regex = Utility.urlRegexVersion1;
        }
        Pattern pattern = Pattern.compile(regex);
        return pattern.matcher(content);
    }

    public static String formatTestStats(String algorithmName, int maxDepth,
long timeTaken, boolean result) {
        return "Algorithm: " + algorithmName + " - Max Depth: " + maxDepth +
" - Time: " + timeTaken + " ms" + " - Url Found: " + result;
    }

    public static void printCurrentDepth(int depth, int maxDepth) {
        int padding = String.valueOf(maxDepth).length() -
String.valueOf(depth).length();
        String s = " ".repeat(padding) + depth + " / " + maxDepth;
```



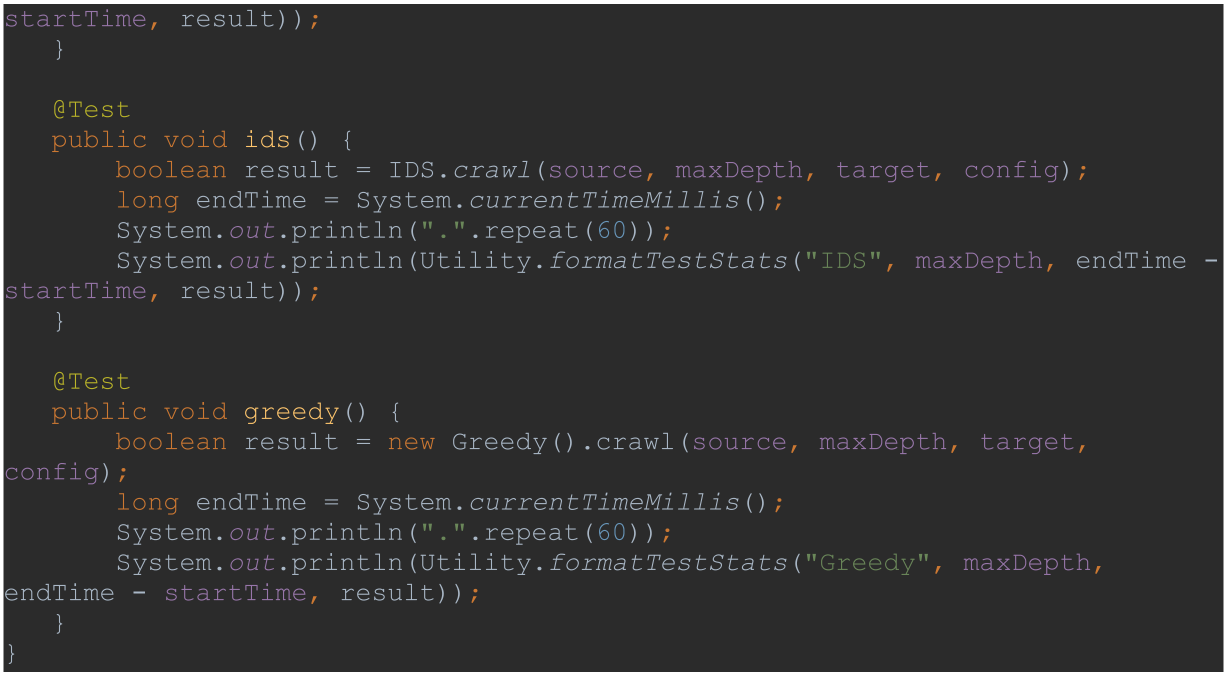
```java
        System.out.println("Current Depth: " + s);
    }

    public static void printList(Queue<String> queue) {
        int n = String.valueOf(queue.size()).length();
        int i = 1;
        for (String value : queue) {
            int padding = n - String.valueOf(i).length();
            String s = " ".repeat(padding) + i + ". " + value;
            System.out.println(s);
            i += 1;
        }
    }

    public static void printListOfHeuristics(Queue<Heuristic> queue) {
        int n = String.valueOf(queue.size()).length();
        int i = 1;
        for (Heuristic h : queue) {
            int padding = n - String.valueOf(i).length();
            String s = " ".repeat(padding) + i + ". " + h.url;
            System.out.println(s);
            i += 1;
        }
    }

    public static void printList(HashSet<String> urlSet) {
        int n = String.valueOf(urlSet.size()).length();
        int i = 1;
        for (String value : urlSet) {
            int padding = n - String.valueOf(i).length();
            String s = " ".repeat(padding) + i + ". " + value;
            System.out.println(s);
            i += 1;
        }
    }

    public static void printList(Map<String, Integer> map) {
        int n = String.valueOf(map.size()).length();
        int i = 1;
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            int padding = n - String.valueOf(i).length();
            String s = " ".repeat(padding) + i + ". " + entry.getKey();
            System.out.println(s);
            i += 1;
        }
    }

    public static long getApproximateMemoryUsage(List<Stack<String>> stacks)
{
        long value = 0L;
        for (Stack<String> stack : stacks) {
            for (String s : stack) {
                value += ObjectSizeFetcher.sizeOf(s.getClass());
            }
        }
        return value;
```



```
    }

    public static <R> long getApproximateMemoryUsage(Queue<R> queue) {
        long value = 0L;
        for (Object o : queue) {
            value += ObjectSizeFetcher.sizeOf(o.getClass());
        }
        return value;
    }

    public static long getApproximateMemoryUsage(Stack<String> stack) {
        long value = 0L;
        for (String s : stack) {
            value += ObjectSizeFetcher.sizeOf(s.getClass());
        }
        return value;
    }

    public static <K, V> long getApproximateMemoryUsage(Map<K, V> map) {
        long value = 0L;
        for (Map.Entry<K, V> entry : map.entrySet()) {
            value += ObjectSizeFetcher.sizeOf(entry.getKey().getClass());
            value += ObjectSizeFetcher.sizeOf(entry.getValue().getClass());
        }
        return value;
    }
}
```

Graph Algorithm Test

```
import GraphAlgorithms.*;
import org.junit.Before;
import org.junit.Test;

import java.util.HashMap;
import java.util.Map;

public class GraphAlgorithmsTest {
   private final int maxDepth = 2;
   private long startTime;
   private String source = "";
   private String target = "";

   private final Map<String, Boolean> config = new HashMap<>() {{
       put("print", false);
       put("memory", false);
   }};

   @Before
   public void setUp() throws Exception {
       source = "https://basuanw.sites.northeastern.edu/";
       target = "https://fonts.googleapis.com";
       target = "";
       config.put("print", false);
       config.put("memory", false);
       startTime = System.currentTimeMillis();
   }

   @Test
   public void bfs() {
       boolean result = BFS.crawl(source, maxDepth, target, config);
       long endTime = System.currentTimeMillis();
       System.out.println(".".repeat(60));
       System.out.println(Utility.formatTestStats("BFS", maxDepth, endTime -
startTime, result));
   }

   @Test
   public void dfs() {
       boolean result = DFS.crawl(source, maxDepth, target, config);
       long endTime = System.currentTimeMillis();
       System.out.println(".".repeat(60));
       System.out.println(Utility.formatTestStats("DFS", maxDepth, endTime -
startTime, result));
   }

   @Test
   public void dls() {
       boolean result = DLS2.crawl(source, maxDepth, target, config);
       long endTime = System.currentTimeMillis();
       System.out.println(".".repeat(60));
       System.out.println(Utility.formatTestStats("DLS", maxDepth, endTime -
```



```
startTime, result));
    }

    @Test
    public void ids() {
        boolean result = IDS.crawl(source, maxDepth, target, config);
        long endTime = System.currentTimeMillis();
        System.out.println(".".repeat(60));
        System.out.println(Utility.formatTestStats("IDS", maxDepth, endTime -
startTime, result));
    }

    @Test
    public void greedy() {
        boolean result = new Greedy().crawl(source, maxDepth, target,
config);
        long endTime = System.currentTimeMillis();
        System.out.println(".".repeat(60));
        System.out.println(Utility.formatTestStats("Greedy", maxDepth,
endTime - startTime, result));
    }
}
```