Michał Fudała

Wyszukiwanie geometryczne - przeszukiwanie obszarów ortogonalnych

Drzewa obszarów

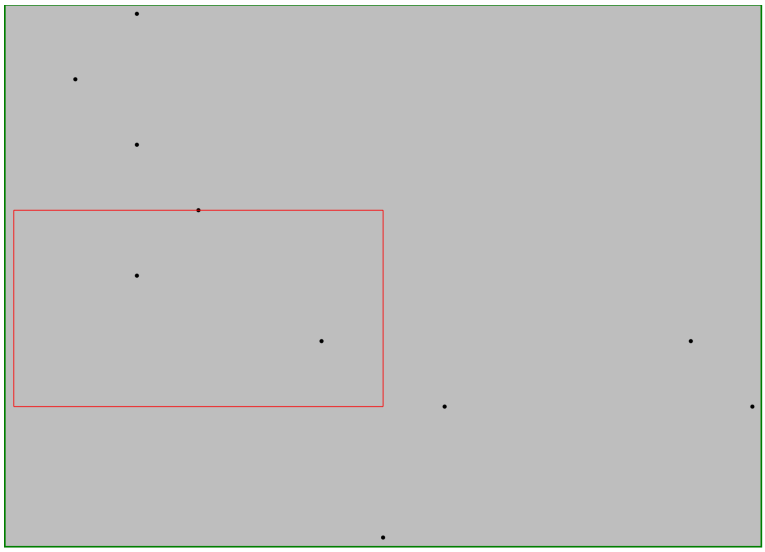
1. Abstrakt

Projekt został stworzony na potrzeby zajęć z przedmiotu Geometrii Obliczeniowej na uczelni Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, AGH.
Celem projektu była implementacja struktury Drzewa obszarów służącej do optymalnego przeszukiwania obszarów ortogonalnych.

2. Definicja problemu

Dane: zbiór punktów **P** na płaszczyźnie.

Problem: dla zadanych x_1, x_2, y_1, y_2 znaleźć punkty q ze zbioru **P** takie, że $x_1 \leq qx \leq x_2$,$y_1 \leq qy \leq y_2$



3. Opis algorytmu

n - liczba punktów wejściowych
k - liczba punktów wyjściowych

Projekt zawiera 2 implementacja algorytmu przeszukiwania:

- implementacja drzewa obszarów 2D (budowanie - O(nlogn), wyszukiwanie - $O(\log^2 n+k)$)
- przeszukiwanie liniowe (O(n)) - przejście po wszystkich punktach - w celach automatycznych testów porównawczych)

Drzewo 1D

Budowa drzewa

metoda `NodePtr buildTree(const Vec &elements);` :

1) Jeżeli punkty nie były posortowane, posortuj po kluczu (y)
2) wywołaj `buildSubTree`

metoda `NodePtr buildSubTree(const Vec &els, size_t beg, size_t end);` :

1) Zbuduj węzęł `N` drzewa którego wartością jest mediana podzbioru punktów (od `beg` do `end`)
2) Rekursywnie wywołaj algorytm dla punktów mniejszych bądź równych od mediany i przypisz do lewego dziecka nowo stworzonego węzła `N`
3) Rekursywnie wywołaj algorytm dla punktów większych od mediany i przypisz do prawego dziecka nowo stworzonego węzła `N`
4) Zwróć `N`

Wyszukiwanie w drzewie

metoda `NodePtr findVSplit(NodePtr tree, C from, C to) const;` :

1) znajdź węzeł dzielący

metoda `collectAll(NodePtr root, C from, C to, VSplitSubtree subtreeType, Set &collector) const;` :

1) subtreeType określa, która strona drzewa `root` czyli węzła dzielącego ma być rozważana
2) w zależności od parametru `subtreeType(LEFT/RIGHT)`:
    1) jeżeli klucz rozważanego node'a jest większy/mniejszy od `x_min/x_max`
        1) idź w lewo/prawo drzewa
        2) zapisz wszystkie liście prawego/lewego poddrzewa do zbioru `collector`
    2) w przeciwnym wypadku
        1) `idź w prawo/lewo drzewa`

metoda `void search(C keyFrom, C keyTo, set<T> &collector) const;` :

1) skorzystaj z `findVSplit` aby znaleźć węzeł dzielący `N`
2) Wywołaj `collectAll` z węzłem `N j`ako root zbierającej punkty dla prawych poddrzew
3) Wywołaj `collectAll` z węzłem `N j`ako root zbierającej punkty dla lewych poddrzew

Drzewo 2D

Różnice przy budowie drzewa

`NodePtr buildTree(const vector<T> &elements); :`

1) Punkty wejściowe są konwertowane na punkty z koordynatami `ComparableTuple: toUniquePoint(keyF(data)), data);`
2) Tworzymy strukturę posortowanych punktów `SortedPoints` (po y i po x)

```
NodePtr buildSubTree(const SortedPoints<T, C> &preSorted, size_t xBeg,
size_t xEnd); :
```

1) Kluczem drzewa 2D jest koordynat x punktu
2) Przy pobieraniu punktów zawsze korzystamy z listy `preSorted`
3) Każdy węzeł drzewa zawiera wskaźnik do drzewa 1D, które jest utworzone z punktów powstałych w wyniku wywołania metody `getAllY(xBeg, xEnd)`

```
auto associatedStructureData = preSorted.getAllY(xBeg, xEnd);
auto associatedStructure =
            make_range_search_tree_shared_ptr<…>(
                    associatedStructureData,
                    getY<T, C>,
                    true
            );
```

Różnice przy wyszukiwaniu w drzewie

`set<T> search(const Rect<> &r) const; :`

1) Argumenty wejściowe x1, x2, y1, y2 są konwertowane tak by uwzględnić strukturę `ComparableTuple` (i pozwalać na wyszukiwanie punktów z takimi samym koordynatami x lub y)

```
void collectAll(NodePtr tree, const Rect<C> &area, VSplitSubtree
subtreeType, set<TreePointWithData> &collector) const; :
```

1) Jeżeli spełniony jest warunek - klucz rozważanego węzłem jest większy/mniejszy od `x_min/ x_max` to zapisujemy wszystkie punkty z struktury powiązanej (drzewa 1D związanego z rozważanym węzłem)

4. Zalety implementacji

Generyczność
Drzewo jest strukturą generyczną i pozwala na przetrzymywanie danych wraz z punktami.
W konstruktorze drzewa możemy przekazać funkcję, która wydobędzie klucz z danej.

Przykład pokazujący zalety takiego podejścia:

Mając zbiór miast (dane w drzewie - **nazwa** miasta, klucz drzewa - współrzędne geograficzne), możemy znaleźć wszystkie **nazwy** miasta mieszczące się w pewnym prostokącie zadanym przez parametry x1, x2, y1, y2. Widać, że dużą zaletą trzymania danych w drzewie jest to, że na wyjściu wraz z pasującymi punktami dostaniemy również **nazwy** miast.

Wydajność
Program został napisany w języku C++ co daje znaczącą przewagę wydajnościową nad innymi językami. Dodatkowo program był pisany ze zwróceniem uwagi na używanie małej ilości operacji kopiowania struktur.

Testy
Program zawiera dużą liczbę testów jednostkowych (wraz z całkowicie generowanymi losowo danymi i parametrami algorytmu - porównanie z prostym liniowym wyszukiwaniem)

5. Użyte struktury danych

Node
Węzeł drzewa binarnego:
```
struct Node {
  Node *left;
  Node *right;
  C key;
  bool isLeaf;
  T value;
}
```
Struktura generyczna, przechowuje dane:
`isLeaf` - czy węzeł jest liściem
`value` - wartość skojarzona z tym węzłem (tylko dla liści) (np nazwa miasta na mapie)
`key` - klucz `(double lub ComparableTuple)` (np współrzędne miasta na mapie)
`left, right` - lewe i prawe dziecko węzła

ComparableTuple
Struktura rozwiązująca problem przetrzymywania wielu punktów o takich samych współrzędnych x lub y w drzewie.
```
class ComparableTuple {
  double c1;
  double c2;
```

```
}
```
Oprócz samych danych implementuje operator porównania

Przejście z punktu o współrzędnych `double` na współrzędne o typie `ComparableTuple` (w kodzie funkcja `Point<ComparableTuple> toUniquePoint(Point<double> p);`):

$$\hat{p} = ((p_x|p_y), (p_y|p_x))$$

Implementacja operatorów porównania (w kodzie wnętrze klasy `ComparableTuple`):

$$(a|b) < (\acute{a}|\acute{b}) \Leftrightarrow a < \acute{a} \vee (a = \acute{a} \wedge b < \acute{b})$$

Przejście z argumentów problemu w typie `double` na typy `ComparableTuple` punktu jest realizowane w następujący sposób (w kodzie metoda `set<T> search(const Rect<> &r) const;`)

$$\hat{R} = [(x|-\infty), (x'|+\infty)] \times [(y|-\infty), (y'|+\infty)]$$

Point
Generyczna struktura opisująca punkt (zawiera współrzędne punktu)

Rect
Prosta struktura opisująca argumenty x1,x2,y1,y2 problemu

SortedPoints
Struktura przechowująca dwie listy punktów posortowanych po współrzędnej y oraz posortowanych po współrzędnej x.
Punkty są sortowane tylko raz przed budową całego drzewa. Jest wykorzystywana tylko przy budowie drzewa. Celem jej istnienia jest możliwość wykonywania operacji:
- `getX(size_t i)` - pobiera punkt o indeksie `i` z list posortowanej po współrzędnej x
- `getAllY(size_t fromX, size_t toX)` - pobiera wszystkie punkty o indeksach od `fromX` do `toX` z listy punktów posortowanych po x, zwraca punkty posortowane po y korzystając z drugiej przechowywanej listy oraz `reverseYIndex`.

RangeSearchTree
Opisana wcześniej struktura drzewa obszarów 1D

RangeSearchTree2D
Opisana wcześniej struktura drzewa obszarów 2D

6. Testy jednostkowe

Projekt zawiera bardzo dużą ilość testów jednostkowych
- ręcznie zdefiniowane przypadki testowe, obejmujące przypadki graniczne (jak punkty o takich samych współrzędnych x, y)
- punkty generowane losowo o charakterystycznej strukturze z ręcznie zadanymi argumentami problemu x1, x2, y1, y2 i oczekiwanymi punktami wyjściowymi
  - na kwadracie
  - wewnątrz kwadratu
  - na kole
  - wewnątrz koła
  - kilka losowych klastrów danych oddalonych od siebie
- losowo generowane punkty i argumenty problemu (test przez porównanie wyników z prostą implementacja sprawdzającą wszystkie punkty)
- test przez graficzne wprowadzenie zbioru punktów (tekstowo)
- testy czy drzewo jest zbalansowane (dla zminimalizowania błędów na początku implementacji)

Testy znajdują się w katalogu **test**:
- **testRangeSearchTree.cpp -** testy wyszukiwania w drzewie obszarów 1D
- **testRangeSearchTree2D.cpp -** testy wyszukiwania w drzewie obszarów 2D
- **testSortedPoints.cpp** - testy struktury punktów posortowanych po współrzędnej x oraz y

Generatory punktów losowych znajdują się w pliku **Random.h**, opis funkcji

- **randomRect** - generowanie prostokąta wewnątrz innego prostokąta
- **randomPointsInRect** - generowanie punktów wewnątrz prostokąta
- **randomPointsInCircle** - generowanie punktów na okręgu
- **randomPointsInCircle** - generowanie punktów wewnątrz okręgu
- **randomPoints** - generowanie punktów wewnątrz prostokąta

7. Użyte biblioteki, środowisko

Projekt został napisany w języku C++ w standardzie C++14. Projekt można zbudować za pomocą CMake.

Użyte biblioteki:
- gogui - biblioteka do wizualizacji
- catch - biblioteka do testów jednostkowych
- cxxopts - biblioteka do parsowania argumentów wejścia programu (terminal)
- Boost - biblioteka dostarcza funkcje pomocnicze ułatwiające pisanie w C++

Wymagane biblioteki zainstalowane w systemie (przez proces linkowania):
- Boost

Inne wymagania:

- kompilator obsługujący standard C++14

8. Budowanie i uruchomianie

```
cmake .
make
```

Wygenerowane pliki:
- **test_range_tree** - program uruchamiający testy jednostkowe
- **range_tree** - program rozwiązujący problem dla zadanych punktów (czyta ze standardowego wejścia)