Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka Algorytmiczna**
Specjalność: **-**

Praca Dyplomowa Inżynierska

Aplikacja rozwiązująca nonogramy

Mateusz Wałejko

Opiekun pracy
dr Marcin Michalski

Słowa kluczowe: nonogramy, solver, aplikacja

Wrocław 2022

Spis treści

Spis rysunków

Spis tabel

Wstęp

Praca poświęcona jest aplikacji rozwiązującej nonogramy. W pracy opisane są zagadnienia teoretyczne potrzebne do zrozumienia problemu, układ i sposób użycia samej aplikacji, a także porównanie podejść do rozwiązywania nonogramów.

Praca podzielona jest na 7 części.

W pierwszej części opisany jest problem rozwiązywania nonogramów. Wytłumaczone jest czym są nonogramy. Podane są definicje potrzebne do zrozumienia problemu. Na końcu, za pomocą opisanych zagadnień wyznaczona jest trudność problemu.

Druga część zawiera schematyczny opis aplikacji — listę wymagań spełnianych przez aplikację oraz zarys nawigacji po niej.

W trzecim rozdziale zawarte są informacje dotyczące implementacji: użyte technologie wraz z ich opisem oraz schemat wykorzystanej bazy danych.

Czwarta część skupia się na opisie solvera wykorzystanego w aplikacji. Nakreślona jest ewolucja solvera w trakcie rozwoju aplikacji, zaproponowane są także heurystyki mogące usprawnić działanie solvera. Różne implementacje wraz z usprawnieniami są poddane testom, by ocenić ich wydajność, oraz podane są wnioski wyciągnięte z badań.

Piąty rozdział to krótka instrukcja dla użytkownika, opisująca wymagania do uruchomienia aplikacji.

Na końcu, w szóstej części, zawarte jest podsumowanie pracy oraz kierunki, w których można dokonać dalszego rozwoju aplikacji.

Rozdział 1

Analiza problemu

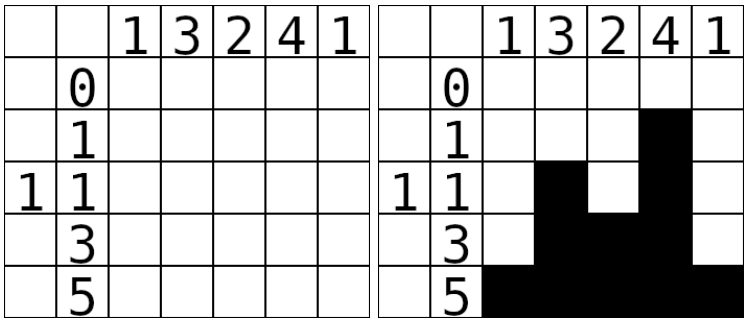
W tym rozdziale przedstawiona jest definicja nonogramów oraz sposób ich rozwiązywania. W kolejnej części rozdziału wypisane są także definicje potrzebne do zrozumienia złożoności problemu, jakim jest rozwiązywanie nonogramów.

1.1 Przedstawienie nonogramów

1.1.1 Opis

Nonogramy (znane także jako *Paint by Number* oraz *Picross*) to łamigłówki, w których celem jest odpowiednie wypełnienie komórek na siatce tak, by uzyskać określony wzór (np. obrazek). W tym celu należy kolorować pola zgodnie ze wskazówkami umieszczonymi obok każdego wiersza oraz kolumny na siatce. Wskazówki mają postać ciągu liczb — każda z liczb oznacza ilość wypełnionych komórek z rzędu, a pomiędzy grupami wypełnionych komórek znajduje się przynajmniej jedna pusta komórka.

Uściślając powyższą, nieformalną definicję, nonogram to łamigłówka na siatce wielkości $w \times h$, gdzie w oznacza szerokość planszy wyrażoną w ilości komórek, a h wysokość planszy wyrażoną w ilości komórek. Dla każdego wiersza i kolumny mamy przedstawiony ciąg liczb H_n będący wskazówką dla danej linii. Dany element h_i opisuje blok stworzony z h_i wypełnionych komórek z rzędu i, jeśli h_i nie jest pierwszym elementem ciągu, to blok opisany przez h_i jest oddzielony przynajmniej jedną pustą komórką od bloku opisanego przez h_{i-1}, oraz jeśli h_i nie jest ostatnim elementem ciągu, to blok opisany przez h_i jest oddzielony przynajmniej jedną pustą komórką od bloku opisanego przez h_{i+1}. Linię spełniającą zadaną wskazówkę opisuje wyrażenie regularne: $l = \hat{}0^*1^{h_1}0^+1^{h_2}0^+ \ldots 0^+1^{h_n}0^*\$$, gdzie 0, 1 oznaczają odpowiednio pustą i wypełnioną komórkę, h_i jest i-tym elementem ciągu H_n, będącego wskazówką dla wiersza/kolumny, a $|l| = n$, gdzie $n = w$ dla wiersza i $n = h$ dla kolumny. Rozwiązaniem nonogramu jest wypełnienie zadanej planszy w taki sposób, by dla każdego wiersza oraz każdej kolumny wskazówki dla nich były spełnione.



(a) przed rozwiązaniem (b) po rozwiązaniu

Rysunek 1.1: Przykładowa plansza

1.1.2 Definicje

Definicja 1.1 *Wskazówką nazywamy ciąg H_i liczb, opisujący ułożenie wypełnionych komórek w danej linii.*

Uwaga. Dla uproszczenia opisu algorytmów następuje założenie, że pusta linia jest opisywana przez wskazówkę będącą ciągiem pustym.

Definicja 1.2 *Instancją problemu nonogramu jest czwórka $N = (w, h, R_n, C_n)$, gdzie w, h to odpowiednio szerokość i wysokość planszy, a R_n i C_n są ciągami wskazówek dla wierszy oraz kolumn.*

1.2 Wymagane zagadnienia matematyczne

1.2.1 Problem decyzyjny

Definicja 1.3 *Problem decyzyjny to problem, na który odpowiedź stanowi 'tak' lub 'nie'.*

Problem decyzyjny to pojęcie kluczowe dla klasyfikacji problemu do wybranej klasy złożoności. By móc sklasyfikować wybrany problem (np. rozwiązanie nonogramu) do jakiejś klasy złożoności, należy przedstawić go w postaci problemu decyzyjnego.

Przykład 1.1 *Czy zadany nonogram $N = (w, h, R_n, C_n)$ ma rozwiązanie?*

1.2.2 Klasa złożoności P

Definicja 1.4 *Klasa złożoności P zawiera wszystkie problemy decyzyjne, których rozwiązanie można znaleźć w czasie wielomianowym.*

Przykład 1.2 *Znalezienie najkrótszej ścieżki między dwoma punktami w grafie należy do klasy P, ponieważ algorytm Dijkstry znajduje najkrótszą ścieżkę w czasie wielomianowym. Sformułowanie tego problemu w postaci problemu decyzyjnego mogłoby brzmieć następująco: Czy dla danego wejściowego grafu $G = (V, E)$ istnieje ścieżka z punktu $v_1 \in V$ do punktu $v_2 \in V$ o długości nie większej niż x?*

1.2.3 Klasa złożoności NP

Definicja 1.5 *Klasa złożoności NP zawiera wszystkie problemy decyzyjne, których rozwiązanie dla odpowiedzi pozytywnej można zweryfikować w czasie wielomianowym.*

Przykład 1.3 *Mając zbiór I przedmiotów, gdzie przedmiot i_n to dwójka (v_n, w_n), gdzie v_n to wartość, a w_n to waga, oraz ograniczenie górne na sumę wag wybranych przedmiotów w_{max}, czy można wybrać przedmioty w taki sposób, by nie przekroczyć limitu wagi w_{max}, a by suma wartości wybranych przedmiotów była większa lub równa c?*

Tak zadany problem to wersja decyzyjna problemu plecakowego. Być może nie istnieje algorytm znajdujący przydział przedmiotów w czasie wielomianowym, ale mając przedstawione rozwiązanie $S \subseteq I$ można zsumować wartości przedmiotów z S i sprawdzić, czy jest to poprawne rozwiązanie.

Należy zauważyć, że każdy problem z klasy P należy także do klasy NP, ponieważ rozwiązanie problemu decyzyjnego jest jednym ze sposobów weryfikacji poprawności jego rozwiązania. To czy $P = NP$ jest jak do tej pory nierozwiązanym problemem.

1.2.4 Redukcja wielomianowa

Definicja 1.6 *Problem A jest redukowalny do problemu B w czasie wielomianowym, jeśli wejścia dla problemu A można przekształcić na wejścia dla problemu B w czasie wielomianowym, a następnie rozwiązać problem A wywołując procedurę rozwiązującą problem B wielomianową ilość razy.*

Przykład 1.4 *Mnożenie liczb a·b można zdefiniować za pomocą operacji dodawania w następujący sposób:*

$$a \cdot b = \underbrace{a + a + \ldots + a}_{b}$$

Należy zauważyć, że jeśli problem A jest redukowalny do problemu B w czasie wielomianowym, a problem B należy do klasy P, to problem A także należy do klasy P, jako że sposobem na jego rozwiązanie jest użycie redukcji wielomianowej, by traktować go jako instancję problemu B, a następnie rozwiązanie go za pomocą algorytmu działającego w czasie wielomianowym.

Wniosek 1.1 *Jeśli istnieje redukcja z A w B w czasie wielomianowym, to B jest co najmniej tak złożony, jak A.*

1.2.5 Klasa problemów NP-trudnych

Definicja 1.7 *Problem H należy do klasy problemów NP-trudnych, jeśli każdy problem w klasie NP jest redukowalny do H w czasie wielomianowym.*

W przypadku klasy problemów NP-trudnych nie ma wymogu, by należące do niej problemy były problemami decyzyjnymi.

Przykład 1.5 *Przykładem problemu NP-trudnego jest problem spełnialności (SAT): 'Czy dla danej formuły logicznej istnieje wartościowanie, dla którego zadana formuła jest spełniona?'. Przynależność tego problemu do tej klasy została udowodniona w 1971 roku przez Stephena Cooka i Leonida Levina w dowodzie twierdzenia Cooka-Levina [21].*

Dla udowadniania przynależności problemu do tej klasy kluczowa jest obserwacja, że istnienie redukcji wielomianowej z A w B implikuje przynależność B do tej klasy, o ile A także do niej należy.

1.2.6 Klasa problemów NP-zupełnych

Definicja 1.8 *Problem decyzyjny C należy do klasy problemów NP-zupełnych, jeśli należy do klas problemów NP-trudnych oraz NP.*

Wniosek 1.2 *Pokazanie, że problem decyzyjny A jest NP-zupełny sprowadza się do pokazania, że istnieje redukcja wielomianowa z problemu H z klasy problemów NP-trudnych oraz że rozwiązanie problemu A można zweryfikować w czasie wielomianowym.*

1.3 Przypisanie problemu rozwiązania nonogramów do odpowiedniej klasy złożoności

Mając zdefiniowane pojęcia potrzebne do klasyfikacji problemu do odpowiedniej klasy złożoności, należy znaleźć klasę, do jakiej należy rozwiązywanie nonogramów. Z uwagi na specyfikę klas, klasyfikacji poddana zostanie decyzyjna wersja problemu, tj. *'Czy zadany nonogram $N = (w, h, R_n, C_n)$ ma rozwiązanie?'*.

1.3.1 Problem rozwiązania nonogramu jest w NP

Niech $M_{h,w}$ będzie macierzą oznaczająca rozwiązanie zadanego nonogramu $N = (w, h, R_n, C_n)$, gdzie $m_{i,j}$ oznacza stan komórki w wierszu i i kolumnie j, oraz $m_{i,j} = 1$, jeśli komórka jest wypełniona, a $m_{i,j} = 0$, jeśli komórka jest pusta. Macierz $M_{h,w}$ jest mapowana na $h + w$ list, będących ciągami stanów komórek w kolejnych wierszach, i kolumnach.

Do weryfikacji rozwiązania użyjemy następującej procedury:

Pseudokod 1.1: Poprawność rozwiązania w osi

```
Input: Lista linii L, lista wskazówek LH, długość linii n
Output: Poprawność rozwiązania w osi (true/false)
1  for i ← 1 to |L| do
2      Li ← L[i];
3      Hi ← LH[i];
4      a ← 0;
5      A ← [];
6      for c ∈ Li do
7          if c = 1 then
8              a ← a + 1;
9          else
10             if a > 0 then
11                 A.push(a);
12                 a ← 0;
13     if a > 0 then
14         A.push(a);
15     for j ← 1to|Hi| do
16         if A[j] ≠ Hi[j] then
17             return false;
18 return true;
```

W procedurze 1.1 następuje weryfikacja rozwiązania w danej osi. Przykładowo, wywołując procedurę 1.1 dla listy wierszy i ich wskazówek, weryfikujemy Poprawność rozwiązania w poziomie. Weryfikacja rozwiązania następuje przez wywołanie procedury dwukrotnie, dla wierszy oraz kolumn. Jeśli w obu przypadkach procedura zwróci `true`, to rozwiązanie jest poprawne.

Czas wykonania procedury jest zależny od wielkości planszy. Zewnętrzna pętla wykonuje się tyle razy, ile jest linii w osi (h w przypadku wierszy, w w przypadku kolumn). Na początku pętli dochodzi do ekstrakcji pewnych danych do lokalnych zmiennych oraz inicjalizacji tablicy — w zależności od języka użytego do implementacji, ta grupa operacji zajmuje czas stały bądź liniowy. Następnie uruchamiana jest pierwsza wewnętrzna pętla. W tej pętli analizowane są dane w danej linii, by zmapować układ jej komórek do wskazówki, jaką reprezentuje. Złożoność operacji w każdej iteracji jest stała, jeśli założymy, że powiększenie tablicy o dodatkowy element wymaga stałego czasu — w p.p. czas wykonania iteracji może być liniowy. Ilość wykonań tej pętli zależy od długości linii. Po wykonaniu pierwszej pętli, w zależności od układu stanu komórek w linii, może dojść do kolejnego powiększenia tablicy o dodatkowy element — złożoność nie przekracza liniowej. Na końcu zewnętrznej pętli wykonywana jest druga pętla, która iteruje po elementach wskazówki zadanej w rozwiązaniu, i porównuje ich wartość do analogicznych elementów we wskazówce odtworzonej z układu linii. Rozbieżność oznacza, że rozwiązanie nie jest prawidłowe, i procedura przedwcześnie zakańcza wykonanie. Długość wskazówki można z góry ograniczyć przez $\lceil \frac{x}{2} \rceil$, gdzie x jest długością linii.

Zadana procedura sprawdza poprawność rozwiązania nonogramów, a jej złożoność, w zależności od implementacji operacji na tablicach, może wynosić $\mathcal{O}(n^2)$ bądź $\mathcal{O}(n^3)$. Zaproponowana procedura ma złożoność wielomianową, zatem problem decyzyjny rozwiązywania nonogramów należy do klasy *NP*.

1.3.2 Problem rozwiązania nonogramu jest NP-trudny

Dowód NP-trudności rozwiązywania nonogramów jest obszerny i wykracza poza zakres tej pracy. Przykładowy dowód jest opisany w pracy [22] i jego zarys jest następujący. Autor rozpoczyna dowód od powołania się na NP-trudność gry na grafach, nazwanej jako *Bounded Nondeterministic Constraint Logic*. Następnie, poprzez redukcję, autor udowadnia NP-trudność zmodyfikowanej wersji gry, określonej na grafach planarnych. Po udowodnieniu tego faktu autor konstruuje redukcję wielomianową z planarnej *Bounded Nondeterministic Constraint Logic* w rozwiązywanie nonogramów, tym samym udowadniając ich przynależność do tej klasy problemów.

1.3.3 Problem rozwiązania nonogramu jest NP-zupełny

Problem rozwiązania nonogramu został zaklasyfikowany do klasy problemów NP, oraz klasy problemów NP-trudnych. Wobec tego, rozważany problem należy do klasy problemów NP-zupełnych.

Rozdział 2

Projekt systemu

W tej części opisane zostały wymagania dla aplikacji w kontekście możliwości interakcji przez użytkownika.

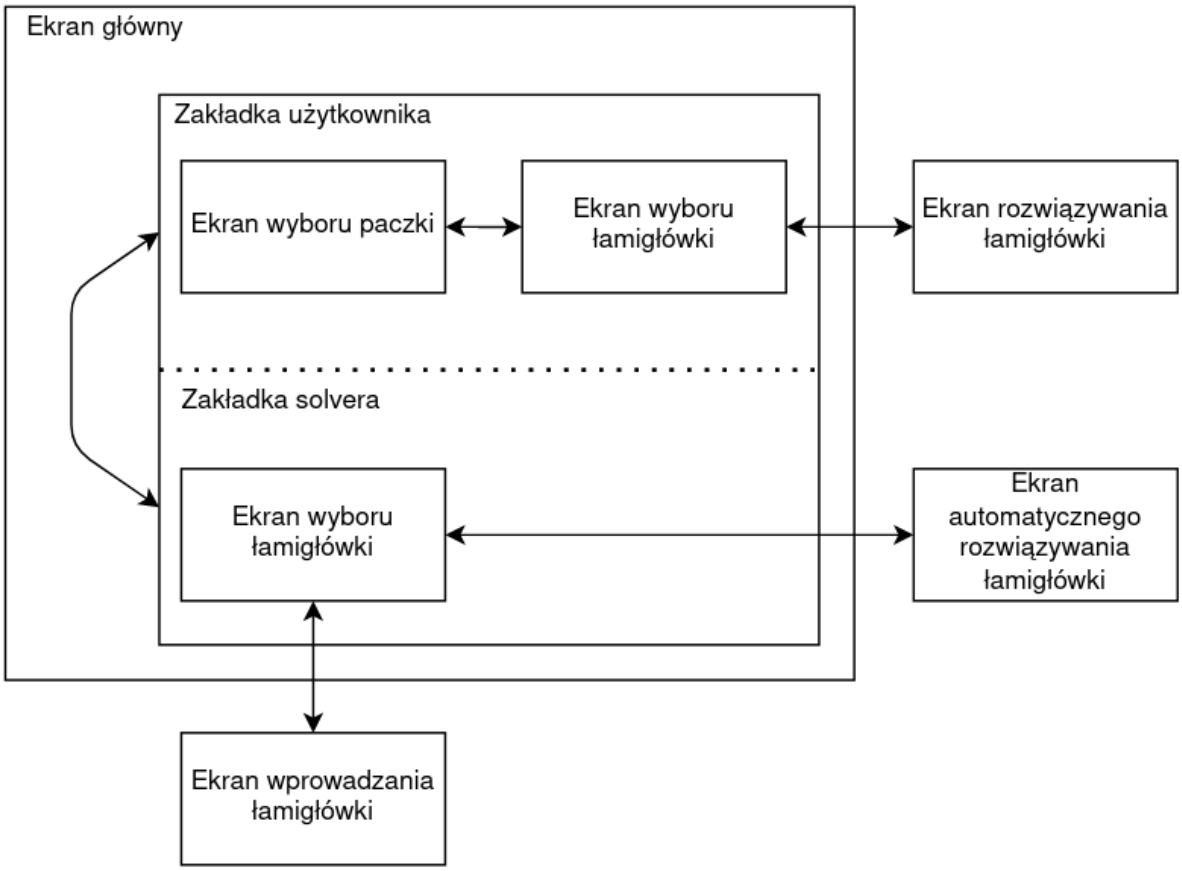
2.1 Wymagania aplikacji

1. Użytkownik może wybrać jedną z predefiniowanych paczek łamigłówek, by przejść do wyboru łamigłówki
2. Użytkownik może wybrać jedną z predefiniowanych łamigłówek w paczce, by przejść do jej rozwiązywania
3. Użytkownik może rozwiązać łamigłówkę przy użyciu wyświetlanej planszy
4. Użytkownik może oznaczać pola, co do których ma pewność, że są puste
5. Aplikacja śledzi błędy użytkownika w trakcie rozwiązywania łamigłówki i przerywa jej rozwiązywanie w przypadku popełnienia zbyt wielu błędów
6. Postęp rozwiązywania łamigłówki jest zapisywany przy wyjściu do poprzedniego ekranu
7. W menu wyboru łamigłówki wyświetlany jest stan łamigłówki tak, by można było ocenić:
   - czy została rozpoczęta,
   - czy została ukończona bez przegranej,
   - czy została ukończona z przegraną.
8. Użytkownik może wprowadzić łamigłówkę dla solvera za pomocą ekranu wprowadzania łamigłówki
9. Użytkownik może wybrać łamigłówkę do rozwiązania dla solvera
10. Użytkownik może nakazać rozwiązanie wprowadzonej łamigłówki

2.2 Nawigacja pomiędzy aktywnościami

Aplikacja otwiera się w menu głównym. W menu głównym dostępne są dwie zakładki: zakłada użytkownika i zakładka w solvera. W zakładce użytkownika użytkownik najpierw wybiera paczkę łamigłówek, a następnie łamigłówkę do rozwiązania, po czym przechodzi do ekranu rozwiązywania. W zakładce solvera dostępna jest lista wprowadzonych i predefiniowanych łamigłówek dla solvera. Użytkownik może przejść do ekranu wprowadzania łamigłówki — by dodać kolejną łamigłówkę — bądź do ekranu automatycznego rozwiązywania, gdzie nakazuje solverowi rozwiązanie wprowadzonej łamigłówki.

Zależności między opisanymi aktywnościami są ukazane na diagramie.



Rysunek 2.1: Diagram zależności między aktywnościami

Rozdział 3

Implementacja systemu

W tym rozdziale opisane są technologie i biblioteki użyte do stworzenia aplikacji, jak i schemat użytej bazy danych.

3.1 Opis technologii

3.1.1 React Native

Aplikacja została napisana w bibliotece React Native [4]. Jest to framework umożliwiający tworzenie aplikacji mobilnych, komputerowych oraz internetowych. React Native jest oparty na bibliotece React.js [9], stworzonej przez Jordana Walke, a rozwijanej przez Meta Platforms Inc. oraz społeczność, i jej rozwój także jest nadzorowany przez tę firmę. Framework jest dostępny na licencji MIT.

Głównymi pojęciami potrzebnymi do tworzenia aplikacji w React Native są komponenty oraz stany. Komponenty reprezentują nie tylko podstawowe składniki interfejsu graficznego, takie jak pola tekstowe czy przyciski, ale także zestawy składników realizujące określoną funkcję, np. plansza do gry lub lista gier. Komponenty opierają się o stan, czyli zestaw informacji przechowywany w komponencie i komunikowany do użytkownika za pomocą interfejsu. Framework odświeża wygląd interfejsu przy zmianie stanu, która najczęściej następuje w wyniku interakcji użytkownika z interfejsem. Wtedy to dochodzi do aktualizacji komponentów korzystających z danego stanu.

Programowanie w React Native odbywa się za pomocą języka skryptowego JavaScript. Znak towarowy należy do Oracle [2], standard jest utrzymywany przez ECMA [1], a uruchamiany jest na różnych silnikach rozwijanych zgodnie ze standardem (np. V8 [11] od firmy Google, czy SpiderMonkey [10] od Mozilli).

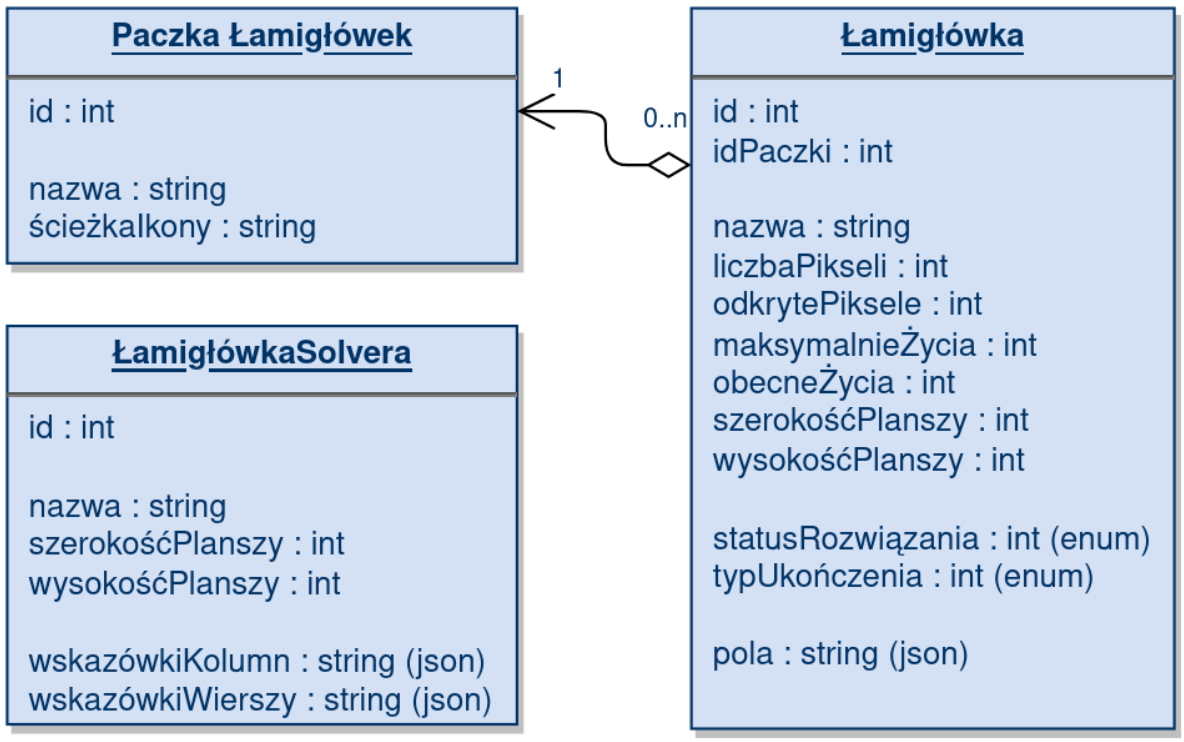
3.1.2 Dodatkowe biblioteki

Do stworzenia aplikacji zostały wykorzystane biblioteki tworzone przez społeczność. Są to między innymi:

- React Native Elements [5] - biblioteka zawierająca podstawowe komponenty zgodne z Material Design
- React Native Navigation [6] - biblioteka obsługująca nawigację między aktywnościami
- React Native SQLite Storage [7] - biblioteka pozwalająca korzystać z lokalnej bazy SQLite
- React Native Zoomable View [8] - biblioteka dodająca komponent obsługujący przybliżanie ekranu

3.2 Baza danych

Dane aplikacji są przechowywane w bazie danych opartej na systemie SQLite. Schemat bazy jest przedstawiony na diagramie poniżej.



Rysunek 3.1: Diagram bazy danych

Baza składa się z 3 tabel: paczek, łamigłówek i łamigłówek solvera. Paczki zawierają jedynie niezbędne dane do reprezentacji grupy łamigłówek, czyli nazwa i ikona. W tabeli łamigłówek zawarte są proste dane takie jak nazwa, maksymalna ilość żyć (ilość błędów, po których się przegrywa). Przechowywane są tam również informacje o stanie gry, tj. `statusRozwiązania` — łamigłówka nierozpoczęta i nierozwiązana, łamigłówka rozpoczęta, łamigłówka rozwiązana — oraz `typUkończenia` — łamigłówka nieukończona bez przegranej, łamigłówka nieukończona z przegraną, łamigłówka ukończona bez przegranej, łamigłówka ukończona z przegraną. Pod zmienną `pola` przechowywana jest lista pól wraz z ich stanami, pod postacią stringa w formacie JSON. Tabela łamigłówek solvera także zawiera łamigłówki, ale w innej formie: łamigłówka zapisana jest jako listy wskazówek w formacie JSON oraz dane do identyfikacji, np. nazwa czy wielkość planszy.

Rozdział 4

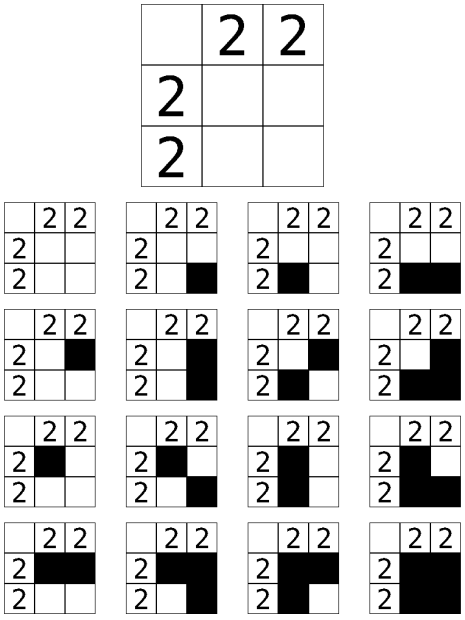
Solver nonogramów

W tym rozdziale opisany jest rozwój solvera. Opisane są kolejne główne wersje solverów. Zbadany został także wpływ zastosowanych heurystyk na wydajność w rozwiązywaniu wybranych łamigłówek.

4.1 Wersje solverów

4.1.1 Solver całościowy

Solver całościowy jest najprostszym z solverów implementowanych w toku pisania aplikacji. Jego implementacja opiera się na założeniu, że obrazek ukryty w łamigłówce jest ciągiem pustych i pełnych pikseli. Solver sprawdza wszystkie możliwe kombinacje pól, aż do wykrycia rozwiązania, bądź stwierdzenia jego braku. Wskazówki umieszczone obok planszy służą jedynie do walidacji rozwiązania i nie są wykorzystywane w trakcie rozwiązywania nonogramu.



Rysunek 4.1: Solver całościowy sprawdza wszystkie możliwe układy planszy w celu znalezienia rozwiązania.

Solver ten zaczyna od pustej planszy. Następnie, dla pierwszego pola wywoływana jest rekursyjna metoda: jeśli indeks pola mieści się w zakresie planszy, to najpierw jego status ustawiany jest na pusty, i następuje wywołanie metody dla następnego indeksu, a jeśli rozwiązanie nie zostanie znalezione, to pole jest wypełniane i ponownie dochodzi do wywołania metody na następnym polu. Jeśli indeks wykracza poza

zakres planszy, to znaczy, że wszystkie pola mają ustawiony status i wywoływana jest metoda sprawdzająca poprawność rozwiązania, podobna do tej opisanej w 1.1. Jeśli solver zakończy działanie zwracając `true`, to w przekazanej mu macierzy pól (równoznaczne z listą wierszy) znajdzie się rozwiązanie łamigłówki.

Pseudokod 4.1: SolverCałościowy

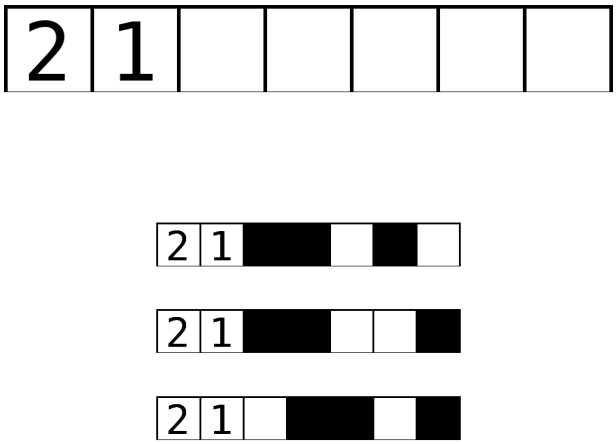
Input: Lista wierszy R, indeks pola i, lista wskazówek wierszy Hr i kolumn Hc, szerokość w i wysokość h planszy
Output: Czy znaleziono rozwiązanie `true/false`

```
1  if i ⩾ w · h then
2      return Verify(R, Hr, Hc, w, h);
3  else
4      iWiersza ← ⌈i/w⌉;
5      iKolumny ← i mod w;
6      R[iWiersza][iKolumny] ← 0;
7      if SolverCałościowy(R, i + 1, Hr, Hc, w, h) then
8          return true;
9      R[iWiersza][iKolumny] ← 1;
10     return SolverCałościowy(R, i + 1, Hr, Hc, w, h);
```

Złożoność czasowa tego solvera jest bardzo duża. Procedura sprawdzająca poprawność rozwiązania może zostać wywołana 2^{w*h} razy, czyli inaczej 2^n, gdzie n to ilość pól na planszy. Wynika to z faktu, że każde kolejne pole wymaga sprawdzenia pól poprzednich, a samo może znajdować się w dwóch stanach, więc podwaja ilość wywołań procedury sprawdzającej.

4.1.2 Solver osiowy

W odróżnieniu od solvera całościowego, solver osiowy korzysta ze wskazówek przy szukaniu rozwiązań. Opiera się on na fakcie, że każda linia (wiersz lub kolumna) może znajdować się w jednym z możliwych stanów, których liczba nigdy nie dojdzie do 2^x, gdzie x jest długością linii. Sprawdzając rozwiązanie w tym solverze, gwarantowana jest poprawność w jednej z osi, co dodatkowo znacząco skraca czas szukania rozwiązania.



Rysunek 4.2: Dla linii na grafice solver osiowy spradza jedynie 3 stany. Dla tej samej linii solver całościowy sprawdziłby $2^5 = 32$ stany

Solver zaczyna od pustej planszy. Przed rozpoczęciem rozwiązywania sprawdzana jest ilość wszystkich kombinacji w danej osi (iloczyn możliwości każdej z linii) i wybierana jest oś z mniejszą liczbą możliwości. Następnie generowane są kombinacje dla każdej z linii. Solver korzysta z rekursyjnej metody i ustawia pierwszą kombinację dla pierwszej linii. Następnie wywołuje metodę dla kolejnej linii, aż do ostatniej, i

wtedy weryfikuje rozwiązanie. Jeśli dla danego ustawienia w linii łamigłówka nie ma rozwiązania, to solver przechodzi do kolejnego ustawienia i wywołuje metodę w kolejnej linii.

Pseudokod 4.2: SolverOsiowy

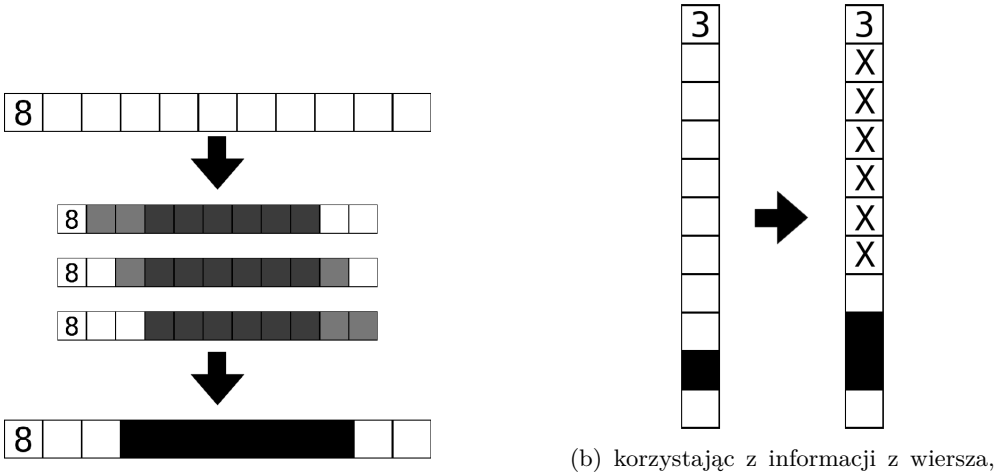
```
Input: Lista linii L, indeks linii i, lista wskazówek linii prostopadłych H, ilość linii n
Output: Czy znaleziono rozwiązanie true/false
1 if i = n then
2     return Verify(L, H, n);
3 else
4     linia ← L[i];
5     foreach komb ∈ linia.kombinacje do
6         NalozKombinacje(linia, komb);
7         if SolverOsiowy(L, i + 1, H, n) then
8             return true;
9     return false;
```

Dzięki eliminacji kombinacji sprzecznych ze wskazówkami w danej osi, procedura sprawdzania poprawności rozwiązania jest wywoływana o wiele rzadziej niż w przypadku solvera całościowego. O ile ilość kombinacji w linii przy rozpatrywaniu każdej komórki z osobna to 2^n, gdzie n to długość linii, tak w przypadku rozważania poprawnych kombinacji dla linii jest ona zależna od długości i zawartości wskazówki, i można ją ograniczyć z góry przez $\binom{n+1-h}{h}$, a h to ilość wskazówek w linii. To przybliżenie jest zawyżone, ponieważ zakłada występowanie jedynie bloków długości jeden we wskazówce. W przeciętnym przypadku, bloki wypełnionych komórek będą dłuższe, oraz będzie ich mniej. Co więcej, jak zostało wspomniane na początku, weryfikacja jest wymagana jedynie w jednej z dwóch osi, jako że konstrukcja potencjalnych rozwiązań opiera się o zestawianie poprawnych kombinacji z linii.

4.1.3 Solver eliminacyjny

Solverem, którego wariant znajduje się w aplikacji, jest solver eliminacyjny. W przeciwieństwie do wcześniej opisanych solverów, solver ten nie zakłada układów komórek w liniach tak długo, jak to możliwe. W przypadku tego solvera, generowane są możliwe kombinacje dla każdej z linii (zarówno wierszy, jak i kolumn). W danym przejściu eliminowane są kombinacje sprzeczne z dostępnymi informacjami (np. kombinacje posiadające pełną pierwszą komórkę, podczas gdy pewne jest, że jest ona pełna) oraz wyciągane są części wspólne kombinacji (np. wszystkie kombinacje mają pustą drugą komórkę), które dostarczają informacji dla innych linii. Co więcej, w przeciwieństwie do poprzednich solverów, nie jest konieczna walidacja rozwiązania, jako że rozwiązanie jest poprawne w momencie, gdy każdy wiersz i każda kolumna ma dostępną jedną możliwą kombinację.

Solver zaczyna od pustej planszy. Na początku generowane są wszystkie kombinacje dla każdej z linii, a linie wrzucane są do kolejki *last-in first-out*. Następnie solver przechodzi do rozwiązywania. Dopóki kolejka nie jest pusta, to są linie, których komibnacje należy zweryfikować. Z kolejki usuwana jest sprawdzana linia. Dla tej linii następują dwa kroki: najpierw, eliminowane są kombinacje sprzeczne z układem danej linii. W wypadku tego solvera, każda komórka znajduje się w jednym z trzech stanów: *pełny*, *pusty* i *nieznany*. Stan *nieznany* komórki dopuszcza kombinacje zawierające komórkę pełną lub pustą; pozostałe stany wymagają zgodności stanu ze stanem komórki w kombinacji. Po eliminacji sprzecznych kombinacji, dochodzi do porównania kombinacji. Jeśli istnieje komórka w linii, której stan jest *nieznany*, a wszystkie pozostałe kombinacje mają ustawiony dla niej ten sam stan, to stan komórki jest aktualizowany, a prostopadła linia zostaje dodana do kolejki do weryfikacji (następuje przy tym upewnienie, że w kolejce nie ma powtórzeń). Kiedy kolejka zostanie opróżniona, sprawdzane są linie. Jeśli wszystkie linie mają jeden możliwy stan, to znaczy, że zostało znalezione rozwiązanie. Jeśli któraś z linii nie ma możliwej kombinacji, to nie istnieje rozwiązanie przy dokonanych założeniach. W przeciwnym wypadku, solver zakłada poprawność jednej z kombinacji dla linii o kilku możliwych kombinacjach. Jako że wskutek założenia stan linii zmienił się, to do kolejki dodawane są linie prostopadłe. Następnie wywoływana jest w sposób rekursyjny metoda rozwiązująca nonogram dla obecnego stanu planszy. Jeśli rozwiązanie nie zostanie znalezione w tej gałęzi,



(a) pola wspólne dla wszystkich kombinacji zostają naniesione na planszę

(b) korzystając z informacji z wiersza, solver wnioskuje stan większości pól w kolumnie (x oznacza pole definitywnie puste)

Rysunek 4.3: Przykład wnioskowania na podstawie możliwych kombinacji

to solver zakłada poprawność kolejnej kombinacji dla tej linii.

Pseudokod 4.3: SolverEliminacyjny

```
   Input: Lista wierszy R i kolumn C, kolejka Q
   Output: Czy znaleziono rozwiązanie true/false
 1 while Q.zawieraElementy() do
 2     linia ← Q.pop();
 3     SprawdzLinie(linia);
 4 if (∀linia ∈ R⋃C)(|linia.kombinacje| = 1) then
 5     return true;
 6 else if (∃linia ∈ R⋃C)(|linia.kombinacje| = 0) then
 7     return false;
 8 else
 9     zakladanaLinia ← pierwsza linia z wieloma kombinacjami;
10     foreach komb ∈ zakladanaLinia.kombinacje do
11         kopiaR ← kopiuj(R);
12         kopiaC ← kopiuj(C);
13         NalozKombinacje(zakladanaLinia, komb);
14         UzupelnijKolejke(Q);
15         if SolverEliminacyjny(kopiaR, kopiaC, Q) then
16             R ← kopiaR;
17             C ← kopiaC;
18             return true;
19     return false;
```

Istotna uwaga dotyczy charakterystyki wierszy i kolumn. W celu umożliwienia działania procedury został wykorzystany mechanizm obecny w wielu powszechnie używanych językach programowania, mianowicie mechanizm płytkiej kopii. Mimo że listy wierszy i kolumn zawierają inne obiekty (listy), to obiekty komórek przechowywane w tych zagnieżdżonych listach są takie same. Dzięki temu, modyfikując stan komórki w liście wierszy w n-tym wierszu i m-tej komórce, modyfikujemy także stan komórki zawartej w n-tej komórce w m-tej kolumnie w liście kolumn.

Pseudokod 4.4: SprawdzLinie

```
Input: Linia l
foreach komb ∈ l.kombinacje do
    if !WeryfikujKombinacje(l) then
        l.kombinacje.usun(l);
foreach i ∈ {1, 2, ..., |l|} do
    if l[i] = nieznany ∧ pole identyczne we wszystkich kombinacjach then
        l[i] = nowy stan;
        Q.dodaj(linia prostopadła do l o indeksie i);
```

4.2 Porównanie wydajności solverów

W tej części zbadane zostały wydajności poszczególnych wersji solverów oraz wpływ wybranych modyfikacji i heurystyk na ich wydajność.

4.2.1 Metodyka badań

Metoda prowadzenia badań jest następująca: dla każdej łamigłówki prowadzonych jest 5 prób rozwiązania. Próby z minimalnym oraz maksymalnym czasem rozwiązywania zostają odrzucone, w celu eliminacji błędów powstałych na skutek zaburzeń występujących w trakcie prowadzenia testu. Następnie czas pozostałych 3 prób zostaje uśredniony i podany jako wynik badania. Jeśli czas wykonania iteracji testu przekroczy 5 minut, to test zostaje przerwany i do komórki zostaje wpisany wynik *przekroczono*.

Testy zostały przeprowadzone na komputerze z procesorem Intel Core i5-6300HQ, z 16GB pamięci RAM o taktowaniu 2133Mhz. Testy zostały uruchomione w środowisku Node.js [3] w wersji v14.18.2. Czas rozwiązywania łamigłówek na urządzeniu mobilnym może znacząco odbiegać od czasów zaprezentowanych w wynikach, z powodu mniejszych zasobów i zastosowanego środowiska uruchomieniowego.

Wyniki zostały zaprezentowane w tabelach o następującym układzie: w pierwszych dwóch kolumnach zostały podane nazwy łamigłówek i ich wielkości (łamigłówki są zadane na planszach kwadratowych). W kolejnych kolumnach, nazwanych odpowiednio do zastosowanej modyfikacji bądź jej braku, podany jest czas rozwiązywania nonogramu wyrażony w milisekundach. Czas jest zaokrąglony do pełnych milisekund, zatem dla prostych łamigłówek może być wyrażony jako 0.

Zestaw danych użyty do testów został przedstawiony na końcu rozdziału.

4.2.2 Solver całościowy

Wydajność solvera całościowego została sprawdzona na zestawie 4 łamigłówek.

Wyniki

Nazwa	Rozmiar	Czas rozwiązywania w ms
Cross	3	1
Flame	5	46
Jellyfish	8	*przekroczono*
Candy	8	*przekroczono*

Tablica 4.1: Wyniki testów dla solvera całościowego

Interpretacja

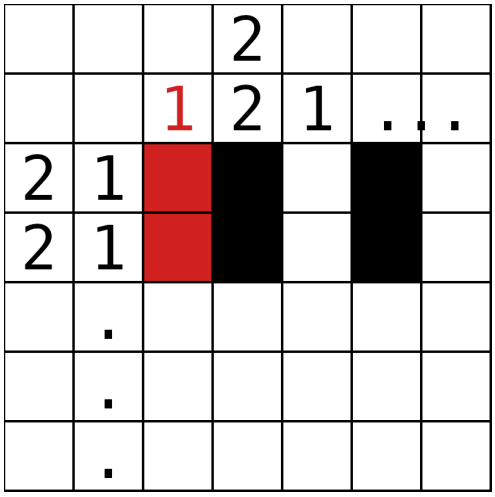
Solver całościowy jest w stanie dość szybko rozwiązać niewielkie łamigłówki, ale czas jego pracy rośnie bardzo szybko wraz ze wzrostem wielkości planszy.

4.2.3 Solver osiowy

Wydajność solvera całościowego została sprawdzona na pełnym zestawie 18 łamigłówek. Został on poddany jednej modyfikacji.

Modyfikacje

Częściowa weryfikacja rozwiązania polega na wykorzystaniu procedury częsciowej weryfikacji. Jest ona podobna do procedury weryfikacji w osi, ale nie sprawdza poprawności całej planszy, a jedynie sprawdza, czy dany układ może prowadzić do poprawnego rozwiązania (czyli czy nie ma sprzeczności na tym etapie). Procedura otrzymuje głębokość, dla jakiej ma zweryfikować poprawność. Jeśli do danej głębokości nie występuje sprzeczność, to rozwiązanie jest konstruowane dalej. W przeciwnym przypadku, w danej gałęzi nie może być poprawnego rozwiązania, i solver przechodzi do innej gałęzi.



Rysunek 4.4: Na tym etapie można zdyskwalifikować każde rozwiązanie zawierające pierwszy i drugi wiersz w układzie widocznym na grafice

Wyniki

Nazwa	Rozmiar	Czas rozwiązywania w ms	
		bez modyfikacji	z częściową weryfikacją
Cross	3	0	0
Flame	5	0	0
Jellyfish	8	463	2
Candy	8	0	0
Shield	10	*przekroczono*	3
Dog	10	12004	1
Duck	16	*przekroczono*	12
Shrimp	16	*przekroczono*	14
Cherries	20	*przekroczono*	25
Potion	20	*przekroczono*	315
Tape	25	*przekroczono*	1091
Lighter	25	*przekroczono*	60945
Ghost	32	*przekroczono*	40040
Demon	32	*przekroczono*	3997
Butcher	48	*przekroczono*	*przekroczono*
Ranger	48	*przekroczono*	59703
Orc	64	*przekroczono*	*przekroczono*
Swordsman	64	*przekroczono*	*przekroczono*

Tablica 4.2: Wyniki testów dla solvera osiowego

Interpretacja

W podstawowej wersji solver osiowy ma niewiele lepszą wydajność od solvera całościowego, będąc w stanie rozwiązać niektóre łamigłówki wielkości 10 przy nałożonym ograniczeniu czasowym. Jednak po dodaniu częściowej weryfikacji, wydajność solvera drastycznie wzrasta, i może zostać wykorzystany do rozwiązywania łamigłówek o wiele większych niż w przypadku podstawowej wersji. Pokazuje to, jak kluczowa jest wstępna eliminacja niepożądanych gałęzi przy rozwiązywaniu danego problemu.

4.2.4 Solver eliminacyjny

Wydajność solvera całościowego została sprawdzona na pełnym zestawie 18 łamigłówek.

Modyfikacje

Rozwiązywanie bez przerywania (*bez przerw.*) to modyfikacja polegająca na usunięciu warunkowego przerwania szukania rozwiązania. Zamiast przerwać w momencie znalezienia kolumny niezawierającej żadnej poprawnej kombinacji, solver szuka dalej, aż do opróżnienia kolejki, i dopiero wtedy informuje o braku rozwiązania. Z uwagi na częste korzystanie z kolejki, eliminacja tej instrukcji mogłaby skrócić czas wykonywania jednego sprawdzenia linii.

Zakładanie dla kolumn (*kolumny*) zmienia kolejność zakładania. Linia z wieloma kombinacjami jest szukana najpierw na liście kolumn, a dopiero potem na liście wierszy. Ta modyfikacja opiera się na założeniu, że układ wypełnionych pól w łamigłówkach powoduje lepszą wydajność przy rozwiązywaniu pionowym.

Zakładanie dla linii z minimalną/maksymalną liczbą kombinacji (*min/max komb.*) dokonuje założenia dla linii z najmniejszą/największą liczbą poprawnych kombinacji. W ten sposób badany jest wpływ doboru sposobu zakładania na czas szukania rozwiązania.

Kolejność linii w kolejce to seria modyfikacji sprawdzających wpływ typu kolejki na czas rozwiązania. W kolumnie *lifo* podane są wyniki dla kolejki, w której nowo dodane bądź ponownie dodane linie

są przesuwane na początek kolejki. W *fifo* przedmioty są wrzucane na koniec kolejki. *bez zm.* oznacza, że przedmioty są ustawiane na początku kolejki, ale ponowne dodanie nie zmienia kolejności przedmiotów w kolejce. W kolejnych trzech kolumnach (*śl. ind. lifo*, *śl. ind. fifo*, *śl. ind. bez zm.*) badane są te same własności kolejki, ale struktura ta zostaje zmodyfikowana tak, by śledzić linie wraz ze zmodyfikowanymi indeksami. Dzięki temu nie jest konieczne sprawdzenie każdego indeksu w linii, a jedynie tych komórek, dla których zaszła zmiana.

Wyniki

		Czas rozwiązywania w ms				
Nazwa	Rozmiar	bez modyfikacji	bez przerw.	kolumny	min komb.	max komb.
Cross	3	0	0	0	0	0
Flame	5	0	0	0	0	0
Jellyfish	8	1	1	1	1	1
Candy	8	0	0	0	0	0
Shield	10	2	2	2	2	1
Dog	10	1	0	0	0	1
Duck	16	2	1	1	1	1
Shrimp	16	1	1	1	1	1
Cherries	20	3	3	3	3	3
Potion	20	12	13	13	13	13
Tape	25	2	2	2	2	2
Lighter	25	13	13	13	12	13
Ghost	32	47	47	48	45	47
Demon	32	1787	1817	1806	1800	1822
Butcher	48	1811	1809	1819	1790	1791
Ranger	48	46	48	47	47	48
Orc	64	1463	1454	1446	1465	1456
Swordsman	64	1440	1467	1451	1429	1432

Tablica 4.3: I część wyników dla solvera eliminacyjnego

Interpretacja

Solver eliminacyjny jest najszybszym z zaprezentowanych solverów. Jest w stanie rozwiązać nawet największe, 64x64 nonogramy w założonym ograniczeniu czasowym. Każdą ze sprawdzanych łamigłówek rozwiązuje szybciej niż solver osiowy. Na szczególną uwagę zasługuje szybkość rozwiązania nonogramów *Ghost* i *Ranger*. Dzięki oparciu się o informacje, które zostały wywnioskowane z poprzednio rozpatrywanych linii, solver może rozwiązać nawet łamigłówki o dużych rozmiarach bez zakładania układu linii, lub z minimalną jego ilością.

Pierwsza seria modyfikacji nie miała dużego wpływu na czas rozwiązywania. Część modyfikacji z drugiej serii okazała się mieć kluczowy wpływ na wydajność solvera przy rozwiązywaniu danej łamigłówki. Użycie kolejki typu *first-in first-out* z przerzucaniem na koniec przy aktualizacji linii znacznie spowolniło czas rozwiązywania łamigłówki *Demon* w przypadku bazowej wersji solvera, a w przypadku wersji ze śledzeniem indeksów różnica była mniejsza, ale zauważalna. Oprócz tego, modyfikacja struktury danych w celu śledzenia modyfikowanych indeksów okazała się dobrym pomysłem, jako że solver korzystający z niej (*śl. ind. bez zm.*) osiągnął najlepsze czasy dla trudniejszych łamigłówek (np. *Demon*, *Butcher*), będąc nieznacznie wolniejszym dla łamigłówek prostszych (np. *Ghost*, *Ranger*).

Nazwa	Rozmiar	Czas rozwiązywania w ms					
		lifo	fifo	bez zm.	śl. ind. lifo	śl. ind. fifo	śl. ind. bez zm.
Cross	3	0	0	0	0	0	0
Flame	5	0	0	0	0	0	0
Jellyfish	8	1	2	2	2	1	1
Candy	8	0	0	0	0	0	0
Shield	10	2	1	1	2	3	2
Dog	10	1	1	1	1	0	1
Duck	16	2	1	2	2	2	2
Shrimp	16	1	1	1	1	1	1
Cherries	20	3	4	4	3	4	3
Potion	20	12	13	14	16	16	13
Tape	25	2	2	2	4	3	3
Lighter	25	13	13	13	13	15	13
Ghost	32	47	47	52	56	72	54
Demon	32	1787	2380	1980	1194	1257	1127
Butcher	48	1811	1884	1798	1854	1962	1717
Ranger	48	46	44	53	61	62	59
Orc	64	1463	1413	1437	1413	1415	1311
Swordsman	64	1440	1489	1544	1534	1608	1451

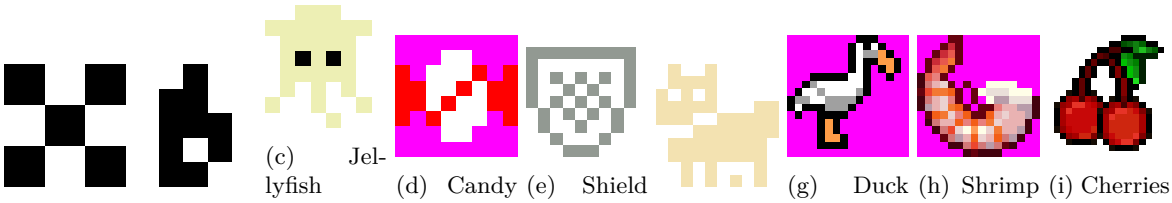
Tablica 4.4: II część wyników dla solvera eliminacyjnego

4.2.5 Wnioski

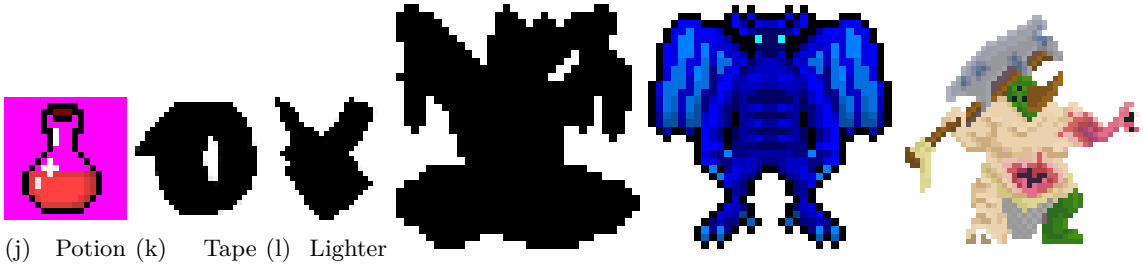
Solver eliminacyjny jest najwydajniejszym z zaproponowanych solverów. Dzięki oparciu na pewnych informacjach i ograniczeniu zakładania, a co za tym idzie back-trackingu, może rozwiązać zadaną łamigłówkę szybbiej niż inne solvery. W przypadku solvera osiowego, introdukcja częściowej weryfikacji bardzo poprawiła jego wydajność.

4.2.6 Zestaw danych

Do przeprowadzenia badań zostały użyte następujące obrazy, które po konwersji zostały przekazane do solverów jako zestaw wskazówek. Grafiki zawierające kolor biały zostały nałożone na tło w kolorze magenta w celu wyróżnienia tego koloru na tle białej kartki.



(a) Cross (b) Flame (c) Jellyfish [24] (d) Candy [24] (e) Shield [23] (f) Dog [23] (g) Duck [13] (h) Shrimp [17] (i) Cherries [14]



(j) Potion [14] (k) Tape [18] (l) Lighter [18] (m) Ghost [16] (n) Demon [16] (o) Butcher [12]

(p) Ranger [15] (q) Orc [19]

(r) Swordsman [20]

Rysunek 4.5: Zestaw danych

Rozdział 5

Instalacja i wdrożenie

5.1 Środowisko

Aplikacja opisana w tej pracy jest aplikacją mobilną. Do jej uruchomienia potrzebne jest urządzenie mobilne z systemem Android. Działanie aplikacji zostało przetestowane na telefonie działające na Android 8.0.

5.2 Instalacja

By zainstalować aplikację, należy przenieść załączony plik formatu `.apk` na urządzenie. Następnie, należy przejść do lokacji z plikiem i rozpocząć jego instalację — domyślnie można tego dokonać przez kliknięcie ikony pliku instalacyjnego.1

5.3 Obsługa

Po uruchomieniu aplikacji wyświetlony zostanie ekran główny, czyli ekran wyboru paczki łamigłówek. Kolejne interakcje można prowadzić nawigując po systemie, zgodnie z diagramem przedstawionym w rozdziale trzecim (2.1).

Rozdział 6

Podsumowanie

6.1 Aplikacja i praca

Aplikacja przedstawiona w tej pracy ma umożliwić zarówno rekreacyjne rozwiązywanie predefiniowanych nonogramów, jak i zapewnić dostęp do szybkiego solvera nonogramów.

W pracy zostały opisane pojęcia teoretyczne związane z rozwiązywaniem nonogramów, między innymi klasy złożoności problemów. Nakreślona została złożoność tej klasy łamigłówek oraz pokazane zostały różne podejścia do ich rozwiązywania. Porównanie algorytmów rozwiązujących nonogramy oraz zbadanie wpływu heurystyk na ich wydajność pozwala lepiej zrozumieć istotę złożoności danych plansz i umożliwić łatwiejsze określenie ich trudności.

6.2 Możliwości rozwoju

Aplikacja może zostać poszerzona o moduł generujący łamigłówki żądanej trudności. Dzięki temu program zyskałby powtarzalność i umożliwił użytkownikowi dobranie odpowiedniego wyzwania na daną chwilę. Dodanie modułu sczytującego plansze ze zdjęć znacznie ułatwiłoby wprowadzanie łamigłówek dla solvera. W dłuższej perspektywie aplikacja skorzystałaby z dodania funkcji publikacji łamigłówek i rozwiązywania łamigłówek stworzonych przez innych graczy.

Bibliografia

[1] Javascript ecma. https://www.ecma-international.org/.

[2] Javascript oracle. https://developer.oracle.com/javascript/.

[3] Node.js. https://nodejs.org/en/.

[4] React native. https://reactnative.dev/.

[5] React native elements. https://reactnativeelements.com/.

[6] React native navigation. https://reactnavigation.org/.

[7] React native sqlite storage. https://github.com/andpor/react-native-sqlite-storage.

[8] React native zoomable view. https://github.com/openspacelabs/react-native-zoomable-view.

[9] React.js. https://reactjs.org/.

[10] Spidermonkey javascript engine. https://spidermonkey.dev/.

[11] V8 javascript engine. https://v8.dev/.

[12] O. M. Army. plagueking 48x48. https://opengameart.org/content/plagueking-48x48.

[13] cimeto. 16x16 strategy bronze age icons. https://cimeto.itch.io/16x16-strategyrpg-bronze-iron-age-icons.

[14] drakzin. 20x20 tileset. https://opengameart.org/content/20x20-tileset.

[15] C. Gabriel. 48x48 faces. http://cgartsenal.blogspot.com/.

[16] C. Hamons. dungeon crawl 32x32. https://opengameart.org/content/dungeon-crawl-32x32-tiles.

[17] henrysoftware. pixel food. https://henrysoftware.itch.io/pixel-food.

[18] ikaytobello. inventory items. https://ilkaytobello.itch.io/inventory-items.

[19] LordNeo. orc 64x64. https://opengameart.org/content/orc-static-64x64.

[20] LordNeo. swordsman 64x64. https://opengameart.org/content/swordsman-static-64x64.

[21] L. L. Stephen Cook. The complexity of theorem-proving procedures, 1971.

[22] J. N. van Rijn. Playing games. the complexity of klondike, mahjong, nonograms and animal chess. https://liacs.leidenuniv.nl/assets/2012-01JanvanRijn.pdf, 2012.

[23] vexed. 10x10 top down rpg tiles. https://opengameart.org/content/bit-bonanza-10x10-top-down-rpg-tiles.

[24] Zakchaos. 8x8 food tiles. https://opengameart.org/content/43-8x8-food-tiles.