МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Лабораторная работа № 1 по дисциплине «Параллельное программирование»

Выполнил студент группы ИВТ-31 _____/Седов М. Д./
Проверил доцент кафедры ЭВМ _____/Долженкова М. Л./

Киров 2020

1 Задание

Необходимо реализовать поиск разрешающей последовательности ходов в пятнашках произвольной размерности.

1. Изучить алгоритм
2. Провести доказательную оценку алгоритма по временной сложности и затратами по памяти
3. Реализовать алгоритм с помощью языка C++
4. Построить набор тестовых примеров и провести оценку эффективности реализованного алгоритма

2 Изучение предметной области

Алгоритм представляет из себя поиск пути в графе, вершинам которого являются состояние каждого элемента пятнашек. Сам граф является бинарным деревом поиска, в качестве значения которого является оценочная стоимость состояния пятнашек, что позволяет ускорить нахождение разрешающей последовательности. В качестве алгоритма для поиска наилучшей последовательности используется A*, а в качестве оценочной функции используется эвристика расстояние городских кварталов(manhattan distance). При достижении состояния со значением оценочной функции равной 0 будет найден разрешающая последовательность ходов.

Асссимптотическая сложность для данного алгоритма: O(lgV*V), где V – число состояний игры для писка в дереве.

3 Программная реализация

Листинг программной реализации алгоритма на C++ приведен в приложении А.

4 Тестирование

В ходе тестирования матрица со значениями пятнашек была сгенерированна случайным образом. Количество строк и столбцов пятнашек и результаты тестирования приведены в таблице 1.

Таблица 1 – Результаты тестирования

Количество строк и столбцов	Среднее время из 10 запусков, с
2x3	0,0000734
3x3	0,0011579
3x4	0,0126833
4x4	0,112261
4x5	2,50988
5x5	61,3944

5 Вывод

В ходе лабораторной работы был реализован алгоритм поиска разрешающей последовательности ходов в пятнашках произвольной размерности. В ходе тестирования было замечено, что время работы алгоритма увеличивается как при увеличении строк матрицы, так и при увеличении столбцов.

Приложение А
(обязательное)
Листинг программной реализации

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <vector>
#include "Map.h"
#include "BinTree.h"
#include "State.h"
#include <ctime>
#include <limits>

#define TESTS 10

using namespace std;


bool check (Map*);
void printMap(Map*);

Map* generateMap(int lines, int cols) {
int len = lines * cols;
Map* map = new Map(lines, cols);

for (int i = 0; i < len; ++i)
{
map->map[i] = i + 1;
}
map->map[len - 1] = 0;
int i = 0;
```

```
int shift_pos;
while (i<=len*20) {
int zero = map->find(0);
shift_pos = rand() % 4;
switch (shift_pos){
case 0:
	if (zero / map->getCols() != 0) {
		map = map->shift(shift_pos);
		i++;
	}
	continue;
case 1:
	if (zero % map->getCols() != map->getCols() - 1) {
		map = map->shift(shift_pos);
		i++;
	}
	continue;
case 2:
	if (zero / map->getCols() != map->getLines() - 1) {
		map = map->shift(shift_pos);
		i++;
	}
	continue;
case 3:
	if (zero % map->getCols() != 0) {
		map = map->shift(shift_pos);
		i++;
	}
	continue;
}
}

return map;
}

void printMap(Map* map) {
std::cout << std::endl;
for (int i = 0; i < map->lines; ++i) {
for (int j = 0; j < map->cols; ++j) {
	std::cout << map->map[i*map->cols + j] << '\t';
}
std::cout << std::endl;
}
std::cout << std::endl;
}

vector<State*> a(Map* map) {

BinTree open = BinTree(new State(map, NULL));
BinTree close = BinTree();
State* min = open.min();

close.add(min);
```

```
open.del(min);

for (; min->getCost() != 0; min = open.min(), close.add(min), open.del(min))
{
int zero = min->getMap()->find(0);

if (zero / map->getCols() != 0) {
        State* s = new State(min->getMap()->shift(0), min);
        if ((open.find(s) == NULL) && (close.find(s) == NULL)) {
                open.add(s);
        }
}

if (zero % map->getCols() != map->getCols() - 1) {
        State* s = new State(min->getMap()->shift(1), min);
        if ((open.find(s) == NULL) && (close.find(s) == NULL)) {
                open.add(s);
        }
}

if (zero / map->getCols() != map->getLines() - 1) {
        State* s = new State(min->getMap()->shift(2), min);
        if ((open.find(s) == NULL) && (close.find(s) == NULL)) {
                open.add(s);
        }
}

if (zero % map->getCols() != 0) {
        State* s = new State(min->getMap()->shift(3), min);
        if ((close.find(s) == NULL) && (open.find(s) == NULL)) {
                open.add(s);
        }
}

}
std::vector<State*> solution;

State* s = min;
do
{
solution.push_back(s);
s = s->getParent();
} while (s != NULL);

return solution;
}


int main(int argc, char const *argv[]) {
int lines, cols;
Map* map;

do {
```

```
system("cls");
cout << "Enter field sizes: " << endl;
cin >> lines >> cols;
cin.clear();
if (cin.good() == false || cols < 2 || lines < 2) {
		system("cls");
		std::cout << "The size of the field is wrong" << std::endl;
		cin.ignore(numeric_limits<streamsize>::max(), '\n');
		getchar();
}
} while (cols*lines <= 4);

double t = 0;
std::vector<State*> ans;

for (int i = 0; i < TESTS; i++) {
srand(i);
map = generateMap(lines, cols);

printMap(map);

std::cout << "\n" << "Case #" << i + 1 << ": ";
clock_t time = clock();
ans = a(map);
time = clock() - time;

std::cout << "\n" << "Solution:";

for (int k = ans.size()-1; k >= 0; k--) {
		printMap(ans[k]->getMap());
}

std::cout << "Time:" << (double)time / CLOCKS_PER_SEC << std::endl;
t += (double)time / CLOCKS_PER_SEC;

}

std::cout << "\n" << "---------- " << std::endl;
std::cout << "Average time = " << t / TESTS << std::endl;
system("pause");
return 0;
}

Map.h
#pragma once

class Map
{
public:
int* map;
int lines;
int cols;
public:
```

```
Map(int lines, int cols) {
this->map = new int[lines*cols];
this->lines = lines;
this->cols = cols;
}

int getLines() {
return this->lines;
}

int getCols() {
return this->cols;
}

int find(int n) {
for (int i = 0; i < lines * cols; ++i) {
        if (map[i] == n) return i;
}
return  -1;
}

Map* shift(int angle) {
if ((angle < 0) || (angle > 3)) return this;

int len = lines * cols;
Map* mapc = new Map(lines, cols);

int ind = 0;

for (int i = 0; i < len; ++i) {
        mapc->map[i] = map[i];
        if (map[i] == 0) ind = i;
}

switch (angle) {
//up
case 0: {
        mapc->map[ind] = mapc->map[ind - cols];
        mapc->map[ind - cols] = 0;
        break;
}
//right
case 1: {
        mapc->map[ind] = mapc->map[ind + 1];
        mapc->map[ind + 1] = 0;
        break;
}
//down
case 2: {
        mapc->map[ind] = mapc->map[ind + cols];
        mapc->map[ind + cols] = 0;
        break;
}
```

```
//left
case 3: {
		mapc->map[ind] = mapc->map[ind - 1];
		mapc->map[ind - 1] = 0;
		break;
}
}
return mapc;
}
};
```

State.h

```
#pragma once
#include <cstdlib>
#include <functional>
#include "Map.h"

int costFunc(Map* map);
unsigned int calcHash(Map* map);

class State
{
public:
int cost;
State* parent;
Map* map;
int hash;
public:
State(Map* map, State* parent) {
this->cost = costFunc(map);
this->parent = parent;
this->map = map;
this->hash = calcHash(map);
}

State() {}

const State* copy() const {
return this;
}

Map* getMap() const {
return this->map;
}

int getHash() const {
return this->hash;
}

int getCost() const {
return this->cost;
}
```

```
State* getParent() {
return this->parent;
}
};

int costFunc(Map* map) {
int len = map->lines*map->cols;
int sum = 0;
for (int i = 0; i < len; ++i)
{
if (map->map[i] == 0) continue;
int dx = abs((i % map->cols) - ((map->map[i] - 1) % map->cols));
int dy = abs((i / map->cols) - ((map->map[i] - 1) / map->cols));
sum += dx + dy;
}

return sum;
}


unsigned int calcHash(Map* map) {
int len = map->lines * map->cols;
unsigned int h = 0;
for (int i = 0; i < len; ++i) {
h += std::hash<int>{}(map->map[i] * (1 << i));
}

return h;
}
```

BinTree.h

```
#pragma once
#include <cstdlib>
#include "Map.h"
#include "State.h"

struct Node {
Node* left;
Node* right;
State* elem;
};

class BinTree
{
private:
Node* node;
public:
int len;

BinTree(State* s) {
this->node = new Node();
this->node->elem = s;
len = 1;
```

```
}

BinTree() {
this->node = new Node();
len = 0;
}

~BinTree() {
}

int getLen() {
return len;
}

void add(State* s) {
Node* n = this->node;

len += 1;

if (n->elem == NULL) {
	n->elem = s;
	return;
}
do
{
	if (s->getCost() >= n->elem->getCost()) {
		if (n->right == NULL) {
			n->right = new Node();
			n->right->elem = s;
			break;
		}
		else {
			n = n->right;
		}
	}
	else {
		if (n->left == NULL) {
			n->left = new Node();
			n->left->elem = s;
			break;
		}
		else {
			n = n->left;
		}
	}

} while (true);
}

void del(State* s) {
Node* n = this->node;
Node* p = NULL;
```

```
while (n->elem != s)
{
	p = n;
	if (s->getCost() > n->elem->getCost()) {
		n = n->right;
	}
	else {
		n = n->left;
	}

	if (n == NULL) return;
}

len -= 1;

if ((n->left == NULL) && (n->right == NULL)) {
	if (p == NULL) {
		n->elem = NULL;
		return;
	}
	if (p->left == n) p->left = NULL;
	else p->right = NULL;
	delete n;
	return;
}

if (n->left == NULL) {
	if (p == NULL) {
		this->node = n->right;
		delete n;
		return;
	}
	if (p->left == n) p->left = n->right;
	else p->right = n->right;
	delete n;
	return;
}

if (n->right == NULL) {
	if (p == NULL) {
		this->node = n->left;
		delete n;
		return;
	}
	if (p->left == n) p->left = n->left;
	else p->right = n->left;
	delete n;
	return;
}

if (n->right->left == NULL) {
	n->elem = n->right->elem;
	n->right = n->right->right;
```

```
		return;
	}
	else {
		Node* k = n->right;

		while (k->left->left != NULL) {
			k = k->left;
		}

		n->elem = k->left->elem;
		k->left = k->left->right;
	}

}

bool cmp(State* a, State* b) {
Map* am = a->getMap();
Map* bm = b->getMap();
int len = am->getCols() * am->getLines();
for (int i = 0; i < len; i++) {
		if (am->map[i] != bm->map[i]) return false;
}
return true;

}


State* find(State* s) {
Node* n = this->node;

if (n->elem == NULL) return NULL;

while ((n->elem->getHash() != s->getHash()) || (!cmp(n->elem, s))) {
		if (s->getCost() >= n->elem->getCost()) {
			n = n->right;
		}
		else {
			n = n->left;
		}

		if (n == NULL) return NULL;
}
return n->elem;
}


State* min() {
Node* n = this->node;

while (n->left != NULL) n = n->left;

return n->elem;
}
```

```
};
```