Dokumentacja Projektu

Zespół projektowy nr. 5

12 czerwca 2025

Spis treści

1 Skład zespołu i role poszczególnych osób

Imię i nazwisko	Rola w projekcie	Zakres obowiązków
Kacper Pawlicki	Koordynator projektu i Programista Frontend	Planowanie pracy zespołu, podział zadań, implementacja interfejsu użytkownika, integracja z backendem
Paweł Witkowski	Programista backend	Implementacja algorytmów oraz struktur danych
Marcin Adamczyk	Programista backend	Implementacja algorytmów oraz struktur danych
Łukasz Szmołda	Dokumentacja, Testy	Przygotowanie dokumentacji technicznej, testowanie funkcjonalności, zgłaszanie błędów

2 Tytuł projektu

BeerLand – aplikacja do rozwiązania problemu projektowego

Celem projektu jest stworzenie aplikacji, która w sposób funkcjonalny i nowoczesny rozwiązuje postawiony problem, głównie przy użyciu odpowiednich i dobrze zaimplementowanych algorytmów. Aplikacja została zaprojektowana na potrzeby realizacji zadania akademickiego i ma na celu symulację rzeczywistego procesu tworzenia oprogramowania – od analizy wymagań, przez projektowanie, aż po implementację i testowanie.

3 Specyfikacja problemu

Celem projektu jest stworzenie rozwiązania informatycznego, które umożliwi efektywne zarządzanie transportem jęczmienia oraz produkcją i dystrybucją piwa w fikcyjnym kraju Shire. Problem został przedstawiony jako symulacja rzeczywistego zagadnienia logistycznego, w którym konieczne jest zapewnienie maksymalnej ilości piwa dostarczanej do lokalnych karczm przy uwzględnieniu ograniczeń związanych z infrastrukturą transportową oraz wydajnością pól i browarów.

Dane wejściowe

- Położenie i wydajność pól jęczmienia (ilość zboża dostępna na każdym poletku).
- Lokalizacje browarów wraz z ich maksymalną przepustowością produkcyjną.
- Lokalizacje karczm będących punktami docelowymi dla transportu piwa.
- Lokalizacje skrzyżowań
- Sieć dróg reprezentująca możliwości transportu zboża i piwa pomiędzy skrzyżowaniami (wraz z informacją o przepustowości i ewentualnym koszcie naprawy).

Opis problemu

Problem można sformalizować jako przepływ w sieci, w której:

- Pola są źródłami surowca (jęczmień),
- Browary są ujściami przepływu jęczmienia oraz źródłem dla przepływu piwa o ograniczonej przepustowości (produkują piwo z jęczmienia),
- Karczmy są punktami końcowymi (odbiorcami piwa),
- Drogi reprezentują możliwe przepływy (z ograniczoną przepustowością i opcjonalnym kosztem naprawy).
- Skrzyżowania są wierzchołkami, które nie są ani źródłami w sieci.

Zadania

1. Wyznaczenie maksymalnego przepływu piwa z pól do karczm przy zachowaniu wszystkich ograniczeń.
2. Minimalizacja kosztu naprawy dróg przy zachowaniu maksymalnego przepływu (problem minimum-cost maximum-flow).
3. Uwzględnienie regionalnych różnic w wydajności upraw – każda ćwiartka Shire ma inną efektywność.
4. Implementacja efektywnego mechanizmu wyszukiwania słów kluczowych (np. *„piwo”, „jęczmień”, „browar”*).

Wymagania wobec rozwiązania

Ostateczne rozwiązanie powinno uwzględniać:

- Dobór odpowiednich struktur danych.
- Zastosowanie klasycznych algorytmów grafowych.
- Teoretyczny opis rozwiązania wraz z analizą złożoności obliczeniowej i pamięciowej.
- Implementację w wybranym języku programowania.
- Testy działania aplikacji na różnych zestawach danych (również losowych i skrajnych przypadkach).

Projekt pełni rolę ćwiczenia inżynierskiego, łącząc elementy algorytmiki, modelowania danych, optymalizacji oraz przetwarzania tekstu, umożliwiając zespołowi studentów kompleksową realizację procesu tworzenia rozwiązania programistycznego.

4 Podział problemu na podproblemy

Projekt BeerLand został podzielony na szereg mniejszych zadań (podproblemów), które razem umożliwiają kompleksowe rozwiązanie postawionego problemu. Każdy z podproblemów wymaga odpowiednich kompetencji w zakresie algorytmiki, programowania oraz analizy danych.

1. **Modelowanie danych wejściowych**
   - Reprezentacja pól, browarów, karczm, dróg i skrzyżowań jako elementów grafu. Gdzie pola, browary, karczmy i skrzyżowania są wierzchołkami, a drogi są krawędziami.
   - Wczytywanie i walidacja danych wejściowych.
2. **Wyznaczenie maksymalnego przepływu z pól do karczm**
   - Implementacja algorytmu maksymalnego przepływu.
   - Uwzględnienie etapów: pole $\rightarrow$ browar $\rightarrow$ karczma.
3. **Minimalizacja kosztu naprawy dróg przy zachowaniu maksymalnego przepływu**
   - Zastosowanie zmodyfikowanego algorytmu min-cost max-flow.
   - Dodanie kosztów do krawędzi grafu.
4. **Uwzględnienie regionalnych różnic w wydajności upraw**
   - Przypisanie pól do odpowiednich ćwiartek mapy.
   - Przeliczanie produkcji jęczmienia na podstawie regionu (wielokąty wypukłe).
5. **Wyszukiwanie wzorców testowych w poprzednich rozwiązaniach**
   - Implementacja struktur danych do wyszukiwania słów.
   - Zaimplementowanie różnych algorytmów do tego zadania.
6. **Zapisywanie skompresowanych rozwiązań do pliku**
   - Dopisywanie kolejnych rozwiązań do wspólnego skompresowanego pliku.
7. **Front-end aplikacji**
   - Budowa intuicyjnego i estetycznego GUI obsługującego wszystkie funkcjonalności aplikacji.
8. **Testowanie i walidacja rozwiązania**
   - Tworzenie danych testowych, w tym przypadków brzegowych.
   - Pomiar czasu działania i zużycia pamięci.
9. **Dokumentacja i raport końcowy**
   - Opis teoretyczny algorytmów oraz analiza złożoności.
   - Instrukcja obsługi aplikacji oraz omówienie wyników.

5 Harmonogram realizacji projektu

Tabela 2: Harmonogram zadań zespołu projektowego — uporządkowany chronologicznie

Zadanie	**Data rozpoczęcia**	**Czas trwania (tyg.)**	**Data zakończenia**	**Osoba odpowiedzialna**
Reprezentacja danych	07.04	1	13.04	Paweł
Algorytm przepływu bez kosztu napraw	07.04	1	13.04	Marcin
Tworzenie menu głównego programu, warstwa graficzna	07.04	1	13.04	Kacper
Wstępny zarys dokumentacji	07.04	4	04.05	Łukasz
Zrobienie testów (szkic)	13.04	3	04.05	Łukasz
Rozbudowa menu o okna do wczytywania danych	13.04	2	27.04	Kacper
Algorytm przepływu z kosztem napraw	13.04	2	27.04	Marcin
Parsowanie plików tekstowych	13.04	1	20.04	Paweł
Możliwość zapisywania danych do pliku tekstowego	20.04	1	27.04	Paweł
Uporządkowanie kodu	28.04	1	04.05	Marcin
Poprawienie logiki wczytywania danych	28.04	1	04.05	Kacper
Przygotowanie algorytmów geometrycznych	28.04	1	04.05	Paweł
Rozbudowa testów i rozwój dokumentacji	05.05	2	18.05	Łukasz
Wizualizacja wyników algorytmów	05.05	2	18.05	Kacper
Uporządkowanie kodu	05.05	1	11.05	Paweł
Implementacja algorytmu Boyera–Moore'a	05.05	1	11.05	Marcin
Dodanie możliwości zapisywania i wczytywania przeprowadzonych symulacji	12.05	2	25.05	Kacper
Implementacja algorytmu Huffmana oraz algorytmu KMP	12.05	1	18.05	Marcin
Implementacja logiki dzielenia płaszczyzny na ćwiartki	12.05	1	18.05	Paweł
Implementacja algorytmu Rabina–Karpa	12.05	1	18.05	Kacper
Refaktoryzacja pod kątem wielowątkowości	19.05	1	25.05	Paweł
Wstępny projekt prezentacji	19.05	1	25.05	Marcin
Przygotowanie finalnej wersji dokumentacji i rozbudowa testów klas implementujących algorytmy	19.05	1	25.05	Łukasz
Refaktoryzacja pod kątem integracji GUI i logiki aplikacji	19.05	1	25.05	Kacper

6 Ogólny opis przebiegu prac nad projektem

Główne prace nad projektem rozpoczęły się w dniu 7 kwietnia i trwały do 25 maja. Projekt został podzielony na szereg zadań, które realizowane były etapami przez członków zespołu.

Przebieg prac przedstawia się następująco:

- W początkowej fazie (07.04–13.04) skupiono się na przygotowaniu podstawowych komponentów programu, takich jak reprezentacja danych (Paweł), algorytm przepływu bez kosztu napraw (Marcin), graficzne menu główne aplikacji (Kacper) oraz wstępny zarys dokumentacji (Łukasz).
- W kolejnym etapie (13.04–27.04) kontynuowano rozwój funkcjonalności – wprowadzono możliwość parsowania plików tekstowych (Paweł), rozbudowano menu o możliwość wczytywania danych (Kacper) oraz zaimplementowano algorytm przepływu z kosztami napraw (Marcin). Dodatkowo, rozpoczęto prace nad testami (Łukasz).
- W trzecim etapie (20.04–04.05) rozwijano możliwość zapisywania danych do pliku (Paweł), uporządkowano kod (Marcin), poprawiono logikę wczytywania danych (Kacper) oraz przygotowano algorytmy geometryczne (Paweł). Równolegle kontynuowano prace nad dokumentacją oraz testami (Łukasz).
- W dniach 05.05–18.05 wdrażano bardziej zaawansowane funkcje, w tym wizualizację wyników algorytmów (Kacper), implementację algorytmów przeszukiwania tekstu: Boyera-Moore'a, Huffmana, KMP oraz Rabina-Karpa (Marcin, Kacper), a także algorytm dzielenia płaszczyzny na ćwiartki (Paweł). Kod był również optymalizowany i porządkowany.
- W ostatnim etapie projektu (19.05–25.05) przeprowadzono refaktoryzację pod kątem wielowątkowości (Paweł) oraz integracji logiki i interfejsu graficznego (Kacper). Powstał również wstępny projekt prezentacji (Marcin), a także finalna wersja dokumentacji wraz z dodatkowymi testami klas implementujących algorytmy (Łukasz).

Każdy z członków zespołu miał przypisane konkretne zadania, a prace były realizowane równolegle w sposób zorganizowany. Dzięki wyraźnemu podziałowi obowiązków i dobrej współpracy projekt został zrealizowany w zaplanowanym terminie.

7 Zastosowane algorytmy – opis, złożoność i poprawność

7.1 Algorytm Edmondsa-Karpa do wyznaczania maksymalnego przepływu w sieci

7.1.1 Opis problemu

Algorytm Edmondsa-Karpa to ulepszona wersja algorytmu Forda-Fulkersona, która do znajdowania ścieżek powiększających wykorzystuje przeszukiwanie wszerz (BFS). Celem algorytmu jest obliczenie maksymalnego przepływu od źródła (`source`) do ujścia (`sink`) w skierowanym grafie z przypisanymi przepustowościami (ang. `capacity`) dla każdej krawędzi.

7.1.2 Argument poprawności

Algorytm Edmondsa-Karpa opiera się na założeniu, że każdy znaleziony przepływ powiększający zwiększa całkowity przepływ w grafie rezydualnym. Wykorzystanie BFS zapewnia,

że w każdej iteracji wybierana jest najkrótsza możliwa ścieżka (pod względem liczby krawędzi), co ogranicza liczbę iteracji. Zgodnie z twierdzeniem o maksymalnym przepływie i minimalnym przekroju, algorytm kończy działanie wtedy, gdy nie istnieje już żadna ścieżka powiększająca – oznacza to osiągnięcie maksimum. Ponieważ przepływy są aktualizowane zgodnie z zasadami zachowania przepływu i pojemności krawędzi, wynik końcowy jest poprawny.

7.1.3 Własności wejścia i wyjścia

- **Wejście:**
  - Macierz sąsiedztwa `capacity[n][n]` reprezentująca przepustowości między wierzchołkami.
  - Indeks `source` — wierzchołek źródłowy.
  - Indeks `sink` — wierzchołek docelowy (ujście).
- **Wyjście:**
  - Liczba całkowita oznaczająca maksymalny możliwy przepływ z `source` do `sink`.

7.1.4 Opis działania

Algorytm wykonuje następujące kroki:

1. Inicjalizuje macierz przepływu `flow` na same zera.
2. Dopóki istnieje ścieżka powiększająca z `source` do `sink` w grafie rezydualnym (znaleziona za pomocą BFS):
   - Wyznacza minimalną przepustowość na tej ścieżce — tzw. *bottleneck capacity*.
   - Aktualizuje przepływy wzdłuż ścieżki — dodając przepływ na krawędziach w kierunku i odejmując w przeciwnym (dla grafu rezydualnego).
   - Sumuje przepływ do całkowitego przepływu `maxFlow`.
3. Gdy nie da się znaleźć kolejnej ścieżki powiększającej — algorytm kończy działanie.

7.1.5 Wzory i zależności

- Warunek ścieżki powiększającej:

$$\text{capacity}[u][v] - \text{flow}[u][v] > 0$$

- Aktualizacja przepływu:

$$\text{flow}[u][v] += \text{pathFlow}, \quad \text{flow}[v][u] -= \text{pathFlow}$$

7.1.6 Złożoność obliczeniowa

- **Złożoność czasowa:**
  - BFS wykonuje się w czasie $O(V + E)$, gdzie E to liczba krawędzi.
  - Maksymalna liczba iteracji BFS to $O(V \cdot E)$ (każda ścieżka powiększająca zwiększa odległość od źródła do ujścia w grafie rezydualnym).
  - Całkowita złożoność: $\mathcal{O}(V \cdot E^2)$.
- **Złożoność pamięciowa:** $\mathcal{O}(V^2)$ na przechowanie macierzy przepustowości i przepływu.

7.1.7 Zastosowania i uwagi

- Algorytm jest prosty w implementacji i dobrze działa dla grafów o umiarkowanym rozmiarze.
- Stosowany w problemach optymalizacyjnych, np. przydzielaniu zasobów, planowaniu produkcji, optymalizacji sieci transportowych, rozdzielaniu zadań itp.
- Ze względu na użycie BFS, algorytm znajduje ścieżki o najmniejszej liczbie krawędzi, co wpływa na lepszą kontrolę nad wydajnością.

7.2 Zmodyfikowany algorytm Busackera-Gowena do maksymalnego przepływu z kosztami aktywacji krawędzi

7.2.1 Opis problemu

Algorytm oblicza maksymalny przepływ w skierowanej sieci przepływowej z dodatkowym ograniczeniem w postaci kosztów aktywacji krawędzi. Każda krawędź może zostać aktywowana (jednorazowo), co umożliwia jej użycie w przepływie. Koszt aktywacji może być różny dla każdej krawędzi. Algorytm obsługuje wiele źródeł i ujść oraz ograniczenia pojemności źródeł.

7.2.2 Argument poprawności

Poprawność zmodyfikowanego algorytmu wynika z faktu, że każdy przepływ jest realizowany tylko po aktywowanych krawędziach, a aktywacja odbywa się zgodnie z minimalnym możliwym kosztem. Dzięki zastosowaniu algorytmu znajdowania ścieżki o najmniejszym koszcie (Dijkstra), przy każdej iteracji wybierana jest najtańsza możliwa aktywacja ścieżki z dostępnych. Ograniczenia źródeł i koszt aktywacji są respektowane na każdym etapie, a przepływ nie przekracza zadanych pojemności. W związku z tym, algorytm zawsze kończy działanie i zwraca maksymalny możliwy przepływ, natomiast koszt aktywacji w pewnych pesymistycznych przypadkach może nie być optymalny.

7.2.3 Własności wejścia i wyjścia

- **Wejście:**
  - Macierz całkowita `capacity`$[n][n]$ — pojemności krawędzi w grafie.
  - Macierz całkowita `activationCost`$[n][n]$ — koszty aktywacji krawędzi.

- Lista wierzchołków `sources` — zbiór wierzchołków źródłowych.
- Lista wierzchołków `sinks` — zbiór wierzchołków ujściowych.
- Mapa `sourceCapacities` — ograniczenia przepustowości dla źródeł.

- **Wyjście:**
  - Obiekt `FlowResult` zawierający:
    * `maxFlow` — wartość maksymalnego przepływu.
    * `totalActivationCost` — całkowity koszt aktywacji użytych krawędzi.
    * `flow`$[n][n]$ — macierz przepływów w sieci.
    * `activated`$[n][n]$ — macierz logiczna wskazująca, które krawędzie zostały aktywowane.

7.2.4 Opis działania

1. Tworzony jest rozszerzony graf poprzez dodanie superźródła i superujścia.
2. Z superźródła do każdego źródła w grafie właściwym prowadzi krawędź o przepustowości równej zadanej w `sourceCapacities` (lub nieskończoność domyślnie).
3. Każde ujście w grafie łączy się z superujściem krawędzią o nieskończonej przepustowości.
4. Właściwy algorytm znajduje ścieżki od źródła do ujścia o minimalnym koszcie aktywacji krawędzi (dostosowana wersja algorytmu Dijkstry).
5. Każda nieaktywna, użyta krawędź zostaje aktywowana i jej koszt aktywacji dodany do całkowitego kosztu.
6. Przepływ na ścieżce jest zwiększany, dopóki istnieją ścieżki z dodatnim przepustem.

7.2.5 Złożoność obliczeniowa

- **Złożoność czasowa:**
  - Każde przebiegnięcie algorytmu Dijkstry: $O(n^2)$ (dla macierzy sąsiedztwa).
  - Liczba iteracji: do $O($`maxFlow`$)$.
  - Łączna złożoność: w najgorszym przypadku $O(n^2 \cdot$ `maxFlow`$)$.

- **Złożoność pamięciowa:**
  - $O(n^2)$ — macierze: przepustowości, kosztów aktywacji, przepływów i statusów aktywacji.

7.2.6 Zastosowania i uwagi

- Modelowanie sieci infrastrukturalnych, w których uruchomienie połączenia generuje koszt (np. sieci wodociągowe, energetyczne).
- Algorytm przydatny w optymalizacji kosztów inicjalizacji zasobów w systemach rozproszonych.
- Możliwe rozszerzenie na grafy z wagami ujemnymi przez zastosowanie algorytmu Bellmana-Forda zamiast Dijkstry.
- Koszt aktywacji jest brany pod uwagę tylko raz dla każdej krawędzi — dalsze przepływy tą krawędzią są już darmowe.

7.3 Algorytm wyznaczania otoczki wypukłej (algorytm Grahama)

7.3.1 Opis problemu

Algorytm służy do wyznaczenia otoczki wypukłej dla zbioru punktów na płaszczyźnie kartezjańskiej. Otoczka wypukła to najmniejszy wypukły wielokąt zawierający wszystkie punkty zbioru. Algorytm Grahama realizuje to zadanie poprzez sortowanie punktów względem kąta nachylenia względem punktu referencyjnego oraz konstrukcję otoczki przy pomocy stosu, eliminując punkty wewnętrzne.

7.3.2 Argument poprawności

Algorytm Grahama jest dobrze znanym i udowodnionym algorytmem konstrukcji otoczki wypukłej. Sortowanie punktów według kąta zapewnia ich ułożenie w porządku rosnącym względem p_0. Usuwanie punktów współliniowych gwarantuje, że otoczka nie zawiera punktów „wewnętrznych". Kontrola orientacji podczas budowy stosu zapewnia, że zachowany jest wypukły charakter powstającego wielokąta. Końcowy wynik jest minimalnym wypukłym obszarem zawierającym wszystkie punkty.

7.3.3 Własności wejścia i wyjścia

- **Wejście:**
  - Lista punktów (wierzchołków) na płaszczyźnie, reprezentowanych jako obiekty klasy `Point` lub ich odpowiedniki (np. `Node` z pozycją).
  - Liczba punktów musi być co najmniej równa 3.
- **Wyjście:**
  - Lista punktów tworzących otoczkę wypukłą w kolejności przeciwnej do ruchu wskazówek zegara.

7.3.4 Szczegółowy opis implementacji

1. Wybór punktu początkowego (p_0) Wybierany jest punkt o najmniejszej współrzędnej y, a w przypadku remisu — o najmniejszej współrzędnej x. Punkt ten stanowi punkt odniesienia do sortowania pozostałych punktów względem kąta względem p_0.

2. Sortowanie punktów według kąta Pozostałe punkty są sortowane rosnąco względem kąta tworzonego z osią x i punktem p_0. W przypadku współliniowości (równych kątów), pozostawiany jest jedynie punkt najbardziej oddalony od p_0 (pozostałe są usuwane).

3. Usuwanie punktów współliniowych Po sortowaniu usuwane są wszystkie punkty pośrednie, które leżą na tej samej linii co p_0 i kolejny punkt z listy, pozostawiając tylko punkt najdalej oddalony od p_0.

4. Konstrukcja otoczki na stosie Stos jest inicjalizowany trzema pierwszymi punktami z posortowanej listy. Następnie kolejno dla każdego punktu sprawdzany jest obrót tworzony przez dwa ostatnie punkty na stosie i aktualny punkt:

- Jeśli obrót jest w prawo lub punkty są współliniowe (orientacja $\neq 1$), ostatni punkt jest zdejmowany ze stosu.
- Dopiero gdy obrót jest w lewo (orientacja $= 1$), punkt jest dodawany na stos.

W ten sposób eliminowane są punkty wnętrza otoczki.

7.3.5 Definicje funkcji pomocniczych

- **Orientacja** (p, q, r) — wyznacza kierunek obrotu trzech punktów:

$$\text{orientacja} = \text{sign}\big((q_x - p_x)(r_y - p_y) - (r_x - p_x)(q_y - p_y)\big)$$

  gdzie zwracana wartość:

  - 1 — obrót w lewo,
  - -1 — obrót w prawo,
  - 0 — punkty współliniowe.

- **Odległość** kwadratowa między punktami p i q:

$$d^2 = (p_x - q_x)^2 + (p_y - q_y)^2$$

  stosowana do rozstrzygania współliniowości przy sortowaniu.

7.3.6 Testowanie i walidacja

- Testowanie na zbiorach punktów o różnej liczbie oraz rozmieszczeniu (losowe, regularne, współliniowe).
- Walidacja zachowania przy małych liczbach punktów (np. 3 punkty, 4 punkty w linii).
- Porównanie wyniku z innymi metodami wyznaczania otoczki (np. algorytm Jarvisa).
- Analiza zachowania dla przypadków krawędziowych: punkty współliniowe, punkty powtarzające się, punkty z ujemnymi współrzędnymi.

7.3.7 Złożoność

- **Złożoność czasowa:** Dominującym etapem jest sortowanie punktów względem kąta, co daje złożoność $O(n \log n)$, gdzie n to liczba punktów wejściowych. Budowa otoczki przy pomocy stosu jest liniowa, czyli $O(n)$.
- **Złożoność pamięciowa:** $O(n)$ na przechowanie listy punktów oraz stosu do budowy otoczki.

7.4 Algorytm naiwny wyszukiwania wzorca w tekście (Naive String Search)

7.4.1 Opis problemu

Algorytm naiwny służy do znalezienia wszystkich wystąpień danego wzorca (ciągu znaków) w tekście. Działa poprzez przeszukiwanie tekstu znak po znaku, porównując kolejno fragmenty tekstu z wzorcem, bez stosowania żadnych zaawansowanych struktur danych ani heurystyk.

7.4.2 Argument poprawności

Algorytm sprawdza wszystkie możliwe przesunięcia wzorca w tekście i porównuje znaki jeden do jednego. Ponieważ żadne potencjalne dopasowanie nie jest pomijane, a każde porównanie jest dokładne, wszystkie wystąpienia wzorca zostaną znalezione. Brak założeń o strukturze tekstu i wzorca sprawia, że algorytm działa poprawnie niezależnie od danych wejściowych.

7.4.3 Własności wejścia i wyjścia

- **Wejście:**
  - Ciąg znaków `pattern` — wzorzec, którego wystąpienia chcemy odnaleźć.
  - Ciąg znaków `text` — tekst, w którym szukamy wzorca.
- **Wyjście:**
  - Lista indeksów całkowitych, wskazujących na pozycje w tekście, od których zaczynają się wystąpienia wzorca.

7.4.4 Opis działania

Algorytm przegląda każdą możliwą pozycję w tekście, na której wzorzec może się zaczynać (od 0 do $n - m$, gdzie n to długość tekstu, a m długość wzorca). Dla każdej pozycji porównuje kolejne znaki wzorca z odpowiadającymi im znakami tekstu:

- Jeśli wszystkie znaki wzorca zgadzają się z fragmentem tekstu, indeks tej pozycji zostaje dodany do listy wystąpień.
- W przypadku napotkania różnicy, przesuwamy się do następnej pozycji w tekście.

7.4.5 Analiza złożoności

- **Złożoność czasowa:** W najgorszym przypadku algorytm może wykonać $O(n \cdot m)$ porównań, gdzie n to długość tekstu, a m długość wzorca. Najgorszy przypadek występuje np. dla tekstu i wzorca z powtarzającymi się znakami, np. tekst = "aaaaaa"i wzorzec = "aaaab".
- **Złożoność pamięciowa:** $O(k)$, gdzie k to liczba znalezionych wystąpień wzorca — przechowywane są indeksy w liście wyników.

7.4.6 Zastosowania i ograniczenia

- Algorytm jest prosty w implementacji i dobrze sprawdza się dla krótkich wzorców lub niewielkich tekstów.
- Nie jest efektywny dla długich tekstów i wzorców, gdzie bardziej zaawansowane algorytmy (np. KMP, Boyer-Moore) oferują lepszą wydajność.
- Nie wymaga dodatkowych struktur danych ani preprocessingu wzorca.

7.5 Algorytm Rabina-Karpa do wyszukiwania wzorca w tekście

7.5.1 Opis problemu

Algorytm Rabina-Karpa służy do znajdowania wszystkich wystąpień wzorca w tekście, wykorzystując technikę haszowania znaków ciągu znaków. W przeciwieństwie do podejścia naiwnego, algorytm korzysta z obliczeń wartości haszu dla podciągów tekstu, co pozwala efektywniej porównywać potencjalne dopasowania.

7.5.2 Argument poprawności

Poprawność algorytmu Rabina-Karpa opiera się na zasadzie, że identyczne ciągi znaków mają takie same wartości haszu. Nawet przy możliwych kolizjach, algorytm wykonuje porównanie znaków, jeśli wartości haszy się zgadzają. Dzięki temu fałszywe trafienia są eliminowane, a prawdziwe wystąpienia wzorca są zawsze odnajdywane. Mechanizm przesuwania okna pozwala na efektywne przeszukiwanie bez utraty poprawności.

7.5.3 Własności wejścia i wyjścia

- **Wejście:**
  - Ciąg znaków `pattern` — wzorzec, który chcemy odnaleźć.
  - Ciąg znaków `text` — tekst, w którym szukamy wzorca.
- **Wyjście:**
  - Lista indeksów całkowitych określających pozycje w tekście, gdzie wzorzec występuje.

7.5.4 Opis działania

Algorytm działa w oparciu o następujące kroki:

1. Oblicza wartość haszu wzorca `patternHash` i wartości haszu pierwszego podciągu tekstu o długości wzorca `textHash`, korzystając z funkcji haszującej opartej na bazie i modulo liczby pierwszej.

2. Przesuwa "okno"o długości wzorca wzdłuż tekstu, w każdej iteracji:

   - Jeśli wartości haszu wzorca i aktualnego podciągu tekstu się zgadzają, wykonuje dokładne porównanie znaków (aby uniknąć fałszywych trafień spowodowanych kolizją haszów).
   - Oblicza nową wartość haszu dla kolejnego podciągu tekstu z wykorzystaniem techniki przesuwania okna (rolling hash), usuwając wpływ najstarszego znaku i dodając wpływ nowego znaku.

7.5.5 Parametry

- $BASE$ — podstawa systemu.
- $PRIME$ — liczba pierwsza używana jako moduł w obliczeniach haszu, by ograniczyć wartość i zminimalizować kolizje.

7.5.6 Złożoność obliczeniowa

- **Złożoność czasowa:**
  - Obliczanie początkowych wartości haszu: $O(m)$.
  - Przesuwanie okna i aktualizacja haszu: $O(n-m+1)$, każda aktualizacja haszu odbywa się w czasie stałym $O(1)$.
  - W najgorszym przypadku (kolizje haszów) możliwe jest wykonanie dokładnego porównania znak po znaku dla każdego przesunięcia, co prowadzi do złożoności $O(m(n-m+1))$.
  - W praktyce algorytm działa średnio w czasie około $O(n+m)$.
- **Złożoność pamięciowa:** $O(k)$, gdzie k to liczba znalezionych dopasowań — potrzebne do przechowania wyników.

7.5.7 Zastosowania i uwagi

- Wartość $PRIME$ i $BASE$ powinny być dobrane tak, aby minimalizować kolizje haszów i uniknąć przepełnień.
- Pomimo średniej dobrej wydajności, algorytm może mieć złożoność kwadratową w najgorszym przypadku z powodu kolizji.

7.6 Algorytm Knuth-Morris-Pratt (KMP) do wyszukiwania wzorca w tekście

7.6.1 Opis problemu

Algorytm KMP rozwiązuje problem efektywnego wyszukiwania wzorca w tekście, eliminując konieczność powtarzania porównań znaków przy dopasowywaniu, poprzez wcześniejsze przetworzenie wzorca w postaci tablicy `lps` (longest prefix suffix).

7.6.2 Argument poprawności

Tablica `lps` przechowuje wiedzę o strukturze wzorca, co pozwala unikać zbędnych porównań i ponownego analizowania tych samych fragmentów. Dzięki temu algorytm nigdy nie przechodzi przez ten sam znak tekstu więcej niż raz. Ostatecznie każde potencjalne dopasowanie jest sprawdzane dokładnie, a dopasowania są poprawnie identyfikowane. Poprawność wynika z właściwości tablicy `lps`, która zachowuje spójność prefiksów i sufiksów wzorca.

7.6.3 Wejście i wyjście

- **Wejście:**
  - `pattern` — wzorzec do znalezienia (ciąg znaków).
  - `text` — tekst, w którym szukamy wzorca.
- **Wyjście:**
  - Lista indeksów pozycji w tekście, w których wzorzec zaczyna się (dopasowania).

7.6.4 Idea algorytmu

Algorytm KMP opiera się na preprocesowaniu wzorca, które umożliwia „przeskakiwanie" znaków wzorca podczas dopasowania w tekście.

- Tworzy tablicę `lps`, w której dla każdej pozycji wzorca przechowywana jest długość najdłuższego właściwego prefiksu, który jest jednocześnie sufiksem podciągu wzorca od początku do tej pozycji.
- Podczas dopasowywania w tekście, w przypadku niedopasowania, indeks wzorca jest przesuwany na podstawie wartości z tablicy `lps`, zamiast powrotu do początku wzorca.

7.6.5 Tablica LPS (Longest Prefix Suffix)

Tablica `lps` jest kluczowa dla optymalizacji:

$$lps[i] = \max\{k : k < i + 1, \quad pattern[0..k-1] = pattern[(i-k+1)..i]\}$$

Dzięki temu możemy ominąć ponowne sprawdzanie znaków, które już zostały dopasowane.

7.6.6 Złożoność obliczeniowa

- Preprocesowanie wzorca (obliczanie tablicy `lps`): $O(m)$, gdzie m to długość wzorca.
- Przeszukiwanie tekstu z wykorzystaniem tablicy `lps`: $O(n)$, gdzie n to długość tekstu.
- Całkowita złożoność: $O(n + m)$, co jest optymalne dla problemu wyszukiwania wzorca.

7.6.7 Praktyczne zastosowania

- Efektywne wyszukiwanie wzorców w dużych tekstach, np. przeszukiwanie dokumentów, edycja tekstu, analiza DNA.
- Podstawa wielu algorytmów w bioinformatyce, przetwarzaniu języka naturalnego i systemach informacyjnych.
- Wydajne dopasowanie wzorca bez konieczności powtarzania porównań, co jest kluczowe przy bardzo długich danych.

7.7 Algorytm Boyer-Moore do wyszukiwania wzorca w tekście

7.7.1 Opis problemu

Algorytm Boyer-Moore jest jednym z najszybszych algorytmów służących do wyszukiwania wzorca w tekście. Optymalizuje proces dopasowywania dzięki wykorzystaniu dwóch głównych heurystyk: reguły *złego znaku* oraz reguły *dobrego sufiksu*, które pozwalają na efektywne przesunięcia wzorca względem tekstu.

7.7.2 Argument poprawności

Poprawność algorytmu Boyera-Moore wynika z faktu, że każde przesunięcie wzorca obliczane jest na podstawie reguł, które nigdy nie pomijają potencjalnego dopasowania. Obie heurystyki (zły znak i dobry sufiks) są zaprojektowane tak, aby przesunięcia były bezpieczne – gwarantują, że żadne prawdziwe wystąpienie wzorca nie zostanie ominięte. Algorytm sprawdza wszystkie pozycje, które mogą zawierać wzorzec, więc każde dopasowanie jest wykrywane.

7.7.3 Wejście i wyjście

- **Wejście:**
  - `pattern` — wzorzec do znalezienia (ciąg znaków).
  - `text` — tekst, w którym wyszukujemy wzorzec.
- **Wyjście:**
  - Lista indeksów początkowych pozycji w tekście, na których wzorzec został znaleziony.

7.7.4 Idea algorytmu

Algorytm Boyer-Moore porównuje wzorzec z tekstem od prawej do lewej strony wzorca, co umożliwia zastosowanie dwóch heurystyk przesunięć:

1. **Reguła złego znaku (Bad Character Rule):** W przypadku niedopasowania znaku w tekście, wzorzec jest przesuwany tak, aby wyrównać go z ostatnim wystąpieniem tego znaku we wzorcu. Jeśli znak nie występuje w wzorcu, przesunięcie jest maksymalne.

2. **Reguła dobrego sufiksu (Good Suffix Rule):** W przypadku dopasowania pewnego sufiksu wzorca, ale niedopasowania dalej na lewo, wzorzec jest przesuwany tak, aby dopasować kolejny występujący w nim ten sam sufiks lub prefiks.

Przesunięcie wzorca jest maksymalnym z wartości wynikających z obu reguł.

7.7.5 Preprocesowanie wzorca

Tablica przesunięć złego znaku Tworzymy mapę $badCharShift$, która dla każdego znaku przechowuje jego ostatnią pozycję we wzorcu:

$$badCharShift[c] = \max\{i : pattern[i] = c\} \quad \text{lub} \quad -1, \quad \text{jeśli } c \notin pattern$$

Tablica przesunięć dobrego sufiksu Tablica $goodSuffixShift$ przechowuje przesunięcia wzorca dla każdej pozycji j, które umożliwiają dopasowanie znanego sufiksu wzorca do innego wystąpienia tego samego sufiksu lub do prefiksu wzorca.

Proces budowy tablicy obejmuje:

- Obliczenie tablicy długości sufiksów wzorca ($suffixLength$).
- Ustawienie przesunięć zgodnie z dopasowaniami sufiksów do prefiksów wzorca lub innym wystąpieniom sufiksów.

7.7.6 Opis kroków wyszukiwania

1. Porównujemy wzorzec z tekstem zaczynając od znaku na pozycji shift $+ m - 1$, gdzie m to długość wzorca.

2. Idziemy od prawej strony wzorca do lewej, porównując kolejne znaki.

3. Jeśli napotkamy niedopasowanie na pozycji j, wyliczamy przesunięcie wzorca:

$$shift \mathrel{+}= \max\left(j - badCharShift[text[shift + j]], \quad goodSuffixShift[j]\right)$$

4. Jeżeli wzorzec został w całości dopasowany (czyli $j < 0$), dodajemy aktualną pozycję shift do listy wyników i przesuwamy wzorzec zgodnie z regułą złego znaku.

7.7.7 Złożoność obliczeniowa

- Preprocesowanie wzorca (budowa tablic $badCharShift$ i $goodSuffixShift$): $O(m + |\Sigma|)$, gdzie $|\Sigma|$ to rozmiar alfabetu, a m to długość wzorca.
- Wyszukiwanie wzorca w tekście: średnio $O(n/m)$ w praktyce, w najgorszym przypadku $O(n + m)$, gdzie n to długość tekstu.

7.7.8 Zastosowania praktyczne

- Wyszukiwanie wzorców w dużych plikach tekstowych.
- Analiza sekwencji biologicznych (np. DNA).
- Implementacje w edytorach tekstu i systemach indeksujących.

7.8 Algorytm Huffmana do kompresji danych

7.8.1 Opis problemu

Algorytm Huffmana jest metodą optymalnego kodowania znaków na podstawie ich częstotliwości występowania, umożliwiającą bezstratną kompresję danych. Jego celem jest minimalizacja średniej długości kodu znaków poprzez przypisanie krótszych kodów tym znakom, które pojawiają się częściej, oraz dłuższych kodów znakom rzadziej występującym.

7.8.2 Argument poprawności

Poprawność algorytmu Huffmana opiera się na jego konstrukcji: drzewo Huffmana budowane jest tak, że znaki o większej częstotliwości znajdują się bliżej korzenia, co zapewnia krótsze kody. Ponieważ drzewo jest binarne i żaden kod nie jest prefiksem innego (właściwość prefiksowa), możliwe jest jednoznaczne dekodowanie każdego ciągu zakodowanego. Fakt, że konstrukcja drzewa zawsze łączy dwa najrzadsze elementy, gwarantuje minimalną długość zakodowanego ciągu w porównaniu z innymi kodowaniami jednostajnymi.

7.8.3 Wejście i wyjście

- **Wejście:**
  - Tekst (ciąg znaków) do kompresji.
- **Wyjście:**
  - Zakodowany ciąg bitów reprezentujący oryginalny tekst w formie skompresowanej.
  - Mapa kodów Huffmana: przypisanie każdemu znakowi unikalnego ciągu bitów (kodów).

7.8.4 Idea algorytmu

Algorytm Huffmana składa się z kilku kroków:

1. **Analiza częstotliwości:** Obliczenie liczby wystąpień każdego znaku w tekście (mapa częstotliwości).
2. **Budowa drzewa Huffmana:** Na podstawie częstotliwości tworzy się drzewo binarne, gdzie każdy liść reprezentuje znak, a ścieżka od korzenia do liścia odpowiada kodowi binarnemu znaku. Drzewo budowane jest iteracyjnie, łącząc dwa najmniej częste węzły w nowy węzeł nadrzędny, aż zostanie jeden korzeń.

3. **Generowanie kodów:** Przypisanie każdemu znakowi unikalnego kodu binarnego na podstawie ścieżki w drzewie (np. „0” dla przejścia do lewego dziecka, „1” dla prawego).

4. **Kodowanie tekstu:** Zamiana każdego znaku w tekście na odpowiadający mu kod binarny i połączenie ich w ciąg bitów.

7.8.5 Struktura danych i implementacja

Węzeł drzewa Huffmana Węzeł drzewa zawiera znak (tylko w liściach), częstotliwość oraz wskaźniki na lewego i prawego potomka. Węzły porównywane są względem częstotliwości, co umożliwia ich efektywne przechowywanie i łączenie w kolejce priorytetowej (PriorityQueue).

Mapa kodów Huffmana Mapa znaków na ciągi bitów (String) przechowuje wygenerowane kody dla szybkiego kodowania i dekodowania.

Kompresja Kompresja obejmuje:

- obliczenie częstotliwości (klasa `FrequencyAnalyzer`),
- zbudowanie drzewa Huffmana (klasa `HuffmanTree`),
- wygenerowanie kodów i zakodowanie tekstu,
- zapisanie mapy kodów i zakodowanego ciągu bitów do pliku (klasa `FileHandler`).

Dekompresja Proces odwrotny do kompresji:

- wczytanie mapy kodów i zakodowanego tekstu z pliku,
- odbudowanie drzewa Huffmana na podstawie mapy,
- dekodowanie ciągu bitów na oryginalny tekst.

7.8.6 Format zapisu skompresowanych danych

Plik wyjściowy zawiera serializowaną mapę kodów Huffmana oraz zakodowany tekst w formie bitów, zapakowanych w bajty wraz z informacją o liczbie bitów ważnych w ostatnim bajcie. Pozwala to na efektywne odczytanie i dekodowanie danych.

7.8.7 Złożoność obliczeniowa

- Obliczenie częstotliwości: $O(n)$, gdzie n to długość tekstu.
- Budowa drzewa Huffmana: $O(m \log m)$, gdzie m to liczba unikalnych znaków (rozmiar alfabetu).
- Generowanie kodów i kodowanie tekstu: $O(n)$ (przy szybkim dostępie do mapy kodów).
- Łączna złożoność: $O(n + m \log m)$, efektywna dla praktycznych zastosowań.

7.8.8 Praktyczne zastosowania

- Kompresja tekstu i danych binarnych bezstratnie, w narzędziach archiwizujących i systemach plików.
- Podstawa dla wielu nowoczesnych algorytmów kompresji, takich jak DEFLATE, wykorzystywanych np. w formatach ZIP, PNG.
- Optymalizacja przesyłu danych w systemach komunikacyjnych, gdzie ważna jest minimalizacja rozmiaru danych.

8 Testowanie algorytmów - testy jednostkowe i testy dla dużych danych

8.1 Cele testowania

Celem testowania zaimplementowanych algorytmów było zapewnienie ich poprawności, niezawodności i efektywności działania w różnych scenariuszach wejściowych. Proces testowania został zaprojektowany tak, aby objąć zarówno testy funkcjonalne, jak i testy wydajnościowe. Do głównych celów należało:

- **Weryfikacja poprawności działania** każdego z algorytmów poprzez testy jednostkowe, obejmujące przypadki brzegowe, dane syntetyczne oraz testy regresyjne. Testami objęto wszystkie algorytmy, lecz głównie skupiliśmy się na:
  - Algorytmie Edmondsa-Karpa (maksymalny przepływ),
  - Zmodyfikowanym algorytmie Busackera-Gowena z kosztami aktywacji krawędzi,
  - Algorytmie Grahama (otoczka wypukła).
- **Detekcja błędów implementacyjnych i logicznych**, które mogłyby prowadzić do niepoprawnych wyników, błędnych ścieżek w grafie, błędów w alokacji pamięci lub nieprzewidywalnych zachowań.
- **Ocena wydajności algorytmów** dla dużych instancji danych — m.in. testowanie przepływu w grafach zawierających tysiące wierzchołków i krawędzi, oraz sprawdzenie stabilności obliczeń geometrycznych dla dużych chmur punktów w algorytmie otoczki wypukłej.
- **Zbudowanie zaufania do implementacji** poprzez automatyzację testów i powtarzalność wyników w niezależnych środowiskach testowych.

Testowanie stanowiło integralną część procesu inżynierskiego i było przeprowadzane na każdym etapie implementacji, co pozwoliło na wczesne wykrycie oraz eliminację potencjalnych problemów.

8.2 Testy jednostkowe

Testy jednostkowe zostały przygotowane w celu weryfikacji poprawności działania poszczególnych komponentów algorytmicznych w projekcie. Testowano głównie logikę algorytmiczną oraz zachowanie metod w sytuacjach standardowych i brzegowych. Łącznie wykonano 60 testów jednostkowych obejmujących klasy implementujące algorytmy oraz modułu struktury mapy (ShireMap).

8.2.1 Cele testów jednostkowych

Celem testów jednostkowych było:

- weryfikacja poprawności implementacji algorytmów dla różnych przypadków wejściowych,
- potwierdzenie poprawności działania metod pomocniczych,
- testowanie zachowania komponentów w sytuacjach brzegowych,
- wykrycie potencjalnych błędów logicznych jeszcze przed testami wydajnościowymi.

8.2.2 Zakres testów

Testami objęto następujące komponenty:

- Implementacja algorytmu Edmondsa-Karpa dla maksymalnego przepływu.
- Zmodyfikowany algorytm Busackera-Gowena do maksymalnego przepływu z kosztami aktywacji krawędzi.
- Algorytm wyznaczania otoczki wypukłej metodą Jarvisa.

8.2.3 Podejście do testowania

Każdy z testowanych algorytmów został objęty zestawem testów sprawdzających:

- poprawność wyników dla prostych grafów / zbiorów punktów,
- zachowanie dla danych granicznych,
- spójność z oczekiwanym zachowaniem teoretycznym algorytmu.

8.2.4 Struktura testów

Testy jednostkowe zostały pogrupowane w klasy testowe odpowiadające poszczególnym modułom aplikacji. Dla każdej klasy testowej przedstawiono jej zakres oraz wybrane przypadki testowe, które ilustrują najważniejsze aspekty przetestowanej funkcjonalności.

ShireMapTest Testy jednostkowe klasy `ShireMap` weryfikują poprawność operacji na strukturze reprezentującej mapę miasteczka Shire. Sprawdzają m.in. dodawanie wierzchołków (pól, karczm i skrzyżowań), krawędzi (połączeń), operacje czyszczenia oraz zarządzanie identyfikatorami węzłów.

Przetestowane przypadki obejmują:

- `testIsEmptyInitially()` – sprawdzenie, czy nowo utworzona mapa jest pusta.
- `testAddNodeIncreasesNodeCount()` – dodanie węzła zwiększa licznik i umożliwia poprawny odczyt.
- `testAddEdgeIncreasesEdgeCount()` – dodanie poprawnej krawędzi rejestruje ją prawidłowo jako krawędź wychodzącą/przychodzącą.
- `testAddEdgeWithMissingNodeDoesNothing()` – zabezpieczenie przed dodaniem krawędzi do nieistniejącego wierzchołka.
- `testClearRemovesAllData()` – metoda `clear()` skutecznie usuwa wszystkie dane z mapy.
- `testFindMaxId()` – poprawne określenie maksymalnego identyfikatora w zbiorze wierzchołków.

ConvexHullTest Testy jednostkowe klasy `ConvexHull` mają na celu weryfikację poprawności algorytmu tworzenia otoczki wypukłej na podstawie zbioru punktów w przestrzeni dwuwymiarowej. W testach wykorzystano pomocniczą klasę implementującą interfejs `Node`, aby zapewnić odpowiednią strukturę danych.

Testy obejmują:

- inicjalizację algorytmu z zestawem 20 punktów o zróżnicowanych współrzędnych,
- generację otoczki wypukłej za pomocą metody `createConvexHull()`,
- weryfikację, że każdy punkt otoczki należy do oryginalnego zbioru punktów,
- sprawdzenie minimalnej liczby punktów na otoczce (w tym przypadku co najmniej 5),
- dodatkowo, dla celów debugowania, wypisanie punktów otoczki.

PlaneQuarterPartitionerTest Testy jednostkowe klasy `PlaneQuarterPartitioner` mają na celu weryfikację poprawności podziału zbioru punktów (węzłów) na ćwiartki płaszczyzny względem punktu środkowego oraz zarządzanie parametrami pól w każdej ćwiartce.

Przetestowane przypadki obejmują:

- `testFindCenterPoint_SimpleCase()` – weryfikację poprawności wyznaczania punktu środkowego metodą średniej arytmetycznej współrzędnych.
- `testAssignNodesToQuarters_EdgeAndBoundaryCases()` – sprawdzenie przypisania węzłów do odpowiednich ćwiartek zgodnie z pozycją względem punktu środkowego, z uwzględnieniem przypadków granicznych i pustych ćwiartek.

- `testSetBarleyAmountForQuarters_BoundaryAndInvalidInput()` – testowanie ustawiania ilości jęczmienia dla wszystkich pól w danej ćwiartce oraz weryfikację poprawnej propagacji tych wartości do pól, a także walidację poprawnego rzucania wyjątków dla nieprawidłowych indeksów ćwiartek.

EdmondsKarpTest Testy jednostkowe dla implementacji algorytmu Edmondsa-Karpa weryfikują poprawność wyznaczania maksymalnego przepływu w grafie skierowanym o pojemnościach na krawędziach.

Testowane przypadki obejmują:

- `testMaxFlowSimpleCase` – klasyczny przykład grafu z 6 wierzchołkami, dla którego spodziewany maksymalny przepływ wynosi 23,
- `testNoPath` – graf bez żadnej ścieżki między źródłem a ujściem, gdzie maksymalny przepływ powinien wynosić 0,
- `testZeroCapacity` – graf z pojedynczą krawędzią o ograniczonej pojemności, sprawdzający poprawność uwzględnienia pojemności krawędzi,
- `testDisconnectedGraph` – przypadek grafu niespójnego, w którym nie istnieje ścieżka od źródła do ujścia,
- `testMultiplePathsSameCapacity` – sytuacja z kilkoma ścieżkami o równej przepustowości sumarycznie zwiększającymi maksymalny przepływ,
- `testLoopEdgeIgnored` – test ignorowania krawędzi pętlowej (z wierzchołka do samego siebie), która nie powinna wpływać na wynik,
- `testBackEdgeScenario` – weryfikacja poprawnej obsługi krawędzi zwrotnych (back edges) w sieci przepływowej,
- `testBottleneck` – test uwzględniający wąskie gardło ograniczające maksymalny przepływ, mimo dużej przepustowości innych krawędzi.

MaxFlowWithActivationTest Testy jednostkowe implementacji algorytmu maksymalnego przepływu z kosztami aktywacji krawędzi mają na celu weryfikację poprawności wyznaczania maksymalnego przepływu wraz z minimalizacją łącznego kosztu aktywacji w skierowanym grafie przepływowym.

Testowane przypadki obejmują:

- `testSimpleGraph` – klasyczny graf 4-wierzchołkowy ze zdefiniowanymi pojemnościami i kosztami aktywacji, gdzie maksymalny przepływ wynosi 4, a minimalny koszt aktywacji to 7,
- `testNoPath` – graf bez dostępnej ścieżki ze źródła do ujścia, w którym maksymalny przepływ oraz koszt aktywacji wynoszą 0,
- `testSingleEdge` – pojedyncza krawędź o określonej pojemności i koszcie aktywacji, sprawdzająca poprawność obliczeń na najprostszym przypadku,
- `testEdgeAlreadyActivated` – symulacja ponownego uruchomienia algorytmu z aktywowanymi już krawędziami, które nie generują dodatkowych kosztów aktywacji,

FrequencyAnalyzerTest Test jednostkowy klasy `FrequencyAnalyzer` weryfikuje poprawność obliczania częstotliwości występowania znaków w ciągu znaków.

Testowane przypadki obejmują:

- `calculateFrequencyShouldReturnCorrectMap()` – analiza prostego ciągu znaków "aabbc", gdzie oczekiwane jest poprawne zliczenie wystąpień poszczególnych liter: 2 dla 'a', 2 dla 'b' oraz 1 dla 'c'.

HuffmanCodingTest Testy jednostkowe klasy `HuffmanCoding()` sprawdzają poprawność kompresji i dekompresji tekstu z użyciem algorytmu Huffmana.

Testowane przypadki obejmują:

- `compressAndDecompressShouldReturnOriginalText()` – weryfikuje, czy po skompresowaniu i następnie zdekompresowaniu tekstu `sampleText` (w tym przypadku "hello huffman") wynik jest identyczny z oryginałem, co potwierdza poprawność działania obu procesów.

Dodatkowo zastosowano metodę `cleanup()` oznaczoną adnotacją `@AfterEach`, która usuwa plik wynikowy po każdym teście, zapewniając izolację i czystość środowiska testowego.

HuffmanTreeTest Testy jednostkowe klasy `HuffmanTree()` weryfikują poprawność budowy drzewa Huffmana oraz kodowania i dekodowania tekstu.

Testowane przypadki obejmują:

- `encodeAndDecodeShouldReturnOriginalText()` – sprawdza, czy dla podanego ciągu znaków po zakodowaniu i następnie zdekodowaniu otrzymany tekst jest identyczny z oryginałem, co potwierdza poprawność kodowania i dekodowania zbudowanego na podstawie częstotliwości znaków,

- `rebuildTreeShouldReconstructCorrectly()` – weryfikuje, czy drzewo Huffmana może zostać poprawnie odtworzone na podstawie mapy kodów Huffmana, a zdekodowany na jej podstawie zakodowany ciąg znaków odpowiada oryginałowi, co dowodzi poprawnej rekonstrukcji i funkcjonowania drzewa.

BoyerMooreTest Testy jednostkowe klasy `BoyerMoore` sprawdzają poprawność implementacji algorytmu Boyera-Moore'a, służącego do efektywnego wyszukiwania wzorców w tekście.

Testowane przypadki obejmują:

- `testSearchSimpleMatch()` – wyszukiwanie pojedynczego wystąpienia wzorca w tekście,

- `testSearchMultipleMatches()` – wykrywanie wielu wystąpień wzorca w ciągu znaków,

- `testSearchNoMatch()` – sytuacja, gdy wzorzec nie występuje w tekście,

- `testSearchEmptyPattern()` – obsługa pustego wzorca, który powinien zwrócić pustą listę wyników,

- `testSearchPatternLongerThanText()` – przypadek, gdy wzorzec jest dłuższy niż tekst, co uniemożliwia znalezienie dopasowania,
- `testSearchPatternEqualsText()` – test dopasowania wzorca identycznego z całym tekstem,
- `testSearchHeavyInput()` – test wydajności i poprawności na dużym tekście z jednym ukrytym wystąpieniem wzorca.

KmpTest Testy jednostkowe klasy `Kmp` weryfikują poprawność implementacji algorytmu Knutha-Morrisa-Pratta do wyszukiwania wzorców w tekście.

Testowane przypadki obejmują:

- `testSimpleMatch()` – znalezienie pojedynczego wystąpienia wzorca w tekście,
- `testMultipleMatches()` – wykrywanie wielu nakładających się i rozłącznych wystąpień wzorca,
- `testNoMatch()` – sytuacja, gdy wzorzec nie występuje w tekście,
- `testPatternLongerThanText()` – wzorzec dłuższy niż tekst, co uniemożliwia dopasowanie,
- `testExactMatch()` – wzorzec identyczny z całym tekstem,
- `testEmptyPattern()` – obsługa pustego wzorca, który powinien zwrócić pustą listę wyników,
- `testEmptyText()` – wyszukiwanie w pustym tekście, co powinno skutkować brakiem dopasowań,
- `testHeavyInputWithLateMatch()` – test wydajności i poprawności na dużym tekście z jednym późnym wystąpieniem wzorca,
- `testSelfOverlappingPattern()` – test wzorca, który może się nakładać sam na siebie, co wymaga prawidłowego obsłużenia indeksów wystąpień.

NaiveStringSearchTest Testy jednostkowe klasy `NaiveStringSearch` sprawdzają poprawność implementacji prostego, naiwnym algorytmem wyszukiwania wzorca w tekście.

Testowane scenariusze obejmują:

- `testFindAllOccurrencesSimpleMatch()` – znalezienie pojedynczego, prostego dopasowania wzorca w tekście,
- `testFindAllOccurrencesMultipleMatches()` – wykrycie wielu, powtarzających się wystąpień wzorca,
- `testFindAllOccurrencesNoMatch()` – sytuację, gdy wzorzec nie występuje w tekście,
- `testFindAllOccurrencesEmptyPattern()` – obsługę pustego wzorca, który nie powinien zwracać żadnych dopasowań,

- `testFindAllOccurrencesPatternLongerThanText()` – przypadek, gdy wzorzec jest dłuższy niż tekst, skutkujący brakiem dopasowań,
- `testFindAllOccurrencesPatternEqualsText()` – sytuację, gdy wzorzec jest identyczny z całym tekstem,
- `testFindAllOccurrencesHeavyInput()` – test na dużym tekście z jednym późnym wystąpieniem wzorca, weryfikujący poprawność i wydajność algorytmu.

RabinKarpTest Testy jednostkowe implementacji algorytmu Rabina-Karpa służą do weryfikacji poprawności wyszukiwania wzorca w tekście z wykorzystaniem haszowania i efektywnego porównywania znaków.

Obejmują one następujące przypadki:

- `testSearchSimpleMatch()` – wykrycie pojedynczego, prostego dopasowania wzorca w tekście,
- `testSearchMultipleMatches()` – znalezienie wielu powtarzających się wystąpień wzorca,
- `testSearchNoMatch()` – przypadek, gdy wzorzec nie występuje w tekście,
- `testSearchEmptyPattern()` – obsługa sytuacji, gdy wzorzec jest pusty i nie powinien zwracać żadnych wyników,
- `testSearchPatternLongerThanText()` – scenariusz, w którym wzorzec jest dłuższy niż tekst, skutkujący brakiem dopasowań,
- `testSearchExactMatch()` – sytuacja, gdy wzorzec jest dokładnie równy całemu tekstowi,
- `testSearchHeavyInputWithLateMatch()` – test na dużym tekście z pojedynczym, późnym wystąpieniem wzorca, sprawdzający zarówno poprawność, jak i wydajność algorytmu.

8.2.5 Podsumowanie wyników

Wszystkie testy jednostkowe zakończyły się pozytywnym wynikiem. Nie stwierdzono błędów w implementacjach w testowanych przypadkach.

8.3 Narzędzia i środowisko testowe

W celu przeprowadzenia testów jednostkowych zastosowano następujące narzędzia i konfiguracje środowiskowe:

- **Konfiguracja sprzętowa:** Komputer z procesorem Intel Core i5-14600KF, 32 GB pamięci RAM oraz dyskiem SSD NVMe.
- **Język programowania:** Java 17 (OpenJDK).
- **Framework testowy:** JUnit 5.10.0.
- **Środowisko IDE:** IntelliJ IDEA 2023.2 (Community Edition).

- **System operacyjny:** Windows 11 64-bit.
- **System budowania:** Maven 3.9.5.
- **Struktura projektu:** Testy jednostkowe umieszczone w katalogu `src/test/java`, konfiguracja zawarta w pliku `pom.xml`.
- **Charakter testów:** Wszystkie testy uruchamiane były lokalnie.

Generowanie danych testowych W przypadku testów wymagających dużej liczby węzłów lub złożonych struktur wejściowych dane testowe generowano automatycznie za pomocą dedykowanego skryptu w języku Python. Skrypt umożliwia parametryzację liczby wierzchołków, ich rozmieszczenia przestrzennego oraz wartości atrybutów, co pozwala na elastyczne testowanie wydajności i poprawności działania algorytmów.

Listing 1: Skrypt generujący dane testowe

```
import random
import math

def generuj_graf(n_wierzcholkow, n_krawedzi, nazwa_suffix):
    typy_wierzcholkow = ["Pole", "Browar", "Karczma", "Skrzyzowanie"]
    wierzcholki = []
    pola_indeksy = []

    for i in range(n_wierzcholkow):
        typ = random.choice(typy_wierzcholkow)
        x = random.randint(-100000, 100000)
        y = random.randint(-100000, 100000)
        if typ == "Pole":
            zboze = random.randint(50, 150)
            wierzcholki.append(f"{typ} {x} {y} {zboze}")
            pola_indeksy.append(i)
        else:
            wierzcholki.append(f"{typ} {x} {y}")

    krawedzie = set()
    max_proba = 10 * n_krawedzi
    proby = 0
    while len(krawedzie) < n_krawedzi and proby < max_proba:
        a = random.randint(0, n_wierzcholkow - 1)
        b = random.randint(0, n_wierzcholkow - 1)
        if a != b:
            przepustowosc = random.randint(10, 50)
            koszt = random.randint(0, 5)
            if (a, b) not in krawedzie:
                krawedzie.add((a, b, przepustowosc, koszt))
        proby += 1

    nazwa_pliku = f"{n_wierzcholkow}_{nazwa_suffix}.txt"
```

```
    with open(nazwa_pliku, 'w', encoding='utf-8') as f:
        f.write(f"{n_wierzcholkow}\n")
        for w in wierzcholki:
            f.write(f"{w}\n")
        f.write(f"{len(krawedzie)}\n")
        for a, b, p, k in krawedzie:
            f.write(f"{a} {b} {p} {k}\n")

    print(f"Wygenerowano: {nazwa_pliku} ({len(krawedzie)} krawedzi)")

def main():
    random.seed(42)
    rozmiary = [100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000]
    for n in rozmiary:
        m_rzadki = 2 * n
        m_sredni = int(n * math.log2(n))
        m_gesty = n * (n - 1) // 2
        generuj_graf(n, m_rzadki, "rzadki")
        generuj_graf(n, m_sredni, "sredniogesty")
        generuj_graf(n, min(m_gesty, 500000), "gesty")

if __name__ == "__main__":
    main()
```

8.4 Testy wydajnościowe

8.4.1 Charakterystyka danych testowych

Dane testowe generowano przy użyciu dedykowanego skryptu losującego grafy skierowane o różnych gęstościach i rozmiarach. Testowane były trzy klasy grafów:

- **Grafy rzadkie:** liczba krawędzi skierowanych rzędu $m = 2n$
- **Grafy średnio-gęste:** liczba krawędzi skierowanych rzędu $m \approx n \log_2 n$
- **Grafy gęste:** liczba krawędzi skierowanych rzędu $m \approx n(n-1)$, ograniczona do maksymalnie 500000 krawędzi ze względów praktycznych

Próbki danych obejmowały liczby wierzchołków: 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000. Dla każdego rozmiaru wygenerowano odpowiednie grafy w trzech kategoriach gęstości.

Wierzchołki zostały oznaczone różnymi typami, a krawędzie posiadały losowe pojemności z zakresu $[10, 50]$ oraz koszty aktywacji z zakresu $[0, 5]$.

Tabela 3: Liczba krawędzi w grafach o różnej gęstości w zależności od liczby wierzchołków

Liczba wierzchołków	Graf rzadki	Graf średniogęsty	Graf gęsty
100	200	664	49 500
1 000	2 000	9 970	499 500

ciąg dalszy na następnej stronie

Tabela 3 – cd.

Liczba wierzchołków	Graf rzadki	Graf średniogęsty	Graf gęsty
2 000	4 000	21 940	1 999 000
3 000	6 000	34 170	4 498 500
4 000	8 000	46 560	7 998 000
5 000	10 000	59 220	12 497 500
6 000	12 000	72 130	17 997 000
7 000	14 000	85 280	24 496 500
8 000	16 000	98 640	31 996 000
9 000	18 000	112 180	40 495 500
10 000	20 000	125 970	49 995 000

8.4.2 Metodyka testów

Dla każdego grafu wykonano pomiary czasu działania zmodyfikowanego algorytmu Busackera-Gowena obliczającego maksymalny przepływ z kosztami aktywacji krawędzi.

- Dla mniejszych zestawów danych (do *2*000 wierzchołków) testy były powtarzane wielokrotnie w celu wyeliminowania wpływu krótkotrwałych fluktuacji obciążenia systemu oraz zwiększenia wiarygodności wyników.
- Dla większych zestawów danych (powyżej *2*000 wierzchołków) testy wykonano jednokrotnie ze względu na znaczący czas wykonania oraz ograniczenia zasobowe.
- W związku z tym wyniki dla dużych danych należy traktować jako orientacyjne, natomiast wyniki dla mniejszych zestawów jako uśrednione i bardziej stabilne.
- Monitorowano również wykorzystanie pamięci, które pozostawało na poziomie $O(n^2)$ ze względu na reprezentację macierzową. W praktyce średnie zużycie pamięci RAM wynosiło około 1 GB lub mniej, natomiast górne wartości oscylowały między 2 GB a nieco ponad 4 GB.

8.4.3 Wyniki pomiarów dla Algorytmu Edmondsa-Karpa

Poniżej przedstawiono zestawienie wyników czasowych dla różnych rozmiarów grafów i ich gęstości.

Tabela 4: Czas wykonania algorytmu w sekundach dla grafów o różnej gęstości

Liczba wierzchołków	Graf rzadki	Graf średniogęsty	Graf gęsty
100	0,0008	0,0020	0,0010
1 000	0,2004	0,9263	0,1862
2 000	1,7027	6,4932	0,8748
3 000	5,8676	26,6598	3,2753
4 000	14,0062	52,6241	8,6309
5 000	26,9193	103,6005	18,1439
6 000	40,2044	144,7313	29,1180
7 000	65,2934	240,6895	44,5770
8 000	107,2146	409,5329	75,4426

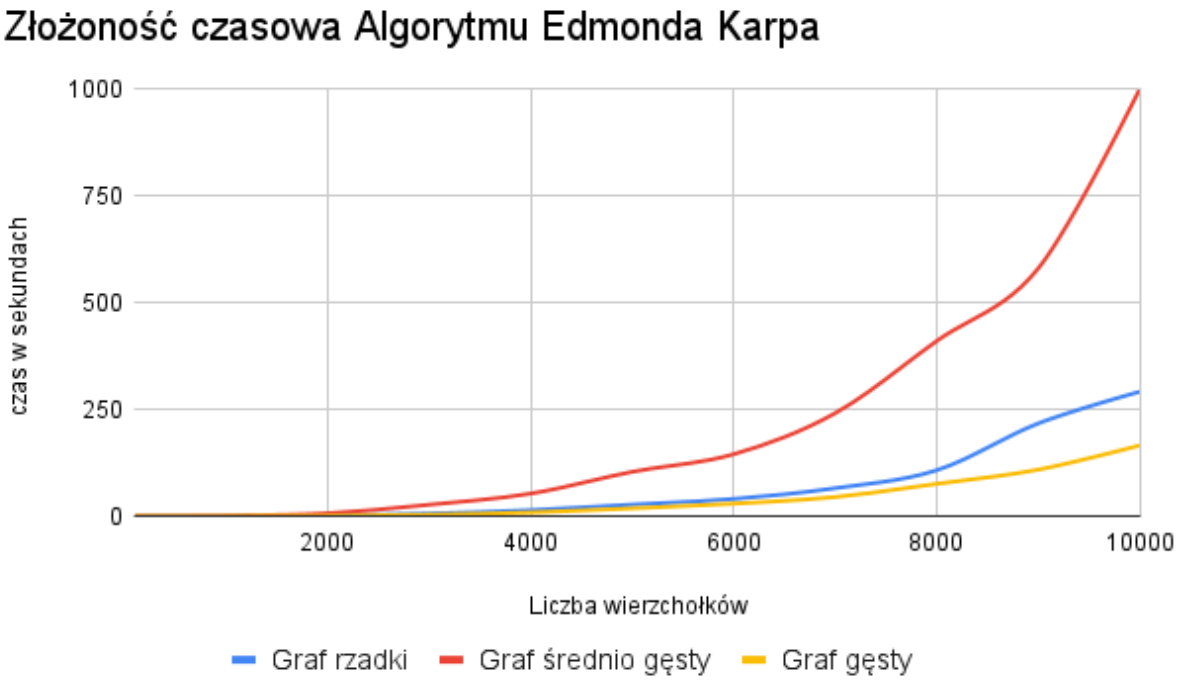
Kontynuacja na następnej stronie

Kontynuacja tabeli 4

Liczba wierzchołków	Graf rzadki	Graf średniogęsty	Graf gęsty
9 000	216,6656	579,0583	108,6273
10 000	290,8848	998,6867	165,2637

Analiza wykresów potwierdza, że czas działania algorytmu rośnie wraz ze wzrostem liczby wierzchołków oraz gęstości krawędzi:

- Dla grafów rzadkich czas wykonania jest stosunkowo niski i rośnie mniej więcej w granicach $O(n^3)$, co jest charakterystyczne dla algorytmu Edmondsa-Karpa na rzadkich sieciach.
- Grafy średniogęste wykazują znaczący wzrost czasu, co odzwierciedla kwadratową lub nawet sześcienną złożoność w zależności od parametrów grafu i pojemności krawędzi.
- W przypadku grafów gęstych czas wykonania pozostaje najniższy w porównaniu z innymi typami grafów, co może wynikać z właściwości testowanego zbioru danych (np. mniejsze pojemności lub specyficzna struktura).



Rysunek 1: Zależność czasu działania od liczby wierzchołków dla różnych gęstości grafu

8.4.4 Wyniki pomiarów dla Algorytmu Busackera-Gowena

Poniżej przedstawiono zestawienie wyników czasowych dla różnych rozmiarów grafów i ich gęstości.

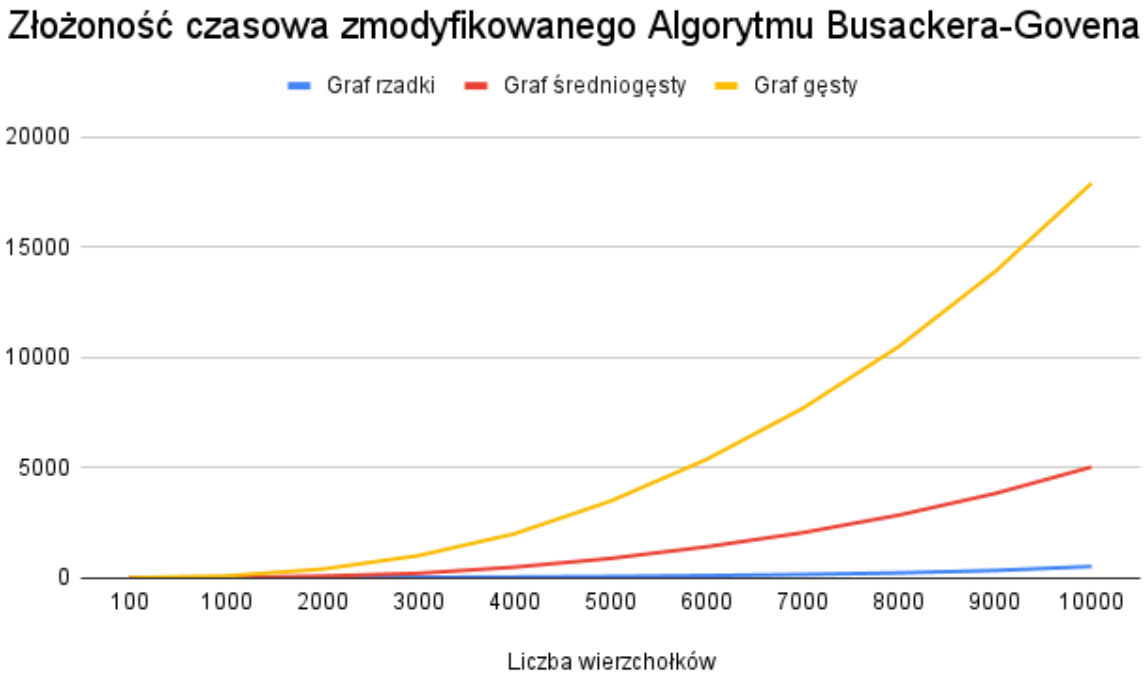
Tabela 5: Czas wykonania algorytmu w sekundach dla grafów o różnej gęstości

Liczba wierzchołków	**Graf rzadki**	**Graf średniogęsty**	**Graf gęsty**
100	0,007	0,0113	0,1079
1 000	0,8032	6,5745	75,2766
2 000	3,7068	66,5163	383,4938
3 000	11,3555	192,9521	994,0673
4 000	27,9872	474,7621	1989,232
5 000	51,5078	870	3472
6 000	89,0778	1400	5370
7 000	142,8184	2040	7690
8 000	213,5527	2840	10500
9 000	323,4046	3820	13900
10 000	504,8711	5020	17900

Analiza wykresów potwierdza, że czas działania algorytmu rośnie wraz ze wzrostem liczby wierzchołków oraz gęstości krawędzi:

- Grafy rzadkie charakteryzują się najszybszym czasem działania, przy czym czas rośnie mniej więcej liniowo z rozmiarem n.
- Dla grafów średnio-gęstych czas wykonania wykazuje wzrost zbliżony do $O(n^2)$, co jest zgodne z przewidywaniami teoretycznymi.

- Grafy gęste wykazują najsilniejszy wzrost czasu, osiągając wartości, które w praktyce ograniczają skalę problemu do około kilku tysięcy wierzchołków.



Rysunek 2: Zależność czasu działania od liczby wierzchołków dla różnych gęstości grafu

8.5 Wyniki testów i wnioski

Przeprowadzone testy miały na celu przede wszystkim **weryfikację poprawności implementacji algorytmów** oraz **określenie ich zachowania czasowego w zależności od rozmiaru i gęstości grafów**. Analiza wyników pozwoliła nam zweryfikować, czy implementacje działają zgodnie z przewidywaniami teoretycznymi oraz ocenić ich praktyczną efektywność.

1. **Weryfikacja poprawności i złożoności**
   Uzyskane czasy wykonania są zgodne z oczekiwaniami wynikającymi z analizy teoretycznej złożoności obu algorytmów. Zarówno wzrost czasu działania wraz ze zwiększaniem liczby wierzchołków, jak i zależność od gęstości grafu potwierdzają, że implementacje funkcjonują poprawnie i zgodnie z założeniami.

2. **Spełnienie celów projektowych**
   Testy obejmowały rozmiary grafów i ich gęstości charakterystyczne dla zakładanego zakresu zastosowań — mianowicie grafy o liczbie wierzchołków w przedziale 100 do 5000 oraz odpowiadającej im liczbie krawędzi typowej dla grafów rzadkich i średnio gęstych. Choć algorytmy zostały zaprojektowane tak, aby obsługiwać również większe zbiory danych, przyjęto, że docelowy zakres, np. miasto Shire, będzie mieścił się w przedziale od 5 000 do 10 000 wierzchołków. W tym obszarze wyniki są satysfakcjonujące i potwierdzają, że algorytmy są gotowe do zastosowania w przewidzianych scenariuszach.

3. **Rekomendacje dalszych działań**
   Ze względu na rosnące czasy działania dla bardzo dużych i gęstych grafów, dalsze prace mogłyby obejmować optymalizację implementacji lub zastosowanie bardziej

efektywnych algorytmów na potrzeby szerszego zakresu danych. Jednak dla obecnych wymagań testowane algorytmy są wystarczająco wydajne i stabilne.

Podsumowując, testy potwierdziły zarówno poprawność implementacji, jak i zgodność wyników z przewidywaniami teoretycznymi, co stanowi solidną podstawę do wykorzystania algorytmów w docelowych zastosowaniach.

8.6 Uwagi o braku walidacji danych wejściowych

W obecnej wersji aplikacji nie zaimplementowano automatycznej walidacji danych wejściowych. Przyjmuje się, że dane przekazywane do systemu są zgodne z wymaganym formatem oraz spełniają podstawowe założenia algorytmu, takie jak:

- brak ujemnych wag krawędzi (jeśli algorytm tego wymaga),
- spójność struktury danych, np. brak odwołań do nieistniejących wierzchołków w definicji krawędzi.

Decyzja o pominięciu warstwy walidacyjnej została podjęta świadomie — priorytetem było skoncentrowanie się na implementacji i testowaniu podstawowej logiki algorytmicznej. Uwzględnienie mechanizmów walidacji danych wejściowych może zostać rozważone jako element przyszłej rozbudowy systemu.

9 Rodzaj wykorzystanej technologii, języków programowania

- **Backend:** Java
- **Frontend:** JavaFX (desktopowa aplikacja graficzna)
- **System kontroli wersji:** Git (repozytorium hostowane na GitHub)
- **Zarządzanie zależnościami i budowaniem projektu:** Maven
- **Ułatwienia w kodzie:** Lombok (automatyzacja generowania kodu — np. getterów, setterów)