Analiza drzew samoorganizujących

(Analysis of self adjusting binary search trees)

Julia Majkowska

Praca licencjacka

Promotor: dr hab. Marcin Bieńkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

26 maja 2019

Streszczenie

Binarne drzewa wyszukiwań (BST) to jedna z najbardziej podstawowych struktur danych w informatyce. Struktura ta jest reprezentacją zbioru elementów i umożliwia wyszukiwanie elementów. Dla konkretnego ciągu zapytań można znaleźć optymalne drzewo BST, które zrealizuje go w minimalnym czasie. Wymaga to jednak znajomości całego ciągu zapytań przy tworzeniu struktury, przez co nie jest to praktyczne rozwiązanie. Binarne drzewa wyszukiwań mają bardzo wiele zastosowań w informatyce. Z tego powodu powstało wiele algorytmów reorganizacji tej struktury danych, umożliających efektywne wykonywanie operacji wyszukiwania bez znajomości wszystkich zapytań. Możemy wyróżnić dwa rodzaje takich algorytmów: algorytmy balansujące ograniczające pesymistyczny czas wyszukiwania (przykładami są drzewa AVL, AA oraz czerwono-czarne) oraz algorytmy samoorganizujące wykorzystujące wiedzę o obsłużonych zapytaniach do przyspieszenia obsługi następnych zapytań. W tej pracy zbadano efektywność implementacji wybranych algorytmów samoorganizujących. Porównano czasy działania samoorganizujących algorytmów Tango i Splay z własną implementacją drzewa czerwono-czarnego, implementacją drzewa czerwono-czarnego ze standardowej biblioteki C++, oraz z optymalnym drzewem statycznym.

Binary search trees (BST) are one of the most elementary data structures in computer science. The data structure is a representation of a set of elements and supports lookup of particular element. For a defined sequence of queries it is possible to create an optimal BST, that answers the queries in minimal time. It requires, however, prior knowledge of the entire query sequence, which makes this solution impractical. Binary search trees have a multidute of applications in computer science. This prompted the invention of several algorithms for reorganising this data structure, and enabling the structure to handle queries more efficiently, without knowing the entire sequence of queries. We can destinguish two kinds of these algorithms : balancing algorithms,which limit the pessimistic of query processing (such as AVL, AA and black-red trees) and self adjusting algorithms, which use already the knowledge of already processed queries to speed up the processing of the coming queries. This paper analyses the efficiency of implementations of selected self adjusting algorithms. I will compare the runtimes of self adjusting Tang and Splay algorithms and compare them to my implementation of black-red tree, the C++ standard template library implementation of black-red tree and the optimal static binary search tree

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Wstęp

Binarne drzewo przeszukiwań (inaczej BST) to jedna z najbardziej podstawowych struktur danych w informatyce. Stosuje się je w wielu systemach informatycznych i algorytmach. Ideą struktury jest trzymanie elementów w wierzchołkach, zawierających wartość elementu oraz wskaźniki na lewe i prawe poddrzewo. Elementy umieszcza się w strukturze w taki sposób, aby wszystkie wartości elementów znajdujące się w lewym poddrzewie wierzchołka w były elementami mniejszymi od w w zdefiniowanym porządku. Analogicznie wszystkie elementy w prawym poddrzewie są elementy większe w tym porządku. Na tak uporządkowanych danych można wyszukiwać daną wartość v poprzez zagłębianie się w drzewo wybierając odpowiednio lewe lub prawe poddrzewo w zależności od wyniku porównania v z wartością korzenia poddrzewa. W najbardziej podstawowej formie tej struktury operacje wstawiania, wyszukiwania i usuwania wykonuje się w czasie $O(H)$, gdzie H to wysokość drzewa. W pesymistycznym wypadku jest to czas zależny liniowo od liczby elementów w strukturze. Złożoność operacji słownikowych na drzewie BST zależy od głębokości drzewa, dlatego powstało wiele struktur danych ograniczających maksymalną głębokość drzewa. Jako przykłady można wymienić drzewa AA, AVL oraz czerwono-czarne, obsługujące operacje słownikowe w pesymistycznym czasie $O(\log n)$.

1.2. Problem optymalnego drzewa binarnego

W wielu zastosowaniach dane nie podlegają rozkładowi jednostajnemu tylko przejawiają jakąś lokalność. Przykładowo, jeśli dla drzewa 100-elementowego będą się pojawiać głównie zapytania o jeden element, można by znacząco zmniejszyć czas działania przesuwając dany element do korzenia. W takim wypadku struktura naszego drzewa zależy od zapytań. Problem znajdowania drzewa binarnego, które przy użyciu najmniejszej liczby operacji obsłuży ciąg zapytań to problem optymalnego

drzewa binarnego. Możemy rozróżnić dwa warianty tego problemu: statyczny oraz dynamiczny.

1.2.1. Statyczne optymalne drzewo binarne

Zdefiniujmy uporządkowany ciąg kluczy $e_1...e_n$, ciąg prawdopodobieństw $A_1...A_n$, będące prawdopodobieństwami wykonania wyszukiwania e_i i $B_0 \dots B_n$, oznaczające prawdopodobieństwa wyszukania elementu z przedziału $[e_i, e_{i+1}]$. Zgodnie z definicją Knutha [4] optymalne statyczne drzewo binarne to drzewo, które minimalizuje oczekiwany czas obsługi dowolnego ciągu zapytań wylosowanego zgodnie ze zdefiniowanym rozkładem.

Wraz z definicją problemu Knuth opublikował algorytm opierający się na programowaniu dynamicznym, znajdujący optymalne drzewo binarne w czasie $\mathrm{O}(n^2)$, który zostanie opisany szczegółowo w dalszej części pracy. W 1975 r. Mehlhorn opublikował algorytm aproksymujący ten problem, działający w czasie $O(n)$ [5]. Dla wariantu, w którym $B_i = 0$, istnieje algorytm Garsia-Wachsa, który znajduje optymalne drzewo w czasie $O(n \log n)$ [3].

1.2.2. Dynamiczne optymalne drzewo binarne

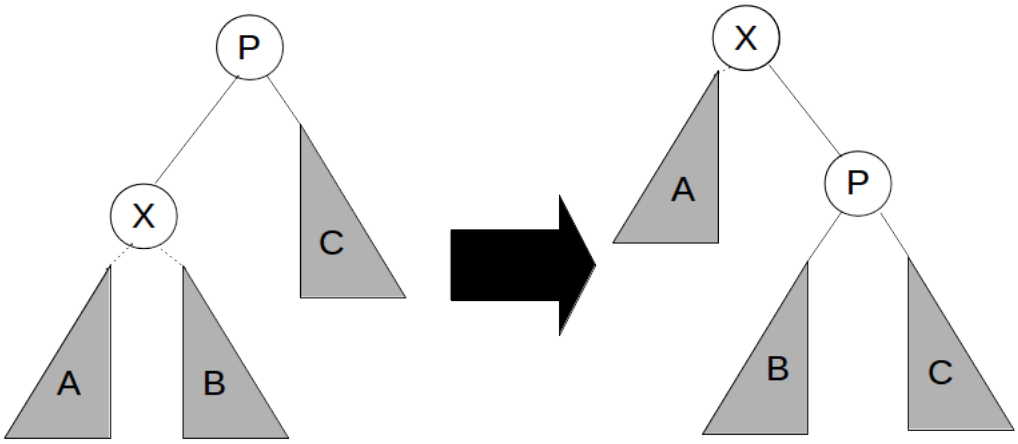
W dynamicznym wariancie problemu mamy na wejściu ciąg zapytań $X = \{x_1, x_2, ..., x_n\}$ o wyszukanie klucza $x_i \in 1, 2, ..., n$. Dla każdego zapytania o wartość w zaczynamy ze wskaźnikiem na korzeń drzewa i wykonując tylko niżej wymienione operacje chcemy przesunąć wskaźnik na wierzchołek zawierający klucz o wartości w. Dozwolone operacje to:

1. Przesunąć wskaźnik na lewego syna obecnie wskazanego wierzchołka

2. Przesunąć wskaźnik na prawego syna obecnie wskazanego wierzchołka

3. Przesunąć wskaźnik na rodzica obecnie wskazanego wierzchołka

4. Wykonać pojedynczą rotację obecnie wskazywanego wierzchołka. Poniżej ilustracja pojedynczej rotacji punku p w prawo



Rysunek 1.1: Pojedyncza rotacja w prawo w punkcie p

W tym problemie chcemy znaleźć drzewo, które minimalizuje liczbę operacji konieczną do obsłużenia ciągu zapytań x_i.

Definicja 1.1 (Konkurencyjność)**.** *Oznaczmy wejscie dla algorytmu jako σ, koszt algorytmu dla tego wejścia jako $ALG(\sigma)$ i optymalny koszt jako $OPT(\sigma)$. Mówimy, że algorytm jest k konkurencyjny, jeśli*

$$\forall_\sigma ALG(\sigma) \leq k * OPT(\sigma)$$

Liczbę k będziemy nazywali współczynnikiem konkurencyjności.

Dla każdego ciągu zapytań istnieje minimalny ciąg operacji odpowiadający na te zapytania. Znalezienie takiego rozwiązania wymaga znajomości całego ciągu zapytań z góry, co w wielu sytuacjach nie jest możliwe. Będziemy mówić, że drzewo jest dynamicznie optymalne ma stały współczynnik konkurencyjności. Udowodnienie dynamicznej optymalności dowolnego algorytmu BST jest problemem otwartym.

Rozdział 2.

Statyczne drzewo optymalne

2.1. Opis struktury

Oznaczmy ciąg zapytań jako $X = \{x_1, x_2, ..., x_m\}$, i zbiór elementów, które występują w tym ciągu jako $Y = \{y_1, y_2, ..., n\}$, gdzie $y_{i-1} < y_i < y_{i+1}$. Dodatkowo niech $C(y)$ będzie liczbą wystąpień y w ciągu X. Przyjmijmy, że nasze klucze y_i umieszczamy na drzewie BST $\mathcal{B}$. Zdefiniujmy $d(\mathcal{B}, y_i)$ jako głębokość wierzchołka zawierającego y_i w $\mathcal{B}$. Zdefiniujmy sobie koszt obsługi zapytania na drzewie $\mathcal{B}$ jako $K(x_i) = d(\mathcal{B}, x_i)$. Statyczne drzewo optymalne to takie drzewo, które dla danego ciągu zapytań, obsługuje je minimalnym sumarycznym kosztem. Na tym drzewie nie wkonujemy żadnych rotacji, a jego struktura pozostaje stała, przez cały czas obsługi zapytań. Ze względu na swój statyczny charakter drzewo to obsługuje jedynie operację wyszukiwania na drzewie i jest konstruowane offline.

2.2. Algorytm konstrukcji drzewa

Statyczne drzewo optymalne można skonstruuować przy pomocy programowania dynamicznego. Wprowadźmy sobie następujące oznaczenia, niech $Cost(i, j)$ będzie oznaczał minimalny koszt obsłużenia $\{x : x \in X \wedge x \in \{y_i, y_{i+1}, ..., y_j\}\}$ optymalnym drzewem BST składającym się z elementów $\{y_i, y_{i+1}, ..., y_j\}$, a $Root(i, j)$ niech będze indeksem korzenia tego optymalnego drzewa. Dodatkowo niech $Sum(i, j) = \sum_{k=i}^{j} C(y_k)$.

Zacznijmy od zdefiniowania przypadków bazowych:

$$Cost(i, i) = C(y_i)$$
$$Root[i][i] = i$$

Zauważmy, że jeśli $Root(i, j) = r$ wtedy $Cost(i, j) = Cost(i, r-1) + Cost(r+1, j) +$

$Sum(i,j)$. W takim razie możemy zapisać następujące zależności rekurencyjne:

$$Root(i,j) = argmin_r Cost(i,r-1) + Cost(r+1,j) + Sum(i,j)$$
$$Cost(i,j) = Cost(i,Root(i,j)-1) + Cost(Root(i,j)+1,j) + Sum(i,j)$$

W ten sposób otrzymujemy algorytm tworzenia drzewa o złożoności $O(n^3)$.

Twierdzenie 2.1. *[4]* $Root(i,j-1) \leq Root(i,j) \leq Root(i+1,j)$

Obserwacja 2.2. Korzystając z 2.1 możemy ograniczyć koszt działania tworzenia optymalnego drzewa BST do $0(n^2)$

Dowód. Policzmy sumarycznyczny koszt działania algorytmu.

$$\sum_{i=1}^{n}\sum_{j=i}^{n} Root(i+1,j) - Root(i,j-1) + 1$$
$$\leq n^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} Root(i+1,j) - \sum_{i=1}^{n}\sum_{j=1}^{n} Root(i,j-1)$$
$$= n^2 + \sum_{i=2}^{n+1}\sum_{j=1}^{n} Root(i,j) - \sum_{i=1}^{n}\sum_{j=0}^{n-1} Root(i,j)$$
$$\leq n^2 + \sum_{i=1}^{n+1}\sum_{j=1}^{n} Root(i,j) - \sum_{i=1}^{n}\sum_{j=1}^{n-1} Root(i,j)$$
$$= n^2 + (\sum_{i=1}^{n}\sum_{j=1}^{n} Root(i,j) - \sum_{j=1}^{n} Root(n+1,j)) - (\sum_{i=1}^{n}\sum_{j=1}^{n} Root(i,j) - \sum_{i=1}^{n} Root(i,n))$$
$$= n^2 + \sum_{i=1}^{n} Root(i,n) - \sum_{j=1}^{n} Root(n+1,j)$$
$$= O(n^2)$$

□

2.3. Opis implementacji

Implementacja statycznego drzewa optymalnego składa się z dwóch klas `static_tree` i `tree_vert`. Klasa `tree_vert` zostanie opisana w kolejnym rozdziale.

2.3.1. static_tree

Klasa `static_tree<T>` udostępnia następujące metody:

- Konstruktor `static_tree(const vector<int>&)` - konstruuje drzewo korzystając z listy zapytań do obsłużenia. Zakłada się, że wyszystkie elementy z zapytań występują w strukturze.
- `bool find(T)` - wykonuje wyszukiwanie na optymalnym drzewie binarnym i zwraca czy element znajduje się w strukturze.
- Destruktor niszczący całą strukturę.

Z racji, że struktura ta jest statyczna operacje wstawiania oraz usuwania elementów nie są obsługiwane.

Rozdział 3.

Drzewa Splay

3.1. Opis struktury

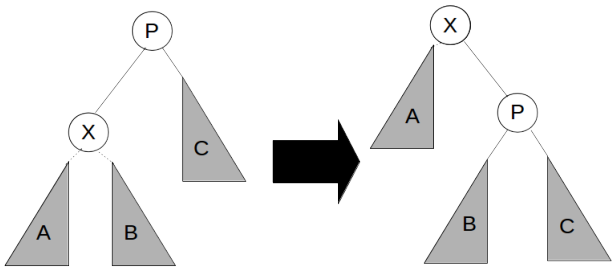
Drzewo Splay zostało stworzone w 1985 przez Sleatora i Trajana [6].

Drzewo Splay jest drzewem samoorganizującym się. Podstawą działania tej struktury jest operacja splay, polegająca na przesunięciu odpowiedniego wierzchołka do korzenia, korzystając z rotacji.

3.1.1. Splay

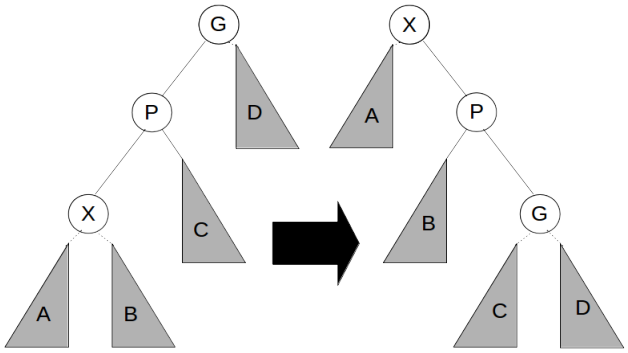
Operacja splay dla danego wierzchołka odbywa się w krokach, do momentu, kiedy wierzchołek nie stanie się korzeniem.Oznaczmy interesujący nasz wierzchołek jako x, niech p będzie ojcem, a g dziadkiem x na początku kroku. Możemy wyróżnić 3 przypadki kroków operacji splay:

1. ZIG - Jeśli x jest lewym synem korzenia wykonujemy pojedynczą rotację korzenia w prawo. Analogicznie kiedy x jest prawym synem korzenia wykonujemy rotację korzenia w lewo.



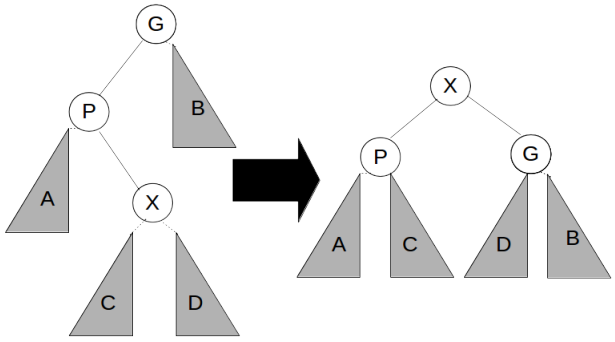
Rysunek 3.1: Krok ZIG operacji Splay

2. ZIGZIG - Jeśli zarówno x i p są lewymi synami wykonujemy u g rotację w prawo, a następnie rotujemy w prawo p. Analogicznie postępujemy kiedy x i p są prawymi synami.



Rysunek 3.2: Krok ZIGZIG operacji Splay

3. ZIGZAG - Jeśli x jest prawym synem, a p jest lewym synem, rotujemy x w lewo a następnie g w prawo. Analogicznie postępujemy kiedy x jest lewym synem, a p jest lewym synem.



Rysunek 3.3: Krok ZIGZAG operacji Splay

3.1.2. Wyszukiwanie

Drzewo splay spełnia zależność BST, więc można wyszukiwać w nim elementy jak w normalnym drzewnie BST. Po znalezieniu odpowiedniego wierzchołka wykonujemy na nim operację splay przenosząc go do korzenia.

3.1.3. Rozdzielanie drzewa względem elementu

Aby rozdzielić drzewo względem danego elementu, wyszukujemy go w drzewie i wykonujemy operację splay na elemencie, na którym zakończyliśmy poszukiwanie. Wynikiem będą lewe i prawe poddrzewo korzenia zmodyfikowanego drzewa.

3.1.4. Łączenie dwóch drzew

Będziemy łączyć drzewa d_1 i d_2 o własności, że $\forall_{k_1 \in d_1, k_2 \in d_2} k_1 < k_2$. Aby to zrobić wykonujemy operację splay na wierzchołku zawierającym minimalny element d_2. Następnie podłączamy d_1 jako lewe poddrzewo zmodyfikowanego d_2.

3.1.5. Wstawianie

Aby wstawić element e do drzewa najpierw wyszukujemy go w drzewie, jeśli element już znajduje się w drzewie to wykonujemy operację splay na odpowiadającym mu wierzchołku. Jeśli elementu nie ma w drzewie, wtedy rozdzielamy drzewo względem e na drzewa d_1 i d_2. Następnie podłączamy d_1 jako lewe poddrzewo elementu e, a d_2 jako prawe poddrzewo.

3.1.6. Usuwanie

Aby usunąć element, wyszukujemy go w drzewie i wykonujemy na odpowiadającym wierzchołku operację splay. Następnie łączymy lewe i prawe poddrzewo w jedną strukturę.

3.2. Analiza złożoności

3.2.1. Złożoność pamięciowa

W każdym węźle trzymamy dwa wskaźniki na synów oraz wartość klucza. Cała struktura zajmuje pamięć $O(n)$.

3.2.2. Analiza zamortyzowana złożoności czasowej

Twierdzenie 3.1. *Dla ciągu zapytań $X = \{x_1, x_2, ..., x_m\}$, gdzie $\forall_{1 \leq i \leq m} x_i \in \{1, 2, ..., n\}$, drzewo splay działa w czasie $O((m+n)logn)$.*

Dowód. Zauważmy, że główną składową kosztu operacji słownikowych jest operacja splay. Pozostałe części operacji można wykonać w czasie stałym. Przeprowadzimy

analizę zamortyzowaną kosztu wykonywania operacji splay. Wprowadźmy następujące oznaczenia :

- $s(w)$ - liczba wierzchołków w poddrzewie wierzchołka w
- $r(w) = \lfloor log(s(w)) \rfloor$
- Funkcja potencjału $\Phi = \sum_w r(w)$

Przeanalizujmy najpierw zmianę potencjału Φ przy poszczególnych krokach operacji Splay. Będziemy korzystali z notacji na schematach 3.1, A14, 3.3

1. **ZIG** : Zmienia się tylko potencjał x i p

$$\begin{aligned}\Delta\Phi &= 1 + \Delta r(x) + \Delta r(p) \\ &= 1 + r'(x) - r(x) + r'(p) - r(p) \\ &= 1 + r'(p) - r(x) && \text{ponieważ } r'(x) = r(p)\end{aligned}$$

2. **ZIGZIG** : Zmienia się potencjał x, p i g

$$\begin{aligned}\Delta\Phi &= 2 + r'(x) - r(x) + r'(p) - r(p) + r'(g) - r(g) \\ &= 2 - r(x) + r'(p) - r(p) + r(g) && \text{ponieważ } r'(x) = r(g) \\ &\leq 2 + r'(g) + r'(x) - 2r(x) && \text{ponieważ } r(x) < r(p) \text{ i } r'(x) > r(p) \\ &\leq 3(r'(x) - r(x)) && \text{z wypukłości funkcji logarytmicznej}\end{aligned}$$

3. **ZIGZAG** : Zmienia się potencjał x, p i g

$$\begin{aligned}\Delta\Phi &= 2 + r'(x) - r(x) + r'(p) - r(p) + r'(g) - r(g) \\ &= 2 - r(x) + r'(p) - r(p) + r(g) && \text{ponieważ } r'(x) = r(g) \\ &\leq 2 + r'(g) + r'(p) - 2r(x) && \text{ponieważ } r(x) < r(p) \\ &\leq 2(r'(x) - 2r(x)) && \text{ponieważ } 2r'(x) - r'(p) - r'(g) \geq 2\end{aligned}$$

We wszystkich przypadkach koszt operacji jest zatem nie większy niż $3(r'(x)-r(x))+1$, gdzie czynnik 1 występuje tylko przy operacji ZIG, a zatem tylko raz przy każdej operacji Splay. Dla całej operacji Splay(x_i) musimy zsumować zmianę potencjału wszytkich kroków. Otrzymujemy w ten sposób sumę teleskopową skracającą się do $3(r(\text{korzeń})-r(x_i))+1 \leq logn$. Aby otrzymać oszacowanie całego czasu działania na całej sekwencji X, musimy jeszcze uwzględnić zmianę potencjału pomiędzy stanem początkowym a końcowym : $\sum_x r'(x) - r(x) \leq \sum_x O(logn) = O(nlogn)$. Zatem sumaryczny czas działania to $O((m+n)logn)$.

□

3.3. Konkurencyjność

Hipoteza 3.1. *Drzewo splay jest dynamicznie optymalne tzn.* $SPLAY(X) = O(OPT(X)$ *gdzie* $SPLAY(X)$ *to koszt obsłużenia ciągu zapytań* X *przez drzewo splay, a* $OPT(X)$ *przez optymalne dynamiczne drzewo działające offline.*

3.4. Opis implementacji

Implementacja składa się z dwóch klas `splay_tree` i `tree_vert`.

3.4.1. tree_vert

Ta klasa reprezentuje wierzchołek drzewa binarnego oraz jego poddrzewo. Udostępnia następujące metody:

- Konstruktory `tree_vert(T val, tree_vert<T>*f = NULL, tree_vert<T>*l = NULL, tree_vert<T>*r = NULL, bool is_null = false)` tworzący wierzchołek trzymający wartość. Umożliwia ustawienie ojca wierzchołka (f), lewego syna (l), prawego syna (r) oraz wartość oznaczającą czy wartość w wierzchołku ma być traktowana jako cześć drzewa (is_null)
- Konstruktor `tree_vert()` tworzący pusty wierzchołek którego wartość nie jest częścią drzewa.
- Desktruktor zwalniający wszystkie wskaźniki w poddrzewie.
- `void disown_left()` i `void disown_right()` - obcinają odpowiednio lewego i prawego syna wierzchołka.
- `bool hook_up_left(tree_vert<T>*)` i `bool hook_up_right(tree_vert<T>*)` - podpinają wierzchołek z argumentu jako odpowiednio lewego i prawego syna wierzchołka.
- `bool get_disowned()` - rozcina krawędź pomiędzy wierzchołkiem a jego ojcem.
- `bool is_root(), bool is_left(), bool is_right()` - sprawdzają czy wierzchołek jest korzeniem, lewym synem czy prawym synem.
- `void rotate_left(), void rotate_right()` - wykonują rotację ojca wierzchołka.
- `void splay()` - wykonuje operację splay na danym wierzchołku.
- `tree_vert<T>* search( T )` - wyszukuje klucza danego jako argument w poddrzewie wierzchołka i zwraca ostatni napotkany wierzchołek.

- `tree_vert<T>* next() , tree_vert<T>* prev()` - zwracają natępnika oraz poprzednika wierzchołka w drzewie.

3.5. splay_tree

Ta klasa reprezentuje całe drzewo splay. Udostępnia następujące metody:

- Konstruktory `splay_tree(),splay_tree(tree_vert<T>*)` tworzące puste drzewo i drzewo danym korzeniu.
- Desktruktor zwalniający wszystkie wskaźniki w drzewie.
- `void splay(T)` - wykonuje operację splay na ostatnim wierzchołku na ścieżce wyszukiwania klucza podanego jako argument.
- `bool find(T)` - sprawdza czy element znajduje się w drzewie.
- `tree_vert<T>* lower_bound( T ), tree_vert<T>* upper_bound( T )` - wyszukują klucza danego jako argument w drzewie i zwraca wskaźniki na odpowiednio najmniejszy element większy równy i najmniejszy element mniejszy równy. Gdy wynik nie istnieje zwraca NULL.
- `bool insert(T)` - wstawia element do drzewa zwraca `false` jeśli element znajdował sie wcześniej w drzewie oraz `true` w przeciwnym wypadku.
- `bool erase(T)` - wstawia element do drzewa zwraca `true` jeśli element znajdował sie wcześniej w drzewie oraz `false` w przeciwnym wypadku.
- `vector<splay_tree<T>* > split(splay_tree<T>* tree, T searched)` - rozdziela drzewo tree na drzewa z elementami mniejszymi, równymi i większymi od podanej wartości.

Dodatkowo na drzewach splay są dostępne następujące funkcje :

- `splay_tree<T>* join( splay_tree<T>* lesser, splay_tree<T>* greater)` - wykonuje operację join na dwóch drzewach

Rozdział 4.

Drzewo Tango

4.1. Opis struktury

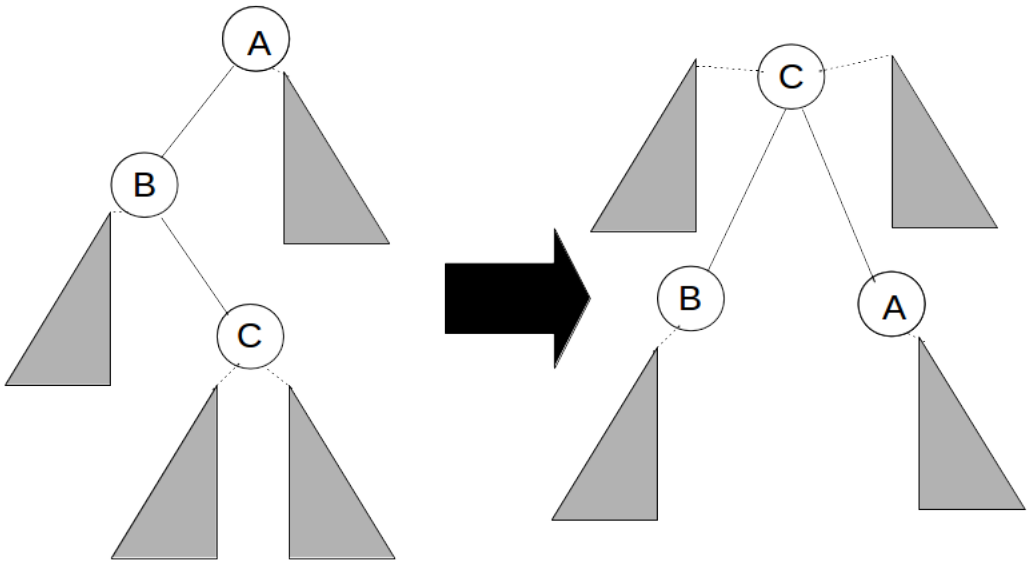
Drzewo Tango jest strukturą samoorganizującą, symulującą działanie zbilansowanego drzewa BST, w której wyróżniamy ścieżkę od korzenia do ostatnio odwiedzonego elementu. Opublikowane zostało w 2004 roku przez Demaine'a, Harmona, John Iacono'a, and Mihai Patraşcu [2]. Jako że symulowana struktura jest statyczna, drzewo Tango nie wspiera operacji wstawiania ani usuwania elementów, a konstrukcja struktury odbywa się offline. W 2006 roku opublikowany został algorytm struktura multi-splay, który opiera się na podobnej koncepcji, a wspiera operacje wstawiania i usuwania [7]. Do opisu działania algorytmu Tango wprowadźmy następujące pojęcia :

- Drzewo referencyjne - zbilansowane drzewo BST zawierające wszystkie te same elementy co nasza struktura. Jeśli liczba elementów nie jest potęgą dwójki, w ostatniej warstwie może brakować liści z prawej strony drzewa.
- Preferowana ścieżka wierzchołka– ścieżka prowadząca od wierzchołka (korzenia poddrzewa) do wyróżnionego elementu, w naszym wypadku będzie to najpóźniej żądany element w poddrzewie.
- Preferowany syn – syn wierzchołka leżący na preferowanej ścieżce wierzchołka. Jeżeli ostatnim odwiedzonym wierzchołkiem poddrzewa jest korzeń, wtedy korzeń nie ma preferowanego syna.
- Krawędź preferowana – krawędź pomiędzy wierzchołkiem a preferowanym synem
- Drzewo pomocnicze – drzewo czerwono-czarne [1] zawierające wierzchołki ścieżki preferowanej.
- Niepreferowany syn – syn wierzchołka nie leżący na preferowanej ścieżce

- Niepreferowana krawędź – krawędź prowadząca do nie preferowanego syna
- Głębokość wierzchołka $\mathcal{D}(w)$ – odległość wierzchołka w od korzenia drzewa
- Maksymalna i minimalna głębokość wierzchołka – maksimum i minimum głębokości w poddrzewie wierzchołka w w drzewie pomocniczym.

4.1.1. Konstrukcja struktury

Oznaczmy zbiór elementów w naszym drzewie jako $\mathcal{E}$. Analogicznie niech $\mathcal{E}(v)$ będzie zbiorem elementów. Rozpatrzmy drzewo BST $\mathcal{B}$ zawierające elementy $\mathcal{E}$. Niech $\mathcal{S}$ będzie zbiorem wierzchołków preferowanej ścieżki w $\mathcal{B}$. Wierzchołki $\mathcal{S}$ umieszczamy na drzewie czerwono czarnym. Niepreferowanych synów podpinamy pod odpowiadających ojców z preferowanej ścieżki dodatkowymi krawędziami. Dodane krawędzie nie są częścią drzewa czerwono czarnego. Powtarzamy procedurę dla poddrzew, w którym korzeniami są niepreferowani synowie wierzchołków z $\mathcal{S}$.



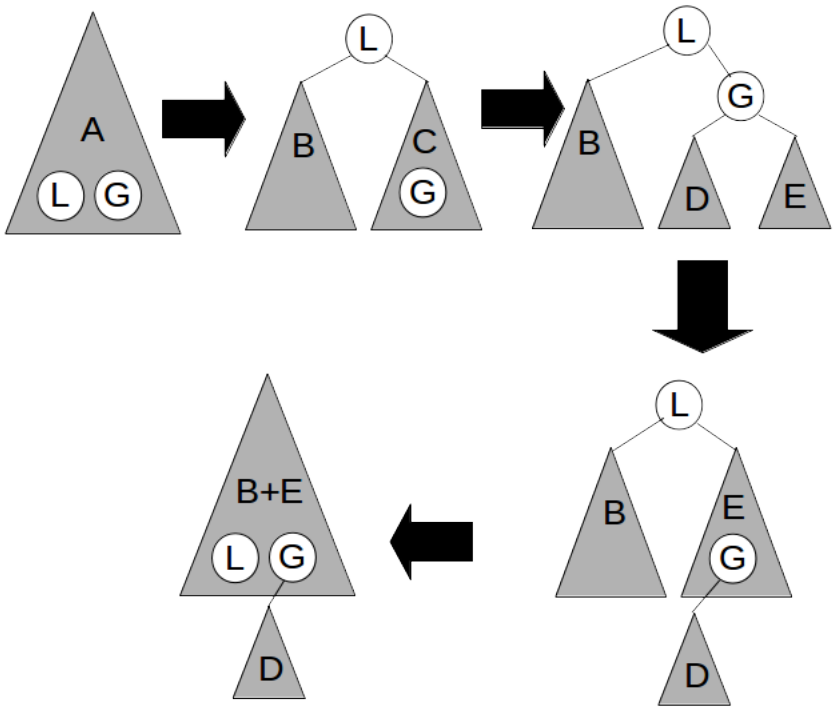
4.1.2. Operacje na drzewie pomocniczym

Do zdefiniowania operacji słownikowych na drzewie Tango będziemy korzystali z następujących operacji na drzewie czerwono-czarnym.

1. Split- Rozdzielenie drzewa czerwono-czarnego na dwa osobne zbilansowane drzewa, zawierające klucze mniejsze i większe od danego
2. Join- Połączenie dwóch drzew czerwono-czarnych w jedno

4.1.3. Łączenie i rozdzielanie drzew preferowanych ścieżek

Aby aktualizować preferowane ścieżki, potrzebujemy umieć połączyć dwa drzewa preferowanych ścieżek oraz rozdzielić drzewo preferowanej ścieżki. Do połączenia dwóch drzew wystarczy wykorzystać operację join. Aby rozdzielić ścieżkę w danym wierzchołku w, potrzebujemy znaleźć w drzewie pomocniczym wszystkie wierzchołki o głębokości większej niż $\mathcal{D}(w)$ i zrobić z nich osobne drzewo pomocnicze. Zauważmy, że istnieje przedział kluczy, dla których wszystkie wierzchołki w drzewie pomocniczym są głębokości większej niż $\mathcal{D}(w)$. Korzystając z maksymalnej głębokości, możemy znaleźć minimalny i maksymalny klucz o głębokości większej niż $\mathcal{D}(w)$. Następnie przy pomocy dwóch operacji Split wydzielamy dolną część ścieżki do osobnego drzewa pomocniczego i łączymy z powrotem pozostałe klucze przy pomocy operacji join.



4.1.4. Aktualizacja preferowanych ścieżek

Mając wierzchołek v chcemy zaktualizować drzewo w taki sposób, aby ścieżka do wierzchołka v była ścieżką preferowaną całego drzewa. Niech wierzchołek v będzie częścią preferowanej ścieżki $\mathcal{P}$, jakiegoś poddrzewa drzewa referencyjnego. $\mathcal{P}$ może być pojedynczym wierzchołkiem. Zaczynamy od rozdzielenia $\mathcal{P}$ w wierzchołku v. Preferowany syn wierzchołka, jeżeli istnieje, staje się nie preferowanym synem. Następnie wykonujemy poniższe kroki, tak długo, aż nie dotrzemy do korzenia całego drzewa.

1. Niech v będzie ostatnim elementem preferowanej ścieżki z wierzchołka r. Znajdujemy ojca o wierzchołka r.

2. Rozdzielamy ścieżkę preferowaną do której należy o w wierzchołku o. Preferowany syn wierzchołka staje się nie preferowanym.

3. Do górnej części ścieżki preferowanej dołączamy ścieżkę z r do v.

4.1.5. Wyszukiwanie

Do wyszukiwania klucza k będziemy symulować wyszukiwanie na drzewie referencyjnym. Najpierw wyszukujemy k w drzewie pomocniczym. Jeśli nie znaleźliśmy klucza k w ścieżce preferowanej, wtedy kontynuujemy wyszukiwanie w poddrzewie niepreferowanego syna ostatniego wierzchołka, na którym zakończyliśmy wyszukiwanie. Na koniec aktualizujemy preferowane ścieżki.

4.2. Analiza złożoności

4.2.1. Złożoność czasowa

Oszacujmy złożoność operacji na drzewach pomocniczych.

Lemat 4.1. *Złożoność dowolnej operacji na drzewie pomocniczym to* $O(\log\log n)$ *gdzie* n *to liczba elementów w strukturze.*

Dowód. Preferowane ścieżki, są ścieżkami zbilansowanego drzewa binarnego, zatem ich długość jest nie większa niż $\lceil \log n \rceil$. Operacje na drzewach pomocniczych są implementowane przy pomocy drzewa czerwono-czarnego i są wykonywane w czasie logarytmicznym od rozmiaru drzewa. Zatem koszt operacji to $O(\log\log n)$. □

Następnie oszacujmy koszt pojedynczego dostępu do elementu w drzewie Tango.

Lemat 4.2. *Złożoność pojedynczego dostępu do elementu* x_i *w drzewie Tango to* $O((k+1)(1+\log\log n))$*, gdzie* n *to liczba elementów w strukturze, a* k *liczba nie preferowanych krawędzi na ścieżce z korzenia do* x_i*.*

Dowód. Koszt dostępu można rozdzielić na koszt wyszukiwania i koszt aktualizacji preferowanych ścieżek w drzewie. Przy wyszukiwaniu odwiedzimy nie więcej niż $k+1$ drzew pomocniczych. Z 4.1 wynika, że wyszukiwanie w drzewie pomocniczym ma złożoność $O(\log\log n)$. Zatem sumaryczny koszty wyszukania to $O((k+1)(1+\log\log n))$. Analogicznie przy aktualizacji dla każdego odwiedzonego drzewa pomocniczego wykonujemy jedna operację Split i dwie operacje Join, wiec sumaryczny koszt to również$O((k+1)(1+\log\log n))$. □

4.2.2. Złożoność pamięciowa

Dla każdego wierzchołka będziemy pamiętać maksymalnie dwa wskaźniki na synów w drzewie pomocniczym, dwa wskaźniki na nie preferowanych synów w drzewie referencyjnym, jeden wskaźnik na ojca w drzewie pomocniczym, oraz klucz. Zatem złożoność pamięciowa to $O(n)$.

4.3. Konkurencyjność

4.3.1. Przeplotowe ograniczenie dolne

Zdefiniujemy sobie model do szacowania ograniczenia dolnego na liczbę operacji na optymalnym drzewie binarnym. Stwórzmy na kluczach $x_i \in \{1, ..., n\}$ idealne drzewo binarne B. Jeśli $n \neq 2^k - 1$ wtedy nasze drzewo będzie pełne, a nie idealne. Struktura drzewa nie będzie się zmieniać w czasie.

Definicja 4.1 (Lewy i prawy obszar)**.** *Dla danego wierzchołka y i drzewa B jego lewym obszarem $L(y)$ będzie lewe poddrzewo y oraz y, a prawy obszar $P(y)$ to prawe poddrzewo y.*

Przy każdym zapytaniu x_i dla wszystkich wierzchołków na ścieżce od korzenia do x_i odnotowujemy czy x_i znajduje się w jego prawym czy lewym poddrzewie.

Definicja 4.2 (Przeplot)**.** *Dla danego wierzchołka y, drzewa B i ciągu zapytań X, przeplotem nazwiemy parę zapytań (x_i, x_j) spełniającą następujące warunki :*

1. $(x_i \in P(y) \wedge x_j \in L(y)) \vee (x_i \in L(y) \wedge x_j \in P(y))$
2. $\neg\exists_{k\in[i,j]}(x_k \in L(y) \vee x_k \in P(y)$

Zdefiniujmy, także funkcję $IB(X, i)$ jako liczbę przeplotów wprowadzonych przez i-te zapytanie, w ciągu zapytań X, a $IB(X) = \sum_i IB(X, i)$.

Będziemy starali oszacować liczbę operacji w optymalnym algorytmie BST, korzystając z liczby przeplotów. Niech T_i będzie stanem drzewa binarnego po wykonaniu zapytań $x_1, x_2, ..., x_i$, przez dowolny deterministyczny algorytm wyszukiwania i reorganizacji.

Definicja 4.3. *Punkt przejściowy wierzchołka y w momencie i to wierzchołek z , o najmniejszej głębokości w T_i, taki że ścieżka z korzenia T_i przechodzi przez jakiś wierzchołek zarówno z $L(y)$ jaki i $P(y)$ w pełnym drzewie B.*

Lemat 4.4. *Dla każdego wierzchołka y i momentu i istnieje dokładnie jeden punkt przejściowy.*

Dowód. Niech l i r będą najniższymi wspólnymi przodkami odpowiednio wierzchołków z $L(y)$ i z $P(y)$ w T_i . Ponieważ klucze w lewym obszarze stanowią spójny przedział posortowanego ciągu wszystkich kluczy, a w drzewie BST wspólny przodek dwóch wierzchołków ma wartość klucza pomiędzy wartościami synów, w takim razie l jest elementem lewego poddrzewa. Analogicznie r jest elementem prawego poddrzewa. Zauważmy również, że klucze całego poddrzewa y w B stanowią spójny przedział posortowanego ciągu wszystkich kluczy, zatem najniższy wspólny przodek należy albo do lewego albo do prawego obszaru. Z tego wynika, że l lub r jest najniższym wspólnym przodkiem całego poddrzewa. Załóżmy bez straty ogólności, że jest to l. Zauważmy, że r będzie punktem przejściowym dla y w momencie i. Z definicji, najniższego wspólnego przodka, jest jedynym wierzchołkiem o najmniejszej głębokości, którego ścieżka przechodzi przez wierzchołek z $R(y)$. Ścieżka do r przechodzi też przez l, które jest elementem $L(y)$. □

Lemat 4.5. *Jeśli w momencie i z był punktem przejściowym wierzchołka y i algorytm BST obsługując zapytania $x_j, ..., x_k$ nie przechodzi przez z, ani nie wykonuje rotacji, która by zmniejszyła głębokość z, to z jest punktem przejściowym w każdym momencie z przedziału $[j, k]$.*

Dowód. Zdefiniujmy l i r jak w poprzednim dowodzie i załóżmy bez straty ogólności, że l jest wspólnym przodkiem wszystkich wierzchołków z $L(y)$ i $P(y)$ w momencie j. Zatem r jest punktem przejściowym y w momencie j. Skoro algorytm BST nie przechodzi w zapytaniach przez r to r pozostaje najniższym wspólnym przodkiem wierzchołków $P(Y)$ w T_i. Dodatkowo, skoro nie została wykonana, żadna rotacja zmniejszająca głębokość r to poddrzewo r w T_i pozostaje niezmienne. W czasie wykonywania zapytań $x_j, ..., x_k$, żaden element z $L(y)$ nie stał się synem r, zatem jakiś element $L(Y)$ musi być wspólnym przodkiem wszystkich wierzchołków z $L(y)$ i $P(y)$. Z tego wynika, że r pozostaje punktem przejściowym y. □

Lemat 4.6. *Każdy wierzchołek z może być w momencie i punktem przejściowym co najwyżej jednego wierzchołka y.*

Dowód. Weźmy dowolne dwa wierzchołki y_1 i y_2, pokażemy, że ich punkty przejściowe w momencie i są różne. Zdefiniujmy dla nich podobnie jak w dowodach poprzednich lematów odpowiednio l_1, r_1 i l_2, r_2. Jeśli y_1 i y_2 nie znajdują się na jednej ścieżce do korzenia (żaden nie jest przodkiem drugiego), to ich lewe i prawe obszary są rozłączne, a zatem ich punkty przejściowe muszą być różne. Jeśli tak nie jest to załóżmy bez straty ogólności, że y_1, jest przodkiem y_2. Jeśli punkt przejściowy y_1 znajduje się w innym obszarze y_1 niż y_2, wtedy punkty przejściowe muszą być różne. W przeciwnym wypadku punkt przejściowy y_1 musi być wspólnym przodkiem elementów z $L(y_2)$ i $R(y_2)$. Zatem głębokość punktu przejściowego y_1 będzie nie większa niż głębokość l_2 i r_2. Z kolei punkt przejściowy y_2 ma głębokość nie mniejszą niż głębokości l_2 i r_2. Skoro l_2 i r_2 muszą mieć różne głębokości, to punkty przejściowe y_1 i y_2 muszą być różne. □

Twierdzenie 4.7. *$OPT(X) \geq \frac{IB(X)}{2} - n$, gdzie OPT to koszt działania optymalnego algorytmu BST na ciagu zapytań X.*

Dowód. Będziemy szacować z dołu liczbę operacji wykonywanych przez optymalny algorytm przez liczbę punktów przejściowych odwiedzonych (przechodzi przez nie ścieżka z korzenia do x_i lub wykonuje się na nich rotacje zmniejszające ich głębokość) w trakcie wykonywania zapytań. Korzystając z 4.6 będziemy zliczać te wydarzenia dla każdego wierzchołka y osobno, a następnie je zsumujemy. Niech $x_{a_1}, xa_2, ..., xa_k$ będzie maksymalnym podciągiem zapytań, w którym każde dwa kolejne zapytania są o elementy w różnych obszarach y. p będzie liczbą przeplotów w y. Załóżmy bez straty ogólności, że zapytania $x_{a_{2i}}$ będą zapytaniami a elementy lewego obszary, $x_{a_{2i+1}}$ prawego. Rozpatrzmy l i r takie jak w poprzednich dowodach. Wszystkie zapytania o prawy obszar y odwiedzają r, a o lewy odwiedzają l. Zatem przy dwóch kolejnych zapytaniach albo algorytm albo będzie przechodził przez punkt przejściowy przy wyszukiwaniu, albo punkt przejściowy się zmieni, co również wymaga odwiedzenia punktu przejściowego. Zatem dla dwóch kolejnych zapytań z naszego podciągu algorytm przynajmniej raz odwiedza punkt przejściowy y. Dla danego wierzchołla otrzymujemy odwiedzamy $\lfloor \frac{p}{2} \rfloor \geq \frac{p}{2} - 1$. Sumując po wszystkich y otrzymujemy $OPT(X) \geq \frac{IB(X)}{2} - n$. □

4.3.2. Zastosowanie przeplotów w oszacowaniu konkurencyjności

Lemat 4.8. *Liczba nie preferowanych krawędzi na ścieżce z korzenia do x_i jest równa $IB(X, i)$.*

Dowód. Obecność niepreferowanej krawędzi $\{a, b\}$ na ścieżce oznacza, że ostatnie zapytanie o element w poddrzewie wierzchołka y w innym poddrzewie niż x_i. To odpowiada przeplotowi. □

Twierdzenie 4.9. *Dla ciągu zapytań $X = \{x_1, x_2, ..., x_m\}$, takich że $\forall_i x_i \in \{1, 2, ..., n\}$ sumaryczny koszt działania drzewa Tango to $O((OPT(X) + n)(1 + loglogn))$, gdzie $OPT(X)$ to koszt optymalnego dynamicznego drzewa BST działającego offline.*

Dowód. Z 4.8 i 4.2 wynika, że koszt obsługi pojedynczego zapytania na drzewie Tango to $O((IB(X, i) + 1)(1 + loglogn))$.Dodatkowo, dla każdego wierzchołka co najwyżej raz będziemy zmieniać jego stan z nie posiadania preferowanego syna na posiadanie jednego. Zatem sumaryczny koszt zapytania to $O((IB(X) + m + n)(1 + loglogn))$. Z 4.7 wiemy, że $OPT(X) \geq IB(X) - n$. Oczywiste też jest, że $OPT(X) \geq m$. Z tego wynika, że koszt algorytmu to $O(OPT(X) + n)(1 + loglogn))$. □

4.4. Opis implementacji

Implementacja składa się z czterech klas `br_tree`, `br_vert`, `tango_tree` i `tango_vert`. Dodatkowo funkcja split zwraca klasę pomocniczą `splitted_tree`.

4.4.1. br_vert

Ta klasa reprezentuje wierzchołek drzewa czerwono-czarnego oraz jego poddrzewo. Poza standardowymi właściwościami drzewa czerwono czarnego, wierzchołek utrzymuje minimalną i maksymalną głębokość potrzebne do wykonywania operacji na drzewach pomocniczych. Ta klasa dziedziczy po wcześniej wymienionej klasie `tree_vert`. Rozszerza funkcjonalność klasy bazowej o aktualizowanie czarnej wysokości przy zmianach w strukturze drzewa oraz o następujące metody:

- Konstruktor `tree_vert(T val, tree_vert<T>*f = NULL, tree_vert<T>*l = NULL, tree_vert<T>*r = NULL, bool is_null = false)` tworzący wierzchołek trzymający wartość. Umożliwia ustawienie ojca wierzchołka (f), lewego syna (l), prawego syna (r) oraz wartość oznaczającą czy wartość w wierzchołku ma być traktowana jako cześć drzewa (is_null).

- `br_vert<T>* left_son(), br_vert<T>* right_son(),br_vert<T>* parent(), br_vert<T>* grandparent(),br_vert<T>* uncle(),br_vert<T>* brother()` - zwracają odpowiednio synów, ojca, dziadka, drugiego syna dziadka oraz drugiego syna ojca wierzchołka.

- `void update_black_height()` - aktualizuje czarną wysokość wierzchołka oraz minimalną oraz maksymalną głębokość poddrzewa.

4.5. br_tree

Ta klasa reprezentuje całe drzewo czerwono-czarne. Aby zachować poprawne własności maksymalnej i minimalnej głębokości przy wstawianiu i łączeniu drzew aktualizacja głębokości musi się odbyć na całej ścieżce do korzenia, a nie do pierwszego czerwonego wierzchołka jak w normalnym drzewie czerwono czarnym. Udostępnia następujące metody:

- Konstruktory `br_tree(),splay_tree(br_vert<T>*)` tworzące puste drzewo i drzewo danym korzeniu.

- `void destroy()` - zwalnia pamięć zajmowana przez wszystkie wierzchołki w drzewie. Destruktor nie zwalnia wierzchołków.

- `void join_right(br_tree<T>* r, br_vert<T>* pivot)` - łączy drzewo z drzewem r i wierzchołkiem *pivot*. Wszystkie klucze drzewa r i klucz wierzchołka *pivot* muszą być większe niż wszystkie klucze w drzewie. Operacja niszczy strukturę drzewa r, ale nie zwalnia pamięci obiektu.

- `void join_right(br_tree<T>* l, br_vert<T>* pivot)` - łączy drzewo z drzewem l i wierzchołkiem *pivot*. Wszystkie klucze drzewa l i klucz wierzchołka *pivot* muszą być mniejsze niż wszystkie klucze w drzewie. Operacja niszczy strukturę drzewa l, ale nie zwalnia pamięci obiektu.

- `int height()` - zwraca czarną wysokość drzewa.

- `bool empty()` - sprawdza czy drzewo jest puste.

- `br_tree<T>* tree_union(br_tree<T>* A).` - dołącza do drzewa wszystkie wierzchołki z drzewa A.

- `pair<br_tree<T>*,br_tree<T>* > split(T).` - rozdziela drzewo na elementy mniejsze i większe równe od argumentu. Operacja niszczy strukturę drzewa, ale nie zwalnia pamięci obiektu.

- `splitted_tree<T> split2(T)` - rozdziela drzewo na elementy mniejsze, większe i wydziela element równy kluczowi (jeśli taki istnieje). Operacja niszczy strukturę drzewa, ale nie zwalnia pamięci obiektu.

- `bool find(T)` - sprawdza czy element znajduje się w drzewie.

- `tree_vert<T>* lower_bound( T ), tree_vert<T>* upper_bound( T )` - wyszukują klucza danego jako argument w drzewie i zwraca wskaźniki na odpowiednio najmniejszy element większy równy i najmniejszy element mniejszy równy. Gdy wynik nie istnieje zwraca NULL.

- `bool insert(T)` - wstawia element do drzewa zwraca `false` jeśli element znajdował sie wcześniej w drzewie oraz `true` w przeciwnym wypadku.

- `bool insert_vert(br_vert<T>*)` - wstawia element do drzewa zwraca `false` jeśli element znajdował sie wcześniej w drzewie oraz `true` w przeciwnym wypadku.

- `bool erase(T)` - wstawia element do drzewa zwraca `true` jeśli element znajdował sie wcześniej w drzewie oraz `false` w przeciwnym wypadku.

Dodatkowo na drzewach pomocniczych jest dostępna funkcja :

- `br_tree<T>* join( br_tree<T>* lesser, br_tree<T>* greater)` - łączy dwa drzewa. Pierwszy argument ma wszystkie klucze mniejsze niż drugi argument

4.5.1. tango_vert

Ta klasa reprezentuje wierzchołek drzewa tango oraz jego poddrzewo. Ta klasa dziedziczy po klasie `br_vert` i jest odpowiedzialna za obsługiwanie operacji na niepreferowanych krawędziach. Udostępnia następujące metody:

- Konstruktor `tango_vert(T val, int d, tango_tree<T> * npls = NULL, tango_tree<T> * nprs = NULL, tango_vert<T>* ps = NULL)` tworzący wierzchołek trzymający wartość *val*. Umożliwia ustawienie niepreferowanych ojca wierzchołka (ps), lewego syna (npls), prawego syna (nprs)
- Destruktor zwalniający wszystkie wierzchołki w poddrzewie wierzchołka
- `tango_vert<T>* left_son(), tango_vert<T>* right_son()` - zwracają odpowiednio lewego i prawego syna wierzchołka.
- `bool has_left(), bool has_right()` - sprawdzają czy wierzchołek ma niepreferowane lewe i prawe poddrzewo.
- `bool add_left(tango_tree<T>* ), bool add_right(tango_tree<T>*)` - dodają drzewo jako niepreferowane lewe i prawe poddrzewo.
- `void remove_left(), void add_right()` - usuwają niepreferowane lewe i prawe poddrzewo.
- `bool reorganize_left(T), bool reorganize_right(T)` - wywołują reorganizację względem klucza na lewym i prawym niepreferowanym poddrzewie (jeśli istnieje).

4.6. tango_tree

Ta klasa reprezentuje całe tango. Udostępnia następujące metody:

- Konstruktory `tango_tree(vector<T>),tango_tree(const tango_tree<T> &)` tworzące nowe drzewo o danych kluczach i kopiujący.
- Deskturktor zwalniający pamięć zajmowana przez wszystkie wierzchołki w drzewie.
- `tango_vert<T>* Root()` - zwraca korzeń drzewa.
- `void become_unpreferred()` - podpina drzewo pod ojca korzenia w drzewie referencyjnym.
- `bool reorganize(T)` - wyszukuje wartości w drzewie i wykonuje reorganizacje, aby ostatni element na ścieżce wyszukiwania był na ścieżce preferowanej całego drzewa.

- `bool find(T)` - sprawdza czy element znajduje się w drzewie.

Rozdział 5.

Badanie wydajności drzew

5.1. Generowanie danych

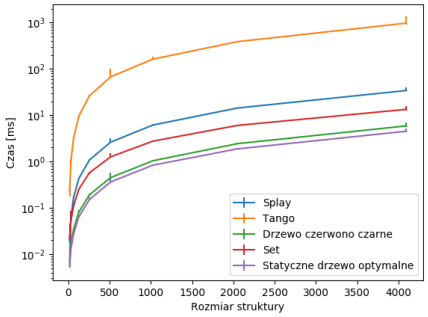
W niżej wymienionych eksperymentach drzewa są testowane na danych losowych typu `int`. Każda struktura jest testowana na tym samym zestawie danych.

5.2. Testowanie dla problemu dynamicznej optymalności

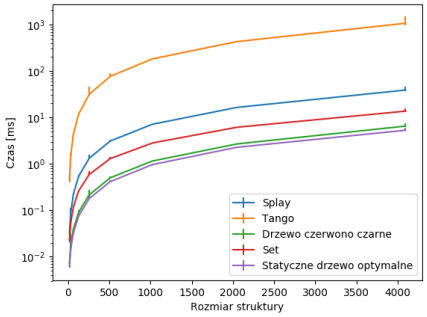
Zbadamy w jakim czasie obsługują zapytania implementacji poszczególnych struktur. Porównuję czasu działania drzew: Tango, Splay, czerwono-czarnego, statycznego drzewa optymalnego oraz standardowej struktury set. W tym eksperymencie nie bierzemy pod uwagę czasu budowania struktury. Wszystkie struktury zawierają elementy $e_i \in [1, ..., n]$. Do struktury, do których elementy dodaje sie pojedynczo, czyli drzewa Tango, drzewa Splay oraz set, wstawiałam elementy w kolejności losowej (takiej samej dla wszystkich struktur). Zapytania są losowane tak aby $\forall_i x_i \in [1, ..., n]$. Eksperyment powtarzam dla rosnących geometrycznie danych $n = \{16, 32, 64, ..., 1024\}$ wykonując po 50 losowych prób. W tym rozdziale przedstawione są wyniki dla liczby zapytań równej rozmiarowi struktury. Bardziej szczegółowe wyniki znajdują się w A.

5.2.1. Rozkład Gaussa

Dane generowane są z rozkładem zbliżonym do rozkładu normalnego nakładając podłogę na wynik funkcji `normal_distribution` i powtarzając losowania jeśli wynik wychodzi poza porządany przedział.



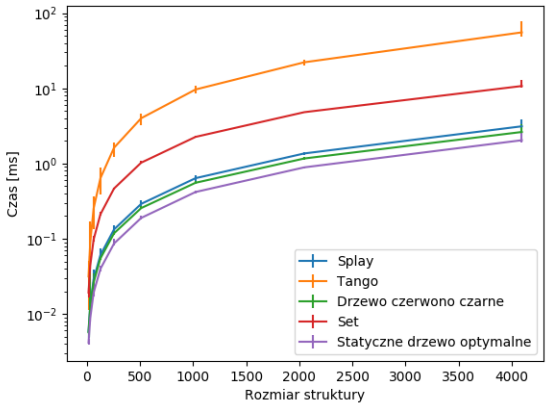
Rysunek 5.1: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{n}{10})$.



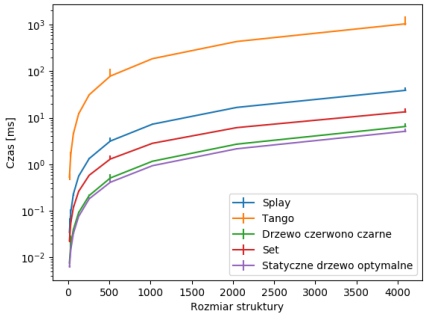
Rysunek 5.2: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{n}{4})$.

5.2.2. Rozkład jednostajny z prawdopodobieństwem powtórzenia wierzchołka

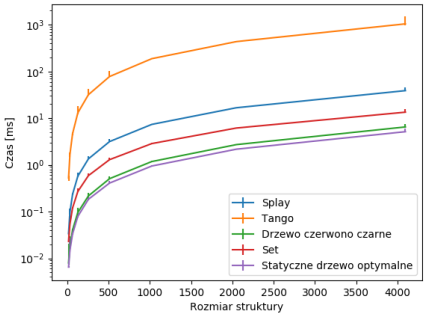
Zapytania są generowane w dwóch krokach. Najpierw losowany jest z określonym prawdopodobieństwem czy zostanie powtórzone poprzednie zapytanie. Jeśli, nie kolejne zapytanie jest losowane z rozkładem jednostajnym korzystając ze funkcji `uniform_int_distribution` ze standardowej biblioteki `random`.



Rysunek 5.5: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 95% powtórzenia wierzchołka.



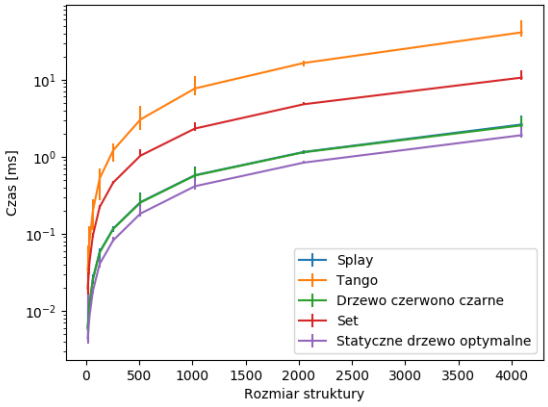
Rysunek 5.3: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{n}{2})$.



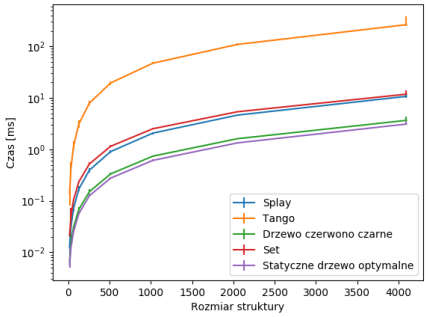
Rysunek 5.4: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{3n}{4})$.

5.2.3. Losowanie ścieżki w grafie losowym z prawdopodobieństwem zmiany wierzchołka

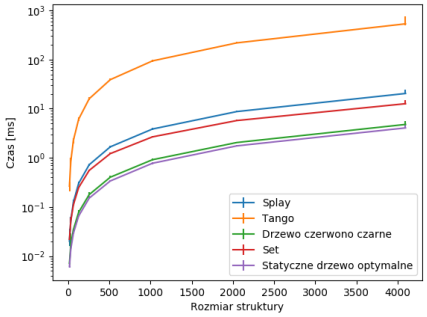
Zapytania są generowane przy pomocy symulacji przechodzenia po losowym drzewie nieskierowanym. Najpierw tworzone jest losowe drzewo przy pomocy kodów Prüfera i kolejne zapytania odpowiadają kolejnym wierzchołkom na losowej ścieżce (nie musi być do ścieżka właściwa).



Rysunek 5.10: Wykresy zależności czasu liczby zapytań przy danym rozmiarze struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 95% powtórzenia wierzchołka.



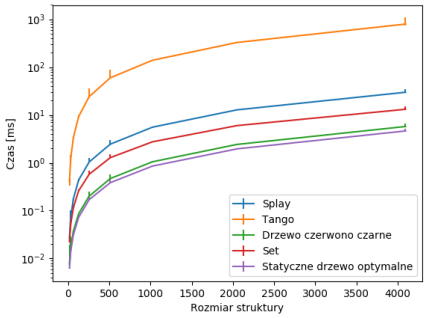
Rysunek 5.6: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 75% powtórzenia wierzchołka.



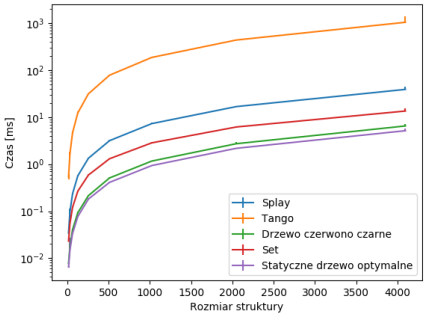
Rysunek 5.7: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 50% powtórzenia wierzchołka.

5.3. Testowanie wydajności wstawiania i usuwania elementów

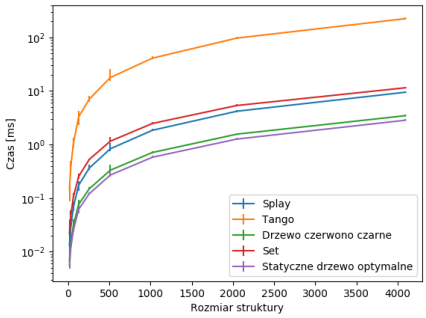
Porównamy czasy obsługi wstawiania i usuwania elementów dla drzew Splay, czerwono-czarnego i struktury set. W tym eksperymencie do struktur są wstawiane elementy $e_i \in [1, ..., n]$ w losowej kolejności. Mierzony jest sumaryczny czas realizacji wszystkich operacji insert. Następnie ze struktury usuwane są wszystkie elementy również w losowej kolejność, mierząc czas wykonania wszystkich operacji delete dla danej struktury. Eksperyment powtarzam dla rosnących geometrycznie danych $n = \{16, 32, 64, ..., 1024\}$ wykonując po 50 losowych prób.



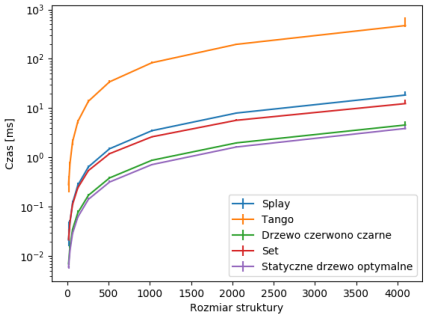
Rysunek 5.8: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 25% powtórzenia wierzchołka.



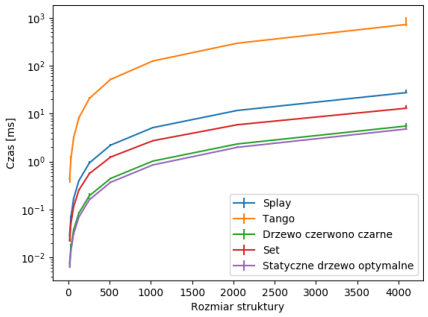
Rysunek 5.9: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 0% powtórzenia wierzchołka.



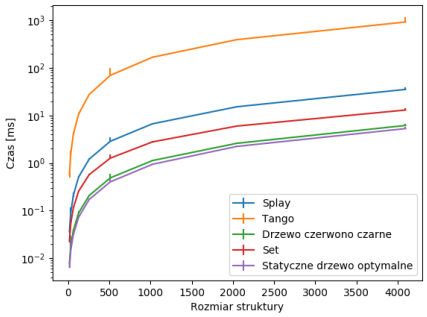
Rysunek 5.11: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 75% powtórzenia wierzchołka.



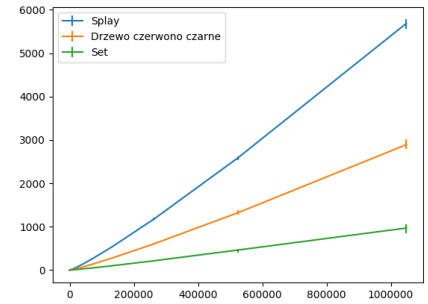
Rysunek 5.12: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 50% powtórzenia wierzchołka.



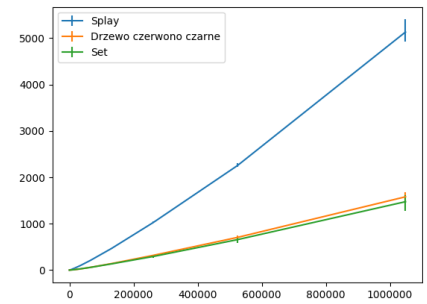
Rysunek 5.13: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 25% powtórzenia wierzchołka.



Rysunek 5.14: Wykresy zależności czasu od rozmiaru struktury, przy liczbie zapytań równej rozmiarowi struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 0% powtórzenia wierzchołka.



Rysunek 5.15: Wykresy zależności czasu wstawienia wszystkich elementów od liczby elementów.



Rysunek 5.16: Wykresy zależności czasu usunięcia wszystkich elementów od liczby elementów.

Bibliografia

[1] Thomas H. Cormen i in. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[2] E. Demaine i in. "Dynamic Optimality—Almost". W: *SIAM Journal on Computing* 37.1 (2007), s. 240–251. DOI: 10.1137/S0097539705447347. eprint: https://doi.org/10.1137/S0097539705447347. URL: https://doi.org/10.1137/S0097539705447347.

[3] Adriano M Garsia i Michelle L Wachs. "A new algorithm for minimum cost binary trees". W: *SIAM Journal on Computing* 6.4 (1977), s. 622–642.

[4] Donald E Knuth. "Optimum binary search trees". W: *Acta informatica* 1.1 (1971), s. 14–25.

[5] Kurt Mehlhorn. "Nearly optimal binary search trees". W: *Acta Informatica* 5.4 (1975), s. 287–295.

[6] Daniel Dominic Sleator i Robert Endre Tarjan. "Self-Adjusting Binary Search Trees". W: *J. ACM* 32.3 (1985), s. 652–686. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.

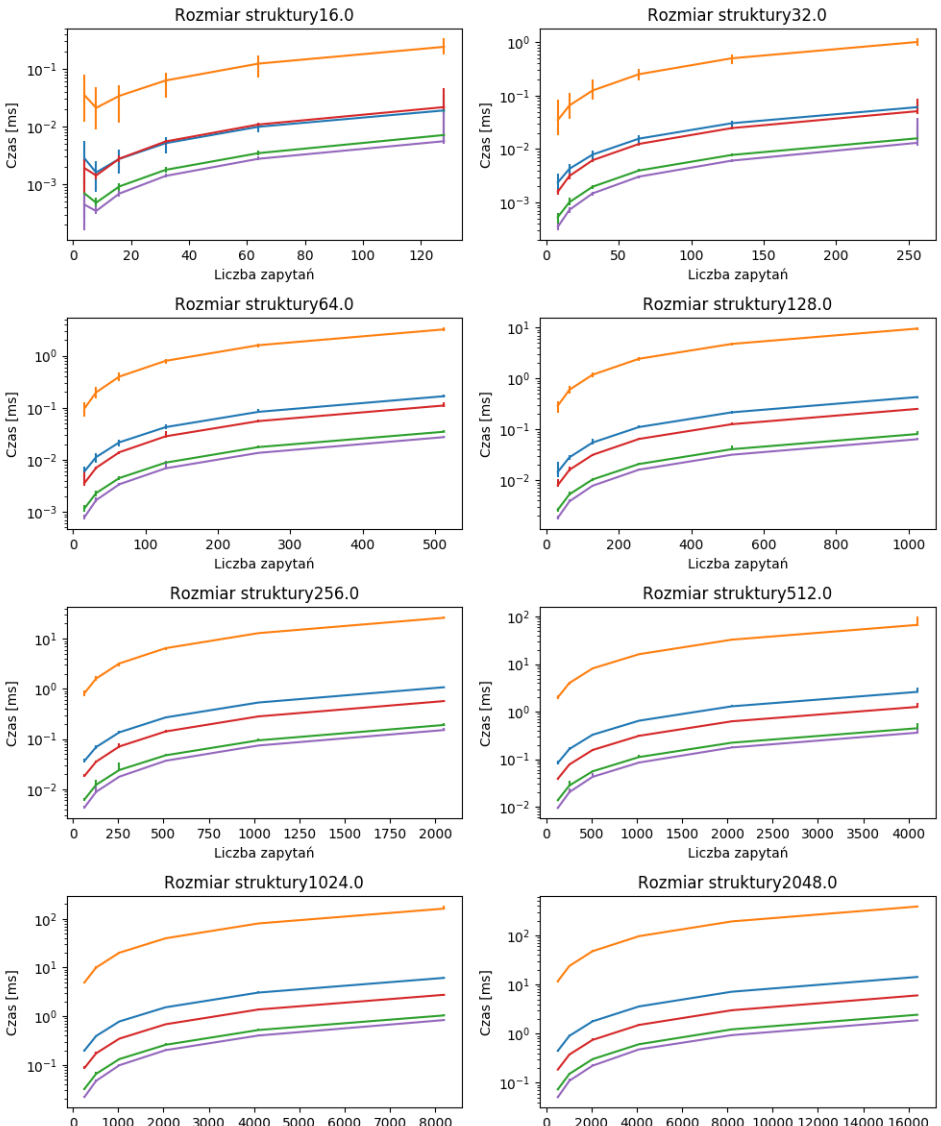
[7] Chengwen Chris Wang. *Multi-splay trees*. 2006.

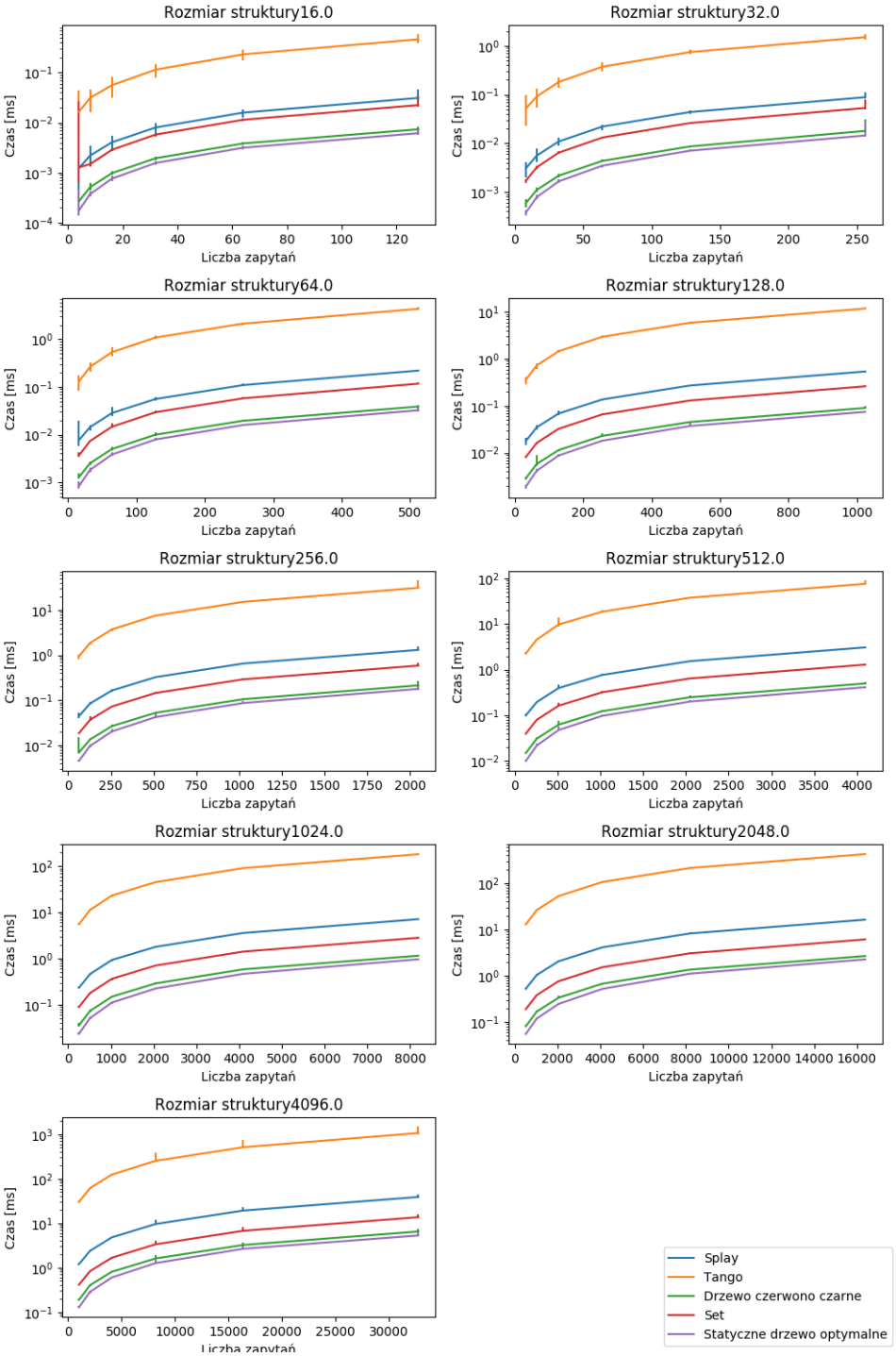
Dodatek A

Szczegółowe wyniki eksperymentów

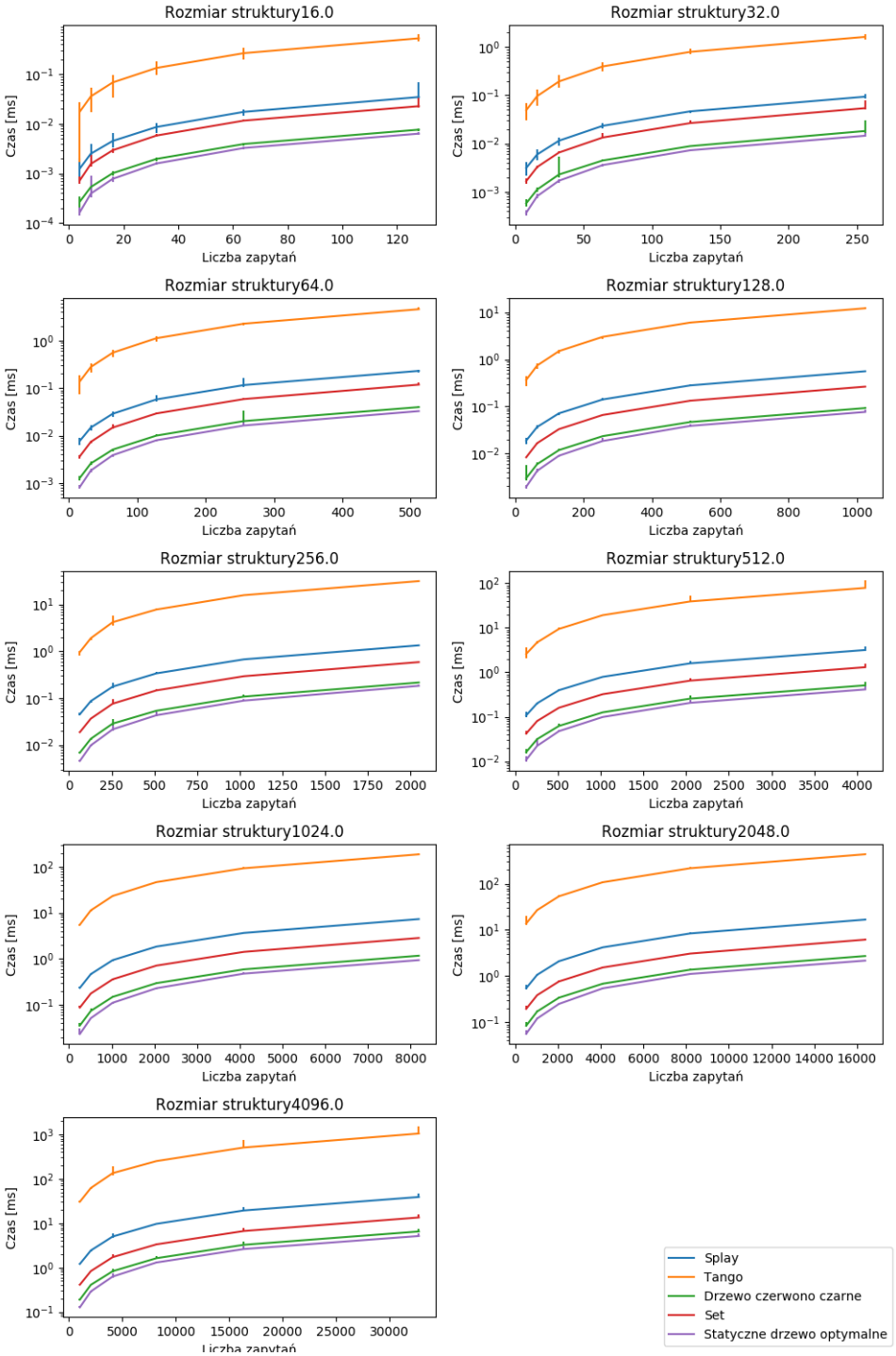
A.1. Testowanie dla problemu dynamicznej optymalności

A.1.1. Rozkład Gaussa

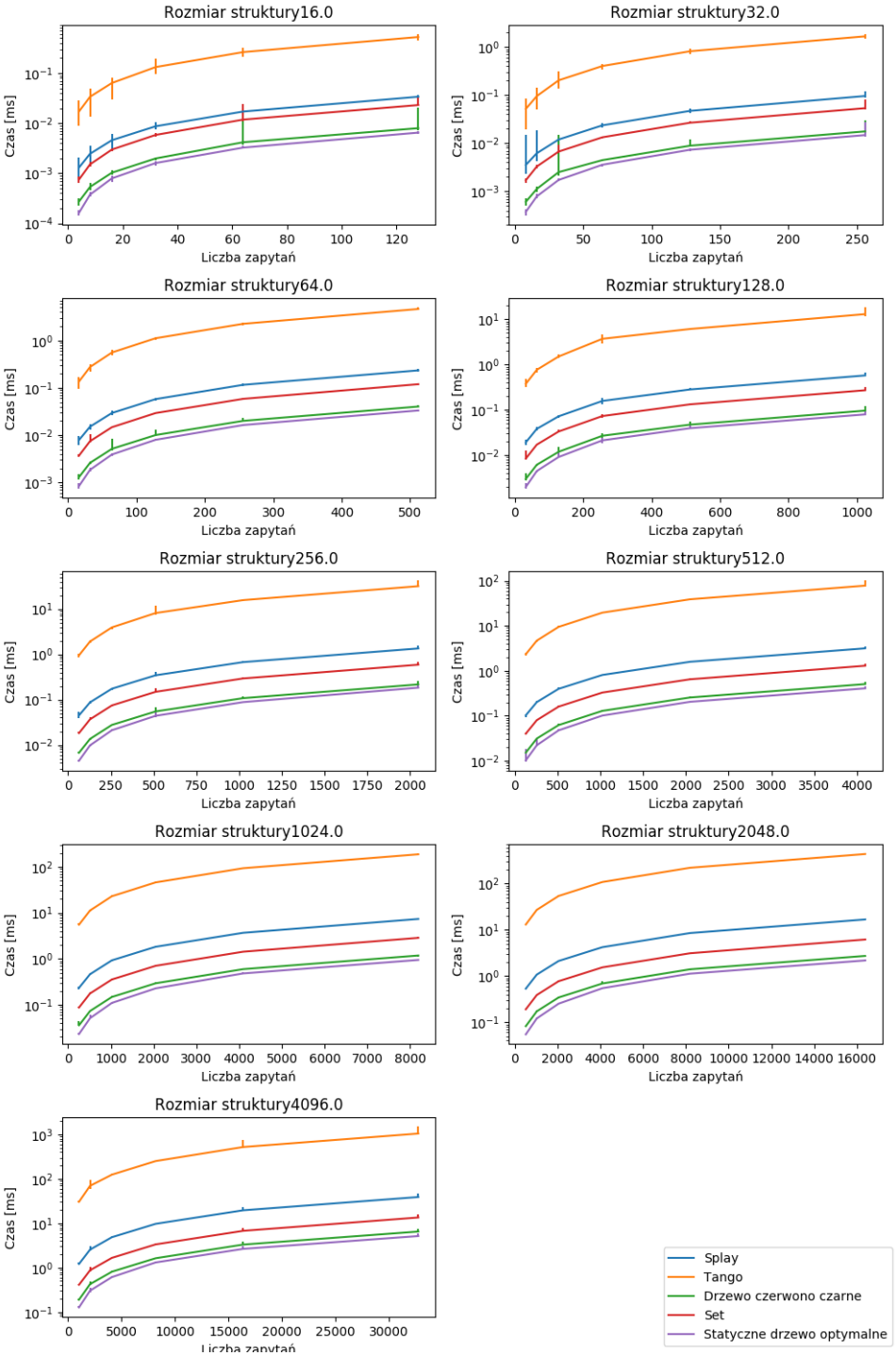




Rysunek A2: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{n}{4})$.

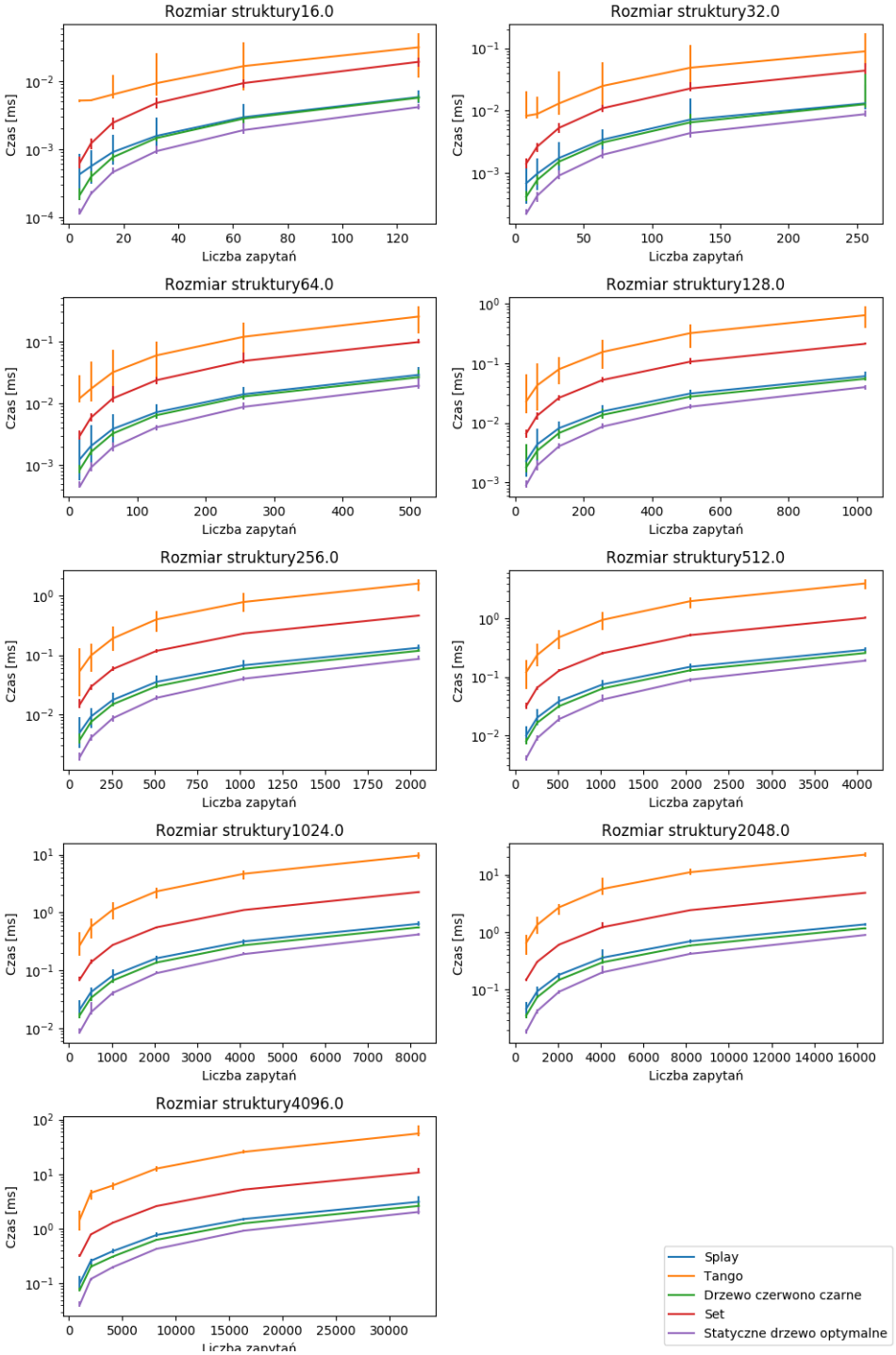


Rysunek A3: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{n}{2})$.

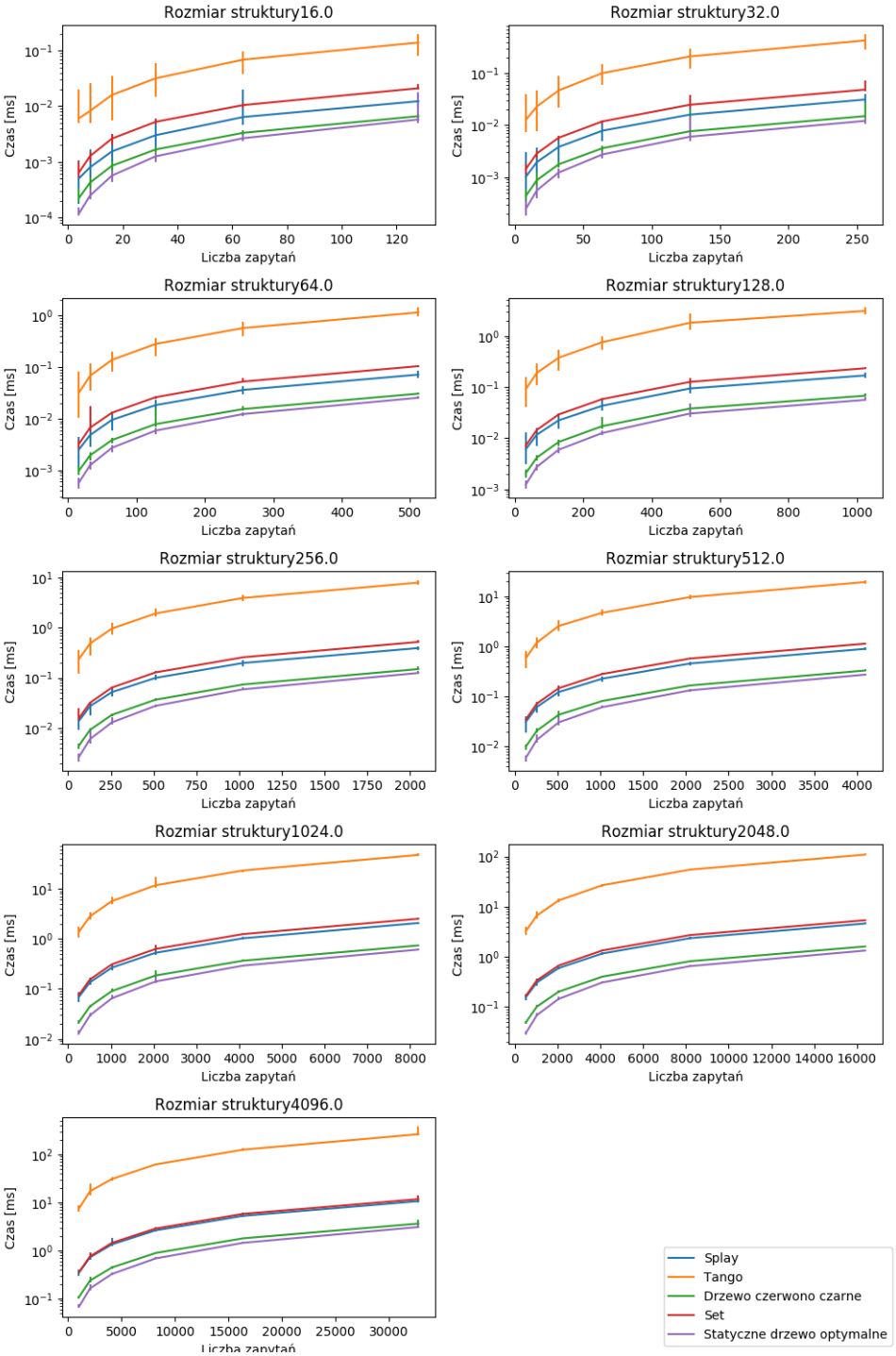


Rysunek A4: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem $\mathcal{N}(\frac{n}{2}, \frac{3n}{3})$.

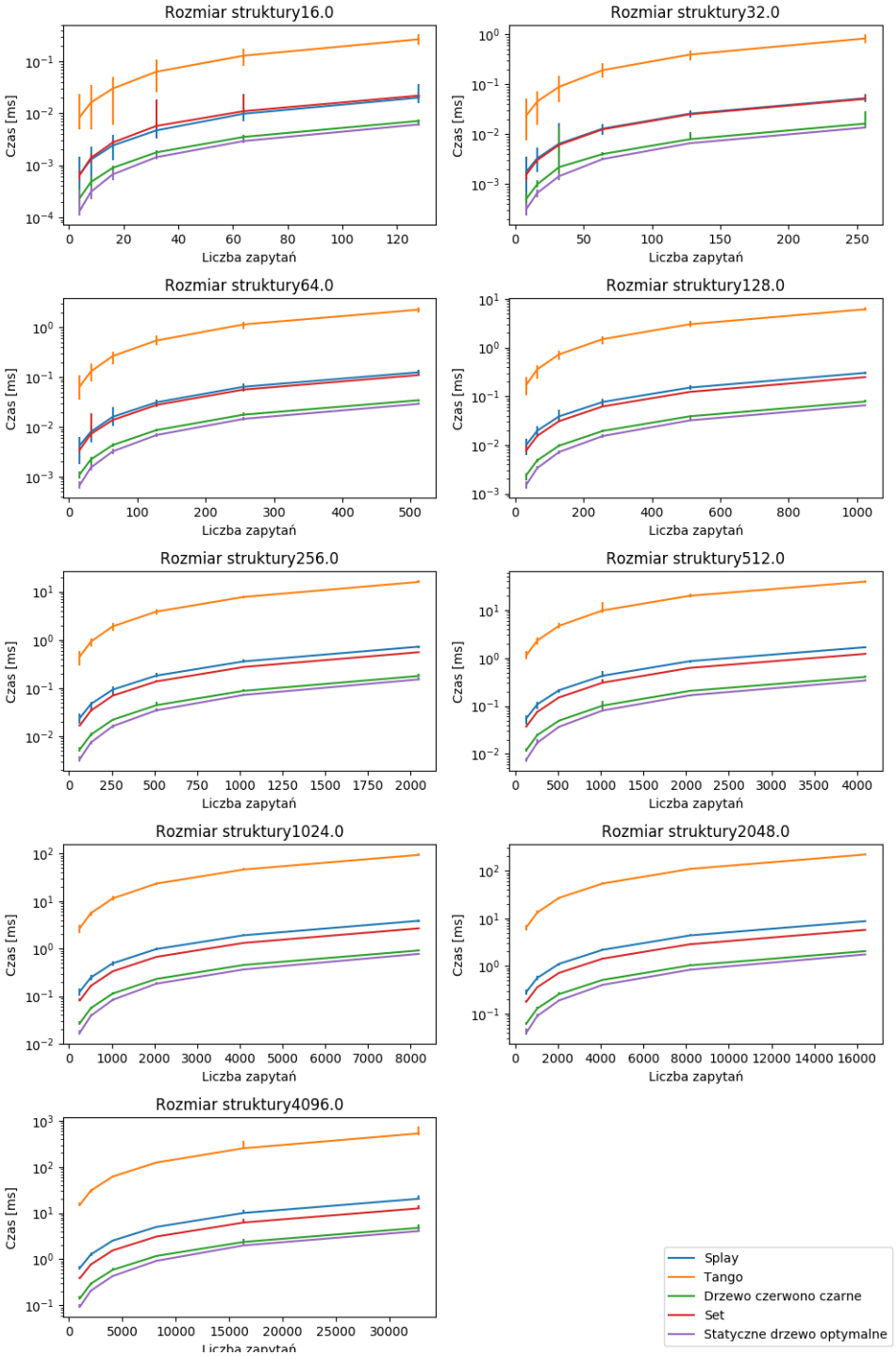
A.1.2. Rozkład jednostajny z prawdopodobieństwem powtórzenia wierzchołka



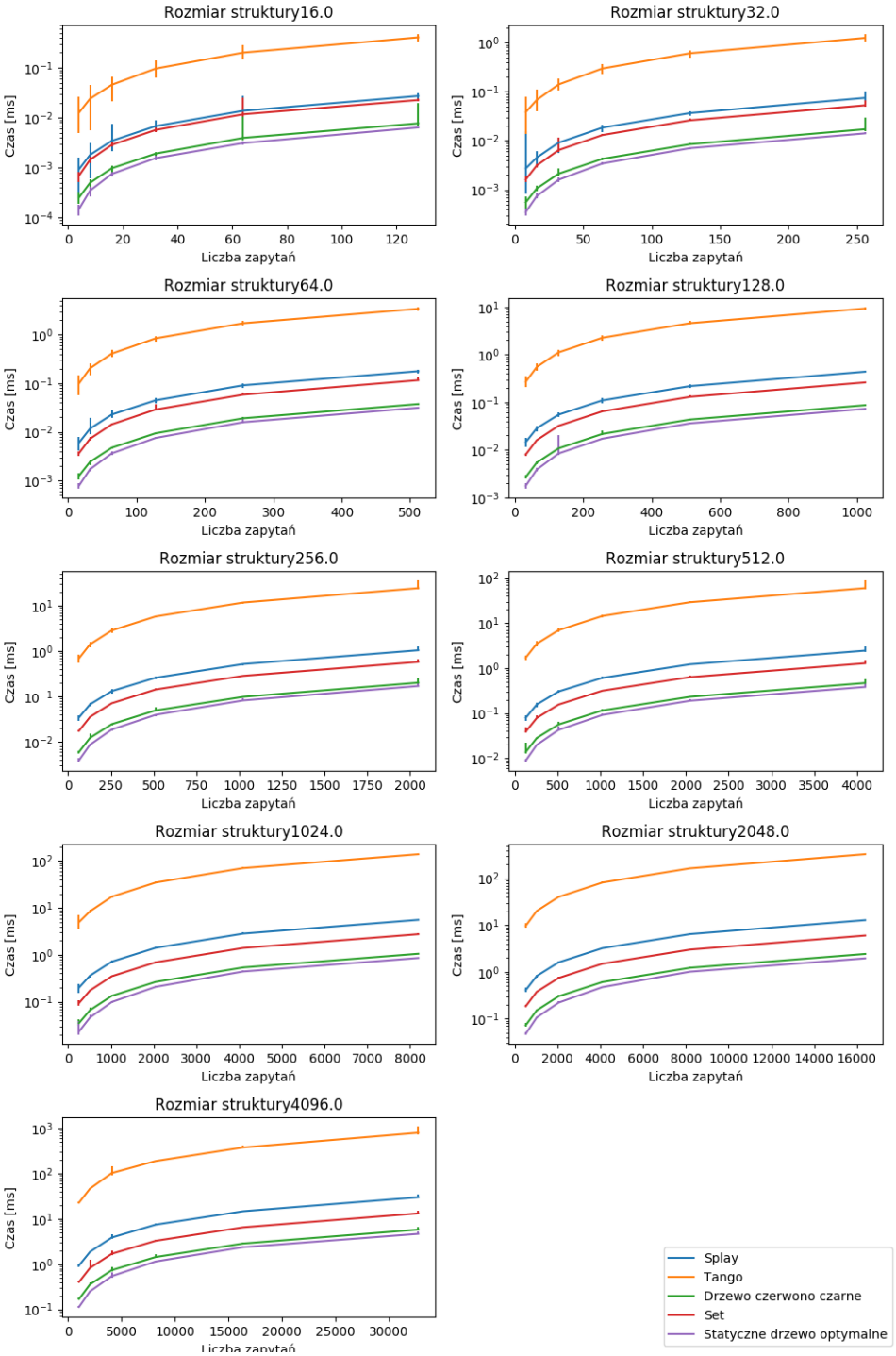
Rysunek A5: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 95% powtórzenia wierzchołka.



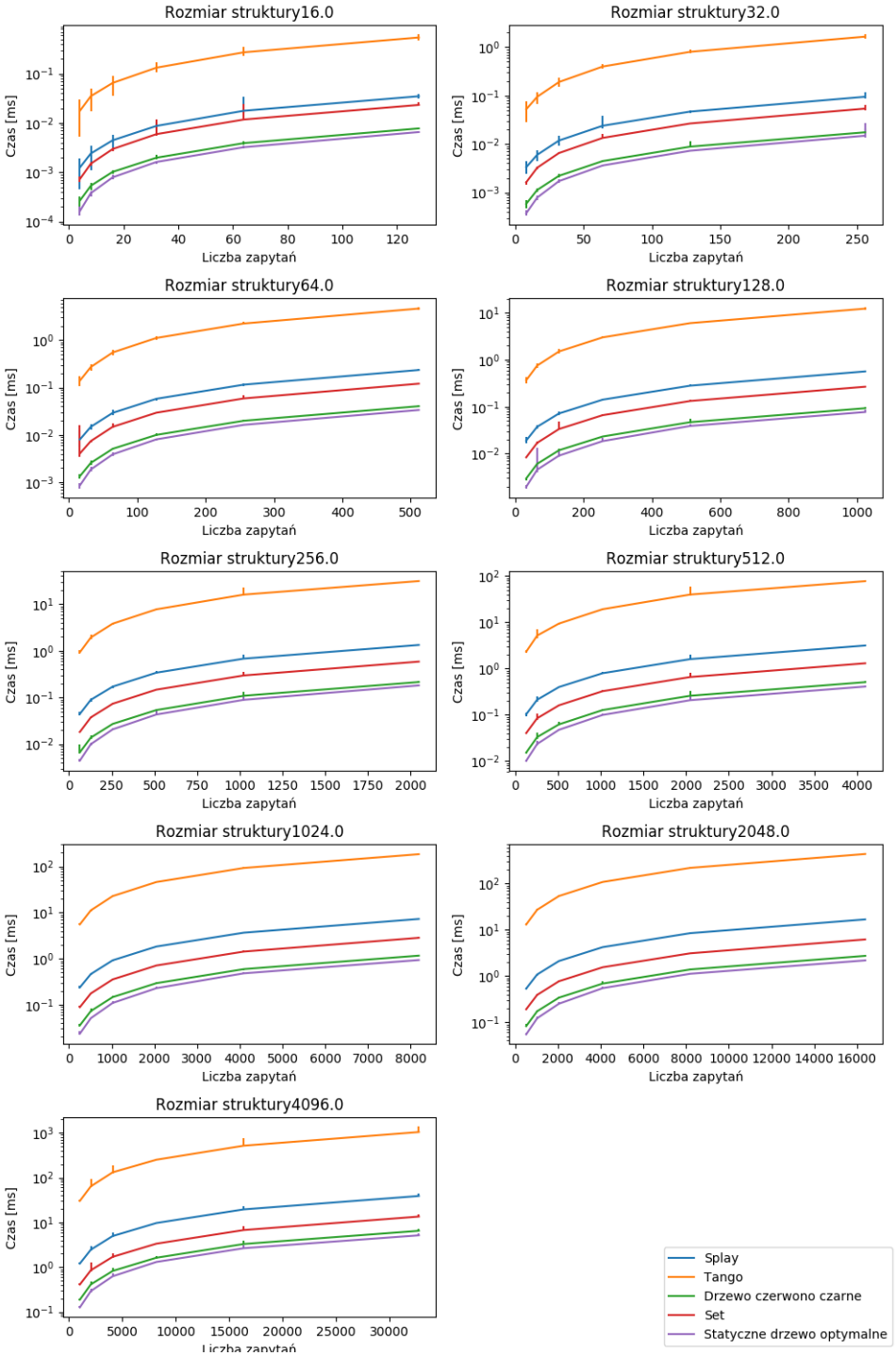
Rysunek A6: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 75% powtórzenia wierzchołka.



Rysunek A7: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 50% powtórzenia wierzchołka.

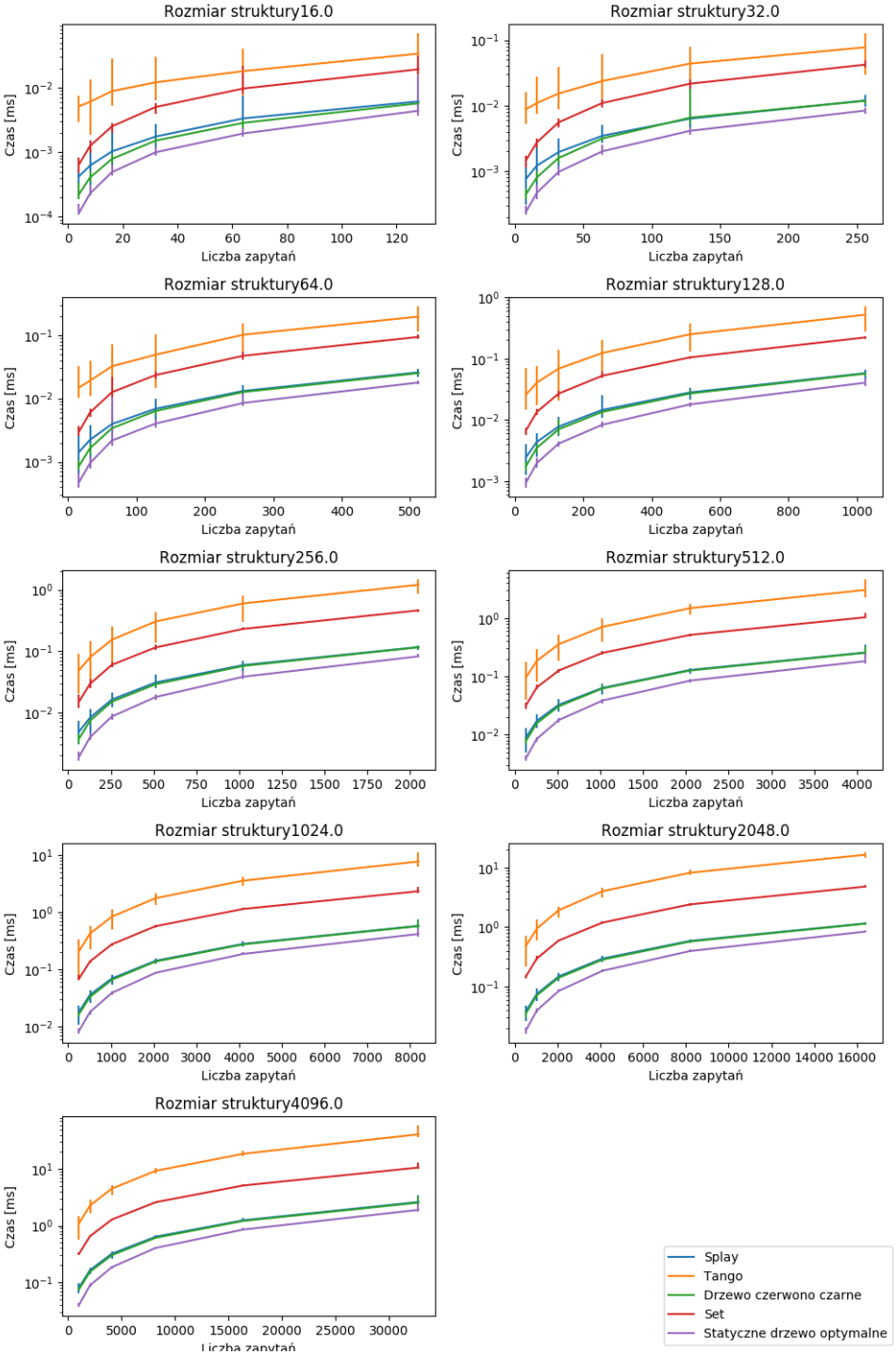


Rysunek A8: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 25% powtórzenia wierzchołka.

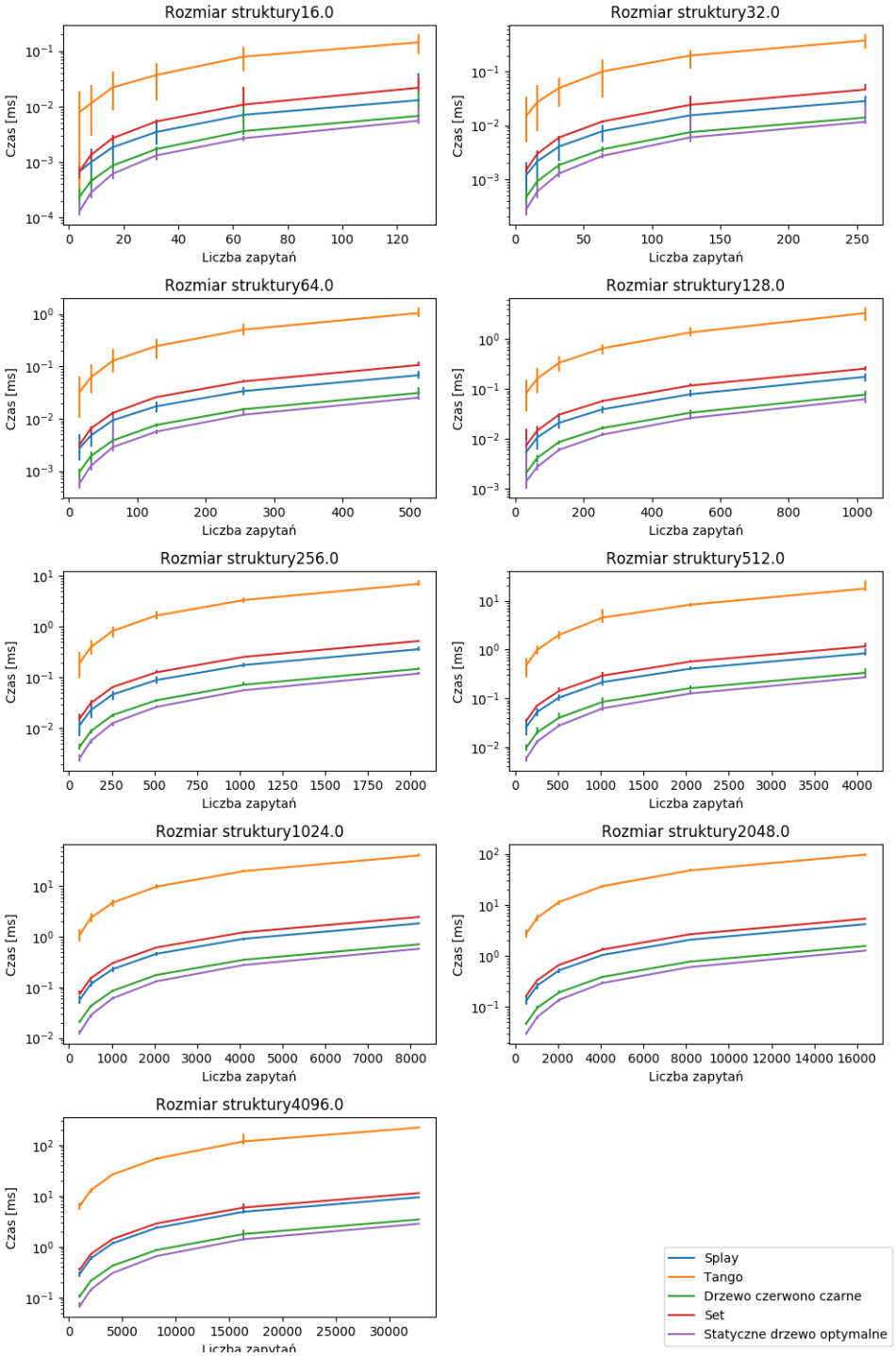


Rysunek A9: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane rozkładem jednostajnym z prawdopodobieństwem 0% powtórzenia wierzchołka.

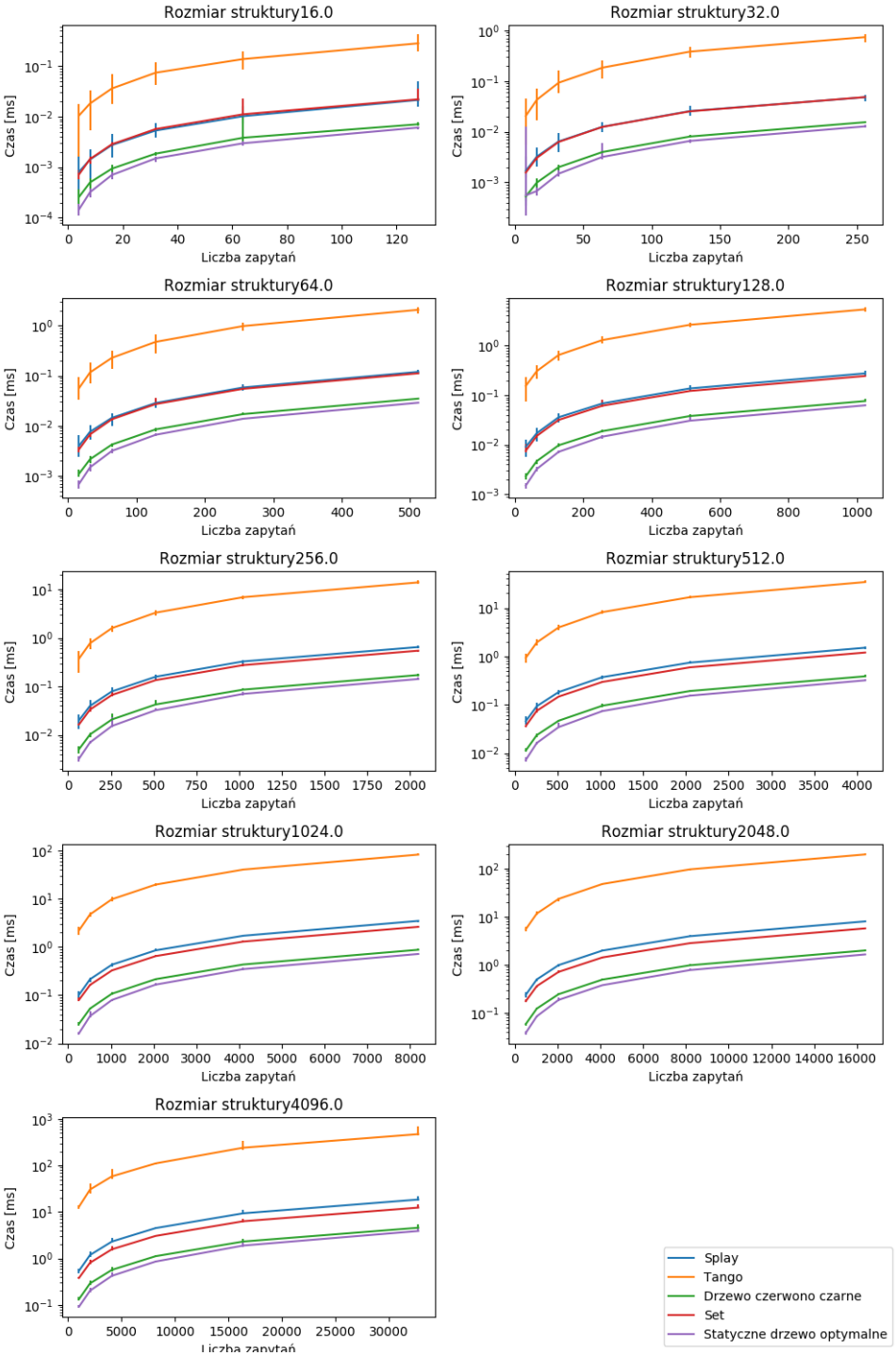
A.1.3. Losowanie ścieżki w grafie losowym z prawdopodobieństwem zmiany wierzchołka



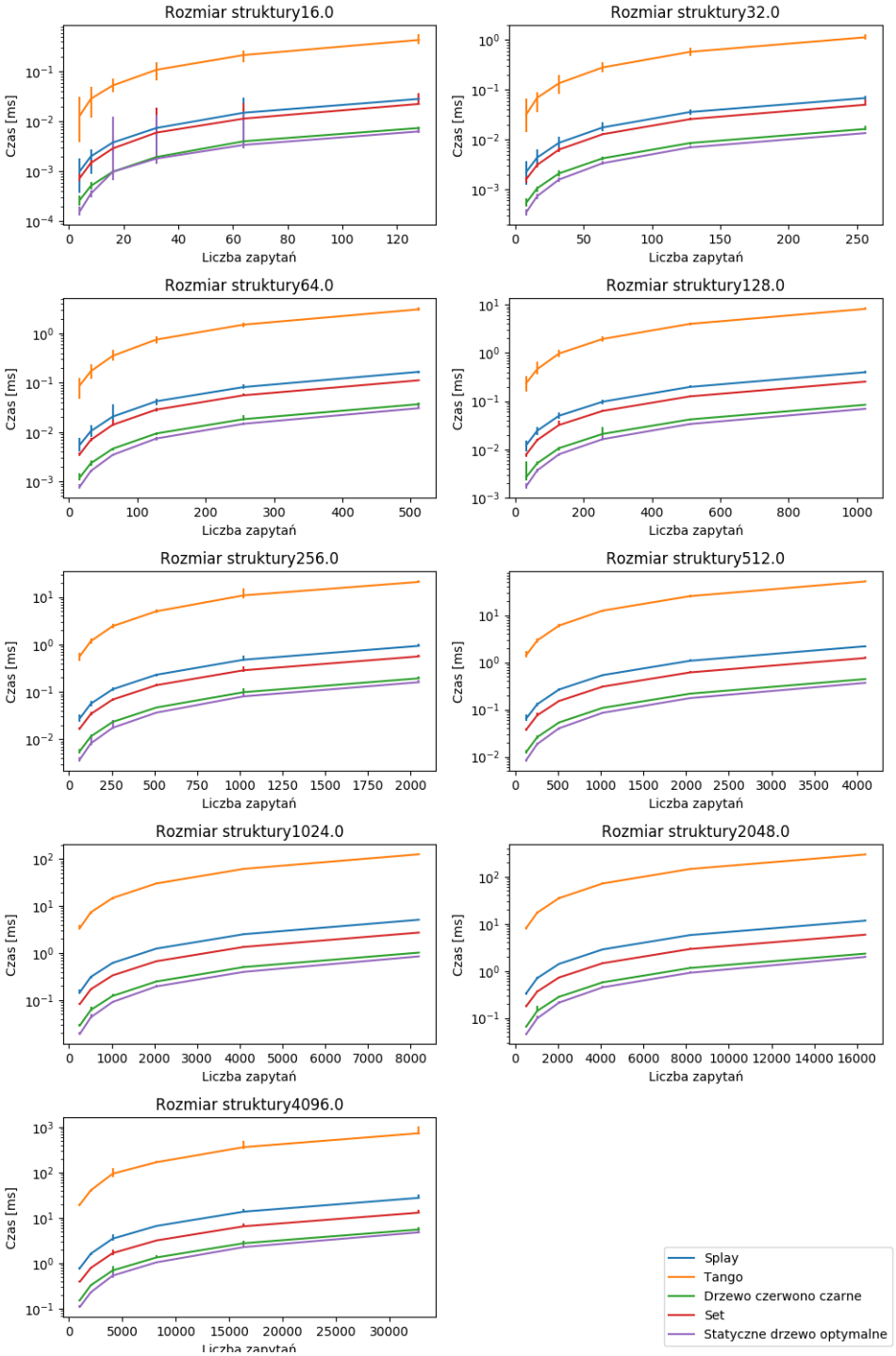
Rysunek A10: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 95% powtórzenia wierzchołka.



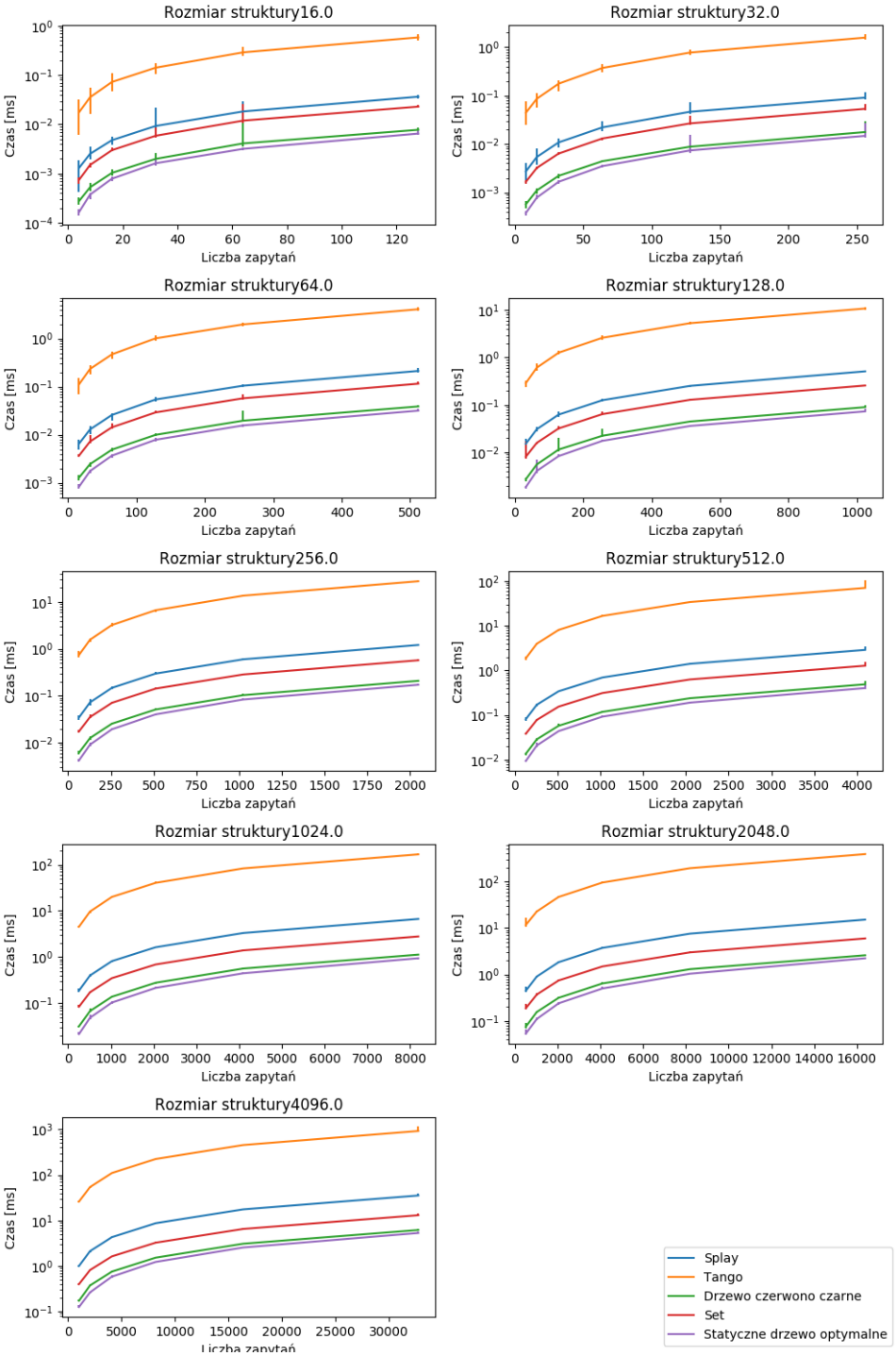
Rysunek A11: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 75% powtórzenia wierzchołka.



Rysunek A12: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 50% powtórzenia wierzchołka.



Rysunek A13: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 25% powtórzenia wierzchołka.



Rysunek A14: Wykresy zależności od czasu liczby zapytań przy danym rozmiarze struktury - dane generowane przez symulowanie przechodzenia po grafie losowym z prawdopodobieństwem 0% powtórzenia wierzchołka.