МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

**Факультет программной инженерии и компьютерной техники**

**ЛАБОРАТОРНАЯ РАБОТА №2**

по дисциплине

“Системы искусственного интеллекта”

Изучение алгоритмов поиска

Вариант №4 (Рига - Уфа)

**Студент:**

Воробьев Кирилл Олегович

            Группа P33302

**Преподаватель:**

Королёва Ю.А.



Санкт-Петербург, 2022

**Цель работы:**

Исследование алгоритмов решения задач методом поиска

**Задание:**

Имеется транспортная сеть, связывающая города СНГ. Сеть представлена в виде таблицы связей между городами. Связи являются двусторонними, т.е. допускают движение в обоих направлениях. Необходимо проложить маршрут из одной заданной точки в другую.

Этап 1. Неинформированный поиск. На этом этапе известна только топология связей между городами. Выполнить:

1) поиск в ширину;

2) поиск глубину;

3) поиск с ограничением глубины;

4) поиск с итеративным углублением;

5) двунаправленный поиск.

Отобразить движение по дереву на его графе с указанием сложности каждого вида поиска. Сделать выводы.

Этап 2. Информированный поиск. Воспользовавшись информацией о протяженности связей от текущего узла, выполнить:

1) жадный поиск по первому наилучшему соответствию;

2) затем, используя информацию о расстоянии до цели по прямой от каждого узла, выполнить поиск методом минимизации суммарной оценки А\*.

Отобразить на графе выбранный маршрут и сравнить его сложность с неинформированным поиском. Сделать выводы.

**Выполнение работы:**

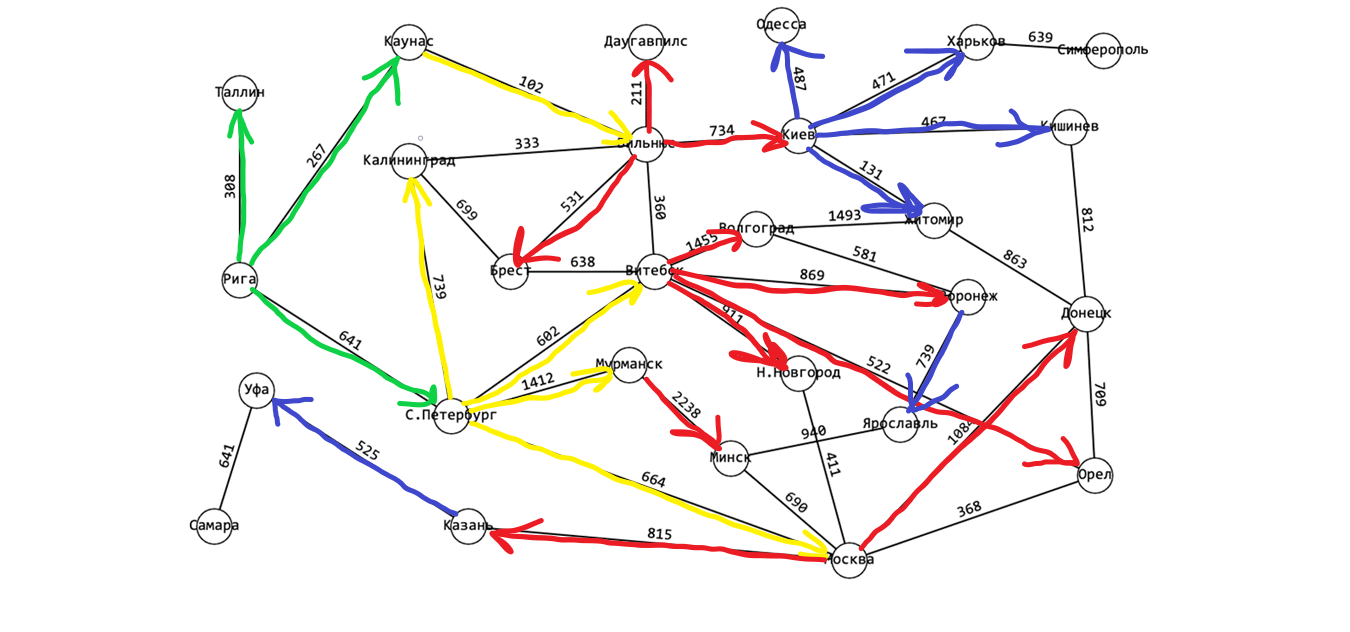
***Неинформативный поиск***

map<string, vector<pair<string, int>>>  
 cities = **{** {"Brest", {  
 make\_pair("Vilnius", 531),  
 make\_pair("Vitebsk", 638),  
 make\_pair("Kaliningrad", 699)  
 }},  
  
 {"Vilnius", {  
 make\_pair("Brest", 531),  
 make\_pair("Vitebsk", 360),  
 make\_pair("Daugavpils", 211),  
 make\_pair("Daugavpils", 211),  
 make\_pair("Kaliningrad", 333),  
 make\_pair("Kaunas", 102),  
 make\_pair("Kiev", 734)  
 }},  
  
 {"Vitebsk", {  
 make\_pair("Brest", 638),  
 make\_pair("Vilnius", 360),  
 make\_pair("Voronezh", 869),  
 make\_pair("Volgograd", 1455),  
 make\_pair("Nizhniy Novgorod", 911),  
 make\_pair("Saint-Petersburg", 602),  
 make\_pair("Orel", 522)  
 }},  
  
 {"Voronezh", {  
 make\_pair("Vitebsk", 869),  
 make\_pair("Volgograd", 581),  
 make\_pair("Yaroslavl", 739)  
 }},  
  
 {"Volgograd", {  
 make\_pair("Vitebsk", 1455),  
 make\_pair("Voronezh", 581),  
 make\_pair("Zhytomyr", 1493)  
 }},  
  
 {"Nizhniy Novgorod", {  
 make\_pair("Vitebsk", 911),  
 make\_pair("Moscow", 411)  
 }},  
  
 {"Daugavpils", {  
 make\_pair("Vilnius", 211)  
 }},  
  
 {"Kaliningrad", {  
 make\_pair("Brest", 699),  
 make\_pair("Vilnius", 333),  
 make\_pair("Saint-Petersburg", 739)  
 }},  
  
 {"Kaunas", {  
 make\_pair("Riga", 267),  
 make\_pair("Vilnius", 102)  
 }},  
  
 {"Kiev", {  
 make\_pair("Zhytomyr", 131),  
 make\_pair("Vilnius", 734),  
 make\_pair("Chisinau", 467),  
 make\_pair("Odessa", 487),  
 make\_pair("Kharkov", 471)  
 }},  
  
 {"Zhytomyr", {  
 make\_pair("Kiev", 131),  
 make\_pair("Donetsk", 863),  
 make\_pair("Volgograd", 1493)  
 }},  
  
 {"Donetsk", {  
 make\_pair("Zhytomyr", 863),  
 make\_pair("Moscow", 1084),  
 make\_pair("Chisinau", 812),  
 make\_pair("Orel", 709)  
 }},  
  
 {"Chisinau", {  
 make\_pair("Kiev", 467),  
 make\_pair("Donetsk", 812)  
 }},  
  
 {"Saint-Petersburg", {  
 make\_pair("Vitebsk", 602),  
 make\_pair("Kaliningrad", 739),  
 make\_pair("Riga", 641),  
 make\_pair("Moscow", 664),  
 make\_pair("Murmansk", 1412)  
 }},  
  
 {"Riga", {  
 make\_pair("Saint-Petersburg", 641),  
 make\_pair("Kaunas", 267),  
 make\_pair("Tallin", 308)  
 }},  
  
 {"Moscow", {  
 make\_pair("Saint-Petersburg", 664),  
 make\_pair("Kazan", 815),  
 make\_pair("Nizhniy Novgorod", 411),  
 make\_pair("Minsk", 690),  
 make\_pair("Donetsk", 1084),  
 make\_pair("Orel", 368)  
 }},  
  
 {"Kazan", {  
 make\_pair("Moscow", 815),  
 make\_pair("Ufa", 525)  
 }},  
  
 {"Minsk", {  
 make\_pair("Moscow", 690),  
 make\_pair("Yaroslavl", 940),  
 make\_pair("Murmansk", 2238)  
 }},  
  
 {"Murmansk", {  
 make\_pair("Saint-Petersburg", 1412),  
 make\_pair("Minsk", 2238)  
 }},  
  
 {"Orel", {  
 make\_pair("Moscow", 368),  
 make\_pair("Vitebsk", 522),  
 make\_pair("Donetsk", 709)  
 }},  
  
 {"Odessa", {  
 make\_pair("Kiev", 487)  
 }},  
  
 {"Tallin", {  
 make\_pair("Riga", 308)  
 }},  
  
 {"Kharkov", {  
 make\_pair("Kiev", 471),  
 make\_pair("Simferopol", 639)  
 }},  
  
 {"Simferopol", {  
 make\_pair("Kharkov", 639)  
 }},  
  
 {"Yaroslavl", {  
 make\_pair("Voronezh", 739),  
 make\_pair("Minsk", 940)  
 }},  
  
 {"Ufa", {  
 make\_pair("Kazan", 525),  
 make\_pair("Samara", 461)  
 }},  
  
 {"Samara", {  
 make\_pair("Ufa", 461)  
 }}  
**}**;

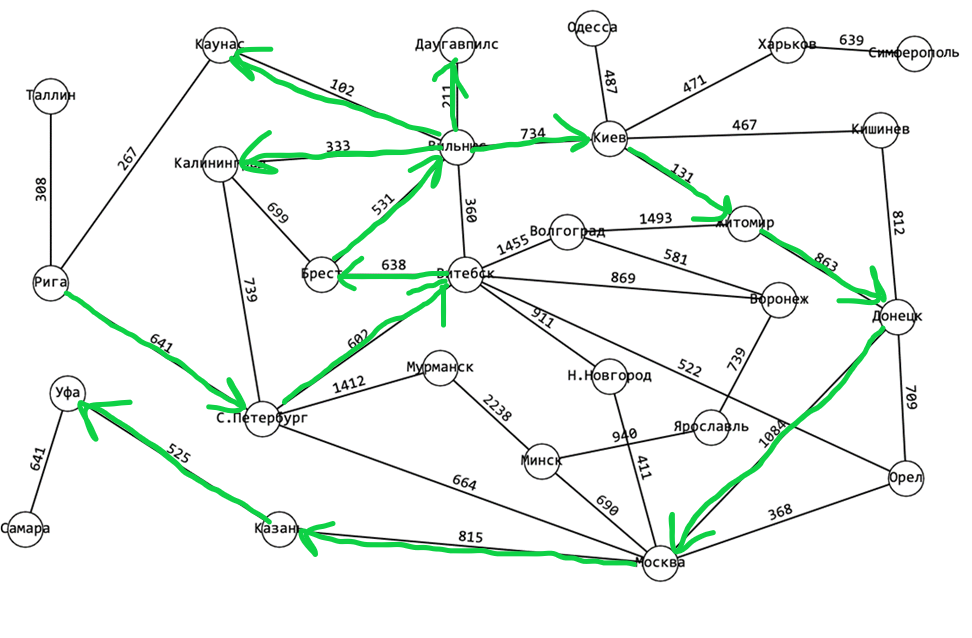
map<string, bool> visited;  
map<string, string> parent;

string bfs(const string &from, const string &to) {  
 queue<string> q;  
 q.push(from);  
 visited[from] = true;  
 while (!q.empty()) {  
 string current = q.front();  
 q.pop();  
 for (auto &city: cities[current]) {  
 string second\_city = city.first;  
 if (!visited[second\_city]) {  
 visited[second\_city] = true;  
 q.push(second\_city);  
 parent[second\_city] = current;  
 }  
 }  
 }  
 string city = to;  
 string way = to;  
 while (!parent[city].empty() && parent[city] != "null") {  
 way = parent[city] + " -> " + way;  
 city = parent[city];  
 }  
 return way;  
}  
  
string dfs(const string &from, const string &to) {  
 visited[from] = true;  
 for (auto it = cities[from].begin(); it != cities[from].end(); it++) {  
 string current = it->first;  
 if (!visited[current]) {  
 parent[current] = from;  
 dfs(current, to);  
 }  
 }  
 string city = to;  
 string way = to;  
 while (!parent[city].empty() && parent[city] != "null") {  
 way = parent[city] + " -> " + way;  
 city = parent[city];  
 }  
 return way;  
}  
  
string dls(const string &from, const string &to, int limit) {  
 visited[from] = true;  
 if (limit == 0)  
 return "Way not found";  
 if (from == to)  
 return "true";  
 for (auto it = cities[from].begin(); it != cities[from].end(); it++) {  
 string current = it->first;  
 if (!visited[current]) {  
 parent[current] = from;  
 if (dls(current, to, limit - 1) == "true")  
 return "true";  
 }  
 }  
 if (parent[to].empty() || parent[to] == "null") return "Way not found";  
 string city = to;  
 string way = to;  
 while (!parent[city].empty() && parent[city] != "null") {  
 way = parent[city] + " -> " + way;  
 city = parent[city];  
 }  
 return way;  
}  
  
string iddfs(const string &from, const string &to) {  
 for (int i = 1;; i++) {  
 clear\_visited();  
 if (dls(from, to, i) != "Way not found")  
 break;  
 }  
 if (parent[to].empty() || parent[to] == "null") return "Way not found";  
 string city = to;  
 string way = to;  
 while (!parent[city].empty() && parent[city] != "null") {  
 way = parent[city] + " -> " + way;  
 city = parent[city];  
 }  
 return way;  
}  
  
void bfs\_modified(queue<string> \*q, set<string> \*visited\_map, map<string, string> \*parent\_map) {  
 string current = q->front();  
 q->pop();  
 for (auto it = cities[current].begin(); it != cities[current].end(); it++) {  
 string to = it->first;  
 if (visited\_map->count(to) == 0) {  
 visited\_map->insert(to);  
 q->push(to);  
 (\*parent\_map)[to] = current;  
 }  
 }  
}  
  
string is\_intersecting(const set<string> &s\_visited, const set<string> &t\_visited) {  
 vector<string> common\_data;  
 set\_intersection(s\_visited.begin(), s\_visited.end(), t\_visited.begin(), t\_visited.end(),  
 back\_inserter(common\_data));  
 if (!common\_data.empty())  
 return common\_data.back();  
 else  
 return "null";  
}  
  
string bidirectional\_search(const string &from, const string &to) {  
 map<string, string> s\_parent, t\_parent;  
 queue<string> s\_queue, t\_queue;  
 set<string> s\_visited, t\_visited;  
 string result\_way;  
 s\_queue.push(from);  
 s\_visited.insert(from);  
 s\_parent[from] = "null";  
 t\_queue.push(to);  
 t\_visited.insert(to);  
 t\_parent[to] = "null";  
 string result;  
 while (!s\_queue.empty() && !t\_queue.empty()) {  
 bfs\_modified(&s\_queue, &s\_visited, &s\_parent);  
 bfs\_modified(&t\_queue, &t\_visited, &t\_parent);  
 string intersection = is\_intersecting(s\_visited, t\_visited);  
 if (intersection != "null") {  
 result = intersection;  
 }  
 }  
 if (result == "null") {  
 return "Way not found";  
 } else {  
 if (!(s\_parent[to].empty() || s\_parent[to] == "null")) {  
 string city\_s = to;  
 string way\_s = to;  
 while (!s\_parent[city\_s].empty() && s\_parent[city\_s] != "null") {  
 way\_s = s\_parent[city\_s] + " -> " + way\_s;  
 city\_s = s\_parent[city\_s];  
 }  
 result\_way += way\_s;  
  
 if (!(t\_parent[to].empty() || t\_parent[to] == "null")) {  
 string city\_t = to;  
 string way\_t = to;  
 while (!t\_parent[city\_t].empty() && t\_parent[city\_t] != "null") {  
 way\_t = t\_parent[city\_t] + " -> " + way\_t;  
 city\_t = t\_parent[city\_t];  
 }  
 result\_way += way\_t;  
 }  
 }  
 return result\_way;  
 }  
}

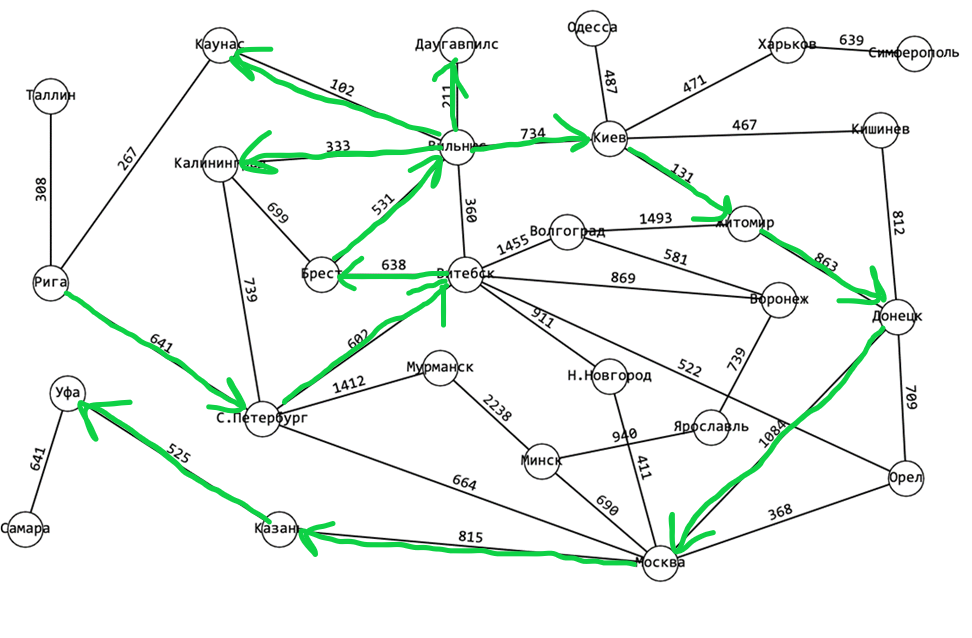
*Поиск в ширину*

**

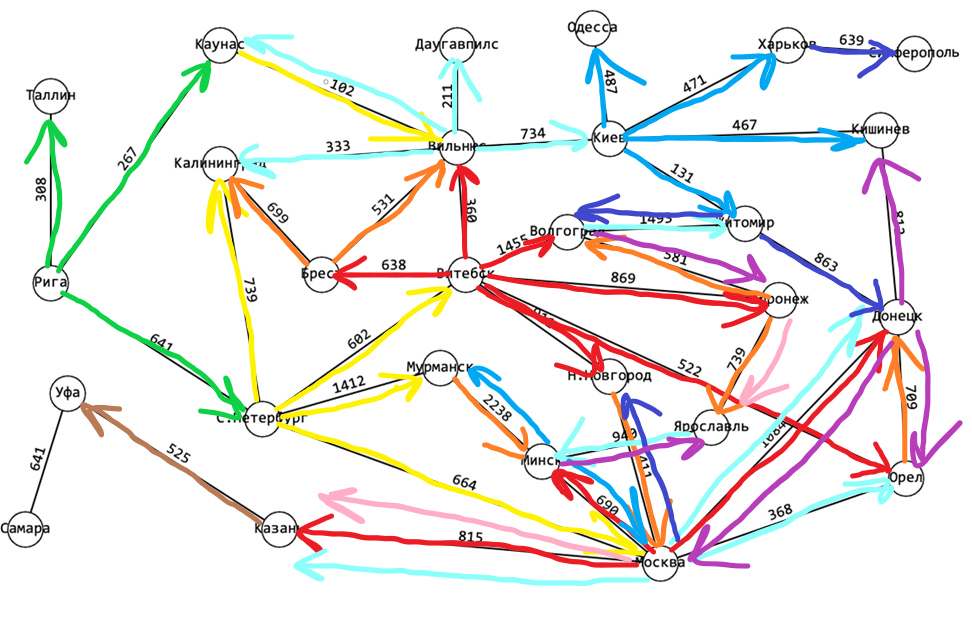
*Поиск в глубину*

**

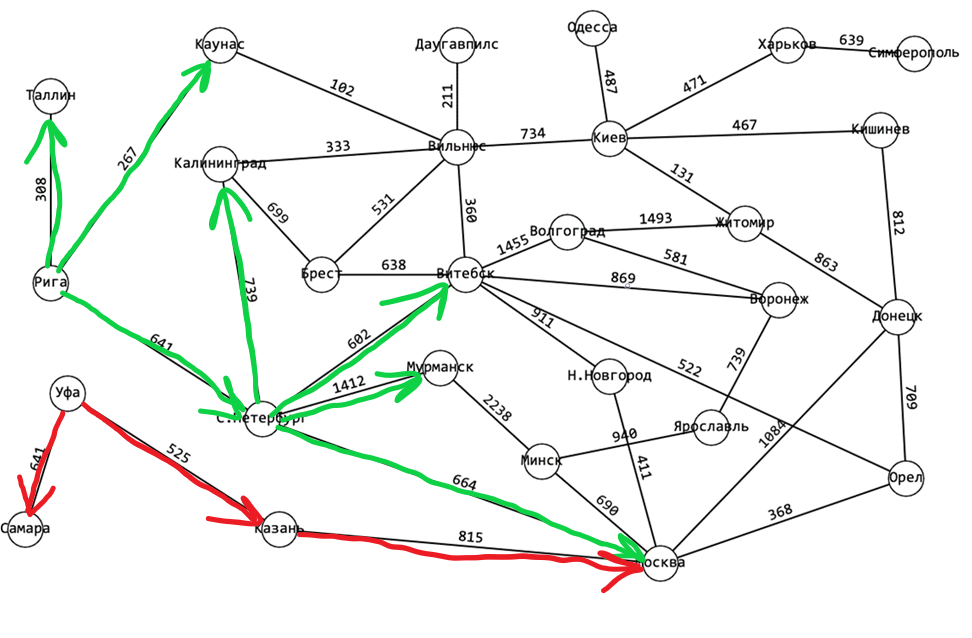
*Поиск с ограничением глубины*

**

*Поиск с итеративным углублением*

**

*Двунаправленный поиск*



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Алгоритм | Поиск в ширину | Поиск в глубину | Поиск с ограничением глубины | Поиск с итеративным углублением | Двунаправленный поиск |
| Сложность | O(N + E) | O(N + E) | O(N + E) | O(N + E) | O(N + E) |

Выводы:

В данной ситуации все алгоритмы поиска имеют одинаковую асимптотическую сложность, но в силу входных данных и особенностей каждого алгоритма – поиск с итеративным углублением будет занимать больше времени, чем остальные алгоритмы.

***Информативный поиск***

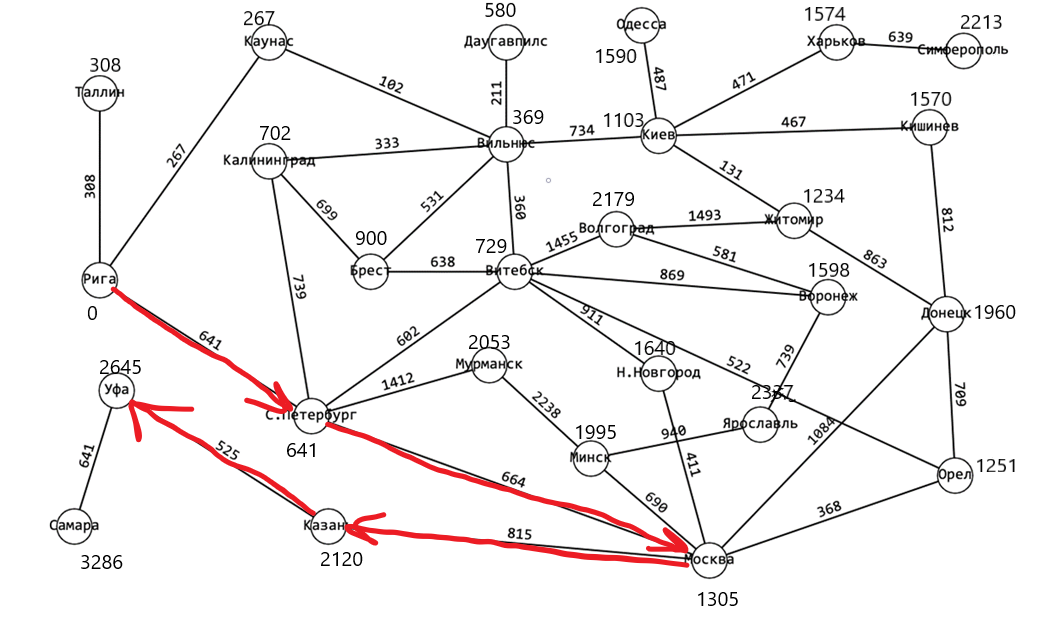
map<string, map<string, int>> straight\_ways = **{** {"Ufa", {  
 make\_pair("Brest", 2147),  
 make\_pair("Vilnius", 1962),  
 make\_pair("Vitebsk", 1641),  
 make\_pair("Voronezh", 1168),  
 make\_pair("Volgograd", 1035),  
 make\_pair("Nizhniy Novgorod", 774),  
 make\_pair("Daugavpils", 1860),  
 make\_pair("Kaliningrad", 2260),  
 make\_pair("Kaunas", 2044),  
 make\_pair("Kiev", 1777),  
 make\_pair("Zhytomyr", 1907),  
 make\_pair("Donetsk", 1462),  
 make\_pair("Chisinau", 2076),  
 make\_pair("Saint-Petersburg", 1637),  
 make\_pair("Riga", 1992),  
 make\_pair("Moscow", 1169),  
 make\_pair("Kazan", 451),  
 make\_pair("Minsk", 1838),  
 make\_pair("Murmansk", 1967),  
 make\_pair("Orel", 1319),  
 make\_pair("Odessa", 1993),  
 make\_pair("Tallin", 1943),  
 make\_pair("Kharkov", 1438),  
 make\_pair("Simferopol", 1900),  
 make\_pair("Yaroslavl", 1045),  
 make\_pair("Samara", 421),  
 }}  
**}**;

map<string, bool> visited;

pair<string, int> greedy\_best\_first\_search\_algorithm(const string &from, const string &to) {  
 string current = from;  
 visited[from] = true;  
 string way = " -> ";  
 int way\_length = 0;  
 while (current != to && !current.empty()) {  
 if (current != from) way += current + " -> ";  
 int min\_len = INT\_MAX;  
 string city;  
 int plus\_way;  
 for (auto &second\_city: cities[current]) {  
 if (straight\_ways[to][second\_city.first] < min\_len && !visited[second\_city.first]) {  
 plus\_way = second\_city.second;  
 city = second\_city.first;  
 min\_len = straight\_ways[to][second\_city.first];  
 }  
 }  
 way\_length += plus\_way;  
 current = city;  
 visited[current] = true;  
 }  
 if (current != to) return make\_pair("Way is not found", 0);  
 else return make\_pair(from + way + to, way\_length);  
}  
  
int a\_star\_search\_algorithm(const string &from, const string &to) {  
 map<int, string> mapa;  
 int function = 0;  
 mapa[function] = from;  
 map<string, int> minimum;  
 while (!mapa.empty()) {  
 auto it = mapa.begin();  
 int current\_function = it->first;  
 if (it->first != 0) {  
 current\_function -= straight\_ways[to][it->second];  
 }  
 string current\_city = it->second;  
 mapa.erase(it);  
 visited[current\_city] = true;  
 if (current\_city == to) {  
 return current\_function;  
 }  
 for (const auto &iter: cities[current\_city]) {  
 string city = iter.first;  
 function = current\_function + straight\_ways[to][city] + iter.second;  
 if (!visited[city] && (minimum.count(city) == 0 || minimum[city] > function)) {  
 minimum[city] = function;  
 parent[city] = current\_city;  
 mapa[function] = city;  
 }  
 }  
 }  
 return INT\_MAX;  
}

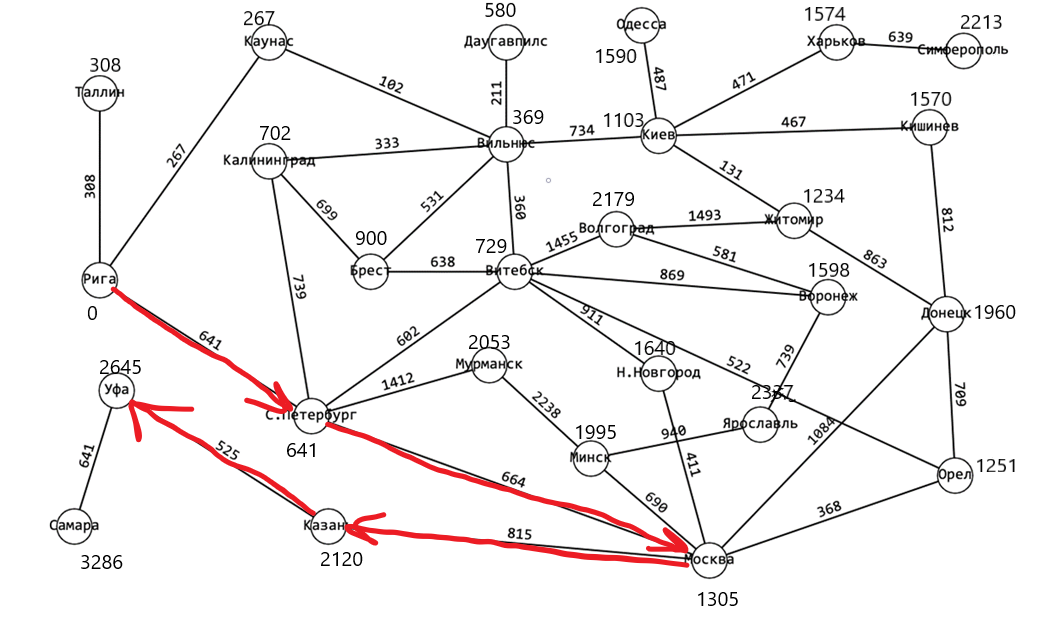
Изначально самый короткий путь:

Рига -> Санкт-Петербург -> Москва -> Казань -> Уфа 2645



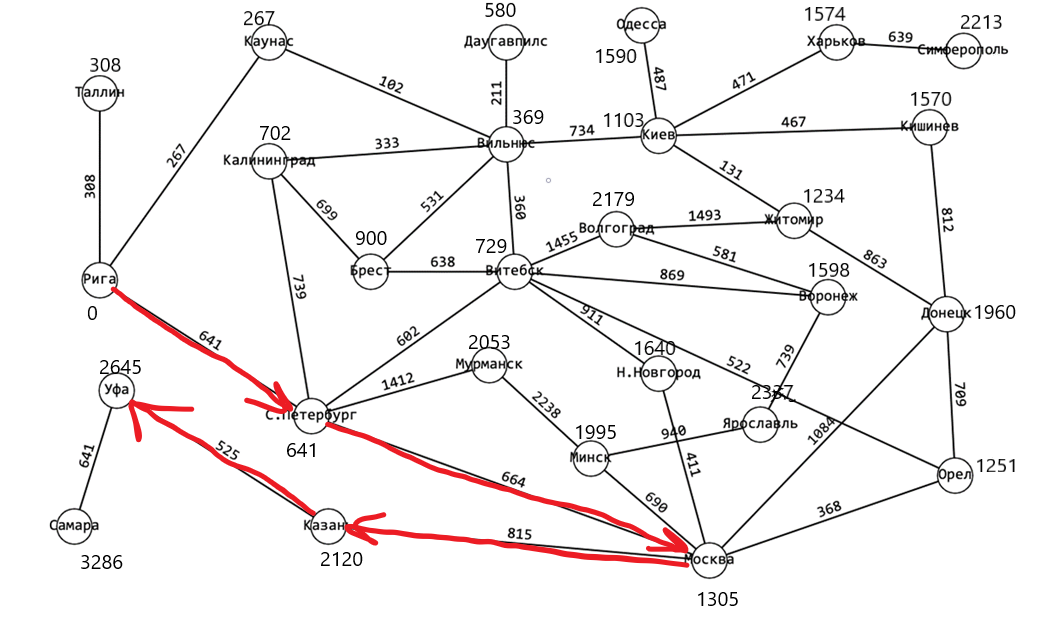
Жадный поиск

Рига -> Санкт-Петербург -> Москва -> Казань -> Уфа 2645



А\*

Рига -> Санкт-Петербург -> Москва -> Казань -> Уфа 2645



Выводы:

|  |  |  |
| --- | --- | --- |
| Алгоритм | Жадный поиск | A\* |
| Сложность | O(N) | O(N^M)  N - глубина решения  M – фактор ветвления |

Жадный алгоритм поиска работает быстрее, чем А\*. В случае моих входных данных оба алгоритма оказались точными (выбрали верный путь) и одинаково быстрыми, но в случае же других данных существует вероятность, что жадный поиск выберет не максимально точный (не наикратчайший по длине пути), а А\* будет работать дольше в силу увеличения ветвления (в сравнении с жадным). А\* выполняет условия полноты и оптимальности, а это значит, что найденное решение обязательно будет оптимальным из всех.