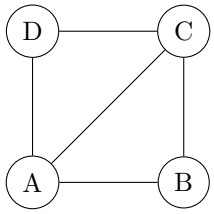
Scientific Computing | Fitchburg State University, Department of Mathematics, Shortest Distance Project | November 10, 2022

This project involves solving problems that are shortest distance problems. Following the general instructions of the project and complete all of the questions involved here.

There are two types of problems in this project. One is called the traveling Salesperson (TSP) problem. The other is called the short path problem (SPP). Before asking any questions, we'll cover the needed background.

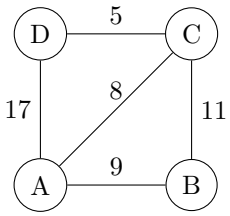
Graphs and Weighted Graphs

A **graph** is set of nodes (or vertices) and edges. Each edge has two nodes and a node can have 0 or more edges that connect to it. Here's a simple example:



The nodes are labelled A, B, C, D and the line segment indicates an edge. Nodes A and C connect to each of the other nodes. And B and D are not connected to each other.

In this project we will be using weighted graphs, which means that each edge has a number associated with it. We will often think of the nodes as either cities (in the TSP) or maybe intersections or addresses (in the SPP). The weights then are distances between nodes. We can then create a visual representation of a weighted graph by adding numbers to the edges:



Stage 1

There is a rudimentary `WeightedGraphs` package that you can use for this project. It can be found on the git repository in the `projects` directory. It is based on the `Graphs` with some simplifications made. For the first stage, we will update this package with some additional functionality and do some testing.

1. Add `Base.show` methods for `Vertex`, `Edge` and `WeightedGraph` to make it easier to read.
2. Add the following to your module:
   (a) an `addVertices!` function that just adds a vector of `Vertex` to a `WeightedGraph`
   (b) an `addVertices!` function that takes a variable number of vertices as arguments.
   (c) an `addVertices!` function that takes in a string of some format and adds the vertices.
   (d) an `addEdges!` function that takes a vector of `Edge` objects as arguments.
   (e) an `addEdges!` function that takes a variable number of edges as arguments.
   (f) an `addEdges!` function that takes in a string of some format and adds the vertices.

3. Write a test suite for your module with the following:

   (a) Test that creating a vertex works.

   (b) Test that creating an edge works.

   (c) Test that creating a basic weighted graph works.

   (d) Tests that adds a single vertex, a vector of vertices or any number of vertices.

   (e) Tests that should throw an `ArgumentError` when adding vertices adds a vertex that already exists.

4. This will create a `Path` data type and some functions related to the path.

   (a) Create a `Path` data type that contains a vector of vertices. It should check that each of the vertices are in the graph and that consecutive vertices are edges, if not it should throw an error.

   You may also wish to write a constructor that takes a string in some format and produces a `Path`.

   (b) Test your `Path` constructors on at least 2 different graphs and a few different paths that are in and not in the graph.

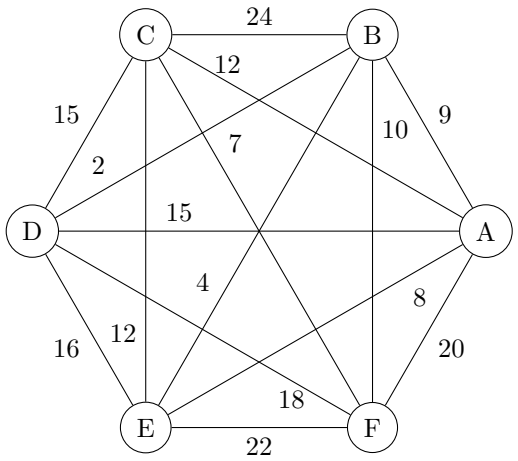
   (c) Create a function called `distance` with the argument of type `Path` and if the path is not in the graph, return `Inf` and if the path is in the graph, return the total distance (sum of the weights).

   (d) Test your `distance` function on the same graphs and paths as above.

Stage 2

For stage 2 we use the data types above to solve some small problems.

1. Consider the following graph:



   (a) Set up a `WeightedGraph` that represents this graph.

   (b) Create the `Path`, $C \to A \to F \to B$ and determine the total distance of the `Path`.

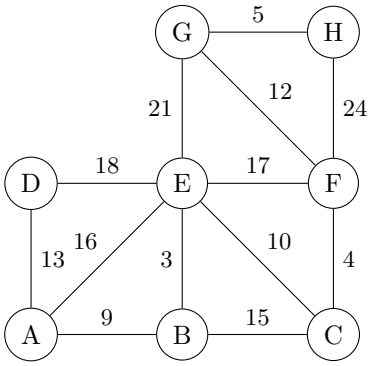
   (c) Add a function to the `WeightedGraphs` module called `solveTSP` which uses brute force (goes through all paths) and determines the shortest path. You should use the distance function that you wrote.

   (d) Use `solveTSP` to solve the above problem.

2. This problem uses simulated annealing to solve the the traveling salesperson problem.

   (a) Add a function to the `WeightedGraphs` module called `simAnnealTSP` which uses simulated annealing to determine the shortest path. The input should be a `WeightedGraph` and a positive integer number of trials to run.

3. Consider the following weighted graph:



Construct the graph using the methods you wrote for the module. Create a few paths and test the distance function on these.

4. Construct a function called `minDistTrop`, which uses the tropical math module from class, and add this to your `WeightedGraphs` module. This should take in a `WeightedGraph` a starting `Vertex` and an ending `Vertex`. This should return a `Path` and a distance as a named tuple.

5. Find the shortest path on the above graph from A to H.

6. Add some vertices and edges to graph above and test your `minDistTrop` function on at least two other paths.

Stage 3–realistic(ish) Problems

1. This problem solves a real-world Traveling Salesperson Problem (TSP) using distance data. We will find the shortest total path between the Massachusetts cities/towns: Boston, Fall River, Fitchburg, Lowell, North Adams, Springfield and Worcester. Follow these steps to do this:

   (a) Use the brute force algorithm that goes through each path to determine the shortest total distance. Find the distance and write down the path as it goes through each town. Start and end with Fitchburg and write down the town names of the shortest path. (Hint: it is highly recommended to sign up for the Google Maps Distance Matrix API. See Appendix 1 of the notes for information on this.)

   (b) Add the following towns: Amherst, Braintree, Lawrence, Newton, Natick, Pittsfield, Plymouth, Provincetown, Salem, Woburn to the list above. Produce an array of distances in alphabetical order.

   (c) Use a Simulated Annealing algorithm to find a minimum path through all of the town in (a) and (c). Use as large a number of randomizations in your algorithm as possible. Find the total distance as well as the path of the towns. Start and end with Fitchburg and write down the town names of the shortest path. Comment on your results. Does this appear to be a shortest path? If not, what is going on?

2. Use the Dijkstra's Algorithm to find the shortest path from A to H for the graph in the previous problem.

3. This problem finds the shortest distance from Fitchburg State to the Entertainment Cinemas in Leominster which mimics Google Maps or another mapping/GPS program.

   (a) Create a graph that has Fitchburg State as the starting point (use 160 Pearl St.) and the Entertainment Cinemas (45 Sack Blvd, Leominster, MA 01453) as the end. Include the following intersections in your graph:

   - Pearl Street and John Fitch Hwy (Fitchburg)
   - North Street and Main Street (Fitchburg)
   - John Fitch Hwy and Lunenburg Street (Fitchburg)
   - White Street and Massachusetts Avenue (Fitchburg)

- Electric Avenue and Massachusetts Avenue (Fitchburg)
- Electric Avenue and Whalom Road (Fitchburg)
- Whalom Road and Summer Street (Fitchburg)
- South Street and Laurel Street (Fitchburg)
- South Street and Wanoosnoc Rd (Fitchburg)
- Water Street and Bemis Rd (Fitchburg)
- Summer Street and John Fitch Hwy (Fitchburg)
- North Street and Main Street (Lunenburg)
- Main Street and Prospect Street (Lunenburg)
- N. Main Street and Rte 2 (Lunenburg)
- Merriam Ave and Rte 2 (Lunenburg)
- Haws Street and Commercial Road (Lunenburg)

In your graph, do not connect all of the nodes. You should have at most 4 nodes (intersections) that connect and choose the shortest distances between intersections. If possible produce a map of your graph.

(b) Use the Tropical Matrix Arithmetic method to solve this problem. List the path travelled that minimizes the travel distance and the nodes (intersections) travelled through.

(c) Use Dijkstra's method to solve the problem. List the path travelled that minimizes the travel distance and the nodes (intersections) travelled through.