Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»
на тему
«Эвристический поиск в графе»

Студент: Н. С. Токарев
Преподаватель: Н. А. Зацепин
Группа: М8О-307Б-18
Дата:
Оценка:
Подпись:

Москва
2020

Условие

Разработать программу на языке C или C++, реализующую указанный алгоритм согласно заданию.
Задана карта, состоящий из n*m клеток. Необходимо найти путь из вершины с номером start в вершину с номером finish при помощи алгоритма A*. Также на карте возможно наличие непроходимых точек.

Метод решения

Данное задание предполагает реализацию алгоритма A* по нахождению пути.

1. Добавить стартовую клетку в открытый список (при этом её значения G, H и F равны 0).

2. Повторять следующие шаги:
   Ищем в открытом списке клетку с наименьшим значением величины F, делаем её текущей.
   Удаляем текущую клетку из открытого списка и помещаем в закрытый список.
   Для каждой из соседних, к текущей клетке, клеток:
   Если клетка непроходима или находится в закрытом списке, игнорируем её.
   Если клетка не в открытом списке, то добавляем её в открытый список, при этом рассчитываем для неё значения G, H и F, и также устанавливаем ссылку родителя на текущую клетку.
   Если клетка находится в открытом списке, то сравниваем её значение G со значением G таким, что если бы к ней пришли через текущую клетку. Если сохранённое в проверяемой клетке значение G больше нового, то меняем её значение G на новое, пересчитываем её значение F и изменяем указатель на родителя так, чтобы она указывала на текущую клетку.
   Останавливаемся, если:
   В открытый список добавили целевую клетку (в этом случае путь найден). Открытый список пуст (в этом случае к целевой клетке пути не существует).

3. Сохраняем путь, двигаясь назад от целевой точки, проходя по указателям на родителей до тех пор, пока не дойдём до стартовой клетки.

Описание программы

Мой проект состоит из трех файлов:main.cpp, map.hpp, search.hpp. Поиск происходит на карте с закрытыми точками. В качестве эвристики я выбрал манхэттенское расстояние, а перемещение происходит по 4 направлениям.
map.hhp: класс map представляет собой карту и содержит один основной метод: Construct SymbolsMap(unsigned int NoPointAmount). В качестве параметра указывается количество добавляемых непроходимых точек, которые необходимо ввести.Непроходимые точ-

ки добавляются в закрытый список. Соответственно карта представляет собой двумерный массив, содержащий информацию о точке: вес, значение F = вес + эвристика, координаты родителя.
search.hpp: Данный класс представляет собой реализацию эвристического поиска. Открытый список реализован с помощью приоритетной очереди по отсортированной по возрастанию. Также использую дополнительную структура для хранения актуального списка точек в открытом списке. Данный класс также содержит объект map. Соответственно соседние точки от текущей добавляются в вектор. После же данные точки анализируются согласно алгоритму. Ответ формируется ввиде карты, где посещенные точки отмечаются восклицательным знаком. Пример:

```
// путь(4 направления и эвристика манхэттенское расстояние)
// стартовая точка:(2,2), конечная точка:(9,9)
# # # # # # # # # # #
# 0 ! ! ! 0 0 0 0 0 #
# 0 ! # ! 0 0 0 0 0 #
# 0 # 0 ! 0 0 0 0 0 #
# 0 0 0 ! 0 0 0 0 0 #
# 0 0 0 ! 0 0 0 0 0 #
# 0 # 0 ! 0 0 0 # 0 #
# 0 # 0 ! 0 # # 0 0 #
# 0 0 # ! ! ! 0 # # #
# 0 0 0 # # ! ! ! ! #
# # # # # # # # # # #
```

Пояснения: закрытый список - список рассмотренных вершин. Открытый список - список вершин, которые необходимо рассмотреть.

Исходный код

```
// main.cpp
#include "source/map.hpp"
#include "source/search.hpp"

int main() {
    unsigned int n,m;
    std::cin >> n >> m;
    Map* testMap = new Map(n,m);
    unsigned int amount;
    std::cin >> amount;
    testMap->ConstructSymbolsMap(amount);
    Search* testSearch = new Search(testMap);
    std::cout << "Start(x,y) && Finish(x,y)" << std::endl;
    unsigned int x,y,x1,y1;
```

```
    std::cin >> x >> y >> x1 >> y1;
    auto start = std::make_pair(x,y);
    auto end = std::make_pair(x1,y1);
    testSearch->Result(start, end);
    testMap->Print();
    delete testMap;
    delete testSearch;
    return 0;
}

//map.hpp
#ifndef MAP_HPP
#define MAP_HPP
#include <iostream>
#include <vector>
#include <cmath>
#include <set>
#include <queue>


struct Information {
    std::string symbol; // 0 - true, # - false
    unsigned int allScore; // weight + heuristic(finish);
    unsigned int weight;
    std::pair<unsigned int, unsigned int> from; // parent_coordinate


    Information() {
        symbol = "0";
        allScore = 0;
        weight = 0;
        from = std::make_pair(0,0);
    }
};


class Map {
private:
    unsigned int lines;
    unsigned int colums;
    std::vector<std::vector<Information>> symbolsMap;
    std::set<std::pair<unsigned int, unsigned int>> noPoints; //new # in symbolsMap
```

```
public:
    friend class Search;

    Map(unsigned int lines, unsigned int colums) {
        this->lines = lines + 2;
        this->colums = colums + 2;
        symbolsMap.resize(this->lines);
    }


    void Print() {
        for(auto i = 0; i < lines;i++) {
            for(auto j = 0; j < colums;j++) {
                std::cout << symbolsMap[i][j].symbol << " ";
            }
            std::cout << std::endl;
        }
    }


    bool CheckNewPoint(int x, int y) {
    if(x > lines - 1 || y > colums - 1 || x <= 0 || y <= 0) {
        return false;
    }
    return true;
    }


    void ConstructSymbolsMap(unsigned int NoPointAmount) {
        std::string status;
        int temp = 0;
        for(auto i = 0; i < lines; i++) {
            for(auto j = 0; j < colums; j++) {
                symbolsMap[i].push_back(Information());
            }
        }
        for(auto j = 0; j < colums;j++) {
            symbolsMap[0][j].symbol = '#';
            symbolsMap[lines - 1][j].symbol = '#';
            noPoints.insert(std::make_pair(0, j));
            noPoints.insert(std::make_pair(lines - 1, j));
        }
```

```
        for(auto i = 0; i < lines;i++) {
            symbolsMap[i][0].symbol = '#';
            symbolsMap[i][colums - 1].symbol = '#';
            noPoints.insert(std::make_pair(i, 0));
            noPoints.insert(std::make_pair(i, colums - 1));
        }
        // add new #(noPoint)
        while(temp < NoPointAmount) {
            unsigned int pointX, pointY;
            std::cin >> pointX >> pointY;
            if(CheckNewPoint(pointX, pointY)) {
                symbolsMap[pointX][pointY].symbol = '#';
                noPoints.insert(std::make_pair(pointX, pointY));
                std::cout << "ADD" << std::endl;
            }
            temp++;
        }
        Print();
    }
};
#endif

//search.hpp
#ifndef SEARCH_HPP
#define SEARCH_HPP
#include "map.hpp"
#include <map>
#include <algorithm>
#define const_weight 5
using point = std::pair<unsigned int, unsigned int>;
using pointAndScore = std::pair<point, unsigned int>;


struct MyCompare {
    constexpr bool operator()(std::pair<point, unsigned int> const & a,
    std::pair<point, unsigned int> const & b) const noexcept {
        return a.second >= b.second;
    }
};


class Search {
public:
```

```
    Search(Map* map) {
        this->map = map;
    }

    void Result(point start, point finish) {
        if(map->CheckNewPoint(start.first,start.second) == true
            && map->CheckNewPoint(finish.first, finish.second) == true
            && AStar(start,finish) == true) {
                std::cout << "YES" << std::endl;
            ChangeSymbolMapAndPrintResult(start,finish);
        } else {
            std::cout << "ERROR" << std::endl;
        }
    }

private:
    Map* map;
    // открытый список
    std::priority_queue <pointAndScore,
            std::vector <pointAndScore>,MyCompare> openQueue;
    // множество точек в открытом списке
    std::set<point> pointsInOpenList;

    void ChangeSymbolMapAndPrintResult(point start, point finish) {
        point current = finish;
        int i = 1;
        while(current != start) {
            //map->symbolsMap[current.first][current.second].symbol = std::to_string(i);
            map->symbolsMap[current.first][current.second].symbol = '!';
            current = map->symbolsMap[current.first][current.second].from;
            i++;
        }
        //map->symbolsMap[start.first][start.second].symbol = std::to_string(i);
        map->symbolsMap[start.first][start.second].symbol = '!';
    }


    bool FindElement(std::set<point>& pointsInOpenList, point& old) {
        auto it = pointsInOpenList.find(old);
        if(it == pointsInOpenList.end()) {
            return false;
        }
```

```
    return true;
}


void Erase(std::set<point>& pointsInOpenList, point& old) {
    for(auto it = pointsInOpenList.begin();it != pointsInOpenList.end();it++) {
        if((*it) == old) {
            pointsInOpenList.erase(it);
            return;
        }
    }
}


unsigned int Heuristik(point start, point finish) {
    //return (std::max(abs(start.first - finish.first),abs(start.second -
    //finish.second)) * const_weight);
    //return (sqrt((pow((start.first - finish.first),2) + pow((start.second -
    //finish.second),2))) * const_weight);
    return ((abs(start.first - finish.first) + abs(start.second - finish.second))
    * const_weight);
}


void GetFourVertex(std::vector<point>& adjacentVertex,point& current) {
    int i_const,j_const;
    i_const = current.first;
    j_const = current.second;
    int i_f,j_f,i_s,j_s;
    j_f = current.second - 1;
    j_s = current.second + 1;
    adjacentVertex.push_back({i_const,j_f});
    adjacentVertex.push_back({i_const,j_s});
    i_f = i_const - 1;
    i_s = i_const + 1;
    adjacentVertex.push_back({i_f, j_const});
    adjacentVertex.push_back({i_s, j_const});
}


void GetAllVertex(std::vector<point>& adjacentVertex,point& current) {
    for(auto j = current.second - 1; j <= current.second + 1; j++) {
```

```
            if(j == current.second) {
                unsigned int first = j - 1;
                unsigned int second = j + 1;
                adjacentVertex.push_back({current.first, first});
                adjacentVertex.push_back({current.first, second});
            }
            unsigned int high, low;
            high = current.first + 1;
            low = current.first - 1;
            adjacentVertex.push_back({low, j});
            adjacentVertex.push_back({high, j});
        }
    }


    bool AStar(point start, point finish) {
        openQueue.push({start,0});
        pointsInOpenList.insert(start);
        while(openQueue.size() > 0) {
            pointAndScore current = openQueue.top();
            if(current.first == finish) {
                return true;
            }
            openQueue.pop();
            Erase(pointsInOpenList, current.first);
            map->noPoints.insert(current.first);
            std::vector<point> adjacentVertex;
            unsigned int tenativeScore =
                map->symbolsMap[current.first.first][current.first.second].weight +
                const_weight;
            GetFourVertex(adjacentVertex,current.first);
            //GetAllVertex(adjacentVertex,current.first);
            for(size_t j = 0;j < adjacentVertex.size(); j++) {
                if(map->noPoints.find(adjacentVertex[j]) != map->noPoints.end()) {
                    continue;
                }
    if(FindElement(pointsInOpenList,adjacentVertex[j])) {
if(tenativeScore <
map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].weight) {
        map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].from =
        current.first;
        map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].weight =
```

```
        tenativeScore;
        map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].allScore =
        tenativeScore + Heuristik(adjacentVertex[j],finish);
        }
    } else {
    map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].from =
    current.first;
    map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].weight =
    tenativeScore;
    map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].allScore =
    tenativeScore + Heuristik(adjacentVertex[j],finish);
    //std::cout << adjacentVertex[j].first << " " << adjacentVertex[j].second << " " <<
    << std::endl;
    openQueue.push({adjacentVertex[j],
    map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].allScore});
    pointsInOpenList.insert({adjacentVertex[j].first,adjacentVertex[j].second});
    }
            }
        }
        return false;
    }
};

#endif
```

Недочеты

Использование менее оптимальных сруктур данных для хранения информации.

Использование дополнительного множества для хранения актуального списка точек открытого списка.

Выводы

Таким образом алгоритм A* в некотором образе является наследником алгоритма Дейкстры. A* использует эвристику, с помощью которой можно отсекать некоторые неоптимальные пути. Существуют несколько эвристик и соответсвенно для разных случаев. Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин исследуемых алгоритмом, растет экспоненциально по сравнению с диной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию: $|h(v) - h^*(v)| <= O(\log(h^*(v)))$, где h^* - оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка $h(x)$не должна расти быстрее, ч ем логарифм от оптимальной эвристики.