Project: Distributed WebCrawler
Report

Team Members:
Jyotheeswar(2019101099)
Trinadh(2019101043)
Sri Ram(2019101032)
Kalyan(2019101063)

Basic Idea:

Our aim in this project is to search through the internet to find the websites and form a directed graph out of them. Using this directed graph we need to run the PageRank algorithm on it. We will take a link and the depth as parameters to start with. Our algorithms will search through the given website for links in it. Say in a webpage a0 we found links l1, l2. In the graph we need to add directed edges from a0->l1, a0->l2. We will repeat the above process for each unexplored node (once a node is explored we will never run this process on that node again).

In this way, we will find the graph of depth given as a parameter. Now to run Page Rank is quite simple. The basic idea is we assume this graph is a Markov chain. Now we will find the stationary state of this Markov process. We will discuss in-depth about these in corresponding topics.

WebCrawler

The job of a web crawler is given a link to output all valid links which can be found from the website. We have used regex to differentiate between normal text and links. We have carefully handled the cases like self redirection, and the cases where there is href as #. We will search for the links which are there on the website using the above procedure and then iteratively repeat this for all the new links we found during this process. During the entire process, we will never re-explore the node which has already been visited. We have written functions when given nodes will output links that are there in the links with depth 1. One can run this function recursively to get the total idea of the graph.

```
def scrape_urls(page):
    try:
        reqs = requests.get(page)
        soup = BeautifulSoup(reqs.text, 'html.parser')
        urls = []
        for link in soup.find_all('a'):
            temp_url = link.get('href')
```

```
            if temp_url[0] == ".":
                temp_url = page + temp_url[1:]
            elif temp_url[0] == "#":
                temp_url = page
            if is_url(temp_url):
                urls.append(temp_url if temp_url[-1] ≠ "/"
else temp_url[:-1])
        return list(set(urls))
```

Map Reduce Framework responsible for web crawling

We have used the map-reduce framework for implementing distributed web crawlers. Now if we just implement the naive method for the above logic it will take a significant chunk of time. There are multiple methods for implementing this in a distributed environment. We have chosen map-reduce to solve this issue. We will send nodes in the format (node, flag) to each mapper, here node is the link, and flag=0 means the node is yet to be explored and flag=1 means the link is already explored.

- **Mapper:**

  In each mapper, we will receive multiple lines of the form (node, flag). For each unexplored node which is there in the current mapper, we will see the links which are there in the node/link. Let's say the links we found in the node are l1,l2. Now we will print edges (node,l1), (node, l2) to make the reducer know that these are edges. We will also print (node, 1) and (l1,0), (l2,0). This is because we have explored the node and are yet to explore l1, l2. So in mapper we are making sure the reduce the current status of the nodes, and some of the new edges we have explored.

- **Reducer:**

  In each reducer, we receive two types of line, one which will be of the form (node, flag) which talks about the exploration state of the nodes. And the other form of the line is (node, l1) which says that there is a directed graph between node->l1. Now we will use the fact that all the lines which will have the same key will be present in the same reducer. So we can explore the state of the node's status. Say there is (node, 0), (node,1) in the reducer, this means the node is explored. We will take the OR of all the flags for the same node and will update the status of the node before passing to the next iteration of the MapReduce.

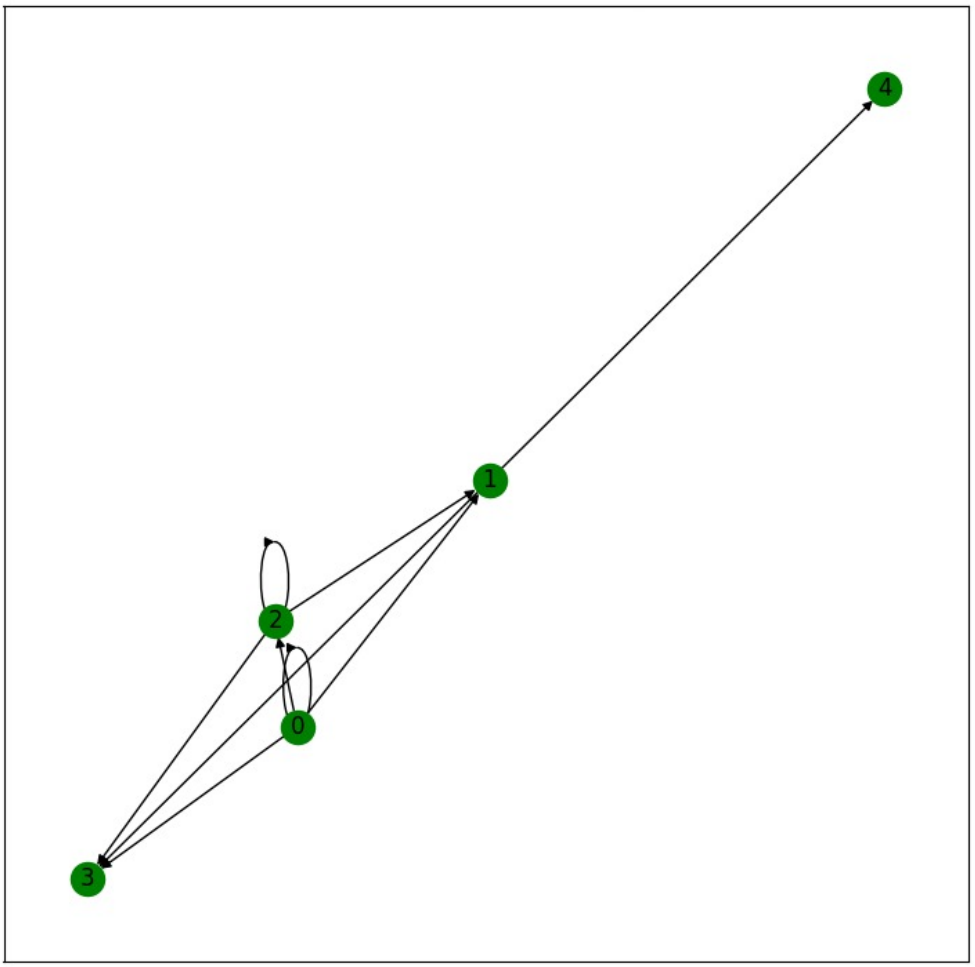
- **Other Mapper-Reducers:**

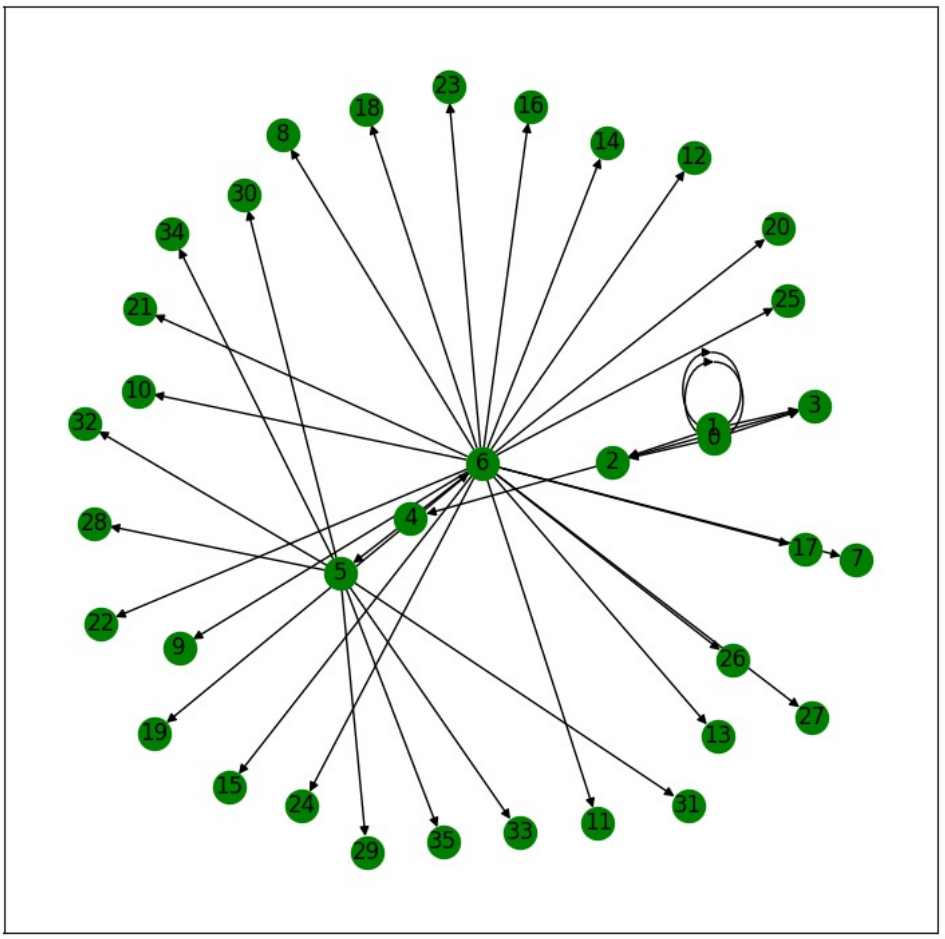
  Along with the above-mentioned map-reduce framework, we will have two more map-reduce frameworks running on the output of the first map-reduce. One of the map-reduce is responsible for separating out the edges which are there in the output. And the other map-reduce is responsible for preparing the input for the next map-reduce. In other words the job of the second map-reduce is to pass the updated statuses to the mappers.

Distributed Nature in files passed

One important aspect of the implementation is that we have not merged the files which are outputted in each iteration from the map-reduce. We have passed all the files which are generated from the previous command using input_directory/* command in -input parameter in the command. Using this we need not concatenate the files which defeats the whole purpose of distributed computation and the fact that edges can be huge in number makes it very difficult for us to store in the same file. We are storing each iteration output of the edges in different folders and we are maintaining the naming convention same across iterations.

Visualisation
These are a few graphs that we got when we ran the algorithm. Each of them have directed edges and each node is a link. We hashed links so that there is better representation for us to see.


4
1
2
0
3

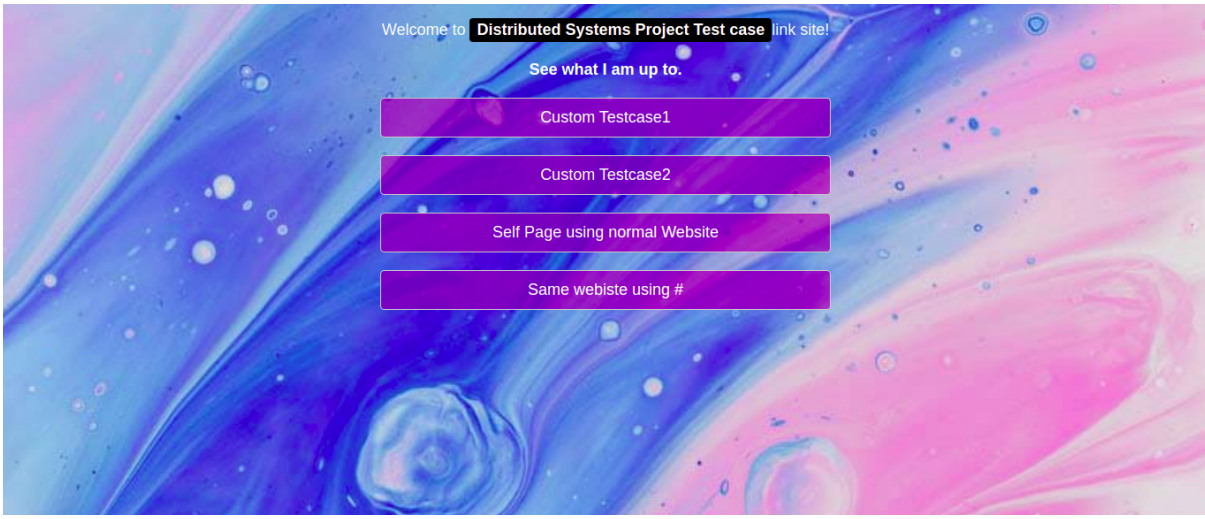


Testing

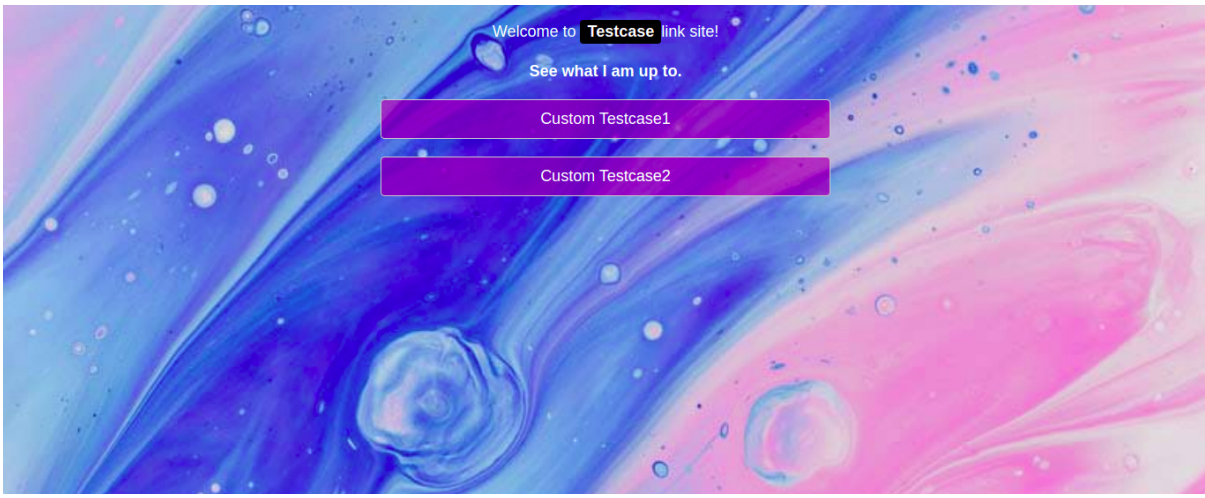
For testing, we have used the following two ways of testing our code.

- **Custom Testing**

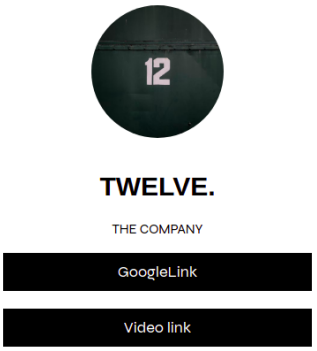
  We have prepared static websites in w3 schools with a few links attached to them. The links which are given are also the static websites that we have prepared (So that there is no dense network that we are getting for any random link that we chose). Finally for one of the nodes we have given google.com, some youtube video links so the graph will explode after that. And if we run for one more iteration then the graph will explode even more as there will be many links on those websites.



Basic starting point of test case



Custom test case in main screen

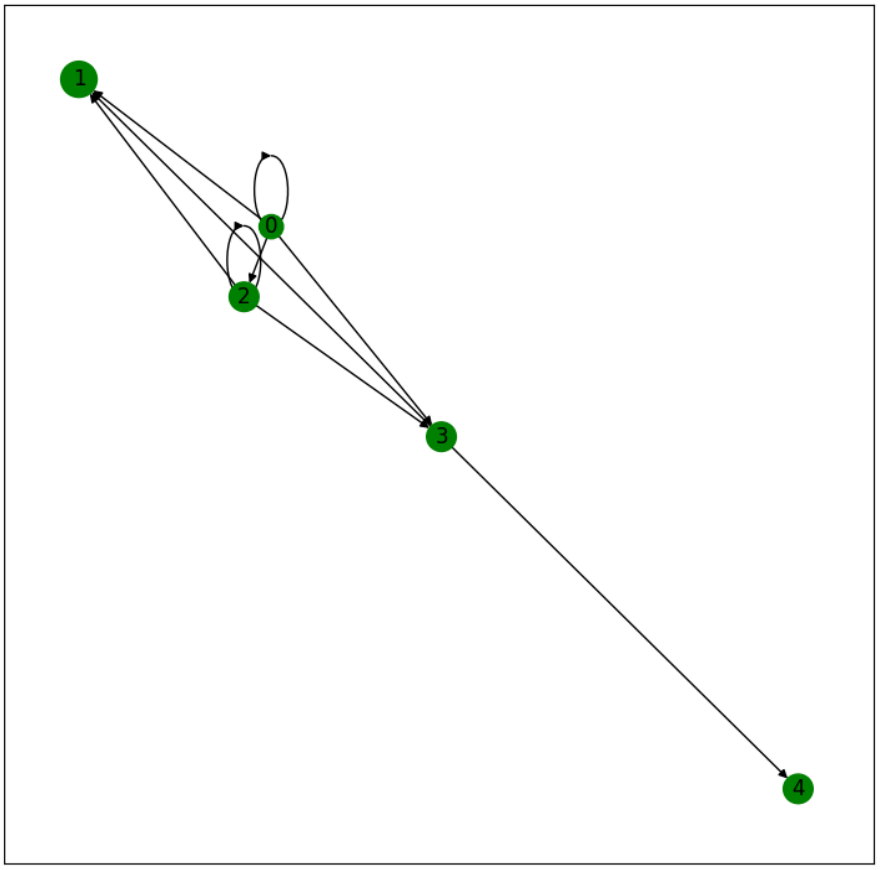


Screen where we have given external sites link

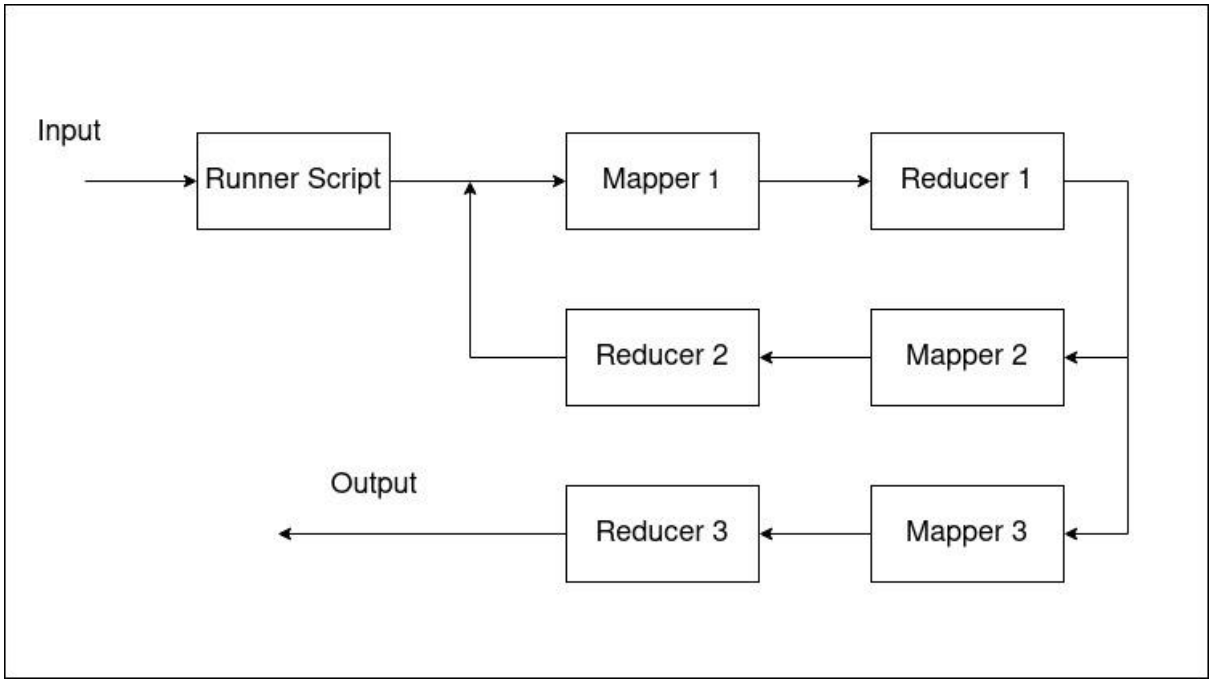
- **General Testing**
  We have taken some famous websites like GitHub, google, and youtube to check our algorithm. Although there is no definite way of judging the graph we feel our algorithm has explored all the edges which are possible. And explored the edges only once.

Page Rank Algorithm

Once we get the directed graph of the links we need to apply a page rank algorithm. Page rank algorithm is simply calculation of the stationary probabilities in a markov chain. While calculation of the page rank algorithm we assume the given directed graph as the markov chain. Now we can use the technique to find stationary probabilities on this. We will iteratively update the probabilities of each node. One thing which we should observe is that when there are dangling nodes we will assign outgoing edges with uniform distribution to all nodes or else the stationary state won't exist for this markov chain. We have also plotted the graph after page rank algorithm with node sizes representing how large the probabilities are.



Architecture:



Final outputs delivered

- Graph obtained from the web crawler.
- Page rank graph of the graph obtained.
- Edges of the graph
- Probabilities after running page rank algorithm