Marc Bachelet
Michael Linder

Project report : Archipelago

1 Workflow

1.1 Organization

In the first part of the project, until Rendu1, the focus being mainly building up a good data structure, the code was mainly produced together since it is difficult at this stage to work on different parts without having a different view of the model. Then, for Rendu 2 and 3, the project was mainly coded independently, each one having different responsabilities :
while Michael mainly coded the gui, its events and dijkstra, Marc managed the subclasses of the model and its functionalities such as reading files, criteria, modifying the city through the gui etc.
This method, in hindsight, could have been problematic since it can lead to a misfunctioning of the modules when they are assembled. However, planning the implementation of the modules before made it possible to reduce those unpleasant surprises.

1.2 Order of work

After a planning on the data structure and responsibilities of each module we built a minimal working model bottom-up (tools → node → city). Afterwards, having implemented and tested the basic functionalities of the model, we could simultaneously add new functionalities in different places and test them one by one.
Looking back, creating the gui before rendu 1 in order to better debug and test our model turned out to be very practical since it is often difficult to understand where errors can occur without having a graphical interface. Having a nearly finished gui allowed us to add step by step the functions for each button and click event.

1.3 Most annoying bug

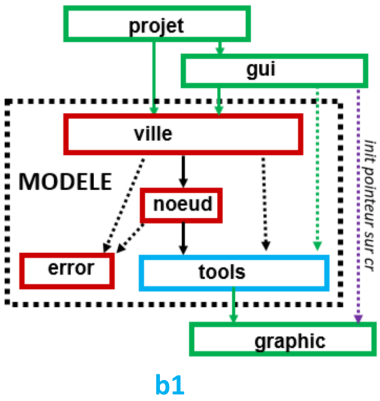
While testing our nearly finished project we encountered randomly appearing segmentation faults. Strangely, defining an empty vector in city solved the problem. Our theory was that we corrupted memory that was now occupied by this new vector. We did not encounter a segfault because the leak was never detected (the vector was never used). Removing the empty vector and using valgrind we found out the memory leak happened while deleting connections from a node that gets deleted. It turned out out our del_connection function manipulated a map to which del_node, who called del_connection, had a pointer stored. This caused a wrong directed pointer which led to the segmentation fault. We solved this by using map.find again instead of storing the previously found position.

1.4 Most frequent bug

We had some bugs due to wrong datatypes of local variables in intermediate steps. An example is an integer variable used while calculating the MTA criteria. This resulted in intermediate results being rounded to whole numbers and therefore to a wrongly calculated MTA criteria.

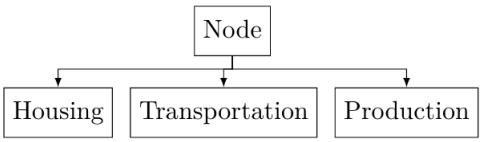
2 Architecture and implementation

2.1 Architecture



The architecture we chose for the project is the **model b1** as described in the project definition.
In addition to initializing the context pointer through the graphic module, the gui accesses other information like the zoom aswell.

2.2 Class hierarchy



The class node has three subclasses, each of them describing a node type.
However, our data structure relies on `std::maps` which are not able to store heterogenous data and in order to avoid the use of pointers, we decided to use the subclasses only for their constructors.
Once the node is stored in the map, it is no longer a specific node type, but its caracteristics have been set as attributes by the constructor of the subclass.
This way of inserting the node into the data structure has the main advantage of leaving it possible to add new node types for example, just by defining a new constructor.

2.3 Data structure

The data structute of the city, stored in the module city, consists in three maps :

- `node_by_uid` : each node is indexed
- `connections_by_id` : each connection is indexed
- `outgoing_connection_id_by_uid` : each uid is associated to a vector of outgoing connection ids

The choice of maps has mainly been made because it provides several advantages - especially compared to a vector. The short lookup time as well as the reasonable insertion complexity of O(log(n)) which are totally independent from the position in the map turned out to fit well the needs of the project.

2.4 Model - who does what ?

In order to make use of the principles of object oriented programming, the model is split into several different objects which we list below :

`City`	This is the actual model, containing the city data structure (the maps) and the methods which make it possible to add/delete nodes/connections. It test for all the possible errors internally.
`City::Dijkstra` `City::Criteria`	As the functionalities of the dijkstra algorithm and criteria measurement are very close to the model (and need a direct access to the data structure) we decided to implement them as nested classes of the city itself.
`Filereader` and `Filewriter`	Quite self-explaining, these two tools are used by the Gui to open and save files.
`City_Gui`	It is through City_gui that the gui communicates with the city. We found it important to seperate the basic functioning of the city and the more complex functionalities that the gui makes possible. For example, it is possible to change the size of a node through the gui, however, it is a responsibility of City_Gui to update the connections around the node.

2.5 Dijkstra

We changed the proposed algorithm because we observed that iterating trough a vector containing every node of the city takes a lot of time for big cities. Given the context of city planning a production and transport node is most likely reached within one to three neighbors. Therefore we started with a vector containing only the starting node and added every neighbor of a visited node later on. See Algorithm for the pseudocode of our implementation. We use a boolean time_only to backtrack and display the shortest path only when needed.

The pseudocode uses the following functions :

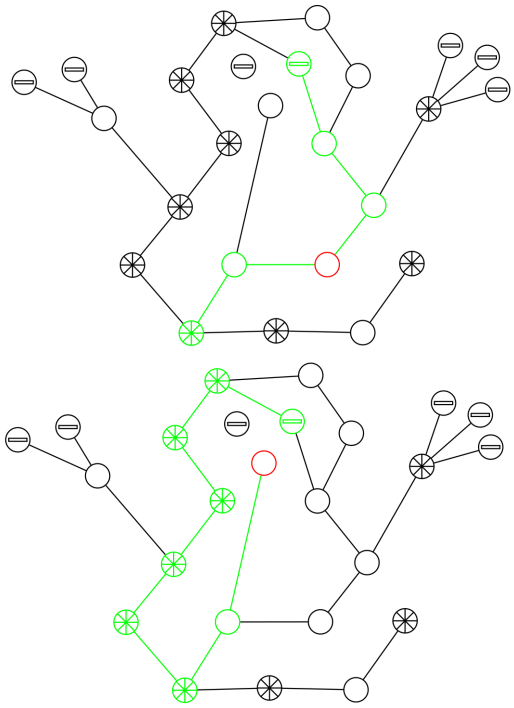
`init_data`	Define time_by_id and set the access time for the starting node to 0. Define parent_by_id and set the parent to no_link for every node. Define and set fastest_production_found and fastest_transport_found to false, fastest_production_time and fastest_transport_time to zero
`fastest_access_time`	return the lowest access time in time_by_uid.
`fastest_access_id`	return the id of the node with the lowest access time in time_by_uid.
`backtrack_shortest_path`	takes an id as an argument and uses parent_by_uid to store all nodes and connections that are part of the shortest path by going from parent to parent until reaching no_link.
`Connection::get_other_end`	takes an id as an argument and returns the id of the other node the connection connects to

We denote the access of a element in a map with map_name[key].

Pseudocode : Dijkstra

```
Require: nodes_by_uid, connections_by_id, outgoing_connections_id_by_uid, time_only

  init_data()

  while time_by_uid is not empty and not (production_found and transportation_found) do
    fastest_access_time ← get_fastest_access_time()
    fastest_node_id ← get_node_id_lowest_access_time()
    fastest_node ← nodes_by_uid[fastest_node_id]
    Remove fastest_node_id from time_by_uid

    // check if fastest transport or production found
    if fastest_node.get_type() = TRANSPORT and not fastest_transport_found then
      fastest_transport_found ← true
      fastest_transport_time ← fastest_access_time
      fastest_transport_uid ← fastest_node.id
    if fastest_node.get_type() = PRODUCTION and not fastest_production_found then
      fastest_production_found ← true
      fastest_production_time ← fastest_access_time
      fastest_production_uid ← fastest_node.id
      Continue with next iteration of while loop

    // update neighbors
    for connection_id in outgoing_connections_id_by_uid do
      connection ← connections_by_id[connection_id]
      neighbor_id ← connection.get_other_end(fastest_node)
      travel_time ← connection.get_travel_time() + fastest_access_time
      if time_by_uid[neighbor_id] ≥ travel_time then
        time_by_uid[neighbor_id] ← travel_time
        parent_by_uid[neighbor_id] ← fastest_node_id
        Add (neighbor_id, time_by_uid) to time_by_uid

  // backtrack shortest path and return time
  if time_only = false and production_found = true then
    backtrack_shortest_path(fastest_production_id)
  if time_only = false and transport_found = true then
    backtrack_shortest_path(fastest_transport_id)
  return fastest_transport_time, fastest_production_time
```

2.6 Examples of shortest path



One typical example where there are two different paths, one leading to the nearest production node and the other leading to the transport node.

This other example is interesting to show **the different speeds a connection** can have. Instead of taking the path through the housing nodes which is shorter in length, the fastest path is in fact the longer path through the transport nodes for which the connection speed is 4 time faster.

Conclusion

Looking back, we think we worked fairly well paced and never had time problems as we were always about 2 weeks ahead of the normal schedule. However, this led to us creating some over-complicated solutions as we lacked the know-how. One example is the coordinate transformation and change of the window-size which we hardcoded without the functions showed in the course later on.
But this puzzeling was definitely a nice part of the work. We would have loved to do even more problem solving instead of just coding. A possibility to implement this in the project would have been for example not to give us the dijkstra algorithm but letting us search an algorithm on our own.