Open-shop scheduling problem

OSSP jest problemem NP-zupełnym, czyli takim, dla którego nie ma szybkich algorytmów rozwiązujących.

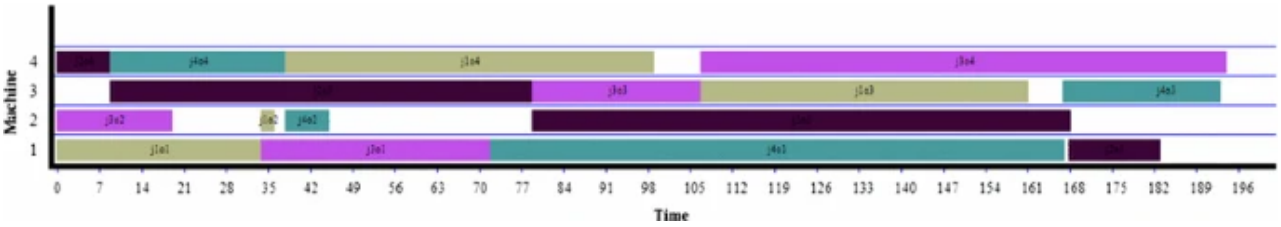
W tym problemie, n prac(jobs) jest wykonywane przez m machin(machines). Dla każdej nj pracy, każda mi machina ma przeznaczony czas dataTable[njmi] w jakim wykonuje tą operacje. W danym czasie, poszczególna maszyna, może przetwarzać tylko jedną operacje tej pracy. Operacje te mogą być wykonywane tylko, gdy dana praca nie jest wykonywana przez inną maszynę. Podczas operacji, nie dozwolone jest żadne przerywanie, bądź spowolnienie, operacje są niezależne. Celem problemu jest znalezienie najbardziej optymalnego planu z jak najkrótszym czasem maksymalnym potrzebnym na przeprowadzenie wszystkich prac na każdej maszynie.

Problem rozwiązujemy przy użyciu „Genetic Algorithm” i „Tabu Search”. Rozwiązania są zaimplementowane w języku Python(3.10).

Paczki używane:

- PyGad
- Random
- SolidPy
- Copy
- Time
- Numpy

Przykładowy wygląd rozwiązania:



Rozwiązania testujemy na 9 danych:

- 3 małe - 3x3
    - [12, 31, 30], [40, 34, 17], [47, 10, 13]
    - [25, 31, 14], [13, 14, 12], [37, 6, 11]
    - [6, 50, 49], [20, 39, 11], [35, 44, 5]
- 3 średnie - 5x5
    - [30, 14, 7, 72, 34], [33, 66, 24, 3, 67], [68, 34, 92, 6, 17], [59, 94, 41, 79, 27], [8, 21, 38, 67, 14]
    - [8, 11, 54, 24, 6], [46, 79, 60, 36, 85], [8, 57, 46, 32, 13], [18, 30, 95, 47, 69], [40, 40, 41, 36, 42]
    - [13, 6, 35, 50, 30], [33, 66, 40, 28, 34], [19, 4, 29, 20, 27], [42, 65, 87, 40, 57], [97, 28, 79, 76, 31]
- 3 duże - 7x7
    - [43, 15, 68, 48, 38, 14, 44], [54, 14, 68, 17, 64, 38, 19], [77, 14, 1, 1, 33, 18, 2], [71, 8, 36, 35, 68, 8, 63], [11, 90, 39, 24, 21, 22, 59], [42, 49, 64, 22, 67, 4, 66], [11, 56, 80, 84, 77, 49, 19]
    - [90, 95, 26, 35, 99, 81, 34], [60, 5, 59, 97, 50, 66, 44], [89, 50, 25, 93, 38, 23, 59], [47, 31, 61, 6, 15, 57, 87], [39, 84, 56, 53, 88, 49, 77], [60, 71, 81, 24, 5, 86, 84], [79, 56, 21, 44, 99, 6, 35]
    - [77, 1, 51, 1, 75, 42, 45], [85, 7, 84, 26, 53, 72, 45], [56, 43, 28, 23, 22, 10, 35], [62, 37, 63, 70, 23, 42, 13], [100, 69, 100, 76, 7, 6, 75], [46, 95, 25, 48, 40, 66, 44], [47, 27, 70, 84, 15, 36, 91]

Implementacja algorytmu genetycznego

Musimy zdefiniować parametry chromosomu. W naszym przypadku będą to kolejne liczby od 0 do ilości danych. Każda liczba informuje nas w jakiej kolejności praca na danej maszynie będzie wykonywana. Chromosomy to lista kolejności danych z inputu np. [5,4,8,2,0,1,3,7,6] gdzie każda liczba oznacza kolejność wykonywania operacji na danym miejscu. Chromosom można odczytywać w sposób: rozpoczynamy od machin, iterując po chromosomie, najpierw iterujemy po machinach, potem po pracach. Czyli w tym przypadku [m0j0,m0j1,m0j2,m1j0,m1j1,m1j2,m2j0,m2j1,m2j3]. Definiując input, musimy podać w jakiej postaci chcemy przekazać dane, tzn zdefiniować ile maszyn ma być i jaka ilość prac..Przykładowy chromosom:

8	4	0	3	5	2	7	6	1

Funkcja fitness działa w następujący sposób:

Dane z każdego rozwiązania są rozdzielane na poszczególne maszyny. Każda praca ma swój interwał. Jeżeli czas aktualny czas machiny i dodanej do niej operacji jest mniejszy od początku wykonania tej pracy na innej machinie, to praca ta jest wrzucana pomiędzy ostatnią prace aktualnej machiny, a aktualną pracę wykonywaną przez inną maszynę. Jeżeli czas się nie mieści w tych przedziałach(czyli praca nakładałaby się z inną machiną), praca dodawany jest koniec czasu aktualnej pracy na innej machinie(praca jest przesuwana na sam koniec, aż ta praca na innej machinie się skończy). Wynikiem(fitness) jest maksymalny czas w jakim machiny skończyły wszystkie prace.

Parametry Algorytmu:

Sol_per_pop – ilość chromosomów w populacji: 20

Num_genes – długość chromosomu(ile genów), zależne od problemu, u nas ilość danych: 9,25,49

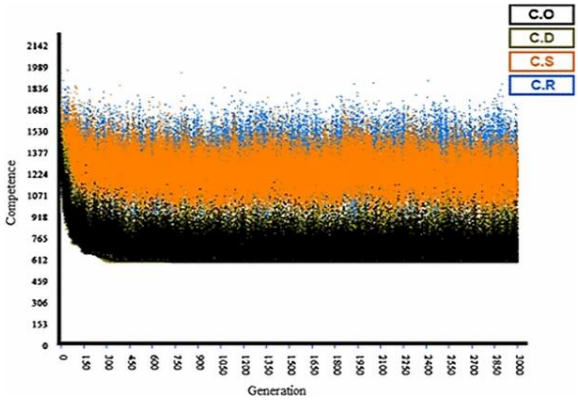
Num_parents_mating – ile spośród rodziców musimy rozmnożyć, 50% populacji: 10

Num_generations – liczba pokoleń: 20

Keep_parents – ilu rodziców zachowujemy dla kolejnej populacji(kilka procent zwykle): 2

Parent_selection_type – selekcja rodziców, wybieramy opcję rankingową: rank

Crossover_type – w ilu punktach przeprowadzane jest krzyżowanie. Na podstawie wykresu fig. 11 z artykułu: https://link.springer.com/article/10.1007/s00500-018-3177-y, wybieramy opcje single-point, gdyż będzie najbardziej efektywna. (single-point – C.O, one-point crossover)



Mutation_type – typ mutacji: random

Mutation_percent_genes – procent genów na jakich przeprowadzamy mutacje, ustawiając musimy zwrócić uwagę na liczbę genów: 10

Allow_duplicate_genes – powtarzanie się genów, gen reprezentując kolejność musi być inny od reszty, więc ustawiamy False

Wyniki:

Dane(input)	Mały 1	Mały 2	Mały 3	Sredni1	Sredni2	Sredni3	Duzy 1	Duzy 2	Duzy 3
Średni czas (sekundy)	0.179	0.107	0.104	0.368	0.364	0.381	0.989	0.974	0.974
Efektywność	100%	99%	98%	85%	59/100	49/100	18%	17%	18%

Algorytm działa bardzo dobrze dla małych inputów. Wraz ze wzrostem danych efektywność spada i czas potrzebny na wykonanie wzrasta. Im większy input, tym mniejsza efektywność. Prawdopodobnie, jest to spowodowane przez większą ilość mniejszych czasów, które często można pomiędzy prace wcisnąć. Dla tak zakresowo rozbieżnych danych przy większej ich ilości algorytm ma problem z dobraniem idealnych parametrów.

Implementacja Przeszukiwania Tabu

Powyższy problem można rozwiązać też za pomocą „tabu search”. polega na iteracyjnym przeszukiwaniu przestrzeni rozwiązań, wykorzystując sąsiedztwo pewnych elementów tej przestrzeni oraz zapamiętując przy tym przeszukiwaniu ostatnie ruchy, dopóki nie zostanie spełniony warunek końcowy(u nas koniec iteracji). Ruchy zapisywane są w postaci atrybutów przejścia (parametry opisujące jednoznacznie wykonany ruch) na liście (zwanej też czasami zbiorem) tabu. Obecność danego ruchu na liście tabu jest tymczasowa (zazwyczaj na określoną liczbę iteracji od ostatniego użycia) i oznacza, że danego ruchu nie można wykonać przez określoną liczbę iteracji – chyba, że ruch spełnia tzw. kryterium aspiracji (aspiration criterion). Lista tabu ma za zadanie wykluczyć (w praktyce jednak jedynie zmniejszyć) prawdopodobieństwo zapętleń przy przeszukiwaniu i zmusić algorytm do przeszukiwania nowych, niezbadanych regionów przestrzeni przeszukiwań.

Do rozwiązania tego problemu używamy zaimportowaną wcześniej biblioteke SolidPy. W tym algorytmie potrzebujemy zdefiniować funkcję _score (działa podobnie do fitness w GA) i _neighborhood. Naszym najlepszym rozwiązaniem będzie kolejność operacji na maszynach ułożonych w taki sam sposób jak chromosom w GA - [0, 2, 6, 4, 3, 8, 1, 5, 7]m = 3, j = 3 czyli poszczególne kolejności dla [m0j0,m0j1,m0j2,m1j0,m1j1,m1j2,m2j0,m2j1,m2j2]

Funkcja score jest podobnie zaimplementowana do fitness w GA. Trzeba zdefiniować dodatkowo funkcję neighborhood. Struktura sąsiedztwa polega na stworzeniu sąsiedztwa naszego wyniku. W naszym rozwiązaniu wybieramy losowe dane i zamieniamy miejscami(permutacja). Następnie algorytm z utworzonego sąsiedztwa wybiera najlepszego sąsiada(nawet jeśli spowoduje to pogorszenie wyniku), którego podmieniamy za poprzedniego. Algorytm zawiera w sobie liste tabu - liste zachowań zabronionych. przetrzymuje ona ostatnie ruchy, które nie mogą zostać wykonane do zwykle określonej liczby iteracji. Rozwiązanie to pozwala zmniejszyć przestrzeń poszukiwań przez co zmniejszamy czas obliczeń. Celem jest uniknięcie przejść cyklicznych i powrotów do lokalnych minimów.

Parametry „tabu search”

Tabu_size – długość listy tabu czyli pojemność tablicy która zawiera zakazane ruchy

Max_steps – ilość iteracji algorytmu

Max_score – kryterium stopu, u nas nie wiemy jaki ma spełniać więc jest None(zatrzymuje się po ostatniej iteracji)

Wyniki Tabu Search

Dane(input)	Mały 1	Mały 2	Mały 3	Sredni1	Sredni2	Sredni3	Duzy 1	Duzy 2	Duzy 3
Średni czas (sekundy)	0.992	0.957	0.942	1.002	1.025	1.013	2.327	2.331	2.356
Efektywność	61%	98%	70%	1%	3%	1%	1%	1%	1%

Podsumowanie

Algorytm genetyczny radzi sobie bardzo dobrze przy mniejszych ilościach danych. Ta efektywność spada w miare proporcjonalnie do ilości danych. Tabu Search radzi sobie dobrze przy mniejszych danych, lecz ta efektywność spada bardzo mocno przy zwiększaniu ich. Jest to zapewne spowodowane blokowaniem nawet dobrych ruchów, zwiększających fitness dla naszych danych. Zmiana parametrów wyszukiwania nie poprawia sytuacji. Na tej podstawie można stwierdzić, że "Przeszukiwanie Binarne" nie jest dobrym wyborem dla rozwiązania Open-shop scheduling problem. Należy pamiętać, że jest to problem optymalizacyjny, więc operacje na większym zakresie liczb [0-100] z większymi ilościami danych może nie mieć znaczącego wpływu na wynik (np fitness 451 a 449), lecz dla efektywności pod względem najlepszego wyniku odgrywa dużą rolę.

Bibliografia

https://link.springer.com/article/10.1007/s00500-018-3177

https://pygad.readthedocs.io/en/latest/README_pygad_ReadTheDocs.html

https://en.wikipedia.org/wiki/Open-shop_scheduling

https://github.com/100/Solid/blob/master/tests/test_tabu_search.py

http://www.cs.put.poznan.pl/mkomosinski/lectures/optimization/TS.pdf

https://ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/szuk_tabu.opr.pdf