# Project 4: Graphs

**Due: 04/19/2024 at 11:59PM**

---

## General Guidelines:

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not need to be public should generally be kept private (instance variables, helper methods, etc.). Additional file creation isn't allowed as Vocareum will only compile the files given in the startercode.

*In general, you are not allowed to import any additional classes or libraries in your code without explicit permission from the instructor! Adding any additional imports will result in a 0 for the entire project.*

### Note on Academic Dishonesty:

Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. **You MUST do your own work**! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online!

### Note on implementation details:

Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

### Note on grading and provided tests:

The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

## Project Overview:

In this project, you will implement various properties of graphs and spanning trees. The learning objectives are understanding spanning trees, single source shortest path and walk properties, strongly connected components, and implementation of the above from scratch.

## Part 0: Graph Implementation (graph.cpp)

You will implement the functions below that will be used for all parts of the assignment. The graph should support non-simple, directed or undirected graphs with edge weights. Additionally, the edges should be able to store a “color” for use in part 2. Please use adjacency list to represent the graphs due to their complexity advantage

`void read_edge_weights(const std::string& filePath)`

Read the graph from the file specified by filePath, storing an integer weight with each edge:

- Line 1 of the file contains two integers $n$ and $m$ specifying the number of vertices and number of edges, respectively.
- For each edge $e$ such that $0 \leq e < m$, line $2 + e$ of the file is a space-separated list of the source vertex index, destination vertex index, and integer edge weight.
- For undirected graphs, if an edge $u - v$ exists, then the edge will appear twice in the list with the same weight: once in $u \rightarrow v$ order and once in $v \rightarrow u$ order.

An example file is listed below. This graph has 3 vertices and 3 edges.

```
3 3
0 1 15
1 0 40
2 1 76
```

`void read_edge_colors(const std::string& filePath)`

Read the graph from the file specified by filePath, storing both integer weights and colors with each edge. This will be used in Part 2. The format is identical to the edge weight format above, except that each edge also has an associated color.

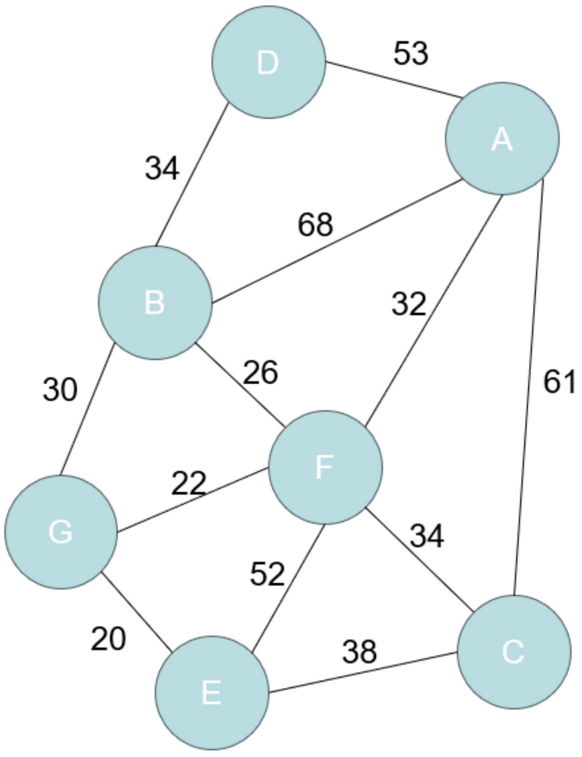
- Each line specifying an edge ends with a character representing the color of the edge.
- The only allowed colors are ‘R’, ‘G’, ‘B’.

An example file is listed below. It is the same graph as above, but each edge has a color.

```
3 3
0 1 15 R
1 0 40 G
2 1 76 B
```

## Part 1: Maximum Height in a Road Network (20 points) (max_height.cpp)

A network of roads connects various cities and is often limited to vehicles with respect to their height and weight. In this part, assume a similar connected network, where the vehicles are limited by their height. Consider the example below:



Here, the nodes represent cities, edges represent nodes and the edge weights represent the height constraint on the vehicle. The network is also connected. Your task here is to determine the maximum height a vehicle can have, to travel from **any** city to **any** city in the network.

You may choose 20 as it is the minimum height in the network, but it isn't the maximum height a vehicle can have. You might be tempted by 32 as it is a big enough height, but one cannot reach G with this height. On careful examination, 30 is the correct answer. With height 30, a vehicle can travel to any city from any city.

There are multiple solutions to this problem, but an efficient way to determine the maximum height can be through Spanning trees. Hint: Think about constructing the Maximum Spanning Tree. You can assume the graph is undirected and has one connected component.

Implement the function:
*int max_height::calculate(const graph& g)*,
returning the maximum allowed height. The runtime should be O(|E| log |E|) or O(|E| log |V|).

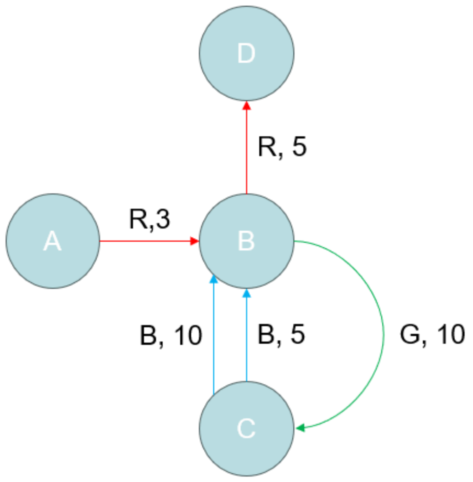
## Part 2: Color Coordinated Weighted Shortest Walk (50 points) (color_walk.cpp)

You are given a graph G = (V,E), whose edge weights are *positive* and each edge is assigned exactly one of the colors R, G, or B. Consider a path between two vertices in the graph, where each edge along the path follows a predefined color order: R followed by G, followed by B, followed by R, repeating until you reach the destination vertex. We call such a path a “walk”. Note that it may not be possible to reach the destination vertex if there is no sequence of edges between the source and destination that follows this pattern. Also note that an intermediate vertex may need to be visited more than once in order to reach the destination.

Your task is to determine the single source shortest weighted walk from a source vertex s to every other vertex in the graph with the constraints being:

- You can start with an outgoing edge of s that has color R or color G or color B
- If you start with R, then the sequence of edges in the shortest walk must follow the color scheme of R,G,B,R,G,B,...
- If you start with G, then the sequence of edges in the shortest walk must follow the color scheme of G,B,R,G,B,R,...
- If you start with B, then the sequence of edges in the shortest walk must follow the color scheme of B,R,G,B,R,G,...

The final result should be a color coordinated weighted shortest walk. A shortest weighted walk allows you to visit a vertex multiple times but an edge exactly once. A color coordinated weighted shortest walk implies finding the walk from s that obeys the color coordination described above, is a walk and the sum of weights along the walk is minimum, when compared with every other walk following a valid color scheme. You are to return a tuple *(starting color, dist)* for every vertex in the graph. We illustrate the problem with a set of examples from source A below.

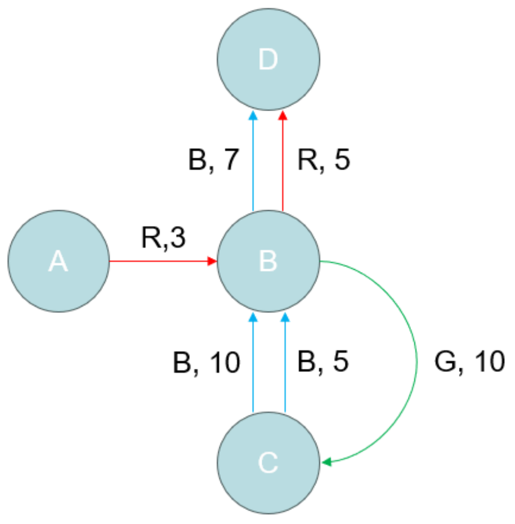


Example 1 (source A):

| Vertex: | A | B | C | D |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| (start color, dist): | (-, 0) | (R, 3) | (R, 13) | (R, 23) |

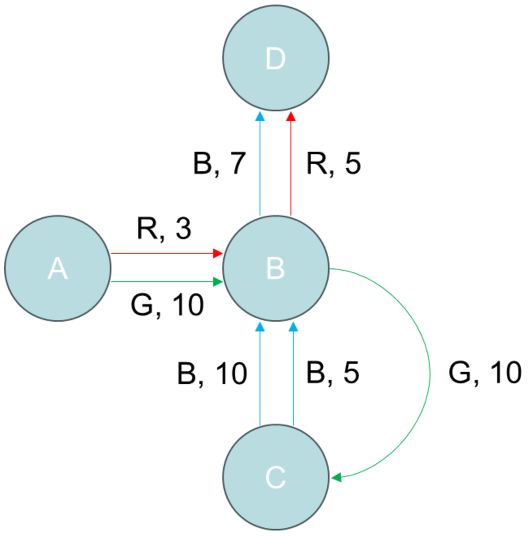
Example 1 asks you to determine the output for source A. A has just one outgoing edge of color R, so the only sequence allowed is R,G,B. Note, there are two edges from C to B of the same color and different weight. Hence, parallel edges of the same color may be present where the weight may differ. The output in the caption represents the output for all vertices. The first pair (0,-) states the shortest weighted walk from A to A is 0 with color '-'. *We will have the special color '-' for source vertex and unreachable vertices*. (R, 3) is for vertex B as the shortest walk is A - B with the starting color as R. (R, 13) represents the shortest walk to C which is A - B - C with a valid color scheme of R,G. Lastly, (R, 23) represents the shortest walk to D which is A - B - C - B - D with the color scheme of R,G,B,R. Note: the shortest walk from A - D is 8 (A - B - D) but it does not have a valid color scheme. Also, we have a parallel edge with color B and the shortest walk picks the edge weight 5 to arrive at 23.



Example 2 (source A):

| Vertex: | A | B | C | D |
|---|---|---|---|---|
| (start color, dist): | (-, 0) | (R, 3) | (R, 13) | (R, 23) |

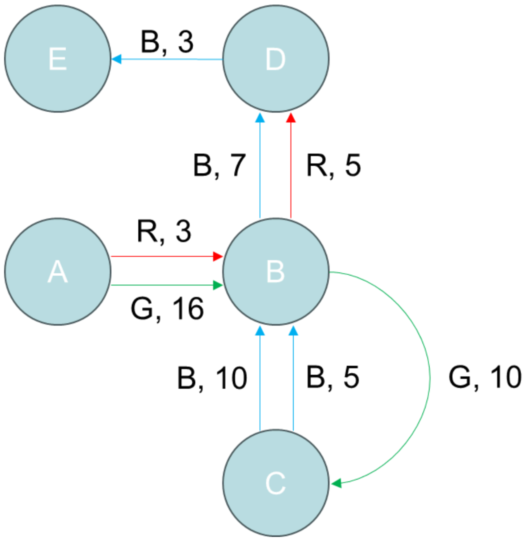
Example 2 has the same structure and result as Example 1 with the addition of a blue edge with weight 7 from B to D. This will not change the result as the color blue doesn't follow the color scheme from A to D (The walk A - B - D, with blue edge selected is a violation as the expected pattern is R,G and we observe R,B in this case. Secondly, A - B - C - B - D will have two blue edges B-C and B - D which is not valid).



Example 3 (source A):

| Vertex: | A | B | C | D |
|---|---|---|---|---|
| (start color, dist): | (-, 0) | (R, 3) | (R, 13) | (G, 17) |

Example 3 has the same structure and result as Example 2 with the addition of a green edge with weight 10 from A to B. Note that this changes the shortest walk to D (A - B - D being the walk) starting with the color G with a weight of 17, whereas the walk A - B - C - B - D has a larger weight of 23.



Example 4 (source A):

| Vertex: | A | B | C | D | E |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| (start color, dist): | (-, 0) | (R, 3) | (R, 13) | (R, 23) | (-, -1) |

Example 4 has a similar structure as example 3 with the addition of a vertex E, and the green edge from A to B changed to weight 16 instead of 10. Thus, the minimum distance from A to D is still (R, 23) (A-B-C-B-D), but there is a tie (G, 23) (A-B-D). In case of a tie, choose the earlier starting color R instead of G. Meanwhile, we can't reach E from A since the edge coming out from D should be either R or G, so the distance is set to -1 and the start color is listed as '-'.

You will return an array of length n that represents the shortest walk to each vertex from the source vertex. Each element in the array is a pair that indicates the starting color of the shortest walk to that vertex, along with the weight of the shortest walk. *The source vertex should have a distance 0 and color '-'. Any vertices that are unreachable should have distance -1 and color '-'.* If there are ties among shortest walks, red has the first preference, green has the second and blue has the third. The example below demonstrates tie resolution.

Example 5: Tie resolution for an arbitrary graph (source A)

| Vertex: | A | B | C | D | E |
|---|---|---|---|---|---|
| (start color, dist): | **(-, 0)** | **(R, 3)**<br>(G,3) | (R, 10)<br>**(G, 4)**<br>(B, 4) | (R, 13)<br>(G, 13)<br>**(B,2)** | **(-, -1)** |

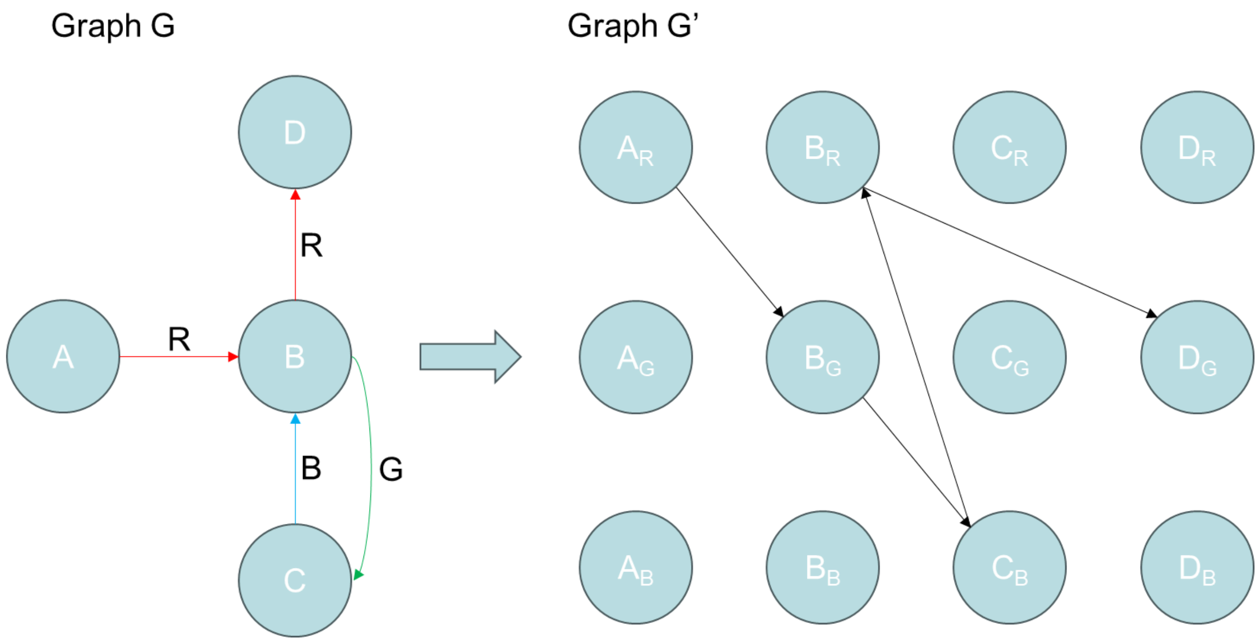
Example 5 represents the state of an arbitrary graph (this is not one of the graphs drawn above), where the bold pair represents the final pair, and others represent other valid weighted walks from A. For vertex B, we see R and G are tied at 3, so we select R, as red has the first preference. For vertex C, we see G and B have the least values and there is a tie between G and B. We select (G,4) as G has a higher preference over B. For D, we observe that B has the least weight (no ties), so we select B. Finally, vertex E is not reachable from A.

Hint: One possible approach to this problem is to transform the original graph G to G' by splitting every original vertex into three separate vertices. For example, vertex A can be split into $A_R$, $A_G$, and $A_B$, where the R, G, B represent the color of the edges exiting that vertex. The edges in the transformed graph thus no longer require a color, since the color information is "encoded" in the source vertex of the edge. You can accordingly treat G' as a standard edge-weighted graph without colors. Then for the source vertex S in G, we can calculate three single source shortest weighted walks from $S_R$, $S_G$, and $S_B$ in G', and select the least cost walk among the three to every vertex $V_R$, $V_G$, $V_B$ in G'. A shortest walk to one of $V_R$, $V_G$, $V_B$ in G' is equivalent to a shortest walk to V in G.

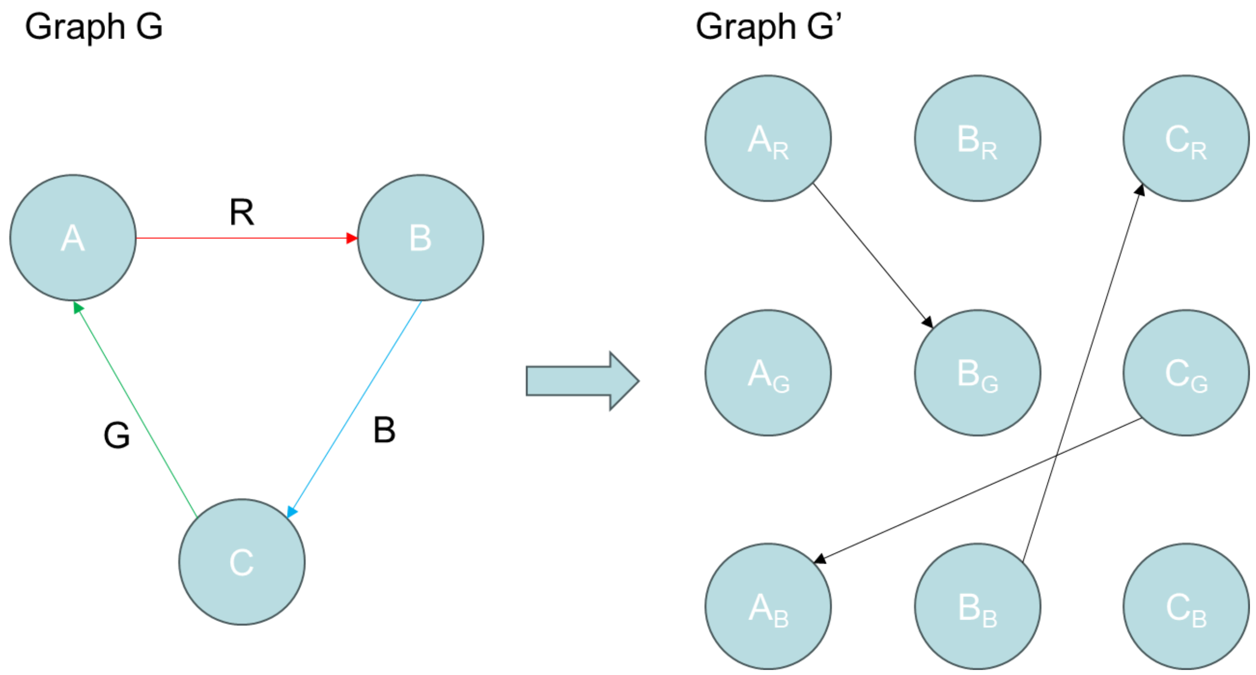
Table 1. From edge color in G to new edge endpoints in G'

| Edge Color in G | Source vertex (U) in G' | Destination vertex (V) in G' |
|---|---|---|

| | | |
|---|---|---|
| Red | $U_R$ | $V_G$ |
| Green | $U_G$ | $V_B$ |
| Blue | $U_B$ | $V_R$ |



Example 6: There is a valid walk from source A to every other vertex for graphs G and G'



Example 7: There is no valid walk from A to C for graphs G and G'

You need to implement the following functions:

*static std::vector<std::pair<char,int>> calculate(const directed_graph& g, handle startHandle)*

The runtime should be O((|V|+|E|) log |V|).

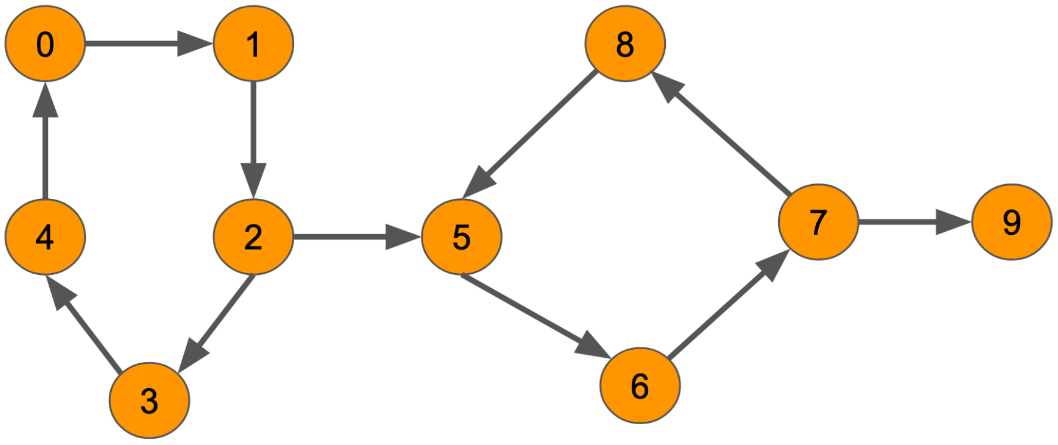
## Part 3: Clustering web pages within a website (30 points) (scc.cpp)

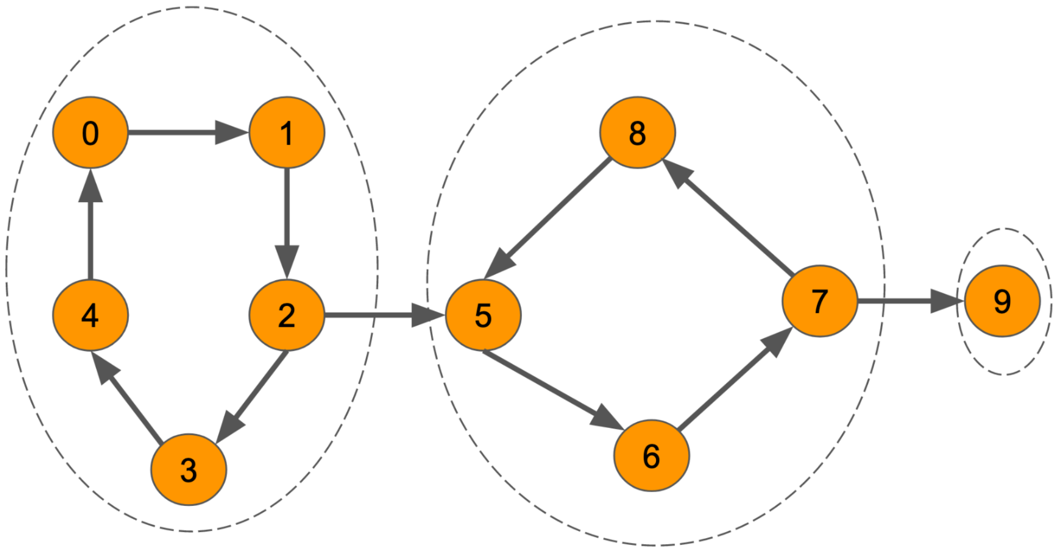
The goal of this part is to develop a tool that analyzes a website's structure by identifying strongly connected components (SCCs) in its internal hyperlink graph. This tool aims to reveal clusters of interconnected web pages, providing insights into the website's content organization and offering opportunities for optimization.

You will create a tool that takes a graph representing a website's structure, where each vertex is a web page, and each edge represents a hyperlink from one page to another. The primary task is to **compute the number of strongly connected components** (SCCs) in this graph. These SCCs represent clusters of web pages that are highly interlinked, suggesting thematic or contextual similarities. With this tool, the user will understand how content is organized and interconnected within a website, which is crucial for enhancing user navigation and search engine optimization (SEO).

Real-World Background:

For website owners, content creators, and SEO specialists, understanding the internal linking structure of a website is vital. It helps in identifying how well the content is interrelated, which in turn affects the site's search engine ranking and user experience. Clusters of strongly connected pages can indicate well-organized content silos, a key factor in effective SEO strategies. Conversely, identifying weakly connected components can highlight areas for improvement in the website's structure.





In the above example, there are **three** strongly connected components. Implement the function:

*int scc::search(const graph& g)*

## Building and Testing Your Code:

The following instructions are for Linux or Mac systems. For Windows, we recommend installing WSL as detailed in the Windows section below.

### CMake:

Included with the starter code is a CMake configuration file. CMake is a cross-platform build system that generates project files based on your operating system and compiler. The general procedure for building your program is to run CMake, and then build using the generated project files or makefile. On Linux that process looks like this:

1. Navigate to your project directory, where CMakeLists.txt is located
2. Create a build directory for compiled executables:
   ```
   $ mkdir build
   ```
   *Note: the $ symbol indicates the shell prompt - do not actually type $*
3. Navigate to the new build directory
   ```
   $ cd build
   ```
4. Run CMake, which generates a Makefile
   ```
   $ cmake ..
   ```
   *The .. indicates the parent directory where your CMake configuration is*
5. Compile the code using the Makefile
   ```
   $ make
   ```

If you want to compile with debug symbols, replace step 4 with
```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

If successful, the program will compile one executable: graph-app. The program takes 2 or 3 arguments: the part number, the input graph, and (only for part 2) the expected output. It will print the results to stdout. For example:

```
./build/graph-app 1 sample_tests/input/part1_test_00.txt
./build/graph-app 2 sample_tests/input/part2_test_00.txt sample_tests/expected/part2_test_00.txt
./build/graph-app 3 sample_tests/input/part3_test_00.txt
```

Each of the above commands runs a test from part 1, part 2, and part 3. For details on how it works, consult the source code in src/graph_app.cpp.

### Testing:

Provided with the starter code are several sample inputs and expected outputs for graph-app. To see if your implementation is correct, run the programs with the inputs, and compare the output with the expected outputs. For example:

```
$ ./graph-app 1 ../sample_tests/input/part1_test_00.txt > part1_test_00_output.txt
$ diff -qsZB part1_test_00_output.txt ../sample_tests/expected/part1_test_00.txt
```

If diff reports the files as identical, then your implementation matches the expected output.

**Windows Setup:**

For Windows users, we recommend using Windows Subsystem for Linux (WSL) and Visual Studio Code as your development environment. Using WSL, you can use all the same commands above to build and run on your local Windows computer. WSL can be installed through the Microsoft Store or by following the instructions here. The link also contains some customization/setup directions. We also recommend downloading Ubuntu from the Microsoft Store and using it to navigate your WSL and install packages.

You will need to install compilers and CMake in order to run and test your project. The required compilers as well as the "make" command can be installed using Ubuntu's build-essential package. This and CMake can be installed with the commands:

```
$ sudo apt update

$ sudo apt install build-essential cmake
```

You will have to enter your WSL password before installation begins to allow privileged access. You can check that the needed items were installed by running the corresponding command with the --version argument. If a version is printed, then that command was installed.

After downloading the work folder from Vocareum, you can move it into your desired directory in WSL using the WSL command line. You can access your Windows home directory from WSL using the path "/mnt/c/Users/<user>/"

To work on your code in WSL, you could use a text editor like Vim or Nano, or you could install the WSL extension in Visual Studio Code. After installing the extension, you can select the blue box at the bottom left of the Visual Studio window and select "Connect to WSL" from the dropdown window. This will allow you to open any WSL files and directories in your Visual Studio environment, and if you open a Terminal you can navigate your directories using the WSL Command Line. (Note: you may need to reinstall other extensions for WSL that you have already installed on your Windows version of VSCode.) From the Terminal, you can execute the `cmake` and `make` commands as specified above to build locally.

**You should also test your code on Vocareum**. After uploading your files, press the **Run** button on the top right. This will execute a script that will compile your code and run it against all of the sample test cases, and print a summary of which tests failed. You should run your code before submitting it to make sure it compiles and executes properly, since you will have a limited number of submissions.

Watch out for infinite loops! If your program enters into an infinite loop during a test, you will receive a 0 for the entire assignment. Make sure you have safeguards in your code to prevent this from happening.

Note that the sample test cases are not exhaustive and purposefully do not cover all possible cases. When you submit your code it will be tested against many more tests with more thorough cases, which will not be made available to inspect. It is your responsibility to ensure your program is correct, by devising new tests and manually inspecting their outputs.