Change Log

We may make minor changes to the spec to address/clarify some outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the online forum discussion and the change log regularly.

Version 1.0 (2024-03-15 17:00)	• Initial release.

Background

As we have mentioned in lectures, the Internet can be thought of as a graph (a very large graph). Web pages represent vertices and hyperlinks represent directed edges.

With almost 1.1 billion unique websites (as of February 2024), and each website having multiple webpages, and each webpage having multiple hyperlinks, it can understandably be a very difficult job to remember the URL of every website you want to visit.

In order to make life easier, from the very early days of the internet, there have been search engines that can be used to find websites.

But the job of a search engine is very difficult: First it must index (create a representation of) the entire (or as close to it as possible) World Wide Web. Next it must rank the webpages it finds.

In this assignment we will be implementing algorithms to solve each of these problems, and figure out the fastest way to navigate from one page to another.

1. To index the internet we will be creating a web crawler.
2. To rank webpages we will implement the PageRank algorithm.
3. To find the shortest path between two pages we will implement Dijkstra's algorithm

The Assignment

Starter Files

Download this zip file.

Unzipping the file will create a directory called 'assn' with all the assignment start-up files.

Alternatively, you can achieve the same thing from a terminal with commands such as:

```
curl https://www.cse.unsw.edu.au/~cs9024/24T1/assn/assn.zip -o assn.zip
unzip assn.zip -d assn
```

The first command will download *assn.zip* into the current working directory, then the second command will extract it into a sub-directory *assn*.

You can also make note of the following URLs:

- `http://www.cse.unsw.edu.au/~cs9024/micro-web`
- `http://www.cse.unsw.edu.au/~cs9024/mini-web`

Here is a visual representation of the *micro-web*:

Once you read the assignment specification, hopefully it will be clear to you how these URLs might be useful. You may also find it useful to construct a similar visual representation for the *mini–web*.

Overall File Structure

Below is a reference for each file and their purpose.

Note: You cannot modify ANY of the header (`.h`) files.

Provided File	Description	Implemented In
`crawler.c`	A driver program to crawl the web	—
`dijkstra.h`	Interface for the Shortest Path functions (Subset 4)	`graph.c`
`graph.h`	Interface for the Graph ADT (Subset 1b)	`graph.c`
`list.h`	Interface for the List ADT (Subset 1a)	`list.c`
`Makefile`	A build script to compile the `crawler` into an executable	—
`pagerank.h`	Interface for the PageRank functions (Subset 3)	`graph.c`

Your task will be to provide the necessary implementations to complete this project.

Subset 1 – Dependencies

Before we can start crawling we need to be able to store our crawled data. As the internet is a Graph, this means we need a Graph ADT. We will also need a Set ADT and one of a Queue ADT or a Stack ADT, in order to perform web scraping (for a BFS or DFS).

Subset 1a – Implement the List (Queue, Stack, Set) ADT

You have been provided with a file `list.h`. Examine the file carefully. It provides the interface for an ADT that will provide Queue, Stack, and Set functionality.

Your task is to implement the functions prototyped in the `list.h` header file within `list.c`.

- You **must** create the file `list.c` to implement this ADT.
- You **must** store *string* (`char *`) data within the ADT.
- You **must** allocate memory dynamically.
- You **must not** modify the `list.h` file.
- You **must not** modify the function prototypes declared in the `list.h` file.
- You **may** add utility functions to the `list.c` file.
- You **may** use the `string.h` library, and other standard libraries from the weekly exercises.
- You **may** reuse code previously submitted for weekly assessments and provided in the lectures.
- You **may** use whatever internal representation you like for your list ADT, provided it does not contradict any of the above.
- You **may** assume that any instance of your list ADT will only be used as a queue *or* a stack *or* a set.
- You **should** write programs that use your ADT to test and debug your code.
- You **should** use `valgrind` to verify that your ADT does not leak memory.

As a reminder:

- Queue – First In, First Out
- Stack – First In, Last Out
- Set – Only stores **unique** values.

See `list.h` for more information about each function that you are required to implement.

Testing

We have created a script to automatically test your list ADT. It expects to find `list.c` in the current working directory. Limited test cases are provided, so you should always do your own, more thorough, testing.

```
9024 dryrun assn_list
```

Subset 1b – Implement the Graph ADT

You have been provided with a file `graph.h`. Examine the file carefully. It provides the interface for an ADT that will provide Graph functionality. The graph is both weighted and directed.

Your task is to implement the functions prototyped in the `graph.h` header file within `graph.c`.

- You **must** create the file `graph.c` to implement this ADT.
- You **must** use an adjacency list representation, but the exact representation is up to you.
- You **must** use *string* (`char *`) data to label the vertices.
- You **must** allocate memory dynamically.
- You **must not** modify the `graph.h` file.
- You **must not** modify the function prototypes declared in the `graph.h` file.
- You **may** add utility functions to the `graph.c` file.
- You **may** use the `string.h` library, and other standard libraries from the weekly exercises.
- You **may** reuse code previously submitted for weekly assessments and provided in the lectures.
- You **should** write programs that use your ADT to test and debug your code.
- You **should** use `valgrind` to verify that your ADT does not leak memory.

See `graph.h` for more information about each function that you are required to implement.

Subset 2 – Web Crawler

We are now going to use the `list` and `graph` ADTs you have created to implement a web crawler.

Assuming your ADTs are implemented correctly, you should be able to compile the crawler using the provided build script:

```
make crawler
```

Note: `crawler.c` requires external dependencies (`libcurl` and `libxml2`). The provided `Makefile` will work on CSE servers (ssh and vlab), but may not work on your home computer.

After running the executable, check that the output aligns with the navigation of the sample website.

Carefully examine the code in `crawler.c`. Uncomment the block of code that uses `scanf` to take user input for the `ignore_list`.

The ignore list represents the URLs that we would like to completely ignore when we are calculating PageRanks, as if they did not exist in the graph. This means that any incoming and outgoing links from these URLs are treated as non-existent. You are required to implement this functionality locally – within the `graph_show` function – and NOT change the representation of the actual graph strcuture within the ADT. For further details see the `graph.h` file.

If you have correctly implemented the ADTs from the previous tasks, this part should be mostly free.

`crawler.c` is a complete implementation of a web crawler; you do not need to modify the utility functions, only the bottom part of the `main` function. However, you should look at the program carefully and understand it well so that you can use it (i.e., modify it appropriately) for later tasks.

Sample Output

Using a modified `crawler.c` that simply calls `graph_show` on the *micro-web*, and without ignoring any pages, the output should be:

```
./crawler http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter a page to ignore or type 'done': done
http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html
http://www.cse.unsw.edu.au/~cs9024/micro-web/Y.html
http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html
http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html 1
http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html http://www.cse.unsw.edu.au/~cs9024/micro-web/Y.html 1
```

```
http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html 1
http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html 1
http://www.cse.unsw.edu.au/~cs9024/micro-web/Y.html http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html 1
```

Now let's add *index.html* to the ignore list:

```
./crawler http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter a page to ignore or type 'done': http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter another page to ignore or type 'done': done
http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html
http://www.cse.unsw.edu.au/~cs9024/micro-web/Y.html
http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html
```

All traces of *index.html* have been removed. This means that only the remaining vertices are displayed as there are no longer any edges. Note that the order of the output matters. It should follow the BFS that is performed by the crawler. If your result does not follow this order, you will be marked as incorrect, even if your graph is valid.

Testing

We have created a script to automatically test your `list` and `graph` ADTs. It expects to find `list.c` and `graph.c` in the current working directory. Limited test cases are provided, so you should always do your own, more thorough, testing.

```
9024 dryrun assn_crawler
```

Subset 3 – PageRank

Background

Now that we can crawl a web and build a graph, we need a way to determine which pages (i.e. vertices) in our web are important.

We haven't kept page content so the only metric we can use to determine the importance of a page is to check how much other pages rely on its existence. That is, how easy is it to follow a sequence of one or more links (edges) and end up on the page.

In 1998, Larry Page and Sergey Brin (a.k.a. Google), created the [PageRank] algorithm to evaluate this metric.

Google still uses the PageRank algorithm to score every page it indexes on the internet to help order its search results.

Task

In `graph.c` implement the two new functions `graph_pagerank` and `graph_show_pagerank`.

First, `graph_pagerank` should calculate and store the PageRank of each vertex (i.e. page) in the graph.

The algorithm must exclude the URLs that are provided in an 'ignore list' to the function. Do not remove the pages from the graph, only **skip** (i.e. ignore) them from calculations. This means that you will need to understand which parts of the PageRank algorithm need to be modified.

Using the ignore list, you will be able to see what happens to the PageRanks as certain pages are removed. What should happen to the PageRank of a particular page if you remove all pages linking to it?

Second, `graph_show_pagerank` should print the PageRank of every vertex (i.e. page) in the graph that is NOT in the ignore list.

Pages (vertices) should be printed from highest to lowest rank, based on their rounded (to 3 d.p.) rank. You should use the `round` function from the `math.h` library. If two pages have the same rounded rank then they should be printed lexiographically.

- You **may** add more utility functions to `graph.c`.
- You **may** (and most likely will need to) modify your struct definitions in `graph.c`.
- You **must not** modify the file `graph.h`.
- You **must not** modify the file `pagerank.h`.
- You **must not** modify the function prototypes for `graph_pagerank` and `graph_show_pagerank`.

Algorithm

For $t = 0$:

$$PR(p_i; t) = \frac{1}{N}$$

for $t > 0$:

$$PR(p_i; t) = \frac{1-d}{N} + d \times ((\sum_{p_j \in M(p_i)} \frac{PR(p_j; t-1)}{D(p_j)}) + (\sum_{p_j \in S} \frac{PR(p_j; t-1)}{N}))$$

Where:

- N is the number of vertices
- p_i and p_j are each some vertex
- t is the "time" (iteration count)
- $PR(p_i; t)$ is the PageRank of vertex p_i at some time t
- d is the damping_factor
- $M(p_i)$ is the set of vertices that have an outbound edge towards $M(p_i)$
- $PR(p_j; t-1)$ is the PageRank of vertex p_j at some time $t-1$
- $D(p_j)$ is the degree of vertex p_j, ie. the number of outbound edges of vertex p_j
- S is the set of sinks, ie. the set of vertices with no outbound edges, ie. where $D(p_j)$ is 0

This formula is equivalent to the following algorithm:

```
procedure graph_pagerank(G, damping_factor, epsilon)
    N = number of vertices in G
    for all V in vertices of G
        oldrank of V = 0
        pagerank of V = 1 / N
    end for
    while |pagerank of V - oldrank of V| of any V in vertices of G > epsilon
        for all V in vertices of G
            oldrank of V = pagerank of V
        end for
        sink_rank = 0
        for all V in vertices of G that have no outbound edges
            sink_rank = sink_rank + (damping_factor * (oldrank of V / N))
        end for
        for all V in vertices of G
            pagerank of V = sink_rank + ((1 - damping_factor) / N)
            for all I in vertices of G that have an edge from I to V
                pagerank of V = pagerank of V + ((damping_factor * oldrank of I) / number of outbound edges from I)
            end for
        end for
    end while
end procedure
```

In order to test your PageRank functions, you should modify `crawler.c` to `#include "pagerank.h"`, and change the last part of the `main` function to something like:

```
    ...
    graph_show(network, stdout, ignore_list);
    graph_pagerank(network, damping, epsilon, ignore_list);
    graph_show_pagerank(network, stdout, ignore_list);
    list_destroy(ignore_list);
    graph_destroy(network);
```

where you choose appropriate values for *damping* and *epsilon*.

Again, it is noted that the changes you make to `crawler.c` are purely for you to test whether your PageRank functions are working. We will use a different `crawler.c` for testing your PageRank functions.

Sample Output

Here we're using a modified `crawler.c` that calculates `graph_pagerank` and prints `graph_show_pagerank`. Damping has been set to 0.85 and epsilon to 0.00001. For the *micro-web*, and without ignoring any pages, the output should be:

```
./crawler http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter a page to ignore or type 'done': done
http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html: 0.412
http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html: 0.196
```

```
http://www.cse.unsw.edu.au/~cs9024/micro-web/Y.html: 0.196
http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html: 0.196
```

Now let's add *index.html* to the ignore list:

```
./crawler http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter a page to ignore or type 'done': http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter another page to ignore or type 'done': done
http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html: 0.333
http://www.cse.unsw.edu.au/~cs9024/micro-web/Y.html: 0.333
http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html: 0.333
```

X.html, *Y.html* and *Z.html* have no connections anymore and as such have the same ranks. Note that the sum is still (approximately) equal to 1, and N, the number of vertices, is equal to 3 in this case, since there were a total of 4 nodes originally, and 1 of the nodes has been ignored.

Testing

We have created a script to automatically test your PageRank functions. It expects to find `list.c` and `graph.c` in the current working directory. Limited test cases are provided, so you should always do your own, more thorough, testing.

```
9024 dryrun assn_rankings
```

Subset 4 – Degrees of Separation (Shortest Path)

In `graph.c`, implement the two functions prototyped in `dijkstra.h`: `graph_shortest_path` and `graph_show_path`.

First, `graph_shortest_path` should calculate the shortest path between a source vertex and all other vertices.

`graph_shortest_path` should use Dijkstra's algorithm to do so.

Note that an ignore list is also passed to `graph_shortest_path`. Similar to above, you will need to ensure these URLs are treated as non-existent. For example if there was a path `A->B->C`, but `B` is ignored, then there is no path from `A` to `C`.

Unlike a regular implementation of Dijkstra's algorithm, your code should minimise the number of edges in the path (not minimise the total weight of the path – consider each edge's weight to be 1).

Second, `graph_show_path` should print the path from the previously given source vertex to a given destination vertex. With the ignore list, there can be no path between two vertices. In this case, output nothing.

- You **may** add more utility functions to `graph.c`.
- You **may** (and most likely will need to) extend your struct definitions in `graph.c`.
- You **must not** modify the file `dijkstra.h`.
- You **must not** modify the file `pagerank.h`.
- You **must not** modify the file `graph.h`.
- You **must not** modify the function prototypes for `graph_shortest_path` and `graph_show_path`.

In order to test your Dijkstra functions, you should modify `crawler.c` to `#include "dijkstra.h"`, and change the last part of the main function to something like:

```
    ...
    graph_show(network, stdout, ignore_list);
    graph_shortest_path(network, argv[1], ignore_list);
    char destination[BUFSIZ];
    printf("destination: ");
    scanf("%s", destination);
    graph_show_path(network, stdout, destination, ignore_list);
    list_destroy(ignore_list);
    graph_destroy(network);
```

The changes you make to `crawler.c` are purely for you to test whether your Dijkstra functions are working. We will use a different `crawler.c` for testing your Dijkstra functions.

Sample Output

Using a modified `crawler.c` that accepts a source page as a command line argument from which to calculate `graph_shortest_path`, and a destination page to output `graph_show_path`, for the *micro-web*, and without ignoring any pages, the output in tracing a path from *X.html* to *Z.html* should be:

```
./crawler http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html
destination: http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html
Enter a page to ignore or type 'done': done
http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html
 -> http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
 -> http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html
```

Now let's add *index.html* to the ignore list:

```
./crawler http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html http://www.cse.unsw.edu.au/~cs9024/micro-web/X.html
destination: http://www.cse.unsw.edu.au/~cs9024/micro-web/Z.html
Enter a page to ignore or type 'done': http://www.cse.unsw.edu.au/~cs9024/micro-web/index.html
Enter another page to ignore or type 'done': done
```

Since *index.html* has been ignored, the path cannot be completed and nothing is returned. Your algorithm should iterate vertices/pages in the same order as the crawler. This ensures that when your algorithm finds the shortest path, it will return the first path it would encounter from the BFS in the crawler. If your result does not follow this order, you will be marked as incorrect, even if your path is valid.

Testing

We have created a script to automatically test your shortest path algorithm. It expects to find `list.c` and `graph.c` in the current working directory. Limited test cases are provided, so you should always do your own, more thorough, testing.

```
9024 dryrun assn_path
```

Assessment

Due Date

Wednesday, 17 April, 11:59:59.

Late Penalty:

- The UNSW standard late penalty for assessment is 5% per day for 5 days – this is implemented hourly for this assignment.
- Each hour your assignment is submitted late reduces its mark by 0.2%.
- For example, if an assignment worth 60% was submitted 10 hours late, it would be awarded 58.8%.
- Beware – submissions more than 5 days late will not be accepted and will receive zero marks. This again is the UNSW standard assessment policy.

Submission

You should submit your `list.c` and `graph.c` files using the following `give` command:

```
give cs9024 assn list.c graph.c
```

Alternatively, you can select the option to "Make Submission" at the top of this page to submit directly through WebCMS3.

Important notes:

- Make sure you spell all filenames correctly.
- You can run `give` multiple times. Only your last submission will be marked.
- Ensure both files are submitted together. If you separate them across multiple submissions, each submission will replace the previous one.

- Whether you submit through the command line or WebCMS3, it is your responsibility to ensure it reports a successful submission. Failure to submit correctly will not be considered as an excuse.
- You cannot obtain marks by e-mailing your code to tutors or lecturers.

Assessment Scheme

This assignment will contribute 12 marks to your final COMP9024 mark.

11 marks will come from automated testing, and 1 mark will come from manual inspection of your code.

The specific breakdown of marks is as follows:

Description	**Marks**
List ADT	3
Graph ADT	3
PageRank	2
Shortest Path	2
Memory Management	1
Code Quality	1
Total	**12**

Important:

- Any submission that does not allow us to follow the aforementioned marking procedure "normally" (e.g., missing files, compile or run-time errors) may result in delays in marking your submission. Depending on the severity of the errors/problems, we may ask you to resubmit (with max late penalty) or assess your written code instead (e.g., for some "effort" marks only).
- Ensure your submitted code compiles on a CSE machine using the standard options `-Wall -Werror`.

Memory management will be assessed using `valgrind`. You may refer to the Week 4 Practical for guidance on how you can compile your code and run it through `valgrind`. Note, this will require you to write some sort of "driver" or "test" program for your ADT.

Code quality will be assessed on:

- Readability – your code is generally easy to understand, follows typical spacing and indentation, and uses a consistent style.
- Documentation – your code is documented in places where it is harder to understand.

While you are not required to follow it, you may refer to the CSE C Coding Style Guide.

Collection

Once marking is complete you can collect your submission using the following command:

```
9024 classrun -collect assn
```

You can also view your marks using the following command:

```
9024 classrun -sturec
```

You can also collect your submission directly through WebCMS3 from the "Collect Submission" tab at the top of this page.

Plagiarism

Group submissions will not be allowed. Your programs must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar assessments in previous years, if applicable) and serious penalties will be applied, including an entry on UNSW's plagiarism register.

- ***Do not copy ideas or code from others***
- ***Do not use a publicly accessible repository or allow anyone to see your code***

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Policy Statement
- UNSW Plagiarism Procedure

Copyright