Podstawy Podejmowania Decyzji Projekt

Rozwiązanie problemu: Latin Square za pomocą IBM ILOG CPLEX oraz Symulowane Wyżarzanie Latin Square i Tabu Search Latin Square

Bartosz Borkowski

Spis treści

1. Wstęp

Cel Sprawozdania

Celem niniejszego sprawozdania jest przedstawienie metodologii oraz wyników rozwiązania problemu Latin Square (Kwadratu Łacińskiego) przy użyciu dwóch różnych podejść: optymalizacji matematycznej za pomocą CPLEX oraz heurystycznej metody symulowanego wyżarzania. Sprawozdanie ma na celu porównanie efektywności obu metod w kontekście jakości uzyskanych rozwiązań oraz czasu obliczeń.

Definicja Problemu

Problem Latin Square polega na znalezieniu kwadratu n x n, w którym każda z 1..n różnych wartości pojawia się dokładnie raz w każdym wierszu i dokładnie raz w każdej kolumnie. Jest to problem klasyczny z zakresu kombinatoryki i znajduje zastosowanie w wielu dziedzinach, takich jak projektowanie eksperymentów, kryptografia czy teoria kodów.

Tło

Nazwa "Latin square" została zainspirowana przez prace matematyczne Leonharda Eulera (1707–1783), który używał łacińskich znaków jako symboli, ale dowolny zbiór symboli może być użyty: w powyższym przykładzie, sekwencję liter A, B, C można zastąpić sekwencją liczb całkowitych 1, 2, 3. Euler zapoczątkował ogólną teorię kwadratów łacińskich.

1	2	3
2	3	1
3	1	2

Rys. 1 Przykładowy latin square 3x3

2. Sformułowanie problemu optymalizacji

2.1 Opis modelu

Model będzie miał za zadanie przy ustalonym rozmiarze i wejściowej kwadratowej macierzy (może być też pusta) wypełnić ją w taki sposób, aby nie łamała żadnej z zasad Latin Square:

1. Żadna z cyfr lub liczb nie może się powtarzać w wierszu
2. Żadna z cyfr lub liczb nie może się powtarzać w kolumnie
3. Ważnym założeniem jest też to, żeby cyfry lub liczby którymi wypełniamy macierz pochodziły z określonego przedziału

Możemy więc wyróżnić dwie zmienne wejściowe, o których decyduje użytkownik:

Rozmiar kwadratu:

$$n$$

Oraz

Macierz wejścia:

$$Z = [\boxdot]_{n\ x\ n}$$

$$Z' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}_{4\ x\ 4}$$

Rys. 2

Rysunek 2 prezentuje przykładową macierz Z’ przy n = 4, gdzie wypełnione zostały 3 komórki macierzy.

Komentarz do punktu 3
w tym sprawozdaniu przedziałem z którego będą pochodziły cyfry lub liczby będzie przedział $\{1, 2, \dots, n\}$ *, ale można go dowolnie modyfikować według uznania użytkownika*

Co jest optymalizowane?

IBM ILOG CPLEX jest środowiskiem, które pozwala na rozwiązanie nie proceduralne, gdzie ważne są ograniczenia problemu i w tym wypadku decyzje będą podejmowane zgodnie z ograniczeniami, w przeciwieństwie do symulowanego wyżarzania, gdzie bezpośrednio optymalizujemy funkcje energii.

2.2 Przedstawienie i uzasadnienie zmiennych decyzyjnych

Zmienne decyzyjne w modelu optymalizacyjnym reprezentują decyzje, które muszą być podjęte, aby uzyskać poprawnie wypełniony kwadrat łaciński. W kontekście CPLEX, Symulowanego Wyżarzania oraz Tabu Search, zmienną decyzyjną będzie macierz X, która jest wynikową macierzą algorytmu.
Innymi słowy, X jest poprawnie wypełnioną (lub bliską poprawnego wypełnienia w przypadku heurystyk) wersją macierzy wejściowej Z.
Model będzie wypełniać macierz Z tam, gdzie nie została podana wartość wejściowa, wartościami x_{ij},

gdzie:

i – indeks wiersza, i ∈ $\{1, 2, \dots, n\}$,

j – indeks kolumny, j ∈ $\{1, 2, \dots, n\}$,

x_{ij} – wartość wpisana przez algorytm w i-tym wierszu i j-tej kolumnie

$X = \left[x_{ij}\right]_{n\,x\,n}$ – macierz wynikowa (wypełniony Latin Square)

2.3 Przedstawienie i uzasadnienie funkcji celu i ograniczeń

Ograniczenia

Jak wspomniałem w Opisie Modelu (2.1) w tym problemie mamy do czynienia z trzema ograniczeniami:

1. Żadna z cyfr lub liczb nie może się powtarzać w wierszu
2. Żadna z cyfr lub liczb nie może się powtarzać w kolumnie
3. Ważnym założeniem jest też to, żeby cyfry lub liczby którymi wypełniamy macierz pochodziły z określonego przedziału.

Przedstawmy je zatem matematycznie:

1. $\forall \mathrm{i} \in \{1, \dots, \mathrm{n}\}, \quad \forall\, \mathrm{j}_1, \mathrm{j}_2 \in \{1, \dots, \mathrm{n}\}, \quad \mathrm{j}_1 < \mathrm{j}_2 \quad \Rightarrow \quad \mathrm{x}_{ij_1} \neq \mathrm{x}_{ij_2}$
2. $\forall \mathrm{j} \in \{1, \dots, \mathrm{n}\}, \quad \forall\, \mathrm{i}_1, \mathrm{i}_2 \in \{1, \dots, \mathrm{n}\}, \quad \mathrm{i}_1 < \mathrm{i}_2 \quad \Rightarrow \quad \mathrm{x}_{i_1 j} \neq \mathrm{x}_{i_2 j}$
3. $\forall \mathrm{i} \in \{1, \dots, \mathrm{n}\}, \ \forall \mathrm{j} \in \{1, \dots, \mathrm{n}\} \ \Rightarrow \ 1 \leq \mathrm{x}_{ij} \leq n$

Funkcja Celu

Funkcja celu dla tego problemu może być zdefiniowana na wiele sposobów. W tym artykule zaimplementowałem metodę powtórzeń, która będzie zliczać powtórzenia w wierszach i kolumnach.
Innymi słowy, za każde powtórzenie cyfry/liczby funkcja celu będzie zwiększała się o jeden.
Wybrałem akurat te funkcje, ponieważ naszym celem jest uzyskanie idealnego Latin Square, co oznacza, że wówczas gdy wystąpią powtórzenia Latin Square przestaje być Latin Squarem.

Jak wytłumaczyłem wcześniej w ***„Co jest optymalizowane?”*** W środowisku CPLEX nie jest potrzebna funkcja celu dla tego problemu.
W kwestii Symulowanego Wyżarzania i Tabu Search rozwinę się w osobnych sekcjach, kolejno 2.4 i 2.5.

2.4 Dodatkowe informacje o Symulowanym wyżarzaniu

Symulowane Wyżarzanie jest algorytmem, który korzysta z:

1. **Startowej propozycji** rozwiązania, w tym sprawozdaniu nazwałem ją **funkcją startu** (bo będę ja generować z określnej macierzy wejściowej Z) (Rys. Kod. 1)
2. **Funkcji kandydatów**, czyli funkcji, która generuje „sąsiednie rozwiązanie” w celu porównania ja z dotychczasowym najlepszym wynikiem, w tym sprawozdaniu będzie ona polegała na zamianie albo w wierszu, albo w kolumnie (decydować o tym będzie rozkład jednostajny) dwóch cyfr (które nie były wpisane pierwotnie w macierzy Z) (decydować będzie o tym rozkład jednostajny dla indeksów)
   (Rys. Kod. 2)
3. **Funkcja energii**, czyli funkcji celu, dla tej metody, która zlicza powtórzenia w wierszach i kolumnach.
   (Rys. Kod. 3)
4. **Funkcja akceptacji**, czyli funkcji, która na podstawie Temperatury i różnicy energii będzie odrzucać lub akceptować propozycje macierzy uzyskanej z Funkcji kandydatów.
   (Rys. Kod. 4)

Kody można znaleźć w sekcji „5. Fragmenty kodu źródłowego”

2.5 Dodatkowe informacje o Tabu Search

Tabu Search został zaimplementowany bardzo podobnie do Symulowanego Wyżarzania więc korzysta on z tych samych/podobnych funkcji:

1. **Startowej propozycji** rozwiązania, w tym sprawozdaniu nazwałem ją **funkcją startu** (bo będę ja generować z określnej macierzy wejściowej Z) (Rys. Kod. 1)
2. **Funkcji kandydatów**, czyli funkcji, która generuje „sąsiednie rozwiązanie” w celu porównania ja z dotychczasowym najlepszym wynikiem,
   w tym sprawozdaniu będzie ona polegała na zamianie albo w wierszu, albo w kolumnie (decydować o tym będzie rozkład jednostajny) dwóch cyfr (które nie były wpisane pierwotnie w macierzy Z) (decydować będzie o tym rozkład jednostajny dla indeksów),
   tutaj pojawia się jedyna różnica, ponieważ tabu search realizuje tę funkcje odrobine inaczej, zaznaczyłem je komentarzami, które można zobaczyć na Rys. Kod. 5 w sekcji „5. Fragmenty kodu źródłowego”
   (Rys. Kod. 5)
3. **Funkcja energii**, czyli funkcji celu, dla tej metody, która zlicza powtórzenia w wierszach i kolumnach.
   (Rys. Kod. 3)

Kody można znaleźć w sekcji „5. Fragmenty kodu źródłowego”

3. Rozwiązanie

3.1 Wejścia dla każdej metody

Wykonam 7 wejść dla każdej z metod.
Będą one wyglądać następująco:

1. Latin square 3x3 z pustym wejściem

   Rozmiar kwadratu:

   $$n = 3$$

   Oraz

   Macierz wejścia:

   $$Z_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}_{3x3}$$

2. Latin square 4x4 z prostym wejściem

   Rozmiar kwadratu:

   $$n = 4$$

   Oraz

   Macierz wejścia:

   $$Z_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}_{4x4}$$

3. Latin square 5x5 z prostym wejściem

   Rozmiar kwadratu:

   $$n = 5$$

   Oraz

   Macierz wejścia:

   $$Z_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}_{5x5}$$

4. Latin square 5x5 z większym wejściem

Rozmiar kwadratu:

$$n = 5$$

Oraz

Macierz wejścia:

$$Z_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 3 \\ 0 & 2 & 0 & 3 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ 3 & 0 & 0 & 0 & 5 \end{bmatrix}_{5x5}$$

5. Latin square 8x8 z pustym wejściem

Rozmiar kwadratu:

$$n = 8$$

Oraz

Macierz wejścia:

$$Z_5 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{8x8}$$

6. Latin square 8x8 z prostym wejściem

Rozmiar kwadratu:

$$n = 8$$

Oraz

Macierz wejścia:

$$Z_6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}_{8x8}$$

7. Latin square 4x4 z **<u>BŁĘDNYM</u>** wejściem

Rozmiar kwadratu:

$$n = 4$$

Oraz

Macierz wejścia:

$$Z_7 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}_{4x4}$$

3.2 CPLEX ILOG

Tutaj znajdują się wyniki uzyskane poprzez CPLEX ILOG dla podanych wcześniej 7 wejść,
Rozwiązany kwadraty Z_x będzie oznaczony jako Z_x^*,
a jako C_x^* oznaczony będzie czas potrzebny na wykonanie
(gdzie x to numer wejścia):

1. Latin square 3x3 z pustym wejściem

$Z_1^* =$

3	1	2
1	2	3
2	3	1

$C_1^* =$ **00:00:00:16**

2. Latin square 4x4 z prostym wejściem

$Z_2^* =$

1	4	2	3
4	1	3	2
3	2	4	1
2	3	1	4

$C_2^* =$ **00:00:00:15**

3. Latin square 5x5 z prostym wejściem

$Z_3^* =$

1	4	5	2	3
5	2	1	3	4
4	1	3	5	2
3	5	2	4	1
2	3	4	1	5

$C_3^* =$ **00:00:00:15**

4. Latin square 5x5 z wiekszym wejściem

$Z_4^* =$

1	5	4	2	3
5	2	1	3	4
4	1	3	5	2
2	3	5	4	1
3	4	2	1	5

$C_4^* =$ **00:00:00:16**

5. Latin square 8x8 z pustym wejściem

$Z_5^* =$

1	7	2	6	4	3	5	8
4	2	8	1	5	6	3	7
6	3	4	5	8	7	1	2
7	8	3	4	2	1	6	5
2	4	6	7	1	5	8	3
8	6	5	3	7	2	4	1
5	1	7	8	3	4	2	6
3	5	1	2	6	8	7	4

$C_5^* =$ **00:00:00:20**

6. Latin square 8x8 z prostym wejściem

$Z_6^* =$

1	5	7	8	6	3	4	2
8	1	4	5	3	2	6	7
5	4	3	2	8	6	7	1
4	6	1	3	7	8	2	5
3	2	8	7	5	4	1	6
7	8	6	1	2	5	3	4
2	7	5	6	4	1	8	3
6	3	2	4	1	7	5	8

$C_6^* =$ **00:00:00:17**

7. Latin square 4x4 z **<u>BŁĘDNYM</u>** wejściem

$Z_7^* =$ No value

$C_7^* =$ **00:00:00:31**

3.3 Symulowane wyżarzanie w Python

Tutaj znajdują się wyniki uzyskane poprzez Symulowane Wyżarzanie dla podanych wcześniej 6 wejść,
Rozwiązany kwadraty Z_x będzie oznaczony jako Z_x^* (jest to najlepszy wynik, rozpatrując kryterium jakości jako funkcje energii z maksymalnie 10 prób wyżarzania),
a jako C_x^* oznaczony będzie czas potrzebny na wykonanie (w sekundach) (jest to czas z sumy maksymalnie 10 prób wyżarzania)
dodatkowo przez F_x^* oznaczone będzie końcowa wartość Funkcji Energii
(gdzie x to numer wejścia):

1. Latin square 3x3 z pustym wejściem

$$Z_1^* = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 3 & 2 \\ 3 & 2 & 1 \end{bmatrix}_{3x3}$$

$$C_1^* = 0.028955459594726562$$

$$F_1^* = 0$$

2. Latin square 4x4 z prostym wejściem

$$Z_2^* = \begin{bmatrix} 1 & 3 & 2 & 4 \\ 4 & 1 & 3 & 2 \\ 3 & 2 & 4 & 1 \\ 2 & 4 & 1 & 3 \end{bmatrix}_{4x4}$$

$$C_2^* = 0.3311138153076172$$

$$F_2^* = 0$$

3. Latin square 5x5 z prostym wejściem

$$Z_3^* = \begin{bmatrix} 1 & 3 & 4 & 5 & 2 \\ 4 & 2 & 5 & 1 & 3 \\ 5 & 1 & 3 & 2 & 4 \\ 3 & 5 & 2 & 4 & 1 \\ 2 & 4 & 1 & 3 & 5 \end{bmatrix}_{5x5}$$

$C_3^* = 0.6353309154510498$

$F_3^* = 0$

4. Latin square 5x5 z wiekszym wejściem

$$Z_4^* = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 \\ 5 & 2 & 1 & 3 & 4 \\ 4 & 5 & 3 & 1 & 2 \\ 2 & 3 & 5 & 4 & 1 \\ 3 & 1 & 4 & 2 & 5 \end{bmatrix}_{5x5}$$

$C_4^* = 0.15358591079711914$

$F_4^* = 0$

5. Latin square 8x8 z pustym wejściem

$$Z_5^* = \begin{bmatrix} 3 & 6 & 5 & 7 & 2 & 4 & 8 & 1 \\ 2 & 8 & 4 & 3 & 5 & 1 & 7 & 6 \\ 5 & 3 & 7 & 2 & 1 & 6 & 4 & 8 \\ 6 & 2 & 8 & 5 & 3 & 7 & 1 & 4 \\ 4 & 8 & 1 & 6 & 7 & 5 & 3 & 2 \\ 1 & 5 & 2 & 8 & 4 & 7 & 6 & 3 \\ 7 & 1 & 6 & 4 & 8 & 3 & 2 & 5 \\ 8 & 4 & 3 & 1 & 6 & 2 & 5 & 7 \end{bmatrix}_{8x8}$$

$C_5^* = 2.622025728225708$

$F_5^* = 2$

6. Latin square 8x8 z prostym wejściem

$$Z_6^* = \begin{bmatrix} 1 & 2 & 6 & 5 & 8 & 4 & 3 & 7 \\ 3 & 1 & 4 & 7 & 6 & 8 & 5 & 2 \\ 7 & 6 & 3 & 8 & 1 & 4 & 2 & 5 \\ 8 & 5 & 1 & 3 & 7 & 2 & 6 & 4 \\ 2 & 3 & 8 & 4 & 5 & 7 & 1 & 6 \\ 6 & 8 & 2 & 1 & 4 & 5 & 7 & 3 \\ 5 & 4 & 7 & 6 & 2 & 3 & 8 & 1 \\ 6 & 7 & 5 & 2 & 3 & 1 & 4 & 8 \end{bmatrix}_{8x8}$$

$C_6^* = 2.626970052719116$

$F_6^* = 2$

7. Latin square 4x4 z **<u>BŁĘDNYM</u>** wejściem

$$Z_7^* = \begin{bmatrix} 1 & 3 & 2 & 4 \\ 3 & 2 & 4 & 1 \\ 4 & 1 & 3 & 2 \\ 1 & 4 & 2 & 3 \end{bmatrix}_{4x4}$$

$C_7^* = 0.7350313663482666$

$F_7^* = 2$

3.4 Tabu Search w Python

Tutaj znajdują się wyniki uzyskane poprzez Symulowane Wyżarzanie dla podanych wcześniej 6 wejść,
Rozwiązany kwadraty Z_x będzie oznaczony jako Z_x^*
a jako C_x^* oznaczony będzie czas potrzebny na wykonanie (w sekundach) dodatkowo przez F_x^* oznaczone będzie końcowa wartość Funkcji Energii (gdzie x to numer wejścia):

1. Latin square 3x3 z pustym wejściem

$$Z_1^* = \begin{bmatrix} 3 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}_{3x3}$$

$$C_1^* = 0.17357087135314941$$

$$F_1^* = 0$$

2. Latin square 4x4 z prostym wejściem

$$Z_2^* = \begin{bmatrix} 1 & 4 & 2 & 3 \\ 2 & 1 & 3 & 4 \\ 3 & 2 & 4 & 1 \\ 4 & 3 & 1 & 2 \end{bmatrix}_{4x4}$$

$$C_2^* = 0.30713963508605957$$

$$F_2^* = 0$$

3. Latin square 5x5 z prostym wejściem

$$Z_3^* = \begin{bmatrix} 1 & 1 & 5 & 3 & 2 \\ 4 & 2 & 4 & 5 & 3 \\ 2 & 5 & 3 & 1 & 4 \\ 5 & 3 & 2 & 4 & 1 \\ 3 & 4 & 1 & 2 & 5 \end{bmatrix}_{5x5}$$

$C_3^* = 0.3719921112060547$

$F_3^* = 2$

4. Latin square 5x5 z wiekszym wejściem

$$Z_4^* = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 \\ 4 & 2 & 5 & 3 & 1 \\ 2 & 5 & 3 & 1 & 4 \\ 5 & 3 & 1 & 4 & 2 \\ 3 & 1 & 4 & 2 & 5 \end{bmatrix}_{5x5}$$

$C_4^* = 0.43283915519714355$

$F_4^* = 0$

5. Latin square 8x8 z pustym wejściem

$$Z_5^* = \begin{bmatrix} 4 & 8 & 5 & 5 & 3 & 6 & 1 & 7 \\ 7 & 2 & 3 & 8 & 6 & 1 & 4 & 8 \\ 5 & 4 & 2 & 1 & 6 & 7 & 3 & 8 \\ 1 & 7 & 6 & 3 & 5 & 2 & 6 & 4 \\ 2 & 3 & 7 & 6 & 1 & 4 & 8 & 5 \\ 6 & 7 & 1 & 8 & 4 & 3 & 5 & 2 \\ 6 & 1 & 8 & 4 & 7 & 5 & 2 & 3 \\ 3 & 5 & 4 & 2 & 8 & 2 & 7 & 1 \end{bmatrix}_{8x8}$$

$C_5^* = 0.7221004962921143$

$F_5^* = 10$

6. Latin square 8x8 z prostym wejściem

$$Z_6^* = \begin{bmatrix} 1 & 2 & 8 & 4 & 3 & 6 & 7 & 5 \\ 5 & 1 & 6 & 8 & 7 & 4 & 2 & 3 \\ 2 & 1 & 3 & 5 & 4 & 8 & 6 & 7 \\ 8 & 4 & 1 & 3 & 2 & 6 & 5 & 7 \\ 7 & 3 & 1 & 8 & 5 & 2 & 6 & 4 \\ 6 & 2 & 7 & 4 & 8 & 5 & 3 & 1 \\ 5 & 7 & 4 & 6 & 1 & 3 & 8 & 2 \\ 4 & 5 & 2 & 3 & 6 & 1 & 7 & 8 \end{bmatrix}_{8x8}$$

$C_6^* = 0.7420132160186768$

$F_6^* = 11$

7. Latin square 4x4 z **<u>BŁĘDNYM</u>** wejściem

$$Z_6^* = \begin{bmatrix} 1 & 4 & 2 & 3 \\ 2 & 4 & 3 & 1 \\ 3 & 2 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}_{4x4}$$

$C_6^* = 0.21640920639038086$

$F_6^* = 2$

4. Podsumowanie

W niniejszej pracy porównano trzy algorytmy optymalizacyjne: CPLEX, symulowane wyżarzanie oraz tabu search, w kontekście ich efektywności i dokładności w rozwiązywaniu problemu Latin Square. Wyniki badań jednoznacznie wskazują, że CPLEX zdecydowanie przewyższa pozostałe metody zarówno pod względem szybkości, jak i precyzji rozwiązania.

Analiza Algorytmów

1. **CPLEX**: CPLEX, zaprojektowany do rozwiązywania problemów optymalizacyjnych, wykazuje najwyższą efektywność. Algorytm ten szybko i precyzyjnie znajduje rozwiązania dla każdego zadania, nie obsługując jednak błędnych danych wejściowych. CPLEX uznałem za najbardziej odpowiedni do rozważanego problemu ze względu na jego deterministyczny charakter i brak elementu losowości, co gwarantuje powtarzalność wyników.
2. **Symulowane Wyżarzanie**: Wyżarzanie, choć czasami wolniejsze od CPLEX, w większości przypadków radzi sobie z problemem stosunkowo dobrze. Algorytm ten charakteryzuje się elementem losowości, co może prowadzić do pewnej nieprzewidywalności wyników. Niemniej jednak, przy odpowiedniej liczbie prób, metoda ta osiąga wyniki zbliżone do CPLEX. Ponadto, wyżarzanie potrafi obsługiwać błędne dane wejściowe, co może być zaletą lub wadą w zależności od kontekstu zastosowania.
3. **Tabu Search**: Tabu search okazał się najmniej efektywny spośród analizowanych metod. Algorytm ten jest zarówno wolny, jak i niedokładny, co znacząco ogranicza jego użyteczność w rozważanym problemie. Podobnie jak wyżarzanie, tabu search jest metodą probabilistyczną, co może prowadzić do zmienności wyników. Mimo iż również obsługuje błędne dane wejściowe, jego niska efektywność czyni go mniej atrakcyjnym wyborem.

Wnioski

Badania wykazały, że wyniki generowane przez poszczególne algorytmy rzadko się powtarzają. W niektórych przypadkach zaobserwowano identyczne rozwiązania w dwóch wierszach lub kolumnach, co jednak było rzadkością. Warto podkreślić, że zarówno wyżarzanie, jak i tabu search są metaheurystykami, a ich probabilistyczna natura wprowadza element losowości. Może to prowadzić do sytuacji, w której czasami algorytmy te są niezwykle dokładne, a innym razem ich wyniki są niewystarczające.

Podsumowując, CPLEX jest najefektywniejszym narzędziem do rozwiązywania analizowanego problemu optymalizacyjnego, podczas gdy wyżarzanie może być stosowane jako alternatywa, oferując pewną elastyczność kosztem deterministycznej pewności. Tabu search, choć mniej efektywny, może znaleźć zastosowanie w specyficznych przypadkach wymagających obsługi błędnych danych wejściowych.

5. Fragmenty kodu źródłowego

```python
def f_startowa(kwadrat, przedzial):
    for i in range(len(kwadrat)):
        dostepne_cyfry = set(przedzial) - set([kwadrat[i][j] for j in range(len(kwadrat)) if kwadrat[i][j] != 0])
        dostepne_cyfry = list(dostepne_cyfry)
        random.shuffle(dostepne_cyfry)
        for j in range(len(kwadrat)):
            if kwadrat[i][j] == 0:
                kwadrat[i][j] = dostepne_cyfry.pop(0)
    return kwadrat
```

Rys. Kod. 1

```python
def f_kandydatow(kwadrat):
    def zamien_wiersz(kwadrat):
        i = random.randint(0, len(kwadrat) - 1)
        j, k = random.sample(range(len(kwadrat)), 2)
        while(kwadrat[i][j] == zarezerwowane_spoty[i][j] or kwadrat[i][k] == zarezerwowane_spoty[i][k]):
            i = random.randint(0, len(kwadrat) - 1)
            j, k = random.sample(range(len(kwadrat)), 2)
        kwadrat[i][j], kwadrat[i][k] = kwadrat[i][k], kwadrat[i][j]

    def zamien_kolumne(kwadrat):
        j = random.randint(0, len(kwadrat) - 1)
        i, k = random.sample(range(len(kwadrat)), 2)
        while(kwadrat[i][j] == zarezerwowane_spoty[i][j] or kwadrat[k][j] == zarezerwowane_spoty[k][j]):
            j = random.randint(0, len(kwadrat) - 1)
            i, k = random.sample(range(len(kwadrat)), 2)
        kwadrat[i][j], kwadrat[k][j] = kwadrat[k][j], kwadrat[i][j]

    # Losowo wybieramy, czy chcemy zamienić wiersz czy kolumnę
    if random.random() < 0.5:
        zamien_wiersz(kwadrat)
    else:
        zamien_kolumne(kwadrat)

    return kwadrat
```

Rys. Kod. 2

```python
def f_energi(kwadrat):
    wartosc_f = 0

    # kolumny
    for i in range(len(kwadrat)):
        for j in range(len(kwadrat)):
            for k in range(j + 1, len(kwadrat)):
                if kwadrat[i][j] == kwadrat[i][k]:
                    wartosc_f += 1
    # wiersze
    for j in range(len(kwadrat)):
        for i in range(len(kwadrat)):
            for k in range(i + 1, len(kwadrat)):
                if kwadrat[i][j] == kwadrat[k][j]:
                    wartosc_f += 1
    return wartosc_f
```

Rys. Kod. 3

```python
def f_akceptacji(T, d_E):
    if d_E < 0:
        return True
    else:
        r = random.random()
        if r < math.exp(-d_E / T):
            return True
        else:
            return False
```

Rys. Kod. 4

```python
def f_kandydatow(kwadrat, tabu_list):
    def zamien_wiersz(kwadrat):
        i = random.randint(0, len(kwadrat) - 1)
        j, k = random.sample(range(len(kwadrat)), 2)
        while (kwadrat[i][j] == zarezerwowane_spoty[i][j] or kwadrat[i][k] == zarezerwowane_spoty[i][k]):
            i = random.randint(0, len(kwadrat) - 1)
            j, k = random.sample(range(len(kwadrat)), 2)
        kwadrat[i][j], kwadrat[i][k] = kwadrat[i][k], kwadrat[i][j]
        return i, j, k  # Zwracamy ruch

    def zamien_kolumne(kwadrat):
        j = random.randint(0, len(kwadrat) - 1)
        i, k = random.sample(range(len(kwadrat)), 2)
        while (kwadrat[i][j] == zarezerwowane_spoty[i][j] or kwadrat[k][j] == zarezerwowane_spoty[k][j]):
            j = random.randint(0, len(kwadrat) - 1)
            i, k = random.sample(range(len(kwadrat)), 2)
        kwadrat[i][j], kwadrat[k][j] = kwadrat[k][j], kwadrat[i][j]
        return i, j, k  # Zwracamy ruch

    # Generowanie kandydatów z uwzględnieniem tabu_list
    while True:
        if random.random() < 0.5:
            ruch = zamien_wiersz(kwadrat)
        else:
            ruch = zamien_kolumne(kwadrat)

        # Sprawdzamy, czy ruch jest dozwolony (nie ma go na liście tabu)
        if ruch not in tabu_list:
            return kwadrat, ruch
```

Rys. Kod. 5