Sprawozdanie - Lista 1

Jakub Zdancewicz

Spis treści

1 Wstęp

Sortowanie to jeden z fundamentalnych problemów informatyki, mający szerokie zastosowanie w wielu dziedzinach. Algorytmy sortowania odgrywają kluczową rolę w wielu aplikacjach, a ich wydajność istotnie wpływa na czas działania. Wybór odpowiedniego algorytmu sortującego jest zatem istotną decyzją podczas tworzenia oprogramowania. W niniejszym sprawozdaniu przeanalizujemy trzy popularne algorytmy sortowania wraz z ich wybranymi modyfikacjami:

- `Insertion Sort`
- `Merge Sort`
- `Heap Sort`

oraz porównamy ich efektywność na zestawach danych o różnych rozmiarach, badając liczbę operacji przypisań i porównań.

2 Opis zaimplementowanych algorytmów

Każdy z algorytmów został przetestowany na 10 losowo wygenerowanych tablicach dla każdej z 15 wybranych długości. Elementy tablic to liczby wymierne z zakresu $[-10^6, 10^6]$. Minimalna długość tablicy wynosiła 2, a maksymalna 100000. Średnia liczba operacji przypisań i porównań dla tablic o danej długości n jest wyliczana według wzoru:

$$L_n = \sum_{i=1}^{10} \left\lceil \frac{porownania_i}{10} \right\rceil + \left\lceil \frac{przypisania_i}{10} \right\rceil$$

gdzie:

L_n - średnia liczba operacji dla tablicy o długości n,

porownania$_i$ - liczba porównań wykonanych dla i-tej tablicy,

przypisania$_i$ - liczba przypisań wykonanych dla i-tej tablicy.

2.1 Insertion Sort

Insertion Sort to algorytm działający w czasie $\Theta(n^2)$. Poniżej przedstawiamy kluczowy fragment jego implementacji:

```
void insertion_sort(float A[], int n)
{
  for (int i = 1; i < n; ++i)
  {
    float x = A[i];
    int j = i - 1;
    while (j > -1 && A[j] > x)
    {
      A[j + 1] = A[j];
      --j;
    }
    A[j + 1] = x;
  }
}
```

Kod 1: Implementacja `Insertion Sort`

Algorytm działa poprzez wstawianie w i-tym kroku pętli **for** elementu $A[i]$ do wcześniej posortowanej części tablicy $A[0] \leqslant A[1] \leqslant \cdots \leqslant A[i-1]$. W pętli **while** przesuwamy wszystkie elementy większe od $A[i]$ o jeden indeks w prawo, a następnie wstawiamy $A[i]$ na odpowiednią pozycję, czyli po pierwszym elemencie mniejszym od $A[i]$ lub na początku tablicy, jeśli wszystkie elementy są większe.

2.1.1 Modyfikacja Insertion Sort

Rozważmy modyfikację algorytmu *Insertion Sort* polegającą na jednoczesnym wstawianiu dwóch kolejnych elementów tablicy. Kluczowy fragmenty implementacji przedstawiono poniżej:

```
void insertion_sort2(float A[], int n)
{
  for (int i = 1; i < n; i += 2)
  {
    float max = A[i];
    float min = A[i + 1];
    if (max < min)
    {
      float temp = max;
      max = min;
      min = temp;
    }
    int j = i - 1;
    while (j > -1 && A[j] > max)
    {
      A[j + 2] = A[j];
      --j;
    }
    A[j + 2] = max;
    while (j > -1 && A[j] > min)
    {
      A[j + 1] = A[j];
      --j;
    }
    A[j + 1] = min;
  }
  if (n % 2 == 0)
  {
    int j = n - 2;
    float key = A[n - 1];
    while (j > -1 && A[j] > key)
    {
      A[j + 1] = A[j];
      --j;
    }
    A[j + 1] = key;
  }
}
```

Kod 2: Implementacja Modyfikacji `Insertion Sort`

W tej modyfikacji algorytm przetwarza elementy tablicy parami, co umożliwia jednoczesne wstawienie dwóch kolejnych elementów. Na początku algorytm porównuje te dwa elementy i przypisuje większy do zmiennej **max**, a mniejszy do zmiennej **min**. Następnie przesuwa elementy większe od **max** o dwie pozycje w prawo. Gdy napotka element mniejszy $A[j_0]$ od **max** lub osiągnie początek tablicy ($j_0 = 0$), wstawia **max** na właściwe miejsce. Nastepnie, podobnie jak w oryginalnej wersji algorytmu, wstawia **min** do wcześniej posortowanej części tablicy $A[0] \leqslant \cdots \leqslant A[j_0]$.

W przypadku tablicy o parzystej liczbie elementów algorytm wykonuje dodatkowy krok: wstawia ostatni element tablicy do pozostałej części. Intuicyjnie, zmniejszenie liczby iteracji powinno skutkować mniejszą liczbą porównań i przypisań, co powinno wpłynąć na zwiększenie efektywności algorytmu.

Zakładając, bez utraty ogólności, że na wejściu otrzymujemy tablicę o nieparzystej liczbie elementów n, w pesymistycznym przypadku w każdej iteracji pętli **for** wykonujemy około 9 operacji oraz 4 operacje za każdą iterację w pętli **while**. Możemy zatem ilość operacji zapisać jako sumę:

$$T(n) = \sum_{i=1}^{\frac{n-1}{2}} 4i + 9 = 4 \cdot \frac{\left(\frac{n-1}{2}\right)\left(\frac{n-1}{2}+1\right)}{2} + 9 \cdot \frac{n-1}{2} = \Theta(n^2)$$

W przypadku gdy tablica ma parzystą długość, dodajemy dodatkowe $4(n-1)$ operacji, co nie zmienia ogólnej złożoności algorytmu.

2.1.2 Wyniki dla Insertion Sort

Tabele poniżej przedstawiają wyniki eksperymentów dla algorytmu *Insertion Sort* oraz jego modyfikacji na tablicach o różnej wielkości:

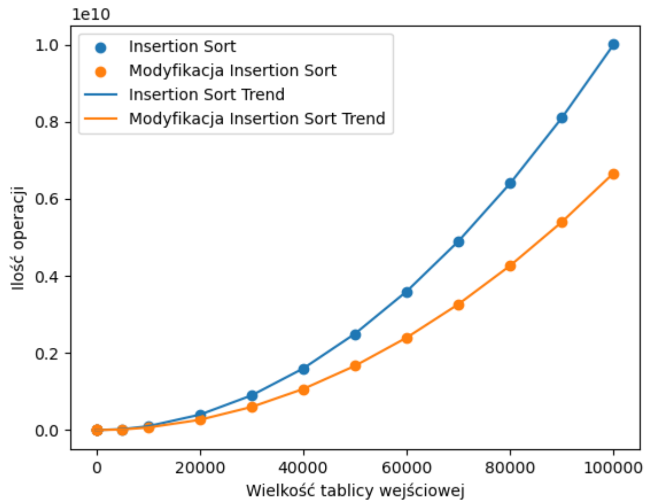
Wielkość tablicy	**Ilość przypisań**	**Ilość porównań**	**Łączna liczba operacji**
5	29	25	54
10	80	71	151
100	5,342	5,343	10,585
5,000	12,610,131	12,610,132	25,215,263
10,000	50,094,315	50,084,316	100,178,631
50,000	1,246,816,445	1,246,816,446	2,493,582,891
80,000	3,200,278,104	3,200,278,105	6,400,476,209
100,000	5,000,376,268	5,000,376,269	10,000,652,537

Tabela 1: Liczba operacji dla algorytmu `Insertion Sort` przy różnych rozmiarach tablicy

Wielkość tablicy	**Ilość przypisań**	**Ilość porównań**	**Łączna liczba operacji**
5	23	21	44
10	66	61	127
100	3,656	3,605	7,261
5,000	8,350,361	8,347,866	16,698,227
10,000	33,267,559	33,262,542	66,530,101
50,000	833,429,925	833,404,889	1,666,834,814
80,000	2,131,445,810	2,131,405,792	4,262,851,602
100,000	3,332,835,201	3,332,785,140	6,665,620,341

Tabela 2: Liczba operacji dla modyfikacji algorytmu `Insertion Sort` przy różnych rozmiarach tablicy

Co ciekawe, dla tablicy o długości n algorytm *Insertion Sort* wykonuje średnio n^2 operacji. Z tabeli jednoznacznie wynika, że modyfikacja algorytmu *Insertion Sort* jest wydajniejsza dla większych tablic. Różnice te są dobrze zobrazowane na Wykresie 1.



Wykres 1: Ilość operacji w zależności od rozmiarów tablicy wejściowej

2.2 Merge Sort

Merge Sort to algorytm sortowania działający w czasie $\Theta(n \log n)$. Poniżej przedstawiamy kluczowy fragment jego implementacji:

```
void merge_sort(float A[], int p, int k)
{
  if (p < k)
  {
    int s = p + (k - p) / 2;
    merge_sort(A, p, s);
    merge_sort(A, s + 1, k);
    merge(A, p, s, k);
  }
}
```

Kod 3: Implementacja `Merge Sort`

Algorytm działa poprzez rekurencyjne dzielenie tablicy na dwie podtablice aż do momentu, gdy każda z nich ma tylko jeden element. Takie tablice jednoelementowe są naturalnie posortowane. Następnie następuje proces łączenia dwóch posortowanych podtablic w jedną większą.

Funkcja **MERGE**, odpowiedzialna za łączenie posortowanych podtablic, jest zdefiniowana w następujący sposób:

```
void merge(float A[], int p, int s, int k)
{
  int n1 = s - p + 1;
  int n2 = k - s;
  float L[n1 + 1];
  float R[n2 + 1];
  L[n1] = numeric_limits<float>::infinity();
  R[n2] = numeric_limits<float>::infinity();
  for (int i = 0; i < n1; ++i)
  {
    L[i] = A[i + p];
  }
  for (int j = 0; j < n2; ++j)
  {
    R[j] = A[j + s + 1];
  }
  int i = 0;
  int j = 0;
  for (int l = p; l <= k; ++l)
  {
    if (L[i] <= R[j])
    {
      A[l] = L[i];
      ++i;
    }
    else
    {
      A[l] = R[j];
      ++j;
    }
  }
}
```

Kod 4: Implementacja `Merge`

Funkcja **MERGE** przypisuje elementy z tablicy $A[p \ldots s]$ do tablicy L oraz elementy z tablicy $A[s+1 \ldots k]$ do tablicy R. Następnie iteruje przez tablice L i R, wybierając za każdym razem mniejszy element i wstawiając go w odpowiednie miejsce w tablicy A. Na wyjściu otrzymujemy posortowaną tablicę, składającą się z elementów $A[p \ldots s]$ i $A[s+1 \ldots k]$.

2.2.1 Modyfikacja Merge Sort

W tej modyfikacji algorytm dzieli tablicę na trzy części zamiast dwóch. Kluczowe różnice w implementacji tego algorytmu w porównaniu do oryginalnej wersji obejmuje dodanie nowych miejsc podziału oraz dodanie dodatkowego rekurencyjnego wywołania w funkcji **MERGE_SORT**:

```
int s1 = p + (k - p) / 3;
int s2 = p + 2 * ((k - p) / 3);
merge_sort(A, p, s1);
merge_sort(A, s1 + 1, s2);
merge_sort(A, s2 + 1, k);
merge(A, p, s1, s2, k);
```

Kod 5: Fragment implementacji modyfikacji `Merge Sort`

Dodatkowo w funkcji **MERGE** wprowadzamy nową tablicę M:

```
float M[n2 + 1];
```

Kod 6: Fragment implementacji modyfikacji `Merge`

Działanie zmienionej funkcji jest analogiczne do oryginalnej, z tą różnicą, że łączenie odbywa się teraz dla trzech tablic zamiast dwóch:

```
int i = 0;
int j = 0;
int z = 0;

for (int l = p; l <= k; ++l)
{
    if (L[i] <= R[j] && L[i] <= M[z])
    {
        A[l] = L[i];
        ++i;
    }
    else if (M[z] <= R[j] && M[z] <= L[i])
    {
        A[l] = M[z];
        ++z;
    }
    else
    {
        A[l] = R[j];
        ++j;
    }
}
```

Kod 7: Fragment implementacji modyfikacji `Merge`

Ilość operacji wykonywanych przez modyfikację algorytmu *Merge Sort* możemy opisać poprzez rekurencyjne równanie:

$$T(n) = 3T(n/3) + n$$

Możemy zatem zapisać:

$$T(n) = \sum_{k=0}^{\log_3 n} 3^k \left(\frac{n}{3^k}\right) = n \sum_{k=0}^{\log_3 n} 1 = \Theta(n \log n)$$

Intuicyjnie, dzięki szybszemu przejściu do zbiorów jednoelementowych, modyfikacja powinna być efektywniejsza od orginału.

2.2.2 Wyniki dla Merge Sort

Tabele poniżej przedstawiają wyniki eksperymentów dla algorytmu *Merge Sort* oraz jego modyfikacji na tablicach o różnej wielkości:

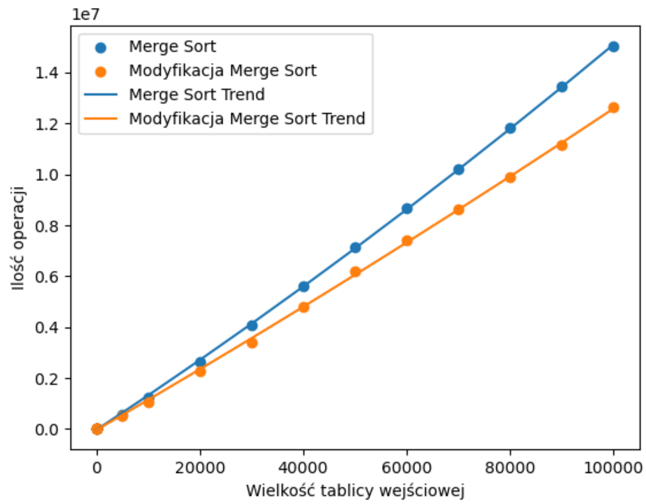
Wielkość tablicy	**Ilość przypisań**	**Ilość porównań**	**Łączna liczba operacji**
5	108	57	165
10	278	148	426
100	4,548	2,512	7,060
5,000	369,028	210,420	579,448
10,000	788,068	450,844	1,238,912
50,000	4,522,308	2,603,388	7,125,696
80,000	7,504,628	4,326,780	11,831,408
100,000	9,544,628	5,506,780	15,051,408

Tabela 3: Liczba operacji dla algorytmu `Merge Sort` przy różnych rozmiarach tablicy

Wielkość tablicy	**Ilość przypisań**	**Ilość porównań**	**Łączna liczba operacji**
5	99	67	166
10	256	180	436
100	3,560	2,791	6,351
5,000	277,738	234,773	512,511
10,000	555,727	481,819	1,037,546
50,000	3,318,096	2,880,715	6,198,811
80,000	5,238,096	4,635,715	9,873,811
100,000	6,706,625	5,931,401	12,638,026

Tabela 4: Liczba operacji dla modyfikacji algorytmu `Merge Sort` przy różnych rozmiarach tablicy

Z tabel wynika, że modyfikacja algorytmu *Merge Sort* jest wydajniejsza dla większych tablic. Co ciekawe, ilość porównań w modyfikacji algorytmu *Merge Sort* rośnie szybciej niż w orginalym algorytmie. Różnice w ilości wykonanych operacji są dobrze zobrazowane na Wykresie 2.



Wykres 2: Ilość operacji w zależności od rozmiarów tablicy wejściowej

2.3 Heap Sort

Heap Sort to algorytm sortowania działający w czasie $\mathcal{O}(n \log n)$. Poniżej przedstawiamy kluczowy fragment jego implementacji:

```
void heap_sort(heap &A, int n)
{
  build_heap(A, n);
  for (int i = n - 1; i > 0; --i)
  {
    float temp = A[i];
    A[i] = A[0];
    A[0] = temp;
    --A.heap_size;
    heapify(A, 0);
  }
}
```

Kod 8: Implementacja `Heap Sort`

Algorytm jest oparty na strukturze kopca binarnego (dokładniej, na kopcu typu maksymalnego), czyli drzewie binarnym, w którym wartość każdego węzła jest mniejsza niż wartość jego rodzica(własność **MAX_HEAP**). W naszej implementacji kopiec jest reprezentowany jako tablica, wraz z dodatkowym parametrem **heap_size**, który przechowuje aktualny rozmiar kopca:

```
struct heap
{
  float *array;
  int heap_size = 0;

  float &operator[](int idx)
  {
    return array[idx];
  }
  heap(float *arr)
  {
    array = arr;
  }
};
```

Kod 9: Implementacja `Heap`

Dodatkowo definiujemy dwie funkcje **LEFT** oraz **RIGHT** zwracające lewe i prawe dziecko węzła o indeksie i:

```
int left(int i)
{
  return 2 * i + 1;
}

int right(int i)
{
  return 2 * i + 2;
}
```

Kod 10: Implementacja `Left` i `Right`

Definiujemy również funkcję **HEAPIFY**, która rekurencyjnie naprawia własność **MAX_HEAP** dla węzła o indeksie i, pod warunkiem, że poddrzewa o wierzchołkach będących dziećmi węzła o indeksie i mają już własność **MAX_HEAP**. Funkcja działa poprzez zamianę węzła z dzieckiem o największej wartości, jeśli jest to konieczne:

```
void heapify(heap &A, int i)
{
  int l = left(i);
  int r = right(i);
  int largest = i;
  if (l < A.heap_size && A[l] > A[i]) {
    largest = l;
  }
  if (r < A.heap_size && A[r] > A[largest]) {
    largest = r;
  }
  if (i != largest)
  {
    float temp = A[i];
    A[i] = A[largest];
    A[largest] = temp;
    heapify(A, largest);
  }
}
```

Kod 11: Implementacja `Heapify`

Algorytm działa poprzez zbudowanie kopca z tablicy A przy pomocy funkcji **BUILD_HEAP**, która wywołuje funkcję **HEAPIFY** dla każdego węzła, który nie jest liściem:

```
void build_heap(heap &A, int n)
{
  A.heap_size = n;
  for (int i = (n / 2); i >= 0; --i)
  {
    heapify(A, i);
  }
}
```

Kod 12: Implementacja `Build Heap`

Następnie, algorytm iteracyjnie zamienia wierzchołek (największy element) z ostatnim liściem (węzłem o największym indeksie), wywołuje funkcję **HEAPIFY** na wierzchołku oraz zmniejsza parametr **heap_size** o jeden. Po zakończeniu pętli **for** otrzymujemy posortowaną tablicę A.

2.3.1 Modyfikacja Heap Sort

W tej modyfikacji algorytm używa kopców trenarnych zamiast binarnych. Kluczowe różnice w implementacji tego algorytmu w porównaniu do oryginalnej wersji obejmuje dodanie funkcji **MID**, która zwraca środkowe dziecko węzła, oraz zmiana kodu funkcji **LEFT** i **RIGHT**:

```
int left(int i)
{
  return 3 * i + 1;
}

int mid(int i)
{
  return 3 * i + 2;
}

int right(int i)
{
  return 3 * i + 3;
  ;
}
```

Kod 13: Implementacja `Left`, `Mid` i `Right`

Dodatkowo, w funkcji **HEAPIFY** wybieramy największe dziecko z trzech zamiast dwóch:

```
  int l = left(i);
  int m = mid(i);
  int r = right(i);
  if (l < A.heap_size && A[l] > A[i])
  {
    largest = l;
  }
  comparisions += 2;
```

```
if (m < A.heap_size && A[m] > A[largest])
{
  largest = m;
}
comparisions += 2;
if (r < A.heap_size && A[r] > A[largest])
{
  largest = r;
}
```

Kod 14: Część implementacji zmodyfikowanego `Heapify`

Reszta kodu pozostaje analogiczna. Dzięki posiadaniu większej liczby liści w kopcu trenarnym, które są pomijane w procedurze **BUILD_HEAP**, efektywność modyfikacji algorytmu powinna być wyższa.

BUILD_HEAP działa w czasie $\mathcal{O}(n)$, zatem zmodyfikowany algorytm działa w czasie $\mathcal{O}(n \log n)$.

2.3.2 Wyniki dla Heap Sort

Tabele poniżej przedstawiają wyniki eksperymentów dla algorytmu *Heap Sort* oraz jego modyfikacji na tablicach o różnej wielkości:

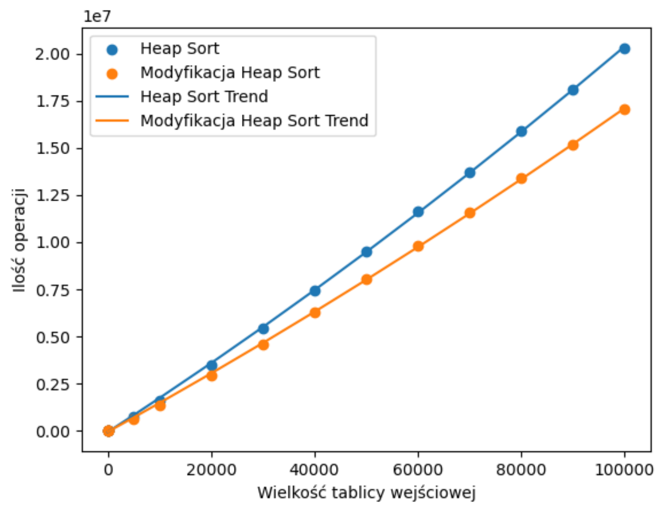
Wielkość tablicy	Ilość przypisań	Ilość porównań	Łączna liczba operacji
5	85	70	155
10	232	183	415
100	4,562	3,308	7,870
5,000	438,290	305,525	743,815
10,000	951,673	661,035	1,612,708
50,000	5,631,385	3,887,342	9,518,727
80,000	9,412,168	6,487,715	15,899,883
100,000	12,012,335	8,274,443	20,286,778

Tabela 5: Liczba operacji dla algorytmu `Heap Sort` przy różnych rozmiarach tablicy

Wielkość tablicy	Ilość przypisań	Ilość porównań	Łączna liczba operacji
5	83	78	161
10	211	195	406
100	3,803	3,290	7,093
5,000	347,976	289,958	637,934
10,000	749,096	621,005	1,370,101
50,000	4,405,541	3,633,093	8,038,634
80,000	7,338,882	6,037,845	13,376,727
100,000	9,348,531	7,686,258	17,034,789

Tabela 6: Liczba operacji dla modyfikacji algorytmu `Heap Sort` przy różnych rozmiarach tablicy

Z tabel wynika, że modyfikacja algorytmu *Heap Sort* jest wydajniejsza dla większych tablic. Różnice te są dobrze zobrazowane na Wykresie 3.

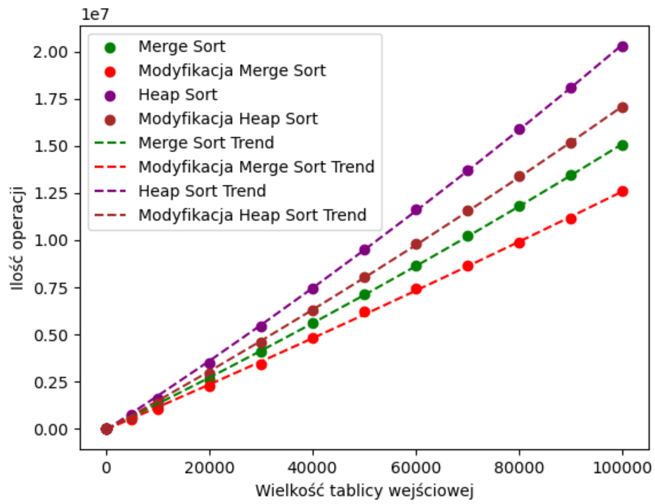


Wykres 3: Ilość operacji w zależności od rozmiarów tablicy wejściowej

3 Wnioski

Na podstawie przeprowadzonej analizy algorytmów, *Insertion Sort* może być wydajny dla niewielkich zestawów danych, jednak w miarę zwiększania się rozmiaru tablicy liczba wykonywanych operacji rośnie nawet o trzy rzędy wielkości w porównaniu do *Merge Sort* i *Heap Sort*.

Porównując *Merge Sort* i *Heap Sort* możemy zauważyć, że pierwszy z nich jest zauważalnie szybszy:



Wykres 4: Ilość operacji w zależności od rozmiaru tablicy wejściowej

Jednak *Merge Sort* posiada jedną znaczącą wadę – złożoność pamięciową. Z powodu konieczności definiowania dodatkowych tablic **L** i **R**, ilość pamięci potrzebnej do działania algorytmu jest nieporównywalnie większa niż w przypadku *Heap Sort*. Wybór odpowiedniego algorytmu zależy więc od dostępnych zasobów.

Zaproponowane modyfikacje algorytmów okazują się efektywniejsza od ich oryginałów; różnica jest szczególnie widoczna w przypadku dużych zestawów danych. Warto zatem rozważyć ich zastosowanie, zwłaszcza gdy zasoby są mocno ograniczone.