NOTE: ***The document is in draft mode till 10/12/2020***

TAU Advanced Topics in Programming - 2021A - Exercise 2

Amir Kirsh, Adam Segoli Schubert

Link to Ex1   **Direct link to changes and additions in Ex2**

Requirements and Guidelines

In this exercise you are requested to add the ability to navigate (suggest routes) based on your GIS data.

Navigation would be based on the following new classes:

```
enum class Direction {A_to_B, B_to_A}; // would be added to GISNamedTypes.h

// your new class
class Route {
public:
    const std::vector<std::pair<EntityId, Direction>>& getWays() const;
    const Coordinates& getWayStartPoint() const;
    const Coordinates& getWayEndPoint() const;
    Meters totalLength() const;
    Minutes estimatedDuration() const; // Minutes would be added to GISNamedTypes.h
                                       // as NamedType<Double<2>>
};                                     // ( 14.12 - changed from NamedType<int> )

// your new class
class Routes {
public:
    bool isValid() const; // routes are valid

    // following functions can be called only if isValid is true - undefined otherwise
    const Route& shortestDistance() const;
    const Route& shortestTime() const;

    // following function can be called only if isValid is false - undefined otherwise
    const std::string& getErrorMessage() const;
};

// The class below would be provided by the course staff, see Dependencies comment below
// This is a Proxy class for your GIS, the purpose of it is to:
// [1] restrict the navigation system to use only certain methods of the GIS class
// [2] allow monitoring the calls to GIS by the navigation system via this proxy
class NavigationGIS {
  const GIS& gis;
public:
  NavigationGIS(const GIS& gis): gis(gis) {}

  std::vector<EntityId> getWaysByJunction(const EntityId& junctionId) const {
       return gis.getWaysByJunction(junctionId);
  }

  std::pair<Coordinates, EntityId> getWayClosestPoint(const Coordinates& coords) const {
       return gis.getWayClosestPoint(coords);
  }

  std::pair<Coordinates, EntityId>
  getWayClosestPoint(const Coordinates& coords, const Restrictions& res) const {
       return gis.getWayClosestPoint(coords, res);
  }

  const Way& getWay(const EntityId& wayId) const {
       return gis.getWay(wayId);
  }

};
```

NOTE: ***The document is in draft mode till 10/12/2020***

```
// this is your main new class for ex2
class Navigation {
public:
    Navigation(const NavigationGIS& gis); // note that you get a const ref
    Routes getRoutes(const Coordinates& start, const Coordinates& end) const;
    // bonus methods - not mandatory:
    Routes getRoutes(
        const Coordinates& start, const Coordinates& end, const Restrictions& res) const;
};
```

The Navigation::getRoutes method gets a start point (that might be on a way or not) and an end point. The return value is a *Routes* object holding shortest by distance and shortest by time *Route* objects, each with the following data:

- A vector of all ways to travel on, in proper order.
- The ***start point*** on the first way closest to the source - this point must be a valid point on the first way in the vector (index 0) .
- The ***end point*** on the last way closest to the destination - this point must be a valid point on the last way in the vector.
- Distance and duration of the route (addition 15.12: from ***start point*** to ***end point*** as described above).

The bonus methods get an additional parameter of type “Restrictions” you should define this type and allow passing the information that toll road should be avoided with the following simple syntax: `Restrictions("toll").` A toll way is a way that is marked with the attribute “toll_road” = true .

You are required also to implement the following class for your tests:

```
// your new testing class for ex2
class NavigationValidator {
public:
    NavigationValidator(const GIS& gis); // note that you get a const ref
    bool validateRoute(
       const Coordinates& start, const Coordinates& end, const Route& r) const;
    // bonus method - not mandatory unless you go for the avoid_toll bonus:
    bool validateRoute(
       const Coordinates& start, const Coordinates& end, const Restrictions& res,
       const Route& r) const;
};
```

The validateRoute method gets a start point (that might be on a way or not), a destination point and a route, it then validates that the route information which corresponds to the given first two parameters - by using the GIS object that was provided to it in its constructor, but ***without*** *using the Navigation class!*
Note that the method is not required to check that this is the shortest route possible! Just that this is a valid route and the distance and time measured are correct.

Navigation Rules:

1. The first / last way in a route **cannot be** a highway - as you cannot stop on a highway - **unless** the provided point is on a highway (3 meters from a highway means on a highway, if there is no other closer way).
2. Note that the category_tags may have several tags separated by a comma (e.g. “highway”, “interstate”).
3. You should use way’s *speed_limit* field and assume: actual speed = speed_limit

Efficiency instructions

Your data structures shall be efficient enough to allow fast routing.
Your navigation class may cache data *during a single search of a route* (i.e. within one call) **but not between calls**, which means it shall not have member variables for caching data.
Note: in ex3 your navigation algorithm would be tested also for efficiency based on the number of calls to the GIS - less calls to the GIS is better.

Error Handling

The new API shall not throw exceptions nor should it write to any error log files. The only required error handling is to properly handle *Route::isValid()*.

Bonus

There are two bonus parts for ex2:

1. Supporting POIs polygon shapes, which is left as a special bonus from ex1 (see details in ex1).
2. Supporting a restriction on the route - specifically to avoid toll roads

To qualify for any one of the bonuses or both, you must adhere to the following instructions:

- Aiming for a bonus cannot break required behavior
- Include the *bonus.polygon* and/or *bonus.avoid_tolls* folder that demonstrates the applicability of your bonus implementation

Additions to GIS class and Entity Classes

The following methods shall be added to your GIS class:

```
std::vector<EntityId> GIS::getWaysByJunction(const EntityId& junctionId) const;
```

The function shall get EntityId of a Junction and return a vector of Ids of all the ways that start at this junction and those which end at this junction and are bidirectional. If the given Id is not known or is not of a junction the function shall return an empty vector (you may log an error).

```
std::pair<Coordinates, EntityId> GIS::getWayClosestPoint(
        const Coordinates&, const Restrictions& res) const;
```

The function shall get Coordinates and Restrictions and return the closest Coordinates along a Way that complies to the restrictions provided and the EntityId of this *way*, Restrictions can be provided as: Restrictions("highway") means avoid highway, Restrictions(" toll ") means avoid toll, Restrictions("highway , toll ") or Restrictions("toll,highway") both mean avoid highway and toll, etc.

```
const Way& getWay(const EntityId& wayId) const;
```

The function shall get EntityId of a Way and return the Way itself. In case the id does not represent a valid way throw an exception of your choice.

Also, you should now have a well defined Way class with the following methods (you may have additional methods if you need):

```
class Way {
public:
    std::pair<EntityId, EntityId> getJunctions() const; // from, to
    Meters getLength() const; // might be lazily calculated (with a mutable field)
    bool isBidirectional() const;
    int getSpeedLimit() const; // as Km/h <- added on 14.12
    bool isHighway() const;
    bool isToll() const;
};
```

NOTE: ***The document is in draft mode till 10/12/2020***

Parts of the exercise

Tests
We would provide a few simple google tests that present how the API can be tested.
You shall add 3 additional tests of your own, to have a better test coverage of your code.
The tests shall include relevant map files.

Submission
You shall submit a zip named ex2_<student1_id>_<student2_id>.zip that contains:

- ***makefile -*** use the same template makefile as in ex2.
- ***students.txt*** - a text file that includes one line per submitter: <user_name> <id> (do not include the character '<' nor '>'
- ***README.md*** - info and remarks about your implementation
- ***bonus.polygon*** folder if you choose to implement this bonus
- ***bonus.avoid_tolls*** folder if you choose to implement this bonus
- **Your code**
- ***tests*** folder containing at least 3 google tests in "*tests.cpp*" file that test functionality introduced in ex2. Relevant map files for your tests should be included in the aforementioned ***tests*** folder.
- The *NavigationValidator* utility test class - **inside your tests folder**

DO NOT SUBMIT THE FOLLOWING FILES

- binary files
- external libraries (you may only use standard C++ libraries and rapidjson)

Testing Environment

The exercise should be implemented to run on a Linux environment, specifically on the docker created for the course.
For details about the testing environment see:
https://docs.google.com/document/d/1IEQFSW7Q6sNpfdmHmAfnSammGpEqb075cgNj9KX6N_c/edit?usp=sharing
Make sure to either develop directly in this environment or to leave enough time to integrate and test your code on it.

Additional Notes and Requirements

1. **More than one corresponding "best" route**
   - If there is more than one shortest route (i.e. two or more with the same distance / same travel time) prefer the one which is **better for both distance and time**
   - if the above rule doesn't result with a single route, prefer the routes resulting from the above rule, with **minimal number of ways** (i.e. minimal junctions)
   - If the above rules do not result with a single route, return any of the routes resulting from above rules (an arbitrary one).
2. **Direction of driving:** assume the direction of the car is not given, thus for bidirectional way, the possible driving direction can be in both directions (for start and end points).
3. **Additional SW requirements**
   - You should not have "new" and "delete" in your code from ex2 and on, you may use smart pointers but you are not obliged to do so (do it only if it helps your code)

- Wherever “move” is required, make sure to properly use the rule of zero (default move) or declare a default move, or implement move with a proper noexcept signature! This applies for both move ctor and move assignment. Also make sure to use *std::move* where and if relevant (but not where not relevant!).
- Remember to block your copy constructor and assignment operator (with *=delete*) in case the default is not good and you do not need these operations. Do not block them if the default is fine.
- Remember to have a virtual destructor if the class is a base of a polymorphic inheritance hierarchy (i.e. you use a pointer or reference to base to manage derived objects, including through destruction).
- Make sure to go through the code remarks that would be published for ex1 and do the proper changes if necessary.

4. **Dependencies**
   You should use the dependencies provided by the course staff from a git repository that would be published. This would include: *NamedType.h*, *Double.h*, *GISNamedTypes.h*, *CoordinatesMath.h, NavigationGIS.h*

5. **Additions following students questions in the course forum**
   (a) The part of the first and last way that is **not part of the Route** is to be calculated as **the total length** of the first/last way **minus air distance** from the start/end location to the junction **that is part of the Route**.
   (b) For A* algorithm you should assume a max_speed_limit: *assume **130 km/h** using a proper constant that you can add in a proper code location.*
   (c) For A* algorithm as well as for item (a) above, there is a need to get the location of a given junction. In Ex.2 you are allowed to use any addition that you create to the API for that purpose. In Ex3 the addition to the API would be uniform and defined and you’d have to use it - two new methods on class Way:

```
        const Coordinates& getFromJunctionCoordinates() const;
        const Coordinates& getToJunctionCoordinates() const;
```

Note that you may have some open questions. If the questions are on the *requirements* - open a new topic in the course forum. If the questions are on the design of your solution: this is your ballpark, you should decide on a design that fits the requirements. However you may still ask design questions in the forum, but such questions should include your thoughts and considerations, what are the options you have in mind and why you hesitate to take a certain approach.

A few notes towards next exercises:

Exercise 3
In ex3 you would be required to implement a simulator that can run a tournament of navigation algorithms over different maps and routes. This exercise would also include a bonus for the best navigation algorithms.
The exact requirements would be published later on.

Exercise 4
Exercise 4 is a separate final exercise, to be submitted individually.
(It will not be based on the code developed in previous exercises).

Good Luck!