Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа №5
по дисциплине "Анализ Алгоритмов"

Тема Конвейерная обработка

Студент Светличная А.А.

Группа ИУ7-53Б

Преподаватель Волкова Л. Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Конвейерная обработка данных — выполнение каждой команды складывается из ряда последовательных этапов (шагов, стадий), суть которых не меняется от команды к команде. С целью увеличения быстродействия процессора и максимального использования всех его возможностей в современных микропроцессорах используется конвейерный принцип обработки информации. Этот принцип подразумевает, что в каждый момент времени процессор работает над различными стадиями выполнения нескольких команд, причем на выполнение каждой стадии выделяются отдельные аппаратные ресурсы. По очередному тактовому импульсу каждая команда в конвейере продвигается на следующую стадию обработки, выполненная команда покидает конвейер, а новая поступает в него.

Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах [1].

1 Аналитическая часть

1.1 Цель и задачи

Цель работы: изучить принципы конвейерной обработки данных и реализовать алгоритм Дейкстры с генерацией данных по заданным параметрам и выводом результата в определенном формате.

Задачи лабораторной работы:

- изучить понятие конвейерной обработки данных;
- реализовать алгоритм Дейкстры;
- реализовать последовательную обработку данных на основе алгоритм Дейкстры;
- реализовать конвейерную обработку данных на основе алгоритм Дейкстры;
- провести сравнительный анализ времени работы линейной и конвейерной обработки данных на основе экспериментальных данных.

1.2 Алгоритм Дейкстры

Алгоритм Дейкстры — это последовательность действий, позволяющих оптимально найти на графе кратчайший путь от некоторой его вершины до всех остальных [2].

Основная идея данного алгоритма заключается в том, что на каждом шаге помечается определённым образом выбранная вершина, а далее просматриваются все последующие (ещё не отмеченные) вершины графа и вычисляется длина пути до каждой из них от начальной точки. Помеченная на каждом этапе новая вершина (та, до которой путь оказался кратчайшим) становится определяющей для следующего шага. И этот построенный до неё кратчайший путь закрашивается. Повторяя данные действия, последовательно просматривая и поочерёдно отмечая вершины графа, в

итоге алгоритм добирается до конечного пункта. Те рёбра графа, которые в результате оказались закрашенными (вместе с отмеченными их крайними точками), образуют ориентированное дерево кратчайших путей (с корнем в исходной вершине графа). Далее остаётся выбрать на нём путь, который соединяет начальную точку с конечной. Это и будет искомый путь.

Вывод

В данном разделе был изучен и рассмотрен алгоритм Дейкстры, хорошо подходящий для конвейерной обработки, так как делится на три этапа: генерация матрицы смежности, решения алгоритма Дейкстры, вывод данных.

2 Конструкторская часть

Описание алгоритмов

На рисунке 2.1 представлены схема алгоритма Дейкстры.

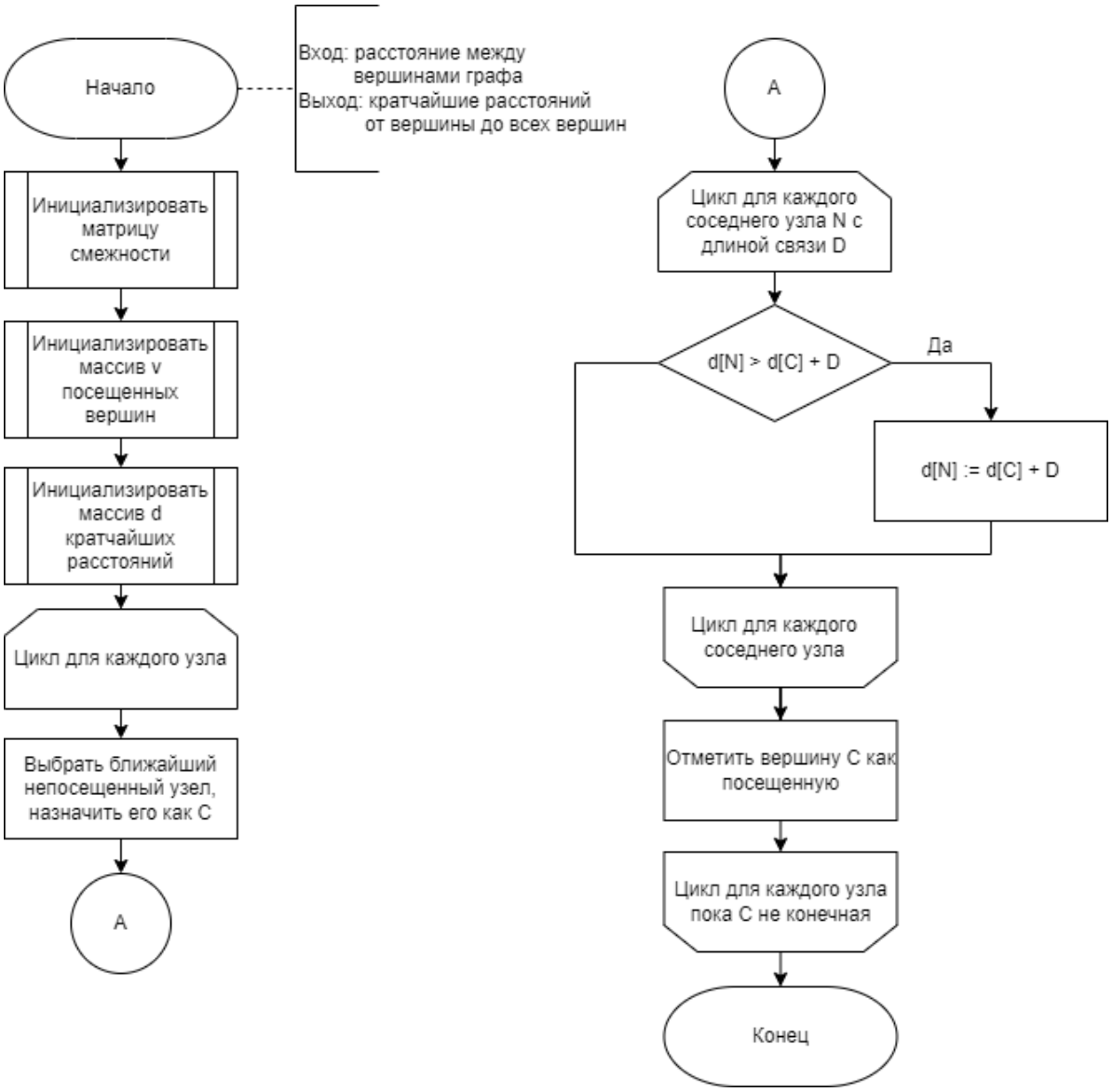

Рисунок 2.1 – Схема алгоритма Дейкстры

Вывод

В данном разделе на основе теоретических данных была построена схема требуемого алгоритма.

3 Технологическая часть

3.1 Требования к программному обеспечению

В программе должна присутствовать возможность обрабатывать линейно и параллельно следующие заявки:

1) генирировать входные данные по заданным вручную параметрам (каждый элемент матрицы высчитывается по формуле $a + / - da$);

2) находить кратчайшие расстояния от нулевой вершины до всех остальных вершин алгоритмом Дейкстры;

3) выводить результат в виде последовательностей номеров городов, расстояние до которых равно k, для всех возможных k.

3.2 Выбор языка программирования

Для реализации программного обеспечения был выбран язык C++ в силу возможности создания потоков с помощью класса std::thread.

3.3 Выбор библиотеки и способа для замера времени

Для замера процессорного времени выполнения реализаций агоритмов была выбрана не стандартная функция библиотеки <time.h> языка C — clock(), которая недостаточно четко работает при замерах небольших промежутков времени, а реализован макрос с параметрами как асемблерная вставка, представленный на листинге 3.1.

Листинг 3.1 – Функция замеров процессорного времени

```
1 #define get_time(time) __asm__ __volatile__("rdtsc" :"=A"(time))
```

В силу существования явления вытеснения процессов из ядра, квантования процессорного времени все процессорное время не отдается какой-либо одной задаче, поэтому для получения точных результатов необходимо усреднить результаты вычислений: замерить совокупное время выполнения реализации алгоритма N раз и вычислить среднее время выполнения.

3.4 Реализации алгоритмов

В листинге 3.2, приведена реализация алгоритма Дейкстры.

Листинг 3.2 – Реализация алгоритма Дейкстры

```
1  void dijkstra(int m[SIZE][SIZE], int w[SIZE], int v[SIZE], int n)
2  {
3      for (int i = 0; i < n; i++)
4          w[i] = INT_MAX, v[i] = 1;
5
6      int minindex, min, begin_index = 0;
7      w[begin_index] = 0;
8
9      do {
10         minindex = min = INT_MAX;
11         for (int i = 0; i < n; i++)
12         {
13             if ((v[i] == 1) && (w[i] < min))
14             {
15                 min = w[i];
16                 minindex = i;
17             }
18         }
19
20         if (minindex != INT_MAX)
21         {
22             for (int i = 0; i < n; i++)
23                 if (m[minindex][i] > 0)
24                     w[i] = get_min(2, w[i], min + m[minindex][i]);
25             v[minindex] = 0;
26         }
27     } while (minindex < INT_MAX);
28 }
```

В листинге 3.3, приведена реализация алгоритмов создания и обработки заявки по стадиям.

Листинг 3.3 – Реализация алгоритмом связанных со стадями обработки заявок

```
typedef struct
{
  int n, a, d;
    int m[SIZE][SIZE];
  int w[SIZE];
  int v[SIZE];
  FILE *f;
} request_t;

typedef struct request_state
{
    bool stage_1;
    bool stage_2;
    bool stage_3;
} request_state_t;

static request_t generate_request(FILE *f, int n, int a, int d)
{
  request_t request;
  request.n = n;
  request.a = a;
  request.d = d;
  request.f = f;
  return request;
}

static void init_request_state_array(std::vector<request_state_t>
   &request_state_array, int num_of_requests)
{
    request_state_array.resize(num_of_requests);
    for (int i = 0; i < num_of_requests; i++)
    {
        request_state_t request_state;
        request_state.stage_1 = false;
        request_state.stage_2 = false;
        request_state.stage_3 = false;
        request_state_array[i] = request_state;
```

```
37      }
38  }
39  void log_data(const request_t *const request)
40  {
41      fprintf(request->f, "Matrix:\n");
42      for (int i = 0; i < request->n; i++)
43      {
44          for (int j = 0; j < request->n; j++)
45              fprintf(request->f, "%5d", request->m[i][j]);
46          fprintf(request->f, "\n");
47      }
48      fprintf(request->f, "\n");
49
50      fprintf(request->f, "Distances:\n");
51      for (int i = 0; i < request->n; i++)
52          fprintf(request->f, "%5d", request->w[i]);
53      fprintf(request->f, "\n");
54
55      bool flag;
56      for (int i = 0; i < request->n; i++)
57      {
58          flag = true;
59          for (int j = i - 1; j >= 0; j--)
60              if (request->w[i] == request->w[j])
61                  flag = false;
62
63          if (flag)
64          {
65              fprintf(request->f, "Distance %d to vertices: %d ",
                    request->w[i], i);
66              for (int j = i + 1; j < request->n; j++)
67                  if (request->w[i] == request->w[j])
68                          fprintf(request->f, "%d ", j);
69              fprintf(request->f, "\n");
70          }
71      }
72      fprintf(request->f, "\n\n");
73  }
74
75  static void stage_1(std::queue<request_t> &q1, std::queue<
        request_t> &q2)
```

```
76 {
77     std::mutex m;
78     m.lock();
79     request_t request = q1.front();
80     m.unlock();
81     init_data(request.m, request.n, request.a, request.d);
82     m.lock();
83     q2.push(request);
84     m.unlock();
85     m.lock();
86     q1.pop();
87     m.unlock();
88 }
89
90 static void stage_2(std::queue<request_t> &q2, std::queue<
       request_t> &q3)
91 {
92     std::mutex m;
93     m.lock();
94     request_t request = q2.front();
95     m.unlock();
96     dijkstra(request.m, request.w, request.v, request.n);
97     m.lock();
98     q3.push(request);
99     m.unlock();
100    m.lock();
101    q2.pop();
102    m.unlock();
103 }
104
105 static void stage_3(std::queue<request_t> &q3, int task_num)
106 {
107    std::mutex m;
108    m.lock();
109    request_t request = q3.front();
110    m.unlock();
111    log_data(&request);
112    m.lock();
113    q3.pop();
114    m.unlock();
115 }
```

В листинге 3.4, приведена реализация алгоритма последовательной обработки данных.

Листинг 3.4 – Реализация алгоритма последовательной обработки данных

```
void linear_processing(int num_of_requests, int n, int a, int d)
{
    size_t time_start = 0, time_end = 0;
    FILE *f = fopen("linear.log", "w");
    std::queue<request_t> q1, q2, q3;
    std::mutex m;
    for (int i = 0; i < num_of_requests; i++)
    {
        request_t request = generate_request(f, n, a, d);
        q1.push(request);
    }
    get_time(time_start);
    for (int i = 0; i < num_of_requests; i++)
    {
        stage_1(std::ref(q1), std::ref(q2));
        stage_2(std::ref(q2), std::ref(q3));
        stage_3(std::ref(q3), i + 1);
    }
    get_time(time_end);
    printf("Time: %zu ticks.\n", time_end - time_start);
    fclose(f);
}
```

В листинге 3.5, приведена реализация алгоритма конвейерной обработки данных.

Листинг 3.5 – Реализация алгоритма конвейерной обработки данных

```
static void parallel_stage_1(std::queue<request_t> &q1, std::queue
    <request_t> &q2,
                        std::vector<request_state_t> &
                            request_state_array,
                        bool &q1_is_empty)
{
    int task_num = 1;
    while(!q1.empty())
    {
        stage_1(std::ref(q1), std::ref(q2));
        request_state_array[task_num - 1].stage_1 = true;
```

```
10          task_num++;
11      }
12      q1_is_empty = true;
13  }
14
15  static void parallel_stage_2(std::queue<request_t> &q2, std::queue
        <request_t> &q3,
16                          std::vector<request_state_t> &
                                request_state_array,
17                          bool &q1_is_empty, bool &q2_is_empty)
18  {
19      int task_num = 1;
20      while(true)
21      {
22          if (!q2.empty())
23          {
24              if (request_state_array[task_num - 1].stage_1 == true)
25              {
26                  stage_2(std::ref(q2), std::ref(q3));
27                  request_state_array[task_num - 1].stage_2 = true;
28                  task_num++;
29              }
30          }
31          else if (q1_is_empty)
32              break;
33      }
34      q2_is_empty = true;
35  }
36
37  static void parallel_stage_3(std::queue<request_t> &q3,
38                          std::vector<request_state_t> &
                                request_state_array,
39                          bool &q2_is_empty)
40  {
41      int task_num = 1;
42      while(true)
43      {
44          if (!q3.empty())
45          {
46              if (request_state_array[task_num - 1].stage_2 == true)
47              {
```

```
48                  stage_3(std::ref(q3), task_num);
49                  request_state_array[task_num - 1].stage_3 = true;
50                  task_num++;
51              }
52          }
53          else if(q2_is_empty)
54              break;
55      }
56  }
57
58  void conveyor_processing(int num_of_requests, int n, int a, int d)
59  {
60      size_t time_start = 0, time_end = 0;
61      FILE *f = fopen("conveyor.log", "w");
62      std::queue<request_t> q1, q2, q3;
63      std::mutex m;
64      for (int i = 0; i < num_of_requests; i++)
65      {
66          request_t request = generate_request(f, n, a, d);
67          q1.push(request);
68      }
69      bool q1_is_empty = false, q2_is_empty = false;
70      std::vector<request_state_t> request_state_array;
71      init_request_state_array(request_state_array, num_of_requests)
            ;
72      std::thread threads[THREADS_COUNT];
73      get_time(time_start);
74      threads[0] = std::thread(parallel_stage_1, std::ref(q1), std::
            ref(q2), std::ref(request_state_array), std::ref(
            q1_is_empty));
75      threads[1] = std::thread(parallel_stage_2, std::ref(q2), std::
            ref(q3), std::ref(request_state_array), std::ref(
            q1_is_empty), std::ref(q2_is_empty));
76      threads[2] = std::thread(parallel_stage_3, std::ref(q3), std::
            ref(request_state_array), std::ref(q2_is_empty));
77      get_time(time_end);
78      for (int i = 0; i < THREADS_COUNT; i++)
79          threads[i].join();
80      printf("Time: %zu ticks.\n", time_end - time_start);
81      fclose(f);
82  }
```

3.5 Тестирование алгоритмов

В таблице 3.1 приведены функциональные тесты для функции, реализующей алгоритм Дейкстры. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Матрица A	Ожидаемый результат
$\begin{pmatrix} 0 \end{pmatrix}$	0
$\begin{pmatrix} 0 & 37 \\ 37 & 0 \end{pmatrix}$	0, 37
$\begin{pmatrix} 0 & 59 & 13 & 14 & 85 \\ 59 & 0 & 98 & 25 & 80 \\ 13 & 98 & 0 & 8 & 92 \\ 14 & 25 & 8 & 0 & 99 \\ 85 & 80 & 92 & 99 & 0 \end{pmatrix}$	0, 39, 13, 14, 85

Вывод

В данном разделе были реализованы алгоритм Дейкстры, алгоритмы последовательной и конвейерной обработки данных, а также проведено функциональное тестирование алгоритма Дейкстры.

4 Экспериментальная часть

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование программного обеспечения:

1) операционная система Windows-10, 64-bit;
2) оперативная память 8 ГБ;
3) процессор Intel(R) Core(TM) i3-7020U CPU @ 2.30GHz, 2304 МГц, ядер 2, логических процессоров 4.

4.2 Замеры времени

В таблице 4.1 приведены результаты замеров в тиках времени работы алгоритмов последовательной и конвейерной обработки для разного количества заявок.

Таблица 4.1 – Замеры времени выполнения алгоритмов последовательной и конвейерной обработки для разного количества заявок

Кол-во заявок	Последовательная обработка	Конвейерная обработка
10	10115424	1838815
20	15957317	578113
30	26678399	1234720
40	22571050	740568
50	30451649	841950
60	34528847	757620
70	34071463	3726378
80	31809828	875277

Зависимость времени работы алгоритмов последовательной и конвейерной обработки для разного количества заявок 4.1.

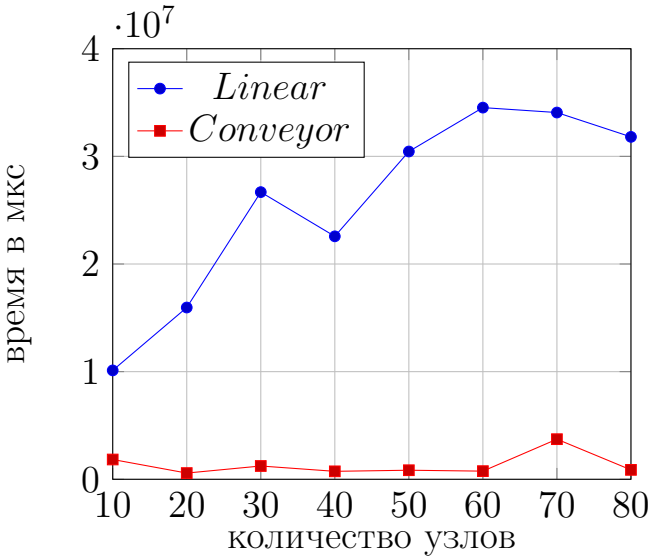

Рисунок 4.1 – Зависимость времени работы алгоритмов последовательной и конвейерной обработки для разного количества заявок

4.3 Файл журнала

В листинге 4.1 показан пример файла журнала, который генерируется программой после обработки заявок.

Листинг 4.1 – Реализация алгоритма Дейкстры

```
Matrix:
    0    1    7    4    0    9    4    8    8    2
    1    0    4    5    5    1    7    1    1    5
    7    4    0    2    7    6    1    4    2    3
    4    5    2    0    2    2    1    6    8    5
    0    5    7    2    0    7    6    1    8    9
    9    1    6    2    7    0    2    7    9    5
    4    7    1    1    6    2    0    4    3    1
    8    1    4    6    1    7    4    0    2    3
    8    1    2    8    8    9    3    2    0    3
    2    5    3    5    9    5    1    3    3    0

Distances:
    0    1    4    4    3    2    3    2    2    2
Distance 0 to vertices: 0
Distance 1 to vertices: 1
Distance 4 to vertices: 2 3
Distance 3 to vertices: 4 6
Distance 2 to vertices: 5 7 8 9


Matrix:
    0    4    1    1    3    8    7    4    2    7
    4    0    7    9    3    1    9    8    6    5
    1    7    0    0    2    8    6    0    2    4
    1    9    0    0    8    6    5    0    9    0
    3    3    2    8    0    0    6    1    3    8
    8    1    8    6    0    0    9    3    4    4
    7    9    6    5    6    9    0    6    0    6
    4    8    0    0    1    3    6    0    6    1
    2    6    2    9    3    4    0    6    0    8
    7    5    4    0    8    4    6    1    8    0

Distances:
    0    4    1    1    3    5    6    4    2    5
Distance 0 to vertices: 0
Distance 4 to vertices: 1 7
Distance 1 to vertices: 2 3
Distance 3 to vertices: 4
Distance 5 to vertices: 5 9
```

```
Distance 6 to vertices: 6
Distance 2 to vertices: 8


Matrix:
    0    4    9    6    3    7    8    8    2    9
    4    0    1    3    5    9    8    4    0    7
    9    1    0    6    3    6    1    5    4    2
    6    3    6    0    0    9    7    3    7    2
    3    5    3    0    0    6    0    1    6    5
    7    9    6    9    6    0    7    5    4    1
    8    8    1    7    0    7    0    2    0    0
    8    4    5    3    1    5    2    0    1    4
    2    0    4    7    6    4    0    1    0    6
    9    7    2    2    5    1    0    4    6    0

Distances:
    0    4    5    6    3    6    5    3    2    7
Distance 0 to vertices: 0
Distance 4 to vertices: 1
Distance 5 to vertices: 2 6
Distance 6 to vertices: 3 5
Distance 3 to vertices: 4 7
Distance 2 to vertices: 8
Distance 7 to vertices: 9
```

Вывод

В данном разделе был проведен эксперимент по измерению времени работы линейной и конвейерной обработки заявок. Согласно полученным при проведении эксперимента данным, конвейерная обработка данных работает быстрее при всех рассматренных условиях.

Заключение

Цель работы достигнута: изучены принципы конвейерной обработки данных и реализован алгоритм Дейкстры с генерацией данных по заданным параметрам и выводом результата в определенном формате.

Все **задачи** лабораторной работы решены:

- изучено понятие конвейерной обработки данных;
- реализован алгоритм Дейкстры;
- реализована последовательная обработка данных на основе алгоритм Дейкстры;
- реализована конвейерная обработка данных на основе алгоритм Дейкстры;
- проведен сравнительный анализ времени работы линейной и конвейерной обработки данных на основе экспериментальных данных.

На основании проведенных экспериментов было определено, что конвейерная обработка данных работает быстрее в среднем в 50 раз.

Список использованных источников

[1] Конвейерная обработка [Электронный ресурс]. URL: https://opengl.org.ru/informatsionnye-sistemy-i-tekhnologii-v-zkonomike/konveiernaya-obrabotka.html (дата обращения: 13.12.2022).

[2] Лебедев Сергей Сергеевич, Новиков Федор Александрович Необходимое и достаточное условие применимости алгоритма Дейкстры // КИО. 2017. №4. URL: https://cyberleninka.ru/article/n/neobhodimoe-i-dostatochnoe-uslovie-primenimosti-algoritma-deykstry (дата обращения: 06.12.2022).