TAU Advanced Topics in Programming - 2021A - Exercises

Amir Kirsh, Adam Segoli Schubert

Requirements and Guidelines

In the exercises, this semester, you will be requested to implement a GIS (Geographic Information System) for navigation - to allow planning travels from Coordinates A to Coordinates B.

The GIS data is based on:

- *Coordinates*: described by Longitude and Latitude
- *Point of Interest*
- Junctions and roads, represented as coordinates connected by ways/arcs
- A separation of data into blocks or cells, to allow quick search of given coordinates (which road I'm on? what is near me?) without the need to search the entire map

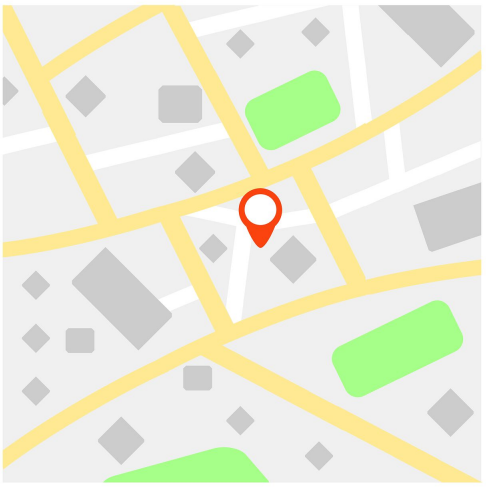
---

To get a glimpse into the domain of GIS and navigation systems, data structures and existing applications see the links below (note that the links are for *general information only*, the exact requirements are detailed below and may vary from the way other systems manage the data):

- https://wiki.openstreetmap.org/wiki/Node
- https://wiki.openstreetmap.org/wiki/Way
- http://wiki.gis.com/wiki/index.php/Main_Page
- https://www.e-education.psu.edu/geog468/l2_p3.html

Algorithms for navigation problems (relevant for the ex2 and ex3, not for ex1):

- https://en.wikipedia.org/wiki/Dijkstra's_algorithm
- https://en.wikipedia.org/wiki/A*_search_algorithm
- https://en.wikipedia.org/wiki/Contraction_hierarchies
- https://en.wikipedia.org/wiki/Category:Routing_algorithms



Exercise 1

In the first exercise we shall deal with organizing the GIS data and providing a structured API for searching data.

The following entities is to be managed by your GIS system:

- ***coordinates***, described by either:
    - An ordered pair of [longitude, latitude]
    - An ordered sequence of (JSON array) of [longitude, latitude] pairs.

Types:

- ***Entity***, described by:
    - [optional in loaded files / mandatory in saved files] id: unique across all type of entities (string)
    - name: a short descriptive name (string)
    - [optional] description: longer description (string)
    - [optional] category_tags: a list of tags (strings) - may be empty

- ***POI* (point of interest)**, described by:
    - all *Entity* fields
    - [optional] accessibility: may be tagged with accessibility flags (“by car”, “by foot”, “by wheelchair”)
    - geometry: there are *three* supported types:
        - Circle:
            - *coordinates:* as center of circle
            - Radius: in meters
        - Polygon:
            - A collection of coordinates: a sequence of at least 3 different coordinate pairs). Note that order of coordinate pairs matters
        - ~~Collection of shapes. Note that it is legal to have combination of shapes that share common area and it is also legal to have combination of shapes that do not share common area and are not adjacent (i.e. creating a shape with disconnected parts)~~

  Note: the section marked in yellow, above, is a bonus for ex2.

- **Junction**, described by:
    - all *Entity* fields
    - *coordinates*

- **Way**, described by:
    - all *Entity* fields
    - *from*: EntityId of a junction
    - *to*: EntityId of a junction
    - [optional]: curves: coordinates setting the Way’s layout, in order (*from* => *to*)
    - direction: “bidirectional” (two-ways) OR “unidirectional” (*from* => *to*)
    - speed_limit: in km/h
    - toll_road: true / false

- [optional] restricted: unauthorized traffic (multiple selection), such as: “public transportation” / “trucks” / “cars” / “motorcycles” / “horses” / “bicycles” / “pedestrians”
- **Note:** the optional category_tags (listed in Entity) may be used to tag way: such as “highway”, “interstate”, “local road”, etc.

Map File

You should be able to parse a map file, as described below, into a GIS object. You may use http://rapidjson.org/ for doing so. It is already installed on our docker container (see below).

See map example below (legend, in grey: `optional entry`):

```
[
    {
        "type": "POI",
        "id": "P1001",
        "name": "Washington Square Park",
        "description": "A public park in Lower Manhattan",
        "category_tags": [],
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [40.731051, -73.999611],
                [40.729606, -73.996602],
                [40.730805, -73.995674],
                [40.732179, -73.998624]
            ]
        }
    },

    {
        "type": "POI",
        "id": "P1002",
        "name": "Washington Square Fountain",
        "description": "",
        "category_tags": ["Public"],
        "geometry": {
            "type": "Circle",
            "coordinates": [40.730811, -73.997455],
            "radius": 0.635
        }
    },

    {
        "type": "Junction",
        "id": "J1001",
        "name": "5th Ave & Washington Square N",
        "description": "",
        "category_tags": [],
        "coordinates": [40.731437, -73.996967]
    },

    {
        "type": "Junction",
        "id": "J1002",
        "name": "5th Ave & 8th St",
        "description": "",
        "category_tags": [],
        "coordinates": [40.732254, -73.996394]
    },
```

```
    {
        "type": "Junction",
        "id": "J1010",
        "name": "6th Ave & 8th St",
        "description": "6 Ave & Greenwich Ave & 8th St",
        "category_tags": [],
        "coordinates": [40.732254, -73.996394]
    },

    {
        "type": "POI",
        "id": "P1010",
        "name": "Elmer Holmes Bobst Library",
        "description": "The main library at NYU",
        "category_tags": [],
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [40.729824, -73.997302],
                [40.729541, -73.996723],
                [40.729055, -73.997136],
                [40.729338, -73.997717]
            ]
        }
    },

    {
        "type": "Way",
        "id": "W2001",
        "name": "5th Ave b/w Washington Square N & 8th St",
        "description": ".*",
        "category_tags": [],
        "from": {
            "entity_id": "J1002"
        },
        "to": {
            "entity_id": "J1001"
        },
        "direction": "unidirectional",
        "speed_limit": 40,
        "toll_road": false,
        "restricted": ["trucks"]
    },

    {
        "type": "Way",
        "id": "W2002",
        "name": "8th St b/w 6th Ave & 5th Ave",
        "description": ".*",
        "category_tags": [],
        "from": {
            "entity_id": "J1010"
        },
        "to": {
            "entity_id": "J1002"
        },
        "curves": [[40.733566, -73.999446]],
        "speed_limit": 40,
        "direction": "unidirectional",
        "toll_road": false,
        "restricted": ["trucks"]
    }
]
```

API

Your GIS object should support the following API:

Types

```
template<typename T>
class NamedType {
	T val;
public:
	explicit NamedType(const T& t): val(t) {}
	operator const T&() const {
		return val;
	}
};

struct Longitude: NamedType<double> { using NamedType<double>::NamedType;  };
struct Latitude: NamedType<double> { using NamedType<double>::NamedType; };
struct Meters: NamedType<double> { using NamedType<double>::NamedType; };

struct EntityId: NamedType<std::string> {
  using NamedType<std::string>::NamedType;
};

~~using Coordinates = std::pair<Longitude, Latitude>;~~ // NOTE - changed to:
class Coordinates {
	Longitude _longitude;
	Latitude _latitude;
public:
	Coordinates(Longitude longitude, Latitude latitude) :
_longitude(longitude), _latitude(latitude) {}
	Longitude longitude() const { return _longitude; }
	Latitude latitude () const { return _latitude; }
	bool operator==(const Coordinates& rhs) const {
		return _longitude == rhs._longitude && _latitude == rhs._latitude;
	}
};
```

Functions

Note: all functions below may or may not be const (i.e. may oblige not to alter the GIS system, or decide not to oblige), the tests would not assume constness of the functions, but you can make them const if you see it appropriate.

```
std::size_t GIS::clear();
```

the function clears all data from the system and returns the number of entities that were cleared

```
std::vector<EntityId> GIS::loadMapFile(const std::string& filename);
```

the function loads a map file and returns a vector of all the EntityIds that were loaded (either Ids that appeared in file or generated) in the exact order as the objects that appeared in the file

- in case the system had existing data, the data loaded from file is added to the existing data
- in case an entity in file doesn't have an Id, a new Id would be generated for it automatically
- in case entityId from file exists already the old data related to this id is replaced with the new data - (see NOTE remark below!)

NOTE:

The requirements for this section have been relaxed: You may decide not to support Entity updates. In this case YOU MUST write an error message into *errors.log* when an existing ID is provided (treating it as a bad entity, as described here).

Regarding loading mixed map files (some entities have ids and others don't): You may accept all entities or decide to only accept one of the cases, i.e. the ones with ids or the ones without.

`std::size_t GIS::saveMapFile(const std::string& filename);`

the function saves all data into a map file and returns the number of entities that were saved, the saved entities must include their Ids

`std::vector<EntityId> GIS::getEntities(const std::string& search_name);`

the function returns a vector of EntityIds for all entities that match the search_name
all exact matches (case sensitive) shall be first in the vector
right after shall ***optionally*** appear "partial matches", in any order (e.g. case insensitive match, matching based on the longer description, match based on the category tags etc. partial match may work differently when the search is coordinates based or not - note: partial match implementation is not mandatory, however it may grant you a bonus.
To qualify for bonus you must include in your submission zip a folder named *bonus.partial_search*, that contains **three** items:

- *partial_search.txt:* explaining what you did and why qualify for bonus - include where code handels this exceptional partial search
- *mapFile.json* with 10-20 entities
- A file named: partial_search_test.*cpp* that includes a google test that **passes** and demonstrates your exceptional search

`std::vector<EntityId> GIS::getEntities(const std::string& search_name, const Coordinates&, Meters radius);`

same as above, but restricted to a search area of the given circle

~~`std::vector<EntityId> GIS::getPOIs(const std::string& search_name);`~~

~~the function returns a vector of EntityIds for all POIs that match the search_name~~
~~all other instructions are the same as in the previous function~~

~~`std::vector<EntityId> GIS::getPOIs(const std::string& search_name, const Coordinate&, Meters radius);`~~

~~same as above, but restricted to a search area of the given circle~~

`std::optional<Coordinates> GIS::getEntityClosestPoint(const EntityId&, const Coordinates&);`

the function shall get EntityId and Coordinates
if the Id is not known, the function returns an empty optional
otherwise, return the closest Coordinates, on entity's perimeter, to the provided Coordinates.
(the Coordinates shall be a point on a the *entity's geometry* that is closest to the given Coordinates)

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

the function shall get Coordinates and return the closest Coordinates of a Way (the Coordinates may be a point <u>on</u> a the *way* that is closest to the given Coordinates) and the EntityId of this *way*

~~`std::vector<EntityId> GIS::getPOIsInCoordinates(const Coordinates&);`~~

~~the function shall get a Coordinates and return a vector of all entity Ids of POIs that contain this Coordinates inside their "area" including on the "perimeter" of the POI, the returned vector may be empty~~

~~`std::vector<EntityId> GIS::getPOIsNearCoordinates(const Coordinates&, Meters radius);`~~

~~the function shall get a Coordinates and radius and return a vector of all entity Ids of POIs that touch or contain partially or in a whole this circle, the returned vector may be empty~~

Efficiency instructions

Your data structures shall be efficient enough to allow fast searches.
Assume that there might be millions of POIs, Ways, etc.
Manage entities in squares of a configurable size. Test different square sizes in your tests!
Make sure to test searches that go beyond a single square.
Manage names in a way that would allow fast searches.
Add to your README.md a description of your design considerations for efficiency.
Note: this is part of the requirements and would be checked as part of the code check.

Error Handling

- Program shall never crash - in any case and regardless of bad input!
- Ignore bad entities in the map file, log into *errors.log* file (~~on the same directory of the provided map file~~ `// NOTE - changed to:` in the current working directory[1]) all the errors that were detected - a line per error. For example: there shall be a line per bad entity.
- Exact error messages in log file is up to you - however a bonus may be given on proper error messages, indicate inside the *readme.bonus* file the special error handling and the tests that you added to check it
- To qualify for bonus you must include a folder named *bonus.error_handling*, containing **four** items:
    - error_handling.*txt: explaining what you did and why you qualify for bonus*
    - *mapFile.json* with **no more** than 10 entities
    - errors.gold.log
    - A file named: error_handling.*cpp including* a google test that **passes** and produces the *errors.log* **identical** to the provided *errors.gold.log*

Bonus

Two items in this ex1 are classified as bonus:

- Partial string search in the proper API functions, as described above
- Advanced error handling, as described above

To qualify for bonus points you must adhere to the following instructions:

- Aiming for a bonus cannot break required behavior (e.g. you cannot “improve” the partial search by not providing all the exact search results first, you cannot “improve” error handling by having more than a single line per error in the errors log file)
- Include the *bonus.partial_search* and/or *bonus.error_handling* that demonstrates the applicability of your bonus implementations

[1] To simplify the exercise specification, **all errors must be logged into a single file**, ***errors.log***.
This file will be created **in the current working directory**.

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 ex1_<student1_id>_<student2_id>.zip that contains:

- ***makefile -*** see template makefile here.
- ***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.partial_search*** and/or ***bonus.error_handling*** folders as described above
- Your code + your tests
- ***tests*** folder containing at least 3 google tests in "*tests.cpp*" file. Relevant map files for your tests should be included in the aforementioned ***tests*** folder.

DO NOT SUBMIT THE FOLLOWING FILES
The following files shall not be submitted:

- 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

Note that you may have some open questions. If the questions are on the *requirements* - open a new topic in the course forum (questions on exercise 1). 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 2

In ex2 you would be required to implement two navigation algorithms that would inquire the GIS model to get a proposed route from a given Coordinates A to B, based on specific requirements (e.g. no toll roads). The navigation algorithms would be implemented as independent modules, separated from the GIS itself.
You would also be required to implement a test that would get a route, check that it is indeed a valid route between the provided Coordinates and calculate the total distance and time on that route.
The exact requirements would be published later on.

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!