Maria Koprowska 136738

Informatyka, 3 semestr, grupa i3

Sprawozdanie

Zastosowanie algorytmów genetycznych w problemie szeregowania zadań

Spis treści

1. Treść zadania

- Problem nr 2,
- Flow-shop, liczba maszyn m = 2,
- liczba zadań n, liczba operacji w ramach zadania = m = 2,
- operacje niewznawialne,
- na drugiej maszynie okresy bezczynności maszyny ustalane przez algorytm wyszukiwania (czas trwania – losowy, ale minimum 1.5 x średni czas operacji dla M2),
- każda operacja poza pierwszą w uszeregowaniu na M2 oraz poza pierwszą po każdym okresie bezczynności na M2 ma realny czas wykonywania większy o 10% (kumulacja!). Okres bezczynności resetuje tę karę. Kumulacja = +0%, +10%, +20%, +30%, itd. aż do okresu bezczynności na M2, potem znowu +0%, +10%, itd.),
- minimalizacja sumy czasów zakończenia wszystkich operacji

Implementacja problemu została wykonana w języku Python 3.6 w środowisku programistycznym PyCharm Professional 2018.2.1.

2. Implementacja

a) Constants – zawiera stałe, które wynikają z treści zadania.

```
MAX_TASK_TIME = 15
MIN_TASK_TIME = 1
JOBS_NUMBER = 50  # n z treści zadania
MAX_MAINTENENANCE_TIME = 30
```

b) Machine (maszyna)

Klasa reprezentująca maszynę 1 lub 2 (przyjmujemy oznaczenia M1 i M2). Przelicza czas wykonania tasków, zwraca sumę czasów zakończenia wszystkich operacji. Druga maszyna do każdego kolejnego tasku dodaje karę – wydłuża czas wykonywania zadania o 10%, kary się kumulują.

c) Job (zadanie)

Obiekt składa się z dwóch tasków, które odpowiednio przekazywane są na maszynę 1 lub 2.

d) Task (część zadania)

Task to część zadania wykonywana na odpowiedniej maszynie. Pierwsza część musi zostać wykonana na pierwszej maszynie zanim na drugiej rozpocznie się wykonywanie drugiej części.

e) Maintenance (okres bezczynności)

Maintenance'y występują tylko na drugiej maszynie, ich wystąpienie resetuje karę. Minimalny czas trwania okresu bezczynności wynosi 1,5 x średni czas operacji na maszynie 2, maksymalny został ustawiony na 30 jednostek.

f) Instance (instancja)

Klasa Instance służy generowaniu i enkapsulacji zbioru zadań o losowych czasach trwania.

g) Solution (rozwiązanie)

Klasa Solution modeluje rozwiązanie w postaci zadanej kolejności operacji na M1 i M2 Oraz tablicy parametrów czasowych uporządkowania.

h) Population (populacja)

Populacja zawiera rozwiązania problemu dla zadanej instancji. Zawiera interfejs służący do operowania na wszystkich zawartych w niej rozwiązaniach.

3. Mutacje

a) Mutacja1 zamienia dwa losowe taski na maszynie M1 oraz dwa taski na M2 o tych samych numerach.

```
def mutate1(self):  # get two random ints and swap tasks with these numbers on M1 and
M2
 x, y = randint(0, JOBS_NUMBER - 1), randint(0, JOBS_NUMBER - 1)
 self.M1[x], self.M1[y] = self.M1[y], self.M1[x]
 self.M2[x], self.M2[y] = self.M2[y], self.M2[x]
 self.score = 0
```

b) Mutacja2 zamienia dwa losowe taski na maszynie M1 oraz dwa losowe taski na maszynie M2.

```
def mutate2(self):  # swap random tasks on M1 and other randoms tasks on M2

r = tuple(randint(0, JOBS_NUMBER - 1) for _ in range(4))
    self.M1[r[0]], self.M1[r[1]] = self.M1[r[1]], self.M1[r[0]]
    self.M1[r[2]], self.M1[r[3]] = self.M1[r[3]], self.M1[r[2]]
    self.score = 0
```

c) Mutacja3 zamienia dwa kolejne taski na M1 i dwa kolejne o tych samych indeksach na M2.

```
def mutate3(self):  # swap two consecutive tasks on M1 and M2
 x, y = randint(0, JOBS_NUMBER - 2), randint(0, JOBS_NUMBER - 2)
 self.M1[x], self.M1[x + 1] = self.M1[x + 1], self.M1[x]
 self.M1[y], self.M1[y + 1] = self.M1[y + 1], self.M1[y]
 self.score = 0
```

d) Mutacja4 zamienia dwa kolejne taski na M1 lub M2.

```
def mutate4(self):  # swap two consecutive tasks on M1 OR M2
    m = (self.M1, self.M2)[randint(0, 1)]
    x = randint(0, JOBS_NUMBER - 2)
    m[x], m[x + 1] = m[x + 1], m[x]
    self.score = 0
```

e) Mutacja5 zamienia dwa losowe taski na M1 lub M2.

```
def mutate5(self): # swap random tasks on M1 or M2
    m = (self.M1, self.M2)[randint(0, 1)]
    x, y = randint(0, JOBS_NUMBER - 2), randint(0, JOBS_NUMBER - 2)
    m[x], m[y] = m[y], m[x]
    self.score = 0
```

4. Crossover (krzyżowanie)

Crossover – krzyżuje każde dwa kolejne rozwiązania z populacji (tzn. pierwsze z drugim, drugie z trzecim itd.), tworząc z nich dwóch potomków, pozostawia oryginalne.

```
elif all(type(i) is Solution for i in argv):  # crossover
 self.instance = argv[0].instance
 self.score = 0
 self.M1 = Machine(1, self)
 self.M2 = Machine(2, self)
 m1h = argv[0].M1[:int(JOBS_NUMBER / 2)]
 self.M1.extend(m1h + [task for task in argv[1].M1 if task not in m1h])
 m2h = argv[0].M2[:int(len(argv[0].M2) / 2)]
 self.M2.extend(m2h + [task for task in argv[1].M2 if task not in m2h])
```

5. Selekcja

Turniej – algorytm układa rozwiązania według ich malejącej jakości i usuwa najgorsze, pozostawiając ich tyle, ile wynosi zadana wielkość populacji.

6. Dobór par do rozmnażania

a) Turniej – rozwiązania rankingujemy i rozmnażamy najlepszy wynik z drugim w kolejności itd.

b) Every Other – rozwiązania rankingujemy, zamieniamy co drugie parzyste z nieparzystymi (tzn. $(1,2,3,4,5,6,7) \rightarrow (1,3,2,4,5,7,6)$).

c) First to Half – rozwiązania rankingujemy, pierwszy wynik rozmnażamy z wynikiem występującym dokładnie w połowie. Dla populacji o wielkości 20 rozmnażamy rozwiązanie 1 z rozwiązaniem 10.

d) Random – miesza za pomocą metody shuffle całą populację.

7. Losowy generator rozwiązań

N zadań zadanych instancją problemu zostaje losowo uporządkowanych (za pomocą metody shuffle) na M1 i M2, po czym na M2 z zadanym prawdopodobieństwem po każdej operacji może zostać wygenerowany maintenance.

8. Strojenie meta-heurystyki

a) Skuteczność poszczególnych mutacji oraz krzyżowania.

Parametry:

100 – liczba generowanych instancji,

1000 – liczba obiegów pętli algorytmu,

50 zadań na maszynach, każde składa się z task1 oraz task2,

20 – wielkość populacji,

25 – procentowe prawdopodobieństwo wystąpienia maintenance'u po każdym zadaniu,

metoda doboru par do rozmnażania: ranking.

Mutacja 1	Mutacja 2	Mutacja 3	Mutacja 4	Mutacja 5	Crossover
39,59%	17,28%	16,59%	24,79%	40,83%	15,68%

Tabela 1: Procentowe poprawy wyników w stosunku do rozwiązań wygenerowanych losowo

b) Skuteczność poszczególnych mutacji złożonych z krzyżowaniem.

Parametry:

100 – liczba generowanych instancji,

1000 – liczba obiegów pętli algorytmu,

50 zadań na maszynach, każde składa się z task1 oraz task2,

20 – wielkość populacji,

25 – procentowe prawdopodobieństwo wystąpienia maintenance'u po każdym zadaniu,

metoda doboru par do rozmnażania: ranking.

Mutacja 1 + C	Mutacja 2 + C	Mutacja 3 + C	Mutacja 4 + C	Mutacja 5 + C
46,53%	32,31%	35,86%	34,87%	49,42%

Tabela 2: Procentowe poprawy wyników w stosunku do rozwiązań wygenerowanych losowo

c) Skuteczność Mutacji5 + krzyżowania w zależności od wielkości populacji

Parametry:

100 – liczba generowanych instancji,

1000 – liczba obiegów pętli algorytmu,

50 zadań na maszynach, każde składa się z task1 oraz task2,

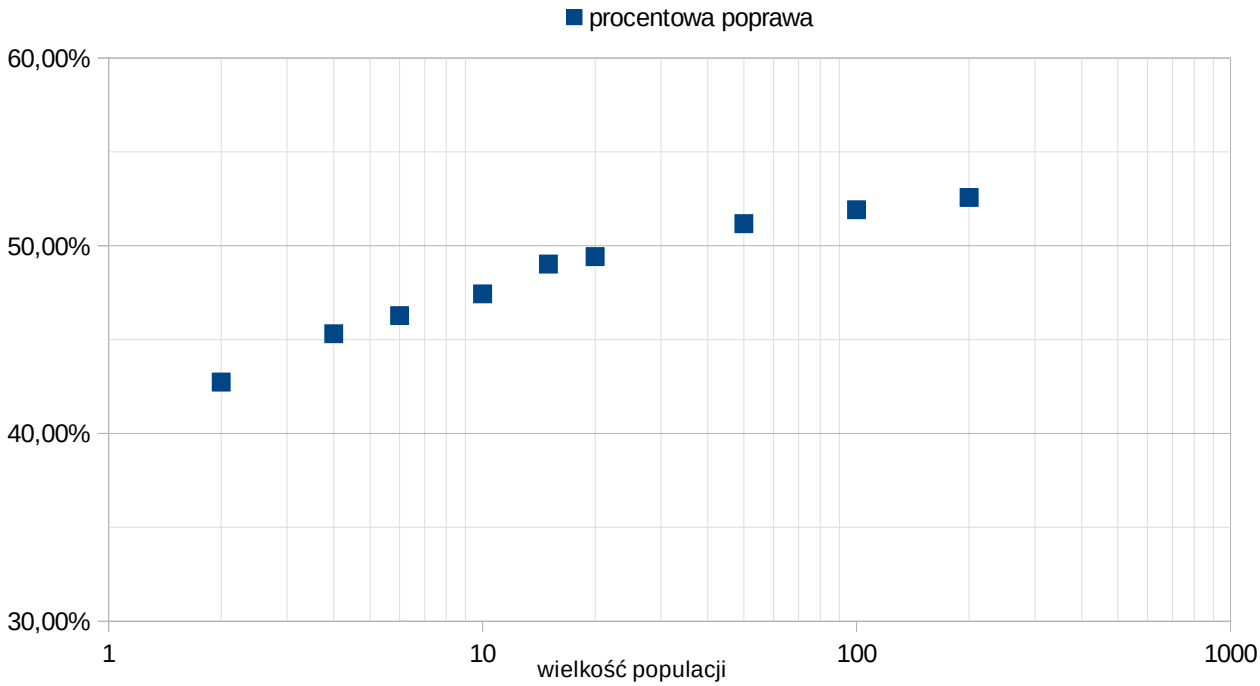
n – wielkość populacji,

25 – procentowe prawdopodobieństwo wystąpienia maintenance'u po każdym zadaniu,

metoda doboru par do rozmnażania: ranking.

n	procentowa poprawa
2	42,74%
4	45,31%
6	46,28%
10	47,43%
15	49,02%
20	49,42%
50	51,17%
100	51,91%
200	52,57%

Tabela 3: Procentowe poprawy wyników w stosunku do rozwiązań wygenerowanych losowo dla różnych wielkości populacji



Wykres 1: Procentowe poprawy wyników w stosunku do rozwiązań wygenerowanych losowo dla różnych wielkości populacji

d) Skuteczność Mutacji5 + krzyżowania w zależności od prawdopodobieństwa wystąpienia maintenance'u.

Parametry:

100 – liczba generowanych instancji,

1000 – liczba obiegów pętli algorytmu,

50 zadań na maszynach, każde składa się z task1 oraz task2,

20 – wielkość populacji,

metoda doboru par do rozmnażania: ranking,

n – procentowe prawdopodobieństwo wystąpienia maintenance'u po każdym zadaniu.

n	
1	44,37%
10	46,67%
25	49,42%
50	47,14%
75	44,56%
100	40,94%

Tabela 4: Procentowe poprawy wyników w stosunku do rozwiązań wygenerowanych losowo dla zadanego prawdopodobieństwa wystąpienia maintenance'u

e) Zmiana metody doboru par do rozmnażania. Parametry:

Mutacja5 + krzyżowanie

100 – liczba generowanych instancji,

1000 – liczba obiegów pętli algorytmu,

50 zadań na maszynach, każde składa się z task1 oraz task2,

20 – wielkość populacji,

25 – procentowe prawdopodobieństwo wystąpienia maintenance'u po każdym zadaniu.

Ranking	49,42%
First to Half	43,08%
Every Other	43,55%
Random	42,75%

Tabela 5: Procentowe poprawy wyników w stosunku do rozwiązań wygenerowanych losowo dla różnych sposobów doboru par do rozmnażania

9. Testy wystrojonej meta-heurystyki

Parametry:

Mutacja5 + krzyżowanie,

100 – liczba generowanych instancji,

1000 – liczba obiegów pętli algorytmu,

50 zadań na maszynach, każde składa się z task1 oraz task2,

100 – wielkość populacji,

25 – procentowe prawdopodobieństwo wystąpienia maintenance'u po każdym zadaniu,

metoda doboru par do rozmnażania: ranking.

Dla podanych wyżej parametrów poprawa w stosunku do losowych rozwiązań wynosi średnio 57,28%.

10. Wnioski

Strojenie meta-heurystyki jest bardzo istotne, dzięki niemu możemy znacznie poprawić. Po odpowiednim dobraniu parametrów otrzymaliśmy dwukrotnie mniejsze sumy czasów zakończenia wszystkich operacji w stosunku do rozwiązań losowych.

Ważne jest badanie realnego czasu w którym następuje zauważalny spadek sumy czasów zakończenia wszystkich operacji. W pewnym momencie skuteczność algorytmu maleje i poprawa jest niewielka, a czas trwania meta-heurystyki znacząco się wydłuża.