РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

дисциплина: Интеллектуальные системы

Студент: Койфман К.Д.

Группа: НПИбд-01-21

№ ст. билета: 1032217058

МОСКВА

2022 г.

Введение.

Цель работы.
Изучение работы и реализация алгоритмов “Поиска в ширину”,” A*” и “Дейкстры”.

Задачи.

1. Реализовать алгоритмы - A*, Дейкстры.

2. Реализовать поддержку различных эвристических функций - Euclid, Manhattan, Octile.

3. Реализовать поддержку коэффициента эвристики (f = g+w*h)

4. Реализовать поддержку 4- и 8-связных графов. (При проверке диагональных переходов необходимо учитывать проходимость смежных вершин).

5. Протестировать реализацию на всех заданиях, используя различные комбинации входных параметров:

5.1. BFS (другие параметры игнорируются)

5.2. Dijkstra (другие параметры игнорируются)

5.3. AStar, connections=4, metrictype=Euclid, hweight=1

5.4. AStar, connections=4, metrictype=Manhattan, hweight=1

5.5. AStar, connections=8, metrictype=Euclid, hweight=1

5.6. AStar, connections=8, metrictype=Octile, hweight=1

5.7. AStar, connections=8, metrictype=Manhattan, hweight=1

5.8. AStar, connections=8, metrictype=Octile, hweight=2

Ход работы.

1 задание.

Реализуем алгоритм Дейкстры и A*.

Определяем начальные условия работы алгоритма и формируем ключевые списки “OPEN” и “CLOSED” (рис.1).

```
117  class AStar//алгоритмы "A*" и "Дейкстры"
118  {
119  public:
120      //Данная функция находит путь от start до goal на карте grid
121      Result find_path(Node start, Node goal, Map grid, std::string metrictype = "Octile", int connections = 8, double hweight = 1)
122      {
123          //Создаём переменную time_now для подсчёта времени, за которое происходит поиск оптимального пути и его полное построение
124          auto time_now = std::chrono::high_resolution_clock::now();
125
126          /* Переменная result будет хранить данные о пути, который будет
127          постепенно строится( например, "время построения", "длина пути" и т.п. */
128          Result result;
129
130          int steps = 0;
131
132          //определим вес и значение эвристики для стартовой вершины
133          start.g = 0;
134          start.h = count_Hvalue(start, goal, metrictype, hweight);
135
136          //В списке OPEN будут хранится затронутые вершины( в очереди на рассмотрение)(сразу добавляем в неё start)
137          std::list<Node> OPEN;
138          OPEN.push_back(start);
139
140          //В списке CLOSED будут хранится просмотренные вершины( уже посещённые)(в него тоже добавляем start)
141          std::set<Node> CLOSED;
142          CLOSED.insert(start);
143
144          //Переменная pathfound будет использоваться для прекращения поиска пути и для начала его постройки
145          bool pathfound = false;
146
```

рис.1

Открываем цикл while, в течении работы которого алгоритм будет просматривать и раскрывать выгодные для перехода вершины(рис.2):

```
146
147          //Пока список OPEN не пуст(т.е. все вершины будут рассмотрены) и НЕ pathfound=false(т.е. путь будет найден)
148          while (!OPEN.empty() && !pathfound)
149          {
150              /*current - переменная будет хранить данные о ТЕКУЩЕЙ вершине(т.е.рассматриваемой в данный момент,
151              точке) такой, которая содержит НАИМЕНЬШЕЕ F-значение, из списка OPEN(которое будет определено в цикле ниже)*/
152              Node current = OPEN.front();
153              for (std::list<Node>::iterator iter = OPEN.begin(); iter != OPEN.end(); ++iter)
154                  if ((*iter).f < current.f) current = (*iter);
155
156              //согласно алгоритму предварительно удаляем вершину из OPEN(т.о. отмечаем её как "уже рассмотренную")
157              OPEN.remove(current);
158
159              //и добавляем её в CLOSED, как уже "рассмотренную"
160              CLOSED.insert(current);
161
162              steps++;
163
164              //если текущая вершина является целевой, тогда..
165              if (current == goal)
166              {
167                  //перестравиваем конечный путь из start в goal и сохраняем его в path
168                  result.path = reconstruct_path(current);
169
170                  //сохраняем стоимость пути
171                  result.cost = current.f;
172
173                  //т.о. путь найден
174                  pathfound = true;
175                  break;
176              }
177
```

рис.2

Найдём вершины, соседние для текущей, доступные для перехода, и проверим не были ли они образованы и/или раскрыты ранее(рис.3):

```
178             //находим соседние(т.е. доступные для перемещения) вершины
179             auto neighbors = grid.get_neighbors(current, connections);
180
181             //Просматриваем все вершины, в которые можно переместиться из current
182             for (auto n : neighbors)
183             {
184                 //если рассматриваемая вершина n отсутствует в списке CLOSED, то тогда..
185                 if (CLOSED.find(n) == CLOSED.end())
186                 {
187                     //..рассчитываем и сохраняем вес перехода из рассматриваемой вершины n
188                     n.g = current.g + count_Gvalue(current, n);
189
190                     //..рассчитываем и сохраняем значение эвристики для рассматриваемой(соседней к текущей) вершины
191                     n.h = count_Hvalue(n, goal, metrictype, hweight);
192
193                     //вычисляем F-значение
194                     n.f = n.g + n.h;
195
196                     //std::cout << "F-value: " << n.f << std::endl;
197
198                     /*сохраняем для данной вершины в качестве "родителя" точку current(т.е.обозначаем,
199                     что переход в вершину (n) был произведён из вершины (current) ),*/
200                     n.parent = &(*CLOSED.find(current));
201
202                     //дабавляем данную вершину в OPEN, чтобы далее рассматривать её как current при следующей итерации
203                     OPEN.push_back(n);
204
205                     //также дабавляем данную вершину в CLOSED, предварительно обозначая, что она УЖЕ рассмотрена
206                     CLOSED.insert(n);
207                 }
208
```

рис.3

Проверим, является ли рассматриваемая вершина дубликатом (т.к. она уже могла быть рассмотрена и наделена f-значением). Если – да, то тогда необходимо проверить является ли новое f-значение меньше того, которое было сохранено ранее(рис.4):

```
209                 //если рассматриваемая вершина есть в списке OPEN(т.е. уже была рассмотрена ранее), то тогда..
210                 for (std::list<Node>::iterator iter = OPEN.begin(); iter != OPEN.end(); ++iter)
211                     if (n == *iter)
212                     {
213                         //..рассчитываем и сохраняем вес перехода из рассматриваемой вершины n
214                         n.g = current.g + count_Gvalue(current, n);
215
216                         //..рассчитываем и сохраняем значение эвристики для рассматриваемой(соседней к текущей) вершины
217                         n.h = count_Hvalue(n, goal, metrictype, hweight);
218
219                         //вычисляем F-значение
220                         n.f = n.g + n.h;
221
222                         /*если полная стоимость пути(т.е. сумма g-значения,
223                         точной уже накопленной стоимости от start до n, и
224                         h-значения, значения эвристики, предполагаемой стоимости пути от n до goal) найденная для новой n
225                         меньше, чем хранящаяся в OPEN, то тогда передаём её св-ва в вершину в списке OPEN*/
226                         if (n.f < (*iter).f) (*iter).f = n.f;
227                     }
228
229             }
230         }
231         //сохраняем данные, собранные в ходе работы алгоритма
232         result.steps = steps;
233         result.nodes_created = CLOSED.size();
234         result.runtime = (std::chrono::high_resolution_clock::now() - time_now).count() / 1e+9;
235         return result;
236     }
```

рис.4

Функции, применённые при работе алгоритма(рис.5):

```
    //функция, которая вычисляет расстояние между текущей и рассматрвиаемой вершинами
    double count_Gvalue(Node current, Node neighbor)
    {
        if ((std::abs(neighbor.j - current.j) == 1) && (std::abs(neighbor.i - current.i) == 1)) return std::sqrt(2);
        else return 1;
    }

    //функция, которая вычисляет значение эвристики, исходя из выбранного вида расчёта расстояния, указанного в xml-файле
    double count_Hvalue(Node current, Node goal, std::string metrictype, int hweight)
    {
        if (metrictype == "Manhattan")
            return hweight * (abs(goal.i - current.i) + abs(goal.j - current.j));

        else if (metrictype == "Octile")
            return hweight * (abs(abs(goal.i - current.i) - abs(goal.j - current.j)) + sqrt(2 * hweight) * std::min(abs(goal.i - current.i), abs(goal.j - current.j)));

        else if (metrictype == "Euclidean")
            return hweight * sqrt((goal.i - current.i) * (goal.i - current.i) + (goal.j - current.j) * (goal.j - current.j));
    }

    //функция, которая перестраивает и сохраняет путь от start до goal
    std::list<Node> reconstruct_path(Node n)
    {
        //TODO - reconstruct path using back pointers
        std::list<Node> path;
        while (n.parent != nullptr)
        {
            path.push_front(n);
            n = *n.parent;
        }
        path.push_front(n);
        return path;
    }
};
```

рис.5

2,3 задание.

Реализуем поддержку эвристических функций - Euclid, Manhattan, Octile и поддержку коэффициента эвристики (f = g+w*h) (рис.6):

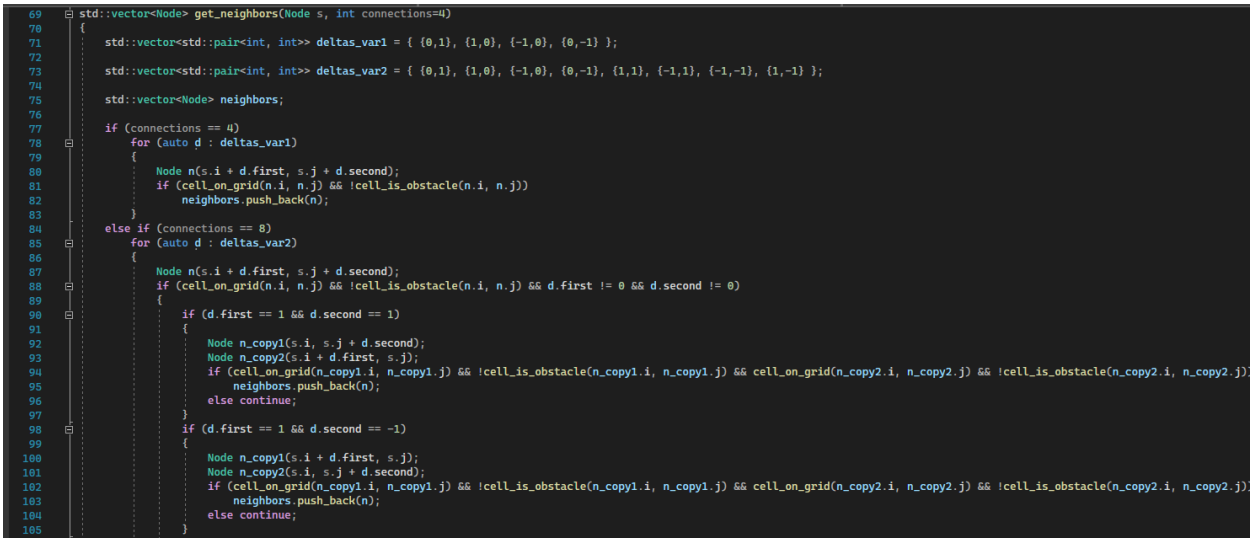
```
//функция, которая вычисляет значение эвристики, исходя из выбранного вида расчёта расстояния, указанного в xml-файле
double count_Hvalue(Node current, Node goal, std::string metrictype, int hweight)
{
    if (metrictype == "Manhattan")
        return hweight * (abs(goal.i - current.i) + abs(goal.j - current.j));

    else if (metrictype == "Octile")
        return hweight * (abs(abs(goal.i - current.i) - abs(goal.j - current.j)) + sqrt(2 * hweight) * std::min(abs(goal.i - current.i), abs(goal.j - current.j)));

    else if (metrictype == "Euclidean")
        return hweight * sqrt((goal.i - current.i) * (goal.i - current.i) + (goal.j - current.j) * (goal.j - current.j));
}
```

рис.6

4 задание.

Реализуем поддержку 4- и 8-связных графов так, чтобы при проверке диагональных переходов учитывалась проходимость смежных вершин (рис.7, рис.8):

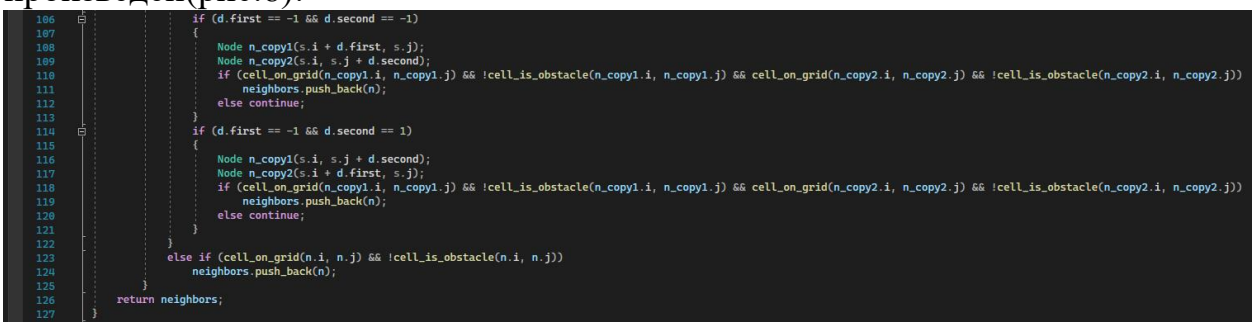
Для этого зададим 2 вектора: 1-ый – для переходов в 4 направлениях, и 2-ой – для переходов в 8 направлениях (т.е. также и по диагонали). Определять, какая вершина является диагональной, мы будем, утверждая, что мы изначально находимся в точке с координатами (0,0) и диагональная вершина будет обладать не нулевыми значениями обеих координат (т.е. x! =0 &&
y! =0):



```
std::vector<Node> get_neighbors(Node s, int connections=4)
{
    std::vector<std::pair<int, int>> deltas_var1 = { {0,1}, {1,0}, {-1,0}, {0,-1} };

    std::vector<std::pair<int, int>> deltas_var2 = { {0,1}, {1,0}, {-1,0}, {0,-1}, {1,1}, {-1,1}, {-1,-1}, {1,-1} };

    std::vector<Node> neighbors;

    if (connections == 4)
        for (auto d : deltas_var1)
        {
            Node n(s.i + d.first, s.j + d.second);
            if (cell_on_grid(n.i, n.j) && !cell_is_obstacle(n.i, n.j))
                neighbors.push_back(n);
        }
    else if (connections == 8)
        for (auto d : deltas_var2)
        {
            Node n(s.i + d.first, s.j + d.second);
            if (cell_on_grid(n.i, n.j) && !cell_is_obstacle(n.i, n.j) && d.first != 0 && d.second != 0)
            {
                if (d.first == 1 && d.second == 1)
                {
                    Node n_copy1(s.i, s.j + d.second);
                    Node n_copy2(s.i + d.first, s.j);
                    if (cell_on_grid(n_copy1.i, n_copy1.j) && !cell_is_obstacle(n_copy1.i, n_copy1.j) && cell_on_grid(n_copy2.i, n_copy2.j) && !cell_is_obstacle(n_copy2.i, n_copy2.j))
                        neighbors.push_back(n);
                    else continue;
                }
                if (d.first == 1 && d.second == -1)
                {
                    Node n_copy1(s.i + d.first, s.j);
                    Node n_copy2(s.i, s.j + d.second);
                    if (cell_on_grid(n_copy1.i, n_copy1.j) && !cell_is_obstacle(n_copy1.i, n_copy1.j) && cell_on_grid(n_copy2.i, n_copy2.j) && !cell_is_obstacle(n_copy2.i, n_copy2.j))
                        neighbors.push_back(n);
                    else continue;
                }
```

рис.7

Далее в зависимости от координат диагональной вершины будем делать следующее. Для проверки того, можно ли совершить переход по диагонали будет служить условие, согласно которому, если 2 соседние вершины, находящиеся по направлению к вершине, в которую планируется совершить переход, являются клетками карты (т.е. находятся в её пределах) и не являются препятствиями, то переход в указанную вершину может быть произведён(рис.8):



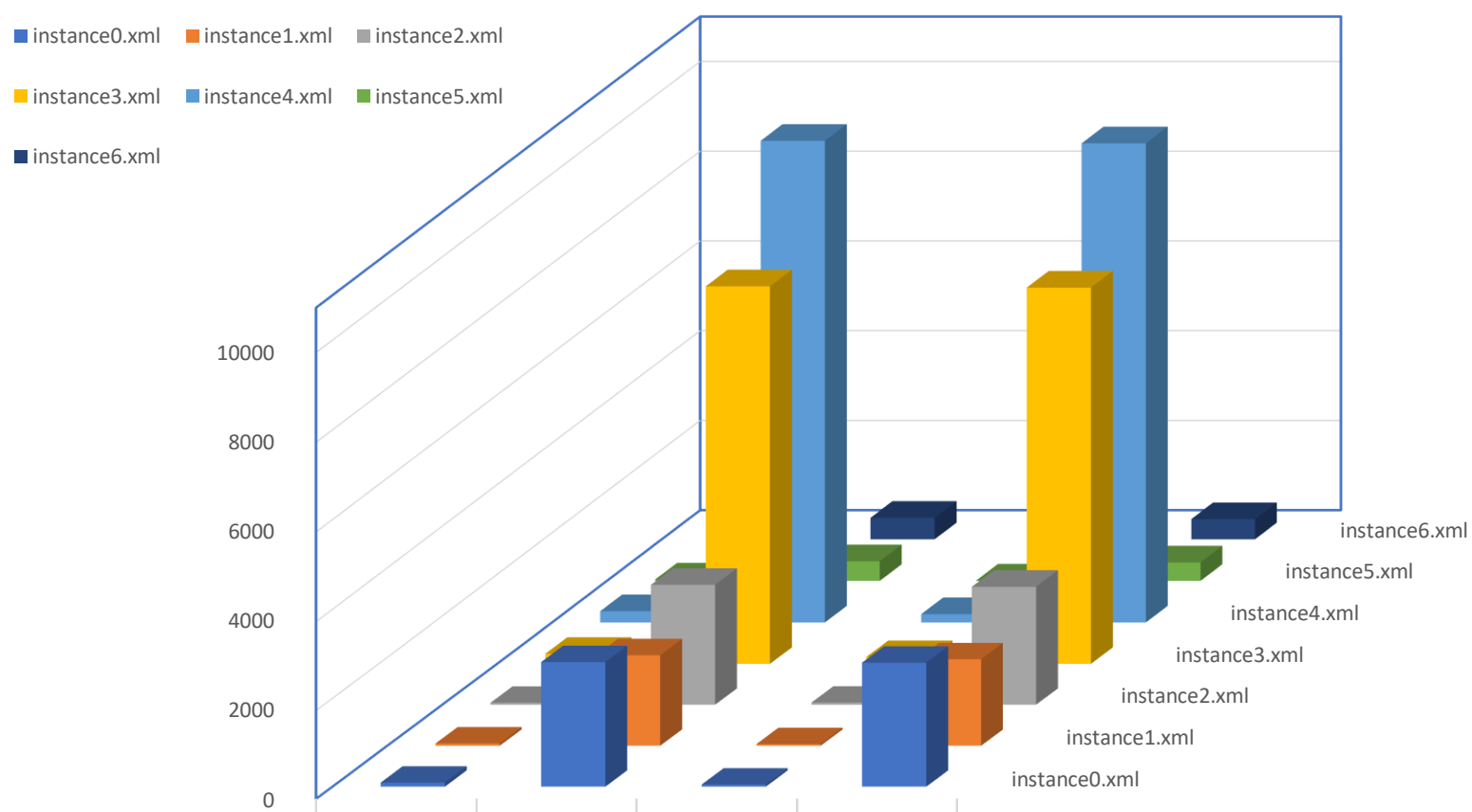
```
                if (d.first == -1 && d.second == -1)
                {
                    Node n_copy1(s.i + d.first, s.j);
                    Node n_copy2(s.i, s.j + d.second);
                    if (cell_on_grid(n_copy1.i, n_copy1.j) && !cell_is_obstacle(n_copy1.i, n_copy1.j) && cell_on_grid(n_copy2.i, n_copy2.j) && !cell_is_obstacle(n_copy2.i, n_copy2.j))
                        neighbors.push_back(n);
                    else continue;
                }
                if (d.first == -1 && d.second == 1)
                {
                    Node n_copy1(s.i, s.j + d.second);
                    Node n_copy2(s.i + d.first, s.j);
                    if (cell_on_grid(n_copy1.i, n_copy1.j) && !cell_is_obstacle(n_copy1.i, n_copy1.j) && cell_on_grid(n_copy2.i, n_copy2.j) && !cell_is_obstacle(n_copy2.i, n_copy2.j))
                        neighbors.push_back(n);
                    else continue;
                }
            }
            else if (cell_on_grid(n.i, n.j) && !cell_is_obstacle(n.i, n.j))
                neighbors.push_back(n);
        }
    return neighbors;
}
```

рис.8

5 задание.

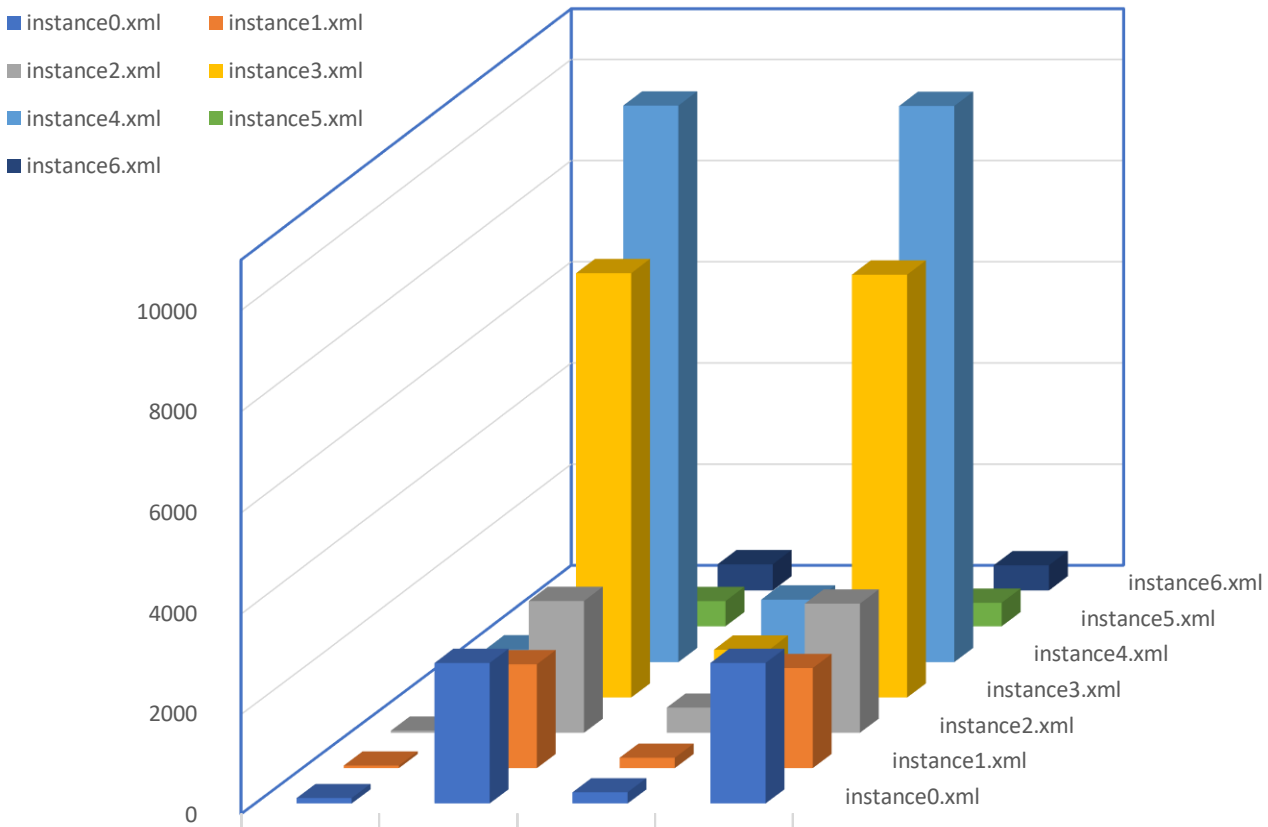
Протестируем реализацию на всех заданиях, используя различные комбинации входных параметров:

5.1. BFS (другие параметры игнорируются):



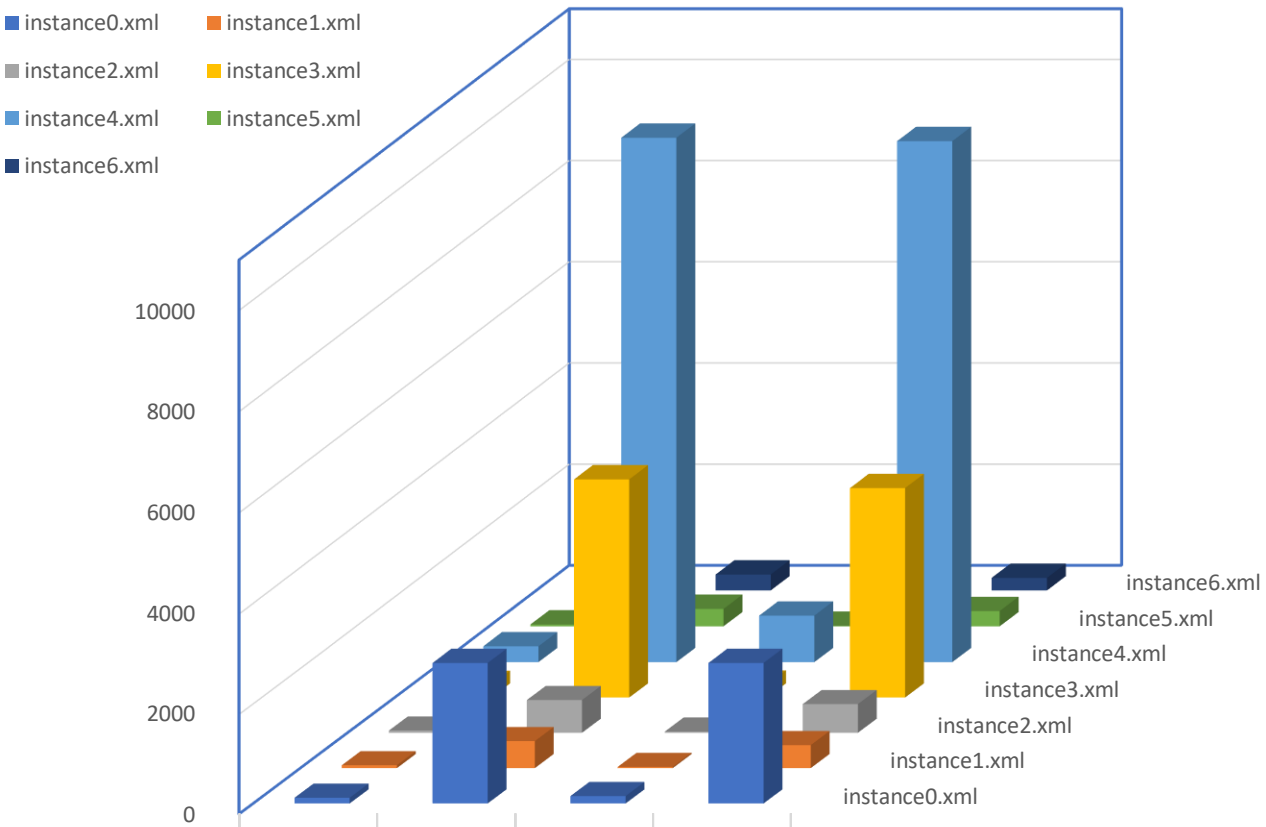
	Стоимость пути	Число раскрытых вершин	Время работы алгоритма(мл сек.)	Число итераций
instance0.xml	92	2820	47	2806
instance1.xml	51	2046	37	1956
instance2.xml	44	2691	49	2648
instance3.xml	226	8418	153	8389
instance4.xml	251	10734	185	10678
instance5.xml	30	437	6	410
instance6.xml	26	478	7	449

5.2. Dijkstra (другие параметры игнорируются):



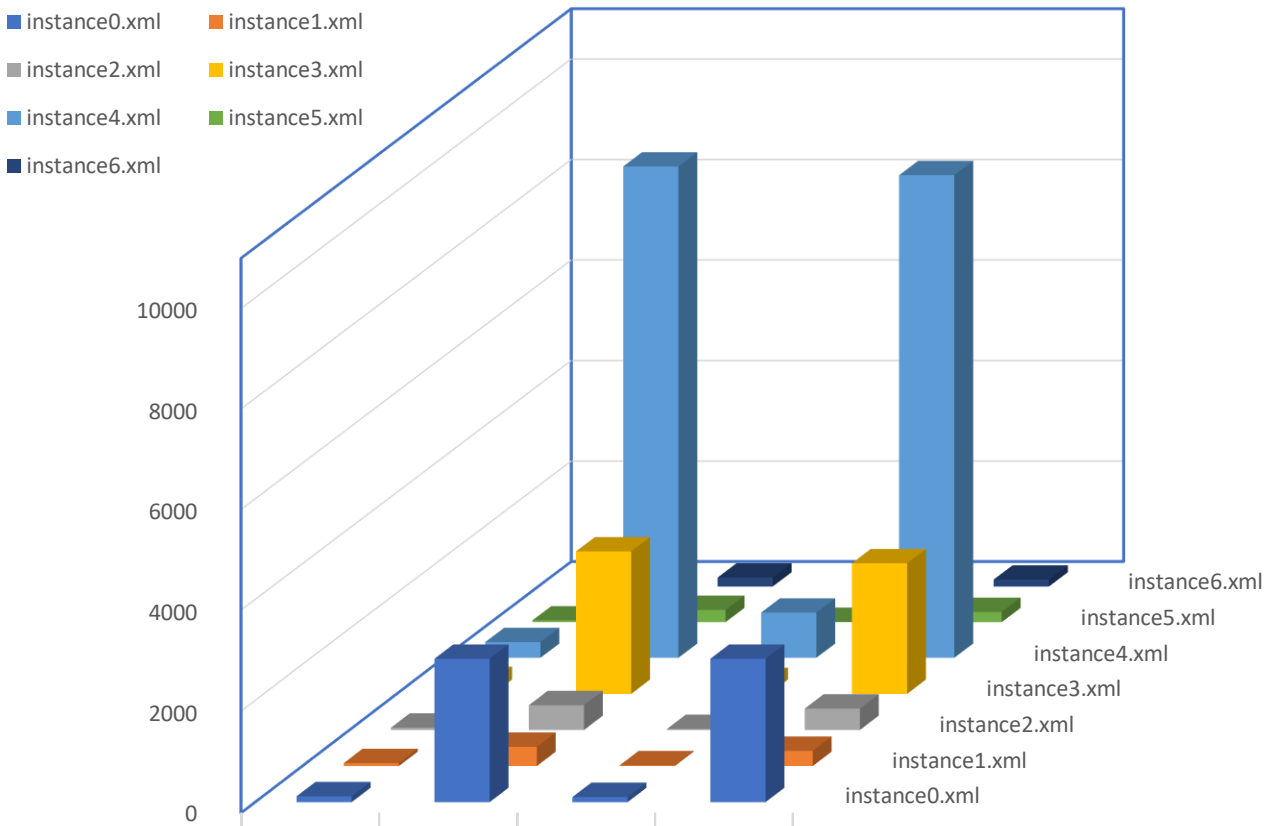
	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	108	2820	224	2820
instance1.xml	51	2085	208	2009
instance2.xml	44	2628	506	2575
instance3.xml	223	8397	956	8368
instance4.xml	251	11053	1236	10993
instance5.xml	30	498	22	464
instance6.xml	26	516	26	496

5.3. AStar, connections=4, metrictype=Euclid, hweight=1



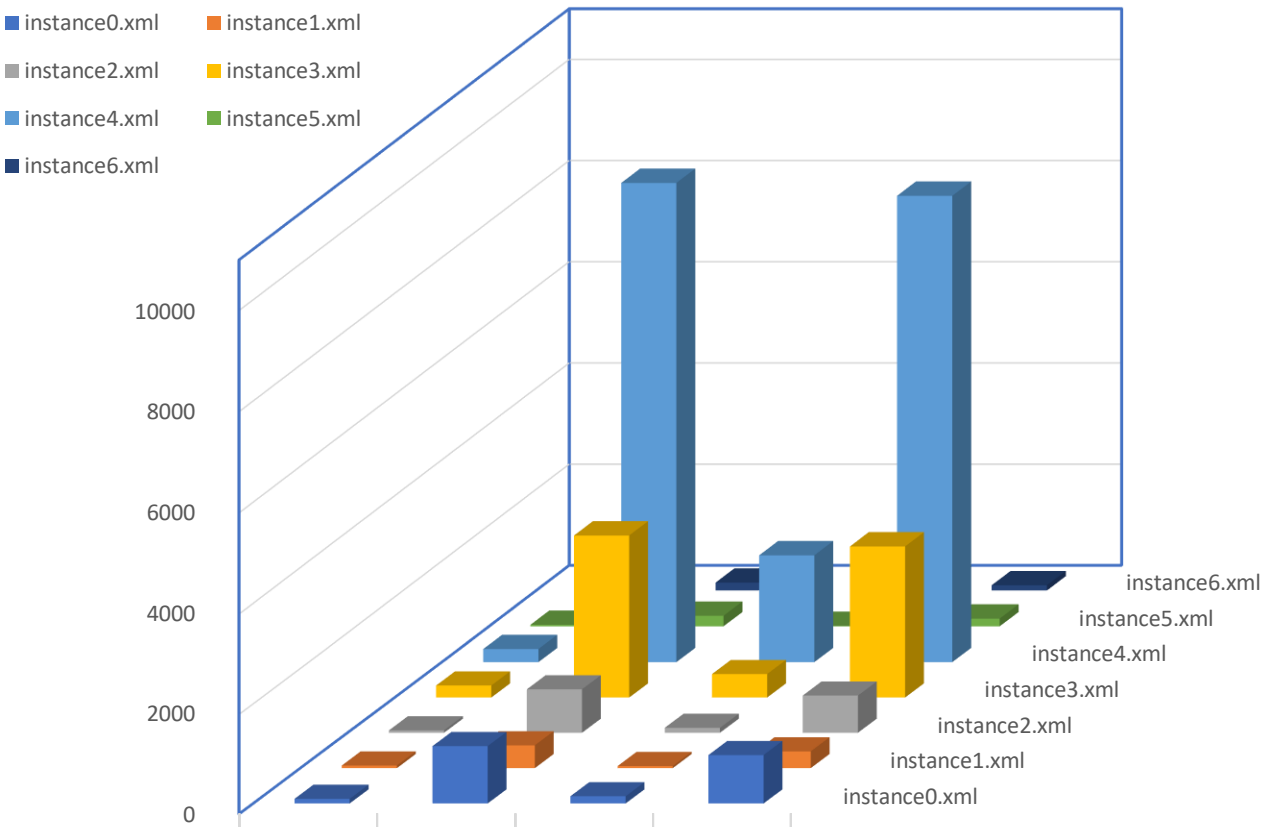
	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	114	2820	148	2820
instance1.xml	57	542	24	461
instance2.xml	45	658	25	576
instance3.xml	255	4322	260	4150
instance4.xml	314	10365	920	10298
instance5.xml	36	344	10	301
instance6.xml	32	314	26	248

5.4. AStar, connections=4, metrictype=Manhattan, hweight=1



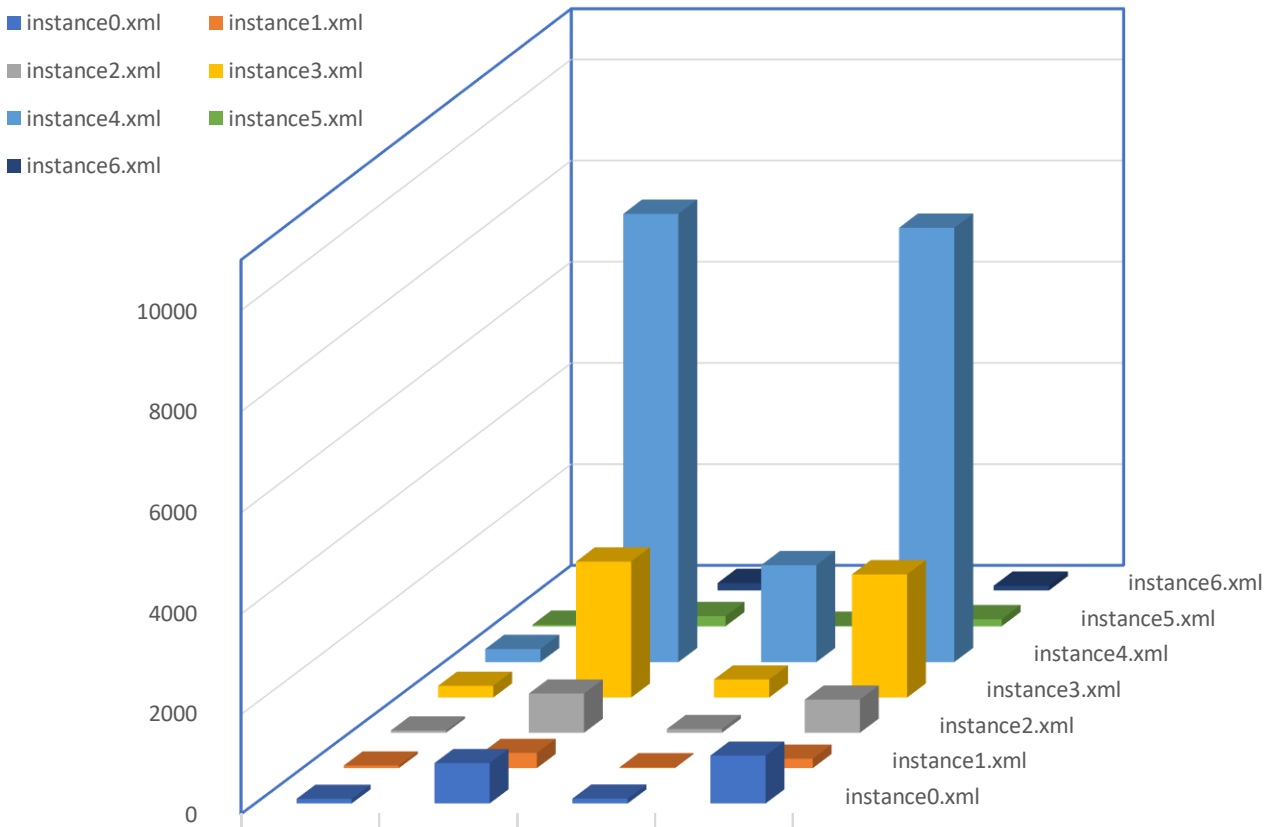
	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	114	2820	100	2820
instance1.xml	57	379	14	298
instance2.xml	45	487	18	420
instance3.xml	255	2827	234	2591
instance4.xml	314	9776	900	9603
instance5.xml	36	244	5	205
instance6.xml	32	178	4	138

5.5. AStar, connections=8, metrictype=Euclid, hweight=1



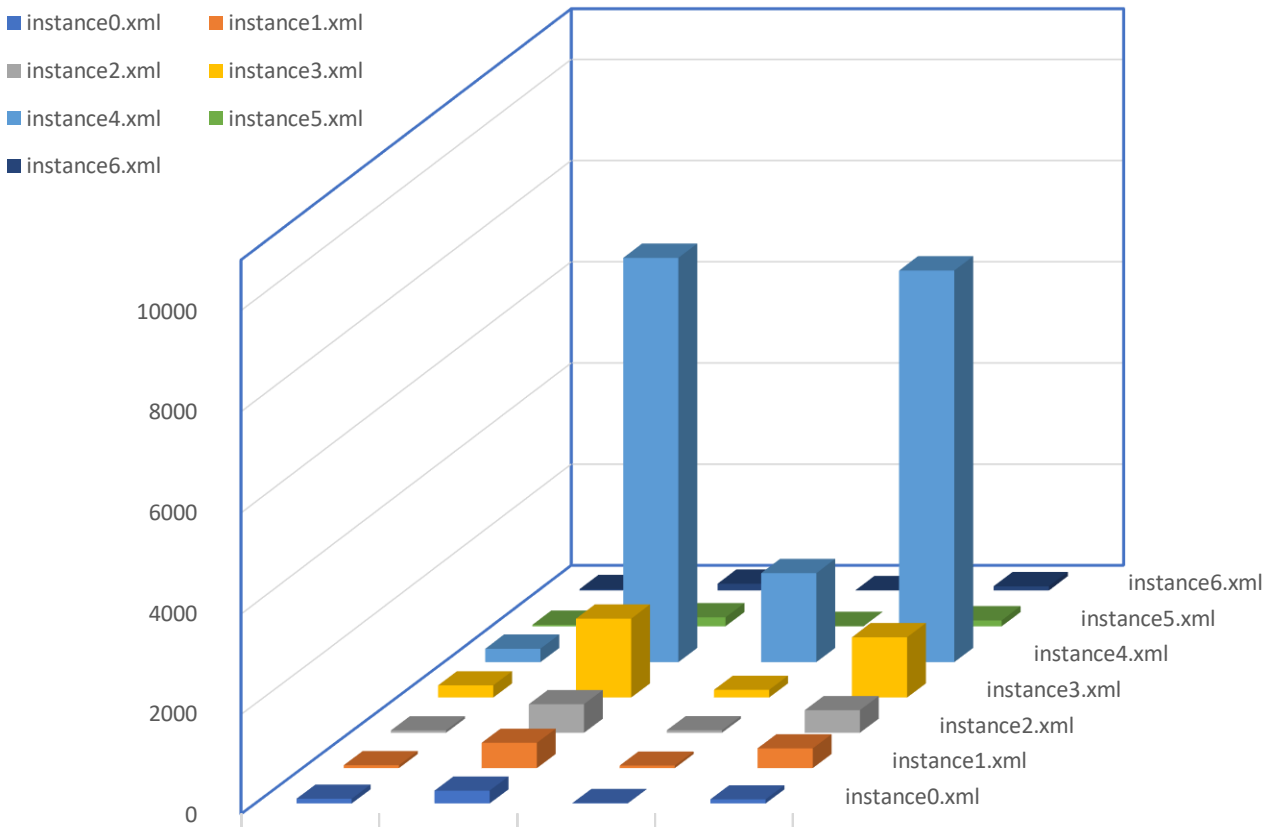
	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	92	1151	144	971
instance1.xml	52	457	41	336
instance2.xml	49	873	100	747
instance3.xml	240	3214	470	2997
instance4.xml	260	9472	2112	9218
instance5.xml	30	208	8	152
instance6.xml	26	154	5	102

5.6. AStar, connections=8, metrictype=Octile, hweight=1



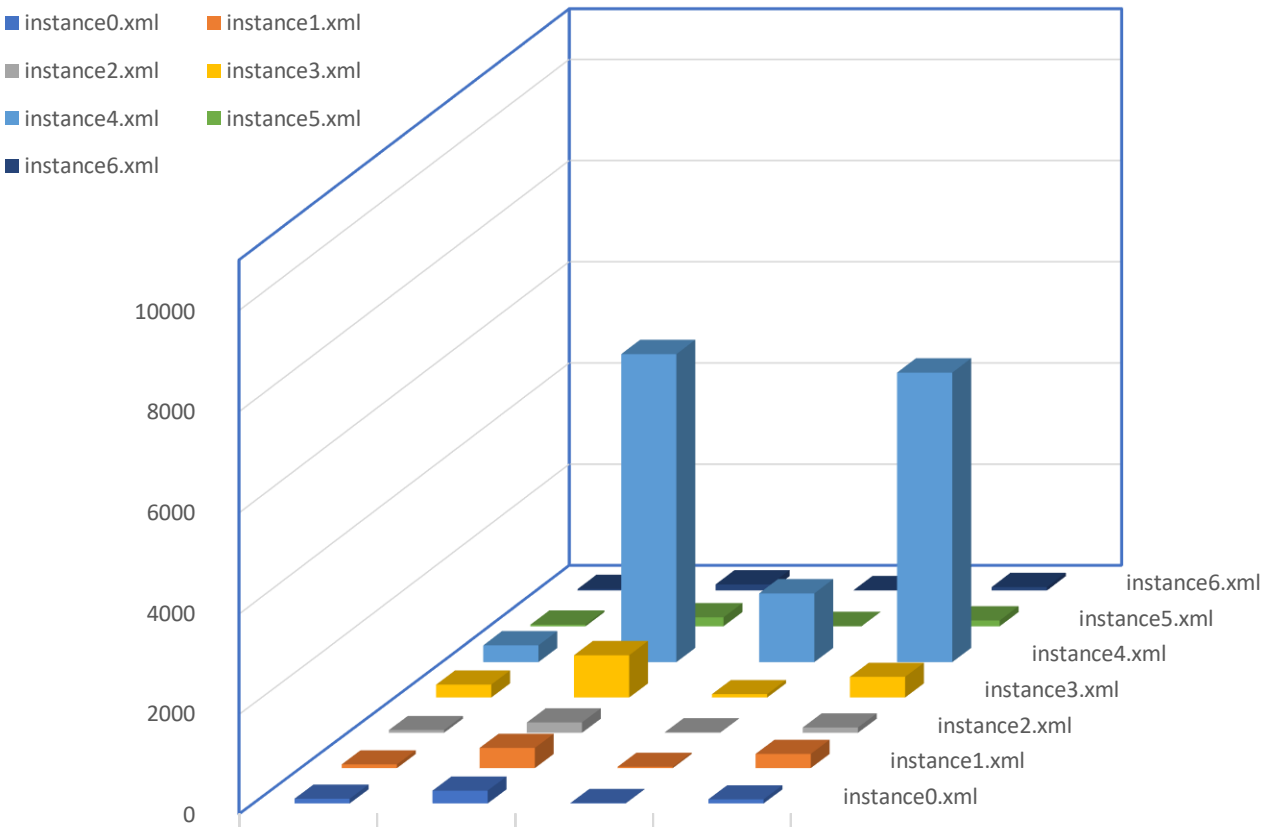
	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	92	810	97	960
instance1.xml	51	304	18	189
instance2.xml	47	788	75	666
instance3.xml	235	2697	361	2444
instance4.xml	260	8861	1917	8586
instance5.xml	30	201	7	139
instance6.xml	27	145	5	92

5.7. AStar, connections=8, metrictype=Manhattan, hweight=1



	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	95	257	8	82
instance1.xml	58	508	50	397
instance2.xml	49	575	51	454
instance3.xml	243	1569	154	1202
instance4.xml	266	7991	1760	7741
instance5.xml	30	177	5	116
instance6.xml	26	133	4	82

5.8. AStar, connections=8, metrictype=Octile, hweight=2



	Стоимость пути	Число раскрытых вершин	Время работы алгоритма (млсек.)	Число итераций
instance0.xml	95	257	8	82
instance1.xml	74	406	26	285
instance2.xml	55	208	11	106
instance3.xml	258	847	67	414
instance4.xml	333	6089	1360	5724
instance5.xml	32	177	5	115
instance6.xml	28	118	3	66

Исходя из полученных результатов, можно сделать следующие выводы:

- Алгоритм поиска кратчайшего пути A* является наиболее эффективным среди остальных, так как срабатывает за минимальное число итераций
- За счёт использования формулы расчёта диагонального расстояния (“Octile metric type”) в связке с алгоритмом A* значительно сокращается время, число раскрытых вершин и число итераций работы алгоритма
- Алгоритм BFS тратит меньше всех остальных времени на поиск оптимального пути
- Алгоритм Dijkstra чаще остальных находит самый коротки путь

Заключение.

В ходе проделанной лабораторной работы, мной были усвоены основные принципы работы алгоритмов “A*”, “Дейкстры” и “Поиска в ширину”.