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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Жадный алгоритм и A*

Студент гр. 9383 _____ Крейсманн К.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритмы поиска пути в ориентированном графе (A* и жадный). Написать программы, реализующие алгоритмы.

Задание.

1) 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.

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.

2) Алгоритм A*. (Вариант 4)

Модификация A* с двумя финишами: требуется найти путь до каждого, а затем найти мин. путь между ними, при этом начальная вершина не должна находиться в окончательном ответе.

Теория

Жадный алгоритм – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Алгоритм A* - один из самых популярных методов решения задач на поиск кратчайшего пути. Ищет первое наилучшее совпадение на графе,

который находит маршрут с наименьшей стоимостью от одной вершины к другой.

Описание алгоритма

Жадный алгоритм:

Каждая вершина хранит информацию о смежных вершинах в виде двоичной кучи, чтобы быстро можно было выбрать вершину, ребро к которой имеет наименьший вес. Алгоритм, начиная со стартовой вершины проходит по ребрам с наименьшим весом, пока не найдет финишную вершину. Если алгоритм заходит в тупик, то происходит возврат к предыдущей вершине. Первый найденный путь от старта к финишу будет ответом.

Модифицированный A*:

Все вершины хранятся в двоичной куче. Каждая вершина имеет метку, равную текущему расстоянию от стартовой вершины к данной + значение эвристической функции. Изначально стартовой вершине присваивается метка равная эвристической функции, а остальным вершинам – максимальное значение типа double.

Сначала ищется путь от стартовой вершины к двум финишным. Алгоритм каждый раз берет вершину с минимальной меткой, удаляет ее из двоичной кучи (удаляет из списка нерассмотренных вершин) и совершает пересчет меток, смежных с текущей. Если на каком-то шаге вершин с наименьшей меткой является одной из финишных, то значит путь до нее найден, он сохраняется. Когда оба пути найдены алгоритм завершает свою работу. Если на каком-то шаге из кучи достается вершина с меткой равной максимальному значению типа double, то это означает, что один или оба пути не найдены (они не существуют).

Затем ищется 2 пути от первого финиша ко второму и обратно, при этом в пути не должно быть стартовой вершины. Пути ищутся аналогично, только если при обновлении меток среди смежных вершин попадается начальная (которой не должно быть в пути) , она пропускается, т.е. через нее алгоритм никогда не пройдет.

Сложность

Временная сложность модифицированного алгоритма A* - O(n*log(n)). Так как алгоритм проходит по всем вершинам за O(n), достает элемент и перестраивает кучу за O(log(n)), обновляет соседние вершины за O(m) (m – количество смежных вершин).

Описание основных функций и структур данных

Graph – граф, представляет собой вектор вершин.

Link – пара (вершина, число) – описывает смежную вершину и расстояние до нее.

Links – вектор link-ов.

Path – путь в виде двусторонней очереди (в жадном алгоритме) или стека (в A*).

Vertex – структура вершины, содержит имя, вектор смежных вершин, указатель на предыдущую вершину в пути, метку.

checkLinksInput(),checkStartFinishInput() – функции, проверяющие корректность входных данных.

heuristic() – эвристическая функция.

findVertexByName() – поиск индекса вершины в графе.

initGraph() – инициализация графа.

findTwoPaths() – функция для модифицированного A*, ищет путь до двух вершин.

findPathWithoutOneVertex() – функция для модифицированного A*, ищет путь между двумя вершинами, не учитывая указанную вершину.

findPath() – ищет путь жадным алгоритмом.

freeMemory() – освобождение памяти.

printPath() – выводит путь.

clearMarksAndPrev() – восстанавливает на исходные значения меток и предыдущих вершин в графе.

printAnswer()- выводит ответ (в A*).

Тестирование модифицированного A*

Входные данные	Выходные данные
a b d a b 2 a c 1 a c 3 d a 1 b e 3 e d 1 a f 1 f e 1	The path between start and finish1:ab The path between start and finish2:afed The path between finish1 and finish2:bed The path between finish2 and finish1:could not be found
a b c a e 3 e c 3 e m 1 m c 1 a l 4 l c 1 l b 3 b a 3 c a 2	The path between start and finish1:alb The path between start and finish2:aec The path between finish1 and finish2:could not be found The path between finish2 and finish1:could not be found
a b c a b 2 a c 3 a d 1 d c 1 c b 2 b l 1 l c 2	The path between start and finish1:ab The path between start and finish2:adc The path between finish1 and finish2:blc The path between finish2 and finish1:cb

b c 10	

Таблица 1 – Тестирование 1

Тестирование жадного алгоритма

Входные данные	Выходные данные
a z a b 10 a c 2 c g 3 g m 1 b l 10 l z 2 l n 1 n z 100	ablnz
a h a h 2 a e 2 e h 1 a b 1 b c 100 c d 1000 d e 10000	abcdeh
a z a b 1 b c 3 c a 4 a l 3 l m 8 g z 10 z k 12	the path does not exist

z a 9	

Вывод.

Произведено знакомство с жадным алгоритмом и алгоритмом A* для поиска путей в графе. Написаны программы реализующие алгоритмы.

Приложение А

A- star/main.cpp

```
#include "astar.hpp"

int main()
{
    Vertex* start, * finish1, * finish2;
    std::ifstream file("Input.txt");
    Graph graph = initGraph(file, &start, &finish1, &finish2);

    file.close();

    PairPaths paths = findTwoPaths(graph, start, finish1, finish2);

    std::optional<Path> path1to2;
    std::optional<Path> path2to1;
    if (paths.first && paths.second)
    {
        clearMarksAndPrev(graph);
        path1to2 = findPathWithoutOneVertex(graph, finish1, finish2, start);
        clearMarksAndPrev(graph);
        path2to1 = findPathWithoutOneVertex(graph, finish2, finish1, start);
    }

    printAnswer(paths.first, paths.second, path1to2, path2to1, std::cout);
    freeMemory(graph);
    return 0;
}
```

A- star/astar.hpp

```
#pragma once

#include <iostream>
#include <algorithm>
#include <vector>
#include <cctype>
```

```
#include <fstream>
#include <optional>
#include <stack>
#include <cfloat>

struct Vertex;
using Link = std::pair<Vertex*, double>;
using Links = std::vector<Link>;
using Graph = std::vector<Vertex*>;
using Path = std::stack<char>;
using PairPaths = std::pair<std::optional<Path>, std::optional<Path>>;
struct Vertex
{
   Vertex(char name = ' ') :name(name) {}
   char name;
   Links linksHeap;
   Vertex* prev = nullptr;
   double mark = DBL_MAX;
};

int heuristic(char a, char b);
bool checkLinksInput(char nameOut, char nameIn, double weight);
bool checkStartFinishInput(char start, char finish1, char finish2);
int findVertexIndexByName(const Graph& graph, char name);
Graph initGraph(std::istream& in, Vertex** start, Vertex** finish1, Vertex**
finish2);
PairPaths findTwoPaths(Graph graph, Vertex* start, Vertex* finish1, Vertex*
finish2);
std::optional<Path> findPathWithoutOneVertex(Graph graph, Vertex* start, Vertex*
finish, Vertex* noneVertex);
void freeMemory(Graph graph);
void printPath(std::optional<Path> path, std::ostream& out);
void clearMarksAndPrev(Graph& graph);
```

```
void printAnswer(std::optional<Path> pathStartFinish1, std::optional<Path> pathStartFinish2, std::optional<Path> pathFinish1Finish2, std::optional<Path> pathFinish2Finish1, std::ostream& out);
```

A- star/astar.cpp

```
#include "astar.hpp"

auto vertexComparator = [](Vertex* vertex1, Vertex* vertex2)
{
   return (vertex1->mark) == (vertex2->mark) ? vertex1->name<vertex2->name : vertex1->mark>vertex2->mark;
};

int heuristic(char a, char b)
{
   return abs(a - b);
}

bool checkLinksInput(char nameOut, char nameIn, double weight)
{
   return std::isalpha(nameOut) && isalpha(nameIn) && weight >= 0;
}

bool checkStartFinishInput(char start, char finish1, char finish2)
{
   return isalpha(start) && isalpha(finish1) && isalpha(finish2) && start != finish1 && start != finish2 && finish2 != finish1;
}

int findVertexIndexByName(const Graph& graph, char name)
{
   for (int i = 0; i < graph.size(); i++)
   {
      if (graph[i]->name == name)
      {
```

```
            return i;
        }
    }
    return -1;
}

Graph initGraph(std::istream& in, Vertex** start, Vertex** finish1, Vertex** finish2)
{
    Graph graph;
    char nameVertexOut, nameVertexIn;
    double weight;
    char nameStart, nameFinish1, nameFinish2;

    while (in >> nameStart >> nameFinish1 >> nameFinish2)
    {
        if (checkStartFinishInput(nameStart, nameFinish1, nameFinish2))
        {
            break;
        }
        std::cerr << "Error input" << '\n';
    }
    *start = new Vertex(nameStart);
    *finish1 = new Vertex(nameFinish1);
    *finish2 = new Vertex(nameFinish2);
    graph.push_back(*start);
    graph.push_back(*finish1);
    graph.push_back(*finish2);


    while (in >> nameVertexOut >> nameVertexIn >> weight)
    {
        if (!checkLinksInput(nameVertexOut, nameVertexIn, weight))
        {
            std::cerr << "Error input!" << '\n';
```

```
            continue;
        }
        Vertex* vertexOut = new Vertex(nameVertexOut);
        Vertex* vertexIn = new Vertex(nameVertexIn);

        int indexOut = findVertexIndexByName(graph, nameVertexOut);
        if (indexOut == -1)
        {
            graph.push_back(vertexOut);
            indexOut = findVertexIndexByName(graph, nameVertexOut);
        }

        int indexIn = findVertexIndexByName(graph, nameVertexIn);
        if (indexIn == -1)
        {
            graph.push_back(vertexIn);
            indexIn = findVertexIndexByName(graph, nameVertexIn);
        }
        graph[indexOut]->linksHeap.push_back(Link(graph[indexIn], weight));
    }
    return graph;
}

PairPaths findTwoPaths(Graph graph, Vertex* start, Vertex* finish1, Vertex*
finish2)
{
    Path pathStartFinish1, pathStartFinish2;
    start->mark = heuristic(start->name, (finish1->name + finish2->name) / 2);
    while (!graph.empty())
    {
        std::make_heap(graph.begin(), graph.end(), vertexComparator);
        Vertex* current = graph.front();
        if (current->mark == DBL_MAX)
        {
            break;
```

```
        }
        if (current == finish1 )
        {
            Vertex* temp = current;
            while (temp != start)
            {
                pathStartFinish1.push(temp->name);
                temp = temp->prev;
            }
            pathStartFinish1.push(temp->name);
            if (!pathStartFinish2.empty())
            {
                break;
            }
        }
        if (current == finish2)
        {
            Vertex* temp = current;
            while (temp != start)
            {
                pathStartFinish2.push(temp->name);
                temp = temp->prev;
            }
            pathStartFinish2.push(temp->name);
            if (!pathStartFinish1.empty())
            {
                break;
            }
        }
        for (auto i : current->linksHeap)
        {
            if (i.first->mark > current->mark - heuristic(current->name, (finish1->name +
finish2->name) / 2) + i.second)
            {
```

```
            i.first->mark = current->mark - heuristic(current->name, (finish1->name + finish2->name) / 2) + heuristic(i.first->name, (finish1->name + finish2->name) / 2) + i.second;
            i.first->prev = current;
          }
        }
        std::pop_heap(graph.begin(), graph.end(), vertexComparator);
        graph.pop_back();
    }
    if (pathStartFinish1.empty() && pathStartFinish2.empty())
    {
        return PairPaths(std::nullopt, std::nullopt);
    }
    if (pathStartFinish1.empty() && !pathStartFinish2.empty())
    {
        return PairPaths(std::nullopt, pathStartFinish2);
    }
    if (!pathStartFinish1.empty() && pathStartFinish2.empty())
    {
        return PairPaths(pathStartFinish1, std::nullopt);
    }
    return PairPaths(pathStartFinish1, pathStartFinish2);
}

std::optional<Path> findPathWithoutOneVertex(Graph graph, Vertex* start, Vertex* finish, Vertex* noneVertex)
{
    Path path;
    start->mark = heuristic(start->name, finish->name);
    while (!graph.empty())
    {
        std::make_heap(graph.begin(), graph.end(), vertexComparator);
        Vertex* current = graph.front();
        if (current->mark == DBL_MAX)
        {
```

```
            return std::nullopt;
        }
        if (current == finish)
        {
            while (current != start)
            {
                path.push(current->name);
                current = current->prev;
            }
            path.push(current->name);
            break;
        }
        if (current == noneVertex)
        {
            std::pop_heap(graph.begin(), graph.end(), vertexComparator);
            graph.pop_back();
            continue;
        }
        for (auto i : current->linksHeap)
        {
            if (i.first == noneVertex)
            {
                continue;
            }
            if (i.first->mark > current->mark - heuristic(current->name, finish->name) +
i.second)
            {
                i.first->mark = current->mark - heuristic(current->name, finish->name) +
heuristic(i.first->name, finish->name) + i.second;
                i.first->prev = current;
            }
        }
        std::pop_heap(graph.begin(), graph.end(), vertexComparator);
        graph.pop_back();
    }
```

```
    if (path.empty())
    {
        return std::nullopt;
    }
    return path;
}

void freeMemory(Graph graph)
{
    for (auto i : graph)
    {
        delete i;
    }
}

void printPath(std::optional<Path> path, std::ostream& out)
{
    if (!path)
    {
        out << "could not be found\n";
        return;
    }
    while (!path.value().empty())
    {
        out << path.value().top();
        path.value().pop();
    }
    out << '\n';
}

void clearMarksAndPrev(Graph& graph)
{
    for (auto i : graph)
    {
        i->mark = DBL_MAX;
```

```
            i->prev = nullptr;
        }
    }

    void printAnswer(std::optional<Path> pathStartFinish1, std::optional<Path>
    pathStartFinish2, std::optional<Path> pathFinish1Finish2, std::optional<Path>
    pathFinish2Finish1, std::ostream& out)
    {
        out << "The path between start and finish1:";
        printPath(pathStartFinish1, out);
        out << "The path between start and finish2:";
        printPath(pathStartFinish2, out);
        out << "The path between finish1 and finish2:";
        printPath(pathFinish1Finish2, out);
        out << "The path between finish2 and finish1:";
        printPath(pathFinish2Finish1, out);
    }
```

Greedy/main.cpp

```
#include "greedy.hpp"

int main()
{
   Vertex* start,*finish;
   std::ifstream file("Input.txt");
   Graph graph = initGraph(file,&start,&finish);
   file.close();
   std::optional<Path> path = findPath(start,finish);
   if(path)
   {
      printPath(path.value(),std::cout);
   }
   else
   {
      std::cerr<<"the path does not exist"<<'\n';
   }
```

```
    freeMemory(graph);
    return 0;
}
```

Greedy/greedy.cpp

```
#include "greedy.hpp"

auto linkComparator = [](Link link1,Link link2)
{
    return link1.second>link2.second;
};

bool checkInput(char nameOut,char nameIn, double weight)
{
    return std::isalpha(nameOut) && isalpha(nameIn) && weight>=0;
}

int findVertexIndexByName(const Graph& graph,char name)
{
    for(int i = 0 ;i <graph.size(); i++)
    {
        if(graph[i]->name==name)
        {
            return i;
        }
    }
    return -1;
}

Graph initGraph(std::istream& in,Vertex** start,Vertex** finish)
{
    Graph graph;
    char nameVertexOut,nameVertexIn;
    double weight;
    char nameStart,nameFinish;
```

```
while(in>>nameStart>>nameFinish)
{
   if(isalpha(nameStart) && isalpha(nameFinish))
   {
      break;
   }
   std::cerr<<"Error input"<<'\n';
}
*start = new Vertex(nameStart);
*finish  = new Vertex(nameFinish);
graph.push_back(*start);
graph.push_back(*finish);

while(in>>nameVertexOut>>nameVertexIn>>weight)
{
   if(!checkInput(nameVertexOut,nameVertexIn,weight))
   {
      std::cerr<<"Error input!"<<'\n';
      continue;
   }
   Vertex *vertexOut = new Vertex(nameVertexOut);
   Vertex *vertexIn = new Vertex(nameVertexIn);

   int indexOut = findVertexIndexByName(graph,nameVertexOut);
   if(indexOut==-1)
   {
      graph.push_back(vertexOut);
      indexOut = findVertexIndexByName(graph,nameVertexOut);
   }

   int indexIn = findVertexIndexByName(graph,nameVertexIn);
   if(indexIn==-1)
   {
      graph.push_back(vertexIn);
      indexIn = findVertexIndexByName(graph,nameVertexIn);
```

```
        }
        graph[indexOut]->linksHeap.push_back(Link(graph[indexIn],weight));
        std::push_heap(graph[indexOut]->linksHeap.begin(),graph[indexOut]-
>linksHeap.end(),linkComparator);
    }
    return graph;
}

std::optional<Path> findPath(Vertex *start,Vertex *finish)
{
    Path path;
    Vertex *current = start;
    path.push_back(current->name);
    while(current->prev!=nullptr || !current->linksHeap.empty())
    {
        if(current==finish)
        {
            break;
        }
        if(!(current->linksHeap.empty()))
        {
            Vertex *next=current->linksHeap[0].first;
            next->prev=current;
            std::pop_heap(current->linksHeap.begin(),current-
>linksHeap.end(),linkComparator);
            current->linksHeap.pop_back();
            current = next;
            path.push_back(current->name);
            continue;
        }
        else
        {
            Vertex* temp = current;
            current = current->prev;
            temp->prev = nullptr;
```

```
            path.pop_back();
        }
    }
    if(current!=finish)
    {
        return std::nullopt;
    }
    return path;
}

void freeMemory(Graph graph)
{
    for(auto i : graph)
    {
        delete i;
    }
}

void printPath(Path path,std::ostream &out)
{
    while(!path.empty())
    {
        out<<path.front();
        path.pop_front();
    }
}
```

Greedy/greedy.hpp

```
#pragma once

#include <iostream>
#include <algorithm>
#include <vector>
#include <cctype>
#include <fstream>
#include <optional>
```

```
#include <stack>
#include <deque>

struct Vertex;
using Link    = std::pair<Vertex*,double>;
using Links   = std::vector<Link>;
using Graph   = std::vector<Vertex*>;
using Path    = std::deque<char>;



struct Vertex
{
   Vertex(char name= ' '):name(name) {}
   char name;
   Links linksHeap;
   Vertex* prev = nullptr;
};

bool checkInput(char nameOut,char nameIn, double weight);
int findVertexIndexByName(const Graph& graph,char name);
Graph initGraph(std::istream& in,Vertex** start,Vertex** finish);
std::optional<Path> findPath(Vertex *start,Vertex *finish);
void freeMemory(Graph graph);
void printPath(Path path,std::ostream &out);
```