
Politechnika Rzeszowska im. Ignacego Łukasiewicza

Wydział Matematyki i Fizyki Stosowanej

**PRACA MAGISTERSKA**

kierunek studiów: inżynieria i analiza danych

# Analiza i zastosowanie algorytmu A* w problemach optymalizacji ścieżek.

Adrian Dereń

Promotor: dr Paweł Bednarz

Rzeszów, 2025

*„Problem najkrótszej ścieżki można rozwiązać za pomocą metody dynamicznego programowania, która polega na stopniowym budowaniu rozwiązania poprzez rozwiązywanie mniejszych pod-problemów.”*

*- Richard Ernest Bellman*

# Spis treści

1. Wstęp ..... 7
1.1. Motywacja do wyboru tematu pracy magisterskiej ..... 7
1.2. Cel pracy ..... 7
1.3. Zakres pracy ..... 7
2. Część teoretyczna ..... 8
2.1 Czym jest algorytm? ..... 8
2.1.1 Definicja algorytmu ..... 8
2.1.2 Cechy dobrego algorytmu ..... 10
2.1.3 Przykład praktyczny algorytmu ..... 11
2.1.4 Złożoność obliczeniowa algorytmu ..... 12
2.2 Wprowadzenie do teorii grafów ..... 13
2.2.1 Definicja grafu ..... 13
2.2.2 Rodzaje grafów ..... 15
2.2.3 Graf ważony ..... 15
2.3 Algorytmy wyszukiwania najkrótszej ścieżki ..... 17
2.3.1 Przeszukiwanie wszerz (BFS) ..... 18
2.3.2 Przeszukiwanie w głąb (DFS) ..... 20
2.3.3 Algorytm Dijkstry ..... 22
2.3.4 Algorytm Bellmana-Forda ..... 23
2.3.5 Zastosowania Algorytmów wyszukiwania ścieżek ..... 25
2.3.6 Funkcja kosztu ..... 26
2.4 Algorytm A* ..... 28
2.4.1 Działanie algorytmu ..... 28
2.4.2 Heurystyka – definicja i rola ..... 29
2.4.3 Rodzaje heurystyk ..... 30
2.4.4 Pseudokod algorytmu A* ..... 31
2.4.5 Przykład – najkrótsza droga z Rzeszowa do Gdańska ..... 32
2.4.6 Wyróżniki praktyczne algorytmu A* ..... 37
3. Część praktyczna ..... 38
3.1 Środowisko testowe i użyte narzędzia ..... 38
3.2 Implementacja w grafie ważonym ..... 39

3.2.1 Fragmenty kodu algorytmów oraz funkcji do wizualizacji ... 39
3.2.1.1 Generowanie grafu ważonego ... 39
3.2.1.2 Algorytm BFS ... 40
3.2.1.3 Algorytm DFS ... 41
3.2.1.4 Algorytm Dijkstry ... 42
3.2.1.5 Algorytm A* ... 43
3.2.1.6 Funkcje heurystyki dla algorytmu A* ... 44
3.2.1.7 Funkcja obliczania parametrów i wizualizacji ... 45
3.2.2 Przykładowy przebieg działania i wizualizacja ... 46
3.2.3 Tabele porównawcze wyników dla grafów ważonych ... 49
3.2.4 Wnioski z analizy porównawczej na grafie ważonym ... 50
3.3 Implementacja w siatce dwuwymiarowej ... 51
3.3.1 Dostosowanie algorytmów do grafu siatkowego ... 51
3.3.1.1 Reprezentacja wierzchołków w siatce 2D ... 52
3.3.1.2 Wizualizacja i odświeżanie siatki ... 53
3.3.1.3 Obsługa klawiatury i uruchamianie algorytmów ... 53
3.3.2 Przykłady działania algorytmów na siatce 2D ... 55
3.3.3 Tabele porównawcze wyników dla siatki 2D ... 62
3.3.4 Wnioski z analizy porównawczej na siatce 2D ... 63
3.4 Implementacja na mapie drogowej Rzeszowa ... 64
3.4.1 Implementacja algorytmów ... 64
3.4.1.1 Pobranie grafu ... 64
3.4.1.2 Wczytanie grafu z pliku ... 65
3.4.1.3 Lokalizacja punktów startowych i końcowych ... 65
3.4.1.4 Integracja algorytmów ... 65
3.4.1.5 Połączenie z interfejsem ... 66
3.4.2 Wizualizacja wyników ... 66
3.4.3 Tabelaryczne zestawienie wyników dla każdego algorytmu ... 69
3.4.4 Interpretacja i podsumowanie wyników na rzeczywistej mapie ... 70
3.5 Wnioski z przeprowadzonych Testów ... 71
4. Podsumowanie ... 74
5. Bibliografia ... 75
6. Załączniki ... 77

7. Streszczenie ..... 78

# 1. Wstęp

## 1.1. Motywacja do wyboru tematu pracy magisterskiej

Wraz z dynamicznym rozwojem technologii coraz większe znaczenie zyskują algorytmy wspierające procesy decyzyjne oraz planowanie działań w złożonych środowiskach. Wśród nich szczególnie istotne są metody umożliwiające znajdowanie rozwiązań optymalnych. Jedną z takich metod jest algorytm A*, który ze względu na swoje praktyczne zastosowania stał się przedmiotem niniejszej analizy.

Wybór tego tematu uzasadniony był nie tylko zainteresowaniem algorytmiką, ale również chęcią połączenia wiedzy teoretycznej z praktyką. W pracy skoncentrowano się na dokładnym omówieniu działania algorytmu A*, jego porównaniu z innymi klasycznymi podejściami oraz analizie wpływu zastosowanej heurystyki na skuteczność i szybkość przeszukiwania.

Dodatkowym motywem była możliwość samodzielnego wykonania implementacji i przeprowadzenia eksperymentów. Praktyczne testy w środowisku programistycznym pozwoliły nie tylko zweryfikować teoretyczne założenia, ale również zrozumieć, w jakich warunkach algorytm A* sprawdza się najlepiej, a w jakich jego działanie może być mniej efektywne.

## 1.2. Cel pracy

Celem pracy było zbadanie skuteczności i przydatności algorytmu A* w różnych środowiskach oraz porównanie go z innymi metodami wyszukiwania ścieżek, takimi jak BFS, DFS i Dijkstra. Szczególny nacisk położono na analizę wpływu rodzaju zastosowanej heurystyki na efektywność działania algorytmu. Praca miała także na celu ocenę praktycznej wartości poszczególnych podejść w kontekście szybkości działania, zużycia zasobów oraz jakości wyznaczonych tras.

## 1.3. Zakres pracy

Praca obejmuje zarówno część teoretyczną, jak i praktyczną. W części teoretycznej przedstawiono podstawy teorii grafów, omówiono różne typy grafów oraz klasyczne algorytmy przeszukiwania. Główna uwaga została poświęcona na algorytmowi A*, jego działaniu oraz roli heurystyki.

W części praktycznej zaimplementowano wybrane algorytmy w języku Python i przeprowadzono testy porównawcze w trzech środowiskach:

- grafach ważonych,
- siatkach 2D,
- rzeczywistej mapie drogowej.

Oceniano czas działania, liczbę odwiedzonych wierzchołków, zużycie pamięci oraz długość tras. Nie analizowano natomiast metod opartych na uczeniu maszynowym ani zagadnień dynamicznych. Zakres został celowo ograniczony do klasycznych podejść, by skupić się na jakościowej ocenie ich praktycznego zastosowania.

# 2. Część teoretyczna.

W ostatnich latach można zauważyć rosnące znaczenie takich zagadnień w kontekście nowoczesnych systemów informatycznych i inżynieryjnych. Algorytmy wyznaczania ścieżek są dziś podstawą działania wielu narzędzi – od prostych aplikacji codziennego użytku po zaawansowane rozwiązania stosowane w logistyce, robotyce czy systemach nawigacyjnych. Graf, jako struktura danych, umożliwia przejrzyste odwzorowanie powiązań między obiektami i analizę możliwych scenariuszy poruszania się w danym środowisku [1].

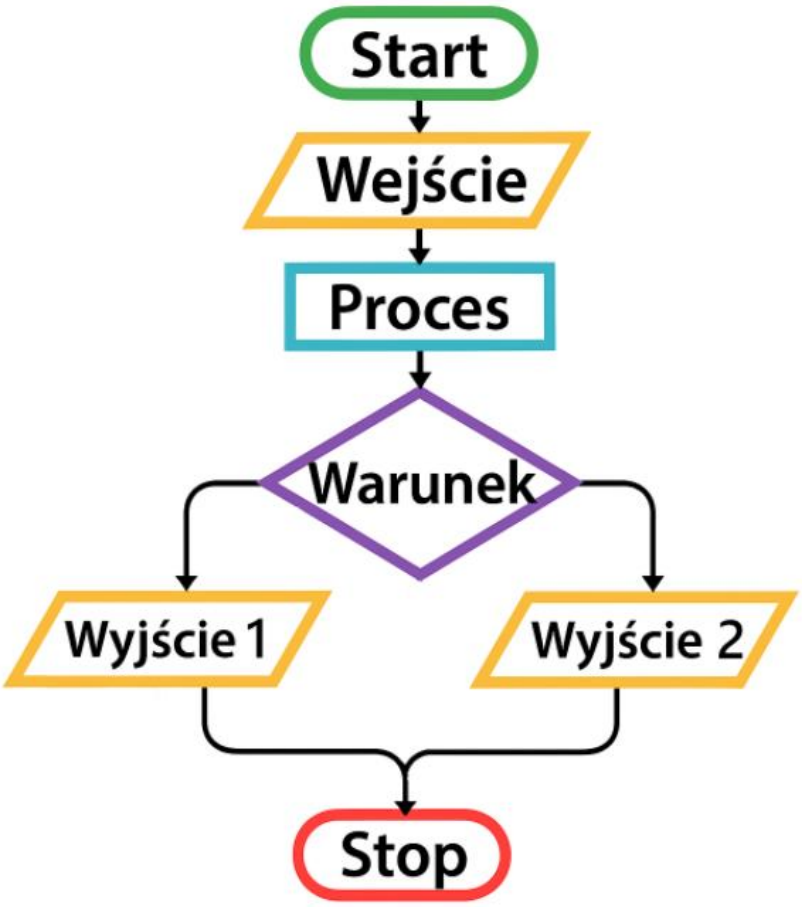
## 2.1 Czym jest algorytm?

Algorytmy są jednym z podstawowych elementów informatyki i wielu dziedzin związanych z przetwarzaniem danych. Umożliwiają rozwiązywanie różnych problemów poprzez wykonywanie kolejnych, dokładnie określonych kroków. Mają duże znaczenie zarówno w teorii, jak i w praktycznych zastosowaniach. Dzięki nim możliwe jest działanie programów komputerowych, systemów sztucznej inteligencji czy aplikacji wykorzystywanych w inżynierii i automatyce [4][5].

### 2.1.1 Definicja algorytmu

Pojęcie algorytmu ma swoje korzenie w pracach perskiego uczonego Muhammada ibn Musy al-Chuwarizmiego, który już w IX wieku przedstawił zestawy reguł potrzebnych do rozwiązywania problemów arytmetycznych. Obecnie, w informatyce, przyjmuje się, że algorytm to zestaw jasno określonych kroków, które prowadzą do znalezienia rozwiązania danego zadania [5]

Według jednej z klasycznych definicji, algorytm *„jest pewną ściśle określoną procedurą obliczeniową, która dla właściwych danych wejściowych „produkuje" żądane dane wyjściowe*[1] *"*. W uproszczeniu, można powiedzieć, że algorytm to dokładny przepis na to, jak przekształcić dane wejściowe w wynik. Każdy krok takiego przepisu musi być jednoznaczny i możliwy do wykonania. Proces ten kończy się po określonej liczbie operacji i zawsze prowadzi do rezultatu, o ile dane wejściowe są poprawne (rys. 2.1). Istotne jest, że algorytm to koncepcja niezależna od konkretnego języka programowania czy systemu komputerowego. Może być zapisany w różnych formach, ale sam jego opis logiczny pozostaje taki sam – niezależnie od tego, na jakim sprzęcie czy w jakim języku zostanie zrealizowany [1][4].




Rys. 2.1. Graficzna reprezentacja algorytmu – schemat blokowy

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, & C. Stein. (2009). Introduction to Algorithms (3rd ed.). MIT Press. Str. 5

### 2.1.2 Cechy dobrego algorytmu

Aby uznać algorytm za poprawnie zaprojektowany, powinien spełniać kilka ważnych warunków. Oto najważniejsze z nich:

**a) Jednoznaczność.**

Każda instrukcja w algorytmie musi być zrozumiała i jednoznaczna. Nie może być tak, że różni wykonawcy zinterpretują ją inaczej. Dzięki temu możliwe jest dokładne powtarzanie tych samych kroków, niezależnie od środowiska [4]

**b) Skończoność**

Algorytm nie powinien działać bez końca, a zatem musi mieć wyraźny moment, w którym przerywa pracę. Wyjątkiem są sytuacje, gdzie ciągłe działanie jest zamierzone, np. w systemach serwerowych czy robotach pracujących w pętli [4][5].

**c) Wydajność**

Dobrze zaprojektowany algorytm nie powinien zużywać zbyt dużo czasu ani pamięci. Przy ocenie tego aspektu często używa się pojęcia złożoności – czyli jak dużo operacji potrzeba w zależności od wielkości danych. Im mniejsza złożoność, tym lepiej dla realnych zastosowań [1].

**d) Uniwersalność**

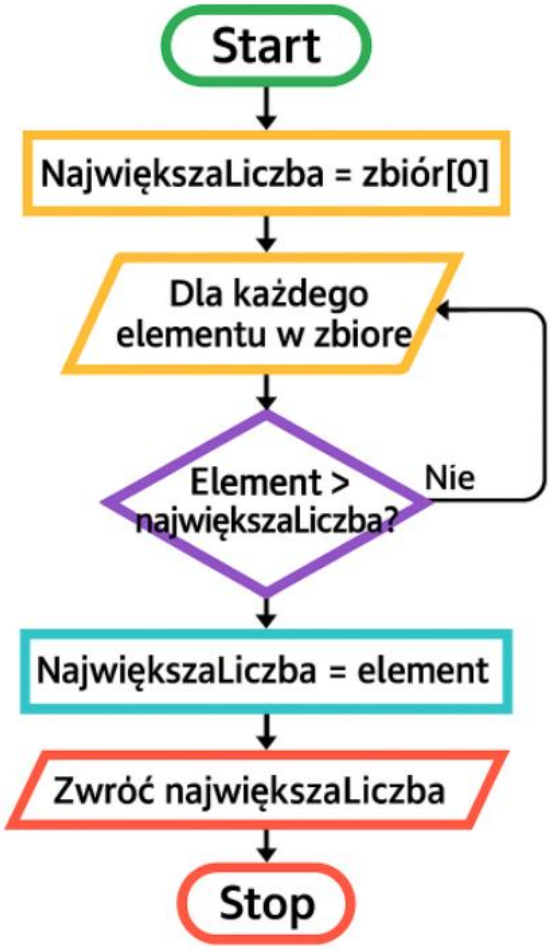
Choć algorytmy są tworzone pod konkretny typ problemu, dobry projekt powinien działać dla różnych danych wejściowych, nie tylko jednego konkretnego przypadku. To pozwala stosować go w szerszym zakresie sytuacji [4].

**e) Praktyczność i czytelność**

Nawet jeśli algorytm działa dobrze, nie może być zbyt trudny do zrozumienia. Zbyt skomplikowane podejście może sprawić problemy przy wdrażaniu, testach czy modyfikacjach. Łatwość implementacji to ważna cecha, zwłaszcza w zespole [5].

### 2.1.3 Przykład praktyczny algorytmu

Do zilustrowania powyższych cech, można sięgnąć po prosty przykład wyszukiwania największej liczby w zbiorze (rys. 2.2). Algorytm działa w ten sposób, że przegląda kolejne elementy listy i porównuje je ze sobą, zapamiętując największą wartość. Wszystkie kroki są jasno określone, więc nie ma tu miejsca na różne interpretacje. Całość kończy się po sprawdzeniu ostatniego elementu, więc działanie jest ograniczone w czasie. Taki algorytm sprawdzi się niezależnie od tego, ile liczb znajduje się w liście, dlatego jest uniwersalny. Dodatkowo, nie wymaga dużych zasobów i łatwo go zaimplementować.




Rys. 2.2. Schemat blokowy algorytmu znajdowania największej liczby w zbiorze

### 2.1.4 Złożoność obliczeniowa algorytmu

Podrozdział 2.1.4 został w całości opracowany w oparciu o [1].

Ocena algorytmu to nie tylko sprawdzenie, czy działa poprawnie, ale też jak dobrze radzi sobie pod względem wydajności. W tym celu analizuje się, ile zasobów potrzebuje do rozwiązania problemu, w zależności od ilości danych wejściowych.

**a) Złożoność czasowa**

Złożoność czasowa opisuje, ile operacji musi wykonać algorytm, żeby uzyskać wynik. Nie chodzi tu o czas w sekundach, ale o liczbę prostych działań, takich jak porównań czy przesunięć danych. Ilość tych operacji rośnie zwykle razem z liczbą danych, najczęściej oznaczaną przez $n$.

Aby uogólnić analizę i uniezależnić ją od sprzętu oraz języka programowania, stosuje się tzw. analizę asymptotyczną. Pokazuje ona, jak algorytm zachowuje się przy dużych rozmiarach wejścia. Używa się do tego notacji *„wielkiego O”*, która opisuje maksymalną liczbę operacji w najgorszym możliwym przypadku.

**Przykładowe interpretacje:**

- $O(1)$ – czas wykonania nie zależy od rozmiaru danych (np. odczyt konkretnego elementu z tablicy).
- $O(n)$ – liczba operacji rośnie proporcjonalnie do wielkości danych (np. algorytm z rys. 2.2, algorytm przeszukiwania liniowego).
- $O(n^2)$ – czas rośnie kwadratowo względem liczby danych (np. algorytm sortowania bąbelkowego).
- $O(\log(n))$ – czas rośnie bardzo wolno, np. w algorytmie wyszukiwania binarnego.

Dla przykładu, algorytm z rys. 2.2 (wyszukiwanie największej liczby) sprawdza każdy element dokładnie raz, więc jego złożoność czasowa to $O(n)$. Dla listy z milionem liczb, algorytm wykona milion porównań.

**b) Złożoność pamięciowa**

Oprócz czasu liczy się też, ile dodatkowej pamięci potrzebuje algorytm. Chodzi o przestrzeń na dane pomocnicze, np. zmienne, tablice, stosy czy kolejki.

Podobnie jak w przypadku czasu, złożoność pamięciowa jest opisywana asymptotycznie, czyli względem liczby elementów wejściowych. Niektóre algorytmy potrzebują tylko jednej zmiennej do

przechowywania wyniku $(O(1))$, natomiast inne mogą tworzyć bardziej rozbudowane struktury pomocnicze $(O(n)), (O(n^2))$.

**Dla przykładu:**

Wspomniany algorytm wyszukiwania maksymalnej liczby z rys. 2.2 potrzebuje jedynie jednej zmiennej pomocniczej, ma więc złożoność pamięciową $O(1)$.

Inne algorytmy sortujące takie jak np. Merge Sort, które rekurencyjnie dzielą dane i scalają wyniki, wykorzystują dodatkową przestrzeń proporcjonalną do wielkości danych, wtedy złożoność pamięciowa wynosi $O(n)$.

**Znaczenie złożoności w praktyce**

Złożoność ma zatem duże znaczenie w praktyce. Pozwala przewidzieć, czy algorytm poradzi sobie z dużą ilością danych albo w środowisku o ograniczonych zasobach jak np. w aplikacjach mobilnych czy systemach wbudowanych. Analiza złożoności już na etapie projektowania pozwala uniknąć problemów z wydajnością i dobrać lepsze rozwiązanie, jeśli jest taka potrzeba.

## 2.2 Wprowadzenie do teorii grafów

Początki teorii grafów sięgają XVIII wieku a dokładniej roku 1741, kiedy Leonhard Euler w swojej pracy „S*olutio problematis ad geometriam situs pertinentis* w *Commentarii academiae scientiarum Petropolitanae*" opisał problem mostów w Królewcu. To właśnie wtedy po raz pierwszy pojawiło się matematyczne podejście do podobnych zagadnień. Grafy to struktury, które pozwalają przedstawić powiązania między obiektami. Są często używane w informatyce, bo znajdują zastosowanie w opisywaniu rzeczy takich jak sieci, zależności czy połączenia [6].

Zrozumienie, jak działają grafy, jest kluczowe dla dalszej części pracy, ponieważ wiele algorytmów – w tym te, które służą do wyszukiwania ścieżek opiera się właśnie na tej strukturze [1].

### 2.2.1 Definicja grafu

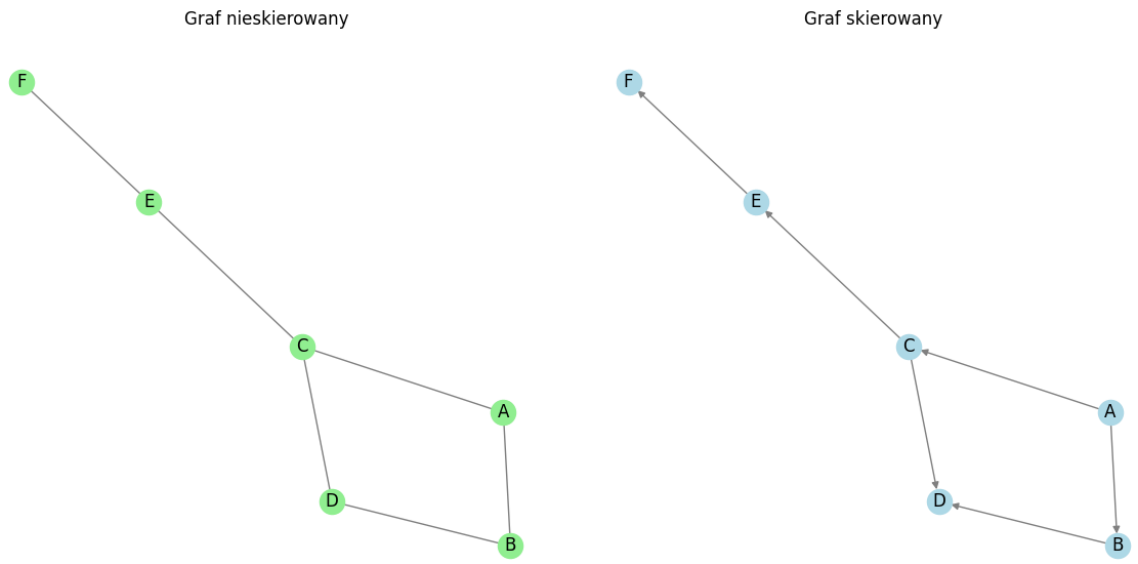
Graf $\boldsymbol{G}$ to para uporządkowana $\boldsymbol{G = (V, E)}$, gdzie:

- $\boldsymbol{V}$– zbiór wierzchołków,
- $\boldsymbol{E \subseteq V \times V}$ – zbiór krawędzi, czyli relacji pomiędzy parami wierzchołków.

W zależności od tego, jak zdefiniowane są połączenia między wierzchołkami, wyróżnia się dwa podstawowe typy grafów:

- **Graf nieskierowany** – krawędzie nie mają kierunku, tzn. że połączenie działa w obie strony,
- **Graf skierowany** (inaczej digraf) – każda krawędź ma określony kierunek, wskazujący, z którego wierzchołka wychodzi i do którego prowadzi [6][7].

Na rys. 2.3 pokazano porównanie grafu nieskierowanego oraz skierowanego. W obu przypadkach struktura wierzchołków i połączeń jest taka sama. W grafie nieskierowanym relacje są dwukierunkowe, natomiast w wersji skierowanej każda krawędź wskazuje określony kierunek przejścia między wierzchołkami.




Rys. 2.3. Przykład grafu nieskierowanego i skierowanego

Jako inny prosty przykład również może posłużyć mapa drogowa, gdzie wierzchołki reprezentują skrzyżowania, a krawędzie odcinki dróg między nimi. Jeśli droga jest jednokierunkowa, mamy doczynienia z grafem skierowanym. W przypadkach takich jak np. sieci komputerowe czy bazy danych, grafy odwzorowują przepływ informacji, zależności między rekordami lub relacje logiczne [6][7].

**Podstawowe elementy grafu:**

- **Wierzchołki** – obiekty, które są ze sobą powiązane,
- **Krawędzie** – połączenia między wierzchołkami,
- **Stopień wierzchołka** – liczba krawędzi wychodzących z danego wierzchołka (w grafie skierowanym rozróżnia się stopień wejściowy i wyjściowy),

- **Ścieżka** – ciąg wierzchołków połączonych krawędziami,
- **Cykl** – ścieżka, która zaczyna i kończy się w tym samym wierzchołku.

Dodatkowo w niektórych typach grafu mogą pojawić się:

- **Pętle –** krawędzie, które łączą wierzchołek z samym sobą. Występują w grafach które dopuszczają tego typu połączenia.
- **Krawędzie wielokrotne –** więcej niż jedna krawędź między tą samą parą wierzchołków. Spotykane są w tzw. multigrafach i pozwalają modelować sytuacje, w których między dwoma elementami istnieje kilka niezależnych relacji.

### 2.2.2 Rodzaje grafów

Oprócz grafów skierowanych i nieskierowanych, które zostały omówione w podrozdziale 2.2.1, w zależności od rodzaju krawędzi oraz charakteru połączeń między wierzchołkami, wyróżnia się również inne przypadki grafów [6]:

- **Graf prosty** – nie zawiera pętli ani krawędzi wielokrotnych pomiędzy tą samą parą wierzchołków.
- **Graf z pętlami** (inaczej pseudograf) – dopuszcza istnienie krawędzi prowadzącej z wierzchołka do samego siebie (pętli).
- **Graf wielokrotny** (inaczej multigraf) – umożliwia więcej niż jedno połączenie między tymi samymi wierzchołkami.
- **Graf ważony** – każdej krawędzi przypisana jest liczba (waga), która może oznaczać m.in. koszt, czas lub odległość.

Dodatkowo grafy można klasyfikować według ich struktury globalnej:

- **Graf spójny** – między każdą parą wierzchołków istnieje ścieżka łącząca je ze sobą.
- **Graf acykliczny** – nie zawiera cykli, czyli ścieżek zamkniętych w tym samym wierzchołku.
- **Drzewo** – to szczególny przypadek grafu, jest spójny i acykliczny, z dokładnie jedną ścieżką pomiędzy dowolnymi dwoma wierzchołkami.

### 2.2.3 Graf ważony

W wielu praktycznych problemach nie wystarcza informacja o samym istnieniu połączenia między wierzchołkami istotne staje się również to, jakie *„koszty”* lub *„wartości”* są z tymi połączeniami związane. W takich przypadkach stosuje się grafy ważone, w których każdej krawędzi przypisana jest pewna liczba, zwana wagą [1].

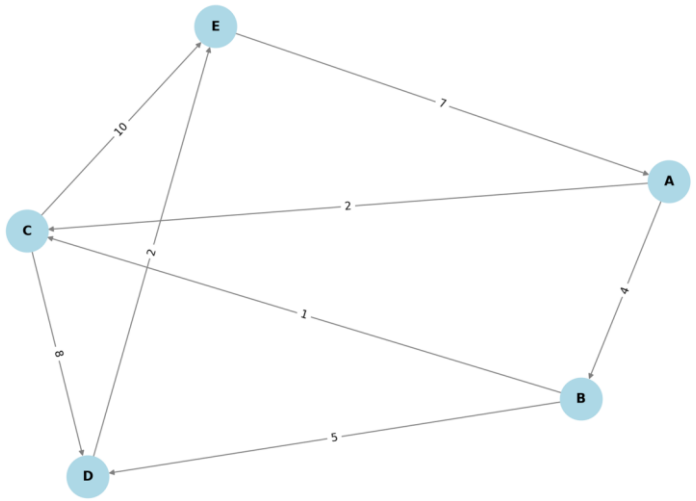
**Graf ważony** to trójka uporządkowana $\boldsymbol{G = (V, E, w)}$**,** gdzie:

- $\boldsymbol{V}$ – to zbiór wierzchołków
- $\boldsymbol{E \subseteq V \times V}$ – zbiór krawędzi
- $\boldsymbol{w : E \rightarrow \mathbb{R}}$ – funkcja wag przypisująca każdej krawędzi wartość liczbową

**Wartość wag może oznaczać:**

- **Długość fizyczną** (np. kilometry między miastami),
- **Czas trwania** (np. czasu przejazdu),
- **Koszt ekonomiczny** (np. cena przesyłki),
- **Przepustowość, ryzyko, opóźnienie** lub inne jednostki zależne od kontekstu zastosowania.

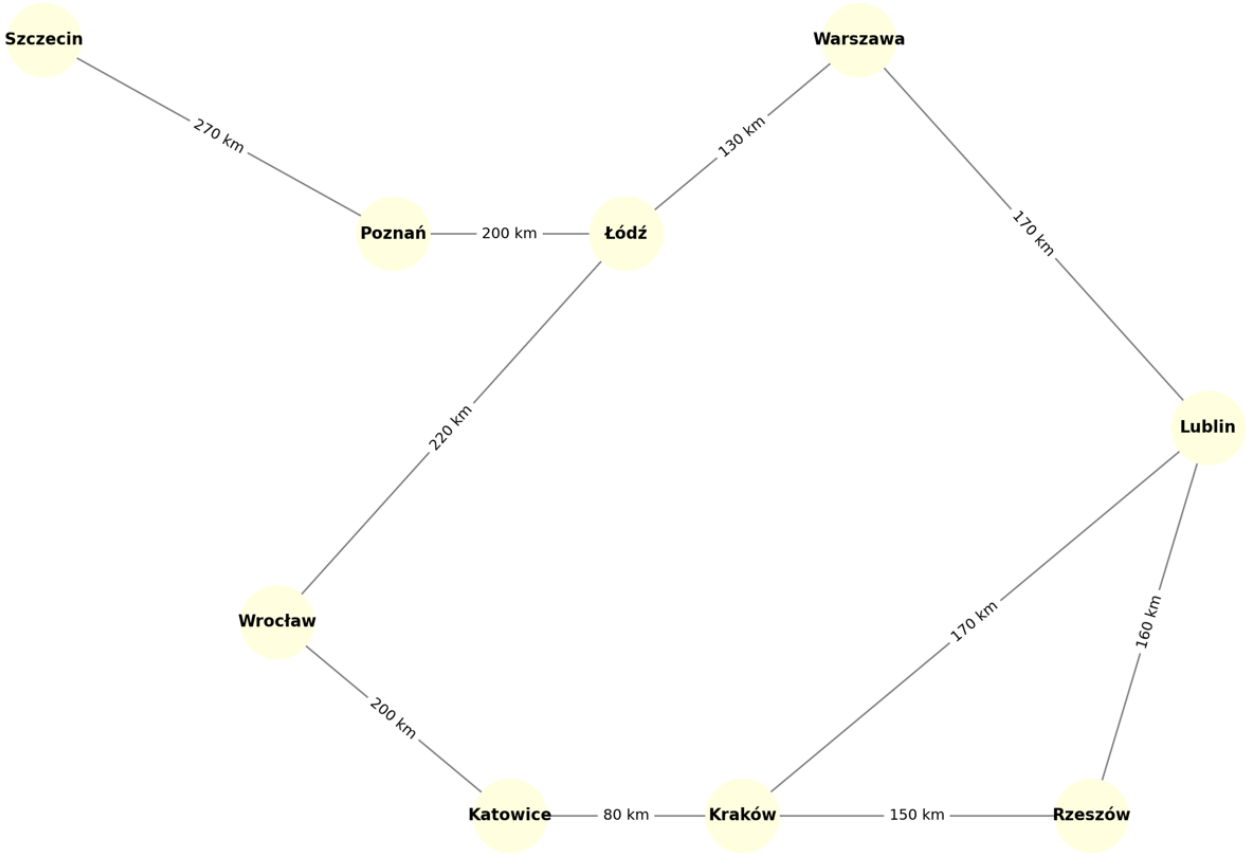
Z punktu widzenia pracy poświęconej algorytmom wyszukiwania najkrótszej ścieżki, graf ważony ma szczególne znaczenie. O ile w grafie nieważonym wystarczy znaleźć trasę z najmniejszą liczbą krawędzi, to w grafie ważonym konieczne staje się uwzględnienie sumy wag na ścieżce a to z kolei prowadzi do zastosowania bardziej skomplikowanych algorytmów [1][2].




Rys. 2.4. Przykładowy graf skierowany z wagami

Grafy ważone pojawiają się m.in. w [1]:

- Systemach nawigacyjnych GPS – gdzie wagi oznaczają czas przejazdu lub długość trasy,
- Sieciach komputerowych – wagi mogą symbolizować opóźnienia transmisji lub obciążenie kanału,
- Planowaniu logistycznym – koszt transportu lub energii potrzebnej do wykonania zlecenia,
- Zarządzaniu ruchem miejskim – dynamiczne wagi zmieniające się w zależności od natężenia ruchu.




Rys. 2.5. Przykład grafu ważonego reprezentującego sieć połączeń drogowych między wybranymi miastami w Polsce

## 2.3 Algorytmy wyszukiwania najkrótszej ścieżki

Problem wyznaczania najkrótszej ścieżki w grafie należy do najważniejszych zagadnień analizowanych w teorii grafów. Ma on istotne znaczenie nie tylko w ujęciu teoretycznym, ale także praktycznym i występuje w wielu obszarach. W zależności od rodzaju grafu, sposób rozwiązania może

się różnić. W grafach nieważonych często wystarcza ustalenie liczby przejść między wierzchołkami, natomiast w grafach z wagami kluczowa staje się suma przypisanych im kosztów [1].

W celu efektywnego przeszukiwania grafu stosuje się różne podejścia. W prostszych przypadkach, gdzie każda krawędź ma tę samą wagę, skuteczne okazują się algorytmy BFS i DFS. Jeśli jednak poszczególne połączenia różnią się wagą, konieczne jest użycie bardziej zaawansowanych metod, takich jak algorytm Dijkstry czy Bellmana-Forda [1].

W środowiskach o dużej złożoności i zmiennych warunkach na przykład w aplikacjach czasu rzeczywistego dużą rolę odgrywają algorytmy heurystyczne. Jednym z nich jest A*, który dzięki szacunkom dotyczącym odległości do celu znacząco skraca czas przeszukiwania dużych przestrzeni grafowych [1][6].

Wybór odpowiedniego algorytmu zależy od charakterystyki konkretnego przypadku: struktury grafu, dostępnych zasobów obliczeniowych czy też wymaganego poziomu dokładności. Często trzeba znaleźć kompromis między czasem działania a jakością wyniku [4].

Przykładowo, w systemach miejskiego transportu publicznego czy w dużych platformach e-commerce, gdzie kluczowe jest przetwarzanie danych „na żywo", istotne staje się dobranie takiego algorytmu, który dobrze skaluje się przy dużych zbiorach danych i zapewnia możliwie niską złożoność obliczeniową [5].

Jak podkreśla T. Cormen, problem najkrótszej ścieżki to nie tylko klasyczny przykład w teorii algorytmów, ale także jedno z najbardziej praktycznych zagadnień wykorzystywanych na co dzień w inżynierii i informatyce stosowanej [1].

### 2.3.1 Przeszukiwanie wszerz (BFS)

Algorytm BFS, czyli przeszukiwanie wszerz (ang. Breadth – First Search), to jedna z najprostszych i najbardziej znanych metod eksploracji grafu. Jego działanie polega na odwiedzaniu kolejnych *„warstw"* wierzchołków, zaczynając od punktu startowego, najpierw analizuje się wszystkich bezpośrednich sąsiadów, potem sąsiadów tych sąsiadów i tak dalej [1].

**Zasada działania**

Do kontroli przebiegu przeszukiwania BFS wykorzystuje strukturę danych w postaci kolejki (FIFO – First-In, First-Out). W pierwszym kroku umieszcza się w niej wierzchołek startowy. Następnie pobierany jest pierwszy element, analizowani są jego sąsiedzi i ci, którzy nie zostali jeszcze odwiedzeni,

trafiają do kolejki. Proces ten powtarza się do momentu, gdy nie pozostanie nic do sprawdzenia lub zostanie osiągnięty cel [1][4].

**Zastosowanie do znajdowania najkrótszej ścieżki**

W grafach, w których krawędzie nie mają wag, BFS umożliwia znalezienie ścieżki z najmniejszą liczbą kroków czyli najkrótszej ścieżki w sensie topologicznym. Zatem w tego typu sytuacjach daje gwarancję optymalności rozwiązania. Znajduje zastosowanie m.in. w analizie siatek drogowych, a także w grafach reprezentujących powiązania społeczne czy strukturę danych [1][8][9].

**Złożoność obliczeniowa:**

- **Złożoność czasowa**:
  **O(n + m),**
- **Złożoność pamięciowa**:
  **O(n + m)**, ponieważ konieczne jest przechowywanie kolejki oraz tablicy odwiedzin,

gdzie:

$\boldsymbol{n}$ – liczba wierzchołków,

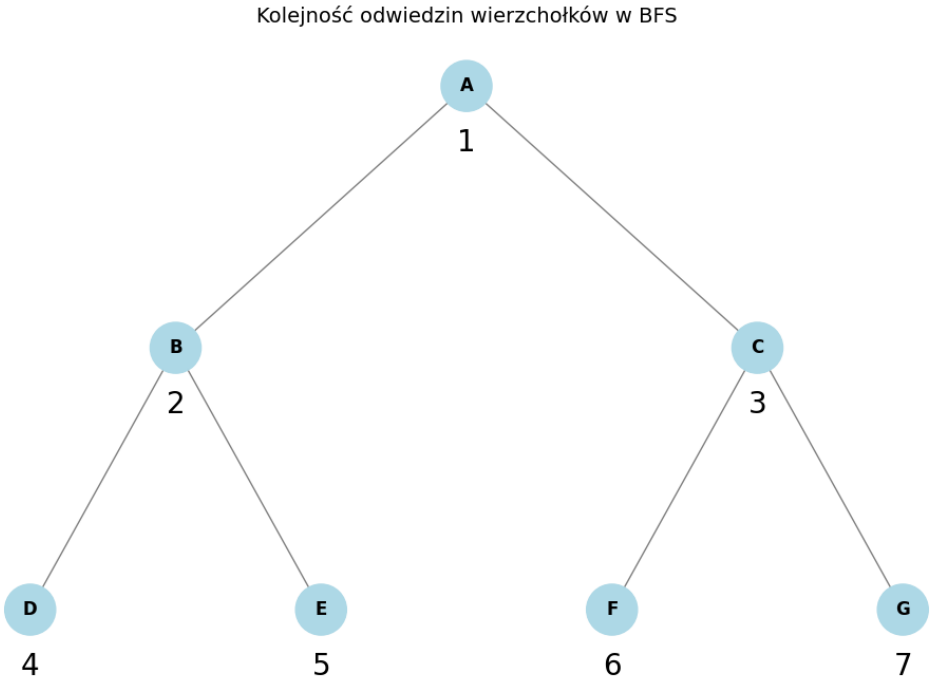
$\boldsymbol{m}$ – liczba krawędzi.

W praktyce, przy odpowiedniej implementacji, BFS może być wykorzystywany do obsługi dużych zbiorów danych, również w systemach działających w czasie rzeczywistym zwłaszcza w wersjach równoległych [10].

**Zalety i ograniczenia**

Największą zaletą tej metody jest prostota i pewność znalezienia najkrótszej trasy w grafach bez wag. Algorytm działa w sposób przewidywalny i nie wymaga skomplikowanych struktur. Wadą BFS jest jednak fakt, że nie uwzględnia różnic w kosztach przejść, dlatego nie nadaje się do grafów, gdzie poszczególne krawędzie mają różne znaczenie.

Na rys 2.6 pokazano przykład jak algorytm BFS przeszukuje graf.




Rys. 2.6. Przykład działania algorytmu przeszukiwania wszerz (BFS)

### 2.3.2 Przeszukiwanie w głąb (DFS)

Algorytm DFS (ang. Depth-First Search) to kolejna klasyczna metoda poruszania się po grafie, polegająca na zagłębianiu się w jego strukturę aż do momentu, gdy dalsze przejście nie jest możliwe. Dopiero wtedy następuje cofnięcie do wcześniejszych punktów i próba podążenia inną ścieżką. Tego typu eksploracja sprawdza się dobrze w zadaniach, gdzie istotne jest pełne przeanalizowanie ścieżek od danego punktu [1].

**Zasada działania**

Procedura rozpoczyna się od wybranego wierzchołka początkowego. Następnie wybierany jest jeden z dostępnych sąsiadów i algorytm podąża nim dalej, schodząc *„w głąb”* grafu. Gdy nie ma już dostępnych nieodwiedzonych sąsiadów, następuje powrót do poprzedniego wierzchołka i eksploracja kontynuowana jest z kolejnego dostępnego punktu. Proces ten można zrealizować rekurencyjnie lub za pomocą jawnego stosu [4].

**Złożoność obliczeniowa**

- **Złożoność czasowa:**
  **O(n + m)**
- **Złożoność pamięciowa:**
  **O(*h*)** – stos rekurencyjny lub jawna struktura stosu,

gdzie:

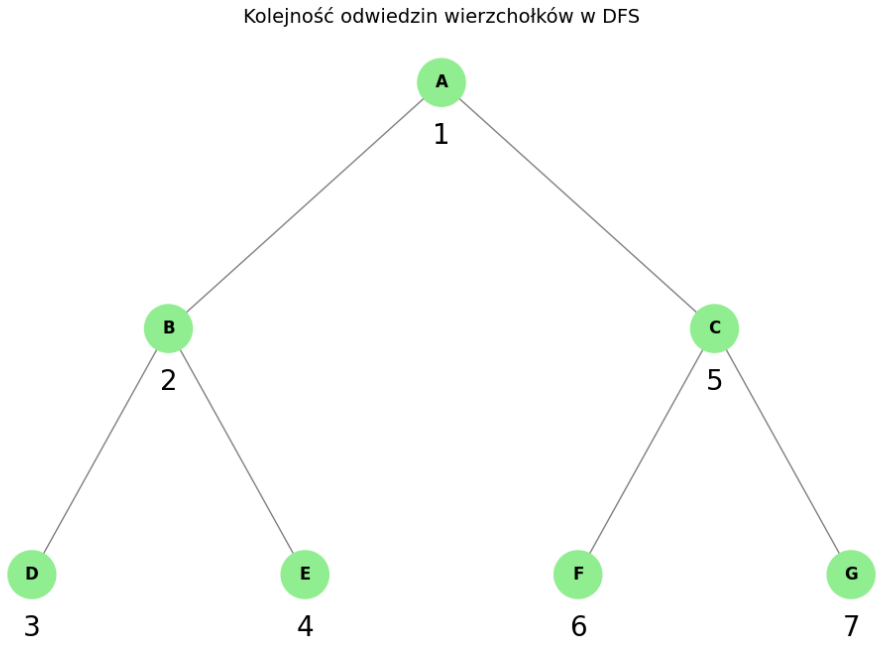
$\boldsymbol{n}$ – liczba wierzchołków,

$\boldsymbol{m}$ – liczba krawędzi,

$\boldsymbol{h}$ – długość najdłuższej prostej ścieżki.

**Ograniczenia w kontekście najkrótszej ścieżki**

DFS nie gwarantuje znalezienia ścieżki optymalnej. Ponieważ *„idzie w głąb”* pierwszą dostępną trasą, może pominąć krótsze drogi, które są osiągalne inną ścieżką. Mimo tego znajduje wiele praktycznych zastosowań, gdzie struktura grafu przypomina drzewo lub labirynt. W przypadkach wymagających minimalizacji odległości, czasu lub kosztu DFS ustępuje miejsca algorytmom takim jak BFS, Dijkstra czy A* [1].




Rys. 2.7. Przykład działania algorytmu przeszukiwania w głąb (DFS)

### 2.3.3 Algorytm Dijkstry

Jedną z najbardziej znanych metod wyznaczania najkrótszych ścieżek w grafach z wagami nieujemnymi jest algorytm opracowany przez Edsgera Dijkstrę w 1959 roku. Do dziś stanowi on podstawę w różnych dziedzinach od systemów GPS po analizę struktur sieciowych [1].

**Zasada działania**

Algorytm operuje na grafie, w którym każda krawędź ma przypisaną nieujemną wagę. Działanie rozpoczyna się od wierzchołka startowego, któremu przypisuje się koszt 0. Pozostałym przypisywane są początkowo wartości nieskończone. W kolejnych krokach wybierany jest wierzchołek o najmniejszym znanym koszcie, a następnie aktualizowane są wartości dojścia do jego sąsiadów – o ile możliwe jest ich poprawienie. Przetworzony wierzchołek zostaje oznaczony jako odwiedzony, by nie przetwarzać go ponownie.

Dla zwiększenia efektywności algorytmu stosuje się kolejkę priorytetową – dzięki niej można szybciej wybierać wierzchołki o minimalnym koszcie [1].

**Złożoność obliczeniowa**

- **Czasowa**:
  **$O(n^2)$** – przy implementacji z tablicą,
  **$O((n + m) \log n)$** – przy implementacji z kolejką priorytetową (np. kopcem Fibonacciego),
- **Pamięciowa:**
  **$O(n + m)$** – do przechowywania kosztów i stanu odwiedzenia,

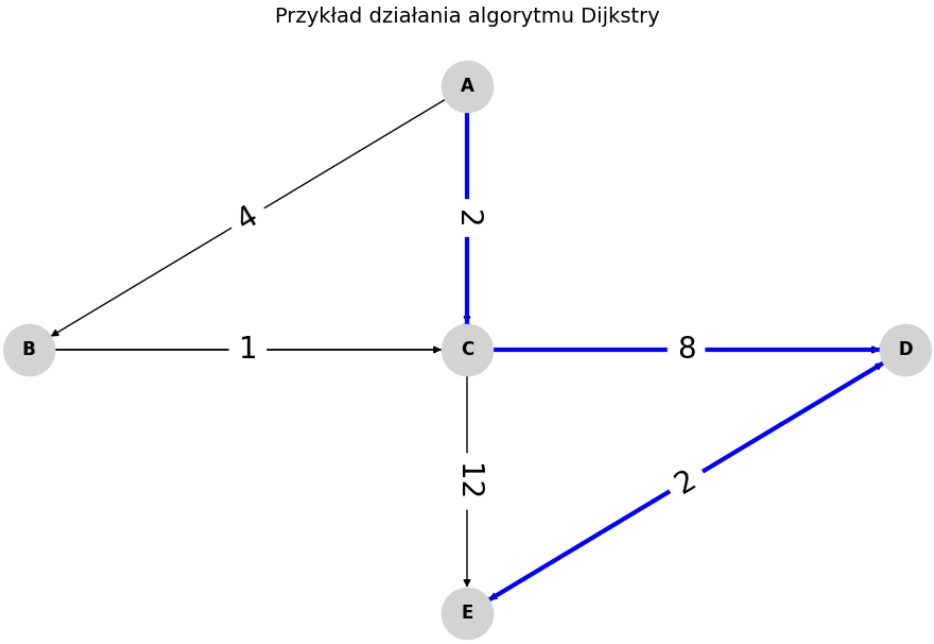
gdzie:

$\boldsymbol{n}$ – liczba wierzchołków,

$\boldsymbol{m}$ – liczba krawędzi.

Algorytm Dijkstry działa poprawnie tylko w przypadku grafów, w których wszystkie krawędzie mają nieujemne wagi. Jeżeli w grafie mogą wystąpić ujemne koszty przejścia, konieczne jest użycie innego rozwiązania – np. algorytmu Bellmana-Forda.

Na rys. 2.8 przedstawiono przykład działania algorytmu Dijkstry w znajdowaniu najkrótszej ścieżki pomiędzy wierzchołkami A – E.




Rys. 2.8. Przykład działania algorytmu Dijkstry

### 2.3.4 Algorytm Bellmana-Forda

W sytuacjach, gdy w grafie mogą występować ujemne wagi krawędzi, stosuje się algorytm Bellmana-Forda czyli klasyczne rozwiązanie umożliwiające wyznaczenie najkrótszej ścieżki w grafach skierowanych oraz ważonych. W odróżnieniu od algorytmu Dijkstry, który działa tylko przy nieujemnych wagach, Bellman-Ford skutecznie radzi sobie także z wartościami ujemnymi [1].

**Zasada działania**

Algorytm rozpoczyna pracę od przypisania każdemu wierzchołkowi wartości nieskończonej, poza punktem startowym (gdzie koszt wynosi zero). Następnie w sposób iteracyjny aktualizuje koszty dojścia do pozostałych wierzchołków, sprawdzając wszystkie dostępne krawędzie. Operacja ta powtarzana jest maksymalnie $(\boldsymbol{n-1})$ razy, gdzie $\boldsymbol{n}$ to liczba wierzchołków w grafie.

Po zakończeniu głównej części działania wykonywane jest dodatkowe przejście po krawędziach, w celu weryfikacji, czy istnieją cykle o ujemnej sumie wag. Ich obecność wskazywałaby na brak jednoznacznego rozwiązania problemu ścieżki minimalnej [1].

**Złożoność obliczeniowa**

- **Czasowa**:
  **O(n * m)**, ponieważ każdy z $(n-1)$ kroków przetwarza wszystkie $m$ krawędzi,
- **Pamięciowa:**
  **O(n)**,

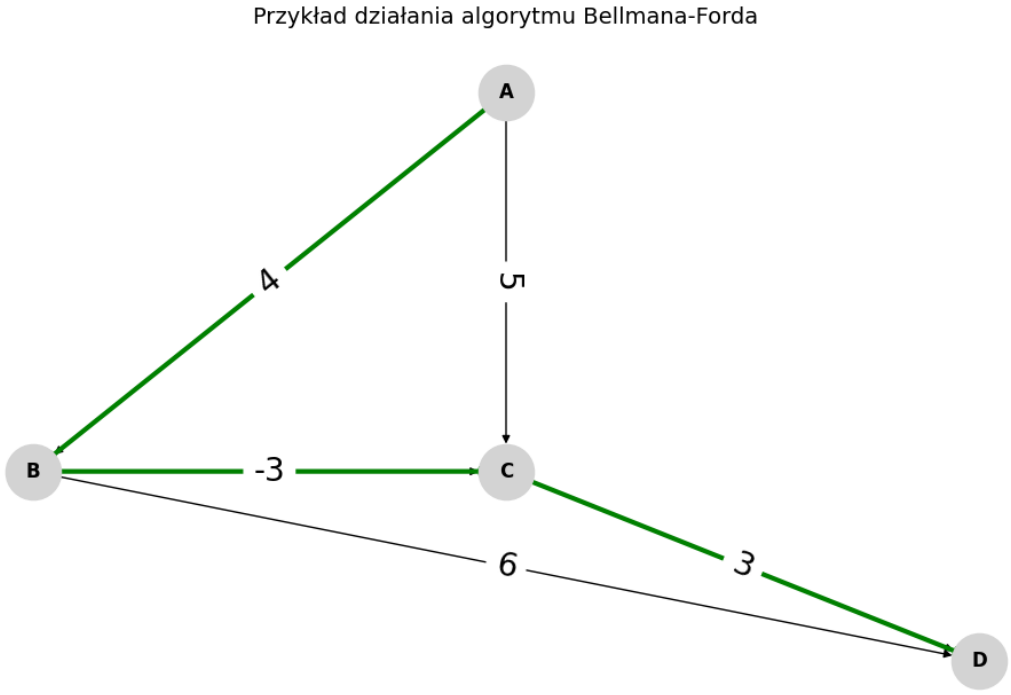
gdzie:

$\boldsymbol{n}-$ liczba wierzchołków,

$\boldsymbol{m}$ – liczba krawędzi.

Choć Bellman-Ford jest wolniejszy niż algorytm Dijkstry, jego zaletą jest uniwersalność i odporność na ujemne wagi krawędzi.

Na rys. 2.9 przedstawiono przykład działania algorytmu Bellmana – Forda w znajdowaniu najkrótszej ścieżki pomiędzy wierzchołkami A – D.




Rys. 2.9. Przykład działania algorytmu Bellmana-Forda

### 2.3.5 Zastosowania Algorytmów wyszukiwania ścieżek

Algorytmy wyznaczania ścieżek w grafach nie są wyłącznie przedmiotem analiz teoretycznych. W rzeczywistości stanowią one trzon wielu systemów informatycznych, wspierając rozwiązania o charakterze praktycznym – od wyznaczania tras, przez komunikację sieciową, po planowanie działań sztucznej inteligencji [1][2][6].

- **Nawigacja i planowanie tras**

W aplikacjach typu GPS czy systemach wspierających zarządzanie flotą, wykorzystywane są algorytmy, których zadaniem jest znalezienie optymalnej trasy pomiędzy punktami. Optymalność może być rozumiana różnie np. jako najkrótszy dystans, minimalny czas przejazdu lub najniższy koszt podróży [1][11].

- **Logistyka i zarządzanie dostawami**

W logistyce grafy służą do modelowania sieci połączeń transportowych np. magazynów, punktów odbioru, tras. Odpowiedni dobór ścieżki pozwala zoptymalizować koszty operacyjne lub czas realizacji zlecenia. Algorytmy grafowe wykorzystywane są także przy planowaniu zaopatrzenia, rozkładzie tras pojazdów i harmonogramowaniu dostaw [3][5].

- **Routing w sieciach komputerowych**

W środowiskach sieciowych jednym z kluczowych wyzwań jest określanie, którą drogą przesłać dane. W tym kontekście stosuje się m.in. algorytm Bellmana-Forda, który potrafi radzić sobie również w przypadku niektórych negatywnych wag, co może odwzorowywać np. niepewność transmisji. W nowocześniejszych sieciach powszechnie korzysta się z bardziej wydajnych algorytmów uwzględniających przepustowość lub opóźnienia [4].

- **Analiza sieci społecznościowych**

Sieci społeczne, modelowane jako grafy, pozwalają badać zależności między użytkownikami. Algorytmy takie jak BFS umożliwiają analizę ścieżek powiązań, obliczanie tzw. odległości społecznej, a także identyfikację osób o największym wpływie. Znajdują tu również zastosowanie metody wykrywania cykli i spójności, co ma znaczenie np. w analizie społecznych baniek informacyjnych [6][9].

- **Gry komputerowe i sztuczna inteligencja**

W grach komputerowych oraz systemach sterowania ruchem agentów (np. robotów czy postaci) wykorzystywane są techniki grafowe do określania możliwych ruchów oraz wyboru najbardziej opłacalnej ścieżki. Algorytmy takie jak A* łączą precyzję z wydajnością i są standardem w środowiskach, gdzie potrzebna jest szybka reakcja systemu na zmieniające się warunki [2][8][11].

- **Planowanie operacyjne w robotyce**

W robotyce mobilnej, grafy mogą reprezentować przestrzeń operacyjną taką jak podłogę hali, sieć korytarzy albo mapę pomieszczeń. Na tej bazie urządzenia autonomiczne (np. roboty przemysłowe czy odkurzacze automatyczne) wyznaczają optymalne ścieżki z uwzględnieniem przeszkód, ryzyka kolizji czy priorytetów zadań [2][5].

### 2.3.6 Funkcja kosztu

W wielu dziedzinach informatyki oraz nauk obliczeniowych, jednym z kluczowych pojęć jest funkcja kosztu – narzędzie umożliwiające ilościową ocenę danego rozwiązania. Polega ona na przypisaniu wartości liczbowej, która może wyrażać długość trasy, czas wykonania zadania, zużycie zasobów, poziom ryzyka lub inne istotne parametry [1][3].

**Zależnie od kontekstu, funkcja kosztu może przyjmować różne postacie, np.:**

- **w grafach** – suma wag krawędzi w ścieżce,
- **w optymalizacji** – funkcja celu do zminimalizowania,
- **w planowaniu** – suma kosztów akcji lub decyzji.

W algorytmach grafowych funkcja kosztu odgrywa rolę wskaźnika, na podstawie którego podejmowane są decyzje o dalszym kierunku przeszukiwania grafu.

**Na przykład:**

- W algorytmie Dijkstry, funkcja kosztu $g(n)$ oznacza sumę wag wszystkich krawędzi pokonanych od wierzchołka startowego do bieżącego, przy czym koszt jest aktualizowany tylko raz dla każdego wierzchołka, jeśli znaleziono tańszą ścieżkę.
- W Bellman-Fordzie, funkcja ta systematycznie wyznacza minimalny koszt dojścia z punktu początkowego do wszystkich pozostałych wierzchołków w grafie, dopuszczając również krawędzie o ujemnych wagach.
  Algorytm ten przyjmuje jako dane wejściowe graf $G = (V, E)$, funkcję wag $w$: $E \rightarrow \mathbb{R}$, oraz wierzchołek początkowy $s \in V$, z którego wyznaczane są najkrótsze ścieżki.

Rola funkcji kosztu jest duża, ponieważ to właśnie od niej zależy, jak sprawnie i poprawnie algorytm znajdzie rozwiązanie. Źle zaprojektowana może prowadzić do nieefektywnej eksploracji, wyboru trasy dalekiej od optymalnej, a nawet błędów logicznych w przebiegu algorytmu.

W rzeczywistych zastosowaniach funkcja kosztu może być rozbudowana i uwzględniać wiele zmiennych jednocześnie, np. czas przejazdu, warunki drogowe, zużycie energii, priorytety użytkownika czy ryzyko opóźnień [2].

## 2.4 Algorytm A*

Algorytm A* (czyt. *„A gwiazdka")* jest jednym z najbardziej znanych i skutecznych algorytmów wykorzystywanych do wyznaczania najkrótszej ścieżki w grafach. Jego zaletą jest połączenie dwóch podejść: deterministycznego znanego z algorytmu Dijkstry, oraz heurystycznego, które wykorzystuje wiedzę szacunkową o problemie. Pozwala to na znacznie efektywniejsze przeszukiwanie przestrzeni stanów w porównaniu do innych algorytmów [1][2].

A* jest algorytmem przeszukiwania informowanego, który prowadzi eksplorację grafu w kierunku celu, ograniczając tym samym liczbę analizowanych węzłów. Jest to możliwe dzięki zastosowaniu heurystyki, która dostarcza szacunkowych informacji o pozostałym koszcie dojścia do celu z danego wierzchołka. A* zawsze znajduje rozwiązanie, jeśli istnieje, oraz zapewnia wyznaczenie najkrótszej możliwej ścieżki, przy odpowiednio dobranej heurystyce [2][5].

Algorytm został opisany w 1968 roku przez Petera E. Harta, Nilsa J. Nilssona i Bertrama Raphaela w pracy zatytułowanej „*A Formal Basis for the Heuristic Determination of Minimum Cost Paths*[2]*"*. Od tego czasu zyskał ogromne znaczenie w wielu dziedzinach informatyki. Jego wszechstronność sprawiły, że A* stał się standardowym narzędziem w rozwiązywaniu problemów przeszukiwania i planowania tras [2][12]

### 2.4.1 Działanie algorytmu

Algorytm A* działa na podstawie funkcji oceny przypisanej każdemu wierzchołkowi grafu, której celem jest określenie, który wierzchołek powinien zostać odwiedzony w kolejnej kolejce przeszukiwania. Funkcja ta łączy koszt rzeczywisty już pokonanej drogi z estymowanym kosztem dojścia do celu [1][2]:

$$\boldsymbol{f(n) = g(n) + h(n)}, \tag{2.1}$$

gdzie**:**

$\boldsymbol{g(n)}$ – znany koszt dojścia z wierzchołka początkowego do bieżącego wierzchołka $\boldsymbol{n}$,

$\boldsymbol{h(n)}$ – heurystyczne oszacowanie kosztu dojścia z $\boldsymbol{n}$ do celu,

$\boldsymbol{f(n)}$ – łączny przewidywany koszt dotarcia do celu przez $\boldsymbol{n}$.

[2] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107

Kluczową rolę w algorytmie odgrywa kolejka priorytetowa, w której przechowywane są otwarte wierzchołki . Każdy z nich jest sortowany względem wartości $\boldsymbol{f(n)}$, a po przetworzeniu przenoszony do zbioru zamkniętego. Proces kontynuuje się, dopóki nie zostanie odnaleziony wierzchołek docelowy lub dopóki kolejka się nie wyczerpie.

Zaletą algorytmu A* jest także jego elastyczność poprzez dobór funkcji heurystycznej można dostosować sposób działania algorytmu do różnych typów problemów i środowisk. Możliwe jest także modyfikowanie funkcji $\boldsymbol{g(n)}$, aby uwzględnić różne metryki kosztów tj. czas, długość, ryzyko, energia itp. [2][5].

### 2.4.2 Heurystyka – definicja i rola

Heurystyka w kontekście algorytmów wyszukiwania to funkcja wspomagająca podejmowanie decyzji na etapie eksploracji grafu, mająca na celu przybliżenie najkorzystniejszego kierunku poszukiwań. Formalnie, heurystyka (oznaczana jako $\boldsymbol{h(n)}$) jest funkcją przypisującą każdemu wierzchołkowi $n$ wartość będącą oszacowaniem kosztu dotarcia z tego wierzchołka do wierzchołka docelowego (celu wyszukiwania) [1][2].

W przeciwieństwie do funkcji rzeczywistego kosztu, która odwzorowuje koszt już pokonanej drogi $g(n)$, heurystyka odnosi się do kosztu przyszłego, jest pewnego rodzaju prognozą, która pomaga algorytmowi ocenić, które kierunki eksploracji są obiecujące. Dzięki temu możliwe jest ograniczenie przestrzeni przeszukiwania i znaczne przyspieszenie procesu znajdowania rozwiązania [2].

Heurystyki w algorytmach takich jak A*, decydują o kolejności eksplorowanych wierzchołków. Ich skuteczność bezpośrednio wpływa na efektywność algorytmu: dobra heurystyka skraca czas wyszukiwania, natomiast zła może spowodować wydłużenie przetwarzania lub nawet brak znalezienia rozwiązania w sensownym czasie [1][5].

Warto zaznaczyć, że heurystyka nie musi być dokładna, zatem wystarczy by była trafna i bezpieczna. Trafność oznacza, że wartości $\boldsymbol{h(n)}$ dobrze odzwierciedlają realne trudności dotarcia do celu, natomiast bezpieczeństwo wiąże się z tzw. dopuszczalnością, czyli warunkiem, że heurystyka nigdy nie przeszacowuje kosztu rzeczywistego. To pozwala zachować optymalność wyników algorytmów tj. A* [2].

Heurystyki wywodzą się z dziedziny sztucznej inteligencji i były pierwotnie stosowane w problemach rozwiązywania łamigłówek, planowania ruchu czy gier strategicznych. Obecnie są

kluczowym elementem inteligentnych metod eksploracji, stosowanych zarówno w zadaniach wyszukiwania ścieżek, jak i w problemach optymalizacji kombinatorycznej oraz uczenia maszynowego [2].

### 2.4.3 Rodzaje heurystyk

Aby heurystyka była poprawna z punktu widzenia algorytmu A*, powinna spełniać określone warunki [2]:

**a) Heurystyka dopuszczalna**

Funkcja $\boldsymbol{h(n)}$ jest dopuszczalna, jeśli nigdy nie przeszacowuje rzeczywistego kosztu dotarcia do celu:

$$\boldsymbol{h(n) \leq h^*(n)}, \tag{2.2}$$

gdzie:

$\boldsymbol{h^*(n)}$ – rzeczywisty najkrótszy koszt od $\boldsymbol{n}$ do celu.

Heurystyki dopuszczalne gwarantują optymalność rozwiązania w algorytmie A*.

**b) Heurystyka spójna (monotoniczna)**

Funkcja $\boldsymbol{h(n)}$ jest spójna, jeśli dla każdej krawędzi $\boldsymbol{(n, n')}$**:**

$$\boldsymbol{h(n) \leq c(n, n') + h(n')}, \tag{2.3}$$

gdzie:

$\boldsymbol{c(n, n')}$ – koszt przejścia między wierzchołkami.

Każda heurystyka spójna jest również dopuszczalna. Spójność heurystyki gwarantuje, że nie trzeba wracać do wcześniej odwiedzonych wierzchołków z lepszym kosztem, co upraszcza implementację A* [11].

**Przykłady funkcji heurystycznych:**

Heurystyki można dostosowywać do konkretnej przestrzeni problemu. Najczęściej spotykane:

- **Odległość Manhattan** (siatki 2D):

$$\boldsymbol{h(n) = |x_n - x_c| + |y_n - y_c|} \tag{2.4}$$

- **Odległość Euklidesowa** (grafy geometryczne):

$$\boldsymbol{h(n) = \sqrt{(x_n - x_c)^2 + (y_n - y_c)^2}} \tag{2.5}$$

- **Odległość Chebysheva** (ruch we wszystkich kierunkach):

$$\boldsymbol{h(n) = max(|x_n - x_c|, |y_n - y_C|)} \tag{2.6}$$

Wybór heurystyki zależy od charakterystyki środowiska, przykładowo w siatkach prostokątnych najlepiej sprawdza się Manhattan, natomiast w grafach z wierzchołkami osadzonymi w przestrzeni 2D lub 3D metryka euklidesowa sprawdza się lepiej [6][11].

### 2.4.4 Pseudokod algorytmu A*

Aby lepiej zobrazować działanie algorytmu A*, poniżej przedstawiono jego uproszczoną wersję w postaci pseudokodu. Algorytm ten wykorzystuje kolejkę priorytetową, w której wybierany jest węzeł o najmniejszej wartości funkcji celu $\boldsymbol{f(n) = g(n) + h(n)}$. Dzięki zastosowaniu heurystyki A* kieruje eksplorację grafu w stronę celu, co pozwala skrócić czas przeszukiwania i ograniczyć liczbę rozwijanych wierzchołków [1][2][11].

```
A*(start, cel)
  open ← {start}
  closed ← ∅
  g[start] ← 0
  f[start] ← h(start)

  while open ≠ ∅:
    n ← węzeł z najmniejszym f(n) w open
    if n == cel:
      return ścieżka od start do cel

    open.remove(n)
    closed.add(n)

    for każdy sąsiad s w n:
```

```
    if s in closed:
      continue
    tentative_g ← g[n] + koszt(n, s)

    if s not in open or tentative_g < g[s]:
      parent[s] ← n
      g[s] ← tentative_g
      f[s] ← g[s] + h(s)
      if s not in open:
        open.add(s)

return brak rozwiązania
```

Listing 2.1. Pseudokod algorytmu A*,
Źródło: [11].

W pseudokodzie:

- open – zbiór wierzchołków otwartych (do przetworzenia),
- closed – zbiór już odwiedzonych wierzchołków,
- g[n] – koszt dojścia od startu do wierzchołka n,
- h(n) – szacowany koszt z n do celu (heurystyka),
- f(n) – suma: realny koszt + szacunek (funkcja celu).

### 2.4.5 Przykład – najkrótsza droga z Rzeszowa do Gdańska

Celem podrozdziału jest pokazanie, w jaki sposób A* działa na realistycznym przykładzie sieci drogowej pomiędzy Rzeszowem a Gdańskiem. Podgraf obejmuje kluczowe miasta znajdujące się na trasie południe–północ Polski. Węzły reprezentują miasta, a krawędzie najważniejsze połączenia drogowe. Koszt krawędzi wyrażony jest w kilometrach, natomiast heurystyka $h(n)$ to odległość w linii prostej do Gdańska.

**Model problemu:**

| Węzeł (Miasto) | $h(n)$ – odległość lotnicza do Gdańska (heurystyka) |
|---|---|
| Rzeszów | 531 km |
| Lublin | 434 km |
| Warszawa | 284 km |
| Olsztyn | 137 km |
| Kraków | 485 km |
| Katowice | 456 km |
| Łódź | 295 km |
| Toruń | 149 km |
| Bydgoszcz | 143 km |
| Gdańsk | 0 km |

Tab. 2.1. Odległości lotnicze pomiędzy miastami

Odległości w linii prostej spełniają warunek dopuszczalności i monotoniczności, więc nadają się na funkcję heurystyczną, ponieważ jest to odległość euklidesowa. Dzięki temu A* gwarantuje znalezienie ścieżki najkrótszej bez potrzeby wstecznej aktualizacji w kolejce otwartej.

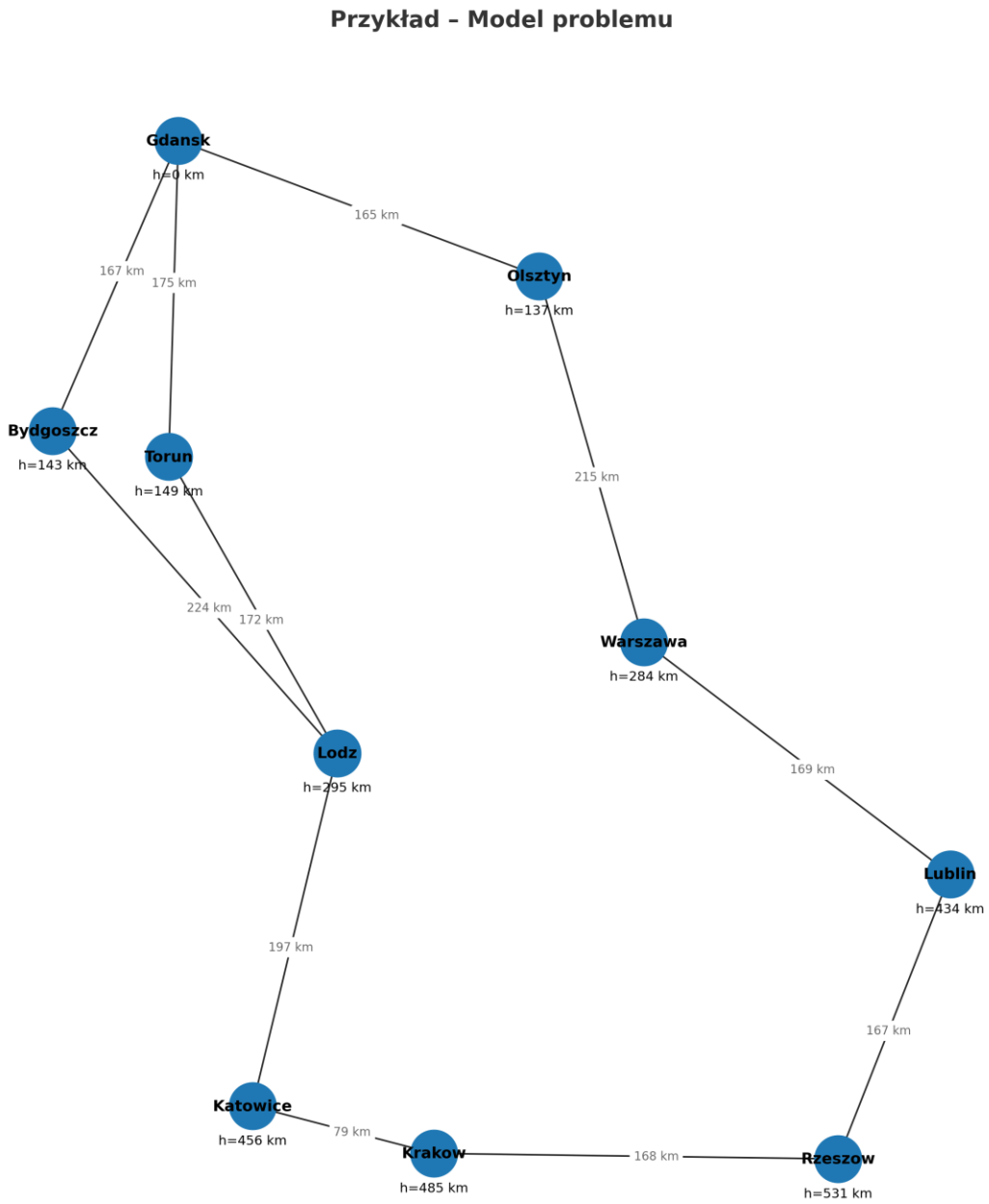
Wybrano tylko najistotniejsze połączenia drogowe ($d$ – wartości kosztu) – te same, które człowiek realistycznie rozważyłby przy planowaniu podróży.

| Węzeł (Miasto) | $d$ – odległość rzeczywista |
|---|---|
| Rzeszów – Lublin | 167 km |
| Rzeszów – Kraków | 168 km |
| Lublin – Warszawa | 169 km |
| Warszawa – Olsztyn | 215 km |
| Olsztyn – Gdańsk | 165 km |
| Kraków – Katowice | 79 km |
| Katowice – Łódź | 197 km |
| Łódź – Toruń | 172 km |
| Toruń – Gdańsk | 175 km |
| Łódź – Bydgoszcz | 224 km |
| Bydgoszcz – Gdańsk | 167 km |

Tab. 2.2. Odległości rzeczywiste pomiędzy miastami

Na rys. 2.10 przedstawiono podgraf głównych dróg między dwoma krańcami Polski (Rzeszowem i Gdańskiem) wraz z naniesioną heurystyką.

- Węzły to miasta (przystanki),
- Krawędzie są opisane rzeczywistym dystansem drogowym w kilometrach (koszt $d = g$ – przyrost).




Rys. 2.10. Graficzna reprezentacja przykładu

**Symulacja krok po kroku Algorytmu A*:**

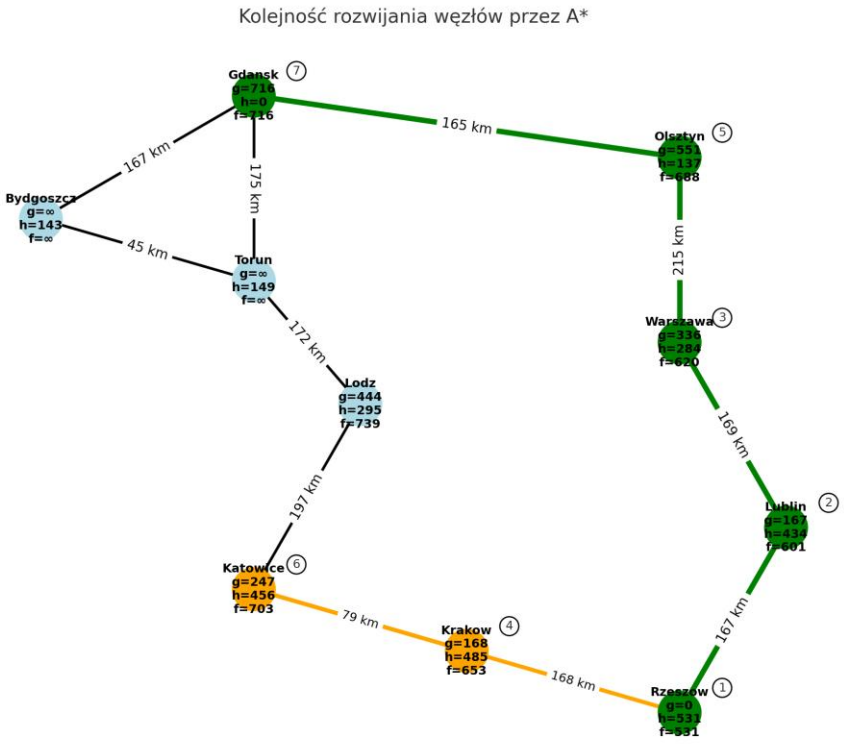
Tab. 2.3. dokumentuje kolejne kroki algorytmu A:

| Krok | Węzeł (Miasto) | $g$ (dotychczasowy koszt) | $h$ | $f = g + h$ | Otwarte alternatywy (posortowane po $f$) |
|---|---|---|---|---|---|
| 0 | Rzeszów | 0 | 531 | 531 | Lublin (601), Kraków (653) |
| 1 | Lublin | 167 | 434 | 601 | Warszawa (620), Kraków (653) |
| 2 | Warszawa | 336 | 284 | 620 | Olsztyn (688), Kraków (653) |
| 3 | Kraków (tańszy $f$ niż Olsztyn dlatego algorytm „nawraca”) | 168 | 485 | 653 | Katowice (703), Olsztyn (688) |
| 4 | Olsztyn | 551 | 137 | 688 | Katowice (703), Gdańsk (715) |
| 5 | Katowice (wciąż tańsze niż Gdańsk) | 247 | 456 | 703 | Łódź (998) |
| 6 | Gdańsk (cel) | 716 | 0 | 716 | - |

Tab. 2.3. Tabela symulacji kroków

Rys. 2.11 numeruje miasta w kolejności ich ekspansji (pobrania z kolejki otwartej). Zgodnie ze wzorem (2.1):

$$f(n) = g(n) + h(n),$$




Rys. 2.11. Kolejność rozwijania węzłów przez algorytm A*

A* za każdym razem wybiera węzeł o najmniejszej wartości $\boldsymbol{f}$. Krótki komentarz do poszczególnych kroków w tab. 2.4:

| Krok | Węzeł (Miasto) | Co się dzieje i dlaczego? |
|---|---|---|
| 0 | Rzeszów | Start, $g = 0$, do kolejki otwartej trafiają sąsiedzi: Lublin i Kraków. |
| 1 | Lublin | Ma niższe $\boldsymbol{f}$ niż Kraków, więc zostaje rozwinięte pierwsze. |
| 2 | Warszawa | Znów najniższe $\boldsymbol{f}$. Warto zwrócić uwagę, że A* nie patrzy na liczbę krawędzi, tylko na sumę $\boldsymbol{g + h}$ |
| 3 | Kraków | Choć jest na południu, pojawia się w otwartej z $\boldsymbol{f < f(Olsztyn)}$, więc algorytm „nawraca", co dowodzi bezstronności selekcji $\boldsymbol{f}$. |
| 4 | Olsztyn | Tutaj $\boldsymbol{f}$ okazuje się niższe niż wszystkie pozostałe alternatywy po rozwinięciu Krakowa. |
| 5 | Katowice | Wciąż tańsze (w sensie $\boldsymbol{f}$) niż osiągnięcie Gdańska wprost z Olsztyna – stąd kolejne zejście w bok. |
| 6 | Gdańsk | Pierwszy raz, gdy węzeł celu trafia na czoło kolejki – kończy, bo heurystyka spełnia warunek monotoniczności. |

Tab. 2.4. Tabela z objaśnieniem kroków.

**Rezultat:**

Zielona ścieżka na rys. 2.11 to droga znaleziona przez algorytm A*.

- **Znaleziony koszt:** $g = 716$ km
- **Ścieżka:** Rzeszów → Lublin → Warszawa → Olsztyn → Gdańsk

Skoro heurystyka jest dopuszczalna i monotoniczna, a węzeł celu został rozwinięty przy pierwszym wyjęciu z otwartej, ścieżka spełnia warunek optymalności.

### 2.4.6 Wyróżniki praktyczne algorytmu A*

Choć A* znajduje zastosowanie w podobnych obszarach co inne algorytmy wyznaczania ścieżek, jego praktyczna przewaga wynika z wykorzystania heurystyki, która umożliwia [1][2]:

- dynamiczne ograniczanie przestrzeni przeszukiwania,
- szybsze znajdowanie optymalnych ścieżek,
- dostosowanie do różnych kryteriów kosztu.

W systemach takich jak [2][11]:

- nawigacja GPS – A* może omijać drogi zamknięte lub uwzględniać czas przejazdu,
- robotyka – pozwala planować ścieżki w czasie rzeczywistym przy zmieniającym się środowisku,
- gry komputerowe – daje bardziej *„inteligentne”* zachowania postaci, które nie tylko idą najkrótszą drogą, ale też np. omijają wrogów lub strefy zagrożenia.

W odróżnieniu od np. algorytmu Dijkstry, który przeszukuje graf równomiernie, A* skupia się na kierunku celu, co w praktyce oznacza mniejsze zużycie zasobów obliczeniowych oraz szybsze czasy odpowiedzi w systemach czasu rzeczywistego [11].

# 3. Część praktyczna

W tej części pracy skoncentrowano się na empirycznej analizie działania wybranych algorytmów grafowych, takich jak BFS, DFS, Dijkstra oraz A*. Celem było porównanie ich skuteczności i efektywności w różnych warunkach, przy wykorzystaniu środowisk o odmiennych strukturach topologicznych. Uwzględniono zarówno algorytmy deterministyczne, jak i heurystyczne, analizując ich zachowanie nie tylko pod względem poprawności, ale także czasu działania, zużycia pamięci i długości wyznaczonej ścieżki.

Szczegółowy opis środowisk testowych, zastosowanych narzędzi oraz założeń eksperymentalnych zawarty został w podrozdziale 3.1.

## 3.1 Środowisko testowe i użyte narzędzia

Wszystkie implementacje algorytmów oraz testy porównawcze przeprowadzono z wykorzystaniem języka Python, w środowisku programistycznym Visual Studio Code. Wybór ten wynikał z wysokiej czytelności składni Pythona oraz dostępności szerokiej bazy bibliotek, wspierających przetwarzanie danych, analizę grafów i wizualizację wyników.

Główne pakiety zastosowane do obsługi i prezentacji danych:

- **networkx** – do reprezentowania grafów oraz przeprowadzania operacji na ich strukturze [13],
- **matplotlib** – do wizualizacji ścieżek, układu wierzchołków i przeszkód w przestrzeni 2D [14],
- **tracemalloc** – służący do pomiaru zużycia pamięci dynamicznej przez konkretne fragmenty kodu [15],
- **random, time** – standardowe biblioteki Pythona wykorzystywane m.in. do losowego generowania danych oraz pomiarów czasów działania funkcji [16],
- **pygame** – popularne narzędzie w języku Python przeznaczone do tworzenia aplikacji multimedialnych i gier 2D [17],
- **queue –** umożliwia realizacje kolejek priorytetowych [18].

Dodatkowo, w przypadku scenariusza wykorzystującego mapę rzeczywistą Rzeszowa, zastosowano bibliotekę osmnx, która pozwala na pobieranie rzeczywistych danych sieci drogowej z serwisu OpenStreetMap i konwertowanie ich do struktury grafowej. Dzięki temu możliwe było testowanie działania algorytmów w bardziej realistycznym środowisku miejskim, uwzględniającym rzeczywistą topologię ulic i skrzyżowań [19].

Dla każdego ze środowisk testowych (graf ważony, siatka 2D, mapa miasta) przygotowano oddzielne pliki skryptowe, które umożliwiały:

- ręczne lub losowe generowanie danych wejściowych,
- uruchamianie wybranych algorytmów (BFS, DFS, Dijkstra, A* z różnymi heurystykami),
- pomiar metryk takich jak czas wykonania, liczba odwiedzonych wierzchołków, długość ścieżki oraz zużycie pamięci operacyjnej.

Każdy test był wykonywany w identycznych warunkach technicznych. Eksperymenty przeprowadzono na komputerze z procesorem AMD Ryzen 5 3550H, kartą graficzną NVIDIA GeForce GTX 1650, 8 GB pamięci RAM DDR4 oraz dyskiem SSD o pojemności 500 GB. Takie parametry odpowiadają typowemu środowisku pracy użytkownika końcowego i umożliwiają płynne testowanie algorytmów bez potrzeby korzystania ze sprzętu o wysokiej wydajności obliczeniowej.

Dzięki przyjętemu podejściu możliwe było nie tylko zautomatyzowane zbieranie wyników, ale także ich wygodna prezentacja – zarówno liczbowo, jak i w formie wizualizacji tras, co ułatwia porównania oraz interpretację różnic w zachowaniu poszczególnych algorytmów.

## 3.2 Implementacja w grafie ważonym

Pierwszym środowiskiem testowym był graf ważony, wygenerowany losowo za pomocą modelu *Erdos-Renyi*. Wygenerowany graf miał zapewnioną spójność (czyli istnienie ścieżki między dowolnymi dwoma wierzchołkami), a wagi krawędzi losowano z przedziału od 1 do 5. Celem było przetestowanie i porównanie kilku podstawowych algorytmów grafowych:

- BFS (Breadth-First Search),
- DFS (Depth-First Search),
- Dijkstra,
- A* z trzema heurystykami: indeksową, euklidesową i manhattan.

### 3.2.1 Fragmenty kodu algorytmów oraz funkcji do wizualizacji

W poniższych podrozdziałach zaprezentowano najważniejsze fragmenty kodu:

#### 3.2.1.1 Generowanie grafu ważonego

W pierwszej kolejności przedstawiono funkcję odpowiedzialną za generowanie spójnego grafu losowego (Listing 3.1)

```python
def generuj_graf(n=30, gestosc=0.1):
    G = nx.erdos_renyi_graph(n=n, p=gestosc)
    while not nx.is_connected(G):
        G = nx.erdos_renyi_graph(n=n, p=gestosc)
    for (u, v) in G.edges():
        G[u][v]['weight'] = random.randint(1, 5)
    return G
```

Listing 3.1. Generowanie grafu spójnego na podstawie modelu Erdos-Rényi

Funkcja generuj_graf służy do tworzenia losowego grafu nieskierowanego na podstawie modelu Erdos-Rényi. Parametry wejściowe umożliwiają określenie liczby wierzchołków oraz gęstości grafu, czyli prawdopodobieństwa wystąpienia krawędzi między każdą parą wierzchołków. Aby zapewnić spójność grafu (istnienie połączenia między wszystkimi wierzchołkami), generowanie jest powtarzane aż do uzyskania struktury spełniającej ten warunek. Każdej krawędzi przypisywana jest losowa waga z zakresu od 1 do 5, co pozwala na testowanie algorytmów na grafach ważonych.

#### 3.2.1.2 Algorytm BFS

Poniższa funkcja (Listing 3.2) odpowiada za przeszukiwanie grafu metodą BFS, czyli *„wszerz”*. Algorytm zaczyna od zadanego wierzchołka i porusza się po grafie warstwowo – najpierw analizując najbliższych sąsiadów, a następnie kolejne poziomy połączeń. Pozwala to na znalezienie ścieżki o najmniejszej liczbie kroków

```python
def bfs(graf, start, cel):
    kolejka = Queue()
    kolejka.put(start)
    odwiedzone = {start}
    sciezka = {}

    while not kolejka.empty():
        obecny = kolejka.get()
        if obecny == cel:
            droga = odwiedzony_na_sciezke(sciezka, start, cel)
            koszt = sum(graf[droga[i]][droga[i+1]]['weight'] for i in
range(len(droga) - 1))
            return droga, odwiedzone, koszt
        for sasiad in graf.neighbors(obecny):
            if sasiad not in odwiedzone:
                odwiedzone.add(sasiad)
                kolejka.put(sasiad)
                sciezka[sasiad] = obecny
    return [], odwiedzone, 0
```

Listing 3.2. Implementacja BFS

Funkcja bfs realizuje przeszukiwanie grafu wszerz. Wykorzystuje do tego kolejkę do przechowywania wierzchołków oczekujących na odwiedzenie, a także zbiór do oznaczania już odwiedzonych. Dodatkowo prowadzony jest słownik, który pozwala odtworzyć drogę od wierzchołka początkowego do celu – dla każdego węzła zapisywany jest jego poprzednik. Gdy znajdzie się wierzchołek docelowy, ścieżka jest rekonstruowana, a jej koszt wyliczany jako suma wag krawędzi, które ją tworzą. W przypadku braku rozwiązania funkcja zwraca pusty wynik.

#### 3.2.1.3 Algorytm DFS

Fragment kodu znajdujący się poniżej przedstawia implementację algorytmu DFS, który służy do przeszukiwania grafu w głąb. W przeciwieństwie do metody BFS, tutaj eksploracja najpierw zagłębia się możliwie jak najdalej w kierunku jednego z sąsiadów, zanim powróci do wcześniejszych poziomów. Taki sposób działania sprawia, że DFS lepiej nadaje się do przeszukiwania rozległych struktur o dużej głębokości.

```
def dfs(graf, start, cel):
    stos = [start]
    odwiedzone = {start}
    sciezka = {}

    while stos:
        obecny = stos.pop()
        if obecny == cel:
            droga = odwiedzony_na_sciezke(sciezka, start, cel)
            koszt = sum(graf[droga[i]][droga[i+1]]['weight'] for i in
range(len(droga) - 1))
            return droga, odwiedzone, koszt
        for sasiad in graf.neighbors(obecny):
            if sasiad not in odwiedzone:
                odwiedzone.add(sasiad)
                stos.append(sasiad)
                sciezka[sasiad] = obecny
    return [], odwiedzone, 0
```

Listing 3.3. Implementacja DFS

Algorytm rozpoczyna pracę od zadanego wierzchołka początkowego, który zostaje umieszczony na stosie. W każdej iteracji zdejmowany jest z niego ostatnio dodany element, który następnie podlega sprawdzeniu. Jeżeli bieżący wierzchołek odpowiada celowi, tworzona jest ścieżka na podstawie słownika odwiedzin, a jej koszt liczony jest jako suma wag poszczególnych krawędzi. Jeśli bieżący wierzchołek nie jest celem, algorytm przechodzi do jego sąsiadów i dodaje do stosu tych, którzy nie zostali jeszcze odwiedzeni. Dzięki temu zachowane jest typowe dla DFS zachowanie rekurencyjne, odwzorowane w sposób iteracyjny.

#### 3.2.1.4 Algorytm Dijkstry

Funkcja przedstawiona w Listingu 3.3 implementuje algorytm Dijkstry, służący do wyznaczania najkrótszej ścieżki w grafie ważonym, w którym wszystkie wagi krawędzi są nieujemne. Algorytm opiera się na klasycznym podejściu zaproponowanym przez E. Dijkstrę, z wykorzystaniem kolejki priorytetowej do selekcji wierzchołków o najniższym koszcie [1].

```
def dijkstra(graf, start, cel):
    kolejka = PriorityQueue()
    kolejka.put((0, start))
    koszt = {start: 0}
    odwiedzone = {start}
    sciezka = {}

    while not kolejka.empty():
        dystans, obecny = kolejka.get()
        if obecny == cel:
            droga = odwiedzony_na_sciezke(sciezka, start, cel)
            return droga, odwiedzone, koszt[cel]
        for sasiad in graf.neighbors(obecny):
            waga = graf[obecny][sasiad]['weight']
            nowy_koszt = dystans + waga
            if sasiad not in koszt or nowy_koszt < koszt[sasiad]:
                koszt[sasiad] = nowy_koszt
                kolejka.put((nowy_koszt, sasiad))
                sciezka[sasiad] = obecny
                odwiedzone.add(sasiad)
    return [], odwiedzone, 0
```

Listing 3.4. Implementacja algorytmu Dijkstry

W pierwszej kolejności do kolejki dodawany jest wierzchołek początkowy z kosztem równym zero. W trakcie przeszukiwania algorytm każdorazowo wybiera wierzchołek o najniższej wartości kosztu i aktualizuje najkrótsze znane ścieżki do jego sąsiadów. Jeśli nowo obliczony koszt dojścia do któregoś z sąsiadów jest mniejszy od dotychczasowego, to koszt zostaje zaktualizowany, a wierzchołek trafia ponownie do kolejki. Proces ten trwa aż do momentu odnalezienia ścieżki do celu lub wyczerpania wszystkich możliwych ścieżek.

Dodatkowo, funkcja zapisuje ślad odwiedzanych wierzchołków oraz tworzy strukturę umożliwiającą późniejsze odtworzenie optymalnej trasy. W końcowym etapie zwracana jest gotowa ścieżka, zbiór odwiedzonych wierzchołków oraz całkowity koszt trasy od startu do celu. Algorytm ten jest deterministyczny i zawsze znajduje rozwiązanie optymalne w kontekście sumy wag.

#### 3.2.1.5 Algorytm A*

Poniższy fragment kodu przedstawia implementację algorytmu A* czyli jednego z najbardziej efektywnych podejść do wyznaczania najkrótszej ścieżki w grafach. W odróżnieniu od metod takich jak BFS czy Dijkstra, A* wykorzystuje dodatkową funkcję heurystyczną, która pozwala oszacować koszt dotarcia do celu i dzięki temu ogranicza niepotrzebne eksplorowanie wierzchołków.

```
def astar(graf, start, cel, heurystyka):
    kolejka = PriorityQueue()
    kolejka.put((0, start))
    koszt = {start: 0}
    odwiedzone = {start}
    sciezka = {}

    while not kolejka.empty():
        _, obecny = kolejka.get()
        if obecny == cel:
            droga = odwiedzony_na_sciezke(sciezka, start, cel)
            return droga, odwiedzone, koszt[cel]
        for sasiad in graf.neighbors(obecny):
            waga = graf[obecny][sasiad]['weight']
            nowy_koszt = koszt[obecny] + waga
            if sasiad not in koszt or nowy_koszt < koszt[sasiad]:
                koszt[sasiad] = nowy_koszt
                priorytet = nowy_koszt + heurystyka(sasiad, cel)
                kolejka.put((priorytet, sasiad))
                sciezka[sasiad] = obecny
                odwiedzone.add(sasiad)
    return [], odwiedzone, 0
```

Listing 3.5. Implementacja algorytmu A*

Na wstępie tworzona jest kolejka priorytetowa, w której umieszcza się wierzchołek startowy z priorytetem równym zeru. W każdej iteracji pobierany jest z niej wierzchołek o najniższej łącznej ocenie kosztu ($\boldsymbol{f = g + h}$), gdzie $g$ oznacza koszt dotarcia z punktu startowego, a $h$ to wartość zwracana przez funkcję heurystyczną – szacująca pozostały dystans do celu. Jeśli analizowany wierzchołek nie prowadzi do celu, jego sąsiedzi są oceniani i ewentualnie dodawani do kolejki. Struktura sciezka pozwala później odtworzyć drogę, a zbiór odwiedzone rejestruje analizowane wierzchołki. Dzięki połączeniu rzeczywistego kosztu z przewidywaniem kierunku, algorytm A* działa szybciej i niż podejścia nieuwzględniające heurystyki.

#### 3.2.1.6 Funkcje heurystyki dla algorytmu A*

Listing 3.6–3.8 zawiera definicje trzech różnych funkcji heurystycznych wykorzystywanych przez algorytm A*. Każda z nich odpowiada za szacowanie pozostałego dystansu z danego wierzchołka do celu, co wpływa na sposób eksplorowania grafu i skuteczność całego algorytmu.

```
def heurystyka_indeksowa(a, b):
    return abs(a - b)
```

Listing 3.6. Heurystyka indeksowa

Pierwsza z heurystyk (indeksowa) jest bardzo uproszczona i opiera się wyłącznie na różnicy identyfikatorów wierzchołków. Jest to celowo niedopasowana funkcja, która może służyć jako przykład źle dobranej heurystyki – jej działanie nie ma związku z rzeczywistą odległością w grafie.

```
def heurystyka_euklidesowa(pos):
    def h(a, b):
        x1, y1 = pos[a]
        x2, y2 = pos[b]
        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)
    return h
```

Listing 3.7. Heurystyka oparta na długości Euklidesowej

Heurystyka euklidesowa (funkcja heurystyka_euklidesowa) oblicza klasyczną odległość w przestrzeni dwuwymiarowej, zakładając, że wierzchołki mają przypisane współrzędne. Jest ona dokładniejsza, ale wymaga dostępu do pozycji wierzchołków.

```
def heurystyka_manhattan(pos):
    def h(a, b):
        x1, y1 = pos[a]
        x2, y2 = pos[b]
        return abs(x1 - x2) + abs(y1 - y2)
    return h
```

Listing 3.8. Heurystyka oparta na odległości manhattan

Heurystyka Manhattan (heurystyka_manhattan) mierzy odległość jako sumę różnic współrzędnych zatem sprawdza się dobrze, gdzie poruszanie się odbywa się tylko w pionie i poziomie. Wybór odpowiedniej funkcji ma kluczowe znaczenie dla efektywności algorytmu A*, ponieważ wpływa na to, w jakim kierunku będzie eksplorowany graf.

#### 3.2.1.7 Funkcja obliczania parametrów i wizualizacji

Poniższa funkcja odpowiada za uruchomienie algorytmu przeszukiwania oraz zebranie i wizualizację wyników jego działania w formie graficznej.

```
def zmierz_i_rysuj(graf, algorytm, nazwa, pos, start, cel,
subplot_index):
    tracemalloc.start()
    start_czas = time.perf_counter()
    sciezka, odwiedzone, koszt = algorytm(graf, start, cel)
    koniec_czas = time.perf_counter()
    pamiec, _ = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    plt.subplot(2, 3, subplot_index)
    nx.draw(graf, pos, node_color='lightgrey', edge_color='black',
with_labels=True, node_size=500, font_size=8)

    if sciezka:
        krawedzie = list(zip(sciezka[:-1], sciezka[1:]))
        nx.draw_networkx_edges(graf, pos, edgelist=krawedzie,
edge_color='green', width=2.5)
        nx.draw_networkx_nodes(graf, pos, nodelist=sciezka,
node_color='limegreen', node_size=500)
        etykiety_krawedzi = nx.get_edge_attributes(graf, 'weight')
        nx.draw_networkx_edge_labels(graf, pos,
edge_labels=etykiety_krawedzi, font_size=6)

    plt.title(
        f"{nazwa}\nCzas: {koniec_czas - start_czas:.4f}s | Pamięć:
{pamiec / 1024:.1f}KB | "
        f"Wierzchołki: {len(odwiedzone)} | Koszt: {koszt}"
    )
```

Listing 3.9. Funkcja pomiaru i wizualizacji wyników działania algorytmu

Funkcja ta umożliwia jednoczesne uruchomienie wybranego algorytmu oraz pomiar jego efektywności. Do rejestracji zużycia pamięci użyto modułu tracemalloc, a czas działania mierzony jest z użyciem time.perf_counter(), co zapewnia wysoką precyzję. Zebrane dane – czas, zużycie pamięci, liczba odwiedzonych wierzchołków i koszt ścieżki – prezentowane są w tytule wykresu. Sama ścieżka wizualizowana jest na grafie w kolorze zielonym, a pozostałe wierzchołki i krawędzie mają neutralne kolory. Dzięki temu można szybko ocenić przebieg oraz efektywność działania danego algorytmu na konkretnym układzie grafu.

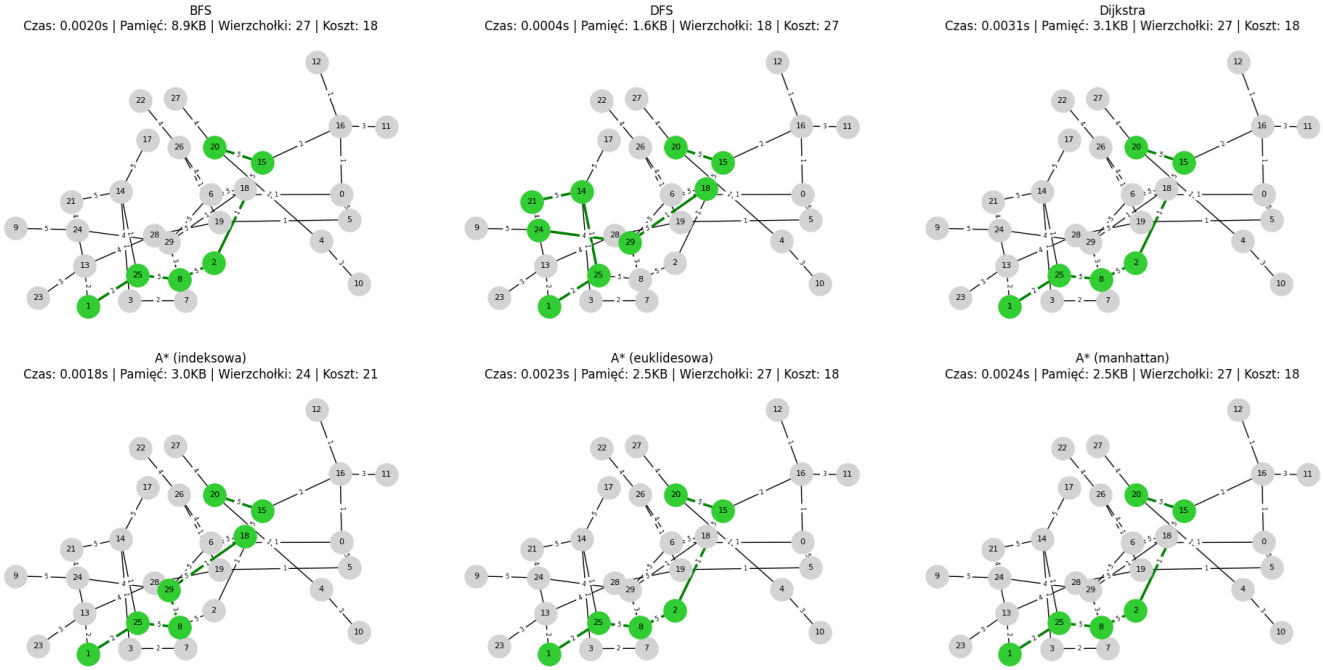
### 3.2.2 Przykładowy przebieg działania i wizualizacja

W celu wizualnej prezentacji działania poszczególnych algorytmów, wykonano serię testów dla różnych losowo wygenerowanych grafów ważonych. Dla każdego przypadku ustalono liczbę wierzchołków, współczynnik gęstości oraz konkretny wierzchołek początkowy i docelowy. Efekty działania zaprezentowano na poniższych ilustracjach (rys. 3.1 – rys. 3.3)

**Otrzymane wizualizacje**

1) Dla losowego grafu o parametrach:

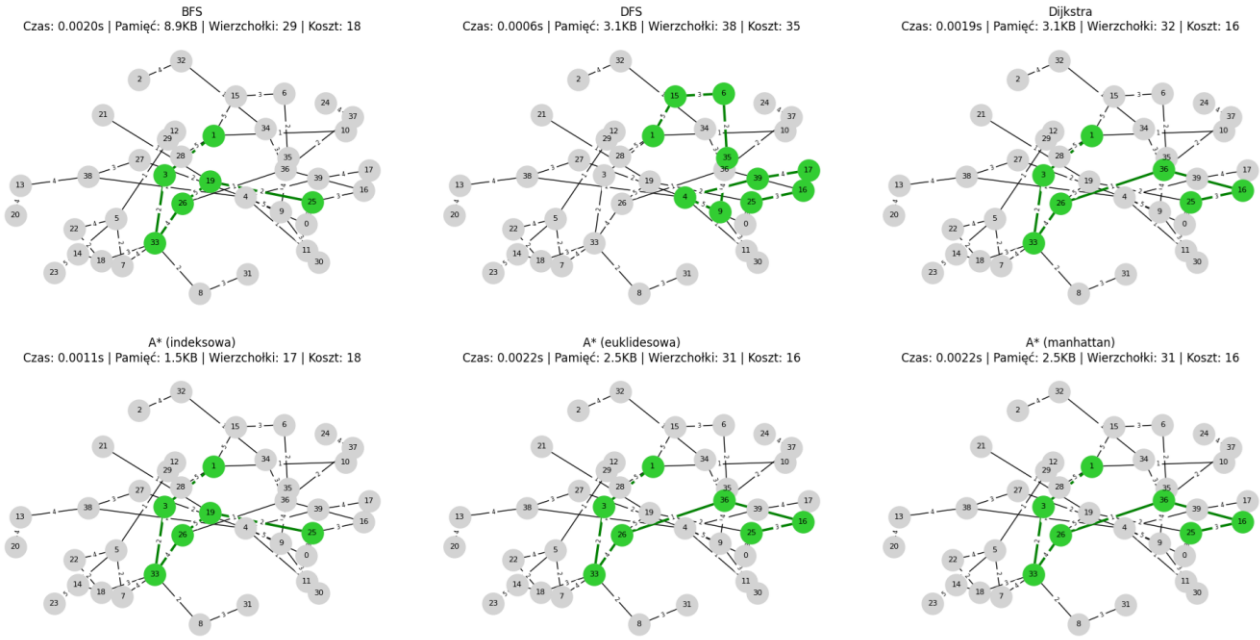
- 30 wierzchołków,
- gęstość = 0.05,
- ścieżka od 1 do 20
- wagi krawędzi losowe (zakres od 1 do 5)




Rys. 3.1. Wyniki działania algorytmów dla grafu (30 wierzchołków, gęstość 0.05, ścieżka od 1 do 20)

2) Dla losowego grafu o parametrach:

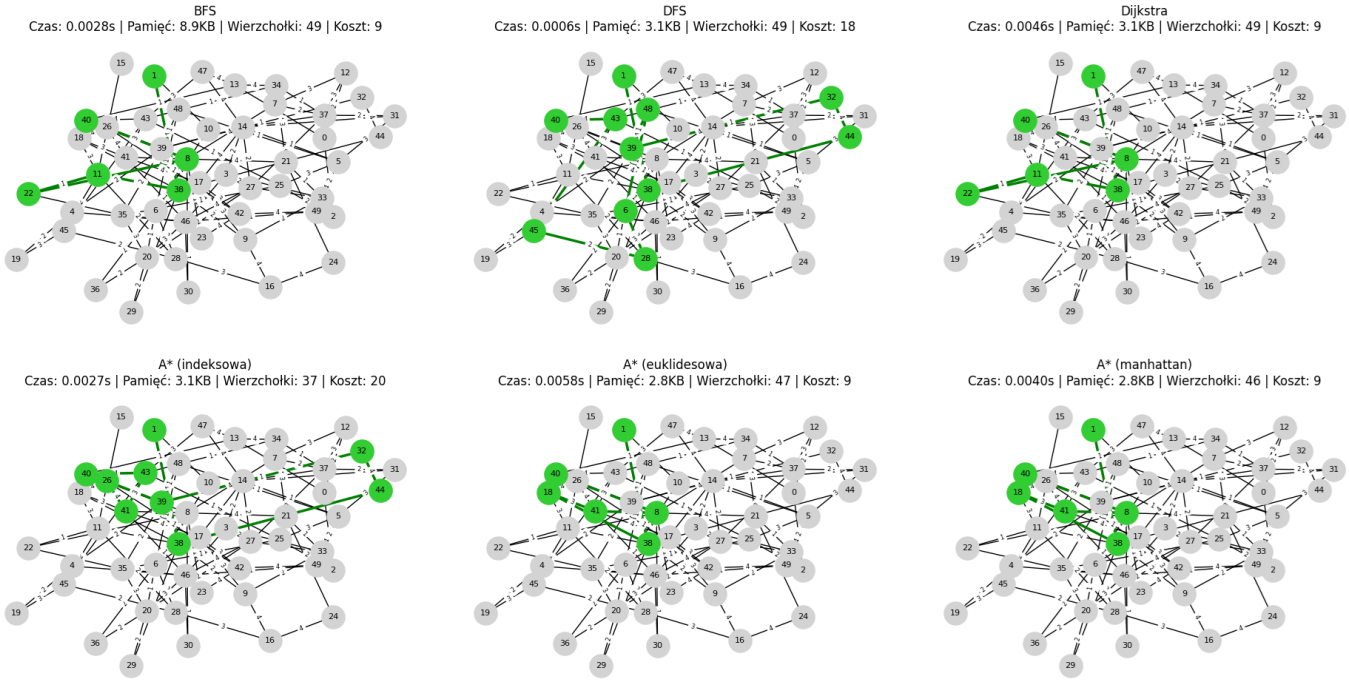
- 40 wierzchołków,
- gęstość = 0.04,
- ścieżka od 1 do 25
- wagi krawędzi losowe (zakres od 1 do 5)




Rys. 3.2. Wyniki działania algorytmów dla grafu (40 wierzchołków, gęstość 0.04, ścieżka od 1 do 25)

3) Dla losowego grafu o parametrach:

- 50 wierzchołków,
- gęstość = 0.07,
- ścieżka od 1 do 40
- wagi krawędzi losowe (zakres od 1 do 3)




Rys. 3.3. Wyniki działania algorytmów dla grafu (50 wierzchołków, gęstość 0.07, ścieżka od 1 do 40)

### 3.2.3 Tabele porównawcze wyników dla grafów ważonych

Poniżej w tab. 3.1 – 3.3 przedstawiono zestawienie wyników działania algorytmów dla trzech różnych grafów testowych z rys. 3.1 – 3.3. Tabele zawierają cztery kluczowe parametry:

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Koszt |
|---|---|---|---|---|---|
| 1 | BFS | 0.0020 | 8.9 | 27 | 18 |
| 2 | DFS | 0.0004 | 1.6 | 18 | 27 |
| 3 | Dijkstra | 0.0031 | 3.1 | 27 | 18 |
| 4 | A* (heurystyka indeksowa) | 0.0018 | 3.0 | 24 | 21 |
| 5 | A* (heurystyka euklidesowa) | 0.0023 | 2.5 | 27 | 18 |
| 6 | A* (heurystyka manhattan) | 0.0024 | 2.5 | 27 | 18 |

Tab. 3.1. Tabela porównawcza działania algorytmów dla testu pierwszego (30 wierzchołków, gęstość 0.05, ścieżka od 1 do 20, wagi 1-5)

Na podstawie tab. 3.1 (test z rys. 3.1) można zauważyć, że BFS, Dijkstra oraz A* z heurystyką euklidesową i manhattan znalazły ścieżki o koszcie 18. DFS, pomimo szybkiego działania, przeszukiwał znacznie mniej optymalnie a znaleziona ścieżka miała wyższy koszt (27). Heurystyka indeksowa w A* spowodowała wybór mniej korzystnej trasy o koszcie 21. Warto także zwrócić uwagę na różnice w liczbie odwiedzonych wierzchołków oraz zużyciu pamięci. A* z dobrze dobraną heurystyką działał bardziej ekonomicznie niż algorytmy przeszukiwania bez heurystyk.

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Koszt |
|---|---|---|---|---|---|
| 1 | BFS | 0.0020 | 8.9 | 29 | 18 |
| 2 | DFS | 0.0006 | 3.1 | 38 | 35 |
| 3 | Dijkstra | 0.0019 | 3.1 | 32 | 16 |
| 4 | A* (heurystyka indeksowa) | 0.0011 | 1.5 | 17 | 18 |
| 5 | A* (heurystyka euklidesowa) | 0.0022 | 2.5 | 31 | 16 |
| 6 | A* (heurystyka manhattan) | 0.0022 | 2.5 | 31 | 16 |

Tab. 3.2. Tabela porównawcza działania algorytmów dla testu drugiego (40 wierzchołków, gęstość 0.04, ścieżka od 1 do 25, wagi 1-5)

Tab. 3.2. reprezentująca wyniki z drugiego testu pokazuje, że algorytmy Dijkstra oraz A* z heurystyką euklidesową i manhattan wyznaczyły ścieżkę o najniższym koszcie (16), przy umiarkowanej liczbie odwiedzeń. BFS również znalazł poprawną trasę (koszt 18), lecz przeszukiwał szerzej. DFS ponownie był najmniej efektywny – zarówno pod względem kosztu (35), jak i liczby odwiedzonych wierzchołków. A* z heurystyką indeksową zakończył działanie na trasie o koszcie 18, jednak odwiedził znacznie mniej węzłów niż pozostałe metody. Widoczne są więc różnice nie tylko w jakości trasy, ale też w sposobie eksploracji grafu.

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Koszt |
|---|---|---|---|---|---|
| 1 | BFS | 0.0028 | 8.9 | 49 | 9 |
| 2 | DFS | 0.0006 | 3.1 | 49 | 18 |
| 3 | Dijkstra | 0.0046 | 3.1 | 49 | 9 |
| 4 | A* (heurystyka indeksowa) | 0.0027 | 3.1 | 37 | 20 |
| 5 | A* (heurystyka euklidesowa) | 0.0058 | 2.8 | 47 | 9 |
| 6 | A* (heurystyka manhattan) | 0.0040 | 2.8 | 46 | 9 |

Tab. 3.3. Tabela porównawcza działania algorytmów dla testu trzeciego (50 wierzchołków, gęstość 0.07, ścieżka od 1 do 40, wagi 1-3)

Tab. 3.3, przedstawiająca dane z ostatniego testu, wskazuje na zbieżność wyników kilku algorytmów tj. BFS, Dijkstra oraz A* z heurystyką euklidesową i manhattan. Wyznaczyły one trasę o identycznym koszcie (9), co świadczy o ich wysokiej skuteczności w gęstym grafie o niewielkich wagach. DFS ponownie wypadł najsłabiej zarówno pod względem kosztu, jak i efektywności przeszukiwania - przeszukał cały graf i wygenerował trasę o koszcie aż 18. Z kolei A* z heurystyką indeksową wyznaczył trasę o koszcie 20, co pokazuje, że niedopasowana heurystyka może prowadzić do pogorszenia wyników. Warto też zauważyć, że pomimo zbliżonych wyników końcowych, poszczególne algorytmy różniły się istotnie pod względem liczby odwiedzonych wierzchołków oraz zużycia pamięci.

### 3.2.4 Wnioski z analizy porównawczej na grafie ważonym

Zestawienie wyników uzyskanych w przeprowadzonych testach pozwala zauważyć pewne prawidłowości w działaniu poszczególnych algorytmów. Szczególnie dobrze wypada A* z heurystyką euklidesową lub manhattan. Niezależnie od parametrów grafu, algorytm ten zazwyczaj wyznaczał trasy o najniższym koszcie a jednocześnie charakteryzował się umiarkowanym zużyciem pamięci oraz stosunkowo małą liczbą odwiedzonych wierzchołków.

Algorytm Dijkstry, choć nie korzysta z funkcji szacującej odległość do celu, również zapewniał poprawne i optymalne wyniki. W porównaniu do A* działał jednak mniej selektywnie, co czasem przekładało się na większy zakres przeszukiwania.

Algorytm BFS, mimo że także znajdował ścieżki o odpowiedniej długości, przeszukiwał graf szeroko, przez co zużywał więcej pamięci i analizował większą liczbę węzłów. DFS natomiast działał szybko, ale nie gwarantował znalezienia optymalnej trasy i często generował drogi o znacznie wyższym koszcie. Potwierdza to, że wybór algorytmu powinien zależeć od priorytetów: szybkości działania czy jakości rozwiązania.

Zastosowanie heurystyki indeksowej w A* pokazało, jak duże znaczenie ma dobór odpowiedniego sposobu szacowania odległości do celu. W każdym z przypadków, gdzie wykorzystano tę prostą

heurystykę, uzyskany koszt trasy był wyższy, a efektywność niższa niż przy bardziej dostosowanych wariantach.

Na koniec warto zwrócić uwagę, że wraz ze wzrostem liczby wierzchołków i zmianą parametrów grafu (takich jak gęstość czy zakres wag), różnice pomiędzy algorytmami czasem się zacierały, jednak różnice w efektywności (czas, pamięć, liczba odwiedzeń) były nadal zauważalne.

## 3.3 Implementacja w siatce dwuwymiarowej

W tej części skupiono się na zastosowaniu wcześniej omówionych algorytmów wyszukiwania ścieżek w środowisku o strukturze dwuwymiarowej siatki. Każde pole siatki odpowiada pojedynczemu wierzchołkowi grafu, a połączenia między nimi są możliwe wyłącznie w czterech kierunkach: góra, dół, lewo oraz prawo. Taka reprezentacja przypomina klasyczne układy map wykorzystywane w grach czy symulacjach nawigacyjnych.

Aby umożliwić czytelne przedstawienie procesu działania algorytmów, środowisko testowe opracowano przy użyciu biblioteki Pygame. Wizualizacja pozwalała na ręczne wyznaczenie punktu początkowego i docelowego oraz na wprowadzenie przeszkód w postaci pól niedostępnych. Każdy algorytm uruchamiano niezależnie, przy zachowaniu identycznych warunków startowych.

Algorytmy (BFS, DFS, Dijkstra oraz A* z heurystyką Manhattan i euklidesową) zostały zaadaptowane w taki sposób, by mogły operować na siatce, traktując komórki jako sąsiadujące wierzchołki. Wyniki testów zebrano w formie zrzutów ekranu oraz zestawiono w tabelach obejmujących czas działania, zużycie pamięci, liczbę odwiedzonych pól i długość znalezionej ścieżki.

### 3.3.1 Dostosowanie algorytmów do grafu siatkowego

W tym środowisku testowym skupiono się na dostosowaniu wcześniej opisanych algorytmów wyszukiwania ścieżek do działania w przestrzeni dwuwymiarowej siatki. Implementacja została zrealizowana przy użyciu biblioteki Pygame, która umożliwia interaktywną wizualizację działania algorytmów w czasie rzeczywistym. Zmiany w logice samych algorytmów były minimalne – najistotniejsze modyfikacje objęły sposób reprezentacji wierzchołków oraz sposób ich prezentacji na ekranie.

Każda komórka siatki odpowiada jednemu wierzchołkowi grafu, a połączenia między komórkami reprezentują krawędzie o jednakowym koszcie. Wierzchołki mają różne kolory zależnie od aktualnego stanu (startowy, końcowy, bariera, przetworzony, itd.), a ich wizualizacja odbywa się w czasie rzeczywistym podczas działania algorytmu.

#### 3.3.1.1 Reprezentacja wierzchołków w siatce 2D

Wszystkie komórki siatki zostały odwzorowane jako obiekty klasy Pole widocznej na Listingu 3.9. Każde takie pole przechowuje informacje o swojej pozycji, stanie oraz liście sąsiednich komórek, które są aktualizowane dynamicznie. Kluczowe metody tej klasy odpowiadają za zmianę koloru pola, sprawdzanie jego typu (np. bariera, start, cel), rysowanie na ekranie oraz resetowanie stanu.

```
class Pole:
    def __init__(self, wiersz, kolumna, rozmiar, ilosc_wierszy):
        self.wiersz = wiersz
        self.kolumna = kolumna
        self.x = wiersz * rozmiar
        self.y = kolumna * rozmiar
        self.kolor = BIALY
        self.sasiedzi = []
        self.rozmiar = rozmiar
        self.ilosc_wierszy = ilosc_wierszy

    def czy_start(self): return self.kolor == POMARANCZOWY
    def czy_cel(self): return self.kolor == TURKUSOWY
    def czy_bariera(self): return self.kolor == CZARNY
    def czy_otwarty(self): return self.kolor == ZIELONY
    def czy_zamkniety(self): return self.kolor == CZERWONY

    def ustaw_start(self): self.kolor = POMARANCZOWY
    def ustaw_cel(self): self.kolor = TURKUSOWY
    def ustaw_bariera(self): self.kolor = CZARNY
    def ustaw_otwarty(self): self.kolor = ZIELONY
    def ustaw_zamkniety(self): self.kolor = CZERWONY
    def ustaw_sciezka(self): self.kolor = FIOLETOWY
    def zresetuj(self): self.kolor = BIALY

    def rysuj(self, okno):
        pygame.draw.rect(okno, self.kolor, (self.x, self.y,
self.rozmiar, self.rozmiar))

    def aktualizuj_sasiadow(self, siatka):
        self.sasiedzi = []
        if self.wiersz < self.ilosc_wierszy - 1 and not
siatka[self.wiersz + 1][self.kolumna].czy_bariera():
            self.sasiedzi.append(siatka[self.wiersz + 1][self.kolumna])
        if self.wiersz > 0 and not siatka[self.wiersz -
1][self.kolumna].czy_bariera():
            self.sasiedzi.append(siatka[self.wiersz - 1][self.kolumna])
        if self.kolumna < self.ilosc_wierszy - 1 and not
siatka[self.wiersz][self.kolumna + 1].czy_bariera():
```

```
            self.sasiedzi.append(siatka[self.wiersz][self.kolumna + 1])
        if self.kolumna > 0 and not siatka[self.wiersz][self.kolumna -
1].czy_bariera():
            self.sasiedzi.append(siatka[self.wiersz][self.kolumna - 1])
```

Listing 3.10. Klasa reprezentująca komórki siatki

Dzięki zastosowaniu kolorów oraz interaktywnego rysowania każde pole natychmiast wizualizuje swój aktualny stan w toku działania programu

#### 3.3.1.2 Wizualizacja i odświeżanie siatki

W celu ciągłego odświeżania okna aplikacji oraz aktualizacji wizualnych zmian w siatce stworzono funkcję rysuj (Listing 3.11). Odpowiada ona za odmalowanie całej siatki oraz wyświetlenie parametrów działania algorytmu.

```
def rysuj(okno, siatka, wiersze, szerokosc):
    global statystyki_algorytmu
    okno.fill(BIALY)
    for rzad in siatka:
        for pole in rzad:
            pole.rysuj(okno)
    rysuj_siatke(okno, wiersze, szerokosc)
    if statystyki_algorytmu:
        czcionka = pygame.font.SysFont(None, 22)
        tekst = czcionka.render(statystyki_algorytmu, True, (0, 0, 0))
        okno.blit(tekst, (10, szerokosc + 10))
    pygame.display.update()
```

Listing 3.11. Funkcja do odświeżania wizualizacji

Takie podejście pozwala użytkownikowi śledzić postępy działania algorytmu w czasie rzeczywistym.

#### 3.3.1.3 Obsługa klawiatury i uruchamianie algorytmów.

W implementacji za pomocą biblioteki Pygame dodano również obsługę klawiszy, która umożliwia szybkie uruchamianie wybranego algorytmu wyszukiwania ścieżki. W momencie, gdy użytkownik zaznaczy w siatce punkt początkowy i końcowy (odpowiednio kolory pomarańczowy i turkusowy), może wybrać metodę działania za pomocą klawiszy numerycznych:

- 1 – Algorytm Dijkstry,
- 2 – A* z heurystyką Manhattan,
- 3 – BFS (Breadth-First Search),
- 4 – DFS (Depth-First Search),

- 5 – A* z heurystyką euklidesową.

Reakcja na naciśnięcie klawisza jest obsługiwana wewnątrz głównej pętli programu, której fragment przedstawiono na Listingu 3.12

```
def start_program(okno, szerokosc):
    WIERSZE = 50
    WYSOKOSC_SIATKI = szerokosc
    WYSOKOSC_OKNA = szerokosc + 40  # dodatkowy pasek na statystyki
    OKNO = pygame.display.set_mode((szerokosc, WYSOKOSC_OKNA))
    siatka = utworz_siatke(WIERSZE, szerokosc)
    poczatek = koniec = None
    dziala = True

    while dziala:
        rysuj(okno, siatka, WIERSZE, szerokosc)

        for zdarzenie in pygame.event.get():
            if zdarzenie.type == pygame.QUIT:
                dziala = False

            if pygame.mouse.get_pressed()[0]:
                poz = pygame.mouse.get_pos()
                w, k = kliknieta_pozycja(poz, WIERSZE, szerokosc)
                pole = siatka[w][k]
                if not poczatek and pole != koniec:
                    poczatek = pole
                    poczatek.ustaw_start()
                elif not koniec and pole != poczatek:
                    koniec = pole
                    koniec.ustaw_cel()
                elif pole != koniec and pole != poczatek:
                    pole.ustaw_bariera()
                 (…)
                if zdarzenie.key == pygame.K_1 and poczatek and koniec:
                    wyczysc_tymczasowe_pola(siatka)
                    for rzad in siatka:
                        for pole in rzad:
                            pole.aktualizuj_sasiadow(siatka)
                    dijkstra(lambda: rysuj(okno, siatka,
WIERSZE,szerokosc), siatka, poczatek, koniec)
```

Listing 3.12. Fragment obsługi klawiatury w pętli głównej

Takie rozwiązanie pozwala szybko i intuicyjnie porównywać zachowanie poszczególnych algorytmów w tej samej konfiguracji start-cel i układzie przeszkód.

### 3.3.2 Przykłady działania algorytmów na siatce 2D

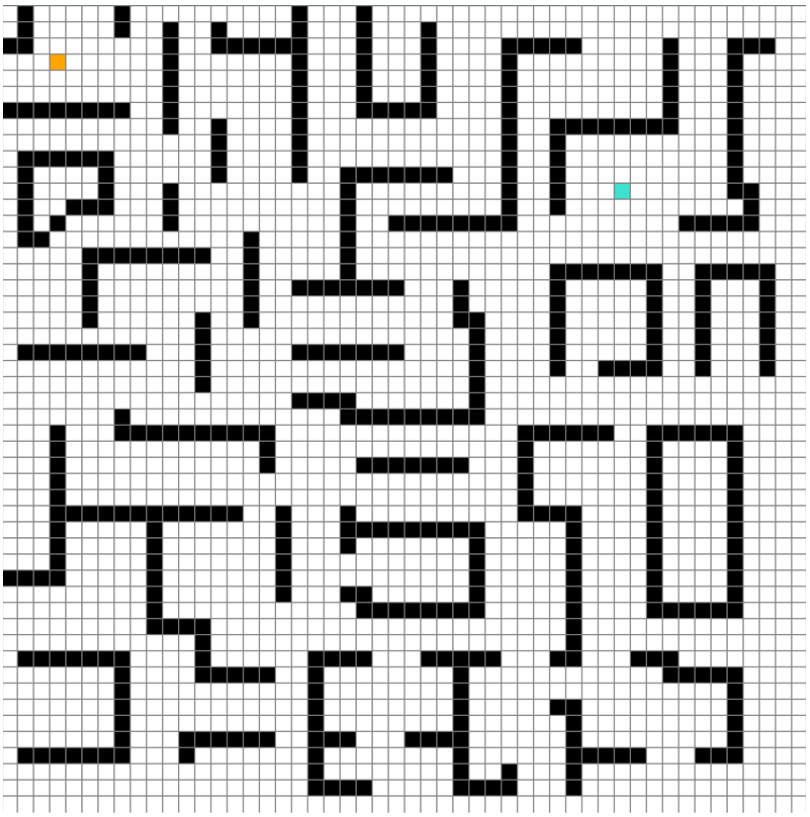
W celu zobrazowania praktycznego działania algorytmów wyszukiwania ścieżek, wykonano eksperymenty na dwóch różnych układach dwuwymiarowej siatki. W każdym przypadku wybrano inny układ przeszkód, a użytkownik ręcznie wskazał punkt początkowy oraz punkt docelowy. Dla każdej konfiguracji uruchomiono pięć różnych algorytmów: BFS, DFS, Dijkstra, A* z heurystyką manhattan oraz A* z heurystyką euklidesową.

Wyniki działania zaprezentowano w formie zrzutów ekranu przedstawiających końcowy stan siatki po znalezieniu ścieżki przez dany algorytm. Na obrazach wyróżniono:

- ścieżkę docelową (kolor fioletowy),
- punkt startowy (pomarańczowy),
- cel (turkusowy),
- odwiedzone wierzchołki (zielone),
- przeszkody (czarne).

Dzięki wizualizacji możliwe jest zaobserwowanie różnic w sposobie eksploracji przestrzeni oraz efektywności poszczególnych metod.

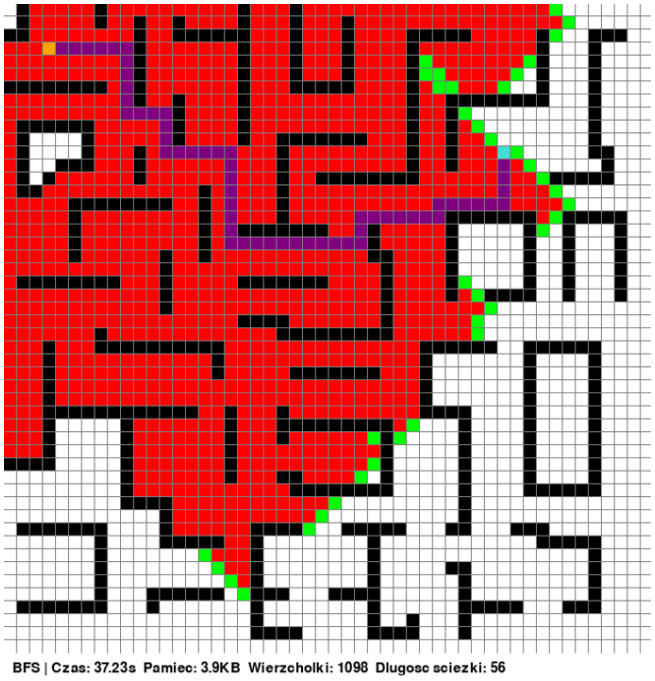
**Przypadek 1** – Siatka 50×50, punkt początkowy w lewym górnym rogu, cel w prawym. Na siatce rozmieszczono przeszkody, które wymuszają znalezienie alternatywnej trasy (rys. 3.4).



Rys. 3.4. Siatka z punktem początkowym i końcowym wraz z przeszkodami dla przypadku nr 1

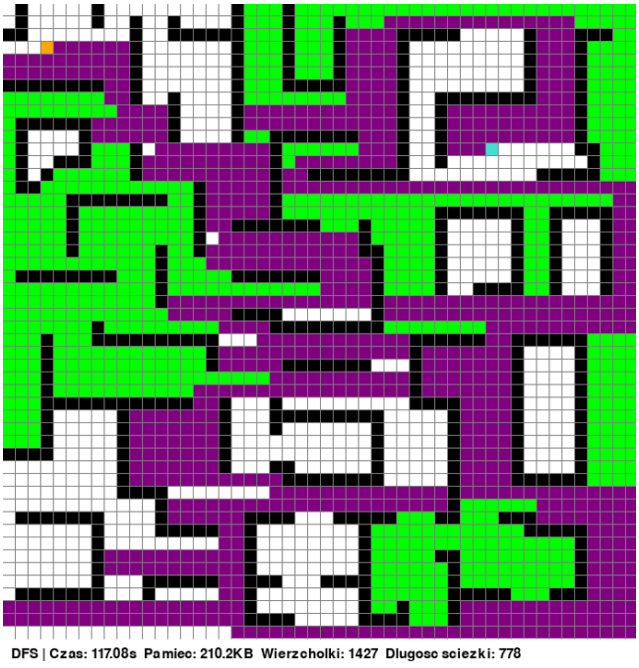
Wyniki wizualizacji dla przypadku 1 przedstawiono na rys. 3.5 – 3.10**:**

1) BFS



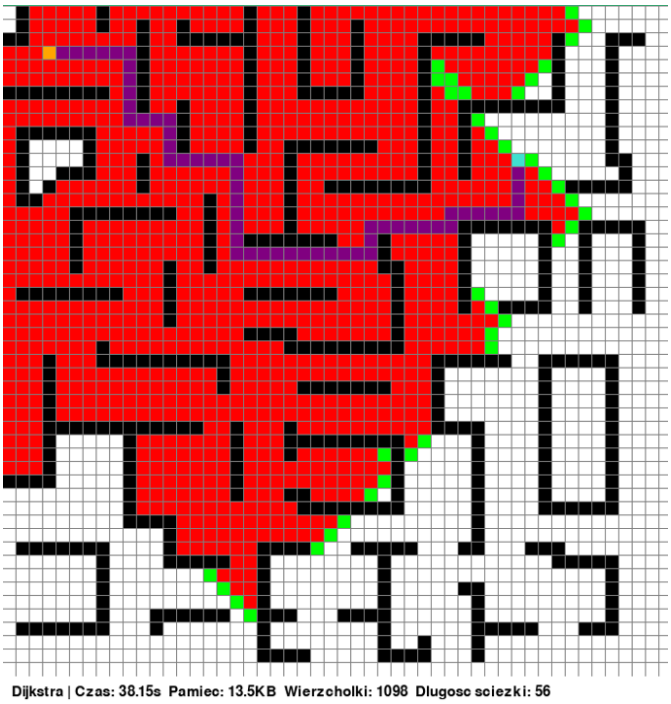

Rys. 3.5. Wynik BFS dla przykładu nr 1

2) DFS




Rys. 3.6. Wynik DFS dla przykładu nr 1

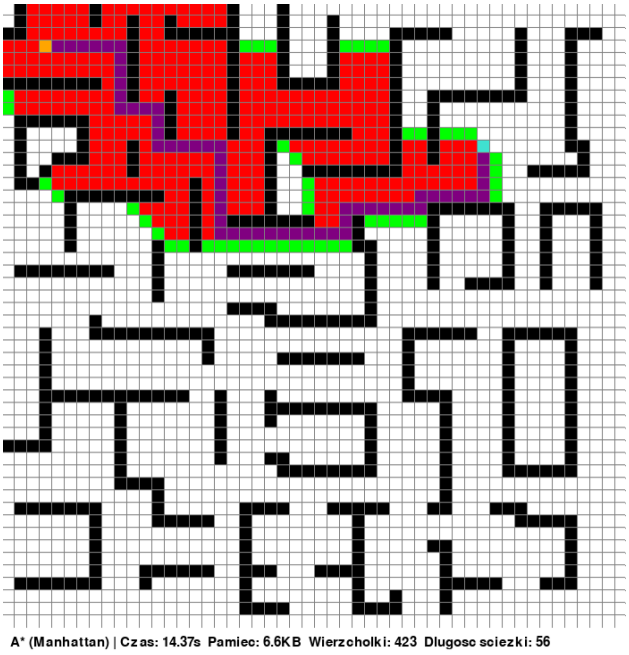
3) Algorytm Dijkstry




Rys. 3.7. Wynik Algorytmu Dijkstry dla przykładu nr 1

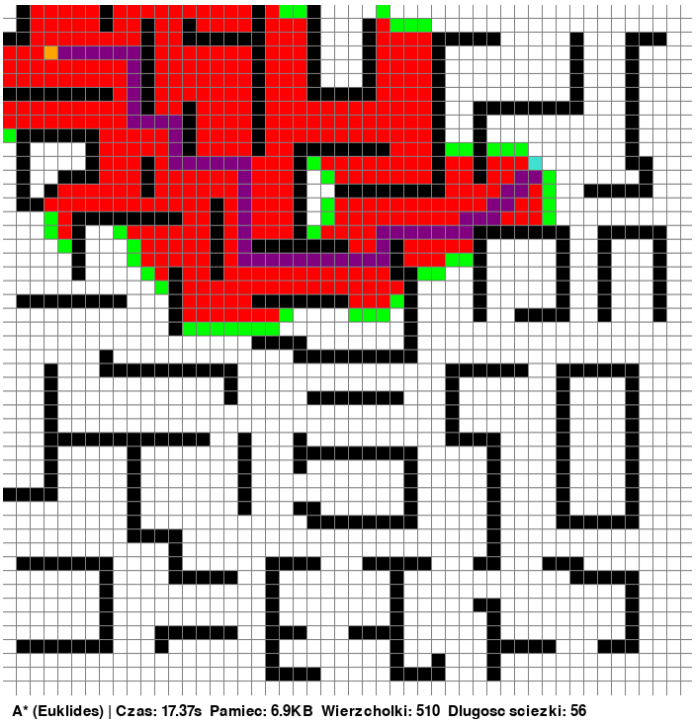
4) Algorytm A*

Manhattan:



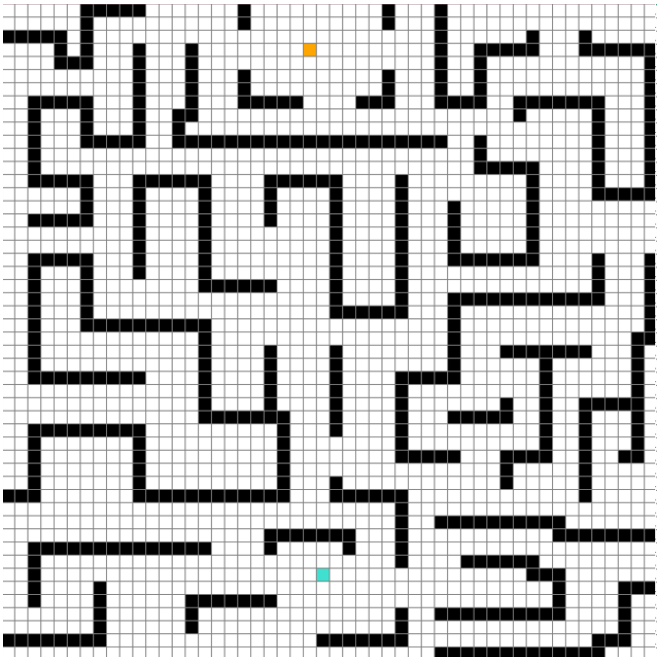

Rys. 3.8. Wyniki Algorytmu A* (Manhattan) dla przykładu nr 1

Euklides:




Rys. 3.9. Wyniki Algorytmu A* (Euklides) dla przykładu nr 1

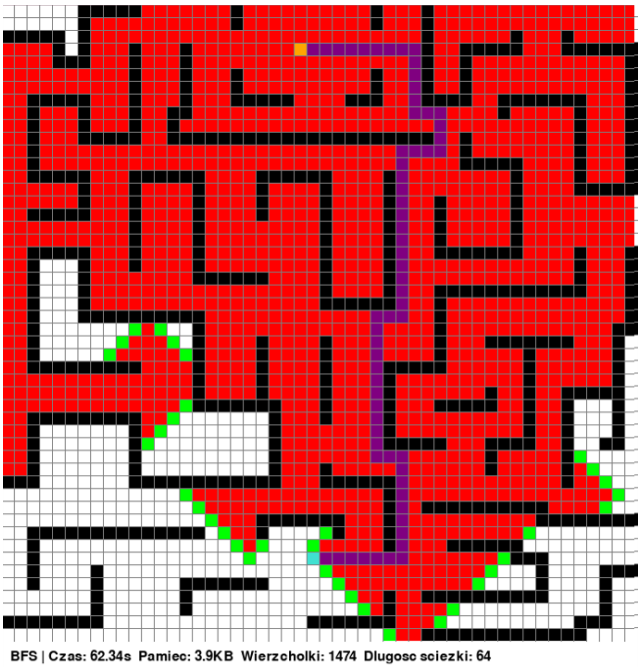
**Przypadek 2** – Siatka 50×50, punkt początkowy w środkowej górnej części, cel - środkowej dolnej (rys. 3.9).



Rys. 3.10. Siatka z punktem początkowym i końcowym wraz z przeszkodami dla przykładu nr 2

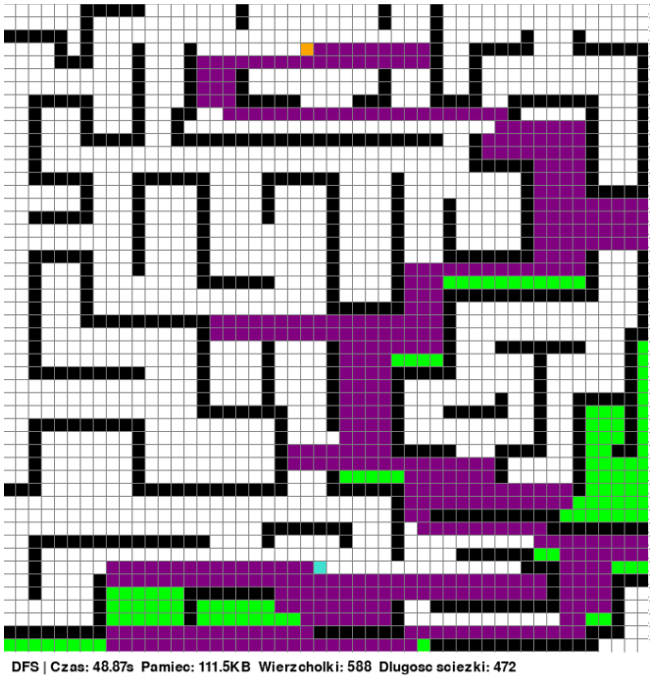
Wyniki wizualizacji dla przypadku 2 przedstawiono na rys. 3.11 – 3.15:

1) BFS



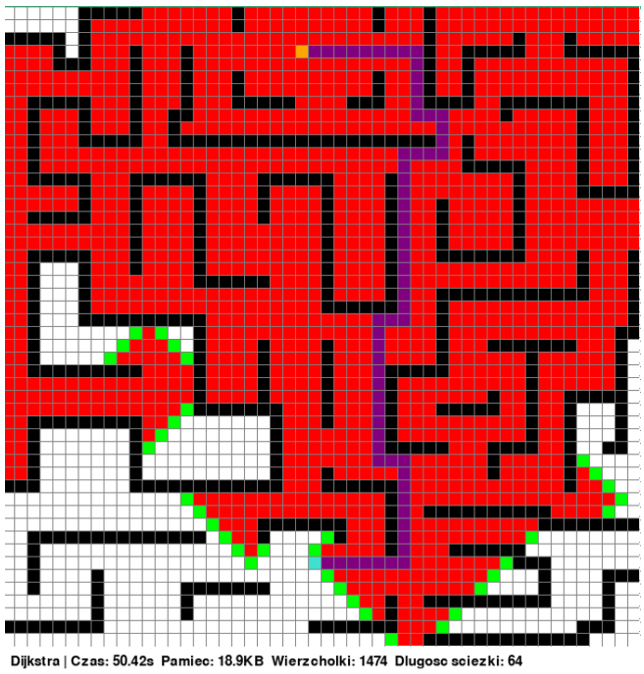

Rys. 3.11. Wynik BFS dla przykładu nr 2

2) DFS




Rys. 3.12. Wynik DFS dla przykładu nr 2

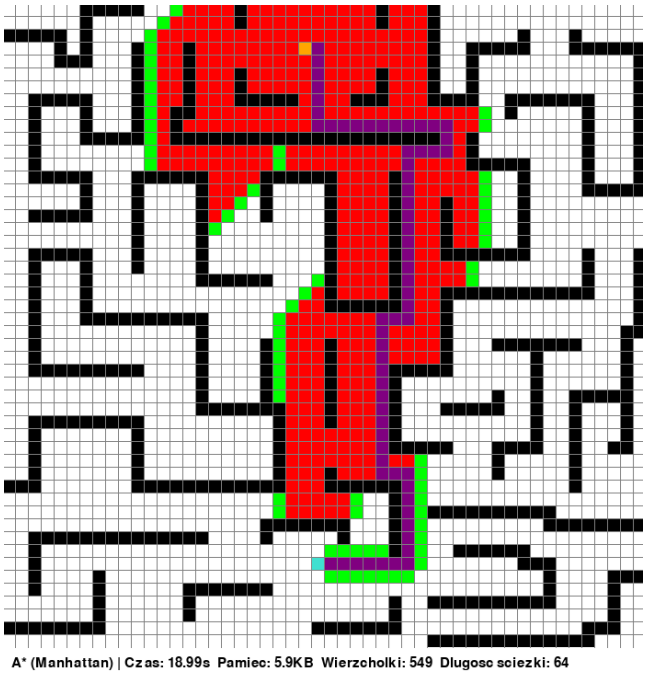
3) Algorytm Dijkstry




Rys. 3.13. Wynik Algorytmu Dijkstry dla przykładu nr 2

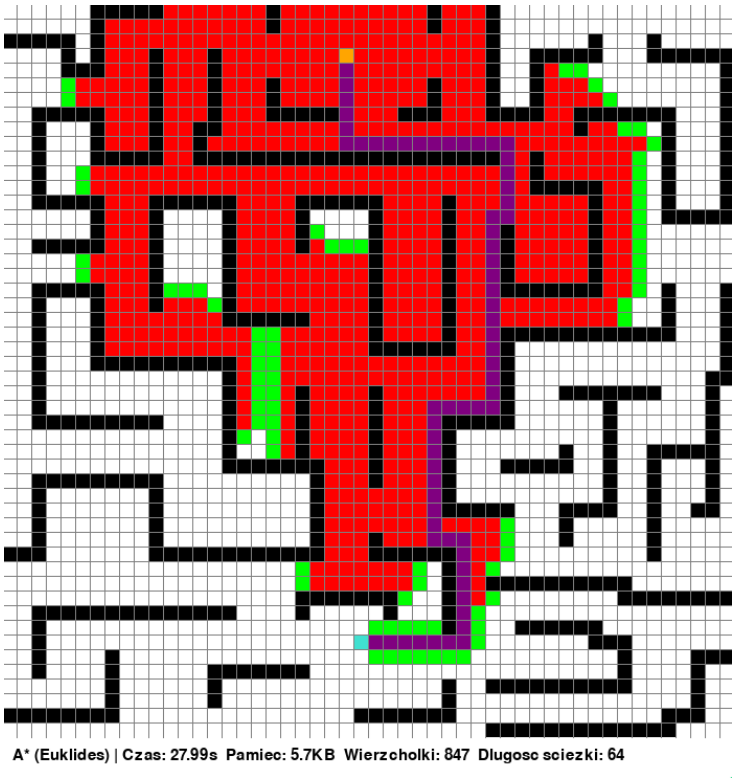
4) Algorytm A*

Manhattan:




Rys. 3.14. Wyniki Algorytmu A* (Manhattan) dla przykładu nr 2

Euklides:




Rys. 3.15. Wyniki Algorytmu A* (Euklides) dla przykładu nr 2

### 3.3.3 Tabele porównawcze wyników dla siatki 2D

Dla obu przypadków siatek testowych z sekcji 3.3.3 przeprowadzono pomiary wydajności wszystkich zaimplementowanych algorytmów. Ocenie poddano cztery podstawowe kryteria tj. czas działania, ilość zużytej pamięci, liczbę odwiedzonych pól oraz długość odnalezionej ścieżki. Wyniki przedstawiono w poniższych zestawieniach tabelarycznych (tab. 3.4 – 3.5).

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Długość ścieżki |
|---|---|---|---|---|---|
| 1 | BFS | 37.23 | 3.9 | 1098 | 56 |
| 2 | DFS | 117.08 | 210.2 | 1427 | 778 |
| 3 | Dijkstra | 38.15 | 13.5 | 1098 | 56 |
| 4 | A* (heurystyka euklidesowa) | 17.37 | 6.9 | 510 | 56 |
| 5 | A* (heurystyka manhattan) | 14.37 | 6.6 | 423 | 56 |

Tab. 3.4. Wyniki dla siatki pierwszej

W pierwszym przypadku większość algorytmów była w stanie odnaleźć ścieżkę o identycznej długości, jednak różnice uwidoczniły się w czasie działania i zasobach pamięciowych. Część metod przeszukiwała znacznie więcej pól niż inne, co może mieć wpływ na wydajność przy bardziej złożonych planszach.

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Długość ścieżki |
|---|---|---|---|---|---|
| 1 | BFS | 62.34 | 3.9 | 1474 | 64 |
| 2 | DFS | 48.87 | 111.5 | 588 | 472 |
| 3 | Dijkstra | 50.42 | 18.9 | 1474 | 64 |
| 4 | A* (heurystyka euklidesowa) | 27.99 | 5.7 | 847 | 64 |
| 5 | A* (heurystyka manhattan) | 18.99 | 5.9 | 549 | 64 |

Tab. 3.5. Wyniki dla siatki drugiej

W drugim teście, pomimo podobnego efektu końcowego w postaci wyznaczonej trasy, zaobserwowano zauważalne różnice w liczbie odwiedzonych komórek oraz w czasie działania. Sposób eksploracji przestrzeni znacząco wpłynął na zużycie pamięci i efektywność działania algorytm.

### 3.3.4 Wnioski z analizy porównawczej na siatce 2D

Po przeanalizowaniu wyników działania algorytmów na siatkach 2D da się zauważyć kilka ważnych różnic, które wpływają na ich praktyczne zastosowanie.

Na pierwszy rzut oka widać, że wszystkie algorytmy – oprócz DFS – były w stanie znaleźć poprawne, czyli najkrótsze możliwe ścieżki. Jednak sposób, w jaki do tych ścieżek dochodziły, był już zupełnie inny. Przykładowo, BFS i Dijkstra odwiedzały bardzo dużo pól, przez co ich działanie trwało dłużej i zużywały więcej zasobów mimo że ostateczny wynik był poprawny. Widać, że działają one *„na ślepo”* – sprawdzają po kolei wiele możliwych dróg, niezależnie od tego, czy prowadzą w dobrym kierunku.

DFS natomiast bardzo często wybierał drogę naokoło, co kończyło się długą, nieoptymalną trasą. Co więcej, przez to że przeszukiwał głęboko, potrafił zużyć zaskakująco dużo pamięci, szczególnie w bardziej skomplikowanych układach.

Zupełnie inaczej wyglądały wyniki algorytmu A*, który w obu wersjach heurystyki (euklidesowej i Manhattan) poradził sobie zdecydowanie najlepiej. Znajdował najkrótszą ścieżkę, ale robił to szybciej i odwiedzał znacznie mniej pól niż inne metody. Działał po prostu bardziej „*inteligentnie”*, bo dzięki heurystyce wiedział, w którą stronę iść, zamiast sprawdzać wszystkie punkty po kolei.

Jeśli zależy nam na szybkości działania i oszczędności zasobów, A* z odpowiednią heurystyką jest zdecydowanie najlepszym wyborem. BFS i Dijkstra też dają dobre wyniki, ale są mniej wydajne. DFS w testach wypadł najsłabiej i raczej nie nadaje się do praktycznego użycia w takich sytuacjach.

## 3.4 Implementacja na mapie drogowej Rzeszowa

W tej części pracy przedstawiono rozszerzenie działania klasycznych algorytmów wyszukiwania ścieżek na bardziej realistyczne środowisko – mapę miasta. Wybór padł na Rzeszów, którego dane drogowe pozyskano z serwisu OpenStreetMap. Dzięki zastosowaniu formatu graphml, udało się stworzyć pełną reprezentację sieci ulicznej jako grafu, który następnie został zaimportowany do aplikacji zbudowanej w języku Python. W celu zapewnienia bardziej przyjaznego interfejsu użytkownika, rozwiązanie zostało wzbogacone o komponent front-endowy w JavaScript oparty na bibliotece Leaflet.

Dzięki temu użytkownik może wyznaczać ścieżki bezpośrednio na mapie, wybierając punkty startowe i końcowe. Po ich zaznaczeniu uruchamiany jest wybrany algorytm, a wynikowa trasa wizualizowana jest w oknie mapy. Obsługiwane metody to m.in. BFS, Dijkstra, A* z heurystyką euklidesową oraz A* z heurystyką Manhattan.

### 3.4.1 Implementacja algorytmów

Warto zaznaczyć, że zaimplementowane algorytmy działają według tych samych zasad, które zostały omówione we wcześniejszych podrozdziałach. Różnica polega jedynie na strukturze danych wejściowych i zamiast siatki 2D przetwarzany jest graf oparty o dane geograficzne. Dzięki temu możliwe jest zastosowanie tych samych rozwiązań do wyznaczania tras w rzeczywistym środowisku miejskim.

W trakcie testów zrezygnowano z użycia algorytmu DFS, ponieważ nie sprawdza się on w zadaniach, gdzie celem jest znalezienie możliwie najkrótszej trasy. DFS nie bierze pod uwagę długości odcinków i często prowadzi przez przypadkowe ścieżki, co czyni go mało użytecznym w kontekście rzeczywistej sieci drogowej. Zamiast tego w analizie znalazł się algorytm BFS, który również działa bez znajomości odległości do celu, ale w przeciwieństwie do DFS zawsze znajduje najkrótszą ścieżkę, przynajmniej jeśli wszystkie przejścia traktowane są jako jednakowe. Dzięki temu stanowi on dobrą podstawę do porównań z bardziej zaawansowanymi podejściami, takimi jak Dijkstra czy A*.

#### 3.4.1.1 Pobranie grafu

Do pobrania i zapisania sieci drogowej miasta Rzeszów wykorzystano bibliotekę OSMnx, która pozwala na bezpośrednią komunikację z serwisem OpenStreetMap. Dzięki jednej komendzie (Listing 3.13) możliwe było pobranie kompletnego grafu drogowego z uwzględnieniem jedynie przejezdnych tras. Wygenerowana struktura została zapisana do formatu GraphML, umożliwiającego późniejsze wykorzystanie danych w analizie i wizualizacji.

```
import osmnx as ox
```

```
G = ox.graph_from_place("Rzeszów, Poland", network_type="drive")
ox.save_graphml(G, "data/rzeszow.graphml")
```

Listing 3.13. Pobranie mapy drogowej Rzeszowa

#### 3.4.1.2 Wczytanie grafu z pliku

Graf drogowy został przygotowany wcześniej w formacie .graphml, a następnie przetwarzany do formy obiektowej za pomocą biblioteki networkx (Listing 3.14). Proces ten został zaimplementowany w module graph_loader.py.

```
import networkx as nx

def wczytaj_graf(sciezka):
    return nx.read_graphml(sciezka)
```

Listing 3.14. Funkcja wczytania mapy drogowej Rzeszowa

W module głównym main.py (Listing 3.15) funkcja ta jest wykorzystywana do załadowania pełnego grafu ulicznego miasta.

```
from graph_loader import wczytaj_graf

graf = wczytaj_graf("data/rzeszow.graphml")
```

Listing 3.15. Wczytanie mapy drogowej Rzeszowa

#### 3.4.1.3 Lokalizacja punktów startowych i końcowych

Wyznaczenie wierzchołków startowego i końcowego odbywa się na podstawie kliknięcia użytkownika na mapie lub wprowadzenia współrzędnych. Aplikacja korzysta z funkcji ox.distance.nearest_nodes, która tłumaczy współrzędne GPS na najbliższy węzeł w grafie.

```
start_node = ox.distance.nearest_nodes(graf, X=lon_start, Y=lat_start)
end_node = ox.distance.nearest_nodes(graf, X=lon_end, Y=lat_end)
```

Listing 3.16. Funkcja znajdowania najbliższego węzła

#### 3.4.1.4 Integracja algorytmów

Algorytmy Dijkstra, BFS oraz A* zostały zaimplementowane w pliku routing.py, a ich struktura została zintegrowana z funkcją find_path, umożliwiającą uruchomienie dowolnego z nich w zależności od wybranego trybu:

```
def find_path(G, start_node, end_node, algorithm_name,
heuristic="euklidesowa"):
    if algorithm_name == "A*":
        # A* z heurystyką
    elif algorithm_name == "Dijkstra":
        #  Dijkstra
```

```
    elif algorithm_name == "BFS":
        # przeszukiwanie wszerz
```

Listing 3.17. Funkcja uruchamiająca odpowiedni algorytm

W przypadku algorytmu A* dostępne są dwie heurystyki – Euklidesowa oraz Manhattana. Warto zauważyć, że ta druga jest mniej efektywna w rzeczywistych mapach drogowych, ponieważ nie uwzględnia krzywizn i orientacji tras.

#### 3.4.1.5 Połączenie z interfejsem

Aplikacja posiada rozbudowany graficzny interfejs użytkownika wykonany z wykorzystaniem PyQt5. Kluczowym elementem jest komponent QWebEngineView, w którym osadzana jest mapa z interaktywnymi elementami.

```
self.map_view = QWebEngineView()
self.map_view.page().setWebChannel(self.channel)
```

Listing 3.18. Utworzenie widoku mapy i połączenie z kanałem komunikacji

Do renderowania mapy wykorzystano bibliotekę folium, generującą kod HTML, który jest osadzany bezpośrednio w przeglądarce w aplikacji. Zdarzenia kliknięcia obsługiwane są przez JavaScript (Leaflet), który komunikuje się z Pythonem przez QWebChannel.

```
fetch('http://localhost:8000/save_coords', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify(coords)
});
```

Listing 3.19. Fragment skryptu JavaScript odpowiadającego za przekazanie punktów do backendu

Mapa generowana jest dynamicznie w Pythonie, a zawartość HTML przekazywana bezpośrednio do QWebEngineView w postaci kodu.

### 3.4.2 Wizualizacja wyników

W ramach aplikacji opracowano interaktywny panel mapowy umożliwiający wyznaczanie trasy pomiędzy dowolnymi punktami na terenie Rzeszowa. Użytkownik może wskazać punkt początkowy i końcowy zarówno klikając bezpośrednio na mapie, jak i wpisując dane adresowe lub współrzędne geograficzne. Aplikacja automatycznie przelicza trasę przy użyciu wybranych algorytmów i wyświetla je jako linie o różnych kolorach.

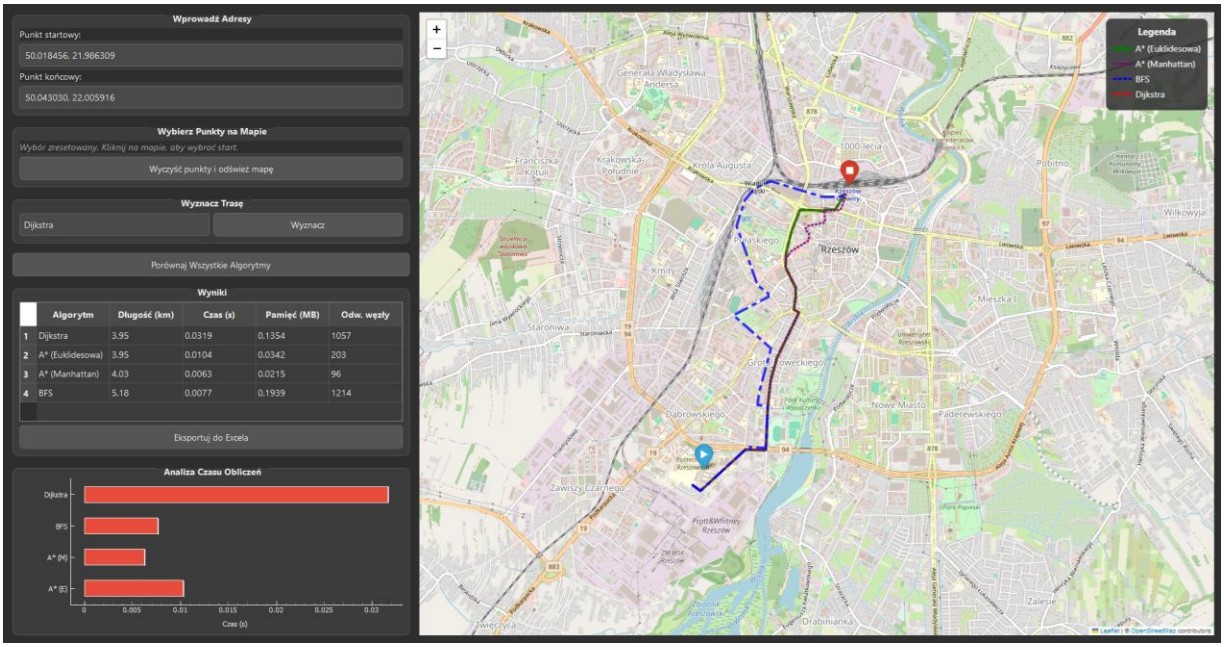
Ze względu na charakter danych pochodzących z OpenStreetMap, zdarza się, że wybrane lokalizacje nie są w pełni zintegrowane z siecią drogową. Może to prowadzić do nieintuicyjnych tras, szczególnie przy punktach na obrzeżach miasta lub w mniej zurbanizowanych rejonach.

Wizualizacja wyników obejmuje:

- prezentację tras na mapie,
- tabelaryczne zestawienie wyników dla każdego algorytmu,
- wykres słupkowy ilustrujący czas działania.

Na poniższych ilustracjach rys. 3.16 – 3.18 przedstawiono przykładowe testy wykonane w różnych rejonach Rzeszowa

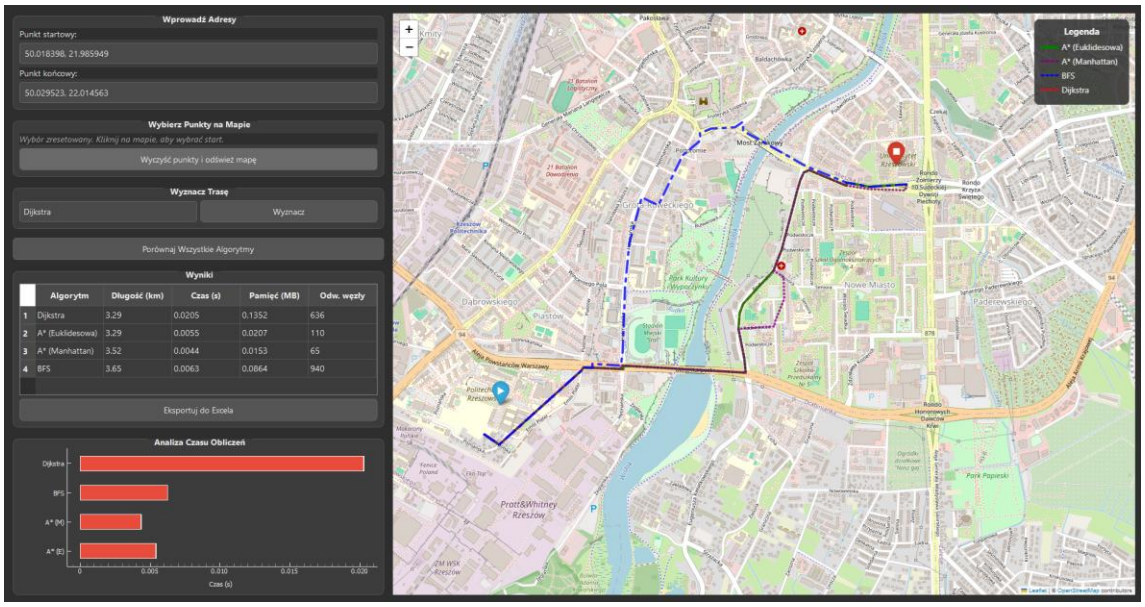
1) Trasa Politechnika Rzeszowska – dworzec kolejowy w Rzeszowie




Rys. 3.16. Porównanie tras między Politechniką Rzeszowską a dworcem kolejowym w Rzeszowie

Na rys. 3.16 wszystkie trasy wiodą przez centrum miasta, lecz tylko A* i Dijkstra utrzymują się przy głównych drogach. BFS znacząco wydłuża ścieżkę, prowadząc mniej bezpośrednio.

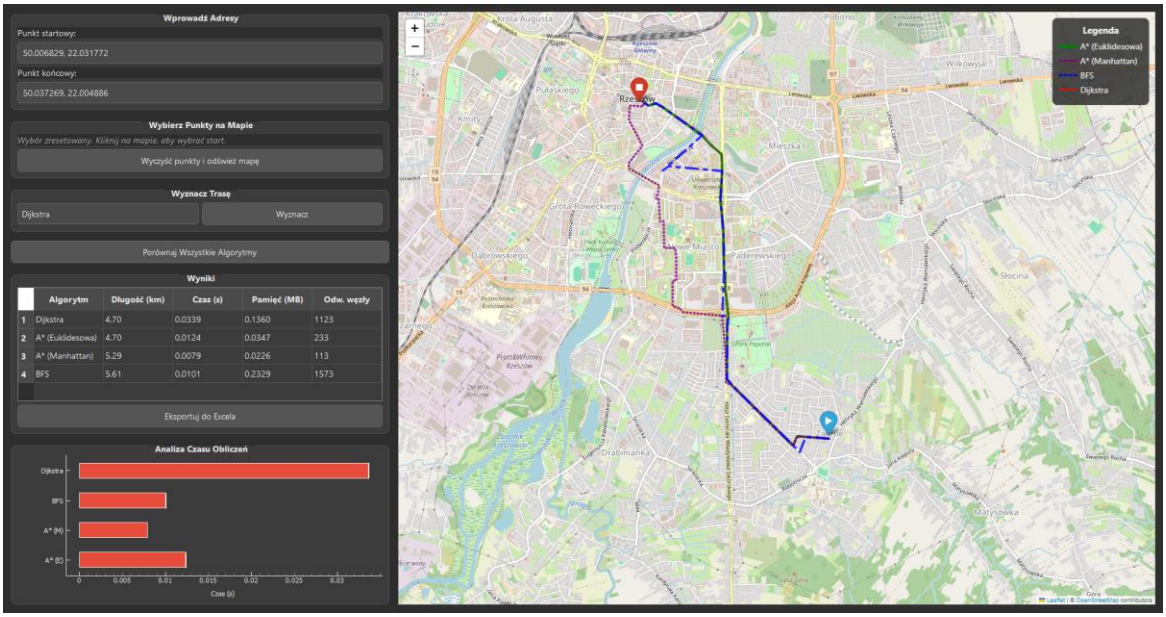
2) Trasa Politechnika Rzeszowska – Uniwersytet Rzeszowski




Rys. 3.17. Porównanie tras między Politechniką Rzeszowską a Uniwersytetem Rzeszowskim

W tej trasie zarówno A* z heurystyką euklidesową, jak i Dijkstra znalazły identyczny przebieg. A* w drugiej wersji lekko odbiega od ścieżki optymalnej, natomiast BFS ponownie wybiera trasę boczną i dłuższą.

3) Trasa okolice Zalesia – rynek.




Rys. 3.18. Porównanie tras z Zalesia na rynek

Trasa pokrywa większy obszar miasta. Algorytmy A* (Euklidesowa) i Dijkstra prowadzą tą samą drogą, podczas gdy BFS i manhattanowa heurystyka znów zbacza i wydłuża trasę.

### 3.4.3 Tabelaryczne zestawienie wyników dla każdego algorytmu

W poniższych tabelach zestawiono podstawowe metryki działania poszczególnych algorytmów trasowania w trzech wybranych scenariuszach miejskich z rys. 3.14 – 3.16. Ponownie porównano czas działania, wykorzystanie pamięci, długość wyznaczonej trasy oraz liczbę odwiedzonych wierzchołków.

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Długość [km] |
|---|---|---|---|---|---|
| 1 | Dijkstra | 0.0319 | 135.4 | 1057 | 3.95 |
| 2 | A* (heurystyka euklidesowa) | 0.0104 | 34.2 | 203 | 3.95 |
| 3 | A* (heurystyka manhattana) | 0.0063 | 21.5 | 96 | 4.03 |
| 4 | BFS | 0.0077 | 193.9 | 1214 | 5.18 |

Tab. 3.6. Porównanie wyników – Politechnika Rzeszowska – Dworzec Główny

Zarówno algorytm Dijkstry, jak i A* z heurystyką euklidesową znalazły trasę o tej samej długości 3.95 km, co wskazuje na ich zgodność przy dobrym dopasowaniu heurystyki. Czas działania A* był jednak wyraźnie niższy i wynosił 0.0104s w porównaniu z 0.0319s Dijkstry. Najmniej efektywnie wypadł BFS, który przeskanował ponad 1200 wierzchołków i wyznaczył trasę o długości aż 5.18 km, czyli o ponad kilometr dłuższą od optymalnej. A* z heurystyką manhattan zadziałał najszybciej, ale jego wynikowa trasa była mniej precyzyjna i miała długość 4.03 km.

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Długość [km] |
|---|---|---|---|---|---|
| 1 | Dijkstra | 0.0205 | 135.2 | 636 | 3.29 |
| 2 | A* (heurystyka euklidesowa) | 0.0055 | 20.7 | 110 | 3.29 |
| 3 | A* (heurystyka manhattana) | 0.0044 | 15.3 | 65 | 3.52 |
| 4 | BFS | 0.0063 | 86.4 | 940 | 3.65 |

Tab. 3.7. Porównanie wyników – Politechnika Rzeszowska – Uniwersytet Rzeszowski

Tutaj A* (Euklidesowa) i Dijkstra osiągnęły identyczny wynik pod względem długości trasy czyli 3.29 km. Algorytm A* wykazał się zdecydowanie lepszą wydajnością (0.0055s), odwiedzając zaledwie 110 wierzchołków, co przy tym samym wyniku Dijkstry (636 wierzchołków) pokazuje dużą oszczędność zasobów. Manhattanowa wersja A* była najszybsza, ale kosztem dłuższej ścieżki wynoszącej 3.52 km. Podobnie jak wcześniej, BFS przeanalizował dużą liczbę punktów i wyznaczył trasę 3.65 km, czyli gorszą jakościowo.

| | Algorytm | Czas [s] | Pamięć [kB] | Odwiedzone wierzchołki | Długość [km] |
|---|---|---|---|---|---|
| 1 | Dijkstra | 0.0339 | 136.0 | 1123 | 4.70 |
| 2 | A* (heurystyka euklidesowa) | 0.0124 | 34.7 | 233 | 4.70 |
| 3 | A* (heurystyka manhattana) | 0.0079 | 22.6 | 113 | 5.29 |
| 4 | BFS | 0.0101 | 232.9 | 1573 | 5.61 |

Tab. 3.8. Porównanie wyników – okolice Zalesia – rynek

W tym scenariuszu Dijkstra oraz A* (euklidesowa) także wyznaczyły identyczną trasę mającą długość 4.70 km. Różnica pojawia się w liczbie odwiedzonych węzłów, mianowicie A* wykorzystał tylko 233 wierzchołki a Dijkstra ponad 1100. Heurystyka manhattan, choć szybka, zwróciła trasę o długości 5.29 km. BFS znowu odwiedził najwięcej węzłów (1573) i wygenerował najdłuższą trasę 5.61 km.

### 3.4.4 Interpretacja i podsumowanie wyników na rzeczywistej mapie.

Wyniki testów przeprowadzonych w oparciu o realną sieć drogową Rzeszowa pokazują wyraźne różnice między algorytmami a zwłaszcza pod względem długości wyznaczonych tras. Najbardziej zbliżone do siebie były trasy wyznaczone przez algorytm Dijkstry oraz A* z heurystyką euklidesową. W większości przypadków dawały one identyczną ścieżkę, zarówno pod względem długości, jak i przebiegu. Dijkstra nie korzysta z heurystyki, przez co potrzebuje więcej zasobów, ale w zamian zwraca ścieżkę o minimalnym koszcie.

Z kolei A* z heurystyką manhattan wypadał znacznie szybciej, ale jego trasy były zazwyczaj nieco dłuższe, czasem nawet o kilkaset metrów. Heurystyka manhattan, choć była efektywna w siatkach regularnych, w przypadku realnych dróg wypada słabiej, szczególnie tam, gdzie ulice nie są rozmieszczone prostopadle. Mimo mniejszej liczby odwiedzanych węzłów, A* (M) często zbaczał z głównych dróg, wybierając mniej intuicyjne warianty.

BFS wypadł najsłabiej, jeśli chodzi o jakość tras i niemal we wszystkich przypadkach wybierał ścieżki wyraźnie dłuższe, nierzadko prowadzące przez boczne ulice lub mniej optymalne połączenia. Jego zaletą był stosunkowo prosty mechanizm działania, ale w środowisku miejskim nie przekładało się to na efektywność.

Podsumowując, testy na rzeczywistych danych drogowych potwierdziły, że algorytm A* z heurystyką euklidesową łączy wysoką skuteczność z szybkością i umiarkowanym zużyciem zasobów. Dijkstra może być dobrą alternatywą tam, gdzie zależy na ścieżce o najniższym koszcie bez potrzeby stosowania heurystyki. A* (M) sprawdza się tylko częściowo, a BFS wypada najmniej korzystnie w przypadku złożonego, nieregularnego układu ulic.

## 3.5 Wnioski z przeprowadzonych Testów

W ramach przeprowadzonych testów zestawiono działanie wybranych algorytmów trasowania w trzech różnych środowiskach: grafach o zróżnicowanej strukturze i wagach, siatkach dwuwymiarowych oraz na rzeczywistej mapie drogowej Rzeszowa. Zróżnicowanie danych wejściowych pozwoliło zaobserwować, jak konkretne algorytmy radzą sobie w praktyce i jakie mają ograniczenia.

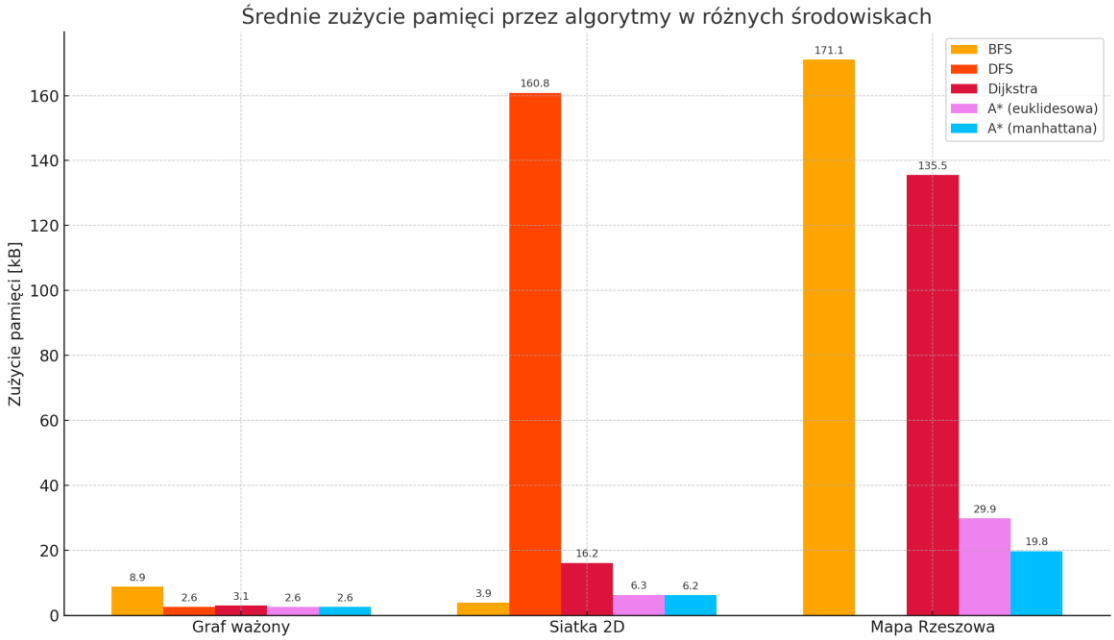
W środowisku grafów ważonych najlepsze rezultaty dawały metody uwzględniające wagi – zwłaszcza Dijkstra oraz A* z heurystyką euklidesową lub manhattan. Oba podejścia zapewniały dobre wyniki przy stosunkowo niskim koszcie obliczeniowym. BFS przeszukiwał szerzej i zużywał więcej pamięci, mimo że również znajdował poprawną ścieżkę. DFS natomiast często zbaczał z optymalnej trasy i generował wynik o wyższym koszcie.

Na siatkach 2D dominowały te same rozwiązania. Algorytm A* ponownie wypadał najlepiej, zarówno pod względem szybkości działania, jak i liczby odwiedzanych pól. BFS oraz Dijkstra dostarczały poprawnych wyników, lecz były mniej efektywne pod względem zasobów. DFS, mimo szybkości działania w niektórych przypadkach, nie gwarantował jakościowych tras i wymagał znacznie więcej pamięci operacyjnej. W tym środowisku można było również zauważyć wpływ wyboru heurystyki na efektywność A* (heurystyka manhattan działa lepiej w układach bardziej regularnych).

W środowisku najbardziej zbliżonym do rzeczywistości czyli na mapie Rzeszowa, algorytmy trasowania zachowywały się nieco inaczej. Dijkstra i A* prowadziły trasy głównymi drogami, tworząc sensowne i intuicyjne przebiegi co potwierdzają wizualizacje. BFS, mimo że działał szybko, częściej wybierał ścieżki boczne, a to w efekcie skutkowało większą liczbą odwiedzonych węzłów oraz wydłużeniem trasy. Heurystyka manhattan, ze względu na brak regularnej struktury ulic, nie przynosiła tu żadnych korzyści. A* z heurystyką euklidesową był w tym przypadku najbardziej uniwersalny i efektywny.

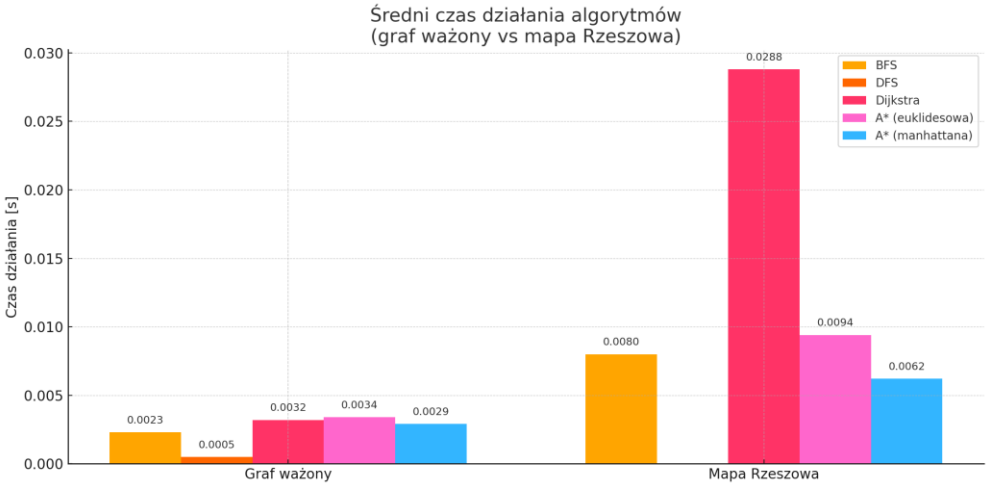
Średnie wyniki dla każdej metody w różnych środowiskach zostały przedstawione na poniższych wykresach.

Na rys. 3.17 zilustrowano średnie zużycie pamięci przez algorytmy w trzech typach środowisk. Szczególnie wysokie wartości dla DFS w siatce 2D rzucają się w oczy, podobnie jak zwiększone zapotrzebowanie pamięciowe A* i Dijkstry w środowisku miejskim, gdzie graf jest znacznie większy i bardziej złożony.



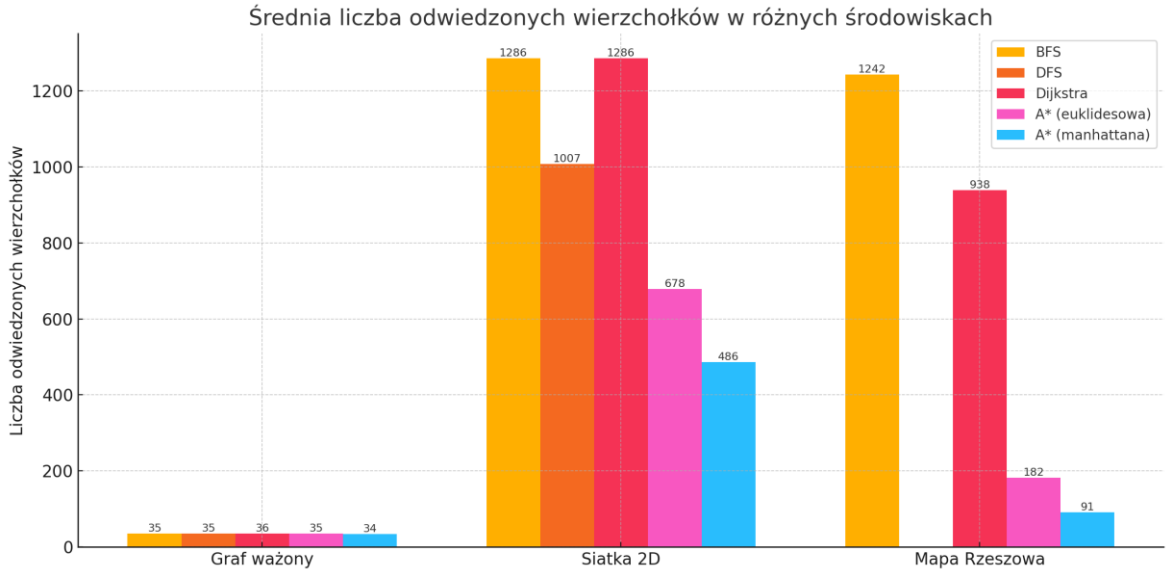

Rys. 3.17. Wykres porównujący średnie zużycie pamięci algorytmu w danych środowiskach

Na rys. 3.18 zestawiono średni czas działania algorytmów na grafie ważonym oraz mapie rzeczywistej. Wyniki pokazują, że różnice w czasie obliczeń zależą głównie od wielkości i struktury danych – w bardziej złożonych grafach czas działania staje się istotnym czynnikiem.




Rys. 3.18. Wykres porównujący średni czas działania algorytmu w grafie ważonym i na mapie Rzeszowa

Z kolei rys. 3.19 przedstawia średnią liczbę odwiedzonych wierzchołków. Wyniki jasno pokazują, że A* dzięki heurystyce potrafi ograniczyć zakres przeszukiwania, w przeciwieństwie do BFS, który analizuje większą część grafu, co potwierdza jego mniej selektywny charakter.




Rys. 3.19. Wykres porównujący liczbę odwiedzonych wierzchołków

Jako podsumowanie można wyprowadzić wniosek, że w kontekście zastosowań praktycznych, najlepszym kompromisem między jakością trasy a czasem działania okazał się algorytm A* z heurystyką euklidesową. Dijkstra to dobre rozwiązanie tam, gdzie nie da się łatwo zdefiniować heurystyki. BFS i DFS znajdują zastosowanie głównie w celach porównawczych lub edukacyjnych, ale ich efektywność w rzeczywistych warunkach jest ograniczona.

# 4. Podsumowanie

Praca została podzielona na część teoretyczną oraz praktyczną, z których każda pełniła istotną rolę w realizacji przyjętego celu.

W części teoretycznej omówiono podstawowe zagadnienia związane z grafami, algorytmami i heurystykami. Przedstawiono definicje, klasyfikację oraz przykłady grafów, a także opisano sposoby przeszukiwania i wyznaczania najkrótszych ścieżek. Osobną uwagę poświęcono algorytmowi A*, który dzięki zastosowaniu heurystyki stanowi jeden z najefektywniejszych sposobów trasowania. Materiał ten stanowił niezbędne przygotowanie do późniejszych implementacji oraz analizy wyników.

Część praktyczna obejmowała budowę środowisk testowych i porównanie działania wybranych algorytmów w trzech różnych kontekstach: grafach losowych z wagami, dwuwymiarowej siatce oraz rzeczywistej mapie drogowej miasta Rzeszowa. W każdym przypadku oceniano podstawowe metryki efektywności, takie jak czas działania, zużycie pamięci, długość wyznaczonej ścieżki oraz liczba odwiedzonych węzłów.

Przeprowadzone testy wykazały, że algorytm heurystyczny A* z funkcją euklidesową, charakteryzuje się najlepszym bilansem między szybkością a jakością wyników. Algorytm Dijkstry uzyskiwał równie dobre rezultaty, lecz przy większym nakładzie operacyjnym. Z kolei metody BFS i DFS, choć prostsze i szybsze w uruchomieniu, w wielu przypadkach prowadziły do mniej korzystnych tras lub zużywały znacznie więcej zasobów.

W efekcie zrealizowano założone cele pracy tj. porównano działanie wybranych algorytmów w różnych typach grafów, oceniono ich skuteczność w praktycznych zastosowaniach oraz wskazano metody najlepiej sprawdzające się w danym kontekście.

# 5. Bibliografia

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, & C. Stein. (2009). Introduction to Algorithms (3rd ed.). MIT Press. PDF: https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction_to_algorithms-3rd%20Edition.pdf, [dostęp: 10.04.2025]

[2] S. Russell, P. Norvig, Sztuczna inteligencja. Nowe spojrzenie, Tom 1, tłum. A. Grażyński, Wydawnictwo Helion, Gliwice 2023. Na podstawie: *Artificial Intelligence*: *A Modern Approach*, 4th ed., Pearson, 2020,

[3] D. Bertsimas, J. Tsitsiklis, *Introduction to Linear Optimization*, Athena Scientific, 1997

[4] A. V. Aho, J. D. Ullman, Foundations of Computer Science, online: http://infolab.stanford.edu/~ullman/focs.html, [dostęp: 10.04.2025].

[5] D. Harel, *Rzecz o istocie informatyki. Algorytmika*, tłum. Z. Płoski, Wydawnictwo Helion, Gliwice 2008,

[6] R. J. Wilson, Wprowadzenie do teorii grafów, Wydawnictwo Naukowe PWN, Warszawa 2012. ISBN: 978-83-0115-0662.

[7] R. Diestel, Graph Theory, 5th ed., Springer, Berlin 2017. Elektroniczna wersja: EMIS Monographs in Mathematics – online: https://www.emis.de/monographs/Diestel/en/GraphTheoryII.pdf, [dostęp: 07.06.2025].

[8] GeeksforGeeks, „Applications of Breadth First Traversal”, https://www.geeksforgeeks.org/applications-of-breadth-first-traversal ,

[9] J. Iyanda, *A Comparative Analysis of Breadth First Search (BFS) and Depth First Search (DFS) Algorithms*, 2023,

https://www.researchgate.net/publication/370751322_Title_A_Comparative_Analysis_of_Breadth_First_Search_BFS_and_Depth_First_Search_DFS_Algorithms, [dostęp: 10.06.2025]

[10] L. Wang, *Parallel Cluster-BFS and Applications to Shortest Paths*, arXiv:2410.17226, 2024, https://arxiv.org/abs/2410.17226, [wersja 1]

[11] GeeksforGeeks, A* Search Algorithm, https://www.geeksforgeeks.org/a-search-algorithm/,[dostęp: 01.06.2025]

[12] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107

[13] NetworkX Developers, *NetworkX – Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks*, dokumentacja online: https://networkx.org, [dostęp: 18.06.2025]

[14] Matplotlib Developers, *Matplotlib documentation*, online: https://matplotlib.org, [dostęp: 18.06.2025]

[15] Python Software Foundation, *tracemalloc — Trace memory usage in your Python program*, dokumentacja, online: https://docs.python.org/3/library/tracemalloc.html, [dostęp: 22.06.2025]

[16] Python Software Foundation, *The Python Standard Library – random, time*, dokumentacja online: https://docs.python.org/3/library/, [dostęp: 22.06.2025].

[17] Shinners P., Pygame Documentation, Pygame Community, [Online]. Available: https://www.pygame.org/docs/ [dostęp: 22.06.2025]

[18] Python Software Foundation. (2023). queue — A synchronized queue class. Python 3 Documentation.

https://docs.python.org/3/library/queue.html#queue.PriorityQueue [dostęp: 22.06.2025]

[19] Boeing, G. (2025). Modeling and Analyzing Urban Networks and Amenities with OSMnx. Geographical Analysis, published online ahead of print. doi:10.1111/gean.70009

# 6. Załączniki

Implementacja na grafie ważonym:

Algorytmy_na_grafie_wazonym.zip

Implementacja na siatce 2D:

Algorytmy_na_siatce_2D.zip

Implementacja na mapie miasta Rzeszów:

Algorytmy_na_mapie.zip

# 7. Streszczenie


POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza Rzeszów, 2025
Wydział Matematyki i Fizyki Stosowanej

**STRESZCZENIE PRACY DYPLOMOWEJ MAGISTERSKIEJ**

**Analiza i zastosowanie algorytmu A* w problemach wyszukiwania najkrótszych ścieżek.**

Autor: Adrian Dereń, nr albumu: FS0 – DU-166724 Promotor: dr Paweł Bednarz
Słowa kluczowe: algorytmy, najkrótsze ścieżki, A gwiazdka, bfs, dijkstra

W pracy przeanalizowano działanie wybranych algorytmów wyznaczania tras w grafach, ze szczególnym uwzględnieniem A*. Porównano ich skuteczność w różnych środowiskach – grafach ważonych, siatkach 2D oraz mapie miejskiej. Oceniono czas działania, zużycie pamięci i jakość tras. Wyniki pokazały, że A* z dobraną heurystyką daje najlepszy kompromis między szybkością a dokładnością.

RZESZOW UNIVERSITY OF TECHNOLOGY Rzeszow, 2025
Faculty of Mathematics and Applied Physics

**DIPLOMA THESIS (MS) ABSTRACT**

**Analysis and application of the A* algorithm to shortest path searching problems.**

Author: Adrian Dereń, code: FS0 – DU-166724 Supervisor: Dr Paweł Bednarz
Key words: algorithms, shortest paths, A star, bfs, dijkstra

The thesis analyzes the performance of selected pathfinding algorithms in graphs, with a focus on A*. Their effectiveness was compared across weighted graphs, 2D grids, and real city maps. Metrics such as execution time, memory usage, and path quality were evaluated. Results showed that A* with a well-chosen heuristic offers the best balance between speed and accuracy.