

Dokumentacja



Zespół 1: Emilia Flisikowska, Julia Kościelak, Anastasiya Tsiarnouskaya, Alesia Zubok

Spis treści

Spis treści.....	1
1.Wprowadzenie.....	3
1.1 Opis.....	3
1.3 Zakres Pracy.....	3
2. Graf.....	4
2.1 Ogólny opis.....	4
2.2 Implementacja.....	5
2.3 Testy jednostkowe.....	10
3. Maksymalny Przepływ.....	12
3.1 Opis.....	12
3.2 Implementacja.....	12
4. Koszt Naprawy Dróg.....	13
4.1 Opis.....	13
4.2 Implementacja.....	13
5. Ćwiartki.....	14
5.1 Ogólny Opis.....	14
5.2 Implementacja.....	14
5.2.1 Algorytm Grahama.....	16
5.2.2 Podział Punktów.....	17
5.2.3 Generowanie Ćwiartek.....	18
5.3 Testy jednostkowe.....	18
6. Wyszukiwanie słów.....	20
6.1 Opis.....	20
6.2 Implementacja.....	21
6.2.1 Algorytm Knutha-Morrisa-Pratta.....	24
6.2.3 Algorytm Boyera-Moore'a.....	24
6.3 Testy jednostkowe.....	25
7. Generator.....	26
7.1 Opis.....	26
7.2 Implementacja.....	26
7.3 Algorytm generowania map.....	29
8. Poprawność.....	30
8.1 Poprawność obliczeń maksymalnego przepływu.....	30
8.2 Poprawność obliczeń maksymalnego przepływu z minimalnym kosztem.....	30
8.3 Poprawność obliczeń otoczki wypukłej.....	31
9. Web-aplikacja.....	32
9.1 Wzorzec Model-View-Controller.....	32
9.2 Struktura aplikacji.....	32
9.2.1 Generowanie nowej mapy.....	32

9.2.2 Wyświetlenie obliczeń przepływów i ćwiartek.....	33
9.2.3 Wyświetlanie logów.....	36
9.3.3 Wyszukiwanie słów.....	36

1.Wprowadzenie

1.1 Opis

Burmistrz Shire dąży do zapewnienia możliwie najlepszego zaopatrzenia lokalnych karczm w piwo, które jest produkowane w browarach z jęczmienia uprawianego na pobliskich polach. Dysponuje on danymi dotyczącymi położenia pól uprawnych wraz z ich wydajnością, lokalizacji browarów oraz ich zdolności przerobowych, rozmieszczenia karczm wraz z ich ograniczoną pojemnością magazynową, a także stanu infrastruktury drogowej- w tym przepustowości dróg oraz kosztów ich naprawy. Dodatkowo kraina została podzielona na ćwiartki o zróżnicowanych warunkach upraw, co wpływa na ilość uzyskiwanego jęczmienia.

Zadanie polega na znalezieniu optymalnego planu transportu jęczmienia i piwa, który maksymalizuje ilość piwa dostarczanego do karczm, jednocześnie minimalizując koszty napraw wykorzystywanych dróg. Rozwiązanie musi uwzględniać zróżnicowane położenie obiektów (pól, browarów i karczm), podział krainy na ćwiartki o różnej wydajności upraw oraz ograniczenia wynikające z przepustowości dróg, wydajności pól i możliwości przerobowych browarów. Dodatkowo projekt obejmuje stworzenie interaktywnego narzędzia, które umożliwia wizualizację przepływów w sieci transportowej, podgląd wyników działania algorytmu oraz wyszukiwanie informacji w dokumentach tekstowych związanych z danymi wejściowymi i logami systemu.

1.2 Zakres Pracy

Alesia Zubok - implementacja algorytmów BM i KMP do wyszukiwania słów.

Anastasiya Tsiarnouskaya - implementacja grafu, implementacja algorytmu Forda-Fulkersona, implementacja serwera,

Emilia Flisikowska - generator danych wejściowych, modyfikacja algorytmu Forda-Fulkersona z uwzględnieniem kosztu naprawy dróg, wizualizacja rozwiązania,

Julia Kościelak - podział Shire na ćwiartki, implementacja Algorytmu Grahama.

Każdy członek zespołu odpowiadał także za dokumentację, implementację, analizę złożoności i poprawności, a także przygotowanie testów jednostkowych dla swojej części projektu.

2. Graf

2.1 Ogólny opis

W naszym projekcie graf jest przechowywany w postaci listy sąsiedztwa. Implementacyjnie zostało to zrealizowane jako mapa, w której kluczem jest Wierzchołek A, a wartością - mapa <Wierzchołek B, Krawędź (A -> B)>.

Podczas tworzenia grafu pierwszym pomysłem było przechowywanie w postaci macierzy sąsiedztwa, ponieważ wydawało się, że to jest najwygodniejsza postać do przeszukiwania (BFS) i szukania MaxFlow. Podczas testowania okazało się, że przy dużej liczbie dróg macierz sąsiedztwa zawiera zbyt wiele zer, co czyni ją nieefektywną pod względem zużycia pamięci. W związku z tym zdecydowaliśmy się zastąpić ją listą sąsiedztwa w postaci mapy.

Dodatkowo rozważaliśmy nadanie każdemu wierzchołkowi unikalnego id (automatycznie aktualizowanego podczas tworzenia obiektu wierzchołka) i przechowywanie grafu w postaci tablicy map typu <Wierzchołek B, Krawędź>. W ten sposób element tablicy o indeksie 0 oznaczał, że Wierzchołek o identyfikatorze 0 jest połączony z Wierzchołkiem B poprzez krawędź. Jednak to rozwiązanie okazało się mało wygodne i nieczytelne w kodzie, a dodatkowo wymagało każdorazowego sprawdzania, czy wierzchołek o danym id rzeczywiście się znajduje w grafie. Z tego powodu zmieniono to na mapę, opisaną wyżej.

2.2 Implementacja

Graf oraz funkcje do jego obsługi zostały zaimplementowane w klasie `Network`. W projekcie wykorzystano również klasy pomocnicze, takie jak:

Vertex

Pola:

- `int x, int y` – współrzędne wierzchołka w przestrzeni dwuwymiarowej.
- `String type` – typ wierzchołka, określany jako wartość tekstowa. W projekcie używane są typy: `Farmland`, `Intersection`, `Brewery`, `Tavern`.
- `int capacity` – pojemność wierzchołka, wykorzystywana przy ustawianiu pojemności farm, browarów i tawern.
- `int gottenFlow` – ilość otrzymanego przepływu. Wartość ta służy do poprawnych obliczeń przepływu jęczmienia: od farm, przez browary browarów, aż do tawern.

Edge

Pola:

- `int maxFlow` – maksymalny przepływ przez krawędź.
- `int currentFlow` – aktualny przepływ.
- `int residualFlow` – przepływ w grafie residualnym.
- `int repairCost` – koszt naprawy krawędzi.
- `Vertex from, Vertex to` – wierzchołki początkowy i końcowy.
- `Edge reverseEdge` – krawędź odwrotna.

-

Jak już było wspomniane, graf jest przechowywany jako mapa <Wierzchołek A, <Wierzchołek B, Krawędź A -> B>>.

Funkcje do obsługi grafu:

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>Vertex addVertex (int x, int y)</code>	<code>int x, y</code> - współrzędne dodawanego wierzchołku	Funkcja tworzy nowy wierzchołek oraz dodaje odpowiedni klucz wraz z pustą mapą do mapy reprezentującej graf, a także do mapy przechowującej wszystkie wierzchołki.	Stworzony wierzchołek.
<code>void addEdge (int maxFlow, int repairCost, int x1, int y1, int x2, int y2)</code>	<code>int maxFlow</code> – maksymalny przepływ krawędzi <code>int repairCost</code> – cena naprawy krawędzi <code>int x1, int y1</code> – współrzędne wierzchołku A <code>int x2, int y2</code> – współrzędne wierzchołku B	Po sprawdzeniu, czy wartości <code>maxFlow</code> i <code>repairCost</code> są nieujemne (w przypadku wartości ujemnych odpowiednie zmienne zostają ustawione na zero), pobierane są wierzchołki A i B z mapy wierzchołków. Jeżeli któryś z nich nie został wcześniej dodany, zostaje utworzony i dodany. Następnie są tworzone krawędzie <code>A->B</code> i <code>B->A</code> , z odpowiednio ustawionymi <code>maxFlow</code> i <code>repairCost</code>	Nic nie zwraca.
<code>void setMaxFlow (int flow, int x1, int y1, int x2, int y2)</code>	<code>int flow</code> – nowy maksymalny przepływ <code>int x1, int y1</code> – współrzędne wierzchołku A <code>int x2, int y2</code> – współrzędne wierzchołku B	Po sprawdzeniu, czy wartość <code>flow</code> jest nieujemna (jeśli jest ujemna, zostaje ustawiona na zero), pobierane są wierzchołki A i B, a następnie aktualizowane są wartości: przepływu maksymalnego, przepływu w sieci residualnej oraz aktualnego	Nic nie zwraca.

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
		<code>przepływu</code> . Funkcja ta jest wykorzystywana do aktualizacji sieci po zakończeniu obliczeń związanych z jęczmieniem, w celu wyznaczenia ilości piwa.	
<pre>void updateEdge (int flow, Vertex from, Vertex to)</pre>	<pre>int flow - aktualny przepływ Vertex from - wierzchołek początkowy Vertex to - wierzchołek końcowy</pre>	Po sprawdzeniu, czy <code>flow</code> jest nieujemny (jeśli jest ujemny, ustawiamy go na zero) oraz czy dane wierzchołki istnieją w grafie (w przeciwnym razie rzuca wyjątek <code>IllegalArgumentException</code>), aktualizujemy wartości <code>przepływu w sieci residualnej</code> oraz <code>aktualnego przepływu</code> .	Nic nie zwraca.
<pre>Vertex addSourceVertex (String sourceName)</pre>	<pre>String sourceName - nazwa wierzchołków, które zostaną połączone wspólnym wierzchołkiem źródłowym</pre>	Dodaje wierzchołek źródłowy o specjalnie zdefiniowanych współrzędnych. Następnie tworzy krawędzie od tego źródła do wszystkich wierzchołków typu <code>sourceName</code> (np. <code>Farmland</code>). Przepustowość tych krawędzi jest równa pojemności odpowiadającego wierzchołka.	Wierzchołek źródłowy.

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<pre>Vertex addSinkVertex (String sinkName)</pre>	<pre>String sinkName</pre> – nazwa wierzchołków, które zostaną połączone wspólnym ujściem	<p>Dodaje wierzchołek ujścia o specjalnie określonych współrzędnych. Następnie tworzy krawędzie z ujścia do wszystkich wierzchołków typu <code>sinkName</code> (np. <code>Tavern</code>). Przepustowość tych krawędzi jest równa pojemności odpowiadającego wierzchołka.</p>	Wierzchołek ujścia.
<pre>void deleteSourceVertex (Vertex sourceVert)</pre>	<pre>Vertex sourceVert</pre> – usuwany wierzchołek	<p>Usuwa wierzchołek źródła oraz wszystkie z nim powiązane krawędzie z grafu.</p>	Nic nie zwraca.
<pre>void deleteSinkVertex (Vertex sinkVert)</pre>	<pre>Vertex sinkVert</pre> – usuwany wierzchołek	<p>Usuwa wierzchołek ujścia oraz powiązane z nim krawędzie.</p>	Nic nie zwraca.
<pre>boolean BFS (Vertex src, Vertex dest)</pre>	<pre>Vertex src</pre> – wierzchołek początkowy <pre>Vertex dest</pre> – wierzchołek końcowy	<p>Implementacja algorytmu <code>BFS (Breadth-First Search)</code> służącego do znajdowania ścieżki z wierzchołka początkowego <code>src</code> do wierzchołka końcowego <code>dest</code>. W trakcie działania uzupełnia mapę <code>previousElements</code>, która przechowuje poprzedników każdego wierzchołka na znalezionej ścieżce.</p> <p>Funkcja ta jest używana wewnętrznie przez</p>	<code>true</code> , jeżeli ścieżka została znaleziona.

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
		<p>algorytm <code>BFSMaxFlow</code>.</p> <p>Złożoność oraz analiza algorytmu zostały przedstawione w części 3 dokumentacji „Maksymalny przepływ”.</p>	
<pre>int BFSMaxFlow (Vertex src, Vertex dest)</pre>	<p><code>Vertex src</code> – wierzchołek początkowy</p> <p><code>Vertex dest</code> – wierzchołek końcowy</p>	<p>Oblicza maksymalny możliwy przepływ z wierzchołka <code>src</code> do <code>dest</code> w grafie przepływu, bazując na znalezionych ścieżkach.</p> <p>W każdej iteracji znajduje ścieżkę przy za pomocą metody <code>BFS()</code> i określa <code>minimalny przepływ</code> możliwy do przesłania tą ścieżką.</p> <p>Następnie aktualizuje przepływy na odpowiednich krawędziach oraz zwiększa licznik <code>maxFlow</code> o wartość przesłanego przepływu.</p> <p>Dodatkowo aktualizuje pole <code>gottenFlow</code> w wierzchołku docelowym (<code>dest</code>), zwiększając je o ilość przesłanego przepływu.</p> <p>Złożoność oraz analiza algorytmu zostały przedstawione w części 3 dokumentacji „Maksymalny przepływ”.</p>	Maksymalny przepływ.

2.3 Testy jednostkowe

Zostały przeprowadzone testy jednostkowe dla następujących funkcji:

Funkcja	Sprawdzone przypadki	Zachowanie funkcji (zgodnie z testem)
<pre>void addEdge (int maxFlow, int repairCost, int x1, int y1, int x2, int y2)</pre>	<code>maxFlow < 0</code> oraz <code>repairCost < 0</code>	Po ustawieniu wartości <code>maxFlow</code> i <code>repairCost</code> na zero, krawędź zostaje pomyślnie dodana do sieci
	<code>maxFlow >= 0</code> oraz <code>repairCost >= 0</code>	Krawędź zostaje pomyślnie dodana do sieci
<pre>void setMaxFlow (int flow, int x1, int y1, int x2, int y2)</pre>	<code>flow < 0</code>	Podana wartość <code>flow</code> zostaje ustawiona na zero, a krawędź zostaje pomyślnie zaktualizowana
	<code>flow >= 0</code>	Podana wartość <code>flow</code> zostaje ustawiona na zero, a krawędź zostaje pomyślnie zaktualizowana
<pre>void updateEdge (int flow, Vertex from, Vertex to)</pre>	<code>flow >= 0</code>	Krawędź zostaje pomyślnie zaktualizowana
	<code>flow < 0</code>	Podana wartość <code>flow</code> zostaje ustawiona na zero, a krawędź zostaje pomyślnie zaktualizowana
	Nieprawidłowy wierzchołek	Zostaje rzucony wyjątek
	<code>from == to</code>	Zostaje rzucony wyjątek
<pre>Vertex addSourceVertex (String sourceName)</pre>	Pomyślnie dodawanie wierzchołków źródłowych (farm, skrzyżowań, browarni, tavern)	Podany wierzchołek zostaje dodany pomyślnie
<pre>Vertex addSinkVertex (String sinkName)</pre>	Pomyślnie dodawanie wierzchołków ujścia (farm, skrzyżowań, browarni, tavern)	Podany wierzchołek zostaje dodany pomyślnie
<pre>void deleteSourceVertex (Vertex sourceVert)</pre>	Pomyślnie usunięcie wierzchołków źródłowych (farm, skrzyżowań, browarni, tavern)	Podany wierzchołek zostaje usunięty pomyślnie
<pre>void deleteSinkVertex</pre>	Pomyślnie usunięcie wierzchołków	Podany wierzchołek zostaje

<code>(Vertex sinkVert)</code>	ujścia (farm, skrzyżowań, browarni, tavern)	usunięty pomyślnie
<code>boolean BFS (Vertex src, Vertex dest)</code>	Graf prosty, droga istnieje	<code>true</code>
	Graf skomplikowany, droga istnieje	<code>true</code>
	Droga nie istnieje	<code>false</code>
	Graf prosty, <code>maxFlow</code> jest równy 0 na jednej z krawędzi	<code>false</code>
	Graf skomplikowany, na wszystkich drogach jest krawędź z zerowym <code>maxFlow</code>	<code>false</code>
	<code>src == dest</code>	<code>true</code>
<code>int BFSMaxFlow (Vertex src, Vertex dest)</code>	Graf skomplikowany, istnieje droga	Wartość <code>maxFlow</code> zostaje obliczona poprawnie
	Graf skomplikowany, droga nie istnieje	0
	Graf prosty z "wąskim gardłem"	Wartość <code>maxFlow</code> zostaje obliczona poprawnie
<code>boolean Dijkstra (Vertex src, Vertex dest, Map<Vertex, Vertex> previousVertices)</code>	Graf prosty, droga istnieje	<code>true</code>
	Graf skomplikowany, droga istnieje	<code>true</code>
	Droga nie istnieje	<code>false</code>
	Graf skomplikowany, na wszystkich drogach jest krawędź z zerowym <code>maxFlow</code>	<code>false</code>
<code>int minCostMaxFlow (Vertex src, Vertex dest)</code>	Graf skomplikowany, istnieje droga	Wartość <code>minCostMaxFlow</code> zostaje obliczona poprawnie
	Graf skomplikowany, droga nie istnieje	0
	Graf prosty z "wąskim gardłem"	Wartość <code>minCostMaxFlow</code> zostaje obliczona poprawnie
	Graf z cyklem	Wartość <code>minCostMaxFlow</code> zostaje obliczona poprawnie
	Graf skomplikowany, istnieje wiele dróg	Wartość <code>minCostMaxFlow</code> zwraca wartość z najniższym kosztem naprawy dróg

3. Maksymalny Przepływ

3.1 Opis

W tej części projektu został rozwiązany problem maksymalnego przepływu w sieci. Do jego realizacji wykorzystano algorytm Forda-Fulkersona.

Podczas implementacji pojawił się problem związany z tym, że klasyczny algorytm Forda-Fulkersona zakłada istnienie jednego źródła i jednego ujścia w sieci przepływowej. W naszym przypadku sieć składała się z wielu farm, browarów i tawern połączonych ze sobą, co wymagało dostosowania algorytmu do nietypowej struktury.

W celu rozwiązania tego podproblemu zaimplementowano funkcje umożliwiające dodawanie oraz usuwanie wspólnych wierzchołków źródłowych i ujściowych. Dzięki temu możliwe było zastosowanie algorytmu w naszej sieci.

3.2 Implementacja

Algorytm Forda-Fulkersona został zaimplementowany z użyciem przeszukiwania wszerz (BFS) (implementacja Edmondsa-Karpa) do znajdowania ścieżek powiększających w grafie przepływowym. Implementacja opiera się na następujących krokach:

1. Łączenie odpowiednich wierzchołków źródłowych i ujściowych (Sieć pierwsza: farmy i browary; sieć druga: browary i tawerny)
2. Wielokrotne wyszukiwanie ścieżki powiększającej za pomocą algorytmu BFS od źródła do ujścia.
3. Wyznaczenie minimalnej przepustowości na znalezionej ścieżce.
4. Aktualizacja przepustowości w grafie rezydualnym – zmniejszenie przepustowości na krawędziach bezpośrednich i zwiększenie na krawędziach przeciwnych (rewersyjnych).
5. Sumowanie całkowitego przepływu, aż do momentu, gdy nie da się znaleźć kolejnej ścieżki powiększającej.

W celu uniknięcia zapętlenia, wszystkie krawędzie mają przepustowości nieujemne, a algorytm zatrzymuje się, gdy BFS nie może znaleźć ścieżki do ujścia.

Wejście: Graf skierowany $G=(V,E)$, przepustowość $c(u,v)$ dla każdej krawędzi $(u,v) \in E$, wierzchołek źródłowy $source \in V$, wierzchołek ujściowy $sink \in V$.

Wyjście: Maksymalny przepływ $maxFlow$ z wierzchołka $source$ do $sink$

Złożoność czasowa: $O(V \cdot E^2)$

Złożoność pamięciowa: $O(V+E)$ (przechowywanie grafu, kolejki dla BFS, tabeli pomocnicze)

4. Koszt Naprawy Dróg

4.1 Opis

Koszt naprawy dróg dla danego przepływu to suma kosztów naprawy poszczególnych odcinków, pomnożonych przez przepływ, który przez nie przechodzi.

W celu uwzględnienia minimalnego kosztu naprawy dróg przy zachowaniu maksymalnego przepływu, zmodyfikowaliśmy nasze rozwiązanie, zastępując algorytm Forda-Fulkersona algorytmem Successive Shortest Path.

Początkowo próba rozwiązania problemu zakładała implementację algorytmu Cycle Cancelling, jednak zrezygnowaliśmy z tego podejścia ze względu na jego złożoność implementacyjną oraz konieczność wprowadzenia wielu zmian w kodzie.

4.2 Implementacja

Do wyszukiwania ścieżek o najmniejszym łącznym koszcie w grafie rezydualnym wykorzystano algorytm Dijkstry.

Implementacja opiera się na następujących krokach:

1. Łączenie odpowiednich wierzchołków źródłowych i ujściowych (Sieć pierwsza: farmy i browary; sieć druga: browary i tawerny)
2. Wielokrotne wyszukiwanie ścieżki o najmniejszym koszcie za pomocą algorytmu Dijkstry od źródła do ujścia
3. Określenie ilości przepływu możliwej do przesłania wzdłuż znalezionej ścieżki
4. Aktualizacja przepustowości i kosztów w grafie rezydualnym – zmniejszenie przepustowości na krawędziach bezpośrednich i zwiększenie na krawędziach przeciwnych (rewersyjnych)
5. Sumowanie całkowitego przepływu i kosztu, aż do momentu, gdy nie da się znaleźć kolejnej ścieżki

W celu uniknięcia zapętlenia, wszystkie krawędzie mają nieujemne koszty naprawy, a algorytm zatrzymuje się, gdy algorytm Dijkstry nie znajdzie ścieżki o dodatnim przepływie od źródła do ujścia.

Wejście: Graf skierowany $G = (V, E)$, przepustowość $c(u,v)$ oraz koszt naprawy $k(u,v)$ dla każdej krawędzi $(u,v) \in E$, wierzchołek źródłowy $source \in V$, wierzchołek ujściowy $sink \in V$

Wyjście: Maksymalny przepływ przy minimalnym łącznym koszcie przesyłu

Złożoność czasowa: $O(F \cdot ((V + E) \log V))$, gdzie F – wartość maksymalnego przepływu

(Złożoność $(V + E) \log V$ wynika z tego, że algorytm Dijkstry przetwarza każdy wierzchołek i każdą krawędź co najwyżej raz, a operacje na kolejce priorytetowej (dodawanie i usuwanie wierzchołków) wykonywane są w czasie $O(\log V)$).

Złożoność pamięciowa: $O(V + E)$ (reprezentacja grafu, struktury pomocnicze w Dijkstrze)

5. Ćwiartki

5.1 Ogólny Opis

Celem rozdziału jest przedstawienie rozwiązania problemu podziału pól uprawnych (Farmlands) na ćwiartki - obszary będące wielokątami wypukłymi, wyznaczanymi za pomocą otoczki wypukłej. Podział ten umożliwia efektywne zarządzanie produkcją jęczmienia, ponieważ wszystkie pola w obrębie danej ćwiartki mają taką samą wydajność produkcyjną.

Szczegółowe opisy algorytmów i rozwiązań są zawarte w kolejnych podsekcjach.

5.2 Implementacja

W naszym projekcie ćwiartki (Quadrant) są budowane na podstawie punktów pól uprawnych (Farmland), które wcześniej są dzielone na grupy.

Najpierw liczba ćwiartek k jest obliczana automatycznie w funkcji `computeK()`. Jest to największa potęga dwójki taka, aby każda ćwiartka miała co najmniej 3 punkty (wymagane do utworzenia otoczki wypukłej). Gdyby pól było za mało, otoczka nie mogłaby powstać, a wtedy wszystkie punkty są "ćwiartką".

Grupowanie punktów odbywa się za pomocą funkcji `dividePoints()`, która działa rekurencyjnie. Punkty są na każdym poziomie sortowane naprzemiennie według współrzędnej X i Y , po czym dzielone na dwie grupy, aż osiągnięta zostanie żądana liczba k . Grupy te nie są oparte na siatce czy stałych przedziałach - wynikają z faktycznego rozkładu punktów w przestrzeni.

Dla każdej grupy wyznaczana jest otoczka wypukła przy pomocy algorytmu Grahama (`grahamAlgorithm()`).

- Najpierw wyszukiwany jest punkt początkowy (o najmniejszej współrzędnej Y , a w przypadku remisu - X)
- Następnie punkty są sortowane według współrzędnej kątowej
- Algorytm skanuje punkty i odrzuca takie, które nie tworzą "skrętu w prawo", budując stos punktów tworzących otoczkę wypukłą.

Gotowe ćwiartki (`Quadrant`) przechowują zarówno listę wszystkich swoich pól (`Farmland`), jak i listę punktów tworzących ich granicę (`hull`). Dodatkowo, każdej ćwiartce przypisywana jest losowa produkcja (`productionPerPlot`), która jest kopiowana do pól w tej ćwiartce.

Podstawowe klasy odpowiedzialne za realizację tej części projektu to:

QuadrantManager - zarządza podziałem pól uprawnych na ćwiartki oraz generowaniem otoczki wypukłej dla każdej ćwiartki.

Pola:

- `ArrayList<Farmland> farmlands` - lista punktów reprezentujących pola uprawne, które mają zostać podzielone na ćwiartki.
- `int k` - liczba ćwiartek, wyliczana jako największa potęga dwójki, tak aby każda grupa miała co najmniej 3 pola uprawne. (w przypadku mniejszej ilości punktów, $k = 1$ a "ćwiartką" jest cały obszar).

Quadrant - klasa reprezentująca ćwiartkę.

Pola:

- `ArrayList<Point2D> hull` – lista punktów tworzących otoczkę wypukłą, definiującą granice ćwiartki.
- `ArrayList<Farmland> farmlands` - lista pól uprawnych należących do tej ćwiartki.
- `int productionPerPlot` – wartość produkcji jęczmienia przypisana do pól w obrębie ćwiartki.

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>int computeK(int numberOfFarmlands)</code>	<code>int numberOfFarmlands</code> - liczba punktów pól uprawnych	<i>Metoda pomocnicza</i> Oblicza największą potęgę dwójki, tak aby w każdej grupie były przynajmniej 3 punkty. Jeśli liczba punktów jest mniejsza od 3, zwraca 1.	Liczba k - liczba ćwiartek.
<code>int calculateDeterminant(Point2D p1, Point2D p2, Point2D p3)</code>	<code>Point2D p1, Point2D p2, Point2D p3</code> - trzy punkty na płaszczyźnie.	<i>Metoda pomocnicza</i> Oblicza wyznacznik, który wskazuje na kierunek skrętu między 3 punktami (do zastosowania w algorytmie Grahama)	Liczba całkowita: dodatnia (skręt w lewo), ujemna (skręt w prawo) lub 0 (punkty współliniowe)

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>ArrayList<Point2D> grahamAlgorithm(ArrayList<Point2D> points)</code>	<code>ArrayList<Point2D> points</code> - lista punktów do otoczenia	Implementacja algorytmu Grahama - zwraca punkty tworzące otoczkę wypukłą (granice danego obszaru) Złożoność oraz analiza algorytmu zostały przedstawione w części 5.2.1 dokumentacji „Algorytm Grahama”.	Lista punktów tworzących otoczkę
<code>ArrayList<ArrayList<Farmland>> dividePoints(ArrayList<Farmland> points, int k, boolean divideByX)</code>	<code>ArrayList<Farmland> points</code> - lista punktów do podziału <code>int k</code> - liczba ćwiartek <code>boolean divideByX</code> - flaga wskazująca czy dzielić względem osi X (true) czy Y (false).	Dzieli wejściową listę punktów na podstawie rekursywnego dzielenia względem osi X i Y naprzemiennie. Szczegóły dotyczące złożoności i działania funkcji <code>dividePoints</code> przedstawiono w części 5.2.2 dokumentacji „Podział Punktów”.	Lista k grup, w których każdy podzbiór zawiera bliskie sobie punkty, tworząc podstawę do dalszego wyznaczania otoczek wypukłych.
<code>ArrayList<Quadrant> createQuadrants()</code>	brak argumentów	Tworzy listę obiektów <code>Quadrant</code> na podstawie wcześniej podzielonych danych i funkcji. Szczegóły dotyczące złożoności i działania funkcji <code>dividePoints</code> przedstawiono w części 5.2.3 dokumentacji „Generowanie Ćwiartek”.	Lista utworzonych ćwiartek (<code>Quadrant</code>)

5.2.1 Algorytm Grahama

Cel: wyznaczenie otoczki wypukłej, która pełni funkcję granicy ćwiartki - obszaru zawierającego grupę punktów reprezentujących pola uprawne.

Opis: Algorytm Grahama rozpoczyna od znalezienia punktu o najmniejszych współrzędnych (najniższy lewy punkt). Następnie sortuje pozostałe punkty według kąta względem tego punktu i buduje otoczkę wypukłą, odrzucając punkty powodujące „obrót w prawo” (nielewe skręty).

Alternatywy: Rozważaliśmy użycie algorytmu Jarvisa, który “owija” punkty wokół zbioru tworząc otoczkę. Choć jest łatwiejszy do zaimplementowania i bardziej intuicyjny, to dla

większych zbiorów jest mniej wydajny. Z tego powodu został odrzucony na rzecz efektywniejszego algorytmu Grahama.

Wejście: Zbiór punktów 2D reprezentujących położenie pól uprawnych.

Wyjście: Uporządkowany (według współrzędnej kątowej względem punktu startowego.) zbiór punktów pól uprawnych tworzących otoczkę wypukłą.

Złożoność czasowa: $O(n \log(n))$

Złożoność pamięciowa: $O(n)$

5.2.2 Podział Punktów

Cel: Podział zbioru punktów reprezentujących pola uprawne na k grup, z zachowaniem przestrzennego porządku punktów i zbalansowanej liczebności. Celem podziału jest przygotowanie danych do późniejszego wyznaczenia granic ćwiartek, poprzez algorytm Grahama opisany wcześniej.

Opis: Algorytm dzieli punkty rekurencyjnie, sortując je naprzemiennie względem współrzędnej X i Y .

Dla każdej iteracji zbiór dzielony jest na dwie części połówkowe, aż do uzyskania k podzbiorów. Liczba k jest wyznaczana jako największa potęga 2 (za pomocą `computeK`), dla każdej grupy która zawiera co najmniej 3 punkty(tak, aby możliwe było wyznaczenie otoczki wypukłej).

Czyli, podczas każdej rekursji, w przypadku kiedy $k == 1$ zwracamy bieżący zbiór bez dalszego dzielenia, a w przeciwnym razie punkty są sortowane według X lub Y , a następnie dzielone na dwie części(dla których rekursywnie wywoływany jest podział z $k/2$).

Alternatywy: Początkowo rozważaliśmy prostszy podział geometryczny - poprzez podział przestrzeni na cztery prostokątne obszary - ćwiartki(I, II, III i IV). Uznałyśmy to rozwiązanie zbyt mało ciekawe. Obecne rozwiązanie pozwala dowolnie zwiększać ilość ćwiartek (1, 2, 4, 8 ...) oraz zachowuje balans liczebności grup.

Wejście: lista punktów pól uprawnych (zawierających ich pozycje), liczba ćwiartek oraz flaga która mówi o tym czy zaczynamy sortowanie od współrzędnej X czy Y .

Wyjście: Lista k podlist punktów, które reprezentują wstępnie podzielone ćwiartki(bez wyznaczonych otoczek).

Złożoność czasowa: $O(n \log(n) * \log(k))$, n - liczba pól uprawnych, k - liczba ćwiartek.

Złożoność pamięciowa: $O(n)$

5.2.3 Generowanie Ćwiartek

Cel: Utworzenie finalnych obszarów ćwiartek, reprezentowanych przez wypukłe wielokąty. Każda granica ma określoną granicę, wyznaczoną na podstawie otoczki wypukłej jej punktów.

Opis: Ten etap integruje funkcje opisane w poprzednich sekcjach. Najpierw, używając algorytmu podziału punktów (5.2.2), początkowy zbiór wszystkich pól uprawnych zostaje podzielony na k mniejszych, zbalansowanych grup. Następnie, dla każdej z tych grup, algorytm Grahama (5.2.1) jest wywoływany w celu wyznaczenia otoczki wypukłej. Punkty tworzące tę otoczkę definiują geometryczną granicę danej ćwiartki. Dodatkowo, każda nowo utworzona ćwiartka otrzymuje losowo wybraną wartość zdolności produkcyjnej, a wszystkie pola uprawne należące do tej ćwiartki dziedziczą tę wartość.

Wejście: Zbiór punktów 2D reprezentujących położenie pól uprawnych.

Wyjście: Lista obiektów Quadrant, gdzie każdy Quadrant zawiera swoją otoczkę wypukłą (granicę), przypisane do niej pola uprawne oraz ustaloną zdolność produkcyjną.

Złożoność czasowa: $O(n \log(n) * \log(k))$, gdzie n to liczba pól uprawnych, a k to liczba wyznaczonych ćwiartek. Złożoność wynika z dominacji funkcji `dividePoints()` oraz kosztu `grahamAlgorithm()` dla wszystkich grup.

Złożoność pamięciowa: $O(N)$.

5.3 Testy jednostkowe

Zostały przeprowadzone testy jednostkowe dla następujących funkcji:

Funkcja	Sprawdzone przypadki	Zachowanie funkcji (zgodnie z testem)
<code>computeK(int numberOfFarmlands)</code>	<code>numberOfFarmlands < 3</code> .	Zwraca 1, ponieważ nie ma wystarczającej liczby pól uprawnych do utworzenia ćwiartki (min. 3 punkty).
	<code>numberOfFarmlands = 3</code> .	Zwraca jeden, ponieważ jest wystarczająco na jedną ćwiartkę.
	Różne wartości <code>numberOfFarmlands</code> (np. 6, 10, 12, 30).	Zwraca największą potęgę 2, dla której każda grupa będzie miała co najmniej 3 punkty (np. dla 30 zwraca 8, ponieważ $30/3=10(8<10)$).

Funkcja	Sprawdzone przypadki	Zachowanie funkcji (zgodnie z testem)
	Wartość niewystarczająca na 2 ćwiartki - <code>numberOfFarmlands</code> = 4.	Zwraca 1, ponieważ nie ma wystarczającej liczby pól uprawnych do podziału na dwie lub więcej ćwiartek, każda po co najmniej 3 punkty.
<code>grahamAlgorithm(ArrayList<Point2D> points)</code>	Pusta lista punktów wejściowych.	Zwraca pustą listę punktów (otoczka wypukła dla pustego zbioru jest pusta).
	Lista zawierająca jeden punkt.	Zwraca listę zawierającą ten sam punkt (punkt jest swoją własną "otoczką" (granica))
	Lista zawierająca dwa punkty.	Zwraca listę zawierającą oba te punkty.
	Punkty współliniowe (np. (0,0), (1,1), (2,2), (3,3))	Zwraca listę zawierającą tylko skrajne punkty współliniowe (np. (0,0) i (3,3)), ignorując punkty wewnętrzne, ponieważ nie są one częścią otoczki wypukłej
<code>dividePoints(ArrayList<Farmland> points, int k, boolean divideByX)</code>	k = 1.	Zwraca oryginalną listę punktów jako pojedynczą grupę, ponieważ dalszy podział nie jest potrzebny
	k = 2, podział względem osi X .	Dzieli punkty na dwie grupy, gdzie pierwsza grupa zawiera punkty o mniejszych współrzędnych X, a druga o większych X
	Nieparzysta liczba punktów.	Dzieli punkty na dwie grupy o zbalansowanej liczebności (jedna grupa może mieć o jeden punkt więcej niż druga)
	Pusta lista punktów.	Zwraca listę zawierającą jedną pustą grupę, ponieważ nie ma punktów do podziału.
<code>createQuadrants()</code>	Pusta lista pól uprawnych (<code>farmlands</code>)	Zwraca pustą listę obiektów Quadrant
	Lista zawierająca jedno pole	Zwraca jedną ćwiartkę

Funkcja	Sprawdzane przypadki	Zachowanie funkcji (zgodnie z testem)
	uprawne.	zawierającą to pole uprawne, a "otoczka" składa się z tego pojedynczego punktu.
	Pięć pól uprawnych na każdej innej ćwiartce wykresu, w tym jedno centralne (np. (0,0) oraz rogi kwadratu).	Zwraca jedną ćwiartkę. Wszystkie pola uprawne należą do tej ćwiartki. Otoczka wypukła składa się z czterech skrajnych punktów, a punkt centralny nie jest jej częścią
	Punkty współliniowe.	Zwraca jedną ćwiartkę. Wszystkie pola uprawne należą do tej ćwiartki. Otoczka wypukła składa się tylko ze skrajnych punktów współliniowych.
	Duża lista pól uprawnych, umożliwiającą podział na 8 ćwiartek (zgodnie z <code>computeK</code>)	Zwraca osiem unikalnych ćwiartek. Każde pole uprawne zostaje przypisane do odpowiedniej ćwiartki, a każda ćwiartka ma swoją otoczkę wypukłą i przypisaną produkcję.

6. Wyszukiwanie słów

6.1 Opis

Wyszukiwanie wzorca w pliku lub własnym tekście przy użyciu dwóch algorytmów:

- KMP (Knuth-Morris-Pratt)
- BM (Boyer-Moore)

Użytkownik ma możliwość wybrania źródła tekstu (plik lub własny input), podania wzorca i wyboru algorytmu do wyszukiwania.

Alternatywy: Rozważono użycie algorytmu naiwnych dopasowań, który polega na przesuwaniu wzorca znak po znaku przez cały tekst i porównywaniu go od początku. Jest prosty w implementacji i zrozumiały, jego złożoność czasowa w najgorszym przypadku wynosi $O(n \cdot m)$, co czyni go nieefektywnym przy dużych danych.

6.2 Implementacja

Zaimplementowano na podstawie następujących kroków:

1. wczytanie tekstu z pliku lub uzyskanie tekstu bezpośrednio od użytkownika,
2. wprowadzenie wzorca przeznaczonego do wyszukania,
3. wybór algorytmu wyszukiwania – KMP, BM bądź KMP i BM razem,
4. przeprowadzenie procesu wyszukiwania wzorca w podanym tekście.

Wyniki są prezentowane użytkownikowi oraz zapisywane do pliku logów w celu ich archiwizacji.

Podstawowe klasy odpowiedzialne za realizację tej części projektu to:

Algorithms implementuje dwa algorytmy wyszukiwania wzorców w tekście:

- Knuth-Morris-Pratt (KMP)
- Boyer-Moore (BM)

Pola:

- **Result result** - obiekt przechowujący wyniki działania algorytmów (KMP, BM).
- **FileWriter writer** - obiekt zapisujący logi algorytmów do pliku output.txt.

Result przechowuje wyniki działania algorytmów KMP i BM.

Pola:

- **String BMResult** - zmienna przechowująca wynik działania algorytmu Boyera-Moore'a.

- `String KMPResult` - zmienna przechowująca wynik działania algorytmu Knutha-Morrisa-Pratta.

ReadFile odczytuje tekst z podanego pliku.

Pola:

- `FileWriter writer` - obiekt zapisujący logi związane z odczytem plików do output.txt.

Menu odpowiada za interfejs, wybór źródła tekstu oraz algorytmu wyszukiwania.

Pola:

- `Scanner scanner` - obiekt klasy `Scanner` do obsługi wejścia użytkownika z konsoli.
- `File file` - plik, w którym będzie wyszukiwany wzorzec.
- `String inputText` - tekst wprowadzony przez użytkownika.
- `String algorithmChoice` - wybór algorytmu.
- `Algorithms alg` - obiekt klasy **Algorithms**, wykonujący operacje wyszukiwania wzorca.
- `FileWriter writer` - obiekt zapisujący logi działania programu do pliku output.txt.

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>void list()</code>	brak argumentów	wyświetla opcje menu	nic nie zwraca
<code>move(String input, String pattern)</code>	<code>String input</code> - wybrana opcja <code>String pattern</code> - podany wzorzec	obsługuje opcję wybraną przez użytkownika	<code>false</code> , jeśli zostało wybrano zakończenie wyszukiwania
<code>void choiceAlgorithm(String text, String pattern)</code>	<code>String text</code> - tekst w którym będzie wyszukiwany wzorzec <code>String pattern</code> - podany wzorzec <code>String</code>	uruchamia wyszukiwanie wzorca w tekście zgodnie z wybranym algorytmem	nic nie zwraca

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>pattern, String algorithmChoice)</code>	<code>algorithmChoice</code> - numer wybranego algorytmu		
<code>int[] createPi(String pattern)</code>	<code>String pattern</code> - podany wzorzec	buduje tablicę prefiksów dla KMP	tablicę prefiksów
<code>int[] createBMNext(String pattern)</code>	<code>String pattern</code> - podany wzorzec	buduje tablicę przesunięć wzorca w good-suffix-heuristic dla BM	tablicę przesunięć BMNext
<code>int[] createLAST(String pattern)</code>	<code>String pattern</code> - podany wzorzec	buduje tablicę przesunięcia wzorca w bad-character-heuristic dla BM	tablicę przesunięć LAST
<code>void KMP(String text, String pattern)</code>	<code>String text</code> - podany tekst <code>String pattern</code> - podany wzorzec	wyszukuje wystąpienia wzorca w tekście za pomocą algorytmu KMP, zapisuje wyniki	nic nie zwraca
<code>void BM(String text, String pattern)</code>	<code>String text</code> - podany tekst <code>String pattern</code> - podany wzorzec	wyszukuje wystąpienia wzorca w tekście za pomocą algorytmu BM, zapisuje wyniki	nic nie zwraca

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>void addBMResult(String result)</code>	<code>String result</code> - tekstowy wynik działania algorytmu BM	dodaje wynik działania algorytmu BM do wewnętrznego bufora	nic nie zwraca
<code>void addKMPSResult(String result)</code>	<code>String result</code> - tekstowy wynik działania algorytmu KMP	dodaje wynik działania algorytmu KMP do wewnętrznego bufora	nic nie zwraca
<code>String getResult(String option)</code>	<code>String option</code> - określa, który wynik ma zostać zwrócony	zwraca rezultat wyszukiwania słów zgodnie z wybranym algorytmem	resultat zgodnie z wybranym algorytmem
<code>String reading(File file)</code>	<code>File file</code> - podany plik, który musi być odczytany	odczytuje tekst z podanego przez użytkownika pliku	odczytany tekst

6.2.1 Algorytm Knutha-Morrisa-Pratta

Opis: Algorytm Knutha-Morrisa-Pratta (KMP) służy do wyszukiwania wzorca w tekście. W celu uniknięcia zbędnych porównań, KMP najpierw przetwarza wzorzec i tworzy tablicę prefiksową (pi), która zawiera informacje o najdłuższych właściwych prefiksach. W momencie napotkania niezgodności algorytm nie cofa się w tekście, lecz przesuwą wzorzec zgodnie z wartościami znajdującymi się w tablicy pi.

Wejście: Tekst, w którym chcemy wyszukać wzorzec, oraz wzorzec, który chcemy odnaleźć.

Wyjście: Przesunięcie, o którym wzorzec występuje w podanym tekście, lub brak dopasowań, jeżeli wzorzec nie występuje.

Złożoność czasowa: $O(m+n)$; m - długość wzorca, n - długość tekstu

Złożoność pamięciowa: $O(m)$; m - długość wzorca

6.2.2 Algorytm Boyera-Moore'a

Opis: Algorytm Boyera-Moore'a służy do wyszukiwania wzorca w tekście. BM rozpoczyna przeszukiwanie od końca wzorca, co często umożliwia pomijanie znacznych fragmentów tekstu. Wykorzystuje przy tym dwie kluczowe heurystyki: bad-character-heuristic (LAST) oraz good-suffix-heuristic (BMNext), które określają, o ile można przesunąć wzorzec po nieudanym dopasowaniu.

Wejście: Tekst, w którym chcemy wyszukać wzorzec, oraz wzorzec, który chcemy odnaleźć.

Wyjście: Przesunięcie, o którym wzorzec występuje w podanym tekście, lub brak dopasowań, jeżeli wzorzec nie występuje.

Złożoność czasowa: $O(n \times m)$ (przypadek pesymistyczny), $O(n / m)$ (przypadek optymistyczny); m - długość wzorca, n - długość tekstu

Złożoność pamięciowa: $O(m + k)$; m - długość wzorca, $k = |AL| = 128$

6.3 Testy jednostkowe

Funkcja	Sprawdzane przypadki	Zachowanie funkcji (zgodnie z testem)
<code>createPi(String pattern)</code>	wprowadzenie wzorca z różnymi typami symboli (cyfry i litery)	zwraca poprawną tablicę pi, niezależnie od typu symboli
<code>KMP(String text, String pattern)</code>	wzorzec istnieje w tekście	wypisuje o jakim przesunięciu występuje wzorzec
	wzorzec nie istnieje w tekście	zawiadamia o braku wystąpień wzorca w tekście
<code>createBMNext(String pattern)</code>	budowa tablicy BMNext na podstawie wzorcy	zwraca poprawna tablice BMNext
<code>createLAST(String pattern)</code>	odczyt z tablicy LAST elementu nie istniejącego we wzorcu	rzuca -1, czyli brak elementu

Funkcja	Sprawdzane przypadki	Zachowanie funkcji (zgodnie z testem)
	odczyt z tablicy LAST elementy istniejącego we wzorcu	zwraca indeks tego elementu
<code>BM(String text, String pattern)</code>	wzorzec istnieje w tekście	wypisuje o jakim przesunięciu występuje wzorzec
	wzorzec nie istnieje w tekście	zawiadamia o braku wystąpień wzorca w tekście

7. Generator

7.1 Opis

Klasa Generator służy do generowania przykładowych danych wejściowych dla symulacji przepływu w sieci transportowej w Shire. Tworzy losową mapę obejmującą:

- drogi
- pola uprawne (farmlands)
- browary (breweries)
- karczmy (taverns)

Dane te są wykorzystywane do testowania algorytmów optymalizacji przepływu zboża i piwa.

7.2 Implementacja

Pola klasy Generator:

- `roadsCount` – liczba dróg do wygenerowania
- `farmlandsCount`, `breweriesCount`, `tavernsCount` – liczby poszczególnych obiektów
- `minRoadLength`, `maxRoadLength` – minimalna i maksymalna długość drogi
- `minDistanceBetweenPoints` – minimalna odległość między obiektami
- `minCoordinates`, `maxCoordinates` – ograniczenia przestrzenne dla współrzędnych

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<code>ArrayList<Road> generateRoads()</code>	brak argumentów	Generuje sieć dróg losowo, rozpoczynając od jednej drogi, a następnie dodając kolejne tak, aby przecinały istniejące i tworzyły realistyczną, spójną topologię. Dodawanie kończy się po osiągnięciu żądanej liczby dróg lub przekroczeniu limitu prób.	Lista wygenerowanych dróg

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
<pre>Road GenerateRandomRoad(double minLength, double maxLength)</pre>	<pre>double minLength-</pre> minimalna dozwolona długość generowanej drogi <pre>double maxLength-</pre> maksymalna dozwolona długość generowanej drogi	Losowo generuje drogę jako odcinek pomiędzy dwoma punktami w przestrzeni 2D. Zapewnia, że długość drogi mieści się w określonym przedziale (<code>minLength</code> , <code>maxLength</code>). Wybór punktów jest losowy w zadanym zakresie współrzędnych.	Nowo wygenerowana droga spełniająca kryteria długości.
<pre>ArrayList<Road> addRoad(ArrayList<Road> _roads)</pre>	<pre>ArrayList<Road> _roads-</pre> lista istniejących dróg	Dodaje nową drogę do sieci, pod warunkiem że przecina ona którąś z istniejących. W przypadku przecięcia, dzieli przecięte drogi w punktach przecięcia i aktualizuje listę dróg. Jeśli nie uda się znaleźć odpowiedniego przecięcia, zwraca oryginalną listę.	Zaktualizowana lista dróg
<pre><T extends Point2D> ArrayList<T> generateObjects(...)</pre>	<pre>ArrayList<Road> roads-</pre> lista istniejących dróg <pre>ArrayList<Point2D> existingObjects-</pre> obiekty już dodane do sieci <pre>IFactory<T> factory-</pre> fabryka tworząca obiekty danego typu <pre>int count-</pre> liczba obiektów do wygenerowania	Losowo generuje obiekty (Farmland, Brewery, Tavern) w pobliżu końców dróg, zachowując minimalny dystans od innych obiektów i dróg. Łączy każdy obiekt z siecią za pomocą nowej drogi.	Lista wygenerowanych obiektów danego typu
<pre>Data generate()</pre>	brak argumentów	Główna funkcja klasy <code>Generator</code> . Odpowiada za wygenerowanie kompletnej sieci – tworzy drogi, rozmieszcza pola uprawne, browary i karczmy, usuwa niepołączone drogi oraz losowo ustawia parametry każdej z nich (np. przepustowość i koszt	Obiekt <code>Data</code> zawierający sieć drogową i rozmieszczone obiekty

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość
		naprawy).	
<pre>Point2D intersection(Road a, Road b)</pre>	<pre>Road a- pierwsza droga Road b- druga droga</pre>	<p><i>Metoda pomocnicza</i> Oblicza punkt przecięcia dwóch dróg metodą wyznaczników.</p>	<p>Punkt przecięcia (<i>Point2D</i>) lub <i>null</i>, jeśli drogi się nie przecinają</p>
<pre>boolean linesEndsTouch(Road line1, Road line2)</pre>	<pre>Road line1- pierwsza droga Road line2- druga droga</pre>	<p><i>Metoda pomocnicza</i> Sprawdza, czy końce dwóch dróg stykają się ze sobą.</p>	<p><i>true</i>, jeśli drogi się stykają; <i>false</i> w przeciwnym przypadku</p>
<pre>boolean RoadWithoutObjects(...)</pre>	<pre>ArrayList<Road> roads ArrayList<Farmland> farmlands ArrayList<Brewery> breweries ArrayList<Tavern> taverns Road r- droga do sprawdzenia</pre>	<p><i>Metoda pomocnicza</i> Sprawdza, czy dana droga jest odizolowana od reszty sieci i nie prowadzi do żadnego obiektu.</p>	<p><i>true</i>, jeśli droga może zostać usunięta; <i>false</i>, jeśli jest powiązana z innym obiektem</p>
<pre>static List<Point2D> findRoadEnds(...)</pre>	<pre>ArrayList<Road> roads- lista wszystkich dróg ArrayList<Point2D> existingObjects- obiekty, które już istnieją</pre>	<p><i>Metoda pomocnicza</i> Zwraca wszystkie końce dróg, które nie są jeszcze powiązane z żadnym obiektem.</p>	<p>Lista punktów końcowych</p>

Nazwa funkcji	Argumenty	Opis działania	Zwracana wartość

7.3 Algorytm generowania map

Generator tworzy sieć dróg oraz rozmieszcza obiekty (farmy, browary i tawerny) w sposób częściowo losowy, z uwzględnieniem pewnych ograniczeń.

Proces zaczyna się od wygenerowania jednej losowej drogi o długości mieszczącej się w ustalonym zakresie. Następnie algorytm próbuje dodawać kolejne drogi, ale tylko takie, które przecinają już istniejące. Gdy nowa droga przecina istniejącą, w miejscu przecięcia tworzone jest skrzyżowanie, a obie drogi zostają podzielone na odcinki. Dodatkowo odrzucane są drogi, których skrzyżowania znalazłyby się zbyt blisko już istniejących punktów.

Po zbudowaniu sieci dróg, generator rozmieszcza obiekty. Umieszczane są one w pobliżu końców istniejących dróg, z zachowaniem minimalnej odległości od innych obiektów i bez przecinania już istniejących tras. Każdy obiekt jest połączony z siecią krótką drogą.

Browarom i tawernom przypisywana jest pojemność, określająca maksymalną ilość towaru, jaką mogą obsłużyć.

Na końcu usuwane są drogi, które nie są połączone z żadnym obiektem i nie prowadzą do innych dróg. Pozostałym drogom przypisywane są losowe wartości maksymalnego przepływu piwa, jęczmienia oraz kosztu naprawy.

Ręcznie wyliczony maksymalny przepływ dla tej sieci wynosi: 5, a koszt naprawy dróg 292 co się zgadza z obliczeniami wykonanymi przez zaimplementowany algorytm.

8.3 Poprawność obliczeń otoczki wypukłej

Rozważmy zbiór punktów:

$(0, 0), (1, 0), (2, 0), (2, 1), (4, 2), (3, 4), (1, 3), (5, 1), (3, 0), (2, 2)$ *przykład z wykładu*

```
Punkty:
  (0, 0)
  (1, 0)
  (2, 0)
  (2, 1)
  (4, 2)
  (3, 4)
  (1, 3)
  (5, 1)
  (3, 0)|
  (2, 2)
Otoczka:
  (0, 0)
  (3, 0)
  (5, 1)
  (3, 4)
  (1, 3)
```

Ręcznie wyliczona otoczka wynosi $(0,0), (3,0), (5,1), (3,4), (1,3)$, co się zgadza z obliczeniami wykonanymi przez zaimplementowany algorytm.

9. Web-aplikacja

W celu ułatwienia korzystania ze stworzonego programu zdecydowaliśmy postawić serwer oparty o framework Java Spring Boot. Także został wykorzystany wzorzec architektoniczny Model-View-Controller, o którym będzie mowa w odpowiedniej części tego rozdziału.

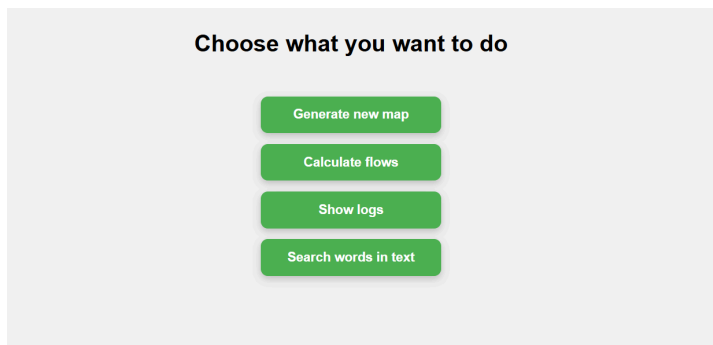
Nasza web-aplikacja pozwala na łatwy wybór opcji, wprowadzanie własnych danych do generowania mapy, łatwe wyszukiwanie słów i oglądanie logów.

9.1 Wzorzec Model-View-Controller

W naszej aplikacji ten wzorzec został wykorzystany w celu stworzenia komunikacji pomiędzy klientem a serwerem. Model reprezentuje dane przesyłane od użytkownika do serwera (np. dane do generowania mapy, tekst do wyszukiwania słów i same słowy) i odwrotnie, widokiem jest strona HTML wyświetlana użytkownikowi, kontrolerami są specjalne funkcje do obsługi zapytań, które pozwalają w odpowiedni moment zwrócić odpowiedni widok HTML.

9.2 Struktura aplikacji

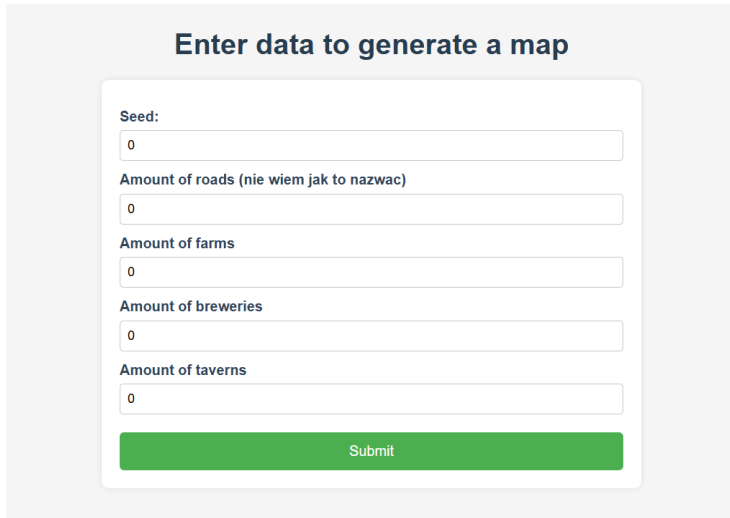
Cała aplikacja się uruchamia poprzez uruchomienie funkcji main w klasie StartServer. Po tym wpisujemy do przeglądarki adres <http://localhost:8080/menu> i pojawia się strona główna aplikacji, która wygląda następująco:



Ten widok jest obsługiwany w kontrolerze `public String menu(Model model)`. Ten kontroler pozwala na otrzymywanie danych od użytkownika, informujących serwer o wyborze dalszego działania aplikacji.

9.2.1 Generowanie nowej mapy

Po wybraniu opcji Generate new map otwiera się strona z formularzem, gdzie możemy, lecz nie musimy wpisać dane dla generatora mapy.



Enter data to generate a map

Seed:
0

Amount of roads (nie wiem jak to nazwac)
0

Amount of farms
0

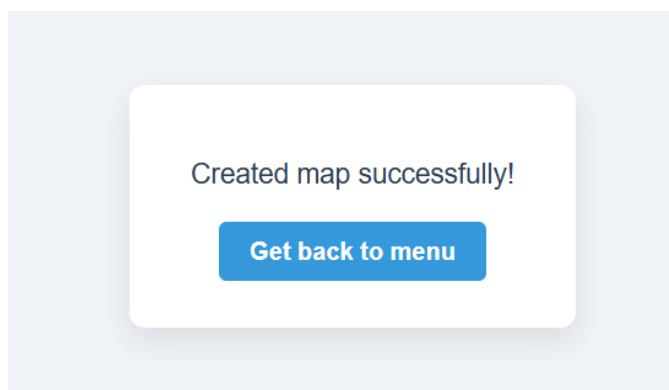
Amount of breweries
0

Amount of taverns
0

Submit

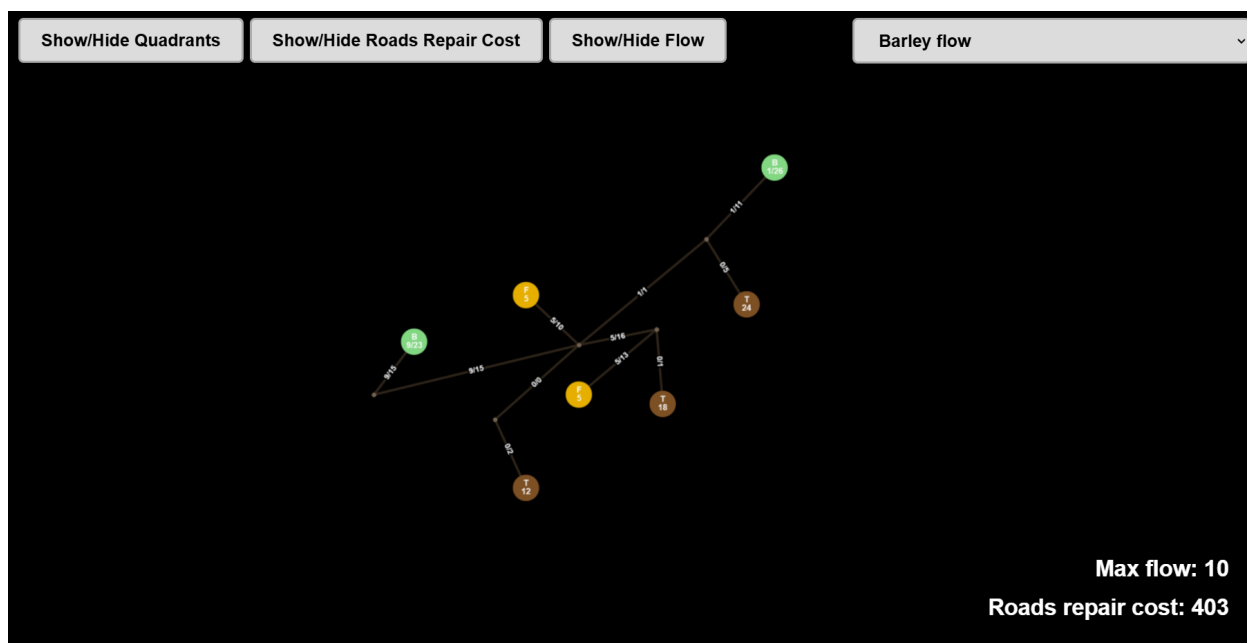
Ten widok jest obsługiwany przez kontrolery `public String sent(@ModelAttribute UserChoice userChoice, Model model)` i `public String generate(@ModelAttribute DataForGenerator dataForGenerator, Model model)`.

Domyślnie wszystkie pola są wyzerowane. Po wpisaniu danych i kliknięciu przycisku submit zostaniemy przekierowani na stronę informującą o tym, że mapa została wygenerowana pomyślnie i możemy wrócić do menu.

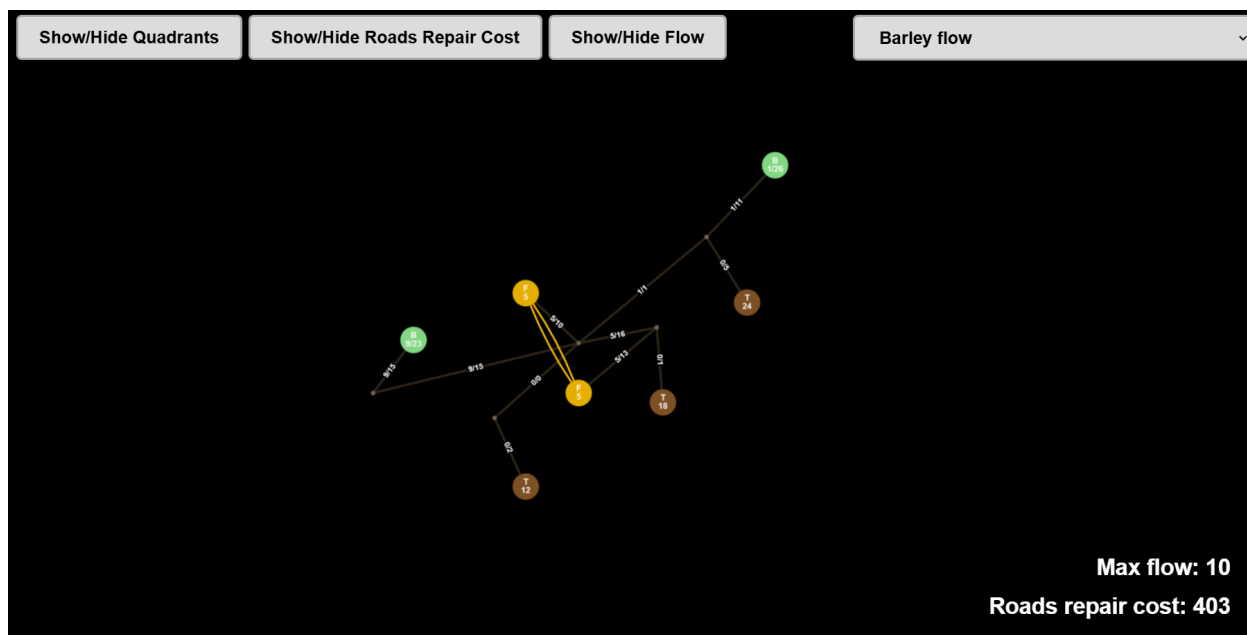


9.2.2 Wyświetlenie obliczeń przepływów i ćwiartek

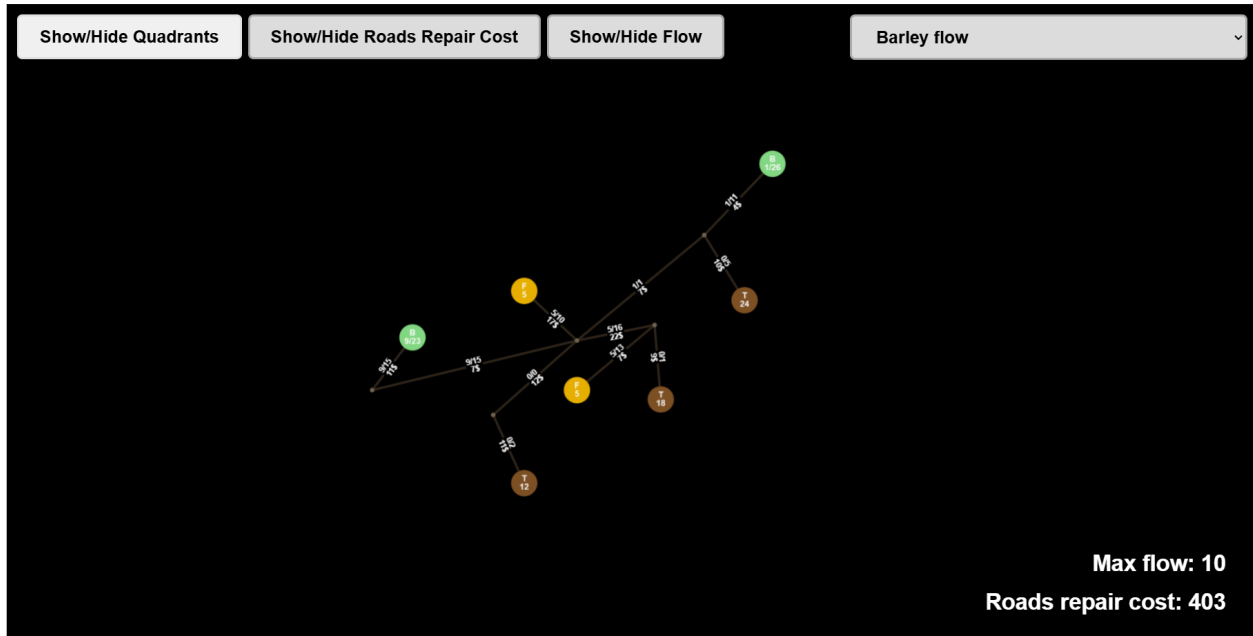
Po wybraniu opcji Calculate flows otwiera się strona z wygenerowaną mapą. Jeżeli na samym początku nie wprowadzimy żadnych danych mapa zostanie wygenerowana na podstawie domyślnych wartości na sztywno napisanych w kodzie dla demonstracji przykładowego działania programu.



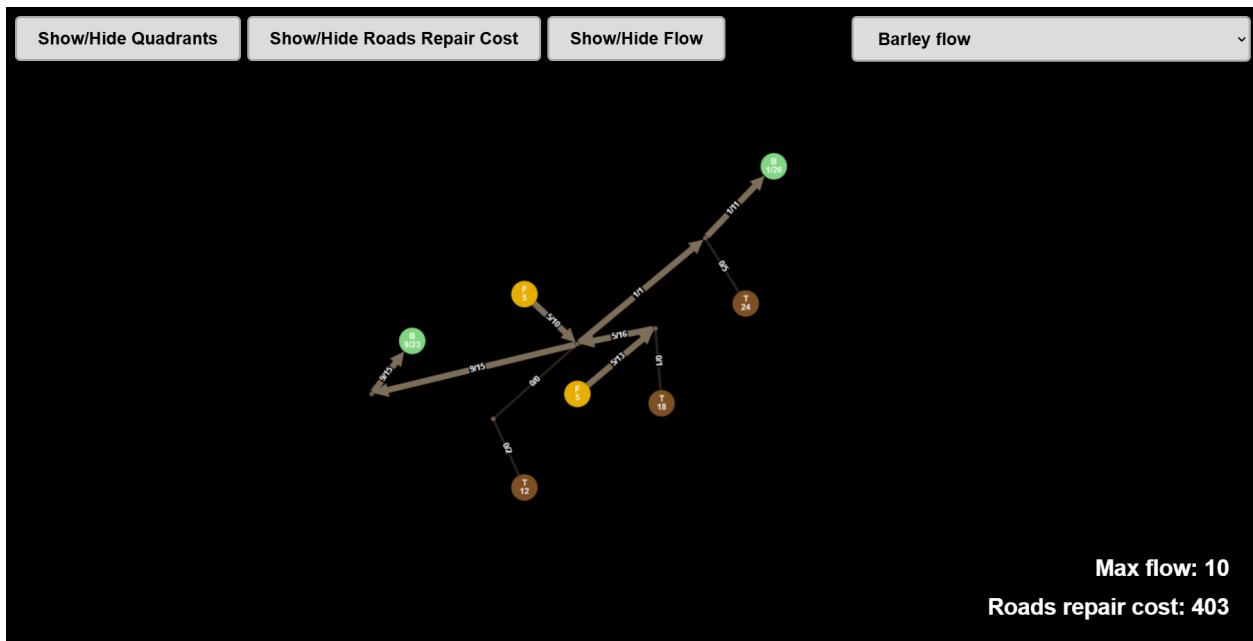
Na tej stronie użytkownik może zaznaczyć różne opcje i w ten sposób pokazać lub schować ćwiartki:



Pokazać lub schować koszty naprawy dróg:



Pokazać lub schować przepływ:



W prawym górnym rogu użytkownik może wybrać jaki przepływ musi zostać wyświetlony:

1. Przepływ jęczmienia
2. Przepływ jęczmienia wraz z naprawą dróg
3. Przepływ piwa
4. Przepływ piwa wraz z naprawą dróg

Natomiast w prawym dolnym rogu zostają wyświetlone dane obliczeń maksymalnego przepływu i kosztu naprawy dróg.

Ten widok jest obsługiwany w kontrolerze `public String sent(@ModelAttribute UserChoice userChoice, Model model)`.

9.2.3 Wyświetlanie logów

Po wybraniu opcji Show logs otwiera się strona z logami aplikacji.

Saved logs:

```
[OK] Server started
[OK] Data is generated without errors
[OK] Data is saved to JSON file without errors
[OK] Data is generated without errors
[OK] Data is saved to JSON file without errors
[OK] Quadrants: 1 were created
Quadrant 1
Production per plot: 5
Hull points:
(821.0, 502.0)
(804.0, 470.0)
Farmlands in this quadrant:
(821.0, 502.0)
(804.0, 470.0)
[OK] The network was created without errors
[OK] Source vertex for farmlands was created
[OK] Sink vertex for breweries was created
[BARLEY FLOW BEFORE DAMAGING] 10
[OK] Source vertex for farmlands was deleted
[OK] Sink vertex for breweries was deleted
[OK] Source vertex for breweries was created
[OK] Sink vertex for taverns was created
[BEER FLOW BEFORE DAMAGING] 8
[OK] Source vertex for breweries was deleted
[OK] Sink vertex for taverns was deleted
[OK] Source vertex for farmlands was created
[OK] Sink vertex for breweries was created
[BARLEY FLOW AFTER DAMAGING] 10
[OK] Source vertex for farmlands was deleted
[OK] Sink vertex for breweries was deleted
[OK] Source vertex for breweries was created
[OK] Sink vertex for taverns was created
[BEER FLOW AFTER DAMAGING] 8
[OK] Source vertex for breweries was deleted
[OK] Sink vertex for taverns was deleted
```

[Get back to menu](#)

W logach się znajduje informacja o generowaniu mapy, dodawaniu/usuwaniu wierzchołków źródłowych, obliczeniach maksymalnych przepływów i innych etapach działania programu.

Ten widok jest obsługiwany w kontrolerze `public String sent(@ModelAttribute UserChoice userChoice, Model model)`.

9.3.3 Wyszukiwanie słów

Po wybraniu opcji Search words in text otwiera się strona z możliwością wyboru typu wyszukiwania. Zostały zaimplementowane trzy możliwości:

1. Wyszukiwanie we własnym ręcznie wprowadzonym tekście (opcja Provide text). W pustym polu możemy wpisać swój tekst, potem słowo do wyszukiwania i wybrać z którego algorytmu chcemy skorzystać.

Enter your text and a word you want to find

Text to search in:

Word to find:

Choose an algorithm:

[Knuth-Morris-Pratt algorithm](#)

[Boyer-Moore algorithm](#)

[Both](#)

Po wybraniu otwiera się strona z wynikami wyszukiwania:

Search Completed

In the text:
kot ma piwo

[KMP and BM]

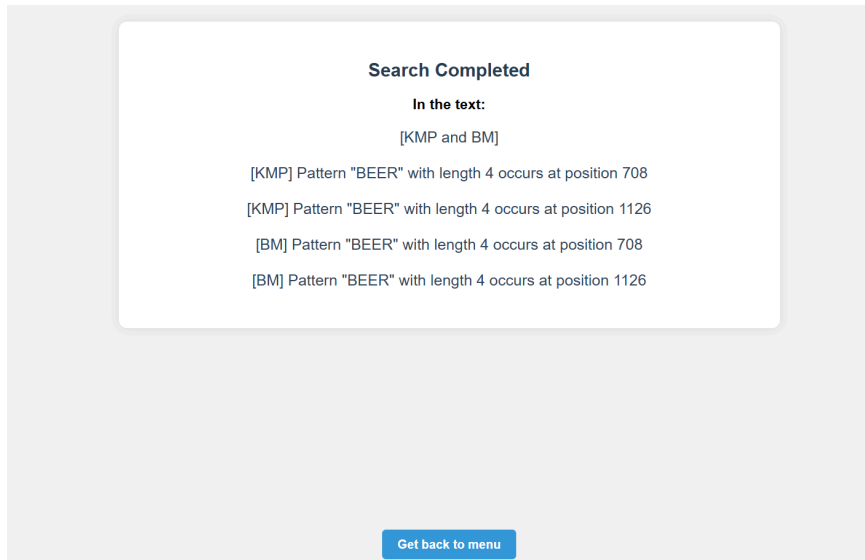
[KMP] Pattern "kot" with length 3 occurs at position 0

[BM] Pattern "kot" with length 3 occurs at position 0

[Get back to menu](#)

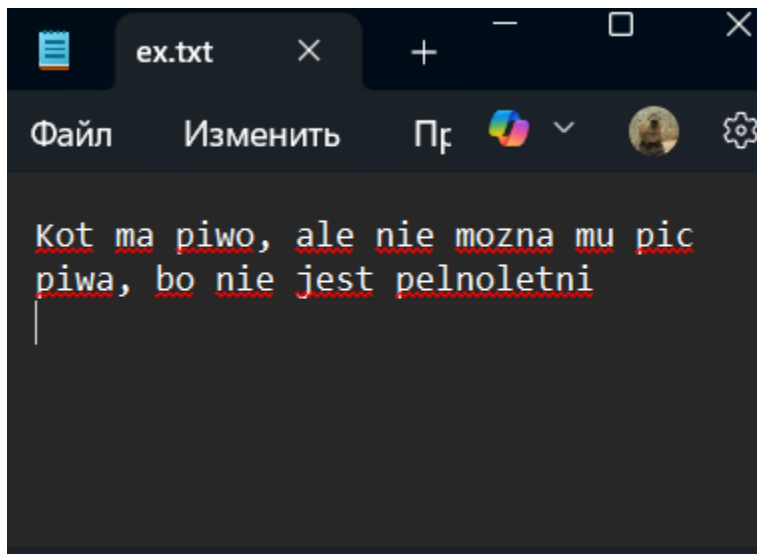
2. Wyszukiwanie słów w logach (opcja Search words in logs). W pustym polu musimy podać słowo do wyszukiwania i wybrać algorytm

Po wybraniu otwiera się strona z wynikami wyszukiwania:



3. Wyszukiwanie słów w pliku (opcja Search words in your own file). Tu możemy dołączyć własny plik i podać słowo do wyszukiwania.

Plik:



Enter your text and a word you want to find

Word to find:

Upload a file:

 ex.txt

Choose an algorithm:

Knuth-Morris-Pratt algorithm

Boyer-Moore algorithm

Both

Search Completed

In the text:

Kot ma piwo, ale nie mozna mu pic piwa, bo nie jest pelnoletni

[KMP and BM]

[KMP] Pattern does not occur

[BM] Pattern does not occur

[Get back to menu](#)