Game of Fibres
readme

Sisällys

Käytetyt tietotyypit

Fibre

Kaikki struct-tietotyypit on siirretty structs.hh tiedoston alle, ja lisäksi joitain vertailija-tyyppien operaattoreja on määritelty structs.cpp tiedostossa. Tämä oli pakko tehdä, koska näillä operaattoreilla on kaksisuuntainen tyyppiriippuvuus. Esittelen tässä kaksi harjoitustyön toisen vaiheen päätietotyyppiä Fibre ja XPoint.

```
// Type for light fibres
struct Fibre
{
    // Fibre cost constructor
    Fibre(const Cost cost) : cost(cost) {}

    // Pointers to Xpoints inside _xPoints
    XPoint* start;
    XPoint* end;

    Cost cost;
};
```

Fibrella eli valokuidulla on osoitin sen molempiin päihin ja valokuidun hinta. Vaikka start-end on nimetty erikseen, valo voi kulkea kuiduissa molempiin suuntiin, kuten harjoitustyön ohjeessakin mainittiin. start ja end auttavat säilyttämään koordinaattien oikean järjestyksen, jota tulostuksessa vaadittiin. Kun uusi Fibre luodaan, niin start XPoint koordinaatti on aina pienempi kuin end XPoint koordinaatti. Tämän avulla, kun Fibreä haetaan koordinaattiparilla, tiedetään vaihtaa parin koordinaatit keskenään ennen varsinaista hakua, jos ensimmäinen on suurempi kuin toinen koordinaatti.

XPoint

```
// Type for Fibre intersections
struct XPoint
{
    // Constructor for a new Xpoint
    XPoint(const Coord &coord) :
        coord(coord), piBfs(nullptr), dBfs(INT_MAX),
        piDij(nullptr), dDij(INT_MAX), hereFrom(nullptr),
        color(White), usedFibre(nullptr)
    {}


    // XPoint is too expensive to copy so delete the implicit copy
    // constructor
    XPoint(const XPoint &copy) = delete;

    // Forbid modifications as _allFibres set is ordered based on this.
    const Coord coord;

    // Stores BFS-search results. pi is the shortest path to a certain
    // XPoint and d is the cost of it.
    const XPoint* piBfs;
    Cost dBfs;

    // Stores Dijkstra-algorithm results.
```

```
    const XPoint* piDij;
    Cost dDij;

    // Used in find_cycle to print the path.
    const XPoint* hereFrom;

    // See XColor (above)
    XColor color;

    // Datastructure for pointers to fibres inside _allFibres.
    std::set<const Fibre*, FibrePtrCompare> fibres;

    // Stores the fibre we used to path to this. Used in trimTreepPrims.
    const Fibre* usedFibre;
};

// Type for XPoint colors. Used in BFS/DFS/Djikstra.
enum XColor
{
    White, Gray, Black
};
```

XPoint on kahdesta tietotyypistä selvästi suurempi. Tyypin olion suuren koon vuoksi, ja koska yksi XPoint attribuutti on kokonainen std::set, jonka kopioiminen oliota käsitellessä olisi täysin turhaa työtä, XPoint implisiittinen kopiorakentaja on poistettu. XPoint saa käsitellä siis vain osoittimena olioon, joka sijaitsee tietorakenteen sisällä. XPoint-rakentaja ottaa parametrikseen kyseisen haarautumispisteen koordinaatin. Koordinaatti tallennetaan vakiona (const), koska kuten kohta tulee ilmi, sitä ei saa muuttaa, eikä sitä tulla koskaan muuttamaankaan. XPoint koordinaatti on sama olion tuhoutumiseen asti.

Muita XPoint talletettavia tyyppejä ovat Dijkstra- ja BFS-haun tulokset (erikseen). find_cycle käytettävä hereFrom ptr, jonka avulla syklin reitti saadaan selville. Lisäksi XPoint tallentaa oman värinsä, joka on tyyppiä XColor, joka näkyy alhaalla. Nämä värit ovat ekvivalentteja Dijkstra-/BFS-/DFS-pseudokoodeissa käytettäviin väreihin. Lisäksi jokainen XPoint tallettaa siitä lähtevät ja siihen tulevat valokuidut std::set tietorakenteeseen. Tallettavat alkiot ovat osoittimia Datastructures-luokan tietorakenteen olioihin.

Kuten nähdään, set alustetaan myös toisella template-argumentilla, joka on structi FibrePtrCompare. Tämä vertailija on esitetty Tämä mahdollistaa Fibre-osoittimien vertailun niiden coordinaateilla, sekä setista hakemisen koordinaattiparin std::pair<Coord, Coord> avulla (std::set heterogeneous lookup). Tämän vuoksi koordinaattipareja etsiessä ei tarvitse luoda uutta Fibre-oliota sen löytääkseen.

Päätietorakenteet

Fibre-tietotyypeille

```
// Set for Fibres supporting heterogenenous find with Coord.
std::set<Fibre, FibreCompare> _allFibres;
```

Fibre-olio luodaan addFibre-private metodissa, ja se talletetaan Fibren koordinaattiparin perusteella järjestettyyn settiin. Tämän ansiosta kaiken muistinhallinnan hoitaa std::set-tietorakenne, jolloin voimme vapaasti käyttää raaka-osoittimia (Fibre*) koko luokan laajuisesti. Kuten kommentistakin tulee ilmi, myös tämä setti tukee heterogeenistä hakua koordinaattiparien kanssa. FibreCompare vertailija löytyy structs.hh tiedostosta.

Valinta std::set ja std::unordered_map/unordered_set välillä on selkeä. Kaikki Fibret täytyy tulostaa add_fibres metodissa koordinaattiensa mukaisessa järjestyksessä, mutta tähän tietorakenteeseen tehdään hakuja harvoin. Lisäksi trim_fibre_network-metodissa std::set järjestyksestä hyödytään erittäin paljon.

Fibres-tietorakenteen valintaan vaikuttavat metodit. Tehokkuuksissa on mainittu vain osat, joihin tietorakenne vaikuttaa, ei koko metodin tehokkuutta.

1. std::set kanssa
    a. add_fibre O (log n)
    b. remove_fibre O(log n)
    c. all_fibres O(n)
    d. trim_fibre_network (ei MST-puussa olevien alkioiden poisto) O(n + m) !!
2. std::unordered_map/set kanssa
    a. add_fibre O(1)
    b. remove_fibre O(1)
    c. all_fibres O(n log n)
    d. trim_fibre_network (poisto) $O(n^2)$

Kuten nähdään ilman std::set trim_fibre_network tulisi erittäin epätehokas, koska meidän tarvitsee tehdä $O(n^2)$ haku etsimään ne valokuidut, jotka eivät ole pienimmässä virittävässä puussa (MST). std::set kanssa voimme käyttää std::set_difference _allFibres ja inSpanning välillä.

XPoint-tietotyypeille

```
// Datastructure for fibre intersections
std::unordered_map<Coord, XPoint, CoordHash> _xPoints;
```

XPoint-olio luodaan myös samassa metodissa kuin Fibre-olio. XPoint olio talletetaan kuitenkin unordered_map, koska siihen tehdään jatkuvasti hakuja. Täten unordered_mapin amortisoidusti vakioaikaisesta hausta hyödytään erittäin paljon. std::map käyttö olisi perusteltua vain all_xpoints perusteella, mutta silloin kaikissa muissa metodeissa jouduttaisiin suorittamaan O(log n) hakuja.

XPoint-tietorakenteen valintaan vaikuttavat metodit. Listataan taas vain osatehokkuudet.

1. std::unordered_map kanssa
    a. all_xpoints O(n log n)
    b. add_fibre O(1)
    c. remove_fibre O(1)

   d. get_fibres_from O(1)
   e. route_any O(1)
   f. route_least_xpoints O(1)
   g. route_fastest O(1)
   h. route_fibre_cycle O(1)
2. std::map kanssa
   a. O(n)
   b. kaikki muut O(log n)…

Kuten nähdään, päätös tietorakenteen käytöstä on harvinaisen selkeä. Sinänsä unordered_map käyttö unordered_set sijaan on perusteltua vain, koska vasta C++20 tukee heterogeenisiä hakuja unordered_settiin. Siis C++20 olisi voitu tehdä

```
// Datastructure for fibre intersections (in C++20)
std::unordered_set< XPoint, XPHash, XPComp> _xPoints;,
```

jolloin XPComp-vertailija olisi voitu määritellä tukemaan hakuja Coord-tyypillä. C++17, jolla harjoitustyö tehtiin, ei tue heterogeenisiä hakuja unordered_set kanssa. Hakeminen C++17 unordered_setistä koordinaatilla vaatisi, että koordinaatilla tehtäisiin uusi XPoint-olio, jolla haku suoritetaan. Kuten XPoint-tietotyyppiä käsitellessä tuli ilmi, että sen kopiominen on raskasta, on sen rakentaminen myös raskasta. Tästä syystä, C++17 tälläisen unordered_set käyttö olisi erittäin epätehokasta.

Yleisesti

Assosiatiivisten tietorakenteiden valitseminen olioiden säilytykseen, oli kyseessä järjestetty tai järjestämätön, motivoi myös erittäin paljon osoittimien (referenssien) säilyminen kaikissa tapauksissa. Jos käyttäisimme esimerkiksi järjestettyä std::vector Fibre-olioiden säilytykseen, niin XPoint fibres tietorakenteessa osoittimien säilyttäminen olisi erittäin vaarallista. Erityisesti kun std::vector joutuisia kasvattamaan kokoaan (allocate), niin aivan kaikki osoittimet jokaisessa XPoint fibres-setissä täytyisi määrittää uudestaan, mikä olisi erittäin kallista.

Sarjojen käyttö Fibres-olioiden säilyttämiseen edellyttäisi, että tallettaisimme näistä olioista kopioita XPoint fibres-tietorakenteessa. Tämä ei ole mahdoton ratkaisu, koska toisin kuin XPoint, Fibre on melko halpa kopioida (2 osoitinta ja kokonaislukumuuttuja = 24 tavua). Tämä on kuitenkin on lähtökohtaisesti huonompi ratkaisu, std::set saatavan hyödyn vuoksi, joka käsiteltiin kappaleessa ”Fibre-tietotyypeille”.

Oheistietorakenteet

Tässä kappaleessa käydään läpi muita harjoitustyössä käytettyjä tietorakenteita. Näitä ovat esimerkiksi tiettyjen metodien lokaalit tietorakenteet tai em. tietotyyppien sisällä säilytettävät tietorakenteet. Aivan jokaista käytettyä tietorakennetta tässä ei käydä läpi, vaan niitä tiettyjä, jotka vaikuttivat metodin tai algoritmin toimintaan merkittävästi, ja joiden valitsemiseen annettiin mahdollisuus vaikuttaa.

CoordPair

```
std::pair<Coord&, Coord&>;
```

Tätä tietorakennetta hyödynnnetään etsiessä Fibreä _allFibres-setistä. Tietorakennetta käyttävät fibresFind, add_fibre ja remove_fibre, mutta sen olennaisin implementaatio on fibresFind-metodissa. Koordinaattipari järjestetään aina ennen hakua, niin että ensimmäinen koordinaatti on pienempi kuin

toinen. fibresFind hyödyntää tässä järjestämisessä std::pair valmista <-vertailijaa, sekä std::swap. Koska CoordPair otetaan fibresFind-metodissa viitteenä, koordinaattien swap välittyy myös sitä kutsuneelle funktiolle.

XPoint->fibres

```
std::set<const Fibre*, FibrePtrCompare> fibres;
```

Samasta syystä kuin Fibres-olioille valittiin std::set, niin XPointista lähtevät ja tulevat valonsäteet tallennettiin myös settiin. Tähän tietorakenteeseen suoritetaan hakuja vain ja ainoastaan remove_fibre-metodissa. Lisäksi hyödymme ylläpidetystä järjestyksestä get_fibres_from-metodissa.

Myönnettäköön, että koska lähtökohtaisesti tämän setin koko on harvoin 10 suurempi, jolloin std::vector toteutus saattaisi olla tehokkaampi. Tämä tehokkuus tulisi, siitä että std::vector tarjoaa paremman muistin lokaalisuuden, jolloin se pystyisi hyödyntämään välimuistia paremmin. Vektorin järjestyksen ylläpitämiseen voitaisiin käyttää puolitushakua.

Vaikka molemmat tietorakenteet voidaan iteroida läpi asymptoottisesti samassa ajassa O(n), niin std::vector kykenee tekemään sen absoluuttisesti nopeammin, juuri tämän muistin lokaalisuuden vuoksi. std::set punamustanpuun iteroimisessa on suuremmat vakiokertoimet. Tästä hyötyisi erityisesti DFS-, BFS- ja Dijkstra-algoritmit, jotka iteroivat tätä fibres-tietorakennetta jokaisella graafin solmulla.

Sanottakoon, että jos tämän vektorin koko pääsee koskaan kasvamaan merkittävän kokoiseksi, niin insert-operaatiot tulevat huomattavan paljon kalliimmaksi, kuin std::set lisääminen. Tämä siksi, että jos lisättävä alkio sattuu menemään esimerkiksi ensimmäiseksi, niin kaikkia std::vector alkiota tarvitsee liikuttaa eteenpäin, jolloin teimme logaritmisen puolitushaun lisäksi lineaarisen operaation.

+ ja – ovat vakiokertoimisesti verrattuna

1. std::set kanssa
    a. add_fibre O(log n) -järj. std::vector
    b. remove_fibre O(log n) -järj. std::vector
    c. get_fibres_from O(n) +järj. std::vector
    d. iteroiminen O(n) +järj. std::vector
2. järjestetty std::vector (m on etäisyys sijoitetun alkion ja lopun välillä)
    a. add_fibre O(log n + m)
    b. remove_fibre O(log n + m)
    c. get_fibres_from O(n) -std::set
    d. iteroiminen O(n) -std::set
3. järjestämätön std::vector
    a. add_fibre O(1)
    b. remove_fibre O(1)
    c. get_fibres_from O(n log n)

Tässä valinta ei ole niin selkeä, kuin muissa tapauksissa. Mutta std::set tarjoaa kuitenkin paljon joustavamman ja helpomman toteutuksen. Esimerkiksi haut järjestettyyn vektoriin koordinaattiparin avulla tarvitsisi tehdä lambdan kanssa, sekä sijoittaminen std::lower_bound kanssa ei ole läheskään niin suoraviivaista kuin std::set::insert. Sijoittaminen ja poistaminen järjestetystä std::vector on myös keskimäärin hieman hitaampaa.

spanningFibres

std::set<Fibre, FibreCompare> spanningFibres;

trim_fibre_network alustettu std::set on lähes ainoa valinta kyseisen metodin nopean toiminnan mahdollistamiseksi. MSTTreePrims Primin algoritmilla toteutettu pienimmän virittävän puun hakualgoritmi, sijoittaa Fibre-olioita sen parametrina saamaan spanningFibres settiin. MSTTreePrims algoritmia suoritetaan, kunnes jokainen graafin solmun väri on musta, eli kunnes jokainen solmu on tässä puussa.

Kuten voidaan huomata, tämä setti käsittelee olioiden kopioita. Tämä ei ole osoitin-toteukseen verrattuna juurikaan kalliimpi, koska kuten aikaisemmin tuli ilmi, Fibre-olio on melko halpa kopioida. Toteutus täytyi tehdä kopioilla, koska osoitin-toteutuksessa std::set_difference käytetty std::back_insert_iterator tarvitsisi toteuttaa itse, koska set_difference suorittaisi sijoitusta Fibre = Fibre*.

On tärkeää huomata, että spanningFibres ei sisällä yksittäisen puun valokuituja, vaan ohjelman metsän kaikki valokuidut, jotka ovat metsän puiden pienimmistä virittävistä puista. Kaikille metsän puille suoritetaan pienimmän virittävän puun haku erikseen. Koska spanningFibres on setti, ja se säilyttää aina järjestyksensä, niin tämä mahdollistaa std::set_difference käytön, josta puhuttiin Fibre-olion päätietorakennetta käydessä läpi.

Tähän tapaukseen järjestetyn vektorin käyttö olisi erittäin epätehokasta. spanningFibres voi sisältää satoja tuhansia valokuituja, jolloin puolitushaku + insert olisi erittäin epätehokas.