Jakub Kosmydel
Norbert Morawski
Bartłomiej Wiśniewski
Przemysław Węglik

# Badania operacyjne
# Projekt

12 maja 2023

# Spis treści

# 1. Wstęp

Celem naszego projektu jest znalezienie optymalnych tras linii dla autobusów, aby maksymalizować liczbę pasażerów, przy minimalnej liczbie linii autobusowych. Aby to osiągnąć, wykorzystywane są algorytmy genetyczne - algorytmy przeszukujące przestrzeń rozwiązań, które opierają się na procesie działania mechanizmu dziedziczenia biologicznego.

W systemie założono, że pozycje oraz popularność przystanków są z góry ustalone. Stosowanie algorytmów genetycznych pozwoliło na wygenerowanie zestawu najlepszych połączeń autobusowych, które można skonfigurować dla lepszego wykorzystania zasobów oraz zwiększenie korzyści z transportu publicznego dla pasażerów.

# 2. Opis zagadnienia

## 2.1. Sformułowanie problemu

Naszym celem w projekcie jest zaprojektowanie sieci linii autobusowych pokrywającej dany obszar miejski, który już posiada sieć przystanków autobusowych. Linie te, powinny mieć możliwość obsłużenia jak największej liczby pasażerów, tworząc jak najmniej postojów oraz zatrzymując się na jak najmniejszej liczbie przystanków.

## 2.2. Model matematyczny

### 2.2.1. Założenia

1. Przystankom przypisujemy ilość punktów w zależności od gęstości zaludnienia w pobliżu oraz ciekawych punktów (teatr, park itp.).
   — Dla każdego przystanku obliczamy liczbę ludzi w pobliżu,
   — Głównym punktom w Krakowie (np. D17, teatry, itp.) nadajemy wartość punktową,
   — Dla każdego przystanku sumujemy powyższe wartości.
2. Rozkładamy linie komunikacyjne po mieście tak, by maksymalizować sumę zebranych punktów przez wszystkie linie.
3. Wprowadzamy koszt dla linii: koszt ścieżki w grafie, po której jedzie + koszt utworzenia nowej linii.
4. Linie przebiegające przez jeden przystanek dzielą się punktami,
5. Maksymalizujemy sumę punktów zebranych przez wszystkie linie.

### 2.2.2. Dane

1. $n$ - liczba linii
2. $m$ - liczba przystanków

**Graf**

1. Wierzchołki to istniejące przystanki z przypisanymi punktami,
2. $p(j)$ - wartość punktowa przystanku:
   — W początkowej wersji liczba ta jest określona z góry,
   — $p(j) = \sum_{i=0}^{n-1} \frac{w_{j,i}}{f(d_{j,i})}$ gdzie $w_{j,i}$ to wartość obiektu (np. liczba mieszkańców w pobliżu) a $d_{j,i}$ to odległość tego bloku od przystanku, f – funkcja skalująca.
   — Funkcja liczona dla danego przystanku $j$
3. Krawędzie to połączenia między przystankami.
4. Koszt krawędzi to odległości między przystankami.

### 2.2.3. Szukane

$x_{i,j}$ - czy linia $i$ zatrzymuje się na przystanku $j$, gdzie:

1. $i \in [0, n-1]$
2. $j \in [0, m-1]$

### 2.2.4. Hiperparametry

1. $\alpha$ - koszt zatrzymania się na przystanku,
2. $\beta$ - koszt nowej linii,
3. $R$ - hiper parametr zbiegania.

### 2.2.5. Funkcja kosztu

$$l_j = \sum_{i=0}^{n-1} x_{i,j} \qquad \text{liczba linii zatrzymujących się na przystanku } j$$

$$q_j = \frac{p_j \cdot (1 + \frac{R}{l_j})^{l_j}}{l_j} \rightarrow \frac{e^R}{l_j} \qquad \text{ile punktów każda linia uzyskuje z przystanku } j$$

$$S_i \qquad \text{długość ścieżki linii } i \text{ w grafie}$$

$$cost_j = \begin{cases} \sum_{i=0}^{n-1} x_{i,j} \cdot (q_j - \alpha) & l_j > 0 \\ -\Delta & l_j = 0 \end{cases} \qquad \text{penalizacja nieodwiedzonych przystanków}$$

$$f(x) = \sum_{j=0}^{m-1} cost_j - \sum_{i=0}^{n-1} [S_i - \beta] \qquad \text{funkcja kosztu}$$

# 3. Opis algorytmów

Nasz problem rozwiązywaliśmy algorytmami genetycznymi.

## 3.1. Reprezentacja środowiska

Jak już zostało wspomniane, zajmowaliśmy się problemem optymalizacji istniejącej sieci komunikacyjnej, bez tworzenia nowych połączeń.

### 3.1.1. Reprezentacja mapy

Mapa z przystankami jest reprezentowana jako ważony graf z biblioteki NetworkX.

### 3.1.2. Reprezentacja genotypu

Genotyp składa się z listy linii autobusowych:

```
class Genotype:
    def __init__(self, lines: list[Line]):
        self.lines = lines
```

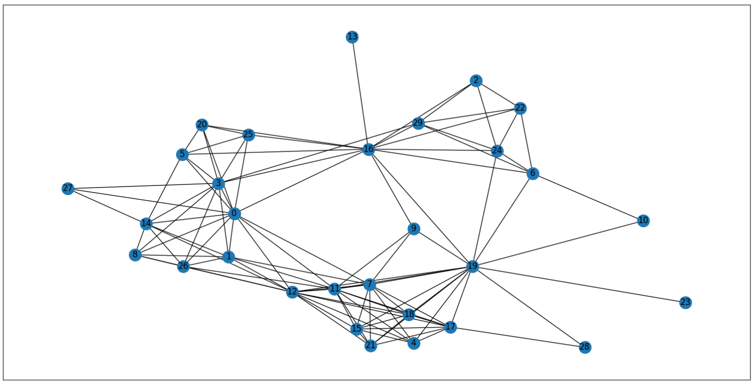
### 3.1.3. Reprezentacja linii

Linia posiada następujące parametry:
1. id - id linii,
2. stops - przystanki, na których się zatrzymuje,
3. edges - wszystkie krawędzie, przez które linia przejeżdża,
4. edge_color - kolor linii; do reprezentacji graficznej,
5. edge_style - styl krawędzi linii; do reprezentacji graficznej,

```
class Line:
    def __init__(self, stops: list[int], best_paths):
        self.id = Line.get_next_id()
        self.stops = stops  # ordered list of stops
        self.edges = []
        self.edge_color = [...]
        self.edge_style = [...]
```

## 3.2. Rozwiązanie początkowe

Na początku, chcąc się skupić na realizacji algorytmu, wygenerowaliśmy losowo sieć połączeń. Powstała ona przez wygenerowanie N punktów na płaszczyźnie, a następnie połączeniu ich między sobą z pewnym prawdopodobieństwem. Dawało to całkiem dobre rezultaty:



Rysunek 3.1. Przykładowa wygenerowana mapa

## 3.3. Symulacja

**Algorytm 1** Symulacja

**function** SYMULUJ(liczba_pokoleń, x)
    populacja = POPULACJA_POCZĄTKOWA( )
    ZAPISZ_POPULACJĘ( )
    **for** $i \leftarrow 0$ **to** liczba_pokoleń $-$ 1 **do**
        populacja = USUŃ_PUSTE(populacja)
        populacja_dopasowanie = FITNESS(populacja)
        populacja = FUNKCJA_PRZETRWANIA(populacja, populacja_dopasowanie)
        populacja_nowa = NOWA_POPULACJA(populacja)    ▷ Tutaj zachodzą mutacje i krzyżowania
        Co x epok:
        ZAPISZ_POPULACJĘ( )
    **end for**
**end function**

Powyżej przedstawiony został podstawowy silnik symulacji. W każdej epoce wykonuje on następujące kluczowe czynności:
— Usuwa niedopuszczalne rozwiązania (linie bez przystanków, organizmy bez linii),
— Oblicza funkcję dopasowania,
— Uruchamia funkcję przetrwania, która likwiduje wybrane osobniki,
— Uruchamia funkcję nowej populacji, która dokonuje mutacji i krzyżowań.

Na tym poziomie nie definiujemy co dana funkcja robi. Zostało to zrobione poniżej.

## 3.4. Selekcja

Została przez nas zaimplementowana najprostsza funkcja zostawiająca 1/5 najlepszych osobników.

## 3.5. Mutacja

### 3.5.1. LineMutator

Tworzy nowe mutacje dla danej linii.
Możliwe mutacje:

1. rotation_to_right
2. cycle_rotation
3. invert - odwraca kolejność przystanków, pomiędzy losowymi indeksami start oraz end,
4. erase_stops - losowo usuwa zadaną liczbę przystanków z linii,
5. add_stops - losowo dodaje zadaną liczbę przystanków, spośród tych, które w linii nie występują.

### 3.5.2. GenotypeMutator

Możliwe mutacje:

1. erase_line - tworzy nowy genotyp, usuwając losową linię,
2. create_line - tworzy nowy genotyp, dodając losowo wygenerowaną linię,
3. split_line - tworzy nowy genotyp, rozdzielając losową, losową linię dwie różne.
4. merge_lines - tworzy nowy genotyp, łącząc w losowej kolejności zadaną liczbę losowych linii,
5. cycle_stops_shift - ???

## 3.6. Krzyżowanie

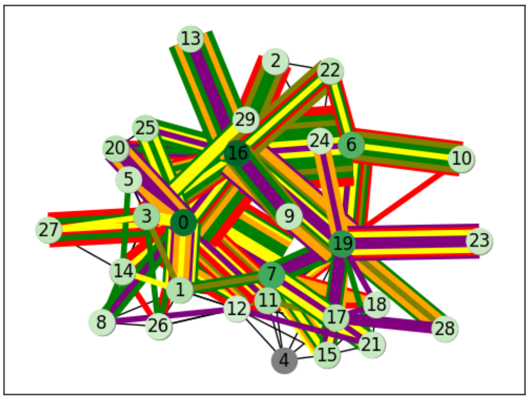
### 3.6.1. GenotypeCrosser

1. merge_genotypes - tworzy nowy genotyp, łącząc losową liczbę losowych linii z dwóch danych genotypów,
2. cycle_stops_shift - ???

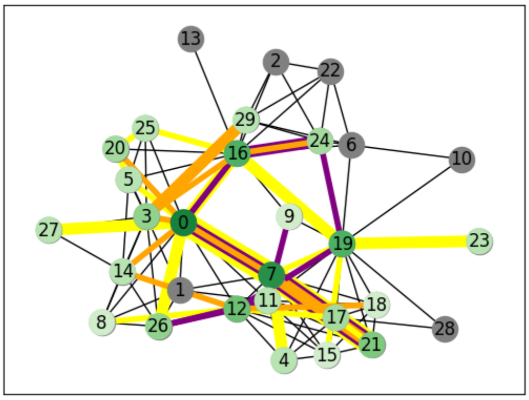
# 4. Aplikacja

# 5. Eksperymenty

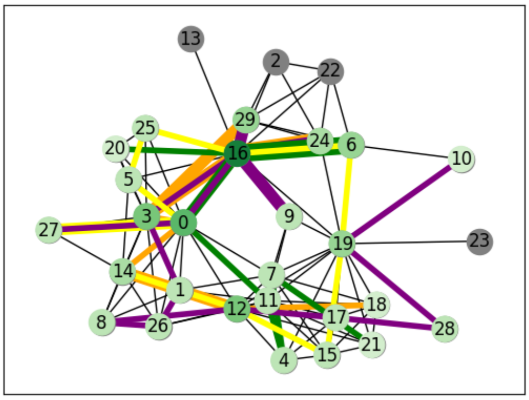
## 5.1. Eksperyment z losowym stosowaniem mutacji i krzyżowań



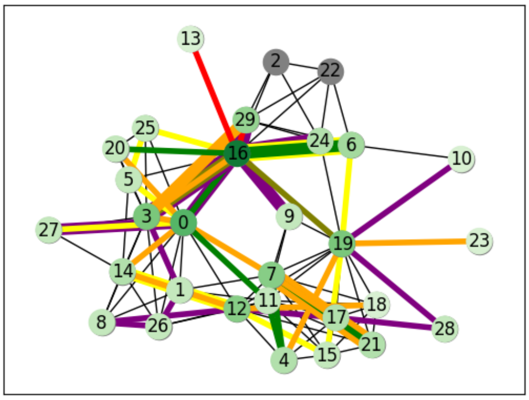

Rysunek 5.1. Populacja 0
dopasowanie −121.46




Rysunek 5.2. Populacja 3
dopasowanie 1.71

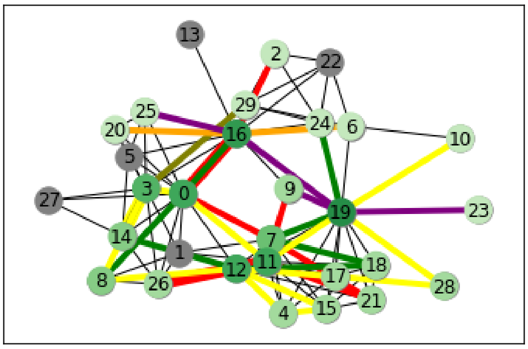



Rysunek 5.3. Populacja 5
dopasowanie 11.08

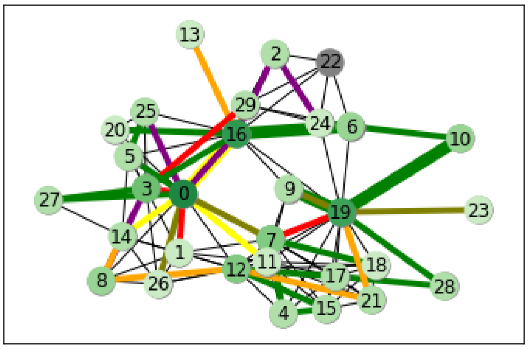



Rysunek 5.4. Populacja 10
dopasowanie 14.40

Jak widzimy, już po 10 epokach sieć połączeń znacznie się wyklarowała. Funkcja dopasowania wzrosła znacząco od generacji 0 do 10.

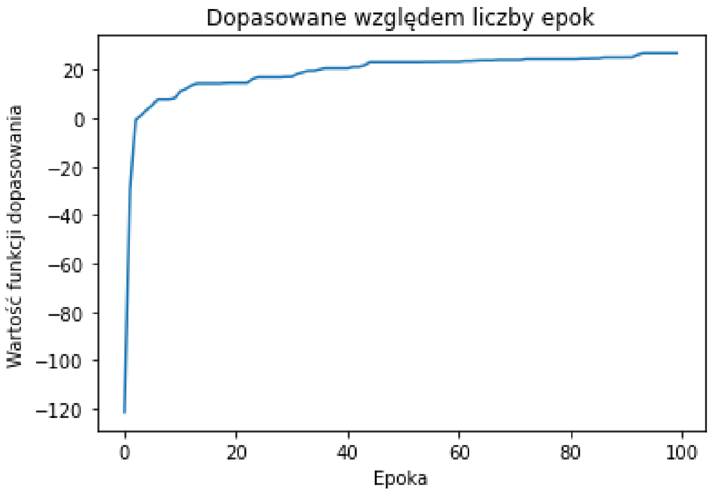


Rysunek 5.5. Populacja 20
dopasowanie 17.18



Rysunek 5.6. Populacja 100
dopasowanie 24.64

Sieć pokryła jeszcze więcej przystanków. Tempo wzrostu funkcji dopasowania zmalało.




Rysunek 5.7. Wykres funkcji dopasowania

Jak widać, rzeczywiście tempo dopasowywania się modelu znacznie spada w poźniejszych etapach symulacji.

# 6. Podsumowanie

Problem generowania linii autobusowych jest bardzo skomplikowany. W celu jego rozwiązania, przydatne są algorytmy genetyczne. Z odpowiednią liczbą nowych generacji jesteśmy w stanie osiągnąć ciekawe wyniki. Nie są one jednak w pełni satysfakcjonujące.