Introduction

We code the Gallager Humblet Spira (GHS) algorithm from scratch in this project. GHS
is an algorithm to construct Minimal Spanning Trees of graphs in a distributed setting.
To do so, we hold a major assumption: every edge in the graph has unique weights and
that, if every edge in a graph is unique, then the MST is unique.

Idea

The idea of GHS (briefly) is as follows:

- A fragment is a part of MST. The goal of the algorithms is to form fragments, join them, and terminate only when one fragment remains. The borders of the fragment are made up of nodes that have one outgoing edge inside the fragment and another outgoing edge towards another fragment. A single node can also be a fragment.
- Initially, each node is a fragment. Fragments fuse together to make larger fragments. Nodes in a fragment run a distributed algorithm to locate the overall minimal outgoing edge. On consensus or a general set of rules, fragments combine with the core edge being the new minimally outgoing edge.
- Each fragment has a unique name. Each fragment has a level. When two fragments combine, one of them changes their name and their levels stay the same / get updated depending on a set of rules.

Rules

Rules for the combination of two fragments

Let the combining fragments be F_1 and F_2 . Let their levels be L_1 and L_2 respectively. F_1 can only combine with F_2 only if L_1 < L_2 . If L_1 = L_2 , the fragments can combine when they reach a consensus on if the outgoing edge chosen by one of them is minimal for the other fragment too.

Rules for a level update on the combination of fragments

If $L_1 < L_2$, All nodes in the new fragment have names F_2 and level L_2 . If $L_1 = L_2$ and the outgoing edges $e_{F_1} = e_{F_2}$ for both the fragments, the new level of the combined fragment is $L_1 + 1$ and the new name of the fragment is e_{F_1} . If these rules don't apply, wait till these rules apply.

Algorithm

Refer to the slides provided in `Resources` to understand the algorithm

- `15.pdf` offers a good explanation of the functions and understanding pseudo code
- `mst.pdf` has the same pseudo code, but it is more readable

Implementation

Most of the message passing and receiving in openMPI is straightforward and has been implemented as such.
After initalization, every node stays in an infinite loop and acts based on the messages it has received until it receives the terminate message when it exits the loop. It then prints its branch edges and exits the program.
The main challenge in implementing the algorithm was how to implement the "wait" functionality. In the ideal implementation, you should answer all the wait queries after a level or name change, but there is no such functionality in openMPI.
We solved this by storing all the wait queries in a queue, and answering them whenever there are pending waiting queries. Worst case, there were no changes and so it still goes to waiting again.
We had use three local locking variables to make sure that if a message sent by p to q is waiting, p does not send the same message to q again, as this will create duplicate waiting queries which can cause unforeseen consequences.