

SPDB

SPRAWOZDANIE Z PROJEKTU

Temat projektu: Wyszukiwanie najszybszej trasy dla wielu punktów zainteresowania

Mateusz Krakowski

Bartosz Latosek

24 kwietnia 2024

Spis treści

1 Wprowadzenie	2
1.1 Treść zadania	2
1.2 Interpretacja	2
2 Architektura	2
2.1 Baza danych	2
2.2 Aplikacja	3
3 Instrukcja użytkowania	5
3.1 Opis interfejsu	5
3.2 Przypisanie funkcji do przycisków	5
4 Użyty algorytm wyszukiwania najszybszej ścieżki	6
4.1 Algorytm Dijkstry	6
4.2 Opis Algorytmu Dijkstry zmodifikowanego na potrzebę aplikacji	6
4.3 Implementacja algorytmu	7
5 Testy	8
5.1 Plan testowania	8
5.2 Baza danych	8
5.3 Testy działania algorytmu	9
6 Literatura	11

1 Wprowadzenie

1.1 Treść zadania

Znajdowanie najlepszej trasy.

Implementacja aplikacji do wyznaczania najlepszej trasy wg zadanego parametrów, opracowanie metody oraz wykonanie testów sprawdzających praktyczną użyteczność zaproponowanych rozwiązań (wyznaczanie trasy w miastach dla kilku lub więcej podanych miejsc do odwiedzenia). Podstawowymi parametrami do wyboru najlepszej trasy jest: czas i odległość. Dodatkowym wymogiem jest minimalizacja skrętów w lewo. Dokumentacja końcowa powinna zawierać:

- Opis metody wyznaczania najlepszej drogi, w tym sposobu wyboru najlepszej trasy. Opis modelu danych.
- Opis architektury aplikacji.
- Informacje o implementacji (wykorzystane algorytmy + skomentowany kod źródłowy).
- Opis i wyniki przeprowadzonych testów.

Proponowana liczba osób w grupie: 2

1.2 Interpretacja

W ramach zadania stworzony zostanie system umożliwiający definiowanie własnych punktów w abstrakcji miasta. Przyjęty zostanie model grafowy, tzn. każdy punkt w mieście będzie miał odpowiadające mu ścieżki, prowadzące do innych zdefiniowanych punktów. Stworzone punkty oraz ścieżki będą zapisywane w bazie danych, z którą będzie się łączyć główny interfejs aplikacji. Użytkownik będzie miał możliwość wybrania interesujących go punktów na mapie a następnie za pomocą wciśnięcia odpowiednich przycisków będzie mógł wybrać tryb optymalizacji (droga / czas) z uwzględnieniem minimalizacji lewoskrętów (lub bez niej). Najkrótsza ścieżka będzie przedstawiona użytkownikowi w formie graficznej i aktualizowana w czasie rzeczywistym w przypadku zmiany wybranego trybu.

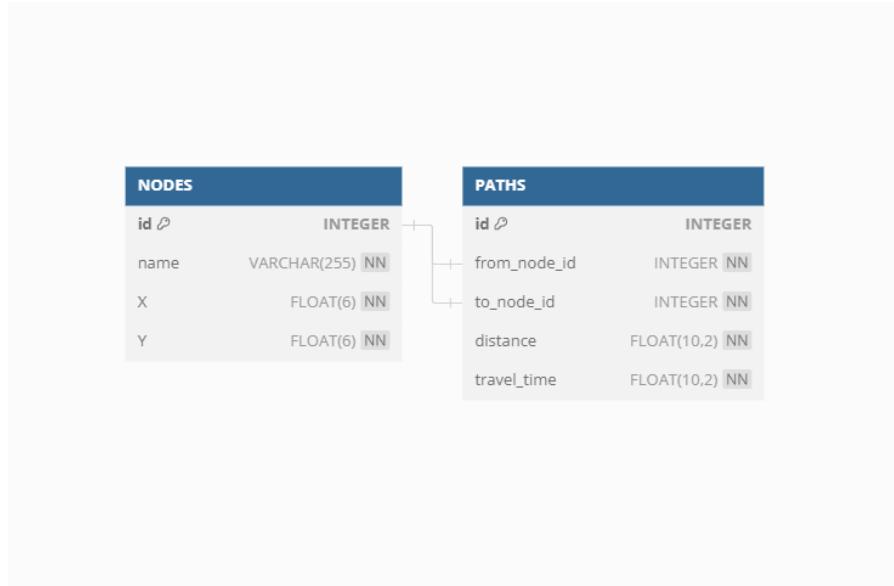
2 Architektura

2.1 Baza danych

Na potrzeby opracowanych założeń, baza danych została zaprojektowana z myślą o przechowywaniu obiektów reprezentujących punkty i połączenia w abstrakcyjnym mieście. Istnieje możliwość konfiguracji własnej mapy węzłów i połączeń, bez zmiany zachowania symulacji.

Baza danych została utworzona w technologii MySQL. Instrukcje instalacji oraz konfiguracji bazy danych zawarte są w pliku *README.md* w głównym repozytorium projektowym[1]. Schematy tabel zawarte są w podfolderze *tables* głównego folderu zawierającego elementy powiązane z bazą danych - *database*. Przykładowe dane do przeprowadzenia symulacji znajdują się w folderze *data*.

się w podfolderze *data*. Zawierają w sobie grupę punktów i połączeń między nimi z arbitralnie dobranymi kosztami. Schemat reprezentujący stworzone encje widoczny jest poniżej:



Rysunek 1: Schemat bazy danych

Tworzenie tabel i zapełnianie ich przykładowymi danymi dzieje się za pośrednictwem skryptów napisanych w języku *Python* zawartych w folderze *database*. W celu utworzenia tabel należy wywołać skrypt *init_tables.py*, w celu wypełnienia ich przykładowymi danymi - *fill_tables.py*, a w celu zresetowania bazy danych i usunięcia tabel wraz z danymi w nich zawartymi - *drop_tables.py*.

Główna część aplikacji łączy się z bazą danych za pomocą biblioteki *MySQL-python*. Zdefiniowane w pliku *.env* referencje do bazy danych, oraz użytkownika pozwalają na połączenie się z lokalnie działającą instancją *MySQL* i wykonywanie działań na bazie danych za pomocą wygodnego *API* biblioteki.

2.2 Aplikacja

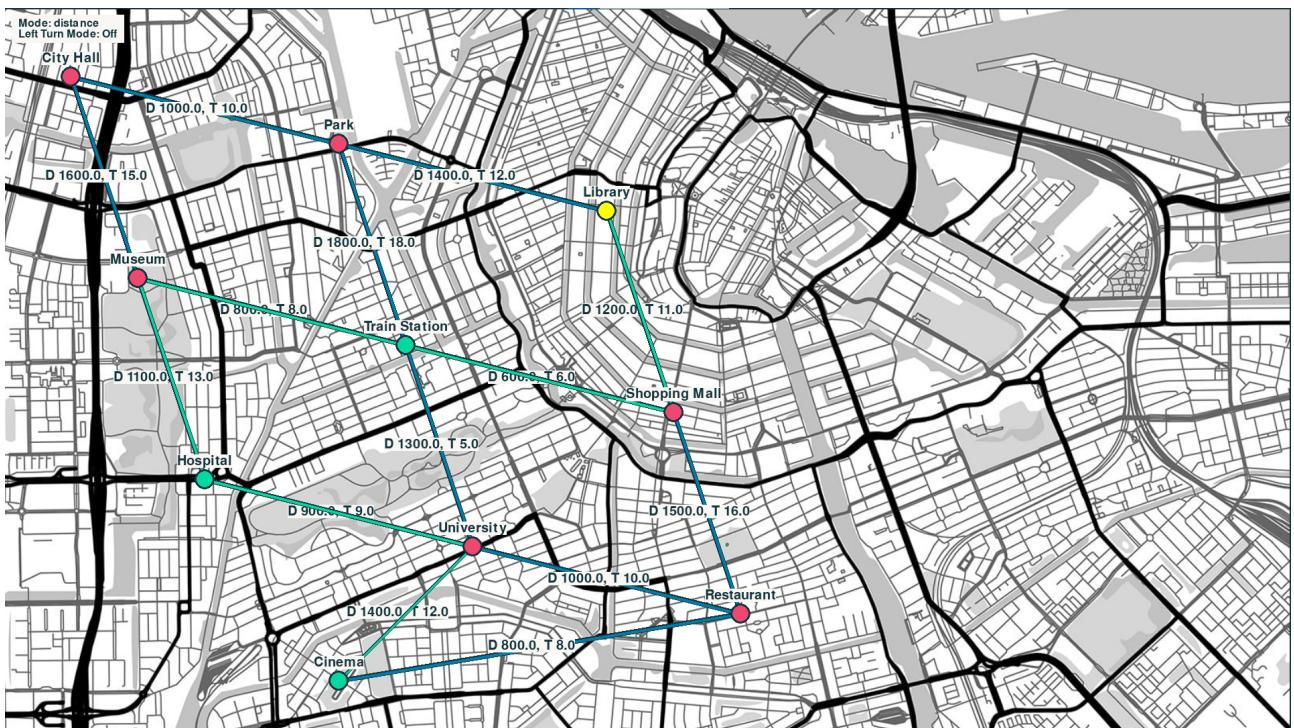
Po uruchomieniu aplikacji, użytkownikowi ukazuje się mapa abstrakcyjnego miasta, z którą może wchodzić w interakcje. Mapa posiada funkcję przybliżania / oddalania widoku oraz przesuwania się po niej - podobnie jak w przypadku powszechnych aplikacji wspomagających nawigację. Po wybraniu 2 lub więcej punktów na mapie (dodatkowo oznaczonych po ich wybraniu), podświetlona zostanie najbardziej optymalna ścieżka pomiędzy wybranymi węzłami zgodnie z zadanymi opcjami.

Interfejs został w całości wykonany za pośrednictwem biblioteki *pygame* języka *Python*. Kod źródłowy został w całości wykonany w języku *Python* i jest podzielony na moduły zgodnie z najlepszymi praktykami programistycznymi w tym języku. Submoduł *datamodels* definiuje

typy danych, na które przekształcane są pobrane z bazy danych surowe informacje. Stanowią abstrakcję ułatwiającą interakcje reszty aplikacji z bazą danych.

Dane ładowane są z bazy do interfejsu za pomocą *database_manager* zawartego w submodule *objects*. Stanowi on abstrakcję, za pomocą której możliwe jest pobieranie danych z bazy i przekształcanie je na odpowiadające obiekty zdefiniowane w wyżej opisywanym submodule *datamodels*. Kolejnymi obiektami zawartymi w tym submodule są *algorithm* - implementujący algorytm Dijkstry, *map* zapewniający interfejs mapy i definiujący zachowania takie jak rysowanie punktów, ścieżek i poruszanie się po mapie, *user_interface* przekazujący akcje użytkownika do pozostałych obiektów oraz główny obiekt zarządzający symulacją i łączący w sobie pozostałe - *simulation_manager*.

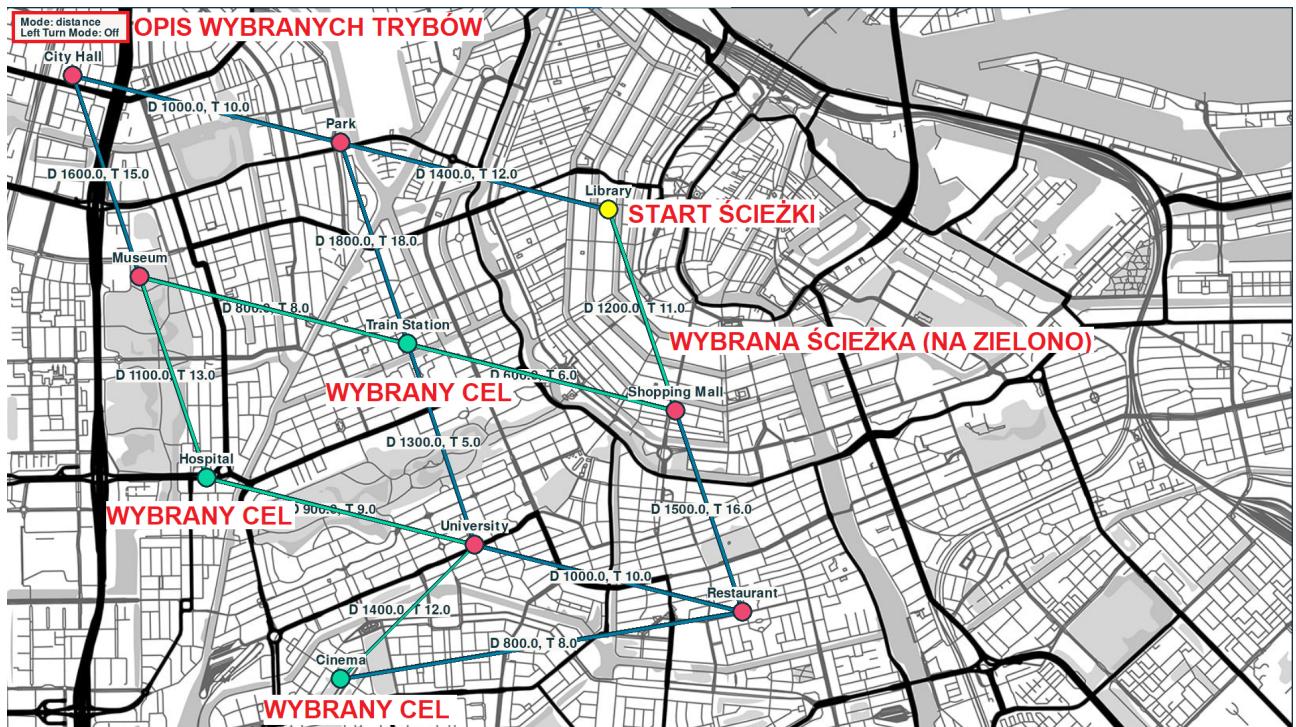
Przykładowy ekran działania aplikacji przedstawiony został na rysunku poniżej:



Rysunek 2: Zrzut ekranu z działającej aplikacji

3 Instrukcja użytkowania

3.1 Opis interfejsu



Rysunek 3: Opis interfejsu

Tak jak przedstawiono na powyższym rysunku, start najszybciej ścieżki reprezentujemy jako żółtą kropkę, kolejne cele są przedstawione jako zielone koła, czerwone koła to niewybrane punkty. Droga standardowo jest koloru niebieskiego, jeśli jest to droga przez którą przechodzi najszybsza trasa, to zmienia ona swój kolor na zielony. Każda ścieżka jest opisana wartościami "D" i "T", kolejno "distance" i "time". Reprezentują one koszty jakie muszą zostać poniesione aby przebyć ów trasę. Dodatkowo w lewym górnym rogu opisane jest, w jakim trybie optymalizacji się znajdujemy ("distance" lub "time") i czy uwzględniamy dodatkową karę za lewoskręt.

3.2 Przypisanie funkcji do przycisków

Na potrzebę wygodnego sterowania stworzono następujące przypisanie klawiszów do funkcji:

- **Przytrzymanie lewego przycisku myszy:** Przesuwanie mapy w dowolną stronę.
- **Lewy przycisk myszy:** Wybór miejsc zainteresowania przez które ma prowadzić trasa.
- **Prawy przycisk myszy:** Wyczyszczenie listy miejsc zainteresowania
- **Klawisz 'Q':** Włączenie trybu szukania najszybszej trasy pod względem Dystansu (na mapie wartości "D").

- **Klawisz 'W':** Włączenie trybu szukania najszybszej trasy pod względem Czasu (na mapie wartości "T").
- **Klawisz 'E':** Włączenie/Wyłączenie dodawania kary za lewoskręt.

4 Użyty algorytm wyszukiwania najszybszej ścieżki

Poniżej przedstawiamy pseudokod podstawowego algorytmu Dijkstry:

4.1 Algorytm Dijkstry

Dijkstra(Graf, StartowyWierzchołek, DocelowyWierzchołek):

1. Inicjalizuj odległości dla wszystkich wierzchołków na nieskończoność (z wyjątkiem StartowegoWierzchołka, ustawionej na 0).
2. Utwórz pustą kolejkę priorytetową Q.
3. Wstaw StartowyWierzchołek do Q z odległością 0.
4. Dopóki Q nie jest puste:
 - a. Pobierz wierzchołek u z najmniejszą odlegością z Q.
 - b. Jeśli u == DocelowyWierzchołek, zakończ algorytm.
 - c. Dla każdej krawędzi wychodzącej z u:
 - i. Oblicz nową odległość dla sasiada v, sumując odległość do u i wagę krawędzi.
 - ii. Jeśli nowa odległość jest mniejsza niż dotychczasowa odległość do v,
 - Zaktualizuj odległość do v.
 - Ustaw poprzednik v na u.
 - Wstaw v do Q z nową odlegością.
5. Odtwórz najkrótszą ścieżkę od DocelowyWierzchołek do StartowyWierzchołek, korzystając z poprzedników.

4.2 Opis Algorytu Dijkstry zmodifikowanego na potrzebę aplikacji

Poniżej przedstawiamy pseudokod zmodyfikowanego na nasze potrzeby algorytmu Dijkstry:

Algorytm Dijkstry(Graf, WierzchołekStartowy, WierzchołekDocelowy):

1. Inicjalizuj odległości dla wszystkich wierzchołków jako nieskończoność.
2. Ustaw odległość dla WierzchołekStartowy na 0.
3. Utwórz kolejkę priorytetową Q, gdzie pierwszym elementem jest WierzchołekStartowy z odlegością 0.
4. Dopóki kolejka Q nie jest pusta:
 - a. Pobierz wierzchołek u z najmniejszą odlegością z Q.
 - b. Jeśli u == WierzchołekDocelowy, zakończ algorytm.
 - c. Dla każdej krawędzi wychodzącej z u:
 - i. Oblicz nową odległość dla sasiada v, sumując odległość do u i wagę krawędzi.
 - ii. Jeśli tryb skrętu w lewo jest aktywny

- i u nie jest WierzchołekStartowy:
 - Jeśli skręt w lewo:
 - Jeśli tryb kosztu to dystans:
 - Aktualizuj koszt do v o dodatkowy koszt skrętu w lewo.
 - Jeśli tryb kosztu to czas:
 - Aktualizuj koszt do v o dodatkowy koszt skrętu w lewo.
- iii. W przeciwnym razie:
 - Jeśli tryb kosztu to dystans:
 - Aktualizuj koszt do v o dystans do sąsiada.
 - Jeśli tryb kosztu to czas:
 - Aktualizuj koszt do v o czas do sąsiada.
- iv. Jeśli nowa odległość jest mniejsza niż dotychczasowa odległość do v:
 - Zaktualizuj odległość do v.
 - Ustaw poprzednik v na u.
 - Wstaw v do Q z nową odlegością.

5. Odtwórz najkrótszą ścieżkę od WierzchołekDocelowy do WierzchołekStartowy, korzystając z poprzedników.

4.3 Implementacja algorytmu

Algorytm zaimplementowano jako klasę Dijkstra z komentarzami opisującymi jej działanie w języku angielskim. Znajduje się ona na repozytorium projektu[1] w katalogu `src\object\algorithm.py`. Sama implementacja skrętu w lewo polega na stworzeniu 2 wektorów, jeden od punktu początkowego do miejsca w którym może dojść do lewoskrętu, drugiego który idzie od punktu potencjalnego lewoskrętu do następnego. Z tych wektorów obliczany zostaje iloczyn wektorowy, jeśli iloczyn jest mniejszy od 0 to doszło do lewoskrętu. W standardowym układzie 2D działałyby się tak dla iloczynu większego od zera, ale z uwagi na to że biblioteka którą my użyliśmy do stworzenia interfejsu użytkownika ma odwróconą oś Y, to i to odwrócenie trzeba zaaplikować do algorytmu wyznaczania lewoskrętu./ wygląda to następująco:

```
def is_left_turn(  
    self, previous_node: Node, current_node: Node, next_node: Node  
) -> bool:  
    """  
    Checks if the turn from the current node to the  
    next node is a left turn.  
    """  
    current_vector = (  
        current_node.coordinates.x - previous_node.coordinates.x,  
        current_node.coordinates.y - previous_node.coordinates.y,  
    )  
    next_vector = (  
        next_node.coordinates.x - current_node.coordinates.x,  
        next_node.coordinates.y - current_node.coordinates.y,  
    )
```

```
# Calculate the cross product of the vectors
cross_product = (
    current_vector[0] * next_vector[1] - current_vector[1] * next_vector[0]
)

# The Y axis is reversed in pygame
# Negative cross_product means it a left turn
return cross_product < 0
```

5 Testy

5.1 Plan testowania

Aby potwierdzić, że aplikacja poprawnie, przeprowadzone zostaną testy bazy danych jak i samego algorytmu wyszukiwania najkrótszej ścieżki. Na te testy będzie składać się przygotowanie skryptów testujących wszystkie funkcje bazy, takie jak:

- Połączenie z bazą danych,
- Stworzenie tablicy,
- Wgranie danych do bazy,
- Wydobycie danych z bazy,
- Usunięcie tablicy z bazy,
- Zamknięcie połączania z bazą danych.

W ranach testu algorytmu, przeprowadzony zostanie eksperyment sprawdzający, czy algorytm faktycznie działa poprawnie według zadanego trybu działania i czy dodatkowo bierze pod uwagę karę wykonania lewoskrętu.

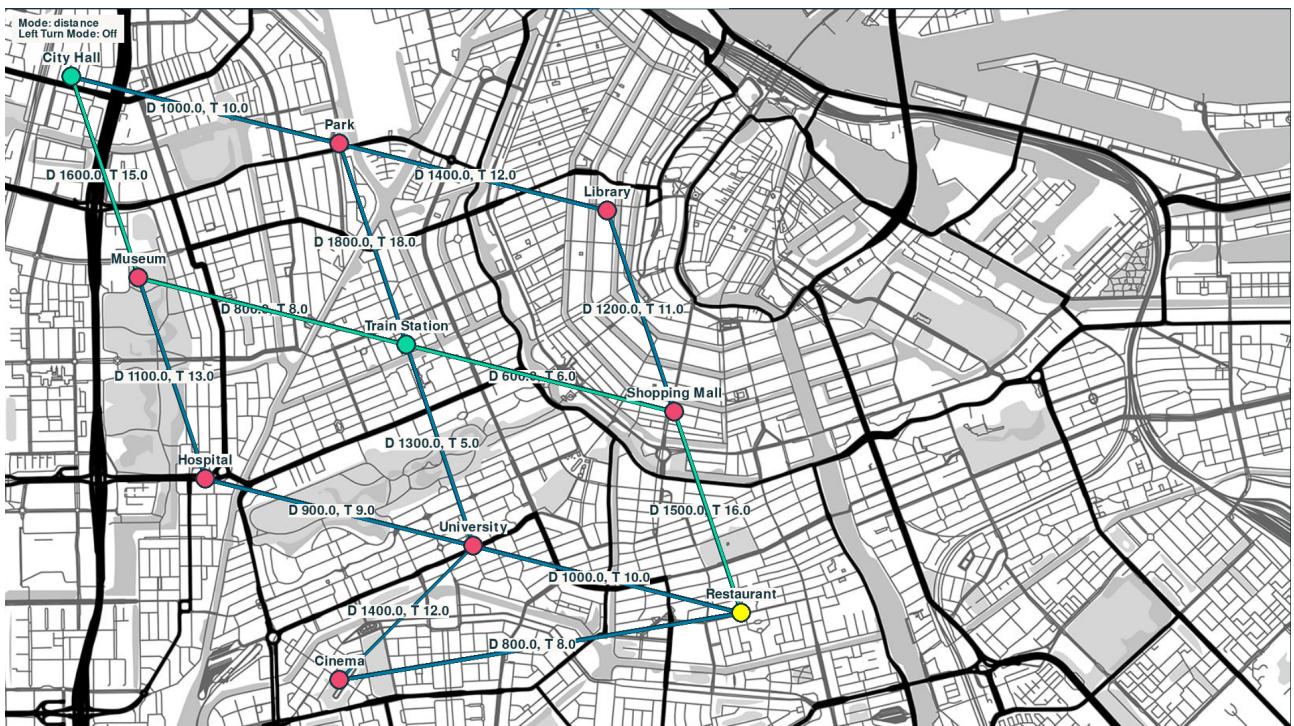
5.2 Baza danych

Prawidłowe działanie bazy danych i jej odpowiednią konfigurację sprawdzić można za pomocą skryptu *db_connectivity_test.py* zawartego w folderze *test*. Przykładowe, prawidłowe działanie skryptu pokazane jest poniżej:

```
PS C:\Users\latos\OneDrive\Desktop\SPDB-Maps_simulator> python3 .\test\scripts\db_connectivity_test.py
Executing function: connect_to_db ✓
Executing function: create_table ✓
Executing function: insert_into_table ✓
Executing function: select_from_table ✓
Executing function: drop_table ✓
Executing function: close_connection ✓
```

Rysunek 4: Wynik uruchomienia skryptu db_connectivity_test.py

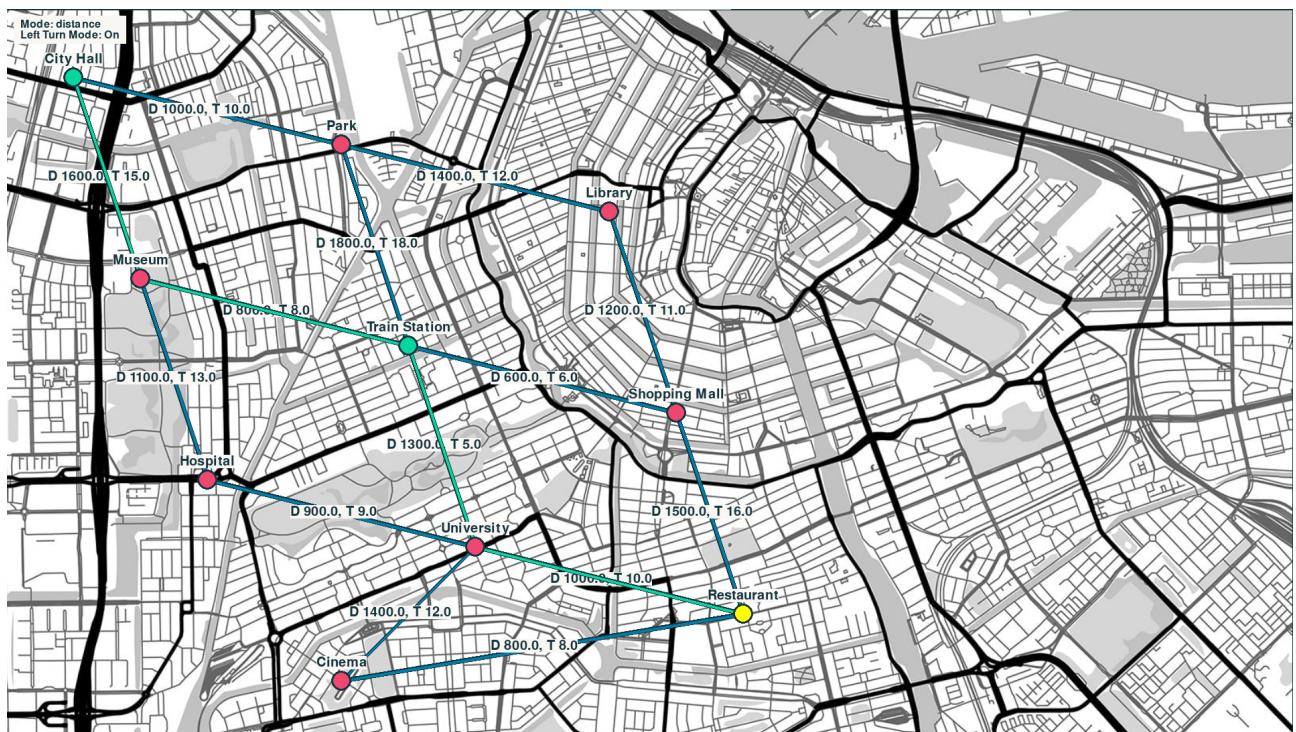
5.3 Testy działania algorytmu



Rysunek 5: Droga bez uwzględnienia kary lewoskrętu

Dla testu, rozważmy drogę "Restaurant > Train Station > City Hall w przypadku trybu "distance", czyli tak jak jest to przedstawione na rysunku 6. Najkrótsza ścieżka z "Restaurant" do "Train Station" prowadzi przez "Shopping Mall" i ma koszt 2100 j.m. Dobra wydaje się też droga przez "University", niestety ma ona 2300 j.m. co czyni ją dłuższą. Kontynuując trasę do "City Hall" porażamy przez "Muzeum" z kosztem 2400 j.m. Dobrą drogą zdaje się także droga przez "Park", niesie ona jednak koszt 2800 j.m. Wygląda na to, że algorytm wykrył najlepszą trasę.

Rozważmy przypadek, w którym wprowadzamy karę za lewoskręt wynoszącą 350 j.m. tak jak to zrobiono na rysunku 5. Tutaj trasa choć porównywalna została zmieniona w następujący sposób: Z "Restaurant" do "Train Station" przepieszczamy się przez "University" z kosztem 2300 j.m. dlatego że droga przez "Shopping Mall" jest obarczona dodatkowym kosztem lewoskrętu, ta



Rysunek 6: Droga z uwzględnieniem kary lewoskrętu

droga w tym przypadku ma cenę 2450 j.m. Droga z "Train Station" do "City Hall" nie zmieniła się, bo mimo potrzeby lewoskrętu w stronę "Museum" to cena poprzedniej drogi zwiększyła się do 2750 j.m. co nie jest większe drogi przez "Park" która zwiększyła się do 3150 j.m.

6 Literatura

- [1] Bartosz Latosek and Mateusz Krakowski. Spdb-maps_simulator. https://github.com/NewtonEiro/SPDB-Maps_simulator, 2024.