Московский авиационный институт
(Национальный Исследовательский Институт)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования.

Курсовой проект

по курсу «Дискретный анализ»

Студент:	Суханов Е. А.
Группа:	М80-306Б-19
Преподаватель:	Сорокин С. А.
Оценка:	
Дата:	

Москва
2021

Оглавление

Введение

Задание. Эвристический поиск на решётках. Реализуйте алгоритм A* для графа на решётке. Первые четыре строки входного файла выглядят следующим образом:

type octile

height <x>

width <y>

map

Где «x» и «y» – высота и ширина карты соответственно. Затем в x строках задана сама карта в виде решётки символов, в которой символы «.» и «G» обозначают проходимые клети. Переход между ячейками возможен только по сторонам.

Далее дано число q и в следующих q строках даны запросы в виде четвёрок чисел на поиск кратчайшего пути между двумя позициями в решётке.

В ответ на каждый запрос выведите единственное число – расстояние между ячейками из запроса.

Программа должна читать входные данные из стандартного потока ввода и выводить ответ на стандартный поток вывода.

Алгоритм работы программы

Программу можно разделить на два компонента:

Ввод-вывод. Решается тривиально. Сначала записываем решетку, будем хранить её как двумерный массив символов. Затем для каждого запроса выполняем функцию поиска кратчайшего пути;

Алгоритм поиска A*. По сути является немного модифицированной версией алгоритма Дейкстры. Отличие заключается лишь в том, что сначала обрабатываются вершины, которые кажутся наиболее близкими к конечной вершине. Таким образом мы уменьшаем количество итераций. Близость определятся с помощью отдельной функции. Функция должна возвращать значение, которое не превосходит значения истинного пути, и может быть любой. Однако, правильно подобранная функция, которая возвращает близкое значение, по сравнению со значением длины истинного пути уменьшает кол-во итераций, по сравнению с другими функциями.

Так как мы можем передвигаться только по сторонам, то есть вверх, вниз, влево и вправо, моя функция возвращает кол-во клеток, которое необходимо пройти по диагонали (т. е. «змейкой», например сначала вверх, а затем право).

f(s, f) = abs(s.x - f.x) + abs(s.y – f.y)

Идея алгоритма Дейкстры достаточно проста. У нас есть множество изпользованных вершин, которое изначально пусто. Начинаем с начальной вершины, говорим, что путь от нее до нее равен 0. Повторяем итерации, пока можем выбрать следующую вершину. Начало итерации. Мы должны выбрать неиспользованную вершину с самым коротким путем. Помещаем выбранную вершину как использованную. Если это конечная вершина, то заканчиваем работу. Теперь нужно обновить значения кратчайших путей до соседей выбранной вершины, если пути стали короче – записываем.

Модификация заключается в следующем: при обновлении пути, нужно прибавлять не только стоимость перехода от выбранной вершины к соседней, но и значение эвристической функции от соседней к конечной.

Код программы

Main.cpp:

```
#include "solution.hpp"
#include <iostream>

int main() {

  TGrid grid;
  int n,m;
  std::cin >> n >> m;
  grid.resize(n);
  for(int i = 0; i < n; ++i) {
    grid[i].resize(m);
    for(int j = 0; j < m; ++j) {
      std::cin >> grid[i][j];
    }
  }

  int q;
  std::cin >> q;
  for(int i = 0; i < q; ++i) {
    TPoint s,f;
    std::cin >> s.first >> s.second >> f.first >> f.second;
    s.first--;
    s.second--;
    f.first--;
    f.second--;
    std::cout << Find(grid, s, f) << std::endl;
  }

  return 0;
}
```

Solution.hpp:

```
#pragma once
#include <vector>
#include <limits>

const long long INF = std::numeric_limits<long long>::max();
typedef std::vector<std::vector<char>> TGrid;
typedef std::pair<int,int> TPoint;

long long Find(const TGrid& grid, const TPoint& start, const TPoint& finish);
```

Solution.cpp:

```
#include "solution.hpp"
#include <set>
#include <iostream>

void Print(const TGrid& grid, const std::vector<std::vector<long long>>& costs) {
  int n = grid.size();
  int m = grid[0].size();
  for(int i = 0; i < n; ++i) {
    for(int j = 0; j < m; ++j) {
      if (grid[i][j] == '.') {
        std::cout << (costs[i][j] == INF ? -1 : costs[i][j]) << '\t';
      }
      else {
        std::cout << grid[i][j] << '\t';
      }
    }
    std::cout << "\n";
  }
}

inline long long Heuristic(const TPoint& start, const TPoint& finish) {
  return abs(start.first - finish.first) + abs(start.second - finish.second);
}

long long Find(const TGrid& grid, const TPoint& start, const TPoint& finish) {
  int n = grid.size();
  int m = grid[0].size();
  std::vector<std::vector<long long>> costs(n, std::vector<long long>(m, INF));
  costs[start.first][start.second] = 0;

  std::set<std::pair<int, TPoint>> q;

  q.insert(std::make_pair(0, start));
  while(!q.empty()) {
    TPoint v = q.begin()->second;
    q.erase(q.begin());
    if (v == finish)
      break;

    int dx[] = {-1, 0, 1, 0};
    int dy[] = {0, 1, 0, -1};

    for(int i = 0; i < 4; ++i) {
      TPoint to = {v.first + dx[i], v.second + dy[i]};
      if (to.first < 0 || to.first >= n || to.second < 0 || to.second >= m)
```

```
        continue;
      if (grid[to.first][to.second] != '.' && grid[to.first][to.second] != 'G')
        continue;

      if (costs[v.first][v.second] + 1 < costs[to.first][to.second]) {
        long long heruisticLen = Heuristic(finish, to);
        q.erase(std::make_pair(costs[to.first][to.second] + heruisticLen, to));
        costs[to.first][to.second] = costs[v.first][v.second] + 1;
        q.insert(std::make_pair(costs[to.first][to.second] + heruisticLen, to));
      }
    }
  }
  return costs[finish.first][finish.second] == INF ? -1 : costs[finish.first]
[finish.second];
}
```

Выводы

Я узнал, что существует множество алгоритмов для поиска путей между вершинами графа. Данные алгоритмы, а так же их модификации, имеют огромное количество применений. Это связано с тем, что графы – очень гибкая абстракция. Одно из самых популярных приложений – поиск кратчайшего пути на графе. При этом граф отображает карту.

Если смотреть конкретные приложения алгоритма A*. То его можно использовать там, где используется Дейкстра, для разового поиска. Или, например, у нас разные начальные вешины. Данный алгоритм часто используется в играх для передвижения различных юнитов из одной точки в другую.

Список источников

1. Введение в алгоритм A* (дата обращения 30.11.2021): https://habr.com/ru/post/331192/