

Факультет программной инженерии и компьютерной техники

Системы искусственного интеллекта

Лабораторная работа №2

Вариант № 5

Преподаватель: Авдюшина Анна Евгеньевна

Выполнил: Хафизов Булат Ленарович

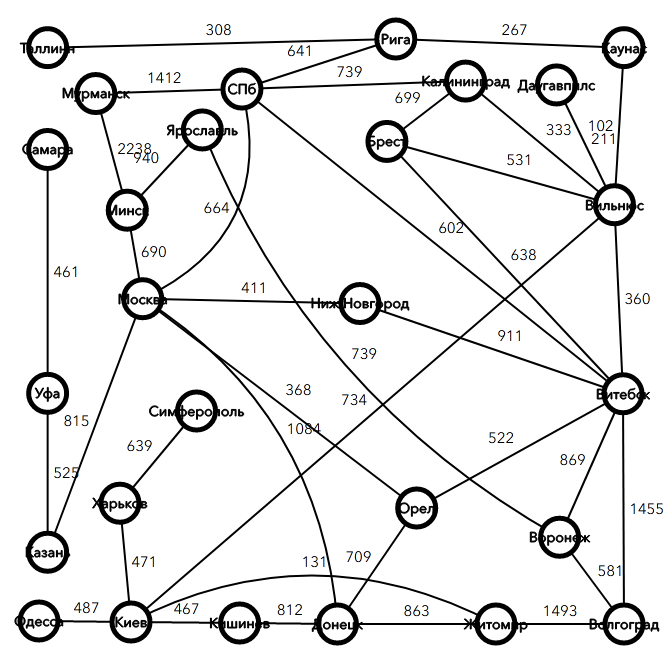
Группа: Р33113

# Задание

Исследование алгоритмов решения задач методом поиска.

Путь от Казани до Таллина.

# Граф



# Код

Main.java

import java.io.\*;  
import java.util.\*;  
import java.util.stream.Collectors;  
  
public class Main {  
  
 private static TreeMap<String, Integer> *idx* = new TreeMap<>();  
 private static TreeMap<Integer, String> *namesByIdx* = new TreeMap<>();  
  
 private static TreeMap<Integer, Integer> *heuristic* = new TreeMap<>();  
 private static int[][] *graph*;  
 private static boolean[] *was*;  
  
 private static int *minDFS* = Integer.*MAX\_VALUE*;  
 private static Stack<String> *minPathDFS* = new Stack<>();  
  
 private static String bfs(int start, int end) {  
 TreeMap<Integer, Integer> path = new TreeMap<>();  
 Queue<Integer> q = new LinkedList<>();  
 path.put(start, -1);  
 q.add(start);  
 while (!q.isEmpty()) {  
 int cur = q.poll();  
 *was*[cur] = true;  
 if (cur != end) {  
 for (int i = 0; i < *idx*.size(); i++) {  
 if (*graph*[cur][i] != 0 && !*was*[i]) {  
 q.add(i);  
 path.put(i, cur);  
 }  
 }  
 } else {  
 break;  
 }  
 }  
 return *printPath*(end, path);  
 }  
  
 private static String bidirectionalBfs(int start, int end) {  
 Queue<Integer> fromStart = new LinkedList<>();  
 fromStart.add(start);  
 Queue<Integer> fromEnd = new LinkedList<>();  
 fromEnd.add(end);  
 List<Integer> wasFromStart = new ArrayList<>();  
 List<Integer> wasFromEnd = new ArrayList<>();  
 wasFromStart.add(start);  
 wasFromEnd.add(end);  
 TreeMap<Integer, Integer> pathFromStart = new TreeMap<>();  
 TreeMap<Integer, Integer> pathFromEnd = new TreeMap<>();  
 pathFromStart.put(start, -1);  
 pathFromEnd.put(end, -1);  
 int intersect = -1;  
 while (!fromStart.isEmpty() && !fromEnd.isEmpty()) {  
 int curFromStart = fromStart.poll();  
 int curFromEnd = fromEnd.poll();  
 if (curFromStart != curFromEnd) {  
 for (int i = 0; i < *idx*.size(); i++) {  
 if (*graph*[curFromStart][i] != 0 && !wasFromStart.contains(i)) {  
 fromStart.add(i);  
 wasFromStart.add(i);  
 pathFromStart.put(i, curFromStart);  
 }  
 }  
 for (int i = 0; i < *idx*.size(); i++) {  
 if (*graph*[curFromEnd][i] != 0 && !wasFromEnd.contains(i)) {  
 fromEnd.add(i);  
 pathFromEnd.put(i, curFromEnd);  
 wasFromEnd.add(i);  
 }  
 }  
 List<Integer> intersection = wasFromStart.stream().distinct().filter(wasFromEnd::contains).collect(Collectors.*toList*());  
 if (!intersection.isEmpty()) {  
 intersect = intersection.get(0);  
 break;  
 }  
 }  
 }  
 Stack<String> ans = new Stack<>();  
 ArrayList<String> listFromStart = new ArrayList<>();  
 int parent = pathFromStart.get(intersect);  
 while (parent != -1) {  
 listFromStart.add(*namesByIdx*.get(parent));  
 parent = pathFromStart.get(parent);  
 }  
 for (int i = listFromStart.size() - 1; i >= 0; i--) {  
 ans.add(listFromStart.get(i));  
 }  
 ArrayList<String> listFromEnd = new ArrayList<>();  
 parent = pathFromEnd.get(intersect);  
 listFromEnd.add(*namesByIdx*.get(intersect));  
 while (parent != -1) {  
 listFromEnd.add(*namesByIdx*.get(parent));  
 parent = pathFromEnd.get(parent);  
 }  
 ans.addAll(listFromEnd);  
 return ans.toString();  
 }  
  
 private static String bestFirstSearch(int start, int end) {  
 PriorityQueue<Heuristic> queue = new PriorityQueue<>((o1, o2) -> Integer.*compare*(o2.getHeuristic(), o1.getHeuristic()));  
 queue.add(new Heuristic(start, 0));  
 TreeMap<Integer, Integer> path = new TreeMap<>();  
 path.put(start, -1);  
 while (!queue.isEmpty()) {  
 int cur = queue.poll().getIdx();  
 *was*[cur] = true;  
 if (cur == end) {  
 break;  
 }  
 for (int i = 0; i < *idx*.size(); i++) {  
 if (*graph*[cur][i] != 0 && !*was*[i]) {  
 queue.add(new Heuristic(i, *heuristic*.get(i)));  
 path.put(i, cur);  
 }  
 }  
 }  
 return *printPath*(end, path);  
 }  
  
 private static String iterativeSearch(int start, int end) {  
 int i = 1;  
 String get;  
 while (true) {  
 get = *dfsWithLimit*(start, end, new Stack<>(), i);  
 if (!get.equals("No ways found")) {  
 break;  
 }  
 i++;  
 }  
 return get;  
 }  
  
 private static String printPath(int end, TreeMap<Integer, Integer> path) {  
 ArrayList<String> list = new ArrayList<>();  
 list.add(*namesByIdx*.get(end));  
 int parent = path.get(end);  
 while (parent != -1) {  
 list.add(*namesByIdx*.get(parent));  
 parent = path.get(parent);  
 }  
 Stack<String> ans = new Stack<>();  
 for (int i = list.size() - 1; i >= 0; i--) {  
 ans.add(list.get(i));  
 }  
 return ans.toString();  
 }  
  
 private static String A\_Search(int start, int end) {  
 PriorityQueue<Heuristic> queue = new PriorityQueue<>((o1, o2) -> Integer.*compare*(o1.getHeuristic(), o2.getHeuristic()));  
 TreeMap<Integer, Integer> cost\_so\_far = new TreeMap<>();  
 cost\_so\_far.put(start, 0);  
 queue.add(new Heuristic(start, 0));  
 TreeMap<Integer, Integer> path = new TreeMap<>();  
 path.put(start, -1);  
 while (!queue.isEmpty()) {  
 Heuristic curH = queue.poll();  
 int cur = curH.getIdx();  
 if (cur == end) {  
 break;  
 }  
 for (int i = 0; i < *idx*.size(); i++) {  
 int new\_cost = cost\_so\_far.get(cur) + *graph*[cur][i];  
 if (*graph*[cur][i] != 0 && (!cost\_so\_far.containsKey(i) || new\_cost < cost\_so\_far.get(i))) {  
 cost\_so\_far.put(i, new\_cost);  
 queue.add(new Heuristic(i, new\_cost + *heuristic*.get(cur)));  
 path.put(i, cur);  
 }  
 }  
 }  
 return *printPath*(end, path);  
 }  
  
 private static void dfs(int cur, int end, int depth, Stack<String> path, int limit) {  
 if (depth == limit + 1) return;  
 if (cur == end) {  
 if (depth + 1 < *minDFS*) {  
 *minDFS* = depth + 1;  
 *minPathDFS*.clear();  
 *minPathDFS*.addAll(path);  
 *minPathDFS*.add(*namesByIdx*.get(end));  
 }  
 return;  
 }  
 *was*[cur] = true;  
 path.add(*namesByIdx*.get(cur));  
 for (int i = 0; i < *idx*.size(); i++) {  
 if (*graph*[cur][i] != 0 && !*was*[i]) {  
 *dfs*(i, end, depth + 1, path, limit);  
 *was*[i] = false;  
 }  
 }  
 path.pop();  
 }  
  
 private static String dfsWithLimit(int cur, int end, Stack<String> path, int limit) {  
 *minPathDFS* = new Stack<>();  
 *minDFS* = Integer.*MAX\_VALUE*;  
 *dfs*(cur, end, 0, path, limit);  
 return *minPathDFS*.isEmpty() ? "No ways found" : *minPathDFS*.toString();  
 }  
  
 public static void main(String[] args) throws IOException {  
 *setIndexes*();  
 *graph* = new int[*idx*.size()][*idx*.size()];  
 *makeGraph*();  
 *setHeuristic*();  
 *was* = new boolean[*idx*.size()];  
 System.*out*.println(*bfs*(*idx*.get("Казань"), *idx*.get("Таллин")));  
 *clearWas*();  
 *dfs*(*idx*.get("Казань"), *idx*.get("Таллин"), 0, new Stack<>(), Integer.*MAX\_VALUE*);  
 System.*out*.println(*minPathDFS*);  
 *clearWas*();  
 System.*out*.println(*iterativeSearch*(*idx*.get("Казань"), *idx*.get("Таллин")));  
 *clearWas*();  
 System.*out*.println(*dfsWithLimit*(*idx*.get("Казань"), *idx*.get("Таллин"), new Stack<>(), 4));  
 *clearWas*();  
 System.*out*.println(*bidirectionalBfs*(*idx*.get("Казань"), *idx*.get("Таллин")));  
 *clearWas*();  
 System.*out*.println(*bestFirstSearch*(*idx*.get("Казань"), *idx*.get("Таллин")));  
 *clearWas*();  
 System.*out*.println(*A\_Search*(*idx*.get("Казань"), *idx*.get("Таллин")));  
 }  
  
 private static void clearWas() {  
 Arrays.*fill*(*was*, false);  
 }  
  
 private static void setIndexes() throws IOException {  
 int curIdx = 0;  
 BufferedReader bf = new BufferedReader(new FileReader("cities.txt"));  
 String line = bf.readLine();  
 while (line != null) {  
 *idx*.put(line, curIdx);  
 *namesByIdx*.put(curIdx, line);  
 curIdx++;  
 line = bf.readLine();  
 }  
 bf.close();  
 }  
  
 private static void makeGraph() throws IOException {  
 BufferedReader bf = new BufferedReader(new FileReader("graph.txt"));  
 String line = bf.readLine();  
 while (line != null) {  
 String[] splited = line.split(" ");  
 *graph*[*idx*.get(splited[0])][*idx*.get(splited[1])] = Integer.*parseInt*(splited[2]);  
 *graph*[*idx*.get(splited[1])][*idx*.get(splited[0])] = Integer.*parseInt*(splited[2]);  
 line = bf.readLine();  
 }  
 bf.close();  
 }  
  
 private static void setHeuristic() throws IOException {  
 BufferedReader bf = new BufferedReader(new FileReader("heuristic.txt"));  
 String line = bf.readLine();  
 while (line != null) {  
 String[] splited = line.split(" ");  
 *heuristic*.put(*idx*.get(splited[0]), Integer.*parseInt*(splited[1]));  
 line = bf.readLine();  
 }  
 bf.close();  
 }  
}

Heuristic.java

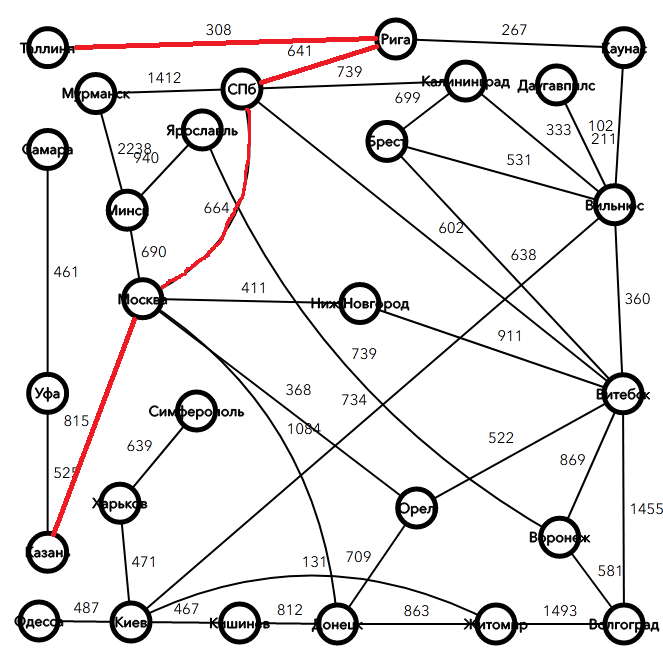
public class Heuristic {  
 private int idx;  
 private int heuristic;  
  
 public Heuristic(int idx, int heuristic) {  
 this.idx = idx;  
 this.heuristic = heuristic;  
 }  
  
 public int getIdx() {  
 return idx;  
 }  
  
 public int getHeuristic() {  
 return heuristic;  
 }  
}

# Вывод

Т.к. последовательность действий можно выразить в виде дерева решений, именно такой подход часто используют в системах искусственного интеллекта для принятия решений. В лабораторной работе необходимо было реализовать простой ИИ, принимающий решение, в какую из вершин графа двигаться, чтобы достичь цели. Основными вопросами, которыми он задаётся являются «Посещена ли вершина?» и «Лучшая ли это вершина из доступных для посещения?» (второй вопрос актуален только для информированного поиска). Для реализации подобного поведения часто используется структура данных «Очередь с приоритетом», где в голове очереди находится самый «выгодный» с точки зрения алгоритма элемент. Так как ответить на эти вопросы не всегда является возможным или возможным однозначно, не все алгоритмы принятия решений являются полными (т.е. могут найти ответ).

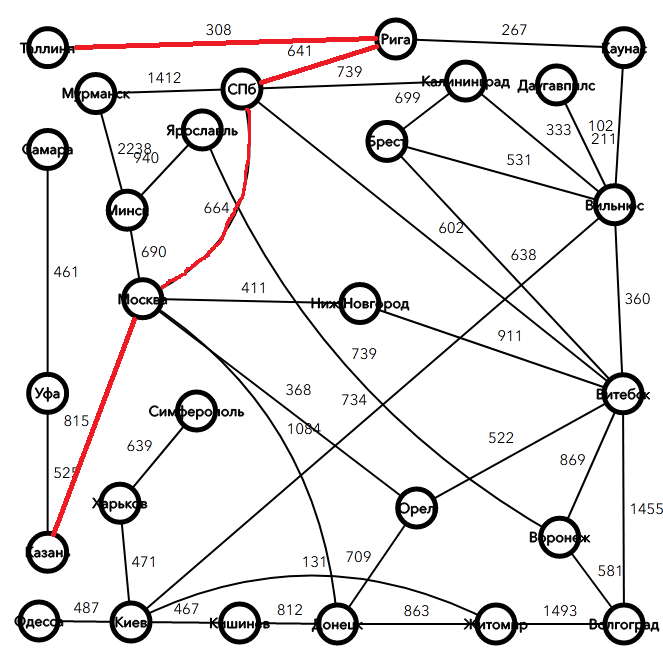
DFS

Т.к. этот алгоритм обладает информацией только о связях между вершинами, его дерево решений будет самым объёмным, ибо следующий рассматриваемый узел будет браться случайным образом, следовательно путь от начала до него может быть не оптимальным (в виду отсутствия эвристической оценки, отображать дерево решений смысла нет).



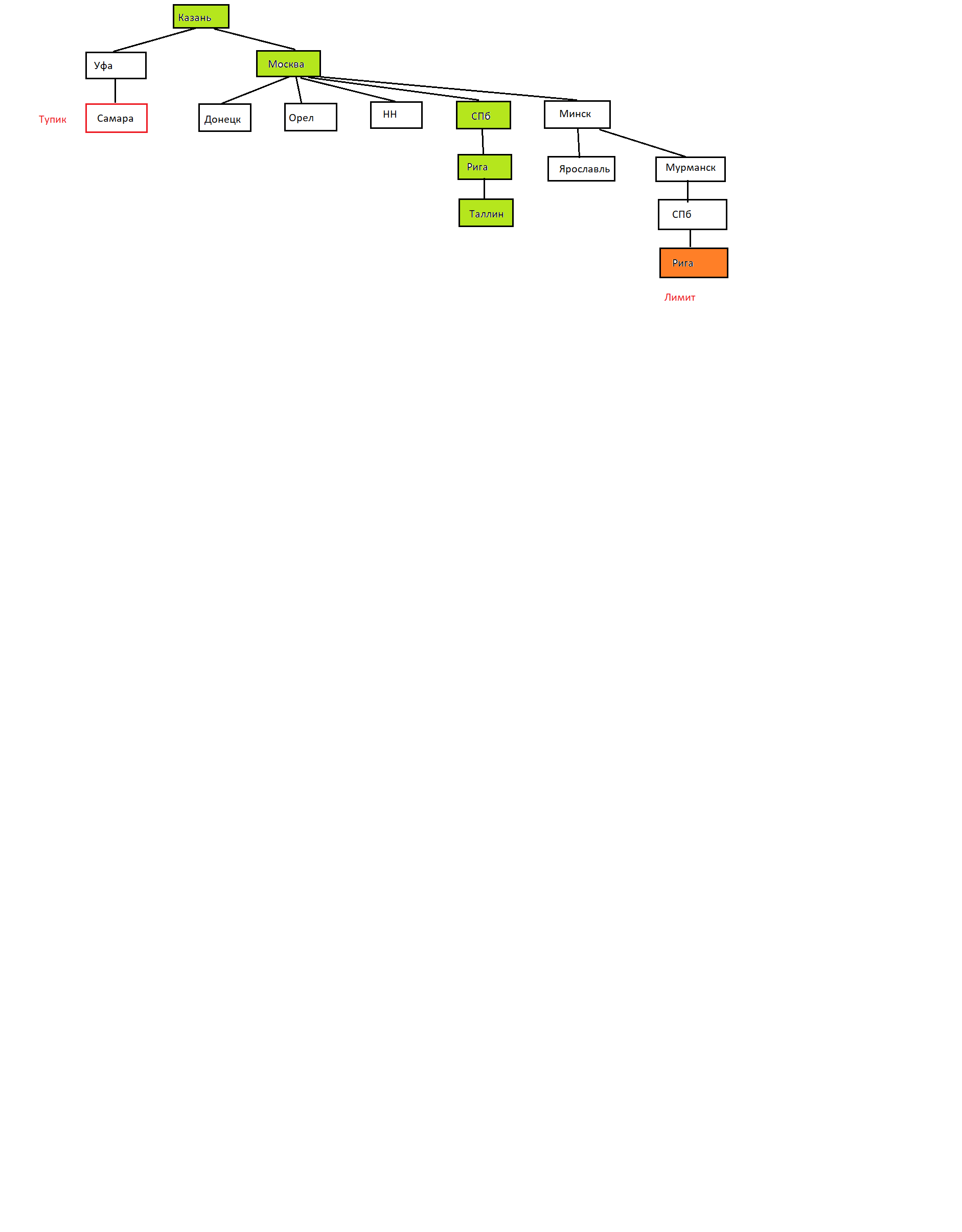
BFS

BFS отличается от DFS порядком обхода вершин, но также не принимает решений, основанный на оценки пройденного пути до вершины. Быстрее обнаружит решение на неглубоких уровнях, но также может пойти по неоптимальному пути и выдать это за решение.



DLS

DLS поступит умнее. Как только будет достигнуть ограничительная глубина, цепочка поиск будет разорвана и будет предпринята попытка найти другой путь. Запустим алгоритм с ограничением в 4 шага.

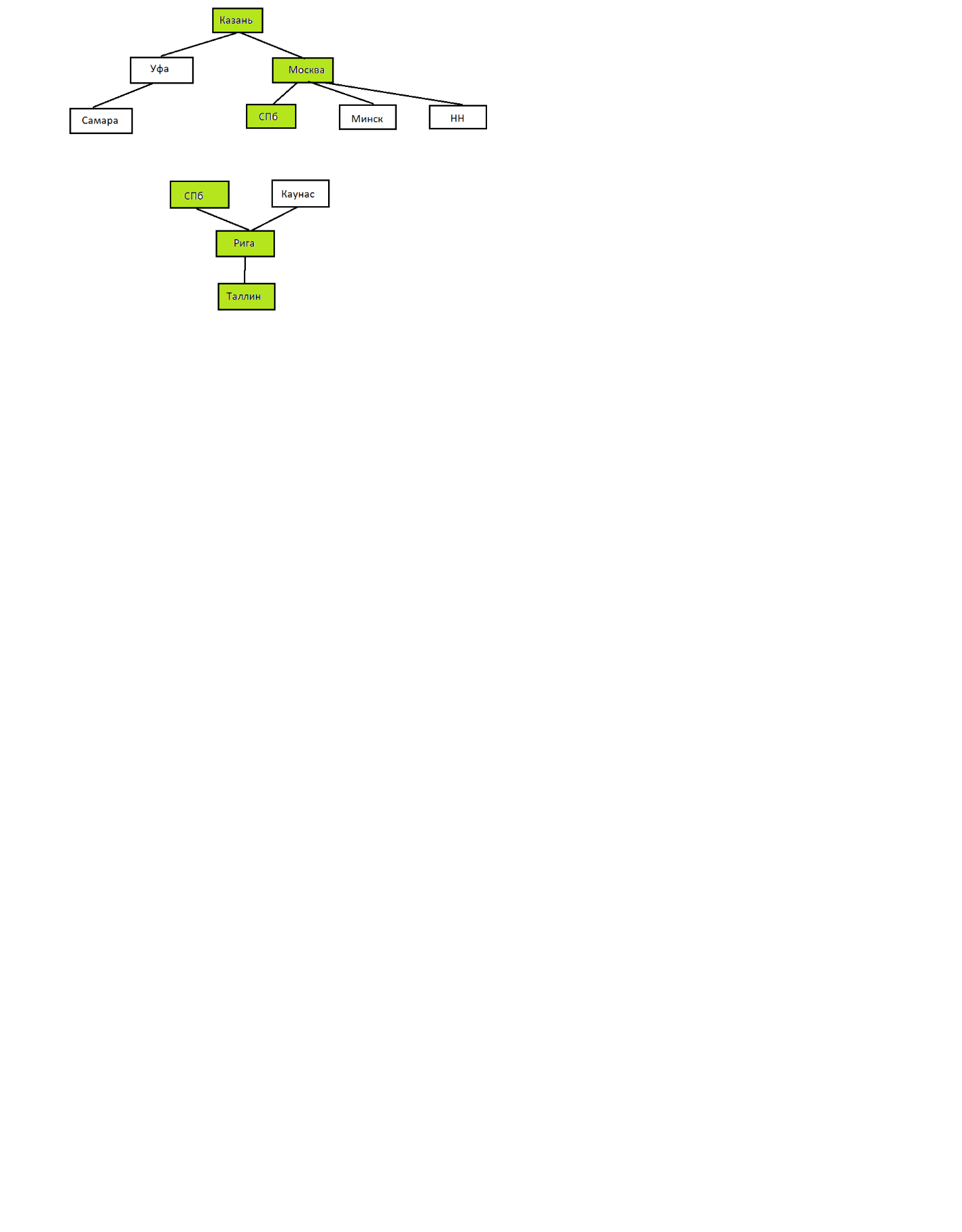


IDDFS

Подобным образом отработает и IDDFS, но сначала запустится с ограничением 1, 2, 3, а на 4-ый найдет то же самое решение.

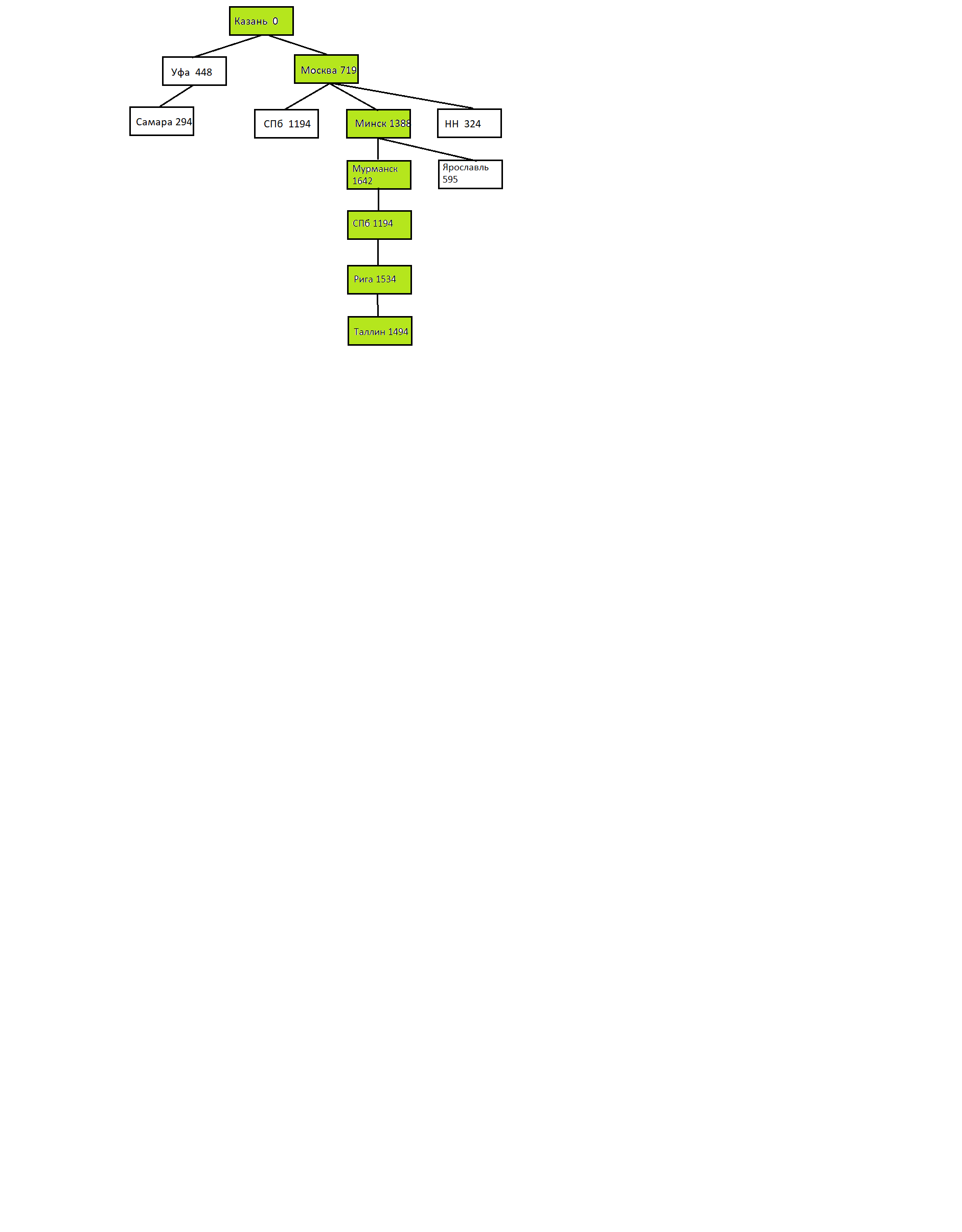
BDS

Один поиск запускается с начального узла, другой – с конечного. Основан на BFS, но как только в оба BFS-а находят общий узел, поиск сворачивается.



BFS

На каждой итерации алгоритм будет выбирать ту вершину из соседей текущей, которая максимально приблизит нас к конечной вершине (максимальная эвристика).



A\*

A\* отличается от BFS функцией оценки. Теперь это сумма эвристики и пути от стартовой вершины (в реальных приложениях используют теорему Пифагора, а не сумму, а сам алгоритм для определения пути движения противников к игроку).

