Линк към хранилище в Github: <https://github.com/danisini/CityMap>

**Проект „Карта на град“**

1. **Увод**

**Описание и идея на проекта**

Разглежданият проект реализира карта на град. Информацията за картата се съдържа в текстов файл. На всеки ред във файла на първа позиция стои името на дадено кръстовище от града, а след него – двойки от имена на други кръстовища и дължината на улицата, свързваща първото кръстовище с тях.  
   
  
**Цели и задачи на разработката**

* Подходящо представяне на картата и определени пътища през града
* Изграждане на леснодостъпно меню.
* Реализиране на основни функционалости на проекта
* Реализиране на допълнителни фунцкионалости на проекта (меню, интерактивен режим)
* Подходяща реализация за четене от текстов файл
* Подходящ подбор на алгоритми за реализация на функционалностите

**Структура на документацията**

Документацията е структуриране в следните основни точки:

1. Увод
2. Проектиране
3. Реализация
4. **Проектиране**

По време на работата се стреми да се следват добрите принципи и практики на ООП дизайна. Реализирани са следните 3 класа и една структура:

* Menu
* CityMap
* Path
* Street структура

Те имат следната йерархия:

* **Клас Menu**: реализира менюто

**Private** член-данни и функции:

* CityMap cityMap; // картата на града
* void help(); // показва командите, които ще са достъпни

**Public** член-функции:

* Menu(); // конструктор
* **Клас CityMap -** реализира картата на града

**Private** член-данни:

* std::unordered\_map <std::string, std::unordered\_map<std::string, double> > adj; // списък на съседство
* std::unordered\_map <std::string, std::vector<std::string> >adjRev; // обърнат списък на съседство
* std::unordered\_map <std::string, int> crossroads; //кръстовища
* std::unordered\_map <std::string, bool> used; //масив за посетени върхове при обхождане
* std::unordered\_map <std::string, bool> closedCrossroads; // затворени кръстовища
* int numCross; // брой кръстовища на картата
* std::string currCrossroad; // текущото кръстовище, на което се намираме
* bool interactive; // казва дали картата е в интерактивен режим

**Private** член-функции –

* void dfs(std::string); // обхождане в дълбочина
* void dfsRev(std::string); // обхождане в дълбочина, но с обърната посока на ребрата
* bool dfsCycle(std::string, std::string); // използване на обхождане в дълбочина за откриване на цикъл
* std::queue<std::string> eulerCycle(std::string); // открива Ойлеров цикъл
* bool isStronglyConnected(); // връща истина, ако картата е силносвързан компонент и лъжа – в противен случай

**Public** член-функции:

* void open(std::string); // функция за взимане на данни от файл
* std::string isThereAPath(std::string, std::string); // проверява дали има път между две кръстовища
* std::string fromOneToAll(std::string); // проверява дали е възможно от едно кръстовище да се стигне до всички останали
* std::vector <Street> deadEnds(); // връща задънени улици
* std::string possibleToReturn(std::string); // проверява възможно ли е да тръгнеш от едно кръстовище и после да се върнеш там
* void possibleToVisitAllStreetsOnce(); // проверява за възможна туристическа обиколка на целия град, минавайки веднъж по всяка улица
* void dijkstraForKPaths(std::string, std::string); // извежда трите най-кратки пътя между две кръстовища
* void dijkstraForKPathsWithClosed(std::string, std::string, std::vector <std::string>); // извежда трите най-кратки пътища между две кръстовища по зададени затворени такива
* void setInteractive(){interactive = true;} // сетър
* bool getInteractive(){return interactive;} // гетър
* void location()const; // указва текущото кръстовище
* void change(std::string); // променя текущото кръстовище
* void neighbours(); // дава информация за съседите на текущото кръстовище
* void close(std::string); // добавя кръстовище към списъка на затворените
* void openCross(std::string); // премахва кръстовище от списъка на затворените
* void closed(); // дава списък от затворените кръстовища
* **Клас Path –** реализира път в града

**Private** член-данни:

* std::vector <std::string> crossroads; // кръстовищата, които са част от пътя, подредени в ред на минаването им за самия път
* double cost; // „тежест‘ на пътя
* int numCros; // брой кръстовища, участващи в пътя

**Public** член-функции:

* Path(); // конструктор
* void addCrossroad(std::string, double); // добавя кръстовище към пътя
* std::string getLastCrossroad()const; // взима последното добавено кръстовище
* void removeLast(std::string, double); // премахва последното добавено кръстовище
* bool operator<(const Path&)const; // оператор за сравнение на пътища
* void print(); // извежда пътя на екрана

1. **Реализация и уточнения**

Клас **Menu:**

* За да се видят командите, след като се въведе пътят към файла, се въвежда числото 8.

Клас **Path**:

* Oператорът < се изпозлва, за да може да се вкарват пътища в приоритетна опашка.

Структура **Street:**

* Структурата се състои от двойка кръстовища и се използва за задънените улици. Тя не е част от класа CityMap, за да може да се тества извън него, тъй като при търсене на задънени улици се връща динамичен масив от тип Street.

Клас **CityMap**:

* Използван е алгоритъм за търсене в дълбочина.
* Използван е алгоритъм за търсене на Ойлеров път
* Използван е алгоритъм на Kosaraju за силносвързани компоненти
* Използван е алгоритъм на Дийкстра за намиране на 3 най-кратки пътя

**Още уточнения:**

* За улеснение приемаме, че кръстовищата не съдържат интервали (това се подразбира и от примера 5Кьошета вместо 5 Кьошета)