Politechnika Wrocławska	Architektura Komputerów 2		
	Detekcja znaku liczby w RNS		
	Izabella Juwa 252786 Patrycja Langkafel 252744	Projekt AK2/OiAK Dr inż. Piotr Patronik	Wrocław, 09.06.2021

Spis treści

1. Wprowadzenie

1.1. Wstęp teoretyczny

RNS (*ang.* residue number system) -system resztowy to system liczbowy służący do reprezentacji liczb całkowitych wektorem reszt z dzielenia względem ustalonego wektora wzajemnie względnie pierwszych modułów. Chińskie twierdzenie o resztach orzeka, że taka reprezentacja jest jednoznaczna dla liczb całkowitych ze zbioru $< 0, m)$, gdzie M jest iloczynem wszystkich modułów. Niech $B = \{m_1, \ldots, m_n\}$, będzie bazą względnie pierwszych modułów, a M ich iloczynem. Wtedy reprezentacją liczby X w systemie resztowym o bazie B jest $\{x_1, \ldots, x_n\}$, gdzie $x_i = X \, mod \, m_i$ dla każdego $1 \leq i \leq n$.

1.2. Sformułowanie problemu

Detekcja znaku w RNS należy do algorytmów trudnych w systemie resztowym. Niemniej jednak odgrywa kluczową rolę i znajduje zastosowanie w różnego rodzaju operacji dotyczących samego RNS (dzielenie i porównywanie liczb), jak i wielu innych zastosowaniach. Problem detekcji znaku w RNS był poruszany przez wielu autorów. W najprostszej wersji może być zrealizowany poprzez konwersję odwrotną i porównanie otrzymanej wartości z odpowiednią liczbą, zazwyczaj równą połowie zakresu dynamicznego. Główne metody porównania można sklasyfikować na te przekształcające liczbę w systemie RNS do systemu MRS w $O(n^2)$ krokach, które uznawane jest za najbardziej efektywne spośród głównych metod zaproponowanych przez Garnera. Jednakże według profesora Donalda Knutha istnieje prawdopodobieństwo na znalezienie lepszej, efektywniejszej metodzie niż ta opisana powyżej. Jego pomysłem było oszacowanie x/M zamiast x, używając Chińskiego Twierdzenia o resztach. Obie metody używają tablic, które stają się ogromne chyba, że liczba bitów bazy jest dostatecznie mała. Wielkość pamięci tego rozwiązania szacuje się na $0((log_2 M)^3/(log_2 log_2 M)^2)$ bitowo. W przedstawionym artykule został zaproponowany nowy efektywniejszy algorytm detekcji znaku. Autorzy użyli Chińskiego Twierdzenia o Reszcie do wprowadzenia równania szacującego wartości x względem m. Dzięki czemu możliwe jest uzyskanie złożoności komputerowej $O(n)$ ze złożonością pamięciową $0(log_2 M)^2)$. Zaproponowane przez nich dwa algorytmy efektywniej szacują dane równanie i wynik znaku w postaci bitu. Jeden z nich bazuje na szeregu potęgowym, gdzie drugi z algorytmów opiera się na tablicy odwrotności o skończonej długości. W przedstawionych rozwiązaniach szacowane błędy stają się coraz mniejsze, aż do momentu w którym możemy określić bit znaku. Gdy to się stanie nasz program kończy działanie. Skupiono się na rozwiązaniu, w którym wielkość zestawu modułów jest skalowalna i łatwa dla aplikacji. Głównym założeniem autorów jest zaproponowanie algorytmów, w których detekcja znaku możliwa jest dla podstawowych baz z większą efektywnością.

2. Notacja oraz baza

W celu rozpoczęcia detekcji znaku w resztowym systemie modularnym, należało określić bazę naszych działań. w będzie określało nam podaną ilość bitów danego słowa

$< x >_m = x \; mod \; m,$ wektor naszych reszt z dzielenia z przez m,

$$gdzie \;\; < x >_m \in < 0, m)$$

Bazą będziemy określać wektor $B = \{m_1, \dots, m_n\}$,

$$gdzie \; \gcd(m_i, m_j) = 1 (i \neq j),$$

przy czym m_i jest liczbą całkowitą z przedziału:

$$m_i = 2^w - \mu_i$$

$$M = \prod_{i=1}^{n} m_i$$

$$M_i = \frac{M}{m_i}$$

$\langle x^{-1} \rangle_{m_i}$, wektor odwrotności multiplikatywnej reszt z dzielenia x przez mi, pod warunkiem, że największy wspólny dzielnik musi się równać 1, co wynika z rozszerzonego algorytmu Euklidesa.

$$gcd(m_i, x) = 1$$

3. Określenie znaku

Jeżeli $\{x\}_B$ będzie stanowić reprezentacje RNS z przedziału $x \in < 0, M - 1 >$, funkcja znaku $\{x\}_B$, może być określona jako:

$$sign(\{x\}_B) = \begin{cases} 0, if \; x < \dfrac{M}{2} \\ 1, if \; x \geq \dfrac{M}{2} \end{cases}$$

4. Chińskie Twierdzenie o Resztach

CRT Jest to jedno z najważniejszych twierdzeń w teorii liczb i kryptografii. Jeśli dwie liczby mają te same reszty z dzielenia przez pewien zbiór modułów, to ich różnica musi być podzielna przez każdy z modułów (definicja kongruencji), a więc także przez najmniejszą wspólną wielokrotność modułów (iloczyn). Dwie różne liczby dające te same reszty dla danego zbioru modułów muszą więc być odległe o co najmniej iloczyn tych modułów. Dla CRT możemy przyjąć zbiór:

$$x = \langle \sum_{i=1}^{n} \langle x_i M_i^{-1} \rangle_{m_i} M_i \rangle_M \text{ , gdzie } x_i =< x >_{m_i}.$$

Jeżeli podzielimy obie strony przez M i podstawimy $\varepsilon_i = \langle x_i M_i^{-1} \rangle_{m_i}$, zyskamy

$$\frac{x}{M} = \langle \sum_{i=1}^{n} \frac{\varepsilon_i}{m_i} \rangle_1.$$

Posługując się $\{W\}_B$, możemy ε_i określić jako $[\varepsilon_1 \;\; \varepsilon_2 \;\; \dots \;\; \varepsilon_n] = \{x\}_B \otimes \{W\}_B$

5. Algorytm detekcji znaku

5.1. Sign detection using a power series (SDPS)

Algorytm ten został stworzony według pewnych zasad:

- Obliczenie μ_i^k, przed rozpoczęciem algorytmu, jako przeglądową tablicę dla $\frac{1}{m_i}$
- Rozpoczęcie obliczania od najbardziej znaczącej pozycji, która zawiera bit znaku.
- Skończenie działania jak tylko bit znaku zostanie określony.

```
//funkcja sing detection using a power series
int SDPS(int X, int n, int w, int mi[]) { //jako paramtery przekazujemy X-liczbę, //dla której liczymy znak, n-liczba liczb w bazie, w długość 2^w, mi to liczby odjęte //od 2^w, aby uzyskać bazę
	int* B = new int[n]; //wektor B={m1...mn}
	int w_pow = pow(2, w); // zmienna 2^w
	for (int i = 0; i < n; i++) {
		B[i] = w_pow - mi[i]; //2^w-mi jako elementy naszej bazy
	}
	int* M_i = new int[n]; //Mi=M/mi, gdzie M to zakres dynamiczny M={m1*m2*...mn}
	for (int i = 0; i < n; i++) {
		M_i[i] = 1;
		for (int k = 0; k < n; k++)
			if (k != i)
				M_i[i] *= B[k];
```

```
}
int M = 1;
for (int i = 0; i < n; i++)
        M *= B[i];  //zakres dynamiczny <-wymnożenie bazy
int* x = new int[n]; //liczba wejściowa
for (int i = 0; i < n; i++)
        x[i] = X % B[i]; //reszty z poszczególnych baz xMODmi
int* M_i_1 = new int[n];//liczba W
for (int i = 0; i < n; i++) {
        for (int k = 0; k < B[i]; k++) {
                if (M_i[i] * k % B[i] == 1) {//odwrotność multiplikatywna z dzielenia Mi, gdzie Mi=M/mi
                        M_i_1[i] = k; //stała wartość dla danej bazy
                        break;
                }
        }//dla każdej pozycji w bazie sprawdzamy wszystkie dostępne cyfry
}     //-------------właściwy algorytm----------------
int* eta = new int[n];
for (int i = 0; i < n; i++) {
        eta[i] = (x[i] * M_i_1[i]) % B[i];  //mnożenie modulo {x}_B*{W}_B
}
int gx_k = 0;
for (int i = 0; i < n; i++)
        gx_k += eta[i];//suma wszystkich et
int low = gx_k % w_pow;  //reszta z dzielenia z 2^w
int sign = low >> (w - 1); //przesunięcie bitowe   int sign=low/(w_pow/2)
if (!sign) {//jeśli sign=0
        low = low + w_pow / 2; //zwiększamy low o 2^(w-1)
}
int high = 0, z = 0, sum = 0, carry = 0; //deklaracja zmiennych
for (int k = 0; k < n; k++) {
        gx_k = 0;
        for (int i = 0; i < n; i++) {
                int mi_pow = pow(mi[i], k);  //wyliczenie gx_k    , mi^k
                gx_k += mi_pow * eta[i];//suma wektora mi^k i et
        }
        high = gx_k >> w; //przesunięcie o w bit
        z = high + low; //suma high i low
        sum = z % w_pow; //sum reszta z dzielenia
        carry = z >> w;  //sprawdzenie czy z się mieści w ciągu w bitów
        if (carry == 1 || sum != w_pow - 1) { //jeśli się nie mieści to mamy overflow
                sign = sign ^ carry;// sign XOR carry
                break;
        }
```

```
            low = gx_k % w_pow; //do low przypisanie reszty z dzielenia gx_k przez 2^w
        }
        return sign;//zwrócenie znaku
}
```

5.2. Sign detecion using reciprocal table (SDRT)

Można stwierdzić, iż algorytm SDRT jest bardzo podobny do wcześniej zaproponowanego algorytmu SDPS. Jednakże ma mniej restrykcji co do parametru μ_i. Baza przyjmuje formę jak poprzednio. Natomiast tablica odwrotności jest tworzona poprzez podzielenie odwrotności modułów reprezentowanych w systemie binarnym na ciąg słów $h_i(k)$

$$\frac{1}{m_i} = \sum_{k=1}^{\infty} h_i(k) \cdot 2^{-kw}$$

Podstawiając otrzymaną tablicę do $\frac{x}{M} = \langle \sum_{i=1}^{n} \frac{\varepsilon_i}{m_i} \rangle_1$, otrzymujemy postać :

$$\frac{x}{M} = \langle \sum_{i=1}^{n} \left(\varepsilon_i \sum_{k=1}^{\infty} h_i(k) \cdot 2^{-kw} \right) \rangle_1$$

```
//funkcja pomocnicza do liczenia tablic odwrotności
int* reciprocal_table(int m_i, int w, int len) { //tablica odwrotności, gdzie parametrem jest m_i=2^w-ui, w-ilosc bitow, len-jak duża tab ma być wygenerowana
            int* tab = new int[len];
            double reverse = 1.0 / m_i;  //  1/mi
            int w_pow = pow(2, w);   // 2^w
            for (int i = 0; i < len; i++) {
                        reverse *= w_pow; //(1/mi)*2^w
                        tab[i] = (int)reverse % w_pow;  //reszta z dzielenia przez 2^w
            }
      return tab; //zwrócenie kolejnych znaków dzielenia
}

int SDRT(int X) {

int n = 3, w = 5;
int mi[3] = { 3, 1, 0 };
int w_pow = pow(2, w);
int B[3];
for (int i = 0; i < n; i++)
            B[i] = w_pow - mi[i];
int x[3];
for (int i = 0; i < n; i++)
```

```
		x[i] = X % B[i];
	int h1, h2, h3;
	int eta[3];
	int M_i_1[3];
	int M_i[3];
	for (int i = 0; i < n; i++) {
		M_i[i] = 1;
		for (int k = 0; k < n; k++)
			if (k != i)
				M_i[i] *= B[k];
	}
	for (int i = 0; i < n; i++) {
		for (int k = 0; k < B[i]; k++) {
			if (M_i[i] * k % B[i] == 1) {
				M_i_1[i] = k;
				break;
			}
		}
	}
	for (int i = 0; i < n; i++) {
		eta[i] = (x[i] * M_i_1[i]) % B[i];
	}
	int** h = new int* [n]; //dwuwymiarowa tab h
	for (int i = 0; i < n; i++)
		h[i] = reciprocal_table(B[i], w, n + 3); //h1 stanowi tab odwrotności  o dł n+3
	h2 = 0;
	h3 = 0;
	for (int i = 0; i < n; i++) {
		h2 += eta[i]; //h2=sumie et
		h3 += eta[i] * h[i][1]; //eta *h_i(1)
	}
	int body, tail, sum, sign, carry;
	for (int k = 3; k < n + 3; k++) {
		h1 = h2;
		h2 = h3;
		h3 = 0;
		for (int i = 0; i < n; i++) //ponowne obliczenie h3
			h3 += h[i][k - 1] * eta[i]; //mnożenie wektorów, indeksujemy od 0, dlatego k-1
		if (k == 3) {
			body = h1 + (h2 >> w) + (h3 >> (2 * w));
			tail = body % 2; //tail jest parzystością body
			sum = body % w_pow; //reszta z dzielenia 2^w z body
			sign = sum >> w - 1;//najwyższy znaku sumy
```

```
                if (((sum % (w_pow / 2)) >> 1) != (w_pow / 4) - 1) //jeżeli reszta z dzielenia sumy przez 2^(w-1) przesunięte o 1 nie równa sie 2^(w-2) -1
                        return sign; //zwracamy znak
            }
            else {//w przeciwnym wypadku wracamy do pętli i dla body przypisujemy
                body = (h1 % w_pow) + ((h2 >> w) % w_pow) + ((h3 >> (2 * w)) % w_pow); //h1mod2^w + h2 przesunięcie bitowo o w modulo 2^w +h3 przesunięcie bitowo o 2w modulo 2^w
                int tmp = (body + tail * w_pow) >> 1; //body i tail(najwyższy znak) *2^w przesunięte o 1
                tail = body % 2; //nowy tail
                carry = tmp >> w;//tmp przesunite o w, czyli też podzielone przez 2^w
                sum = tmp % w_pow; //reszta z dzielenia z tmp przez 2^w
                if (carry == 1 || sum != w_pow - 1) { //sprawdzanie jeżeli carry jest równe 1 albo suma nie rowna sie 2^w-1
                        sign = sign ^ carry; //do znaku przypisujemy signXOR carry
                        return sign;//zwracamy znak
                }
            }
        }
        return sign;
}
```

6. Szacowane błędy

Przy mechanizmie wyznaczania znaku x z $< G(x,d) >_1$, poniższy diagram przedstawia nam wartość $< G(x,d) >_1$ obliczoną dla danego x. Punkty $P(x) = (x, < G(x,d) >_1)$ pojawiają się w pewnych obszarach pomiędzy liniami $y = (\frac{1}{M})x$ i $y = \left(\frac{1}{M}\right)x - e$. Ponadto pojawiają się również w trójkątnym obszarze powyżej $y = \left(\frac{1}{M}\right)x + (1-e)$. Tutaj symbol e jest uproszczonym wyrażeniem granicy błędu $e(d)$. Aby określić znak, najpierw obliczamy $< G(x,d) >_1$z $\{x\}_B$, a następnie określmy wartość według poniższych zasad:

$$< G(x,d) >_1 \in [0, \frac{1}{2} - e) \cup => sign(x) = 0,$$

$$< G(x,d) >_1 \in [\frac{1}{2}, 1 - e) => sign(x) = 1,$$

$$< G(x,d) >_1 \in [\frac{1}{2} - e, \frac{1}{2}) \cup [1 - e, 1) => indeterminate$$

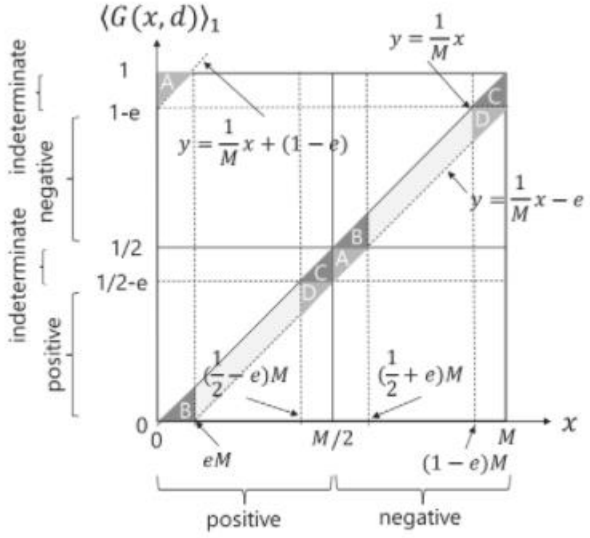
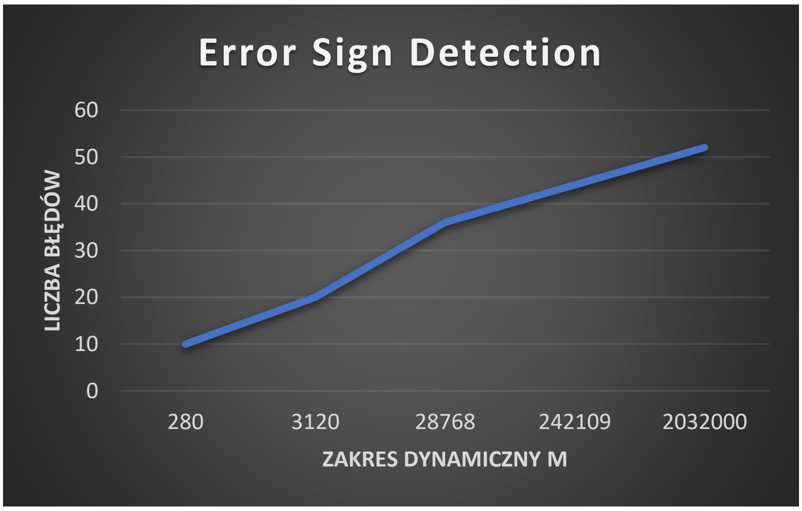
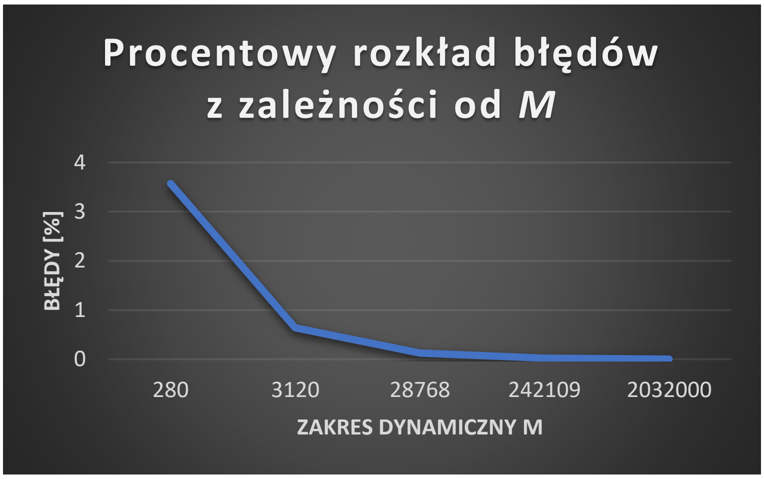


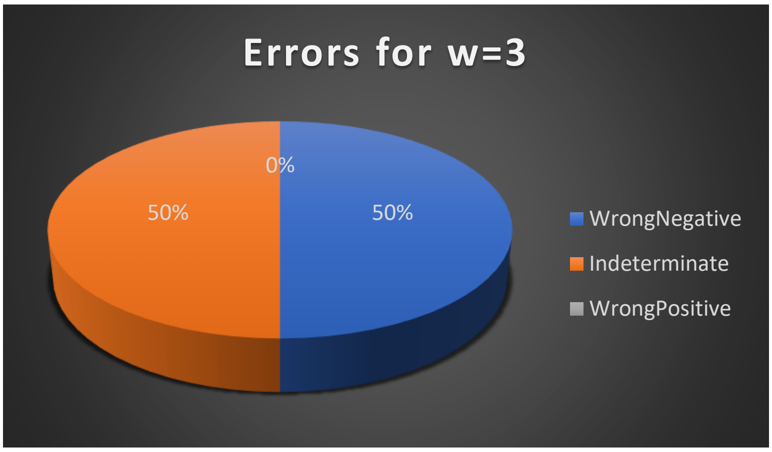
Figure 1: Obszary błędnej detekcji znaku

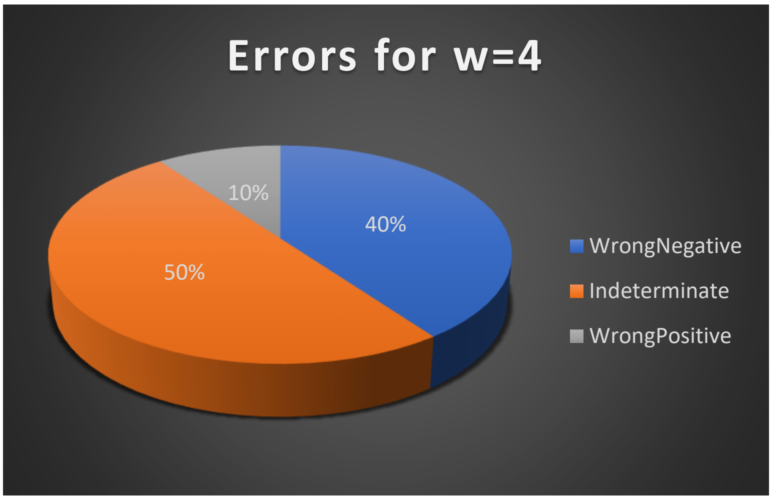
7. Analiza błędów

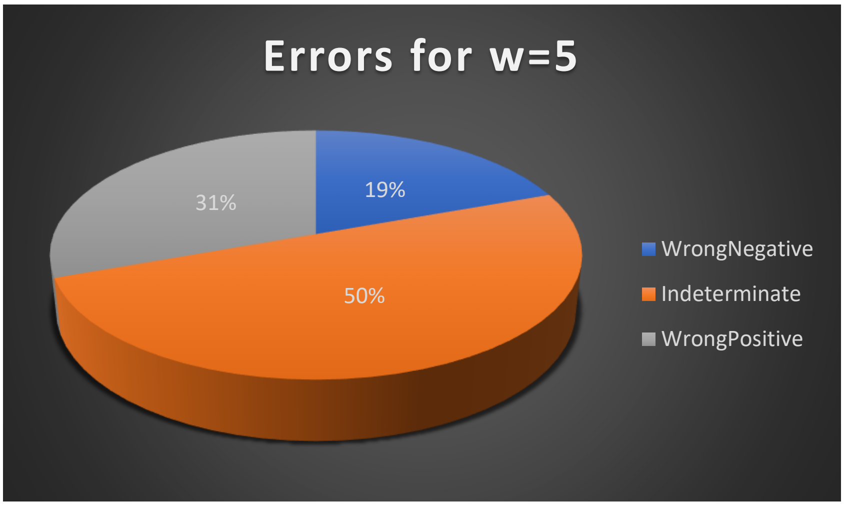
7.1. Liczba błędów a zakres dynamiczny



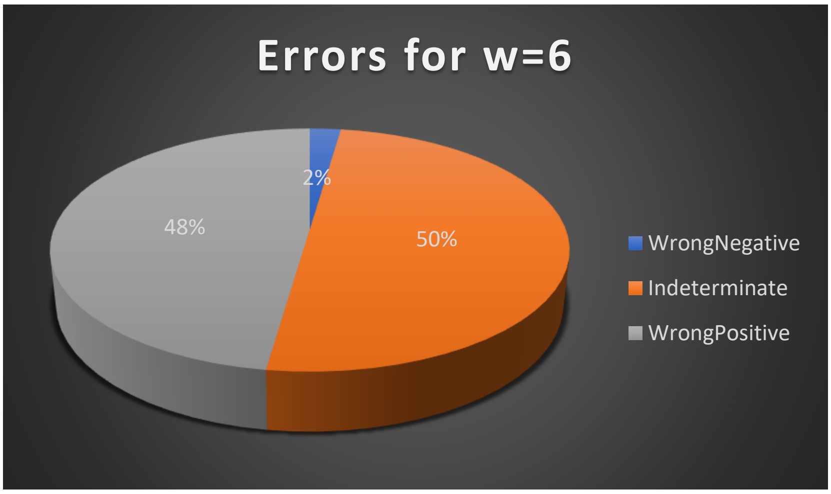

Procentowy rozkład błędów
z zależności od M
4
3
2
1
0
BŁĘDY [%]
280
3120
28768
242109
2032000
ZAKRES DYNAMICZNY M


Errors for w=3
0%
50%
50%
WrongNegative
Indeterminate
WrongPositive

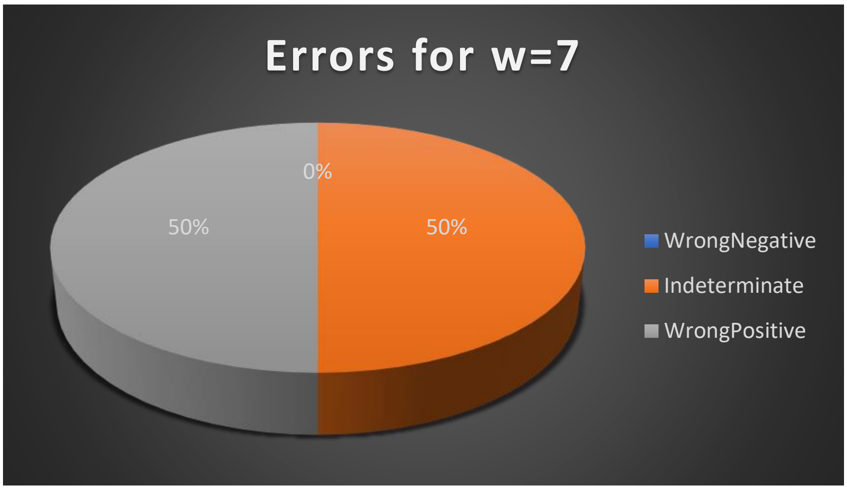

Errors for w=4
10%
40%
50%
WrongNegative
Indeterminate
WrongPositive



Errors for w=5
31%
19%
50%
WrongNegative
Indeterminate
WrongPositive



Errors for w=6
2%
48%
50%
WrongNegative
Indeterminate
WrongPositive



Errors for w=7
0%
50%
50%
WrongNegative
Indeterminate
WrongPositive

8. Podsumowanie oraz wnioski

Zgodnie z założeniami artykułu obszary złej detekcji znaku należą do skrajnych obszarów liczb z zakresu dynamicznego M. Jednakże w przypadku implementacji algorytmu SDRT bazującego na *reciprocal table*, czyli tablicy odwrotności, nie wystąpiło żadne przekłamanie. Dla wszystkich liczb z zakresu [0,M), algorytm wskazał poprawny bit znaku. Natomiast analizując algorytm wykorzystujący *power series*, czyli szereg naszych potęg, można zauważyć pewną zależność wystąpień błędnej detekcji znaku. Przekłamania rzeczywiście znajdują się w obszarach opisanych w powyższym diagramie. Warto jednak zaznaczyć, że wraz ze wzrostem liczby *w*, co wiąże się z zwiększonym zakresem dynamicznym, nasze przekłamania oczywiście zwiększają się liniowo. Biorąc jednakże pod uwagę liczbę błędów na cały zakres dynamiczny, procent złej detekcji znaku spada wraz ze wzrostem zakresu dynamicznego.

9. Literatura:

[1] *https://drive.google.com/file/d/1aMb_AQbGg1qnui1-8mPDSZOZ5Nnm_TOH/view*

[2] *https://www.csee.umbc.edu/~phatak/691a/rns-lnotes/phatak-jpdc-2016-rns-sd.pdf?fbclid=IwAR38QE2hoxkZAdEn97N6dbKp_A7K69oaFZCuw564ssh5-dmhGWwd6S2eh2A*

[3] *rozprawa_lin_v7_full_utf.dvi (dbc.wroc.pl)*