后端测试说明

算法对比

通过单元测试的方式进行常规DFS与本带剪枝的多因素深度优先算法的比较。

两种算法使用相同的数据进行测试，在相同的环境下分别测试出平均时间。

执行后端代码中的gragh包下Test文件即可进行测试。

```
public static void main(String[] args)
{
    List<AllData> data = new ArrayList<>();
    for (int i = 0; i < 3000; i++) {
        initData(data);
    }
    long start = System.currentTimeMillis();
    test1(data);
    long end = System.currentTimeMillis();
    System.out.println("算法一所需时间:" + (end - start));
    start = System.currentTimeMillis();
    test2(data);
    end = System.currentTimeMillis();
    System.out.println("算法二所需时间:" + (end - start));
}
```

这段代码会计算算法执行前后的时间戳，以此为依据比较两种算法的性能。其中test1和test2代表常规DFS算法和带剪枝的多因素深度优先算法

测试结果：

数据条数	算法一所需平均时间 （ms）	算法二所需平均时间 (ms)
100	14	5
200	22	7
500	47	12
1000	124	33
2000	569	200
3000	980	354

注：数据为一天的航班数

从这些数据可以看出我们所采用的算法二也就是带剪枝的多因素深度优先有明显的性能优势。

压力测试

测试环境

- macOS Monterey 12.2.1
- 6核16G内存

测试方式

运行数据库之后，运行后段代码，通过wrk压测工具进行测试。

测试结果

```
# yurunjie @ RUNJIEYU-MB0 in ~/Desktop [16:36:28]
$ wrk -t 16 -c 100 -d 30s --latency --timeout 5s -s post.lua http://127.0.0.1:8333/recommend/second
Running 30s test @ http://127.0.0.1:8333/recommend/second
  16 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     8.29ms   43.07ms 607.58ms   99.05%
    Req/Sec   594.02    291.78   838.00     81.20%
  Latency Distribution
     50%    4.10ms
     75%    4.44ms
     90%    5.18ms
     99%   18.31ms
  16350 requests in 30.09s, 5.30MB read
  Socket errors: connect 0, read 67, write 0, timeout 0
  Non-2xx or 3xx responses: 16350
Requests/sec:    543.42
```

结论

查看测试结果我们可以发现我们的算法也具备比较高的性能，接口也可以提供比较高的并发量，符合赛题对我们的要求。