# Тестирование работы алгоритма, дающего 1.5-приближение.

In [10]:
```
import numpy as np
import itertools
import scipy.stats as sps
from tqdm import tqdm_notebook
import matplotlib.pyplot as plt
import timeit
import time

from MetricTSP import MetricTSP
```

## 1. Решение перебором

In [38]:
```
def brute_force(graph):
    V = len(graph)
    ans = 10**3
    for perm in itertools.permutations(list(range(V-1))):
        permutation = np.concatenate(([V-1],
                                      np.array(perm),
                                      [V-1]))
        cur = permutation[0]
        path = 0
        for v in permutation[1:]:
            path += graph[cur][v]
            cur = v
        ans = min(ans, path)
    return ans
```

## 2. Генерация случайного графа

In [39]:

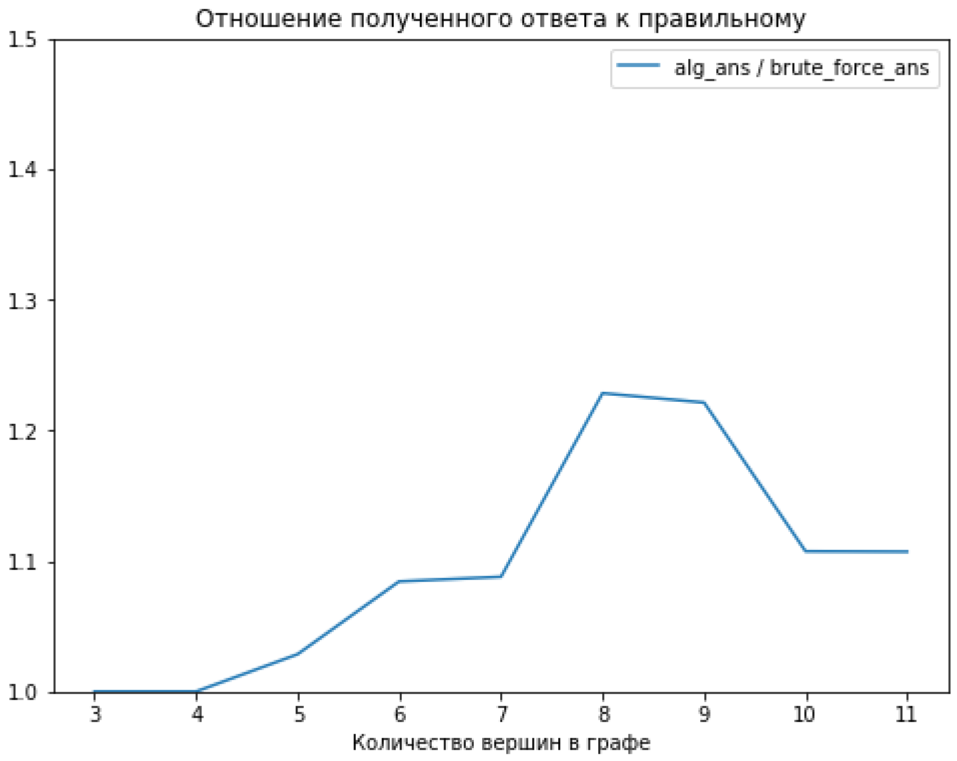
```
def is_metric(graph):
    v = len(graph)
    for i in range(v):
        for j in range(v):
            for u in range(v):
                if graph[i, u] + graph[u, j] < graph[i, j]:
                    return False
    return True


def make_metric(graph):
    v = len(graph)
    for i in range(v):
        for j in range(v):
            for u in range(v):
                if graph[i, u] + graph[u, j] < graph[i, j]:
                    graph[i, j] = graph[i, u] + graph[u, j]
                    graph[j, i] = graph[i, j]


def generate_graph(v):
    g = sps.randint.rvs(1, 50, size=(v, v))
    for i in range(v):
        for j in range(v):
            if i > j:
                g[i, j] = g[j, i]
    while (is_metric(g) == False):
        make_metric(g)
    return g
```

## 3. Сравнение алгоритмов

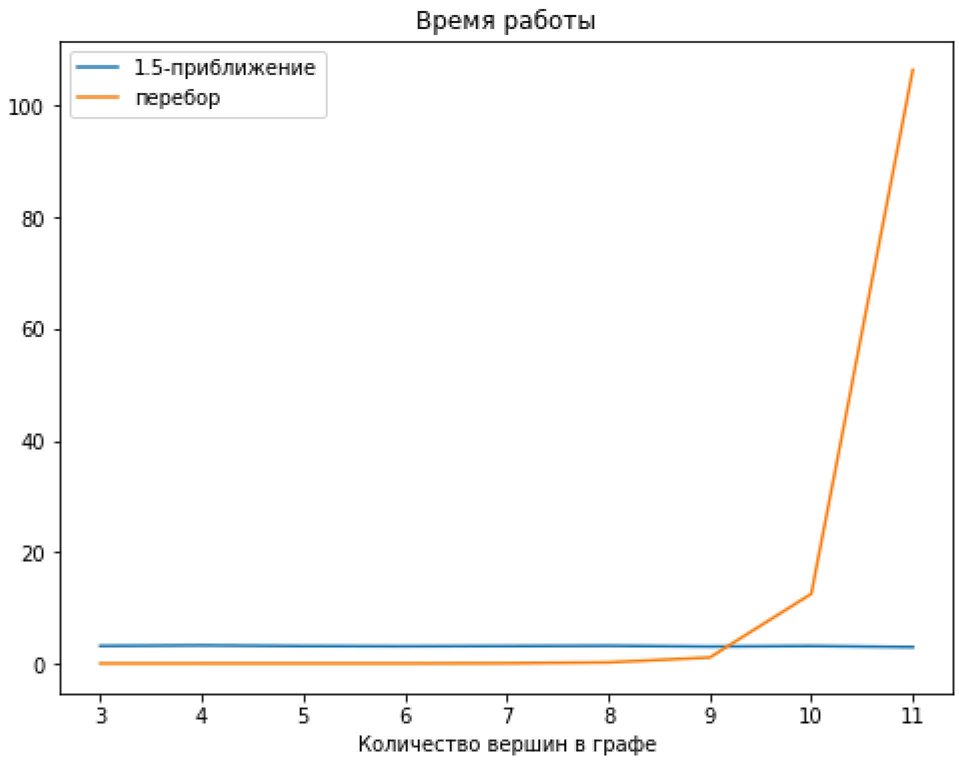
In [54]:
```
arr = np.arange(3, 12)

alg_answers = []
alg_times = []
brute_force_answers = []
brute_force_times = []

for v in tqdm_notebook(arr):
    g = generate_graph(v)

    print('Кол-во вершин: {}'.format(v))
    print(g)
    begin = time.time()
    tsp_solver = MetricTSP(v, g)
    cycle = tsp_solver.ham_cycle()
    cur = cycle[0]
    ans = 0
    for v in cycle[1:]:
        ans += g[cur][v]
        cur = v
    end = time.time()
    alg_answers.append(ans)
    alg_times.append(end - begin)

    print()

    begin = time.time()
    brute_force_answers.append(brute_force(g))
    end = time.time()
    brute_force_times.append(end - begin)


brute_force_answers = np.array(brute_force_answers)
brute_force_times = np.array(brute_force_times)
alg_answers = np.array(alg_answers)
alg_times = np.array(alg_times)
```

```
Кол-во вершин: 10
[[14 34 15 24 14 11 16 35  7 18]
 [34 32 29 24 22 33 33 16 36 35]
 [15 29  8  9 13  4  9 25  8 20]
 [24 24  9 18 11 13  9 28 17 24]
 [14 22 13 11 22 17 20 38 14 13]
 [11 33  4 13 17  8  5 24  4 16]
 [16 33  9  9 20  5  1 19  9 21]
 [35 16 25 28 38 24 19 32 28 40]
 [ 7 36  8 17 14  4  9 28  8 20]
 [18 35 20 24 13 16 21 40 20 19]]
mst:  [(0, 8), (8, 5), (5, 2), (5, 6), (2, 3), (3, 4), (4, 9), (6, 7), (7,
1)]
min_perfect_matching:  [(1, 3), (0, 2)]
eul_cycle:  [2, 0, 8, 5, 6, 7, 1, 9, 4, 3, 2, 0]
ham_cycle:  [2, 0, 8, 5, 6, 7, 1, 9, 4, 3, 2]

Кол-во вершин: 11
[[10  5 25 20 16 14 23 13 10 20 13]
```

```
In [58]:
plt.figure(figsize=(8, 6))

plt.title('Отношение полученного ответа к правильному')
plt.plot(arr, alg_answers / brute_force_answers,
         label='alg_ans / brute_force_ans')
plt.xlabel('Количество вершин в графе')
plt.ylim(1., 1.5)
plt.legend()
plt.show()
```



In [59]:
```
plt.figure(figsize=(8, 6))

plt.title('Время работы')
plt.plot(arr, alg_times, label='1.5-приближение')
plt.plot(arr, brute_force_times, label='перебор')
plt.xlabel('Количество вершин в графе')
plt.legend()
plt.show()
```



**Вывод:**

Алгоритм, дающий 1.5-приближение, действительно даёт ответ, не более чем в 1.5 раза отличающийся от правильного, при этом уже при 10 вершинах время его работы в несколько раз меньше времени работы перебора.