1 Getting Prepared

Copy all of the files from \texttt{\textasciitilde{libsq/asst5}} into your directory \texttt{\textasciitilde{Q/asst5}} (which should have been created when you ran \texttt{q-setup} for Assignment 1).

2 Exercises (100 points)

All work should be done in directory \texttt{\textasciitilde{Q/asst5}}. The \texttt{Makefile} provided includes rules to build all the programs in this assignment; you shouldn’t need to change it at all.

Answers to the written problems should be written out by hand.

1. (20 points) Complete the \texttt{graphConnect} function in \texttt{graph_con.c}. Analyze the running time of your solution, in terms of the size and order of the graph, using big-O notation.

When you are finished, you can build and run \texttt{graph_test} to test your program, or run \texttt{test_con} (a script that runs \texttt{graph_test} on several graphs).

2. (25 points)

(a) Weiss 5.9. (You may find it useful to read through exercise 5.8 before starting 5.9.)

Computing the actual probability in part d of this problem is a very tricky business (especially since we haven’t discussed probability at all), because it depends upon the nature of spelling errors and the actual hash function. To make this problem reasonable, you should assume that all possible misspellings are equally likely (i.e. “qqqqq” is just as likely as “thier”), and the hash function does a “perfect” job; the probability that two randomly chosen words will have the same hash value is $1/300,007$.

(b) Implement this scheme, by completing the functions in \texttt{hash_bit.c}. This completes the implementation of \texttt{hash_test}, a program that reads the standard UNIX dictionary into a bit hash table, and then counts how many bits are actually set.$^1$

When you are finished, you can build and run \texttt{hash_test} to test your functions.

3. (25 points) One approach to dealing with the problem of unbalanced binary search trees is to permit the tree to become unbalanced after some operations (instead of making sure that it is balanced after every operation), and rebalance it only after it becomes notably unbalanced according to some metric. (Note— this approach to balanced trees is not generally effective, although it can be useful in some situations. We do not recommend it as a general technique.)

Devise an algorithm to create a complete binary search tree, given a binary search tree containing \(n\) nodes, in time \(O(n)\), and using \(O(n)\) extra memory.

Recall that a complete tree is a binary tree where all the leaves are at depth \(d\) or \(d - 1\), and all the leaves at depth \(d\) are to the left of all of the leaves at depth \(d - 1\).

The algorithm might be difficult to state correctly for all cases, so please include a detailed description of the strategy and ideas behind your algorithm, so we can understand what you are trying to do even if some of the details are wrong.

$^1$If you browse through the standard UNIX dictionary, you will notice that this dictionary, when used with this scheme, would give a very bad spell-checker— it doesn’t include many common prefixes and suffixes. Spell-checkers typically synthesize by applying rules to a set of root words, instead of trying to keep track of every derivative spelling of every word. The heuristics and algorithms that these spell-checkers use are very interesting, but completely beyond the scope of this problem.
Hint— if \( n \) is exactly \( 2^{d+1} - 1 \), then a perfect binary search tree can be constructed. Try constructing (by hand) perfect binary search trees containing the integers 1 through 3, then 1 through 7, and 1 through 15, and see if a pattern emerges.

4. (15 points) Prove that any DAG (directed acyclic graph) containing \( N \) vertices can contain \textbf{at most} \( \frac{(N-1)N}{2} \) edges.

5. (15 points) Imagine that you have a network-like structure, with the following modification— each \textit{vertex} has a cost associated with it, instead of the edges. The cost of a path through this structure is the total cost of the vertices visited by the path (including both the start and end vertex), instead of the cost of the edges in the path.

Devise an algorithm for computing a minimum cost path from one vertex to another on this structure.

(As an aside - computing the minimum spanning tree for such a structure is easy; any spanning tree will do, since any spanning tree will, by definition, include all of the vertices, and so incur the same cost no matter which edges are selected.)

3 \textbf{Graduate Problem (20 points)}

A graph \( G = < V, E > \) is \textit{bipartite} if \( V \) can be divided into two disjoint partitions \( V_1 \) and \( V_2 \) such that there are no adjacent pairs of vertices within either \( V_1 \) or \( V_2 \). Each edge in the graph (if any) is between a vertex in \( V_1 \) and a vertex in \( V_2 \).

- Give an efficient algorithm for determining whether a graph is bipartite, and if so how \( V \) can be partitioned. (The partitioning is not necessarily unique; there may be many possible partitionings, but your algorithm need only find one of them.)

Note that your algorithm need work only for undirected graphs, not directed graphs.

Hint— if there is a spanning tree for a bipartite graph, what property \textit{must} this spanning tree have? (Note that a graph can be bipartite even if it does not have a spanning tree, however.)

- Give an analysis of the running time of your algorithm, using big-O notation, in terms of the size and/or order of the graph. If your analysis makes any assumptions about the representation of the graph (as adjacency lists or adjacency matrix, etc), be sure to mention them and what role they play in your analysis.

4 \textbf{Submit Your Work}

1. Use the \texttt{submit} program to hand in your work. From your directory \(~/\texttt{q}\), run the \texttt{submit} program. The name of the course is \texttt{cs315}, and the number of the project is \texttt{5}, and the name of the directory to submit is \(~/\texttt{q}/\texttt{asst5}\). (See the UNIX tutorial for more info about \texttt{submit}.)

2. Print out a copy of your \texttt{graph_con.c} and \texttt{hash_bit.c}, and hand them in during the next class or section. (If you modify or create any other files, please print them out as well.)