



**AGH**

**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W  
KRAKOWIE**

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,  
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

Praca dyplomowa

*Projekt i implementacja systemu rekomendacji miejsc do  
parkowania w usłudze współdzielenia samochodów.*

*Design and implementation of a parking space  
recommendation system in the car sharing service*

Autor:

*Paweł Hanzlik*

Kierunek studiów:

*Informatyka*

Opiekun pracy:

*dr. hab. inż. Radosław Klimek prof. uczelni*

Kraków, 2022

*Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.*

*Podziękowania dla prof. Radosława  
Klimka za poświęcony mi czas i meryto-  
ryczne wsparcie*



# Spis treści

<b>1. Wprowadzenie</b>	9
1.1. Możliwy sposób udoskonalenia systemu Car-sharingu	9
1.1.1. System rekomendacji miejsc parkingowych	10
1.1.2. Przykład	10
1.1.3. Ograniczenia	10
1.2. Czynniki wpływające na działanie systemu	10
1.3. Metody wyliczania wskaźników danej strefy	11
1.4. Cel pracy	11
<b>2. Architektura systemu</b>	13
2.1. Wzorzec Model-View-Controller	13
2.2. Diagram klas architektury	14
2.3. Szczegóły warstwy Model	14
2.3.1. Diagram bazy danych	15
2.4. Szczegóły warstwy Controller	16
2.5. Szczegóły warstwy View	16
2.5.1. Komponent main	17
2.5.2. Komponent city	17
2.5.3. Komponent form	18
2.5.4. Komponent result	18
<b>3. Teoretyczne wiadomości dotyczące problemu SAT</b>	19
3.1. Problem spełnialności ( <i>ang. SAT - Satisfiability</i> ) [2]	19
3.1.1. Elementarne pojęcia	19
3.1.2. Sposoby zapisywania formuł zdaniowych [3]	20
3.1.3. Klasy złożoności obliczeniowej problemów	20
3.2. Klasy problemu SAT	21

3.2.1.	Podział na podklasy k-SAT [4] .....	21
3.2.2.	SAT Solver .....	21
3.3.	Zagadnienie Max-SAT .....	22
3.3.1.	Obliczanie przykładowego problemu Max-SAT .....	23
3.4.	Zagadnienie Weighted Max-SAT .....	23
3.4.1.	Obliczanie przykładowego problemu Weighted Max-SAT .....	23
<b>4.</b>	<b>Zaprojektowanie silnika wnioskowania logicznego do rekomendacji miejsc parkingowych jako Ważony Max-SAT Solver</b> .....	<b>25</b>
4.1.	Struktura obszaru miejskiego .....	25
4.2.	Zmienne zdaniowe .....	27
4.2.1.	Proces budowania zmiennych .....	27
4.2.2.	Przykładowe wartościowanie parkingu .....	28
4.3.	Klauzule .....	28
4.3.1.	Proces budowanie klauzul .....	28
4.3.2.	Przykład generowania formuły .....	32
4.4.	Interpretacja odpowiedzi solvera .....	33
4.5.	Wyszukiwanie najlepszych parkingów .....	33
<b>5.</b>	<b>Biblioteka SAT4J w projekcie</b> .....	<b>35</b>
5.1.	Format wejścia .....	35
5.1.1.	Zapisanie problemu SAT jako DIMACS [5] .....	35
5.1.2.	Przykładowa formuła w formacie DIMACS .....	36
5.1.3.	Zapisanie problemu Weighted Max-SAT jako DIMACS .....	36
5.1.4.	Przykładowa formuła Weighted Max-SAT w formacie DIMACS .....	37
5.2.	Użycie biblioteki SAT4J w kodzie aplikacji .....	37
5.2.1.	Tworzenie Solvera oraz reprezentacja formuł logicznych w kodzie .....	37
5.2.2.	Implementacja SAT Solvera [6] .....	38
5.2.3.	Implementacja Ważonego Max-SAT Solvera .....	39
<b>6.</b>	<b>Implementacja systemu</b> .....	<b>41</b>
6.1.	Generowanie danych o położeniu parkingów i stref na obszarze miejskim .....	41
6.2.	Wybór miasta przez klienta .....	42
6.3.	Wyszukiwanie miejsca parkingowego .....	42
6.3.1.	Klasa przechowująca dane parkingu i jego score .....	42

6.4. Przyjmowanie preferencji klienta i tworzenie solvera.....	43
6.5. Wykorzystanie solvera.....	44
6.5.1. Max-SAT Dekorator i ustawienie liczby klauzul oraz zmiennych .....	44
6.5.2. Max-SAT Dekorator i ustawienie liczby klauzul oraz zmiennych .....	44
6.5.3. Sprawdzanie spełnialności formuły .....	45
6.5.4. Proces punktowania parkingów .....	45
<b>7. Weryfikacja działania systemu .....</b>	<b>47</b>
7.1. Generowanie danych .....	47
7.2. Użytkownik A - Kraków .....	49
7.3. Użytkownik B - Kraków .....	52
7.4. Użytkownik A - Warszawa .....	55
7.5. Użytkownik B - Warszawa .....	58
<b>8. Podsumowanie.....</b>	<b>61</b>
8.1. Wnioski .....	61
8.2. Możliwe rozszerzenia systemu.....	62





# 1. Wprowadzenie

Firmy zajmujące się wypożyczaniem samochodów, takie jak Traficar, wprowadzają coraz to nowsze aplikacje zapewniające szereg funkcji ułatwiających interakcję klienta z procesem Car-sharingu [1]. Obejmują one między innymi możliwość płatności bezgotówkowych, prosty sposób otwierania i uruchamiania samochodu za pomocą skanera kodu QR ale również oferowanie mapy z pokazanymi wolnymi miejscami do parkowania czy też odciążenie użytkownika z obowiązku płacenia za postój, gdyż jest to wliczane w koszt wypożyczenia auta. Tak zaprojektowana aplikacja przyciąga rzeszę ludzi i dzięki temu firma może się rozwijać. Jednakże rodzi się również problem dotyczący dostępności samochodów na wynajem. Rosnące zainteresowanie tym rodzajem komunikacji to jeden z czynników lecz większy wpływ na to ma dystrybucja pojazdów. Większość klientów nie zważa na miejsce pozostawienia wypożyczonego auta mając w głowie jedynie potrzebę znalezienia parkingu jak najbardziej ich odpowiadającego. Firmy jednak muszą myśleć przede wszystkim o zmniejszeniu kosztów swojej działalności oraz kreowaniu jak najlepszego środowiska dla swoich użytkowników.

## 1.1. Możliwy sposób udoskonalenia systemu Car-sharingu

Kluczowym elementem są oczywiście koszty samego przedsięwzięcia, które koncerny chcą zmniejszać w celu osiągnięcia wyższych zysków. Najprostszym rozwiązaniem wydaje się oddelegowanie pracowników dystrybuujących samochody po całym obszarze miejskim lecz to generuje konieczności opłacania pensji, koszty paliwa oraz chwilową niedostępność niektórych samochodów na wynajem.

### 1.1.1. System rekomendacji miejsc parkingowych

O wiele lepszym pomysłem zdaje się być moduł aplikacji umożliwiający nakierowywanie kierowców aby oni sami pozostawiali pojazdy w odpowiednich miejscach. Korporacje w swoim interesie będą mogły starać się więc zachęcać użytkownika do postoju w strefach wzmożonego zapotrzebowania na samochody do wynajęcia, zostawiając tym samym klientowi możliwość zaparkowania na wciąż korzystnym dla niego miejscu jak i pozwolić mu na dobór odpowiadających mu cech samego parkingu.

### 1.1.2. Przykład

Weźmy pod uwagę większe obszary miejskie takie jak Kraków czy Warszawa. Podzielone one są na dzielnice o zróżnicowanym stopniu urbanizacji i zaludnienia. Większość obrzeży stanowią obszary mieszkalne, zaś centrum to strefy ekonomiczne, tereny turystyczne oraz miejsca pracy mieszkańców. Na tej podstawie jasno można wyciągnąć wniosek, że te właśnie miejsca będą miały większy współczynnik zapotrzebowania na samochody do wypożyczenia. Tak więc użytkownik podając swój adres docelowy znajdujący się w strefie mniejszego popytu proszony będzie o zostawienie pojazdu na parkingu znajdującym się w innej, bardziej obleganej części miasta.

### 1.1.3. Ograniczenia

Należy jednak pamiętać o swego rodzaju ograniczeniach takiego rozwiązania. Nie można rekomendować stref znajdujących w zupełnie innej lokalizacji, nawet mających wyższe zapotrzebowanie, ponieważ jest to niemożliwe do realizacji. Trzeba więc założyć, że klient nadal będzie znajdował się relatywnie blisko celu. W ramach rekompensaty zaś za to niewielkie unieogodnienie, firma może zaoferować rabaty na przyszłe wynajmy.

## 1.2. Czynniki wpływające na działanie systemu

Główną zmienną zdecydowanie jest wskaźnik zapotrzebowania danej strefy na samochody. Dodatkowo można wprowadzić wskaźniki atrakcyjności, faworyzujące części miasta zlokalizowane wokół często uczęszczanych przez ludzi miejsc, a także miarę masycenia danego obszaru, aby nie doprowadzić do zebrania prawie wszystkich aut w jedną strefę. Preferencje użytkownika będą traktowane drugorzędnie, jako, że priorytet stanowi odpowiednie rozmieszczenie floty pojazdów, którą dysponuje firma tak, aby jak najbardziej ułatwić wynajem aut innym użytkownikom.

### 1.3. Metody wyliczania wskaźników danej strefy

Korporacje w tym celu mogą monitorować, a następnie wykorzystywać informacje o tym, skąd najczęściej ludzi wypożycza samochody. W tym wypadku ważne jest także pora dnia i godzina, kiedy dokonano wynajmu. Z racji na brak możliwości pozyskania takich danych, w tej pracy wygenerowano wartości pseudolosowe. System nie przechowuje również danych o preferencjach klienta, dając mu za każdym razem możliwość swobodnego wyboru cech miejsca parkingowego.

### 1.4. Cel pracy

Celem pracy jest zaprojektowanie i implementacja systemu do Car-sharingu w inteligentnym obszarze miejskim. Aplikacja ma za zadanie uwzględniać wiele czynników, takich jak preferowanie stref o wysokim wskaźniku zapotrzebowania, różne cechy parkingów czy indywidualne upodobania klientów. System, którego silnik wnioskowania logicznego oparty został na ważonym SAT solverze powinien zwracać użytkownikowi listę kilku najlepszych dla niego miejsc parkingowych. Tak zaprojektowany, pozwoli firmie na lepsze zarządzanie i rozmieszczenie samochodów oraz zaoszczędzenie kosztów związanych z użyciem zasobów ludzkich, które należałoby wykorzystać zamiast systemu.



## 2. Architektura systemu

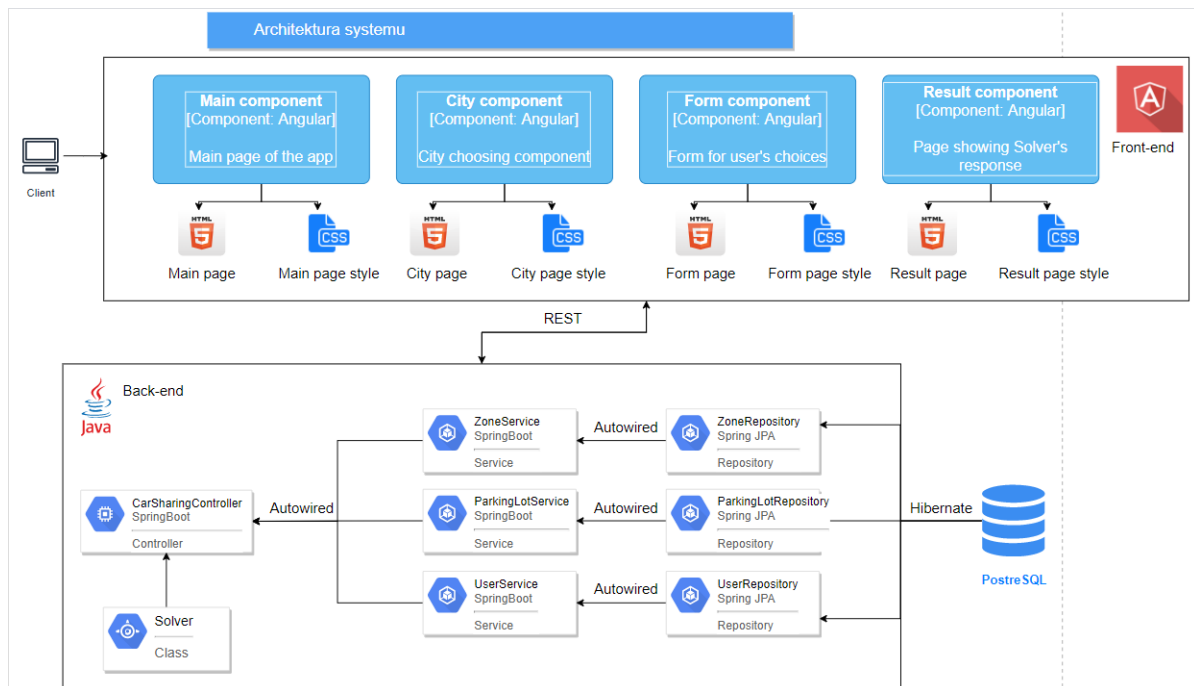
Rozdział ten przedstawia i opisuje architekturę systemu oraz poszczególnych modułów aplikacji.

### 2.1. Wzorzec Model-View-Controller

Struktura aplikacji oparta została o framework MVC służący do organizacji modułów systemu na trzy główne, połączone ze sobą części, z których każda z nich jest tworzona i rozwijana w różnych technologiach. Użycie takiego rozwiązania pozwala na ukrycie części logiki aplikacji przy jednoczesnym udostępnieniu użytkownikowi przyjaznego interfejsu graficznego, a także umożliwia wykorzystanie szeregu wspierających taki model technologii, dając programiście tym samym elastyczność w doborze narzędzi.

1. Element Model - to centralny komponent tego wzorca projektowego. Określa użyte w aplikacji struktury danych, ich atrybuty oraz relacje, jak i wszelką logikę związaną z obsługą rekordów w bazie.
2. Element View - odpowiada za całą logikę dotyczącą interfejsu użytkownika aplikacji. Ma za zadanie przysyłać działania użytkownika do kontrolera oraz odbierać i prezentować wyniki. Zazwyczaj jest to graficzny UI zawierający komponenty wizualizujące wynik pracy systemu oraz końcowy widok umożliwiający klientowi wchodzenie z nim w interakcję.
3. Element Controller - zachowuje się on jak interfejs łączący moduł Model z modulem View, zapewniający przepływ oraz obsługę przychodzących zapytań, manipulację danymi, przesyłanie odpowiedzi systemu do części Front-end, a także sterujący działaniem aplikacji.

## 2.2. Diagram klas architektury



**Rys. 2.1.** Diagram architektury systemu, pokazujący podział na część Back-endu zawierającą repozytoria, serwisy, kontroler oraz klasę Solvera, jak i część Front-endu zawierającą komponenty będące częścią UI.

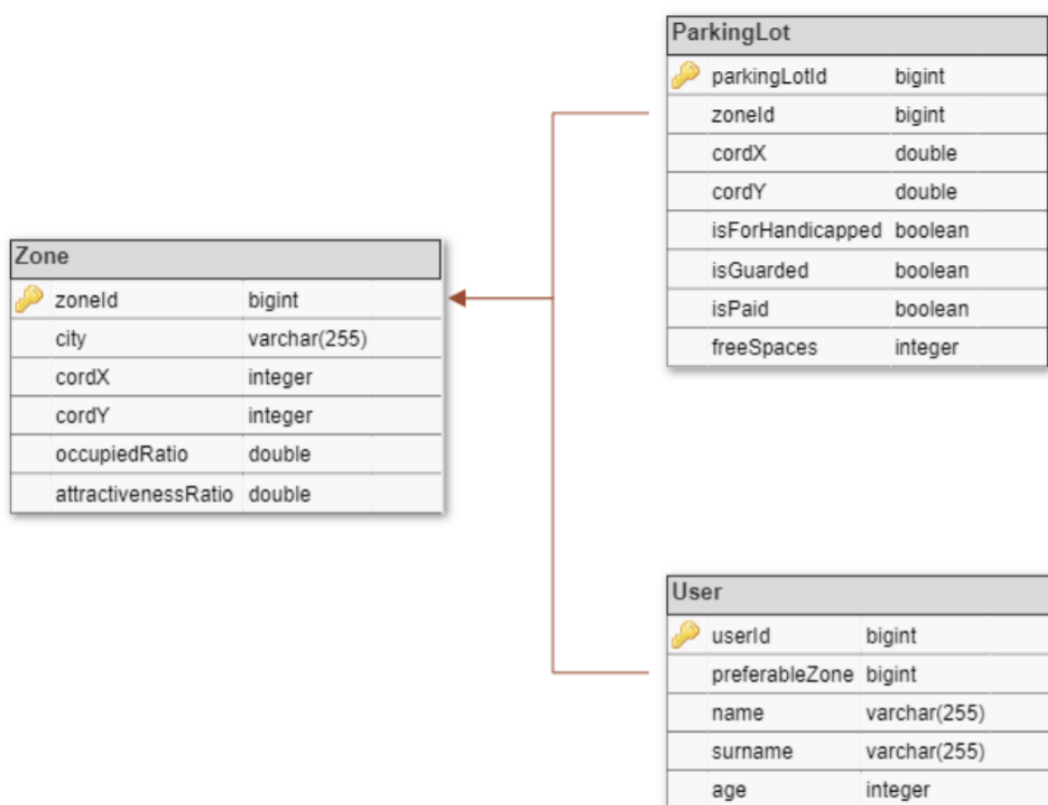
## 2.3. Szczegóły warstwy Model

Składa się ona z trzech tabel przechowywanych w relacyjnej bazie danych

1. Tabela ParkingLot zawiera informacje dotyczące szczególnych miejsc parkingowych na danym parkingu takich jak posiadanie specjalnych części postojowych dla niepełnosprawnych czy koszt pozostawienia samochodu. Oprócz tego przechowuje informacje o numerze strefy, w której się znajduje oraz jego współrzędnych.

2. Tabela Zone zawiera informacje o wartości współczynników zapotrzebowania na samochody w danej strefie oraz współczynnik jej atrakcyjności określany na podstawie znajdowania się w niej miejsc będących częstym celem użytkowników, tj. galerie handlowe bądź stacje komunikacji publicznej.
3. Tabela User posiada informacje o danych osobowych klientów aplikacji.

### 2.3.1. Diagram bazy danych



**Rys. 2.2.** Schemat bazy danych. Przedstawia wszystkie wykorzystywane w aplikacji encje, ich atrybuty oraz wyodrębnione relacje między nimi. Te oznaczone są czerwoną linią i reprezentują relację 1..N, gdzie strzałka występuje przy kluczu głównym, zaś drugi koniec klucz obcy. Symbol klucza w pierwszych wierszach tabel wskazuje na Primary Key tabeli

## 2.4. Szczegóły warstwy Controller

Warstwa ta składa się z następujących części:

1. CarSharingController - jest to klasa kontrolera, który komunikuje się z interfejsem użytkownika, wystawia API REST-owe oraz przyjmuje zapytania HTTP od użytkownika z jego preferencjami dotyczącymi parkingu. Na tej podstawie tworzy obiekt Solvera, którego wynik następnie zwraca na Front-end.

Wystawia on trzy punkty końcowe (*ang. endpointy*).

```
przyjmuje parametry od klienta aplikacji, tworzy obiekt Solvera  
i zwraca wynik  
GET /api/maxsat/sfps/cordXcordYuserChoices  
  
generuje strefy oraz parkingi  
GET /api/maxsat/GenerateData  
  
przypisuje wybrane przez użytkownika miasto  
GET /api/maxsat/AssignCity/city
```

### Listings 2.1. Wystawione endpointy w kontrolerze

2. Solver - klasa reprezentująca ważony MaxSat Solver będący głównym silnikiem systemu (jego struktura zostanie opisane w dalszych rozdziałach).
3. ZoneService, ParkingLotService, UserService - łączą one warstwę modelu z warstwą kontrolera, a także zapewniają szereg operacji typu CRUD (*ang. create, read, update, delete*) i modyfikujących bazę danych.
4. ZoneRepository, ParkingLotRepository, UserRepository - są to interfejsy zapewniające mapowanie obiektowo-relacyjne z warstwą modelu udostępniające metody pozwalające na wyszukiwanie oraz zmianę rekordów w bazie danych.

## 2.5. Szczegóły warstwy View

Ta część składa się z czterech głównych komponentów, z których każdy zawiera plik HTML kodujący wygląd strony internetowej, plik CSS modelujący jej styl oraz dwa pliki Typescript zawierające logikę działania.



### 2.5.1. Komponent main



**Rys 2.3.** Main-page - strona startowa aplikacji

Jest to startowa strona aplikacji, która ma na celu wprowadzenie klienta w system oraz wygenerowanie danych dotyczących stref i parkingów po nowym uruchomieniu programu.

### 2.5.2. Komponent city



**Rys 2.4.** City-page - strona do wyboru miasta

Ta strona pozwala użytkownikowi wybrać miasto, w którym się znajduje.

### 2.5.3. Komponent form

PARKING SPOT RECOMMENDATION SYSTEM

Insert destination location:


CordX  CordY

Do you want a parking spot for disabled people?  
☒ Yes ☐ No

Do you want a paid parking?  
☒ Yes ☐ No

Do you want a guarded parking spot?  
☒ Yes ☐ No

Do you want a parking with at least 15 spots?  
☒ Yes ☐ No

  
AGH

Design and implementation of a parking space recommendation system in the car sharing service

**Rys 2.5.** Form-page - strona do wyboru preferencji klienta

Ta strona pozwala użytkownikowi wskazać współrzędne, dokąd się udaje oraz dobrać odpowiadające jego potrzebom parametry miejsca parkingowego.

### 2.5.4. Komponent result

PARKING SPOT RECOMMENDATION SYSTEM

[ParkingId: 176 Score: 13, ParkingId: 123 Score: 12, ParkingId: 141 Score: 12]

  
AGH

Design and implementation of a parking space recommendation system in the car sharing service

**Rys 2.6.** Result-page - strona do przedstawiania odpowiedzi solvera

Ostania strona interfejsu użytkownika pokazująca listę trzech najlepszych parkingów.

### 3. Teoretyczne wiadomości dotyczące problemu SAT

W tym rozdziale przedstawiona zostanie teoretyczna strona problemu spełnialności formuł logicznych. Przytoczone będą informacje zarówno o podstawowych pojęciach jak i stopniowo bardziej złożonych dążąc na końcu do przytoczenia istoty opisywanej pracy, tj. ważonego Max-SAT solvera.

#### 3.1. Problem spełnialności (*ang. SAT - Satisfiability*) [2]

Występuje on zarówno w logice matematycznej jak i w informatyce. Polega on w głównej mierze na określeniu czy istnieje określone podstawienie pod zmienne zdaniowe by dana formuła była prawdziwa. Wtedy formuła uznana zostaje za spełnialną. Problem ten uznawany jest jako NP (*ang. nondeterministic polynomial*) zupełny, a to oznacza brak konkretnego algorytmu rozwiązującego ten problem w sposób efektywny. W kolejnych podrozdziałach bardziej przybliżona zostanie struktura SAT oraz szereg pojęć z nim związanych.

##### 3.1.1. Elementarne pojęcia

1. Zdanie - to wyrażenie logiczne będące przedmiotem analizy jego prawdziwości. Przypisuje mu się logiczną wartość '0' gdy jest określone jako fałszywe bądź '1' gdy zdanie jest prawdą.
2. Zmienna zdaniowa - jest to najprostszy symbol reprezentujący dowolne zdanie lub traktowany jako formuła atomowa. Oznaczana małymi literami np. ("p", "q") często również z dolnymi indeksami. W procesie wartościowania przyporządkowana zostaje jej cyfra 1 bądź 0.

3. Formuła zdaniowa - wyrażenie składające się z jednej bądź wielu zmiennych zdaniowych lub ich negacji oraz operatorów logicznych takich jak alternatywa ( $\vee$ ) czy koniunkcja ( $\wedge$ ).
4. Klauzula - formuła logiczna złożona ze zmiennych zdaniowych zespolonych spójnikiem alternatywy np.

$$\neg p \vee q \vee \neg r \vee s$$

5. Klauzula dualna - klauzula, w której alternatywy zastąpione zostają koniunkcjami.

$$p \wedge \neg q \wedge \neg r \wedge s$$

### 3.1.2. Sposoby zapisywania formuł zdaniowych [3]

Wyróżnia się 3 główne postacie normalne zapisu form zdaniowych. Są to:

- CNF - Koniunkcyjna postać normalna (ang. *Conjunctive Normal Form*) oznacza zapis wyrażenia jako koniunkcja klauzul.

$$(\neg p \vee q) \wedge (\neg a \vee b)$$

- DNF - Dysjunkcyjna postać normalna (ang. *Disjunctive Normal Form*) oznacza zapis wyrażenia jako alternatywa klauzul dualnych.

$$(p \wedge \neg q) \vee (a \wedge \neg b)$$

- INF - Implikacyjna postać normalna (ang. *Implicative Normal Form*) to formy zdaniowe składające się z literałów połączonych implikacjami, które są połączone implikacjami, a ostatni krok to wynikanie lewej strony z fałszu.

$$q \Rightarrow (\neg p \Rightarrow \neg q \Rightarrow p \Rightarrow \perp) \Rightarrow (p \Rightarrow q \Rightarrow \perp) \Rightarrow \perp$$

### 3.1.3. Klasy złożoności obliczeniowej problemów

- Klasa problemów P (ang. *deterministic polynomial*) - reprezentuje on zbiór problemów decyzyjnych dających się rozwiązać w wielomianowym czasie.
- Klasa problemów NP (ang. *nondeterministic polynomial*) - reprezentuje on zbiór problemów decyzyjnych, którego działanie w przypadku otrzymania odpowiedzi "tak" można zweryfikować w wielomianowym czasie.

- Klasa problemów NP-zupełnych (*ang. NP-complete*) - reprezentuje on zbiór problemów decyzyjnych, które należą do klasy NP oraz można w wielomianowym czasie zredukować do niego dowolny inny problem NP. Nie można jednoznacznie stwierdzić czy istnieje dla nich rozwiązanie w czasie wielomianowym.
- Klasa problemów NP-trudnych (*ang. NP-hard*) - reprezentuje on zbiór problemów decyzyjnych, których rozwiązanie jest przynajmniej tak trudne jak rozwiązanie problemów klasy NP-zupełnych. Nie może zostać rozwiązany w wielomianowym czasie.

## 3.2. Klasy problemu SAT

Spełnialność formuł jest rozstrzygalna, jednak pesymistyczny przypadek zakłada wykładniczą złożoność obliczeniową. Wyróżnia się więc podklasy problemu o niższym bądź równym czasie wykonywania algorytmu.

### 3.2.1. Podział na podklasy k-SAT [4]

Są to problemy zapisane w koniunkcyjnej formie normalnej, których klauzule mają co najwyżej  $k$  literałów. Poprzez literał rozumiana jest zmienna zdaniowa bądź zmienna zanegowana.

1. 1-SAT - to wyrażenia zbudowane z klauzul mających pojedyncze zmienne zdaniowe. Zaliczane do klasy P. Przykładem takiej formuły może być:

$$\neg p \vee q \vee \neg r \vee s$$

2. 2-SAT - to wyrażenia zbudowane z klauzul mających dwie zmienne zdaniowe. Zaliczane do klasy P. Przykładem takiej formuły może być:

$$(\neg p \vee q) \wedge (\neg a \vee b)$$

3. 3-SAT - to wyrażenia zbudowane z klauzul mających trzy zmienne zdaniowe. Zaliczane do klasy NP. Przykładem takiej formuły może być:

$$(\neg p \vee q \vee \neg r) \wedge (\neg a \vee b \vee \neg c)$$

### 3.2.2. SAT Solver

SAT Solver to narzędzie służące do rozstrzygania spełnialności podanej mu formuły zdaniowej. Wykonuje on szereg podstawień wartości pod zmienne zdaniowe i sprawdza czy którekolwiek dało oczekiwany rezultat. Gdy takie istnieje Solver zwraca listę wartości zmiennych

zaś w innym przypadku przekazuje błąd o niespełnialności formuły. Dla lepszej wizualizacji posłużę się dwoma przykładami formuł logicznych i przedstawię możliwy output Solvera dla każdej z nich.

– Formuła spełnialna

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee q)$$

Podstawienie spełniające formułę:

p	q	$(p \vee q)$	$(\neg p \vee \neg q)$	$(\neg p \vee q)$	$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee q)$
0	0	0	1	1	0
0	1	1	1	1	1
1	0	1	1	0	0
1	1	1	0	1	0

**Tabela 3.1.** Wynik:  $(p, q) = (false, true)$

– Formuła niespełnialna

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)$$

p	q	$(p \vee q)$	$(\neg p \vee \neg q)$	$(\neg p \vee q)$	$(p \vee \neg q)$	$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee q)$
0	0	0	1	1	1	0
0	1	1	1	1	0	0
1	0	1	1	0	1	0
1	1	1	0	1	1	0

**Tabela 3.2.** Wynik: Podany problem SAT nierozwiązywalny

### 3.3. Zagadnienie Max-SAT

Max-SAT to rozbudowany koncept problemu SAT. Z racji, iż formuła nie zawsze musi być spełnialna, Max-SAT stara się zmaksymalizować ilość spełnionych klauzul w podanym wejściu. Ten rodzaj rozszerzenia jest już klasy NP-trudnej, ponieważ ciężko jest znaleźć przybliżone rozwiązania, które spełnia odpowiednią liczbę klauzul w zasięgu określonego stosunku przybliżenia do optymalnego rozwiązania.

### 3.3.1. Obliczanie przykładowego problemu Max-SAT

Patrząc na przytoczoną wcześniej formułę niespełnialną

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)$$

Max-SAT solver zwróciłby odpowiedź mówiącą o tym, że maksymalna liczba spełnionych klauzul wynosi 3. Przykładowe rozwiązanie to  $(p, q) = (true, false)$ , gdzie tylko klauzula  $(\neg p \vee \neg q)$  nie byłaby spełniona

## 3.4. Zagadnienie Weighted Max-SAT

Jest to z kolej rozbudowa problemu Max-SAT, w której oprócz samych klauzul i ich wartościowania, przypisujemy każdej z nich wagę najczęściej w postaci liczby naturalnej. Solver w tym wypadku nie znajduje rozwiązania spełniającego jak najwięcej klauzul lecz maksymalizuje wartość spełnionych wag.

### 3.4.1. Obliczanie przykładowego problemu Weighted Max-SAT

Pochylmy się raz jeszcze nad tą formułą przypisując tym razem wagę każdej klauzuli.

$$\begin{aligned} (p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \\ (p \vee q) \rightarrow [12] \\ (\neg p \vee \neg q) \rightarrow [5] \\ (\neg p \vee q) \rightarrow [15] \\ (p \vee \neg q) \rightarrow [9] \end{aligned}$$

Przy takich danych wejściowych rozwiązanie wyglądałoby tak:

$$(p, q) = (true, true)$$

Solver odrzuciłby bowiem klauzulę z najniższą wagą tj.  $(\neg p \vee \neg q)$





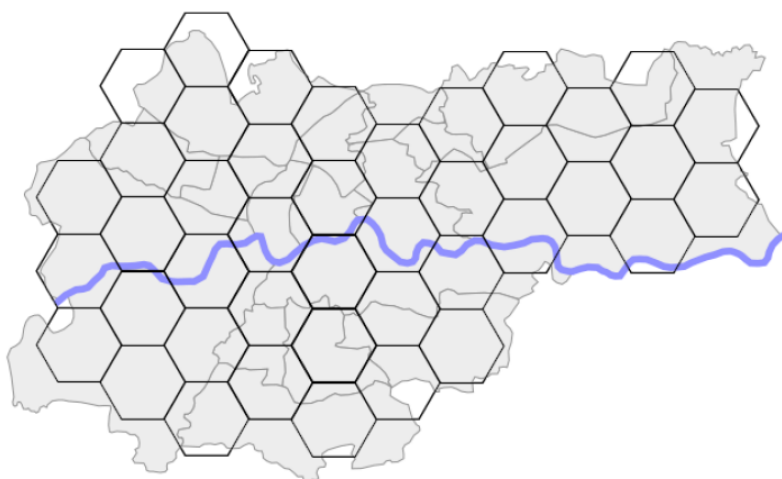
## **4. Zaprojektowanie silnika wnioskowania logicznego do rekomendacji miejsc parkingowych jako Ważony Max-SAT Solver**

Ten rozdział skupi się w głównej mierze na budowie systemu polecającego parkingi w inteligentnym obszarze miejskim. Do tego celu posłuży Weighted Max-SAT, który idealnie nadaje się do takiego zadania, ponieważ będzie wymagana obsługa także niespełnialnych formuł. Oczywiście jest, że nie zawsze istniał będzie idealny parking dla każdego użytkownika, w związku z czym system powinien wybrać ten najbliższy ideałowi. Działanie silnika problemu rekomendacji miejsc parkingowych można przedstawić w następujących krokach.

1. Utworzenie odpowiedniej ilości zmiennych zdaniowych w zależności od danych pobranych od użytkownika.
2. Zbudowanie klauzul złożonych ze stworzonych wcześniej literałów,
3. Analiza odpowiedzi Ważonego Max-SAT solvera oraz mapowanie jej na stosunek polecenia danego parkingu w celu wybrania najlepszych dla klienta aplikacji.

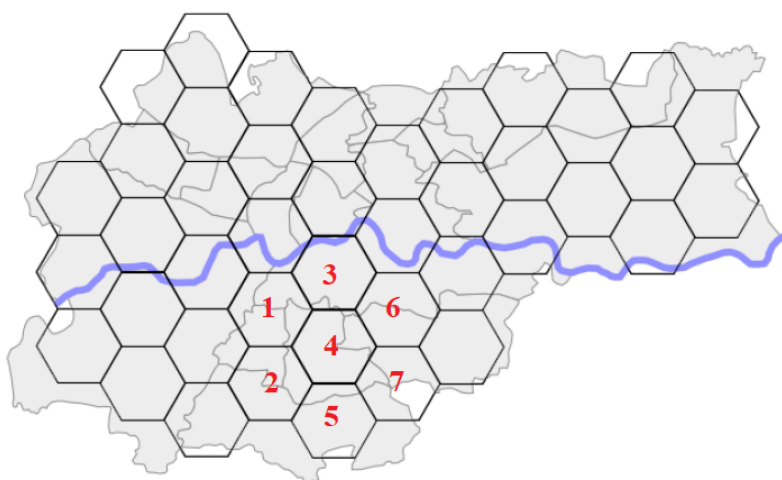
### **4.1. Struktura obszaru miejskiego**

Obszar miasta podzielony zostanie na heksagonalne strefy, z których każda posiadała będzie współczynnik zapotrzebowania na samochody.



**Listings 4.1.** Przykładowy obszar miejski podzielony na strefy

W taki sposób prezentuje się przykładowy obszar, dla którego wykonywać będziemy wyliczenia najkorzystniejszego dla użytkownika parkingu. Nie chcemy jednakże zmuszać go do drastycznej zmiany miejsca pozostawienia samochodu, dlatego przyjmiemy pewne ograniczenie. Mianowicie po wskazaniu przez klienta celu swej podróży, do obliczeń wykorzystamy jedynie tylko strefy przylegające do strefy wybranej przez użytkownika.



**Listings 4.2.** Strefy w otoczeniu użytkownika

Tak więc w większości przypadków silnik wybierał będzie parkingi tylko z siedmiu stref ulokowanych w pobliżu klienta. Dla stref skrajnych będą to minimalnie cztery strefy.

## 4.2. Zmienne zdaniowe

### 4.2.1. Proces budowania zmiennych

Pierwszym krokiem tworzenia Ważonego Max-SAT solvera jest określenia potrzebnych zmiennych zdaniowych oraz ich interpretacji. Ponieważ system opiera się w głównej mierze na parkingach oraz ich parametrach, w oczywisty sposób nasuwa się zbudowanie zmiennych zdaniowych opartych właśnie na tych danych. Należy pamiętać również, że muszą być one skonstruowane w taki sposób, aby możliwe było przypisanie im wartości logicznych '0' oraz '1' ze względu na konieczność wykonania na nich późniejszego procesu wartościowania. W celu stworzenia jak najlepszego systemu, wyszczególnionych zostało 11 zmiennych zdaniowych cechujących pojedynczy parking. Tak prezentuje się zestawienie zmiennych:

- S1 - parking znajduje się w strefie nr 1
- S2 - parking znajduje się w strefie nr 2
- S3 - parking znajduje się w strefie nr 3
- S4 - parking znajduje się w strefie nr 4
- S5 - parking znajduje się w strefie nr 5
- S6 - parking znajduje się w strefie nr 6
- S7 - parking znajduje się w strefie nr 7
- S8 - parking posiada miejsca dla niepełnosprawnych
- S9 - parking jest płatny
- S10 - parking jest strzeżony
- S11 - na parkingu jest przynajmniej 15 wolnych miejsc parkingowych

W przypadku gdy strefa znajduje się na obrzeżach terenu miejskiego, pominięta zostanie odpowiednia liczba zmiennych zdaniowych tak, by dopasować ją do faktycznej liczby sąsiadujących z klientem stref. Naturalnie również parking nie może znajdować się w więcej niż jednej strefie jednocześnie, a same zmienne zdaniowe zbudowane w ten sposób służą pomocniczo do wyboru przez solver wyłącznie najlepszej dla użytkownika.

### 4.2.2. Przykładowe wartościowanie parkingu

Rozważmy istnienie parkingu o następujących parametrach. Znajduje się w strefie o numerze 2, posiada miejsca dla niepełnosprawnych, jest bezpłatny, a przy tym strzeżony lecz nie zostało na nim przynajmniej 15 miejsc do postoju. Taki parking będzie reprezentowany przez następujące wartościowanie zmiennych zdaniowych :

```
S1 = 0
S2 = 1
S3 = 0
S4 = 0
S5 = 0
S6 = 0
S7 = 0
S8 = 1
S9 = 0
S10 = 1
S11 = 0

(S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11) =
(0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0)
```

**Listings 4.3.** Przykładowe wartościowanie zmiennych

## 4.3. Klauzule

### 4.3.1. Proces budowanie klauzul

Kolejnym elementem tworzenie solvera jest określenie klauzul oraz ich wag, ponieważ korzystamy z problemu Weighted Max-SAT. Klauzule będą składać się ze zmiennych zdaniowych połączonych operatorami alternatywy oraz koniunkcji i będą miały na celu odzwierciedlić preferencje użytkownika co do wyboru parkingu, a także zmusić solver do wyboru tylko jednej, najkorzystniejszej dla klienta strefy. Im więcej klauzul, o większej wadze zostanie spełnionych, tym lepiej dopasowane miejsce będzie mogło zostać zarekomendowane przez aplikację.

W przypadku klauzul nie będziemy więc opierać się na parametrach parkingu lecz utworzone zostaną one na podstawie informacji przesłanych przez klienta. Priorytetem dla solvera pozostaje jednak wybór tej strefy w pobliżu użytkownika, która posiada najwyższy priorytet,

czyli jest tam największe zapotrzebowanie na samochody. Wyodrębnionych zostało 12 klauzul wraz z ich wagami. Tak prezentuje się zestawienie klauzul:

- U0 - klauzula pomocnicza wymuszająca wybrania przez solver najkorzystniejszej strefy, w której ma znajdować się rekomendowany parking.
- U1 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 1
- U2 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 2
- U3 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 3
- U4 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 4
- U5 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 5
- U6 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 6
- U7 - klauzula mówiąca solverowi, aby nie wybierał strefy nr 7
- U8 - Klient preferuje parkingi z miejscami dla osób niepełnosprawnych.
- U9 - Klient preferuje parkingi płatne
- U10 - Klient preferuje parkingi strzeżone
- U11 - Klient preferuje przestronne parkingi

Pierwsze 8 klauzul pozostaje niezależnych od wyboru użytkownika i wybór strefy rekomendowanej pozostaje w kwestii samej aplikacji, który używa systemu wag. Oblicza się go w następujący sposób :

$$\text{waga} = \text{Math.round}(1 / (50 * \text{occupiedRatio} + 10 * \text{attractivenessRatio}) * 1000) ;$$

Oznacza to, że najniższą wagę (czyli najwyższy priorytet) uzyska strefa o odpowiednio wysokim współczynniku zapotrzebowania na samochody, który jest najważniejszym wskaźnikiem, ale w pewnym stopniu zależeć on będzie również od atrakcyjności strefy dla klienta.

Pozostałe klauzule dotyczą już bezpośrednio użytkownika i jego wyborów podanych przy tworzeniu zapytania do systemu.

Tak przedstawia się lista klauzul, ich wag oraz sposobu budowania ich ze zmiennych zdaniowych. W nawias podano ich wagi:

- $U0 \rightarrow S1 \vee S2 \vee S3 \vee S4 \vee S5 \vee S6 \vee S7$  (9999)
- $U1 \rightarrow \neg S1$  (priorytet strefy nr 1)
- $U2 \rightarrow \neg S2$  (priorytet strefy nr 2)
- $U3 \rightarrow \neg S3$  (priorytet strefy nr 3)
- $U4 \rightarrow \neg S4$  (priorytet strefy nr 4)
- $U5 \rightarrow \neg S5$  (priorytet strefy nr 5)
- $U6 \rightarrow \neg S6$  (priorytet strefy nr 6)
- $U7 \rightarrow \neg S7$  (priorytet strefy nr 7)
- $U8 \rightarrow S8 \wedge S11$  (30, 25)
- $\neg U8 \rightarrow \neg S8 \wedge (\neg S11 \vee S9)$  (25, 20)
- $U9 \rightarrow S9 \wedge S11$  (35, 30)
- $\neg U9 \rightarrow \neg S9 \wedge (\neg S11 \vee \neg S10)$  (20, 15)
- $U10 \rightarrow S10 \wedge S9$  (30, 20)
- $\neg U10 \rightarrow \neg S10 \wedge \neg S11$  (20, 15)
- $U11 \rightarrow S11 \wedge (S9 \vee S10)$  (15, 10)
- $\neg U11 \rightarrow \neg S11$  (25)

Pierwsza liczba określa wagę klauzli przed znakiem koniunkcji, natomiast druga, za znakiem. Klauzula  $U0$  wraz z klauzulami  $U1 - U7$  tworzą formułę powodującą wybranie przez solver tylko jednej strefy. Działa to w taki sposób, solver musi spełnić klauzulę  $U0$ , ponieważ ma bardzo wysoką wagę, a następnie spełnia jak najwięcej klauzul  $U1 - U7$  do momentu gdy  $U0$  przestałaby być spełniona.

W efekcie końcowym tylko jedna klauzula  $U1 - U7$  ma wartość *true*.

Uzasadnienie budowy klauzul :

$U0 \rightarrow S1 \vee S2 \vee S3 \vee S4 \vee S5 \vee S6 \vee S7$  - klauzula pomocnicza do wyboru najkorzystniejszej strefy.

Dla lepszej wizualizacji zasady działania tej klauzuli rozważmy prostszy przykład, w którym mamy tylko 3 zmienne zdaniowe dotyczące stref. Tak więc:

$$U0 \rightarrow S1 \vee S2 \vee S3$$

$$U1 \rightarrow \neg S1$$

$$U2 \rightarrow \neg S2$$

$$U3 \rightarrow \neg S3$$

Formuła zdaniowa będzie zatem wyglądała następująco:

$$(S1 \vee S2 \vee S3) \wedge \neg S1 \wedge \neg S2 \wedge \neg S3$$

#### **Listings 4.4.** Przykład uzasadniający utworzenie klauzuli U0

Powyższa formuła jest oczywiście niespełnialna lecz ważony Max-SAT solver działa w taki sposób, że próbuje spełnić jak najwięcej klauzul o jak najwyższych wagach. Na pewno więc musi spełnić klauzulę U0, jednak dzięki zastosowaniu alternatywy zmiennych zdaniowych wystarczy, że tylko jedna z nich będzie miała wartość 1. Oznacza to, że solver może spełnić jeszcze którekolwiek 2 z klauzul U1,U2,U3 nadając odpowiadającym im zmiennym zdaniowych wartość 0.

Otrzymane wartościowanie zmiennych mogłoby więc wyglądać tak;

$$S1 = 0, \quad S2 = 0, \quad S3 = 1$$

Widać z tego, że została wybrana dokładnie jedna strefa i jest to dokładnie taki stan, którego aplikacja oczekuje.

$U8 \rightarrow S8 \wedge S11$  - Użytkownik pragnie zaparkować na parkingu, która posiada miejsca dla niepełnosprawnych, zatem parking musi posiadać takie miejsca, a oprócz tego aby ułatwić klientowi, skierujemy go na mniej oblegany parking.

$\neg U8 \rightarrow \neg S8 \wedge (\neg S11 \vee S9)$  - Użytkownik nie chce by parking posiadał miejsca dla niepełnosprawnych, zatem parking nie będzie posiadał takich miejsc oraz ma on być płatny bądź posiadać mniej wolnych miejsc, aby mniej zatłoczone, bezpłatne parkingi udostępnić osobom niepełnosprawnym.

$U9 \rightarrow S9 \wedge S11$  - Użytkownik szuka płatnego parkingu, zatem parking będzie płatny, zaś takie parkingi przeważnie mają sporo wolnych miejsc.

$\neg U9 \rightarrow \neg S9 \wedge (\neg S11 \vee \neg S10)$  - Użytkownik wybrał preferencję dotyczącą bezpłatnego parkingu, zatem parking będzie bezpłatny, jednakże takie parkingi przeważnie nie są strzeżone, a także więcej osób preferuje takie parkingi. Klauzula odzwierciedla więc taką sytuację.

$U10 \rightarrow S10 \wedge S9$  - Użytkownik wybrał opcję strzeżonego parkingu, zatem parking będzie strzeżony, a co za tym idzie często takie parkingi są płatne.

$\neg U10 \rightarrow \neg S10 \wedge \neg S11$  - Użytkownik preferuje niestrzeżone parkingi, zatem parking nie będzie strzeżony oraz nie będzie posiadał przynajmniej 15 wolnych miejsc, ponieważ niestrzeżony parking jest często bezpłatny, więc będzie bardziej oblegany.

$U11 \rightarrow S11 \wedge (S9 \vee S10)$  - Użytkownik preferuje przestronne parkingi więc musi on koniecznie mieć sporo wolnych miejsc do parkowania, a takie parkingi są przeważnie płatne bądź strzeżone.

$\neg U11 \rightarrow \neg S11$  - Użytkownik nie chce by parking miał 15 wolnych miejsc, więc parking nie będzie posiadał tylu wolnych miejsc.

Z takich klauzul tworzona zostaje ostateczna formuła zdaniowa, dla której uruchamiamy solver, wyliczając najlepsze wartościowanie zmiennych zdaniowych.

### 4.3.2. Przykład generowania formuły

Rozważmy następującego użytkownika, który pragnie zaparkować na parkingu, który niekoniecznie musi posiadać miejsca dla niepełnosprawnych, za to powinien być płatny, strzeżony oraz być przestronny:

$U8 = 0$   
 $U9 = 1$   
 $U10 = 1$   
 $U11 = 1$   
  
 $(U8, U9, U10, U11) = (0, 1, 1, 1)$

**Listings 4.5.** Przykładowe wartościowanie klauzul

Przepiszmy teraz powyższe klauzule na zmienne zdaniowe

$\neg U8 \rightarrow \neg S8 \wedge (\neg S11 \vee S9)$   
 $U9 \rightarrow S9 \wedge S11$   
 $U10 \rightarrow S10 \wedge S9$   
 $U11 \rightarrow S11 \wedge (S9 \vee S10)$

**Listings 4.6.** Przykładowe wartościowanie klauzul



Dołączmy jeszcze klauzule związane ze strefami aby utworzyć całą formułę:

$$(S1 \vee S2 \vee S3 \vee S4 \vee S5 \vee S6 \vee S7) \wedge \neg S1 \wedge \neg S2 \wedge \neg S3 \wedge \neg S4 \wedge \neg S5 \wedge \\ \neg S6 \wedge \neg S7 \wedge \neg S8 \wedge (\neg S11 \vee S9) \wedge S9 \wedge S11 \wedge S10 \wedge S9 \wedge S11 \wedge (S9 \vee S10)$$

**Listings 4.7.** Wygenerowana formuła zdaniowa

## 4.4. Interpretacja odpowiedzi solvera

Odpowiedzą solvera będzie lista zmiennych zdaniowych zanegowanych bądź nie, dla każdej zmiennej, oznaczającym czy taka zmienna zdaniowa powinna mieć wartość *true*, czy też *false*. Taki zestaw danych określał będzie parametry idealnego parkingu.

Przykładowo, wynik zwrócony przez solver może wyglądać tak :

$$[-1, 2, -3, -4, -5, -6, -7, 8, -9, 10, 11]$$

**Listings 4.8.** Przykładowy wynik solvera

Takie wartości zmiennych zdaniowych określają parking znajdujący się w strefie nr 2 w otoczeniu klienta, a także parking dostosowany do osób niepełnosprawnych, niepłatny, strzeżony oraz mający więcej niż 15 wolnych miejsc.

## 4.5. Wyszukiwanie najlepszych parkingów

Teraz gdy solver wskazał nam idealne miejsce parkingowe, pozostaje jedynie przeiterować po wszystkich parkingach znajdujących się w branych pod uwagę przez solver strefach i porównać ich parametry z wynikiem. Do tego celu wykorzystam prosty proces punktacji parkingów. Będzie on przydzielał score w następujący sposób :

- +10 punktów jeśli parking znajduje się we wskazanej przez silnik strefie
- +1 punkt za każdą zgodność parametru parkingu z ideałem

Gdy zakończy się proces punktacji, użytkownik będzie wiedział, który parking jest najbardziej dla niego rekomendowany.



## 5. Biblioteka SAT4J w projekcie

SAT4J to biblioteka open source stworzona dla języka Java w celu rozwiązywania problemów logicznych i optymalizacyjnych. Idealnie nadaje się do obliczania zadań związanych z SAT oraz jego wariacjami, takimi jak Max-SAT oraz Weighted Max-SAT. Użyte w niej wzorce projektowe takie jak strategia oraz dekorator sprawiają, iż można dopasować solver do swoich potrzeb. Mimo wszystko jest ona jednak stosunkowo wolna w działaniu co jest jej znacznym minusem.

### 5.1. Format wejścia

SAT4J narzuca odgórnie użytkownikowi przymus stosowania formatu DIMACS przy podawaniu klauzul do solvera. Standard ten to nic innego jak tekstowa reprezentacja formuły logicznej w postaci CNF określająca reguły przekazywania danych.

#### 5.1.1. Zapisanie problemu SAT jako DIMACS [5]

Dane umieszczane zostają w pliku tekstowym zbudowanym w ściśle określony sposób. Składa się on z trzech podstawowych sekcji:

1. Komentarze - nie są konieczne do prawidłowego odczytania przez program danych lecz pomagają opisać plik wejściowy. Znakiem komentującym jest litera 'c'.
2. Wiersz wprowadzający - określa miejsce, od którego zaczynają się linie z klauzulami. Opisuje on również rodzaj problemu SAT, liczbę zmiennych zdaniowych (*nvars*) oraz liczbę klauzul (*nclauses*).  
Ma postać : *p cnf nvars nclauses*
3. Wiersze z danymi - reprezentują każdą z kolei klauzulę formuły logicznej. Ich format to *nvars* liczb z zakresu  $[-nvars; nvars]$ , przy czym dodatnie liczby odpowiadają danej zmiennej zdaniowej, a ujemna jej zaprzeczeniu. Każda linia kończy się cyfrą 0. Spacje uznawane są jak operator logiczny lub ( $\vee$ )

### 5.1.2. Przykładowa formuła w formacie DIMACS

Wykorzystajmy SAT problem o takiej budowie w koniunkcyjnej postaci normalnej:

$$(p \vee q \vee \neg r) \wedge (\neg p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r) \wedge (p \vee \neg q \vee r)$$

po przekształceniu do tekstowej reprezentacji wyglądałaby następująco:

```
c
c CNF w formacie DIMACS
c
p cnf 3 4
1 2 -3 0
-1 2 3 0
-1 -2 -3 0
1 -2 3 0
```

**Listings 5.1.** Formuła logiczna CNF w formacie DIMACS

### 5.1.3. Zapisanie problemu Weighted Max-SAT jako DIMACS

Ogólny zarys formatu pozostaje niezmienny, jednak należy dokonać kilku modyfikacji.

1. Wiersz wprowadzający - tym razem ma postać :  $p \ wcnf \ nvars \ nclauses$ . Zmianie ulega drugi wyraz, wskazujący na fakt, iż mamy do czynienia z klauzulami posiadającymi wagi.
2. Wiersze z danymi - ich format modyfikuje się dodając liczbę reprezentującą wagę klauzuli na początek, po której występuje niezmienna forma zapisu danych.

### 5.1.4. Przykładowa formuła Weighted Max-SAT w formacie DIMACS

Dodajmy wagi do klauzul z wyżej użytej formuły logicznej:

$$\begin{aligned}
 &(p \vee q \vee \neg r) \wedge (\neg p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r) \wedge (p \vee \neg q \vee r) \\
 &\qquad\qquad\qquad (p \vee q \vee \neg r) \rightarrow [4] \\
 &\qquad\qquad\qquad (\neg p \vee q \vee r) \rightarrow [7] \\
 &\qquad\qquad\qquad (\neg p \vee \neg q \vee \neg r) \rightarrow [11] \\
 &\qquad\qquad\qquad (p \vee \neg q \vee r) \rightarrow [9]
 \end{aligned}$$

po przekształceniu do tekstowej reprezentacji wyglądałaby następująco:

```

c
c WCNF w formacie DIMACS
c
p wcnf 3 4
4 1 2 -3 0
7 -1 2 3 0
11 -1 -2 -3 0
9 1 -2 3 0

```

**Listings 5.2.** Formuła logiczna CNF z wagami w formacie DIMACS

## 5.2. Użycie biblioteki SAT4J w kodzie aplikacji

Biblioteka udostępnia użytkownikowi szereg klas oraz metod ułatwiających zakodowanie problemów logicznych w sposób zrozumiały dla solvera. Implementacja jest bardzo łatwa i szybka, składa się bowiem z kilku, bądź kilkunastu linii kodu.

### 5.2.1. Tworzenie Solvera oraz reprezentacja formuł logicznych w kodzie

W przypadku tworzenia aplikacji pobierającej dane od użytkownika, a także z bazy danych, lepszym rozwiązaniem od tworzenia plików z danymi jest budowanie klauzul bezpośrednio w klasie Solvera. Tym sposobem pozbywamy się konieczności mapowania formuł do formatu DIMACS oraz nie zużywamy miejsca na dysku.

### 5.2.2. Implementacja SAT Solvera [6]

Jako przykład posłużymy się problemem z punktu 5.1.2. Na początku utworzymy obiekt solvera. W dalszej kolejności prześlemy strukturę naszej formuły zdaniowej i wywołamy metodę sprawdzającą spełnialność tego problemu SAT.

```
// Stworzenie obiektu Solvera używając dekoratora
WeightedMaxSatDecorator maxSatSolver = new WeightedMaxSatDecorator(
    org.sat4j.maxsat.SolverFactory.newDefault());
ModelIterator solver = new ModelIterator(
    new OptToSatAdapter( new PseudoOptDecorator(maxSatSolver)));

// Podanie ilości klauzul i zmiennych zdaniowych
solver.newVar(3);
solver.setExpectedNumberOfClauses(4);

// Dodanie klauzul do modelu
maxSatSolver.addSoftClause(new VecInt(new int[] {1, 2, -3}));
maxSatSolver.addSoftClause(new VecInt(new int[] {-1, 2, 3}));
maxSatSolver.addSoftClause(new VecInt(new int[] {-1, -2, -3}));
maxSatSolver.addSoftClause(new VecInt(new int[] {1, -2, 3}));

//Sprawdzanie spełnialności formuły logicznej
try {
    if (solver.isSatisfiable()){
        System.out.println(Arrays.toString(solver.model()));
    } else {
        throw new Exception("Niespełnialna formuła");
    }
} catch (Exception e){
    e.printStackTrace(); }
```

**Listings 5.3.** Przykładowa implementacja problemu SAT

W przypadku, gdy formuła jest spełnialna wynikiem działania solvera będzie lista wartościowania zmiennych zdaniowych, dla której problem SAT ma wartość *true*. Przykładowo:  $[-1, 2, -3]$

### 5.2.3. Implementacja Ważonego Max-SAT Solvera

Aby stworzyć solver działający w oparciu o Weighted Max-SAT należy dokonać kilku modyfikacji względem wyżej przedstawionego kodu. Mianowicie korzystamy z udostępnionego przez bibliotekę SAT4J dekoratora, a następnie porównywalnie z wcześniejszym przykładem przekazujemy informacje o ilości zmiennych zdaniowych oraz klauzul. Za przykład posłuży problem z punktu 5.1.4.

```
// Stworzenie obiektu Solvera używając dekoratora
WeightedMaxSatDecorator maxSatSolver = new WeightedMaxSatDecorator(
    org.sat4j.maxsat.SolverFactory.newDefault());
ModelIterator solver = new ModelIterator(
    new OptToSatAdapter( new PseudoOptDecorator(maxSatSolver)));

// Podanie ilości klauzul i zmiennych zdaniowych
solver.newVar(3);
solver.setExpectedNumberOfClauses(4);

// Dodanie klauzul do modelu
maxSatSolver.addSoftClause(4, new VecInt(new int[] {1, 2, -3}));
maxSatSolver.addSoftClause(7, new VecInt(new int[] {-1, 2, 3}));
maxSatSolver.addSoftClause(11, new VecInt(new int[] {-1, -2, -3}));
maxSatSolver.addSoftClause(9, new VecInt(new int[] {1, -2, 3}));

//Sprawdzanie spełnialności formuły logicznej
try {
    if (solver.isSatisfiable()){
        System.out.println(Arrays.toString(solver.model()));
    } else {
        throw new Exception("Niespełnialna formuła");
    }
} catch (Exception e) {
    e.printStackTrace(); }
```

**Listings 5.4.** Przykładowa implementacja problemu Weighted Max-SAT

Wynikiem również będzie lista wartościowania lecz taka, dla której suma wag spełnionych klauzul jest maksymalna.





## 6. Implementacja systemu

W tym rozdziale przedstawione zostaną niektóre fragmenty kodu aplikacji, reprezentujące przede wszystkim elementy budowy solvera, kontrolera przyjmującego zapytania od użytkownika czy też metody generującej strefy oraz parkingi.

### 6.1. Generowanie danych o położeniu parkingów i stref na obszarze miejskim

```
@GetMapping("/GenerateData")
public HttpStatus GenerateZones() {
    List<String> cities = new ArrayList<>(
        List.of("Kraków", "Warszawa", "Wrocław"));
    if (zoneService.getAllZones().isEmpty()) {
        for (String city : cities) {
            for (int x = -4; x <= 5; x++)
                for (int y = -3; y <= 4; y++) {
                    zoneService.addZone(zone); }
        }
        List<ZoneEntity> zones = zoneService.getAllZones();
        for (ZoneEntity zone: zones) {
            for (int i = 0; i < count; i++) {
                // Wygenerowanie losowych parametrów dla parkingu
                parkingService.addParkingLot(parking); }
        }
    }
    return HttpStatus.OK;
}
```

**Listings 6.1.** CarSharingController.java - metoda generująca strefy oraz parkingi

## 6.2. Wybór miasta przez klienta

```
private String pickedCity = "";

@GetMapping("/AssignCity")
public HttpStatus AssignCity(
    @RequestParam(defaultValue = "Kraków") String city) {
    this.pickedCity = city;
    return HttpStatus.OK;
}
```

**Listings 6.2.** CarSharingController.java - metoda służąca do wyboru miasta przez użytkownika

## 6.3. Wyszukiwanie miejsca parkingowego

### 6.3.1. Klasa przechowująca dane parkingu i jego score

```
@GetMapping("/sfps")
public ResponseEntity<String> searchForParkingSpot(...) {

    @Getter
    ZoneTuple {
        public final long ParkingId;
        public final int Score;

        @Override
        public String toString() {
            return String.format("ParkingId: %d Score: %d \n",
                ParkingId, Score); }
    }
}
```

**Listings 6.3.** CarSharingController.java - Klasa pomocnicza reprezentująca odpowiedź systemu

## 6.4. Przyjmowanie preferencji klienta i tworzenie solvera

```
@GetMapping("/sfps")
public ResponseEntity<String> searchForParkingSpot(
    @RequestParam(value = "cordX", defaultValue = "0") int x,
    @RequestParam(value = "cordY", defaultValue = "0") int y,
    @RequestParam() String [] usersChoices) {

    List<ZoneEntity> zones = new ArrayList<>();
    zoneService.getAllZones().forEach(zone -> {
        if (zoneService.isAdjacent(pickedCity, zone, x, y)) {
            zones.add(zone);
        }
    });

    //Implementacja ZoneTuple

    List<ZoneTuple> results = new ArrayList<>();
    Solver solver = new Solver(zones, usersChoices);
    parkingService.getAllParkingLots().forEach(parking ->
        results.add(new ZoneTuple(parking.getParkingLotId(),
            solver.test(parking))));

    results.sort(Comparator.comparingInt(
        ZoneTuple::getScore).reversed());
    List<ZoneTuple> topResults = results.subList(0, 3);

    return new ResponseEntity<>(topResults.toString(),
        HttpStatus.OK);
}
```

**Listings 6.4.** CarSharingController.java - Metoda służąca do wyszukiwania parkingów

## 6.5. Wykorzystanie solvera

### 6.5.1. Max-SAT Dekorator i ustawienie liczby klauzul oraz zmiennych

```
public Solver(List<ZoneEntity> zones, String[] usersChoices) {  
    final int MAX_VAR = 11;  
    final int NB_CLAUSESES = zones.size()+9;  
  
    WeightedMaxSatDecorator maxSatSolver = new  
    WeightedMaxSatDecorator(  
        org.sat4j.maxsat.SolverFactory.newDefault());  
    ModelIterator solver = new ModelIterator(  
        new OptToSatAdapter(  
            new PseudoOptDecorator(maxSatSolver)));  
  
    solver.newVar(MAX_VAR);  
    solver.setExpectedNumberOfClauses(NB_CLAUSESES);  
}
```

**Listings 6.5.** Solver.java - tworzenie części klauzul

### 6.5.2. Max-SAT Dekorator i ustawienie liczby klauzul oraz zmiennych

```
//U8 handicapped  
if (usersChoices[0].equals("Yes")){  
    maxSatSolver.addSoftClause(30,new VecInt(new int[]{8}));  
    maxSatSolver.addSoftClause(25,new VecInt(new int[]{11}));  
}else{  
    maxSatSolver.addSoftClause(25,new VecInt(new int[]{-8}));  
    maxSatSolver.addSoftClause(20,new VecInt(new  
        int[]{9,-11}));  
}
```

**Listings 6.6.** Solver.java - Dodawanie klauzul do solvera

### 6.5.3. Sprawdzanie spełnialności formuły

```
try {
    if (solver.isSatisfiable()) {
        int [] temp = solver.model();
        for (int t : temp) result.add(t);
        System.out.println(result);
    } else {
        throw new Exception("Unsatisfiable formula");
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

**Listings 6.7.** Solver.java - Sprawdzanie spełnialności formuły

### 6.5.4. Proces punktowania parkingów

```
public int test(ParkingLotEntity parking) {
    int score = 0;
    long zone = parking.getZoneId();
    int index = zoneIds.indexOf(zone);

    // is in selected zone
    if (index > (-1) && result.get(index) > 0)
        score += 10;

    // S8 - for disabled
    if (result.contains(8) && parking.getIsForHandicapped())
        score++;
    else if (result.contains(-8) &&
             !parking.getIsForHandicapped())
        score++;
}
```

**Listings 6.8.** Solver.java - Metoda punktująca parkingi

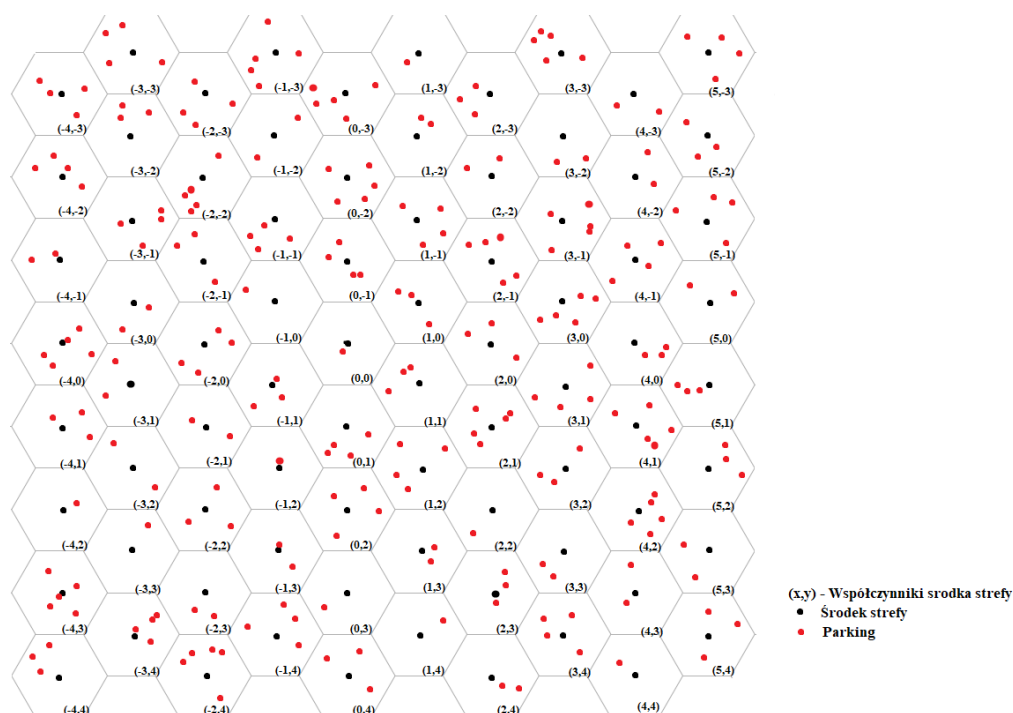


## 7. Weryfikacja działania systemu

Weryfikacja działania aplikacji oraz solvera zostanie przeprowadzona poprzez wygenerowanie przykładowych obszarów miejskich dla dwóch miast, a następnie dla każdego z nich utworzone zostaną po dwa zapytania z różnymi preferencjami użytkownika. Następnie uzyskane wyniki porównane zostaną z parametrami stref oraz parkingów w celu potwierdzenia ich poprawności.

### 7.1. Generowanie danych

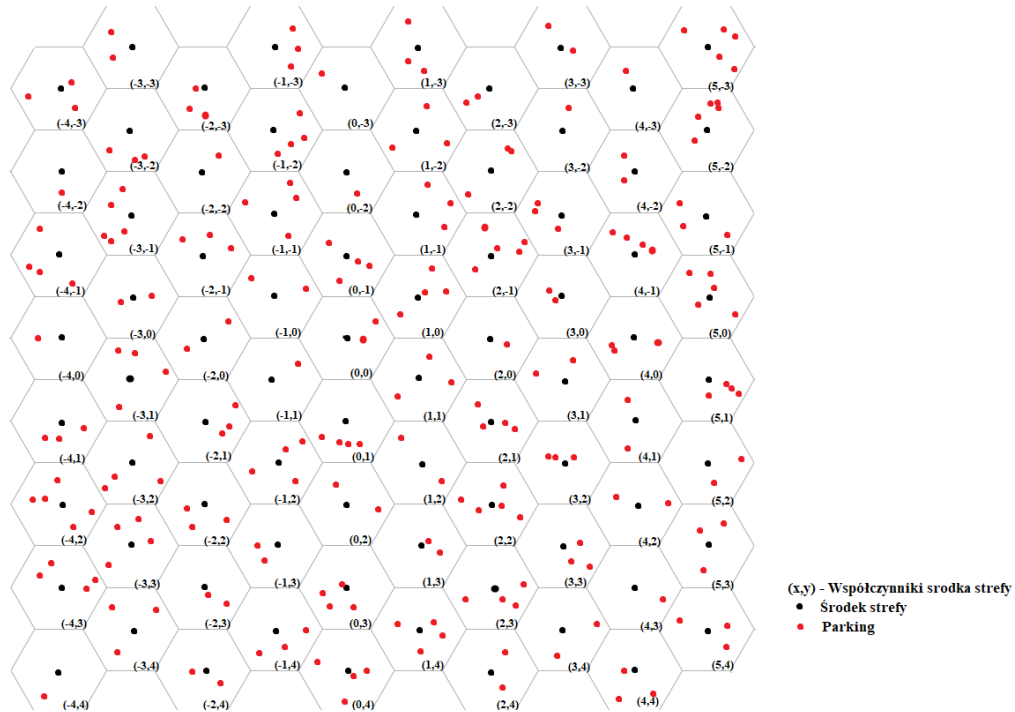
#### 1. Kraków



**Rys 7.1.** Kraków - wygenerowana mapa położenia parkingów

Powyższa mapa przedstawia lokalizację utworzonych przez system stref oraz parkingów. Oznaczenia tłumaczy legenda dołączona do mapy.

## 2. Warszawa



**Rys 7.2.** Warszawa - wygenerowana mapa położeń parkingów

Powyższa mapa przedstawia lokalizacje utworzonych przez system stref oraz parkingów. Oznaczenia tłumaczy legenda dołączona do mapy.



## 7.2. Użytkownik A - Kraków

Rozważmy klienta aplikacji, który szuka miejsca parkingowego w Krakowie oraz o następujących preferencjach.

Insert destination location:

CordX   CordY

Do you want a parking spot for disabled people?

☐ Yes ☒ No

Do you want a paid parking?

☐ Yes ☒ No

Do you want a guarded parking spot?

☒ Yes ☐ No

Do you want a parking with at least 15 spots?

☒ Yes ☐ No

**Rys 7.3.** Użytkownik A - preferencje użytkownika dotyczące parkingu

Taki wybór parametrów oznacza, że klient ma chęć zaparkować na terenie strefy o współrzędnych  $(-2, -2)$ , zaś parking nie posiada miejsc dla niepełnosprawnych oraz jest bezpłatny, jednakże ma on być strzeżony i posiadać co najmniej 15 wolnych miejsc.

Odpowiedź solvera:

```
[-1, -2, -3, -4, 5, -6, -7, -8, -9, 10, -11]
```

```
ParkingId: 61 Score: 12,
```

```
ParkingId: 60 Score: 11,
```

```
ParkingId: 59 Score: 10
```

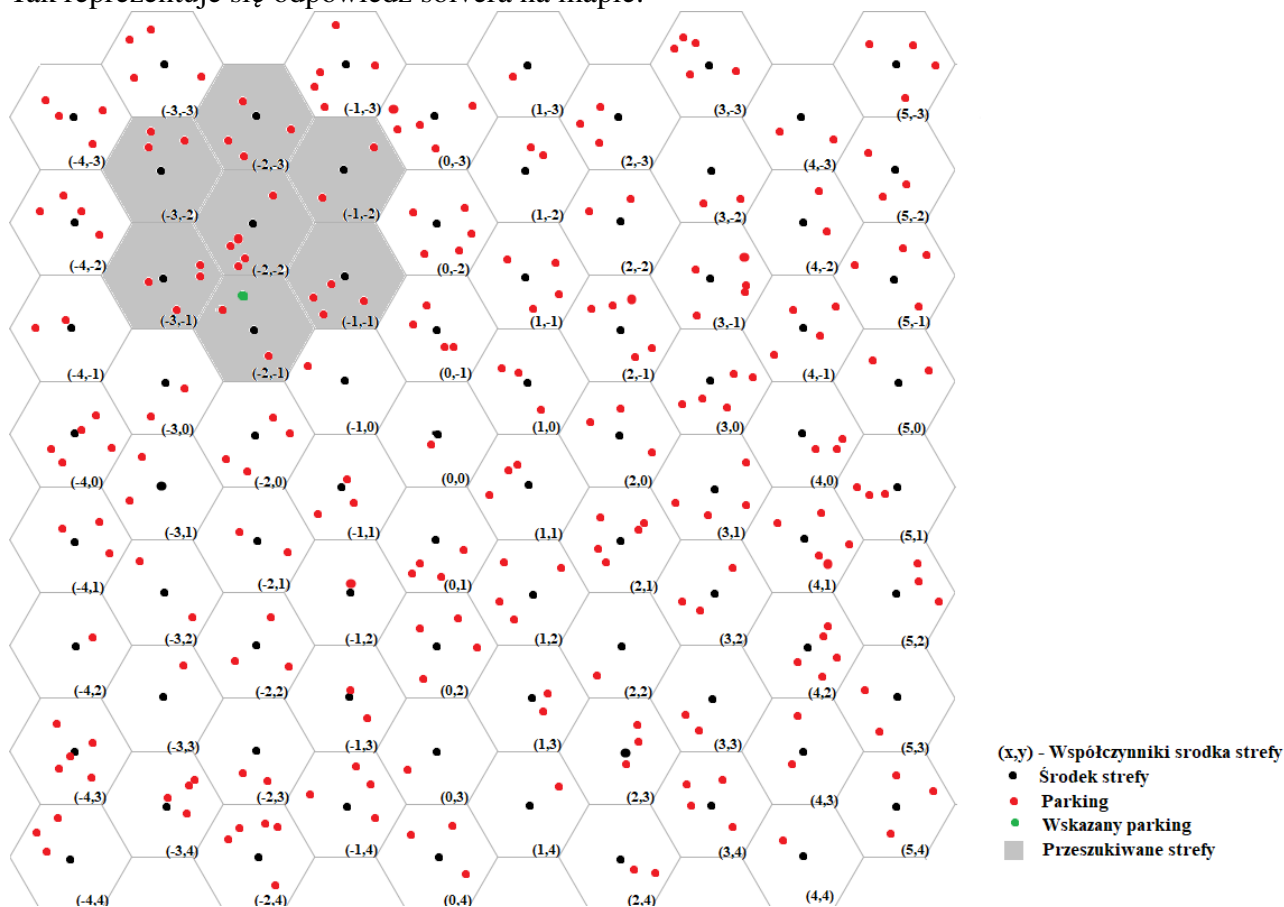
**Listings 7.1.** Użytkownik A - odpowiedź solvera

Otrzymane wartościowanie zmiennych zdaniowych oznacza, iż idealne parametry parkingu wskazane przez solvera są następujące:

```
Numer strefy w otoczeniu klienta: 5
Parking posiada miejsca dla niepełnosprawnych: Nie
Parking jest płatny: Nie
Parking jest strzeżony: Tak
Parking ma co najmniej 15 wolnych miejsc: Nie
```

### Listings 7.2. Użytkownik A - interpretacja wyniku

Tak reprezentuje się odpowiedź solvera na mapie:



Rys 7.4. Użytkownik A - graficzna reprezentacja wyniku

Porównajmy teraz odpowiedź aplikacji ze stanem faktycznym zawartym w bazie danych.

W szczególności dla wybranych do obliczeń stref:

	zone_id [PK] bigint	city character varying (255)	cordx integer	cordy integer	occupied_ratio double precision	attractiveness_ratio double precision
1	10	Kraków	-3	-2	0.26	0.83
2	11	Kraków	-3	-1	0.3	0.06
3	17	Kraków	-2	-3	0.26	0.32
4	18	Kraków	-2	-2	0.77	0.38
5	19	Kraków	-2	-1	0.86	0.11
6	26	Kraków	-1	-2	0.04	0.33
7	27	Kraków	-1	-1	0.51	0.32

**Rys 7.5.** Użytkownik A - rzeczywiste rozłożenie stref

Po wykonaniu obliczeń priorytetów każdej ze stref, czyli iloczynu:

$$priority = 50 * occupied\_ratio * attractiveness\_ratio$$

widać, że solver słusznie dokonał wyboru strefy o ID wynoszącym 19 i współrzędnych  $(-2, -1)$ , ponieważ ma ona najwyższy wynik wskaźnika priorytetu. Oznacza to więc, że w tej strefie potrzeba w tym momencie najwięcej samochodów do wypożyczenia.

W szczególności dla parkingów w wybranej strefie:

	parking_lot_id [PK] bigint	cordx double precision	cordy double precision	free_spaces integer	is_for_handicapped boolean	is_guarded boolean	is_paid boolean	zone_id bigint
1	59	-1.9	-0.68	50	true	false	true	19
2	60	-2.38	-1.2	35	true	true	true	19
3	61	-2.13	-1.36	97	false	false	false	19

**Rys 7.6.** Użytkownik A - rzeczywiste rozłożenie parkingów

Po przeanalizowaniu parametrów poszczególnych parkingów widać, że faktycznie parking o ID wynoszącym 61 posiada największą liczbę parametrów o odpowiedniej wartości w stosunku do idealnego parkingu.

Dla użytkownika A, o wybranych preferencjach, solver poprawnie przeanalizował dane i wskazał najlepszy parking spośród dostępnych, odpowiednio więc spełnił swoje zadanie.

## 7.3. Użytkownik B - Kraków

Rozważmy następnego klienta aplikacji, który również szuka miejsca parkingowego w Krakowie oraz o następujących preferencjach.

Insert destination location:

CordX   CordY

Do you want a parking spot for disabled people?  
☒ Yes ☐ No

Do you want a paid parking?  
☐ Yes ☒ No

Do you want a guarded parking spot?  
☒ Yes ☐ No

Do you want a parking with at least 15 spots?  
☐ Yes ☒ No

**Rys 7.7.** Użytkownik B - preferencje użytkownika dotyczące parkingu

Taki wybór parametrów oznacza, że klient ma chęć zaparkować na terenie strefy o współrzędnych (3, 1), zaś parking posiada miejsca dla niepełnosprawnych oraz jest bezpłatny, a także ma on być strzeżony lecz nie posiadać 15 wolnych miejsc.

Odpowiedź solvera:

```
[-1, 2, -3, -4, -5, -6, -7, 8, -9, 10, -11]
```

```
ParkingId: 160 Score: 13,
```

```
ParkingId: 162 Score: 13,
```

```
ParkingId: 161 Score: 12
```

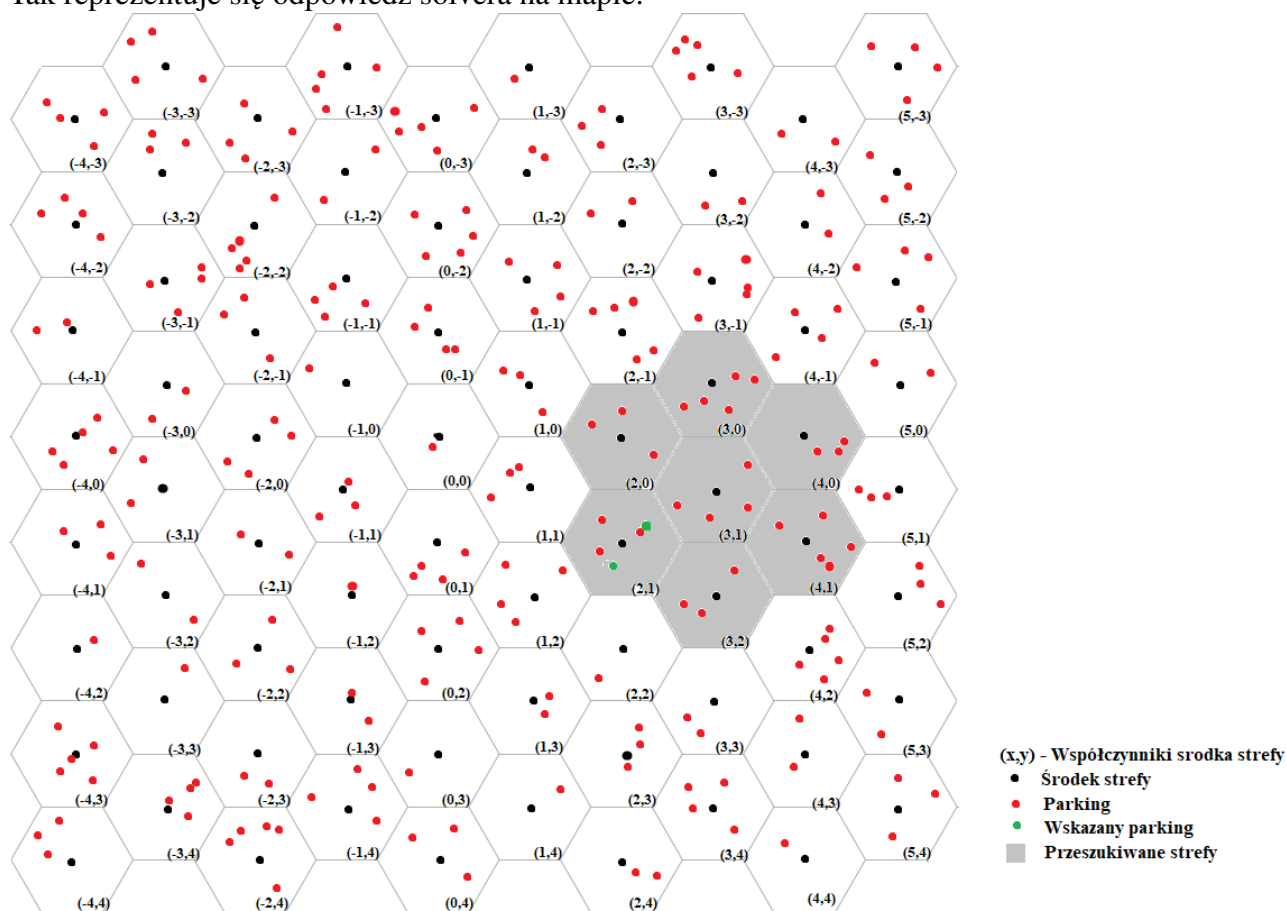
**Listings 7.3.** Użytkownik B - odpowiedź solvera

Otrzymane wartościowanie zmiennych zdaniowych oznacza, iż idealne parametry parkingu wskazane przez solvera są następujące:

Numer strefy w otoczeniu klienta: 2  
 Parking posiada miejsca dla niepełnosprawnych: Tak  
 Parking jest płatny: Nie  
 Parking jest strzeżony: Tak  
 Parking ma co najmniej 15 wolnych miejsc: Nie

**Listings 7.4.** Użytkownik B - interpretacja wyniku

Tak reprezentuje się odpowiedź solvera na mapie:



**Rys 7.8.** Użytkownik B - graficzna reprezentacja wyniku

Porównajmy teraz odpowiedź aplikacji ze stanem faktycznym zawartym w bazie danych.

W szczególności dla wybranych do obliczeń stref:

	zone_id [PK] bigint	city character varying (255)	cordx integer	cordy integer	occupied_ratio double precision	attractiveness_ratio double precision
1	52	Kraków	2	0	0.01	0.02
2	53	Kraków	2	1	0.98	0.34
3	60	Kraków	3	0	0.06	0.49
4	61	Kraków	3	1	0.37	0.03
5	62	Kraków	3	2	0.94	0.38
6	68	Kraków	4	0	0.67	0.96
7	69	Kraków	4	1	0.06	0.2

**Rys 7.9.** Użytkownik B - rzeczywiste rozłożenie stref

Po wykonaniu obliczeń priorytetów każdej ze stref, czyli iloczynu:

$$priority = 50 * occupied\_ratio * attractiveness\_ratio$$

widać, że solver słusznie dokonał wyboru strefy o ID wynoszącym 53 i współrzędnych (2, 1), ponieważ ma ona najwyższy wynik wskaźnika priorytetu. Oznacza to więc, że w tej strefie potrzeba w tym momencie najwięcej samochodów do wypożyczenia.

W szczególności dla parkingów w wybranej strefie:

	parking_lot_id [PK] bigint	cordx double precision	cordy double precision	free_spaces integer	is_for_handicapped boolean	is_guarded boolean	is_paid boolean	zone_id bigint
1	160	1.94	1.17	13	true	true	true	53
2	161	1.84	1.1	96	false	true	false	53
3	162	2.18	0.84	0	true	false	false	53
4	163	2.14	0.83	45	false	true	false	53
5	164	1.79	0.81	85	false	true	false	53

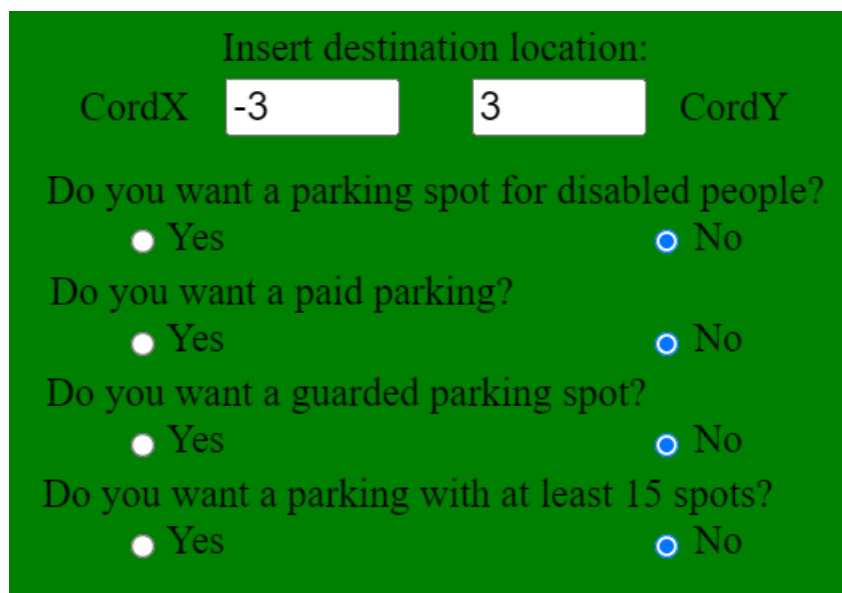
**Rys 7.10.** Użytkownik B - rzeczywiste rozłożenie parkingów

Po przeanalizowaniu parametrów poszczególnych parkingów widać, że aż dwa parkingi o ID wynoszących 160,162 posiada największą liczbę parametrów o odpowiedniej wartości w stosunku do idealnego parkingu.

Dla użytkownika B, o wybranych preferencjach, solver poprawnie przeanalizował dane i wskazał najlepsze parkingi spośród dostępnych, odpowiednio więc spełnił swoje zadanie.

## 7.4. Użytkownik A - Warszawa

Rozważmy następnego klienta aplikacji, który również szuka miejsca parkingowego w Warszawie oraz o następujących preferencjach.



Insert destination location:

CordX   CordY

Do you want a parking spot for disabled people?

☐ Yes ☒ No

Do you want a paid parking?

☐ Yes ☒ No

Do you want a guarded parking spot?

☐ Yes ☒ No

Do you want a parking with at least 15 spots?

☐ Yes ☒ No

**Rys 7.11.** Użytkownik A - preferencje użytkownika dotyczące parkingu

Taki wybór parametrów oznacza, że klient ma chęć zaparkować na terenie strefy o współrzędnych  $(-3, 3)$ , zaś parking nie posiada miejsc dla niepełnosprawnych, jest bezpłatny, niestrzeżony i nie posiada 15 wolnych miejsc.

Odpowiedź solvera:

```
[-1, -2, -3, 4, -5, -6, -7, -8, -9, -10, -11]
```

```
ParkingId: 294 Score: 12,
```

```
ParkingId: 295 Score: 11,
```

```
ParkingId: 292 Score: 10
```

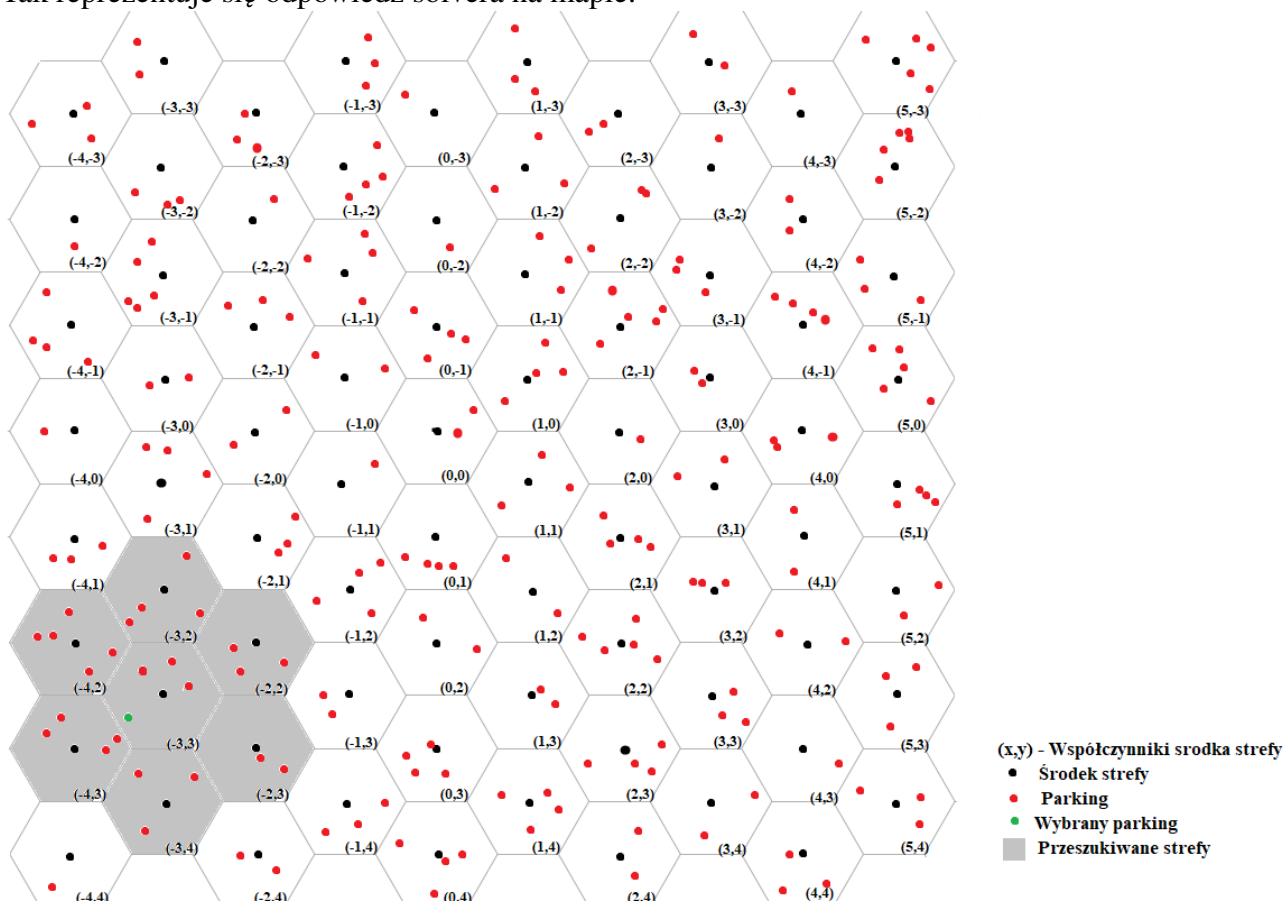
**Listings 7.5.** Użytkownik A - odpowiedź solvera

Otrzymane wartościowanie zmiennych zdaniowych oznacza, iż idealne parametry parkingu wskazane przez solvera są następujące:

Numer strefy w otoczeniu klienta: 4  
 Parking posiada miejsca dla niepełnosprawnych: Nie  
 Parking jest płatny: Nie  
 Parking jest strzeżony: Nie  
 Parking ma co najmniej 15 wolnych miejsc: Nie

**Listings 7.6.** Użytkownik A - interpretacja wyniku

Tak reprezentuje się odpowiedź solvera na mapie:



**Rys 7.12.** Użytkownik A - graficzna reprezentacja wyniku



Porównajmy teraz odpowiedź aplikacji ze stanem faktycznym zawartym w bazie danych. W szczególności dla wybranych do obliczeń stref:

	zone_id [PK] bigint	city character varying (255)	cordx integer	cordy integer	occupied_ratio double precision	attractiveness_ratio double precision
1	86	Warszawa	-4	2	0.23	0.85
2	87	Warszawa	-4	3	0.5	0
3	94	Warszawa	-3	2	0.59	0.07
4	95	Warszawa	-3	3	0.91	0.91
5	96	Warszawa	-3	4	0.44	0.36
6	102	Warszawa	-2	2	0.59	0.27
7	103	Warszawa	-2	3	0.66	0.59

**Rys 7.13.** Użytkownik A - rzeczywiste rozłożenie stref

Po wykonaniu obliczeń priorytetów każdej ze stref, czyli iloczynu:

$$priority = 50 * occupied\_ratio * attractiveness\_ratio$$

widać, że solver słusznie dokonał wyboru strefy o ID wynoszącym 95 i współrzędnych  $(-3, 3)$ , ponieważ ma ona najwyższy wynik wskaźnika priorytetu. Oznacza to więc, że w tej strefie potrzeba w tym momencie najwięcej samochodów do wypożyczenia.

W szczególności dla parkingów w wybranej strefie:

	parking_lot_id [PK] bigint	cordx double precision	cordy double precision	free_spaces integer	is_for_handicapped boolean	is_guarded boolean	is_paid boolean	zone_id bigint
1	292	-2.83	2.97	15	true	true	true	95
2	293	-3.19	2.76	37	true	true	true	95
3	294	-3.22	3.29	21	true	false	false	95
4	295	-2.95	2.72	97	true	true	false	95

**Rys 7.14.** Użytkownik A - rzeczywiste rozłożenie parkingów

Po przeanalizowaniu parametrów poszczególnych parkingów widać, że parking o ID wynoszących 294 posiada największą liczbę parametrów o odpowiedniej wartości w stosunku do idealnego parkingu.

Dla użytkownika A, o wybranych preferencjach, solver poprawnie przeanalizował dane i wskazał najlepszy parking spośród dostępnych, odpowiednio więc spełnił swoje zadanie.

## 7.5. Użytkownik B - Warszawa

Rozważmy następnego klienta aplikacji, który również szuka miejsca parkingowego w Warszawie oraz o następujących preferencjach.

Insert destination location:

CordX   CordY

Do you want a parking spot for disabled people?  
☒ Yes ☐ No

Do you want a paid parking?  
☒ Yes ☐ No

Do you want a guarded parking spot?  
☒ Yes ☐ No

Do you want a parking with at least 15 spots?  
☒ Yes ☐ No

**Rys 7.15.** Użytkownik B - preferencje użytkownika dotyczące parkingu

Taki wybór parametrów oznacza, że klient ma chęć zaparkować na terenie strefy o współrzędnych  $(4, -1)$ , zaś parking posiada miejsca dla niepełnosprawnych, jest płatny, strzeżony i posiada 15 wolnych miejsc.

Odpowiedź solvera:

```
[-1, -2, -3, -4, -5, 6, -7, 8, 9, 10, 11]
```

```
ParkingId: 462 Score: 13,
```

```
ParkingId: 460 Score: 11,
```

```
ParkingId: 461 Score: 11
```

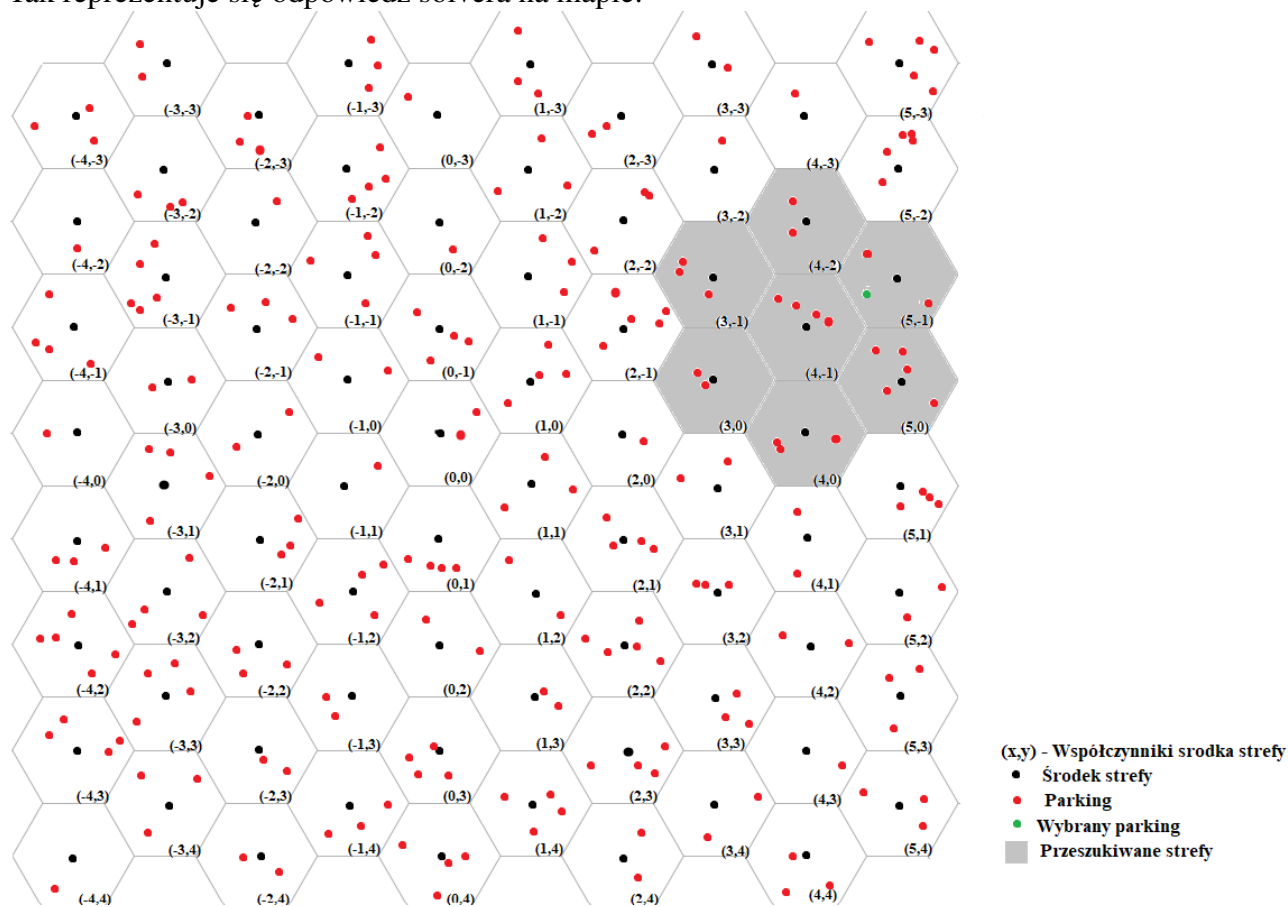
**Listings 7.7.** Użytkownik B - odpowiedź solvera

Otrzymane wartościowanie zmiennych zdaniowych oznacza, iż idealne parametry parkingu wskazane przez solvera są następujące:

Numer strefy w otoczeniu klienta: 6  
 Parking posiada miejsca dla niepełnosprawnych: Tak  
 Parking jest płatny: Tak  
 Parking jest strzeżony: Tak  
 Parking ma co najmniej 15 wolnych miejsc: Tak

**Listings 7.8.** Użytkownik B - interpretacja wyniku

Tak reprezentuje się odpowiedź solvera na mapie:



**Rys 7.16.** Użytkownik B - graficzna reprezentacja wyniku

Porównajmy teraz odpowiedź aplikacji ze stanem faktycznym zawartym w bazie danych.

W szczególności dla wybranych do obliczeń stref:

	zone_id [PK] bigint	city character varying (255)	cordx integer	cordy integer	occupied_ratio double precision	attractiveness_ratio double precision
1	139	Warszawa	3	-1	0.23	0.35
2	140	Warszawa	3	0	0.72	0.05
3	146	Warszawa	4	-2	0.48	0.98
4	147	Warszawa	4	-1	0.69	0.52
5	148	Warszawa	4	0	0.2	0.59
6	155	Warszawa	5	-1	1	0.8
7	156	Warszawa	5	0	0.74	0.96

**Rys 7.17.** Użytkownik B - rzeczywiste rozłożenie stref

Po wykonaniu obliczeń priorytetów każdej ze stref, czyli iloczynu:

$$priority = 50 * occupied\_ratio * attractiveness\_ratio$$

widać, że solver słusznie dokonał wyboru strefy o ID wynoszącym 155 i współrzędnych (5, -1), ponieważ ma ona najwyższy wynik wskaźnika priorytetu. Oznacza to więc, że w tej strefie potrzeba w tym momencie najwięcej samochodów do wypożyczenia.

W szczególności dla parkingów w wybranej strefie:

	parking_lot_id [PK] bigint	cordx double precision	cordy double precision	free_spaces integer	is_for_handicapped boolean	is_guarded boolean	is_paid boolean	zone_id bigint
1	460	5.21	-0.71	11	true	false	false	155
2	461	4.63	-1.13	14	true	false	false	155
3	462	4.69	-0.87	40	false	true	true	155

**Rys 7.18.** Użytkownik B - rzeczywiste rozłożenie parkingów

Po przeanalizowaniu parametrów poszczególnych parkingów widać, że parking o ID wynoszący 462 posiada największą liczbę parametrów o odpowiedniej wartości w stosunku do idealnego parkingu.

Dla użytkownika B, o wybranych preferencjach, solver poprawnie przeanalizował dane i wskazał najlepszy parking spośród dostępnych, odpowiednio więc spełnił swoje zadanie.

## 8. Podsumowanie

Cel pracy jakim był projekt i implementacja systemu do rekomendacji miejsc parkingowych został pomyślnie zrealizowany. Aplikacja oparta o ważony Max-SAT solver poleca użytkownikom parkingi bazując na ich preferencjach, a przy tym uwzględniając takie czynniki jak zapotrzebowanie na samochody w różnych strefach w obszarze miejskim czy atrakcyjność danego terenu. Tak zaprojektowana aplikacja jest w stanie pomóc zarówno klientowi aplikacji w znalezieniu dogodnego miejsca do pozostawienia auta jak i firmie zajmującej się Car-sharingiem poprzez większy i szybszy dostęp do floty samochodów dla użytkowników oraz brak konieczności dystrybuowania większości samochodów, ponieważ będą one już dostępne w miejscach, gdzie inni klienci będą mogli je wynająć.

### 8.1. Wnioski

1. Analizując przykłady działania można stwierdzić, iż aplikacja działa poprawnie i zgodnie z postawionymi celami pracy inżynierskiej oraz ustalonymi założeniami co do funkcjonowania solvera.
2. Wykorzystany w projekcie Max-SAT solver niezwykle dobrze sprawdza się w problemach rekomendacji, które mogą posiadać bardzo skomplikowaną logikę, wiele klauzul logicznych, a przy tym różnorodną strukturę. Korzystając z solvera typu Max-SAT lub Weighted Max-SAT nie musimy przejmować się kwestią spełnialności formuły, ponieważ w przypadku gdy nie można znaleźć takiego wartościowania i tak otrzymamy najlepsze możliwe dopasowanie wartości zmiennych zdaniowych. Jedynym wymogiem jest stosowanie DIMACS, jednakże pomaga to w ujednoliceniu aplikacji tego typu, dzięki czemu stają się czytelniejsze.
3. Z perspektywy klienta utworzono przyjazny i prosty w obsłudze interfejs użytkownika mogący zachęcić do korzystania z aplikacji. Narzędzia i języki programowania, które wykorzystano sprawiają, że system działa sprawnie i w krótkim czasie zwraca użytkownikowi odpowiedź.

## 8.2. Możliwe rozszerzenia systemu

1. Dalszy rozwój mógłby opierać się przede wszystkim na wykorzystywaniu rzeczywistych obszarów miejskich oraz map Google w celu wizualizacji poleceń polecanych użytkownikowi parkingów, bądź połączenia aplikacji z systemem nawigacji samochodowej.
2. Solver mógłby również wykorzystywać jeszcze większą ilość zmiennych zdaniowych i klauzul korzystających na przykład z danych osobowych klienta co do wieku czy płci.
3. Dodanie systemu logowania się do aplikacji oraz tworzenie kont użytkownika w celu przechowywania często odwiedzanych lokacji oraz wykorzystywania historii wykonanych przez klienta tras aby jak najlepiej dopasować solver do każdej osoby indywidualnie.

# Bibliografia

- [1] *Car-sharing*.  
<https://www.traficar.pl/carsharing>.
- [2] *Satisfiability problem*.  
<http://www.cs.ecu.edu/karl/6420/spr16/Notes/NPcomplete/sat.html>.
- [3] *Postacie normalne formuł zdaniowych*.  
<https://sites.google.com/site/problemspelnialnosci/normalform>.
- [4] *k-SAT problems*.  
<https://www.baeldung.com/cs/cook-levin-theorem-3sat>.
- [5] Martin Hořeňovský. *Modern SAT solvers: fast, neat and underused (part 1 of N)*.  
<https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/>. 2018.
- [6] *SAT4J - Howto*.  
<https://www.sat4j.org/howto.php>.