МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритмы поиска пути в графах

Студент гр. 9382 _____ Иерусалимов Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с алгоритмами по поиску пути в графе. Получить навыки решения задач на такие алгоритмы.

Задание.

Вар. 5. Реализовать алгоритм Дейкстры поиска пути в графе (на основе кода A*).

Задача на жадный алгоритм:

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет abcde

Задача на алгоритм A*:

Разработайте программу, которая решает задачу построения кратчайшего пути в *ориентированном* графе **методом A***. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

ade

Описание алгоритмов.

Жадный алгоритм.

Начинается со стартовой вершины, смотрятся соединённые вершины к текущей. Среди них выбирается та, вес которой наименьший среди остальных соединенных вершин. Эта выбранная вершина кладется в стек, а у вершины которой смотрели соседей устанавливается флаг на использованную. Далее все

повторяется с новой вершиной в стеке. Когда все соседние вершины пройдены включая текущую, то надо вернуться на одну вершину назад по стеку. Как только текущая вершина будет равняться конечной и все соседи рассмотрены, то алгоритм завершается.

Сложность:

На каждой итерации перебираются все соседние вершины, из этого выходит что сложность по времени будет O(I * J). Где J – количество соседних вершин, I – количество вершин.

Граф храниться в векторе структур. Где в каждой структуре присутствует имя графа и вектор ребер. Тогда сложность по памяти будет O(I + J) Где J – количество ребер, I – количество вершин.

Алгоритм A*.

На каждом шаге выбирается вершина с наименьшим приоритетом. Функция для оценки приоритета состоит из двух слагаемых: текущего расстояния от начальной вершины и эвристической функции(в данной задаче это разница между ASCII кодами символов или в случае целых чисел просто их разница по модулю). Затем для выбранной вершины рассматриваются смежные ей вершины. Для каждой смежной вершины проверяется ее кратчайший путь до начальной вершины, и если текущий путь короче, то заменяется на него. После этого эта смежная вершина помещается в очередь с приоритетом, где значение приоритетности определяется с помощью функции оценки приоритетности, определенной выше. Алгоритм заканчивает работу, как только из очереди вытащится конечная вершина.

Сложность:

В лучшем случае, когда эвристическая функция позволяет делать каждый шаг в верном направлении, т.е. наиболее подходящая функция тогда сложность по времени составляет O(I + J) Где J – количество ребер, I – количество

вершин.

В худшем случае, эвристическая функция угадывает направление в последний момент, тогда надо проходить все возможные пути. Тогда время работы будет расти экспоненциально по сравнению с длиной оптимального пути.

Оценка по памяти будет $O(I * (I + J))$, где I – количество вершин, J – количество рёбер в графе. Так как в худшем случае мы будем хранить все пути в очереди, то и сложно по памяти будет экспоненциальной.

Алгоритм Дейкстры.

Работает точно так же как и A* изменились только компараторы. Присутствует шаблонная функция которая принимает разные компараторы, и в зависимости от этого меняется работа. Теперь мы не смотрим на дальность символов между кодов ASCII, мы просто проходимся по всем ребрам. Каждой вершине назначается сумма, за которую до нее можно добраться. С помощью нового компаратора сравниваем оптимальная ли сумма у вершины. Новый компаратор просто сравнивает больше ли текущее ребро предыдущего, если текущее ребро больше то мы не меняем сумму у вершины, если нет, то ставим новую сумму. Алгоритм заканчивает работу, когда из очереди вытаскивается конечный элемент.

Сложность:

В этом алгоритме асимптотика работы зависит от реализации.

Разделяют три случая реализации.

1)Наивная реализация. - n раз осуществляем поиск вершины с минимальной величиной d среди O(n) непомеченных вершин и m раз проводим релаксацию за O(1). И тогда скорость будет O($n^2 + m$).

2) Двоичная куча - Используя двоичную кучу можно выполнять операции

извлечения минимума и обновления элемента за O(log n). Тогда время работы алгоритма Дейкстры составит O(n * log n + m * log n)=O($m * \log n$).

3) Фибоначчиева куча - Используя Фибоначчиевы кучи можно выполнять операции извлечения минимума за O(log n) и обновления элемента за O(1). Таким образом, время работы алгоритма составит O(n * log n + m).

Мы используем наивную реализацию.

Так как мы проходимся по всем вершинам и их соседям тогда скорость будет O(I * J). Где J – количество ребер, I – количество вершин. Еще прибавим n к этой сложности так как, когда мы восстанавливаем путь и даем конечный ответ нам надо пройти расстояние от конечной вершины и до, начальной. Тогда сложность по времени будет O(I * J + n), где n – количество узлов между вершинами.

Сложность по памяти будет такая же как и в худшем случае A*.

Описание функций и структур данных.

struct Edge – Структура ребер, хранит длину и имя к какому узлу подключена.

struct Vertex – Структура вершин, хранит имя, использована ли вершина, эвристическую сумму, сумму пути, указатель на прошлую вершину и вектор ребер.

class Graph – класс графов, отвечает за правильное построение графа.

void AddVertex(char name) – Добавляет вершину в граф. Аргументом принимает имя вершины

void AddEdge(char vertex1, char vertex2, int mass) – Добавляет ребро. Первый аргумент имя вершины отправки, второй имя вершины получателя, третий

вес ребра

Vertex *FindVertex(char name) – Находит вершину. Аргумент - имя искомой вершины. Возвращает указатель на вершину.

class GreedyAlgorithm – Класс жадного алгоритма.

GreedyAlgorithm(Graph *a) – Конструктор. Аргумент указатель на граф.

void getShortestPath(char vertex1, char vertex2) – Решение жадного алгоритма. Аргументы, соответственно: начальная вершина, конечная.

class Dijkstra_and_Astar – Класс по решение с помощью Дейкстры и A*

Dijkstra_and_Astar (Graph *a) - Конструктор. Аргумент указатель на граф.

priority_queue<Vertex *, std::vector<Vertex *>, T> quequeAstar - очередь в A* алгоритме. Оценочная функция f = g + h, где g кратчайшее расстояние до этой вершины и h – эвристика. Также для нее был написан специальный компаратор, который справляется с правильным определением приоритета.

template<typename T>
void AstarOrDijkstra(char start, char end) – Шаблонная функция для поиска пути с помощью A* или Дейкстры. T – имя типа компаратора. Char start – название вершины с которой надо начать поиск. Char end – название вершины до которой надо дойти.

struct CmpAstar – структура с компараторами для алгоритма A*
struct CmpDijkstra - структура с компараторами для алгоритма Дейкстры

Тестирование.

Тестирование жадного алгоритма.

№	Входные данные	Выходные данные
1	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	abdefg
2	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe
3	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 f g 1.0	abdefg

4	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	abcd
5	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	abdefg
6	a d a b 1.0 b c 1.0 c a 1.0 a d 8.0	ad
7	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0	abcd

	e d 3.0	
8	a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf
9	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge

Тестирование алгоритма Дейкстры.

№	Входные данные	Выходные данные
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade
2	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	ag
3	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	ag

4	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe
5	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge
6	a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf

7	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	aed
8	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	ag

Тестирование алгоритма A*.

№	Входные данные	Выходные данные
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade

2	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	ag
3	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	ag
4	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe

5	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge
6	a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf
7	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	ag

Выводы.

Были получены навыки решения задач связанные с алгоритмами по поиску пути в графе.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <queue>


int PRINT = 0;

using namespace std;


struct Edge {
    char name;
    int mass;

    Edge(char name, int mass) {
        this->name = name;
        this->mass = mass;

    }


};


struct Vertex {
    Vertex(char name) {
        this->name = name;
        used = false;
        prev = nullptr;
        sum = 0;
        heuristic = 0;
    }

    double sum;
    double heuristic;
    char name;
    bool used;
    Vertex *prev;
    vector<Edge *> edge;

};

struct CmpAstar {
    bool operator()(Vertex *v1, Vertex *v2) {//for queue
        if (v1->heuristic == v2->heuristic) {
            return v2->name > v1->name;
        }
```

```
        return (v2->heuristic) < (v1->heuristic);
    }

    bool cmpAstar(Vertex *num1, double num2) {//for nums
        return num1->heuristic > num2;
    }
};

struct CmpDijkstra {
    bool operator()(Vertex *v1, Vertex *v2) { //for queue
        return (v2->sum) < (v1->sum);
    }

    bool cmpDijkstra(Vertex *num1, double num2) { //for nums
        return num1->sum > num2;
    }
};


class Graph {
public:


    void AddVertex(char name) {
        if (FindVertex(name) == nullptr) {
            Vertex currVertextex(name);
            graph.push_back(currVertextex);
        }

    }

    void AddEdge(char vertex1, char vertex2, int mass) {
        FindVertex(vertex1)->edge.push_back(new Edge(vertex2, mass));
        // FindVertex(vertex2)->edge.push_back(new Edge(vertex1, mass));
    }

    Vertex *FindVertex(char name) {
        for (int i = 0; i < graph.size(); ++i) {
            if (graph[i].name == name) {
                return &graph[i];
            }
        }
        return nullptr;
    }

    vector<Vertex> graph;
};


class GreedyAlgorithm {
private:
    Graph *solve;
    vector<Vertex *> result;
```

```
public:

   GreedyAlgorithm(Graph *a) {
      solve = a;
   }

   void getShortestPath(char vertex1, char vertex2) {
      if (PRINT) {
         cout << "\t\t\tGreedy Algorithm Start!";
      }
      int minSize;
      char cur = vertex1;
      Vertex *currVertex;
      result.push_back(FindVertex(vertex1));
      currVertex = FindVertex(cur);
      int i = 0;
      while (cur != vertex2) {
         if (PRINT) {
            cout << "\n\n_____________________________________" << i
               << "_____________________________________\n\t\tCurrent Vertex != answer. \n\t\t\t"
<< cur << " != "
               << vertex2 << "\n\t\t\tcontinue...\n\n";
         }

         ++i;
         bool found = false;
         //result.push_back(FindVertex(cur));
         Vertex *nextVert;
         minSize = INT32_MAX;
         if (PRINT) {
            cout << "\tWe look at the neighboring vertices at the current node - " << cur << "\n";
            cout << "\tAnd choose the smallest path.\n";
         }


         for (int i = 0; i < currVertex->edge.size(); ++i) {
            if (PRINT) {
               cout << "\tAdjacent vertex: " << cur << " -> " << currVertex->edge[i]->name << "\n";

            }
            if (!FindVertex(currVertex->edge[i]->name)->used && currVertex->edge[i]->mass <
minSize) {

               minSize = currVertex->edge[i]->mass;
               nextVert = FindVertex(currVertex->edge[i]->name);
               found = true;
               if (PRINT) {
                  cout << "\t\tThe edge is minimal " << minSize << ", we keep the path.";
                  cout << "\t\tTemp answer is: " << cur << " -> " << currVertex->edge[i]->name << "\n";
               }

            } else if (FindVertex(currVertex->edge[i]->name)->used) {
               if (PRINT) {
                  cout << "\t\tThe neighbor has already been visited.\n";
```

```
            }
         } else {
            if (PRINT) {
               cout << "\t\tCurrent weight is less than new " << minSize << " < " << currVertex-
>edge[i]->mass
                     << "\n";
            }
         }

      }


      if (!(currVertex->name == vertex1)) {
         currVertex->used = true;
      }


      if (!found) {

         if (!result.empty()) {
            result.pop_back();
            currVertex = result.back();
            cur = currVertex->name;
         }
         if (PRINT) {
            cout << "\nAll neighboring vertices have been visited, we return to the previous vertex - "
                 << currVertex->name << ".\n";
         }
         continue;
      }
      if (PRINT) {
         cout << "\nAdd vertex \"" << nextVert->name << "\" to the response stack with minimum
weight \""
              << minSize
              << "\"\n";
      }

      currVertex = nextVert;
      cur = currVertex->name;
      result.push_back(currVertex);
      if (PRINT) {
         cout << "Answer in the curren elment is : ";


         for (int i = 0; i < result.size(); ++i) {
            cout << result[i]->name << "->";
         }
      }
   }


   if (PRINT) {
      cout
           << "\nWe reached the final vertices. \nCompleting the
```

```
algorithm.\n\n//////////////////////////////////////////////////////////////////////////////\n";
        }
        cout << "Shortest path using greedy algorithm:\n";


        for (int i = 0; i < result.size(); ++i) {
            cout << result[i]->name;
        }
        return;

    }


    Vertex *FindVertex(char nameVer) {
        for (int i = 0; i < solve->graph.size(); ++i) {
            if (nameVer == solve->graph[i].name) {
                return &solve->graph[i];
            }
        }
        cout << "Can't find vertex - " << nameVer << "\n";
        exit(-1);
    }


};


class Dijkstra_and_Astar {
private:
    void RestorePath(const char start, char end) {
        string path(1, end);
        while (end != start) {
            end = graph->FindVertex(end)->prev->name;
            path = string(1, end) + "->" +path;
        }
        cout << path << "\n";
        cout<< "Answer is :"<<path<<"\n";
    }

    vector<char> answ;
    Graph *graph;
public:
    Dijkstra_and_Astar(Graph *a) {
        this->graph = a;
    }

    template<typename T>
    void printQueue(priority_queue<Vertex *, std::vector<Vertex *>, T> queue) {
        while (!queue.empty()) {
            cout << "{ " << queue.top()->name << ", " << queue.top()->sum << "}";
            queue.pop();
        }
    }
```

```
template<typename T>
void AstarOrDijkstra(char start, char end) {
    priority_queue<Vertex *, std::vector<Vertex *>, T> quequeAstar;

    quequeAstar.push(graph->FindVertex(start));

    Vertex *current;
    Vertex *temp;
    int newSum;
    double tempHeuristic;
    bool choiseAlgorithm = 1;
    CmpDijkstra cmp;
    CmpAstar cmpA;
    if (typeid(T) == typeid(CmpDijkstra)) {
        choiseAlgorithm = 0;

            cout << "Using Dijkstra algorithm\n_____\n";
            cout << "\tCalculate the summ of path for each neighbor\n\n";

    } else {

            cout << "Using A* algorithm\n_____\n";
            cout << "\tCalculate the heuristic function for each neighbor\n\n";

    }
    do {

        current = quequeAstar.top();

        if (PRINT) {
            cout << "////////////////////////////////////////////////////////////////\n";
            cout << "\n\tCurrent queue ( ";
            printQueue<T>(quequeAstar);
            cout << " )\n";
            cout << "\tWe took the vertices out of the queue" << "{ " << quequeAstar.top()->name << ",
"
                << quequeAstar.top()->sum << "}" << " and put it in a temporary variable\n";
        }

        quequeAstar.pop();
        current->used = true;

        if (PRINT) {
            cout << "\tWe go through all the neighbors of the vertex  \'" << current->name << "\'\n";
            if (current->edge.empty() && current->name != end) {
                cout << "Vertex \'" << current->name << "\' has no neighbors\nTake the next item in the
queue\n\n";

            }
        }

        for (auto &neighbour : current->edge) {

            tempHeuristic = current->sum + neighbour->mass + (end - neighbour->name);
```

```
            temp = graph->FindVertex(neighbour->name);
            if (PRINT) {
               cout << "\t\t_____" << current->name << " -> " << neighbour->name
                    << "_____\n";
               if (choiseAlgorithm) {
                  cout << "\t\t\tCalculate the heuristic function f = g + h = " << current->sum + neighbour->mass
                       << " + " << (end - neighbour->name) << " = " << tempHeuristic << "\n";
               } else {
                  cout << "\t\t\tcalculate the sum for \"" << neighbour->name << "\', summ = current + next = "
                       << current->sum << " + " << neighbour->mass << " = " << current->sum + neighbour->mass
                       << "\n";
               }

            }

            if (choiseAlgorithm ? cmpA.cmpAstar(temp, tempHeuristic) : cmp.cmpDijkstra(temp, neighbour->mass)) {
               if (PRINT) {
                  if (choiseAlgorithm) {
                     cout << "\t\t\tThe current  heuristic function is less than the previous one ("
                          << temp->name << "), write a new value\n\t\t\t" << temp->heuristic << " > "
                          << tempHeuristic << " ; " << temp->name << " = " << tempHeuristic << "\n";
                  } else {
                     cout << "\t\t\tThe current summ is less than the previous one (" << temp->name
                          << "), write a new value\n\t\t\t" << temp->sum << " > "
                          << current->sum + neighbour->mass << " ; " << temp->name << " = "
                          << current->sum + neighbour->mass << "\n";
                  }
               }
               temp->heuristic = tempHeuristic;
               temp->sum = current->sum + neighbour->mass;
               temp->prev = current;
               answ.push_back(temp->prev->name);
               if (PRINT) {

                  cout << "\t\t\tAdd prev vertex in parent ";
                  while (temp->prev != nullptr) {
                     cout << temp->prev->name << "->";
                     temp = temp->prev;
                  }
                  cout << "\n";

               }

            } else if (choiseAlgorithm ? temp->heuristic == 0 : temp->sum == 0) {

               temp->heuristic = tempHeuristic;
               temp->sum = current->sum + neighbour->mass;
               temp->prev = current;
               answ.push_back(current->name);
```

```
                queqeAstar.push(temp);
                answ.push_back(temp->prev->name);
                if (PRINT) {
                    if (choiseAlgorithm) {
                        cout << "\t\t\tHeuristic function for \'" << temp->name << "\' = " << temp->heuristic
                            << " \n";
                    } else {
                        cout << "\t\t\tThe sum of paths for \'" << temp->name << "\' = " << temp->sum << "
\n";
                    }
                    cout << "\t\t\tAdd in queue \'" << temp->name << "\'. (";
                    printQueue<T>(queqeAstar);
                    cout << " )\n\n";

                    cout << "\t\t\tAdd prev vertex in parent ";
                    while (temp->prev != nullptr && temp->name != start) {
                        cout << temp->prev->name << "->";
                        temp = temp->prev;
                    }
                    cout << "\n";

                }
            }
        }
        if(PRINT){
            if(current->name != end)
                cout<<"\n\t\t\t\t" << current->name << " != " << end << "\n \t\t\t\tContinue...\n\n";
            else  cout<<"\n\t\t\t\t" << current->name << " = " << end << "\n \t\t\t\tEnd\n\n";
        }


    } while (current->name != end);


    RestorePath(start, end);
    return;
  }
};


int main() {
  Graph a;
  string length;
  char start, end;
  char mainVertex, secondVertex;

  int i = 0;
  int choise = 1;
  int b;
  cout << "enable Intermediate data? 1 - Yes 0 - No\n";
  cin >> b;
  PRINT = b;
  //if (PRINT) {
    cout << "Gready Algoritm - 0, Astar - 1, Dijkstra - 2\n";
```

```
    //}
    cin >> choise;
    if (PRINT) {
        cout << "Input data with ')' on end: \n";
    }
    cin >> start >> end;
    while (mainVertex != ')' && cin >> mainVertex) {

        if (mainVertex != ')') {
            cin >> secondVertex >> length;
            a.AddVertex(mainVertex);
            a.AddVertex(secondVertex);
            a.AddEdge(mainVertex, secondVertex, stoi(length));
        }

    }

    if (choise == 0) {
        GreedyAlgorithm *some = new GreedyAlgorithm(&a);

        some->getShortestPath(start, end);
    } else if (choise == 1) {
        Dijkstra_and_Astar *some = new Dijkstra_and_Astar(&a);
        //cout << "Shortest Path using Dijkstra algorithm:\n";
        // cout << some->getShortestPath(start, end);
        some->AstarOrDijkstra<CmpAstar>(start, end);
    } else {
        Dijkstra_and_Astar *some = new Dijkstra_and_Astar(&a);
        some->AstarOrDijkstra<CmpDijkstra>(start, end);
    }

    system("pause>nul");
    return 0;
}
```