```
struct Town {
        TownID town_id = NO_TOWNID;
        Name town_name = NO_NAME;
        Coord coords;
        int tax = NO_VALUE;

        Town *master = nullptr;
        std::vector<Town*> vassals;

        std::unordered_map<Town*, int> roads_to;

        std::string color = "white";
        int source_dist = -1;
        TownID arrived_from = NO_TOWNID;
        Town* went_to = nullptr;
        bool loop_cause = false;

        // We want to use negative values
        // so that priority queue gives the cheapest
        // Dijkstra...or more precisely A*
        uint dijkstra_d = 1;
        uint dijkstra_de = 1;
        bool re_relaxed = false;

    };
```

Structissa on vassalsista eteenpäin apumuuttujia (kutsun niitä kommenteissa myös flageiksi), joiden avulla voidaan toteuttaa BFS, DFS, A* sekä Kruskal. Dijkstraa (eli oikeastaan A*) varten oli otettava käyttöön uint, jotta voidaan negatiivisilla luvuilla hieman huijata priority_queueta järjestämään lisätyt alkiot meidän kannaltamme suotuisella tavalla. Muuttuja re_relaxed on avuksi tilanteessa, jossa jo harmaaksi merkitylle nodelle lasketaan uusi prioriteetti.

Public:
`std::vector<TownID> depth_first_search(TownID fromid, TownID toid, std::vector<TownID> route);`
Hakee reitin kahden kaupungin välillä, kutsutaan “any_route” sisällä. DFS sen takia, että se hakee minkä tahansa ensimmäisenä löytämänsä reitin ja palauttaa sen.

`bool breath_first_search(TownID fromid, TownID toid);`
BFS:n logiikan vuoksi ensimmäinen palautettu reitti on samalla lyhin nodejen lukumäärällä mitattuna. Kutsutaan least_towns_route sisällä.

bool **depth_first_cycle**(TownID startid);
Jälleen tarkastellaan DFS avulla, että löytyykö silmukoita lähtökaupungista lähtien. Kutsutaan road_cycle_route:n sisällä.

bool **dijkstra_enhanced**(TownID start, TownID finish);
Oikeammin kyseessä taitaa olla A*-algoritmi. Etsii lyhintä mahdollista reittiä matkan pituuden pohjalta. Ei siis nodejen lukumäärän, vaan teiden pituuden. Kutsutaan shortes_route sisällä.

Private:
std::multimap<int, std::pair<Town*, Town*>> road_db;
Tänne talletetaan kopio jokaisesta olemassa olevasta tiestä. Järjestyy automaattisesti tien pituuden mukaan. Hyödyllinen Kruskalin trimmausalgoritmin kannalta.

void **relax_A**(Town &u, Town &v, Town &goal);
Kutsutaan dijkstra_enhancedin sisällä. Lasketaan uudet kustannukset ja kustannusarviot prioriteetin muodostamista varten.

void **reset_town_flags**();
Apufunktio apumuuttujien alustukseen, jota tarvitaan etenkin graafialgoritmejä toteuttavissa funktioissa.

MUUTA HUOMIOITAVAA:
Sain trim_road_networkin trimmaamaan verkostoa ihan oikein, mutta välillä testreadillä lukemalla esimerkkitiedostoja ohjelma kaatuu. Voin syöttää testitiedostosta samat komennot käsin ja ohjelma ei kaadu. Ei se toisaalta yleensä kaadu myöskään testitiedostoja suoraan lukemalla. Jäi vähän mysteeriksi, mutta aika loppuu.