KVPY SUMMER PROJECT REPORT 2013

Jay H. Bosamiya

KVPY SA 11110105

ABSTRACT

A stack with constant time push, pop, and findmin was implemented using linked lists and analysed. Binary and ternary Huffman tree generation for Huffman coding was implemented and analysed. Optimal solution to Travelling Salesman Problem (TSP) for complete graphs was implemented using branch and bound technique and analysed.

Project Advisor

Prof. Y. Narahari (Chairman, Computer Science and Automation, Indian Institute of Science, Bangalore)

Duration

25th June 2013 to 11th July 2013

Table of Contents

Introduction

The project was about the studying, understanding, designing, and implementing algorithms for a variety of situations.

The first situation involved designing an implementation for a stack using linked lists which, other than the usual push and pop operations, also must have a “findmin” operation that returned the minimum element in the stack at that point of time. All the three operations were to take constant time for execution.

The second situation involved a compression technique known as Huffman coding. By building Huffman trees, one could generate “Huffman codes” that better encode the data at hand.

The third situation involved optimally solving the famous Travelling Salesman Problem for complete graphs. The technique to be used is called branch and bound.

All the 3 sub-projects have been implemented in C++, following the standard strictly and following best programming practices.

Subproject 1

Linked List Implementation of a Stack with Constant Time Push, Pop, and Findmin

Objective

Suppose one is implementing a stack of integers using a singly linked list. It is trivial to implement push and pop in worst case O(1) time (i.e. constant time) by maintaining the top of the stack to be the first element in the linked list and the bottom of the stack to be the last element in the linked list. The objective is to include another operation called "findmin" which returns the minimum element in the stack. All the 3 operations (push, pop and findmin) must all run in O(1) time. An additional linked list may be used.

Naïve Solution

Implement two stacks using linked lists. Call them nStack and mStack (for "normal stack" and "min stack"). For each element pushed into nStack, compare with mStack's top element and push the smaller of the two into mStack. For each element popped from nStack, pop one element from mStack. For findmin operation, return top element in mStack.

Push() – Θ(1)
Pop() – Θ(1)
FindMin() – Θ(1)

Space used – Θ(N) for nStack, Θ(N) for mStack

Better Solution

Implement two stacks using linked lists. Call them nStack and mStack (for "normal stack" and "min stack"). For each element pushed into nStack, compare with mStack's top element and push the element into mStack only if it is lesser than or equal to the top element. For each element popped from nStack, pop the top element from mStack only if it is equal to the element popped from nStack. For findmin operation, return top element in mStack.

Push() – Θ(1)
Pop() – Θ(1)
FindMin() – Θ(1)

Space used – Θ(N) for nStack, O(N) for mStack

The solution is better since it requires O(N) space for mStack instead of Θ(N) since it requires space of N in worst case (reverse sorted numbers are pushed) but lesser space in any other case

Implementation

Attached in folder MinStack.

MinStack/main.cpp – Handles the user interface of the implementation; allows for usage of the operations

MinStack/MinStack.h – Declares and defines the MinStack class which is the actual implementation of the objective

MinStack/SinglyLinkedListStack.h – Declares and defines SinglyLinkedListStack class which is an implementation of a stack using a singly linked list.

Subproject 2

Huffman Coding

Objective

Huffman Coding is an entropy encoding algorithm that, given a set of alphabets and their relative frequencies, encodes each of the alphabets using variable length codes. This leads to lossless compression. The objective is to find the Huffman binary code using binary trees and an optimal ternary code using ternary trees for a set of relative frequencies for alphabets (denoted by 1, 2, 3… without loss of generality).

Solution for Binary Huffman Coding

Generate the Huffman tree by starting off with a forest of trivial trees (of the alphabets) ordered by relative frequency. Repeatedly combine the two trees with least relative frequencies by making them the children of the root of a new tree which has relative frequency equal to the sum of its children. Continue the above until only one tree remains. This is the optimal Huffman tree. By traversing down to the leaves, assign different symbols to siblings (0 for left sibling, 1 for right sibling). The concatenation of the symbols in the path from root to leaf gives the optimal Huffman code for the alphabet at the leaf.

Naïve Implementation

Store the forest in an array and traverse it to obtain the trees with least relative frequency.

Time taken per combine – O(N)
Number of combines – N
Total time for tree creation – $O(N^2)$ (because $\sum_{i=1}^{N} O(i)$)

Better Implementation

Store the forest in a priority queue and use it to obtain the trees with least relative frequency.

Time taken per combine – $O(\log_2 N)$
Number of combines – N
Total time for tree creation – O(N log N)

Solution for Ternary Huffman Coding

Do the same as Binary Huffman Coding except 3 trees must be combined and the symbols assigned should be 0 to left sibling, 1 to middle sibling and 2 to right sibling.

Implementation

Attached in folder Huffman.

Huffman/main.cpp – Handles the user interface of the implementation; allows for inputting the relative frequencies and displays the Huffman codes for the alphabets

Huffman/HuffmanCoder.h – Declares and defines the HuffmanCoder class which is the actual implementation of the objective

Subproject 3

Optimal Solution to TSP using Branch and Bound

Objective

The Travelling Salesman/person Problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? The naïve solution to solving it would be to enumerate all possible paths and choose the one with minimum path length. A better solution would be to prune away all paths that are necessarily sub-optimal even before enumerating them. This method is called Branch and Bound. The objective is to implement this technique and find the best TSP tour for a given set of cities (represented as a complete weighted graph).

Solution

For each city (vertex) to be visited exactly once, it is trivial to show that, in the TSP tour, there will be exactly two edges that have this vertex as the end point. Also, since the roads are bi-directional and have same distance both ways, we can assign one edge as incoming and other as outgoing for each vertex (without loss of generality). Now, consider the edges that are outgoing from each vertex. These will have weights that are equal to or greater than the smallest outgoing edge from that vertex. This means that we can obtain a lower bound on a TSP tour if even part of it is fixed. At each point of time, one can store the best TSP tour obtained so far and discard any partial tour whose lower bound is greater than (or equal to) the current best. In this way, multiple tours which are sub-optimal are discarded in one swoop without the need to enumerate each one of them.

It becomes useful to implement this using “best first” method of traversal of the state space. This can be done by storing each state as a node in a priority queue ordered by lower bound of the partial tour.

By pre calculating the shortest outgoing edge from each vertex, it becomes even more faster to calculate the lower bound on the partial tour.

When no more state space nodes remain in the priority queue, the optimal tour has been found.

Implementation

Attached in folder TSP.

TSP/main.cpp – Handles the user interface of the implementation; allows for either generation of random complete graphs or inputting complete graphs and then displays the optimal TSP path

TSP/constants.h – Defines constants that are used across the program and allows for easy modification of the program just by modifying these constants

TSP/Graph.h – Declares and defines the Graph data structure used in the program

TSP/OptimalPath.h – Declares and defines the OptimalPath data structure used in the program

TSP/tspSolve.h – Declares and defines the tspSolve() function that takes a Graph as input and gives the OptimalPath as output

Acknowledgements

I would like to thank KVPY for giving me the unique opportunity to do this project.

I would also like to thank Prof. Y. Narahari for guiding me through this project and giving such wonderful topics to work on.

References

[1] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

[2] The Algorithm Design Manual by Steve S. Skiena

[3] http://lcm.csa.iisc.ernet.in/dsa/

[4] http://www.stackoverflow.com/