Programming II

Hyperlinks Checker Project

Prepared by:

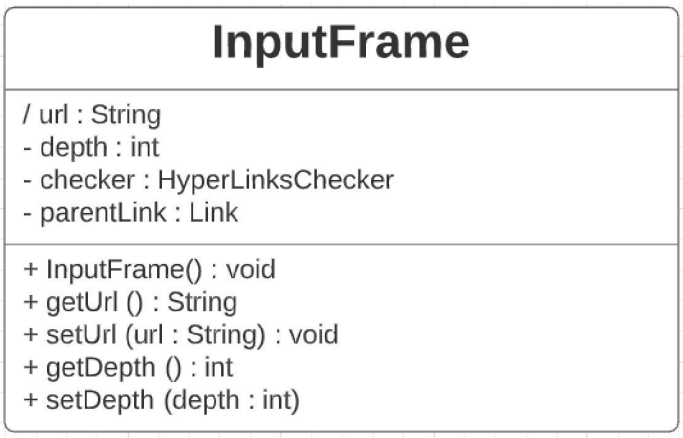
Name
Youssef Samuel Nachaat Labib
Youssef Amr Ismail Othman

Classes Description

Our hyperlinks checker project is implemented by the use of OOP using 3 main classes: **HyperLinkschecker** , **Link** and **ThreadValidator**. In addition of 3 classes for the GUI: **InputFrame**, **OutputFrame**, **ShowInformationFrame** .

We will discuss the state and the behaviour for each class individually and how they interact with each other.

1) InputFrame Class



UML Class Diagram (InputFrame)

The InputFrame class is responsible for getting the input from the user.

- URL: The url that the user wants to validate.
- Depth: The depth where the user wants the program to stop before checking more links.

The **InputFrame** attributes are:

- url of type String: which is the url of the link.
- depth of type int.
- checker : object of class HyperLinksChecker.
- parentLink: object of class Link.

Once an instance of the class **InputFrame** is constructed: The user is asked to enter the url and the depth. When the user press the check button, the input text is extracted using getText() method, and an instance of the class **Link** is created.

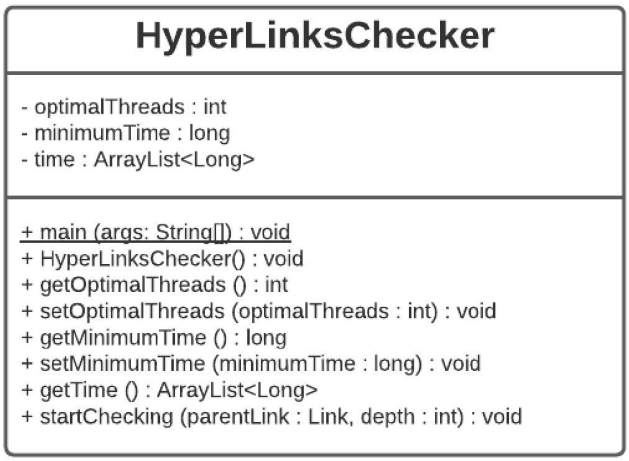
After validating the input, an instance of the class **HyperLinksChecker** is created and the method startChecking() is called, to start checking the link and its sublinks till the specified depth.(which will be explained later).

Validating the input:

1-URL: The method isValidOneURL() is called to check this link entered by the user, if it is not valid, a message in displayed to the user: “Invalid URL input!”, and the user is asked to re-enter the data.

2-Depth: A try-catch block is used when getting the input depth from the user, to catch there is a **NumberFormatException**, if the exception is thrown due to invalid input, a message in displayed to the user: “Invalid depth input!”, and the user is asked to re-enter the data.

2) HyperLinksChecker Class



UML Class Diagram (HyperLinksChecker)

The **HyperLinksChecker** class is the main class in our program. It is the class that has the main method. It is responsible to get the input from the **InputFrame** class, check the link entered and finally show the output to the user using the **OutputFrame** class.

The **HyperLinksChecker** attributes are:

- optimalThreads of type int: which is the best number of threads that the checker used.
- minimumTime of type long: which is the excecution time taken to check the link using the optimal number of threads.
- time: a array list to store the execution every time a specific number of threads is used.

Note: All these attributes are private and there are getters and setters to access them. (Encapsulation OOP principal).

Methods:

- public static void main(String args[]):

The main method in our program only gets the user input from the **InputFrame** class and call the startChecking () method.

- public void startChecking(Link parentLink, int depth) throws IOException, InterruptedException

This method begins with a while loop:

We first set the execution start time in the first loop to the current execution time using System.currentTimeMillis() method.

We set the number of threads used in this first loop to 1 and it will be incremented after each loop.

Then, an if else statement is used:

-If the number the number of threads used is equal to 1: the method validateURLs() is called by the object parentLink entered by the user. After this method is executed we print the number of valid and invalid links in that link using methods: Link.getNumValidURLs() and Link.getNumInvalidURLs().

-Else: The **ExecutorService** interface is used and the maximum number of threads that can be executed at the same time is set to the number of threads determined in the beginning of the loop. The validateURLs() method is called. While the executor still has tasks to do, that means that the validate method did not finish, so the main will stop until it finishes checking using an empty while loop, that ends when the executor has zero tasks left. After that the executor is shutdown and we wait until all tasks are completing using methods: shutdown() and awaitTermination(). Finally the number of valid and invalid links is printed.

After checking the link and its sub links, we get the execution end time, and we add the “time” (end-start) array list the execution time using the number of threads in this loop.

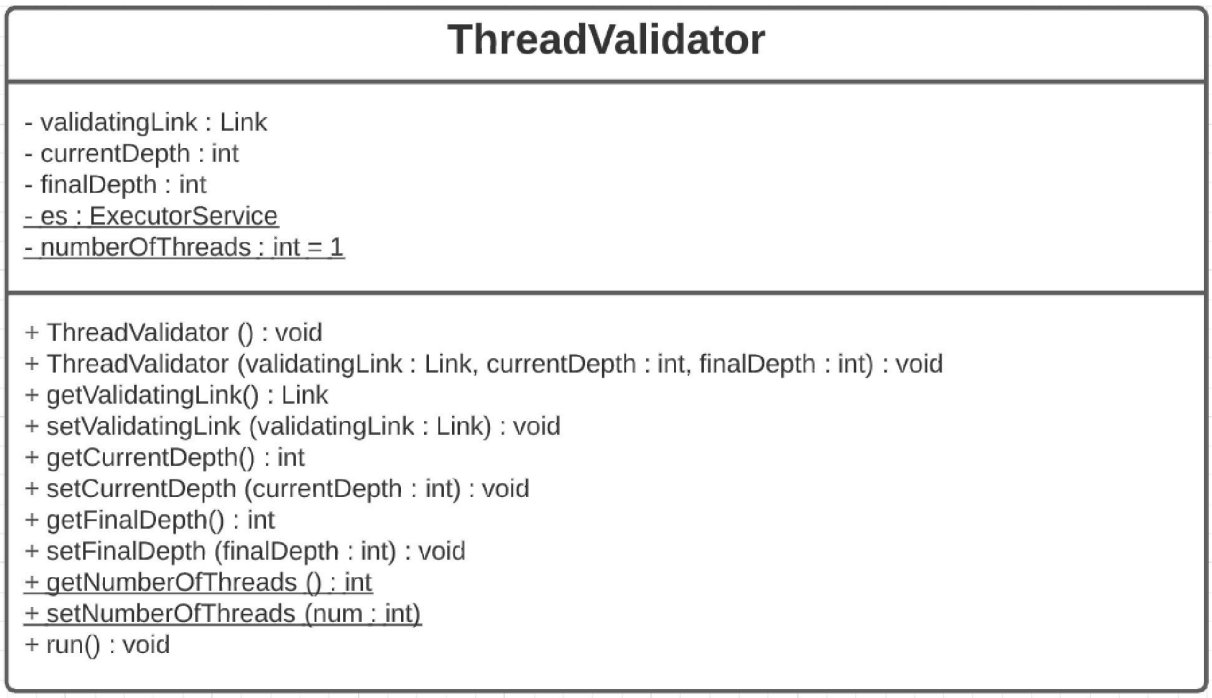
Now we check whether to end the loop or begin a new iteration:

- If the execution time of this iteration is less than the last one but by a small difference (400ms), and it is not the first iteration, the we set the optimal number of threads by the current number of threads, and the minimum time by the current execution time. And we set the Boolean variable “finish” to true to end this loop and do not begin a new one.
- Else if the current execution time is bigger than the last one, we set the optimal number of threads by the current – 1, to get the number of threads used in the last iteration, and we the minimum time by the variable “duration”, which still stores the value of the last execution time. “finish” is also set to true.
- Else, we set the variable “duration” to the current execution time (end - start). And the Boolean variable “finish” remains false so we can begin a new iteration

At the end of the while loop we print the number of threads used in this iteration and the execution time.

After the while loop has ended, we create a new object of the class **OutputFrame** to show results to the user.

3) ThreadValidator Class



UML Class Diagram (ThreadValidator)

ThreadValidator class extends the **Thread** class. It is responsible to create threads to make the process of validating the links faster.

The **ThreadValidator** attributes are:

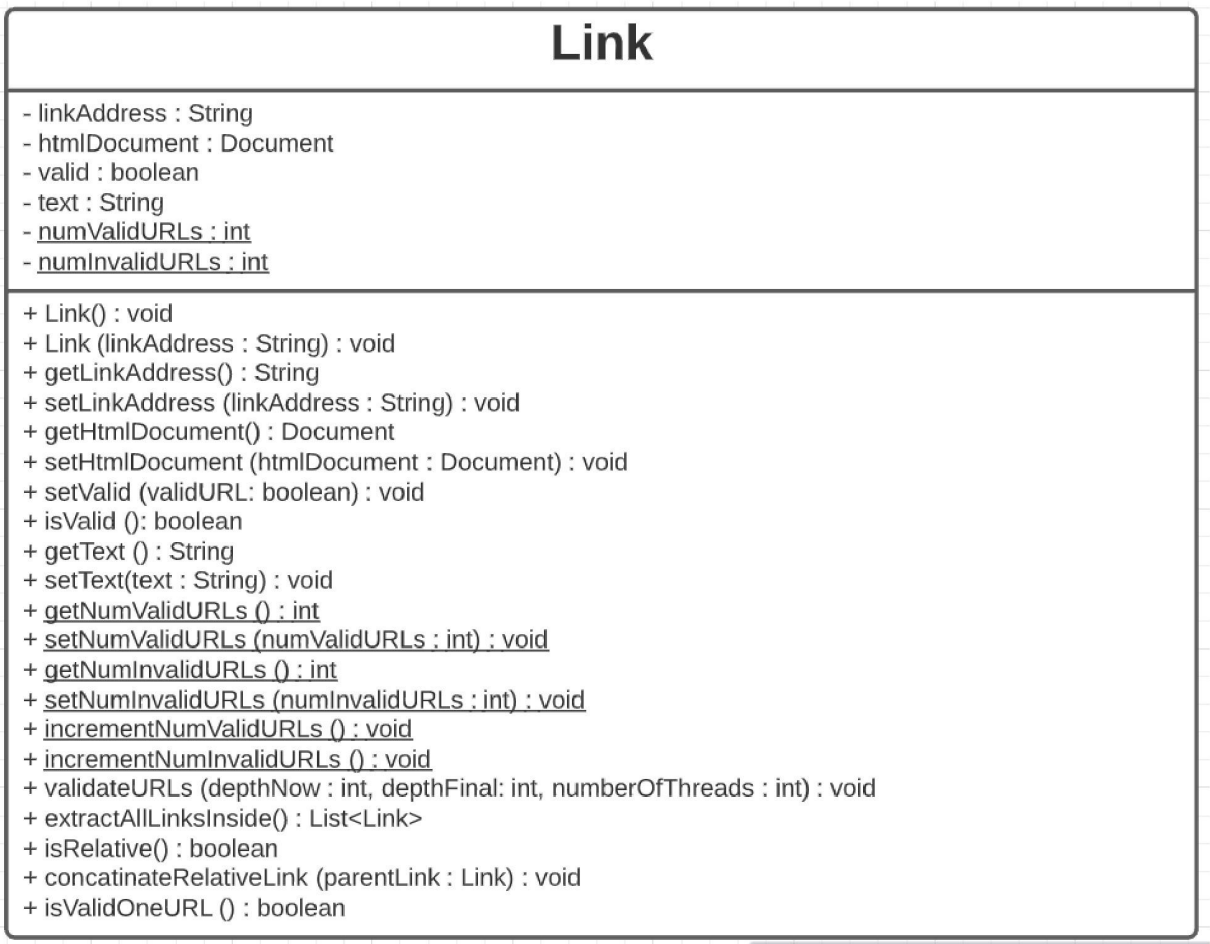
- validatingLink (private): an object of class **Link,** the link that the thread will check.
- currentDepth of type int (private): the depth of the current link from the parent link.
- finalDepth of type int (private): the maximum depth.
- es: the executor that help in executing the threads.
- numberOfThreads: which is the maximum number of threads that can execute at the same time.

Methods:

- Setters and Getters methods for all the private attributes. (Encapsulation OOP principle)
- public void run() :

When a thread starts, run() calls the method validateURLs() to let it check the link that the thread has now, and give it the current and final depth as arguments.

4) Link Class



UML Class Diagram (Link)

The **Link** class attributes are:

Private attributes with getters and setters methods:

- linkAddress of type String: the address of the link to be checked
- htmlDocument: instance of class Document which is the html document of the link.
- valid of type boolean: to decide whether the link is valid or not.
- text of type String: the text of the link written in the html document.

Static attributes to get the total number of valid and invalid Links:

- numValidURLs of type int: the total number of valid links.
- numInvalidURLs of type int: the total number of invalid links.

Methods:

- public List<Link> extractAllLinksInside() :

This method is called by a link object and returns a list of all links inside that link.

The class **ArrayList** is used to create an array of links which we do not know its size. We get the html document of that link with the help of the **jsoup** library, then we extract all the hyperlinks in that document. We loop on all the links in that document and extract the url part from the html tag. Then we create a Link object with the data extracted and add it to our list.

Then a check occurs using isRelative() method, if the link extracted is a relative link, for example "/index.com". We call concatinateRelativeLink() method to fix the link by adding the host name to the url.

Finally, the list of links is returned.

- public boolean isRelative()

This method check whether the link extracted is not complete. It returns true if it is a relative link, false otherwise.

- public void concatinateRelativeLink(Link parentLink)

This method is used to complete the missing part in the URL of a relative link. It takes as input the parent link from which we extracted this relative link. We get the host and the protocol name of the parent link by creating an object of the class **URL** (in java.net package), giving the URL constructor link address of the relative link and using the methods getHost() and getProtocol(). And finally we set the link address of the relative link by a new one which is the concatination of the protocol of the parent link, the host name and the initial URL.

- public boolean isValidOneURL()

This method is used to check a single URL. It tries to connect to the link using the **jsoup** library. If the connection succeded so we set the boolean variable "valid" to true. Is there a HttpStatusException, IOException or Illegal exception caught, we set "valid" to false.

- public void validateURLs(int depthNow, int depthFinal, int numberOfThreads)

This is a recursive method which is considered the most important method in the checking process in our system.

First, we check if the link object that called the method is valid or not using isValidOneURL() method.

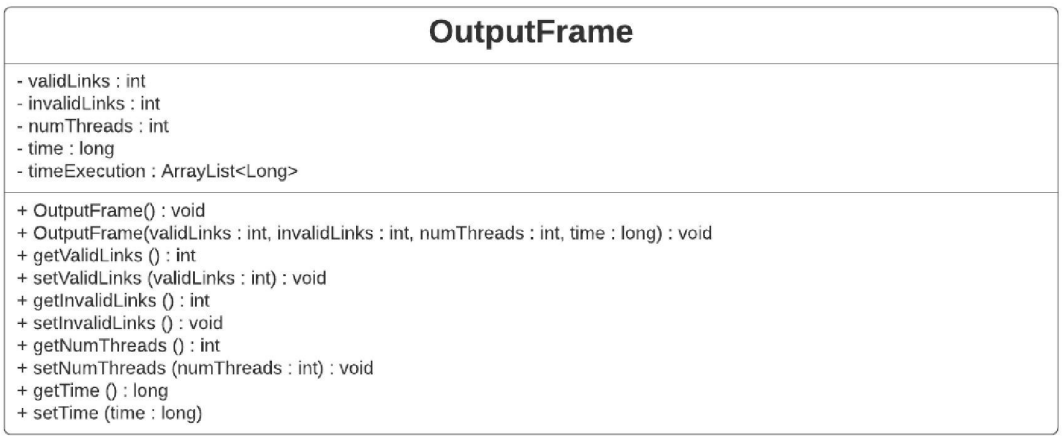
If invalid: We print this url in the console, we increment the total number of invalid urls and the method ends.

Else, if the link is valid: We print the url in the console and we increment the number of valid urls. Then, if the current depth is equal to the maximum depth the method ends because we do not need to check more links. If we continue in the method, we extract all the links in that link using extractAllLinksInside(). We all loop on those links, and for each :

- If the number of threads used equals 1 we call the method recursively again to check each sublink.
- Else, a thread object from the **ThreadValidator** class is created and we use the Executor Service to execute these threads according to the given number of threads. As we mentionned before when the thread starts it calls this recursive method again tith the given link to check it.

So by the end of this recursive method, we checked the first link given by the user, we checked the sublinks inside it according to the depth given by the user.

5) OutputFrame Class



UML Class Diagram (OutputFrame)

The **OutputFrame** class is responsible to show the output to the user.

The attributes:

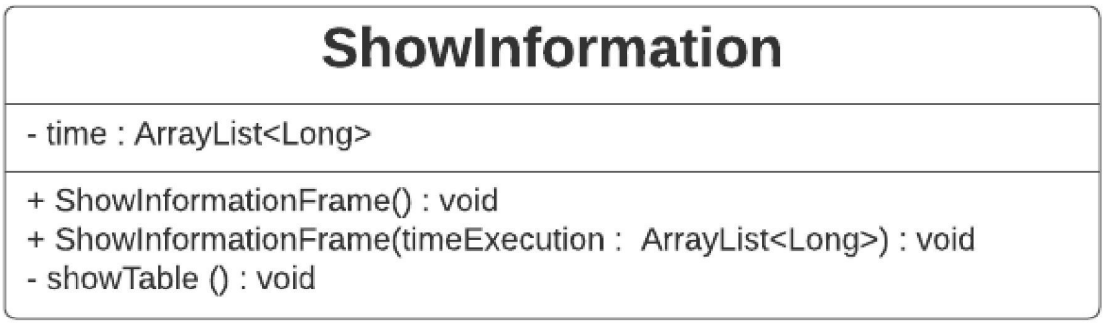
- validLinks of type int: The total number of valid links.
- invalidLinks of type int: The total number of invalid links.
- numThreads of type int: The optimal number of threads used.
- time of type long: The minimum time of execution using the optimal number of threads.
- timeExecution: an object of class ArrayList which stores the values of time execution for different number of threads used each time.

Note: All these attributes are private and have getters and setters methods.

The constructor: public OutputFrame(int validLinks, int invalidLinks, int numThreads, long time, ArrayList<Long> timeEx). When constructing an object of this frame, we show the given data in the text fields of that frame, so the user can see the results. The text fields are set to be uneditable so the user cannot change any data.

By clicking the “More Information button” in that frame, another frame appears, which is an object of the ShowInformation class, and we pass to it the array list that contains the time execution values.

6) ShowInformation Class



UML Class Diagram (ShowInformation)

This class let the user see more information about what happened in the program by letting him know the execution time for every number of threads used.

The arguments:

- time: an object of class **ArrayList** which contains the values of the execution time for different number of thread used.

The constructors:

- The default constructor: public ShowInformation()
- public ShowInformation(ArrayList<Long> timeExecution): This constructor has as input the array list of execution time which was sent from the OutputFrame. The constructor set the attribute time to this list, and call the showTable() method.

The methods:

- private void showTable() :

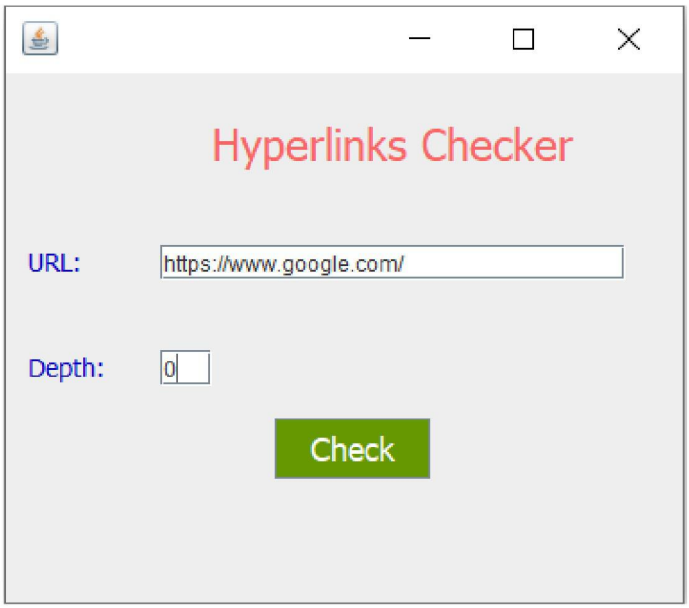
We have only two columns in that table, one for the number of threads and the other for the execution time. So, we loop on the list of execution time and we create a new a row with data from the list.

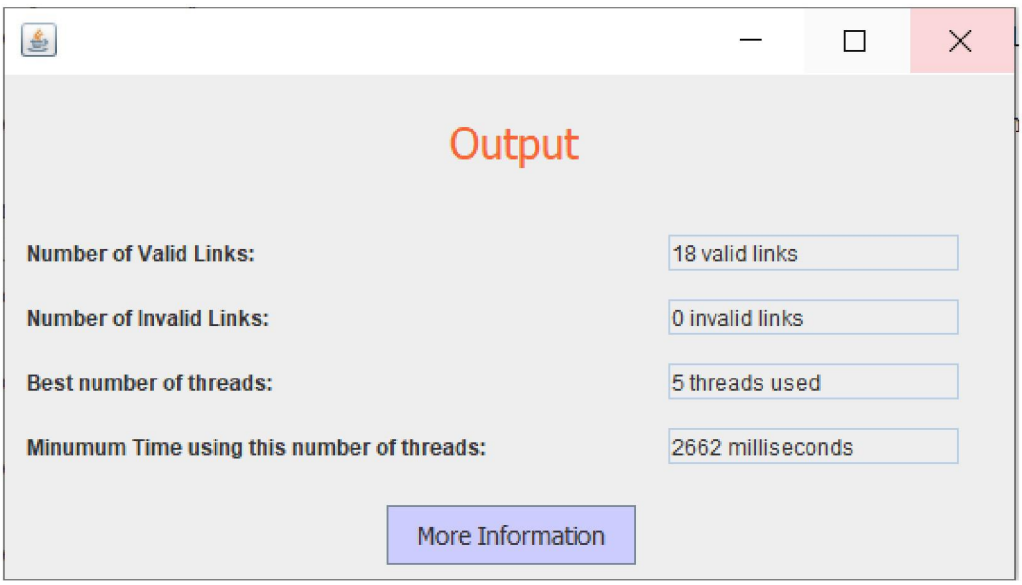
We use the **DefaultTableModel** class to model our table and the **Object** class to make an array of objects called “rowData[]” of size two. The first object will be the number of threads and the second is the time that will be got from the list of execution time. We use the method addRow() to add the array of objects in each iteration.

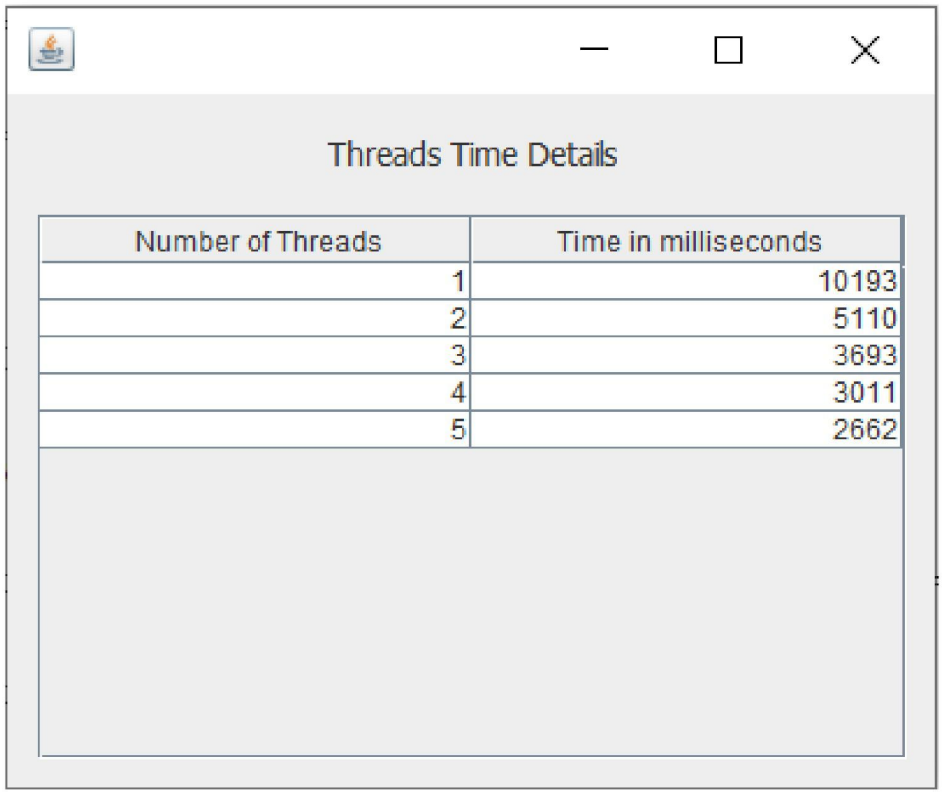
Outputs and charts

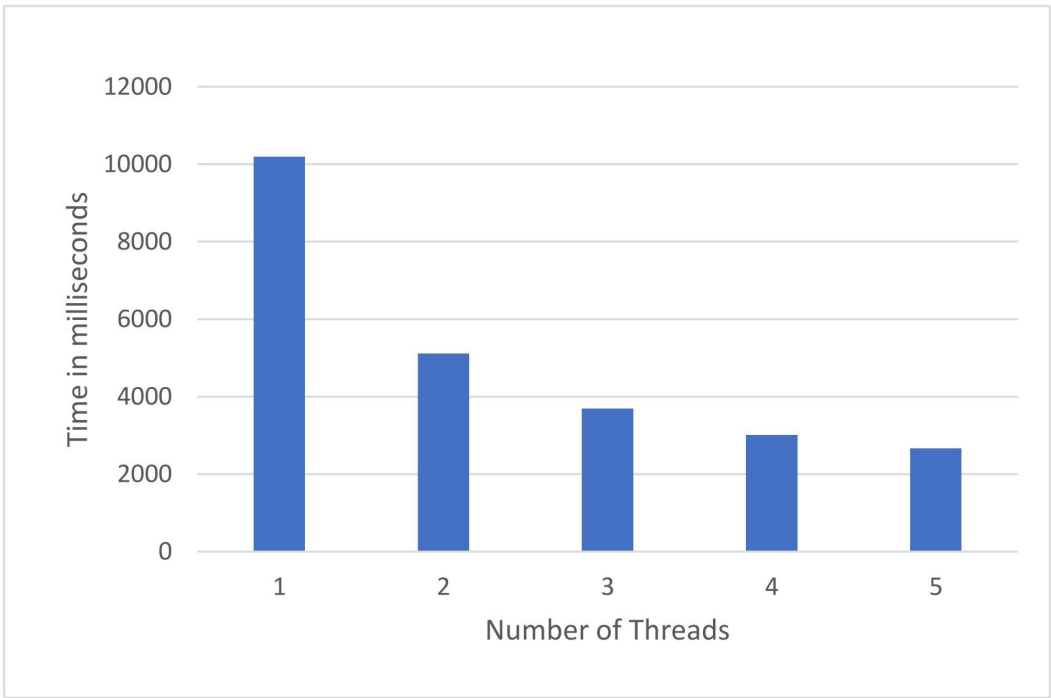
1- Link: https://www.google.com

Depth: 0



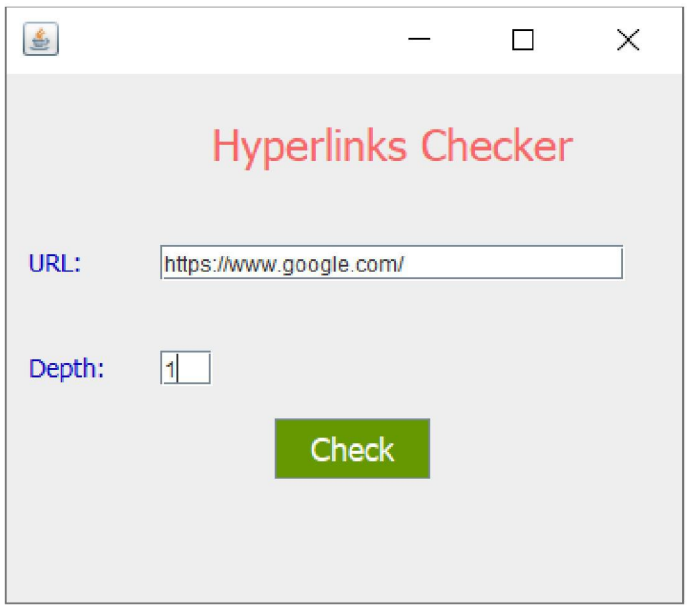


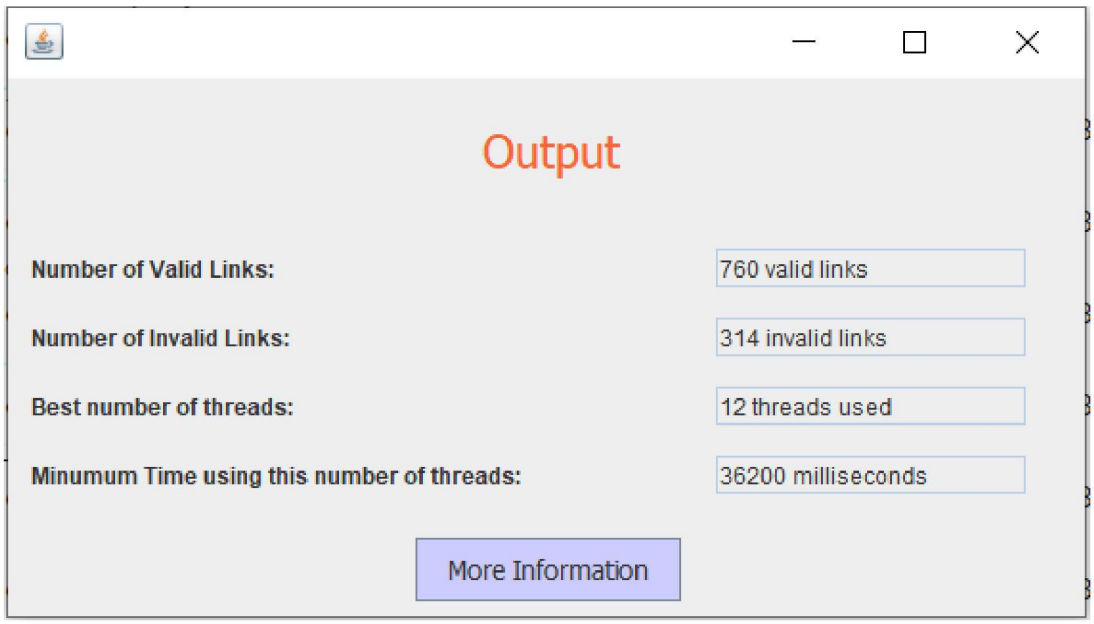

Threads Time Details
Number of Threads
Time in milliseconds
1
10193
2
5110
3
3693
4
3011
5
2662

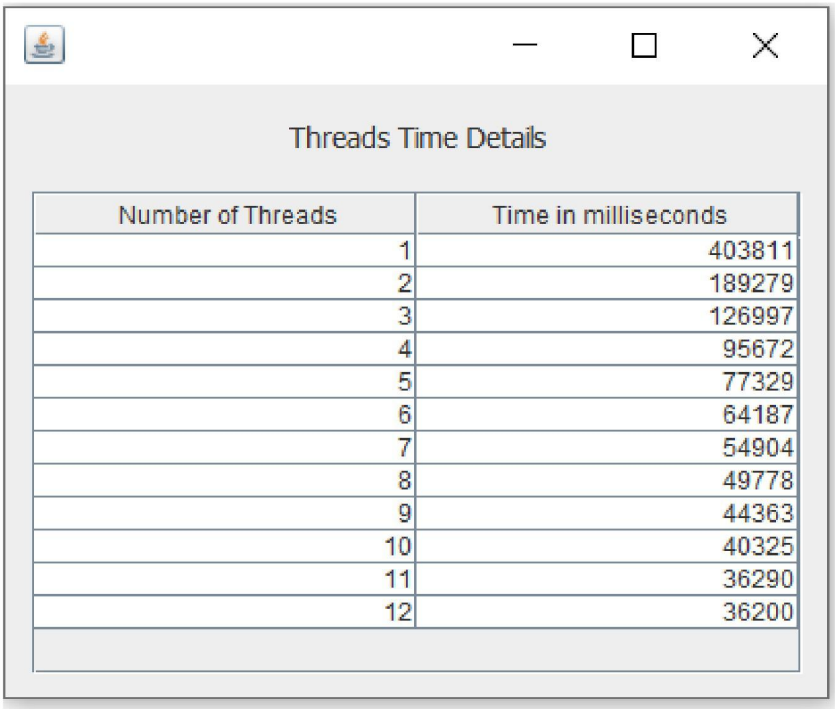

12000
10000
8000
6000
4000
2000
0
Time in milliseconds
1
2
3
4
5
Number of Threads

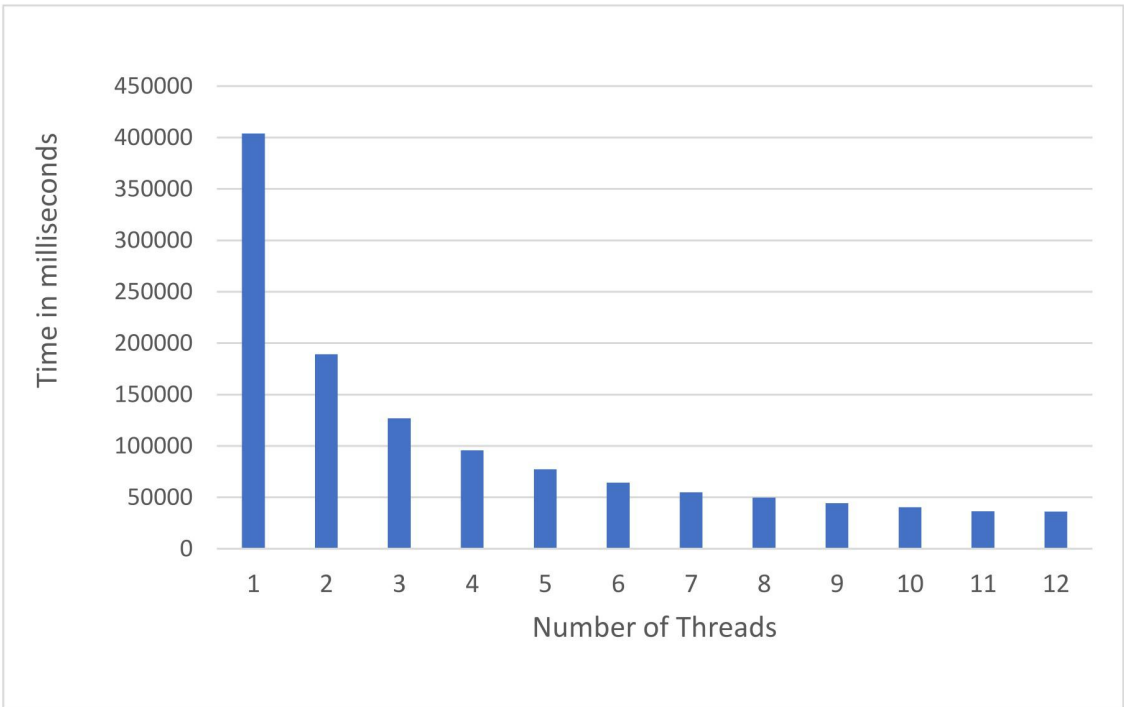
2- Link: https://www.google.com

Depth: 1



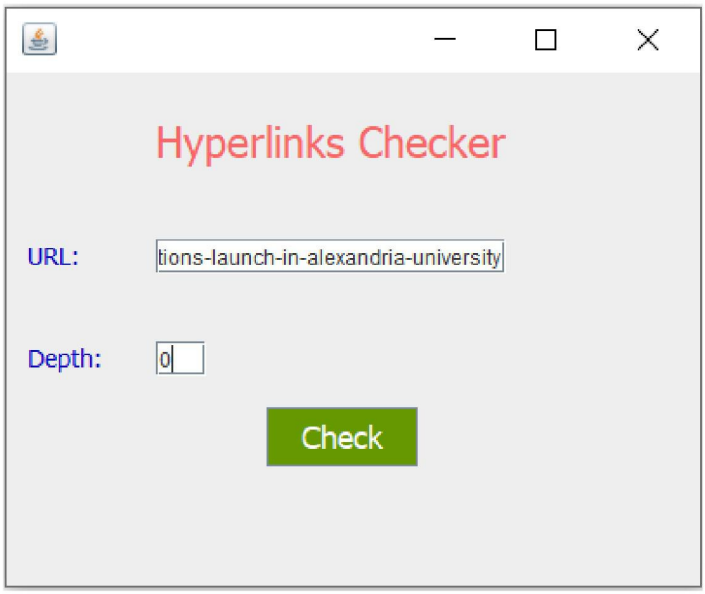


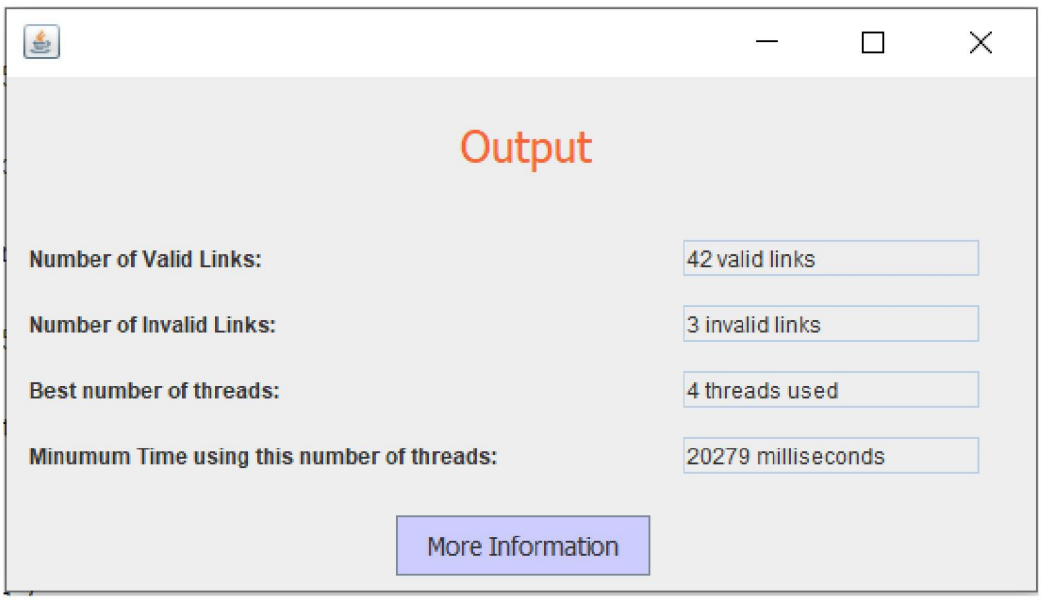

Threads Time Details
Number of Threads
Time in milliseconds
1 403811
2 189279
3 126997
4 95672
5 77329
6 64187
7 54904
8 49778
9 44363
10 40325
11 36290
12 36200

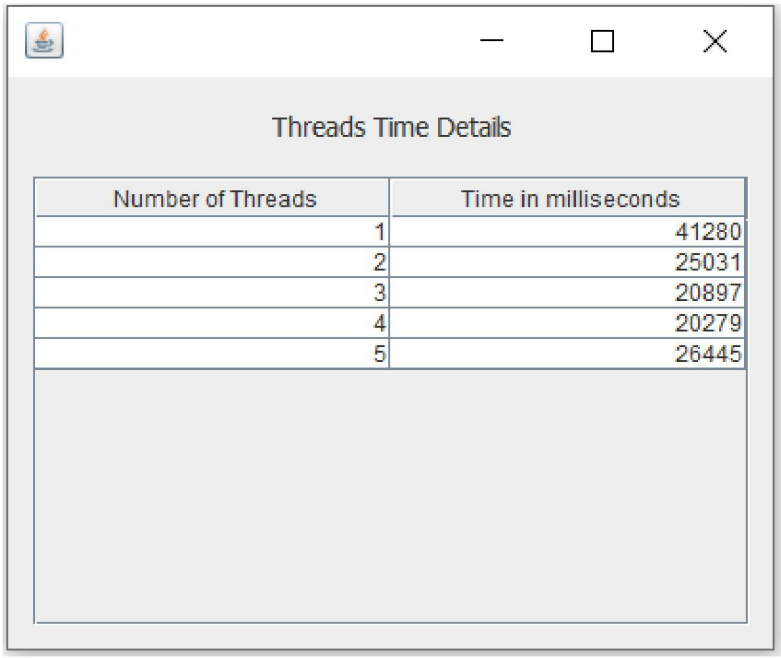

Time in milliseconds
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
1
2
3
4
5
6
7
8
9
10
11
12
Number of Threads

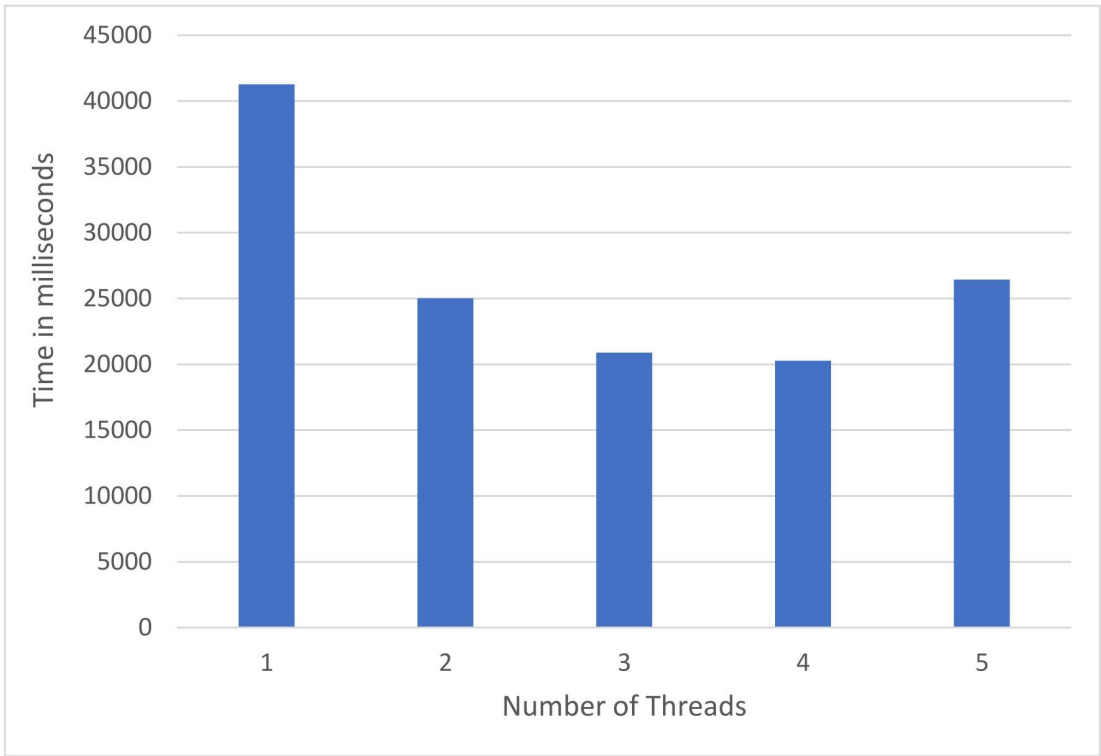
3- Link: https://www.alexu.edu.eg/index.php/en/discover-au/4025-diamond-jubilee-celebrations-launch-in-alexandria-university

Depth: 0



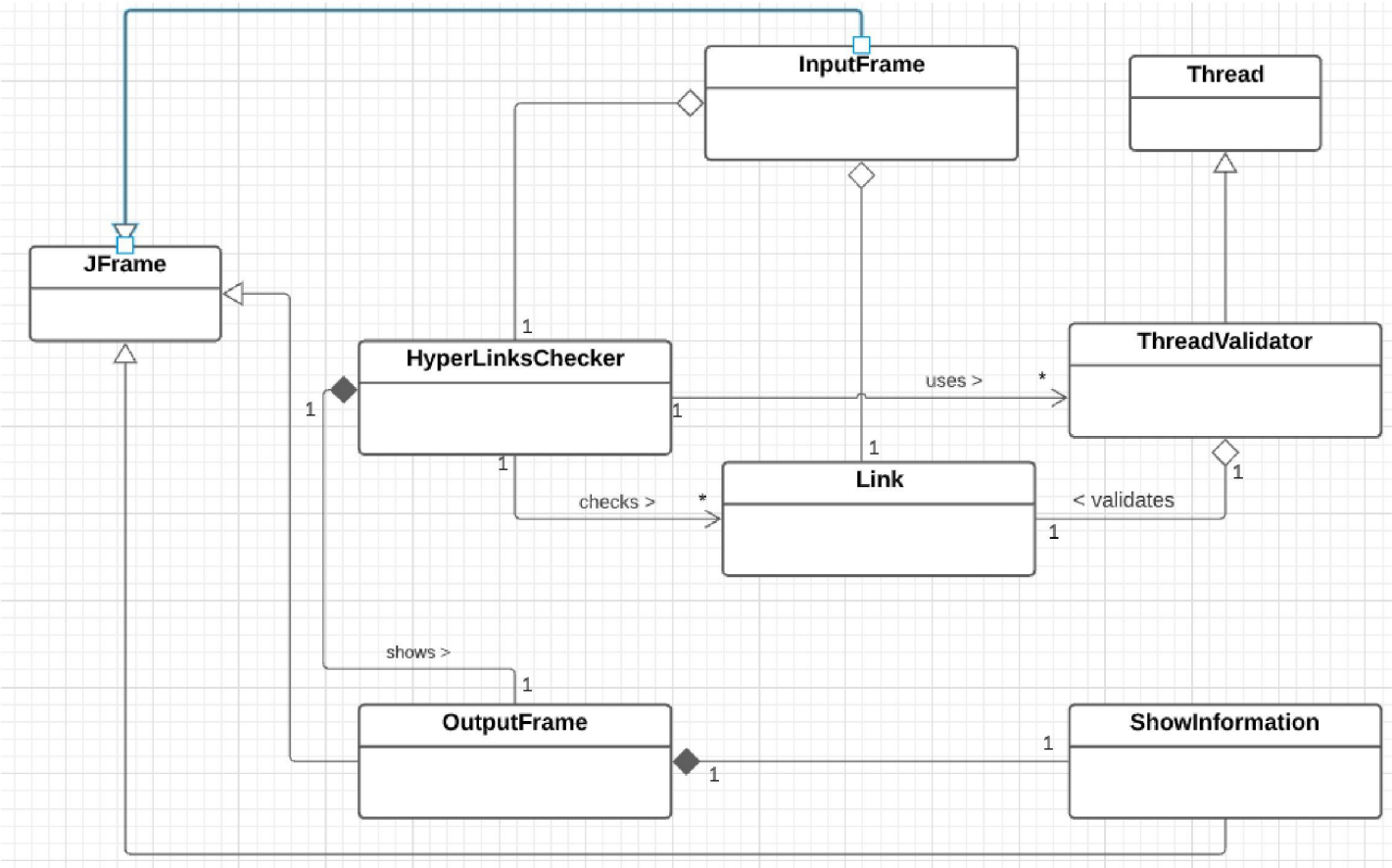



Threads Time Details
Number of Threads
Time in milliseconds
1
41280
2
25031
3
20897
4
20279
5
26445

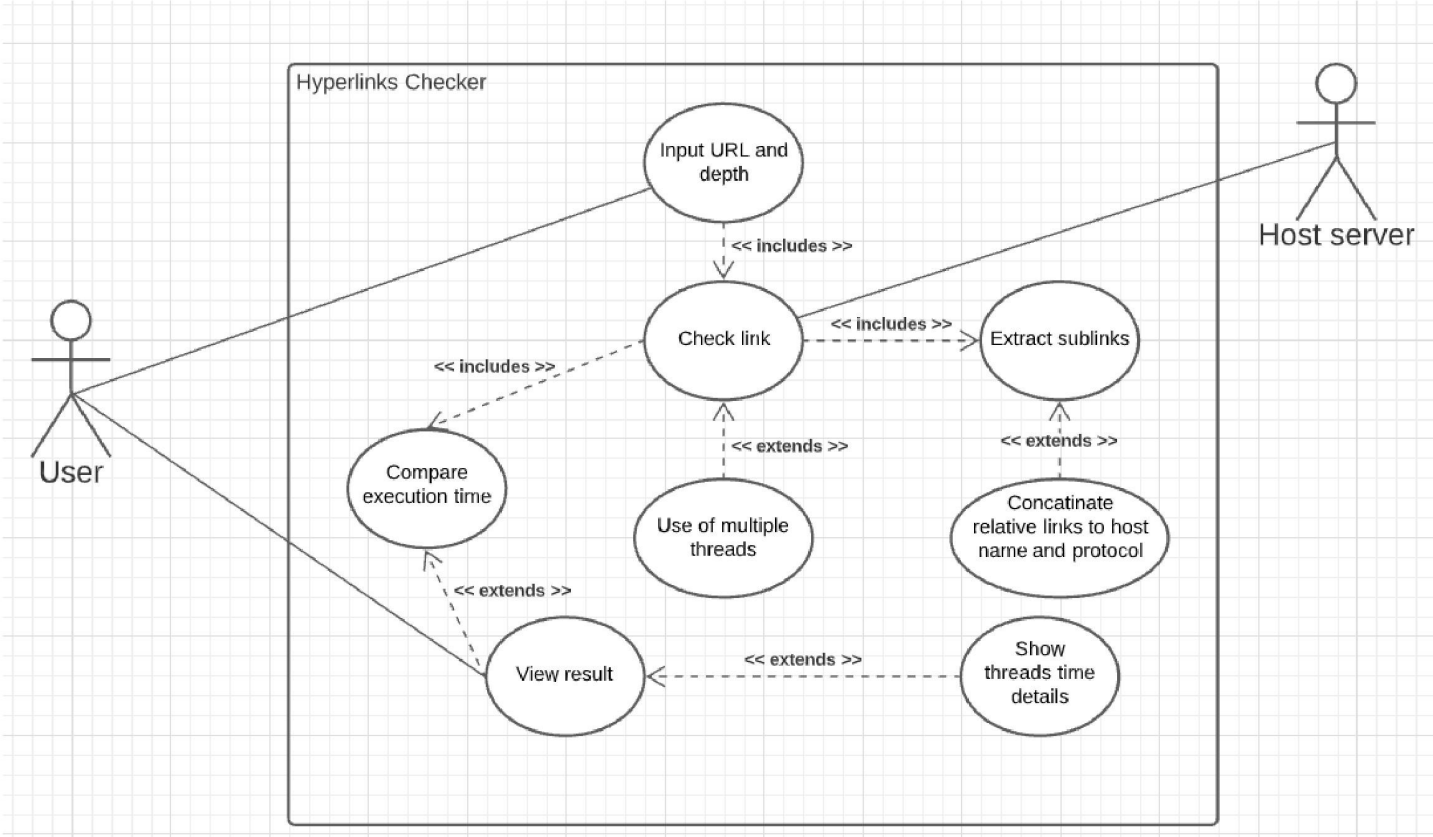

45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Time in milliseconds
1
2
3
4
5
Number of Threads

More UML Diagrams

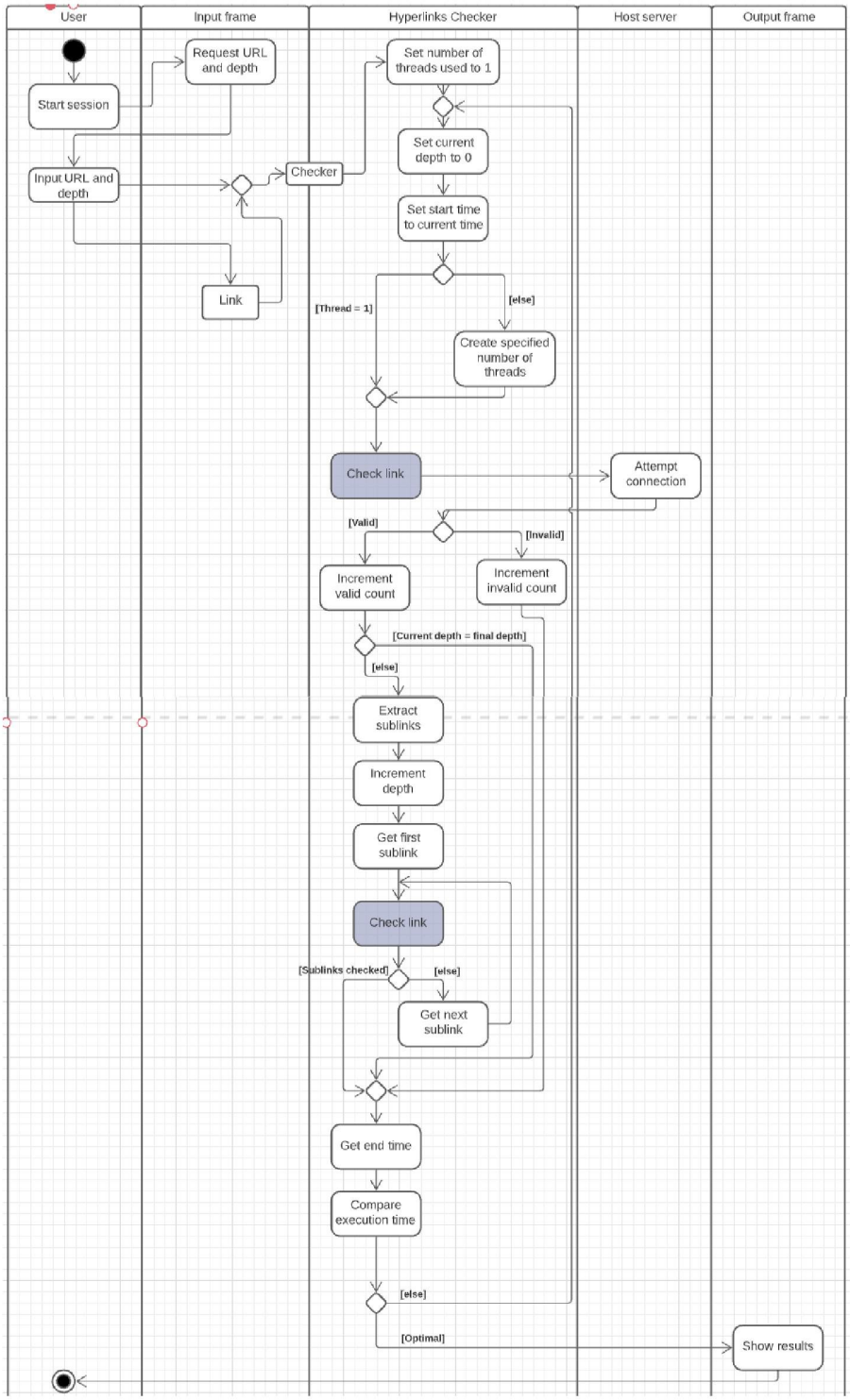
a) Class Diagrams (to show relationships between classes)



b) Use Case Diagram



c) Activity Diagram



d) Sequence Diagram