Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

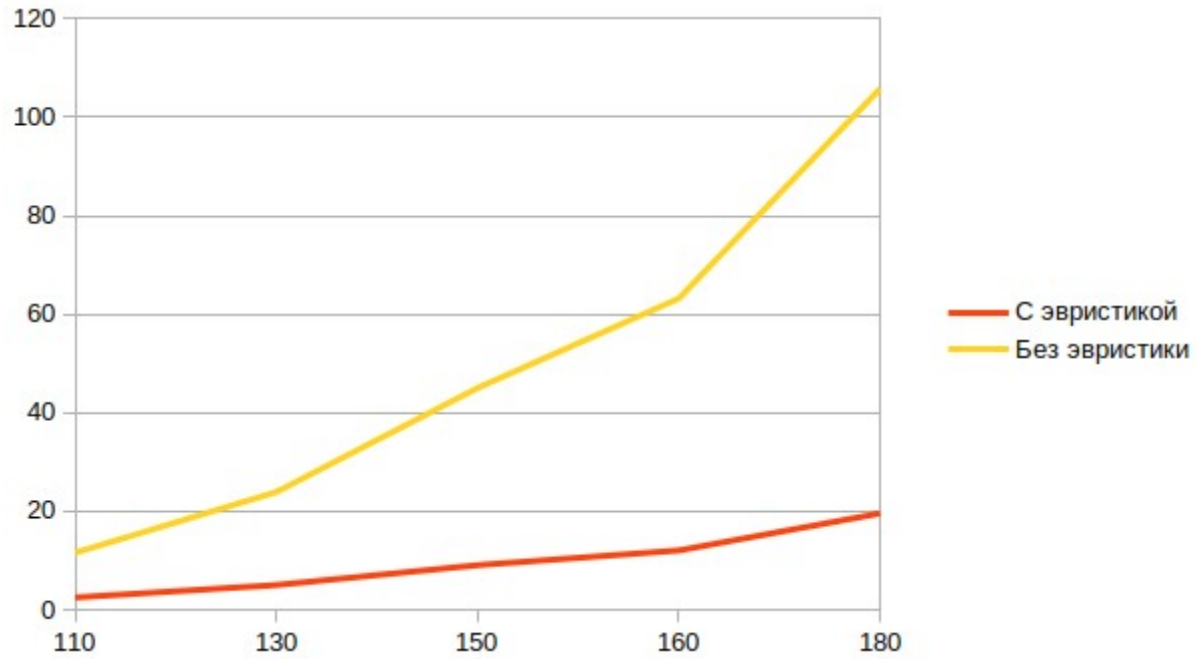
Курсовой проект по теме «Эвристический поиск в графе»

Студент: Литвина А.А.
Преподаватель: Журавлев А.А.
Группа: М8О-306Б-17
Дата:
Оценка:

Москва, 2019

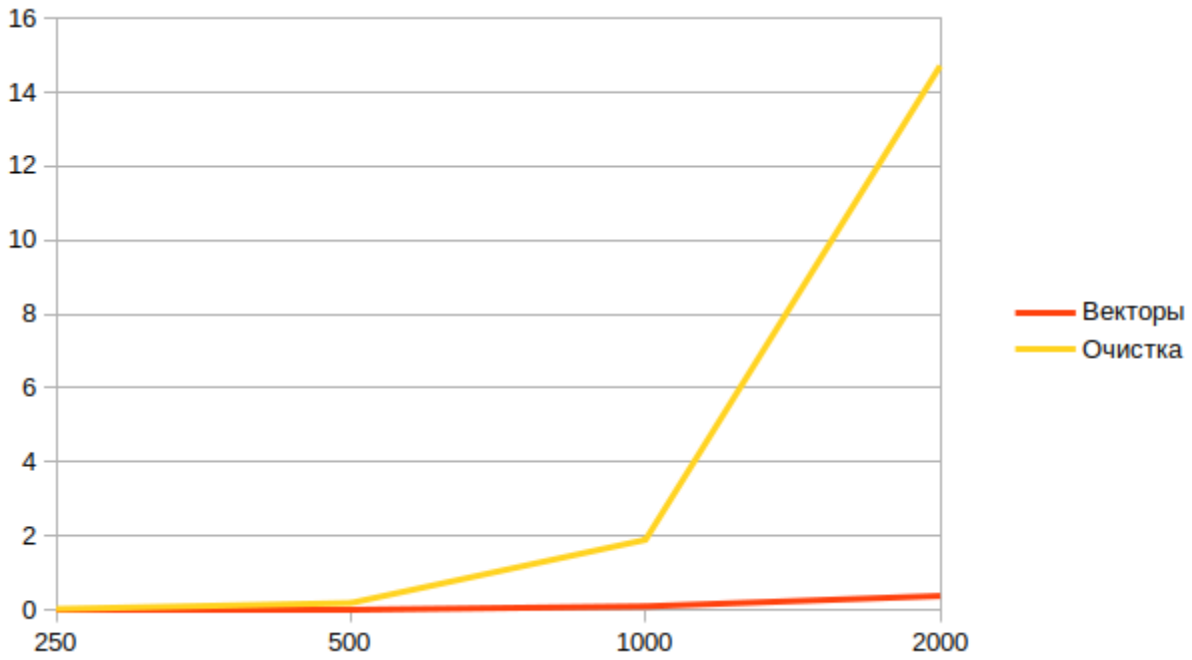
В данном курсовом проекте мною был реализован алгоритм поиска кратчайшего пути в неориентированном графе с помощью эвристики (алгоритм A*). Этот алгоритм представляет собой усовершенствованный вариант алгоритма Дейкстры, который находит кратчайшее расстояние от одной вершины графа до всех остальных. Особенность алгоритма Дейкстры заключается в том, что в отличие, например, от полного перебора вариантов, алгоритм Дейкстры на каждом этапе вычисляет наиболее оптимальное расстояние g(v) от начальной вершины до текущей и, таким образом, быстрее достигает цели. В алгоритме Дейкстры используется очередь с приоритетами для хранения вершин. Данный алгоритм работает только с ребрами неотрицательного веса. Алгоритм A* отличается от Дейкстры тем, что помимо расстояния g(v) рассчитывается эвристическая функция h(v), которая оценивает примерное расстояние от текущей вершины до конечной. Таким образом, алгоритм A* задается функцией f(v)=g(v)+h(v). То есть итоговое расстояние складывается из пройденного расстояния и оценки оставшегося расстояния. A* использует эвристику для исключения узлов, которые кажутся менее выгодными, и посещает только минимальное количество узлов, благодаря чему находит решение быстрее и с меньшими затратами памяти, чем простой алгоритм Дейкстры. В своем курсовом проекте я использовала самую простую и известную эвристику - евклидово расстояние между двумя точками.

Сравним время работы алгоритма Дейкстры с эвристикой и без и убедимся, что с эвристикой алгоритм работает намного быстрее.



Чтобы алгоритм работал быстрее, мне пришлось использовать некоторую хитрость. После каждого нового запроса нужно очищать все используемые контейнеры, но на очистку с помощью стандартных функций уходит слишком много времени. Поэтому я ввела дополнительные векторы типа <int>, в которых я храню значение итерации, на которой поменялось значение в соответствующем основном векторе. Следовательно, если текущая итерация не соответствует итерации в дополнительном векторе, то значение, записанное в основном векторе, нас не интересует и мы объявляем его по дефолту. Таким образом, на каждой итерации очищается только очередь с приоритетами, но это не требует большого количества времени.

Ниже приведены графики времени работы программы со стандартной очисткой контейнеров и с помощью дополнительных векторов.



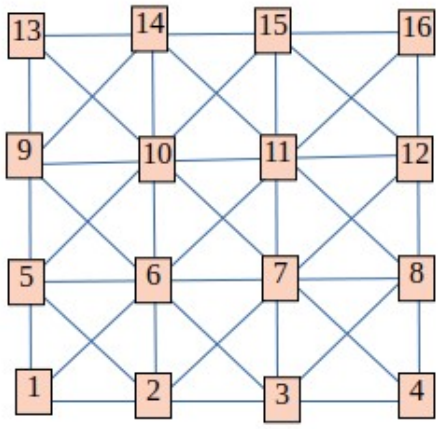
Как видно из графиков, стандартная очистка контейнеров занимает значительно больше времени, чем введение дополнительных векторов.

Код программы

```
#include <iostream>
#include <cstdlib>
#include <queue>
#include <vector>
#include <math.h>
#include <ctime>

using namespace std;

double Distance(int x1,int y1, int x2, int y2) {
	return sqrt(pow((x2-x1),2)+pow((y2-y1),2));
}

int main() {

	int N,M,x,y,v,u, q, start, finish, current, gen;
	double temp;
	cin >> N >> M;
	vector <pair <int, int> > Coord; //вектор с координатами
	vector <vector <pair <int, double> > > Matrix (N+1); //матрица с расстрояниями между вершинами
	priority_queue <pair <double, double>, vector <pair <double, double> >, greater <pair <double, double> > > Open;
	vector <bool> Is_Open (N+1,false); //вектор с неиспользованными вершинами
	vector <bool> Is_Close (N+1,false); //вектор с использованными вершинами
	vector <double> g (N+1,0); //вектор с суммой расстояний
	vector <double> f (N+1,0); //вектор с суммой расстояний + эвристика
	vector <int> generation_op (N+1,0); //вектор для очистки вектора Is_Open;
	vector <int> generation_cl (N+1,0); //вектор для очистки вектора Is_Close;
	vector <int> generation_g (N+1,0); //вектор для очистки вектора g;
	vector <int> generation_f (N+1,0); //вектор для очистки вектора f;


	Coord.push_back(make_pair(0,0));

	for (int i=0; i<N; i++) {
		cin >> x >> y;
		Coord.push_back(make_pair(x,y));
	}


	for (int i=0; i<M; i++) {
		cin >> v >> u;
		double d=Distance(Coord[v].first,Coord[v].second,Coord[u].first,Coord[u].second);
		Matrix[v].push_back(make_pair(u,d));
		Matrix[u].push_back(make_pair(v,d));
	}

	cin >> q;

	gen=0;

	for (int j=0; j<q; j++) {

		Open = priority_queue <pair <double, double>, vector <pair <double, double> >, greater <pair <double, double> > >();
```

```
		cin >> start >> finish;
		Open.push(make_pair(0,start));
		Is_Open[start]=true;
		g[start]=0;
		f[start]=g[start]
+Distance(Coord[start].first,Coord[start].second,Coord[finish].first,Coord[finish].second);
		gen++;

		while (!Open.empty()) {
			current=Open.top().second;
			if (current==finish)
				break;
			Open.pop();
			Is_Open[current]=false;
			generation_op[current]=gen;
			Is_Close[current]=true;
			generation_cl[current]=gen;
			for (int i=0; i<Matrix[current].size(); i++) {
				int cur_u=Matrix[current][i].first;
				double cur_d=Matrix[current][i].second;
				if ((generation_cl[cur_u]==gen)and(!Is_Close[cur_u]) or
(generation_cl[cur_u]!=gen)) {
					if (generation_g[current]!=gen) {
						g[current]=0;
						generation_g[current]=gen;
					}
					if (generation_g[cur_u]!=gen) {
						g[cur_u]=0;
						generation_g[cur_u]=gen;
					}
					temp=g[current]+cur_d;
					if ((temp<g[cur_u])or(generation_op[cur_u]==gen)and(!
Is_Open[cur_u])or(generation_op[cur_u]!=gen)) {
						g[cur_u]=temp;
						f[cur_u]=g[cur_u]
+Distance(Coord[cur_u].first,Coord[cur_u].second,Coord[finish].first,Coord[finish].second);
						generation_f[cur_u]=gen;
						if ((generation_op[cur_u]==gen)and(!Is_Open[cur_u]) or
(generation_op[cur_u]!=gen)) {
							Open.push(make_pair(f[cur_u],cur_u));
							Is_Open[cur_u]=true;
							generation_op[cur_u]=gen;
						}
					}
				}
			}
		}

		if (generation_f[finish]==gen)
			cout << f[finish] << endl;
		else cout << 0 << endl;
	}
}
```

Рассмотрим пример работы алгоритма для графа, изображенного на рисунке.



Расстояние от вершины 6 до всех остальных:

1	2	3	4	5	6	7	8
1.41421	1	1.41421	2.41421	1	0	1	2
9	10	11	12	13	14	15	16
1.41421	1	1.41421	2.41421	2.41421	2	2.41421	2.82843

Вывод

В этом курсовом проекте я познакомилась с алгоритмом A*, узнала что такое эвристика, какие разновидности бывают. Я реализовала самую простую эвристику - евклидово расстояние между точками. Также я выяснила, чем A* отличается от обычного Дейкстры и показала наглядно на графиках, что алгоритм с эвристикой работает значительно быстрее за счет того, что посещает меньшее количество узлов. В ходе выполнения проекта я столкнулась с некоторой трудностью: очистка всех контейнеров при каждом новом запросе занимает слишком много времени. Поэтому я была вынуждена отказаться от очистки с помощью стандартных функций и ввести дополнительные векторы для фиксирования итерации, на которой было изменено значение в основном векторе. Таким образом, очистка оказалась вовсе не нужна. Время работы алгоритма в обоих случаях я также представила на графике и показала, что в большинстве случаев введение дополнительных векторов оказывается выгоднее.