Filip Browarny
Marcin Górski

1. krótkie streszczenie założeń z Projektu Wstępnego

Sieć optyczna 2 - Polska

Zadanie polega na połączeniu każdych dwóch miast ze zbioru danych za pomocą odpowiednich połączeń światłowodowych.

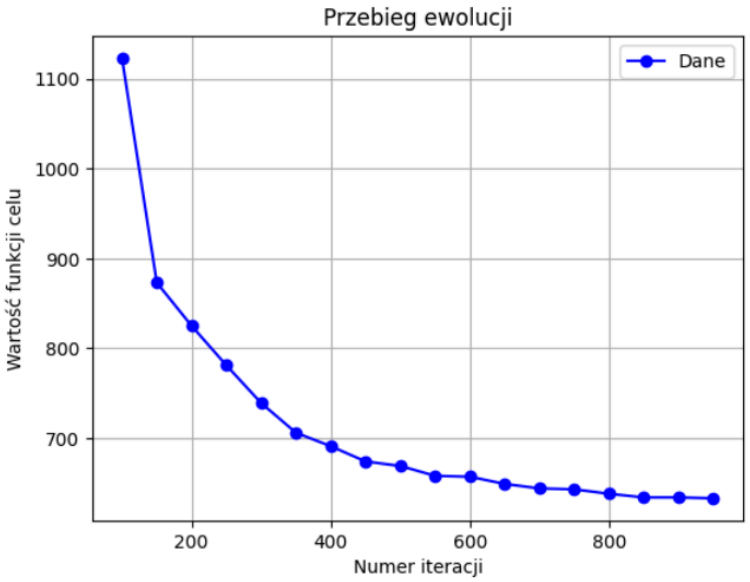
Dostępne karty transponderów to:

- 10 o koszcie 1
- 40 o koszcie 3
- 100 o koszcie 5

Mogą zostać one zmienione. Problem sprowadza się do znalezienia przepływów w grafie nieskierowanym, których koszt jest minimalny. Dodatkowo nałożone jest ograniczenie wynikające z ilości długości fal, które obsługuje światłowód. W praktyce oznacza to, że na danym odcinku maksymalna ilość kart transponderów jest ograniczona do zadanej wartości. W domyślnej wersji jest to 96, natomiast program jest uniwersalny i umożliwia sprawdzenie zachowania algorytmu także dla innych wartości w szczególności jak np 32 lub 64. Do rozwiązania tego problemu użyliśmy algorytmu ewolucyjnego, z typowymi elementami jak inicjalizacja populacji, selekcja (np. turniejowa),, krzyżowanie(np. binarne), mutacja, (np. Gaussowska), a także sukcesja. W projekcie wstępnym opisaliśmy również zbiór danych, a także przemyśleliśmy sposoby krzyżowania i reprezentacji genomu. Bardzo ważne jest także odpowiednie zaplanowanie eksperymentów, które między innymi polegają na sprawdzaniu różnych kombinacji i doboru populacji początkowej, a także jej wielkości.Bardzo ważna jest także rozsądne dobranie liczby iteracji.

2. Pełen opis funkcjonalny

- Program generuje po uruchomieniu plik model.txt zawierający:
    - Otrzymaną wartość funkcji celu.
    - Tabelę zawierającą listę zapotrzebowań, zapewnioną przepustowość, oczekiwaną przepustowość oraz różnice zapewnionej i oczekiwanej przepustowości.
    - Dalej mamy dokładny spis wszystkich transponderów o różnych pojemnościach, na każdej ścieżce, każdego zapotrzebowania.
- Uruchomiony algorytm co 50 iteracji będzie wypisywał na konsolę raport z obecnie znalezionego najlepszego wyniku. Poniżej wykres przykładowego uruchomienia algorytmu dla 1000 iteracji. Dla czytelności pominięte zostały wyniki dla 0 oraz 50 iteracji.



- Model umożliwia zmianę maksymalnej ilości transponderów, oraz ich kosztu.
- Aby uruchomić model z danymi z dowolnej innej sieci np. Niemcy lub USA wystarczy zamienić pliki w folderu networks na takie dotyczące innej sieci.
- Model umożliwia dostosowanie parametrów takich jak:
    - wielkość populacji
    - ilość krzyżowań
    - moc mutacji
    - procent mutowanych genów
    - ilość iteracji
- Model umożliwia wybór algorytmów selekcji, mutacji i krzyżowania.

3. precyzyjny opis algorytmów oraz opis zbiorów danych

- ### Zbiór danych

Wykorzystujemy gotowy zbiór danych ze strony http://sndlib.zib.de/home.action, dla kraju Polska. Na stronie znajduje się także dokładny opis zbioru poszczególnych pól w pliku, dlatego tutaj nie będę tego powtarzał. Jedyną różnicą jaką robimy jest niewykorzystywanie kosztów i pojemności kart transponderów ze strony, tylko ustawienie własnych, stałych i takich samych dla całej sieci, oraz dodanie limitu kart transponderów na danym odcinku.
W celu łatwiejszej obsługi plików poszczególne sekcje takie jak zapotrzebowania czy ścieżki, zostaną przeniesione do osobnych plików. Dzięki temu, łatwo będzie można zmienić sieć, w przypadku np. konieczności dodania nowego miasta.

Za pomocą biblioteki pandas oraz wyrażeń regularnych pliki tekstowe zostały zrzutowane na obiekty pythonowe. Plik z danymi nie są zapisane w żadnym konkretnym formacie dlatego konieczne było wykorzystanie wcześniej wspomnianych wyrażeń regularnych.

- ### Opis algorytmów

Do optymalizacji naszego zadania zastosowaliśmy algorytm ewolucyjny. Jest to rodzaj heurystycznych technik optymalizacyjnych inspirowanych procesami ewolucyjnymi w naturze. Dobrze nadaje się on do rozwiązywania problemów optymalizacyjnych, w których

przestrzeń poszukiwań jest duża i trudno znaleźć globalne optimum. Jego ogólna struktura obejmuje:

-Tworzenie populacji początkowej(potencjalnych rozwiązań problemu)
-Ocenę przystosowania(za pomocą funkcji celu)
-Selekcja
-krzyżowanie
-mutacja
-sukcesja

Wszystkie powyższe kroki prowadzą nas do znajdowania optymalnych rozwiązań problemu.

Na początku musimy zdecydować jak chcemy reprezentować nasze rozwiązanie. W naszym programie zdecydowaliśmy się na rozwiązaniem z użyciem wektora o 1386 wymiarach, gdzie każdy wymiar odpowiada za konkretną wielkość transpondera dla konkretnej ścieżki.

Następnie równie ważna jest inicjalizacja populacji początkowej, w naszym programie genotyp będzie trójwymiarową tablicą numpy (NumPy array). Struktura genotypu będzie miała wymiary (demnand_num, paths_num, trans_num), gdzie:

demnand_num: Liczba wierszy w DataFrame demands_df(ilość wymagań w zbiorze danych)
paths_num: Liczba ścieżek w każdym wymaganiu
trans_num: Liczba rodzajów transponderów

Każdy element genotypu to liczba całkowita losowo wygenerowana z zakresu od min (włącznie) do max (wyłącznie).

Do następnych etapów algorytmu niezbędna będzie dla nas funkcja celu, mająca za zadanie określić jakość danego osobnika. Jest ona sumą kosztu osobnika + wartość funkcji kary. W naszym przypadku funkcja kary jest wykladnicza z dodatkowa stałą wartością. Dzięki temu wyniki niespełniające postawionych celów i ograniczeń są mocno karane i są znacznie gorsze w stosunku do osobników poprawnych, spełniających wymagania. W naszym przypadku wymagania, które powinien spełniać punkt to spełnienie wymaganego przepływu, a także nie przekroczenie maksymalnej ilości transponderów na jednym połączeniu. Sprawdzana jest suma niespełnionych przepływów, a także ilości nadwyżkowych transponderów, w celu wyliczenia kary dla punktu. Obliczona funkcja oceny osobnika będzie nam niezbędna do następnych etapów algorytmu.

Teraz potrzebujemy wybrać punkty z populacji do reprodukcji, odbywa się to na podstawie ich dostosowania, jest to faza selekcji. U nas w algorytmie używamy selekcji turniejowe. Polega na wybraniu kilku osobników do turnieju (u nas domyślnie rozmiar turnieju to dwa) i selekcja wybiera z nich najlepszego osobnika, który przeżywa(u nas domyślnie jeden).

Następnym etapem jest krzyżowanie, w którym dwoje rodziców jest używanych do generowania potomstwa przez wymianę ich genetycznych informacji. U nas używamy krzyżowania jednopunktowego. Losowany jest punkt graniczny, który determinuje jak duża część pierwszego rodzica (dane pierwszego rodzica od “lewej” do tego punktu) i jak duża drugiego (dane drugiego rodzica od punktu do “prawej”) jest brana do stworzenia nowego

osobnika. Te podzielone części są łączone do stworzenia nowego osobnika. W naszym algorytmie bierzemy pod uwagę parametr prawdopodobieństwa krzyżowania, oznaczający to z jakim prawdopodobieństwem dana para rodziców zostanie poddana operacji krzyżowania w procesie ewolucji populacji

Kolejnym etapem jest mutacja, czyli proces wprowadzania losowych zmian lub modyfikacji do genotypu osobników w populacji. Mutacji jest poddawany każdy osobnik, ale bierzemy pod uwagę parametr nazywany prawdopodobieństwem mutacji, określa on szansę na to, że pojedynczy gen w chromosomie (genotypie) osobnika zostanie zmieniony w procesie mutacji. Mamy także parametr mocy mutacji, która wpływa na to jak “szerokie” zmiany mutacyjne będą. W naszym programie używamy mutacji gaussowskiej, która na podstawie prawdopodobieństwa mutacji, modyfikuje poszczególne geny biorac pod uwage liczbe wylosowana z rozkładu normalnego gaussa i parametr mocy mutacji.

Ostatnim etapem algorytmu jest sukcesja. Decyduje ona, które osobniki przechodzą do populacji potomnej.W naszym algorytmie używamy sukcesji elitarnej z 10% elitą, co oznacza, że na początku wybieramy 10% najlepszych osobników z populacji bazowej i oni będą dodani do populacji następnej. Dzięki temu nie tracimy najlepszych osobników z danej iteracji algorytmu.

4. raport z przeprowadzonych testów oraz wnioski

Testowanie polegało na uruchamianiu modelu z różnymi ustawieniami i porównywanie wyników.

Na początku zaimplementowana została selekcja ruletkowa. Algorytm z selekcją ruletkową praktycznie się nie uczył, dlatego zaimplementowaliśmy selekcję turniejową z turniejem o wielkości dwa. To zdecydowanie poprawiło wyniki.

Eksperymentowaliśmy także ze sposobami mutacji. Po testach zdecydowaliśmy się na mutację gaussowską, ze zmiennym procentem mutowanych genów. W trakcie pracy algorytmu co 50 iteracji zmniejszamy go o 1 z 10 do 4. Takie rozwiązanie sprawia, że algorytm na początku skupia się bardziej na eksploracji, a później na eksploatacji.

Algorytm miał problem z uczeniem dopóki nie została ustawiona naprawa poprzez zawijanie. Jeśli po mutacji wartość genu będzie większa niż 2 to zostanie policzone modulo z tej liczby. Wynika to z faktu, że największe zapotrzebowanie między dwoma miastami jest większe niż 200 więc nie ma potrzeby posiadania więcej niż 2 transponderów na żadnej trasie, a wielokrotnie zmniejsza to przestrzeń przeszukiwań.

Kolejnym testem był wybór sukcesji. Konieczne było zastosowanie sukcesji elitarnej z elitą równą 10%.

Testy były utrudnione ze względu na dużą złożoność obliczeniową problemu. Każdy test wymagał kilkunastu minut czekania, żeby wyliczył się model. Przy tak dużej ilości parametrów trzeba było czasem zaufać intuicji, bo niemożliwe jest deterministyczne określenie, który parametr jest najlepszy.

Znalezione przykładowe rozwiązanie, które znajduje się w pliku ‘model.txt’ jest zadowalające, natomiast nie jest optymalne. Niestety złożoność obliczeniowa sprawiła, że 1000 iteracji z 500 osobnikami zajmuje 30 minut. Aby uzyskać optymalny wynik populacja powinna być co najmniej kilka razy większa niż długość genotypu (w naszym przypadku kilka tysięcy). Ilość iteracji także powinna być znacznie większa. Do przyspieszenia kodu wykorzystaliśmy dekorator @njit. Funkcja służąca do mutacji przyspieszyła 42.5 raza po dodaniu tego dekoratora. Nie udało nam się przyspieszyć za pomocą tej biblioteki funkcji

celu, której złożoność obliczeniowa jest największa, ponieważ nie obsługuje ona słowników, na której oparta jest ta funkcja.

5. opis wykorzystanych narzędzi, bibliotek, itp

Projekt pisaliśmy używając języka python, a także jupytera, który umożliwił nam bardzo wygodny podział kodu i większą czytelność.

Użyte biblioteki:

- copy

Biblioteka ta umożliwiła nam wykonywanie głębokich kopii przedstawicieli naszej populacji, dzięki zastosowaniu kopii głębokiej ominęliśmy tworzenie referencji do obiektów.

- random

Pozwoliła nam na generowanie różnorakich losowych liczb. Użyliśmy między innymi funkcji uniform, sample, a także gauss. Była to zdecydowanie niezbędna biblioteka do krzyżowania i mutacji osobników.
Była także konieczna do generowania populacji początkowej lub selekcji.

- re

Jest to moduł umożliwiający korzystanie z wyrażeń regularnych. Pozwoliły nam one na parsowanie i przekształcanie plików danych. Dzięki tej bibliotece odczyt niezbędnych informacji ze zbioru danych był zdecydowanie łatwiejszy.

- pandas

Jest to biblioteka do efektywnego manipulowania danymi. Jej głównym celem jest zwiększenie czytelności i ułatwienie analizy danych.
W naszym projekcie użyliśmy jej do tworzenia Data Frameów, wykorzystanych później w naszej funkcji celu

- numpy

Biblioteka umożliwiająca pracę z dużymi, wielowymiarowymi tablicami i macierzami. Dostarcza także różnorakie funkcje matematyczne do wykonywania działań na tych strukturach.Używamy między innymi w funkcji np.random.randint(...) do tworzenia trójwymiarowej tablicy do generowania losowego genomu.

- numba

Blioteka, która umożliwia przyspieszenie wykonania kodu poprzez kompilację just-in-time (JIT). Działa na zasadzie dekoratorów, które są stosowane do funkcji, co powoduje, że kod tej funkcji jest kompilowany do natywnego kodu maszynowego podczas jego pierwszego wywołania, co zazwyczaj przyspiesza kolejne wywołania tej funkcji. W niektórych miejscach była dla nas bardzo przydatna, lecz nie zawsze można użyć tego dekoratora, ponieważ są sytuacje w których może nawet zwolnić wykonanie programu.