Harjoitustyö: Karttaretki

COMP.CS.300 Tietorakenteet ja algoritmit 1, kevät 2021

Titta Kemppi, 282751

Johdanto

Tietorakenteet ja algoritmit -kurssin ensimmäisessä harjoitustyössä kirjoitetaan algoritmeja retkeilykarttasovellukselle. Kurssihenkilökunnan puolesta annettiin valmis sovellusrunko ja graafinen käyttöliittymä, joten opiskelijan tehtäväksi jäi täydentää datastructures-tiedoston tyhjät metodit ja muodostaa tietorakenteet paikkojen ja alueiden tietojen säilytykseen. Sovelluksella tarkastella kartalla olevia paikkoja ja alueita, niiden välisiä yhteyksiä ja niistä voidaan pyytää eri tietoja eri tavoin jäsenneltynä. Tehtävän tarkoitus on tehdä sovelluksen metodeista mahdollisimman (asymptoottisesti) tehokkaita. Lisäksi opiskelijoille annettiin testausohjelma, jolla pystyi kokeilemaan metodien suoritusaikoja eri määrillä dataa.

Toisessa harjoitustyössä työhön lisättiin reitit ja niiden muodostamiseen tarkoitetut hakualgoritmit.

Käytetyt tietorakenteet

Tietorakenteita suunnitellessa tuli kohtuullisen nopeasti siihen tulokseen, että paikoille ja alueille on syytä tehdä omat tietorakenteensa, sillä niiden välisten yhteyksien luomiselle ei ilmennyt tarvetta. Molemmissa käytin kuitenkin samaa tietorakennetta: unordered_map, jonka avaimena on paikan tai alueen id ja arvona struct, joka sisältää siitä tarpeelliset tiedot.

std::unordered_map<PlaceID, PlaceInfo> places_;

std::unordered_map<AreaID, AreaInfo> areas_;

```
struct PlaceInfo{
Name name_;
        Coord  coords_;
        PlaceType type_;

    };  struct
AreaInfo{
Name name_;
        std::vector<Coord> coords_;
        std::shared_ptr<AreaID> parent_ = nullptr;
std::vector<std::shared_ptr<AreaID>> children_ = {};

  };
```

Eli paikkaan liittyvästä structista löytyy tämän nimi, koordinaatit ja tyyppi. Alueen vastaavasta löytyy myös nimi ja koordinaatit, mutta tyypin sijasta löytyy viite alueen ”vanhempaan” eli sen alueen id:hen, johon kyseinen alue kuuluu. Viimeisenä vector viitteistä alueen alialueisiin, eli lapsinodeihin.

Näiden lisäksi erillisenä tietorakenteena on vector, jossa säilytetään paikkojen id:itä.

std::vector<PlaceID> placeIDs_;

Tällainen ylimääräisen tietorakenteen roikottaminen on yleensä turhaa, mutta pidin sen antamaa tehokkuutta metodeissa place_count ja all_places riittävänä etuna näin toimia. Valinta saattoi vähän kostautua metodissa remove_place.

Tietorakenteen valinta perustuu ajatukseen, että asioita on helppo jäsentää id:nsä perusteella, joten assosiatiivinen tietorakenne oli sopiva. Mapin valinta setin sijaan perustui vain omaan mieltymykseeni. Unordered_mapin valitsin, sillä se on normaalia map-rakennetta tehokkaampi, eikä tässä harjoitukessa ei ollut syytä pitää paikkoja tai alueita järjestyksessä. Structin valinta perustuu lähinnä siihen, että siihen on helppo säilöä useampaa kuin yhtä tietuetta. Alueiden structissa oleva viite vanhempaan osoittautui (itselleni) helpoimmaksi keinoksi toimia alialueiden kanssa.

Harjoitustyön toisessa vaiheessa käytin tietorakenteita:

```
struct Way{
        std::vector<Coord> coords_;
        Coord start_;
        Coord end_;
        Distance length_;

    };

    std::unordered_map<WayID, Way> ways_;

    struct Crossroad{
        Coord coord_;
        int color_;
        std::vector<std::shared_ptr<Way>> ways_connected_;
        std::shared_ptr<Crossroad> prior_crossroad;
        Distance distance_;
        std::shared_ptr<WayID> came_from_;
    };
```

```
std::unordered_map<Coord, Crossroad, CoordHash> crossroads_;
```

Homma alkoi pelkällä Way-rakenteella, mutta homman edetessä oli aika pian ilmeistä, että risteyksille tarvittiin omat rakenteensa. Muuten ratkaisu on aika samantyyppinen kuin ensimmäisessäkin vaiheessa. Crossroad-rakenteessa ways-connected viittaa reitteihin, jotka lähtevät risteyksestä ja prior viittaa risteykseen, josta tultiin tähän risteykseen. Distance kertoo, kuinka pitkä matka tähän risteykseen pääsyssä on kuljettu. Nämä ovat oleellisia parametreja lähinnä reittialgoritmeissa. Way-rakenteessa olevat start ja end ovat turhahkoja, mutta lisäsin ne, jotta koodi olisi paikotellen edes vähän luettavampaa.

Algoritmiratkaisut ja niiden tehokkuudet

Käydäänhän läpi jokainen metodi tehtävänannossa annetussa järjestyksessä ja avataan ratkaisuja.

Pakolliset metodit, ykkösvaihe:

place_count

Metodi palauttaa tiedon järjestelmässä olevien paikkojen määrästä. Toteutus palauttaa vectorin placeIDs koon, eli metodi on vakioaikainen O(1).

clear_all

Metodi tyhjentää tietorakenteet places ja placeIDs clear-metodilla, joka lineaariaikainen. Tämä tekee koko metodista lineaarisen θ(n).

all_places

Palauttaa vectorin, jossa on kaikkien paikkojen id:t. Toteutukseni palauttaa jo valmiina olevan placeID-vectorin. Näin ollen metodi on vakioaikainen O(1).

add_place

Lisää paikan tietorakenteeseen. Algoritmi tarkistaa find-algoritmilla, onko paikka jo olemassa ja jos ei, lisää se insert-metodilla paikan tiedot places-umappiin ja push_back-metodilla id:n placeIDsvectoriin. Find on O(n)≈θ(1), insert on ≈θ(1) ja push_back on O(1). Näin ollen metodi on O(n), mutta keskimäärin θ(1).

place_name_type

Palauttaa paikan nimen ja tyypin. Toteutus tarkistaa, onko haettua paikkaa tietorakenteessa findalgoritmilla ja palauttaa sen tiedot, jos löytyy. Tehokkuuden pullonkaulana toimii find, eli metodin tehokkuus on O(n) (≈θ(1)).

place_coord

Palauttaa paikan koordinaatit. Toteutus sama kuin place_name_typessa. Tehokkuus sama.

places_alphabetically

Palauttaa vectorissa järjestelmässä olevien paikkojen id:t nimien mukaisessa aakkosjärjestyksessä. Toteutuksessa paikkojen nimet ja id:t lisätään for-loopin avulla multimappiin, johon nimet menevät automaattisesti aakkosjärjestykseen.

```
std::multimap<Name, PlaceID> placeholder;
```

Mapista id:t rullataan toisella for-loopilla palautettavaan vectoriin. Metodin tehokkuus on forlooppien takia θ(n).

places_coord_order

Palauttaa paikkojen id:t koordinaattien mukaisessa järjestyksessä. Toteutuksessa mapista rullataan for loopilla id:t ja koordinaatit vectoriin, jonka sisällä on pareja (pair).

```
std::vector<std::pair<Coord, PlaceID>> placeholder;
```

Tietorakennevalintana olisi toki toiminut myös multimap, mutta tällä toteutuksella sain homman pelittämään nopeammin. Alkiot järjestetään sort-algoritmilla, johon on liitetty comp-funktio, joka järjestää alkiot euklidisen etäisyyden mukaiseen järjestykseen. Tämän jälkeen rullataan taas id:t lopulliseen vectoriin, joka palautetaan. Sort-algoritmin takia metodin tehokkuus on O(n log(n)).

find_places_name
Palauttaa vectorin paikoista, jotka ovat pyydetyn nimisiä. Toteutus käy for-loopilla places-mapissa olevat alkiot ja lisää halutun nimisten paikkojen id:t palautettavaan vectoriin. For-loopin takia on tehokkuus θ(n).

find_places_type
Palauttaa vectorin paikoista, joiden tyyppi on haluttu. Sama toteutus ja tehokkuus kuin metodilla find_places_name.

change_place_name
Vaihtaa halutun paikan nimen. Käytetään find-algoritmia id:n etsimiseen places-mapista ja vaihdetaan id:tä vastaavan paikan nimi uuteen. Tehokkuus findin takia on O(n) (≈θ(1)).

change_place_coord
Vaihtaa halutun paikan koordinaatit. Toteutus ja tehokkuus sama kuin change_place_namella.

add_area
Lisää areas-mappiin uuden arean. Tarkistaa find-algoritmilla josko paikka olisikin jo olemassa. Jos ei ole, käytetään insert-metodia id-avaimen ja nimi-koordinaatti–structin lisäämiseksi rakenteeseen. Tehokkuus on jo vanha tuttu O(n) (≈θ(1)).

area_coords
Palauttaa halutun alueen koordinaatit. Tuttuun tapaan tarkistetaan alueen olemassaolo findalgoritmilla. Jos löytyy, palautetaan koordinaatit. Tehokkuus O(n) (≈θ(1)).

all_areas
Palauttaa vectorin kaikista järjestelmän alueiden id:istä. Rullataan areas-map for-loopilla läpi ja push_backilla lisätään vectoriin. Tehokkuus θ(n) loopin takia.

add_subarea_to_area
Tekee halutusta alueesta toisen alueen alialueen. Toteutus tarkistaa (find-algoritmilla), että molemmat id:t löytyvät areas-mapista, ja että alueella ei vielä ole ylialuetta ja jos näin on, se tekee ylialueen id:hen pointerin ja lisää sen alialueen structin parentiksi. Jälleen find-algoritmi toimii rajoittavana tekijänä, ja asymptoottinen tehokkuus on O(n) (≈θ(1)).

subarea_in_areas
Metodi etsii kaikki alueet, joiden alialue kysytty alue on, ja palauttaa ne järjestyksessä olevassa vectorissa. Aluksi tarkistetaan jälleen, että alueen id löytyy järjestelmästä. Tämän jälkeen tehdään pointer alueen parent-alkioon ja while-loopilla etsitään kaikki puussa olevat ylialueet muuttaen pointerin arvoa aina tutkittavan alueen parentiksi, kunnes se osoittaa nullptr:iin. Find-algoritmi on O(n) (≈θ(1)) ja while-loop on O(n), mikä tarkoittaa, että pahimmillaan on tilanne O(n^2), mutta tyypillisesti metodi on lineearinen.

Vapaaehtoiset metodit:

all_subareas_in_area
Metodi palauttaa kaikki annettuun alueeseen kuuluvat alialueet. Ratkaisu hyödyntää rekursiota. Ohjelma lisää tutkittavan alueen alialueet palautettavaan vectoriin ja kutsuu metodia uudestaan kaikille alialueille. Tehokas tämä metodi ei ole kahdella for-loopillaan (θ(n)) ja insertoinnillaan (O(n)),

ja notaatio onkin O(n^2). Testaus kuitenkin osoitti, ettei metodi juuri koskaan ole läheskään noin tehoton, vaan toimii kohtuullisen lineaarisesti.

remove_place

Metodi poistaa paikan tietorakenteesta. Jälleen findilla tarkistetaan, että paikka on olemassa. Maptietorakenteesta poisto käy kauniisti erase-metodilla, mutta placeIDs-vectorista poistossa pitää yhdistää erase- ja remove -metodeja. Väittäisin metodin tehokkuuden olevan O(n), mutta täysin varma tuosta vectorista poistosta en ole.

common_area_of_subareas

Metodi palauttaa kahden alueen yhteisen yläalueen. Ratkaisu kutsuu metodia subarea_in_areas molemmille id:ille ja (ei kovin) kauniisti for-looppaa molempien vanhemmat läpi löytääkseen ensimmäisen yhteisen yläalueen. Tehokkuus on O(n^2), mutta tyypillisesti paljon nopeampi.

Pakolliset metodit, kakkosvaihe:

all_ways

Metodi palauttaa vectorissa kaikki ohjelman reitit. Tehokkuudeltaan se on lineaarinen, koska for-loopataan kaikkien alkioiden läpi.

add_way

Metodi lisää halutun väylän tietorakenteeseen. Lisäksi väylän päädyt lisätään risteyksiksi risteysrakenteeseen. Metodissa on iso kasa vakioaikaisia operaatioita, mutta perftest viittaa metodin lineaarisuuteen.

ways_from

Metodi kaikki halutusta koordinaatista lähtevät tiet ja risteykset, joihin niitä pitkin pääsee. Metodissa rullataan ways_-tietorakenne läpi ja etsitään tiet, joiden alku- tai loppupiste annettu koordinaatti on. Tie lisätään palautettavaan vectoriin ja tien loppu- tai alkupää lisätään risteykseksi riippuen siitä, kumpi lähtöpiste oli. Metodin tehokkuus on lineaarinen for-loopin takia.

get_way_coords

Palauttaa halutun tien koordinaatit. Tähän käytetään umapin find-algoritmia, jonka tehokkuus on O(n).

clear_ways

Metodi tyhjentää ways_ ja crossroads_ -tietorakenteet. Tyhjennys tapahtuu lineaarisella clear-metodilla.

route_any

Metodi palauttaa minkä tahansa reitin annetun lähtökoordinaatin ja loppukoordinaatin välillä. Ratkaisussa käytetään DFS-algoritmia, mikä tekee metodista lineaarisen. Reitin metsästämiseen käytetään takaisinpäin viittausta, eli jokaisesta risteyksestä löytyy viite risteykseen, josta sinne päästiin. Matkaa risteyksestä toiseen lasketaan algoritmin sisällä.

Vapaaehtoiset metodit, kakkosvaihe:

route_least_crossroads

Metodi palauttaa reitin kahden risteyksen välillä, jonka matkalla on vähiten risteyksiä. Toteutuksessa käytetään BFS-algoritmia ja siten se on lineaarinen.

route_shortest_distance

Metodi palauttaa reitin kahden risteyksen välillä, jonka matka on lyhin. Toteutuksessa käytetään Dijkstra-algoritmia ja siten se on lineaarinen.