

Płaszczakowo

Algorytmy i Struktury Danych II (2024)

Adam Cedro
Sajmon Koniec
Jakub Pawłowski
Bartłomiej Wiśniewski

O projekcie

Płaszczakowo

Płaszczakowo jest projektem zaliczeniowym, który powstał w ramach zajęć 'Algorytmy i Struktury Danych II' w roku akademickim 2023/2024. Naszym celem podczas tworzenia tego projektu było dążenie do poszerzenia wiedzy oraz przygotowanie się do nadchodzącego przedmiotu 'Programowanie Zespołowe'.

Podczas realizacji projektu każdy z nas miał okazję nauczyć się czegoś nowego. Od podstaw tworzenia stron internetowych, po skuteczne zarządzanie zespołem.

Skład projektu

Tworzyliśmy projekt w zespole czteroosobowym. Każdy z nas miał przydzielone inne zadanie, ale w razie problemów chętnie sobie nawzajem pomagaliśmy.

Poniżej przedstawiamy obowiązki każdego z nas:

- **Adam Cedro**
 - Interpretacja problemu "Grafik strażników".
 - Implementacja problemu "Grafik strażników".
 - Stworzenie biblioteki do rysowania grafów.
 - Przygotowanie całej oprawy graficznej aplikacji.
- **Sajmon Koniec**
 - Interpretacja problemu "Budowanie Płotu".
 - Implementacja problemu "Budowanie Płotu".
 - Stworzenie pojedynczych grafik.
- **Jakub Pawłowski**
 - Opieka nad projektem na platformie GitHub
 - Prowadzenie zebrań i przydzielanie zadań.
 - Opracowanie szkieletu aplikacji, wybór technologii.
 - Praca nad UX aplikacji
 - Pomoc w rozwiązywaniu problemów.
- **Bartłomiej Wiśniewski**
 - Interpretacja problemu "Maszyna informatyka".
 - Implementacja problemu "Maszyna informatyka".

- Stworzenie naszych karykatur.

Styl graficzny

Zainspirowani wyjątkowym stylem kreskówkowym ilustracji Dr. Mariusza Kanieckiego, postanowiliśmy, że nasz projekt również będzie zawierał elementy w podobnym charakterze. Wszystkie grafiki w naszym projekcie są owocem naszej własnej pracy — nie korzystaliśmy z żadnych grafik dostępnych w Internecie ani generowanych za pomocą generatywnej sztucznej inteligencji. Do tworzenia naszych ilustracji użyliśmy programów Photoshop, Krita oraz tabletu graficznego z aplikacją PENUP.

Naszym celem było stworzenie unikalnych i oryginalnych grafik, które idealnie oddają atmosferę i styl naszego projektu. Poświęciliśmy wiele godzin na dopracowanie każdego szczegółu, dbając o to, aby każda ilustracja była spójna z naszą wizją artystyczną. Praca nad grafikami była dla nas nie tylko wyzwaniem, ale także ogromną przyjemnością, pozwalającą nam w pełni wykorzystać nasze umiejętności i kreatywność. Jesteśmy dumni z efektów naszej pracy i mamy nadzieję, że nasze ilustracje dodadzą uśmiechu podczas korzystania z aplikacji.

Strona główna

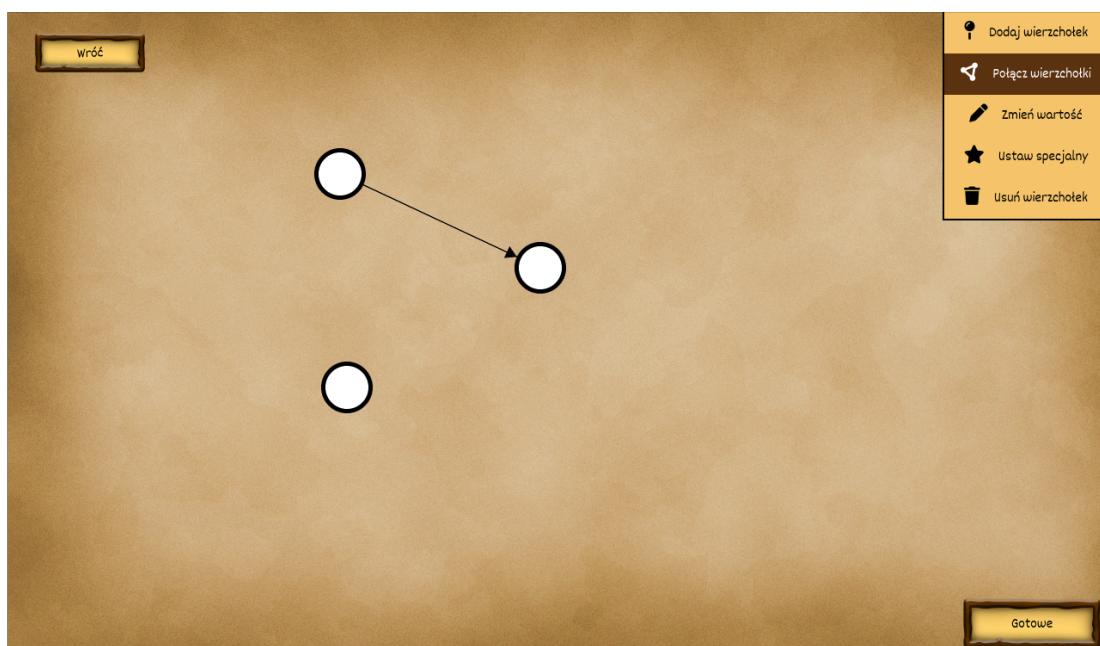


Na stronie głównej można przejść do poszczególnych problemów, z którymi Płaszczaki zetknęły się na swojej drodze. Dodatkowo dostępna jest opcja wyjścia z aplikacji.

W prawym dolnym rogu znajdują się dwa przyciski: przycisk z napisem "i" prowadzi do strony z informacjami o autorach, a przycisk z ikoną książki do dokumentacji.

W tle widoczna jest piękna panorama świata Płaszczaków, którą sami przygotowaliśmy. Użyliśmy efektu paralaksy, aby zasymulować głębię.

Wybór metody wprowadzenia

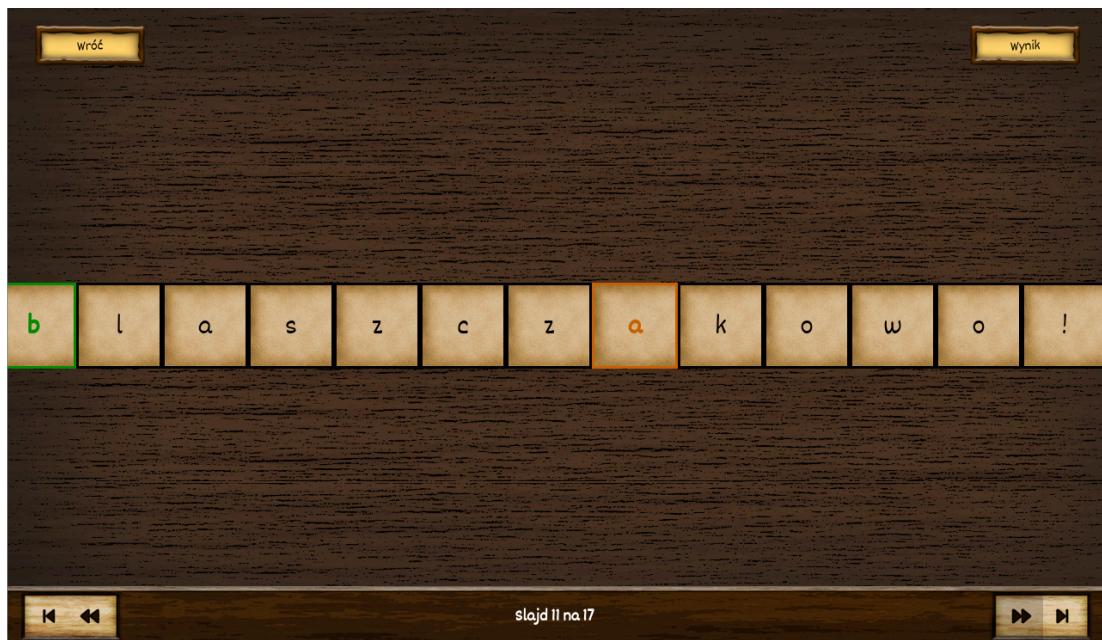


W Płaszczakowie kładziemy duży nacisk na interaktywność oraz ponowne wykorzystanie wcześniejszych danych wejściowych.

Dostępne metody wprowadzania danych to:

1. **Plik:** Użytkownik może wybrać plik JSON zawierający wcześniejsze przygotowane dane wejściowe. Dla każdego problemu istnieją domyślne dane wejściowe.
2. **Kreator grafów:** Użytkownik może za pomocą kliknięć myszki wskazać miejsca, gdzie mają się znajdować wierzchołki grafu, a następnie połączyć je krawędziami.
3. **Klawiatura:** Dostępne w problemie "Maszyna informatyka", gdzie melodie wpisuje się za pomocą klawiatury.

Symulacja przebiegu algorytmu



Ponieważ tworzymy aplikację graficzną, w pełni wykorzystaliśmy jej możliwości, aby w każdym problemie przeprowadzać graficzną symulację algorytmu.

Na dole strony znajduje się pasek, który umożliwia przewijanie slajdów symulacji, dzięki czemu możemy łatwo zobaczyć każdy krok działania algorytmu. Jest tam również informacja o aktualnym slajdzie.

Każdy z nas czerpał ogromną radość z możliwości interaktywnego śledzenia, jak algorytm działa krok po kroku. I naszym zdaniem to jedna z najlepszych funkcji projektu Płaszczakowo.

Dane wyjściowe

The screenshot shows a game interface titled "Grafik strażników" (Guard Schedule). At the top left is a "Wróć" (Return) button, and at the bottom right is a "Zakończ" (Finish) button. The central part is a table with the following data:

Dzień	ID Płaszczaka	Energia	Melodia
1	0	26	0
2	1	19	0
3	2	10	1
4	3	7	1
5	4	5	1
6	5	1	3

Podczas tworzenia danych wyjściowych zależało nam na połączeniu przejrzystości z estetyką. Każde dane wyjściowe są opatrzone atrakcyjną grafiką, dzięki czemu są nie tylko czytelne, ale również przyjemne dla oka.

Szczegóły techniczne

Wykorzystywane technologie

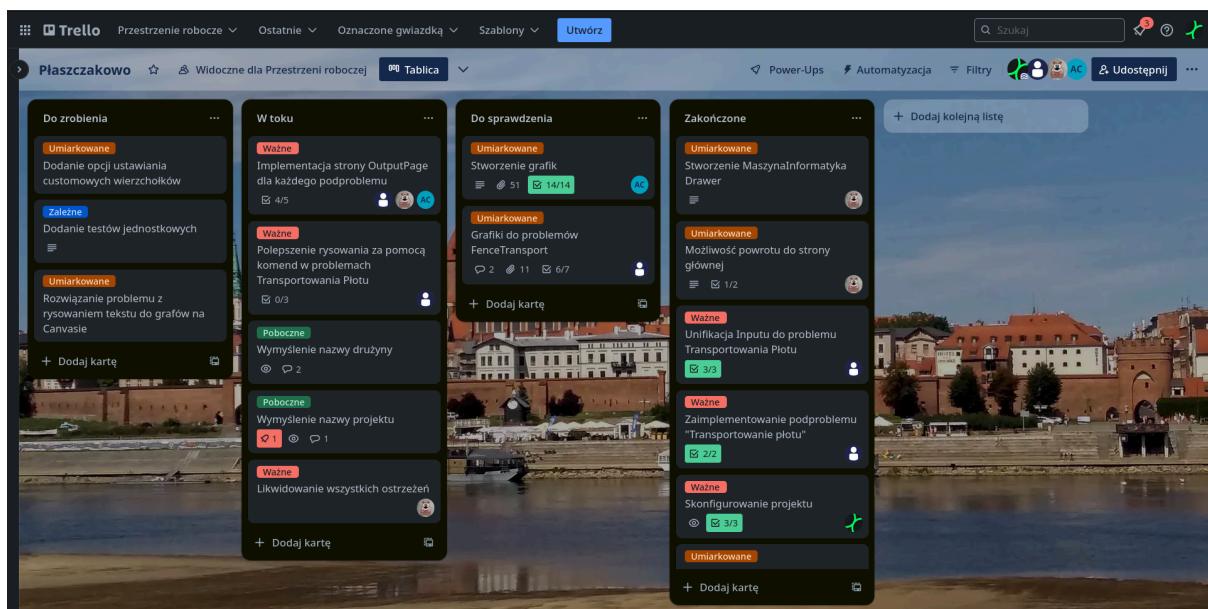
Płaszczakowo to aplikacja desktopowa, którą stworzyliśmy za pomocą języka programowania C# oraz platformy .NET. Projekt ten korzysta z technologii Electron oraz ASP.NET, aby zapewnić wyświetlanie i funkcjonalność aplikacji na różnych systemach operacyjnych, w tym Windows, Linux i macOS.

Electron jest frameworkm, który pozwala tworzyć aplikacje desktopowe przy użyciu technologii webowych. Jego główną zaletą jest możliwość uruchamiania aplikacji stworzonych jako strony internetowe w natywnym środowisku desktopowym. Electron opiera się na Chromium, silniku przeglądarki Google Chrome. Dzięki Electron nasza aplikacja Płaszczakowo może działać płynnie na różnych systemach operacyjnych.

ASP.NET to platforma aplikacji webowych stworzona przez Microsoft, która umożliwia tworzenie dynamicznych stron internetowych, aplikacji i usług. W projekcie Płaszczakowo używamy ASP.NET wraz z Blazor, co pozwala na wykorzystanie języka C#.

Blazor w ASP.NET pozwala na tworzenie komponentów, które mogą być ponownie używane i zarządzane w całej aplikacji.

Trello



Do zarządzania przydziałem zadań wykorzystaliśmy platformę Trello, co znacznie ułatwiło naszą współpracę nad projektem. Na Trello tworzyliśmy tablice z zadaniami do wykonania, które każdy członek zespołu mógł samodzielnie przydzielać sobie do realizacji.

Dzięki temu mieliśmy jasny podział obowiązków i mogliśmy na bieżąco śledzić postęp prac. Każde zadanie było opisane, przypisane do odpowiedniej kategorii, co pozwalało nam zachować porządek.

Możliwość komentowania i dodawania załączników do zadań również usprawniła naszą komunikację oraz wymianę informacji, ponieważ wszystko mieliśmy w jednym miejscu.

Wykorzystanie generatywnej sztucznej inteligencji.

Sztuczna inteligencja jest potężnym narzędziem wspierającym proces tworzenia aplikacji i naukę. Jednakże jej nadmierne użycie może mieć negatywne dla nas skutki. Dlatego podczas tworzenia naszego projektu ograniczyliśmy korzystanie ze sztucznej inteligencji do minimum.

Jesteśmy przeciwnikami bezrefleksyjnego korzystania z AI przy realizacji większości lub całości projektu. Stosujemy zasadę: "Najpierw pomyśl, potem użyj AI."

Oto elementy, w których wykorzystaliśmy AI:

1. Redagowanie dokumentacji.
2. Drobna pomoc, np. w wymyślaniu nazw klas lub zmiennych.
3. Generowanie prostego, ale czasochłonnego kodu.
4. Algorytm rysowania strzałek w grafie skierowanym.

Nasze podejście pozwoliło nam skorzystać z zalet AI, jednocześnie zachowując kontrolę nad procesem twórczym i rozwijając nasze własne umiejętności.

Problem “Budowanie Płotu”

Problem ten obejmuje dobranie Tragarzy w pary, znalezienie punktów orientacyjnych, które stworzą Płot oraz znalezienie sposobu na jak najszybsze zbudowanie Płotu.

Podproblem Dopasowania Tragarzy

Tragarze odpowiedzialni są za transportowanie odcinków płotu. Aby byli w stanie to zrobić muszą dobrać się w pary. Są dwa rodzaje tragarzy: ci z rękoma z przodu i ci z rękoma z tyłu (niemożliwe jest przełożenie rąk z jednej strony na drugą). Pod uwagę trzeba też wziąć relację tragarzy między sobą, gdyż nie wszyscy tragarze się lubią, a to jest kluczowe do szybkiego zbudowania płotu.

Dane wejściowe

Za dane wejściowe weźmy liczbę tragarzy z przodu oraz z tyłu, a także ich relacje między sobą. To nam wystarczy do wyznaczenia jak największej liczby par tragarzy.

Idealnym sposobem do wyznaczania par jest stworzenie sieci przepływowej z relacji między tragarzami oraz wykonać algorytm na maksymalny przepływ.

Stworzenie sieci przepływowowej

Ustawmy tragarzy z rękoma z przodu z jednej strony (lewej) a tragarzy z rękoma z tyłu z drugiej (prawej) oraz połączmy tych, którzy się lubią, krawędziami. Powstanie nam graf dwudzielny. Aby zmienić to w sieć przepływową należy zmienić krawędzie na skierowane oraz dodać przepustowość o wartości 1 do każdej z nich. Następnie utworzyć dwa nowe wierzchołki — źródło i ujście. Od źródła poprowadzić krawędzie skierowane z przepustowością 1 do tragarzy z rękoma z przodu, a od tragarzy z rękoma z tyłu poprowadzić takie same krawędzie do ujścia. Stworzy nam to sieć przepływową.

Algorytm maksymalnego przepływu

Użytem w programie algorytmem na maksymalny przepływ jest algorytm Forda-Fulkersona. Do znalezienia najkrótszej drogi ze źródła do ujścia jest użyty algorytm Depth-First Search (DFS).

Podproblem Znajdowania Płotu

Płot trzeba zbudować wokół krainy, wykorzystując punkty orientacyjne.

Naszimi danymi wejściowymi do tego podproblemu będzie lista punktów orientacyjnych, wraz z ich położeniem.

Algorytm Grahama

Pierwszym krokiem jest znalezienie najniżej położonego punktu na płaszczyźnie (punkt orientacyjny najbardziej na południe), oznaczmy go P_0 . Od tego punktu liczymy kąt między każdym pozostałym punktem. Jako że nie ma punktów poniżej P_0 , zakres kątów będzie od 0 do 180 stopni. Pozostałe wierzchołki sortujemy pod względem uzyskanych kątów, zaczynając od najmniejszego. Tworzymy stos, który początkowo zawiera P_0 oraz pierwszy

punkt z posortowanej listy. Iterujemy przez pozostałe punkty w posortowanej kolejności, dla każdego punktu sprawdzamy, czy dodanie go do stosu tworzy zakręt zgodnie z ruchem wskazówek zegara, czy przeciwnie. Jeśli zakręt jest zgodnie z ruchem wskazówek zegara, usuwamy punkt (lub punkty) ze stosu, ponieważ nie należy on do otoczki wypukłej. W przeciwnym wypadku dodajemy punkt do stosu i sprawdzamy następne punkty. Po przejściu przez wszystkie punkty stos zawiera wierzchołki tworzące otoczkę wypukłą.

Podproblem Transportowania Płotu

Odcinki o długości 1 są produkowane w fabryce i siecią wąskich dróg przenoszone do punktów odbioru przez tragarzy (mogą je ciąć).

Interpretacja

Fabryka jest jednym z punktów orientacyjnych. Punkty odbioru to wierzchołki otoczki wypukłej. Graf powstały z połączenia niektórych punktów orientacyjnych to sieć wąskich dróg. Celem jest jak najszybsze zbudowanie płotu, dlatego zakładam, że przejście z jednego punktu orientacyjnego do drugiego zajmuje jedną godzinę. Pary tragarzy niosą odcinki o długości 1 i jeżeli jest taka potrzeba, tną go na miejscu, w punktach odbioru.

Wejście

Wejściem ostatniego podproblemu jest graf (stworzony z punktów orientacyjnych i sieci wąskich dróg) wraz z zaznaczoną fabryką, a także wyniki poprzednich podproblemów, czyli liczba par tragarzy oraz indeksy otoczki wypukłej.

Przebieg transportu

Tragarze startują w fabryce. Każda para bierze odcinek i szuka najdalszego niezbudowanego punktu płotu. Potem algorytmem BFS znajdujemy najkrótszą ścieżkę z fabryki do wierzchołka płotu. Każda para porusza się jednocześnie, a jedno przejście trwa godzinę. Gdy tragarze dotrą na miejsce, muszą przejść przez krawędź płotu, aby do niego dobudować do niego odcinek, który niosą. Gdy tragarzom skończy się odcinek, muszą wrócić do fabryki po kolejny.

Problem “Maszyna Informatyka”

Podproblem korekcji frazy

Heretyk i Informatyk postanowili zapisać melodię w maszynie dla potomności. Nie wiedzą jednak w jaki sposób naprawić okropny dźwięk “poli”, tak by znów brzmiał poprawnie. Co więcej, inne fragmenty melodii mogły już paść ofiarą podobnej usterki.

Zauważmy, że jedyną różnicą między dźwiękiem budzącym grozę - “poli”, a przyjemną melodią “boli” jest to, że p jest literą “skierowaną w dół”, natomiast b - “skierowaną w góre”. Stwórzmy zatem listę potencjalnych “złych” liter.

“Zła” litera	“Dobra” litera
p	b
q	d
y	h
m	w
t	f
u	n
j	r
a	g

Przyjrzyjmy się algorytmom dopasowania wzorca.

W zasadzie do tego zadania możemy wytypować trzy możliwości - algorytmy Knutha-Morrisa-Pratta, Rabina-Karpa oraz naiwny. W tym przypadku uznamy, że najbardziej optymalnym wyborem jest algorytm naiwny - wyjaśnienie znajduje się w dalszej części rozważań.

Naiwny algorytm wyszukiwania wzorca

Dane wejściowe: ciąg znaków S , wzorzec W

Dane wyjściowe: indeksy wystąpień wzorca W w ciągu znaków S

Przebieg algorytmu

1. Utwórz pustą listę A o typie liczbowym - w niej przechowywać będziemy indeksy, gdzie znaleźliśmy wzorzec.
2. Przyjmijmy, że $x = 0$. Dopóki $x < |S|$:
 - a. $S[x] == W \Rightarrow A.append(x)$

- b. $x = x + 1$
3. Zwracamy listę wynikową A .

W naszym przypadku nie do końca interesuje nas indeks, na którym znajdują się litery, których szukamy, dlatego wprowadziliśmy małą zmianę w naszej implementacji algorytmu. Przez D oznaczmy słownik "złych liter", niech x, y będą odpowiednio "złą" i "dobrą" literą, zakładamy, że $D[x] = y$. Nasza modyfikacja głównie dotyczy punktu 2.a z powyższego schematu: $S[x] == W \Rightarrow S[x] = D[W]$, jak również brak tworzenia tablicy A .

Dzięki zastosowaniu słownika jako struktury danych w języku C# (co daje nam dostęp przykładowo do metody $D.ContainsKey$) oraz temu, że nasze wzorce są długości 1 udało nam się zredukować złożoność obliczeniową tego pod problemu do $O(n)$, gdzie n jest długością wprowadzonego ciągu znaków. Jest to zatem optymalniejsze podejście niż wykorzystanie algorytmu Rabina Karpa (w naszym przypadku $O(8(1 + n))$, $O(m + n)$ ogólnie - mamy 8 wzorców długości 1 [wartość m]), analogicznie dla algorytmu Knutha-Morrisa-Pratta.

Podproblem kompresji danych

Z poprzedniego pod problemu wiemy już, w jaki sposób skorygować ciąg znaków będący melodią na Stronie, pozostaje więc kwestia zapisu jej w maszynie informatyka. Założymy więc, że otrzymujemy poprawną melodię.

Podczas prób jej zapisu wykorzystywał on kodowanie liter do formy 5-bitowej liczby binarnej, natomiast napotkał przeszkodę w postaci niewystarczającej ilości miejsca w swojej maszynie. Musimy zatem znaleźć sposób, aby skompresować ciąg znaków na tyle, aby tego miejsca nie brakło.

Przyjrzyjmy się Kodowaniu Huffmana - jednemu z algorytmów bezstratnej kompresji danych.

Kodowanie Huffmana

Dane wejściowe: ciąg znaków S

Dane wyjściowe: drzewo umożliwiające stworzenie słownika kodów dla każdej z liter w S

Przebieg algorytmu

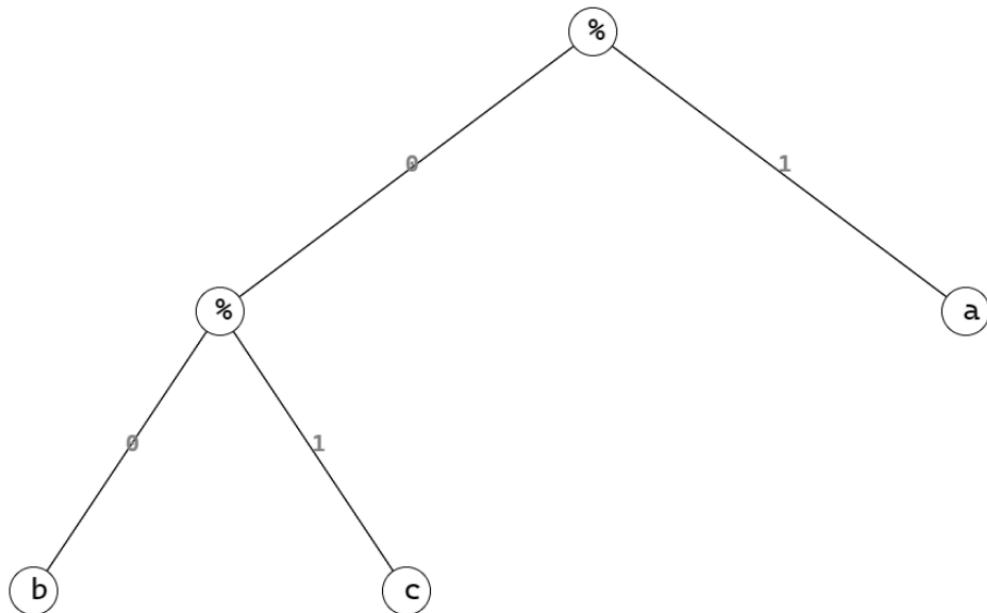
1. Określ częstotliwość występowania wszystkich znaków w S
2. Utwórz listę drzew binarnych, które przechowują **znak** oraz jego **częstotliwość występowania**.
3. Dopóki powyższa lista ma długość > 1 :
 - usuń z listy dwa drzewa o najmniejszej częstotliwości zapisanej w korzeniu
 - utwórz łącznik, którego wartość jest równa sumie częstotliwości usuniętych

wyżej drzew

- dopisz utworzony korzeń do listy

Uwaga. Kodowanie Huffmana jest algorytmem **niedeterministycznym** - nie określa on sposobu postępowania w wypadku, gdy dwa drzewa mają taką samą częstotliwość występowania. Na potrzeby naszego rozwiązania przyjęliśmy, że wówczas decydować będzie kolejność alfabetyczna. Ponadto dla lepszego zobrazowania łączników w wizualizacjach dostępnych w *Płaszczakowie* stały się one specjalnymi wierzchołkami zawierającymi znak %.

Tworzenie kodowania na podstawie otrzymanego drzewa



Utworzyliśmy tutaj drzewo dla prostej frazy "abac". Aby uzyskać kod dla danej litery należy zaczynając od pustego kodu (nazwijmy go K) przejść przez drzewo dodając do K kod 0, jeżeli przechodzimy do lewego potomka, lub kod 1 w przeciwnym wypadku.

Zauważmy, że w danej frazie litera **a** wystąpiła dwukrotnie, czyli była najczęściej występującą, stąd jej kod jest najkrótszy.

Porównajmy otrzymane kody do pierwotnego kodowania zaproponowanego przez Informatyka:

Litera	Pierwotne kodowanie	kodowanie Huffmana
a	00001	1
b	00010	00

c	00011	01
---	-------	----

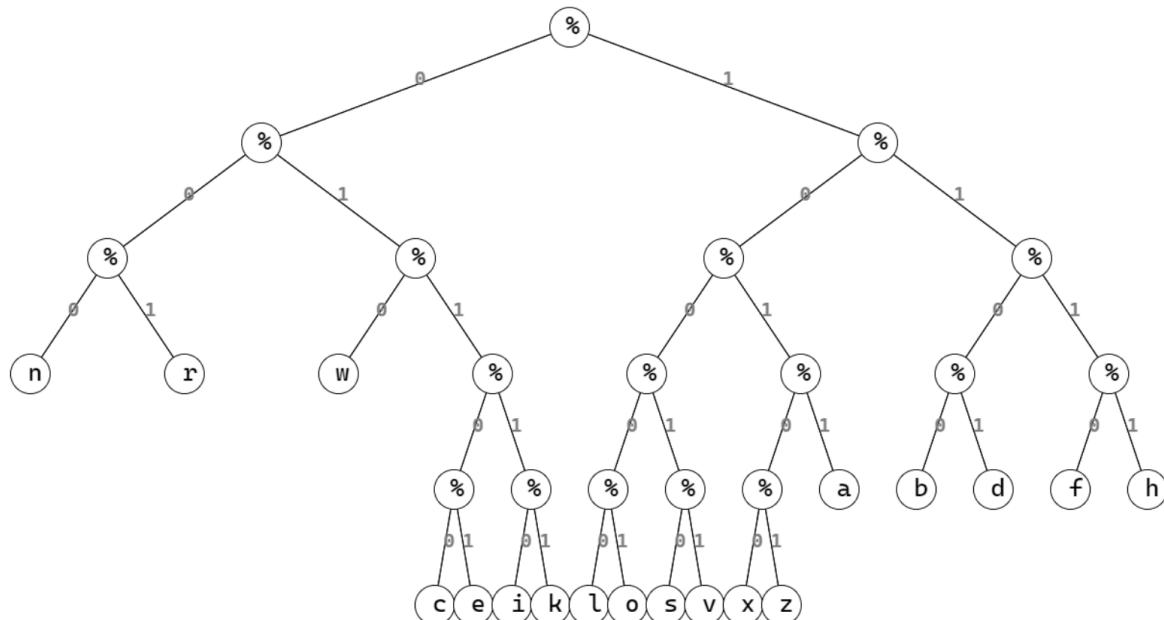
Kodowanie pierwotne	Kodowanie Huffmmana
00001000110000100010	011010
20 bitów	6 bitów

Zauważmy, że zastosowanie kodowania Huffmmana w tym przypadku poskutkowało oszczędnością **70%** miejsca względem pierwotnego rozwiązania zaproponowanego przez Informatyka.

Najbardziej pesymistyczny przypadek

Sprawdźmy również najbardziej pesymistyczny przypadek dla algorytmu. Wprowadźmy jako frazę wejściową wszystkie litery alfabetu łacińskiego, występujące równą ilość (oznaczmy ją jako n) razy.

Wówczas otrzymane drzewo prezentuje się następująco:



Jak będą prezentować się wówczas kody poszczególnych liter?

Znak	Kod binarny
u	0000
v	0001
w	0010
x	0011
y	0100
z	0101
m	01100

Znak	Kod binarny
n	01101
k	01110
l	01111
i	10000
j	10001
a	10010
b	10011
e	10100
f	10101
c	10110
d	10111
o	11000
p	11001
g	11010
h	11011
q	11100
r	11101
s	11110
t	11111

Zauważmy, że dla 6 liter długość kodów wynosi mniej niż 5. Oszczędność miejsca względem kodowania Informatyka wyniesie więc $6 \cdot n$ bitów (pamiętajmy, że każda litera w danej frazie występuje n razy).

Problem “Grafik Strażników”

Analiza problemu

Płot stanowi zbiór punktów orientacyjnych, przez które przechodzi **strażnik**. Każdy punkt orientacyjny ma przypisaną pewną wartość (jasność). Każdy płaszczak posiada pewną ilość **energii**. Cykl rozpoczyna się i kończy **w tym samym** punkcie, tworząc jedną pełną trasę. Aby płaszczak został strażnikiem, musi posiadać **co najmniej** taką samą energię, co **maksymalny** punkt orientacyjny na trasie.

Cel

Ustalić jak najszybciej grafik pracy strażników i jak najmniejszą liczbę odsłuchań melodii dla każdego strażnika.

Dane wejściowe

- Lista płaszczaków o pewnej energii.
- Lista wierzchołków o pewnej jasności.
- Lista krawędzi.
- Maksymalna ilość kroków.

Dane wyjściowe

- Posortowana lista płaszczaków z pewnymi ilościami odsłuchań melodii.

Warunki poruszania się

- Strażnik musi przejść przez wszystkie punkty orientacyjne płotu.
- W trakcie przemierzania płaszczak traci energię równą wartości punktu, przez który przechodzi.
- Strażnik posiada pewną określoną maksymalną liczbę kroków.
- Jeżeli pole poprzednie jest jaśniejsze niż obecne, płaszczak odpoczywa regenerując energię oraz resetuje licznik kroków.
- Płaszczak **musi** zatrzymać się, jeśli nie posiada wystarczająco dużo energii lub osiągnął limit kroków. Regeneruje wtedy energię oraz licznik kroków. Aczkolwiek jeżeli znajduje się na polu jaśniejszym (\geq) niż poprzedni, to **słucha** on melodii.

Złożoność

Ogólna złożoność algorytmu, to $O(P \times V)$, gdzie P , to liczba płaszczaków, a V , to liczba wierzchołków.