UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Szymon Rękawek
nr albumu: 206288

Gra Thuego

Praca magisterska na kierunku:

INFORMATYKA

Promotor:

prof. dr hab. T. Dzido

Gdańsk 01.01.2016

Streszczenie

Słowa kluczowe

Thue

Spis treści

Wprowadzenie

Tematem niniejszej pracy jest Gra Thuego. Axel Thue był norweskim matematykiem żyjącym w latach 1863 - 1922, znanym z prac z zakresu kombinatoryki.

Thue pracował nad problemami, powstałymi w wyniku badań nad sekwencjami symboli. Praca Thue [1] opisywała problem, który autor nazwał *nieredukowalne słowa* (*irreducible words*). Poświęca w niej szczególną uwagę dwu i trzy literowym przypadkom. W skrócie wprowadza pojęcie znane obecnie jako *Thue-Morse word* i pokazuje, że nieskończone słowa bez nasunięć (*Overlap-free*) są pochodnymi tej sekwencji. W swoich pracach definiuje kolejną strukturę, a mianowicie infinite *Square-Free word*, oraz przedstawia sposoby generowania nieskończenie długich słów wolnych zarówno od kwadratów jak i nasunięć.

Gra która powstała na podstawie teorii Thuego w skrócie polegała będzie na utworzeniu jak najdłuższego ciągu znaków nad określonym z góry alfabetem. Zależnie od trybu gry, kończyć się ona będzie w momencie gdy pojawi się zdefiniowany na początku rodzaj powtórzenia w tworzonym przez nas, bądź algorytm ciągu. Jednym z trybów gry jest walka komputera przeciwko niemu samemu, po takiej rozgrywce przedstawione zostaną złożoności czasowe oraz wnioski wynikające z obranej przez oponentów taktyki.

Ostatnia część pracy poświęcona jest analizie algorytmów zarówno pod względem czasu ich wykonywania jak i zdolności do przewidywania ruchów przeciwnika.

ROZDZIAŁ 1

Teorie Axela Thue na temat sekwencji symboli

Zanim przejdziemy do twierdzeń, koniecznym jest wyjaśnienie podstawowych twierdzeń, które w dalszej części pracy będą wielokrotnie wykorzystywane. Niżej wyjaśnione zostały sposoby generowania nieskończenie długich ciągów składających się z dwóch symboli, nie zawierających nasunięć. Po czym ukazane zostały metody tworzenia nieskończenie długich ciągów na trzech znakach, wolnych od kwadratów. Na koniec części teoretycznej przedstawiony został pomysł gry dla dwóch graczy wykorzystujący powyższe własności.

1.1. Definicje

- Alfabet jest skończonym zbiorem symboli lub liter.
- Słowo alfabetu A jest skończoną sekwencją elementów z A.
- Długość słowa ω jest reprezentowana przez $|\omega|$.
- Puste słowo o długości 0 jest reprezentowane przez ε.
- Czynnik (factor) słowa ω jest słowem u, które występuje wewnątrz ω formie $\omega = xuy$, podczas gdy x oraz y również są słowami tego alfabetu.
- Kwadrat (square) jest niepustym słowem w formie uu, gdzie u jest niepuste.
- Square-free, słowo jest wolne od kwadratów, jeśli żaden z jego czynników nie jest kwadratem.

- Nasunięcie (Overlap) jest słowem w formie $xuxux$, gdzie x jest niepuste. Nazwa pojęcia wzięła się z tego, że xux występuje dwa razy w $xuxux$. Pierwszy raz jako prefiks (początkowy czynnik) oraz jako sufiks (końcowy czynnik) i te dwa wystąpienia mają wspólną część - centralne x, a więc *nasuwają* się na siebie.

- Overlap-free - słowo w którym żaden z czynników nie nasuwa się na siebie. W definicji Axela Thue słowo ω w alfabecie długości n jest nieredukowalne jeśli jakiekolwiek dwa wystąpienia tego samego słowa jako czynnik wewnątrz ω są zawsze oddzielone od siebie przez $n-2$ liter. Oznacza to, że nieredukowalne dwuliterowe słowo jest bez nasunięć i nieredukowalne trzyliterowe słowo jest bez kwadratów.

- Morfizm - mapowanie obiektu z jednej matematycznej struktury w inną.

1.2. Thue-Morse word

Rozważmy nieskończone słowo

$$0110100110010110100101100110100 1\ldots$$

Zostało ono nazwane po Thue, który badał jego właściwości w referacie z 1906 roku, oraz Morsie, który odkrył je na nowo w latach 20 XIX wieku. Słowo Thue-Morse'a występuje również o wiele wcześniej w wiadomościach Prouheta[200] z Francuską Akademią Nauk w 1851 roku. Rzeczywiście Prouhet podał więcej ogólnych konstrukcji, uzyskując nie tylko słowo Thue-Morse'a, ale całą rodzinę słów na większych alfabetach mających inne interesujące właściwości. Słowa te czasami odnoszą się do ogólnych słow Thue-Morse'a lub słów Prouheta.

Niech $A = \{a, b\}$ będzie dwuliterowym alfabetem. Rozważmy morfizm μ z monoidu $A*$, który definiuje się następująco:

$$\mu(a) = ab, \qquad \mu(b) = ba$$

Dla $n \geq 0$:

$$u_n = \mu^n(a), \qquad v_n = \mu^n(b)$$

Wtedy:

$$\begin{aligned} u_0 &= a & v_0 &= b \\ u_1 &= ab & v_1 &= ba \\ u_2 &= abba & v_2 &= baab \\ u_3 &= abbabaab & v_3 &= baababba \end{aligned}$$

Wzór ogólny:

$$u_{n+1} = u_n v_n, \qquad v_{n+1} = v_n u_n$$

oraz:

$$u_n = \overline{v}_n, \qquad v_n = \overline{u}_n$$

gdzie $\overline{\omega}$ jest uzyskiwane z ω przez zamianę a oraz b. Słowa u_n i v_n są często nazywane *Blokami morsa*. Można łatwo zauważyc że u_{2n} oraz v_{2n} są palindromami oraz to że $u_{2n+1} = \neg v_{2n+1}$, gdzie $\neg w$ jest negacją w. Morfizm μ może być rozszerzony do nieskończonych słow, które mają dwa stałe punkty:

$$t = abbabaabbaababbabaab... = \mu(t)$$
$$t = baababbaabbabaababba... = \mu(\ t)$$

Przedstawione powyżej słowo t jest sekwencją Thue-Morse'a. Jest wiele innych sposobów na stworzenie tego słowa. Niech t_n będzie n-tym symbolem w t, zaczynając od $n = 0$. Wtedy można pokazać, że:

$$t_n = \left\{ \begin{array}{l} a\ if\ d_1(n) \equiv (mod\ 2) \\ a\ if\ d_1(n) \equiv (mod\ 2) \end{array} \right\}$$

gdzie $d_1(n)$ jest liczbą bitów równych 1 w binarnej reprezentacji n.

Dla $n \leq 12$ oraz $n \in \mathbb{N}$ generowane jest następujące słowo:

$$\begin{aligned} bin(0) &= 0, & d_1(0) &= 0\ mod\ 2 = 0 \rightarrow a \\ bin(1) &= 1, & d_1(1) &= 1\ mod\ 2 = 1 \rightarrow b \\ bin(2) &= 10, & d_1(2) &= 1\ mod\ 2 = 1 \rightarrow b \\ bin(3) &= 11, & d_1(3) &= 2\ mod\ 2 = 0 \rightarrow a \end{aligned}$$

$$
\begin{aligned}
bin(4) &= 100, & d_1(4) &= 1\ mod\ 2 = 1 \to b\\
bin(5) &= 101, & d_1(5) &= 2\ mod\ 2 = 0 \to a\\
bin(6) &= 110, & d_1(6) &= 2\ mod\ 2 = 0 \to a\\
bin(7) &= 111, & d_1(7) &= 3\ mod\ 2 = 1 \to b\\
bin(8) &= 1000, & d_1(8) &= 1\ mod\ 2 = 1 \to b\\
bin(9) &= 1001, & d_1(9) &= 2\ mod\ 2 = 0 \to a\\
bin(10) &= 1010, & d_1(10) &= 2\ mod\ 2 = 0 \to a\\
bin(11) &= 1011, & d_1(11) &= 3\ mod\ 2 = 1 \to b\\
bin(12) &= 1100, & d_1(12) &= 2\ mod\ 2 = 0 \to a
\end{aligned}
$$

$$t = abbabaabbaaba$$

W konsekwencji istnieje skończony automat obliczający wartości t_n. Automat ten ma dwa stany końcowe 0 oraz 1. Na początku czyta łańcuch znaków $bin(n)$ od lewej do prawej, zaczynając od $n = 0$. Ostateczny stan równy jest 0 lub 1 i definiuje czy t_n jest równe a lub b. W skrócie obliczenie jakie wykonuje automat to $d_1(n)$ $modulo$ 2.

1.3. Square-free word

Łatwo można zauważyc, że jedynymi słowami bez kwadratów w alfabecie $A = \{a, b\}$ są: a, b, ab, ba, aba, bab. Istnieje jednak dowolnie długi ciąg znaków wolny od kwadratów dla słów nad alfabetem trzyliterowym. By stworzyć dowolne słowo wolne od kwadratów Thue wymyślił następujący algorytm.

Mając alfabet $A = \{a, b, c\}$ należy zastąpić każde wystąpienie litery a przez $abac$, b przez $babc$ oraz c przez $bcac$, jeśli jest poprzedzone przez a lub $acbc$, jeśli jest poprzedzone przez b. Zaczynając od litery a otrzymujemy nieskończone słowo które nie zawiera kwadratów.

$$abacbabcabacbcacbabcabacbabcacbcabacbabc...$$

W 1912 roku Axel Thue wymyślił inny sposób na generowanie nieskończonego słowa bez kwadratów na trzech literach z użyciem następującego morfizmu.

- $a \to abcab$
- $b \to acabcb$
- $c \to acbcacb$

Po raz kolejny zastępujemy każde wystąpienie z naszych liter przez zdefiniowane sekwencje. Jest to dość skomplikowana struktura, zsumowana długość łańcuchów wynosi 18. A Carpi [7] dowiódł, że morfizm na alfabecie składającym się z trzech liter tworzący słowa wolne od kwadratów musi mieć długość równą co najmniej 18.

1.4. Thue online

Praca J. Grytczuka, P. Szafrugi i M. Zmarza pod tytułem Online version of theorem of Thue[8] opisuje wersję online teorii Thuego. Jest to gra dla dwóch graczy Boba oraz Alicji. Podczas rozgrywki Alicja i Bob naprzemiennie wykonując swoje ruchy tworzą ciąg bez kwadratów. Celem Boba jest jak najszybsze skończenie rozgrywki poprzez utworzenie kwadratu, Alicja natomiast musi tego unikać.

Wartym wspomnienia jest uproszczony tryb gry, podczas którego mamy dwóch graczy - Alicję i Boba, tak jak zostało wspomniane, tylko Alicji zależy na tym by uniknąć kwadratów. Rozgrywka polega na tym, że Alicja i Bob wybierają na przemian symbole z ustalonego zbioru A, oraz dopisują je na końcu istniejącego ciągu. W momencie, gdy pojawia się kwadrat aa, jego druga część, czyli w naszym przypadku prawe a, zostaje usunięte. Zostało udowodnione w [9] J. Grytczuk, J. Kozik, P. Micek, New approach to nonrepetitive sequences. Random Structures Algorithms, DOI 10.1002/rsa.20411. , że Alicja jest wstanie stworzyć dowolnie długi ciąg bez kwadratów, nie zważając na ruchy Boba. Powyższe jest jednak możliwe pod warunkiem, że moc zbioru A wynosi conajmniej 8.

Innym typem gry przedstawionym w pracy[8] jest Online Thue game. Po raz kolejny gracze wykonują swoje ruchy na przemian. W swojej rundzie Bob wybiera indeks w istniejącym ciągu S, który jest sprecyzowany przez liczbę $i \in \{0, 1, ..., n\}$, następnie Alicja wybiera symbol $x \in A$, który jest wstawiany jedną pozycję w prawo od s_i, dając nam nową sekwencję $S' = s_1, ..., s_i, x, s_{i+1}, ..., s_n$ w momencie gdy $i = 0$, x jest ustawiany na początku S. Celem Boba jest zmuszenie Alicji do stworzenia kwadratu, podczas gdy Alicja unika tego najdłużej jak to możliwe. Na przykład, jeśli ustalimy, że $A = \{a, b, c\}$ i $S = acbc$, wtedy Bob wybierając indeks $i = 1$ nie daje Alicji możliwości wybrania symbolu, który nie stworzyłby kwadratu. Rzeczywiście wybierając jakikolwiek $x \in A$ doprowadza do utworzenia kwadratu w S': **aa**cb, a**bcbc**, a**cc**bc. W przypadku gdy Bob obierze dobrą strategię, rozgrywka na 3 symbolach skończy się na ciągu o długości ≤ 5, nie ważne jak dobrą strategię obierze Alicja. Oczywiście im większą moc ma zbiór A, tym więcej ruchów, będzie potrzebował Bob, by zakończyć rozgrywkę.

[9] A. Ku¨ndgen and M. J. Pelsmajer, Nonrepetitive colorings of graphs of bounded treewidth, Discrete Math. 308 (2008), 4473–4478. [10] J. Bara ´t, P. P. Varju´. On square-free vertex colorings of graphs. Studia Sci. Math. Hungar. 44 (2007) 411–422.

Grytczuk, Szafruga i Zmarza[8] wysnuli teorię, że istnieje strategia dla Alicji gwarantująca jej utworzenie dowolnie długiej gry w online Thue game na zbiorze o 12 symboli. Teorię swoją oparli o prace Ku¨ndegena i Pelsmajera[9], oraz Baráta and Varju[10], na temat niepowtarzalnych kolorowań grafów planarnych (outerplanar graphs?).

Autorzy [8] nie zamkneli do końca problemu i zauważyli, że może istnieć strategia dla Alicji, która pozwoliłaby jej na dowolnie długą rozgrywkę nawet przy mocy zbioru A równej 9.

1.5. Komputerowa implementacja Online Thue Game

Komputerowa implementacja gry Thuego opiera się na pomyśle gry z pracy [8]. W grze dostępnych jest kilka trybów zarówno dla jednego oraz dwóch graczy jak i komputerowa symulacja, czyli gra komputera przeciwko niemu samemu.

Implementacja gry Online Thue Game nazywana będzie **Longest Square-Free word**. Zasady pozostają niemal identyczne. Na początku gry, gracze ustalają moc zbioru symboli, oraz otrzymują swoje role. Jeden z nich staje się architektem, drugi malarzem. Rola architekta polega na wybieraniu odpowiedniego indeksu w ciągu tworzonym przez graczy, pod którym powstanie nowy element. Malarz natomiast określa symbol wstawianego elementu. Gra kończy się w momencie, gdy w ciągu tworzonym przez graczy pojawia się **kwadrat**. Indeks i podawany przez architekta nie może być mniejszy od zera oraz większy niż n, gdzie n jest równe liczbie elementów w ciągu. Na początku gry ciąg S zawiera tylko jeden symbol równy 0. Grę rozpoczyna architekt. Gracze wykonują swoje ruchy na przemian. Malarz otrzymuje punkt za każdy pomalowany element, który nie tworzy **kwadratu** wewnątrz ciągu. By wyłonić zwycięzcę potrzebne są dwie rundy. Każdy z graczy musi sprawdzić się zarówno jako malarz i architekt. Wygrywa osoba, która zdobyła więcej punktów jako malarz.

Tryb ten dostępny jest również dla jednego gracza, rolę przeciwnika otrzymuje wtedy komputer, który działa według algorytmu przewidującego określoną przez poziom trudności liczbę ruchów do przodu. Algorytm może pełnić zarówno rolę budowniczego jak i malarza.

Bliźniaczym trybem gry, opierającym się na tych samych zasadach z niewielką różnicą jest **Longest Overlap-Free word**. Różnica polega na tym, że malarz w tworzonym ciągu musi unikać **nasunięcia**.

Podobnie jak w **Longest Square-Free word** jest możliwość gry przeciwko algorytmowi, który jest w stanie przewidywać określoną ilość ruchów do przodu.

ROZDZIAŁ 2

Aplikacja Longest free word

Program, który powstał na bazie gier Longest Square-free word oraz Longest Overlap-free word, został napisany w Javie. Umożliwia on rozgrywkę w obu wersjach gry, zarówno z drugim graczem, jak i z komputerem, oraz jest w stanie zasymulować rozgrywkę dwóch graczy komputerowych, grających przeciwko sobie z ustawionymi przez nas poziomami inteligencji. Dodatkową opcją jest uruchomienie obszernego testu, który zasymuluje rozgrywkę komputerowych graczy na wszystkich poziomach trudności, mniejszych od sprecyzowanego przez nas n. Komunikacja z aplikacją odbywa się poprzez wspomniany plik konfiguracyjny - przed uruchomieniem aplikacji, oraz konsolę aplikacji - po jej uruchomieniu. W programie zostało użyte narzędzie automatyzujące budowę aplikacji - Maven. Dzięki niemu, po pobraniu kodu źródłowego aplikacji jesteśmy w stanie uruchomić ją z poziomu konsoli dzięki dwóm linijkom wpisanym do terminala. Okazało się ono niezwykle pomocne podczas przeprowadzania czasochłonnych testów na maszynie zdalnej, której sterowanie odbywało się właśnie poprzez terminal.

2.1. Plik konfiguracyjny

Opcje dostępne wewnątrz pliku konfiguracyjnego to:

- **gameType** - wartości, jakie możemy wprowadzić to Square oraz Overlap. Jest to typ gry, którego zamierzamy użyć i precyzuje, czy zagramy w Longest Square-free word czy Longest Overlap-free word.
- **gameMode** - tryb gry wartości wpisywane w tym polu mają wpływ na to, czy gra odbywać się będzie z drugim człowiekiem - *humanHuman*,

komputerem z tym warunkiem, że to my jesteśmy budowniczym - *humanBuilder*, ponownie z komputerem, jednak tym razem to on jest budowniczym - *pcBuilder*, oraz walka dwóch komputerów - *pcPc*.

- **setPower** - jest to moc zbioru elementów, do których dostęp będzie miał malarz podczas rozgrywki. Zbiór ten wypełniany jest liczbami $\in \{0, ..., n-1\}$
- **builderNestingLevel** i **painterNestingLevel** - poziom zagnieżdżenia na jaki komputer sobie pozwoli jako budowniczy i malarz. Zasada działania zagnieżdżeń zostanie omówiona w dalszej części pracy.
- **maxThinkTime** - podczas przeprowadzania testów z udziałem komputerowych graczy, czasami nie chcemy by przeciwnik myślał nad swoim ruchem 17 godzin. Właśnie dlatego została wprowadzona ta opcja konfiguracyjna. Jeśli czas jaki komputer spędził nad wyliczeniem kolejnej pozycji lub symbolu, będzie większy niż ustalona przez nas liczba nanosekund rozgrywka zostaje przerwana.
- **makeOverallTest** - gdy opcja ta zostanie ustawiona na true, po uruchomieniu aplikacji zostanie przeprowadzony obszerny test, zawierający w sobie kombinacje rozgrywek komputerów o wszystkich poziomach zagnieżdżeń ≤ 6 na wszystkich mocach zbiorów > 0 i ≤ 7. Czyli zostanie wykonanych $6 * 6 * 6$ rozgrywek. Warto wspomnieć, że podczas tego testu opcja maxThinkTime okazała się niezwykle pomocna.

2.2. Algorytm szukający powtórzeń wewnątrz ciągu

```
List<Integer> findSquare(List<Integer> sequence) {
  List<Integer> squareSeq = null;
  int maxSeqSize = sequence.size()/2;
```

```java
4    int minSeqSize = 1;
5    for(int ąsubSeSize=minSeqSize; subSeqSize<=maxSeqSize;subSeqSize++) {
6      squareSeq = compareSubSeq(subSeqSize, sequence);
7      if(squareSeq != null) {
8        return squareSeq;
9      }
10   }
11   return null;
12 }
```

Powyższy algorytm ma za zadanie znalezienie kwadratu w sekwencji, którą reprezentuje lista Integerów przekazana w parametrze. Zmienna **maxSeqSize** jest to długość najdłuższego podciągu jaki się zmieści w sekwencji, jeśli dostawimy za nim podciąg o identycznej długości.

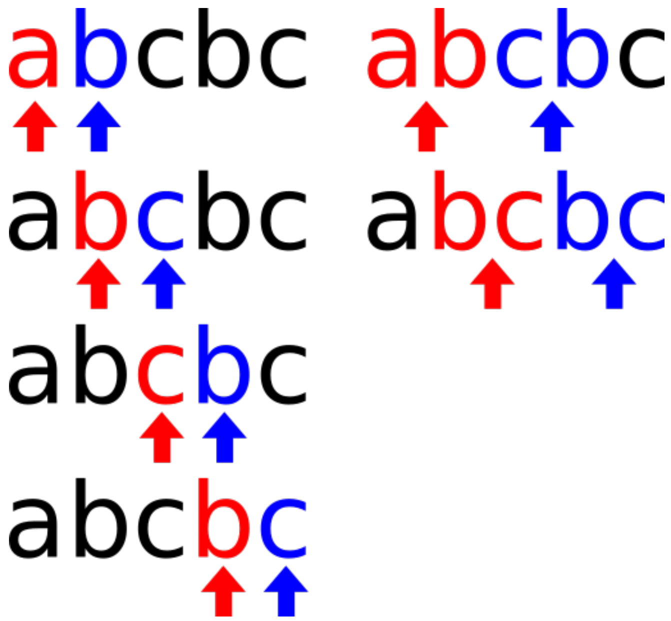
Natomiast zmienna **minSeqSize** jest to minimalna długość podciągu, który może składać się na kwadrat, naturalnie wynosi ona 1.

W linii 5 wykonujemy pętlę, wewnątrz, której do metody **compareSubSeq** przekazywana jest długość podciągu, który składać się będzie na kwadrat, oraz naszą sekwencję. Metoda **compareSubSeq** zwróci nam lewą część znalezionego kwadratu, lub null w przypadku, gdy taki kwadrat w sekwencji nie istnieje.

```java
Subsequence compareSubSeq(int subSeqSize, List<Integer> sequence) {
  List<Integer> left = new ArrayList<>();
  List<Integer> right = new ArrayList<>();
  int comparesFitInSequence = (sequence.size() + 1) - (subSeqSize*2) ;
  for(int i=0; i<comparesFitInSequence; i++) {
    for(int j =0;j<subSeqSize;j++) {
      left.add(sequence.get(i+j));
      right.add(sequence.get(i+j+subSeqSize));
    }
    if(listsAreEqual(left, right)) {
      return new Subsequence(left, i, subSeqSize);
    }
    left.clear();
```

```
14        right.clear();
15      }
16      return null;
17  }
```

Zmienne **left** i **right** są to podciągi, które reprezentują lewą i prawą część kwadratu aa. Zmienna **comparesFitInSequence** jest to ilość porównań jaka zmieści się wewnątrz naszej sekwencji. Dla przykładu, gdy nasz ciąg, jest reprezentowany przez $S = abcabaca$, a wcześniej sprecyzowana długość podciągu, składającego się na kwadrat wynosi 2, to zmienna **comparesFitInSequence** zostanie ustawiona na $(8+1)-(2*2)$ czyli 5, ponieważ możliwe są następujące porównania: **abca**$baca$, a**bcab**aca, ab**caba**ca, abc**abac**a, $abca$**baca** W pętli znajdującej się w linii 5 do listy **left** oraz **right** dodawane są podciągi odpowiedniej długości, natomiast w linii 10 następuje sprawdzenie czy podciągi są identyczne. Jeśli - tak oznacza to, że w naszej sekwencji, rozpoczynając od indeksu i występuje kwadrat długości $2*$**subSeqSize**. Jeśli listy są różne to w linii 13 następuje ich wyczyszczenie, po to by pętla mogła porównać dwa kolejne podciągi.



Rysunek 2.1. Działanie algorytmu szukającego kwadratów.

Dla typu gry w którym unikamy nasunięć powstały osobne metody.

```
Subsequence findOverlap(List<Integer> sequence) {
  Subsequence repeatedSequence = null;
  int maxSeqSize = (sequence.size()/2)+1;
  int minSeqSize = 3;
  for(int subSeqSize=minSeqSize; subSeqSize<=maxSeqSize; subSeqSize++) {
    repeatedSequence = compareSubSeqOverlap(subSeqSize, sequence);
    if(repeatedSequence != null) {
      return repeatedSequence;
    }
  }
  return null;
}
```

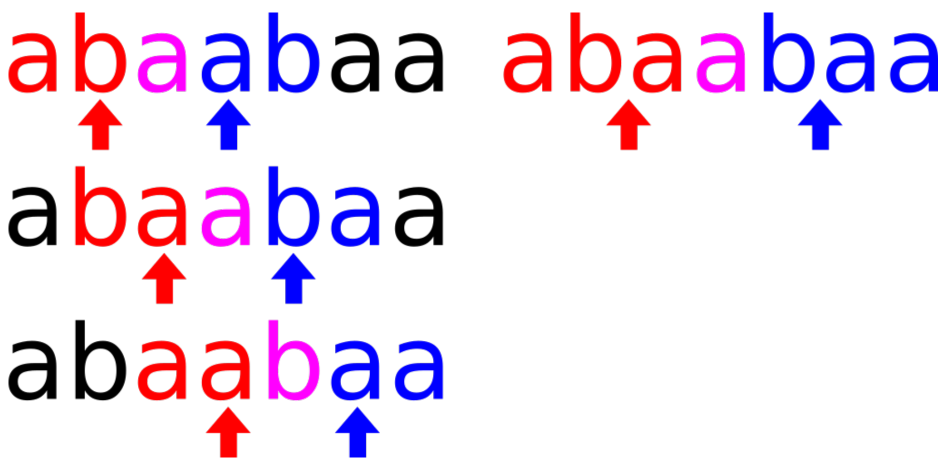
Metoda **findOverlap**, działa w sposób podobny do metody **findSquare**. Różnice to inne wartości zmiennych **maxSeqSize**, **minSeqSize** oraz metoda wywoływana w pętli.

Do zmiennej **maxSeqSize** przypisywana jest liczba o jeden większa niż długość sekwencji, ponieważ szukamy nasunięcia, a więc podciągi będą ze sobą dzieliły jeden znak. Wobec tego dla ciągu $S = \{abcabcb\}$, najdłuższy porównywany ciąg będzie długości 4, ponieważ w ostatniej iteracji pętli będziemy ze sobą porównywali podciągi $abca$ oraz $abcb$, które dzielą ze sobą literę a.

Wartość zmiennej **minSeqSize** wynosi 3, ponieważ jest to warunkiem stworzenia nasunięcia.

```
1
2  Subsequence compareSubSeqOverlap(int subSeqSize, List<Integer> sequence) {
3    List<Integer> left = new ArrayList<>();
4    List<Integer> right = new ArrayList<>();
5    int comparesFitInSequence = (sequence.size() + 2) - (subSeqSize*2);
6    for(int i=0; i<comparesFitInSequence; i++) {
7      for(int j =0;j<subSeqSize;j++) {
8        left.add(sequence.get(i+j));
9        right.add(sequence.get(i+j+subSeqSize-1));
10     }
11     if(listsAreEqual(left, right)) {
12       return new Subsequence(left, i, subSeqSize);
13     }
14     left.clear();
15     right.clear();
16   }
17   return null;
18 }
```

Metoda **compareSubSeqOverlap** również jest analogiczna do metody **compareSubSeq**. Różni się tutaj wartość zmiennej **comparesFitInSequence**, jest ona większa o 1, z takiego samego powodu, co zmienna **maxSeqSize** z metody **findOverlap**. Różni się również podciąg zapisywany do zmiennej **right**, w pętli z 6 linii. Pierwszy indeks owego podciągu jest równy indeksowi ostatniego elementu, lewego podciągu, po to by stworzyć nasunięcie.



Rysunek 2.2. Działanie algorytmu szukającego nasunięć.

2.3. Komunikacja z użytkownikiem

Plik konfiguracyjny nie jest wystarczającym środkiem komunikacji z aplikacją. W związku z tym, po uruchomieniu programu mamy dostęp do konsoli, służy do wyświetlania jak i wprowadzania treści.

Po uruchomieniu aplikacji wypisywane w konsoli wypisywane są najważniejsze opcje konfiguracyjne, takie jak poziom budowniczego, poziom malarza oraz lista dostępnych symboli. Jeśli uruchomiliśmy grę w trybie **humanHuman**, to aplikacja w pierwszej kolejności poprosi nas o indeks. Po wpisaniu indeksu mieszczącego się w przedziale $\in \{0, ..., n\}$, gdy to zrobimy zostaniemy poproszeni o podanie symbolu ≥ 0 i $< setPower$. W momencie gdy podaliśmy prawidłowe wartości na ekran konsoli wyświetlany zostaje ciąg S', który został utworzony poprzez dodanie do istniejącego ciągu odpowiedniego symbolu. Jeśli w ciągu S' pojawi się kwadrat, na ekran zostanie wyświetlony czas rozgrywki, ilość ruchów jaka została do tej pory wykonana oraz indeksy wraz z symbolami, wspomnianego powtórzenia.

Oto przykład prostej rozgrywki:

```
#> W grze ędostpne ąs ęąnastpujce liczby:
0
1
2
 0: { 0 } 1: { }
## =============================================================== ##
#> Podaj indeks:
1
#> Podaj ęliczb:
2
 0: { 0 } 1: { 2 } 2: { }
## =============================================================== ##
#> Podaj indeks:
1
#> Podaj ęliczb:
1
 0: { 0 } 1: { 1 } 2: { 2 } 3: { }
## =============================================================== ##
#> Podaj indeks:
3
#> Podaj ęliczb:
1
 0: { 0 } 1: { 1 } 2: { 2 } 3: { 1 } 4: { }
## =============================================================== ##
#> Podaj indeks:
4
#> Podaj ęliczb:
2
 0: { 0 } 1: { 1 } 2: { 2 } 3: { 1 } 4: { 2 } 5: { }
## =============================================================== ##

#> Znaleziono kwadrat:
 1: { 1 } 2: { 2 } <-> 3: { 1 } 4: { 2 }
```

```

## ============================================================ ##

#> Rozrgrywka łtrwaa 5 óruchw i 20.339 sekund.
```

Jeśli uruchomimy rozgrywkę z komputerem, obok wybranego indeksu lub symbolu pojawia się również czas jaki był mu potrzebny na podjęcie decyzji.

```
#> Komputer łwybra indeks: 1   | Czas trwania ńoblicze: 24.127 s
#> Komputer łwybra ęliczb: 4   | Czas trwania ńoblicze: 0.012 s
```

By ułatwić późniejszą analizę wszystkie informacje zawarte w konsoli, zapisywane są do nowo utworzonego pliku w katalogu output.

2.4. Zachłanny algorytm wyszukiwania symbolu

Jak zostało wcześniej wspomniane można sterować poziomem inteligencji komputerowych oponentów za pomocą zmiennych konfiguracyjnych. Jeśli ustawimy poziom zagnieżdżeń na 0, algorytm będzie działał zachłannie, wybierając opcje, która jest najatrakcyjniejsza w danym momencie, nie zważając na to, co może wydarzyć się w kolejnej turze. Obrazuje to poniższy algorytm szukający pasującego symbolu na wskazanym wcześniej indeksie.

```
int findRightColorGreedy(List<Integer> sequence, int index, int power) {
    for(int symbol=0;symbol<power; symbol++) {
      sequence.add(index, symbol);
      if(pickProperFind(sequence) == null) {
        sequence.remove(index);
        return symbol;
      } else {
        sequence.remove(index);
```

```
9        }
10     }
11     return -1;
12   }
```

Metoda na wejściu dostaje trzy parametry:

- sequence - utworzony wcześniej ciąg.
- index - indeks wybrany przez budowniczoego.
- power - moc zbioru symboli.

Zbiór symboli to liczby należące do przedziału $[0; power - 1]$. W 2 linii metody rozpoczyna się pętla, która iteruje po wszystkich dostępnych symbolach. W kolejnej linii symbol dodawany jest do naszego ciągu na podanej pozycji. Metoda pickProperFind z warunku if zależnie od typu gry wywołuje wcześniej opisane metody findSquare lub findOverlap. Jeśli warunek if zostanie spełniony oznacza to, że po dodaniu aktualnego symbolu na danej pozycji nie powoduje stworzenia kwadratu/nasunięcia, algorytm zatem usuwa dodany element z ciągu i zwraca go w linii 6. Jeżeli okaże się jednak, że dodany symbol tworzy powtórzenie w ciągu, zostaje on również usunięty, a pętla zaczyna się od początku. Metoda zwraca wartość -1 jeśli okaże się, że żaden z symboli nie jest w stanie stworzyć ciągu wolnego od kwadratów/nasunięć.

2.5. Zachłanny algorytm wyszukiwania indeksu

By można było przeprowadzić symulację gry, należy wprowadzić również algorytm budowczniego, starający się znaleźć najmniej atrakcyjny indeks dla malarza. Zachłanny algorytm realizujący to zadanie znajduje się poniżej.

```
int findRightIndexGreedy(List<Integer> sequence, int power) {
  int winner = -1;
```

```
3     int smallestSymbolSize = power;
4     for (int i =0;i<sequence.size()+1;i++) {
5       List<Integer> symbols = getFitableColorList(sequence, i);
6       if (symbols.size() <= smallestSymbolSize) {
7         smallestSymbolSize = symbols.size();
8         winner = i;
9       }
10    }
11    return winner;
12  }
```

Algorytm jako parametry, otrzymuje stworzony wcześniej ciąg, oraz moc zbioru symboli. Zmienna winner ustawiona początkowo na -1, reprezentuje indeks, który zostanie zwrócony jako ten, pod którym jest najmniejsza dowolność wyboru symboli, bez tworzenia powtórzeń. Metoda iteruje po każdym indeksie ciągu i zapisuje do listy symbole, po których wstawieniu w dane miejsce nie utworzy się kwadrat/nasunięcie. Następnie w warunku if z 6 linii sprawdzane jest, czy aktualna lista symboli jest mniejsza niż ta, zarejestrowana wcześniej. Jeśli tak, do zmiennej winner zapisany zostaje aktualny indeks.

2.6. Algorytm wyszukiwania symbolu z zagnieżdżeniami

Algorytm zachłanny nie jest wystarczająco sprytny, żeby przeciwstawić się człowiekowi mającemu odrobinę doświadczenia w Longest free word. Wobec tego powstał algorytm nie działający zachłannie, lecz starający się przewidzieć, jakie konsekwencje w kolejnych turach może nieść ze sobą dany wybór.

Poniższy algorytm wprowadza pojęcie punktacji. Podczas jego działania, dla możliwych wyborów nadawane są punkty. Im więcej punktów uzyska dany symbol, tym atrakcyjniejszym staje się on wyborem. Algorytm sprawdza jakie symbole można dodać w predefiniowanym indeksie, następnie iterując w pętli, dodaje każdy z symboli i sprawdza, ile możliwości będzie miał w kolejnej lub kolejnych turach, biorąc pod uwagę wszystkie dostępne indeksy. Każda dodatkowa możliwość to dodatkowy punkt dla wybranego koloru. Sam algorytm składa się z dwóch głownych części: **findRightColorPredicting** oraz **simulation**.

```
1  int findRightColorPredicting(List<Integer> sequence, int index) {
2    List<Integer> scoreList = initScoreList(power);
3    List<Integer> symbolList = getFitableSymbolList(sequence, index);
4    for(int symbol: symbolList) {
5      sequence.add(index, symbol);
6      simulation(sequence, symbol, scoreList, painterNestingLevel);
7      sequence.remove(index);
8    }
9    return getRandomFromScoreList(scoreList);
10 }
```

Na początku algorytm inicjalizuje listy **scoreList** oraz **symbolList**. Tę pierwszą tyloma zerami ile jest w grze dostępnych symboli, drugą natomiast symbolami, jakie możemy wstawić w zdefiniowany przez budowniczego indeks. Linia 4 rozpoczyna pętlę, która iterując po liście **symbolList**, dodaje jej element, wywołuje metodę **simulation**, przekazując utworzony **ciąg**, dodany **symbol**, **scoreList** oraz p**oziom zagnieżdżenia malarza**, po czym usuwa dodany **symbol** sprawiając, że ciąg pozostaje bez zmian. Na końcu zwraca element, który miał największą liczbę punktów. Jeśli kilka symboli otrzymało ich tyle samo, program wybiera losowy z nich. Dzięki w grze występuje pewna przypadkowość, oraz jest mała szansa na powtórzenie dwóch identycznych rozgrywek przy odpowiednio długim ciągu.

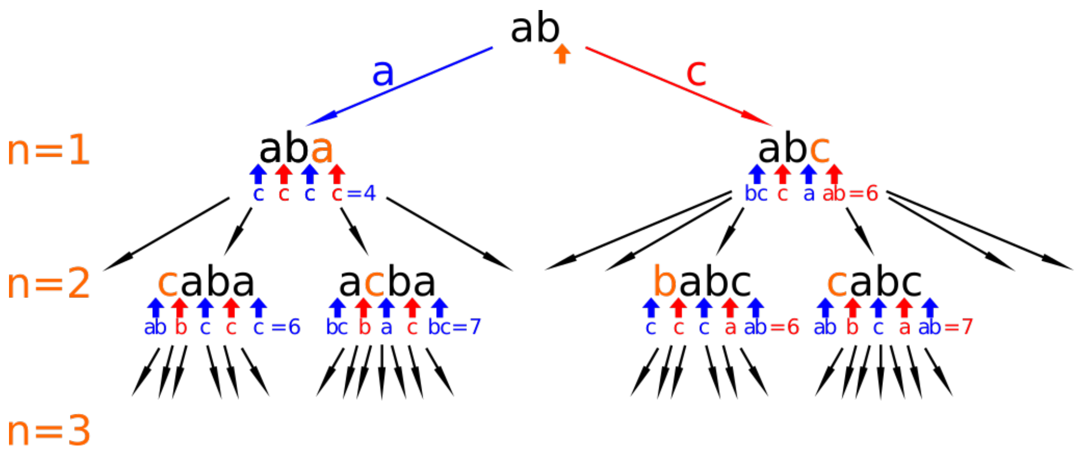
Wewnątrz metody **simulation** nadawane są punkty, oraz za pomocą rekurencji wykonywana jest symulacja kolejnych iteracji gry.

```
void simulation(List<Integer> sequence, int indexInScoreList, List<Integer>
    scoreList, int invokes) {
  invokes--;
```

```
3    for(int j=0;j<sequence.size()+1;j++) {
4      List<Integer> symbolList = getFitableSymbolList(sequence, j);
5      updateScoreList(scoreList, indexInScoreList, symbolList.size());
6      for(int symbol: symbolList) {
7        sequence.add(j, symbol);
8        if(invokes > 0) {
9          simulation(sequence, indexInScoreList, scoreList, invokes);
10       }
11       sequence.remove(j);
12     }
13   }
14 }
```

Na początku dekrementowana zostaje ilość zagnieżdżeń na jaką chcemy się zagłębić. Oznacza to, że jeśli poziom zagnieżdżenia ustawiony jest na 1, to metoda **simulation** zostanie wywołana tylko raz. Pętla z 3 linii iteruje po indeksach pod którymi możliwe jest dodanie symbolu. W 4 linii do listy zapisywane są symbole, które można wcisnąć pod aktualny indeks, nie powodując powtórzenia. Metoda **updateScoreList** zwiększa ilość punktów symbolu przekazanemu z poprzedniej metody. Ilość punktów jest równa liczbie elementów, zmiennej **symbolList**. Pętla z 6 linii iterując po liście pasujących symboli, dodaje element, następnie pod warunkiem, że **invokes** jest większe od 0 wywołuje samą siebie ze zmodyfikowanym ciągiem, tym samym indeksem, który został przekazany na początku, listą punktową oraz pozostałą liczbą wywołań. Na końcu pętli element zostaje usunięty, po to, by ciąg wrócił do pierwotnego stanu.

Algorytm obrazuje poniższa grafika:



Rysunek 2.3. Działanie algorytmu malarza z zagnieżdżeniami.

Mamy ciąg $S = a, b$, budowniczy wybrał indeks 2, czyli malarz wybiera symbol jaki zostanie wstawiony za literką b. Typ gry to **Longest square free word**. Moc zbioru wynosi 3, więc do wyboru są dostępne symbole a, b, c. Wstawienie symbolu b stworzyłoby kwadrat, dlatego algorytm rozważa literki a oraz c. Zmienna n jest to poziom zagnieżdżenia algorytmu, natomiast liczby przy pasujących symbolach oznaczają punkty, jakie zostają przypisane symbolowi a lub c.

Już przy pierwszym zagnieżdżeniu widać, że literka c jest atrakcyjniejsza, ponieważ wstawienie jej zapewnia malarzowi więcej możliwości w kolejnych rundach. Jednak sprawdzenie jednego ruchu do przodu nie zawsze jest wystarczające i zdarza się, że z pozoru nieatrakcyjny symbol w perspektywie kolejnych, tur jest najlepszą opcją.

2.7. Algorytm wyszukiwania indeksu z zagnieżdżeniami

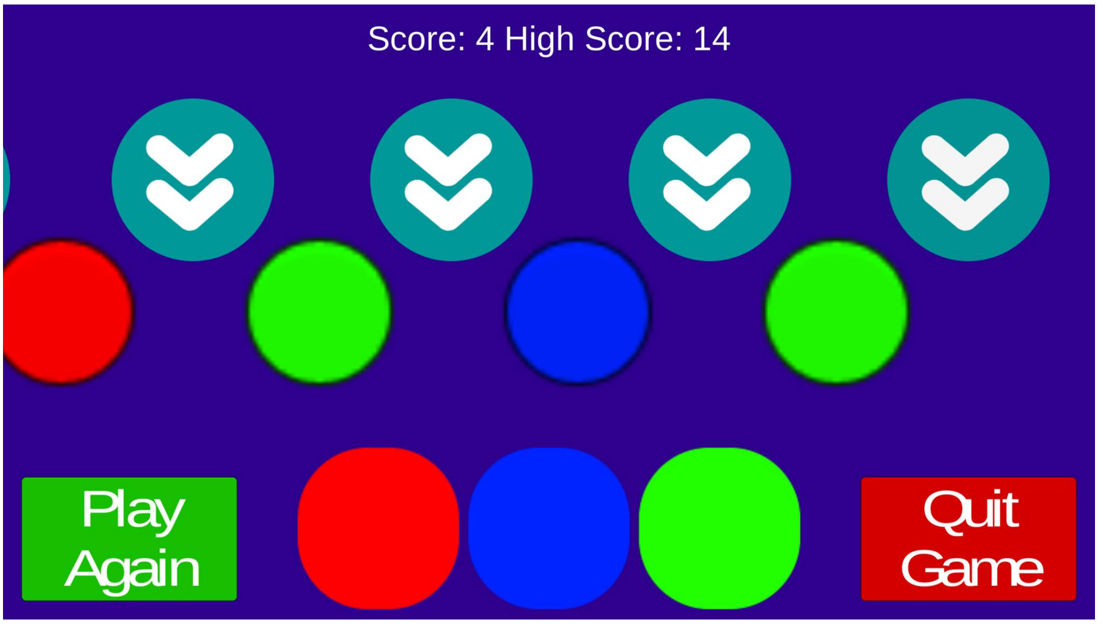
Sposób działania algorytmu z zagnieżdżeniami szukającego indeksu, który ma największą szansę na stworzenie powtórzenia, działa na podobnej zasadzie, co algorytm wyszukiwania symbolu. Różnica polega na tym, że w jego pierwszej części, iterujemy po wszystkich dostępnych symbolach, a nie jednym predefiniowanym, oraz punkty nadawane zostają indeksom i zwracamy ten, który uzyska ich najmniej. Druga część algorytmu czyli metoda **simulation** pozostaje bez zmian.

```
1  int findRightIndex(List<Integer> sequence) {
2    List<Integer> scoreList = initScoreList(sequence.size() + 1);
3    for (int i = 0; i<sequence.size() + 1; i++) {
4      List<Integer> symbols = getFitableSymbolList(sequence, i);
5      for (int symbol : symbols) {
6        sequence.add(i, symbol);
7        simulation(sequence, i, predictList, builderNestingLevel);
8        sequence.remove(i);
9      }
10   }
11   return getRandomMinFromPredict(scoreList);
12 }
```

Na początku metody zainicjalizowana zostaje zmienna scoreList. Do listy dodane zostają zera w ilość odpowiadającej długość ciągu plus jeden, bo właśnie na tylu indeksach możemy dodać nowy element. W 3 linii iterujemy po każdym dostępnym indeksie, natomiast od 4 linii mamy już wszystko to co w algorytmie szukającym symbolu. Na końcu metody zwracany zostaje indeks, który uzyskał najmniej punktów, czyli będzie najmniej atrakcyjny dla malarza. Tak jak poprzednio jeśli istnieje więcej niż jeden indeks z minimalna wartością, element jest wybierany losowo.

2.8. Longest square free word z interfejsem graficznym

Powstała również wersja gry, w której użytkownik nie korzysta z konsoli, lecz z graficznego interfejsu. W tej odmianie użytkownik za pomocą trzech kolorów ma stworzyć jak najdłuższy ciąg bez kwadratów. Czynności jakie gracz wykonuje to wybranie miejsca, w które zostanie wstawiony element oraz jego kolor. Jeśli w tworzonym ciągu pojawi się kwadrat rozgrywka zostaje przerwana i na ekranie zostają podświetlone powtórzenia składające się na kwadrat. Program zapisuje również najwyższy wynik, który jest równy długości utworzonego ciągu. Algorytm szukający kwadratów nie różni się w żaden sposób od algorytmu z podstawowej wersji gry.



Rysunek 2.4. Longest Free Word z interfejsem graficznym

Aplikacja została napisana na popularnym silniku do tworzenia gier - Unity. Pozwala on kompilować kod programu do plików wykonywalnych, które mogą być uruchamiane w przeglądarkach, na komputerach osobistych, konsolach i telefonach komórkowych. W tym przypadku pod uwagę brane były głównie telefony komórkowe, o czym może świadczyć wielkość przycisków i mała ilość szczegółów na ekranie.

ROZDZIAŁ 3

Analiza symulowanych potyczek

3.1. Ilość symboli potrzebna na rozgranie partii

Długość gry zależna jest od dwóch czynników. Ustawionego poziomu zagnieżdżenia i ilości dostępnych symboli. Thue w swoim twierdzeniu[][], udowodnił, że jesteśmy w stanie stworzyć nieskończenie długi ciąg bez kwadratów mając do dyspozycji 3 symbole. Sytuacja zmienia się, gdy ciąg tworzony jest przez dwóch oponentów, z których jeden stara się utworzyć kwadrat. Nawet używając zachłannego algorytmu w przypadku budowniczego i algorytmu z zagnieżdżeniami 7 poziomu dla malarza, rozgrywka zawsze trwała będzie maksymalnie 5 ruchów.

Dla 4 symboli, również nie jest możliwe rozegranie partii dłuższej niż 10 ruchów. Przy zachłannym algorytmie budowniczego, malarz jest w stanie utworzyć ciąg długości 10. Jeśli algorytm budowniczego używa zagnieżdżeń, liczba ta zmniejsza się do 7.

Dopiero rozgrywka na 5 symbolach daje algorytmom pole do popisu, bowiem w przeprowadzonych badaniach, zależnie od ustawionych poziomów, trwała ona od 11 do conajmniej 236 ruchów.

Ilość symboli	Malarz	Budowniczy	Ilość ruchów
3	≥ 0	≥ 0	5
4	0	0	7
4	≥ 1	≥ 1	7
4	0	≥ 1	10

Tabela 3.1. Maksymalna ilość ruchów w stosunku do ilości dostępnych symboli.

Używanie zachłannego algorytmu malarza nadal nie skutkuje zbyt długą grą. Niezależnie od poziomu budowniczego, kwadrat pojawi się zawsze po 11 kroku. Jednak podczas prób, gdy malarz miał ustawiony poziom zagnieżdżenia równy 1, a budowniczy działał zachłannie, rozgrywka trwała od 44 do 236 tur. Średnia arytmetyczna wyciągnięta z ilości ruchów wyniosła 133.83

Ilość symboli	Ilość ruchów	Czas trwania
5	44	0.1 s.
5	56	0.4 s.
5	81	2.4 s.
5	88	4.3 s.
5	100	7.6 s.
5	102	7.3 s.
5	163	1 min. 27 s.
5	171	1 min. 48 s.
5	173	1 min. 59 s.
5	193	4 min.
5	199	3 min. 38 s.
5	236	8 min. 51 s.

Tabela 3.2. Podsumowanie badań dla budownieczego poziomu 0 i malarza poziomu 1.

Przy poziomie budowniczego i malarza ustawionym na 1, podczas 10 prób rozgrywka trwała od 30 do 213 ruchów. Średnia arytmetyczna obliczona na podstawie rozgrywek wynosi 96.9, a więc jest o 37.23 niższa niż, gdy budowniczy używał algorytmu zachłannego.

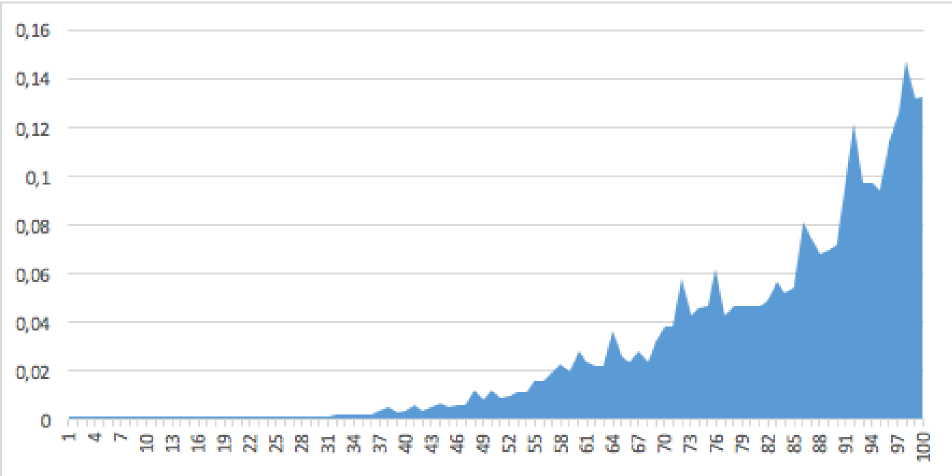
Ilość symboli	Ilość ruchów	Czas trwania
5	30	1.2 s.
5	42	5.1 s.
5	45	11 s.
5	47	7.8 s.
5	73	1 min. 17 s.
5	104	9 min. 38 s.
5	114	29 min. 29 s.
5	134	1 godz. 30 min. 27 s.
5	178	5 godz. 11 min. 22 s.
5	202	9 godz. 42 min. 15 s.
5	213	13 godz. 04 min. 35 s.

Tabela 3.3. Podsumowanie badań dla budownieczego i malarza poziomu 1.

Obecne zasoby sprzętowe i czasowe sprawiły, że malarz poziomu 2 okazał się niepokonany w walce z budowniczym poziomu 0. Przy ponad 10 próbach ani razu nie był zmuszony do stworzenia kwadratu w rozgrywce trwającej ≤ 220 tur i około 26 godzin.

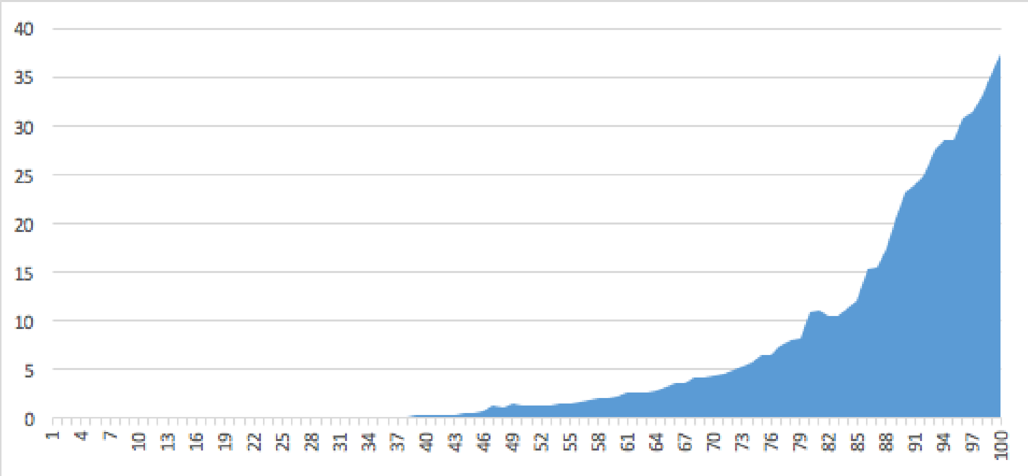
3.2. Pomiary czasów potrzebnych na podjęcie decyzji

Nieodłącznym elementem analizy działania aplikacji jest badanie czasu w jakim wykonuje ona swoje algorytmy. Nie inaczej jest w przypadku Longest Free Word. Najkrótrzy zbadany czas decyzji algorytmu wyniósł 0.001 sekundy, najdłuższy natomiast około 2 godzin. Czas potrzebny na decyzję jest tym dłuższy im dłuższy jest stworzony do tej pory ciąg. Naturalnie więc, na początku rozgrywki algorytm będzie działał szybko, lecz w raz z rozwojem potyczki będzie zwalniał. Zależność tą można zaobserwować na wykresie 3.1. Widoczne odchylenia to prawdopodobnie efekt działania innego procesu działającego na tej samej maszynie.

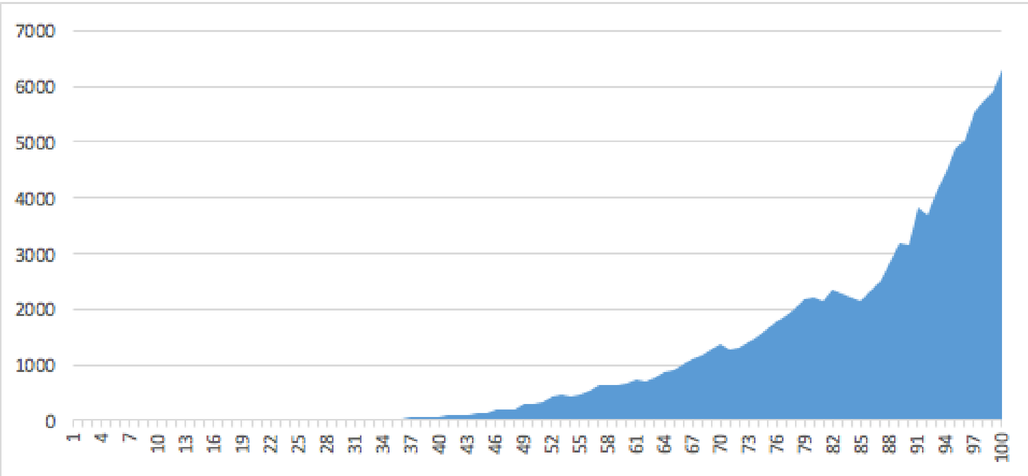


Rysunek 3.1. Ilość sekund potrzebna na podjęcie decyzji jako budowniczy przy poszczególnych długościach ciągu. Budowniczy: 0, Malarz: 1.

Warto zauważyć jak ogromne różnice w wydajności istnieją pomiędzy poziomami zagnieżdżenia algorytmów.



Rysunek 3.2. Ilość sekund potrzebna na podjęcie decyzji jako budowniczy przy poszczególnych długościach ciągu. Budowniczy: 1, Malarz: 1.

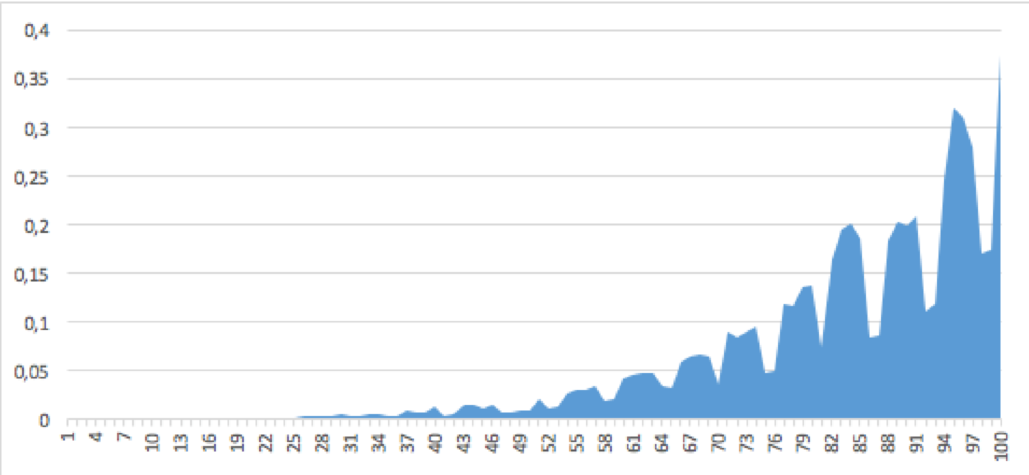


Rysunek 3.3. Ilość sekund potrzebna na podjęcie decyzji jako budowniczy przy poszczególnych długościach ciągu. Budowniczy: 2, Malarz: 1.

Przy zagnieżdżeniu ustawionym na 2, dla ciągu długości 100 wyliczenie odpowiedniej decyzji zajęło około 170 razy więcej czasu niż przy zagnieżdżeniu równym 1.

Odrobinę inaczej wyglądają analogiczne wykresy dla malarza. Algorytm malarza z zagnieżdżeniami symuluje dodawanie kolejnych symboli w wyznaczonym indeksie, jednak jeżeli w danym miejscu da się wstawić tylko 1 symbol, to przeprowadzi mniej symulacji, niż jeśli dałoby się tam wstawić 5 symboli. Dzięki tej zależności na wykresie można zauważyć, przy których ruchach malarz jest w opałach, a przy których ma dostępnych wiele opcji.

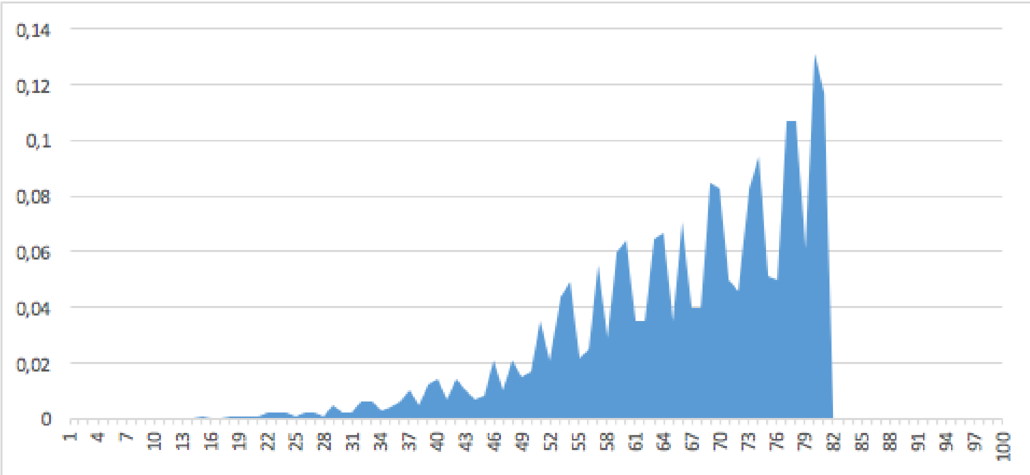
Na wykresie 3.4, widać wzloty i spadki czasu trwania algorytmu. Jak widać zachłanny algorytm budowniczego cyklicznie redukuje liczbę opcji malarza.



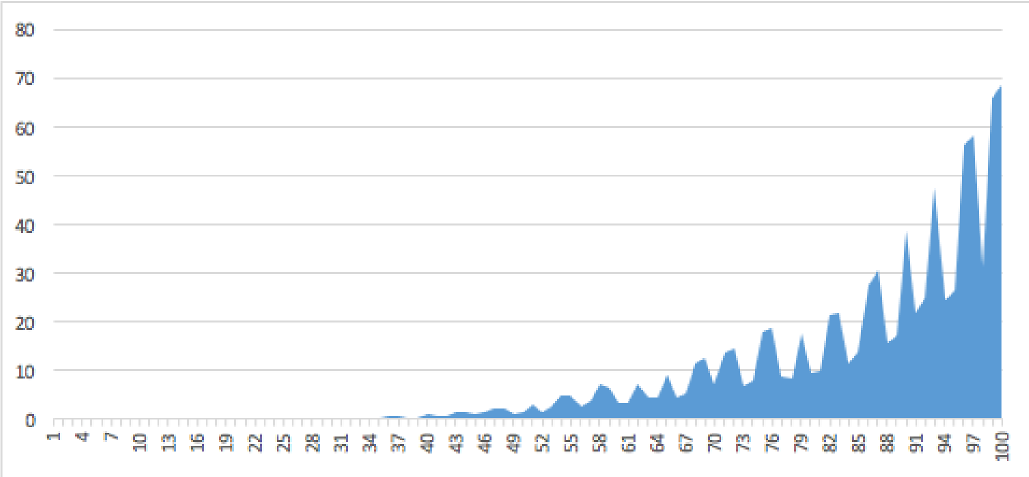
Rysunek 3.4. Ilość sekund potrzebna na podjęcie decyzji jako malarz przy poszczególnych długościach ciągu. Budowniczy: 0, Malarz: 1.

Jeśli przeanalizujemy wykres 3.5 zauważymy, że podczas rozgrywki ze sprytniejszym algorytmem budowniczego, krzywe są o wiele bardziej spiczaste, co oznacza, że jest to trudna rozgrywka dla malarza, która ostatecznie kończy się w 82 ruchu.

Ponownie możemy porównać ze sobą czasy podejmowania decyzji dla algorytmu z zagnieżdżeniem 1 poziomu na wykresie 3.5, oraz 2 poziomu na wykresie 3.6. Wykonanie ruchu, dla ciągu długości 79 pierwszemu algorytmowi zajęło 0.061 sekundy, natomiast temu bardziej złożonemu 17.711. Zatem w tym konkretnym ruchu algorytm z zagnieżdżeniem o 1 większym decydował się około 290 razy dłużej.



Rysunek 3.5. Ilość sekund potrzebna na podjęcie decyzji jako malarz przy poszczególnych długościach ciągu. Budowniczy: 2, Malarz: 1.



Rysunek 3.6. Ilość sekund potrzebna na podjęcie decyzji jako malarz przy poszczególnych długościach ciągu. Budowniczy: 0, Malarz: 2.

3.3. Obserwacja zachowań algorytmu budowniczego

ROZDZIAŁ 4

Narzędzia i standardy pokrewne

Systemy SGML, ze względu na mnogość funkcji jakie spełniają i ich kompleksowe podejście do oznakowywania i przetwarzania dokumentów tekstowych, są bardzo skomplikowane. Możemy wyróżnić dwa podejścia do budowy takich systemów. Z jednej strony, buduje się systemy zindywidualizowane, oparte o specyficzne narzędzia tworzone w takich językach, jak: C, C++, Perl czy Python. Edytory strukturalne, filtry do transformacji formatów czy parsery i biblioteki przydatne do konstrukcji dalszych narzędzi, tworzone są według potrzeb określonych, pojedynczych systemów.

Z drugiej strony, twórcy oprogramowania postanowili pójść krok dalej i połączyć te różne narzędzia w jedną całość. Tą całość miał stanowić DSSSL lub jego XML-owy odpowiednik – standard XSL. Ze względu na oferowane możliwości można twierdzić, że tworzenie i używanie narzędzi implementujących standard DSSSL/XSL, jest najwłaściwszym podejściem. Przemawiają za tym różne argumenty, ale najważniejszym z nich jest to, że mamy tu możliwość stworzenia niezależnego od platformy programowej i narzędziowej zbioru szablonów – przepisów jak przetwarzać dokumenty SGML.

4.1. Przetwarzanie dokumentów SGML – standard DSSSL

DSSSL (*Document Style Semantics and Specification Language*) – to międzynarodowy standard ściśle związany ze standardem SGML. Standard ten, można podzielić na następujące części:

- język transformacji (*transformation language*). To definicja języka służącego do transformacji dokumentu oznaczonego znacznikami zgodnie z pewnym DTD na dokument oznaczony zgodnie z innym DTD.

- język stylu (*style language*) opisujący sposób formatowania dokumentów SGML.
- język zapytań (*query language*) służy do identyfikowania poszczególnych fragmentów dokumentu SGML.

Opisane główne części składowe standardu DSSSL dają obraz tego, jak wiele aspektów przetwarzania zostało zdefiniowanych i jak skomplikowany jest to problem. Jest to głównym powodem tego, że mimo upływu kilku lat od zdefiniowania standardu nie powstały ani komercyjne ani wolnodostępne aplikacje wspierające go w całości. Istnieją natomiast *nieliczne* narzędzia realizujące DSSSL w ograniczonym zakresie, głównie w części definiującej język stylu, który odpowiada za opatrzenie dokumentu czysto strukturalnego w informacje formatujące. Daje to możliwość publikacji dokumentów SGML zarówno w postaci elektronicznej, hipertekstowej czy też drukowanej.

4.2. Przetwarzanie dokumentów XML – standard XSL

Tak jak XML jest *uproszczoną* wersją standardu SGML, tak XSL jest uproszczonym odpowiednikiem standardu DSSSL. W szczególności, wyróżnić można w tym standardzie następujące części składowe:

- język transformacji (XSLT) To definicja języka służącego do transformacji dokumentu.
- język zapytań (XPath) służy do identyfikowania poszczególnych fragmentów dokumentu.
- język stylu deefiniujący sposób formatowania dokumentów XML.

ROZDZIAŁ 5

Przegląd dostępnych narzędzi

W celu wykorzystania standardu SGML do przetwarzania dokumentów, niezbędne jest zebranie odpowiedniego zestawu narzędzi. Narzędzi do przetwarzania dokumentów SGML jest wiele. Są to zarówno całe systemy zintegrowane, jak i poszczególne programy, biblioteki czy skrypty wspomagające.

5.1. Narzędzia do przeglądania dokumentów SGML

Do tej kategorii oprogramowania zaliczamy przeglądarki dokumentów SGML oraz serwery sieciowe wspomagające standard SGML, przy czym rozwiązań wspierających standard XML jest już w chwili obecnej dużo więcej i są dużo powszechniejsze.

Jeżeli chodzi o przeglądarki to zarówno Internet Explorer jak i Netscape umożliwiają bezpośrednie wyświetlenie dokumentów XML; ponieważ jednak nie wspierają w całości standardu XML, prowadzi to ciągle do wielu problemów[1].

[1] Z innych mniej popularnych rozwiązań można wymienić takie aplikacje, jak: HyBrick SGML Browser firmy Fujitsu Limited, Panorama Publisher firmy InterLeaf Inc, DynaText firmy Inso Corporation czy darmowy QWeb. W przypadku serwerów zwykle dokonują one transformacji „w locie" żądanych dokumentów na format HTML (rzadziej bezpośrednio wyświetlają dokumenty XML). Ta kategoria oprogramowania ma, z punktu widzenia projektu, znaczenie drugorzędne.

5.2. Parsery SGML

Program `nsgmls` (z pakietu `SP` Jamesa Clarka) jest doskonałym parserem dokumentów SGML, dostępnym publicznie. Parser `nsgmls` jest dostępny w postaci źródłowej oraz w postaci programów wykonywalnych przygotowanych na platformę MS Windows, Linux/Unix i inne. Oprócz analizy poprawności dokumentu parser ten umożliwia również konwersję danych do formatu ESIS, który wykorzystywany jest jako dane wejściowe przez wiele narzędzi do przetwarzania i formatowania dokumentów SGML. Dodatkowymi, bardzo przydatnymi elementami pakietu `SP` są: program `sgmlnorm` do normalizacji, program `sx` służący do konwersji dokumentu SGML na XML oraz biblioteki programistyczne, przydatne przy tworzeniu specjalistycznych aplikacji służących do przetwarzania dokumentów SGML.

W przypadku dokumentów XML publicznie dostępnych, parserów jest w chwili obecnej kilkadziesiąt. Do popularniejszych zaliczyć można Microsoft Java XML Parser firmy Microsoft, LT XML firmy Language Technology Group, Exapt oraz XP (James Clark)

5.3. Wykorzystanie języków skryptowych

5.4. Wykorzystanie szablonów XSL

Stosując wersję XML typu DocBook można wykorzystać szablony stylów przygotowane w standardzie XSL (autor N. Walsh). W chwili obecnej są dostępne narzędzia umożliwiające przetworzenie dokumentów XML do postaci drukowanej (Adobe PDF) oraz hipertekstowej (HTML).

Podobnie jak w przypadku szablonów DSSSL, szablony stylów XSL są sparametryzowane i udokumentowane i dzięki temu łatwe w adaptacji. Do zamiany dokumentu XML na postać prezentacyjną można wykorzystać jeden z dostępnych publicznie procesorów XSLT (por. tabela 5.1).

XSL:FO jest skomplikowanym językiem o dużych możliwościach, zawierającym ponad 50 różnych „obiektów formatujących”, począwszy od

Nazwa	Autor	Adres URL
`sablotron`	Ginger Alliance	`http://www.gingerall.com`
`Xt`	J. Clark	`http://www.jclark.com`
`4XSLT`	FourThought	`http://www.fourthought.com`
`Saxon`	Michael Kay	`http://users.iclway.co.uk/mhkay/saxon`
`Xalan`	Apache XML Project	`http://xml.apache.org`

Tabela 5.1. Publicznie dostępne procesory XLST

Źródło: Opracowanie własne

najprostszych, takich jak prostokątne bloki tekstu poprzez wyliczenia, tabele i odsyłacze. Obiekty te można formatować wykorzystując przeszło 200 różnych właściwości (*properties*), takich jak: kroje, odmiany i wielkości pisma, odstępy, kolory itp. W tym dokumencie przedstawione jest absolutne miniumum informacji na temat standardu XSL:FO.

Cały dokument XSL:FO zawarty jest wewnątrz elementu `fo:root`. Element ten zawiera (w podanej niżej kolejności):

- dokładnie jeden element `fo:layout-master-set` zawierający szablony określające wygląd poszczególnych stron oraz sekwencji stron (te ostatnie są opcjonalne, ale typowo są definiowane);
- zero lub więcej elementów `fo:declarations`;
- jeden lub więcej elementów `fo:page-sequance` zawierających treść formatowanego dokumentu wraz z opisem jego sformatowania i podziału na strony.

Zakończenie

Możliwości, jakie stoją przed archiwum prac magisterskich opartych na XML-u, są ograniczone jedynie czasem, jaki należy poświęcić na pełną implementację systemu. Nie ma przeszkód technologicznych do stworzenia co najmniej równie doskonałego repozytorium, jak ma to miejsce w przypadku ETD. Jeżeli chcemy w pełni uczestniczyć w rozwoju nowej ery informacji, musimy szczególną uwagę przykładać do odpowiedniej klasyfikacji i archiwizacji danych. Sądzę, że język XML znacznie to upraszcza.

DODATEK A

Tytuł załącznika jeden

Treść załącznika jeden.

DODATEK B

Tytuł załącznika dwa

Treść załącznika dwa.

Spis tabel

Spis rysunków

Oświadczenie

Ja, niżej podpisany(a) oświadczam, iż przedłożona praca dyplomowa została wykonana przeze mnie samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

. . . .	. . . .
data	podpis