KD-TREE

Обща информация и функционалност:

Проектът се фокусира върху структурата KD-TREE и основни нейни функционалности. Дървото е съставено от *d-мерни* точки (произволна размерност) и се конструира по даден списък от такива (конструирането може да стане по няколко стратегии). Поддържат се функции за ***добавяне на точка*** и ***премахване на точка*** *от дървото*, ***преобразуване на дървото в списък*** от точки (изграждащите го).
Има функция за ***търсене в интервал*** (по зададени ограничения за всяка една от осите, връща списък от точките, в хиперинтервала), която може да бъде приложена върху конкретно дърво от точки или върху списък от точки.
Реализирана е функция за ***най-близък съсед*** на произволна точка измежду дадените спрямо произволна метрика за разстояние. Последната функция има по-оптимизирани вариации за разстояния на ***Minkowski (в частност евклидово и манхатъново), мин*** и ***макс*** разстояние***.***
Дърветата могат да бъдат ***сериализирани в JSON формат*** и ***обратно извлечени*** от него.

За реализацията:

Дървото се представя с алгебричен тип, точките са списъци (*има проверка всички да са n-мерни, от еднаква размерност*). (Поначало възнамерявах точките също да бъдат абстрахирани, но това отпадна, защото прекомерно затормозяваше решението, а същевременно не носеше нещо полезно извън абстракцията.)

- *Конструиране:* дървото се строи по даден списък от точки, това прави функцията build. Тя, от своя страна, се обръща към construct функция, която приема и допълнителни аргументи *seed1, seed2, seed3.* Тяхната идея е да конкретизират коя стратегия за строене ще се ползва:
  - За избиране на ос:
    1. гледа се по кое направление разстоянието между две точки е максимално (тоест къде обвивката на точките е най-широка), това се избира и за ос на разделяне; *seed1 = -1*
    2. Алтерниращa стратегия (seed1 <- [0.. d-1]) - коренът разбива точките по направление 0, децата му - по направление 1 и т.н.;
  - За избиране на стойност:
    1. Избира медианата на точките (така броят точки намалява двойно)
    2. Избира средата на обхващащия ги интервал по съответното направление (така пространството намалява двойно)
  - За избиране на leafSize:
    1. По един връх в листо;
    2. sqrt (n/d) - rule of thumb? (някъде пишеше, че sqrt е добра горна граница);
    3. Избор спрямо размерността на точките: между 10 и 40 за d<5, между 5 и 15 иначе (самите стойностите са от интернет/предложени от бота);

- *Точки в ограничен интервал:* Минава се през дървото, като, ако в някой връх исканият интервал не се пресича с този на точките в поддървото (който е precompute-нат в дървото), се връщаме нагоре, вместо да влизаме в поддървото. Операцията има сложност грубо O(mlogm.d) (доколкото може да остане такава във функционален стил), където m е броят точки в интервала;

- *Най-близка точка до дадена*: За произволна метрика е ясно, че еднозначен бърз алгоритъм няма (или поне аз не виждам), разбира се, ако не говорим за апроксимиращи такива. Общият случай е решен чрез линейно минаване през точките, тривиалното. По -интересен е случаят с някои конкретни метрики за разстояние, по-конкретно са покрити *Minkowski distance*, *min distance*, *max distance*. Идеята е общо взето следната: функцията се спуска да листото/областта, която потенциално трябва да съдържа

точката, намира се най-близката ѝ точка там. Пр връщането (на всяко ниво от пътя) в другото поддърво се влиза само ако е възможно там да има по-близка точка, проверката става чрез намиране на разстоянието между точката и d-мерната обвивка на точките в съответното друго поддърво и сравняване с най-доброто разстояние до момента.

- *Сериализиране:* Направен е типов клас ToJSON (макар че май има и вграден такъв), на който типът на дървото е инстанция. Реализиране е функция, която обръща структурата в JSON файл. Тук особеността е, че в JSON формата са пропуснати обвивките на точките от поддърветата, тяхно закодиране би било излишен разход на памет, понеже те могат да бъдат допълнително изчислени от самото дърво (типът CompressedTree представя точно това дърво).
  По-трудно е *десериалвзирането*, проблемът е, че на теория трябва да се прави/използва наготово parser от json в дърво. Малко по-безболезнен начин е да използваме факта, че show/read форматът (на compressedTree) по подразбиране е доста подобен на json формата. С помощта на функция минаваме от json в "readable" формат, който read превръща в "CompressedTree". Оттам декомпресиране (изчисляваме обвивките/интервалите на точките поддърветата) и обратно сме в KDTree тип.

- *Допълнителни функционалности:* Операциите по добавяне и махане на точка в дървото са придружени с актуализация на обвивките

За самите функции, по модули:

1. Hmath

Модулът реализира основни помощни математически функции и алгоритми, които се ползват в другите функции. Ето и някои от самите тях:
- **accumulate**
- **QuickSort**
- **dimension :: [a] -> Int;** - колко мерна е точката
- **dimensionality :: [[a]] -> Int;** - в колко мерно е пространството са точките (при коректно зададени равни измерения на всичките)
- **distanceToHull :: ([Double] -> [Double] -> Double) -> [Double] -> [(Double, Double)] -> Double;** по метрика, точка и обвивка/хиперкуб намира разстоянието от точката до него
- **hull :: [[Double]] -> [(Double,Double)];** - намира обвивката (като хиперкуб/) на точки

2. Tree
Грижи се за типовете, взаимодействието им и за сериализирането. Функциите вече бяха споменати по-горе:
- **toJSON :: KDTree -> String**
- **decompress :: CompressedTree -> KDTree**
- **jsonToReadable :: String -> String**, с помощни функции **clearJSON1**, **clearJSON2**
- **serialise, deserialise**

3. Construction
Грижи се за конструкция на дървото и валидацията на размерността на данните.
- **validateDimensions** - проверява дали всики точки имат еднаква размерност, ако не, хвърля грешка;
- **Amplitude** - по направление връща най-дългото разстояние между две точки по него;

- **widestDimension** - намира именно това направление;
- **nextSeed** - гарантира алтерниране при избора на осите (от стратегия 2);
- **setVal, setAxis, leafsize** - спрямо seed избират конкретни стойности за value, axis (оста на разделяне), максимален брой точки в листо;
- **build, construct** - вече споменахме;

4. Queries

Реализира всички допълнителни функционалност около дървото, главно такива от по-високо ниво.

- **insert, remove** (ко съществува) - на точка от дърво
- **treeToList** - връща списък от точките от дървото (в dfs ред)
- **kdSort** - сортира списък от точки по стратегиите, които дървото ползва
- **nearestNeighbour** - най-близка точка до дадена в общия случай
- **nearestCustom** - най-близка точка до дадена в специфичните случаи, уточнени горе, възползва се от това, че сме в Евклидово пространство
- **nnMinkowski, nnMin, nnMax, nnManhattan, nnEuclidean** са конкретните функции
- **filterInside** - по дадено дърво и списък от ограничения (constraints), намира точките от дървото в ограниченията/“d-интервала”;
- **pointsInInterval** - прави същото като горе, само че множеството точки е зададено директно като списък, а не дърво (отдолу се свежда до дърво);

В кода над функциите е разяснено тяхното действие, също най-отдолу в Queries.hs са оставени тестове на различните функции и резултатите им.

Tree-ctor1.json и Tree-ctor2.json показват дървета, направени върху еднакви точки, но различни стратегии за конструиране. treeCopy.json е дървото, генерирано при четене на дърво от json и записването/копирането му в друг (с цел да се илюстрира десериализирането).