**Hot Potato Routing vs. Ant-Colony Routing**

**By: Michael Burwell, Caleb Landis, Tristan Kolat, and Logan Kurker**

**Abstract**

Our project this time around has a similar objective in mind. We are still following the routine where we pass messages through a graph of nodes until they reach their final destination. The difference this time lies in our inclusion of a new algorithm to run this system, namely the Ant-Colony Routing Algorithm. This algorithm uses graphs considered a directed graph, wherein each edge of the node has an associated direction. Each edge has its own “pheromone”, and the pheromone is used within an equation to calculate the probability that the path forward will follow a good path. This is given by the equation: pj = (φkj)n/ΣiENk(φkj)n. The “good path” that this equation calculates essentially gives the probability that the edge that the message will pass through has been passed by many “ants” beforehand. If many ants have passed through, then theoretically this means that it is a good path to follow since it is more likely that the message will reach its destination quicker. From our analysis, we found that the ant-colony routing algorithm worked better than the hot-potato algorithm under certain circumstances. When the graph we used contained nodes with many different edges, it helped if the dilution period, or period in which the pheromones for each edge wears off given time, is higher, then the ant-colony algorithm worked much quicker than the hot-potato algorithm. However, with graphs that contained less edges, the ant-colony algorithm worked better when there was a lower dilution period.

**Methods**

Project four made use of several classes from project three that helped us implement the “hot-potato” routing algorithm. The classes we used for the last project solved a lot of the core issues that were still needed for implementation of the “ant-colony” algorithm, with most of these classes remaining relatively unchanged. These classes included *ThreadGraph, ThreadNode, MessagePacket, TimeInterval,* and *Mailbox.* Although they were described in much greater detail in our report for project three, a quick overview of how the classes work and how they relate to this project will still be useful.

The *ThreadGraph* class is responsible for reading in the graph text file (.dat) containing information for a graph of nodes and populating that data into a data structure that will be useful for initializing each node’s neighbors when creating its threads. This translates into *ThreadNode* where each node’s thread will begin running the operation for the selected algorithm at runtime. This class is where most of the work of project four is displayed, as it not only operates “hot-potato” routing, but “ant-colony” routing as well, and it will be further described in the following paragraphs. Both routing algorithms needed a way to keep track of each message’s travel time, number of hops between nodes, its final destination, and which nodes the message should be sent to and received from, so a *MessagePacket* class was designed as an abstract data type to be used as the data for the message passing. The *TimeInterval* class was created in order to keep track of how long each message packet was in routing. Finally, we also used the *Mailbox* class for sending and receiving messages as well as storing them while they wait to be received.

In project three, we used the *driver.cpp* file for organizing the process of reading from the graph, initializing the threads for each node, joining the threads, and ultimately outputting the data collected for analysis. This same process was needed for project four, now with the added functionality of being able to choose which algorithm will run on runtime. The driver will now take in command line arguments for the duration of how long the simulation will send messages, which algorithm will be used, as well as which graph will be used for routing. Upon instantiation of each *ThreadNode* that corresponds to a node in the graph, each node will create a thread that runs the instance’s *run()* method to begin the sending of messages for the specified duration and consequently the receiving of them based on the chosen algorithm.

Last project, every thread would create a set number of messages to be sent to random destinations, and they would all be sent concurrently. In order to make it so that threads would not overlap in execution, we implemented a staggering algorithm where each thread will wait a fixed interval that allotted enough time in between job executions while still completing all threads’ operations in a timely manner. One of the tasks we were assigned was to determine the shortest possible fixed interval to satisfy this need while making use of the algorithm given to us. This algorithm has each node sleeping an initial amount of time every 𝞴 seconds where 𝞴 was the time that we have determined optimal for this. This is done upon calling each *ThreadNode’s run()* method, where it will then send messages for the specified duration.

Up until the creation of each message, whichever algorithm was chosen had not had an effect on the flow of the program. The process of reading the graph, initializing threads for each node, and creating messages was the same despite which algorithm had been chosen, but that is because neither of the algorithms had been needed yet since they deal specifically with routing the message. When a message needs to be passed, it will need a receiver which may or may not be its final destination. With the “hot-potato” algorithm, the receiver is randomly chosen among its neighbors minus the one it came from, if any, as implemented in the last project. Our implementation of this algorithm has not changed for this project, except for the fact that upon creation of the message packet, this algorithm may not even be used depending on which of the two algorithms were selected. Now, sending a message will react to determine a receiver based upon the chosen algorithm.

If the message passing is “ant-colony” routing based, *ThreadNode* will run the *findTrail* method which will ultimately return which node will be the receiver of the message. This method will first check if the final destination is one of the current node’s neighbors so it can just directly send the message there. Otherwise, a random receiver among the node’s neighbors will be chosen under the influence of each edge’s pheromone. This is done through *getPheremoneNeighbor*() which is a static method through our new *RandomNodes* class that was created to hold this and other static methods that return random values for uses such as random sleeping times and of course choosing a random neighbor. Choosing a random neighbor based on pheromones happens by taking each available edge’s pheromone and using that to calculate a probability that the node may be chosen as a receiver through a random discrete distribution value. This probability is useful because a stronger pheromone will indicate that this trail is taken by “ants” more frequently as the path is more likely to not lead to a node where there is only one singular neighbor - the one that the ant just came from.

With the receiver of the message chosen, the message packet will be sent and received using *Mailbox*. Within *ThreadNode* a second thread for the node was created for receiving. This thread runs *thread\_recv()* to keep receiving messages until all of them have reached their final destination. Every time a message is received, it will check how to handle routing based on the selected algorithm as described above, with the “hot-potato” algorithm still handling this in the same way as the last project. When an “ant-colony” routed message is received, it will increment the pheromone for the trail just taken by a constant value that we were to determine. In order to make sure no pheromone became too strong, we also dilute the same trail exponentially based on time using a half-life constant as another simulation parameter that we had chosen. Regardless of the chosen algorithm, the message will then decide if this node is the final destination; if not, the same process for choosing a new receiver and forwarding the message will continue. Once all messages have been received, the *driver* will join all threads and output the data collected as described earlier.

**Results**

In the last project we found that the hop counts and time it takes a message to pass through a hot potato algorithm is directly related to the ratio of edges compared to nodes. To find a good competitor to our “ant colony” algorithm we started testing the graphs that had the fastest times, or highest ratio of edges per node. These graphs were A19, A21, A37 and A46. This gave us a baseline to test our new algorithm.

Running multiple tests at a duration of 3 seconds on graph A19 gave a start for fine tuning our algorithm. Our “ant colony” algorithm does not allow for a single edge to get lower than the initial pheromone assignment. This allowed us to make the Initial pheromone a constant to test around. Our constant chosen was 42.

Our first rounds of testing were on the increment variable. This variable increments every time a node receives a message from another node. The receiver node increments the edge that the sender sent from by the increment variable. We began testing numbers greater than our initial pheromone of 42 and quickly found that this increased the time and sometimes would set “grooves” in the graph that neglected other nodes less traveled. A much lower dilution half-life would fix this problem, but we wanted to have a higher dilution half-life so the paths would dilute slowly over time. A high number of increments and low dilution half-life would not have a large difference with the “hot potato” algorithm, in fact it would imitate it. So we choose to lower our increment variable and raise our dilution half-life variable. After many tests we were able to land on a value of 9.5 for increments and a half-life of 3000 milliseconds. This allowed us to raise the power coefficient so that the small increments would have large consequences on the probabilities of choosing a path. Our final numbers after the first round of testing on graph A19 where (42, 9.5, 2.9, 3000). These values allowed our “ant colony” algorithm to reliably beat our “hot potato” algorithm, and sometimes reach twice as fast finish times.

Our next round of tests were to focus on the graphs that the “hot potato” algorithm had a hard time with. These graphs have a lower ratio of edges per node and may be a problem for our “ant colony” algorithm, considering as it travels it wears down groves on heavily trafficked areas. We began testing on A20. This graph has only one more node than A19 but much less edges. Our initial tests were either not able to finish or not really much better than “hot potato”. The simple fix was to lower the dilution half-life to allow for the edges to dilute a little bit more. After many tests we found that a half-life of 2500 ms was a good medium between the two types of graphs. This simple change allowed for our “ant colony” algorithm to consistently beat our “hot potato” algorithm at 1.5 to over 2 times faster on the graphs with a low ratio of edges per node.

The final numbers that we landed on for the “ant colony” algorithm are (42, 9.5, 2.9, 2500). Initial Pheromone of 42, pheromone increment constant of 9.5, power coefficient of 2.9, and a dilution half-life of 2500 milliseconds.

**Discussion**

Based on our results, we found that changing the dilution of the half-life and changing the base value of the pheromones had a significant impact on the effectiveness of our ant-routing algorithm. In the case of graphs that had less paths to take due to a lower amount of edges, it makes sense that it would be more difficult for our “ants” to have a harder time finding the optimal route. Because of this, it was important for us to dilute our edges more often so that the “ants” could reroute and find better paths instead of being stuck evenly between the limited number of paths already available to them. To that end, on the other spectrum it also makes sense for the graphs with a higher number of routes to not dilute our paths nearly as much.

In the case of graphs A19, A21, A37, and A46, these graphs had a large number of paths that the ants could potentially take. Because of this, it would become easy for the optimal paths to be found for the ants to travel. In this case, it makes sense that lowering the dilution for these paths would be detrimental to our runtime. If the optimal path is found, then we don’t want our messages straying from that path. Otherwise, they could easily get lost within the high number of edge options that each message could take. This would defeat the whole purpose of the ant-routing algorithm, essentially turning it into the hot-potato routing algorithm. As a result, it was important for us to find the correct parameters when running the program.

It should be noted that the ant-routing algorithm is not fool-proof. It does not always guarantee that it will run faster, as in the case of the graphs with a lower amount of edges. In that case, the algorithm had to be fine-tuned. The ant-routing algorithm simply increases the likelihood of making it to the correct destination, but not necessarily guaranteeing it. It should also be noted that with the ant-colony routing, we must take precaution in ensuring that the cooldown duration is sufficiently long. What we mean by this is as follows: in the case where the ants find a good route for the messages to be passed to, this will significantly increase the traffic that these nodes experience. As a result, it is important that the threads do not overlap with this increased traffic, so the value of 𝞴 might need to be higher with this algorithm in use. Despite this fact, we still found that the ant-colony algorithm was better. Whether or not this remains the case if the graphs were to be larger or the number of threads higher is something we could test in future studies.

**References**

Kolat, T., Landis, C., Kurker, L., & Burwell, M. (2023, March). CSCI403\_Project4.

Retrieved April 30th, 2023, from<https://github.iu.edu/callandi/CSCI403_spring23_P3>.

**Appendix**

**Appendix A: Driver**

| #include <iostream> |  |
| --- | --- |
|  | #include <fstream> |
|  | #include <thread> |
|  | // #include <condition\_variable> |
|  | #include <string.h> |
|  | #include <typeinfo> |
|  | // #include <future> |
|  | #include "src/ThreadGraph.h" |
|  | #include "src/ThreadNode.h" |
|  | // #include "src/MessagePacket.h" |
|  | // #include "src/mbox.h" |
|  |  |
|  | #define MAX 1024 |
|  |  |
|  | struct analysis{ |
|  | unsigned int hops; |
|  | double time; |
|  | }; |
|  |  |
|  | struct options{ |
|  | unsigned int optionD; |
|  | std::string optionR; |
|  | std::string filename; |
|  | }; |
|  |  |
|  | void analyzeResults(std::vector<ThreadNode> nodes, analysis &results); |
|  | void createNodes(std::vector<ThreadNode> &nodes, ThreadGraph graph, unsigned int duration); |
|  | void runThreads(std::vector<std::thread\*> &t, std::vector<ThreadNode> &nodes, std::string algName); |
|  | void joinThreads(std::vector<std::thread\*> &threads); |
|  | options getOptions(int argc, char \*argv[]); |
|  |  |
|  | int main(int argc, char \*argv[]){ |
|  |  |
|  | options parameters = getOptions(argc, argv); |
|  |  |
|  | // output stream for storing the graphs analysis |
|  |  |
|  | // std::ofstream analysisFile; |
|  | // analysisFile.open("Analysis.csv"); |
|  | // analysisFile << "Graph, Nodes, Edges, Hops, Node-Edge Ratio, Time, \n"; |
|  |  |
|  |  |
|  | ThreadGraph graph(parameters.filename); |
|  | std::vector<ThreadNode> nodes; // Node objects |
|  | std::vector<std::thread\*> threads; // Threads |
|  | analysis results = {0,0}; |
|  |  |
|  | std::cout << "\nTest - (" << parameters.filename << ")" |
|  | << " - duration - (" << parameters.optionD << "s)" |
|  | << " - Algorithm - (" << parameters.optionR << ")" << std::endl; |
|  |  |
|  | if(parameters.optionR == "ant"){ |
|  | std::cout << "Initial Pheromone - \t\t" << INIT\_PHEROMONE |
|  | << "\nPheromone Increment Constant - \t" << INCR\_PHEROMONE |
|  | << "\nPower Coefficient - \t\t" << POWER\_COEFF |
|  | << "\nDilution Half Life - \t\t" << DILUTION\_HALF\_LIFE << std::endl; |
|  |  |
|  | } |
|  |  |
|  | createNodes(nodes, graph, parameters.optionD); |
|  | runThreads(threads, nodes, parameters.optionR); |
|  | joinThreads(threads); |
|  | analyzeResults(nodes, results); |
|  |  |
|  | // print the totals for the graph and reset the static counting variables for the |
|  | // group of threads. |
|  | std::cout << "\nMessages Sent: " << ThreadNode::getMessagesSent() |
|  | << "\nMessages Received: " << ThreadNode::getMessagesReceived() << std::endl; |
|  | std::cout << "Graph (" << parameters.filename |
|  | << ") - Total Hops -> " << results.hops |
|  | << " - Total Time -> " << results.time << " ms\n"; |
|  |  |
|  | // analysisFile << filename << ", " |
|  | // << graph.getNumNodes() << ", " |
|  | // << graph.getNumEdges() << ", " |
|  | // << (double)graph.getNumEdges() / graph.getNumNodes() << ", " |
|  | // << results.hops << ", " |
|  | // << results.time << ", \n"; |
|  |  |
|  |  |
|  |  |
|  | // analysisFile.close(); |
|  |  |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | void createNodes(std::vector<ThreadNode> &nodes, ThreadGraph graph, unsigned int duration) |
|  | { |
|  | for (uint16\_t i = 0; i < graph.getNumNodes(); i++) |
|  | { |
|  | ThreadNode temp(i, graph.getNeighbors(i), graph.getNumNodes(), duration); |
|  | nodes.push\_back(temp); |
|  | } |
|  | } |
|  |  |
|  | void runThreads(std::vector<std::thread\*> &threads, std::vector<ThreadNode> &nodes, std::string algorithmName) |
|  | { |
|  | // Use a thread for each of the nodes to use the nodes run function. |
|  | std::vector<ThreadNode>::iterator it; |
|  | for(it = nodes.begin(); it != nodes.end(); it++){ |
|  | threads.push\_back(std::move(new std::thread(&ThreadNode::run, &(\*it), algorithmName))); |
|  | } |
|  | } |
|  |  |
|  | void joinThreads(std::vector<std::thread\*> &threads) |
|  | { |
|  | // Makes main wait for each of the nodes to be joinable |
|  | for(int i = 0; i < threads.size(); i++){ |
|  | if(threads[i]->joinable()) |
|  | threads[i]->join(); |
|  | delete threads[i]; |
|  | threads[i] = nullptr; |
|  | } |
|  |  |
|  | } |
|  |  |
|  | void analyzeResults(std::vector<ThreadNode> nodes, analysis &results) |
|  | { |
|  | // reset the results if they are not already set |
|  | results.hops = 0; |
|  | results.time = 0; |
|  |  |
|  | // loop for storing the values for the total hopcount and total time |
|  | // also prints each nodes hopcount and time to completion |
|  | for(int i = 0; i < nodes.size(); i++){ |
|  | // std::pair<unsigned int, double> p = {nodes[i].getHopCount(), nodes[i].getTotalTime()}; |
|  | results.hops += nodes[i].getHopCount(); |
|  | results.time += nodes[i].getTotalTime(); |
|  | // std::cout << "Node (" << i << ") - Hop Count -> " << nodes[i].getHopCount() << " - Time -> " << nodes[i].getTotalTime() << " ms\n"; |
|  | } |
|  | } |
|  |  |
|  | options getOptions(int argc, char \*argv[]) |
|  | { |
|  | options parameters; |
|  | parameters.filename = "./graph/A10.dat"; |
|  | parameters.optionD = 10; |
|  | parameters.optionR = "hot"; |
|  |  |
|  | for(int i = 0; i < argc; i++){ |
|  | std::string temp = argv[i]; |
|  | if(temp == "-d"){ |
|  | parameters.optionD = std::stoi(argv[i + 1]); |
|  | } |
|  | if(temp == "-r"){ |
|  | std::string temp2 = argv[i + 1]; |
|  | if(temp2 == "hot" || temp2 == "ant"){ |
|  | parameters.optionR = temp2; |
|  | } |
|  | else{ |
|  | throw std::invalid\_argument("-r option needs to be -r <hot|ant>"); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | if(argc > 0){ |
|  | std::string temp = argv[argc - 1]; |
|  | if(temp.find(".dat") > temp.length() || temp.find("./graph/A") > temp.length()){ |
|  | throw std::invalid\_argument("filename needs to follow the structure: ./graph/A[10:50].dat"); |
|  | } |
|  | else{ |
|  | parameters.filename = temp; |
|  | } |
|  | } |
|  |  |
|  | return parameters; |
|  | } |

**Appendix B: ThreadGraph Class**

| #ifndef \_THREAD\_GRAPH\_H |  |
| --- | --- |
|  | #define \_THREAD\_GRAPH\_H |
|  |  |
|  | #include <iostream> |
|  | #include <fstream> |
|  | #include <string> |
|  | #include <queue> |
|  | #include <map> |
|  | #include <vector> |
|  | #include <stdint.h> |
|  |  |
|  |  |
|  | class ThreadGraph |
|  | { |
|  | private: |
|  | uint16\_t numNodes; |
|  | uint16\_t numEdges; |
|  | typedef std::vector<uint16\_t> neighbors; |
|  | std::map<uint16\_t, neighbors> graph; |
|  |  |
|  | public: |
|  | ThreadGraph(std::string); |
|  |  |
|  | uint16\_t getNumNodes() const; |
|  | uint16\_t getNumEdges() const; |
|  | neighbors getNeighbors(uint16\_t index) const; |
|  | std::string nborsToString(uint16\_t index) const; |
|  | }; |
|  |  |
|  | #endif |

|  | #include "ThreadGraph.h" |
| --- | --- |
|  | #include <typeinfo> |
|  |  |
|  | ThreadGraph::ThreadGraph(std::string filename) |
|  | { |
|  | std::ifstream file; |
|  | std::queue<std::string> nodeList; |
|  |  |
|  | try{ |
|  | file.open(filename); |
|  |  |
|  | // loop through file |
|  | while(!file.eof()){ |
|  | std::string line; |
|  | std::getline(file, line); |
|  | nodeList.push(line); |
|  | } |
|  |  |
|  | file.close(); |
|  | } catch(...){ /\* It'll just re-throw from here and be someone else's problem \*/} |
|  |  |
|  |  |
|  | // scrape the node list that is full lof string and need to be turned into |
|  | // usable integer indexes of node names |
|  | while(!nodeList.empty()){ |
|  | // removes the last element of the queue because of that pesky newline character that |
|  | // getline gives at the end. |
|  | if(nodeList.size() == 1){ |
|  | nodeList.pop(); |
|  | continue; |
|  | } |
|  |  |
|  | // create a temp |
|  | std::string temp = nodeList.front(); |
|  |  |
|  | // find the first two rows that include the number of nodes and the number of edges |
|  | // in the graph. with the "#" signifier and find the words "nodes" or "edges" to |
|  | // figure out which line we are on |
|  | if(temp.at(0) == '#'){ |
|  | if(temp.find("nodes") <= temp.length()) |
|  | // split the string by looking for the "= " sign wich is at the end and take everything |
|  | // after it |
|  | this->numNodes = std::stoi(temp.substr(temp.find("= ") + 2, (temp.find("= ") + 2) - (temp.length() - 1))); |
|  | else if (temp.find("edges") <= temp.length()) |
|  | // same as numnodes |
|  | this->numEdges = std::stoi(temp.substr(temp.find("= ") + 2, (temp.find("= ") + 2) - (temp.length() - 1))); |
|  | } |
|  | else{ |
|  | // split from the ":" sign |
|  | if(temp.find(':') <= temp.length()){ |
|  | // store the node ID number before the ":" |
|  | std::string nodeID = temp.substr(0, temp.find(':')); |
|  | // store everything after the ":" sign |
|  | std::string nbors = temp.substr(temp.find(':') + 1, temp.find(':') - (temp.length() - 1) ); |
|  |  |
|  | // loop through everything after the ":" sign neglecting commas "," and store in the graph as |
|  | // integers |
|  | std::string node = ""; |
|  | for(int i = 0; i < nbors.length(); i++){ |
|  | if(nbors.at(i) != ',' && i != nbors.length() - 1){ |
|  | node += nbors.at(i); |
|  | // this->graph[std::stoi(nodeID)].push\_back(uint16\_t(nbors.at(i) - '0')); |
|  | } |
|  | else if (i == nbors.length() - 1) |
|  | { |
|  | node += nbors.at(i); |
|  | this->graph[std::stoi(nodeID)].push\_back(std::stoi(node)); |
|  | } |
|  | else{ |
|  | if(node.length() > 0) |
|  | this->graph[std::stoi(nodeID)].push\_back(std::stoi(node)); |
|  | node = ""; |
|  | } |
|  | } |
|  | } |
|  | } |
|  | nodeList.pop(); |
|  | } |
|  | }; |
|  |  |
|  | // getters for the graph. |
|  | uint16\_t ThreadGraph::getNumNodes() const {return this->numNodes;}; |
|  | uint16\_t ThreadGraph::getNumEdges() const {return this->numEdges;}; |
|  | ThreadGraph::neighbors ThreadGraph::getNeighbors(uint16\_t index) const {return this->graph.at(index);}; |
|  | std::string ThreadGraph::nborsToString(uint16\_t index) const |
|  | { |
|  | ThreadGraph::neighbors nbors = getNeighbors(index); |
|  | std::string info = ""; |
|  | std::vector<uint16\_t>::iterator it; |
|  | for(it = nbors.begin(); it < nbors.end(); it++){ |
|  | info += std::to\_string(\*it) + ","; |
|  | } |
|  |  |
|  | return info; |
|  | } |

**Appendix C: ThreadNode Class**

| #ifndef \_THREAD\_NODE\_H |  |
| --- | --- |
|  | #define \_THREAD\_NODE\_H |
|  |  |
|  | #include <thread> |
|  | #include <mutex> |
|  | #include <condition\_variable> |
|  | #include "Node.h" |
|  | #include "RandomNodes.h" |
|  |  |
|  |  |
|  | #define MAX 1024 |
|  | #define COOL 10 |
|  | #define EXECUTION\_CYCLE 500 |
|  |  |
|  | #define INIT\_PHEROMONE 2.50 |
|  | #define INCR\_PHEROMONE 5.20 |
|  | #define POWER\_COEFF 2.13 |
|  | #define DILUTION\_HALF\_LIFE 600 |
|  |  |
|  | using namespace std::chrono; |
|  |  |
|  | class ThreadNode : public Node |
|  | { |
|  | public: |
|  | ThreadNode(); |
|  | ThreadNode(uint16\_t id, std::vector<uint16\_t> neighbors, uint16\_t totalNodes, unsigned int duration); |
|  | ThreadNode(const ThreadNode& other); |
|  | ~ThreadNode(); |
|  |  |
|  | void run(std::string algName); |
|  | std::map<uint16\_t, double> getEdges() const; |
|  | std::map<uint16\_t, time\_point<high\_resolution\_clock>> getEdgeTimes() const; |
|  | unsigned int getDuration() const; |
|  | static int getMessagesSent(); |
|  | static int getMessagesReceived(); |
|  |  |
|  |  |
|  | // unsigned int getMaxMsgs() const; |
|  | // unsigned int getHopCount() const; |
|  | // double getTotalTime() const; |
|  |  |
|  | private: |
|  | // TimeInterval \_thread\_start\_time; |
|  | // static time\_point<high\_resolution\_clock> \_thread\_start\_t; |
|  | // time\_point<high\_resolution\_clock> \_end\_time; |
|  | unsigned int \_duration; |
|  | std::map<uint16\_t, double> \_edges; |
|  | std::map<uint16\_t, time\_point<high\_resolution\_clock>> \_edge\_times; |
|  | std::string \_algorithmType; |
|  | char \_buffer[MAX]; |
|  |  |
|  | static bool \_stopRecieving; |
|  | static int \_messages\_sent; |
|  | static int \_messages\_recieved; |
|  | static std::default\_random\_engine \_generator; |
|  | static std::mutex \_thread\_mtx; |
|  | static std::condition\_variable \_thread\_cv; |
|  |  |
|  | MessagePacket passPotato(MessagePacket msg); |
|  | MessagePacket moveAnt(MessagePacket msg); |
|  |  |
|  | /\* TODO these are some of the required pheromone functions not developed\*/ |
|  | void incrPheromone(uint16\_t from); |
|  | void dilutePheromones(); |
|  |  |
|  | uint16\_t thread\_send(MessagePacket msg); |
|  | void thread\_recv(); |
|  | void receive(MessagePacket (ThreadNode::\*passAlgorithm)(MessagePacket msg)); |
|  |  |
|  | // function pointer for choosing how to pass the message |
|  | // will replace the two functions above |
|  | // |
|  | // typedef uint16\_t(\*passingAlgorithm)(uint16\_t, uint16\_t); |
|  | // uint16\_t thread\_send(passingAlgorithm alg); |
|  | // void thread\_recv(passingAlgorithm alg); |
|  |  |
|  | void incrMsgSent(unsigned int incr); |
|  | void incrMsgRecieved(unsigned int incr); |
|  | void recordMessage(MessagePacket msg); |
|  |  |
|  | uint16\_t findTrail(uint16\_t prevSender, uint16\_t dest); |
|  | uint16\_t getRandomNeighbor(uint16\_t prevSender, uint16\_t dest); |
|  | uint16\_t createDestination(uint16\_t min, uint16\_t max) const; |
|  | MessagePacket createMessage(uint16\_t (ThreadNode::\*passAlgorithm)(uint16\_t prev, uint16\_t dest)); |
|  |  |
|  | // bool canSend() const; |
|  | bool hasReceivedAllMsgs() const; |
|  |  |
|  | void randSleep(double mean); |
|  | void randCool(double mean); |
|  |  |
|  | void printTestInfo(uint16\_t id, std::string action, uint16\_t sender, uint16\_t trans, uint16\_t recv, uint16\_t destination) const; |
|  | void printRunInfo() const; |
|  | }; |
|  |  |
|  | #endif |

| #include "ThreadNode.h" |  |
| --- | --- |
|  |  |
|  | std::default\_random\_engine ThreadNode::\_generator; |
|  | std::mutex ThreadNode::\_thread\_mtx; |
|  | std::condition\_variable ThreadNode::\_thread\_cv; |
|  | bool ThreadNode::\_stopRecieving; |
|  |  |
|  |  |
|  | int ThreadNode::\_messages\_sent = 0; |
|  | int ThreadNode::\_messages\_recieved = 0; |
|  |  |
|  | ThreadNode::ThreadNode() |
|  | {} |
|  |  |
|  | ThreadNode::ThreadNode(uint16\_t id, std::vector<uint16\_t> neighbors, uint16\_t totalNodes, unsigned int duration) |
|  | : \_duration(duration), Node(id, neighbors, totalNodes) |
|  | { |
|  | // seed the generator, and thread start time only once |
|  | // not sure if we still need the \_stopReceiving boolean value or not |
|  | // but I am initializing it to false any way. |
|  | if(id == 0) |
|  | { |
|  | \_generator.seed(std::chrono::system\_clock::now().time\_since\_epoch().count()); |
|  | \_stopRecieving = false; |
|  | } |
|  |  |
|  | // initiallize the node edges for pheromones |
|  | for(auto &it : neighbors){ |
|  | \_edges[it] = pow(INIT\_PHEROMONE, POWER\_COEFF); |
|  | \_edge\_times[it] = TimeInterval::getNow(); |
|  | } |
|  | // Decrementing the \_messages\_recieved based on the number of nodes created |
|  | // so the independant Receive threads will not finish until the main threads |
|  | // are done |
|  | \_messages\_recieved--; |
|  | } |
|  |  |
|  | ThreadNode::ThreadNode(const ThreadNode& other) |
|  | : \_duration(other.getDuration()), \_edges(other.getEdges()), \_edge\_times(other.getEdgeTimes()), |
|  | Node(other) |
|  | { |
|  |  |
|  | } |
|  |  |
|  | ThreadNode::~ThreadNode() |
|  | {} |
|  |  |
|  | void ThreadNode::run(std::string algName) |
|  | { |
|  | // printTestInfo(getID(), "Running...", -1, -1, -1, -1); |
|  | this->\_algorithmType = algName; |
|  |  |
|  | auto start = TimeInterval::getNow(); |
|  | auto now = TimeInterval::getNow(); |
|  |  |
|  | auto timer = duration\_cast<milliseconds>(now - start); |
|  | auto end = milliseconds(\_duration \* 1000); |
|  |  |
|  | // create the reciever thread and pass it the thread\_recv function |
|  | // of this node |
|  | // |
|  | std::thread reciever(&ThreadNode::thread\_recv, this); |
|  |  |
|  | // time the sending for each thread based on the \_duration till "end" |
|  | while(timer <= end) { |
|  | // the running of each send THREAD needs to be staggered by the EXECUTION\_CYCLE time |
|  | // by adding the EXECUTION time with the node's ID times the EXECUTION\_CYCLE and |
|  | // the total number of nodes |
|  | // ID \* EXECUTION\_CYCLE |
|  | // EXECUTION\_SLEEP\_TIME = EXECUTION\_CYCLE + --------------------- |
|  | // TOTAL\_NODES |
|  | // EX: |
|  | // 0 \* EXECUTION\_CYCLE |
|  | // EST(NODE(0)) = EXECUTION\_CYCLE + --------------------- = EXECUTION\_CYCLE |
|  | // TOTAL\_NODES |
|  | // |
|  | std::this\_thread::sleep\_for(std::chrono::milliseconds(EXECUTION\_CYCLE + (getID() \* EXECUTION\_CYCLE) / getTotalNodes() ) ); |
|  |  |
|  | // randSleep is the old Implementation of the HOT\_POTATO algorithm |
|  | // randSleep(SLEEP); |
|  |  |
|  | // depending on whichever algorithm chosen we need to either use the |
|  | // getRandom or findTrail methods |
|  | if(algName == "hot"){ |
|  | thread\_send(createMessage(&ThreadNode::getRandomNeighbor)); |
|  | incrMsgSent(1); |
|  | } |
|  | else if(algName == "ant"){ |
|  | thread\_send(createMessage(&ThreadNode::findTrail)); |
|  | incrMsgSent(1); |
|  | } |
|  |  |
|  | now = TimeInterval::getNow(); |
|  | timer = duration\_cast<milliseconds>(now - start); |
|  | } |
|  |  |
|  | // after the main threads are done we increment the message recieved |
|  | // because at the creation of the node we decremented by the number |
|  | // of threads created so that when we are running the other independent |
|  | // receive thread it starts at a deficit so it could not finish until |
|  | // the sending threads are done |
|  | // |
|  | // Once all main threads have finished the receive threads could then |
|  | // finish afterwards |
|  | incrMsgRecieved(1); |
|  |  |
|  | if(getID() == 0){ |
|  | printRunInfo(); |
|  | } |
|  |  |
|  | if(reciever.joinable()) |
|  | reciever.join(); |
|  | } |
|  |  |
|  | MessagePacket ThreadNode::passPotato(MessagePacket msg) |
|  | { |
|  |  |
|  | msg.incHopCount(); |
|  |  |
|  | // Choose new neighbor as a receiver to pass the message that was not meant for |
|  | // this node. New neighbor must not be the previous sender |
|  | uint16\_t from = msg.getTransmittor(); // previous sender |
|  | uint16\_t dest = msg.getDestination(); // final destination |
|  | uint16\_t receiver = getRandomNeighbor(from, dest); |
|  | msg.setReceiver(receiver); |
|  | msg.setTransmittor(getID()); |
|  |  |
|  | printTestInfo(getID(), "Pass Potato", msg.getSender(), msg.getTransmittor(), msg.getReceiver(), msg.getDestination()); |
|  |  |
|  | return msg; |
|  | } |
|  |  |
|  | MessagePacket ThreadNode::moveAnt(MessagePacket msg) |
|  | { |
|  | // printTestInfo(getID(), "Move Ant", -1, -1, -1, -1); |
|  |  |
|  | msg.incHopCount(); |
|  | dilutePheromones(); |
|  | incrPheromone(msg.getTransmittor()); |
|  |  |
|  | // printTestInfo(getID(), "Move Ant - After INCREMENT", -1, -1, -1, -1); |
|  | // Choose new neighbor as a receiver to pass the message that was not meant for |
|  | // this node. New neighbor must not be the previous sender |
|  | uint16\_t from = msg.getTransmittor(); // previous sender |
|  | uint16\_t dest = msg.getDestination(); // final destination |
|  | uint16\_t receiver = findTrail(from, dest); |
|  | msg.setReceiver(receiver); |
|  | msg.setTransmittor(getID()); |
|  |  |
|  | printTestInfo(getID(), "Ant on The Move", msg.getSender(), msg.getTransmittor(), msg.getReceiver(), msg.getDestination()); |
|  |  |
|  | return msg; |
|  | } |
|  |  |
|  | void ThreadNode::incrPheromone(uint16\_t from) |
|  | { |
|  | // printTestInfo(getID(), "incrPheronome", -1, -1, -1, -1); |
|  | // add to the pheromone based on an ant walking acrosse the edge |
|  | \_edges[from] += INCR\_PHEROMONE; |
|  | \_edge\_times[from] = TimeInterval::getNow(); |
|  | // since an ant just walked on the edge restart the half-life clock |
|  |  |
|  | } |
|  |  |
|  | void ThreadNode::dilutePheromones() |
|  | { |
|  | // printTestInfo(getID(), "Dilution:", -1, -1, -1, -1); |
|  | for(auto &it : \_edges){ |
|  | // start is the last time an ant walked on the edge or |
|  | // the last time a message was received from this neighbor |
|  | auto start = \_edge\_times[it.first]; |
|  | auto now = TimeInterval::getNow(); |
|  |  |
|  | auto duration = duration\_cast<milliseconds>(now - start); |
|  | it.second = it.second \* pow( 0.5, ((duration.count()/DILUTION\_HALF\_LIFE))); |
|  | if(it.second < INIT\_PHEROMONE) |
|  | it.second = INIT\_PHEROMONE; |
|  | } |
|  | } |
|  |  |
|  | uint16\_t ThreadNode::thread\_send(MessagePacket msg) |
|  | { |
|  | std::string m = msg.getDataStr(); |
|  |  |
|  | // use the mailbox to send the message |
|  | const char \*dataPtr = m.c\_str(); |
|  | uint16\_t bytes = mbox\_send(msg.getReceiver(), dataPtr, strlen(dataPtr)); |
|  |  |
|  | // increment the total amount of messages sent |
|  |  |
|  | return bytes; |
|  | } |
|  |  |
|  | MessagePacket ThreadNode::createMessage(uint16\_t (ThreadNode::\*passAlgorithm)(uint16\_t prev, uint16\_t dest)) |
|  | { |
|  | // printTestInfo(getID(), "Create Message", -1, -1, -1, -1); |
|  | // create a message and get a random neighbor to send that |
|  | // message to |
|  | uint16\_t random\_dest = createDestination(0, getTotalNodes() - 1); |
|  | uint16\_t random\_recv = (\*this.\*passAlgorithm)(getID(), random\_dest); |
|  | MessagePacket msg(getID(), random\_dest, random\_recv); |
|  |  |
|  | printTestInfo(getID(), "Sending", getID(), getID(), msg.getReceiver(), msg.getDestination()); |
|  |  |
|  | // Set current time as start of keeping track of how long message is in network |
|  | msg.timeStart(); |
|  |  |
|  | return msg; |
|  | } |
|  |  |
|  | uint16\_t ThreadNode::createDestination(uint16\_t min, uint16\_t max) const |
|  | { |
|  | // printTestInfo(getID(), "createDestination", -1, -1, -1, -1); |
|  |  |
|  | // This should only be used when creating a message will find |
|  | // The only error checking is to see if the random number is |
|  | // the current ID |
|  | // |
|  | // get a random number from a uniform distribution in the range |
|  | // of min to max which should be 0 and total number of nodes in |
|  | // the graph - 1 |
|  | uint16\_t destination = RandomNodes::rand\_uniform(min, max, \_generator); |
|  | if(destination == getID()) |
|  | destination = RandomNodes::rand\_uniform(min, max, \_generator); |
|  |  |
|  |  |
|  | return destination; |
|  | } |
|  |  |
|  | uint16\_t ThreadNode::getRandomNeighbor(uint16\_t prevSender, uint16\_t destination) |
|  | { |
|  | // printTestInfo(getID(), "getRandomNeighbor", -1, -1, -1, -1); |
|  |  |
|  | /\* First check to see if the destination is one of this nodes neighbors \*\*/ |
|  | uint16\_t random\_recv = getID(); |
|  | std::vector<uint16\_t>::const\_iterator neighbor; |
|  | for(neighbor = getNbors().begin(); neighbor < getNbors().end(); neighbor++){ |
|  | if(\*neighbor == destination){ |
|  | random\_recv = \*neighbor; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | /\* If the random receiver is still this nodes ID then we know the destination |
|  | is not among this nodes neighbors. Get a randomNeighbor from the random |
|  | nodes class |
|  | \*/ |
|  | if(random\_recv == this->getID()){ |
|  | // find a random neighbor and set equal to destination |
|  | random\_recv = RandomNodes::getRandomNeighbor(prevSender, getNbors(), \_generator); |
|  | } |
|  |  |
|  | return random\_recv; |
|  | } |
|  |  |
|  | uint16\_t ThreadNode::findTrail(uint16\_t prevSender, uint16\_t dest) |
|  | { |
|  | /\* First check to see if the destination is one of this nodes neighbors \*\*/ |
|  | uint16\_t antTrail = getID(); |
|  | for(auto &neighbor : getNbors()) { |
|  | if(neighbor == dest){ |
|  | antTrail = neighbor; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  |  |
|  | /\* If the random receiver is still this nodes ID then we know the destination |
|  | is not among this nodes neighbors. Get a randomNeighbor from the random |
|  | nodes class |
|  | \*/ |
|  | if(antTrail == this->getID()){ |
|  | // find a random neighbor and set equal to destination |
|  | antTrail = RandomNodes::getPheromoneNeighbor(prevSender, \_edges, POWER\_COEFF, \_generator); |
|  | } |
|  |  |
|  | return antTrail; |
|  |  |
|  | } |
|  |  |
|  | void ThreadNode::thread\_recv() |
|  | { |
|  | \_thread\_mtx.lock(); |
|  | bool stopReceiving = ThreadNode::\_stopRecieving; |
|  | \_thread\_mtx.unlock(); |
|  |  |
|  | while(!stopReceiving){ |
|  | if(\_algorithmType == "hot") |
|  | receive(&ThreadNode::passPotato); |
|  | else if (\_algorithmType == "ant") |
|  | receive(&ThreadNode::moveAnt); |
|  |  |
|  | // if threads have received all messages then it is time to |
|  | // stop receiving |
|  | if(hasReceivedAllMsgs()){ |
|  | \_thread\_mtx.lock(); |
|  | ThreadNode::\_stopRecieving = true; |
|  | \_thread\_mtx.unlock(); |
|  |  |
|  | stopReceiving = ThreadNode::\_stopRecieving; |
|  |  |
|  | } |
|  | } |
|  | } // end thread\_recv |
|  |  |
|  | void ThreadNode::receive(MessagePacket (ThreadNode::\*passAlgorithm)(MessagePacket msg)) |
|  | { |
|  | // if bytes are 0 then there wasn't anything in the buffer |
|  | // return |
|  |  |
|  | int rbytes = mbox\_recv(getID(), &\_buffer, MAX); |
|  | if(rbytes <= 0) |
|  | return; |
|  |  |
|  | MessagePacket temp(\_buffer); |
|  |  |
|  |  |
|  | // Check if message's final destination is this thread |
|  | // If this is final destination: |
|  | if (temp.getDestination() == getID()) |
|  | { |
|  | printTestInfo(getID(), "Reached Destination", temp.getSender(), temp.getTransmittor(), temp.getReceiver(), temp.getDestination()); |
|  | recordMessage(temp); |
|  | incrMsgRecieved(1); |
|  |  |
|  | // If this is not final destination: |
|  | } else { |
|  | // Cool down for a random time |
|  | randCool(COOL); |
|  | MessagePacket forwardMessage = (\*this.\*passAlgorithm)(temp); |
|  | thread\_send(forwardMessage); |
|  | } |
|  | } |
|  |  |
|  | void ThreadNode::incrMsgSent(unsigned int incr) |
|  | { |
|  | // printTestInfo(getID(), "incrMsgSent", -1, -1, -1, -1); |
|  |  |
|  | // increment the messages by the increement variable |
|  | \_thread\_mtx.lock(); |
|  | this->\_messages\_sent += incr; |
|  | \_thread\_mtx.unlock(); |
|  | } |
|  |  |
|  | void ThreadNode::incrMsgRecieved(unsigned int incr) |
|  | { |
|  | // increment the messages received by the increment variable |
|  | \_thread\_mtx.lock(); |
|  | this->\_messages\_recieved += incr; |
|  | \_thread\_mtx.unlock(); |
|  | } |
|  |  |
|  | void ThreadNode::randSleep(double mean) |
|  | { |
|  | // choose a random number for the node to sleep |
|  | int randNumber = (int)(RandomNodes::rand\_exponential(mean, \_generator) \* 1000); |
|  | std::this\_thread::sleep\_for(std::chrono::milliseconds(randNumber)); |
|  | } |
|  |  |
|  | void ThreadNode::randCool(double mean) |
|  | { |
|  | // choose a random number for the node to cool |
|  | int randNumber = (int)(RandomNodes::rand\_exponential(mean, \_generator) \* 1000); |
|  | std::this\_thread::sleep\_for(std::chrono::milliseconds(randNumber)); |
|  | } |
|  |  |
|  | void ThreadNode::recordMessage(MessagePacket msg) |
|  | { |
|  | // Stop message timer |
|  | msg.timeStop(); |
|  |  |
|  | // record the messages hops and time |
|  | Node::\_total\_hops += msg.getHopCount(); |
|  | Node::\_total\_time += msg.getFinalTimeInterval(); |
|  |  |
|  | if(\_algorithmType == "ant"){ |
|  | incrPheromone(msg.getTransmittor()); |
|  | } |
|  | } |
|  |  |
|  | bool ThreadNode::hasReceivedAllMsgs() const |
|  | { |
|  | bool allMsgsReceived = false; |
|  |  |
|  | // check to see if the number of messages received matches the number |
|  | // of messages sent |
|  | \_thread\_mtx.lock(); |
|  | allMsgsReceived = \_messages\_recieved >= \_messages\_sent; |
|  | \_thread\_mtx.unlock(); |
|  |  |
|  | // printTestInfo(getID(), "HAS RECEIVED ALL MSGS ", (uint16\_t)allMsgsReceived, (uint16\_t)\_messages\_recieved, (uint16\_t)\_messages\_sent, -1); |
|  |  |
|  | return allMsgsReceived; |
|  | } |
|  |  |
|  | void ThreadNode::printTestInfo(uint16\_t id, std::string action, uint16\_t sender, uint16\_t trans, uint16\_t recv, uint16\_t dest) const |
|  | { |
|  | // // print test information for depugging |
|  | // \_thread\_mtx.lock(); |
|  | // std::cout << "Thread - "<< std::to\_string(id) << " - " << action |
|  | // << " - Sender - " << std::to\_string(sender) << " - Transmittor - " << std::to\_string(trans) |
|  | // << " - Receiver - " << std::to\_string(recv) << " - Dest - " << std::to\_string(dest) |
|  | // << " -> (" << std::to\_string(\_messages\_sent) << ":" << std::to\_string(\_messages\_recieved) << ")" << std::endl; |
|  | // \_thread\_mtx.unlock(); |
|  | } |
|  |  |
|  | std::map<uint16\_t, double> ThreadNode::getEdges() const |
|  | { |
|  | return \_edges; |
|  | } |
|  |  |
|  | std::map<uint16\_t, time\_point<high\_resolution\_clock>> ThreadNode::getEdgeTimes() const |
|  | { |
|  | return \_edge\_times; |
|  | } |
|  |  |
|  | unsigned int ThreadNode::getDuration() const |
|  | { |
|  | return \_duration; |
|  | } |
|  |  |
|  | void ThreadNode::printRunInfo() const |
|  | { |
|  |  |
|  | /\* Code written by leemes, user: 592323 published by stack overflow |
|  | Ref: https://stackoverflow.com/questions/14539867/how-to-display-a-progress-indicator-in-pure-c-c-cout-printf\*/ |
|  | \_thread\_mtx.lock(); |
|  | float recvProgress = (float)\_messages\_recieved / \_messages\_sent; |
|  | \_thread\_mtx.unlock(); |
|  |  |
|  | int barWidth = 70; |
|  | while (recvProgress < 1.0) { |
|  | std::cout << "Messages Received: [" |
|  | << int(recvProgress \* 100.0) << "%]\r"; |
|  | std::cout.clear(); |
|  | std::cout.flush(); |
|  |  |
|  | if(int(recvProgress\*100) == 100) |
|  | std::cout << std::endl; // << "--------------- Progress Done --------------------" << std::endl; |
|  |  |
|  | \_thread\_mtx.lock(); |
|  | recvProgress = (float)\_messages\_recieved / \_messages\_sent; |
|  | \_thread\_mtx.unlock(); |
|  | // \_thread\_mtx.unlock(); // <-------- Unlocked |
|  | } |
|  | } |
|  |  |
|  | int ThreadNode::getMessagesSent() |
|  | { |
|  | return \_messages\_sent; |
|  | } |
|  | int ThreadNode::getMessagesReceived() |
|  | { |
|  | return \_messages\_recieved; |
|  | } |

**Appendix D: Message Packet Class**

| #ifndef \_MESSAGE\_PACKET\_H\_ |  |
| --- | --- |
|  | #define \_MESSAGE\_PACKET\_H\_ |
|  |  |
|  | #include <thread> |
|  | #include <vector> |
|  | #include <stdint.h> |
|  | #include <sys/time.h> |
|  | #include <cstddef> |
|  | #include <sstream> |
|  | #include <iostream> |
|  | #include <string.h> |
|  | #include "mbox.h" |
|  | #include "TimeInterval.h" |
|  |  |
|  | class MessagePacket |
|  | { |
|  | public: |
|  | // also sets hop count to 0, transmittor to sendr, and initializes startTime |
|  | MessagePacket(uint16\_t sendr, uint16\_t dest, uint16\_t rcvr); |
|  | MessagePacket(std::string dataStr); |
|  | MessagePacket(const MessagePacket& otherPacket); |
|  | MessagePacket(); |
|  | ~MessagePacket(); |
|  |  |
|  | void init(uint16\_t sendr, uint16\_t dest, uint16\_t rcvr); |
|  |  |
|  | std::string getDataStr() const; |
|  | void setDataStr(std::string dataStr); |
|  |  |
|  | void setTransmittor(uint16\_t trsmtr); |
|  | uint16\_t getTransmittor() const; |
|  | uint16\_t getTransmittor(); |
|  |  |
|  | void setReceiver(uint16\_t rcvr); |
|  | uint16\_t getReceiver() const; |
|  |  |
|  | void setSender(uint16\_t sndr); |
|  | uint16\_t getSender() const; |
|  |  |
|  | void setDestination(uint16\_t dest); |
|  | uint16\_t getDestination() const; |
|  |  |
|  | void incHopCount(); |
|  | uint16\_t getHopCount() const; |
|  | void setHopCount(uint16\_t hpcount); |
|  |  |
|  | void timeStart(); // sets start time to keep track of how long message is in network |
|  | void timeStop(); // sets final time to how long since start time has passed |
|  |  |
|  | void setStartTime(double t); |
|  | double getStartTime() const; |
|  |  |
|  | void setFinalTime(double t); |
|  | double getFinalTime() const; |
|  |  |
|  | void setTimeInterval(TimeInterval t); |
|  | TimeInterval getTimeInterval() const; |
|  |  |
|  | // difference of stop time and start time |
|  | double getFinalTimeInterval(); |
|  |  |
|  | MessagePacket operator=(const MessagePacket& other); |
|  | friend std::ostream& operator<<(std::ostream& out, const MessagePacket& packet); |
|  |  |
|  | private: |
|  | uint16\_t \_transmittor; // threadnode that last sent the message |
|  | uint16\_t \_receiver; // threadnode that last recieved the message |
|  | uint16\_t \_sender; // original sender of message |
|  | uint16\_t \_destination; // threadnode the message needs to get to |
|  | uint16\_t \_hopCount; // number of passes between nodes |
|  | timeval \_startTime; // time message was created. determines time spend in network |
|  | timeval \_finalTime; // time message reached final destination |
|  | TimeInterval timeInterval; // Time interval object for keeping track of time |
|  |  |
|  | void copyMessagePacket(const MessagePacket& other); |
|  | }; |
|  |  |
|  | #endif |

| #include "MessagePacket.h" |  |
| --- | --- |
|  |  |
|  | MessagePacket::MessagePacket() |
|  | : \_transmittor(0), \_receiver(0), \_sender(0), \_destination(0), \_hopCount(0) |
|  | {} |
|  |  |
|  | MessagePacket::MessagePacket(uint16\_t sendr, uint16\_t dest, uint16\_t rcvr) |
|  | : \_transmittor(sendr), \_receiver(rcvr), \_sender(sendr), \_destination(dest), \_hopCount(0) |
|  | {} |
|  |  |
|  | MessagePacket::MessagePacket(std::string dataStr) |
|  | { |
|  | setDataStr(dataStr); |
|  | } |
|  |  |
|  | MessagePacket::MessagePacket(const MessagePacket& otherPacket){ |
|  | copyMessagePacket(otherPacket); |
|  | } |
|  |  |
|  | MessagePacket::~MessagePacket() |
|  | {} |
|  |  |
|  | void MessagePacket::init(uint16\_t sendr, uint16\_t dest, uint16\_t rcvr) |
|  | { |
|  | \_transmittor = sendr; |
|  | \_receiver = rcvr; |
|  | \_sender = sendr; |
|  | \_destination = dest; |
|  | \_hopCount = 0; |
|  | } |
|  |  |
|  | std::string MessagePacket::getDataStr() const |
|  | { |
|  | std::string dataStr = ""; |
|  | std::string currentStr = ""; |
|  | std::stringstream ss; |
|  | ss << \_transmittor << ',' << \_receiver << ',' << \_sender << ',' << \_destination << ',' << \_hopCount << ','; |
|  | while (std::getline(ss, currentStr, ',')) |
|  | { |
|  | dataStr += currentStr + ','; |
|  | } |
|  |  |
|  | ss.clear(); |
|  | currentStr = ""; |
|  | ss << (double)\_startTime.tv\_sec << ','; |
|  | ss >> currentStr; |
|  | dataStr += currentStr; |
|  |  |
|  | ss.clear(); |
|  | currentStr = ""; |
|  | ss << (double)\_finalTime.tv\_sec << ','; |
|  | ss >> currentStr; |
|  | dataStr += currentStr; |
|  |  |
|  | return dataStr; |
|  | } |
|  |  |
|  | void MessagePacket::setDataStr(std::string dataStr) |
|  | { |
|  | std::string trsmtrStr = ""; |
|  | std::string rcvrStr = ""; |
|  | std::string sndrStr = ""; |
|  | std::string destStr = ""; |
|  | std::string hCntStr = ""; |
|  | std::string sTimeStr = ""; |
|  | std::string fTimeStr = ""; |
|  |  |
|  | std::stringstream ss; |
|  | ss << dataStr; |
|  |  |
|  | std::getline(ss, trsmtrStr, ','); |
|  | std::getline(ss, rcvrStr, ','); |
|  | std::getline(ss, sndrStr, ','); |
|  | std::getline(ss, destStr, ','); |
|  | std::getline(ss, hCntStr, ','); |
|  | std::getline(ss, sTimeStr, ','); |
|  | std::getline(ss, fTimeStr, ','); |
|  |  |
|  | setTransmittor(std::stoi(trsmtrStr)); |
|  | setReceiver(std::stoi(rcvrStr)); |
|  | setSender(std::stoi(sndrStr)); |
|  | setDestination(std::stoi(destStr)); |
|  | setHopCount(std::stoi(hCntStr)); |
|  | setStartTime(std::stod(sTimeStr)); |
|  | setFinalTime(std::stod(fTimeStr)); |
|  | timeInterval.setTimes(getStartTime(), getFinalTime()); |
|  | } |
|  |  |
|  | void MessagePacket::setTransmittor(uint16\_t trsmtr) |
|  | {\_transmittor = trsmtr;} |
|  |  |
|  | uint16\_t MessagePacket::getTransmittor() const |
|  | {return \_transmittor;} |
|  |  |
|  | uint16\_t MessagePacket::getTransmittor() |
|  | {return \_transmittor;} |
|  |  |
|  | void MessagePacket::setReceiver(uint16\_t rcvr) |
|  | {\_receiver = rcvr;} |
|  |  |
|  | uint16\_t MessagePacket::getReceiver() const |
|  | {return \_receiver;} |
|  |  |
|  | void MessagePacket::setSender(uint16\_t sndr) |
|  | {\_sender = sndr;} |
|  |  |
|  | uint16\_t MessagePacket::getSender() const |
|  | {return \_sender;} |
|  |  |
|  | void MessagePacket::setDestination(uint16\_t dest) |
|  | {\_destination = dest;} |
|  |  |
|  | uint16\_t MessagePacket::getDestination() const |
|  | {return \_destination;} |
|  |  |
|  | void MessagePacket::incHopCount() |
|  | {\_hopCount++;} |
|  |  |
|  | uint16\_t MessagePacket::getHopCount() const |
|  | {return \_hopCount;} |
|  |  |
|  | void MessagePacket::timeStart() |
|  | { |
|  | timeInterval.start(); |
|  | \_startTime = timeInterval.start\_time; |
|  | } |
|  |  |
|  | void MessagePacket::timeStop() |
|  | { |
|  | timeInterval.stop(); |
|  | \_finalTime = timeInterval.end\_time; |
|  | } |
|  |  |
|  | double MessagePacket::getFinalTimeInterval() |
|  | { |
|  | return timeInterval.GetInterval(); |
|  | } |
|  |  |
|  | void MessagePacket::setHopCount(uint16\_t hpcount){ |
|  | this->\_hopCount = hpcount; |
|  | } |
|  |  |
|  | void MessagePacket::setStartTime(double t) { |
|  | this->\_startTime.tv\_sec = t; |
|  | } |
|  | double MessagePacket::getStartTime() const{ |
|  | return this->\_startTime.tv\_sec; |
|  | } |
|  |  |
|  | void MessagePacket::setFinalTime(double t) { |
|  | this-> \_finalTime.tv\_sec = t; |
|  | } |
|  | double MessagePacket::getFinalTime() const{ |
|  | return this->\_finalTime.tv\_sec; |
|  | } |
|  |  |
|  | void MessagePacket::setTimeInterval(TimeInterval t){ |
|  | this->timeInterval = t; |
|  | } |
|  | TimeInterval MessagePacket::getTimeInterval() const{ |
|  | return this->timeInterval; |
|  | } |
|  |  |
|  | void MessagePacket::copyMessagePacket(const MessagePacket& other){ |
|  | setTransmittor(other.getTransmittor()); |
|  | setReceiver(other.getReceiver()); |
|  | setSender(other.getSender()); |
|  | setDestination(other.getDestination()); |
|  | setHopCount(other.getHopCount()); |
|  | setStartTime(other.getStartTime()); |
|  | setFinalTime(other.getFinalTime()); |
|  | setTimeInterval(other.getTimeInterval()); |
|  |  |
|  | } |
|  |  |
|  | MessagePacket MessagePacket::operator=(const MessagePacket& other){ |
|  | copyMessagePacket(other); |
|  | return \*this; |
|  | } |
|  |  |
|  | std::ostream& operator<<(std::ostream& out, const MessagePacket& packet) |
|  | { |
|  | out << packet.getDataStr(); |
|  | return out; |
|  | } |

**Appendix E: MailBox Class**

|  |  |
| --- | --- |
| /\* |  |
|  | \* mailbox.cc |
|  | \* Copyright(c) 2022 Dongsoo S. Kim |
|  | \*/ |
|  | #include <iostream> |
|  | #include <string.h> |
|  | #include <map> |
|  | #include <queue> |
|  | #include <mutex> |
|  | #include <condition\_variable> |
|  | #include "mbox.h" |
|  | #include "MessagePacket.h" |
|  |  |
|  |  |
|  |  |
|  | class MailBox { |
|  | struct item { |
|  | int length; |
|  | char\* content; |
|  | }; |
|  |  |
|  | typedef std::queue<item> mailbox\_t; // defining a queue of mailboxes to be a type for simple annotation |
|  | std::map<uint16\_t,mailbox\_t> \_mailboxes; // the queue of mailboxes mapped to an unsigned integer |
|  | std::map<uint16\_t, std::mutex> \_mtx; // parallel map of locks to each mailbox ID |
|  | std::map<uint16\_t, std::condition\_variable> cvs; |
|  | std::mutex m; // basic lock for code deletion |
|  | int ID; |
|  | public: |
|  | MailBox() { |
|  | }; |
|  | ~MailBox(); |
|  | bool empty(uint16\_t msgID); |
|  | int send(uint16\_t msgID, const void \*packet, int len); |
|  | int recv(uint16\_t msgID, void \*packet, int max); |
|  | }; |
|  |  |
|  | static MailBox mailbox; |
|  |  |
|  |  |
|  | MailBox::~MailBox() |
|  | { |
|  | struct item xtem; |
|  | m.lock(); |
|  | for (auto x : \_mailboxes) { |
|  | while (!x.second.empty()) { |
|  | xtem = x.second.front(); |
|  | x.second.pop(); |
|  | delete [] xtem.content; |
|  | } |
|  | } |
|  | m.unlock(); |
|  | } |
|  |  |
|  | bool MailBox::empty(uint16\_t msgID){ |
|  | bool isEmpty = false; |
|  |  |
|  | // critical section |
|  | \_mtx[msgID].lock(); |
|  | if(\_mailboxes[msgID].empty() ) |
|  | isEmpty = true; |
|  | \_mtx[msgID].unlock(); |
|  |  |
|  | return isEmpty; |
|  | } |
|  |  |
|  | int MailBox::send(uint16\_t msgID, const void \*packet, int len){ |
|  |  |
|  | int numBytes; |
|  | // could be critical section |
|  | { |
|  | std::lock\_guard<std::mutex> lock(\_mtx[msgID]); |
|  |  |
|  | char\* msg = (char\*)malloc(len + 1); |
|  | strcpy(msg, (char\*)packet); |
|  |  |
|  | item pckt = {len, msg}; |
|  | numBytes = sizeof(pckt); |
|  |  |
|  | // std::cout << msgID << " - MAILBOX SEND: " << msg << std::endl; |
|  | \_mailboxes[msgID].push(pckt); |
|  | // std::cout << "Mailbox\_Send - " << msgID << " - count - " << \_mailboxes[msgID].size() << std::endl; |
|  | } |
|  |  |
|  | cvs[msgID].notify\_all(); |
|  | return numBytes; |
|  | } |
|  |  |
|  | int MailBox::recv(uint16\_t msgID, void \*packet, int max){ |
|  | int numBytes = 0; |
|  |  |
|  | // critical section |
|  | std::unique\_lock<std::mutex> lk(\_mtx[msgID]); |
|  | // bool empty = \_mailboxes[msgID].empty(); |
|  | // std::cv\_status timedOut = cvs[msgID].wait\_for(lk, std::chrono::milliseconds(200)); |
|  | cvs[msgID].wait\_for(lk, std::chrono::milliseconds(1000)); |
|  | // std::cerr << msgID << " - Mailbox - Receive" << std::endl; |
|  | // bool timedOut = cvs[msgID].wait\_for(lk, std::chrono::milliseconds(200), [empty]{return !empty;}); |
|  |  |
|  | // std::cout << "Mailbox\_Recv - " << msgID << " - count - " << \_mailboxes[msgID].size() << std::endl; |
|  | if(!(\_mailboxes[msgID].empty())){ |
|  | strcpy((char\*)packet, \_mailboxes[msgID].front().content); |
|  | free(\_mailboxes[msgID].front().content); |
|  | \_mailboxes[msgID].pop(); |
|  | numBytes = sizeof(packet); |
|  | } |
|  |  |
|  | // std::cerr << msgID << " - Mailbox - Receive - End" << std::endl; |
|  | // } |
|  |  |
|  | return numBytes; |
|  | } |
|  |  |
|  |  |
|  | bool mbox\_empty(uint16\_t msgID) |
|  | { |
|  | return mailbox.empty(msgID); |
|  | } |
|  |  |
|  | bool mbox\_avail(uint16\_t msgID) |
|  | { |
|  | return true; |
|  | } |
|  |  |
|  | int mbox\_send(uint16\_t msgID, const void \*packet, int len) |
|  | { |
|  | return mailbox.send(msgID, packet, len); |
|  | } |
|  |  |
|  | int mbox\_recv(uint16\_t msgID, void \*packet, int max) |
|  | { |
|  | return mailbox.recv(msgID, packet, max); |
|  | } |
|  |  |

**Appendix F: Time Interval Class**

| #ifndef \_TIMEINTREVAL\_H\_ |  |
| --- | --- |
|  | #define \_TIMEINTREVAL\_H\_ |
|  |  |
|  | #include <sys/time.h> |
|  | #include <chrono> |
|  | #include <mutex> |
|  | #include <cstddef> |
|  |  |
|  | using namespace std::chrono; |
|  |  |
|  | class TimeInterval{ |
|  | public: |
|  | time\_point<high\_resolution\_clock> \_start\_time; |
|  | time\_point<high\_resolution\_clock> \_end\_time; |
|  |  |
|  | timeval start\_time; |
|  | timeval end\_time; |
|  |  |
|  | TimeInterval(); |
|  | void start(); |
|  | void chronoStart(); |
|  | void stop(); |
|  | void chronoStop(); |
|  |  |
|  | static time\_point<high\_resolution\_clock> getNow(); |
|  |  |
|  | void setTimes(double st, double ft); |
|  | timeval getStartTime() const; |
|  | timeval getEndTime() const; |
|  | double GetInterval(); |
|  | double getChronoInterval(); |
|  | time\_point<high\_resolution\_clock> projectedEnd(int seconds) const; |
|  |  |
|  | TimeInterval operator=(const TimeInterval& other); |
|  |  |
|  | private: |
|  | static std::mutex \_time\_mutex; |
|  | }; |
|  |  |
|  | #endif //!defined TimeInterval |

**Appendix G: Makefile**

| CXX = g++ -pthread |  |
| --- | --- |
|  | GFLAGS = -g -std=c++11 |
|  | OBJECTS = mailbox.o |
|  | FILES = mailbox.cc |
|  |  |
|  | mailbox.o: mailbox.cc |
|  | $(CXX) $(GFLAGS) -c mailbox.cc |
|  |  |
|  | clean: |
|  | rm -f mailbox.o |
|  | rm \*.o |
|  |  |
|  | valgrind: driver |
|  | valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./driver |

**Appendix H: Console Traces**

./driver -d 3 -r ant ./graph/A19.dat

Test - (./graph/A19.dat) - duration - (3s) - Algorithm - (ant)

Initial Pheromone - 2.5

Pheromone Increment Constant - 5.2

Power Coefficient - 2.13

Dilution Half Life - 600

Messages Received: [98%]

Messages Sent: 91

Messages Received: 91

Graph (./graph/A19.dat) - Total Hops -> 625 - Total Time -> 166880 ms

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

./driver -d 7 -r hot ./graph/A19.dat

Test - (./graph/A19.dat) - duration - (7s) - Algorithm - (hot)

Messages Received: [99%]

Messages Sent: 198

Messages Received: 198

Graph (./graph/A19.dat) - Total Hops -> 3777 - Total Time -> -925490 ms