Programming Assignment 5 (Union-Find, Kruskal's Algorithm, KMP and Strongly Connected Components)

Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

Instructions For Submissions

- **Each group to have at most 2 members**. Although you can work individually, I encourage you to get a partner. One submission per group. **Mention the name of all members.**
- **Submit code and a brief report**. Submission is via Canvas as a single zip file. No need to include the algorithm description in the report.

---

1 Overview

We are essentially going to **implement a list-based Union-Find**, **implement Kruskal's algorithm**, and **implement KMP**. Additionally, you will write a report on **Kosaraju's Strongly Connected Components algorithm**.

To this end, **your task is to implement the following methods**:

- In **UnionFind**: `makeSet`, `find`, `append`, and `doUnion`
- In **Kruskal**: `runKruskal`
- In **KMP**: `runKMP` and `computeFailure`

The project also contains additional files which you do not need to modify (but need to use). You will use `TestCorrectness` to test your code. For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not.

Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

1.1 Testing Correctness

To test the correctness of your graph implementations, I have included: `mst_graph.txt`; the corresponding graph is shown below.
Important: For the graph methods to work, you MUST fill in the path (in **TestCorrectness**) for the folder where the graph files are stored.

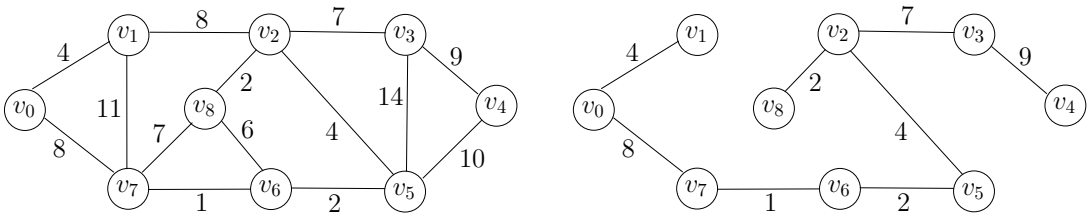


Figure 1: Graph (left) used for testing MST algorithms; corresponding MST on right

1.2 C++ Helpful Tips

For C++ programmers, remember to use DYNAMIC ALLOCATION for declaring any and all arrays/objects. DO NOT forget to clear memory using *delete* (for objects) and *delete*[ ] for arrays when using dynamic allocation.

Remember to return an array from a function, you must use dynamic allocation. So, if you want to return an array x having length 10, it must be declared as **int *x = new int[10];**

2 Union-Find

Implement the `makeSet`, `find`, `append`, and `doUnion` methods using the following pseudo-codes.[1]

MakeSet

- create a linked list, call it LL (Use dynamic allocation in C++)
- insert x at the end of LL
- $representative[x] = LL$

Find

return $representative[x]$

Append

- set next of $arg1$'s tail to point to $arg2$'s head
- set tail of $arg1$ to $arg2$'s tail
- increment $arg1$'s size by $arg2$'s size
- for each ListNode $node$ in $arg2$, set $representative[node's\ value] = arg1$
- set $arg2$'s head and tail to null

Union

- LinkedList $LL_x = \text{FIND}(x)$
- LinkedList $LL_y = \text{FIND}(y)$
- If $(LL_x \neq LL_y)$, do the following:
  - If (size of $LL_x \geq$ size of LL_y) $\text{APPEND}(LL_x, LL_y)$
  - Else $\text{APPEND}(LL_y, LL_x)$

[1] As you may have already understood, *doUnion* is the *union* method. C++ has a default construct by the name *union*; hence, the change in name.

3 Kruskal's Algorithm

Implement the `runKruskal` method using the following pseudo-code.

- Sort the edges of the graph.
- Create a UnionFind object by invoking the constructor with argument $numVertices$. Call this object $objUF$. Create a dynamic array of type $Edge$.
- Initialize $numEdgesAdded = 0$;
- For each edge e in $edgeList$, do the following:
  - Let src and $dest$ be the source and destination of e.
  - If src and $dest$ are in different components (i.e., the $find$ calls on them return different values), then do the following:
    - Call the $doUnion$ method on $objUF$ with src and $dest$ as arguments
    - Add e to the dynamic array
    - Increment $numEdgesAdded$ by one.
    - if $numEdgesAdded$ equals $(numVertices - 1)$, then stop the process.
- Return the dynamic array.

4 Knuth-Morris-Pratt

Implement the `runKMP` and `computeFailure` methods using the following pseudo-code.

Run KMP

- Create an integer dynamic array occ. Call $computeFailure$ to get the failure array F for the $pattern$.
- Let $txtLen$ be the length of the $text$ and $patLen$ be the length of the $pattern$
- Initialize $t = 0$ and $p = 0$
- As long as $(t < txtLen)$, do the following:
  - If ($pattern[p]$ equals $text[t]$), then do the following:
    - If (p equals $patLen - 1$) then the pattern has been matched; so, add $(t - p)$ to occ and set $p = F[p]$. Otherwise, we need to check the next character of pattern; so increment p
    - In either case, we need to check the next character of the text; so increment t
  - Else a mismatch has occurred; so, do the following:
    - If $(p \neq 0)$, then the last matched character of the pattern is at index $(p - 1)$; so, slide the pattern by setting $p = F[p - 1]$. Otherwise, the first character of the pattern has mismatched; so increment t
- Return occ

Compute Failure

- Let $patLen$ be the length of the $pattern$
- Create an array $F[\ ]$ of the length $patLen$
- Initialize $pref = 0$, $suff = 1$, and $F[0] = 0$
- As long as $suff \neq patLen$, do the following:
  - If the characters of the pattern at the indexes $suff$ and $pref$ are the same, then do the following:
    - increment $pref$ by one
    - set $F[suff] = pref$
    - increment $suff$ by one
  - Else if $pref$ is 0, then set $F[suff] = 0$ and increment $suff$ by one
  - Else set $pref = F[pref - 1]$
- Return F

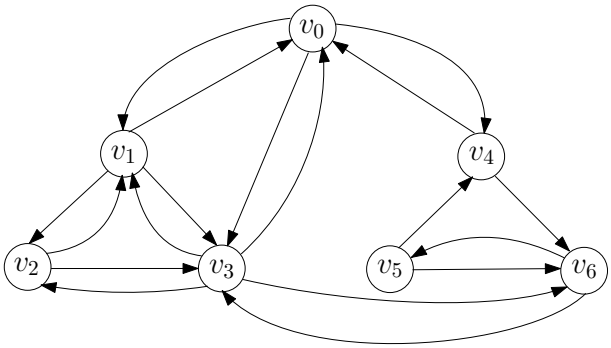
5 Report: Strongly Connected Components

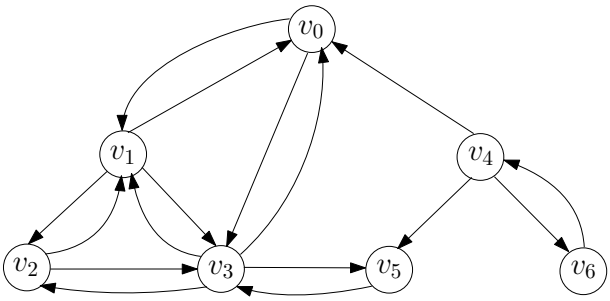
For the `SCC` class, the graphs in the next page have been used to test correctness. The corresponding strongly connected components can be found in the `ExpectedOutput` file.

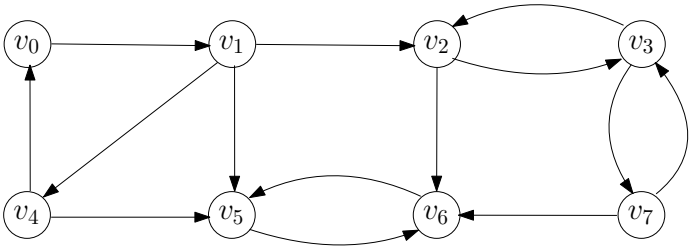
You do not have to write any code. Instead, write a report on the `SCC` class, by using the algorithm description here: https://drive.google.com/drive/folders/1qXH8OMuBRkDewOzMaL2RbJwxAGcPeZSU?usp=sharing. In particular, explain the code by answering the following:

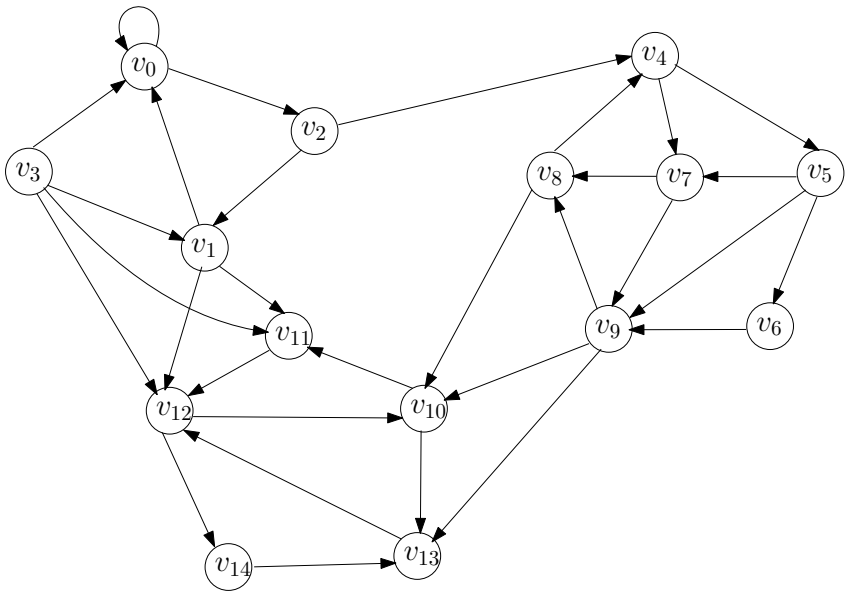
- What is the purpose of the `step1Helper` method? Explain the purpose of the stack here.
- What is the purpose of the `step1` method? Clearly explain what algorithm is being run here.
- What is the purpose of the `step2` method?
- What is the purpose of the `step3` method? In particular, explain the following:
  - What is the purpose of the outer while loop?
  - What is the purpose of the inner while loop, and what is getting added to `component`? What algorithm is being run here?
  - After the inner loop terminates, what is being added to `scc`?

Caution: You should explain/answer the above in the context of what the method/statement/part of code achieves. Writing something like "*this code has a for-loop that goes over all the edges of a vertex*" will get you no credits. I do not want an English description of the code; I want an explanation of the code's purpose.


v_0
v_1
v_4
v_2
v_3
v_5
v_6


v_0
v_1
v_4
v_2
v_3
v_5
v_6


v_0
v_1
v_2
v_3
v_4
v_5
v_6
v_7


v_0
v_4
v_2
v_3
v_8
v_7
v_5
v_1
v_{11}
v_9
v_6
v_{12}
v_{10}
v_{14}
v_{13}