KD drzewa

Szymon Laszczyński

Wrocław, 17 stycznia 2011

Streszczenie

Może się wydawać na pierwszy rzut oka, że bazy danych mają niewiele wspólnego z geometrią. Jednakże na wiele pytań (zapytań) dotyczących baz danych może być interpretowanych w ujęciu geometrycznym. Aby to osiągnąć przekształcamy rekordy bazy danych w punkt przestrzeni k-wymiarowej, natomiast zapytania do bazy w zbiory punktów w takich przestrzeniach. Problemy w takim ujęciu będziemy nazywać przeszukiwaniem obszarów (range searching problem). Jedną ze struktur służących do rozwiązywania takich problemów są kd-drzewa, które to omówimy w poniższym opracowaniu.

1 Ogólny opis

Omówienie idei kd-drzew przedstawimy na przykładzie 2-wymiarowego przeszukiwania obszarów. Ta struktura jest szczególnie przydatna w przypadku 2-wymiarowym i pozwala nam zająć jedynie $O(n)$ pamięci, jednakże będzie nas to kosztować zwiększeniem czasu zapytań w stosunku do innych struktur dla tego problemu (np. drzewa przedziałowe).

Obserwacja 1. *Punkty należące do P możemy zastąpić punktami innej przestrzeni. Współrzędne tych punktów zamienimy na współrzędne w tzw. przestrzeni liczb złożonych w postaci pary liczb. Liczbę złożoną dwóch liczb a i b będziemy oznaczać przez $(a|b)$. Jedyne czego potrzebujemy to porządek liniowy na przestrzeni liczb złożonych, który będzie odpowiadał porządkowi na pierwotnych współrzędnych. Zdefiniujmy go zatem jako porządek leksykograficzny na liczbach złożonych, a więc dla dwóch liczb złożonych $(a|b)$ i $(a'|b')$ mamy:*

$$(a|b) < (a'|b') \iff a < a' \text{ lub } (a = a' \text{ i } b < b') \tag{1}$$

Zatem każdy punkt $p = (p_x, p_y)$ z P możemy zamienić na $\hat{p} = ((p_x|p_y), (p_y|p_x))$. W ten sposób otrzymujemy nowy zbiór punktów $\hat{P}$ z współrzędnymi w przestrzeni liczb złożonych. Łatwo zauważyć, że $\hat{P}$ spełnia ograniczenia których wymagamy, tzn. każde dwa punkty mają różne współrzędne x i y. Oczywiście musimy także dostosować nasze zapytanie (prostokątny przedział) by odpowiadał nowym współrzędnym. Dla przedziału $R = [x : x']x[y : y']$ przedział przekształcony w nowych współrzędnych będzie wyglądał następująco:

$$\hat{R} = [(x| - \infty) : (x'| + \infty)] \times [(y| - \infty) : (y'| + \infty)] \tag{2}$$

Pozostaje wykazać poprawność naszego przekształcenia, tzn. że punkty z $\hat{P}$ spełniające zapytanie $\hat{R}$ odpowiadają punktom z P spełniającym zapytanie R.

Lemat 1. *Niech p będzie punktem, a R będzie prostokątnym przedziałem w przestrzeni 2-wymiarowej. Wtedy:*

$$p \in R \iff \hat{p} \in \hat{R} \tag{3}$$

Dowód. Niech $R = [x : x']x[y : y']$ oraz $p = (p_x, p_y)$. Z definicji, $p \in R \iff x \leq p_x \leq x'$ i $y \leq p_y \leq y'$. To natomiast zachodzi wtedy i tylko wtedy gdy $(x| - \infty) \leq (p_x|p_y) \leq (x'| + \infty)$ i $(y| - \infty) \leq (p_y|p_x) \leq (y'| + \infty)$, a zatem wtedy i tylko wtedy gdy $\hat{p} \in \hat{R}$. □

Zatem nasza obserwacja jest poprawna. Warto zauważyć, że nie ma potrzeby pamiętać współrzędnych w postaci złożonej, a jedynie wykorzystywać naszą obserwację przy wykonywaniu porównań. To podejście można też zastosować w przypadku k-wymiarowym.

Wychodząc z idei 1-wymiarowych drzew przedziałowych chcielibyśmy zbudować drzewo BST do przeszukiwania naszego zbioru punktów. Pojawia się jednak problem, gdyż musimy w naszym przypadku wziąć pod uwagę dwie współrzędne x i y. Aby to obejść będziemy na zmianę brać pod uwagę współrzędną x i współrzędną y.

Dokładniej, w korzeniu budowanego drzewa dzielimy zbiór P pionową linią l na dwa równoliczne podzbiory. Informacje o l przechowujemy w korzeniu. P_{left} - zbiór punktów leżących na lewo od l lub zawartych w l przechowujemy w lewym poddrzewie, natomiast P_{right} - zbiór punktów leżących na prawo od l przechowujemy w prawym poddrzewie. Dla lewego syna korzenia dzielimy P_{left} na dwa równoliczne podzbiory linią poziomą; punkty pod lub na linii podziału przechowujemy w lewym poddrzewie lewego syna korzenia, natomiast punkty nad linią podziału przechowujemy w prawym poddrzewie. Lewy syn korzenia przechowuje linię podziału. Podobnie postępujemy z P_{right} dzieląc go poziomą linią na dwa równoliczne podzbiory przechowywane w synach prawego syna korzenia. Zbiory przechowywane we wnukach korzenia ponownie dzielimy liniami pionowymi. Ogólnie będziemy dzielić liniami pionowymi w wierzchołkach o parzystej głębokości, natomiast poziomymi w wierzchołkach o nieparzystej głębokości.

Tak zbudowane drzewo nazywamy kd-drzewem. Oryginalnie nazwa ta miała określać k-wymiarowe drzewo, więc drzewo którego budowę opisujemy powyżej nazwalibyśmy 2d-drzewem. Z czasem oznaczenie to przestało być stosowane i dziś mówimy o 2-wymiarowym kd-drzewie.

2 Budowa 2-wymiarowego kd-drzewa

Możemy skonstruować kd-drzewo korzystając z rekurencyjnej procedury, którą opiszemy poniżej. Procedura posiada dwa parametry: zbiór punktów i liczbę naturalną. Pierwszy parametr to zbiór punktów, dla których budujemy kd-drzewo (na początku wykonania tym zbiorem jest P), drugi parametr określa głębokość rekursji i pozwala stwierdzić głębokość rozpatrywanego w procedurze wierzchołka (na początku jest to 0). Procedura w wyniku zwraca korzeń kd-drzewa.

2.1 Złożoność czasowa i pamięciowa

Zacznijmy od złożoności czasowej. Najbardziej kosztowny krok jaki wykonujemy to znalezienie punktu podziału zadanego zbioru punktów. Znalezienie mediany może być przeprowadzone w czasie liniowym. Jednakże takie algorytmy są raczej skomplikowane, lepszym podejściem jest posortowanie

Algorithm 1 BuildKdTree(P, depth)

Input: Zbiór punktów P i obecna głębokość *depth*.
Output: Korzeń kd-drzewa dla zbioru P.

```
if P zawiera jeden punkt then
    return liść przechowujący ten punkt
else if depth jest parzysty then
    Podziel P na dwa równoliczne podzbiory linią pionową l przechodzącą przez medianę po współ-
    rzędnych x punktów ze zbiory P. Niech P_1 będzie zbiorem punktów leżących na lewo od l i na
    l, a P_2 będzie zbiorem punktów na prawo od l.
else
    Podziel P na dwa równoliczne podzbiory linią poziomą l przechodzącą przez medianę po współ-
    rzędnych y punktów ze zbiory P. Niech P_1 będzie zbiorem punktów leżących pod l i na l, a P_2
    będzie zbiorem punktów nad l.
end if
v_left ← BuildKdTree(P_1, depth + 1)
v_right ← BuildKdTree(P_2, depth + 1)
Stwórz wierzchołek v w którym będą trzymane informacje o l, jego lewym synem będzie v_left, a
prawym v_right
return v
```

wcześniej naszego zbioru P po współrzędnych x i y. Zamiast P w pierwszym parametrze będziemy przekazywać dwie posortowane listy. W takich listach łatwo będzie znaleźć w czasie liniowym mediany i podzielić następnie te listy zachowując porządek. Zatem złożoność czasową możemy przedstawić równaniem:

$$T(n) = \begin{cases} O(1) & \text{if} \quad n = 1, \\ O(n) + 2T(\lceil n/2 \rceil) & \text{if} \quad n > 1, \end{cases} \tag{4}$$

które rozwiązuje się do $O(n \log n)$. Aby wyznaczyć złożoność pamięciową zwracamy uwagę na to, że każdy liść w kd-drzewie przetrzymuje informacje o innym punkcie z P. Zatem mamy n liści. Ponieważ kd-drzewo jest drzewem BST, każdy wewnętrzny wierzchołek i liść wykorzystuje pamięć $O(1)$, zatem całkowita zajmowana pamięć możemy ograniczyć przez $O(n)$. Możemy zatem podsumować naszą analizę następującym lematem:

Lemat 2. *Kd-drzewo dla zbioru n punktów zajmuje $O(n)$ i może zostać skonstruowane w czasie $O(n \log n)$.*

3 Przeszukiwanie 2-wymiarowego kd-drzewa

Zajmiemy się teraz przeszukiwanie przestrzeni punktów P z wykorzystaniem kd-drzewa. Zbudowane przez nas drzewo dzieli przestrzeń na obszary, a każdy wierzchołek odpowiada jakiemuś obszarowi. Obszar odpowiadający wierzchołkowi v jest prostokątem, który może być nieograniczony z jednej lub więcej stron. Ograniczają go linie podziału wyznaczane przez przodków v - taki obszar będziemy oznaczać przez $region(v)$. Obszar korzenia kd-drzewo to cała rozpatrywana przestrzeń. Łatwo zauważyć, że punkt p jest przechowywany w poddrzewie pod wierzchołkiem v wtedy i tylko wtedy, gdy leży on w obszarze $region(v)$. Zatem musimy przeszukać poddrzewo zaczepione w wierzchołku v wtedy i tylko wtedy, gdy przedział zapytania przecina obszar $region(v)$. Ta obserwacja prowadzi

do następującego algorytmu: Przechodzimy przez wierzchołki drzewa, jednak odwiedzamy tylko

Algorithm 2 SearchKdTree(v, R)

Input: Korzeń (wierzchołek wewnętrzny) kd-drzewa i zapytanie R.
Output: Wszystkie punkty zawarte pod v, które zawierają się w R.

```
if v jest liściem then
  return punkt przechowywany w v jeżeli znajduje się w R.
else
  if region(lc(v)) jest w całości zawarty w R then
    ReportSubtree(lc(v))
  else if region(lc(v)) przecina R then
    SearchKdTree(lc(v), R)
  end if
  if region(rc(v)) jest w całości zawarty w R then
    ReportSubtree(rc(v))
  else if region(lc(v)) przecina R then
    SearchKdTree(rc(v), R)
  end if
end if
```

te, których obszary przecinane są przez R. Gdy osiągamy liście musimy jeszcze sprawdzić, czy zawierają się w R. Kiedy obszar jest w pełni zawarty w R możemy po prostu zwrócić wszystkie wierzchołki zawarte w poddrzewie - zajmuje się tym procedura ReportSubtree(v). Lewego syna v oznaczyliśmy przez $lc(v)$, natomiast prawego przez $rc(v)$.

Najważniejszym testem wykonywanym przez algorytm jest sprawdzenie, czy R przecina się z obszarem określonym przez v. By to sprawdzić moglibyśmy znaleźć wszystkie wierzchołki dla danego obszaru $region(v)$ w fazie preprocessingu i przechowywać je w drzewie, nie będzie to jednak potrzebne. Możemy utrzymywać obecny obszar podczas rekursji wykorzystując linie podziału zawarte w wierzchołkach wewnętrznych. Dla przykładu obszar odpowiadający lewemu synowi v o parzystej głębokości możemy policzyć z $region(v)$ w następujący sposób:

$$region(lc(v)) = region(v) \cap l(v)^{left}, \tag{5}$$

gdzie $l(v)$ jest linią podziału przechowywaną w v, a $l(v)$ jest półprzestrzenią na lewo od $l(v)$ i zawierającą $l(v)$.

3.1 Złożoność czasowa

Algorytm przeszukiwania nigdzie nie zakłada, że R jest obszarem prostokątnym i bez problemu zadziała dla bardziej złożonych zapytań. By jednak przeprowadzić analizę czasową działania algorytmu będziemy zakładać, że R jest prostokątnym przedziałem.

Lemat 3. *Zapytanie będące prostokątnym obszarem dla kd-drzewa przechowującego n punktów działa w czasie $O(\sqrt{n} + k)$, gdzie k oznacza liczbę zwróconych punktów.*

Dowód. Na początek zauważmy, że potrzebujemy przynajmniej $O(k)$ czasu, żeby zwrócić k wierzchołków, zawartych w zapytaniu. Musimy jeszcze wykazać ograniczenie na wierzchołki wewnętrzne,

które jesteśmy zmuszeni odwiedzić w przypadku, gdy rozpatrywany obszar przecina obszar zapytania. By to przeanalizować, znajdziemy ograniczenie na liczbę obszarów przecinanych przez pionową linię. W ten sposób będziemy w stanie wyznaczyć ograniczenie na liczbę obszarów przecinanych przez lewe i prawe ograniczenie obszaru zapytania. Liczbę obszarów przecinanych przez górne i dolne ograniczenia obszaru zapytania wyznaczamy w podobny sposób.

Niech l będzie pionową linią, a T będzie kd-drzewem. Niech $l(root(T))$ będzie linią podziału przechowywaną w korzeniu T. Linia l przecina obszar na lewo albo na prawo od $l(root(T))$, nigdy oba. Stąd wynika prosta rekurencja $Q(n) = 1 + Q(n/2)$. Nie jest ona jednak poprawna, gdyż musimy wziąć pod uwagę, że linie podziału dzieci korzenia są już liniami poziomymi. By zapisać poprawnie rekurencje musimy ująć w niej dwa stopnie naszego drzewa. Każdy z wierzchołków na głębokości 2 odpowiada obszarowi zawierającemu około $n/2$ punktów (dokładniej, zawiera $\lceil \lceil n/2 \rceil /2 \rceil = \lceil n/4 \rceil$ jednak asymptotycznie nie wpływa to na naszą rekurencję). Dwa z czterech wierzchołków o głębokości 2 odpowiadają obszarom przecinanym przez l więc musimy rekurencyjnie policzyć liczbę obszarów w nich zawartych. Dodatkowo l przecina obszar zawarty w korzeniu i powiązany z jednym z jego synów. Zatem $Q(n)$ możemy opisać następującym równaniem:

$$Q(n) = \begin{cases} O(1) & \text{if} \quad n = 1, \\ 2 + 2Q(n/4) & \text{if} \quad n > 1, \end{cases} \tag{6}$$

Ta rekurencja rozwiązuje się do $Q(n) = O(\sqrt{n})$. Oznacza to, że liczba obszarów przecinanych przez linię pionową jest ograniczona przez $O(/sqrtn)$. Podobna analiza wykazuje, że liczba obszarów przecinanych przez linię poziomą także jest ograniczona przez $O(/sqrtn)$. Zatem całkowita liczba obszarów przecinanych przez prostokątny przedział także jest ograniczona przez $O(/sqrtn)$. □

Przeprowadzona analiza złożoności czasowej jest raczej pesymistyczna. W wielu praktycznych sytuacjach czas ten będzie o wiele mniejszy, lecz wciąż ograniczony przez k.

4 Podsumowanie

Wydajność kd-drzew możemy podsumować przez następujące twierdzenie:

Twierdzenie 1. *Kd-drzewo dla zbioru P zawierającego n punktów na płaszczyźnie 2-wymiarowej zajmuje $O(n)$ pamięci oraz może zostać zbudowane w czasie $O(n \log n)$. Prostokątny obszar zapytania na kd-drzewie zajmuje $O(\sqrt{n} + k)$ czasu, gdzie k oznacza liczbę zwróconych wierzchołków.*

4.1 Uogólnienie na k wymiarów

Kd-drzewa można uogólnić z łatwością na d-wymiarowe przestrzenie. Algorytm konstrukcji jest bardzo podobny do 2-wymiarowego przypadku: W korzeniu dzielimy zbiór punktów na prawie równoliczne podzbiory punktów wg. hiperpłaszczyzny odpowiadającej osi X_1. Innymi słowy w korzeniu dzielimy punkty zgodnie z pierwszą współrzędną. W jego dzieciach dzielimy zgodnie z drugą współrzędną i tak postępujemy, aż osiągniemy wierzchołki o głębokości $d - 1$, w których podział wykonujemy po ostatniej współrzędnej. Na głębokości d podział ponownie przeprowadzamy po pierwszej współrzędnej. Zatrzymujemy rekurencję gdy pozostaniemy tylko z jednym punktem, który przechowujemy w liściu. Ponieważ d-wymiarowe kd-drzewo dla n punktów jest drzewem

binarnym o n liściach wykorzystuje ono $O(n)$ pamięci. Konstrukcja odbywa się w czasie $O(n \log n)$ (Oczywiście przyjmujemy, że d jest stałe).

Wierzchołki w d-wymiarowym kd-drzewie odpowiadają obszarom podobnie jak w przypadku 2-wymiarowym. Można zatem w podobny sposób wykazać, że złożoność czasowa dla zapytań prostokątnych ograniczmy przez $O(n^{1-(1/d)} + k)$.

Literatura

[1] *Computational geometry: algorithms and applications*, Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars, 2008

[2] *Advanced Data Structures: Lecture 3 Dr. Andre Schulz*, Jacob Steinhardt and Greg Brockman, 2010, *http://courses.csail.mit.edu/6.851/spring10/scribe/lec03.pdf* .