UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Szymon Rękawek
nr albumu: 206288

Gra Thuego

Praca magisterska na kierunku:
INFORMATYKA
Promotor:
prof. UG dr, hab. T. Dzido

Gdańsk 2016

Streszczenie

Celem pracy było zaprogramowanie i analiza wyników gry opartej na twierdzeniach Axela Thue, związanej z powtórzeniami oraz nasunięciami wewnątrz ciągu znaków. Głównymi funkcjonalnościami aplikacji jest rozgrywka między dwoma graczami oraz symulacja przeprowadzona przez konkurujące ze sobą algorytmy.

Powstały dwa warianty aplikacji, pierwszy z nich napisany został za pomocą silnika *Unity3d*, posiada interfejs graficzny, nie ma jednak innego zastosowania niż poprawianie własnych rekordów. Drugi jest napisany w języku *Java*, bez interfejsu graficznego, w zamian oferuje funkcjonalność dokonywania testów na serwerze zdalnym i zawiera wiele opcji konfiguracyjnych, między innymi ustawianie poziomu rekurencji, którego używają algorytmy.

Przeprowadzone testy wykazały, że algorytmy symulujące graczy działają prawidłowo i zwiększanie poziomu wykorzystywanej przez nich rekurencji skutkuje lepiej podejmowanymi decyzjami. Na podstawie badań ustalona została maksymalna długość rozgrywki przy odpowiedniej taktyce, dla rozgrywek zawierających określoną ilość symboli z których tworzony jest ciąg.

Słowa kluczowe

Thue, Java, ciąg, programowanie, Unity3d

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. Jego prace [2] opisywały pojęcie, które autor nazwał *nieredukowalnymi słowami*. Poświęca w nich szczególną uwagę dwu i trzy literowym przypadkom. W skrócie wprowadza pojęcie znane obecnie jako *Thue-Morse word* i pokazuje, że nieskończone *słowa bez nasunięć*, są pochodnymi tej sekwencji. W swoich pracach definiuje kolejną strukturę, a mianowicie *nieskończone słowo wolne od kwadratów*, oraz przedstawia sposoby generowania nieskończenie długich słów wolnych zarówno od *kwadratów* jak i *nasunięć*.

Gra która powstała na podstawie twierdzeń Thuego, w skrócie polegała będzie na utworzeniu jak najdłuższego ciągu znaków nad określonym z góry alfabetem. Zależnie od trybu gry, kończyć się będzie ona w momencie, gdy pojawi się zdefiniowany na początku rodzaj powtórzenia w tworzonym przez nas, bądź algorytm ciągu. Jednym z trybów gry jest rozgrywka komputera przeciwko niemu samemu, po jej zakończeniu przedstawiony zostanie czas trwania rozgrywki i każdego wykonanego ruchu.

Ostatnia część pracy poświęcona jest analizie algorytmów zarówno pod względem czasu ich wykonywania jak i zdolności do przewidywania ruchów przeciwnika. Badany jest również aspekt tego, ile symboli z których tworzony jest ciąg, jest potrzebnych na rozegranie dostatecznie długiej rozgrywki, przy możliwie najdoskonalszej taktyce.

ROZDZIAŁ 1

Opis struktur pozwalających na stworzenie gry

Zanim przejdziemy do twierdzeń, koniecznym jest wyjaśnienie podstawowych definicji, 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 słowa ω jest słowem u, które występuje wewnątrz ω formie $\omega = xuy$, podczas gdy x oraz y również są słowami tego alfabetu.
- *Kwadrat* jest słowem w formie uu, gdzie u jest niepuste.
- Słowo jest *wolne od kwadratów*, jeśli żaden z jego czynników nie jest *kwadratem*.

- *Nasunięcie* jest słowem w formie $xuxux$, gdzie x jest niepuste. Nazwa pojęcia wzięła się z tego, że xux występuje dwa razy w $xuxux$. Jako *prefiks* oraz jako *sufiks*, oba wystąpienia mają wspólną część – centralne x, a więc *nasuwają* się na siebie.

- Słowo jest *wolne od nasunięć* jeśli żaden z jego 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 matematycznego z jednej struktury w inną.

1.2. Thue-Morse word

Słowo Thue-Morse'a jest nieskończonym ciągiem znaków utworzonym z dwuliterowego alfabetu, nie zawierającym ani jednego nasunięcia. Zostało ono nazwane po Thue, który badał jego właściwości w pracy z 1906 roku [7], oraz Morsie, który pracował nad nim w latach 20 XX wieku. Konstrukcja zwana obecnie słowem Thue-Morse'a występuje również o wiele wcześniej w korespondencji Prouheta [8] z Francuską Akademią Nauk w 1851 roku. Prouhet podał więcej ogólnych konstrukcji, uzyskując nie tylko to słowo, ale całą rodzinę słów na większych alfabetach mających inne interesujące właściwości. Słowa te czasami odnoszą się do ogólnych słów Thue-Morse'a lub słów Prouheta. Dla alfabetu $A = \{0, 1\}$ słowo to może wyglądać w następujący sposób:

$$0110100110010110100101100110100 1\ldots$$

Jednym ze sposobów generowania słowa Thue-Morse'a jest zastosowanie się do poniższego wzoru.

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

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 $u_{2n+1} = v_{2n+1}^{\sim}$, gdzie $w^{\sim}$ jest odwróceniem w. Morfizm μ może być rozszerzony do nieskończonych słów, które mają dwa stałe punkty:

$$t = abbabaabbaababbabaab\ldots = \mu(t)$$
$$\overline{t} = baababbaabbabaababba\ldots = \mu(\overline{t})$$

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:

$$t_n = \left\{ \begin{array}{l} a \text{ if } d_1(n) \equiv 0 \pmod 2 \\ b \text{ if } d_1(n) \equiv 1 \pmod 2 \end{array} \right\}$$

gdzie $d_1(n)$ jest liczbą bitów równych 1 w binarnej rozbudowie $bin(n)$ z n. Na przykład 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 czy 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 kolejną metodę generowanie nieskończonego słowa bez *kwadratów* na trzech literach z użyciem poniższego morfizmu.

- $a \to abcab$

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

Po raz kolejny zastępujemy każde wystąpienie liter przez zdefiniowane sekwencje. Jest to dość skomplikowana struktura, zsumowana długość łańcuchów wynosi 18. A. Carpi [3] w 1983 roku 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* [5] opisuje grę online bazującą na twierdzeniach Thuego. Podczas rozgrywki dwaj gracze naprzemiennie wykonując swoje ruchy, tworzą ciąg bez *kwadratów*. Celem pierwszego gracza jest jak najszybsze skończenie rozgrywki poprzez utworzenie *kwadratu*, drugi natomiast musi tego unikać.

Wartym wspomnienia jest uproszczony tryb gry, podczas którego mamy graczy – Alicję i Boba, tylko Alicji zależy na tym by uniknąć *kwadratów*. Rozgrywka polega na tym, że Alicja i Bob wybierają na przemian symbole z ustalonego alfabetu 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 pracy [4], że Alicja jest w stanie 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 [5] jest *Online Thue Game*. Po raz kolejny gracze wykonują swoje ruchy na przemian. W swojej rundzie Bob wybiera pozycję s_i w ciągu S, gdzie $i \in \{0, 1, \ldots, n\}$, a n to aktualna długość ciągu. Następnie Alicja wybiera symbol x z alfabetu A, który jest wstawiany pomiędzy pozycją s_i i s_{i+1}, tworząc nowy ciąg $S' = s_1, \ldots, s_i, x, s_{i+1}, \ldots, s_n$. Celem Boba jest wybieranie takich pozycji, by w se-

kwencji pojawił się *kwadrat*; Alicja musi wybierać takie symbole by *kwadratu* nie utworzyć. Na przykład, jeśli ustalimy, że alfabet $A = \{0, 1, 2\}$, a sekwencja $S = 0212$, wtedy Bob wybierając pozycję $i = 1$, zmusza Alicję do stworzenia kwadratu w kolejnym ruchu. Każdy z symboli należących do alfabetu A, wstawiony na pozycję $i = 1$ skutkuje jego utworzeniem: $S^0 = \underline{00}21, S^1 = 0\underline{1212}, S^2 = 0\underline{2}212$. W przypadku gdy Bob obierze dobrą strategię, rozgrywka na 3 symbolach skończy się na ciągu o długości ≤ 5, nie ważne jaką strategię obierze Alicja. Oczywiście im więcej elementów znajduje się w alfabecie A, tym więcej ruchów, będzie potrzebował Bob, by wygrać

Grytczuk, Szafruga i Zmarza [5] udowodnili, że istnieje strategia dla Alicji gwarantująca jej rozegranie dowolnie długiej gry w *Online Thue game* na zbiorze 12 symboli. Swoją definicję oparli o prace A. Kündgena i M. J. Pelsmajera [6], oraz J. Baráta i P. P. Varjú. [1], na temat niepowtarzalnych kolorowań grafów planarnych.

1.5. Komputerowa implementacja Online Thue Game

Komputerowa implementacja gry Thuego opiera się na pomyśle gry z pracy [5]. W grze dostępne są dwa typy gry, zarówno dla jednego oraz dwóch graczy jak i komputerowa symulacja, czyli gra komputera przeciwko niemu samemu.

Implementacja gry *Online Thue Game* nazywana będzie *Longest Free Word*. W grze dostępne są dwa typy gry, pierwszy z nich to *Longest Square-Free Word*, a jego zasady są następujące. Na początku gry, gracze ustalają liczbę symboli wchodzących w skład alfabetu A i otrzymują swoje role. Jeden z nich staje się *budowniczym*, drugi *malarzem*. Rola *budowniczego* polega na wybieraniu pozycji w tworzonym ciągu, na którą wstawiony zostanie wybrany przez *malarza* symbol z alfabetu A. Gra kończy się w momencie, gdy w tworzonym przez graczy ciągu pojawi się *kwadrat*. Numer pozycji i podawanej przez *budowniczego* musi należeć do zbioru

$\{0, 1, \ldots, n\}$, gdzie n to aktualna ilość elementów w ciągu. Na początku gry sekwencja S zawiera jeden losowy symbol, z alfabetu A. Grę rozpoczyna *budowniczy*, a gracze wykonują swoje ruchy na przemian. *Malarz* otrzymuje punkt za każdy wstawiony element, który nie tworzy *kwadratu*. By wyłonić zwycięzcę potrzebne są dwie rundy. Każdy z graczy musi sprawdzić się w obu rolach. Wygrywa osoba, która zdobyła więcej punktów jako *malarz*.

Gra dostępna jest też dla jednego gracza, rolę przeciwnika otrzymuje wtedy algorytm, który działa według ustawionej w pliku konfiguracyjnym taktyki. Może on zarówno pełnić rolę budowniczego jak i malarza. Dodatkową możliwością jest przeprowadzenie rozgrywki pomiędzy dwoma algorytmami.

Bliźniaczym typem 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*. Tutaj podobnie jak w *Longest Square-Free Word* jest możliwość gry przeciwko algorytmowi, oraz przeprowadzenie rozgrywki dwóch algorytmów.

ROZDZIAŁ 2

Aplikacja Longest Free Word

Stworzony przeze mnie program, który powstał na bazie gry *Longest Free Word*, napisany jest w języku *Java*. 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 wybranymi przez użytkownika taktykami. Dodatkową opcją jest uruchomienie obszernego testu, który zasymuluje rozgrywkę komputerowych graczy na wielu kombinacjach poziomów zaawansowania i alfabetów o różnych rozmiarach. Komunikacja z grą odbywa się poprzez plik konfiguracyjny – przed uruchomieniem programu, oraz konsolę – po jego uruchomieniu. Dzięki użyciu narzędzia automatyzującego budowę aplikacji – *Maven*, po pobraniu kodu źródłowego, jesteśmy w stanie uruchomić ją z poziomu terminala dzięki wprowadzeniu zaledwie dwóch krótkich instrukcji. 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, które zmienna akceptuje 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* – 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 – `humanBuil-`

`der`, ponownie z komputerem, jednak tym razem to on jest budowniczym – `pcBuilder`, oraz walka dwóch komputerów – `pcPc`.

- *setPower* – jest to liczba symboli, do których dostęp będzie miał malarz podczas rozgrywki. Zbiór ten wypełniany jest liczbami należącymi do zbioru $\{0, 1, ..., n-1\}$.
- *builderNestingLevel* i *painterNestingLevel* – liczba wywołań rekurencyjnej funkcji podczas podejmowania decyzji jako *budowniczy* i *malarz*. Po ustawieniu tych zmiennych na 0 algorytmy działają zachłannie.
- *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, to rozgrywka zostaje przerwana.
- *makeOverallTest* – gdy opcja ta zostanie ustawiona na `true`, po uruchomieniu aplikacji zostanie przeprowadzony obszerny test, zawierający w sobie kombinacje rozgrywek algorytmów z róznymi wartościami zmiennych *builderNestingLevel*, *painterNestingLevel*, *setPower* i *gameType*. Warto wspomnieć, że podczas tego testu opcja *maxThinkTime* okazała się niezwykle pomocna.
- *randomization* – po ustawieniu tej flagi na `true`, algorytmy używające rekurencji będą losować swoją decyzję spośród opcji, które wydają im się tak samo atrakcyjne. Dzięki temu komputerowi oponenci zachowują się nieprzewidywalnie i ciężko jest rozegrać dwie identyczne partie.

2.2. Algorytm szukający powtórzeń wewnątrz ciągu

Metoda 2.1 ma za zadanie znalezienie kwadratu w sekwencji, którą reprezentuje przekazana w parametrze lista obiektów typu `Integer`. Zmienna `maxSeqSize` reprezentuje długość najdłuższego podciągu jaki się zmieści w sekwencji, jeśli dostawimy za nim podciąg o identycznej długości. Zmienna `minSeqSize` jest to minimalna długość podciągu, który może składać się na *kwadrat*, naturalnie jest równa 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.

Kod 2.1. Metoda szukająca *kwadratów* wewnątrz listy.

```
1  List<Integer> findSquare(List<Integer> sequence) {
2    List<Integer> squareSeq = null;
3    int maxSeqSize = sequence.size()/2;
4    int minSeqSize = 1;
5    for(int subSeqSize=minSeqSize; subSeqSize<=maxSeqSize;subSeqSize++) {
6      squareSeq = compareSubSeq(subSeqSize, sequence);
7      if(squareSeq != null) {
8        return squareSeq;
9      }
10   }
11   return null;
12 }
```

Metoda 2.2, porównuje sąsiadujące ze sobą podciągi i sprawdza czy są takie same. 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 $S = 01201020$, a wcześniej sprecyzowana długość podciągu, składającego się na kwadrat wynosi 2, to wartość jaka zostanie przypisana do zmiennej `comparesFitInSequence` wyniesie $(8+1)-(2*2) = 5$, ponieważ możliwe są następujące porównania: $\underline{01201}020$, $0\underline{12010}20$, $01\underline{20102}0$, $012\underline{01020}$ 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 $2 * \texttt{subSeqSize}$. Jeśli listy są różne to w linii 13 następuje ich wyczyszczenie, po to by pętla mogła porównać dwa kolejne podciągi.

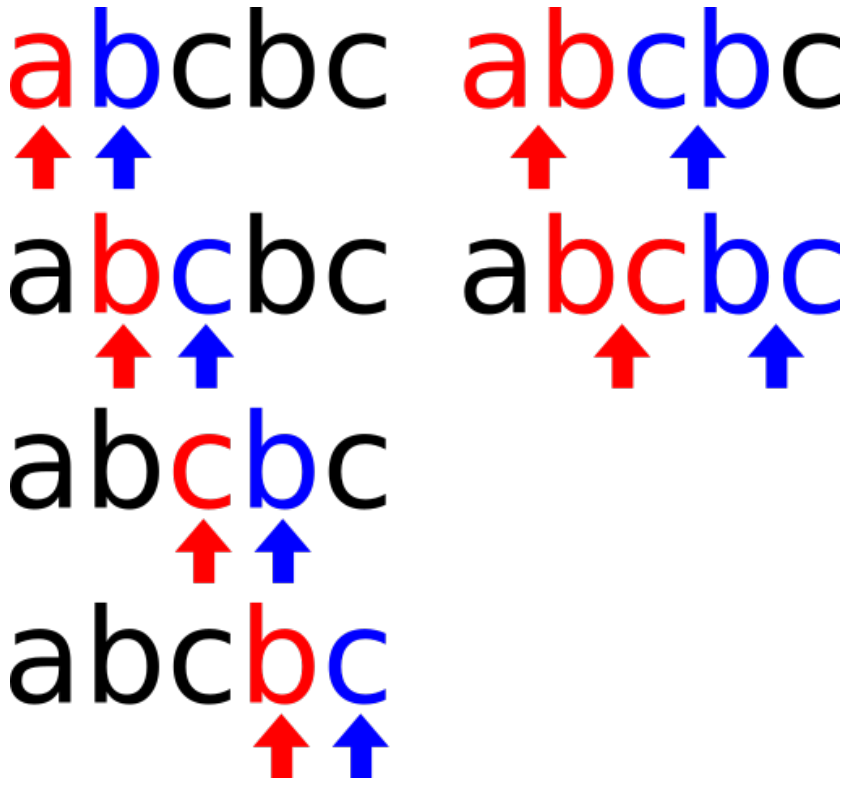
Kod 2.2. Metoda pomocnicza przy szukaniu *kwadratów* wewnątrz listy.

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

Działanie algorytmu szukającego *kwadratów* zilustrowane zostało na rysunku 2.1. Alfabet symboli składa się w tym przypadku ze zbioru liter $\{a, b, c\}$, zamiast ze zbioru cyfr, po to by rysunek był bardziej czytelny. Ciąg

w którym szukamy *kwadratu* to $S = abcbc$, kolumny na rysunku reprezentują pierwszą i drugą iterację pętli z metody 2.1. Wiersze natomiast odpowiadają iteracjom pętli z metody 2.2. Kolorowe symbole na które wskazują strzałki mówią nam o tym, które podciągi są ze sobą porównywane. Jak widzimy w ciągu nie ma *kwadratu* składającego się z podciągów długości 1. Algorytm przechodzi więc do drugiej iteracji i porównuje ze sobą podciągi długości 2. W drugim wierszu, czyli drugiej iteracji pętli z metody 2.2 zostały znalezione dwa takie same podciągi, w tym momencie algorytm kończy swoją pracę i zwraca podciąg bc.

Rysunek 2.1. Działanie algorytmu szukającego *kwadratów*.



Dla typu gry *Longest Overlap-Free Word* potrzebny jest osobny algorytm wyszukujący *nasunięcia*. Metoda 2.3, działa w sposób analogiczny do metody 2.1. Różnice to inne wartości zmiennych `maxSeqSize` i `minSeqSize` oraz metoda porównująca podciągi 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 $S = 0120121$, najdłuższy porównywany ciąg będzie długości 4, ponieważ w ostatniej iteracji pętli będziemy ze sobą porównywali podciągi 0120 oraz 0121, które dzielą ze sobą symbol 0. Wartość zmiennej `minSeqSize` wynosi 3, ponieważ jest to warunkiem stworzenia nasunięcia.

Kod 2.3. Metoda szukająca *nasunięć* wewnątrz listy.

```
1  Subsequence findOverlap(List<Integer> sequence) {
2    Subsequence repeatedSequence = null;
3    int maxSeqSize = (sequence.size()/2)+1;
4    int minSeqSize = 3;
5    for(int subSeqSize=minSeqSize; subSeqSize<=maxSeqSize; subSeqSize++) {
6      repeatedSequence = compareSubSeqOverlap(subSeqSize, sequence);
7      if(repeatedSequence != null) {
8        return repeatedSequence;
9      }
10   }
11   return null;
12 }
```

Metoda 2.4 działa na tej samej zasadzie co metoda 2.2. Różni się tutaj wartość zmiennej `comparesFitInSequence`, jest ona większa o 1, z takiego samego powodu, co zmienna `maxSeqSize` z metody 2.3. Inny jest również podciąg zapisywany do zmiennej `right`, w pętli w 6 linii. Pierwszy indeks owego podciągu jest równy indeksowi ostatniego elementu lewego podciągu, po to by stworzyć nasunięcie.

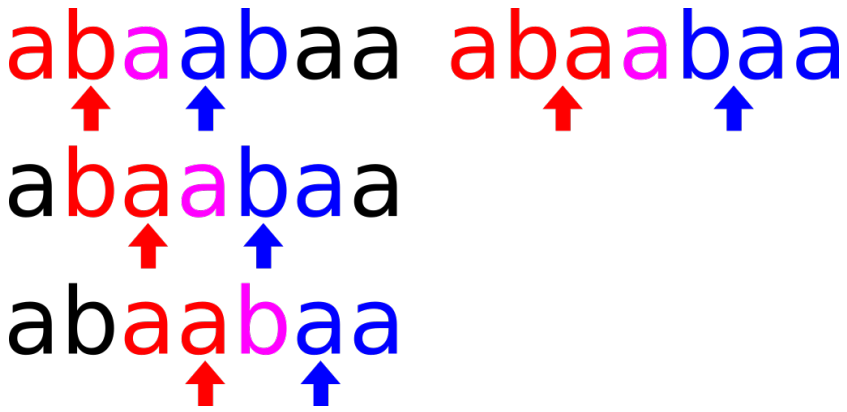
Kod 2.4. Metoda pomocnicza przy szukaniu *nasunięć* wewnątrz listy.

```
1  Subsequence compareSubSeqOverlap(int subSeqSize, List<Integer> sequence) {
2    List<Integer> left = new ArrayList<>();
```

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

Działanie algorytmu ilustruje rysunek 2.2. Tutaj również alfabet symboli składa się ze zbioru liter $\{a, b, c\}$, by poprawić czytelność rysunku. Ciąg wewnątrz, którego szukamy *nasunięcia* to $S = abaabaa$. Kolumny reprezentują pierwszą i drugą iteracjię pętli z metody 2.3, wiersze odpowiadają iteracjom pętli z metody 2.4. Czerwone i niebieskie symbole wskazywane przez strzałki to porównywane podciągi, różowy symbol to element wspólny obu podciągów. *Nasunięcie* znaleziono w drugiej iteracji metody 2.3, przy pierwszym porównaniu – podciągi $abaa$ dzielą ze sobą symbol a. Po znalezieniu *nasunięcia* algorytm zwraca podciąg $abaa$.

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, która służy do wyświetlania jak i wprowadzania treści.

Po uruchomieniu aplikacji na ekranie 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 należącego do zbioru $\{0, 1, \ldots, n\}$, program zażąda podania symbolu, powinien on należeć do naszego alfabetu, czyli symbol musi należeć do zbioru $\{0, 1, \ldots, setPower - 1\}$. 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, wyświetlony zostanie czas rozgrywki, ilość ruchów jaka została do tej pory wykonana oraz indeksy wraz z symbolami, wspomnianego powtórzenia. Przykład rozgrywki przedstawia kod 2.5.

Kod 2.5. Rozgrywka dwóch graczy wypisana w konsoli.

```
#> W grze dostepne sa nastepujace liczby:
0
```

```
1
2
 0: { 0 } 1: { }
## ------------------------------------------------------------ ##
#> Podaj indeks:
1
#> Podaj liczbe:
2
 0: { 0 } 1: { 2 } 2: { }
## ------------------------------------------------------------ ##
#> Podaj indeks:
1
#> Podaj liczbe:
1
 0: { 0 } 1: { 1 } 2: { 2 } 3: { }
## ------------------------------------------------------------ ##
#> Podaj indeks:
3
#> Podaj liczbe:
1
 0: { 0 } 1: { 1 } 2: { 2 } 3: { 1 } 4: { }
## ------------------------------------------------------------ ##
#> Podaj indeks:
4
#> Podaj liczbe:
2
 0: { 0 } 1: { 1 } 2: { 2 } 3: { 1 } 4: { 2 } 5: { }
## ------------------------------------------------------------ ##

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

## ------------------------------------------------------------ ##

#> Rozrgrywka trwala 5 ruchow 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(kod 2.6).

Kod 2.6. Sposób wyświetlania podjętych decyzji algorytmu.

```
#> Komputer wybral indeks: 1  | Czas trwania obliczen: 24.127 s
#> Komputer wybral liczbe: 4  | Czas trwania obliczen: 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 rodzajami taktyk komputerowych graczy za pomocą zmiennych konfiguracyjnych. Jeżeli zmienną *painterNestingLevel* ustawimy na 0, algorytm *malarza* działał będzie zachłannie, wybierając opcje, która jest najatrakcyjniejsza w danym momencie, nie zważając na to, co może wydarzyć się w kolejnej turze. Zasada działania zachłannego algorytmu *malarza* zapisana jest w metodzie 2.7. Definicja metody mówi nam o trzech parametrach jakie są do niej przekazywane:

- `sequence` – utworzony wcześniej ciąg,
- `index` – indeks wybrany przez budowniczoego,
- `power` – moc alfabetu.

Na alfabet, z którego może korzystać *malarz* składają się liczby należące do zbioru $\{0, 1, ..., power - 1\}$. 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*.

Kod 2.7. Metoda zachłannie szukająca symbolu.

```
1  int findRightColorGreedy(List<Integer> sequence, int index, int power) {
2      for(int symbol=0;symbol<power; symbol++) {
3        sequence.add(index, symbol);
4        if(pickProperFind(sequence) == null) {
5          sequence.remove(index);
6          return symbol;
7        } else {
8          sequence.remove(index);
9        }
10     }
11     return -1;
12   }
```

2.5. Zachłanny algorytm wyszukiwania indeksu

By można było przeprowadzić symulację gry, należy wprowadzić również algorytm *budowniczego*, starający się znaleźć najmniej atrakcyjny indeks dla *malarza*. Zachłanny algorytm realizujący to zadanie znajduje się w metodzie 2.8. Metoda 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 lub równa, niż ta zarejestrowana wcześniej. Jeśli tak, do zmiennej `winner` zapisany zostaje aktualny indeks.

Kod 2.8. Metoda zachłannie szukająca indeksu.

```
1  int findRightIndexGreedy(List<Integer> sequence, int power) {
2    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 }
```

2.6. Algorytm wyszukiwania symbolu wykorzystujący rekurencję

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.

Algorytm z metody rekurencyjnie wyszukujący symbolu 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 na predefiniowanej pozycji, następnie iterując w pętli, dodaje każdy z symboli i sprawdza, ile możliwości będzie miał w kolejnej lub kolejnych turach, biorąc pod uwagę wszystkie dostępne pozycje. Każda dodatkowa możliwość to dodatkowy punkt dla wybranego koloru. Sam algorytm składa się z dwóch metod głównych `findRightColorPredicting` (2.9) oraz `simulation` (2.10).

Na początku algorytm inicjalizuje listy `scoreList` oraz `symbolList`. Tę pierwszą tyloma zerami ile jest w grze dostępnych symboli, drugą natomiast symbolami, jakie możemy wstawić na zdefiniowanej przez *budowniczego* pozycji. Linia 4 rozpoczyna pętlę, która iterując po liście `symbolList`, dodaje jej element, wywołuje metodę `simulation`, przekazując utworzony *ciąg*, dodany *symbol*, `scoreList` oraz *poziom 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, a flaga `randomization` ma wartość `true` program wybiera losowy z nich. Dzięki temu w grze występuje przypadkowość, oraz jest mała szansa na powtórzenie dwóch identycznych rozgrywek przy odpowiednio długim ciągu.

Kod 2.9. Metoda szukająca symbolu, wykorzystująca rekurencję.

```
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 }
```

Wewnątrz metody `simulation` nadawane są punkty oraz za pomocą rekurencji wykonywana jest symulacja kolejnych iteracji gry. Na początku dekrementowana zostaje zmienna `invokes` mówiąca o tym, na ile poziomów rekurencji algorytm ma się jeszcze zagłębić. Oznacza to, że jeśli przekazana zmienna `invokes` ustawiona jest na 1, to metoda `simulation` zostanie wywołana tylko raz. Pętla w 3 linii iteruje po pozycjach, na których możliwe jest dodanie symbolu. W 4 linii do listy zapisywane są symbole, które można wstawić na aktualnej pozycji, nie powodując *kwadratu/nasunięcia*. Metoda `updateScoreList` zwiększa ilość punktów symbolu przekazanemu z poprzedniej metody. Ilość punktów jest równa liczbie elementów, zmiennej `symbolList`. Pętla w 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.

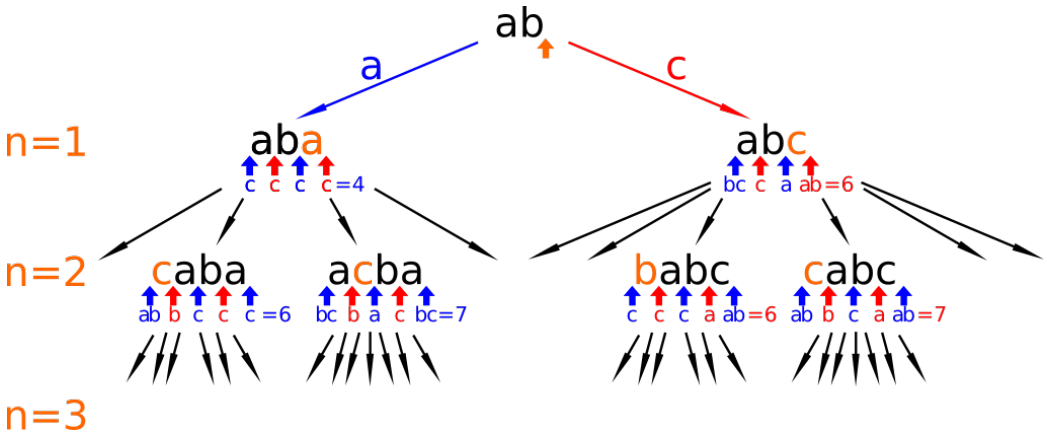
Kod 2.10. Metoda rekurencyjnie wykonująca symulacje kolejnych ruchów.

```
1  void simulation(List<Integer> sequence, int indexInScoreList, List<Integer>
       scoreList, int invokes) {
2    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 }
```

Działanie algorytmu przedstawione zostało na rysunku 2.3. Po raz kolejny w celu zwiększenia przejrzystości rysunku alfabet symboli składa się ze zbioru liter $\{a, b, c\}$, a nie z liczb jak ma to miejsce podczas zwykłej rozgrywki. Mamy ciąg $S = ab$, *budowniczy* wybrał indeks 2, zatem *malarz* wybiera symbol jaki zostanie wstawiony za literką b. Typ gry to *Longest Square-Free Word*. Wstawienie symbolu b, stworzyłoby kwadrat, dlatego algorytm rozważa literki a oraz c. Zmienna n jest to poziom rekurencji, natomiast liczby przy pasujących symbolach oznaczają punkty, jakie zostają przypisane symbolowi a lub c.

Już przy pierwszym poziomie rekurencji widać, że literka c jest atrakcyjniejsza, ponieważ wstawienie jej zapewnia *malarzowi* 6 możliwości w kolejnych rundach, podczas gdy literka a osiągnęła wynik 4. 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.

Rysunek 2.3. Działanie algorytmu rekurencyjnego *malarza*.



2.7. Algorytm wyszukiwania indeksu wykorzystujący rekurencję

Sposób działania algorytmu rekurencyjnego 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, ustalamy punktację iterując po wszystkich dostępnych pozycjach, zamiast sprawdzać ją dla jednej ustalonej przez przeciwnika, oraz punkty nadawane zostają pozycjom i zwracamy tą, która uzyska ich najmniej. Druga część algorytmu czyli metoda `simulation` (2.10) pozostaje bez zmian.

Na początku metody 2.11 zainicjalizowana zostaje zmienna `scoreList`. Do listy dodane zostają zera w ilości odpowiadającej długości ciągu plus jeden, bo właśnie na tylu pozycjach możemy dodać nowy element. W 3 linii iterujemy po każdej dostępnej pozycji, natomiast od 4 linii mamy już wszystko to co w algorytmie szukającym symbolu. Na końcu metody zwracana zostaje pozycja, która uzyskała najmniej punktów, czyli będzie najmniej atrakcyjna dla *malarza*. Tak jak poprzednio jeśli istnieje więcej niż jedna pozycją z minimalną wartością, a flaga `randomization` ustawiona jest na `true`, to element jest wybierany losowo.

Kod 2.11. Metoda szukająca pozycji, wykorzystująca rekurencję.

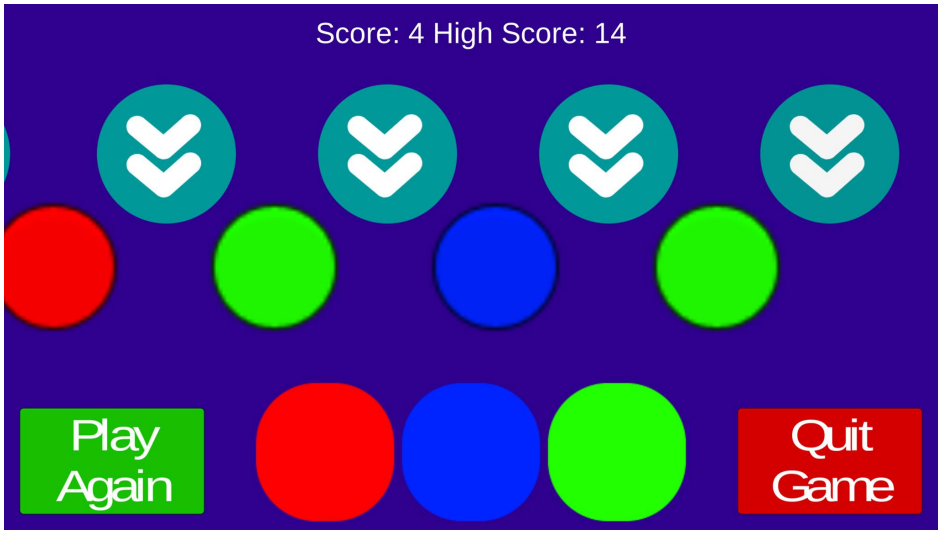
```
1  int findRightIndex(List<Integer> sequence) {
2    List<Integer> scoreList = initScoreList(sequence.size() + 1);
3    for (int i = 0; i<sequence.size() + 1; i++) {
4      List<Integer> symbols = getFitableSymbolList(sequence, i);
5      for (int symbol : symbols) {
6        sequence.add(i, symbol);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

```
11    return getRandomMinFromPredict(scoreList);
12  }
```

2.8. Longest square free word z interfejsem graficznym

Napisana została przeze mnie również druga wersja gry. Użytkownik nie korzysta w niej z konsoli, lecz z graficznego interfejsu. W tej odmianie gracz za pomocą trzech kolorów ma stworzyć jak najdłuższy ciąg bez *kwadratów*. Czynności jakie należy wykonywać podczas rozgrywki to wybranie miejsca, w które zostanie wstawiony element oraz wybranie jego koloru. 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 tego z podstawowej wersji gry. Wygląd aplikacji został przedstawiony na rysunku 2.4.

Rysunek 2.4. *Longest Free Word* z interfejsem graficznym.



Aplikacja została napisana na popularnym silniku do tworzenia gier – *Unity3d*. 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 wyświetlonych szczegółów.

ROZDZIAŁ 3

Analiza symulowanych potyczek

Dobrym sposobem by poznać aplikację jest jej uruchomienie. Jeszcze lepszym, przeprowadzenie jej analizy oraz zobrazowanie wyników za pomocą wykresów i tabel. Rozdział ten poświęcony jest badaniu działania aplikacji *Longest Free Word*. Badania skupiają się na wariancie aplikacji opierającym się na tworzeniu ciągu bez *kwadratów*, a flaga *randomization* ustawiona została na `true`. Wszystkie testy uruchomione były na zdalnym serwerze, stworzonym za pośrednictwem serwisu *digitalocean.com*. Podzespoły maszyny to system operacyjny Ubuntu 15.10, procesor 2,6 Ghz Intel Core i7, pamięć ram 16 GB 1600 MHz DDR3 i dysk SSD.

3.1. Ilość symboli potrzebna na rozegranie partii

Długość gry zależna jest od dwóch czynników. Ustawionego poziomu zagnieżdżenia i ilości dostępnych symboli. Thue w swoim twierdzeniu [2], udowodnił, że jesteśmy w stanie stworzyć nieskończenie długi ciąg bez *kwadratów*, mając do dyspozycji alfabet składający się z 3 symboli. Sytuacja zmienia się, gdy ciąg tworzony jest przez dwóch graczy, z których jeden stara się utworzyć *kwadrat*. Jeśli rolę *budowniczego* pełnił będzie zachłanny algorytm, a rolę *malarza* przejmie algorytm wykorzystujący siedmiopoziomową rekurencję, to przy takim alfabecie, rozgrywka i tak zawsze trwała będzie maksymalnie 5 ruchów.

Dla alfabetu składającego się z 4 symboli, algorytmy również nie są w stanie rozegrać zbyt długiej partii. Przy rozgrywce, w której udział bierze zachłanny algorytm pełniący rolę *budowniczego*, niezależnie taktyki wy-

branej przez algorytm *malarza*, rozgrywka zakończy się maksymalnie po 10 ruchu.

Tabela 3.1, przedstawia jakim wynikiem kończy się każda z rozgrywek po przypisaniu odpowiednich wartości zmiennym konfiguracyjnym *setPower*, *builderNestingLevel* i *painterNestingLevel*. Wartym zauważenia jest, że przy alfabecie o mocy 4, zachłanny algorytm pełniący rolę *malarza* (*painterNestingLevel* $= 0$), jest w stanie rozegrać partię na 10 ruchów, a rozgrywka algorytmów rekurencyjnych (*painterNestingLevel* ≥ 1), kończy się po 7 ruchach. Dowodzi to tego, że taktyka wykorzystująca rekurencje nie jest doskonała i nie zawsze jest najlepszą opcją. Pozostałe warianty nie zostały przedstawione, gdyż rozgrywka w ich przypadku nie skutkowała stałym wynikiem. Oznacza to, że taktyka *budowniczego* nie była wystarczająco dobra by za każdym razem pokonać *malarza* w takiej samej ilości ruchów.

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

setPower	*painterNestingLevel*	*builderNestingLevel*	Długość gry
3	≥ 0	≥ 0	5 ruchów
4	0	0	7 ruchów
4	≥ 1	≥ 1	7 ruchów
4	0	≥ 1	10 ruchów

Dopiero rozgrywka na 5 symbolach daje algorytmom pole do popisu, bowiem w przeprowadzonych badaniach, zależnie od ustawionych wartości zmiennych *builderNestingLevel* i *painterNestingLevel*, trwała ona od 11 do conajmniej 236 ruchów. Wykorzystywanie zachłannego algorytmu do pełnienia roli *malarza* (*painterNestingLevel* $= 0$) przy alfabecie pięcioelementowym, nie skutkuje zbyt długą grą. Niezależnie od wartości zmiennej *builderNestingLevel*, *kwadrat* pojawi się zawsze po 11 kroku. Jednak podczas prób, gdy wartość zmiennej *painterNestingLevel* ustawiona była na 1, a algorytm *budowniczego* działał zachłannie (*builderNestingLevel* $= 0$), rozgrywka trwała od 44 do 236 tur. Średnia arytmetyczna obliczona z ilości

ruchów w badanych rozgrywkach wyniosła $133,83$. Dane analizowanych rozgrywek znajdują się w tabeli 3.2.

Tabela 3.2. Podsumowanie badanych rozgrywek, przy konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 0$, *painterNestingLevel* $= 1$.

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

Przy wartości zmiennej *builderNestingLevel* równej 0 i zmiennej *painterNestingLevel* równej 1, podczas 10 prób rozgrywka trwała od 30 do 213 ruchów. Średnia arytmetyczna obliczona z ilości ruchów w analizowanych rozgrywkach wyniosła $96,9$, a więc jest o $37,23$ niższa, niż gdy rolę *budowniczego* pełnił algorytm zachłanny. Dane analizowanych rozgrywek znajdują się w tabeli 3.3.

Obecne zasoby sprzętowe i czasowe sprawiły, że algorytm pełniący rolę *malarza*, z wartością zmiennej *painterNestingLevel* $= 2$, okazał się niepokonany w walce z zachłannym algorytmem budowniczego. Przy ponad 10 próbach ani razu nie był zmuszony do stworzenia *kwadratu* w rozgrywkach trwających ≤ 220 tur i około 26 godzin.

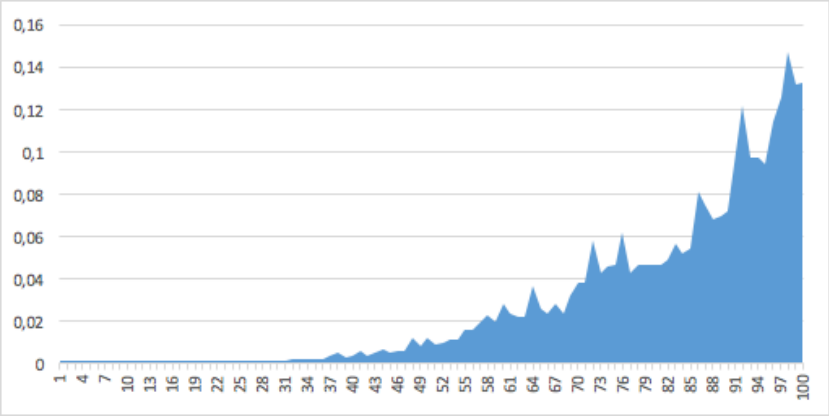
Tabela 3.3. Podsumowanie badanych rozgrywek, przy konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 1$, *painterNestingLevel* $= 1$.

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

3.2. Pomiary czasów potrzebnych na podjęcie decyzji

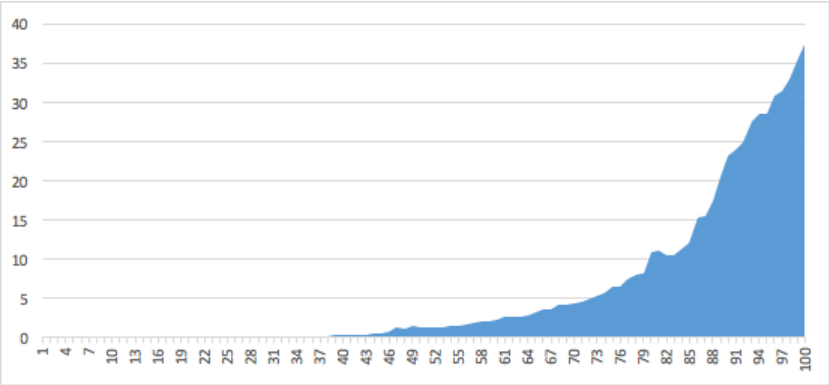
Nieodłącznym elementem analizy działania aplikacji jest badanie czasu w jakim wykonuje ona swoje algorytmy. Nie inaczej jest w przypadku *Longest Free Word*. Najkrótrzy zbadany czas odpowiedzi algorytmu wyniósł jedną tysięczną sekundy, najdłuższy natomiast około dwóch godzin. Czas potrzebny na podjęcie decyzji 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. Liczba sekund jakiej potrzebował algorytm *budowniczego* na podjęcie decyzji, przy poszczególnych długościach ciągu i konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 0$, *painterNestingLevel* $= 1$.

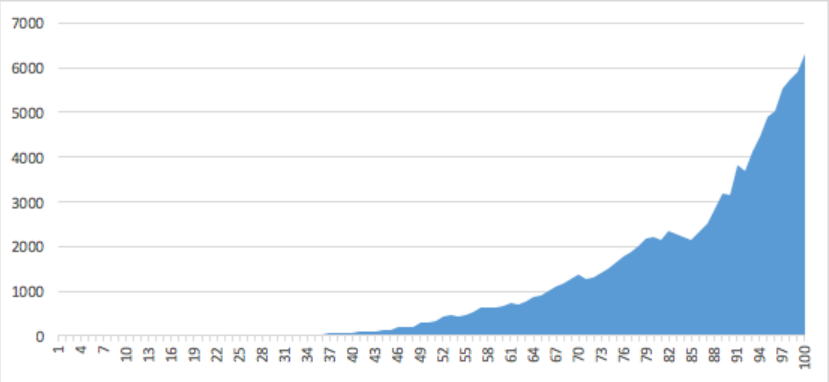


Warto zauważyć jak ogromne różnice w wydajności pojawiają się, gdy wartości zmiennych sterujących rekurencją są różne. Algorytmowi pełniącemu rolę *budowniczego* przy wartości zmiennej *builderNestingLevel* równej 2, dla ciągu długości 100, podjęcie decyzji zajęło około 170 razy więcej czasu, niż po ustawieniu wartości tej zmiennej na 1. Wykresy 3.2 i 3.3, przedstawiają różnice w złożonościach czasowych obu rozgrywek.

Rysunek 3.2. Liczba sekund jakiej potrzebował algorytm *budowniczego* na podjęcie decyzji, przy poszczególnych długościach ciągu i konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 1$, *painterNestingLevel* $= 1$.



Rysunek 3.3. Liczba sekund jakiej potrzebował algorytm *budowniczego* na podjęcie decyzji, przy poszczególnych długościach ciągu i konfiguracji: $setPower = 5$, $builderNestingLevel = 2$, $painterNestingLevel = 1$.

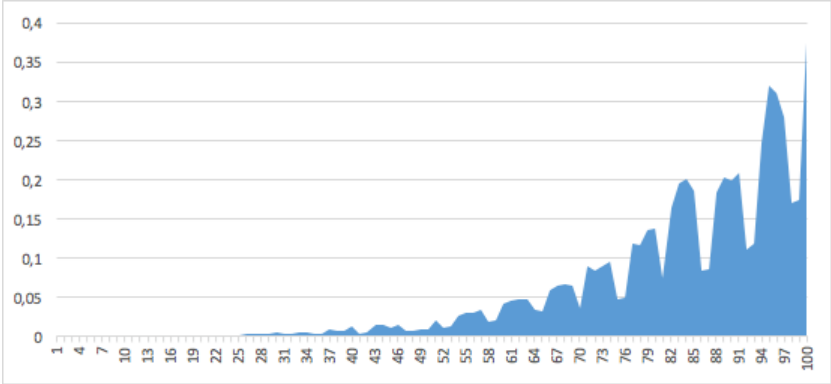


Odrobinę inaczej wyglądają analogiczne wykresy przedstawiające złożoności algorytmów pełniących rolę *malarza*. Algorytm korzystający z rekurencji symuluje dodawanie kolejnych symboli na wyznaczonej pozycji, jeżeli jednak w danym miejscu da się wstawić tylko jeden symbol, to algorytm przeprowadzi mniej symulacji, niż jeśli dałoby się tam wstawić pięć symboli. Dzięki tej zależności na wykresie można zauważyć, przy których ruchach *malarz* jest bliski przegranej, a przy których ma wiele dostępnych opcji. Wykres 3.4 przedstawia czasy potrzebne algorytmowi pełniącemu rolę *malarza* na podjęcie decyzji. Jak widać zachłanny algorytm działający jako *budowniczy* cyklicznie redukuje liczbę opcji *malarza*.

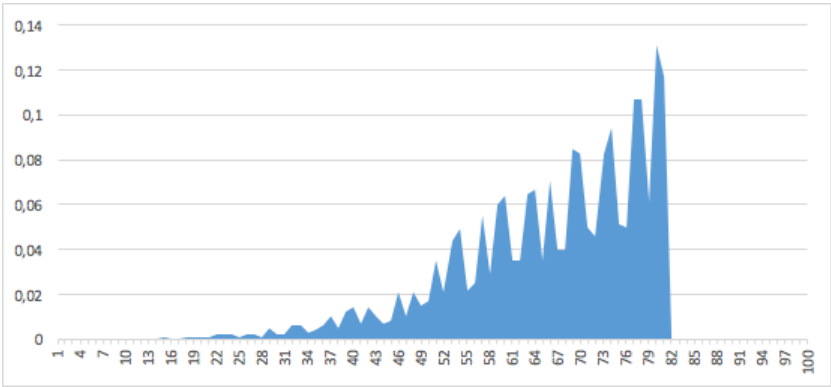
Jeśli przeanalizujemy wykres 3.5 zauważymy, że podczas rozgrywki z algorytmem *budowniczego*, który używa rekurencji, krzywe są o wiele bardziej spiczaste, co oznacza, że dla *malarza* jest to trudna rozgrywka, która ostatecznie kończy się w 82 ruchu. Ponownie możemy porównać ze sobą czasy jakie były potrzebne na podjęcie decyzji dla algorytmu wykorzystującego jednopoziomową rekurencję na wykresie 3.5, z algorytmem wykorzystującym dwupoziomową rekurencję na wykresie 3.6. Wykonanie ruchu, dla ciągu długości 79 pierwszemu algorytmowi zajęło 0.061 sekundy, natomiast temu bardziej złożonemu 17.711 sekund. Zatem w tym konkretnym

ruchu algorytm korzystający z rekurencji głębszej o jedno wywołanie, decydował się około 290 razy dłużej.

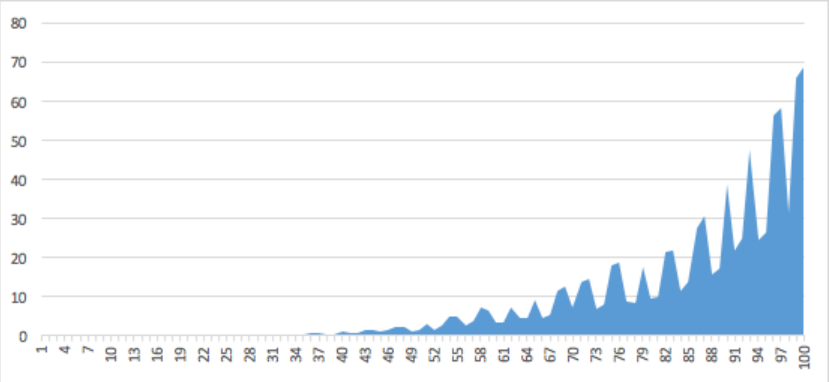
Rysunek 3.4. Liczba sekund jakiej potrzebował algorytm *malarza* na podjęcie decyzji, przy poszczególnych długościach ciągu i konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 0$, *painterNestingLevel* $= 1$.



Rysunek 3.5. Liczba sekund jakiej potrzebował algorytm *malarza* na podjęcie decyzji, przy poszczególnych długościach ciągu i konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 2$, *painterNestingLevel* $= 1$.



Rysunek 3.6. Liczba sekund jakiej potrzebował algorytm *malarza* na podjęcie decyzji, przy poszczególnych długościach ciągu i konfiguracji: *setPower* $= 5$, *builderNestingLevel* $= 0$, *painterNestingLevel* $= 2$.



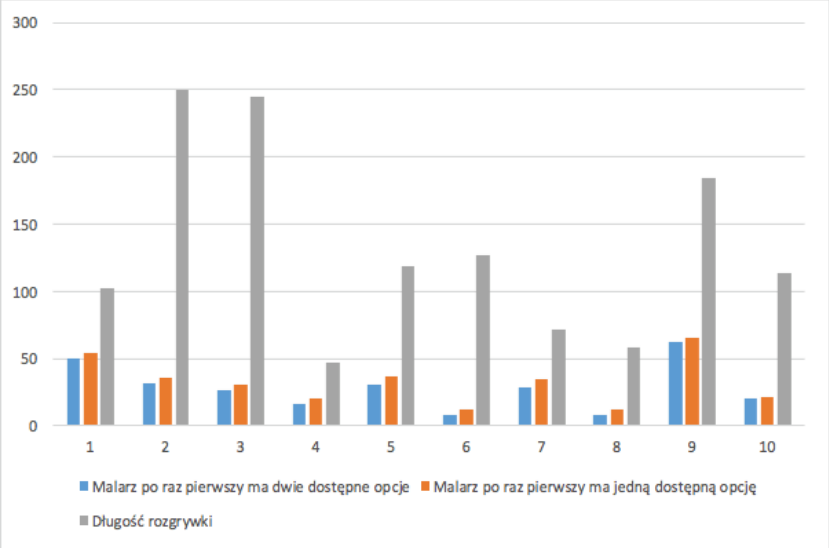
3.3. Porównanie zachowań algorytmu z zagnieżdżeniami i bez zagnieżdżeń

Jeśli przyjrzymy się dokładnie działaniu zachłannego algorytmu wyszukiwania pozycji, to bez trudu zauważymy, że w początkowej fazie gry, kiedy *malarz* ma dużą swobodę w wybieraniu elementów, pozycja, która jest wybierana to w przeważającej większości ostatni indeks ciągu. Spowodowane jest to tym, że pozycje 0 oraz $n + 1$, otoczone są elementami tylko z lewej lub prawej strony, dzięki temu *malarz* ma na nich do wyboru większą gamę symboli. Na reszcie pozycji zazwyczaj dostępne jest tyle samo symboli, zwracany jest więc ostatni z nich. Taktyka tego typu została przeze mnie nazwana *taktyką iteracyjną* i nie jest zbyt emocjonująca. Jeśli nie mamy zbyt dużej puli symboli do wyboru, to po pewnym czasie następuje wyłamanie się z tego stylu gry. Wyłamanie pojawia się zawsze w momencie, gdy algorytm pełniący rolę *budowniczego* znajduje indeks z mniejszą ilością możliwości.

Przeprowadzone przeze mnie badania, przy alfabecie pięcioelementowym, zmiennych *painterNestingLevel* ustawionej na 1, i *builderNestingLevel* ustawionej na 0, wykazały, że w przeciągu kilku kroków od pierw-

szego znalezienia pozycji, pod którą dostępne są dwa symbole, pojawia się pozycja pod którą dostępny jest tylko jeden symbol. Nie wykryłem natomiast korelacji, pomiędzy pierwszym wyłamaniem się z *taktyki iteracyjnej*, a długością rozgrywki. Wykres przedstawia 10 rozgrywek, na którym opisane są powyższe zależności.

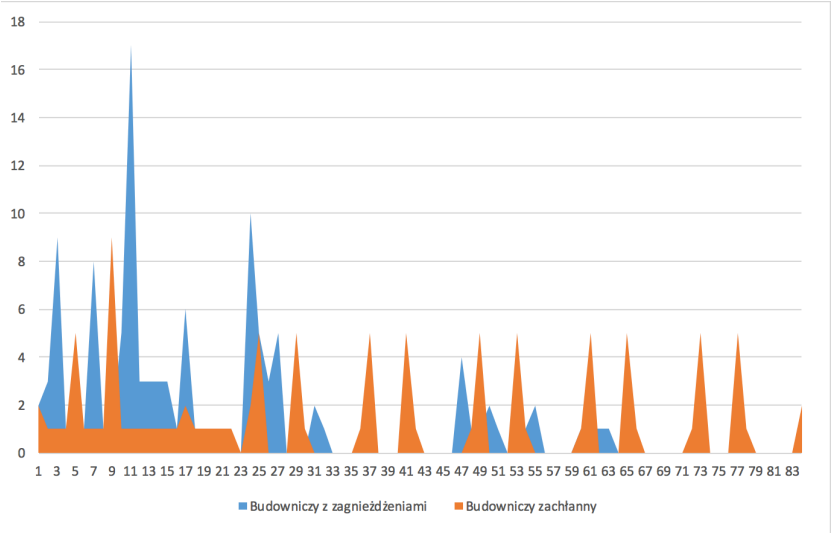
Rysunek 3.7. Stosunek pierwszego wystąpienia indeksu z dwoma dostępnymi elementami, do pojawienia się indeksu z jednym dostępnym elementem i długości rozgrywki. Użyty został pięcioelementowy alfabet, rolę *budowniczego* pełnił zachłanny algorytm, a *malarza* algorytm wykorzystujący jednopoziomową rekurencję.



Ciężko jest dopatrzyć się podobnej cykliczności w algorytmie wykorzystującym rekurencję. Algorytm ten skupia się raczej na grupie pozycji będących obok siebie. Wykres 3.8 przedstawia rozkład wybieranych pozycji przez oba algorytmy. Rozgrywki nie trwały dłużej niż 115 ruchów, na rysunku można zaobserwować, że najczęściej wybierane pozycje przez algorytm rekurencyjny należały do zbioru liczb ≤ 27. Natomiast pozycje wybierane przez algorytm zachłanny, są w miarę równomiernie rozłożone po całej szerokości wykresu. Ponadto, mimo wyłamania się z wcześniej wspomnianej

taktyki iteracyjnej, algorytm zachłanny w dalszym ciągu działa cyklicznie, wybierając pozycje po zero, jeden lub pięć razy.

Rysunek 3.8. Porównanie rozkładu wybieranych pozycji przez algorytm *budowniczego* używający jednopoziomowej rekurencji oraz algorytm *budowniczego* działający zachłannie.



Zakończenie

Twierdzenia Axela Thue na temat sekwencji symboli okazały się doskonałą bazą do dalszych rozważań. Wykorzystali je J. Grytczuk, P. Szafruga i M. Zmarz, tworząc grę dla dwóch graczy wykorzystującą dotychczasowe koncepcje. Podczas pisania pracy dyplomowej, postanowiłem podejść do ich pomysłu od strony programistycznej. Napisana przeze mnie aplikacja obsługuje logikę gry, wprowadza szereg funkcji ułatwiających analizę przeprowadzonych gier, a co najważniejsze, za pomocą algorytmów jest w stanie zasymulować rozgrywkę graczy, korzystając z wprowadzonych przeze mnie taktyk. Można dodatkowo modyfikować stworzone przeze mnie taktyki, przez zmienne konfiguracyjne, które sterują wykorzystywanym poziomem rekurencji. Ustawienie większego poziomu zagnieżdżenia skutkowało lepszymi rezultatami w rozgrywce. Mimo to nie byłem w stanie rzucić nowego światła na problem ilości symboli potrzebnej na dowolnie długą grę. Główną tego przyczyną jest brak zasobów sprzętowych, które pozwoliłyby na przeprowadzenie większej ilości obliczeń.

Bibliografia

[1] János Barát and Péter Varjú. On square-free vertex colorings of graphs. *Studia Scientiarum Mathematicarum Hungarica*, 44(3):411–422, 2007.

[2] Jean Berstel. Axel thue's work on repetitions in words. *Séries formelles et combinatoire algébrique*, 11:65–80, 1992.

[3] Arturo Carpi. On the size of a square-free morphism on a three letter alphabet. *Information processing letters*, 16(5):231–235, 1983.

[4] Jarosław Grytczuk, Jakub Kozik, and Piotr Micek. New approach to nonrepetitive sequences. *Random Structures & Algorithms*, 42(2):214–225, 2013.

[5] Jarosław Grytczuk, Piotr Szafruga, and Michał Zmarz. Online version of the theorem of thue. *Information Processing Letters*, 113(5):193–195, 2013.

[6] Andre Kündgen and Michael J Pelsmajer. Nonrepetitive colorings of graphs of bounded tree-width. *Discrete Mathematics*, 308(19):4473–4478, 2008.

[7] Trygve Nagell, Atle Selberg, Sigmund Selberg, and Knut Thalberg. Selected mathematical papers of axel thue. *Universitetsforlaget, Oslo*, 1977.

[8] Eugéne Prouhet. Mémoire sur quelques relations entre les puissances des nombres. *CR Acad. Sci. Paris*, 33(225):1851, 1851.

[9] Narad Rampersad. *Overlap-free words and generalizations*. PhD thesis, University of Waterloo, 2007.

Spis tabel

Spis rysunków

Spis kodów źródłowych

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