Daniel Hornik, Jakub Książek
studia niestacjonarne informatyka

ALGORYTM BAERA

Praca zaliczeniowa na przedmiot Systemy Operacyjne

Algorytm dla deterministycznych problemów szeregowania zadań, procesorów równoległych oraz zadań zależnych (graf antydrzewo), niepodzielnych, jednostkowych, procesorów jednorodnych dla minimalizacji długości uszeregowania.

Podstawowe pojęcia

W problemach szeregowanie bardzo ważną część odgrywa sprecyzowanie problemów – do tego zadania przychodzi nam z pomocą teoria grafów, która prócz szeregu przydatnych algorytmów dostarcza nam całej gamy struktur, które posiadają swoje indywidualne własności pozwalające na pracę z różnymi typami danych. Niektóre z tych struktur są podstawowymi i ważnymi obiektami wykorzystywanymi w informatyce. W opisie algorytmu Baera ustalimy, że zbiór zadań tworzy graf antydrzewo, jednak żeby dobrze zrozumieć ten termin będziemy musieli wyjaśnić dwa nieco powszechniejsze rodzaje grafów:

a) **graf spójny**, czyli taki graf, w którym z jednego wierzchołka można dotrzeć dokładnie jedną drogą do dowolnego innego wierzchołka.

b) **graf acykliczny**, czyli taki graf, który charakteryzuje brak cykli.

Wykorzystując powyższe definicje możemy zaprezentować definicję grafu typu antydrzewo:

> **Antydrzewo (lub antydendryt)** – jest to graf spójny i acykliczny, który ma dokładnie jeden następnik. Charakterystycznym parametrem tego typu grafu jest jego wysokość.

Aby w pełni zrozumieć czym jest wysokość drzewa, będziemy potrzebowali pojęcia **poziomu węzła** – czyli ilości krawędzi prowadzących od korzenia do węzła. W związku z tym możemy powiedzieć, że **wysokość drzewa** jest to maksymalny poziom węzła.

W dalszej części dokumentacji będziemy równie często posługiwać się takimi terminami jak **zasób**, czyli pewna wymagana przez zadanie wielkość np. pamięć lub procesor, która jest konieczna do wykonania zaprogramowanych instrukcji. Zasoby są wykorzystywane przez **zadanie**, czyli zbiory instrukcji programu komputerowego wykorzystujące jego zasoby w celu realizowania odpowiednich zadań.

Ostatnim ale nie najmniej ważnym terminem jest **Wykres Gantta**, będący graficzną reprezentacją procesu wykonywania zadań w czasie.

Opis działania algorytmu Baera

Algorytm Baera jest to jedyny znany algorytm wielomianowy dający optymalny wynik dla zadań niepodzielnych dla procesorów jednorodnych, przyjmując trzy dodatkowe założenia:

1. Procesory są jednorodne z jednostkowymi współczynnikami prędkości

   $Procesor_1$: $b_1 = 2$

   $Procesor_2$: $b_2 = 1$

2. Jednostkowe czasy wykonywania zadań
3. Zależność zadań między sobą wyraża antydrzewo.

Istotną cechą omawianego algorytmu jest to, że tylko wolniejszy procesor (P_2) może mieć przerwę w wykonywaniu zadania. Jeśli mamy w danej chwili *t* dostępne zadania to w pierwszej kolejności przydzielamy je szybszemu procesorowi – w naszym przypadku P_1. Ponadto algorytm posiada tylko 5 kroków, z tym że po krokach drugim, trzecim i czwartym sprawdzany jest warunek, który w przypadku wykonania przedostatniego zadania kończy działanie pętli i przydziela ostatnie zadanie na szybszy procesor.

1. Określ poziomy zadań (poziom zadania jest to poziom węzła na antydrzewie)
   Ustaw licznik zadań: q = n,
   Ustaw k1 = k2 = 0
2. Przydziel dostępne w chwili k1 zadanie o najwyższym poziomie do procesora P1
   Ustaw: q = q-1, k1 = k1+1
3. Jeśli są dostępne w chwili k2 zadania to przydziel to o najwyższym poziomie do procesora P2
   Ustaw: q = q-1, k2 = k2+2,
4. Jeśli dostępne jest zadanie w chwili k1 to przydziel zadanie o najwyższym poziomie do procesora P1.
   Ustaw q = q-1, k1=k1+1

5. Jeśli q = 1 to przejdź do kroku 5, w przeciwnym wypadku powtórz krok 2.
   Przydziel ostatnie zadanie do procesora P1.
   *C*max = k1+1;*

Powyższy algorytm jest algorytmem optymalnym, tzn. jeśli tylko spełnione są założenia algorytmu daje najlepsze rozwiązanie dla określonej klasy problemów.

Przykład zastosowania:

Szeregowanie zadań może być wykorzystane w systemach komputerowych ale równie często a nawet i częściej jest stosowane w innych dziedzinach nauki np. w obrabianiu metali.

Mamy do wykonania *n* elementów na dwóch obrabiarkach. Pierwsza grupa elementów musi być wykonane tylko wtedy, kiedy mamy elementy z drugiej grupy, aby można było dopasować je do siebie. Zależność wykonywania elementów można przedstawić za pomocą grafu typu antydrzewo. Dodatkowo jedna obrabiarka pracuje dwa razy szybciej niż druga. Wszystkie części są wykonywane w takim samym okresie czasu. Przykład spełnia wszystkie założenia, więc można zastosować algorytm Baera. W naszym przykładzie role procesorów jednorodnych przejmują obrabiarki, a role zadań zajmuje wykonywanie poszczególnych elementów.

Schemat blokowy algorytmu Baera

Złożoność

Złożoność obliczeniowa, czyli czas jaki potrzebuje algorytm na zrealizowanie określonej wielkości problemu, zależy od tego ilości operacji jakie algorytm musi wykonać aby zakończyć swoje działanie. Jeśli algorytm posiada złożoność wykładniczą to znaczy, że nadaje się do obliczenia tylko małej ilości danych wejściowych. W praktyce okazuje się często, że algorytmy te w zupełności wystarczą, gdyż ryzyko posiadania większych ilości danych jest znikome.

Algorytm Baera posiada złożoność wielomianową, która wynosi $O_{(n + c),}$ gdzie *c* to stała ilość dodatkowych operacji. Złożoność algorytmu jest tak mała ponieważ każde zadanie zostaje zdjęte z grafu dokładnie jeden raz. Dodatkowo wymaga się aby zadania miały wyznaczone poziomy. Poziom odpowiada pseudo priorytetowi i ogranicza wykonywanie algorytmu do zadań o najwyższym priorytecie. Dodatkowo do złożoności wchodzą dodatkowe operację sprawdzania i zmian w pamięci.

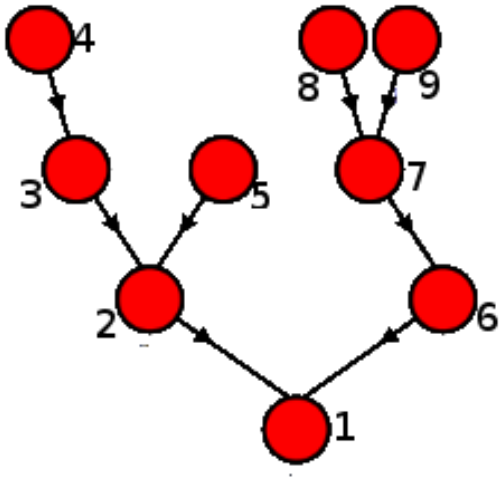
Złożoność pamięciowa zależy od sposobu składowania grafu oraz od tego jakie dodatkowe informacje zawierają zadania – nie można jej jednoznacznie określić, jest ona unikalna dla każdej konkretnej implementacji.

Przykładowo dla danych przechowywanych w tablicy dwuwymiarowej złożoność będzie wynosić: $O_{(n^2 + k * n + c)}$, gdzie *c* to pamięć potrzebna do przechowywania dodatkowych zmiennych, *k * n* pamięć do przechowywania dodatkowych danych np. dla poziomów każdego z wierzchołków. N^2- jest potrzebne tyle pamięci ponieważ dane przechowywane są w postaci macierzy sąsiedztwa reprezentowanej przez dwuwymiarową tablice.

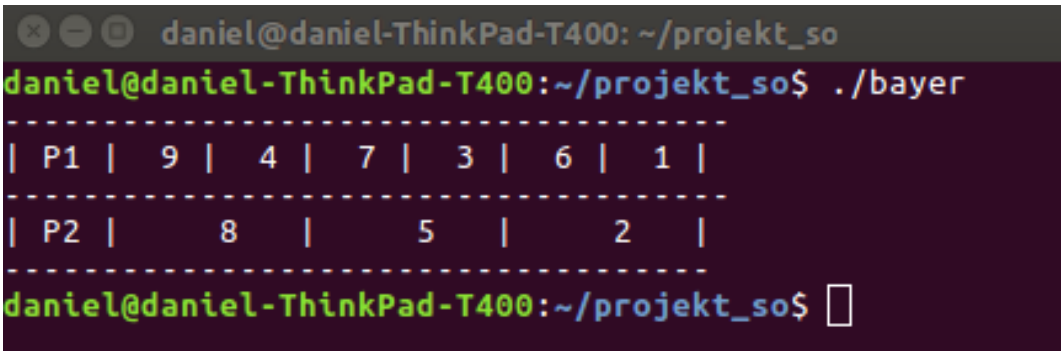
W niektórych przypadkach lepszym rozwiązaniem będzie zastosowanie listy incyndencji, lub innej struktury danych, która nadawałaby się lepiej do realizacji tego typu zadań, ponieważ w grafie typu antydrzewo każdy wierzchołek posiada tylko jeden następnik. Wszystko zależy od potrzeb oraz implementacji.

Wizualizacja

Dynamiczna wizualizacja działania algorytmu Baera znajduje się pod adresem http://baer.jakubksiazek.malopolska.pl. Została ona wykonana w technologii HTML5 wzbogaconej o technologię *Scalable Vector Graphics* (SVG) animowanej przy pomocy *JavaScriptu*. Przykładowe drzewo wygenerowane przy pomocy skryptu prezentuje się następująco:



Na podstawie wyżej zadanego problemu (antydzewa) algorytm zarówno w wersji wizualnej (javascrip) jak i w postaci kodu napisanego w c++ zwróci niżej zaprezentowany wykres Gantta:



Kod algorytmu (C++):

Najlepszym sposobem na zrozumienie i zaprezentowanie algorytmu jest przełożenie go na język zrozumiały dla komputera, niżej prezentujemy kod napisany w C++ opatrzony stosownymi komentarzami wyjaśniającymi poszczególne kroki algorytmu.

```
/**
 * Funkcja zlicza ilość węzłów w grafie przedstawiającym zależności zadań
 */
int countNodes(int levels)
{
        if (levels < 1) {
                return 2;
        }
        return pow(2, levels) + countNodes(--levels);
}

/**
 * Funkcja sprawdza czy przekazany graf jest antydrzewem, jeśli nie to
 * zwraca 0 i wyświetla odpowiedni komunikat
 * graph - wskaźnik na typ integer, wskazuje na macież grafu.
 */
int matrixTest(int *graph) {
        int x;
        int y;
        int nodes = countNodes(MAX_LEVEL-1);
        int realIndex;
        int children;
        for (x=0; x<nodes; x++) {
                children = 0;
                for (y=0; y<nodes; y++) {
                        realIndex = x + nodes*y;
                        if (graph[realIndex] == 1) {
                                children++;
                        }
                }
                if (children > 1) {
                        printf("Przekazana macież nie jest antydrzewem,
algorytm nie działa dla takiego grafu\n");
                        return 0;
                }
        }

        return 1;
}
```

```
/**
 * Funkcja wyznacza poziomy węzłów
 */
int findJumps(int *graph, int x, int jumps) {
        int nodes = countNodes(MAX_LEVEL-1);
        int y;


        for(y=nodes-1; y>=0; y--) {
                int realIndex = x + nodes*y;
                if (graph[realIndex] == 1) {
                        return findJumps(graph, y, ++jumps);
                }
        }

        return jumps;
}

/**
 * Funkcja zbiera węzeł o najwyższym poziomie z tablicy węzłów
 */
int findMaxIndex(int *levels, int nodes) {
        int maxValue = 0;
        int maxIndex = -1;
        int i;
        for (i=nodes-1;i>=1;i--) {
                if (levels[i] > maxValue) {
                        maxIndex = i;
                        maxValue = levels[i];
                }
        }
        if (maxIndex > 0) {
                levels[maxIndex] = 0;
        }

        return maxIndex;
}

/**
 * Pobiera kolejny węzeł z tablicy
 */
int getNextNode(int *levels, int nodes) {
        int node = findMaxIndex(levels, nodes);

        if (node == -1) {
                node = 0;
        }

        return node+1;
}
```

```
/**
 * Narysowanie wizualizacji, przekazane tablice proc1Arr i proc2Arr
trzymają kolejne numery zadan
 * A tablice Amount trzymają ilośc każdego na procesorze
 */
void printVisualalization(int *proc1Arr, int *proc2Arr, int proc1Amount,
int proc2Amount) {
        int j;
        printf("-------");
        for (j=0;j<proc2Amount;j++) {
                printf("----------");
        }
        printf("\n| P1 |");
        for (j=0;j<proc1Amount;j+=2) {
                printf(" %2d | %2d |", proc1Arr[j], proc1Arr[j+1]);
        }
        printf("\n-------");
        for (j=0;j<proc2Amount;j++) {
                printf("----------");
        }
        printf("\n| P2 |");
        for (j=0;j<proc2Amount;j++) {
                printf("    %2d   |", proc2Arr[j]);
        }
        printf("\n------");
        for (j=0;j<proc2Amount;j++) {
                printf("----------");
        }
        printf("\n");
}

void main() {
        int alocatedSize =  countNodes(MAX_LEVEL-1);
        alocatedSize *= alocatedSize;
        int *graph = malloc(alocatedSize * sizeof(int));
        /**
         * Zakodowana maciez antydrzewa
         */
int arr[256] = {
/*  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16*/
        /* 1*/  0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 2*/  0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 3*/  0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 4*/  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 5*/  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 6*/  0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 7*/  0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,
        /* 8*/  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 9*/  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 10*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 11*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 12*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 13*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 14*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 15*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        /* 16*/ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

```
        // 9 8 4 7 5 3 6 2 1
        memcpy(graph, arr, sizeof(int)*alocatedSize);

        if (matrixTest(graph) == 0) {
                return;
        }

        int nodes = countNodes(MAX_LEVEL-1);
        int *levels = malloc(sizeof(int)*nodes);


        int x = 0;
        for (x=0; x< nodes; x++) {
                levels[x] = findJumps(graph, x, 0);
        }


        int maxLevel=-1;
        int maxNode;
        int _nodes = 1;
        for (x=1; x<nodes; x++) {
                if (levels[x] > 0) {
                        _nodes++;
                } else {
                        break;
                }
        }


        /* Właściwa część algorytmu */
        int *proc1 = malloc(sizeof(int) * nodes);
        int *proc2 = malloc(sizeof(int) * nodes);
        int _proc1 = 0;
        int _proc2 = 0;
        int j = _nodes;
        while(j > 0) {
                proc1[_proc1++] = getNextNode(levels, _nodes);
                j--;
                if (j>1) {
                        proc2[_proc2++] = getNextNode(levels, _nodes);
                        proc1[_proc1++] = getNextNode(levels, _nodes);
                        j-=2;
                } else {
                        proc2[_proc2++] = 0;
                        proc1[_proc1++] = getNextNode(levels, _nodes);
                        j--;
                }

        }

        printVisualalization(proc1, proc2, _proc1, _proc2);

}
```