1. ЦЕЛЬ

Изучить и исследовать методы планирования заданий для мультипрограммных систем.

1. ПОСТАНОВКА ЗАДАЧИ

Найти такой порядок выполнения множества заданий, при котором критерий эффективности суммарного времени обработки пакета заданий принимает минимальные значение, используя точный и эвристический алгоритм для трёхфазной модели.

1. ЗАДАНИЕ

Разработать программу, моделирующую работу заданных алгоритмов планирования заданий для мультипрограммной системы.

Исходные данные к программе:

– матрица трудоёмкости.

Результаты:

– план обработки пакета заданий;

– временная диаграмма Ганта.

1. ОПИСАНИЕ ПРОГРАММЫ

При запуске программы предлагается ввести количество заданий, требующих планирования. После этого выводится запрос на заполнение матрицы трудоемкости. Матрица заполняется данными, отражающими потребность каждого задания в устройствах трехфазной модели. Ввод данных осуществляется в порядке увеличения номера задания, для каждого устройства.

После ввода данных, программа приступает к поиску оптимального порядка выполнения заданий с минимизацией суммарного времени выполнения. Поиск происходит в 2 этапа.

На первом этапе проверяются условия приведения данной матрицы, к двухфазной модели для применения точного алгоритма планирования. Если условия не выполняются будет выведено сообщение об ошибке, в случае успеха будет выведен найденный порядок заданий.

На втором этапе к этим же исходным данным применяется эвристический алгоритм и будет выведен результат.

1. ТЕКСТ ПРОГРАММЫ

#include <iostream>

#include <memory>

#include <vector>

#include <algorithm>

#include <functional>

#include <numeric>

using namespace std;

struct matrixIndexes {

int i;

int j;

};

typedef std::vector<int> vect;

typedef std::vector<std::vector<int>> matr;

typedef std::vector<bool> taskVect;

void arrReset(vect &arr) {

for (auto &elem : arr) {

elem = 0;

}

}

void matrixReset(matr &m) {

for (size\_t i = 0; i < m.size(); ++i) {

for (size\_t j = 0; j < m[0].size(); ++j) {

m[i][j] = 0;

} }}

void printArr(vect &a){

for (auto &i : a){

std::cout << i << " ";

}}

int minIndex(const vect &a, const taskVect &t) {

int minIndex = 0;

while (t[minIndex]) {

++minIndex;

}

for (size\_t i = 0; i < a.size(); ++i) {

if (a[i] < a[minIndex] && t[i] == false) {

minIndex = i;

}

}

return minIndex;

}

/\*

if (min(0) or min(2) <= max(1)) return 1 else return 0

\*/

int checkAlgorithm(const matr &m) {

if (m.size() != 3){

return false;

}

vect tempVal(3, 0);

for (size\_t i = 0; i < m.size(); ++i) {

int index = 0;

if (i != 1) {

for (size\_t j = 0; j < m[0].size(); ++j) {

if (m[i][j] < m[i][index]) {

index = j;

}

}

}

else { for (size\_t j = 0; j < m[0].size(); ++j) {

if (m[i][j] > m[i][index]) {

index = j;

} } }

tempVal[i] = index; }

if ((m[0][tempVal[0]] >= m[1][tempVal[1]]) || (m[2][tempVal[2]] >= m[1][tempVal[1]])) {

return 1;

}

return 0;

}

vect matrColumnSum(const matr &m,const size\_t col1, const size\_t col2) {

vect temp;

for (size\_t i = 0; i < m[0].size(); ++i) {

temp.push\_back(m[col1][i] + m[col2][i]);

}

return temp; }

int min(const int &a, const int &b) {

return (a > b) ? b : a;

}

vect generateJonsonOrder(const matr &m) {

if (m.size() != 2) {

throw "invalid vector size";

}

vect orderedVect(m[0].size(), 0);

taskVect tasks(m[0].size(), false);

int forwardIter = 0;

int backIter = orderedVect.size() - 1;

while (forwardIter <= backIter) {

int minI = 0, minJ = 0;

while (tasks[minJ] == true) {

++minJ;

}

for (size\_t i = 0; i < m.size(); ++i) {

for (size\_t j = minJ + 1; j < m[0].size(); ++j) {

if (tasks[j] == false && m[i][j] < m[minI][minJ]) {

minI = i;

minJ = j;

} } }

int minVal = m[minI][minJ];

for (size\_t i = 0; i < m[0].size(); ++i) {

if (tasks[i] == false){

if (m[0][i] == minVal) {

orderedVect[forwardIter++] = i;

tasks[i] = true;

} else if (m[1][i] == minVal) {

orderedVect[backIter--] = i;

tasks[i] = true;

} } } }

return orderedVect;}

vect frontOrder(const vect &a) {

vect temp = a;

sort(temp.begin(), temp.end());

return temp;

}

vect BackOrder(const vect &a) {

vect temp = a;

sort(temp.begin(), temp.end(), greater<int>());

return temp;}

vect generateFrontOrder(const vect &a) {

vect temp;

taskVect tasks(a.size(), false);

for (size\_t i = 0; i < a.size(); ++i){

int tempIndex = minIndex(a, tasks);

temp.push\_back(tempIndex);

tasks[tempIndex] = true;

}

return temp;}

int MaxValIndex(const vect &a) {

int index = 0;

for (size\_t i = 0; i < a.size(); ++i) {

if (a[i] > a[index]) {

index = i;

} }

return index;}

vect runExactAlgo(matr &m) {

vect temp;

if (checkAlgorithm(m)) {

vect A = matrColumnSum(m, 0, 1);

vect B = matrColumnSum(m, 1, 2);

matr T = { A , B};

temp = generateJonsonOrder(T);

return temp; }

throw "Error";}

vect runEvristicAlgo(matr &m){

vect temp;

vect colSumVal(m.size(),0);

for (size\_t i = 0; i < m.size(); ++i) {

colSumVal[i] = accumulate(m[i].begin(), m[i].end(),0);

}

int changeEvristicAlg = MaxValIndex(colSumVal);

switch (changeEvristicAlg) {

case 0: {

vect tempSumCol1Col2 = matrColumnSum(m,1,2);

temp = generateFrontOrder(tempSumCol1Col2);

reverse(temp.begin(), temp.end());

break;

}

case 1: {

m.erase(m.begin() + 1);

temp = generateJonsonOrder(m);

break;

}

case 2: {

vect tempSumCol1Col2 = matrColumnSum(m, 0, 1);

temp = generateFrontOrder(tempSumCol1Col2);

break; } }

return temp;}

int main() {

vect A = { 2, 1, 4, 5, 3 };

vect B = { 3, 1, 2, 3, 4 };

vect C = { 4, 5, 5, 7, 6 };

matr J = { A, B, C };

vect temp1 = runExactAlgo(J);

vect temp2 = runEvristicAlgo(J);

printArr(temp1);

cout << endl;

printArr(temp2);

return 0;

}

1. ТЕСТОВЫЕ ПРИМЕРЫ

Для проверки работоспособности программы, была использована матрица приведенная в методических указаниях к лабораторной работе (табл.1). В результате чего были получены соответствующие данные, в чем можно убедиться на рисунке 1.

Таблица 1 – Матрица трудоемкости №1.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Y1 | Y2 | Y3 |
| J1 | 2 | 3 | 4 |
| J2 | 1 | 1 | 5 |
| J3 | 4 | 2 | 5 |
| J4 | 5 | 3 | 7 |
| J5 | 3 | 4 | 6 |

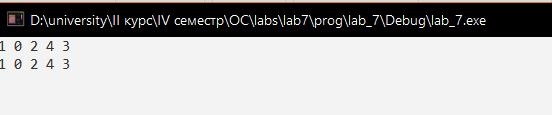


Рисунок 1 – Результат выполнения программы.

Для полученных данных составлена диаграмма Ганта:

Рисунок 2 – Диаграмма Ганта

В ходе построения диаграммы было получено, что суммарная длительность выполнения программ составляет 29 ед. времени,

Для второго тестирования были использованы исходные данные таблицы 2:

Таблица 2 – Матрица трудоемкости №2.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Y1 | Y2 | Y3 |
| J1 | 2 | 4 | 4 |
| J2 | 8 | 1 | 7 |
| J3 | 4 | 2 | 6 |
| J4 | 3 | 2 | 5 |
| J5 | 1 | 3 | 5 |

Т.к. для данной матрицы выполняется условие min(Ri3) ≥ max (Rj2), то может быть применен точный алгоритм планирования заданий.

Приведя матрицу к двухстолбцовой, получим:

Таблица 3 – Приведенная матрица трудоемкости №3.

|  |  |  |
| --- | --- | --- |
|  | Y1 | Y2 |
| J1 | 6 | 8 |
| J2 | 9 | 8 |
| J3 | 6 | 8 |
| J4 | 5 | 7 |
| J5 | 4 | 8 |

Применив алгоритм Джонсона получим следующий порядок выполнения заданий: J5, J4, J1, J3, J2. Аналогичный порядок получим, применив эвристический алгоритм. Выделим номер фазы с наибольшей суммарной продолжительностью заданий: Y1 = 18, Y2 = 12, Y3 = 24. Так как наибольшая продолжительность выполнения заданий у третей фазы, запускать задания требуется в порядке возрастания величины (Ri1 + Ri2).

Результат выполнения программы совпадает с теоретическими расчетами.

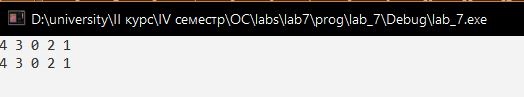


Рисунок 3 – Результат выполнения программы.

Рисунок 2 – Диаграмма Ганта

В результате построения диаграммы было полученно, что суммарная длительность выполнения пакета программ составляет 27 ед. времени.

ВЫВОД

В ходе выполнения лабораторной работы была промоделирована мультипрограммная система, осуществляющая планирование по критерию минимизации суммарного времени обработки пакета заданий. Для этого были применены средства языка С++. Данная задача выполнялась с использованием точного и эвристического алгоритмов для трёхфазной модели.

По полученным результатам и построенным диаграммам, можно сделать вывод, что программа работоспособна.