Dokumentacja do projektu z ALHE

Rafal Babinski Roman Moskalenko

1 Treść zadania

SK.ALHE.4

Złodziej ukradł X gramów złota ze skarbca i wraca do domu pociągiem. Żeby uniknąć schwytania przez policję, musi zamienić złoto na banknoty, więc postanawia sprzedać złoto pasażerom pociągu. Zainteresowanych kupnem jest N pasażerów, każdy z nich zgadza się kupić a_i, $i \in (1, 2, ..., N)$ gramów złota za v_i, $i \in (1, 2, ..., N)$. Złodziej chce uciec przed policją, jednocześnie maksymalizując zysk. Zaimplementuj program bazujący na algorytmie ewolucyjnym, który wskaże pasażerów, którym powinien sprzedać złoto oraz sumę wartości banknotów, którą zarobi. Zastosowanie i porównanie z innym algorytmem będzie dodatkowym atutem przy ocenie projektu.

2 Projekt wstępny

2.1 Założenia ogólne

Ewidentnie mamy do czynienia z problemem plecakowym zero-jedynkowym. Zakładamy, że $a_i, v_i, X \in \mathbb{R}^+$.

W projekcie będziemy nawiązywać się do artykułu[1], w którym badano efektywność algorytmów genetycznych dla problemu plecakowego zero-jedynkowego. Dalej będziemy odnosić się do niego po prostu jako “artykuł”.

Preferowanym językiem programowania jest Python.

2.2 Planowane rozwiązanie

W ramach projektu planujemy porównać działanie trzech algorytmów:

- **Prosty algorytm genetyczny** – algorytm genetyczny, w którym odrzuca się rozwiązania niespełniające ograniczenia problemu.
- **Algorytm genetyczny zmodyfikowany** – algorytm genetyczny bazowany na artykule. Polega on na poradzeniu sobie z rozwiązaniami niespełniającymi ograniczenia. Do tego zaimplementujemy dwie metody, które najlepiej się sprawdziły wg. wspomnianego artykułu: *zastosowanie w funcji celu logarytmicznej funkcji kary* oraz *zachłanne naprawianie rozwiązań* (do wyboru odpowiednio do pojemności plecaka).
- **Branch-and-Bound** – algorytm, który powinien zapewnić nam rozwiązanie optymalne. Znając rozwiązanie optymalne będziemy w stanie dokładniej oszacować efektywność algorytmów genetycznych.

Oczekujemy, że zmodyfikowana wersja algorytmu genetycznego okaże się efektywniejsza niż wersja prosta.

3 Realizacja projektu

3.1 Generowane danych testowych

Na potrzeby projektu generowano instancje problemu plecakowego w sposób opisany w artykule.

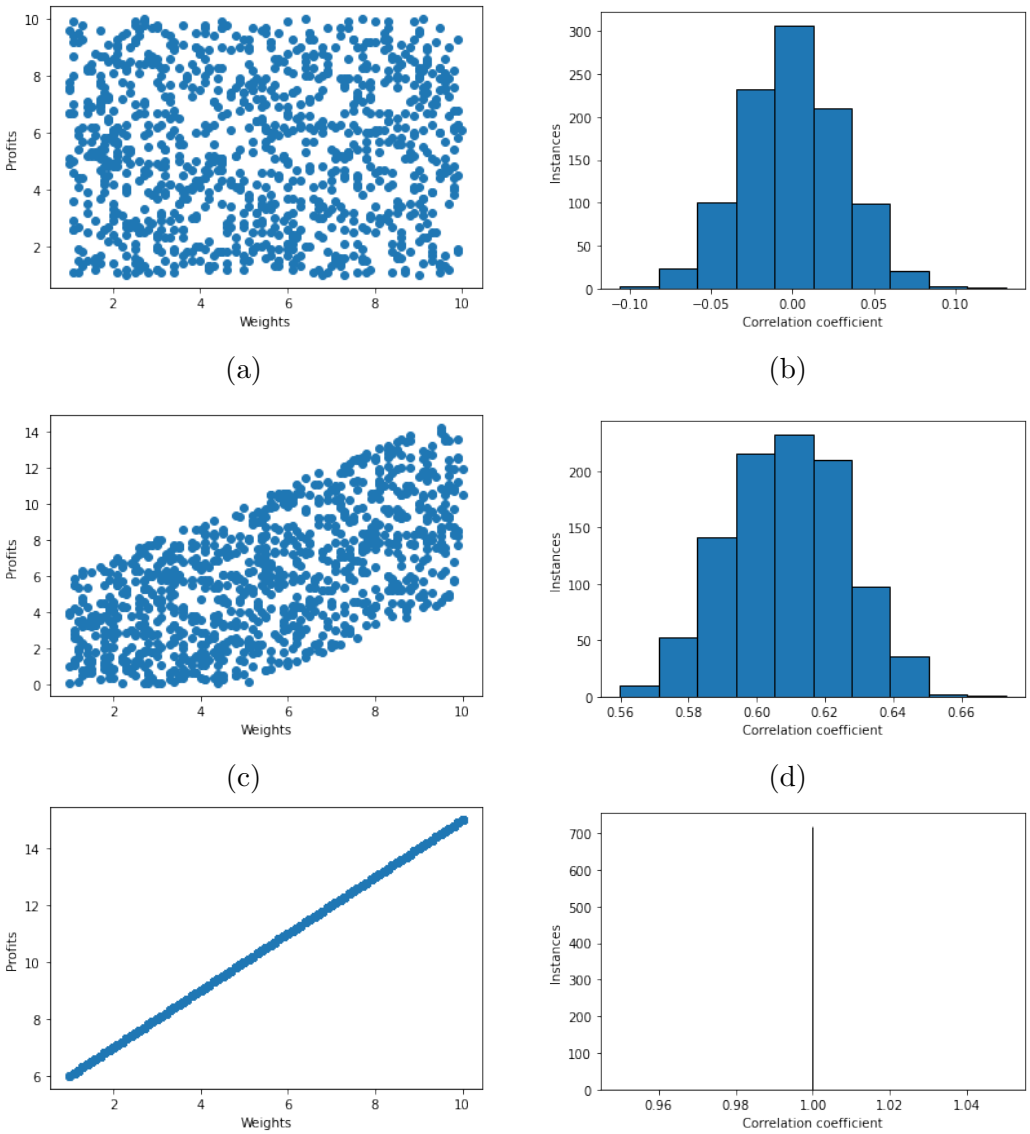
3.1.1 Korelacja

Rozróżnia się instancje, gdzie wagi i zyski są:

- **nieskorelowane** – współczynnik korelacji w przybliżeniu równy 0,
- **skorelowane słabo** – współczynnik korelacji w przedziale (0, 1),
- **skorelowane mocno** – współczynnik korelacji równy 1.

Większa korelacja, jak wspomniano w artykule, implikuje większą trudność problemu.

Duży wpływ na współczynnik korelacji mają parametry v – wartość maksymalna wag oraz r – odchylenie zysku od wagi. Nie będziemy szczególnie badać tej zależności. W dalszych rozdziałach wykorzystane dane zostały wygenerowane z parametrami $v = 10$, $r = 5$.



Rysunek 1: 1000 próbek danych (przy $v = 10, r = 5$) nieskorelowanych (a), skorelowanych słabo (c) i mocno skorelowanych (e). Współczynnik korelacji w 1000 wygenerowanych instancji bez korelacji (b), z korelacją słabą (d) i mocną (f).

Zwrócimy uwagę, że dane skorelowane "słabo" mają współczynnik korelacji w przybliżeniu równy 0.61 co możemy uznać za słabą korelację.

3.1.2 Pojemność plecaka

W artykule opisano 2 typy pojemności:

- **Ograniczona** – $2v$, gdzie v jest wartością maksymalną wag.
- **Rozszerzona** – równa połowie sumy wszystkich wag.

Nasz generator umożliwia również ustawienie pojemności wybranej arbitralnie przez użytkownika.

Dla wygody posługiwania dodaliśmy też listę indeksów do elementów posortowanej wg. stosunku zysk/waga tablicy. Nie wpływa to w żaden sposób na rozwój osobników w naszych algorytmach, w dodatek przyspiesza naprawianie zachłanne.

3.2 Branch-and-Bound

Metoda przeglądu drzewa przestrzeni rozwiązań z heurystycznym odcinaniem gałęzi, które nie mają rozwiązania optymalnego. Nasza implementacja wykorzystuje przegląd drzewa wgłąb.

3.2.1 Wykorzystanie zasobów czasowych i pamięciowych

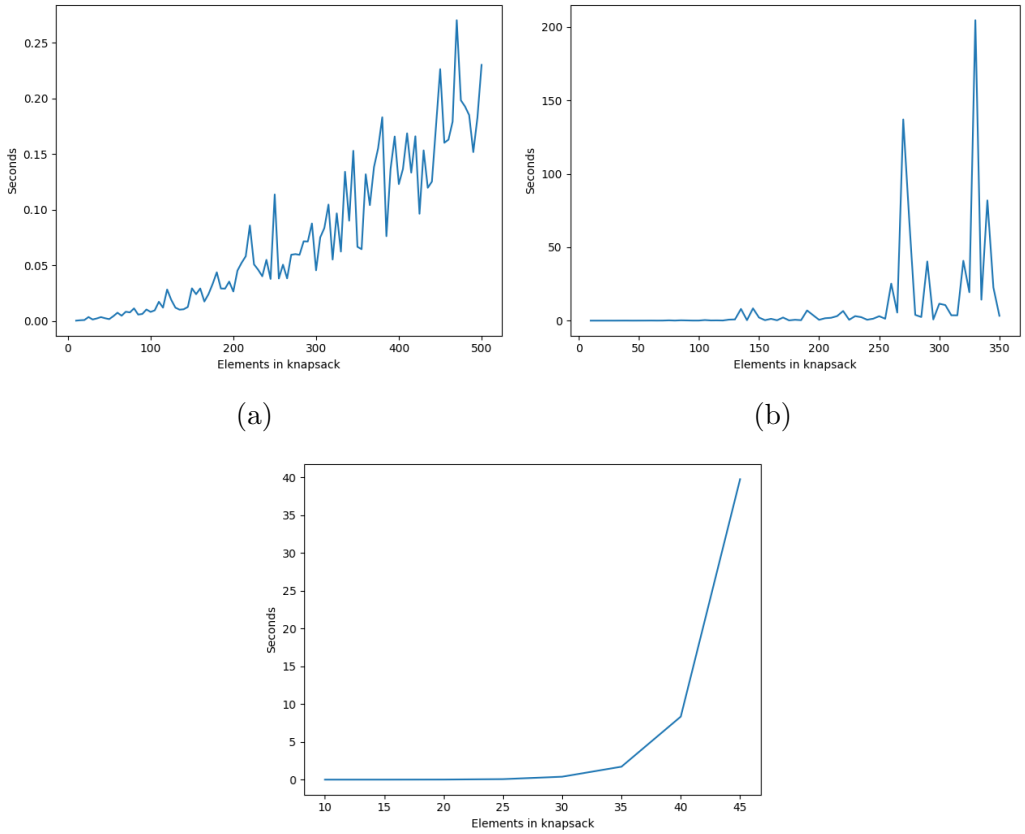
Przedstawimy kilka wykresów opisujących zależność czasu trwania algorytmu od liczby elementów w plecaku przy różnych korelacji i pojemności.

Rysunek 2 dobrze opisuje jak bardzo czas trwania algorytmu zależy od korelacji danych i typu pojemności plecaka. Dla każdego punktu na wykresie algorytm został odpalony 5 razy i został zachowany czas uśredniony.

Jak widać dla **słabej korelacji** i ograniczonej pojemności algorytm działa bardzo szybko (mniej niż 1 sekunda). Tak się dzieje dla tego, że ograniczona pojemność pozwala algorytmowi dość często odrzucać gałęzie drzewa.

Dla **silnej korelacji** czas trwania jest rzędu dziesiątek sekund, czasami sięga kilku minut.

Dla **powiększonej pojemności** plecaka wykres czasu trwania zaczyna przypominać funkcję wykładniczą. Także zauważyliśmy wykorzystanie przez algorytm zbyt dużej ilości pamięci (czasami system operacyjny zabijał program) dla tego postanowiliśmy nie kontynuować pomiaru czasu dla większych wartości.



Rysunek 2: Wykresy zależności czasu działania algorytmu od liczby elementów w plecaku przy różnych właściwościach zbioru: (a) słaba korelacja i ograniczona pojemność plecaka; (b) silna korelacja i ograniczona pojemność plecaka; (c) słaba korelacja i rozszerzona pojemność.

3.2.2 Wykorzystanie algorytmu w projekcie

Czas trwania i pochłanianie pamięci robią algorytm bardzo nieatrakcyjny. Spróbowaliśmy nawet zmodyfikować algorytm tak, aby nie zagłębiał się niżej określonego poziomu. To rzeczywiście przyspieszyło jego działanie, ale zwracane przez algorytm wartości stały się bardzo odległe od wartości optymalnej, co robi modyfikację niezbyt efektywną. Z tego powodu postanowiliśmy zrezygnować z użycia tego algorytmu w przypadku rozszerzonej pojemności plecaka.

3.3 Algorytm genetyczny

Wyróżniamy następujące komponenty zaimplementowanych algorytmów genetycznych (różnice między algorytmami opiszemy w odpowiednich podrozdziałach każdego z komponentów):

1. Stworzenie populacji początkowej
2. Selekcja osobników do krzyżowania
3. Krzyżowanie
4. Mutacja
5. Selekcja osobników do utworzenia następnej populacji

Każdy z tych komponentów postaramy się opisać.

3.3.1 Populacja początkowa

Inicjalizacja populacji początkowej odbywa się w ten sposób, że dla każdego nowego osobnika losowana jest lista zer i jedynek.

Na koniec dla każdego osobnika populacji wyliczana jest jakość (fitness). W przypadku algorytmu z funkcją kary, osobniki nie są naprawiane, natomiast fitness jakość uwzględnia karę. Dla pozostałych algorytmów osobniki przy przekroczeniu pojemności plecaka zostają naprawione naprawianiem losowym.

Po tych czynnościach populacja jest gotowa do wejścia do pętli głównej algorytmu.

3.3.2 Selekcja do krzyżowania

Zgodnie z artykułem zaimplentowaliśmy **selekcję turniejową**. W każdym turnieju uczęstniczy **2 osobników**. Ponieważ przy utworzeniu nowej populacji osobniki zostają posortowane wg. jakości, wystarczy porównać ich indeksy aby wyznaczyć zwycięzcę turnieju.

3.3.3 Krzyżowanie

Zaimplementowaliśmy **krzyżowanie jednopunktowe**, używane w artykule.

W wyniku tej operacji powstają dwa nowe osobniki z przeliczonymi wartościami jakości.

Współczynnik krzyżowania odpowiada za to jak często stosowane jest krzyżowanie: np. dla współczynnika równego 1 krzyżujemy zawsze, dla 0.5 – w przybliżeniu w połowie przypadków, dla 0 – nigdy nie krzyżujemy.

3.3.4 Mutacja

Odwrócenie wybranego bitu.

Każdy bit dowolnego osobnika może zostać zmutowany, niezależnie od tego, czy dany osobnik został wybrany do krzyżowania, czy nie.

Współczynnik mutacji odpowiada za to, jak często mutujemy bity.

3.3.5 Selekcja do następnej populacji

Wszystkie osobniki zostają posortowane malejąco wg. posiadanej jakości. W przypadku algorytmu bez modyfikacji osobniki przekraczające pojemność plecaka zostają zastąpione wylosowanymi na nowo, które już spełniają ten warunek.

3.4 Porównanie algorytmów

W tym rozdziale porównamy działanie algorytmów.

Do porównania: algorytm genetyczny zwykły bez modyfikacji (*vanilla*), algorytm genetyczny z logarytmiczną funkcją kary (*penalty*), algorytm genetyczny z zachłannym naprawianiem osobników (*repair*).

Wydajność algorytmu oceniamy na podstawie najlepszej uzyskanej wartości jakości w ciągu 500 generacji. Współczynniki krzyżowania i mutacji są stałe i wynoszą 0.65 i 0.05 odpowiednio. Różnica między naszym eksperymentem a przedstawionym w artykule jest taka, że dla rozszerzonej pojemności plecaka nie jesteśmy w stanie sprawdzić czy znalezione rozwiązania są optymalne. Także dla algorytmu z funkcją kary po zakończeniu algorytmu rozwiązania często przekraczają pojemność plecaka, dla tego jednorazowo naprawiamy je zachłannie. W tabeli poniżej przedstawiamy uzyskane rezultaty (dokładniejszy przebieg eksperymentu można znaleźć w logach).

Correl.	No. of items	Cap. type	Algorithm		
			vanilla	repair	penalty
none	100	c1	45.3 (68.5)	**68.5 (68.5)**	67.9 (68.5)
		c2	371.4 (-)	411.0 (-)	**414.0** (-)
	250	c1	46.3 (120.6)	**120.6 (120.6)**	116.9 (120.6)
		c2	828.1 (-)	907.6 (-)	**913.1** (-)
	500	c1	36.6 (117.5)	**117.5 (117.5)**	112.2 (117.5)
		c2	1548.1 (-)	1691.2 (-)	**1715.6** (-)
weak	100	c1	35.6 (60.1)	**60.1 (60.1)**	58.9 (60.1)
		c2	359.0 (-)	398.8 (-)	**401.3** (-)
	250	c1	34.0 (69.6)	**69.6 (69.6)**	66.4 (69.6)
		c2	835.8 (-)	**951.6** (-)	941.7 (-)
	500	c1	33.2 (85.2)	**85.2 (85.2)**	81.8 (85.2)
		c2	1676.2 (-)	1848.2 (-)	**1863.2** (-)
strong	100	c1	59.7 (85.0)	**85.0 (85.0)**	84.9 (85.0)
		c2	566.2 (-)	**587.7** (-)	584.0 (-)
	250	c1	49.4 (100.0)	**100.0 (100.0)**	95.0 (100.0)
		c2	1371.4 (-)	**1433.9** (-)	1415.1 (-)
	500	c1	54.6 (105.0)	**105.0 (105.0)**	**105.0 (105.0)**
		c2	2705.8 (-)	**2797.8** (-)	2770.4 (-)

Tabela 1: Efektywność algorytmów dla różnych danych wejściowych

W nawiasach jest podane rozwiązanie optymalne lub "-" gdy nie znamy go. Wartości pogrubione świadczą o tym, że albo została osiągnięta wartość optymalna, albo dana wartość jest największa spośród innych.

Wnioski

Dla danych o ograniczonej pojemności plecaka algorytm z naprawianiem ma wyraźną przewagę nad pozostałymi i zawsze zwraca rozwiązanie optymalne.

Dla danych o rozszerzonej pojemności plecaka niestety nie znaliśmy rozwiązania optymalnego. Rezultaty działania różnią się od przedstawionych w artykule: algorytm z naprawianiem pokazuje lepsze wyniki, ale nie dla wszystkich danych wejściowych.

Widać, że standardowy algorytm genetyczny bez modyfikacji pokazuje gorsze wyniki, niż jego modyfikacje. Nasze oczekiwania odnośnie tego się sprawdziły. Możemy uznać, że cel projektu został osiągnięty.

3.5 Opis funkcjonalny

Projekt składa się z kilku plików .py, które zawierają najistotniejsze komponenty projektu:

- **BNBAlgorithm.py** - implementacja algorytmu Branch-and-Bound
- **Chromosome.py** - plik zawierający klasę *Chromosome*, oraz dotyczące jej metody. W tym pliku znajdują się implementacje funkcji do naprawiania osobników, mutacji, krzyżowania oraz wyliczenia jakości z uwzględnieniem kary.
- **DrawUnit.py** - plik zawierający metody do wizualizacji populacji osobników. Zrezygnowano z korzystania z nich, ponieważ dość trudno interpretować powstające obrazki.
- **Experiments.py** - w pliku algorytmy zostają uruchomione dla różnych zestawów danych wejściowych. Wszystkie wyniki są zapisywane w folderze *logs/*
- **Generator.py** - generowanie danych syntetycznych, które przekazywane są badanym algorytmom. Można ustawiać korelacje generowanych danych, zakres danych, pojemność bądź typ pojemności plecaka, liczbę elementów w plecaku oraz ziarno losowości. Bezpośrednio uruchamiając ten plik można zapisać wygenerowane dane do pliku.
- **GeneticAlgorithm.py** - metody algorytmu genetycznego, w tym inicjalizacja populacji, selekcja, naprawianie populacji, stworzenie następnej populacji itp.
- **main.py** - przeznaczony do uruchamiania algorytmów. Podając argumenty użytkownik może wybrany algorytm oraz plik z danymi wejściowymi. O tym jak korzystać z programu napisano w rozdziale *Uruchamianie*.

W folderze **tests/** znajdują się testy jednostkowe, dla poszczególnych metod programu.

Do folderu **logs/** zapisywany jest wyniki eksperymentu opisanego w pliku *Experiments.py*. Każdy plik logu zawiera przebieg ewolucji populacji (jakość najlepszą, oraz jakość uśrednioną), a także parametry wywołania instancji algorytmu. Nazwa logu opisuje parametry wywołania: liczba elementów w plecaku, typ pojemności, korelacja, wybrany algorytm.

W folderze **img/** przechowywane są obrazki wykorzystane w dokumentacji.

3.6 Uruchamianie

Program main.py pozwala na rozwiązanie problemu algorytmami:

- Branch and Bound
- Genetyczny

Program **main.py** może wygenerować dane testowe sam, lub wczytać je z pliku:

```
python main -i plik.json bnb -d 10
```

Wygenerować dane można programem **Generator.py**.

```
python Generator.py -o plik.json
```

Aby uruchomić program z algorytmem Branch and Bound należy:

```
python main.py bnb -d 10
```

Argument **-d** pozwala na wybranie głebokości przeglądania drzewa w algorytmie BnB.

Aby uruchomić program z algorytmem Genetycznym należy:

```
python main.py genetic -m method
```

Gdzie **method** należy wybrać opcję **vanilla**, **repair** lub **penalty**.

Więcej informacji o argumentach do programu można znaleźć uruchamiając:

```
python main.py -h            Argumenty generatora
python main.py bnb -h        Argumenty algorytmy Branch and Bound
python main.py genetic -h    Argumenty algorytmu Genetycznego
```

Referencje

[1] Zbigniew Michalewicz, Jarosław Arabas. "Genetic Algorithms for the 0/1 Knapsack Problem." 1994.