Extending the capabilities of our graph

In this exercise we will build upon your implementation of a generic graph to make it more useable and useful.

Part 1: Reading from file

We want to be able to read generic graphs which are stored as human readable text files. This would not be much of a problem if it weren't for the genericity of the graph, allowing state and edge data to be arbitrary types.

Without JIT (just in time compilation), full genericity is not possible. However, we can - Strive to give a generic implementation that should work for almost all types - Dispatch dynamically based on the input file for a fixed set of predefined types - Facilitate possible integration of additional user-defined types

The input files have the following format:

- First line corresponds to a header composed of
  (State type) (Edge data type) [(number of states) (number of edges)]
- Every other line represents an edge
  (src state) (dst state) (edge data)

Note that the number of states and number of edges may or may not be present in the header and can be used for optimization (preallocation). They are not mandatory, however if present they are assumed to be correct.

Part 1: How to

The idea is to write ONE generic parsing function which is parametrized by the type of the state and edge data.

Then we can use successive template specialization to deal with the combinatorial explosion. Imagine the possible types for states and edge data are (int, unsigned, float, char, str). Then, we would have to use 25 `if-else` in order to obtain all possible combinations.

If you want to add a new state type: The joke is on you, you have to treat all combinations with edge data types. This is clearly not desirable/maintainable.

Instead we want a first helper function that only specializes on the state type (5 `if-else`) over all state types), which in turn calls a second helper function specializing on the edge data type (again 5 `if-else`). At this point all types are known and we can call the actual parsing function with the correct state and edge type.

Note do not forgot to restrain your templates with concepts where possible.

Note to simplify matters, if state / edge data type is char or str it can be assumed that they do not contain whitespaces.

Part 2: Shortest distance

No graph exercice is complete without a question on shortest distances
-ChatGPT

Recall THEG (Théorie des Graphes). There you have seen an exercice where edge weights have been semi-rings and one can compute *shortest distances* in this abstract setting.

Recall a semiring is a set R equipped with two binary operations $\oplus$ and $\otimes$ such that

- $(R, \oplus)$ is a commutative monoid with identity $\mathbb{0}$
- $(R, \otimes)$ is a monoid with identity $\mathbb{1}$
- $\otimes$ distributes over $\oplus$
- $\mathbb{0}$ annihilates $\otimes$: for all $a \in R$ $\mathbb{0} \otimes a = \mathbb{0} = a \otimes \mathbb{0}$

The standard distance computation over edges weights in $\mathbb{R}_0^+$ is called the *min tropical semiring* $(\mathbb{R} \cup \infty, \oplus = \min, \otimes = +)$ and $\mathbb{0} = \infty$ and $\mathbb{1} = 0$.

However you have also seen that you can compute the path with the highest chance of survival by using $([0., 1.], \oplus = \max, \otimes = *)$ and $\mathbb{0} = 0.$ and $\mathbb{1} = 1.$.

We want to code such a generic shortest distance algorithm.

Part 2: How to

If the type of the edge data verifies the *concept* of being a semiring, we want to use this semiring to compute the distances.

If the edge data is a "classical" number (int, unsigned, float, double), we want to compute the basic distance by default (min tropical semiring).

The notation $\oplus$ and $\otimes$ are only used here to disambiguate their meaning. To keep notation close to classical shortest path algorithms we will use $\oplus = min$ and $\otimes = +$. If the data type of the edges implements these operations, it can be considered a semi-ring. Note that $\mathbb{0}$ and $\mathbb{1}$ must also be accessible.

Part 2: expected

Your code must at least work for "normal numbers".

You should do something special for char and string to work.

As a bonus: Create your own semiring and adapt the code to work from file as well.