Team Pedestrians

Caitlin Klein and Peter Lee

Overview

We are trying to model foot traffic over a period of time. There are several major components to do this. There are people who have a schedule they have to keep. There is a map which is modeled through a graph that contains nodes (destinations) and edges (paths) which have weights that correspond to distance. These weights will adjust according to the number of people on that path. So edges will have a minimum weight to represent distance and that number may increase to represent the increased amount of traffic. The weight only increases if the capacity of the edge has been reached. If so the weight increases linearly for each additional person over the capacity.

Modules

Person Module

description: models an individual, each person has a schedule and a current location
interactions:receives time from proxy and returns current location. The person then sends its current location and destination to the map server in a shortest path query, and receives the next edge to go to. When a person receives its next destination, it sends a message to the map to reflect it is now on that path and similarly it tells the map when it has left the path, i.e. "subscribes" and "unsubscribes" from the path.

Map Module

description: models the map server, using the ugraph module to represent a map as a graph. Contains a ugraph with the vertices and edges connecting vertices along with weights.
interactions: Answers queries for next edge a person has to go to, spawns processes to handle calculating next edge in shortest path, receives messages from persons to alter the weight of the edges.

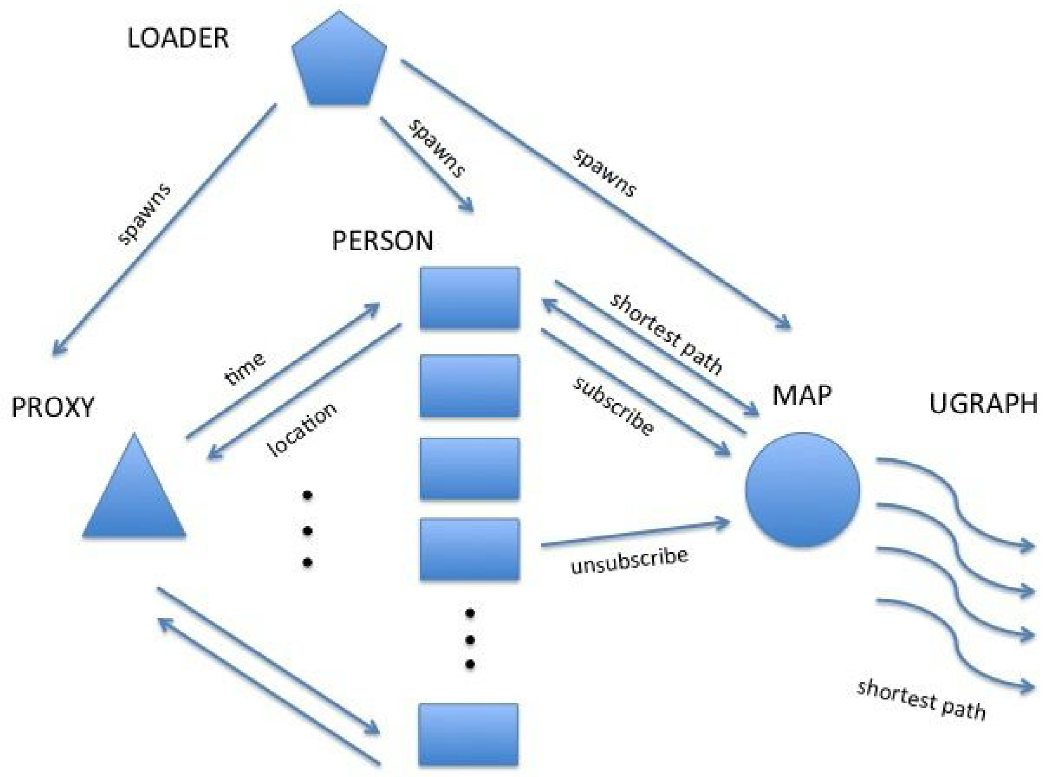
Proxy Module

description: acts as a proxy, pings each person telling them the current time and receives in return a current position, writes to a file in JSON format.
interactions: sends time to person and receives their current location

Ugraph Module

description: acts as a wrapper for the digraph library, digraph is for unweighted directed graphs while we need weighted undirected graphs, however digraph has labels that can store the weight and for every edge there exists another edge going the opposite way
interactions: used by the map module to represent the map

Loader Module
description: loads the initial data
interactions: creates a bunch of persons based off of reading from a file containing schedules, spawns those persons and gives the pids to the proxy, from another file generates the graph for the map module, registers the map for the persons to use



Problem to Address

The most interesting problem to address is a person querying the map and finding out its next position. Because the weights on the graph change according to the amount of people currently on them, shortest path can not just be calculated once as that could change depending on the volume of people. The map module also can't be a bottleneck for the persons. Therefore map will receive messages from the persons and spawn processes to calculate the next destination. A simple way to do this is to just calculate the shortest path for each person every time they have to make a decision about where to go. However, with a high volume of nodes and people this could result in a large amount of potentially unnecessary computation. Alternatives to consider for optimization have been distributing this work over multiple machines, breaking up the work to lessen the load on a single machine. However, a better approach would be to optimize the shortest path algorithm to lessen the overall workload.

There was much discussion over how to factor in the changing weights on the map. We struggled with how to accurately represent the number of people on the path as they move. We decided on a subscription model where persons send a message to the map "subscribing" to a path. The map increases the weight of that edge linearly once a capacity has been reached. Persons can also "unsubscribe" to decrease that weight if the path is over capacity. Any discrepancies can be attributed to real world synchronization problems of real people walking a path.

Minimum Deliverable

- working model that simulates foot traffic
- program outputs to a file who was on what path at particular times

Maximum Deliverable

- model can be accessed while running to query number of people on each path / how often a path is used
- a visualization of the model
- additional data based on how people move about the pathways
- providing a file of people's schedules and creating a model based off those schedules

Outcome

We achieved our minimum deliverables and some of our maximum deliverables. We ended up with a working model that outputs to a file. In addition we were able to provide a visualization for that model using the outputted file. We also wrote a python script to generate test data to supply the people's schedules. We did not have enough time to perform data analytics. When we pitched our project proposal we did not intend to complete all of the maximum deliverables but to try to complete as many as we could.

The best decision that we made was discussing extensively the design of our model before any coding happened. It was important to us to have solid logic of how our model would work since we knew debugging would be difficult and feasibility of having the program complete within a reasonable time. Without extensive discussion, debugging would have been more difficult and more code would have had to be rewritten. In the beginning we decided to use digraph to help model the map as a graph. However, digraph implements directed unweighted graphs while we needed undirected weighted graphs. We built ugraph on top of digraph to achieve the functionality we wanted. If we were to do this again we would have just written ugraph from scratch with the same functionality and inclusion of ETS tables but without the calls to digraph underneath.

As a two person team, it was quite easy to divide the work. Because of the heavy discussion in the beginning we spent a lot of time together working on it. While implementing the program we still tended to work together even if we divvied up tasks to be able to consult with the other if problems arose. Since there are several modules it was easy to divide, work concurrently, and integrate.

Most Difficult Bug Report

The most difficult bug that we encountered was a problem resulting from writes to an ETS table from a process with read-only access to the table. By default, only the process that creates an ETS table has write access to the table. Since our graph was created by a loader process and given to the map process, the map process could not update the weights of the graph, which led to our program crashing. The bug was fixed by allowing the map process to create the graph, eliminating any problems with ownership. We also later discovered that the ets module provides functionality for giving away ownership tables, and the digraph module that we used provides table IDs that could be used for this purpose.

To find the bug, we first isolated the line of code that crashed our program, which happened to be when we tried to update the weight of an edge. Finding the reason behind the problem took much longer, as we only discovered the problem after reading more on ETS tables and how they work. If we had known more about the properties of ETS tables, we would have likely found the reason for the problem much more quickly, so perhaps more initial research into the ets tables as used in the digraph module would have allowed us to solve this bug much more quickly.

Overview of Code

- proxy.erl

- The server that synchronizes and sends the time to person processes and outputs time and location information to a file.

- loader.erl

- The loader module for the program. Reads in file input and spawns the map server, proxy, and the person processes according to the data.

- map.erl

- The module for the map server.

- ugraph.erl

- The module representing undirected graphs.

- person.erl

- The module representing a person as a process.

The following files are auxiliary files for the pedestrian traffic modeling program:

- gen_sched.py

- Generates random schedules given a graph for use with the program above. See below for usage.

- viz.html

- Implements a visualization of the output of the pedestrian traffic simulation.

- **compile**

- Compiles the erl modules