CSCE 2202-01, -02 Term Project Spring 2022
Guidelines

The term project is **NOT OPTIONAL.** The project grade represents **20%** of the total course grade.

Guidelines:

- The project is preferred to be a teamwork.
- You should select ONE of the 3 projects presented here.
- The project topic should be different from the topic of the term paper.
- A group **of 1 - 4 members** is to complete the project by the deadline and report the work in a project report (format attached).
- All group members **must be from the same section**. Group members may **NOT BE MIXED** between the sections of 2202
- The project group can be the same as that presenting the term paper
- Names of group members and the section should be sent by e-mail to the course TA Eng. Mohamed Hany: mohamed.ihany@aucegypt.edu by **Monday April 11th, 2022. No allowance to change the names of group members once they have been sent to Mr. Hany.**
- **Names** of group members and the section **must be submitted** in the final project report with **no allowance to change the names from those sent before.**
- The final project report should be submitted on blackboard.
- **Final date** of submission of the complete project report is **Thursday May 12, 2022**.
- There will be NO late policy for the submission of the project. **No submissions will be accepted after the due date.**

Academic Integrity

Students are expected to commit to the principles of academic integrity. Any plagiarism detected will result in zero grade for the project.

Project Components

- A problem to be defined
- Adopted methodology for the solution
- Implementation of one or more algorithms in the adopted methodology
- Choice of your input experimental data
- Demonstrations by your processing of input experimental data
- Analysis of algorithms used and your output experimental results as well as critique of methodology used
- Conclusions

CSCE 2202-01, -02 Term Project Spring 2022
Suggested Project Report Format

Project Title

Group Name/s
Department of Computer Science and Engineering, AUC

Abstract:
This is just a suggested format. You may change or modify it in the way that suits the project work
……
……
Keywords:

1. Introduction
(Introduce the topic of the project here)
……
2. Problem Definition
(Present here a more detailed definition of the topic or problem)
……
3. Methodology
(Outline the methodology chosen to solve the problem)
……
4. Specification of Algorithms to be used
……
5. Data Specifications
(Specify the input data to be used in your work)
……
6. Experimental Results
(Give your results of processing the input data in text, tabular or graphical forms)
……
7. Analysis and Critique
(Your own analysis and critique of the output results and the methodology/algorithms used)
……
8. Conclusions
……
Acknowledgements
(Acknowledge other people who helped you in producing the project)
…….
References
(List here the references you used to produce the project)
……..…….

Appendix: Listing of all Implementation Codes
……..
……..

Project (1)
Text Compression Utility

Introduction

ASCII files can be compressed to smaller sizes by using variable length Huffman Coding. By analyzing the different probabilities of the various symbols that occur in a given file, we can build a Huffman tree to come up with a minimal, optimized binary list of code words for these symbols.

Algorithm

Huffman coding is a famous example of a greedy algorithm. The problem deals with the lossless compression of a message by coding its symbols using a variable length binary prefix-free coding scheme. The algorithm was introduced in 1952 by David Huffman based on a greedy approach (see Lecture Slides and Lecture Notes on Greedy Algorithms). The algorithm produces an encoding that minimizes the average length $< L >$ of a code word and gets as close to Shannon's bound (H) as possible. Huffman's idea is to use the concept of optimal merge patterns to construct the binary code tree. Since in such merge tree the paths from the root to a symbol might not be equal, the resulting code words will be of variable length. In particular, more frequent symbols will be in leaves nearer to the root, i.e. will be of shorter code lengths, while leaves of less frequent symbols will occur at deeper levels, i.e. having longer code lengths.

The Project:

In this project, you will be implementing a compression utility for ASCII-encoded text. Your program should be able to both compress and decompress text files, and to compute the compression ratio and the efficiency of your encoding. The steps to implement this program are as follows:

1. Determine the symbols / characters used in the file, build your alphabet and probabilities of the symbols.
2. Build a merge tree to find the optimal prefix binary coding for each symbol.
3. Encode your characters with the new binary codes.
4. Save the coded characters along with your code table in the output compressed file.

For decompression, your program should read the code table / tree from the compressed file and use that to recover the original non-compressed text file.

Optimally, your program should have three different parameters passed to it at runtime:

1. A flag indicating whether it is being used to compress or decompress the input file.
2. The path of the input text file.
3. The path to write the output file (compressed file in case of compression, text file in case of decompression).

When used for compression, your program should output the compression ratio (for ASCII-encoded files where each character is written in 8 bits, the compression ratio will be $< L > / 8$) and the efficiency (η) of your encoding.

Submission Instructions

Your source code and executable binary file should be submitted on blackboard by the due date, along with a README file that describes how the command line parameters should be passed (or help instructions to be printed by the program when no parameters are passed)

Grading

The program will be tested against a sample test file to compress it, check the size of the compressed file, and then used again to decompress the file to obtain the original text file. If your program works, it should be capable of producing a smaller sized compressed file, and it should be able to decompress the file to obtain the original file. In this case, grading will be on both the program and the project report.

Project (2)
Simple Plagiarism Detection Utility using String Matching

Introduction

Plagiarism is a serious problem in study and research. In this project, you will implement a simple plagiarism detector. Your input will be a collection (corpus) of existing documents and a potentially plagiarized document. Your output will be the set of documents from which the document was plagiarized.
Given a test file, treat each sentence in the file as a potential pattern. Search for the pattern in the existing documents and find the matches.

This problem is typically approached through string matching. In the string matching problem, we are given a string of n characters called the *text* and a string of m characters ($m \leq n$) called the *pattern* and we want to find a substring of the *text* that matches the *pattern*. More precisely, we want to find in the text the start index of the first substring that matches the pattern, if it exists.

Algorithms

There are several algorithms designed to find the location where one or several strings (patterns) are found within a larger string or text. A summary of the most famous methods for string matching is given in the following:

1. Brute Force Matching using Hamming Distance:
   This method uses Brute–Force sliding window matching (see course slides and Lecture Notes on Brute-Force Algorithms). It can be used to detect approximate matches and to find suggested closest matchings. This method has the highest complexity among other algorithms.
   The definition of closest follows the Hamming-distance concept. In this method, the Hamming Distance between two strings follows the algorithm below:

   For two strings X, Y, where length(X)| ≤ length (Y)
   A match occurs at position k if A(k) = B(k).
   M = Number of matches
   Distance = Number of mismatches D = length(Y)) – M
   D = 0 if X and Y are identical

2. The Rabin-Karp Algorithm
   This is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text.

3. The KMP Algorithm
   This is a string-searching algorithm created by Donald Knuth, James H. Morris, and Vaughan Pratt (1977). The algorithm searches for occurrences of a pattern string within a main "text string" by employing the observation that when a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

4. The Boyer–Moore Algorithm
   This is an efficient string-searching algorithm that is the standard benchmark for practical string-search. It was developed by Robert S. Boyer and J Strother Moore in 1977. The

algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). It is thus well-suited for applications in which the pattern is much shorter than the text or where it persists across multiple searches. The Boyer–Moore algorithm uses information gathered during the preprocess step to skip sections of the text, resulting in a lower constant factor than many other string search algorithms. In general, the algorithm runs faster as the pattern length increases. The key features of the algorithm are to match on the tail of the pattern rather than the head, and to skip along the text in jumps of multiple characters rather than searching every single character in the text.

The Project:

Your project is to build a **Simple Plagiarism Detection Utility** using string matching algorithms.

You are required to:

- Build a collection (corpus) of existing documents and a potentially plagiarized document (the choice and number of corpus documents, the size of the document and the potentially plagiarized document are left to your discretion).
- Implement the following basic detectors:
  1. Approximate string matching using Brute Force and Hamming Distance
  2. Rabin–Karp algorithm
  3. Knuth–Morris–Pratt algorithm (KMP Algorithm)
  4. Boyer–Moore string-search algorithm
- Using the corpus, find the documents from which the potential document was plagiarized. Given a test file, treat each sentence in the file as a potential pattern. Using the above algorithms, search for the pattern in the existing documents and find the matches.
- Compare the performance of the implemented algorithms (measures of performance are left to your discretion).

Submission Instructions

Your source code and executable binary file(s) should be submitted on blackboard by the due date, along with a README file that describes how to use the utility.

Grading

The program will be tested against a sample potentially plagiarized document. If your program works, it should be capable of detecting plagiarism. In this case, grading will be on both the program and the project report.

Project (3)
Finding the Minimum Spanning Tree (MST) for a Graph

Introduction

In many problems, we choose to represent the relationships between problem entities as a connected, edge-weighted undirected graph with |V| vertices and |E| edges. For such graph *G(V,E)*, a sub-graph *S(V,T)* is a spanning tree of the graph (*G*) if:

1. *V(S) = V(G)* and $T \subseteq E$
2. *S* is a tree, i.e., *S* is connected, has no cycles and $|T| = |V|-1$

In this case, a spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles.
For a connected edge-weighted undirected graph, there could be more than one spanning tree. The cost of a spanning tree is the sum of the weights of its edges. A Minimum Spanning Tree (*MST*) is a spanning tree of the graph with minimum cost.

There are several problems that can be solved by finding minimum spanning trees. These include design of computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids. An example is cable laying for telecommunication among communication points and road paving between houses.

Algorithms

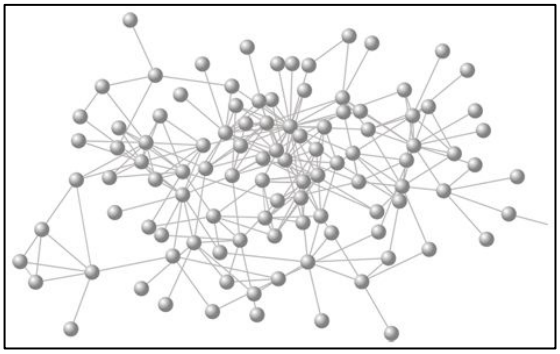
1. A commonly used algorithm to determine the *MST* is **Kruskal's algorithm**. This is a greedy algorithm in which the partial solution may consist of a forest of *MST's* that are joined together by selected edges of the graph to form the final *MST* (see Lecture Slides and Lecture Notes on Greedy Algorithms). The algorithm takes $O(|E| \log |V|)$ time.

2. A second famous algorithm is **Prim's algorithm**, which is also a greedy algorithm. This was invented by Vojtěch Jarník in 1930 and rediscovered by Prim in 1957 and Dijkstra in 1959. Basically, it grows the MST (*T*) one edge at a time so that at any stage, the set of selected edges form a single tree. Its run-time is also $O(|E| \log |V|)$ (see Lecture Slides and Lecture Notes on Greedy Algorithms).

The Project

Your project is to build a **Minimum Spanning Tree Utility** for a connected, edge-weighted undirected graph.

As an application, a telecommunications authority wishes to lay cables in a new housing compound. It is constrained to bury the cables only along certain roads. This can be represented as a graph containing the houses connected by the cables (edges). Some of the edges might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Cable laying cost is an acceptable unit for edge weight.



The cable network can be represented by a graph with V nodes (houses) and the weights on the edges represent the cable laying cost between

pairs of houses. In this case, our problem would be to find a **Minimum Cost Spanning Tree (MST)** for the given network. Such a MST would be connected (i.e., there is a cable from every house to every other house) and the sum of weights of edges in that tree would be minimum (i.e., the cost of cable laying be minimum).

You are required to:

- Generate text files that contain the adjacency matrix representations for connected edge-weighted non-directed graphs for layouts with different number V of houses (say V = 8, 16 , 32 , 64). Each graph is connected and the weights are all positive integers in the range 20 – 100. Zero weight represents the absence of a road, or the distance between a house and itself. The houses can be numbered (0,1,2,…) or (A,B,C,….). The first number in the file is the number of houses (V). The following V numbers represent the first row in the adjacency matrix, followed by V numbers representing the second row, and so on.
- **Implement both Kruskal's and Prim's algorithms** for finding the MST for a graph.
- Using the above algorithms, find the MST's for the graphs generated.
- Compare the performance of the implemented algorithms (measures of performance are left to your discretion).

Notes on the design and implementation:

- You may use an adjacency matrix representation of the graph, i.e., a 2-D array of size V_{max} by V_{max} where V_{max} is the maximum number of vertices (e.g V_{max} = 100).

- An edge e = (u,v,w) has one vertex u, a second vertex v and a weight w (a positive integer). It can be represented as an object of a class "Edge". You may use a 1-D array of size E_{max} to store the non-zero edges in the graph, where E_{max} is the maximum number of possible edges = $V_{max}*(V_{max}-1)/2$. The actual number of such edges in the graph will be E so that the edges will be stored in locations (0 .. E-1).
- The vertices can be given numbers (0,1,….V-1) where V is the actual no. of vertices in the graph given in the file. These numbers can be mapped to names (e.g A,B,C…).

- In Kruskal's algorithm, you may use a PQ (Minimum heap) to insert edges into the heap and then remove one edge at a time. The algorithm may also use a "Sets" class for disjoint sets in order to test for cycles in the graph.

- The zip file "**Test_Kruskal.rar**" contains a test graph file **"testgraph.txt"** representing a graph of 7 vertices. You can test your implementation using this file. The sample output for that file is also given in file **"sample output.txt".**

- You may use the following algorithm to generate a random connected edge-weighted non-directed graph with positive weights and store it in a text file:

    1. Set number of vertices V
    2. Set minimum and maximum edge weights W_{min} , W_{max}
    3. Seed random number generator
    4. Generate a 2-D array A[0..V-1] [0..V-1] with random weights between 0 and W_{max} (you may also convert the weights to integer values here)
    5. Replace values less than W_{min} by zeros
    6. Put zeros on the diagonal A(i,i), i = 0..V-1

7. For each row above the diagonal, copy it to the corresponding column below the diagonal
   This is the final adjacency matrix
8. Reshape the 2-D array A into a 1-D array B of size V^2 (row by row)
9. In a text file, write the dimension V followed by the array B

Submission Instructions

Your source code and executable binary file(s) should be submitted on blackboard by the due date, along with a README file that describes how to use the utility.

Grading

The program will be tested against a sample graph. If your program works, it should be capable of producing the graph MST. In this case, grading will be on both the program and the project report.