Optymalizacja kolonią mrówek jednowymiarowego problemu upakowania

Joanna Kołodziejczyk

1 Problem do rozwiązania

Celem laboratorium jest implementacja algorytmu mrówkowego dla jednowymiarowego problemu upakowania (bin packing problem).

1.1 Definicja problemu

Biorąc pod uwagę n elementów o różnych rozmiarach a_j i nieskończoną liczbę pojemników (każdy o stałej, skończonej pojemności b), należy upakować elementy w **minimalną** liczbę pojemników, aby żaden pojemnik nie został przepełniony.

Przy rozwiązywaniu zadania trzeba brać pod uwagę następujące ograniczenia:

- Każdy element może być przypisany tylko do jednego pojemnika.
- Całkowita waga/rozmiar umieszczonych w jednym pojemniku elementów musi być mniejsza lub równa zadanej pojemności pojemników. Jeżeli i jest indeksem pojemnika, a k jest znalezioną lub minimalną liczbą pojemników to ograniczenie można wyrazić jako nierówność:

$$\exists poj : \forall_{i \in 0..k-1} : \left(\sum_{\forall j \in poj[i]} a_j \right) \leq b$$

Celem jest znalezienia najmniejszej liczby k pojemników o wielkości b, które mogą zawierać wszystkie obiekty wymienione w a. Alternatywnie można by spróbować znaleźć mapowanie x, dla którego funkcja f przyjmuje najniższą wartość.

$$f(x) = \sum_{i=0}^{k-1} \max \left\{ 0, \left(\sum_{\forall j \in x[i]} a_j \right) - b \right\} \tag{1}$$

2 Algorytm

2.1 Pseudokod algorytmu ACO

Główne założenia ACO to:

1. Każda ścieżka, po której porusza się mrówka, jest kandydatem do rozwiązania danego problemu;

2. Po wygenerowaniu rozwiązania kandydata, ilość feromonu odkładana na ścieżce jest proporcjonalna do jakości rozwiązania;

3. Ścieżka o relatywnie większej ilości feromonu ma większe szanse na bycie wybraną przez mrówkę.

```
Initialization
for idx = 0 to max_iter do
    ant ← RandomlyInitializeAnts
    M ← initialize_ant_memory(M)
    for m = 1 to colony_size do
        while m nie włoży wszystkich elementów do kontenerów. do
            p_m ← compute_prob(τ, M)
            next_element ← max(p_m)
            M ← update(M)
        end while
    end for
    τ ← deposit_pheromon(τ)
    result ← find_best_ant()
    if f(result) < f(best) then
        best ← result
    end if
end for
```

2.2 Istotne elementy algorytmu

1. **Kodowanie mrówki**:

   Mrówka znajdzie kolejność, w której będzie umieszczać przedmioty w pojemnikach.

   - *ant* — lista indeksów elementów wkładanych do kontenerów (permutacja z n). np. $\{2, 3, 1, 5, 4\}$ oznacza, że wkłada się do kontenerów elementy w kolejności 2, 3, 1, 5 i 4.
   - M — Ant Memory przechowuje listę elementów już włożonych do kontenerów (takich, których już nie można wybierać) lub tych wolnych (takich, które można dołożyć do rozwiązania budowanego przez mrówkę). Może mieć zmienną długość lub stałą. Stała długość wektora to n - liczba elementów. Zawiera wartości 0 - nie użyty element i 1 - element użyty.
   - f – funkcja celu, szukamy jej maksimum

     $$f(ant) = \frac{\sum_{i=1}^{N} (F_i/b)^k}{N},$$

     gdzie:

- F_i – to wypełnienie i-tego pojemnika,
- b – maksymalna pojemność pojemników,
- N – liczba pojemników użyta w rozwiązaniu,
- k wartość skalująca - karząca za zostawienie dużo wolnego miejsca.

N trzeba wyznaczyć analizując ciąg mrówki. Zaczynamy od elementu pierwszego dodajemy jego wagę do pierwszego pojemnika, potem następny element, dopóki mieszczą się w pojemniku (czyli suma $< b$). Jeżeli włożenie kolejnego elementu przekroczy b, to dodajemy kontener.

2. **Tablica wymiany informacji w mrowisku**

   τ — rozkład feromonu, tabla przechowuje informacje dla kolonii o różnych rozkładach elementów w kontenerach (kolejnościach) i ich atrakcyjności. Rozmiar $n \times n$ - $\tau_{i,j}$ oznacza atrakcyjność pojawienia się kolejno elementów i i j. np. jeżeli $n = 5$ mrówka równa się $\{2, 3, 1, 5, 4\}$. Przy takim kodowaniu mrówka wykorzystuje z tablicy τ elementy $\tau_{2,3}, \tau_{3,1}, \tau_{1,5}, \tau_{5,4}$. Tablica na przekątnej ma ustawione 0.

3. **Parametry początkowe algorytmu**

   Przed uruchomieniem algorytmu należy arbitralnie podać wielkości parametrów:

   - $colony\_size$ — liczba mrówek np. 10.
   - α — parametr kontrolujący wpływ feromon $1 < \beta < 5$ np 1.
   - β — parametr kontrolujący wpływ heurystyki $1 < \beta < 5$ np. 1.
   - ρ — współczynnik parowania feromonu $[0, 1]$ np. 0.9.
   - max $\_iter$ — maksymalna liczba iteracji np. 100
   - k — współczynnik funkcji celu $f(ant)$ np. 2.

4. **Inicjalizacja**

   Ustawienie początkowych parametrów tablicy τ.
   Inicjalizacja *best* może być dowolną losowa permutacją.

```
1: best ← randperm(n)
2: for i = 1 to n do
3:    for j = 1 to n do
4:       if i ≠ j then
5:          τ_{i,j} = 1/n
6:       else
7:          τ_{i,j} = 0
8:       end if
9:    end for
10: end for
```

5. **Obliczenie prawdopodobieństwa wyboru elementu do ścieżki** $p(m) \leftarrow compute\_prob$

$$p_{m_{i,j}} = \begin{cases} \frac{\tau_{i,j}^{\alpha} \cdot \eta_{i,j}^{\beta}}{\sum_{s \in \mathcal{N}_i^m} \tau_{i,s}^{\alpha} \cdot \eta_{i,s}^{\beta}}, & \text{if } j \in \mathcal{N}_i^m \\ 0, & \text{otherwise} \end{cases}$$

gdzie:

- $\eta_{i,j}$ może być dowolną dopasowaną heurystyką. W tym przykładzie realizowana jest heurystyka - najpierw najcięższy(największy). Zatem heurystyka powinna być wprost proporcjonalna do wagi elementu.

$$\eta_{i,j} = a_j$$

- $\mathcal{N}_i^m$ to zbiór wszystkich węzłów dostępnych z i (sąsiadów) określonych na podstawie pamięci mrówki M.

6. **Uaktualnienie wektora M** – $update(M)$

   Należy dodać wybrany w kroku poprzednim element (o ile lista ma zmienną długość), lub zmienić 0 (element nie włożony do pojemnika) na 1 (element włożony do pojemnika).

7. **Uaktualnienie feromonu** $deposit\_pheromon(A)$

   Każda mrówka m po zbudowaniu rozwiązania odkłada feromon proporcjonalnie do jakości zbudowanego rozwiązania.

   (a) Oblicza swoją poprawkę dla feromonu

$$\Delta\tau_{i,j}^m = \begin{cases} f(m) & \text{jeżeli } (i,j) \text{ było odwiedzone przez mrówkę(m)} \\ 0, & \text{otherwise} \end{cases}$$

   (b) Następnie oblicza się nowe wartości feromonu:

$$\tau_{i,j} \leftarrow \rho \cdot \tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^m$$

3 Wymagania dla implementacji

Język implementacji algorytmu jest dowolny.

Do testowania należy wykorzystać paczkę z testowymi przykładami (benchmarki) do rozwiązania. Każdy plik testowy zawiera następujące dane [1]

1. wiersz pierwszy to liczba elementów n,

[1] Więcej plików testowych można znaleźć na stronie `https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm`

2. wiersz drugi to wartość b czyli maksymalną pojemność pojemników,

3. kolejne n wierszy to wartości wag/wielkości elementów $a_j, j \in \{0..n-1\}$.

W nazwie benchmarku pierwszy człon przed znakiem „_" jest minimalnym rozwiązaniem globalnym.

Wymagane elementy programu:

- Wczytać benchmark/plik testowy z pliku.
- Zaimplementować algorytm zgodnie z wytycznymi podanymi w punkcie 2 i 2.1.
- Program po wykonaniu optymalizacji powinien wyświetlać:
  - stan, który został wskazany przez program jako rozwiązanie i jego wartość: ile pojemników użyto i jak rozdzielono elementy do pojemników oraz podać sumę wag z każdego pojemnika
  - demonstrować w postaci wykresu (ewentualnie listy) zmienność wartości funkcji f w procesie optymalizacji. Na osi odciętych powinny być przyrastające wartości idx a na osi rzędnych wartości funkcji f.
- Ponadto uruchomić algorytm dziesięciokrotnie i podać średnią arytmetyczną z funkcji f w 10 uruchomieniach oraz odchylenie standardowe.

Każde własne (rozsądne) propozycje modyfikacji lub wykorzystane z literatury zmiany w algorytmie lub reprezentacji lub prezentacji wyników ponad te, wymagane w instrukcji, będą dodatkowo punktowane.

3.1 Przekazanie programu

Kod z rozwiązaniem proszę wysłać na mail: jkolodziejczyk@zut.edu.pl