CMPT 409 Assignment 1

Heather Li, Ekjot Singh Billing, Manshant Singh Kohli

January 19, 2018

1 Erdos Numbers

(C++11) In this question, you could model the relationships between the authors and the papers as an unweighted graph, where the vertices of the graph would be authors and their would be an edge between any two vertices if they had written a paper together. The Erdos number of an author would be the length of the shortest path between the vertex representing Erdos and the author. This could be easily accomplished by running a BFS starting at the node represented by Paul Erdos.

We chose to represent our graph as an adjacency list. We processed the input given regarding the authors and their papers. Each author's adjacency list (reprenting people they had co-authored a paper with) was stored as an unordered set. The names of every author were used as a key in a hash table (as an unordered map in C++), which stored their adjacency list and their Erdos number (initialized to 0).

Once we had processed all the input and inserted it into adjacency lists, we ran a BFS starting at the "P. Erdos" adjacency list with an Erdos number of 0. As each one of Erdos' co-authors were visited, they were marked as visited assigned a number greater than 1 and added to a queue. As we continued through the BFS, we added authors to the queue if they had not yet been visited and assigned them an Erdos number 1 greater than the neighbour from which they were referred.

After this processing, we had the Erdos number of every author listed in the original input. We could then answer Erdos number queries for authors - if an author had an Erdos number of 0, this would be because they were never visited by the Erdos BFS, at which point we could say their Erdos number was infinity.

2 Poker Hands

(C++ 11) For this question, we stored the 5 cards as a 64-bit integer. The 8 most-significant bits of this integer were used to denote the different hands the cards could be scored by (e.g. straights, flushes, pairs, etc.). We used the remaining bits as tiebreakers (i.e. if the highest hand our 5 cards contained was a pair, we would store the value of the pair).

During input, we preprocessed all 5 cards, before storing them in our 64-bit integer. If all the input cards had the same suit, we marked the hand as having a flush. Aside from this, suits do not matter to the problem, as it is stated that no suit is ranked higher than another. Having done so, we could discard the suit data and now only consider the denominations. The cards' denominations were stored in an i with 13 entries, where each entry $v[i]$ determines the number of cards with denomination $i + 2$ We checked for straights using our v vector by seeing if there are 5 consecutive entries in v with values of 1.

From this data, we could start populating our 64-bit integer. We stored the highest value hand that our 5 cards represented, along with its tiebreaking characteristic. We could modify the bits in our integer through masking and bitshifts.

Once we had transformed the white and the black hands into integers, we could compare the two integers to see which one would win.

3 Flea Circus

(C++) We solved this problem by modelling the fleas and the branches between them vertices and edges of a tree. There is exactly one path between any two vertices on a connected tree.

We created a tree, which we implemented as an adjacency list from the input describing the tree. The adjacency list for each node was referred to be a pointers in an array of size n, the size of the tree. For each set of fleas (a, b), we ran a modified depth-first search (implemented with recursion) starting from a flea at position a to a flea at position b. The path taken was recorded in an array. Once the target location of flea b was reached, we checked the middle position(s) in the array and outputted them.

4 Check the Check

(C++) We stored the input board as an 8x8 character array. Once we had collected all of the input values, we iterated through the board. If we received a non-null chess piece, we would check along its lines of movement (e.g. check all diagonals for a bishop). If we ran into a king of the opposite colour, we would return a flag signifying that either the white/black king was under check. If we ran into another chess piece piece before we ran into the king/ran into the end of the board, we would simply start scanning another direction. If we checked every position and no checks were found, we would return a tie.

We used helper arrays to describe the directions chess pieces could take and another helper function to determine when a position was in range. Note that we did not need to check kings as moving a king so it is adjacent to another king is an illegal move in chess. (If both kings were adjacent to each other, they would both be in check, something which is prohibited by the problem parameters).