Tristan Basil

12/15/2016

CS315-002 -- Final Project

*As easy as cab* – Programming Competition Analysis and Topological Sort Using Adjacency Matrices

***Section 1: Problem Introduction***

* 1. **Abstract**

“As Easy as CAB” was a problem assigned during the ACM Mid-Central Regional programming competition, having to do with topological sort and DAGs (Directed acyclic graphs). It can be found on KATTIS and solved by anyone today (<https://open.kattis.com/problems/easyascab>). This problem highlights the usefulness of different data structures, as well as clever algorithms to avoid redundant processing, particularly within the context of programming competitions and programming with strict runtime constraints. This project seeks to explain the analytic process that must be undertaken to solve such a programming competition problem by analyzing this example in-depth. 2 possible solutions and their effectiveness in addressing the problem statement will be provided.

* 1. **Importance of the problem**

This programming problem highlights the usefulness of different data structures, as well as clever algorithms to avoid redundant processing. Programming competitions are based on the idea of the need to solve different types of problems by using efficient, established algorithms. Running time, as well as actual implementation time, are frequently the limiting factor in creating a successfully submitted solution. Therefore, a simple, efficient solution employing the correct algorithm for the problem saves not only implementation time, but running time as well.

Beyond addressing the specific objective of the problem’s original programming competition context, this problem’s basis in language analysis could be applied to the analysis of real languages as well (assuming one has a dictionary of the unknown language). Analysis of the problem also provides a model for how one should approach such programming problems not only within a competition, but in a situation that forces strict run-time and implementation time constraints.

**1.3 Problem statement**

While the original problem statement can be found at the link provided in section 1.2, a summary of the problem statement for “As easy as cab” follows:

*Given a set of strings in a new ‘alphabetical’ order, write a program that can discover the alphabet of a given language, or discover that the language is impossible or ambiguous.*

*For example, given the strings: cab, cda, ccc, badca,*

*Derive the alphabet of the given language. Here, it is “adcb” (an English dictionary would yield “abcd…”). It may be that a list contains inconsistencies that make it impossible to be ordered under any proposed alphabet.* *Additionally,* *some lists may not provide enough clues to derive a unique alphabet order.*

*The first line of input will contain L and N, separated by a space, where L is a lowercase character b≤L≤z representing the highest character in the traditional alphabet that appears in the derived alphabet, and N is an integer 1≤N≤1000 that is equal to the number of strings in the list. Each of the next N lines will contain a single nonempty string of length at most 1000.*

***Section 2: Analysis***

**2.1 Input/Output**

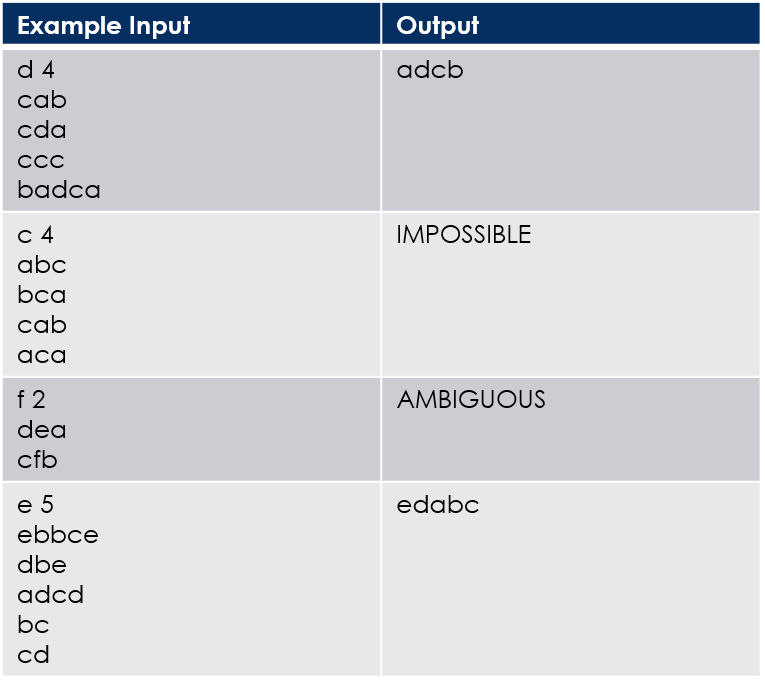
 Various example inputs and outputs to better understand the problem are included in *Table 1* to the right. Note that the first line defines the highest letter that appears in the language, and the amount of words that will follow it. This simplifies allocating arrays and allows the program to expect exact input.

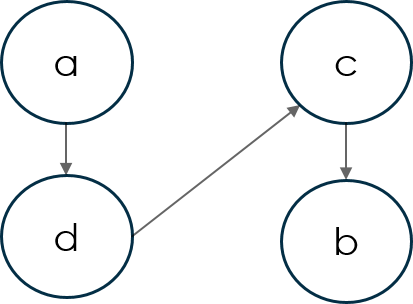
Table 1: Basic I/O for "As easy as cab"

In order to distinguish the hierarchy of the alphabet, each pair of strings should be compared, letter by letter. If the letters are same in the relative position from the front, continue to the next pair of letters in the pair of strings. If the letters differ, it is clear that the letter of the second string is preceded by the corresponding letter in the first, creating a ‘hint’ as to the order of the language.

**2.2 Problem Analysis**

The ‘hints’ described in *Section 2.*1 create a directed flow of hierarchy between each letter for each hint: therefore, it is clear that this problem models a graph problem. Each vertex of the graph serves as a letter, while each edge represents the higher position of one letter to another. For example, the first example input provided in Table 1 includes the pair of strings “cab” and “cda”, indicating that a is followed by d: therefore, a directed edge from vertex a to vertex d can be created in the graph.

However, with further analysis, it is clear that the graph should be, in particular, a **Directed Acyclic Graph.** In a directed acyclic graph, all nodes must be interconnected, and all edges must be directed in the same direction, i.e. they should all possess a distinct depth value (number of edges from root node). The following page contains 3 examples from *Table 1* which illustrate this principle for this problem.

*Example 1:*

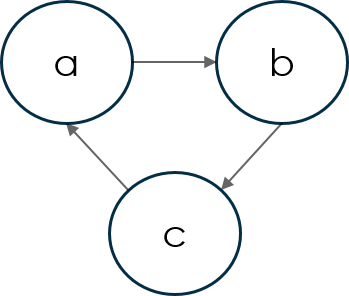
a🡪d

d🡪c

c🡪b

This configuration is valid. Solution: adcb

Example 1: A valid, directed asymmetrical graph (DAG)

*Example 2:*

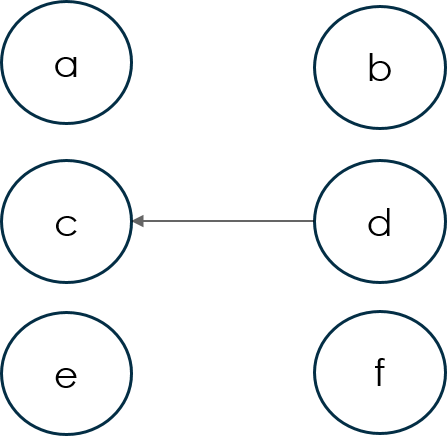
a🡪b

b🡪c

c🡪a

There cannot be loops for a logical solution. It is clear that this configuration is impossible. The correct output would be “IMPOSSIBLE”.

Example 2: An invalid graph with a loop

*Example 3:*

d🡪c

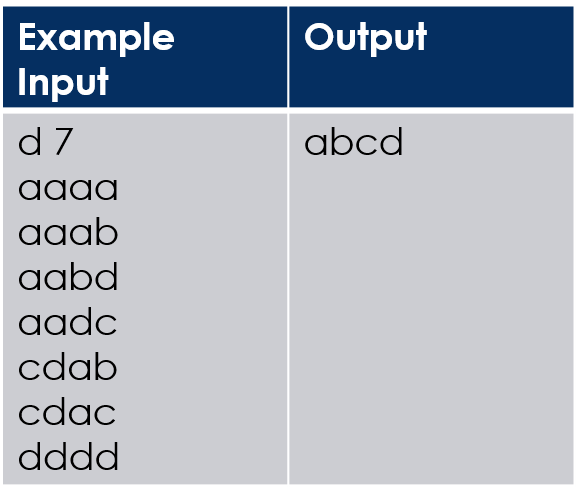
The other nodes do not have connections, but the graph must be interconnected in order for a hierarchy to be established (depth). This configuration is ambiguous, and the correct output would be “AMBIGUOUS”.

*Conclusion:*

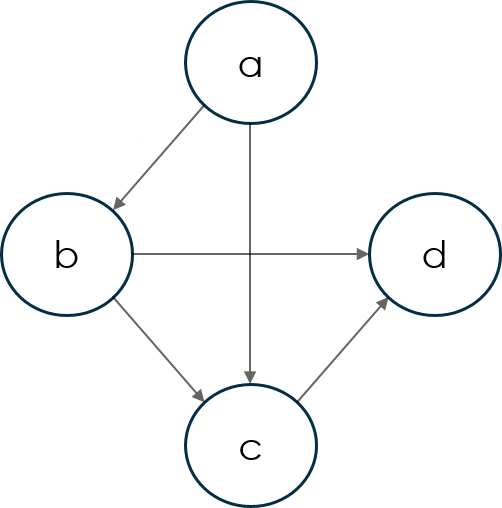
It is clear that, from the given examples, only **directed acyclic graphs** (graphs with directed edges in one direction, where all such nodes are interconnected with no loops) are accepted as valid solutions to the problem.

Example 3: An invalid graph that is not interconnected

***Section 3: Solution Analysis***

**3.1 Possible (Naive) Solution: DFS**

A possible, albeit inefficient way to approach this problem is via a DFS:

1. First, establish the graph using each pair. We can use the C++ <map> to map each letter to its particular struct, containing the number of predecessors and whether or not it has been visited. We can use a <set> to keep its followers, which ensures that duplicate “hints” will not be stored in the list. An example graph that this process builds is provided to the right in *Example 4*.

Example 4: Example DFS input. Note that a🡪b connection occurs twice, but using the <set> circumvents its placement twice.

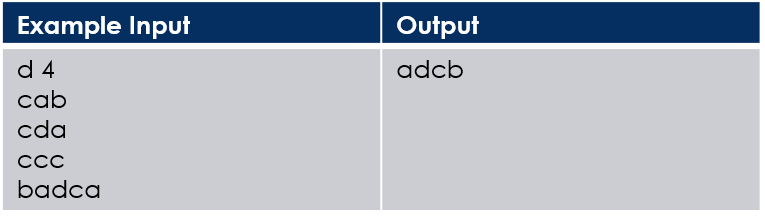
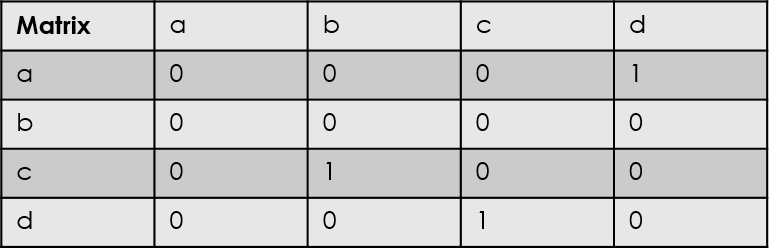
1. As we establish the graph, count the number of direct predecessors to each node.
2. After the graph is established, find the node with no predecessors. If there is more than 1, it is ambiguous. If there are none, then there must be a cycle; this is impossible.
3. Recursively backtrack through the graph starting at the beginning, marking the depth of each node. If a particular node has more than 1 depth from different paths, the longest should be used.
4. Use these depths to build the final string.

While this problem does correctly model the graph and correctly solve the problem, it is ultimately inefficient and does not meet the problem’s strict 1 second time constraint for a number of reasons:

1. DFS, while exhaustive, is very redundant in this case. Looping through every possible character yields paths that are too short, and only 1 path ultimately contains the correct depth of the letter (the longest). For example, in the language above, only the path a🡪b🡪c indicates c’s true position (depth 2), but the a🡪c path is still taken. The a🡪c path would indicate a depth of 1, but is meaningless with the existence of the longer path: therefore, the traversal down this edge is useless.
2. After establishing the depth of each letter, this particular solution must rebuild the final alphabet string, making note of any nodes at identical depths and returning “AMBIGUOUS” if they appear: this acts as additional overhead to the problem.

**3.2 Better solution: Topological Sort, using adjacency matrix**

A far more efficient (and clever) solution follows, which makes use of an adjacency matrix and takes advantage of topological sort:

1. Instead of maintaining a literal structure representation of each node, we keep a 2D array of adjacent, i.e. immediately connected nodes. This is known as an **adjacency matrix**. A parallel array keeps track of the amount of immediate predecessors each node has (int numPredecessors[]), and a Boolean array represents if it has >0 (bool hasNoPredecessors[]). As indicated in the matrix to the right, the first pair indicates d must follow a: therefore, Matrix[a][d] receives value 1 to indicate boolean True.

Example 5: Topological sort using adjacency matrix. This abstraction is far more efficient than DFS, and avoids traversing repeat ‘hints’ by only traversing direct edges from the root node (initially, from a)

1. We loop through the array of predecessors to find the letter(s) without predecessors. Similar to before, if there are 0, the graph is impossible; >1, ambiguous.
2. The single letter we find with no predecessors is the first letter in our alphabet. We then effectively remove the node from the graph by decrementing the amount of predecessors each of its immediately following nodes has (determined from hints from pairs of strings), and marking the following node’s hasNoPredecessors[] value true if it indeed has none. In the first iteration of this loop using the a🡪d hint, this step would decrement d’s number of predecessors from 1 to 0, indicating d to the be the next valid letter in the sequence.
3. We repeat step 3 with the modified graph. This process concatenates the final language string.

The topological sort and adjacency matrix solution is better than the possible DFS solution for a number of reasons:

1. As mentioned before, DFS is redundant in this problem, while the topological sort and adjacency matrix ensures that the absolute path of the DAG is implicit by using the number of immediate predecessors. It is clear that a node with no immediate predecessors is the next node in the sequence.
2. Adjacency matrices are much less cumbersome than C++ <set>, <map>, and <vector>. By using the various assumptions that the problem provided, the adjacency matrix not only simplifies the problem but makes use of much simpler and faster data structures to address it.
3. By mapping each letter to indexes within the array, this solution also circumvents the need for the <map> abstraction used in the first solution.
4. Ultimately, the first solution is too slow for this particular problem. Only the second is accepted due to the runtime constraint of 1 second, which becomes apparent for large inputs.

***Section 4: Conclusions and Final Remarks***

There are various ways of approaching programming problems, and “Easy as CAB” is no exception. Both provided solutions were perfectly correct with respect the inputs and outputs of the problems, but only the second fully addressed its specification of 1 second runtime. Correctness, therefore, should never be used as the sole basis of a program’s design: rather, every specification, particularly runtime, should be considered in the design or implementation of an algorithm.

Similarly, each particular interpretation/implementation of a problem can lead to its own issues and benefits. Understanding the inefficiencies of algorithms and planning beforehand can circumvent efficiency problems that must eventually be debugged or profiled later.

Finally, generic graph algorithms, such as topological sort, can be applied to various different problems. Programming competition problems such as this one exemplify the infinite number of applications that graphing algorithms can apply to, and the various different ways in which those problems may be approached. Understanding the appropriate use cases of such algorithms, therefore, is just as important, if not more so, than creating a solely ‘correct’ solution.

***Section 5: Works cited***

* KATTIS and ACM for the original “As easy as cab” problem spec: <https://open.kattis.com/problems/easyascab>
* The ACM Mid-Central Regional Programming Competition judge panel for creating the original topological sort and adjacency matrix solution analyzed in Section 3.2: <http://www.icpc-midcentral.us/archives.html>