Uniregistry Challenge

Caveats

My background and experience are primarily windows development related as such I used C# and visual studio to develop. I chose to use .NET Core as that is platform independent so should be able to be run on any os. In the root directory I also included a Postman collection (Unireg.postman_collection.json) that has the calls to the api I tested with. The connection string for the db is contained in the appsettings.json

Due to the time crunch I used some libraries to speed up development. Some of these libraries I would probably not use in a real-life example because they can cause performance issues. One example of this is entity framework. I wrote most of the code first before attempting to connect with the database. As such my db table names are all upper case to decrease mapping code needed. I am aware this is not the standard for PostgreSQL and is more a pain to work with. I also had some challenges working with entity framework for postgres.

Part 1 (XML to graph)

The xml structure was quite straight forward so instead of a parser I decided to use a simple xmlserializer. The data objects are defined in GraphApi.Data project. Typically, I would not use the same objects that are retrieved from the db for so many purposes but due to time constraint I did. This makes the objects a little messier as some properties are only needed by entityframework.

For validation I used data annotations. This allowed me to define property requirements in an easily readable format. Looking at Graph.cs one can tell what validation occurs on the object. Some of the validation required custom validation attributes. These can be found in GraphApi.Framework.

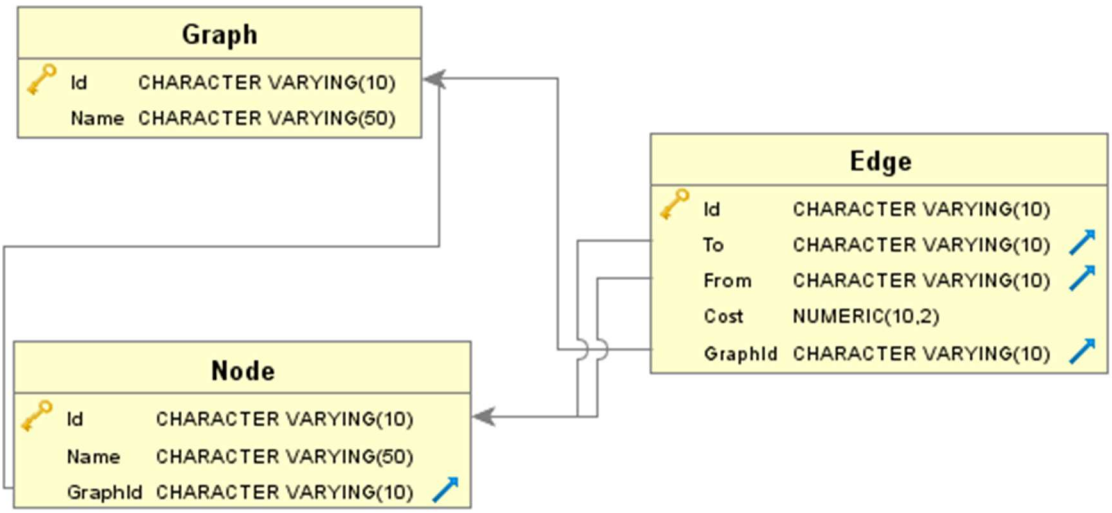
For the requirement that there must be only one from node and one to node I needed to put an event handler on the serializer. The serializer would have just ignored the additional properties.

The challenge did not specify how the program was to get the url to download the file but since the last part of the challenge required an api I added an upload endpoint that takes the following structure

```
{
"Url" : "http://localhost/Test1.xml"
}
```

Part 2 (Database)

Structure



The blue arrows indicate foreign key relationships.

Find Cycles Query

```
WITH RECURSIVE circular("To", "From", depth, path, cycle) AS (
        SELECT u."To", u."From", 1,
                ARRAY[md5(u."To")],
                false
        FROM "Edge" u
                                where "GraphId" = '1'
        UNION ALL
        SELECT u."To", u."From", cm.depth + 1,
                path || md5(u."To"),
                md5(u."To") = ANY(path)
        FROM "Edge" u, circular cm
        WHERE u."To" = cm."From" AND NOT cycle and  "GraphId" = '1'
        )
```

```
select * from circular where cycle order by depth desc
```

Part 3 (Queries)

Note : The json structure provided in the example was not valid (I put it through an online json validator). It requires {} around each of the objects in the queries array.

I used the [json.NET] to deserialize the json which is the go to json framework for .net.

The expected json is

```
{
	"graphid":"1",
	"queries": [{
	"paths": {
		"start": 'a',
		"end": 'e'
	}
	},
	{
		"cheapest": {
		"start": 'a',
		"end": 'e'
	}
	}
	]
}
```

For both queries I used code. There are sql procedures that can do this but they do not scale well. I was also able to utilize asynchronization to speed things up.

All paths.

Decided to use a recursive algorithm starting at start node and following the graph until either a cycle is detected, a node with no outgoing paths is reached or the ‘end’ node specified is reached. When a node has multiple outgoing nodes the recursive call is made asynchronously so multiple paths are investigated at once.

Cheapest path

Used the same as above but added a global variable that stores the cheapest path found. If the current path being investigated has a cost greater than the cheapest path found so far the path is abandoned so no additional work is done.