KIT205 Data Structures and Algorithms  
Assignment 3

# Introduction

[Ticket to Ride](https://www.daysofwonder.com/tickettoride/en/usa/) is a popular board game that involves connecting cities in a given railroad network.  In this assignment you will prototype some potential approaches for creating an AI player for this game (since the AI players for the computerised version are currently terrible!)

The basic gameplay of Ticket to Ride requires players to fulfill "tickets" that are randomly selected.

* A ticket consists of 2 cities that need to be connected.
  + Adjacent cities are connected by placing the required number of train tokens on the track.

While there are other complications in the full game, the basis of a good strategy is to fulfill your tickets using the least number of train tokens – i.e. can fulfill more tickets with the fixed number of train tokens available.

## Data Structures and Input

We will represent the map using the following data structures, as used in tutorials.

typedef struct edge{  
 int to\_vertex;  
 int weight;   
} Edge;  
  
typedef struct edgeNode{  
 Edge edge;  
 struct edgeNode \*next;  
} \*EdgeNodePtr;  
  
typedef struct edgeList{  
 EdgeNodePtr head;  
} EdgeList;  
  
typedef struct graph{  
 int V;  
 EdgeList \*edges;  
} Graph;

In this case,

* the vertices represent the cities on the map, and
* the edge weight will be the number of train tokens required to connect adjacent cities.

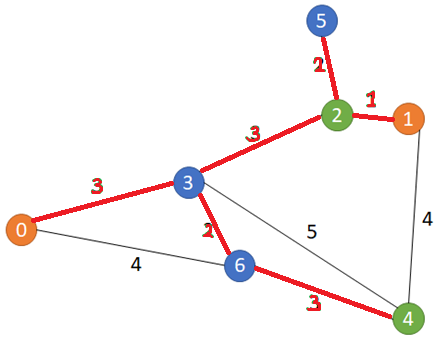
We will also use the same redirected input method used in tutorials to create the graph.  However, the graph in this case is *undirected*, so **two** edges need to be added for each pair of adjacent cities.

* i.e. if cities 2 and 7 are adjacent 🡪 an edge needs to be added from vertex 2 to 7 *and* from vertex 7 to 2.

The graph input will include only edges from cities with a lower index to cities with a higher index, but you must also add the edges in the other direction.

The graph input will then be followed by input for a given number of tickets.  For example, the following input will create a graph of **7** cities,

with **2** tickets from cities 2 to 4 and 0 to 1:

**7**  
2  
6,4 3,3  
2  
4,4 2,1  
2  
5,2 3,3   
2  
6,2 4,5  
1  
6,3  
0  
0  
**2**  
2,4 0,1

Tickets to be fulfilled



# Part A

The problem of finding the cheapest way to fulfill tickets is obviously related to the minimal spanning tree problem.  So part A is a warm-up exercise where you will implement Prim's MST algorithm.  The following high-level pseudocode should be followed:

## Prim's Minimal Spanning Tree

([from Wikipedia](https://en.wikipedia.org/wiki/Prim%27s_algorithm))

1. Associate with each vertex *v* of the graph
   * a number *C[v]* 🡪 cheapest cost of a connection to *v*
   * an edge *E[v]* 🡪 edge providing that cheapest connection

To initialize these values,

* + set all values of *C[v]* to *+∞* (or to any number larger than the maximum edge weight)
  + set each *E[v]* to a special flag value 🡪 indicating that there is no edge connecting *v* to earlier vertices.

1. Initialize an empty forest*F* and a set *Q* of vertices that have not yet been included in *F* (initially, all vertices).
2. Repeat the following steps until *Q* is empty:
   1. Find + remove a vertex *v* from *Q* having the minimum possible value of *C[v]*
   2. Add *v* to *F* and, if *E[v]* is not the special flag value, also add *E[v]* to *F*
   3. Loop over the edges *vw* connecting *v* to other vertices *w*. For each such edge, if *w* still belongs to Q and *vw* has smaller weight than *C[w]*, perform the following steps:
      1. Set *C[w]* to the cost of edge *vw*
      2. Set *E[w]* to point to edge *vw*
3. Return *F*

**Your implementation must follow the above pseudocode and use the following C function prototype**:

Graph prims\_mst(Graph \*self);

## Hints

* In the pseudocode *F* will be a Graph data structure
* Since you know that the graph is connected, you may start by initialising *F* in step 2 as a graph containing all vertices (but no edges).
  + This means that you can skip the part "Add *v* to *F*" in step 3b, you will only need to add the edge *E[v]* to *F*
* You can combine the *C[v]* and *E[v]* lists by using an array of Edges
* *Q* can also be implemented as a membership array (i.e. array representation of a set)
* You can use a simple search for step 3a.  You do not need to use any of the more sophisticated heap-based approaches.

For part A, you can ignore the tickets.  You just need to return the MST for the given fully connected graph.  Having returned the graph, you should print all of the edges that have been added and the total cost of the MST.

# Part B

You will now write an algorithm to find the cheapest way to fulfill your tickets.  This is not a trivial task.

* For example, a naive solution might be to find the shortest path for each ticket and then add all of the paths together.
  + However, for certain tickets, this may be very wasteful, as illustrated below.
* In this case the top map shows the shortest paths for tickets A and B.
* The bottom map shows one way that these tickets can be fulfilled more efficiently.

Diagram

Description automatically generated with medium confidence

So, the ticket fulfillment problem is related to the MST problem and also to the [Steiner tree problem](https://en.wikipedia.org/wiki/Steiner_tree_problem) of finding an MST for a subset of the vertices in the graph.  While there are many polynomial time algorithms for MST, there are suprisingly no known efficient algorithms for the Steiner tree problem - it is an [NP-hard](https://en.wikipedia.org/wiki/NP-hardness) problem.

Like the Steiner tree problem, I suspect that the problem of finding the most efficient way to fulfill tickets is also NP-hard.  Luckily, I do not expect you to find an exact solution. Your task is to find a good approximate solution of the *optimal* set of edges. Some of the approaches for finding an [approximate solution](https://en.wikipedia.org/wiki/Steiner_tree_problem#Approximating_the_Steiner_tree) to the Steiner tree problem, may suggest approaches that you can use here.

Example: Find an approximate solution to the Steiner tree problem

1. Find the minimal spanning tree for the subset of vertices that are destinations in any of your tickets
2. Remove any edges that are not required to fulfill any ticket.