Patryk Milczarek
Michał Janczyk

OAST – projekt, zadanie pierwsze

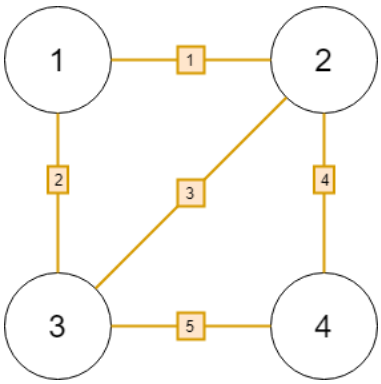
Implementacja algorytmu ewolucyjnego do projektowania sieci optycznych DWDM

1. Opis problemu

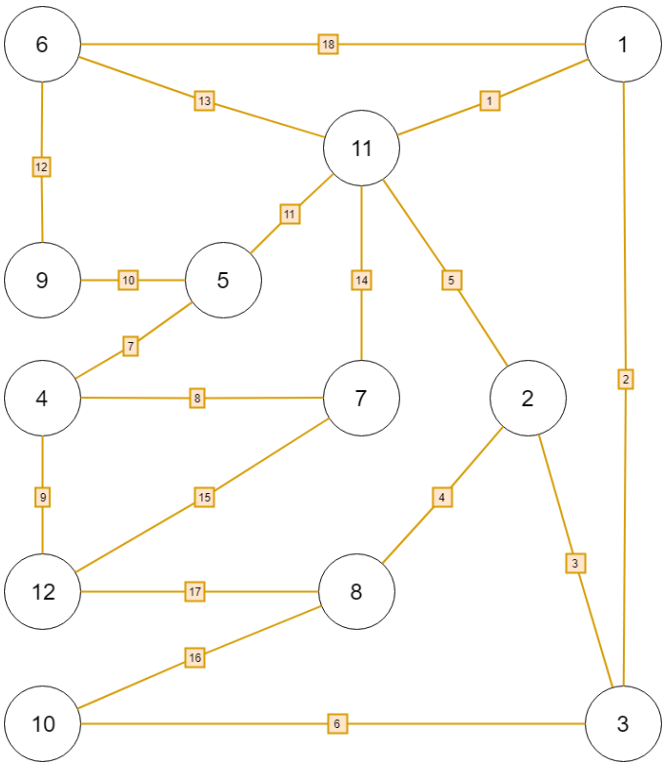
Problem DAP(Demand Allocation Problem) - dla wybranej sieci złożonej z n węzłów połączonych p ścieżkami, każda o określonych zapotrzebowaniach d, należy znaleźć najoptymalniejsze ścieżki P(d,p), dla których obciążenia łączy są optymalnie najmniejsze, nie przekraczające wydajności tychże łącz.

Poniżej przedstawione zostały sieci dla których szukane będą najlepsze rozwiązania problemu DAP.

1.1. Sieć złożona z czterech węzłów



1.2. Sieć złożona z dwunastu węzłów



2. Algorytm Brute Force

2.1. Opis algorytmu

Algorytm brute force polega na sprawdzeniu i znalezieniu pierwszego rozwiązania z listy wszystkich możliwych, które spełnia przyjęte kryterium. Funkcja celu problemu DAP przedstawia się następująco:

$$F(x) = \max \{ \max\{ 0, y(e,x) - c(e) \} : \ e \in E \}$$

gdzie:

e - id łącza,
c(e) - liczba par włókien na łączu e pomnożona przez liczbę lambd we włóknie

Przyjętym kryterium zatrzymania działania algorytmu jest znalezienie takiego rozwiązania, którego wartość funkcji celu spełnia równość F(x) = 0.
Algorytm ten sprawdza się jedynie dla niewielkich sieci, w przypadku większych i bardziej skomplikowanych niż nasza sieć 1.1, wystąpi ogromne zapotrzebowanie na zasoby zarówno czasowe, jak i pamięciowe komputera, przez co jest on wtedy wyjątkowo nieskuteczny.

2.2. Implementacja

Do prawidłowego działania algorytmu brute force należy wygenerować wszystkie możliwe rozwiązania, których liczba w przypadku naszej małej sieci (patrz pkt. **1.1**) wynosi 810 000.

Mając wszystkie możliwe rozwiązania należało znaleźć obciążenie każdego łącza, a następnie obliczyć jego pojemność. Poniżej znajduje się pseudokod zaimplementowanego algorytmu *(w programie funkcja link_computation)*.

```
begin
   for e:=1 to E do l(e,x) := 0;
   for d := 1 to D do
      for p:= 1 to m(d) do
         for e:=1 to E do
            if e ∈ P(d,p) then l(e,x) := l(e,x) + x(d,p);
   for e:=1 to E do y(e,x) := l(e,x)
end
```

Następnie, zgodnie z przyjętym kryterium zatrzymania algorytmu (F(x) = 0) należało zaimplementować podaną funkcję celu. Dla każdego węzła szukamy najmniejszego obciążenia z(e), a następnie wybieramy najgorszy przypadek Fmax. Po znalezieniu najlepszego rozwiązania F(x) = 0 algorytm kończy działanie i zapisujemy wynik do pliku. Poniżej przedstawiono pseudokod zastosowanego algorytmu *(funkcja solve)*:

```
for e:=1 to E do
    z(e) := max{0, y(e,x) - c(e)}

F := max {z} : e ∈ E

if (F == 0)
  create_solution_file
  exit
```

3. Algorytm Ewolucyjny

W przypadku dużej sieci (patrz pkt 1.2) spodziewamy się, że liczba możliwych rozwiązań będzie tak duża, że algorytm brute force nie będzie mógł zostać zastosowany. Czas sprawdzenia wszystkich możliwych rozwiązań zajmie zbyt dużo czasu. Do znalezienia rozwiązania według różnych kryteriów zostanie wykorzystany algorytm ewolucyjny.

3.1. Opis algorytmu

Algorytm ewolucyjny polega na wykonywaniu na zapotrzebowaniach sieci (chromosomach) wygenerowanych wcześniej w losowy sposób operacji genetycznych z określonym prawdopodobieństwem. Do takich operacji należą:

- Mutacja - polega na zamianie kolejności lambd w danym zapotrzebowaniu. Po ich zamianie w sposób losowy, zmutowany chromosom jest dodawany do zbioru chromosomów początkowych.
- Krzyżowanie - Zbiór wszystkich zapotrzebowań jest dzielony na dwa równe zbiory i z każdego z nich, z określonym prawdopodobieństwem, pobierane są po jednym chromosomie(rodzice), aby następnie w sposób losowy krzyżować ich geny, czyli rozwiązania dla jednego zapotrzebowania. W wyniku krzyżowania rodziców powstaje potomek, czyli nowy chromosom, który jest dodawany do zbioru wszystkich chromosomów.

Po wykonaniu wyżej wymienionych operacji, zbiór chromosomów jest sortowany od najmniejszej wartości rozwiązania sieci, obliczanej tak jak w punkcie **2.1**, a następnie najgorsze rozwiązania są z niego usuwane, aby zostało tyle chromosomów ile było na początku działania algorytmu.
Operacje genetyczne są wykonywane wielokrotnie aż do osiągnięcia zadanego kryterium stopu lub najlepszego możliwego wyniku, każdy wynik dla kolejnej iteracji wykonania operacji jest nie gorszy od poprzedniego.

3.2. Implementacja

Implementacja algorytmu polega na wygenerowaniu określonej liczby możliwych rozwiązań danej sieci(populacji).

Mając wygenerowane możliwe rozwiązania, wykonywane są na nich operacje genetyczne z określonymi prawdopodobieństwami aż do osiągnięcia kryterium stopu. Poniżej znajduje się pseudokod zaimplementowanego algorytmu *(w programie funkcja evolution)*.

```
begin
   n:= 0; initialize(P(0));
   while stopping criterion not true
       O(n):= ∅
       for i:= 1 to K do O(n):= O(n)∪crossover[P(n)];
       for x ∈ O(n) do mutate(x);
       n:= n+1;
       P(n):= select_best_N[O(n-1)∪P(n-1)];
   end while
end
```

Do szukania najlepszych rozwiązań została wykorzystana funkcja opisana w punkcie 2.2 *(funkcja solve)*.

4. Uruchomienie programu

Aby uruchomić program na komputerze należy:

a. Zainstalować środowisko Node.js
   https://nodejs.org/en/download/
b. Otworzyć konsolę, przejść do katalogu projektowego i zainstalować moduł ‘seedrandom’:
   `npm install seedrandom`
c. Umieścić plik źródłowy *app.js* i plik wejściowy o nazwie *input.txt*, zawierający rozkład sieci w tym samym folderze.
d. Otworzyć plik app.js, a następnie ustawić parametry algorytmu na początku programu:

method - rodzaj algorytmu, może przyjąć wartości “bruteforce” dla metody brute force lub “evolution” dla algorytmu ewolucyjnego
table_length - liczność populacji
seed - ziarno losowości
p_cross - prawdopodobieństwo krzyżowania
p_mutate - prawdopodobieństwo mutacji
max_cross_number - maksymalna liczba krzyżowań
max_mutation_number - maksymalna liczba mutacji
max_no_change_generations - maksymalna liczba wykonanych iteracji algorytmu po których nie pojawia się lepszy wynik
max_time - maksymalny czas działania algorytmu

e. Będąc w odpowiednim katalogu (tym, gdzie znajduje się plik app.js) użyć poniższej komendy do uruchomienia programu.
   `node app.js`
f. Pliki wynikowe zostaną utworzone w tym samym katalogu.

`evolution-solution.txt`, `bruteforce-solution.txt` - plik końcowy działania programu w zależności od użytej metody, zgodnie z załącznikiem nr 2 dołączonym do zadania pierwszego na stronie przedmiotu

`solution_trajectory.txt` - plik, w którym znajduje się trajektoria najlepszych rozwiązań wg. następującego schematu: <numer iteracji> <funkcja celu F> <EOL>

5. Opis najlepszego uzyskanego rozwiązania

Sieć 4-węzłowa (brute force)

Wartość funkcji kosztu: 0
Liczba wykonanych iteracji AE do znalezienia rozwiązania: 1
Czas optymalizacji: < 0s
Wartości parametrów algorytmu:
- Liczność populacji: 810000

Wynikowe obciążenie łączy (sygnały): |
Wymiary łączy (włókna): | => zawartość pliku bruteforce-net4.txt
Rozkład zapotrzebowań na poszczególne ścieżki: |

Sieć 4-węzłowa (algorytm ewolucyjny)

Wartość funkcji kosztu: 0
Liczba wykonanych iteracji AE do znalezienia rozwiązania: 1
Czas optymalizacji: 0s
Wartości parametrów algorytmu:
- Liczność populacji: 1000

- Prawdopodobieństwo krzyżowania: 0.3
- Prawdopodobieństwo mutacji: 0.04
- **Ziarno prawdopodobieństwa: “oast”**

Wynikowe obciążenie łączy (sygnały): |
Wymiary łączy (włókna): | => zawartość pliku evolution-net4.txt
Rozkład zapotrzebowań na poszczególne ścieżki |

Sieć 12-węzłowa (algorytm ewolucyjny):

Wartość funkcji kosztu: 0
Liczba wykonanych iteracji AE do znalezienia rozwiązania: 226
Czas optymalizacji: 47s
Wartości parametrów algorytmu:
- Liczność populacji: 1000
- Prawdopodobieństwo krzyżowania: 0.3
- Prawdopodobieństwo mutacji: 0.04
- **Ziarno prawdopodobieństwa: “oast”**

Wynikowe obciążenie łączy (sygnały) |
Wymiary łączy (włókna) | = > zawartość pliku oast-evolution-net12.txt
Rozkład zapotrzebowań na poszczególne ścieżki |

Sieć 12-węzłowa (algorytm ewolucyjny):

Wartość funkcji kosztu: 5
Liczba wykonanych iteracji AE do znalezienia rozwiązania: 32
Czas optymalizacji: 6s
Wartości parametrów algorytmu:
- Liczność populacji: 1000
- Prawdopodobieństwo krzyżowania: 0.3
- Prawdopodobieństwo mutacji: 0.04
- **Ziarno prawdopodobieństwa: “tamagotchi”**

Wynikowe obciążenie łączy (sygnały) |
Wymiary łączy (włókna) | = > zawartość pliku tama-evolution-net12.txt
Rozkład zapotrzebowań na poszczególne ścieżki |

6. Wnioski

Algorytm Brute force jest łatwy do zaimplementowania, ale sprawdza się jedynie dla małych sieci. W przypadku gdy istnieje ogromna liczba rozwiązań dla sieci(rzędu paruset milionów, miliardów), jest to metoda nieefektywna. Optymalne jest wtedy zastosowanie innego algorytmu - algorytmu ewolucyjnego, który znajduje rozwiązania za pomocą krzyżowania i mutowania. Na ich podstawie znajduje rozwiązania nie gorsze.