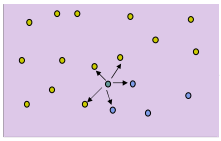
Przyspieszenie *kNN*

Ulepszone metody indeksowania przestrzeni danych:
R-drzewo,
R*-drzewo,
SS-drzewo,
SR-drzewo.

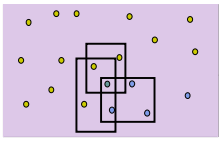
Plan wykładu

- Klasyfikacja w oparciu o przykładach
- Problem indeksowania przestrzeni obiektów
- Struktura przestrzeni obiektów
  - R-tree
- Operacje na *R* – drzewie
  - *Insert*
  - *Delete*
  - *Search*
- Wyszukiwania *k* najbliższych sąsiadów w oparciu o *R*-drzewie
- Ulepszenia

Klasyfikacja w oparciu o przykładach



IBL: k najbliższych sąsiadów



Reguły decyzyjne

Algorytm klasyfikacji

- **Wejście**: Tablica decyzyjna ***D***, nowy obiekt x_q
- **Wyjście**: Klasa decyzyjna, do której należy x_q (czyli $Dec(x_q)$)
- **Parametry**: k - liczba sąsiadów, funkcja odległości d

- **Krok 1**: Szukaj w zbiorze ***D***, k najbliżej położonych obiektów $\{x_1, x_2,..., x_k\}$
- **Krok 2**: Wyznacz klasę dla x_q na podstawie $Dec(x_1)$, $Dec(x_2)$,..., $Dec(x_k)$

Funkcja odległości

- Niech każdy obiekt *x* będzie zdefiniowany wektorem wartości: $<a_1(x),...,a_k(x)>$
- Odległość Euklidesowa:

$$d(x,y)=\sqrt{\sum_i (a_i(x)-a_i(y))^2}$$

- Dla atrybutów symbolicznych a_i:

$$a_i(x)-a_i(y)=\begin{cases}0 & a_i(x)=a_i(y)\\ 1 & a_i(x)\neq a_i(y)\end{cases}$$

- Atrybuty rzeczywiste: Unormować

Problematyki związane z klasyfikacją w oparciu o przykładach

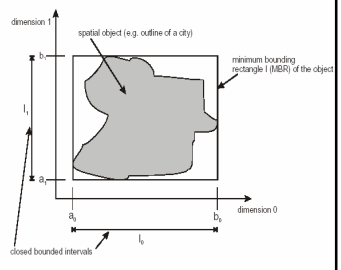
- Funkcja odległość
- Szukanie *k* najbliższych sąsiadów
- Strategia głosowanie

Wyszukiwanie *k* najbliższych sąsiadów

- Tablica danych:
  - Sekwencyjne poszukiwanie: $O(n)$ (n – liczba obiektów w zbiorze ***D***)
- Strukturalny zbiór danych (indeksowany zbiór danych):
  - Struktura:
    - Obiekty są prezentowane jako punkty w przestrzeni R^n
    - Przestrzeń obiektów jest podzielona na małe fragmenty
    - Fragmenty są zapamiętane w strukturze drzewiastym
  - Wyszukiwanie:
    - Przeglądaj drzewo i wybieraj fragmenty, gdzie mogą zawierać najbliższych sąsiadów dla danego obiektu
    - Szukać w znalezionych fragmentach najbliższych sąsiadów
  - Czas wyszukiwania: $O(\log n)$

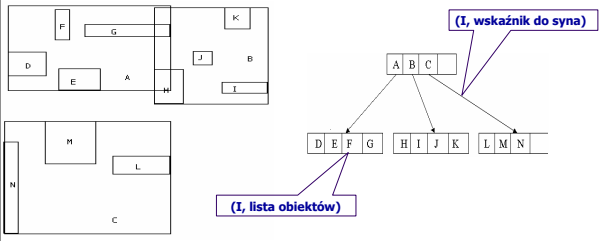
Podział przestrzeni obiektów

- Przestrzeń obiektów jest podzielona na *n*-wymiarowe prostokąty (*niekoniecznie rozłączne*).
- Prostokąty pokrywają wszystkie obiekty w zbiorze danych
- Prostokąty są minimalne (MBR - **M**inimum **B**ounding **R**ectangle)
- Opis prostokąta:
  $I = \{[a_1, b_1],...,[a_n, b_n]\}$



Struktura R-drzewa

- Drzewo zrównoważone
- Węzeł wewnętrzny zawiera klucze (opis prostokąta *I*) i wskaźniki do synów
- Węzeł zewnętrzny zawiera listę obiektów zawierających się w odpowiednim prostokącie



Własności R - drzewa

- Korzeń ma co najmniej dwóch synów
- Każdy węzeł wewnętrzny ma pomiędzy *m* a *M* synów
- Każdy liść zawiera pomiędzy *m* a *M* obiektów
- Wszystkie liście znajdują się na jednym poziomie drzewa
  - *m*: minimalna liczba rozgałęzienia w węźle
  - *M*: maksymalna liczba rozgałęzienia w węźle

  $2 \le m \le M$
- **Węzeł przepełniony**: zawiera więcej niż *M* rekordów (kluczy)
- **Węzeł niedomiarowy**: zawiera mniej niż *m* rekordów

Operacje na drzewie

- Wstawianie (*insert*)
- Usuwanie (*delete*)
- Wyszukiwanie (*search*)

Wstawianie obiektu do drzewa

Wejście:
- ***T*** – dane R-drzewo,
- x_{new} – nowy obiekt do wstawiania

Wyjście:
- R-drzewo ***T*** zawierające obiekt x_{new}

Wstawianie obiektu do drzewa

Wejście: ***T***: *R* - drzewo, x_{new} - obiekt do wstawiania
Wyjście: ***T*** z obiektem x_{new}

Algorytm *Insert*:
Krok 1. Wybieraj odpowiedni liść *L* do wstawiania x_{new} *używając ChooseLeaf*
Krok 2.
Jeśli jest miejsce w *L*, wstaw x_{new} w *L*
wpp.
2.1 Wstaw x_{new} w *L*
2.2 Podziel *L* na L_1 i L_2
2.3 Modyfikuj drzewo używając *AdjustTree*
Krok 3. Jeśli korzeń musi być podzielony, utwórz nowy korzeń, którego synami są dwie części poprzedniego korzenia

Funkcja *ChooseLeaf*

Wejście: ***T***: *R* - drzewo, x_{new} - obiekt do wstawiania
Wyjście: odpowiedni liść *L* dla x_{new}
(funkcja rekurencyjna)

Krok 1. *L* ← korzeń(***T***)
Krok 2. **if** *L jest liściem* **return** *L*;
else *// L jest węzłem wewnętrznym*
wybierz prostokąt $F_k \in L$ leżący „najbliżej" obiektu x_{new}
Jeśli takich jest kilka, wybierz najmniejszy prostokąt
Krok 3. Zastosuj *ChooseLeaf* dla F_k

Funkcja *AdjustTree*

Zadania:
- **Powiększać** istniejące prostokąty, żeby zawierały nowy obiekt
- **Podzielić** duże węzły na mniejsze, jeśli zawierają więcej niż *M* elementów

Wejście: ***T***: drzewo, *L* – liść, od którego drzewo ma być modyfikowane

Wyjście: ***T*** spełniające ograniczenia *R*-drzewa

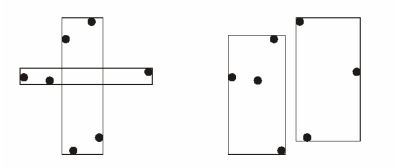
Funkcja *AdjustTree*

Funkcja *AdjustTree*(*L*)

Krok 1. *ActiveNode* ← *L*
Krok 2.
if *ActiveNode* jest korzeniem **stop**
else
P = *ActiveNode.Parent*();
Powiększ prostokąt w P związany z ActiveNode,
żeby zawierał on wszystkie prostokąty w *ActiveNode*;
Krok 3.
if *P* zawiera więcej niż *M* elementów
SpliteNode (*P*);
AdjustTree (*P.Parent*());

Funkcja *Splite*

- **Zadanie**: Podzielić przepełniony węzeł na dwa nowe węzły
- **Kryterium**: Nowe powstające prostokąty są małe

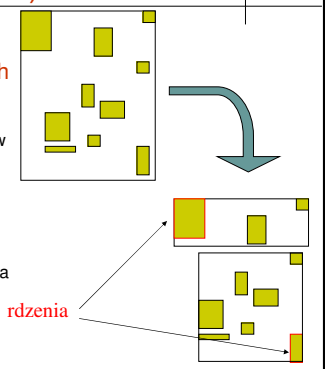


Dobry podział

Zły podział

Funkcja *Splite* (c.d.)

Idea:
- Wybierz rdzenia nowych grup:
  - wybierz dwa najdalej położone prostokąty i wstaw je do różnych węzłów
- Rozdziel pozostałe elementy:
  - wstaw *p* do tej grupy, która wymaga najmniejszego powiększania, aby zawierała *p*



Algorytm *Splite*

Wejście: Węzeł *w* zawiera M+1 elementów
Wyjście: Dwa węzły, każdy zawiera pomiędzy *m* a *M* elementów

Krok 1: Wybierz rdzenia grup i wstaw je do dwóch nowych węzłów
Krok 2: **repeat**
przydziel elementy
until wszystkie elementy są przydzielone do grup lub jedna grupa ma *m* (lub *M*+1-*m*) elementów
Krok 3: dołącz pozostałe elementy do drugiej grupy

Usuwanie obiektu z drzewa

Algorytm usuwania obiektu

Wejście: ***T***: *R* - drzewo, *x* - obiekt do usuwania
Wyjście: ***T*** : drzewo bez *x*

Krok 1: Szukaj liścia *L*, który zawiera x (używając *FindLeaf*)
Krok 2: Usuń *x* z *L*
Krok 3: Modyfikuj drzewo używając *CondenseTree*
Krok 4: Jeśli korzeń zawiera tylko jednego syna, usuń stary korzeń, syn będzie korzeniem.

Funkcja *FindLeaf*

Wejście: ***T***: *R*-drzewo, *x*: obiekt
Wyjście: liść, który zawiera *x*

Krok 1: Jeśli *T* jest liściem zwracaj *T*
Krok 2: Dla każdego syna *T.s* , jeśli prostokąt zawiera *x* to *FindLeaf*(*T.s*)
Krok 3: Jeśli *T* jest liściem, sprawdź, czy w nim x występuje. Jeśli tak, zwracaj *T*

Funkcja *CondenseTree*

Wejście: ***T***: *R* - drzewo, *L* – liść, od którego drzewo ma być poprawione
Wyjście: ***T***: skondensowane drzewo

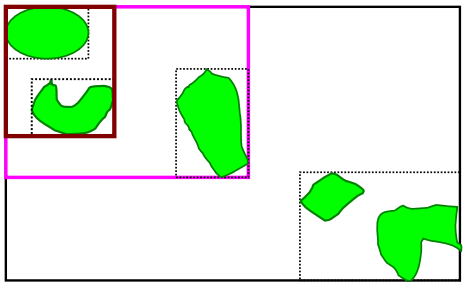
Krok 1: *N = L;*
Jeśli N nie jest korzeniem P = N.Parent()
Krok 2: **if** *N* zawiera mniej niż *m synów*
usuń *N z drzewa*
zapisz wszystkie elementy N na liście Q
Krok 3: **if** *N jest dobry*
zmniejsz prostokąt w N, żeby prostokąt w N był *MBR*
Krok 4: N = P; goto Krok 1
Krok 5: Wstaw wszystkie elementy Q do drzewa.

Wyszukiwanie k najbliższych sąsiadów za pomocą R- drzewa

MBR – Minimal Bounding Rectangle

- MBR jest n-wymiarowym prostokątem używanym w R-drzewie, który ogranicza obiekty .
- Własność MBR:
  - Każda ściana prostokąta zawiera co najmniej jeden punkt bazy danych.

Własność MBR



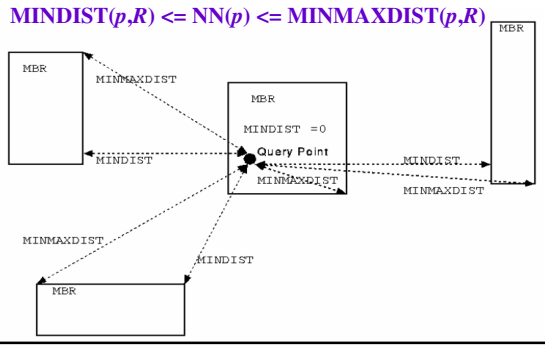
Funkcje odległości używane do wyszukiwania k-NN

- *MINDIST (p, R)*: minimalna odległość punktu p do prostokąta R.
- MINDIST(p, R) = $\min_{o \in R} d(p,o)$
- Własności:
  - Jeśli *p* znajduje się wewnątrz prostokąta, to *MINDIST(p,R)* = 0
  - Dla każdego $o \in D \cap R$,
    $MINDIST(p, R) \le d(p,o)$

Funkcje odległości używane do wyszukiwania k-NN

- MINMAXDIST *(p, R)* jest minimalna odległość punktu *p* do najdalszych punktów ścian prostokąta *R*
- Jeśli $d^*(p,S)$ oznacza odległość *p* do najdalszego punktu ściany *S*, to
  $MINMAXDIST\ (p, R) = min_{S:ściana\ R}\ d^*(p,S)$
- Własność:
  - Istnieje obiekt w *R*, którego odległość do *p* jest mniejsza niż MINMAXDIST
    $\exists_{o \in D \cap R}\ d(p,o) \le MINMAXDIST(p, R)$

MINDIST & MINMAXDIST



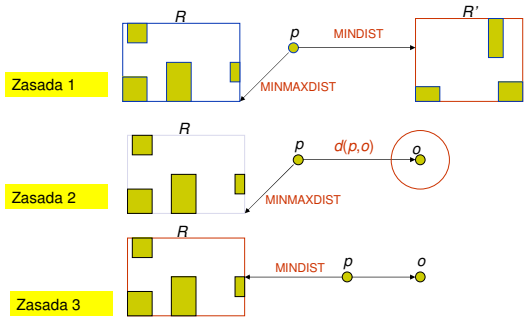
Algorytm szukanie najbliższego sąsiada

- Idea:
  - Przeglądaj drzewo od korzenia w dół.
  - Dla każdego węzła usuń rozgałęzienia, które nie powinny zawierać najbliższego sąsiada
  - Jeśli doidziesz do liścia, szukaj w nim najbliższego sąsiada.

Usuwanie zbędnych rogałęzień

- **Zasada 1**: Prostokąt *R'* nie zawiera najbliższego sąsiada dla *p* (więc będzie usuwany) jeśli istnieje prostokąt *R* taki, że:
  MINDIST(*p*,*R'*) > MINMAXDIST(*p*,*R*)
- **Zasada 2**: Obiekt *o* nie jest najbliższym sąsiadem *p* jeśli istnieje prostokąt *R* taki, że
  d(*p*,*o*) > MINIMAXDIST(*p*,*R*)
- **Zasada 3**: Prostokąt *R* nie zawiera najbliższego sąsiada punktu *p* jeśli istnieje obiekt *o* taki, że
  MINDIST(*p*,*R*) > d(*p*,*o*)

MINDIST vs MINMAXDIST



NN Search Algorithm

Krok 1: Inicjalizuj odległość do najbliższego sąsiada = ∞;
Krok 2: Obchodź drzewo od korzenia w dół.
Przy każdym nieodwiedzonym węźle *t* sortuj prostokąty zapisane w *t* rosnąco względem MINDIST i zapisz jego synów w liście ***ABL*** (**A**ctive **B**ranch **L**ist)
Krok 3: Usuń z ***ABL*** zbędne rozgałęzienia stosując strategie przycinania 1 i 2.
Krok 4: Dla każdego węzła *t* z ***ABL*** rekurencyjnie szukaj najbliższego sąsiada *p;*
Usuń węzeł *t* z ***ABL***
Krok 5: Jeśli węzeł jest liściem, poszukaj najbliższego sąsiada. Aktualizuj odległość do najbliższego sąsiada.
Krok 6: Po powrocie z rekursji skorzystaj z zasady przycinania 3 w celu usunięcia niezbędnych rozgałęzień
Krok 7: Jeśli ***ABL*** = ∅ zwróć najbliższego sąsiada.

Ulepszanie

- Tylko zasada przycinania 3 jest stosowana;
- Tylko MINDIST jest potrzebny;
- Przycinanie tylko gdy nowy sąsiad jest znaleziony

Szukanie k najbliższych sąsiadów

- Zapamiętać k aktualnych sąsiadów w uporządkowanym buforze (wg. rosnącej odległości sąsiada od punktu p)
- Przycinanie względem najdalszego punktu na liście

Modyfikacje R-drzew

- R*- tree: Modyfikacja operacja insert, zmniejsza część wspólną między regionami
- SS – tree: przestrzeń obiektów jest pokryta sferami
  - Zaleta: prostszy opis
  - Wada: regiony zachodzą się
- SR – tree: kombinacja R-tree i SS tree
- X – tree: regiony nie mają częsci wspólnej

R*-drzewo

Struktura i główne operacje

Struktura R*-drzewa

- Podobieństwo do R - drzewa:
  - Używa prostokątów **MBR** (**M**inimal **B**ounding **R**ectangle) do grupowania punktów w przestrzeni danych
  - Drzewo ma wszelkie własności jak *R*-tree (ograniczenia na liczbę rozgałęzień,...)
- Różnica: ulepszony wariant R-drzewa
  - Minimalizuje części wspólne między prostokątami (regionami) na jednym poziomie drzewa
  - Minimalizuje pola (objętości) prostokątów (regionów)
  - Minimalizuje wielkość drzewa

Struktura R* - drzewa (c.d.)- Modyfikacje

- Modyfikacja operacji Insert:
  - Optymalna heurystyka podziału węzła (inna kryterium oceny jakości podziału)
  - Nowa operacja do rekonstrukcji drzewa: wymuszone wstawianie (forced re-insert)

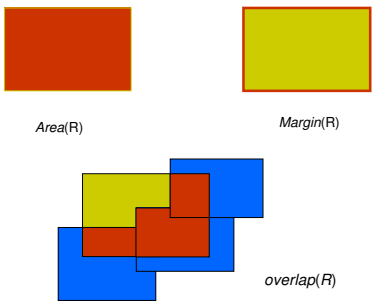
Podstawowe oznaczenia

- Pole prostokąta (objętość prostopadłościanu): *area*(*R*)
- Obwód prostokąta (powierzchnia prostopadłościanu): *margin*(R)
- Pokrycie (*overlap*): prostokątami w węźle *t*:

$$overlap(R_k) = \sum_{i=1,k\neq i}^{p} area\left(R_k.Rectangle \cap R_i.Rectangle\right)$$

gdzie R_1, R_2,...,R_p są rekordami węzła *t*

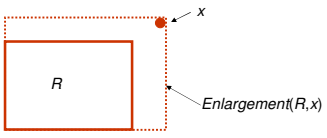
Pojęcia podstawowe – Ilustracja



Podstawowe operacje na prostokątach

- Powiększanie prostokąta: (*enlargement*(*R*,*x*))

Zwiększać prostokąt *R* minimalnie, żeby pokrywać obiekt *x*



Podstawowe operacje na drzewach

- Wstawianie (*insert*)
- Usuwanie (*delete*)
- Wyszukiwanie (*search*)

Wstawianie

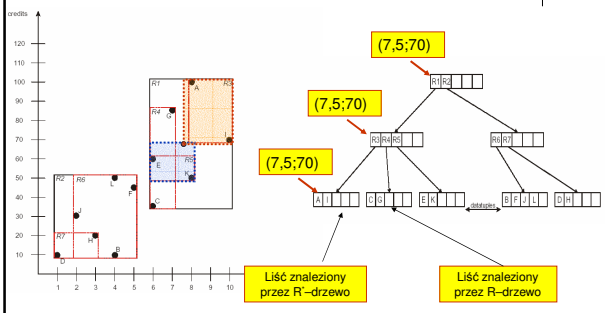
- **Ogólny schemat**:
  - Szukaj liścia do wstawiania (*ChooseLeaf*(x))
  - Wstaw nowy obiekt
  - Modyfikuj drzewo

ChooseLeaf – Idea

Przeglądaj drzewo top-down:

- W każdym węźle wewnętrznym:
  - wybierz poddrzewo zawierające prostokąt, który wymaga najmniejszego powiększenia, żeby pokrywać nowy obiekt
- W każdym węźle zewnętrznym (liść):
  - wybierz prostokąt, który po powiększaniu ma najmniejsze pokrycie (*overlap*) z innymi prostokątami tego samego liścia

ChooseLeaf – Ilustracja



Funkcja *ChooseLeaf*

Wejście: ***T***: *R** - drzewo, *x* - obiekt do wstawiania
Wyjście: odpowiedni liść *L* dla *x*

Krok 1. *L* ← korzeń(***T***)
Krok 2. **if** *L jest liściem*
wybierz prostokąt R taki, że *enlargement* (*R*,*x*) ma najmniejszą część wspólną z innymi prostokątami w tym liściu;
else // *L* jest *węzłem wewnętrznym*
wybierz syna *S* , który jest związany z prostokątem *R* o najmniejszym *enlargement* (*R*,*x*)
Krok 3. Zastosuj *ChooseLeaf* dla *S*

Modyfikuj drzewo – Ogólny schemat

- Jeśli liść *L* nie jest przepełniony to
  - Powiększ prostokąt w *L*, do którego nowy obiekt był wstawiany
  - Rekurencyjnie powiększ wszystkie prostokąty w przodkach *L*, żeby zawierały prostokąty w liściu *L*.
- Jeśli liść *L* jest przepełniony to
  - Podziel liść *L* na 2 liście L_1 i L_2 lub
  - Wykonuj *re-insert* dla części prostokątów w *L*
  - W sposób rekurencyjny modyfikuj ojca *L*

Podzielić przepełniony węzeł - Idea

Krok 1: Wybierz oś *S*, względem której podział ma być wykonywany

Krok 2: Wybierz na osi *S* najlepszy podział

Krok 3: Podziel obiekty

Oceny jakości podziału

- Niech *P* oznacza podział zbioru prostokątów *S* na dwa rozłączne podzbiory S_1 i S_2
- $bb(S_i)$: najmniejszy prostokąt zawierający S_i
  - *area-value*(*P*) = *area*[$bb(S_1)$] + *area*[$bb(S_2)$]
  - *margin-value*(*P*) = *margin*[$bb(S_1)$] + *margin*[$bb(S_2)$]
  - *overlap-value*(*P*) = *area*[$bb(S_1) \cap bb(S_2)$]
- Podział *P* jest dobry, jeśli wartości *area*, *margin* i *overlap* są małe

Algorytm wyboru osi

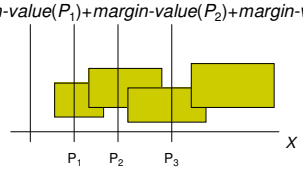
Krok 1: Dla każdej osi *X*

- Uporządkuj prostokąty względem *X*
- Oblicz wartość *margin-value*(*X*): suma *margin-value* wszystkich podziałów na *X*

Krok 2: Wybierz oś o najmniejszej wartości *margin-value*

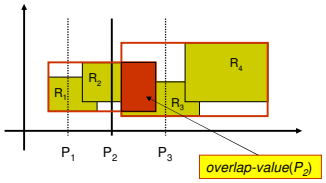
margin-value(X) =

= *margin-value*(P_1)+*margin-value*(P_2)+*margin-value*(P_3)



Algorytm wyboru miejsca podziału

- Dla wybranej osi *X*
  - Dla każdego podziału P na *X* oblicz *overlap-value*(*P*)
  - Wybierz podział o najmniejszej wartości *overlap-value*
  - Jeśli istnieje kilka podziałów o najmniejszej wartości *overlap-value*, wybierz ten z najmniejszą *area-value*

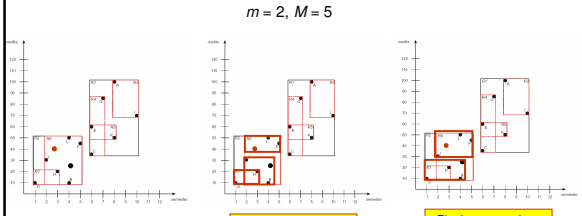


Rekonstrukcja drzewa

- W *R*-drzewie: regiony są niezmienne podczas wstawiania obiektu (chyba że jest podzielony na dwa regiony);
- Wada: utworzone na początku prostokąty są optymalne dla pierwszych obiektów, ale nie są optymalne dla nowych wstawianych obiektów
- Potrzeba narzędzi do rekonstrukcji drzewa
- R*-tree proponuje operację wymuszonego wstawiania (*forced reinsert*)

Wstawianie z rekonstrukcją - Ilustracja

m = 2, *M* = 5



Regiony przed wstawianiem

R – drzewo: regiony po wstawianiu

*R** -drzewo: regiony po wstawianiu

Idea algorytmu „*reinsert*”

(Przy wstawianiu nowego obiektu)
Jeśli węzeł *t* jest ***przepełniony*** (ma M+1 synów) i *t* nie był ***reorganizowany*** **to**

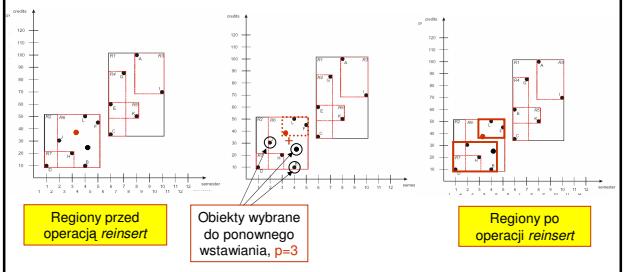
Krok 1: wytnij *p* synów węzła *t*
Krok 2: zmniejsz prostokąt w *t* tak, żeby nadal zawierał pozostałych synów
Krok 3: wstaw z powrotem prostokąty związane z wybranymi synami.

Algorytm *reinsert*

Dla przepełnionego węzła *t* (ma M+1 synów):

Krok 1: Dla każdego syna *s*, oblicz odległość środka prostokąta związanego z *s* do środka prostokąta w *t*
Krok 2: Uporządkuj malejąco synów względem odległości od swojego środka do środka prostokąta w *t*
Krok 3: Wybierz *p* pierwszych synów na liście
Krok 4: Usuń je i zmniejsz prostokąt w *t*
Krok 5: Wstaw z powrotem *p* usuniętych prostokątów

Reinsert - Ilustracja



Ocena R* -drzewa

- R*- drzewo jest stabilny wobec złośliwych danych
- Średni koszt wstawiania i usuwania na R*-drzewie jest niższy niż na R- drzewie
- Czas wyszukiwania jest szybszy niż na R-drzewie

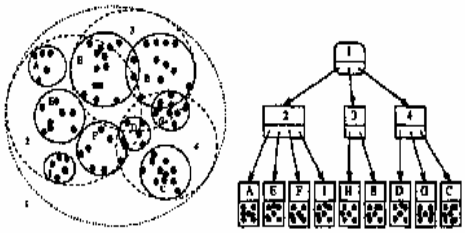
SS-drzewo

Struktura i główne operacje

Struktura SS-drzewo

- Podobna struktura do R-drzewa
- Regiony są kołami
- Opis regionu: para (środek, promień)
  - **środek:** środek ciężkości figury tworzonej przez środki kół zapisanych w synach
  - **promień:** odległość do najdalszego koła zapisanego w synach.

Struktura SS-drzewo



(a) Representation of high dimensional feature space in 2D

(b) Tree representation used in memory or on disk

Algorytm *Insert*

Krok 1. Szukaj liścia *L* do wstawiania
Krok 2. Wstaw nowy obiekt *x* do *L*
Krok 3. Modyfikuj drzewo (*od liścia *L* do korzenia*)
3.1 Jeśli zbadany węzeł *t* nie jest przepełniony
Powiększ promień w *t*
3.2 Jeśli węzeł *t* jest przepełniony i *t* nie był rekonstruowany
re-insert (*t*)
3.3 Jeśli *t* jest przepełniony i *t* był rekonstruowany
splite (t)

Algorytm podziału: *Splite*

Cel: Podziel zbiór kół na dwa podzbiory, których koła ograniczające mają małe promienie i jak najmniej zachodzą się na siebie

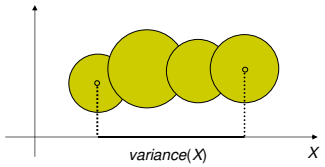
Krok 1: Wybierz oś *S*, względem której podział ma być wykonywany
Krok 2: Wybierz na osi *S* najlepszy podział
Krok 3: Podziel obiekty

Ocena podziału

- Niech *X* będzie osią współrzędnych
- Niech *S* będzie zbiorem kół
- *variance*(*S*,*X*) = *odległość między środkami dwóch najdalszych kół na osi X*
- Niech P oznacza podział zbioru kół *S* na dwa rozłączne podzbiory S_1 i S_2
- variance(P) = *variance*(S_1)+*variance*(S_2)
- Podział P jest dobry jeśli variance(*P*) jest minimalny

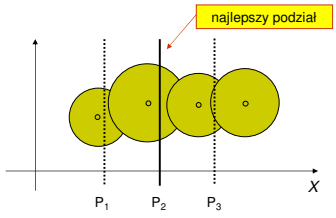
Wybór osi

- Dla każdej osi *X*
  - *Uporządkuj koła względem X*
  - *Oblicz odległość najdalszych kół na osi X. ozn. variance(X)*
- *Wybierz oś z największą wartością variance*



Szukanie miejsca podziału

- Dla wybranej osi *X*
  - Dla każdego podziału P na *X* oblicz *variance*(*P*)
  - Wybieraj podział o najmniejszej wartości *variance*



Ocena SS -drzewa

- Zaleta:
  - Mała informacja potrzebna do opisu koła niż prostokata
  - Średnica koła zwiększa się dość wolno względem wzrostu wymiaru danych (liczby atrybutów) → szybki czas wyszukiwania w jednym regionie.
- Wada:
  - Koło ma większą objętość niż prostokąt → większa część wspólna z innymi kołami (more overlap)

SR-drzewo

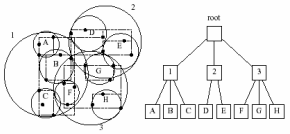
Struktura i główne operacje

Obserwacja

- Prostokąty dzielą przestrzeń danych na regiony o małej objętości, ale mają one dużą średnicę niż koło
- Koła dzielą przestrzeń danych na regiony o małej średnicy, ale ich objętości są większe niż prostokąty
- Rozwiązanie: kombinacja prostokątów i kół → SR - drzewo

Struktura SR - drzewa

- Podobna struktura do R-drzewa
- Region jest przecięciem koła i prostokąta
- Opis regionu: para (prostokąt, koło)
  - Prostokąt : I =[(a_1,b_1),...,(a_k,b_k)]
  - Koło: (środek, promień)



Algorytm: Insert

- Prostokąt jest aktualizowany tak jak w R*- drzewie
- Koło węźle *t* jest aktualizowane: (C_1,...,C_k – koła związane z synami *t*)
  - Środek:

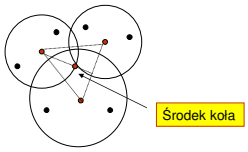
$$x_i = \frac{\sum_{i=1}^{n} C_i.x_i \times C_i.w}{\sum_{k=1}^{n} C_k.w} \quad (1 \le i \le D),$$

gdzie

C_i: koło związane z i-tym synem

$C_i.x_i$: i-ta współrzędna środka koła C_i

$C_i.w$: liczba obiektów zawartych w kole C_i

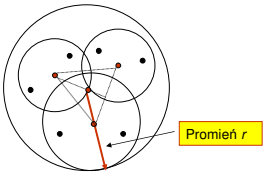


Algorytm Insert (c.d.)

- Promień:

$$r = \min(d_s, d_r),$$
$$d_s = \max_{1 \le k \le n} (\|x - C_k.x\| + C_k.r),$$
$$d_r = \max_{1 \le k \le n} (MAXDIST(x, C_k.R)),$$

- d_s: maksymalna odległość środka koła w węźle *t* do kół w synach
- d_r: maksymalna odległość środka koła w węźle *t* do prostokątów w synach

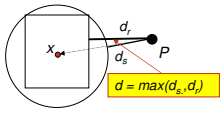


Szukanie najbliższego sąsiada

- Odległość *d* punktu *P* do regionu zapisanego w węźle C_k jest definiowana:

$$\begin{aligned} d &= \max(d_s, d_r), \\ d_s &= \max(0, \|p - C_k, x\| - C_k.r), \\ d_r &= MINDIST(p, C_k.\mathbf{R}), \end{aligned}$$

- d_s: minimalna odległość punktu *P* do koła w węźle C_k
- d_r: minimalna odległość punktu *P* do prostokąta w węźle C_k



Ocena SR-drzewa

- Zaleta:
  - Zmniejsza i objętość i średnicę regionu w porównaniu z R*-drzewem i SS-drzewem
  - Czas wyszukiwania krótszy niż SS-drzewo i R*-drzewo
- Wada:
  - Koszt budowania SR-drzewa jest większy niż SS-drzewa.
  - Rozmiar węzła większy od rozmiaru węzła SS-drzewa i R*-drzewa

Podsumowanie

- R*-drzewo: ulepszony wariant R-drzewa
  - Nowa heurystyka dla optymalizacji podziału -> modyfikacja w funkcji *splite*
  - Nowa operacja rekonstrukcji drzewa: re-insert
- SS-drzewo: regiony są kołami
- SR-drzewo: regiony są przecięcem prostokąta i koła