Algorithm - Introduction

Iulian Rotaru

Mohamed Bentorcha

Part 1: Explaining why our solution runs in polynomial time

To begin with, I will explain the logic behind our algorithm. We start by sorting all the houses (leftmost/negative houses first and rightmost ones last). Once we have this list of sorted houses, our next step is to find the position of the snowplow. For this, we use a dichotomic search. Once we have this index, we clearly identify two options: going left or going right. The main question now is: How deep ?

We create two indexes at each extremity of the sides: left index and right index. Then we compute the preferable index with the following equation:

For left side: (right_house_count * leftmost_house_position) / left_house count

For right side: (left_house_count * rightmost_house_position) / right_house_count

The lowest score wins, and the other index is brought closer to the snowplow position by one. We keep doing this operation until one of the indexes crosses the snowplow. Once this occurs, we know for sure that taking the index that did not cross is the best solution right now.

We move the snowplow to the corresponding index, adding all houses on the path to the result. Then we remove this section from the initial list and start all this process again with the new position of the snowplow. In the end, the initial list will be empty and we will have a list of travels that should spare the most waiting time thanks to our estimation

This is the action tree of our program:

- Parcours => **1**
    - Merge Sort => **n * log(n)**
    - Main loop => **n**
        - Center Index search => **log(n)**
        - Limit Check Loop => **n - 1**

Parcours is our program, called once.

The Merge sort is called only

The Main Loop can make **2** to **n** iterations (**2** when we end up by doing a complete left search then complete right search or the opposite, **n** if we would change our direction for each house)

Center Index Search is a binary search

Limit Check Loop can iterate up to **n - 1** times (if we bring closer and closer the right and left indexes, one by one, **n - 1** is the maximum possible iterations. The minimal number depends on the position of the snowplow and the houses).

So the Main Loop has a total complexity of **n * log(n) + n ^ 2 - n**

Mixing all these operations, we have a total complexity of **n * log(n)** + **n * log(n) + n ^ 2 - n**

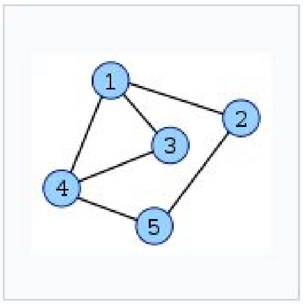
=> 2 * n * log(n) + n ^ 2 - n

Part 2: Line Graphs, Hamiltonian Path Problem and Eulerian Path Problem

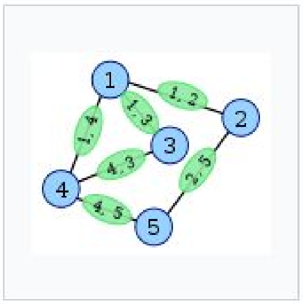
For the second part we decided to chose the Line Graph exercise. The Line Graph generator working with graphviz can be found in the line_graph directory.

Line Graph

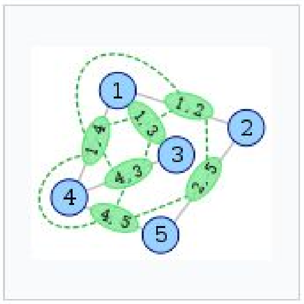
What is a Line Graph ? A Graph is composed of Vertices and Edges. The Line Graph of a Graph is the representation of the adjacency of edges on the graph. The edges of the graph are now represented as vertices, and linked to other adjacent edges.



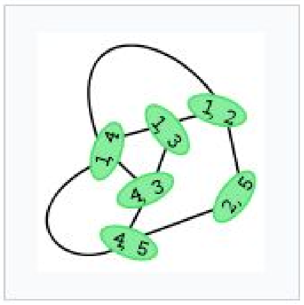
Graph G



Vertices in L(G) constructed from edges in G



Added edges in L(G)

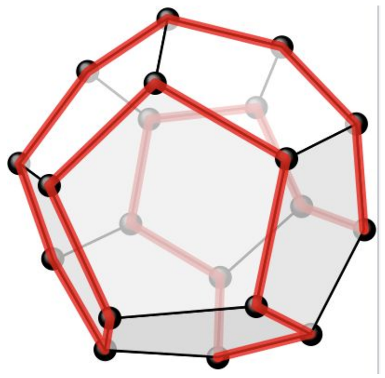


The line graph L(G)

This is the example given by wikipedia. The repository contains PDFs generated by Graphviz where I reused the initial graph provided by wikipedia and got the same final line graph.

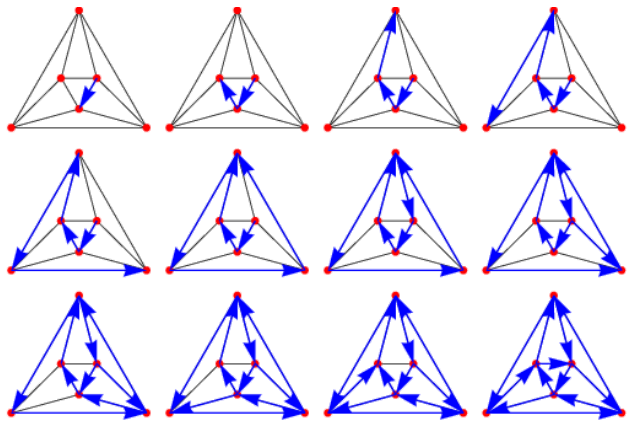
Hamiltonian Path Problem

The Hamiltonian Path Problem is a famous graph theory problem. The goal is to link all vertices together and form a complete loop between them, basically find a cyclic path that captures all the vertices. We call Hamiltonian a Path that is cycling in a graph and reaching all the vertices.



Eulerian Path Problem

The Eulerian Path Problem is similar to the Hamiltonian Path Problem, only its goal is to go through all the edges once. It should not be cyclic, but the path should cover all possible edges.



Where are we going ?

Line Graphs have a fundamental role in the Eulerian Path Problem and the Hamiltonian Path Problem: the Line Graph of any Eulerian Graph is Hamiltonian, and the Line Graph of any Hamiltonian Graph is Hamiltonian too ! A lot of famous algorithms use this property when trying to resolve Hamiltonian paths.