Name: Brian Jon Stout
Date: 20FEB2017
Current Module: Data Structures and Algorithms 2
Project Name: zergmap

Project Goals:

The goal of the zergmap program is to take any amount of zerg packets, parse them for information, store the data appropriately, then to create a graph based around the distance between the zerg unit (derived from longitude and latitude) and remove any units from the graph which does not have two unique routes to any other unit in the graph. After this the program should print out a list of zerg that are underneath a specified health percentage

Considerations:

- The program must be able to sanitize data from a file, and recover from errors to avoid crashing
- The program must be able to tell the difference between IPV6 and IPv4
- The program needs to solve the graph with the least amount of changes possible
- - options must be allowed in the program

Initial Design:

zergmap.c contains the most basic program flow, and maintains the data structure data between function calls. input.h and input.c are responsible for parsing the binary data from the user input and grabbing the relevant data. Input also contains information on the basic zerg structure. Output.c and output.h are responsible for printing out the end information. Tree.c and tree.h contain a AVL tree data structure which stores zerg units and consolidates repeat information, as well as holds the zerg status information the zerg graph doesn't need. List.c and list.h created a list data structure that acts as an intermediary between the graph structure and the tree structure. An unpack function grabs the nodes from the tree and only inserts them into the list if it contains GPS data. The list contains a count of the members which is important for the graph to create is matrix table. Graph.c and graph.h contain all the programming relating to generating the graph, and solving it. Binary.c and binary.h contain functions related to handling ntoh calls for oddly sized packet fields, as well as converting uint_t data types to doubles.

Data Flow:

The program starts by sending argc and argv into an argument_checker function that stops the program if the command line arguments are parseable. It then transitions into the getopt() portion of the program which evaluates if there's a -h option and it's followed by a valid input (>0 < 100)

After this the tree data structure is started and the program opens up each file one by one, running it through the read_pcap_packet function which returns a zerg_unit structure based on the data read. The zerg_unit is inserted into the tree and sorted based on the zergID.

After all the zerg packets are read the program calls an unpack_tree function which generates a list structure, and inserts each node of the tree into the list if it contains GPS information.

The list structure is passed to the create_graph function. In the create_graph function it goes through each member of the list and generates storage for the zerg unit stored there into a table for quick access. Afterwards it finds the distances between each zerg unit based on the haversine function and generates a adjacency matrix table based on those differences.

After the graph is created it's based to the cleanup_graph function which finds all leafs (nodes with only one adjacency) and gets rid of them. After this the graph is checked for weaknesses in the check_for_weakness() function. It checks any node with more than 2 adjacencies, and runs routes from every point in the graph which can't touch that node. If it can't find a route it's declared a weakness. The function deletes an adjacent node and the cleanup function is ran again.

The program quickly checks to see if the graph has deleted too many nodes. If it has it declares the graph unsolvable and stops the program.

If it isn't, then it prints out all the zerg units that have been deleted as suggested removals.

It then prints out any zerg units that are below a certain health.

Communication Protocol:

Not applicable.

Potential Pitfalls:

Choke points are the biggest pitfalls and create the most problems with the program.

Test Plan:

User Test:

Two nodes connected

Three nodes connected

Four nodes connected, with one as a leaf

The bow tie

Providing bad user input.

Test Cases:

All the base cases are solvable, though the bow tie solution could easily be broken. Also the choke point solution doesn't really resolve which side is greater and could delete the wrong side breaking the solution.

Conclusion:

The project's main challenge was finding a solution for choke points. The complexity quickly escalated and which out a good base code the program could easily propagate into even worse code, and break further (it definitely did for me.) The project definitely demonstrates the necessity to have a strong base on data structures, and having module and clean code prepared already for basic data structures, something I'm definitely going to work on during our 5 day break.