Team S (Link Analysis)

Team Logistics and Success Metrics

Asad Ul Haq: Liaised with other teams. Helped on implementation plan, DFD and unit tests. Wrote add_node unit test.

Kellie Ng: I helped with making the DFDs, system tests, and implementation plan. I also set up meeting times that would work for everyone, as well as made sure everyone was on track to get things done by a certain day.

Siyan Zuhayer: I talked with other teams to finalize the architectural divisions, helped make the DFDs and system tests, and made the success metrics.

Erika Ingersoll: I created the FDDs and the get_metadata and update_metadata unit tests.

Communication Methods:
We use Discord to communicate.

Where Our Project Work Will Be Stored:
We will be using GitHub to store our project work.
https://github.com/anotherAsad/LSPT_semester_project

Internal Success Metrics:

1. Readability / Understandability – Our code should be easy to understand by anyone. This metric will be met by following a consistent coding style standard and isn't something that can be measured.
2. Simplicity – Our code shouldn't be complex in order to make it easier to understand, maintain, and extend it. This metric will be measured by cyclomatic complexity and will be met by conducting code reviews where we can make suggestions on where to simplify our code to reduce complexity.
3. Testability – Our code must be structured in a way that makes it easy to test. This metric will be measured by measuring simplicity, as the simpler our code is, the easier it'll be to test.

External Success Metrics:

1. Efficiency – The link analysis component should be efficient so that getting ranking can quickly get up-to-date scores from us without being slowed down too much. This will be measured by timing functions of link analysis such as the PageRank algorithm and retrieving a PageRank score. The metric will be met if the average time it takes for our functions meets the times we want.

2. Scalability – The link analysis component should be efficient at all large scales. This will be measured by testing the efficiency of link analysis on very large webgraphs and the metric will be met if the average times of functions meets the times we want.
3. Correctness – The link analysis component should give accurate scores to ranking to ensure the correct ranking of results. This will be measured through testing and the metric will be met if our tests cover most cases and all of those tests are passed.

Architectural Divisions

UI/UX (webgraph team)

getGraph():

- ➢ Input - none
- ➢ Output - webgraph (immutable)
- ➢ Side effects - none
- ➢ Called by UI/UX (Webgraph) when the webgraph visualization needs to be made
    - ○ Probably need to keep this API for UI/UX and then make one that gives a subgraph for Ranking

getSubgraph():

- ➢ Input - url, depth
- ➢ Output - subgraph of webgraph with <depth> levels
- ➢ Side effects - none

Ranking

getScore():

- ➢ Input - url
- ➢ Output - PageRank score / -1 (if given a URL not in the webgraph)
- ➢ Side effects - none
- ➢ Called by Ranking when they need to get PageRank scores for urls relevant to their query

Crawling

updateLinkGraph():

- ➢ Inputs - url, outlinks
- ➢ Output - success/failure
- ➢ Side effects - add/remove outlinks for url to webgraph
    - ○ If outlink is found in inputs that wasn't in graph, add to graph
    - ○ If an edge for url isn't found in outlinks, remove from graph
    - ○ If an outlink is found in inputs that's already in graph, do nothing
        - ■ PageRank scores get updated in next update cycle (and time of update gets recorded)

- When crawler updates a page in doc data store, then doc data store calls this function so we can check if there are any new/removed outlinks

removeNode():

- Inputs - url pattern
- Output - success/failure
- Side effects - remove all nodes whose url matches the pattern and their associated edges from webgraph
- When crawler sees a new page in robots.txt, it tells doc data store to remove that page and then doc data store calls this function to tell us to remove it from the graph
    - When crawler crawls a new robots.txt, it'll insert it into DDS and DDS will send us all url patterns to remove from the graph

Evaluation

reportMetric():

- Inputs - json of components of number of added nodes that succeeded or failed, and how long the page rank update took
- Output - none
- Side effects - updates evaluation data for link analysis
- Called by Link Analysis

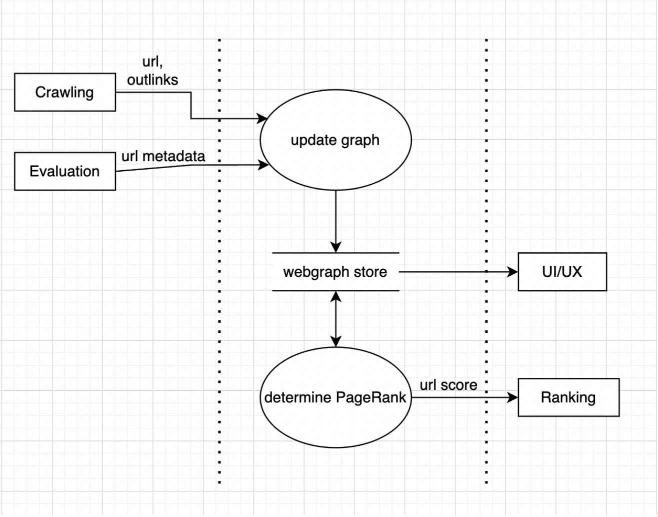
updateLinkGraph():

- Inputs - list of nodes with data about them, e.g., was it clicked/ignored, timestamp at which query happened
- Output - none
- Side effects - updates nodes in webgraph with the sent metadata
- Called by Evaluation
    - Reason why eval sends us this data is so that we can pass it off to UI/UX along with the webgraph

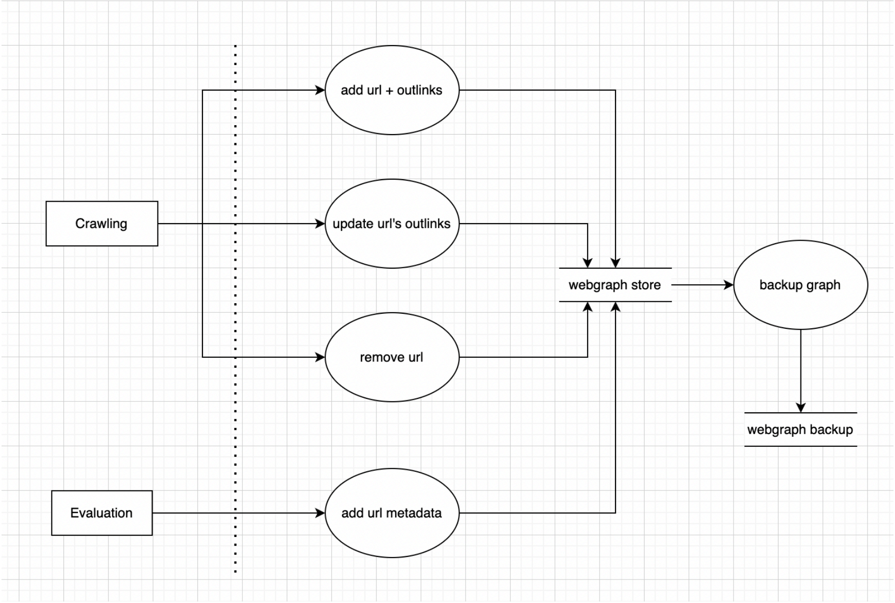
SE Design - DFDs and FDDs

DFDs:

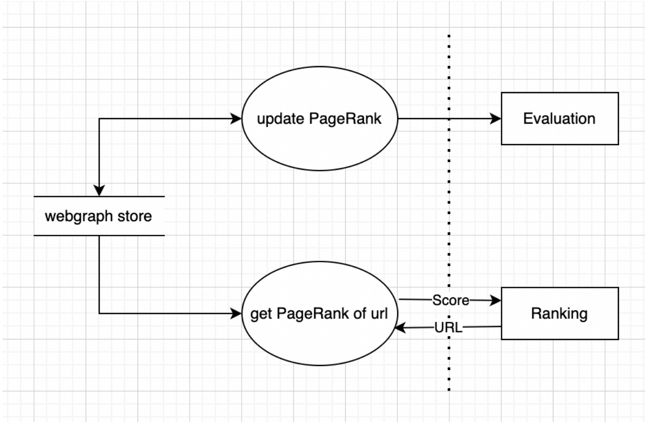
Level 0:



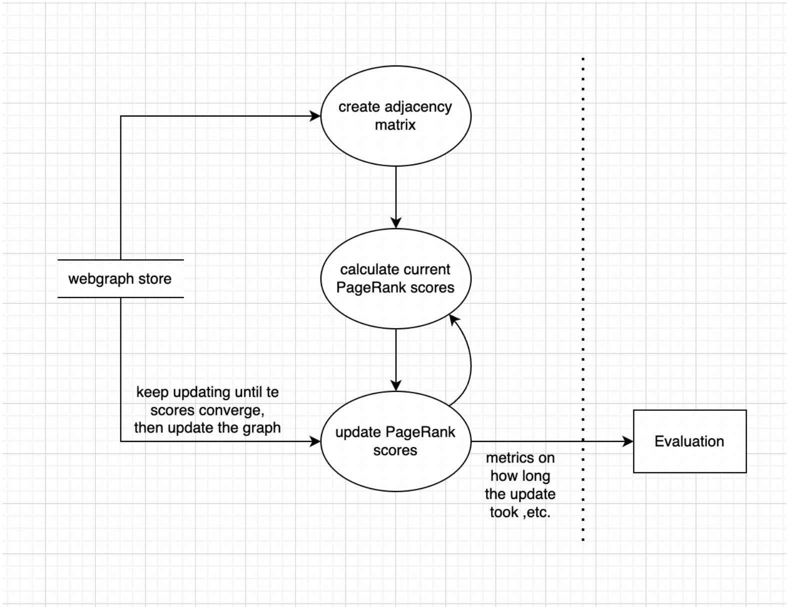
Level 1 - update graph:



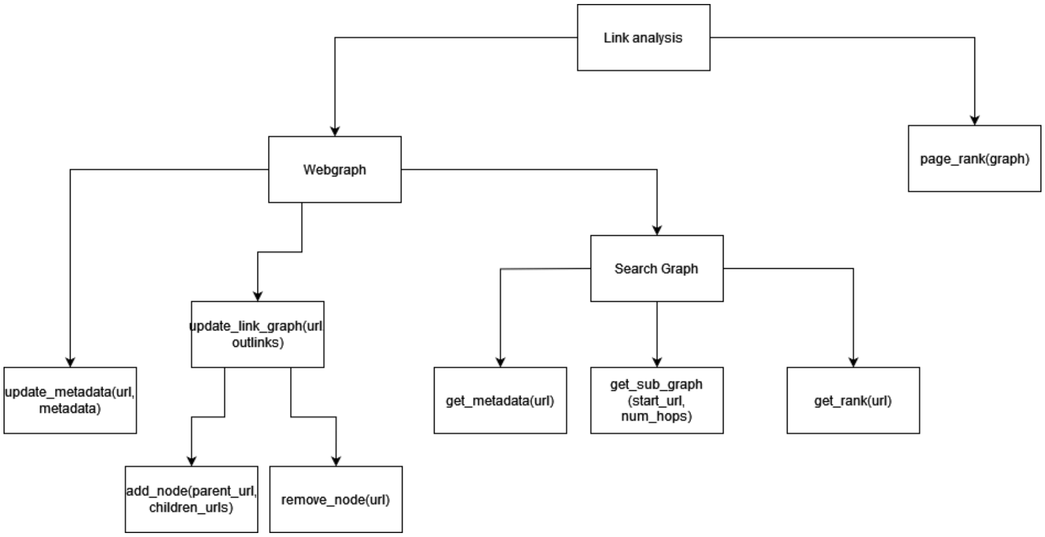
Level 1 - determine pagerank:



Level 2 - update pagerank:


create adjacency matrix
calculate current PageRank scores
webgraph store
keep updating until te scores converge, then update the graph
update PageRank scores
metrics on how long the update took ,etc.
Evaluation

FDD:


Link analysis
Webgraph
page_rank(graph)
Search Graph
update_link_graph(url outlinks)
update_metadata(url, metadata)
get_metadata(url)
get_sub_graph (start_url, num_hops)
get_rank(url)
add_node(parent_url, children_urls)
remove_node(url)

Design Reviews

Meeting Notes 11/17

Review #1:

- Reviewers: Aaron, William (Team V)
- What was reviewed:
  - Level 1 DFD
    - Does LA need an outlink towards Evaluation giving them data on how long the PageRank calculation took?
- What remained the same:
  - Level 0 DFD
  - Level 2 DFD
- What should change:
  - No notes

Review #2:

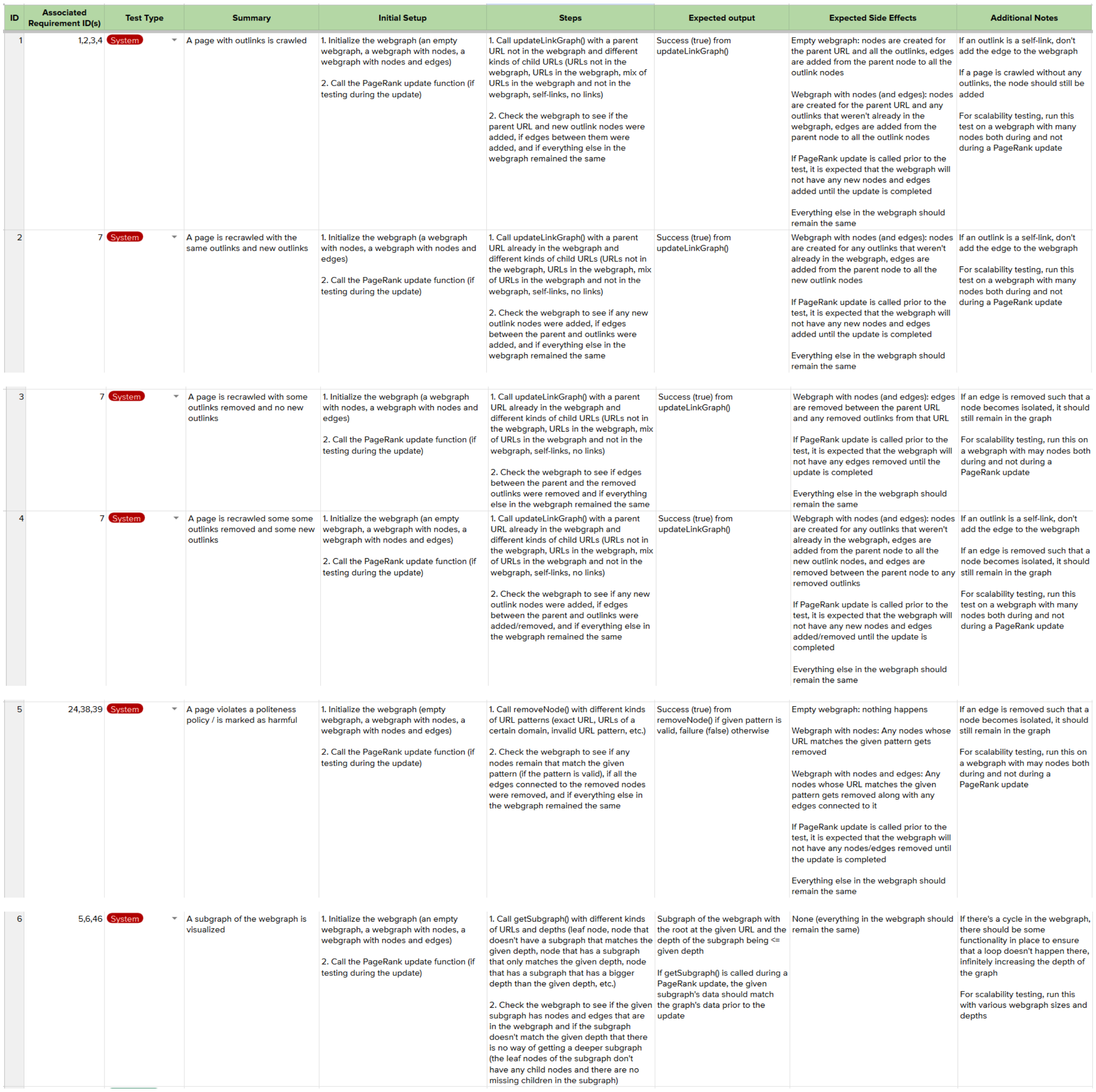
- Reviewers: Our Team
- What was reviewed:
  - What happens when the VM shuts down while the webgraph is getting updated
- What remained the same:
  - DFDs and FDD
- What should change:
  - We added a test case to check for what will happen when the VM shuts down during an update
  - It is expected that the graph just loads in the previous backup and then restarts the execution of updates that were happening at the time of the shutdown

System Tests

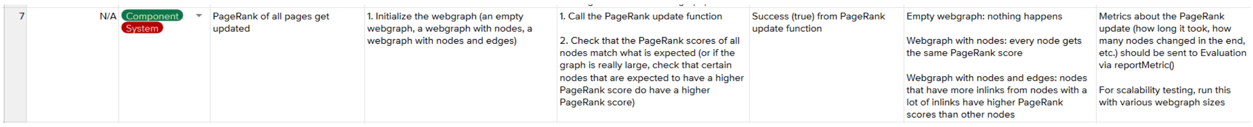
Spreadsheet (test cases are on the "system and component tests" sheet): https://docs.google.com/spreadsheets/d/1g8wKU98GcBYmHCal0Oaf_sJAYElerPe1b28pZhKwRaU/edit?usp=sharing

8 interesting system tests:

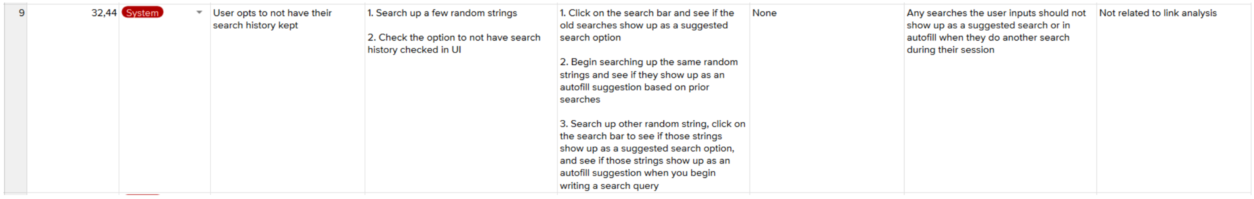
- 1, 2, 3, 4, 5, 6, 7 relate to scalability
- 5, 9 relate to social awareness



ID	Associated Requirement ID(s)	Test Type	Summary	Initial Setup	Steps	Expected output	Expected Side Effects	Additional Notes
1	1,2,3,4	System	A page with outlinks is crawled	1. Initialize the webgraph (an empty webgraph, a webgraph with nodes, a webgraph with nodes and edges) 2. Call the PageRank update function (if testing during the update)	1. Call updateLinkGraph() with a parent URL not in the webgraph and different kinds of child URLs (URLs not in the webgraph, URLs in the webgraph, mix of URLs in the webgraph and not in the webgraph, self-links, no links) 2. Check the webgraph to see if the parent URL and new outlink nodes were added, if edges between them were added, and if everything else in the webgraph remained the same	Success (true) from updateLinkGraph()	Empty webgraph: nodes are created for the parent URL and all the outlinks, edges are added from the parent node to all the outlink nodes Webgraph with nodes (and edges): nodes are created for the parent URL and any outlinks that weren't already in the webgraph, edges are added from the parent node to all the outlink nodes If PageRank update is called prior to the test, it is expected that the webgraph will not have any new nodes and edges added until the update is completed Everything else in the webgraph should remain the same	If an outlink is a self-link, don't add the edge to the webgraph If a page is crawled without any outlinks, the node should still be added For scalability testing, run this test on a webgraph with many nodes both during and not during a PageRank update
2	7	System	A page is recrawled with the same outlinks and new outlinks	1. Initialize the webgraph (a webgraph with nodes, a webgraph with nodes and edges) 2. Call the PageRank update function (if testing during the update)	1. Call updateLinkGraph() with a parent URL already in the webgraph and different kinds of child URLs (URLs not in the webgraph, URLs in the webgraph, mix of URLs in the webgraph and not in the webgraph, self-links, no links) 2. Check the webgraph to see if any new outlink nodes were added, if edges between the parent and outlinks were added, and if everything else in the webgraph remained the same	Success (true) from updateLinkGraph()	Webgraph with nodes (and edges): nodes are created for any outlinks that weren't already in the webgraph, edges are added from the parent node to all the new outlink nodes If PageRank update is called prior to the test, it is expected that the webgraph will not have any new nodes and edges added until the update is completed Everything else in the webgraph should remain the same	If an outlink is a self-link, don't add the edge to the webgraph For scalability testing, run this test on a webgraph with many nodes both during and not during a PageRank update
3	7	System	A page is recrawled with some outlinks removed and no new outlinks	1. Initialize the webgraph (a webgraph with nodes, a webgraph with nodes and edges) 2. Call the PageRank update function (if testing during the update)	1. Call updateLinkGraph() with a parent URL already in the webgraph and different kinds of child URLs (URLs not in the webgraph, URLs in the webgraph, mix of URLs in the webgraph and not in the webgraph, self-links, no links) 2. Check the webgraph to see if edges between the parent and the removed outlinks were removed and if everything else in the webgraph remained the same	Success (true) from updateLinkGraph()	Webgraph with nodes (and edges): edges are removed between the parent URL and any removed outlinks from that URL If PageRank update is called prior to the test, it is expected that the webgraph will not have any edges removed until the update is completed Everything else in the webgraph should remain the same	If an edge is removed such that a node becomes isolated, it should still remain in the graph For scalability testing, run this on a webgraph with may nodes both during and not during a PageRank update
4	7	System	A page is recrawled some some outlinks removed and some new outlinks	1. Initialize the webgraph (an empty webgraph, a webgraph with nodes, a webgraph with nodes and edges) 2. Call the PageRank update function (if testing during the update)	1. Call updateLinkGraph() with a parent URL already in the webgraph and different kinds of child URLs (URLs not in the webgraph, URLs in the webgraph, mix of URLs in the webgraph and not in the webgraph, self-links, no links) 2. Check the webgraph to see if any new outlink nodes were added, if edges between the parent and outlinks were added/removed, and if everything else in the webgraph remained the same	Success (true) from updateLinkGraph()	Webgraph with nodes (and edges): nodes are created for any outlinks that weren't already in the webgraph, edges are added from the parent node to all the new outlink nodes, and edges are removed between the parent node to any removed outlinks If PageRank update is called prior to the test, it is expected that the webgraph will not have any new nodes and edges added/removed until the update is completed Everything else in the webgraph should remain the same	If an outlink is a self-link, don't add the edge to the webgraph If an edge is removed such that a node becomes isolated, it should still remain in the graph For scalability testing, run this test on a webgraph with many nodes both during and not during a PageRank update
5	24,38,39	System	A page violates a politeness policy / is marked as harmful	1. Initialize the webgraph (empty webgraph, a webgraph with nodes, a webgraph with nodes and edges) 2. Call the PageRank update function (if testing during the update)	1. Call removeNode() with different kinds of URL patterns (exact URL, URLs of a certain domain, invalid URL pattern, etc.) 2. Check the webgraph to see if any nodes remain that match the given pattern (if the pattern is valid), if all the edges connected to the removed nodes were removed, and if everything else in the webgraph remained the same	Success (true) from removeNode() if given pattern is valid, failure (false) otherwise	Empty webgraph: nothing happens Webgraph with nodes: Any nodes whose URL matches the given pattern gets removed Webgraph with nodes and edges: Any nodes whose URL matches the given pattern gets removed along with any edges connected to it If PageRank update is called prior to the test, it is expected that the webgraph will not have any nodes/edges removed until the update is completed Everything else in the webgraph should remain the same	If an edge is removed such that a node becomes isolated, it should still remain in the graph For scalability testing, run this on a webgraph with may nodes both during and not during a PageRank update
6	5,6,46	System	A subgraph of the webgraph is visualized	1. Initialize the webgraph (an empty webgraph, a webgraph with nodes, a webgraph with nodes and edges) 2. Call the PageRank update function (if testing during the update)	1. Call getSubgraph() with different kinds of URLs and depths (leaf node, node that doesn't have a subgraph that matches the given depth, node that has a subgraph that only matches the given depth, node that has a subgraph that has a bigger depth than the given depth, etc.) 2. Check the webgraph to see if the given subgraph has nodes and edges that are in the webgraph and if the subgraph doesn't match the given depth that there is no way of getting a deeper subgraph (the leaf nodes of the subgraph don't have any child nodes and there are no missing children in the subgraph)	Subgraph of the webgraph with the root at the given URL and the depth of the subgraph being <= given depth If getSubgraph() is called during a PageRank update, the given subgraph's data should match the graph's data prior to the update	None (everything in the webgraph should remain the same)	If there's a cycle in the webgraph, there should be some functionality in place to ensure that a loop doesn't happen there, infinitely increasing the depth of the graph For scalability testing, run this with various webgraph sizes and depths



7	N/A	Component System	PageRank of all pages get updated	1. Initialize the webgraph (an empty webgraph, a webgraph with nodes, a webgraph with nodes and edges)	1. Call the PageRank update function 2. Check that the PageRank scores of all nodes match what is expected (or if the graph is really large, check that certain nodes that are expected to have a higher PageRank score do have a higher PageRank score)	Success (true) from PageRank update function	Empty webgraph: nothing happens Webgraph with nodes: every node gets the same PageRank score Webgraph with nodes and edges: nodes that have more inlinks from nodes with a lot of inlinks have higher PageRank scores than other nodes	Metrics about the PageRank update (how long it took, how many nodes changed in the end, etc.) should be sent to Evaluation via reportMetric() For scalability testing, run this with various webgraph sizes



9	32,44	System	User opts to not have their search history kept	1. Search up a few random strings 2. Check the option to not have search history checked in UI	1. Click on the search bar and see if the old searches show up as a suggested search option 2. Begin searching up the same random strings and see if they show up as an autofill suggestion based on prior searches 3. Search up other random string, click on the search bar to see if those strings show up as a suggested search option, and see if those strings show up as an autofill suggestion when you begin writing a search query	None	Any searches the user inputs should not show up as a suggested search or in autofill when they do another search during their session	Not related to link analysis

Datasets for testing:

- For generating webgraphs:
  - Randomly generate a set of URLs (none of which have any outlinks)
  - Randomly generate a set of URLs and assign a random number of outlinks to a random number of URLs
- For crawling interface testing:
  - Randomly generate a set of URLs such that none of them are in a webgraph
  - Randomly generate a set of URLs such that all of them are in a webgraph
  - Randomly generate a set of URLs such that some of them are in a webgraph
- For ranking interface testing:
  - Randomly select a set of URLs that are in the webgraph
  - Randomly generate a set of URLs that are not in the webgraph
- For UI/UX interface testing:
  - Randomly select a set of URLs that are in the webgraph
  - Randomly generate a set of URLs that are not in the webgraph
  - Randomly generate some depth values
- For evaluation interface testing:
  - Randomly select a set of URLs
  - For each selected URL, randomly generate the number of times it was and wasn't clicked

Implementation Plan

For our component, we decided to use Python. For Python, we decided to use it because it's easy to use. It has a graph library tool that was built in C++, so it is reliable for performance. It also has a built-in graph-tool method that we plan on using to build our web graph. Additionally, we plan on using snake_case for functions and variables, adhering to the Python naming conventions. We plan on using a default of four spaces/soft tabs. The libraries we will be using include JSON and the graph-tool library. We will also add detailed comments with fully fleshed out parameters, functionality, and return values, if any. We plan on implementing the PageRank algorithm in Python as well for consistency. We originally planned on putting it in C, but figured that keeping everything in one language would be more cohesive and pragmatic. We will also conduct code reviews when necessary, as well as keep our code on GitHub and commit consistently for version control history. For coherence and relevance, we have decided to stick with the pip 24.0 (python 3.12) and python 3.12.3, running on the standard CPython interpreter. We also require our team members to have graph-tool installed to make for easier code revisions and building.

Unit Tests

Unit tests:

1. test_add_node
2. test_get_metadata
3. test_update_metadata

1. test_add_node()

   Tests the `add_node(url, child_list)` function comprehensively.

   Where, `add_node()` looks as follows:

```
# INPUT: (1) url: a string that corresponds to a node to be
# added/already inside the graph. (2) list_of_outlinks is a list of
# strings that are outlink urls from the original url.
#
# OUTPUT: true if the nodes are added. false if the nodes are not added.
#
# DESCRIPTION: (1) looks for a node that corresponds to the url. If the
# node is not found, it is created. (2) looks through the
# list_of_outlinks, and adds/creates these nodes just like step 1.
# (3) Updates directed edges info from url to outlinks in the graph.
# (4) Returns true if the operation succeeded, false if any of the urls
# is invalid.
def add_node(url, list_of_outlinks):
    ...
    return success_status
```

Below is a list of tests for the `add_node()` function.

```
# Test 1: Add new url and new list_of_outlinks.
#     Preconditions: The webgraph is instantiated. Has no nodes.
#     Returns: success
print("returned: " + str(add_node("www.google.com", [
    “www.wikipedia.com”,
    “www.cnn.com”
]))


# Test 2: Add a url that is already in the graph, with new outlinks
#     Preconditions: The webgraph is instantiated, has node with url
#     “www.google.com”, but doesn’t have “www.rpi.edu”, “www.rpi.com”
#     Returns: success
print("returned: " + str(add_node("www.google.com", [
    “www.rpi.edu”,
    “www.rpi.com”
]))

# Test 3: Add a url that is not in the graph, but has outlinks that are
# in the graph.
#     Preconditions: The webgraph is instantiated, does not have node with
#     url “www.boogle.com”, but has “www.rpi.edu”, “www.rpi.com”
#     Returns: success
print("returned: " + str(add_node("www.boogle.com", [
    “www.rpi.edu”,
    “www.rpi.com”
]))

# Test 4: Add a url that is in the graph, with outlinks that are already
# in the graph.
#     Preconditions: The webgraph is instantiated, has nodes with
#     urls “www.boogle.com”
#     Returns: success
print("returned: " + str(add_node("www.boogle.com", [
    “www.rpi.edu”,
    “www.rpi.com”
]))

# Test 5: Add a url that is in the graph, with a mix of outlinks; (some new, #
some old)
#     Preconditions: The webgraph is instantiated, has nodes with
#     urls “www.boogle.com”, “www.rpi.edu”. Doesn’t have “www.cnn.com”
#     Returns: success
print("returned: " + str(add_node("www.boogle.com", [
    “www.rpi.edu”,
    “www.rpi.com”,
    “www.cnn.com”
]))

# Test 6: Test with non-string argument for url, but valid list_of_outlinks
#     Preconditions: The webgraph is instantiated.
#     Returns: failure
print("returned: " + str(add_node(1234, [
    “www.rpi.edu”,
    “www.rpi.com”,
    “www.cnn.com”
]))

# Test 7: Test with valid string url, and non-list argument for
# list_of_outlinks
#     Preconditions: The webgraph is instantiated.
#     Returns: failure
print("returned: " + str(add_node("www.boogle.com", 1234))


# Test 8: Test with not-string and non-list argument for url and
# list_of_outlinks respectively
#     Preconditions: The webgraph is instantiated.
#     Returns: failure
print("returned: " + str(add_node([], 1234))

# Test 9: Test with valid url, and empty list_of_strings
#     Preconditions: The webgraph is instantiated.
#     Returns: success
print("returned: " + str(add_node("www.boogle.com", []))

# Test 10: Test with valid url, and a list_of_outlinks, some of whose
# arguments are not strings.
#     Preconditions: The webgraph is instantiated.
#     Returns: failure
print("returned: " + str(add_node("www.boogle.com", [
    “www.cnn.com”
    1234
]))
```

2. test_get_metadata

```
# INPUT: (1) url: a string that corresponds to a node to be
# added/already inside the graph.
#
# OUTPUT: the metadata contained in the node, in JSON.
#
# DESCRIPTION: (1) looks for the node corresponding to the url given
#              (2) fetches metadata contained by the node
#              (3) returns metadata in JSON format

def get_metadata(url):
    ...
    return metadata
```

Below is a list of tests for the `get_metadata()` function.

```
#TEST 1: get metadata from an existing node that has metadata
#     Preconditions: node "www.google.com" exists, has metadata, and is
#                    connected to other nodes
#     Returns: metadata in JSON format
print("metadata: " + str(get_metadata("www.google.com"))

#TEST 2: get metadata from an existing node that has metadata and is not
#     connected to other nodes
#     Preconditions: node "www.google.com" exists, has metadata, and is not #
connected to other nodes
#     Returns: metadata in JSON format
print("metadata: " + str(get_metadata("www.rpi.edu"))


#TEST 3: get metadata from an existing node that does not have metadata and #
is connected to other nodes
#     Preconditions: node "www.google.com" exists, does not have metadata,
#                    and is connected to other nodes
#     Returns: empty data
print("metadata: " + str(get_metadata("www.google.com"))

#TEST 4: get metadata from an existing node that does not have metadata and is
not connected to other nodes
#     Preconditions: node "www.google.com" exists, does not have metadata,
#                    and is not connected to other nodes
#     Returns: empty data
print("metadata: " + str(get_metadata("www.google.com"))
#TEST 5: get metadata from a node that doesn't exist
#     Preconditions: node "www.google.com" does not exist
#     Returns: None
print("metadata: " + str(get_metadata("www.google.com"))

#TEST 6: get metadata from a node that doesn't exist while page_rank is
#        running
#     Preconditions: node "www.google.com" does not exist, page_rank is
#                    running
#     Returns: None
print("metadata: " + str(get_metadata("www.google.com"))

#TEST 7: get metadata from a node that exists while page_rank is running
#     Preconditions: node "www.google.com" exists, page_rank is running
#     Returns: empty data
print("metadata: " + str(get_metadata("www.google.com"))
```

3. test_update_metadata

```
# INPUT: (1) url: a string that corresponds to a node in the graph.
#        (2) metadata to be added, in JSON format
#
# OUTPUT: true if the metadata is updated. false if the metadata is not
#         updated.
#
# DESCRIPTION: (1) looks for a node that corresponds to the url.
#              (2) if metadata already exists, replaces the field indicated #
in the input with new value (or creates new field if the
#                  field doesn't exist). Leaves the other fields as is.
#              (3) if metadata does not exist, creates metadata
#              (4) if metadata is updated successfully, returns true.
#                  Otherwise, returns false.

def update_metadata(url, metadata):
    ...
    return success_status
```

Below is a list of tests for the `update_metadata()` function.

```
# TEST 1: Add valid metadata to a node that exists
#     Preconditions: node "www.google.com" exists. metadata is valid JSON.
#     Returns: success
print("metadata: " + str(update_metadata("www.google.com", metadata))

# TEST 2: Add valid metadata to a node that does not exist
#     Preconditions: node "www.boogle.com" does not exist.
#     metadata is valid JSON.
#     Returns: failure
print("metadata: " + str(update_metadata("www.boogle.com", metadata))
# TEST 3: Add invalid metadata (non JSON string) to a node that exists
#     Preconditions: node "www.boogle.com" exists.
#     metadata is invalid JSON.
#     Returns: failure
print("metadata: " + str(update_metadata("www.boogle.com", metadata))

# TEST 4: Add invalid metadata to a node that does not exist
#     Preconditions: node "www.boogle.com" does not exist.
#     metadata is invalid JSON.
#     Returns: failure
print("metadata: " + str(update_metadata("www.boogle.com", metadata))

# TEST 5: Add valid metadata to a non-string url
#     Preconditions: an instantiated graph.
#     metadata is valid JSON.
#     Returns: failure
print("metadata: " + str(update_metadata(1234, metadata))

#TEST 6: update metadata in a node that exists and does not already have
#        metadata while page_rank is running
#     Preconditions: node "www.google.com" exists, has no metadata.
#                    page_rank is running
#     Modifies: nothing
#     Returns: false
print("update_metadata: " + str(update_metadata("www.google.com", data))

#TEST 7: update metadata in a node that exists and already has metadata
#        while page_rank is running
#     Preconditions: node "www.google.com" exists, has metadata with field
#                    to be changed. page_rank is running.
#     Modifies: nothing
#     Returns: false
print("update_metadata: " + str(update_metadata("www.google.com", data))
```

Resources

https://submitty.cs.rpi.edu/courses/f24/csci4460/course_material/lectures/csci4460-f24-week-03-search.pdf
https://www.cs.cornell.edu/home/kleinber/networks-book/networks-book-ch14.pdf
https://en.wikipedia.org/wiki/PageRank