Podstawy sztucznej inteligencji - sprawozdanie

Dawid Kania

February 2025

1 Wstęp

Kod źródłowy znajduje się pod linkiem: `https://github.com/Kaniek99/AIbasics`
Wszystkie problemy zostały rozwiązane zgodnie z algorytmem omawianym na zajęciach – tzn. generujemy jednego rodzica a następnie przez mutację tworzymy dziecko. Zastosowanie dla poszczególnych problemów zostanie omówione szczegółowo w rozdziałach poświęconych danym problemom. Rozwiązanie każdego problemu znajduje się w oddzielnym pliku, część wspólna znajduje się w pliku genotype.go i zawiera interfejs *Genotype* z metodami $Mutate$, $Swap$, $Crossover$.

2 Problem plecakowy

Problem plecakowy to problem optymalizacyjny, w którym maksymalizujemy wartość przedmiotów spakowanych do plecaka. Przedmioty mają swoją wagę/objętość a liczba spakowanych rzeczy jest ograniczona przez pojemność plecaka.

2.1 Założenia

Niech I będzie zbiorem przedmiotów, wówczas i_i jest i-tym przedmiotem z tego zbioru. Przez w_i oznaczamy wagę i-tego przedmiotu a przez v_i jego wartość. Rozwiązaniem jest ciag binarny s o liczebności równej liczebności zbioru przedmiotów. Maksymalną dopuszczalną wagę plecaka oznaczamy przez w_{max} natomiast wyrażenie $i_i \in P$ oznacza, że i-ty przedmiot należy do ciągu binarnego plecaka P. Wówczas dane wejściowe to:

- $|I| = |s| = 100$,
- $v_i \in [10, 90] \ \wedge \ v_i \in \mathbb{N}$,
- $w_i \in [10, 90] \ \wedge \ w_i \in \mathbb{N}$,
- $w_{max} = 2500$,
- $s_i = \begin{cases} 1, \ i_i \in P, \\ 0, \ i_i \notin P. \end{cases}$

2.2 Rozwiązanie problemu

Niech s będzie ciągiem binarnym będącym rozwiązaniem problemu, wówczas współczynnik dopasowania dla s wyznaczamy według następującej formuły:

$$f(s) = \begin{cases} \sum_{i \in P} v_i, \ gdy \ \sum_{i \in P} w_i < w_{max}, \\ w_{max} - \sum_{i \in P} w_i, \ w \ przeciwnym \ przypadku. \end{cases}$$

Algorytm genetyczny służący do rozwiązania problemu plecakowego
Warunkiem kończącym działanie algorytmu jest nieznalezienie lepszego rozwiązania przez 100 iteracji.

Krok 1. Wygenerowanie zbioru przedmiotów I
Krok 2. Wygenerowanie rodzica s
Krok 3. Obliczamy $f(s)$.
Krok 4. Sprawdzamy czy licznik iteracji zakończonych niepowodzeniem jest mniejszy niż 100. Jeśli nie to algorytm kończy działanie.
Krok 5. Dziecko c powstaje przez mutację rodzica – losowo wybrany gen zmienia wartość na przeciwną.
Krok 6. Obliczamy $f(c)$.
Krok 7. Rozważamy teraz dwa przypadki:

- $f(s) < f(c)$ Zatem lepszym rozwiązaniem jest dziecko – staje się ono teraz rodzicem, $s = c, f(s) = f(c)$. Licznik iteracji zakończonych niepowodzeniem ustawiamy na 0. Wracamy do kroku 4.

- $f(s) \geqslant f(c)$
  Zwiększamy licznik iteracji zakończonych niepowodzeniem. Wracamy do kroku 4.

Na wyjściu otrzymujemy rozwiązanie suboptymalne.
Implementacja rozwiązania powyższego problemu znajduje się w pliku knapsack.go

Przykładowe rozwiązanie

```
Items: {weight: 46, value: 64} {weight: 23, value: 43} {weight: 23, value: 70} {weight: 54, value: 44} {weight: 10, v
alue: 43} {weight: 60, value: 18} {weight: 29, value: 25} {weight: 29, value: 64} {weight: 44, value: 87} {weight: 73
, value: 73} {weight: 48, value: 49} {weight: 57, value: 16} {weight: 40, value: 25} {weight: 89, value: 31} {weight:
 12, value: 13} {weight: 35, value: 80} {weight: 66, value: 28} {weight: 74, value: 86} {weight: 47, value: 48} {weig
ht: 37, value: 56} {weight: 85, value: 34} {weight: 75, value: 44} {weight: 11, value: 79} {weight: 36, value: 50} {w
eight: 33, value: 27} {weight: 56, value: 85} {weight: 74, value: 50} {weight: 31, value: 58} {weight: 65, value: 34}
 {weight: 12, value: 13} {weight: 82, value: 39} {weight: 49, value: 29} {weight: 16, value: 63} {weight: 51, value:
48} {weight: 52, value: 81} {weight: 60, value: 51} {weight: 25, value: 71} {weight: 65, value: 38} {weight: 64, valu
e: 56} {weight: 40, value: 61} {weight: 43, value: 25} {weight: 87, value: 33} {weight: 54, value: 79} {weight: 44, v
alue: 29} {weight: 73, value: 74} {weight: 66, value: 15} {weight: 38, value: 17} {weight: 75, value: 19} {weight: 14
, value: 36} {weight: 33, value: 23} {weight: 24, value: 89} {weight: 45, value: 12} {weight: 48, value: 61} {weight:
 19, value: 13} {weight: 74, value: 53} {weight: 74, value: 57} {weight: 63, value: 28} {weight: 53, value: 83} {weig
ht: 39, value: 14} {weight: 79, value: 66} {weight: 66, value: 81} {weight: 84, value: 43} {weight: 61, value: 50} {w
eight: 21, value: 61} {weight: 18, value: 54} {weight: 56, value: 57} {weight: 27, value: 30} {weight: 75, value: 37}
 {weight: 86, value: 58} {weight: 71, value: 30} {weight: 51, value: 51} {weight: 33, value: 71} {weight: 35, value:
21} {weight: 62, value: 42} {weight: 89, value: 67} {weight: 58, value: 56} {weight: 13, value: 87} {weight: 25, valu
e: 41} {weight: 46, value: 49} {weight: 16, value: 17} {weight: 72, value: 37} {weight: 32, value: 89} {weight: 17, v
alue: 34} {weight: 44, value: 75} {weight: 44, value: 77} {weight: 87, value: 61} {weight: 13, value: 59} {weight: 31
, value: 50} {weight: 79, value: 54} {weight: 15, value: 13} {weight: 26, value: 49} {weight: 51, value: 10} {weight:
 51, value: 43} {weight: 78, value: 75} {weight: 14, value: 17} {weight: 27, value: 50} {weight: 60, value: 52} {weig
ht: 19, value: 31} {weight: 65, value: 89} {weight: 35, value: 75}
Value of all items: 4776, Weight of all items: 4843
Solution: [0 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 0 0 0 1 0 0 1 1 1 0 1 1 0 1 0 0 1 1 1
1 1 1 0 1 1 0 0 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 0 1 0 1 1 0]
Solution value: 2599
```

3 Problem alokacji zadań

Problem alokacji zadań to problem optymalizacyjny, w którym należy przydzielić wszystkie zadania na dostępne rdzenie procesora w taki sposób aby łączny czas ich wykonania był jak najkrótszy. Każdy z rdzeni ma pewien mnożnik, z którym wykonuje zadania.

3.1 Założenia

Niech T będzie zbiorem zadań, wówczas $t : T \to \mathbb{Z}_+$ jest ciągiem, który $i-temu$ zadaniu przypisuje czas t_i potrzebny do jego wykonania. Zbiór C jest zbiorem ID rdzeni procesora gdzie c_i to ID i-tego rdzenia a C_i to zbiór zadań wykonywanych na i-tym rdzeniu. Rozwiązaniem jest ciag $s : T \to C$, o liczebności równej liczebności zbioru zadań, którego elementami są liczby całkowite. Ciąg z mnożnikami rdzeni oznaczamy przez m, jego liczebność jest równa zbiorowi rdzeni a i-ty element ciągu to mnożnik i-tego procesora. Wówczas dane wejściowe to:

- $|T| = |s| = 100$
- $|C| = |m| = 4$
- $i \in \{0,\ 1,\ 2,\ ...\ ,\ 99\}$
- $c_i \in \{0,\ 1,\ 2,\ 3\}$
- $s_i \in C$
- $m = (1,\ 1.25,\ 1.5,\ 1.75)$
- $|s| = |T| = 100$

3.2 Rozwiązanie problemu

Niech s będzie rozwiązaniem problemu, wówczas współczynnik dopasowania dla s wyznaczamy według następującej formuły:

$$f(s) = max(\sum_{i \in C_0} t_i m_0, \sum_{i \in C_1} t_i m_1,\ ...\ , \sum_{i \in C_3} t_i m_3)$$

Algorytm genetyczny służący do rozwiązania problemu alokacji zadań
Warunkiem kończącym działanie algorytmu jest nieznalezienie lepszego rozwiązania przez 100 iteracji.
Krok 1. Wygenerowanie zbioru zadań T oraz funkcji $t : T \to \mathbb{Z}_+$
Krok 2. Wygenerowanie rodzica s
Krok 3. Obliczamy $f(s)$.
Krok 4. Sprawdzamy czy licznik iteracji zakończonych niepowodzeniem jest mniejszy niż 100. Jeśli nie to algorytm kończy działanie.
Krok 5. Dziecko d powstaje przez mutację rodzica – losowo wybrany gen zmienia wartość – odpowiada to przypisaniu zadania do innego rdzenia.
Krok 6. Obliczamy $f(d)$.
Krok 7. Rozważamy teraz dwa przypadki:

- $f(s) < f(d)$ Zwiększamy licznik iteracji zakończonych niepowodzeniem. Wracamy do kroku 4.
- $f(s) \geqslant f(d)$
  Zatem lepszym rozwiązaniem jest dziecko – staje się ono teraz rodzicem, $s = d, f(s) = f(d)$. Licznik iteracji zakończonych niepowodzeniem ustawiamy na 0. Wracamy do kroku 4.

Na wyjściu otrzymujemy rozwiązanie suboptymalne.
Implementacja rozwiązania powyższego problemu znajduje się w pliku allocation.go
Przykładowe rozwiązanie

```
Core multipliers: [1 1.25 1.5 1.75]
Tasks time: [88 63 18 30 62 84 30 83 37 80 85 47 89 82 76 59 84 35 78 67 76 79 19 19 11 89 50 79 72 18 65 88 31 57
 69 22 46 40 10 32 86 27 87 34 29 20 13 63 47 89 47 46 50 60 16 21 48 26 76 69 48 56 21 45 82 54 82 82 28 59 36 26
 46 31 46 19 24 44 87 72 12 86 28 68 19 31 89 19 74 35 90 39 41 81 10 26 30 90]
Solution: [0 0 0 3 2 1 3 2 0 3 2 1 0 1 1 1 2 2 1 1 3 0 2 0 0 3 1 1 0 2 0 2 1 0 0 1 1 0 1 0 3 2 0 0 2 3 3 2 3 0 0 2
0 0 2 1 2 2 2 1 2 0 0 1 0 1 3 3 2 1 1 0 3 1 1 1 1 2 0 3 0 3 2 0 3 2 3 2 0 1 3 0 1 0 3 1 0 1 0]
Time required 1703.75
```

4 Problem komiwojażera

Problem komiwojażera to problem optymalizacyjny, w którym należy wyznaczyć jak najkrótszą trasę prowadzącą przez wszystkie miasta.

4.1 Założenia

Niech C będzie zbiorem identyfikatorów miast, wówczas $d : C \times C \to \mathbb{R}_+$ jest funkcją, która parze miast (c_i, c_j) przypisuje odległość d_{ij} pomiędzy nimi. Rozwiązaniem problemu nazywamy dowolną permutację zbioru C i oznaczamy ją przez s. Wówczas dane wejściowe to:

- $C = \{0,\ 1,\ 2,\ \dots\ ,\ 99\}$
- $s : C \to C$
- $\begin{cases} d(i,j) = d(j,i), \\ d(i,j) > 0,\ i \neq j, \\ d(i,j) = 0,\ i = j. \end{cases}$

4.2 Rozwiązanie problemu

Niech s będzie rozwiązaniem problemu, wówczas współczynnik dopasowania dla s wyznaczamy według następującej formuły:

$$f(s) = \sum_{i=1}^{|C|-1} d(s_{i-1}, s_i)$$

Algorytm genetyczny służący do rozwiązania problemu alokacji zadań
Warunkiem kończącym działanie algorytmu jest nieznalezienie lepszego rozwiązania przez 100 iteracji.
Krok 1. Wygenerowanie odległości pomiędzy miastami – $d : C \times C \to \mathbb{R}_+$
Krok 2. Wygenerowanie rodzica s
Krok 3. Obliczamy $f(s)$.
Krok 4. Sprawdzamy czy licznik iteracji zakończonych niepowodzeniem jest mniejszy niż 100. Jeśli nie to algorytm kończy działanie.
Krok 5. Dziecko k powstaje przez zamienienie miejscami dwóch losowo wybranych genów u rodzica.
Krok 6. Obliczamy $f(k)$.
Krok 7. Rozważamy teraz dwa przypadki:

- $f(s) < f(k)$ Zwiększamy licznik iteracji zakończonych niepowodzeniem. Wracamy do kroku 4.
- $f(s) \geqslant f(k)$
  Zatem lepszym rozwiązaniem jest dziecko – staje się ono teraz rodzicem, $s = k, f(s) = f(k)$. Licznik iteracji zakończonych niepowodzeniem ustawiamy na 0. Wracamy do kroku 4.

Na wyjściu otrzymujemy rozwiązanie suboptymalne.
Implementacja rozwiązania powyższego problemu znajduje się w pliku salesman.go

Przykładowe rozwiązanie

```
Solution: [44 18 35 91 9 85 11 53 5 61 25 32 56 16 43 34 48 71 99 7 84 63 88 27 82 52 42 68 12 26 28 92 39 29 45
 57 1 47 83 75 77 90 24 3 66 86 13 49 37 76 6 67 50 23 54 2 21 0 60 51 80 33 15 62 97 78 30 79 95 72 94 40 19 73
 96 55 46 17 8 64 93 81 20 4 69 14 87 59 10 74 65 36 41 89 58 22 31 70 98 38]
Total distance: 7337
```