15.04.2021

AMHE

Dokumentacja końcowa

Piotr Pakulski | *Uladzislau Lahoikin*

Opis problemu

Problem plecakowy (Knapsack problem) jest problemem optymalizacyjnym polegającym na maksymalizacji sumarycznej wartości przedmiotów w plecaku przy jednoczesnym uwzględnieniu wielkości plecaka (przedmioty muszą mieścić się w plecaku). Każdy przedmiot ma zdefiniowaną wagę i wartość. Dla losowo generowanych problemów dane są nieskorelowane, co znaczy, że nie można określić zależności pomiędzy wartością, a wagą przedmiotów. W tym przypadku algorytm szybciej się uczy ponieważ różnice pomiędzy wartościami, a wagami są większe i łatwiej jest znaleźć bardzo wartościowe i bardzo tanie przedmioty. Natomiast dla danych skorelowanych wartość przedmiotu jest zależna od jego wagi, jest mniejsza różnica w relacji wagi do wartości poszczególnych przedmiotów. Zazwyczaj w tym przypadku wartość stanowi funkcję wagi, a rozwiązania są cięższe do znalezienia.

Cel projektu

Celem projektu jest rozwiązanie problemu plecakowego dla danych skorelowanych i nieskorelowanych poprzez implementację algorytmów PBIL i GA, porównanie algorytmów oraz przeprowadzenie analizy statystycznej wyników.

Wykorzystane technologie

Język programowania: Python + biblioteki (matplotlib, pandas, json, sys, random, time, os, datetime, copyfile, operator, math, csv)

Zbiory danych

Wybrane problemy: http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/
Pełna lista wygenerowanych problemów (na potrzeby pracy [3]): http://hjemmesider.diku.dk/~pisinger/codes.html
W finalnym porównaniu wykorzystano wybrane problemy z pełnej listy problemów.

Algorytm PBIL

Algorytm PBIL (Population-Based Incremental Learning) jest optymalizacyjnym algorytmem genetycznym, w którym ewolucja następuje głównie dla genotypu całej populacji niż dla jednostki. Często określany jest prostszą wersją algorytmu ewolucyjnego, jednak w wielu przypadkach osiąga lepsze wyniki niż standardowy algorytm ewolucyjny. W poniższej ramce znajduje się pseudokod algorytmu PBIL opisany przez A Baluja[2].

Po inicjalizacji wektora prawdopodobieństwa występuje pętla generacji, w której na początku każdej iteracji generowane są próbki, które następnie są oceniane, żeby wybrać najlepszą, następnie aktualizowany jest wektor prawdopodobieństwa po czym dochodzi do jego mutacji.

```
P ⟵ initialize probability vector. (Each position = 0.5)

loop # GENERATIONS

        # Generate Samples
        i ⟵ loop #SAMPLES
                sample_i ⟵ generate sample vector according to probabilities in P
                evaluation_i ⟵ evaluate (sample_i)

        # Find Best Sample
        max ⟵ find vector corresponding to maximum evaluation

        # Update Probability Vector
        l ⟵ loop #LENGTH
                P_l ⟵ P_l * (1.0 - LR) + max_l * (LR)

        # Mutate Probability Vector
        l ⟵ loop #LENGTH
                if (random (0,1] < MUT_PROBABILITY)
                        P_l ⟵ P_l * (1.0 - MUT_SHIFT) + random (0.0 or 1.0) * (MUT_SHIFT)
```

USER DEFINED CONSTANTS:
GENERATIONS: number of iterations to allow learning.
SAMPLES: the population size, number of samples to produce per generation
LENGTH: length of encoded solution
MUT_PROBABILITY: probability of mutation occurring in each position
MUT_SHIFT: amount for mutation to affect the probability vector
LR: learning rate

VARIABLES:
P: probability vector
$sample_{i..SAMPLES}$: solution vectors
$evaluation_{i..SAMPLES}$: evaluations of the solution vectors
max: solution vector corresponding to maximum evaluation

Algorytm GA

GA to algorytm przeznaczony do wyszukiwania i globalnej optymalizacji, oparty na teorii ewolucji i genetyki Darwina. Przy stosowaniu GA zakłada się, że w pewnej populacji najlepsze osobniki mają większe szanse na przeżycie i generowanie coraz bardziej sprawnych osobników, co gwarantuje, że populacja ewoluuje i znajduje rozwiązanie problemu. W ramce poniżej znajduje się pseudokod algorytmu GA opisany przez Andre L. S. Pessoa[4].

```
1 - initialize the population
                current_generation := 1
                for i := 1 to N do
                        for j := 1 to L do
                                p[i][j] := random_number(0,1)
2 - evaluate the population
                for i := 1 to N do
                        evaluate ( p[i] )
3 - selection
                for i := 1 to N do
                        parents[i] := tournament ( p, tournament_size )
4 - crossover
                for i := 1 to N step 2 do
                        for j := 1 to L do
                                if ( j <= crossover_point )
                                        offspring[i][j] := parents[i][j]
                                        offspring[i+1][j] := parents[i+1][j]
                                else
                                        offspring[i][j] := parents[i+1][j]
                                        offspring[i+1][j] := parents[i][j]
5 - mutation
                for i := 1 to N do
                        if ( random_number < mutation_rate )
                                mutation(offspring[i])
6 - evaluate the offspring
                for i := 1 to N do
                        evaluate(offspring[i])
7 - new population
                p := offspring
                current_generation += 1
8 - termination condition
                if ( current_generation < max_generation )
                        return to step 3
9 - the solution is the individual with the best evaluation
```

Opis implementacji

Implementacja umożliwia na uruchomienie algorytmów PBIL i GA dla problemów plecakowych z ustawionymi w plikach konfiguracyjnych wartościami parametrów.

W przypadku obu algorytmów wszystkie ścieżki są względne i dla przykładowych problemów najprościej jest wywoływać komendy z lokalizacji plików .py

PBIL:

W celu uruchomienia algorytmu należy ustawić parametry algorytmu w pliku parameters.json, a następnie wywołać komendę: `python main.py <ścieżka do parameters.json>`
W pliku z parametrami można konfigurować takie parametry jak:

- verbose_details - czy pokazywać dodatkowe logi informujące o przebiegu algorytmu,
- problem_path - ścieżka do pliku z problemem,
- early_stopping_patience - czy i w ciągu ilu generacji sprawdzać wcześniejsze zatrzymanie (-1 oznacza nie sprawdzać),
- repeat_times - ile razy powtarzać wykonywanie algorytmu
- population_size - wielkość populacji,
- generations - liczba generacji,
- mutation_probability - prawdopodobieństwo mutacji,
- mutation_value - wielkość mutacji,
- learning_rate1 - szybkość uczenia się,
- learning_rate2 - negatywana szybkość uczenia się.

Przykładowy plik z parametrami został umieszczony w plikach projektu.

Po każdym przebiegu algorytmu tworzone są 3 pliki:
- pbil-<data>.txt - zawiera ustawienia algorytmu
- pbil-<data>.csv - zawiera przebieg algorytmu w każdej generacji (średnia wartość, najlepsza wartość i waga dla najlepszej wartości)
- final.csv - po każdym przebiegu dodawany jest jeden wpis z finalnym rezultatem algorytmu (nazwa pliku z logami przebiegu, długość problemu, wartość najlepszego plecaka, optymalna wartość dla problemu, waga najlepszego plecaka, maksymalna waga dla problemu, czas obliczania rozwiązania, wektor obrazujący najlepszy plecak)

Logi z przebiegu algorytmu do utworzenia tabeli porównawczej umieszczono w plikach projektu.

Aby wyświetlić wykresy lub podsumowanie danych należy użyć pliku misc.py, który został przygotowany w formacie PyCharm notebook wskazując ścieżkę do konkretnych logów.

GA:

W celu uruchomienia algorytmu trzeba w terminalu wpisać komendę:

```
python genetic.py <ścieżka do pliku z problemem> <0> <1> <2> <3> <4> <5> g
```

Powyższa komenda wymaga podania następujących parametrów:
- <0> - liczba osobników w populacji (rozmiar plecaka)
- <1> - rodzaj selekcji: rs - rank, ts - tournament, bs - Boltzmann
- <2> - rodzaj krzyżowania: 1c - 1-point, uc - uniform
- <3> - prawdopodobieństwo krzyżowania
- <4> - prawdopodobieństwo mutacji
- <5> - liczba generacji

Wartości parametrów mogą się różnić w zależności od rozwiązywanego problemu.
Cały przebieg rozwiązywania problemu plecakowego jest wypisywany do konsoli. Po rozwiązaniu problemu plecakowego użytkownik zostanie poinformowany o tym w konsoli.

Wyniki z logami działania algorytmu są zapisywane do plików o nazwie:
- log_avgFit_<data>.csv

- log_knapVal_<data>.csv

Pliki mają postać tablicy zawierającej numery iteracji i wartości funkcji fitness.

Następnie użytkownik może wyświetlić wykresy dla rozwiązanego problemu plecakowego za pomocą wygenerowanych logów, wpisując komendy:

```
python misc-genetic.py
python misc-genetic-boxplt.py <ścieżka do pliku log_knapVal>
```

Porównanie algorytmów

Zaimplementowane algorytmy PBIL oraz GA zostały porównane pod kątem otrzymanej średniej maksymalnych wartości funkcji przystosowania(fitness) dla 50 uruchomień poszczególnych algorytmów przy różnych wartościach zmiennych:

- N – długości plecaka: 50, 100, 200
- R – zakresu współczynnika wagowego: 1000, 10000,
- Danych skorelowanych i nieskorelowanych

Do porównania algorytmów wykorzystano testy automatyczne, w trakcie pojedynczego testu wybrany algorytm z ustawionymi parametrami był uruchamiany wielokrotnie według wartości zdefiniowanej w pliku konfiguracyjnym. Po każdym uruchomieniu algorytmu, wyniki były zapisywane w formie 3 plików (wspólny plik z finalnymi wynikami, plik z przebiegiem algorytmu i plik z parametrami algorytmu). Finalne wyniki po przetworzeniu były ręcznie wpisywane do finalnej tabeli. Ostatecznie w celu porównania uruchomiono algorytmy dla 12 problemów (3 wartości N * 2 wartości R * 2 rodzaje danych).

PBIL:

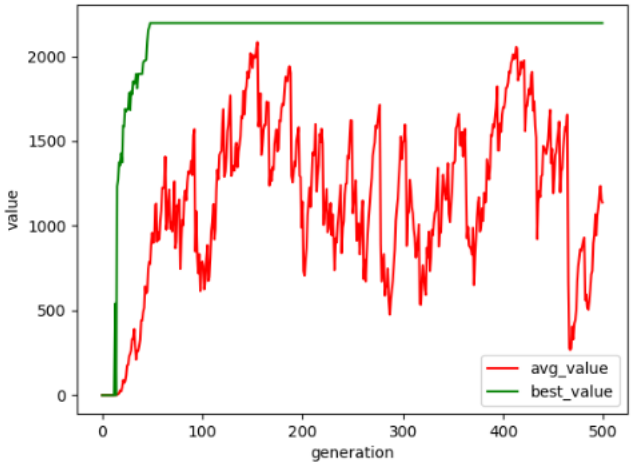
Porównanie wykonano dla tych samych ustawień algorytmu PBIL: wielkość populacji: 500 i liczba generacji: 500, brak wcześniejszego zatrzymania.

Problem						**PBIL**				
Nazwa problemu (zbiór danych)	N (długość plecaka)	R (zakres współczynnika wagowego)	Rodzaj danych (Skorelowane-S/Nieskorelowane-NS)	Maksymalna pojemność plecaka	Optimum	Odchylenie średniej najlepszej wartości plecaka od optimum	Średnia najlepsza wartość plecaka	Odchylenie najlepszej wartości plecaka od optimum	Najlepsza wartość plecaka	Średni czas (s)
p_knapPI_1_50_1000	50	1000	NS	995	8373	0.12%	8363.14	0.00%	8373	3.28
p_knapPI_1_50_10000	50	10000	NS	9984	68946	0.84%	68363.72	0.00%	68946	3.23
p_knapPI_3_50_1000	50	1000	S	994	1894	2.14%	1853.38	0.00%	1894	3.12
p_knapPI_3_50_10000	50	10000	S	9984	20862	2.65%	20310.18	0.00%	20862	3.27
p_knapPI_1_100	100	1000	NS	995	9147	2.87%	8884.92	0.00%	9147	5.69

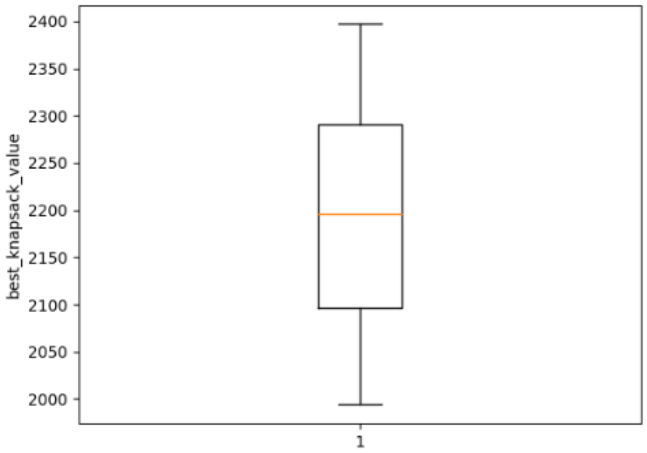
_1000										
p_knapPI_1_100_10000	100	10000	NS	9984	81021	3.80%	77943.50	0.00%	81021	5.77
p_knapPI_3_100_1000	100	1000	S	997	2397	8.72%	2188	0.00%	2397	5.77
p_knapPI_3_100_10000	100	10000	S	9984	23982	8.04%	22053.16	0.04%	23973	5.69
p_knapPI_1_200_1000	200	1000	NS	1008	11238	9.71%	10146.66	1.74%	11042	10.83
p_knapPI_1_200_10000	200	10000	NS	10295	108079	11.43%	95726.82	1.08%	106907	10.47
p_knapPI_3_200_1000	200	1000	S	997	2697	12.77%	2352.72	0.11%	2694	10.70
p_knapPI_3_200_10000	200	10000	S	10330	26330	14.81%	22430.72	3.85%	25315	10.81

Przykładowe wykresy dla problemu p_knapPI_3_100_1000:

Pojedyncze uruchomienie



Rozkład najlepszych wartości plecaka



GA:

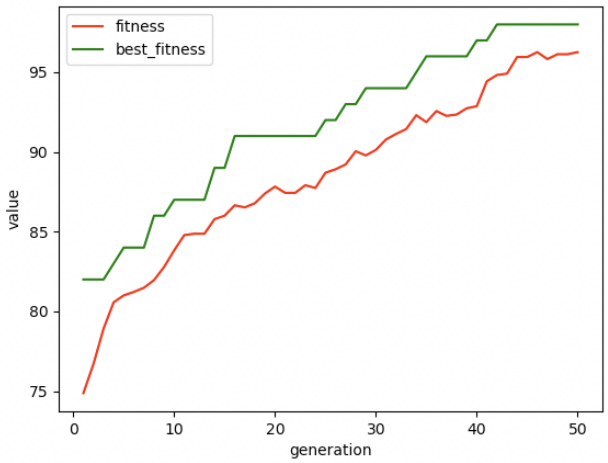
Porównanie wykonano dla tych samych ustawień algorytmu GA:

Problem						**GA**				
Nazwa problemu (zbiór danych)	N (długość plecaka)	R (zakres współczynnika wagowego)	Rodzaj danych (Skorelowane-S/Niesko relowane-NS)	Maksymalna pojemność plecaka	Optimum	Odchylenie średniej najlepszej wartości plecaka od optimum	Średnia najlepsza wartość plecaka	Odchylenie najlepszej wartości plecaka od optimum	Najlepsza wartość plecaka	Średni czas (s)
p_knapPI_1_50_1000	50	1000	NS	995	8373	11.42%	7417	0.0%	8373	1.94

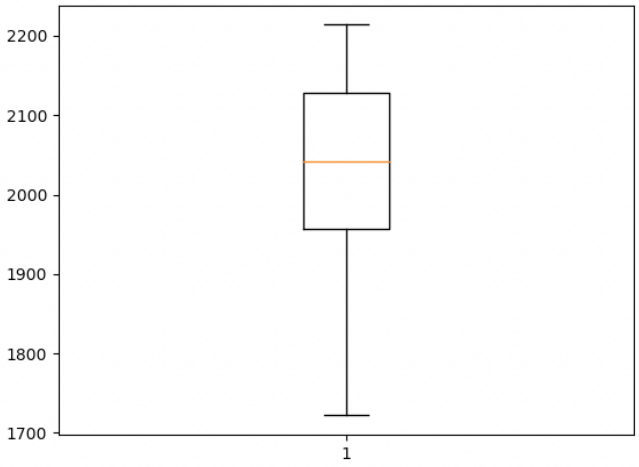
p_knapPI_1_50_10000	50	10000	NS	9984	68946	4.32%	65968	0.0%	68946	3.23
p_knapPI_3_50_1000	50	1000	S	994	1894	8.64%	1812	0.0%	1894	2.13
p_knapPI_3_50_10000	50	10000	S	9984	20862	5.64%	19686	0.0%	20862	3.38
p_knapPI_1_100_1000	100	1000	NS	995	9147	6.02%	8597	0.0%	9147	4.92
p_knapPI_1_100_10000	100	10000	NS	9984	81021	3.61%	78097	0.0%	81021	7.64
p_knapPI_3_100_1000	100	1000	S	997	2397	5.82%	2258	0.0%	2397	5.03
p_knapPI_3_100_10000	100	10000	S	9984	23982	3.7%	23090	0.0%	23982	7.95
p_knapPI_1_200_1000	200	1000	NS	1008	11238	6.96%	10456	0.21%	11214	9.73
p_knapPI_1_200_10000	200	10000	NS	10295	108079	5.28%	102373	1.21%	106772	14.5
p_knapPI_3_200_1000	200	1000	S	997	2697	7.28%	2501	1.25%	985	11.08
p_knapPI_3_200_10000	200	10000	S	10330	26330	6.56%	23743	3.5%	25409	19.21

Przykładowe wykresy dla problemu p_knapPI_3_100_1000:

Pojedyncze uruchomienie



Rozkład najlepszych wartości plecaka



Podsumowanie wyników porównania i wnioski

Po dokonaniu analizy powyższych tabel można wysunąć kilka wniosków:

Po pierwsze algorytm PBIL miał lepsze średnie wyniki w przypadku mniejszych problemów (N = 50) niż GA. Odwrotną tendencję widać w przypadku większych problemów, w przypadku których algorytm GA osiągał lepsze średnie wyniki,

W przypadku trudniejszych/większych problemów możliwe jest otrzymanie wartości optimum po modyfikacji parametrów algorytmu np. zwiększenia wielkości populacji czy liczby generacji. Zarówno algorytm PBIL jak i GA nie znalazły optymalnego rozwiązania dla ustalonych parametrów

W przypadku algorytmu PBIL średni czas wykonywania algorytmu zwiększa się wraz z wielkością problemu. Można zauważyć 3 grupy czasów, dla N równego 50, 100, 200. Nie widać wyraźnego związku pomiędzy rodzajem danych i wartością współczynnika wagowego R, a czasem wykonywania algorytmu.
Dla algorytmu GA nie widać wyraźnych grup ze względu na wielkość problemu, jednak można zauważyć, że algorytm pracował dłużej dla danych skorelowanych niż dla danych nieskorelowanych. Dodatkowo zwiększenie wartości współczynnika wagowego R powodowało wydłużenie czasu działania algorytmu.

Algorytm PBIl był szybszy niż algorytm GA, w przypadku większych problemów, jednak w przypadku mniejszych relacja była odwrotna. Jednak jeśliby sprawdzać średni czas po którym algorytm znajdował najlepsze dla danego przebiegu rozwiązanie to prawdopodobnie algorytm PBIL byłby szybszy w każdym przypadku (Jednak tego typu badanie zostało przeprowadzone tylko dla kilku przykładów więc jest to bardziej hipoteza).

Literatura

[1] Adaptive Algorithms for solving the Knapsack Problem, Sharif Samir Hamed, Marco A. Wiering, 2019
[2] Population-Based Incremental Learning:A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning, Shumeet Baluja, 1994
[3] Where are the hard knapsack problems?, David Pisinger, 2004
[4] Application of Simple and Compact Genetic Algorithms for Estimating Harmonic Components, By Andre L. S. Pessoa, 2015