Adrian Frydmański 209865
Dawid Gracek 209929

Data oddania: **26 I 2016r.**
Prowadzący: **dr inż. Jarosław Mierzwa**

Projektowanie efektywnych algorytmów – laboratorium

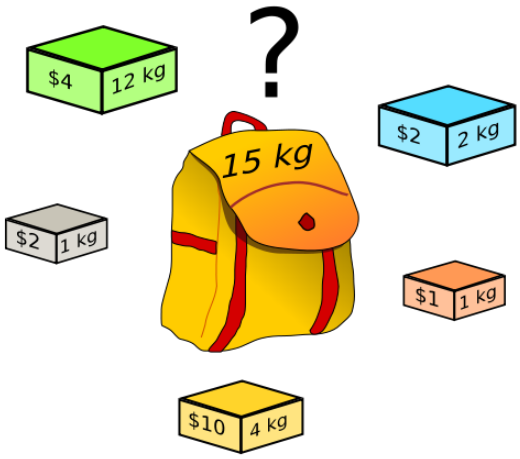
Temat: „Rozwiązywanie problemu plecakowego schematem aproksymacyjnym”

Zadanie

Należy zaimplementować schemat aproksymacyjny dla optymalizacyjnego problemu plecakowego oraz dokonać testów polegających na pomiarze czasu działania algorytmu w zależności od wielkości instancji.

Informacje wstępne

Dyskretny problem plecakowy (ang. discrete knapsack problem) jest jednym z najczęściej poruszanych problemów optymalizacyjnych, którego nazwa pochodzi od maksymalizacyjnego problemu wyboru przedmiotów, tak by ich sumaryczna wartość była jak największa i jednocześnie mieściły się w plecaku. Przy podanym zbiorze elementów o podanej wadze i wartości, należy wybrać taki podzbiór by suma wartości była możliwie jak największa, a suma wag była nie większa od danej pojemności plecaka.



Rysunek 1 Graficzne przedstawienie problemu

Problem plecakowy często przedstawia się jako problem złodzieja rabującego sklep – znajduje on N towarów; i–ty przedmiot jest wart c_i oraz waży w_i. Złodziej chce zabrać jak najwartościowszy łup, przy czym nie może zabrać więcej niż *B* kilogramów. Nie może też zabierać ułamkowej części przedmiotów, gdzie jest to możliwe w ciągłym problemie plecakowym.

W pełni wielomianowy schemat aproksymacji (ang. Fully Polynomial-Time Approximation Scheme) to algorytm aproksymacyjny pozwalający na uzyskanie dowolnie dobrego rozwiązania przybliżonego danego problemu optymalizacyjnego, którego to złożoność czasowa jest wielomianowa i rośnie wielomianowo w miarę wzrostu żądanej dokładności.

Opis działania algorytmu

Algorytm zaprojektowano dla problemu plecakowego z podziałem na 2 części. Najpierw algorytm zmienia wartości przedmiotów zgodnie ze schematem. Potem następuje generowanie rozwiązań przy pomocy algorytmu dokładnego opartego o programowanie dynamiczne. Wygenerowane wyniki przechowywane są w tablicy dwuwymiarowej, a następnie skalowane. Skutkuje to zmniejszeniem tablicy wartości.

```
WE:
        ϵ ∈ [0,1]
        L // tablica n przedmiotów o wartościach vi i wagach wi
WY:
        S // rozwiązanie
-----------------------------------------------------------------
V = max {vi | 1 <= i <= n}    // największa wartość przedmiotu
K = ϵ * V / n
for i from 1 to n do
        vi = ⌊ vi / K ⌋            // przeskalowanie wartości
end for
return dynamic(L)
```

Przykład działania

Dany jest plecak z przedmiotami:

Nr	Waga	Wartość
1	7	75
2	8	150
3	6	250
4	4	35
5	3	10
6	9	100

Szukany jest najdroższy przedmiot. Największą V wartość ma przedmiot 3 – 250. Wartości przedmiotów są zmieniane w zależności od zadanego ε – przyjmijmy 0,5.

<table>
<tr><th>Nr</th><th>Waga w</th><th>Wartość v</th><th></th><th>Nowa wartość v</th></tr>
<tr><td>1</td><td>7</td><td>75</td><td rowspan="6">K = ε * V / n = 20,83

$v_i = \lfloor v_i / K \rfloor$</td><td>3</td></tr>
<tr><td>2</td><td>8</td><td>150</td><td>7</td></tr>
<tr><td>3</td><td>6</td><td>250</td><td>12</td></tr>
<tr><td>4</td><td>4</td><td>35</td><td>1</td></tr>
<tr><td>5</td><td>3</td><td>10</td><td>0</td></tr>
<tr><td>6</td><td>9</td><td>100</td><td>4</td></tr>
</table>

Dla nowopowstałych wartości wykonywany jest algorytm dynamiczny wg poniższego schematu:

```
y =0;
do
    {
    y = y + 1;
        for (k = 1; k<=n; k++)
          {
            if (y - pk < 0 || F[k-1][y- pk] == ∞)
                F[k][y] = F[k-1][y];
            else  F[k][y] = min(F[k-1][y], F[k-1][y- pk] + wk);
            if (F[n][y] <= V)  profit = y;
          }
    }while (y < n*p_max);

return profit;
```

Gdzie F jest funkcją rekurencyjną:

$$F(k,y)=\begin{cases} 0 \quad gdy \quad k=1...n, \quad y=0 \\ \infty \quad gdy \quad k=0, y\geq 1 \\ \min\left(F(k-1,y), F(k-1,y-p_k)+w_k\right) \quad k=1...n,\ y\geq 1 \end{cases}$$

Złożoność obliczeniowa algorytmu jest równa O(n²).

Implementacja algorytmu

Funkcją znajdującą rozwiązanie algorytmem dynamicznym jest `PlecakFPTAS` z vectorem rozmiarów przedmiotów `sizes` i ich wartości `values`, całkowitym rozmiarem plecaka `totalsize`, liczbą przedmiotów `itemsnum` i wartością błędu `epsilon` w argumentach. `Vector` jako struktura danych został wykorzystany ze względu na jego wygodę użycia i wystarczającą wydajność.

```
vector<short> PlecakFPTAS(vector<int> sizes, vector<int> values, int totalsize, int itemsnum, float epsilon)
{
   int bestValue = 0;
   int maxValue = 0;
   double scale;
   int size = sizes.size();

   vector <bool> bestChoice(size);
   vector <vector <int>> table(size+1);

   for (int i = 0; i < size; i++)
   {
      bestChoice[i] = false;
   }
   for (int i = 0; i < size; i++)
      if (maxValue < values[i])
         maxValue = values[i];
   scale = (epsilon * maxValue) / (double)size;
   maxValue = (double)maxValue / scale;
   for (int i = 0; i < size; i++)
   {
      values[i] = floor((double)values[i] / scale);
      table[i].resize(size * maxValue + 1);
   }
   table[size].resize(size * maxValue + 1);
   for (int i = 0; i <= size; i++)
      table[i][0] = 0;
   for (int i = 1; i <= size * maxValue; i++)
      table[0][i] = INT_MAX;
   for (int i = 1; i <= (size * maxValue); i++)
   {
      for (int j = 1; j <= size; j++)
      {
         if (i < values[j - 1])
            table[j][i] = table[j - 1][i];
         else if (table[j - 1][i - values[j - 1]] == INT_MAX)
            table[j][i] = table[j - 1][i];
         else
            table[j][i] = table[j-1][i]<table[j-1][i-values[j-1]]+sizes[j-1]?table[j-1][i]:table[j-1][i-values[j-1]]+sizes[j-1];
      }
      if (table[size][i] <= totalsize)
      {
         bestValue = i;
      }
   }
   for (int k = size-1; k >= 0; k--)
   {
      if (bestValue - values[k] >= 0)
      {
         if (table[k+1][bestValue] == table[k][bestValue - values[k]] + sizes[k])
         {
            bestChoice[k] = true;
            bestValue = bestValue - values[k];
         }
         cout << k << ": " << bestChoice[k] << ", ";
      }
   }
   cout << '\n';
   vector<short> wynik; // rozwiązanie
   wynik.resize(0);
   for (int i = 0; i < size; i++)
      if (bestChoice[i])
         wynik.push_back(i);
   for (int i = 0; i < wynik.size(); i++)
      cout << wynik[i] << " ";
   cout << '\n';

   // zwrócenie rozwiązania
   return wynik;
}
```

Testowanie

Pliki wejściowe dla programu mają następującą postać: liczba elementów plecaka i pary opisujące rzeczy (rozmiar i wartość).

Funkcja generująca rzeczy do umieszczenia w plecaku dopuszcza tylko takie rozwiązania, w których suma wartości wszystkich wygenerowanych przedmiotów przekracza całkowitą wielkość plecaka zwiększoną o 25%. W testowaniu losowych instancji wielkość plecaka jest ustawiana automatycznie według wzoru:

*Rozmiar = 1,5 * liczba_elementów*

Początkowy rozmiar plecaka to 75, a liczba elementów to 50. Rozmiar plecaka jest zwiększany o 75 w każdej iteracji aż do 900 i liczba elementów wzrasta od 50 do 600.

W rozszerzonych testach program wykonuje algorytm dla powyższych rozmiarów, dla każdego ustawiając wszystkie liczby elementów od 50 do 600 (co 50).

Do zbadania czasu działania algorytm był wykonywany po 100 razy (dla losowo generowanych instancji o zadanym rozmiarze), a następnie jego wyniki zostały uśrednione.

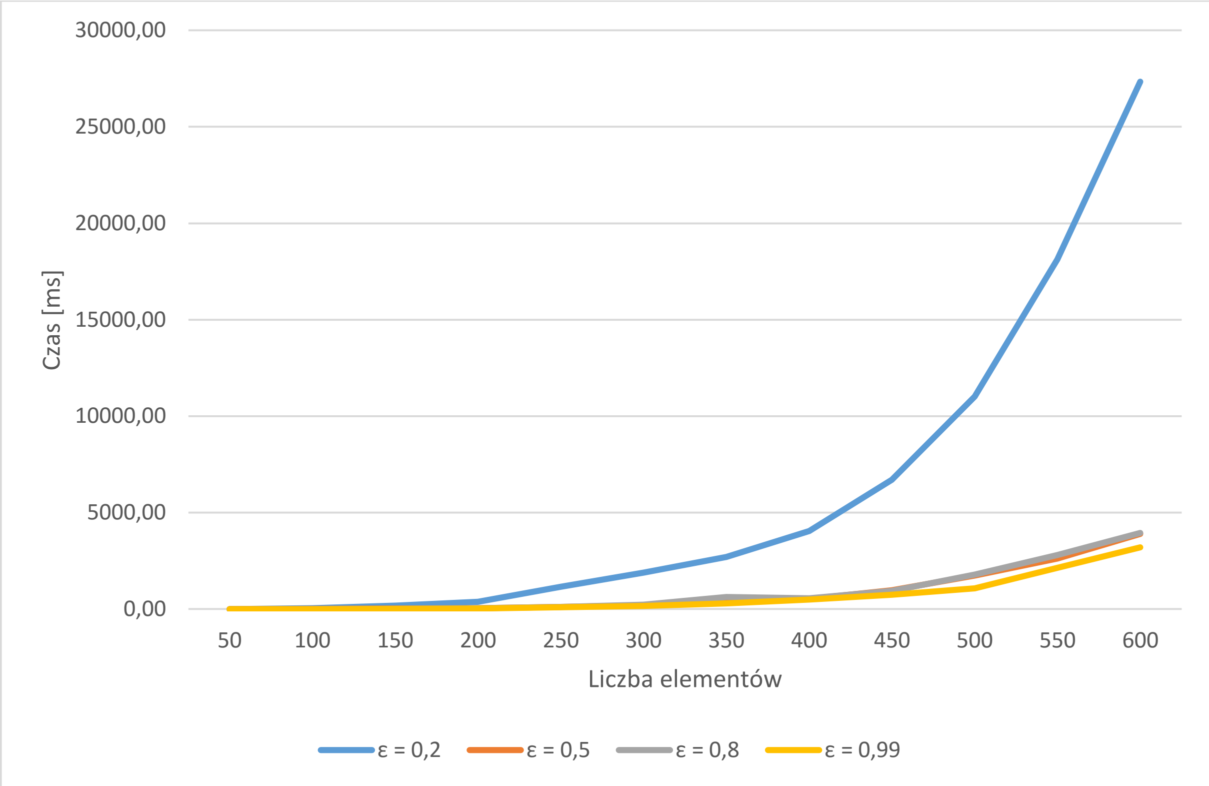
Czas wykonywania algorytmu był mierzony dzięki własnej klasie `MyTimer`. Zlicza ona takty zegara procesora i dzieli tę liczbę przez jego częstotliwość, a następnie mnoży przez 1000 aby wynik był podany w milisekundach, lub 1000000 dla wyniku w sekundach.

Wyniki

Poniżej przedstawiono wyniki pomiarów. Najpierw przeprowadzony został szybki test dla pojedynczych wielkości plecaka dla danej liczby elementów, potem został on rozszerzony o więcej wartości. W takiej samej kolejności prezentowane są wyniki.

Tabela 6 Pomiar czasów wykonywania się algorytmu dla pojemności równej 1,5 liczby elementów i ε = {0,2, 0,5, 0,8, 0,99}

Liczba elementów	Czas wykonywania [ms]			
	ε = 0,2	ε = 0,5	ε = 0,8	ε = 0,99
50	5,56	0,59	0,66	0,50
100	53,38	7,42	5,28	4,41
150	180,80	22,89	19,62	15,25
200	378,73	51,01	46,32	37,80
250	1160,72	112,45	117,46	90,79
300	1891,04	202,69	222,70	170,72
350	2698,08	587,47	630,01	294,51
400	4042,19	523,73	562,09	502,62
450	6701,16	971,44	942,64	751,30
500	11020,10	1733,05	1783,40	1081,49
550	18127,11	2614,83	2805,68	2138,62
600	27332,81	3890,45	3956,22	3200,66



Wykres 1 Czas wykonywania się algorytmu w zależności od liczby elementów dla różnych wartości ε

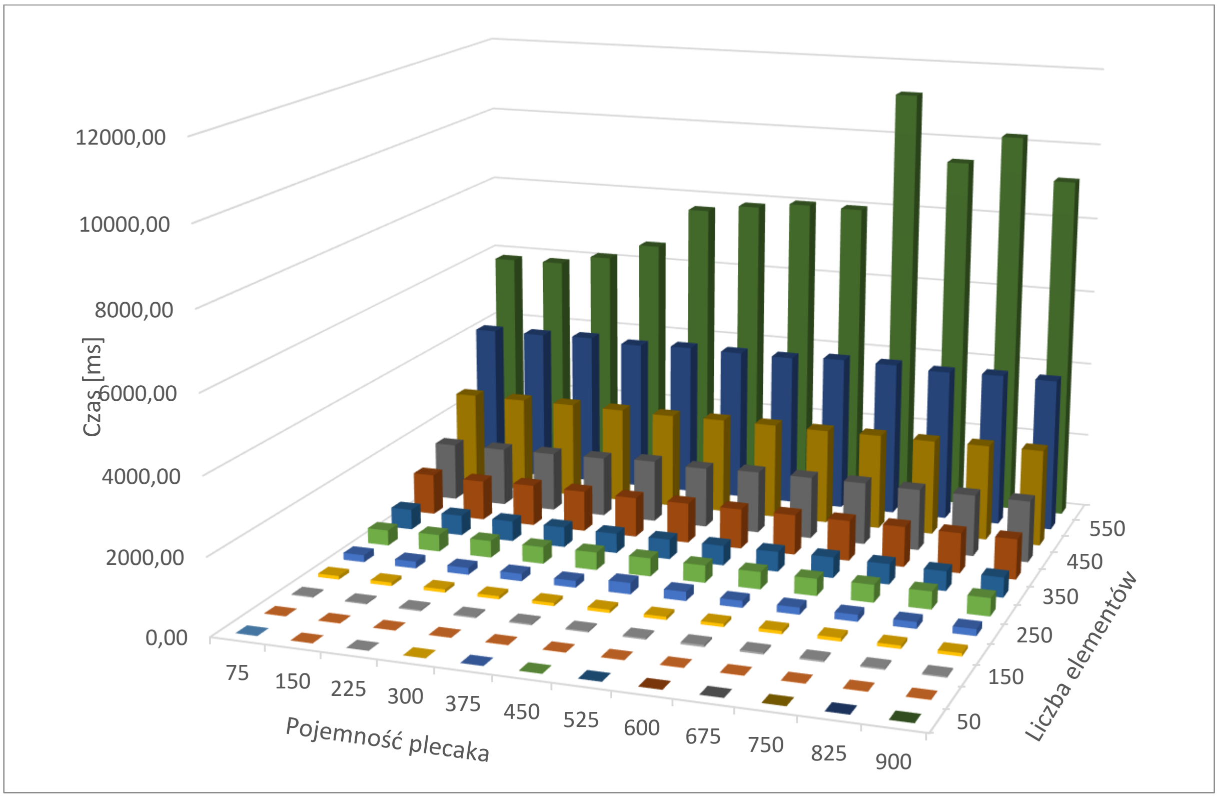
Widać, że dla ε = {0,5, 0,8, 0,99} czas jest bardzo podobny. Dla ε = 0,2 czas jest dużo większy.

Niewielkie rozbieżności między poprzednią, a następną tabelą wynikać mogą z wykonywania się obliczeń niezależnie od siebie przy różnym wykorzystaniu procesora i pamięci przez inne procesy.

Poniższe dane wyznaczono na podstawie obliczeń ze współczynnikiem ε = 0,5.

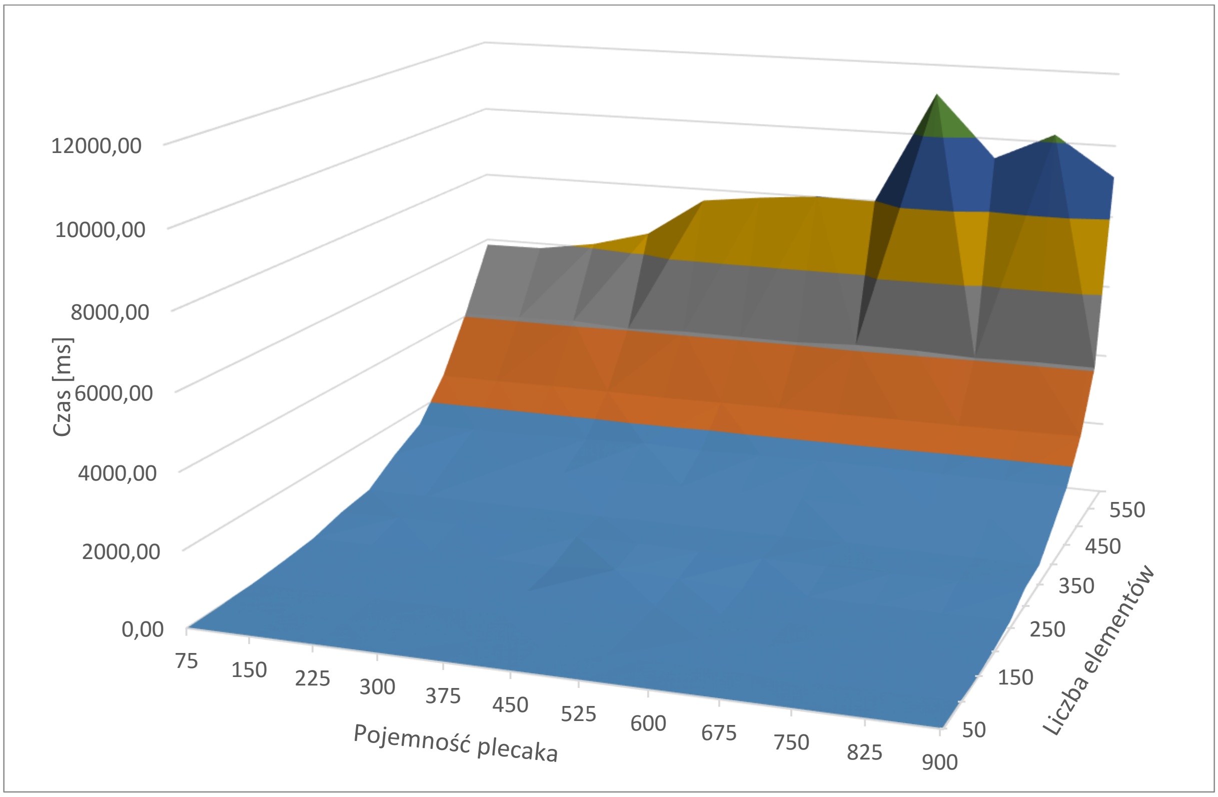
Tabela 7 Pomiar czasów wykonywania się algorytmu dla różnych pojemności i liczb elementów

Pojemność	Czas [ms]											
75	0,90	8,45	27,60	92,56	186,42	408,05	539,52	1086,97	1532,86	2574,94	4090,44	5841,87
150	0,97	8,41	30,02	91,02	180,47	445,51	533,95	1061,31	1551,33	2569,95	4096,24	5858,74
225	0,89	8,61	32,78	82,55	182,25	445,39	535,64	1095,83	1572,72	2574,57	4133,55	6111,84
300	1,02	8,61	34,21	81,55	194,10	446,95	533,01	1082,71	1594,97	2561,77	4044,32	6550,18
375	0,93	8,80	34,12	81,19	179,21	466,93	520,76	1061,44	1654,81	2540,20	4099,84	7664,73
450	1,02	8,68	32,03	82,88	296,45	479,65	524,07	1064,68	1606,58	2556,99	4079,64	7863,18
525	0,89	8,72	36,12	81,83	238,84	459,04	522,72	1063,33	1653,57	2555,45	4070,03	8016,52
600	1,00	8,84	51,87	81,91	180,33	456,49	523,47	1064,43	1657,37	2540,91	4140,43	7989,78
675	0,92	8,68	43,52	82,86	184,16	448,69	545,61	1068,86	1659,33	2551,11	4125,23	11186,87
750	1,04	8,60	40,59	88,95	177,88	466,06	536,29	1070,55	1632,01	2540,48	4057,52	9449,43
825	0,97	8,62	36,14	86,75	173,97	467,06	517,00	1066,02	1634,65	2556,01	4092,47	10217,98
900	0,94	8,53	32,87	82,85	175,83	472,61	523,71	1066,59	1629,68	2572,05	4077,02	9122,52
Liczba elementów	50	100	150	200	250	300	350	400	450	500	550	600



Wykres 2 Czas wykonywania się algorytmu w zależności od pojemności i liczby elementów (słupkowy)

Widać tu, że nie pojemność, a liczba elementów ma duży wpływ na czas obliczeń.



Wykres 3 Czas wykonywania się algorytmu w zależności od pojemności i liczby elementów (powierzchniowy)

Podsumowanie i wnioski

Widać, że pojemność plecaka nieznacznie zwiększa (w większości przypadków) czas wykonywania się algorytmu, ale największy wpływ ma liczba elementów. Ta też dla większych wartości bardziej zwiększa czas wykonywania się algorytmu.

Zgodnie z założeniami widoczny jest wielomianowy wzrost czasu wykonywania.

Bibliografia

- „Algorytmy aproksymacyjne”, http://www.asdpb.republika.pl/wyk78.pdf
- „Knapsack problem”, https://en.wikipedia.org/wiki/Knapsack_problem