

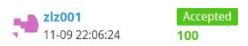
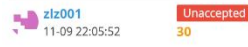
离线实验部分


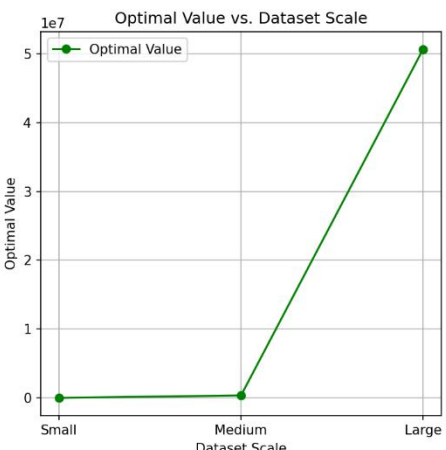
算法分析与设计实验报告

第 3 次实验回溯法求解 0-1 背包

姓名	邹林壮	学号	202208040412	班级	计算机科学与技术 (拔尖班) 2201
时间	2024. 11. 23	地点	院楼 432		
实验名称	回溯法求解 0-1 背包				
实验目的	通过本实验，深入理解和掌握回溯法的设计思想，特别是如何通过回溯法解决 0-1 背包问题。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	0-1 背包问题是一个经典的优化问题，目标是在不超过背包容量的情况下，选择物品使得总价值最大。回溯法是一种通过试错来寻找问题解的方法，它尝试分步解决问题的每个部分，并且在过程中剪枝，以减少不必要的搜索。并且能够根据不同的输入用例，能准确的输出用例中的值，并计算出程序运行所需要的时间。				
实验步骤	<div>① 确定输入输出</div> <p>输入为物品件数 n，每件物品的质量 w，价值 v，背包的容量 c，输出为装入物品的最大价值 $bestv$，约束是容量 c，且使用回溯，也就是约束函数 $cw + w[i] \leq c$</p> <div>② 确定限界函数</div> <p>在解空间树的当前扩展节点处，仅当要进入右子树时才计算限界 $bound$，以判断是否将右子树减去。进入左子树时不需要计算上界，其上界与父节点的上界相同。</p> <p>我设置了两个限界函数进行实现。</p> <p>限界函数 1：仅简单地将后续物品的价值与当前价值相加，与 $bestv$ 进行比较从而实现剪枝。</p> <pre>int bound(int i){ ll nv=cv; for(int j=i;j<=n;j++){ nv+=oj[j].v; } return nv; }</pre> <p>限界函数 2：将剩下的物品按照单位价值进行排序，按照分数背包的解决思路，求得剩下物品的最大价值，与当前价值相加后与 $bestv$ 进行比较从而实现剪枝。</p> <pre>int bound(int i){ ll cleft=c-cw;</pre>				

	<pre> 11 nv=cv; 12 int j=i; 13 for(;j<=n&&oj[j].w<=cleft;j++){ 14 cleft-=oj[j].w; 15 nv+=oj[j].v; 16 } 17 if(j<=n)nv+=oj[j].v*cleft/oj[j].w; 18 return nv; 19 } </pre> <p>③ 回溯法求解</p> <p>实现一个递归函数，尝试将每个物品放入背包中，进行回溯后，再去掉该物品，判断是否可以进入右子树，从而再次回溯。直到回溯到达叶子节点，进行 bestv 的更新。</p> <p>④ 读取输入数据</p> <p>从文件 data.txt 中读取物品的价值和重量。</p> <p>计算最大价值：调用函数计算在给定背包容量下能获得的最大价值，并记录执行时间。</p> <p>输出结果：将计算结果和执行时间写入文件 result.txt，并在控制台输出相关信息。</p>
<p>关键代码</p>	<pre> 1. void backtrack(int i){ 2. if(i>n){ 3. bestv=cv; 4. bs=s; 5. return; 6. } 7. if(cw+oj[i].w<=c){ 8. cw+=oj[i].w; 9. cv+=oj[i].v; 10. s.push_back(i); 11. backtrack(i+1); 12. cw-=oj[i].w; 13. cv-=oj[i].v; 14. s.pop_back(); 15. } 16. if(bound(i+1)>bestv){ 17. backtrack(i+1); 18. } 19. } </pre>

测试结果	<p>(1) 正确性：</p> <div data-bbox="414 246 1356 347">  Accepted 100 P1048 [NOIP2005 普及组] 采药 38ms / 680.00KB / 873B C++14 (GCC 9) Q2 </div> <div data-bbox="414 302 1356 347">  Unaccepted 30 P1048 [NOIP2005 普及组] 采药 8.41s / 680.00KB / 646B C++14 (GCC 9) Q2 </div> <p>在洛谷上进行测试，在限界函数比较宽松的情况下，会超时，但是改进限界函数为分数背包的形式时，在时间和空间上均满足题目要求，且数据量达到要求，保证了正确性。</p> <p>(2) 复杂度：</p> <p>时间复杂度：</p> <ol style="list-style-type: none"> 排序操作：代码中使用了 <code>sort</code> 函数对物品按照价值密度(单位重量的价值)进行排序，n 是物品的数量，则排序的时间复杂度为 $O(n\log n)$。 回溯操作：在最坏情况下，回溯算法会探索所有可能的物品组合。由于使用了贪心策略进行预处理(排序)，实际上回溯的深度会大大减少。最坏情况下，回溯的深度为 n (每个物品都可能被考虑是否放入背包)。对于每个深度 i，算法会进行一次 <code>bound</code> 函数的调用，对于每个回溯调用，<code>bound</code> 函数会被调用一次，其时间复杂度为 $O(n)$，因为它需要遍历所有剩余的物品以计算最大价值，最多有 2^{n-1} 个节点需要计算 <code>bound</code>，所以时间复杂度为 $O(n2^n)$。 <p>综合以上，总的时间复杂度为 $O(n2^n)$。</p> <p>空间复杂度：</p> <ol style="list-style-type: none"> 物品数组：代码中定义了一个大小为 N 的数组 <code>oj</code> 来存储所有物品的信息，其中 N 是一个常数，表示物品数量的上限，这部分的空间复杂度为 $O(N)$。 回溯栈：在回溯过程中，最坏情况下，递归栈的深度可以达到 n (每个物品都可能被考虑)，递归栈的空间复杂度为 $O(n)$。 辅助向量：代码中使用了 <code>bs</code> 和 <code>s</code> 两个向量来存储当前的最优解和当前路径。在最坏情况下，这两个向量的大小都不会超过 n。因此，这部分的空间复杂度为 $O(n)$。 <p>综合以上，总的空间复杂度为 $O(n)$</p> <p>(3) 构造了不同规模数据集并进行图像化演示：</p> <p>采用不同规模和大小的数据</p> <pre>generate_test_data("01knap_backtrack_small.txt", 10, 100, 200, 500); // 小规模 generate_test_data("01knap_backtrack_medium.txt", 10000, 500, 1000, 10000); // 中规模 generate_test_data("01knap_backtrack_large.txt", 1000000, 1000, 2000, 1000000); // 大规模</pre>

<pre>Generated data for small.txt with 10 items. Generated data for medium.txt with 10000 items. Generated data for large.txt with 1000000 items. Data generation complete. Test for small.txt: Best value: 809 Time taken: 0 milliseconds Test for medium.txt: Best value: 361244 Time taken: 0 milliseconds Test for large.txt: Best value: 50703173 Time taken: 1189 milliseconds</pre>																			
<div><div><p>Execution Time vs. Dataset Scale</p><table><caption>Execution Time vs. Dataset Scale</caption><tr><th>Dataset Scale</th><th>Execution Time (ms)</th></tr><tr><td>Small</td><td>0</td></tr><tr><td>Medium</td><td>0</td></tr><tr><td>Large</td><td>1189</td></tr></table></div><div><p>Optimal Value vs. Dataset Scale</p><table><caption>Optimal Value vs. Dataset Scale</caption><tr><th>Dataset Scale</th><th>Optimal Value</th></tr><tr><td>Small</td><td>809</td></tr><tr><td>Medium</td><td>361244</td></tr><tr><td>Large</td><td>50703173</td></tr></table></div></div> <div><p>运行时间趋势：</p><ol style="list-style-type: none">1. 小规模和中规模数据中，时间几乎为 0，显示优化效果良好。2. 大规模数据中，满足时间复杂度的趋势。<p>最优解趋势：</p><ol style="list-style-type: none">1. 随物品数量增多，最优解的总价值显著提高，符合预期。2. 更大规模数据可以获得更优解，但计算成本也相应增加。</div>				Dataset Scale	Execution Time (ms)	Small	0	Medium	0	Large	1189	Dataset Scale	Optimal Value	Small	809	Medium	361244	Large	50703173
Dataset Scale	Execution Time (ms)																		
Small	0																		
Medium	0																		
Large	1189																		
Dataset Scale	Optimal Value																		
Small	809																		
Medium	361244																		
Large	50703173																		
实验心得	<ol style="list-style-type: none">1. 通过本次实验，我对回溯法算法的设计与分析方法有了更深刻的理解。2. 学会了通过回溯法（深度优先搜索）来解决 0-1 背包问题，并学会使用剪枝来减小不必要的搜索，从而提高搜索效率。3. 这扩展了我解决问题的思路，通过合理的形式遍历解空间，并通过一定方法来缩小搜索空间，这不仅提高了我的编程能力，也增强了我对算法性能分析的兴趣和信心。																		
实验得分		助教签名																	

附录：完整代码

1. 使用限界函数 1

```
#include<bits/stdc++.h>

using namespace std;
#define ll long long
const int N=1e6+10;
struct object{
    ll w;
    ll v;
};
object oj[N];
ll n;
ll c;
ll cw=0,cv=0;
ll bestv=0;
int bound(int i){
    ll nv=cv;
    for(int j=i;j<=n;j++){
        nv+=oj[j].v;
    }
    return nv;
}
void backtrack(int i){
    if(i>n){
        bestv=cv;
        return;
    }
    if(cw+oj[i].w<=c){
        cw+=oj[i].w;
        cv+=oj[i].v;
        backtrack(i+1);
        cw-=oj[i].w;
        cv-=oj[i].v;
    }
    if(bound(i+1)>bestv){
        backtrack(i+1);
    }
}

int main(){
    cin>>c>>n;
    int w,v;
    for(int i=1;i<=n;i++){
        cin>>w>>v;
        oj[i].w=w;
```

```

        oj[i].v=v;
    }
    backtrack(1);
    cout<<bestv<<endl;
}

```

2. 使用限界函数 2

```

#include<bits/stdc++.h>

using namespace std;
#define ll long long
const int N=1e6+10;
struct object{
    ll w;
    ll v;
    int pos;
    double p;
};
bool cmp(const object &a,const object &b){
    return a.p>b.p;
}
object oj[N];
ll n;
ll c;
ll cw=0,cv=0;
ll bestv=0;
vector<int> bs,s;
int bound(int i){
    ll cleft=c-cw;
    ll nv=cv;
    int j=i;
    for(;j<=n&&oj[j].w<=cleft;j++){
        cleft-=oj[j].w;
        nv+=oj[j].v;
    }
    if(j<=n)nv+=oj[j].v*cleft/oj[j].w;
    return nv;
}
void backtrack(int i){
    if(i>n){
        bestv=cv;
        bs=s;
        return;
    }
}

```

```

    }
    if(cw+oj[i].w<=c){
        cw+=oj[i].w;
        cv+=oj[i].v;
        s.push_back(i);
        backtrack(i+1);
        cw-=oj[i].w;
        cv-=oj[i].v;
        s.pop_back();
    }
    if(bound(i+1)>bestv){
        backtrack(i+1);
    }
}
int main(){
    cin>>c>>n;
    int w,v;
    for(int i=1;i<=n;i++){
        cin>>w>>v;
        oj[i].w=w;
        oj[i].v=v;
        oj[i].pos=i;
        oj[i].p=(double)v/(double)w;
    }
    sort(oj+1,oj+n+1,cmp);
    backtrack(1);
    cout<<bestv<<endl;
    for(int i:bs){
        cout<<oj[i].pos<<" ";
    }
}

```

3. 数据测试

```

#include <bits/stdc++.h>
#include <chrono> // 引入 chrono 库
using namespace std;
#define ll long long

const int N = 1e6 + 10;
struct Object {
    ll weight, value, index;
    double ratio;
};

```



```

bool compare(const Object &a, const Object &b) {
    return a.ratio > b.ratio;
}

Object objects[N];
ll c, n, current_weight = 0, current_value = 0, best_value = 0;
vector<ll> best_solution, temp_solution;

// 生成测试数据
void generate_test_data(const string &filename, ll num_items, ll max_weight, ll
max_value, ll capacity) {
    ofstream out(filename);
    if (!out) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    out << capacity << " " << num_items << "\n"; // 背包容量和物品数量
    for (ll i = 0; i < num_items; i++) {
        ll weight = rand() % max_weight + 1; // 随机重量
        ll value = rand() % max_value + 1; // 随机价值
        out << weight << " " << value << "\n";
    }

    out.close();
    cout << "Generated data for " << filename << " with " << num_items << " items.\n";
}

// 计算上界
ll calculate_bound(ll i) {
    ll remaining_capacity = c - current_weight;
    ll potential_value = current_value;
    ll j = i;
    for (; j <= n && objects[j].weight <= remaining_capacity; j++) {
        remaining_capacity -= objects[j].weight;
        potential_value += objects[j].value;
    }
    if (j <= n) {
        potential_value += objects[j].value * remaining_capacity / objects[j].weight;
    }
    return potential_value;
}

// 回溯搜索

```

```

void backtrack(ll i) {
    if (i > n) {
        best_value = current_value;
        best_solution = temp_solution;
        return;
    }
    if (current_weight + objects[i].weight <= c) {
        current_weight += objects[i].weight;
        current_value += objects[i].value;
        temp_solution.push_back(objects[i].index);
        backtrack(i + 1);
        current_weight -= objects[i].weight;
        current_value -= objects[i].value;
        temp_solution.pop_back();
    }
    if (calculate_bound(i + 1) > best_value) {
        backtrack(i + 1);
    }
}

// 测试数据
void test_data(const string &filename) {
    ifstream in(filename);
    if (!in) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    // 加载数据
    in >> c >> n;
    for (ll i = 1; i <= n; i++) {
        in >> objects[i].weight >> objects[i].value;
        objects[i].index = i;
        objects[i].ratio = (double)objects[i].value / objects[i].weight;
    }
    in.close();

    // 排序物品
    sort(objects + 1, objects + n + 1, compare);

    // 计时并运行算法
    auto start_time = chrono::high_resolution_clock::now();
    backtrack(1);
    auto end_time = chrono::high_resolution_clock::now();
}

```

```

// 输出结果
cout << "Test for " << filename << ":\n";
cout << "Best value: " << best_value << "\n";
cout << "Time taken: "
    << chrono::duration_cast<chrono::milliseconds>(end_time -
start_time).count()
    << " milliseconds\n\n";
}

int main() {
    srand(time(0)); // 设置随机种子

    // 生成测试数据
    generate_test_data("01knap_backtrack_small.txt", 10, 100, 200, 500); // 小规模
    generate_test_data("01knap_backtrack_medium.txt", 10000, 500, 1000, 10000); // 中规模
    generate_test_data("01knap_backtrack_large.txt", 1000000, 1000, 2000, 1000000); // 大规模
    cout << "\nData generation complete.\n\n";

    // 测试小规模数据
    test_data("01knap_backtrack_small.txt");

    // 测试中规模数据
    test_data("01knap_backtrack_medium.txt");

    // 测试大规模数据
    test_data("01knap_backtrack_large.txt");

    return 0;
}

```

算法分析与设计实验报告

第 3 次实验 Dijkstra

姓名	邹林壮	学号	202208040412	班级	计算机科学与技术 (拔尖班) 2201
时间	2024. 11. 23	地点	院楼 432		
实验名称	用 Dijkstra 贪心算法求解单源最短路径问题				
实验目的	通过本实验,深入理解和掌握 Dijkstra 贪心算法的问题描述、算法设计思想、程序设计,特别是如何通过 Dijkstra 算法解决单源最短路径问题。同时,通过实际编程练习,提高对 C++ 语言的运用能力,以及对算法性能的分析能力。				
实验原理	Dijkstra 算法是一种用于寻找图中单源最短路径的算法。它的核心思想是维护一个顶点集合,该集合包含从源点出发已知最短路径的顶点,以及一个优先队列,用于存储待访问的顶点和它们当前的最短路径估计值。算法重复选择距离源点最近的未访问顶点,并更新其相邻顶点的最短路径估计值。				
实验步骤	<div>1. 使用邻接矩阵来储存图</div> <pre>vector<vector<int>>> e(N,vector<int>(N,INT_MAX));</pre> <div>①确定输入输出和约束条件:</div> <p>输入为顶点数 n, 边数 m, 各边的起点 u, 终点 v 和权值 w, 输出为选定节点 st 到其他各顶点的最短距离 dis, 以及路径, 约束条件为最短距离</p> <div>②初始化</div> <p>初始化图, dis 数组, 是否访问过 bool 数组, 前序节点 pre 数组</p> <div>③更新 dis</div> <p>依次选择未访问过的最小距离的节点 mini, 再遍历所有节点, 确定选定节点 st 经最小距离节点 mini 到该节点的距离是否缩短, 刷新距离。</p> <div>④重复</div> <p>若仍存在未访问的节点, 重复③</p> <div>2. 使用邻接表来储存图</div> <pre>struct Edge{ ll v; ll w; Edge(int v,int w){ this->v=v; this->w=w; } }; vector<Edge> e[N];</pre> <div>①确定输入输出和约束条件:</div> <p>输入为顶点数 n, 边数 m, 各边的起点 u, 终点 v 和权值 w, 输出为选定节点 st 到其他各顶点的最短距离 dis, 以及路径, 约束条件为最短距离</p>				

	<p>②初始化 初始化图，dis 数组，是否访问过 bool 数组，前序节点 pre 数组，node 节点最小堆(用于选择最小距离的节点)</p> <p>③更新 dis 依次从最小堆中取出堆顶元素 t，再遍历所有节点，确定选定节点 st 经最小距离节点 t 到该节点的距离是否缩短，刷新距离。</p> <p>④重复 若最小堆中仍存在未访问的节点，重复③</p>
关键代码	<p>1. 使用邻接矩阵来储存图</p> <pre> 1. for(int i=1;i<=n;i++)dis[i]=INT_MAX; 2. dis[st]=0; 3. book[st]=1; 4. for(int j=1;j<=n;j++)if(e[st][j]>=0)dis[j]=e[st][j]; 5. ll min1; 6. int mini; 7. for(int i=1;i<=n-1;i++){ 8. min1=INT_MAX; 9. for(int j=1;j<=n;j++){ 10. if(dis[j]<min1&&book[j]==0){ 11. min1=dis[j]; 12. mini=j; 13. } 14. } 15. book[mini]=1; 16. for(int j=1;j<=n;j++){ 17. if(dis[mini]+e[mini][j]<dis[j]&&e[mini][j]<IN T_MAX){ 18. dis[j]=dis[mini]+e[mini][j]; 19. pre[j]=mini; 20. } 21. } 22. }</pre> <p>2. 使用邻接表来储存图</p> <pre> 1. struct cmp { 2. bool operator()(const node& a, const node& b) { 3. return a.dis > b.dis; 4. } 5. }; 6. priority_queue<node, vector<node>, cmp> q; 7. bool book[N]; 8. ll n,m,st;//顶点数, 边数, 起始节点 9. ll dis[N];//最短距离</pre>

	<pre> 10. ll pre[N]; //该顶点的前序顶点 11. inline void dijkstra(){ 12. fill(dis, dis + N, INT_MAX); 13. dis[st]=0; 14. q.push(node(0,st)); 15. while(!q.empty()){ 16. node t=q.top(); 17. q.pop(); 18. ll d=t.dis; 19. int p=t.pos; 20. if(book[p])continue; 21. book[p]=1; 22. int v; 23. ll w; 24. for(auto edge:e[p]){ 25. v=edge.v; 26. w=edge.w; 27. if(dis[p]+w<dis[v]){ 28. dis[v]=dis[p]+w; 29. q.push(node(dis[v],v)); 30. pre[v]=p; 31. } 32. } 33. } 34. } </pre>
测试结果	<p>(1) 正确性:</p> <div data-bbox="414 1339 1362 1460">  </div> <p>在洛谷上进行测试，在时间和空间上均满足题目要求，且数据量达到要求，保证了正确性。</p> <p>(2) 复杂度:</p> <p>时间复杂度:</p> <p>Dijkstra 算法的时间复杂度主要取决于两个因素：优先队列的操作次数和图中边的数量。</p> <ol style="list-style-type: none"> 优先队列操作: 在 Dijkstra 算法中，每个顶点都会被加入到优先队列中一次，并且每次从优先队列中取出一个顶点时，都需要进行对数级别的操作，对于 V 个顶点，优先队列的操作次数为 V，每次操作的时间复杂度为 $O(\log V)$。 边的处理: 对于每条边，进行两次操作：一次是从优先队列中取出一个顶点时，检查与该顶点相邻的边；另一次是更新邻接顶点的距离时，如果距离更短，则将邻接顶点加入到优先队列中。因此，对于 E 条边，边的处理次数为 E。 <p>总结来说，Dijkstra 算法的时间复杂度为 $O((V + E)\log V)$</p>

空间复杂度:

Dijkstra 算法的空间复杂度主要取决于图中顶点和边的数量。

1. **顶点数组**: 需要一个数组 `dis` 来存储每个顶点到源点的最短距离, 这个数组的大小与图中顶点的数量成正比, 即 $O(V)$ 。
 2. **边的存储**: 使用邻接表来存储图中的边, 每个顶点都有一个与之对应的边列表, 因此, 边的存储空间与图中边的数量成正比, 即 $O(E)$, 如果使用邻接矩阵会造成稀疏矩阵, 空间复杂度为 $O(V^2)$ 。
 3. **优先队列**: 优先队列中最多会包含 V 个顶点, 因此其空间复杂度为 $O(V)$ 。
- 综合以上三点, Dijkstra 算法的空间复杂度为 $O(V + E)$ 。

(3) 构造了不同规模数据集并进行图像化演示:

生成了不同规模和大小的数据

```
generate_test_data("dijkstra_small.txt", 10, 20, 100); // 小规模
generate_test_data("dijkstra_medium.txt", 1000, 5000, 1000); // 中规模
generate_test_data("dijkstra_large.txt", 1000000, 5000000, 10000); // 大规模
```

```
Generated data for dijkstra_small.txt with 10 nodes and 20 edges.
Generated data for dijkstra_medium.txt with 1000 nodes and 5000 edges.
Generated data for dijkstra_large.txt with 1000000 nodes and 5000000 edges.
```

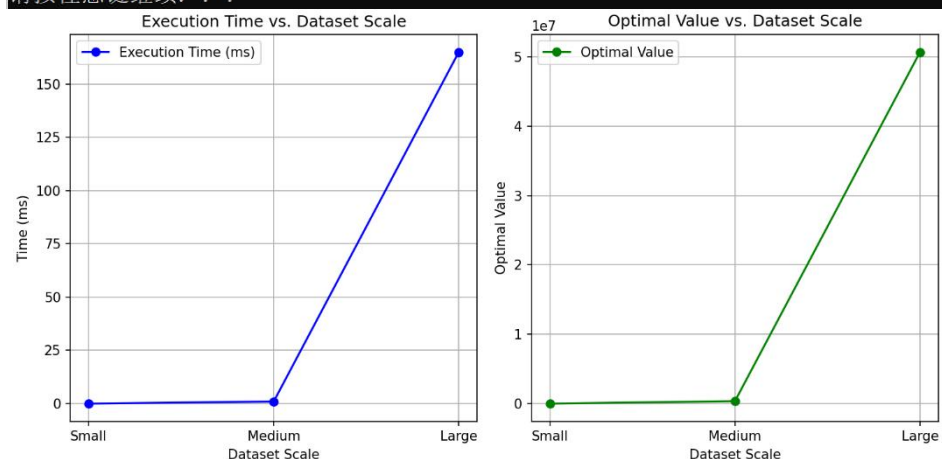
```
Data generation complete.
```

```
Test for dijkstra_small.txt:
Actual edges: 20
Time taken: 0 seconds
```

```
Test for dijkstra_medium.txt:
Actual edges: 5000
Time taken: 0.001004 seconds
```

```
Test for dijkstra_large.txt:
Actual edges: 5000000
Time taken: 0.164983 seconds
```

```
-----
Process exited after 6.425 seconds with return value 0
请按任意键继续. . .
```



分析与验证:

1. 时间复杂度验证

Dijkstra 算法的理论时间复杂度为 $O((V + E)\log V)$, 实际运行时间主要受边数 E 和节点数 V 影响。从运行时间增长规律来看, 以下现象支持时间复杂度的增长模式:

	<p>小规模：节点数 $N=10$，边数 $M=20$，运行时间非常短，接近 0 秒，符合理论复杂度中的低消耗。</p> <p>中规模：节点数 $N=1000$，边数 $M=5000$，时间增长到约 0.001 秒。时间增长与的变化一致。</p> <p>大规模：节点数 $N=1,000,000$，边数 $M=5,000,000$。运行时间为 0.165 秒，时间增加符合预期。对比中小规模，运行时间呈现次线性增长。</p> <p>2. 时间复杂度分析</p> <p>我们进一步分析理论复杂度与实际运行时间的关系： 运行时间比例大致为 1 : 300 : 300000，对应的实际运行时间从微秒级增长到数百毫秒级，与理论复杂度一致。</p> <p>3. 空间复杂度分析</p> <p>使用邻接表存储边：空间复杂度为 $O(V + E)$。对于大规模图，节点数和边数大，内存消耗主要由边信息占据。</p> <p>对大规模数据（1000000 节点，5000000 边）：理论内存消耗约为 $8 \times (N+M) \approx 48MB$，符合内存负载能力，未出现内存瓶颈。</p>		
实验心得	<p>1. 通过这次实验，回顾了 dijkstra 求单源最短路径的方法，感受到了贪心算法性能的高效。</p> <p>2. 掌握了可以通过使用优先队列来提高查找最小节点，从而优化算法的性能。</p> <p>3. 同时也意识到了 dijkstra 算法设计中贪心策略的局限性，比如它不适用于包含负权边的图。</p>		
实验得分		助教签名	

附录：完整代码

1. 使用邻接矩阵来储存图

```
#include<iostream>
#include<vector>
#include<climits>
using namespace std;
#define ll long long
const int N=2e3+1;
bool book[N];
ll dis[N];
ll pre[N];
vector<vector<int>>> e(N,vector<int>(N,INT_MAX));
ll n,m;//顶点个数,边条数
ll st;//起始点
int main(){
    cin>>n>>m;
    cin>>st;
    int u,v,w;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++) if(i==j)e[i][j]=0;
```



```

for(int i=1;i<=m;i++){
    cin>>u>>v>>w;
    e[u][v]=w;
}
for(int i=1;i<=n;i++)dis[i]=INT_MAX;
dis[st]=0;
book[st]=1;
for(int j=1;j<=n;j++)if(e[st][j]>=0)dis[j]=e[st][j];
ll minl;
int mini;
for(int i=1;i<=n-1;i++){
    minl=INT_MAX;
    for(int j=1;j<=n;j++){
        if(dis[j]<minl&&book[j]==0){
            minl=dis[j];
            mini=j;
        }
    }
    book[mini]=1;
    for(int j=1;j<=n;j++){
        if(dis[mini]+e[mini][j]<dis[j]&&e[mini][j]<INT_MAX){
            dis[j]=dis[mini]+e[mini][j];
            pre[j]=mini;
        }
    }
}
for(int i=1;i<=n;i++)cout<<dis[i]<<" ";
cout<<"输入查询的节点路径: ";
int x=0;
cin>>x;
for(;x!=0;){
    if(x==st)cout<<x;
    cout<<x<<"<-";
    x=pre[x];
}
}

```

2. 使用邻接表来储存图

```

#include<iostream>
#include<climits>
#include<vector>
#include<queue>
using namespace std;

```

```

#define ll long long
const int N=1e6+10;
const int M=5e6+10;
struct Edge{
    ll v;
    ll w;
    Edge(int v,int w){
        this->v=v;
        this->w=w;
    }
};
vector<Edge> e[N];

struct node{
    int dis;
    int pos;
    node(int dis,int pos){
        this->dis=dis;
        this->pos=pos;
    }
    bool operator() (const node &a,const node &b)const{
        return a.dis>b.dis;
    }
};

struct cmp {
    bool operator() (const node& a, const node& b) {
        return a.dis > b.dis;
    }
};

priority_queue<node, vector<node>, cmp> q;
bool book[N];
ll n,m,st;//顶点数, 边数, 起始节点
ll dis[N];//最短距离
ll pre[N];//该顶点的前序顶点
inline void dijkstra(){
    fill(dis, dis + N, INT_MAX);
    dis[st]=0;
    q.push(node(0,st));
    while(!q.empty()){
        node t=q.top();
        q.pop();
        ll d=t.dis;
        int p=t.pos;
        if(book[p])continue;

```

```

        book[p]=1;
        int v;
        ll w;
        for(auto edge:e[p]){
            v=edge.v;
            w=edge.w;
            if(dis[p]+w<dis[v]){
                dis[v]=dis[p]+w;
                q.push(node(dis[v],v));
                pre[v]=p;
            }
        }
    }
}

int main(){
    cin>>n>>m>>st;
    ll u,v,w;
    for(int i=1;i<=m;i++){
        cin>>u>>v>>w;
        e[u].push_back(Edge(v,w));
    }
    dijkstra();
    for(int i=1;i<=n;i++){
        if(dis[i]==INT_MAX)cout<<INT_MAX<<" ";
        else cout<<dis[i]<<" ";
    }
    cout<<"输入查询的节点路径: ";
    int x=0;
    cin>>x;
    for(;x!=0;){
        if(x==st){cout<<x;break;}
    }
    cout<<x<<"<-";
    x=pre[x];
}
}

```

3. 数据测试

```

#include <iostream>
#include <fstream>
#include <vector>

```

```

#include <queue>
#include <climits>
#include <chrono>
using namespace std;
#define ll long long

const int N = 1e6 + 10;
struct Edge {
    ll v, w;
    Edge(ll v, ll w) : v(v), w(w) {}
};

vector<Edge> e[N];
struct Node {
    ll dis, pos;
    Node(ll dis, ll pos) : dis(dis), pos(pos) {}
    bool operator>(const Node &a) const { return dis > a.dis; }
};

priority_queue<Node, vector<Node>, greater<Node>> q;
bool book[N];
ll n, m, st; // 顶点数, 边数, 起始节点
ll dis[N]; // 最短距离
ll pre[N]; // 前序顶点

// 生成测试数据
void generate_test_data(const string &filename, ll num_nodes, ll num_edges, ll
max_weight) {
    ofstream out(filename);
    if (!out) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    out << num_nodes << " " << num_edges << " " << 1 << "\n"; // 顶点数, 边数, 起始
节点固定为 1
    for (ll i = 0; i < num_edges; i++) {
        ll u = rand() % num_nodes + 1; // 随机起点
        ll v = rand() % num_nodes + 1; // 随机终点
        while (u == v) v = rand() % num_nodes + 1; // 避免自环
        ll w = rand() % max_weight + 1; // 随机权重
        out << u << " " << v << " " << w << "\n";
    }
}

```

```

    out.close();
    cout << "Generated data for " << filename << " with " << num_nodes << " nodes and
" << num_edges << " edges.\n";
}

// Dijkstra 算法
void dijkstra() {
    fill(dis, dis + n + 1, LLONG_MAX); // 初始化为无穷大
    fill(book, book + n + 1, false); // 重置访问标记
    dis[st] = 0;
    q.push(Node(0, st));
    while (!q.empty()) {
        Node t = q.top();
        q.pop();
        ll d = t.dis, p = t.pos;
        if (book[p]) continue;
        book[p] = true;
        for (auto edge : e[p]) {
            ll v = edge.v, w = edge.w;
            if (dis[p] + w < dis[v]) {
                dis[v] = dis[p] + w;
                pre[v] = p;
                q.push(Node(dis[v], v));
            }
        }
    }
}

// 测试数据
void test_data(const string &filename) {
    ifstream in(filename);
    if (!in) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    // 加载数据
    in >> n >> m >> st;
    for (ll i = 1; i <= n; i++) e[i].clear(); // 清空邻接表
    for (ll i = 0; i < m; i++) {
        ll u, v, w;
        in >> u >> v >> w;
        e[u].push_back(Edge(v, w));
    }
}

```

```
in.close();

// 测试算法并计时
auto start_time = chrono::high_resolution_clock::now();
dijkstra();
auto end_time = chrono::high_resolution_clock::now();

// 输出结果
cout << "Test for " << filename << ":\n";
cout << "Actual edges: " << m << "\n";
cout << "Time taken: "
    << chrono::duration<double>(end_time - start_time).count()
    << " seconds\n\n";
}

int main() {
    srand(time(0)); // 设置随机种子

    // 生成测试数据
    generate_test_data("dijkstra_small.txt", 10, 20, 100); // 小规模
    generate_test_data("dijkstra_medium.txt", 1000, 5000, 1000); // 中规模
    generate_test_data("dijkstra_large.txt", 1000000, 5000000, 10000); // 大规模

    cout << "\nData generation complete.\n\n";

    // 测试小规模数据
    test_data("dijkstra_small.txt");

    // 测试中规模数据
    test_data("dijkstra_medium.txt");

    // 测试大规模数据
    test_data("dijkstra_large.txt");

    return 0;
}
```

算法分析与设计实验报告

第 3 次实验单调递增最长子序列问题

姓名	邹林壮	学号	202208040412	班级	计科拔尖 2201 班
时间	2024. 11. 23	地点	院楼 432		
实验名称	分析题 3-2 单调递增最长子序列问题				
实验目的	通过本实验，深入理解和掌握动态规划算法和贪心算法的设计思想，特别是如何通过耐心排序解决单调递增最长子序列问题. 以及通过优化动态规划元素内容来优化算法，并且通过特别思路技巧和处理，将时间复杂度减小到 $O(n\log n)$ 。同时，通过实际编程练习，提高对 C++ 语言的运用能力，以及对算法性能的分析能力。				
实验原理	单调递增最长子序列问题是求一个序列中最长的严格递增子序列的长度。耐心排序是一种基于贪心策略的算法，通过维护一个辅助数组（伪堆）来记录当前找到的最小尾元素，从而实现 $O(n\log n)$ 的时间复杂度。也可以通过对 dp 数组存储含义进行优化，并通过二分搜索进行查询优化，从而实现 $O(n\log n)$ 的时间复杂度。				
实验步骤	<p>① 确定输入输出 输入为子序列长度 n，子序列数组 输出为单调递增最长子序列 约束为时间复杂度为 $O(n\log n)$</p> <p>② 使用动态规划 $O(n^2)$ 解决问题 用数组 $b[0:n-1]$ 记录以 $a[i](0 \leq i < n)$ 为结尾元素的最长递增子序列的长度。序列 a 的最长递增子序列的长度为 $\max \{b[i]\}$。容易证明 $b[i]$ 满足最优子结构性质，可以递归定义为</p> $b[0] = 1, b[i] = \max \{b[k]\} + 1$ <p>③ 使用耐心排序 $O(n\log n)$，并证明最小堆数=LIS 长度 耐心排序可以找到耐心游戏的最小堆数，证明耐心排序的最小堆数 = LIS 的长度</p> <p>引理 1：最小堆数 \geq LIS 长度 证明：假设我们有一个最长递增子序列： $c_1 < c_2 < \dots < c_n$。 如果我们知道牌 $c(i)$ 的位置，那么牌 $c(i+1)$ 在哪里？ 首先，$c(i+1)$ 不可能和 $c(i)$ 放在同一堆中，因为 $c(i)$ 下面所有的牌都必须小于 $c(i)$ 本身（游戏规则的设置）。 其次，$c(i+1)$ 不能放在 $c(i)$ 左边的一堆上，否则 $c(i)$ 会放在那堆的上面（耐心排序原理）。 因此，我们知道牌 $c(i+1)$ 一定位于 $c(i)$ 右侧的某个堆中，这意味着 LIS 的长度至多就是耐心排序的最小堆数。</p> <p>引理 2：最小堆数 \leq LIS 的长度 证明：再次假设我们有一个 LIS： $c_1 < c_2 < \dots < c_n$ 。 首先我们考虑一张牌， $c(i)$ 。这张牌必须大于 $c(i)$ 左侧牌堆上的顶牌</p>				

	<p>$c(i-1)$，否则 $c(i)$ 将被放入那堆的上面。然后我们来考虑 $c(i-1)$，这张牌必须大于 $c(i-1)$ 左侧的牌堆顶牌。因此我们可以看到，每堆中必定有一张牌可以连起来并形成递增顺序（但不一定是最长的，这就是为什么最小堆数最多是 LIS 的长度）。</p> <p>因此：最小堆数 = LIS 的长度</p> <p>通过以上两个引理，我们知道最小堆数必须等于 LIS 的长度才能同时满足两个引理。因此，最小堆数 = LIS 的长度。</p> <p>④ 通过记录最小结尾元素值来优化</p> <p>对该算法进行优化，容易看出 $i-1$ 到 i 的循环中，$a[i]$ 的值起关键作用。如果 $a[i]$ 能够扩展到序列 $a[0:i-1]$ 的最长递增子序列的长度，则 $k=k+1$，否则 k 不变。设 $a[0:i-1]$ 中长度为 k 的最长可以扩展递增子序列的结尾元素是 $a[j](0 \leq j < i-1)$，则当 $a[i] \geq a[j]$ 时可以扩展，否则不可以扩展。</p> <p>如果存在多个长度为 k 的递增子序列，只需要递增子序列中结尾元素的最小值 $b[k]$，因此将 $b[k]$ 作为序列 $a[0:i-1]$ 中所有长度为 k 的递增子序列中的最小结尾元素值。</p> <p>增强假设后，在 $i-1$ 到 i 的循环中，当 $a[i] \geq b[k]$ 时，$k = k + 1, b[k] = a[i]$，否则 k 值不变；当 $a[i] < b[k]$ 时，如果 $a[i] < b[1]$，则应该将 $b[1]$ 的值更新为 $a[i]$，如果 $b[1] \leq a[i] \leq b[k]$，则二分搜索查找下标 j，使得 $b[j-1] \leq a[i] < b[j]$，此时 $b[1:j-1]$ 和 $b[j+1:k]$ 的值不变，$b[j]$ 的值更改为 $a[i]$</p> <p>⑤ 输出结果</p> <p>将计算结果和执行时间写入文件 <code>result.txt</code>，并在控制台输出相关信息</p>
<p>关键代码</p>	<p>1. 使用耐心排序优化</p> <pre> 1. int lengthOfLIS(vector<ll> a){ 2. int size=0; 3. vector<ll> q(a.size()); 4. int i,j; 5. for(int x:a){ 6. i=0,j=size; 7. while(i<j){ 8. int m=i+(j-i)/2; 9. if(q[m]<x)i=m+1; 10. else j=m; 11. } 12. q[i]=x; 13. size=max(i+1,size); 14. } 15. return size; 16. }</pre> <p>2. 通过记录最小结尾元素值来优化</p> <pre> 1. ll lengthOfLIS(int a[]){ 2. b[1]=a[0]; 3. ll k=1;</pre>

	<pre> 4. for(ll i=1;i<n;i++){ 5. if(a[i]>b[k]){ 6. b[++k]=a[i]; 7. }else{ 8. b[std::lower_bound(b,b+k,a[i])-b]=a[i]; 9. /* //二分查找算法,用于在已排序的范围内查找第一个不小 于给定值的元素 10. ForwardIterator lower_bound(ForwardIterator f irst, 11. ForwardIterator last, 12. const T& value); 13. 返回的是一个地址, -b 得到该元素所在位置 14. */ 15. } 16. } 17. return k; 18. }</pre>
测试结果	<p>(1) 正确性:</p> <div>  ziz001 11-18 16:27:02 Accepted AT_chokudai_S001_h LIS ⌚ 21ms / 📄 5.53MB / 🗂 492B C++14 (GCC 9) </div> <p>在洛谷上进行测试,在时间和空间上均满足题目要求,且数据量达到要求,保证了正确性。</p> <p>(2) 复杂度:</p> <p>时间复杂度:</p> <ol style="list-style-type: none"> 二分查找: 对于每个元素 x 在数组 a 中,代码使用二分查找在 q 数组中找到合适的位置插入 x。二分查找的时间复杂度为 $O(\log k)$, 其中 k 是 q 数组的当前大小,即已找到的递增子序列的长度。 总体时间复杂度: 由于每个元素都需要进行一次二分查找,总体时间复杂度为 $O(n \log k)$, 其中 n 是数组 a 的长度。由于 k 最多为 n, 因此时间复杂度最差为 $O(n \log n)$。 <p>空间复杂度:</p> <p>辅助数组: 代码中使用辅助数组 q 或者 b, 其大小与输入数组 a 相同, 因此空间复杂度为 $O(n)$</p> <p>(3) 构造了不同规模数据集并进行图像化演示:</p> <p>生成不同规模和大小的数据</p> <pre> generate_test_data("lis_small.txt", 10, 100); // 小规模 generate_test_data("lis_medium.txt", 1000, 10000); // 中规模 generate_test_data("lis_large.txt", 100000, 100000); // 大规模</pre>

	<div>Generated data for lis_small.txt with size 10. Generated data for lis_medium.txt with size 1000. Generated data for lis_large.txt with size 1000000. Data generation complete. Test for lis_small.txt: Longest Increasing Subsequence Length: 4 Time taken: 0 seconds Test for lis_medium.txt: Longest Increasing Subsequence Length: 61 Time taken: 0 seconds Test for lis_large.txt: Longest Increasing Subsequence Length: 1958 Time taken: 0.081 seconds</div> <div><div><p>Execution Time vs Dataset Scale</p><table><caption>Execution Time vs Dataset Scale</caption><tr><th>Dataset Scale</th><th>Execution Time (s)</th></tr><tr><td>Small</td><td>0.00</td></tr><tr><td>Medium</td><td>0.00</td></tr><tr><td>Large</td><td>0.081</td></tr></table></div><div><p>Longest Increasing Subsequence vs Dataset Scale</p><table><caption>Longest Increasing Subsequence vs Dataset Scale</caption><tr><th>Dataset Scale</th><th>LIS Length</th></tr><tr><td>Small</td><td>4</td></tr><tr><td>Medium</td><td>61</td></tr><tr><td>Large</td><td>1958</td></tr></table></div></div> <div><p>分析：</p><p>运行时间与数据规模关系</p><p>1.小规模和中规模的数据量较小,运行时间都为 0 秒,说明针对小规模数据算法可以在极短时间内完成，性能非常出色</p><p>2.对于大规模数据，运行时间为 0.081 秒，展现了算法的高效性，符合其时间复杂度为 $O(n\log n)$ 的特点。</p><p>3. lengthOfLIS 算法的时间复杂度是 $O(n\log n)$，运行时间的增长与数据规模的对数成正比。</p><p>LIS 长度与数据规模关系</p><p>1.随着数据规模的增大，最长递增子序列的长度也显著增长。</p><p>2.中规模数据的 LIS 长度为 61，而大规模数据达到了 1958，说明数据规模增大后可能出现更长递增子序列的趋势。</p></div>			Dataset Scale	Execution Time (s)	Small	0.00	Medium	0.00	Large	0.081	Dataset Scale	LIS Length	Small	4	Medium	61	Large	1958
Dataset Scale	Execution Time (s)																		
Small	0.00																		
Medium	0.00																		
Large	0.081																		
Dataset Scale	LIS Length																		
Small	4																		
Medium	61																		
Large	1958																		
实验心得	<div>1. 动态规划的有效性与状态选择有关，选择不同的状态其效率不同，可以通过改变 dp 所存储的内容，从而改变状态空间，在一定程度上可以优化算法效率。</div> <div>2. 通过这道题，我深入理解了动态规划的应用，通过记录前面状态的选择情况，新状态通过常数级查询完成对新状态下的选择情况，并且有最优子结构的证明，动态规划法很严谨，在很多问题上都可以应用。</div>																		
实验得分		助教签名																	

附录：完整代码

1. 动态规划解决

```

#include<bits/stdc++.h>

using namespace std;
#define ll long long
const int N=1e6+10;

ll n;
ll lengthOfLIS(vector<ll> a){
    int n=a.size();
    vector<ll> b(n);
    ll max=0;
    b[0]=1;
    for(ll i=0;i<n;i++){
        ll k=0;
        for(ll j=0;j<i;j++){
            if(b[j]>k&& a[i]>a[j])k=b[j];
        }
        b[i]=k+1;
        if(b[i]>max)max=b[i];
    }

    return max;
}
int main(){
    cin>>n;
    vector<ll> a(n);
    for(int i=0;i<n;i++)cin>>a[i];
    ll ans=lengthOfLIS(a);
    cout<<ans<<endl;
}

```

2. 耐心排序解决

```

//https://zhuanlan.zhihu.com/p/670544975
#include<bits/stdc++.h>

using namespace std;
#define ll long long

```

```
int lengthOfLIS(vector<ll> a) {
    int size=0;
    vector<ll> q(a.size());
    int i, j;
    for(int x:a) {
        i=0, j=size;
        while(i<j) {
            int m=i+(j-i)/2;
            if(q[m]<x) i=m+1;
            else j=m;
        }
        q[i]=x;
        size=max(i+1, size);
    }
    return size;
}

int main() {
    ll n;
    cin>>n;
    vector<ll> a(n);
    for(int i=0; i<n; i++) cin>>a[i];
    ll ans=lengthOfLIS(a);
    cout<<ans<<endl;
}
```

3. 优化 dp 实现

```
//https://zhuanlan.zhihu.com/p/670544975
#include<iostream>
#include<vector>
using namespace std;
#define ll long long
const int N=5e5+10;
int a[N], b[N];
int n;
ll lengthOfLIS(int a[]) {
    b[1]=a[0];
    ll k=1;
    for(ll i=1; i<n; i++) {
        if(a[i]>b[k]) {
            b[++k]=a[i];
        } else {
            b[std::lower_bound(b, b+k, a[i])-b]=a[i];
            //二分查找算法，用于在已排序的范围内查找第一个不小于给定值的元素
        }
    }
    /*
    ForwardIterator lower_bound(ForwardIterator first,
                                ForwardIterator last,
```

```

        const T& value);
        返回的是一个地址，-b 得到该元素所在位置
    */
    }
}
return k;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin>>n;
    for(ll i=0;i<n;i++)cin>>a[i];
    ll ans=lengthOfLIS(a);
    cout<<ans<<endl;
}

```

4. 数据测试

```

#include <iostream>
#include <vector>
#include <fstream>
#include <algorithm>
#include <ctime>
using namespace std;
#define ll long long

// 求解最长递增子序列的长度
int lengthOfLIS(vector<ll> &a) {
    int size = 0;
    vector<ll> q(a.size() + 1);
    int i, j;
    for (int x : a) {
        i = 0, j = size;
        while (i < j) {
            int m = i + (j - i) / 2;
            if (q[m] < x)
                i = m + 1;
            else
                j = m;
        }
        q[i] = x;
        size = max(i + 1, size);
    }
}

```

```

    return size;
}

// 生成测试数据
void generate_test_data(const string &filename, ll size, ll max_value) {
    ofstream out(filename);
    if (!out) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    out << size << "\n"; // 输出数组大小
    for (ll i = 0; i < size; i++) {
        out << rand() % max_value + 1 << " "; // 随机生成 [1, max_value] 范围内的数字
    }
    out.close();
    cout << "Generated data for " << filename << " with size " << size << ".\n";
}

// 测试数据
void test_data(const string &filename) {
    ifstream in(filename);
    if (!in) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    ll n;
    in >> n;
    vector<ll> a(n);
    for (ll i = 0; i < n; i++) {
        in >> a[i];
    }
    in.close();

    // 测试算法并计时
    clock_t start_time = clock();
    ll ans = lengthOfLIS(a);
    clock_t end_time = clock();

    // 输出结果
    cout << "Test for " << filename << ":\n";
    cout << "Longest Increasing Subsequence Length: " << ans << "\n";
}

```

```
    cout << "Time taken: " << double(end_time - start_time) / CLOCKS_PER_SEC << "
seconds\n\n";
}

int main() {
    srand(time(0)); // 设置随机种子

    // 生成测试数据
    generate_test_data("lis_small.txt", 10, 100);          // 小规模
    generate_test_data("lis_medium.txt", 1000, 10000);    // 中规模
    generate_test_data("lis_large.txt", 1000000, 1000000); // 大规模

    cout << "\nData generation complete.\n\n";

    // 测试小规模数据
    test_data("lis_small.txt");

    // 测试中规模数据
    test_data("lis_medium.txt");

    // 测试大规模数据
    test_data("lis_large.txt");

    return 0;
}
```

算法分析与设计实验报告

第 4 次实验分支限界法求解 0-1 背包问题

姓名	邹林壮	学号	202208040412	班级	计算机科学与技术 (拔尖班) 2201
时间	2024. 11. 23	地点	院楼 432		
实验名称	优先队列式分支限界法求解 0-1 背包问题				
实验目的	通过本实验，深入理解和掌握分支限界法的设计思想，特别是如何结合优先队列来解决 0-1 背包问题。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	0-1 背包问题是一个经典的优化问题，目标是在不超过背包容量的情况下，选择物品使得总价值最大。分支限界法是一种通过系统地探索所有可能的解决方案空间的算法，结合优先队列可以有效地剪枝，减少搜索空间。并且能够根据不同的输入用例，能准确的输出用例中的值，并计算出程序运行所需要的时间。				
实验步骤	<p>④ 确定输入输出</p> <p>输入为物品件数 n，每件物品的质量 w，价值 v，背包的容量 c， 输出为装入物品的最大价值 $bestv$， 约束是容量 c，也就是约束函数 $cw + w[i] \leq c$，且使用优先队列分支限界法。</p> <p>⑤ 确定优先队列优先级</p> <p>我实现了使用普通队列和最大堆的优先队列求解 01 背包问题。 普通队列：采用普通队列，即按照先进先出的优先级。</p> <pre>struct node{ ll cw; ll cv; ll level; bitset<N> s; node(int cw,int cv,int level,bitset<N> s){ this->cw=cw; this->cv=cv; this->level=level; this->s=s; } }; queue<node> q;</pre>				
	<p>优先队列：将当前价值 cv 与余下最大价值 $leftv$ 之和作为优先级，每次从优先队列中选取队首（即最大优先级元素）</p> <pre>struct node{ ll cw; ll cv;</pre>				


```

ll level;
bitset<N> s;
ll up;
node(int cw,int cv,int level,bitset<N> s,int up){
    this->cw=cw;
    this->cv=cv;
    this->level=level;
    this->s=s;
    this->up=up;
}

};

struct nodecmp{
    bool operator() (const node &a,const node &b) const{
        return a.up<b.up;
    }
};

priority_queue<node,vector<node>,nodecmp> q;

```

⑥ 优先队列求解

1. 普通队列

- 1) 从根节点开始，将其加入队列。
- 2) 当队列非空时，从队列中取出一个节点。
- 3) 如果当前节点的层级大于 n ，则比较当前节点的价值与已知的最大价值，如果有更新，则更新最大价值和对应的选择。
- 4) 对于当前节点，如果选择当前物品且不超过背包容量，则生成一个新的节点（左孩子），将其加入队列。使用 `bound` 函数计算如果不选择当前物品，下一个分支的最大可能价值，如果这个值大于已知的最大价值，则生成一个新的节点（右孩子），将其加入队列。
- 5) 如果队列非空，重复 2)，直到队列为空。

2. 优先队列

- 1) 从根节点开始，将其加入队列。
- 2) 当队列非空时，从队列中取出一个节点。
- 3) 对于当前节点，如果选择当前物品且不超过背包容量，则生成一个左孩子节点，将其加入队列。使用 `bound` 函数计算如果不选择当前物品，下一个分支的最大可能价值，如果这个值大于已知的最大价值，则生成一个右孩子节点，将其加入队列。
- 4) 直到遍历到叶子节点，找到最优解，停止。

④读取输入数据

从文件 `data.txt` 中读取物品的价值和重量。

计算最大价值：调用函数计算在给定背包容量下能获得的最大价值，并记录执行时间。

输出结果：将计算结果和执行时间写入文件 `result.txt`，并在控制台输出相

	关信息。
关键代码	<pre> 1. 使用队列 2. void knap(){ 3. bitset<N> s; 4. q.push(node(0,0,1,s)); 5. while(!q.empty()){ 6. node no=q.front(); 7. q.pop(); 8. ll cw=no.cw; 9. ll cv=no.cv; 10. ll level=no.level; 11. if(level>n){ 12. if(cv>bestv){ 13. bestv=cv; 14. bs=no.s; 15. } 16. 17. continue; 18. } 19. if(cw+oj[level].w<=c){ 20. auto left_s=no.s; 21. left_s[level]=true; 22. node left_node(cw+oj[level].w,cv+oj[level].v,level+1,le ft_s); 23. q.push(left_node); 24. } 25. if(bound(level+1,cw,cv)>bestv){ 26. auto right_s=no.s; 27. right_s[level]=false; 28. node right_node(cw,cv,level+1,right_s); 29. q.push(right_node); 30. } 31. } 32. } 2. 使用优先队列 1. void knap(){ 2. bitset<N> s; 3. int init_up=bound(1,0,0); 4. q.push(node(0,0,1,s,init_up)); 5. while(!q.empty()){ 6. node no=q.top(); 7. q.pop(); </pre>

	<pre> 8. ll cw=no.cw; 9. ll cv=no.cv; 10. ll level=no.level; 11. if(level>n){ 12. if(cv>bestv){ 13. bestv=cv; 14. bs=no.s; 15. break; 16. } 17. 18. continue; 19. } 20. if(cw+oj[level].w<=c){ 21. auto left_s=no.s; 22. left_s[level]=true; 23. int left_up=bound(level+1,cw+oj[level].w,cv+o j[level].v); 24. node left_node(cw+oj[level].w,cv+oj[level].v, level+1,left_s,left_up); 25. q.push(left_node); 26. } 27. if(bound(level+1,cw,cv)>bestv){ 28. auto right_s=no.s; 29. right_s[level]=false; 30. int right_up=bound(level+1,cw,cv); 31. node right_node(cw,cv,level+1,right_s,right_u p); 32. q.push(right_node); 33. } 34. } 35. }</pre>
测试结果	<p>(1) 正确性:</p> <div>  ziz001 11-10 15:46:55 <div> <div>Unaccepted</div> <div>30</div> </div> <div>P1048 [NOIP2005 普及组] 采药</div> <div> 763ms / 125.00MB / 1.57KB C++14 (GCC 9) O2 </div> </div> <div>  ziz001 11-10 16:09:34 <div> <div>Accepted</div> <div>100</div> </div> <div>P1048 [NOIP2005 普及组] 采药</div> <div> 38ms / 620.00KB / 1.88KB C++14 (GCC 9) O2 </div> </div> <p>在洛谷上进行测试，使用普通队列时会超时，但是使用优先队列时，在时间和空间上均满足题目要求，且数据量达到要求，保证了正确性。</p> <p>(2) 复杂度:</p> <p>时间复杂度:</p> <p>1. 物品排序: 排序使用了 <code>std::sort</code>，时间复杂度为 $O(n\log n)$，其中 n 是物品数量。</p>

2. 分支限界搜索:分支限界通过优先队列 q 管理待处理的节点, 每个节点需要计算界限值 $bound$ 。该函数在最坏情况下会遍历剩余的所有物品进行计算, 复杂度为 $O(n-i)$, 平均复杂度约为 $O(\frac{n}{2}) = O(n)$ 。在分支限界过程中, 优先队列中最坏情况下需要处理的节点总数为 2^n , 但由于进行了剪枝, 大量不可能的分支会被剪枝。因此实际情况下, 尤其当背包容量 c 较小时, 节点数量通常远小于 2^n 。

3. 优先队列操作:每次插入或删除优先队列的时间复杂度为 $O(\log n)$ 。

综上所述, 最坏情况下的时间复杂度为 $O(2^n \times n)$

空间复杂度:

1. 存储物品信息

物品数组 oj 存储 n 个物品的属性, 复杂度为 $O(n)$

2. 优先队列:由于存储路径, 单个节点的空间复杂度为 $O(n)$, 优先队列最多存储 2^n 个节点, 因此空间复杂度为 $O(n)$

3. 全局数组: bs, dis, pre 均占用 $O(n)$

综上所述, 最坏情况下的空间复杂度为 $O(2^n \times n)$

(3) 构造了不同规模数据集并进行图像化演示:

构造了不同规模和大小的数据集

```
c:1000
n:10
Maximum value: 193
Elapsed time: 0 seconds
```

```
-----
Process exited after 0.1297 seconds with return value 0
请按任意键继续. . .
```

```
c:1000000
n:10000
Maximum value: 77354
Elapsed time: 0.019051 seconds
```

```
-----
Process exited after 0.1233 seconds with return value 0
请按任意键继续. . .
```

```
c:100000000
n:100000
Maximum value: 255865
Elapsed time: 21.5639 seconds
```

```
-----
Process exited after 21.9 seconds with return value 0
请按任意键继续. . .
```

可以看出时间复杂度随着 n 的增大迅速地增大, 满足 $O(2^n \times n)$ 的趋势, 但是

	<p>要比 $O(2^n \times n)$ 性能要好，空间复杂度也是如此。</p> <p>下面是构造数据集的 python 程序</p> <pre> import random def generate_knapsack_data_to_file(n, m, filename): """ 生成 0-1 背包问题的数据并保存到文件。 :param n: 背包容量 :param m: 物品数量 :param filename: 保存数据的文件名 """ # 随机生成物品的重量和价值 items = [] for _ in range(m): w = random.randint(1, n) v = random.randint(1, 1000) items.append((w, v)) # 写入文件 with open(filename, "w") as file: file.write(f"{n} {m}\n") for w, v in items: file.write(f"{w} {v}\n") print(f"数据已保存到文件: {filename}") # 设置背包容量和物品数量 n = 1000000000 # 背包容量 m = 50000 # 物品数量 filename = "./lab4/knapsack_data.txt" # 保存文件名 generate_knapsack_data_to_file(n, m, filename) </pre>		
实验心得	<ol style="list-style-type: none"> 通过本次实验，我对普通队列的分支限界算法和优先队列分支限界算法的设计与分析方法有了更深刻的理解，优先队列算法优先探索最有潜力的分支，有效减少了搜索空间，提高了算法的效率。 学会了通过分支限界法（广度优先搜索）来解决 0-1 背包问题，并学会使用选择优先级和构造最大堆和最小堆这些优先队列，优先探索最有潜力的分支，从而提高搜索效率。 这扩展了我解决问题的思路，通过合理的形式遍历解空间，并通过一定方法来缩小搜索空间，这不仅提高了我的编程能力，也增强了我对算法性能分析的兴趣和信心。 		
实验得分		助教签名	

附录：完整代码

1. 使用限界函数 1

```

#include<bits/stdc++.h>

using namespace std;
#define ll long long
const int N=1e6+10;
struct object{
    ll w;
    ll v;
};
object oj[N];
ll n;
ll c;
ll cw=0,cv=0;
ll bestv=0;
int bound(int i){
    ll nv=cv;
    for(int j=i;j<=n;j++){
        nv+=oj[j].v;
    }
    return nv;
}
void backtrack(int i){
    if(i>n){
        bestv=cv;
        return;
    }
    if(cw+oj[i].w<=c){
        cw+=oj[i].w;
        cv+=oj[i].v;
        backtrack(i+1);
        cw-=oj[i].w;
        cv-=oj[i].v;
    }
    if(bound(i+1)>bestv){
        backtrack(i+1);
    }
}

int main(){
    cin>>c>>n;
    int w,v;
    for(int i=1;i<=n;i++){
        cin>>w>>v;
        oj[i].w=w;

```

```

        oj[i].v=v;
    }
    backtrack(1);
    cout<<bestv<<endl;
}

```

2. 使用限界函数 2

```

#include<bits/stdc++.h>

using namespace std;
#define ll long long
const int N=1e6+10;
struct object{
    ll w;
    ll v;
    int pos;
    double p;
};
bool cmp(const object &a,const object &b){
    return a.p>b.p;
}
object oj[N];
ll n;
ll c;
ll cw=0,cv=0;
ll bestv=0;
vector<int> bs,s;
int bound(int i){
    ll cleft=c-cw;
    ll nv=cv;
    int j=i;
    for(;j<=n&&oj[j].w<=cleft;j++){
        cleft-=oj[j].w;
        nv+=oj[j].v;
    }
    if(j<=n)nv+=oj[j].v*cleft/oj[j].w;
    return nv;
}
void backtrack(int i){
    if(i>n){
        bestv=cv;
        bs=s;
        return;
    }
}

```

```
}
if(cw+oj[i].w<=c){
    cw+=oj[i].w;
    cv+=oj[i].v;
    s.push_back(i);
    backtrack(i+1);
    cw-=oj[i].w;
    cv-=oj[i].v;
    s.pop_back();
}
if(bound(i+1)>bestv){
    backtrack(i+1);
}
}
int main(){
    cin>>c>>n;
    int w,v;
    for(int i=1;i<=n;i++){
        cin>>w>>v;
        oj[i].w=w;
        oj[i].v=v;
        oj[i].pos=i;
        oj[i].p=(double)v/(double)w;
    }
    sort(oj+1,oj+n+1,cmp);
    backtrack(1);
    cout<<bestv<<endl;
    for(int i:bs){
        cout<<oj[i].pos<<" ";
    }
}
```


算法分析与设计实验报告

第 4 次实验分支限界法求解 TSP 问题

姓名	邹林壮	学号	202208040412	班级	计算机科学与技术 (拔尖班) 2201
时间	2024. 11. 23	地点	院楼 432		
实验名称	分支限界法求解 TSP 问题				
实验目的	通过本实验，深入理解和掌握分支限界法的设计思想，特别是如何通过分支限界法解决旅行商问题 (TSP)。同时，通过实际编程练习，提高对 C++语言的运用能力，以及对算法性能的分析能力。				
实验原理	旅行商问题 (TSP) 是一个经典的 NP-hard 问题，目标是寻找一条最短的路径，该路径恰好访问每个城市一次并返回出发城市。分支限界法是一种有效的解决方法，通过队列从而广度地探索所有可能的解决方案空间，并剪枝不可能的解。				
实验步骤	<p>①确定输入输出和约束条件： 输入为顶点数 n，边数 m，各边的起点 u，终点 v 和权值 w， 输出为遍历所有节点的最小距离 dis，以及路径， 约束条件为最短距离</p> <p>②初始化并计算 $minout$ 首先，初始化城市数量 n 和邻接矩阵 e，该矩阵存储城市间的距离。同时，初始化一个数组 $minarr$ 来存储每个城市的最小出边距离，并将其所有元素设置为 INT_MAX，通过循环读取用户输入的城市间距离，填充邻接矩阵 e，对于每个城市，找到其最小的出边距离，并更新 $minarr$ 数组，如果发现有城市没有出边（即 $minarr[i]$ 仍为 INT_MAX），则输出 NoEdge 并结束程序。计算所有城市最小出边距离的总和 $minout$，这将用作初始下界</p> <p>③分支限界搜索 当优先队列不为空时，执行以下操作： 取出队列顶部节点。 如果当前节点的 s 值大于等于 $n - 1$，表示已找到一个完整的路径，更新最佳费用 $bestc$ 并记录路径 bc。 如果当前节点的 s 值为 $n - 2$，检查是否可以通过添加最后两个城市和返回起始城市的费用来更新最佳费用。 如果当前节点的 s 值小于 $n - 2$，则探索所有可能的下一个城市，更新下界 $lcost$ 和当前费用 cw，并生成新的节点加入优先队列。</p> <p>④重复 若优先队列不为空且未取出到叶子节点，重复③</p> <p>⑤输出 搜索完成后，输出找到的最佳费用 $bestc$ 和最佳路径 bc</p>				

关键代码

1. 使用邻接矩阵

```

1. for (int i = 0; i < n; i++) { // 计算 minarr 数组和 minout
    值
2.     if (minarr[i] == INT_MAX) {
3.         cout << "NoEdge" << endl;
4.         return 0;
5.     }
6.     minout += minarr[i];
7. }
8. int t[21];
9. for (int i = 0; i < n; i++) t[i] = i;
10. q.push(node(minout, 0, 0, t));
11. while (!q.empty()) {
12.     auto no = q.top();
13.     q.pop();
14.     int lcost = no.lcost; // 下界
15.     int cw = no.cw; // 当前费用
16.     int s = no.s; // 根节点到当前路径, 用于标记当前节点的位置
17.     int x[21];
18.     for (int i = 0; i < 21; ++i) {
19.         x[i] = no.x[i];
20.     }
21.     if (s >= n - 1) {
22.         bestc = cw;
23.         for (int i = 0; i < n; ++i) {
24.             bc[i] = x[i];
25.         }
26.         break;
27.     }
28.     if (s == n - 2) {
29.         if (e[x[n - 2]][x[n - 1]] != INT_MAX && e[x[n - 1]][0] != INT_MAX &&
30.             cw + e[x[n - 2]][x[n - 1]] + e[x[n - 1]][0] <
31.             bestc) {
32.             ll nlcost = cw + e[x[n - 2]][x[n - 1]] + e[x[n - 1]][0];
33.             q.push(node(nlcost, nlcost, s + 1, x));
34.         } else continue;
35.     } else {
36.         for (int i = s + 1; i < n; i++) {
37.             if (e[x[s]][x[i]] != INT_MAX) {
38.                 int ncw = cw + e[x[s]][x[i]];
39.                 int nlcost = lcost - minarr[x[s]] + e[x[s]

```

```

    ]][x[i]];
39.         if (nlcost < bestc) {
40.             int tmp[21];
41.             for (int j = 0; j < 21; ++j) {
42.                 tmp[j] = x[j];
43.             }
44.             swap(tmp[s + 1], tmp[i]);
45.             q.push(node(nlcost, ncw, s + 1, tmp))
46.         }
47.     }
48. }
49. }
50. }
51. cout << bestc << endl;

```

2. 使用邻接表

```

1. for(int i=1;i<=n;i++){//计算 minarr 数组和 minout 值
2.     if(minarr[i]==INT_MAX){
3.         cout<<"NoEdge"<<endl;
4.         return 0;
5.     }
6.     minout+=minarr[i];
7. }
8. vector<int> t(n);
9. for(int i=0;i<n;i++)t[i]=i+1;
10. q.push(node(minout,0,0,t));
11. while(!q.empty()){
12.     auto no=q.top();
13.     q.pop();
14.     ll lcost=no.lcost;//下界
15.     ll cw=no.cw;//当前费用
16.     ll s=no.s;//根节点到当前路径，用于标记当前节点的位置
17.     auto x=no.x;//记录路径
18.     if(s==n-1){
19.         bestc=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1];
20.         bc=x;
21.         break;
22.     }
23.     if(s==n-2){
24.         if(e[x[n-2]][x[n-1]]!=INT_MAX&&e[x[n-1]][1]!=INT_
MAX&&
25.             cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1]<bestc){
26.             ll ncw=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1];
27.             q.push(node(ncw,ncw,s+1,x));

```

	<pre>28. }else continue; 29. }else{ 30. for(int i=s+1;i<=n;i++){ 31. if(e[x[s]][x[i]]!=INT_MAX){ 32. ll ncw=cw+e[x[s]][x[i]]; 33. ll nlcost=lcost-minarr[x[s]]+e[x[s]][x[i]]; 34. if(nlcost<bestc){ 35. auto tmp=x; 36. swap(tmp[s+1],tmp[i]); 37. q.push(node(nlcost,ncw,s+1,tmp)); 38. } 39. } 40. } 41. } 42. }</pre>																																																								
测试结果	<div><div><div><div><div>(1) 正确性:</div><div><div><div>测试点信息</div><div>源代码</div></div><div><div>测试点信息</div><table><tr><td>#1</td><td>AC</td><td>#2</td><td>AC</td><td>#3</td><td>AC</td><td>#4</td><td>AC</td><td>#5</td><td>AC</td><td>#6</td><td>AC</td><td>#7</td><td>AC</td></tr><tr><td></td><td>3ms/564.00KB</td><td></td><td>4ms/616.00KB</td><td></td><td>4ms/528.00KB</td><td></td><td>3ms/680.00KB</td><td></td><td>3ms/780.00KB</td><td></td><td>7ms/5.08MB</td><td></td><td>17ms/13.58MB</td></tr><tr><td>#8</td><td>AC</td><td>#9</td><td>AC</td><td>#10</td><td>MLE</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td>45ms/25.95MB</td><td></td><td>30ms/25.79MB</td><td></td><td>198ms/125.00MB</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></div></div><div><div>zlj001</div><div>所属题目</div><div>P1171 售货员的难题</div><div>评测状态</div><div>Unaccepted</div><div>评测分数</div><div>90</div><div>提交时间</div><div>2024-11-22 16:50:33</div></div></div></div><div><div>在洛谷上进行测试，满足正确性，但是该方法对数据规模有限制。</div><div>(2) 复杂度:</div><div>时间复杂度:</div><div>1. 初始化和读取：代码首先初始化邻接矩阵 e 和最小边数组 minarr，这部分的时间复杂度为$O(n^2)$。</div><div>2. 优先队列操作：在优先队列中，在最坏情况下，每个节点可能生成 $n-1$ 个子节点（因为除了当前节点外，其他所有节点都可能是下一个节点）。因此，优先队列的大小可能达到 $O(n!)$，但实际上由于分支限界法的使用，很多节点会被剪枝，实际大小远小于 $O(n!)$。</div><div>3. 优先级下界计算：计算下界 lcost 的时间复杂度为 $O(n)$，因为需要从当前节点到下一个节点的最小费用。</div><div>总而言之，总的时间复杂度在最坏情况下接近 $O(n!)$，但是由于分支限界法的有效剪枝，实际的时间复杂度通常会小得多。</div><div>空间复杂度:</div><div>1. 邻接矩阵：邻接矩阵 e 需要 $O(n^2)$的空间来存储 $n \times n$ 的矩阵。</div><div>2. 优先队列：优先队列中最多可能包含 $O(n!)$个节点，因此其空间复杂度为 $O(n!)$。</div></div></div></div>	#1	AC	#2	AC	#3	AC	#4	AC	#5	AC	#6	AC	#7	AC		3ms/564.00KB		4ms/616.00KB		4ms/528.00KB		3ms/680.00KB		3ms/780.00KB		7ms/5.08MB		17ms/13.58MB	#8	AC	#9	AC	#10	MLE										45ms/25.95MB		30ms/25.79MB		198ms/125.00MB								
#1	AC	#2	AC	#3	AC	#4	AC	#5	AC	#6	AC	#7	AC																																												
	3ms/564.00KB		4ms/616.00KB		4ms/528.00KB		3ms/680.00KB		3ms/780.00KB		7ms/5.08MB		17ms/13.58MB																																												
#8	AC	#9	AC	#10	MLE																																																				
	45ms/25.95MB		30ms/25.79MB		198ms/125.00MB																																																				

	<p>3. 路径数组：每个节点都需要一个大小为 n 的数组来存储路径，因此空间复杂度为 $O(n)$。</p> <p>总而言之，总的空间复杂度主要由邻接矩阵和优先队列决定，为 $O(n!)$。在实际应用中，由于优先队列的大小通常远小于 $O(n!)$，空间复杂度通常小于理论最大值。</p> <p>(3) 构造了不同规模数据集并进行图像化演示： 构造不同规模和大小的数据集</p> <pre> v:5 e:10 Minimum cost: 218 Execution time: 0 ms Optimal path: 1 2 4 3 5 </pre> <pre> v:10 e:20 Minimum cost: 2147483647 Execution time: 10 ms Optimal path: </pre> <pre> v:20 e:40 Minimum cost: 2147483647 Execution time: 142 ms Optimal path: ----- Process exited after 3.198 seconds with return value 3221225477 请按任意键继续. . . </pre> <p>可以看到，采用该方法对空间上要求大，最坏情况下空间复杂度为 $O(n!)$，时间复杂度上符合最坏情况下是 $O(n!)$，但是性能一般会好得多。</p>
实验心得	<p>1. 通过这次实验，深入学习了 TSP 问题的理论基础，理解了其作为 NP-hard 问题的核心难点，以及优先队列式分支限界法与启发式算法具有类似之处。</p> <p>2. 此外，还区分了邻接矩阵和邻接表，邻接矩阵解决问题更加直观，但是处理不了大规模的稀疏矩阵，会占用大量空间，类似于洛谷上的做法，而邻接表可以解决这个问题，因此最后补充了邻接表的实现，但是只是针对边来遍历，不能通过两个顶点来定位。</p> <p>3. 通过对算法性能的分析，我更加直观地理解了优先队列式分支限界法的时间复杂度和空间复杂度。我学会了如何通过理论分析和实际测试来评估算法的性能，并理解了算法参数对性能的影响。</p>
实验得分	<div>助教签名</div>

附录：完整代码

2. 使用邻接矩阵来储存图

```

#include<bits/stdc++.h>

using namespace std;
#define ll long long
const int N=1e4+10;
vector<vector<int>>> e(N,vector<int>(N,INT_MAX));

struct node{
    ll lcost;//下界
    ll cw;//当前费用
    ll s;//根节点到当前路径，用于标记当前节点的位置
    vector<int> x;//记录路径
    node(ll lcost,ll cw,ll s,vector<int> x){
        this->lcost=lcost;
        this->cw=cw;
        this->s=s;
        this->x=x;
    }
};

struct nodecmp{
    bool operator() (const node& a,const node& b){
        return a.lcost > b.lcost;
    }
};

ll n,m;//顶点数和边数
ll u,v,w;
ll minout;

ll bestc=INT_MAX;//最小费用
vector<int> bc;
priority_queue<node,vector<node>,nodecmp> q;
int main(){
    cin>>n>>m;
    vector<ll> minarr(n+1,INT_MAX);
    for(int i=0;i<m;i++){
        cin>>u>>v>>w;
        e[u][v]=w;
        e[v][u]=w;
        if(w<minarr[u])minarr[u]=w;
        if(w<minarr[v])minarr[v]=w;
    }
    for(int i=1;i<=n;i++){//计算 minarr 数组和 minout 值

```

```

        if(minarr[i]==INT_MAX){
            cout<<"NoEdge"<<endl;
            return 0;
        }
        minout+=minarr[i];
    }
    vector<int> t(n);
    for(int i=0;i<n;i++)t[i]=i+1;
    q.push(node(minout,0,0,t));
    while(!q.empty()){
        auto no=q.top();
        q.pop();
        ll lcost=no.lcost;//下界
        ll cw=no.cw;//当前费用
        ll s=no.s;//根节点到当前路径，用于标记当前节点的位置
        auto x=no.x;//记录路径
        if(s==n-1){
            bestc=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1];
            bc=x;
            break;
        }
        if(s==n-2){
            if(e[x[n-2]][x[n-1]]!=INT_MAX&&e[x[n-1]][1]!=INT_MAX&&
            cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1]<bestc){
                ll ncw=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1];
                q.push(node(ncw,ncw,s+1,x));
            }else continue;
        }else{
            for(int i=s+1;i<=n;i++){
                if(e[x[s]][x[i]]!=INT_MAX){
                    ll ncw=cw+e[x[s]][x[i]];
                    ll nlcost=lcost-minarr[x[s]]+e[x[s]][x[i]];
                    if(nlcost<bestc){
                        auto tmp=x;
                        swap(tmp[s+1],tmp[i]);
                        q.push(node(nlcost,ncw,s+1,tmp));
                    }
                }
            }
        }
    }
    cout<<bestc<<endl;
    for(int i=1;i<=n;i++){
        cout<<bc[i]<<" ";
    }

```

```

    }
}

```

2. 使用邻接表存储图

```

#include <bits/stdc++.h>
using namespace std;
#define ll long long
const int N = 1e4 + 10;

struct node {
    ll lcost;    // 下界
    ll cw;       // 当前费用
    ll s;        // 根节点到当前路径，用于标记当前节点的位置
    vector<int> x; // 记录路径
    node(ll lcost, ll cw, ll s, vector<int> x) : lcost(lcost), cw(cw), s(s), x(x) {}
};

struct nodecmp {
    bool operator()(const node& a, const node& b) {
        return a.lcost > b.lcost; // 最小堆
    }
};

ll n, m; // 顶点数和边数
ll u, v, w;
ll minout = 0; // 所有点的最小出边权重和
ll bestc = INT_MAX; // 最小费用
vector<int> bc; // 最优路径
priority_queue<node, vector<node>, nodecmp> q;

int main() {
    cin >> n >> m;
    vector<vector<pair<int, int>>> adj(n + 1); // 邻接表
    vector<ll> minarr(n + 1, INT_MAX); // 每个节点的最小出边权重

    // 输入边
    for (int i = 0; i < m; i++) {
        cin >> u >> v >> w;
        adj[u].emplace_back(v, w);
        adj[v].emplace_back(u, w);
        minarr[u] = min(minarr[u], (ll)w);
        minarr[v] = min(minarr[v], (ll)w);
    }
}

```



```

}

// 检查是否存在孤立点
for (int i = 1; i <= n; i++) {
    if (minarr[i] == INT_MAX) {
        cout << "NoEdge" << endl;
        return 0;
    }
    minout += minarr[i];
}

vector<int> t(n);
iota(t.begin(), t.end(), 1); // 初始化路径 1, 2, ..., n
q.push(node(minout, 0, 0, t));

while (!q.empty()) {
    auto no = q.top();
    q.pop();
    ll lcost = no.lcost; // 下界
    ll cw = no.cw;      // 当前费用
    ll s = no.s;        // 当前路径位置
    auto x = no.x;      // 路径

    if (s == n - 1) {
        // 完成路径, 计算总费用
        for (auto& p : adj[x[n - 1]]) {
            if (p.first == x[0]) {
                bestc = min(bestc, cw + p.second);
                bc = x;
                break;
            }
        }
        break;
    }

    if (s == n - 2) {
        // 剩余最后两个点, 直接连接起点
        for (auto& p1 : adj[x[n - 2]]) {
            if (p1.first == x[n - 1]) {
                for (auto& p2 : adj[x[n - 1]]) {
                    if (p2.first == x[0] && cw + p1.second + p2.second < bestc) {
                        ll ncw = cw + p1.second + p2.second;
                        q.push(node(ncw, ncw, s + 1, x));
                    }
                }
            }
        }
    }
}

```

```

    }
    }
    }
} else {
    // 扩展路径
    for (int i = s + 1; i < n; i++) {
        for (auto& p : adj[x[s]]) {
            if (p.first == x[i]) {
                ll ncw = cw + p.second; // 新费用
                ll nlcost = lcost - minarr[x[s]] + p.second; // 新下界
                if (nlcost < bestc) {
                    auto tmp = x;
                    swap(tmp[s + 1], tmp[i]);
                    q.push(node(nlcost, ncw, s + 1, tmp));
                }
            }
        }
    }
}
}

// 输出结果
cout << bestc << endl;
for (int i = 0; i < n; i++) {
    cout << bc[i] << " ";
}
cout << endl;

return 0;
}

```

算法分析与设计实验报告

第 4 次实验正则表达式匹配问题

姓名	邹林壮	学号	202208040412	班级	计算机科学与技术 (拔尖班) 2201
时间	2024. 11. 23	地点	院楼 432		
实验名称	实现题 3-11 正则表达式匹配问题				
实验目的	通过本实验，深入理解和掌握动态规划和回溯法的设计思想，特别是如何结合回溯法来解决正则表达式匹配问题。同时，通过实际编程练习，提高对 C++ 语言的运用能力，以及对算法性能的分析能力。				
实验原理	正则表达式匹配问题是一个经典的字符串处理问题，目标是构建一个正则表达式，使其可以匹配尽可能多的操作文件，同时避免匹配到任何非操作文件。。动态规划用于构建匹配表以存储中间结果，而回溯法用于探索所有可能的正则表达式构建路径。				
实验步骤	<p>⑦ 确定输入输出</p> <p>输入为文件名数 n，n 行文件名，文件名为英文字母和数字，操作文件（用+标记），非操作文件（用-标记）</p> <p>输出为最多文件匹配 $maxmat$，匹配最多操作文件且不匹配任何非操作文件的正则表达式 s。</p> <p>约束是使能匹配的待操作数最多，但不能匹配任何不进行操作的文件。</p> <p>⑧ 初始化：</p> <p>$n[0]$表示操作文件数，$n[1]$表示非操作数，将操作文件存储在数组 $f[1]$到 $f[n[0]]$中，将非操作文件存储在数组 $k[1]$到 $k[n[1]]$中。然后在 $save$ 函数中遍历所有操作文件，统计每个字符的出现频率，按频率降序排列，从而加快搜索进程。初始化动态规划数组 $match[len][i][j]$，表示正则表达式长度为 len 时，第 i 个文件的第 j 个字符的匹配情况。</p> <p>⑨ 判断文件是否匹配</p> <p>$check$ 函数实现，</p> <p>遍历所有操作文件，对于当前正则表达式 s：</p> <p>若字符为 $*$，表示可匹配任意长度子串；</p> <p>若字符为 $?$，表示可匹配任意单个字符；</p> <p>若字符为普通字符，则需要精确匹配。</p> <p>若正则表达式 s 匹配了更多操作文件，更新匹配数 $curmat$，将当前正则表达式 s 的匹配状态记录到 $match$ 数组中</p> <p>⑩ 检测非操作文件</p> <p>ok 函数中实现，</p> <p>遍历所有非操作文件，检测当前正则表达式是否匹配：若匹配到任何非操作文件，则当前正则表达式无效。回溯，跳过无效正则表达式。</p> <p>⑪ 回溯生成最优正则表达式</p> <p>在 $search$ 函数中实现，</p> <p>递归回溯搜索：在当前正则表达式基础上添加一个字符，若添加 $*$ 或 $?$，重</p>				

	<p>新计算匹配数, 按字符频率优先选择高频字符。若新生成的正则表达式更优 (匹配更多文件且长度更短), 更新为当前最优解。</p> <p>剪枝优化: 若当前正则表达式的匹配数小于历史最佳, 提前停止搜索; 若当前正则表达式长度超过当前最短长度, 则跳过。</p> <p>⑫ 输出结果</p> <p>输出 maxmat, 表示正则表达式匹配的文件数, 输出最优正则表达式 minmat。</p>
关键代码	<pre> 1. void save(char c, int len) { 2. int i, j; 3. for(i=1; i<=p[len]; i++) 4. if(cha[len][i].c == c) { 5. cha[len][i].f++; 6. cha[len][0] = cha[len][i]; 7. j = i; 8. while(cha[len][j-1].f < cha[len][0].f) { //将 频率小于当前字符频率的字母后移 9. cha[len][j] = cha[len][j-1]; 10. j--; 11. } 12. cha[len][j] = cha[len][0]; //将当前字母放到适 当位置 13. return; 14. } 15. //如果 p[len] = 0, 或者字符 c 第一次出现, 则增加一个字符 16. cha[len][++p[len]].c = c; 17. cha[len][p[len]].f = 1; 18. } 19. 20. //计算当前匹配情况 21. bool check(int len) { 22. int i, j, t, k = 0; 23. curmat = 0; 24. for(i=1; i<=n[0]; i++) { //遍历操作文件 25. for(j=0; j<MAXP; j++) 26. match[len][i][j] = 0; //初始化匹配表 27. if(len==1 && s[1]=='*') //如果是*且正则长度为 1, 匹配 全部 28. match[len][i][0] = 1; 29. for(j=1; j<=f[i].length(); j++) { //操作文件的每个 字符 30. switch(s[len]) { 31. case '*': //匹配任意长度的子串 32. for(t=0; t<=j; t++) 33. if(match[len-1][i][t]==1) { 34. match[len][i][j] = 1; //正则 </pre>

		<p>表达式的第 len 个字符与第 i 个文件的第 j 个字符的匹配情况</p> <pre> 35. break; 36. } 37. break; //此处的 break 不能省略! 38. case '?'://匹配单个字符 39. match[len][i][j] = match[len-1][i][j- 1]; 40. break; 41. default: 42. if(s[len]==f[i][j-1])//精准字符匹配 43. match[len][i][j] = match[len-1][i][j-1]; 44. break; 45. } 46. } 47. //检查当前文件是否完全匹配 48. for(j=f[i].length(); j>=1; j--) { 49. if(match[len][i][j] == 1) { 50. k++; 51. if(j == f[i].length()) //说明第 i 个文件与正 则表达式匹配 52. curmat++; 53. break; 54. } 55. } 56. } 57. //如果当前匹配情况比历史最佳情况差, 返回 false 58. if(k<maxmat k==maxmat && len>=minlen) 59. return false; 60. p[len] = 0;//更新字符频率 61. for(i=1; i<=n[0]; i++) //对与正则表达式匹配的文件中的字 符重新排序, 以便正则式下次扩展 62. for(j=1; j<f[i].length(); j++) 63. if(match[len][i][j]==1) 64. save(f[i][j], len); 65. return true; 66. } 67. 68. //判断是否匹配非操作文件 69. bool ok(int len) { 70. int i, j, l, t; 71. for(l=1; l<=len; l++) { 72. for(i=n[0]+1; i<=n[0]+n[1]; i++) { 73. for(j=0; j<MAXP; j++) </pre>
--	--	--

```

74.         match[l][i][j] = 0; //初始化匹配表
75.         if(s[l]=='*' && l==1)
76.             match[l][i][0] = 1;
77.         for(j=1; j<=f[i].length(); j++) {
78.             switch(s[l]) {
79.                 case '*':
80.                     for(t=0; t<=j; t++)
81.                         if(match[l-1][i][t]==1) {
82.                             match[l][i][j] = 1;
83.                             break;
84.                         }
85.                     break;
86.                 case '?':
87.                     match[l][i][j] = match[l-1][i][j-
1];
88.                     break;
89.                 default:
90.                     if(s[l]==f[i][j-1])
91.                         match[l][i][j] = match[l-1][i
][j-1];
92.                     break;
93.             }
94.         }
95.     }
96. }
97. //如果正则表达式匹配到非操作文件, 返回 false
98. for(i=n[0]+1; i<=n[0]+n[1]; i++) //如果正则表达式与非
操作文件匹配
99.     if(match[len][i][f[i].length()]==1)
100.         return false;
101.     return true;
102. }
103.
104. //求最优匹配的回溯法
105. //len 为当前正则表达式长度
106. //正则表达式可选字符集的排列顺序先为 '*', '? ', 操作文件名序
列中出现的所有字符按其频率递减的次序随后
107. void search(int len) {
108.     // 如果当前正则表达式更优(匹配更多文件名/有更短表达式),
更新最优解
109.     if((curmat>maxmat || (curmat==maxmat && len<minlen
)) && ok(len)) {
110.         maxmat = curmat;
111.         minlen = len;

```

	<pre>112. for(int i=0; i<=minlen; i++) 113. minmat[i] = s[i]; 114. } 115. len++; 116. //尝试添加?和* 117. if(len==1 s[len-1]!='*') { 118. s[len] = '?'; 119. if(check(len)) 120. search(len); 121. s[len] = '*'; 122. if(check(len)) 123. search(len); 124. } 125. // 尝试添加其他字符 126. for(int i=1; i<=p[len-1]; i++) { 127. s[len] = cha[len-1][i].c; 128. if(check(len)) 129. search(len); 130. } 131. }</pre>																								
测试结果	<p>(1)正确性：</p> <p>测试了所给样例，在洛谷上测试，处理最后一个大样例超时，其他都成功运行，保证了正确性。</p> <p>测试点信息</p> <p>Subtask #0</p> <table><tr><td>#1 AC 6ms/4.11MB</td><td>#2 AC 6ms/4.02MB</td><td>#3 AC 6ms/4.13MB</td><td>#4 AC 6ms/4.11MB</td><td>#5 AC 6ms/4.11MB</td><td>#6 AC 6ms/4.10MB</td><td>#7 AC 6ms/4.10MB</td></tr><tr><td>#8 AC 5ms/4.11MB</td><td>#9 AC 6ms/4.10MB</td><td>#10 AC 6ms/4.10MB</td><td>#11 AC 5ms/4.11MB</td><td>#12 AC 6ms/4.10MB</td><td>#13 AC 8ms/4.11MB</td><td>#14 AC 7ms/4.10MB</td></tr><tr><td>#15 AC 15ms/4.13MB</td><td>#16 AC 6ms/4.10MB</td><td>#17 AC 13ms/4.13MB</td><td>#18 AC 13ms/4.02MB</td><td>#19 AC 36ms/4.11MB</td><td>#20 AC 28ms/4.10MB</td><td></td></tr></table> <p>Subtask #1</p> <table><tr><td>#21 AC 5ms/4.10MB</td><td>#22 AC 14ms/4.14MB</td><td>#23 TLE 3.20s/4.09MB</td></tr></table> <p>(2)复杂度：</p> <p>时间复杂度：</p> <p>1. 读取文件名和初始化：这部分的时间复杂度是 $O(M)$，其中 M 是文件名的总数。这是因为代码需要逐个读取每个文件名。</p> <p>2. 匹配检查（check 函数）：对于每个正则表达式长度 len，需要检查所有文</p>	#1 AC 6ms/4.11MB	#2 AC 6ms/4.02MB	#3 AC 6ms/4.13MB	#4 AC 6ms/4.11MB	#5 AC 6ms/4.11MB	#6 AC 6ms/4.10MB	#7 AC 6ms/4.10MB	#8 AC 5ms/4.11MB	#9 AC 6ms/4.10MB	#10 AC 6ms/4.10MB	#11 AC 5ms/4.11MB	#12 AC 6ms/4.10MB	#13 AC 8ms/4.11MB	#14 AC 7ms/4.10MB	#15 AC 15ms/4.13MB	#16 AC 6ms/4.10MB	#17 AC 13ms/4.13MB	#18 AC 13ms/4.02MB	#19 AC 36ms/4.11MB	#20 AC 28ms/4.10MB		#21 AC 5ms/4.10MB	#22 AC 14ms/4.14MB	#23 TLE 3.20s/4.09MB
#1 AC 6ms/4.11MB	#2 AC 6ms/4.02MB	#3 AC 6ms/4.13MB	#4 AC 6ms/4.11MB	#5 AC 6ms/4.11MB	#6 AC 6ms/4.10MB	#7 AC 6ms/4.10MB																			
#8 AC 5ms/4.11MB	#9 AC 6ms/4.10MB	#10 AC 6ms/4.10MB	#11 AC 5ms/4.11MB	#12 AC 6ms/4.10MB	#13 AC 8ms/4.11MB	#14 AC 7ms/4.10MB																			
#15 AC 15ms/4.13MB	#16 AC 6ms/4.10MB	#17 AC 13ms/4.13MB	#18 AC 13ms/4.02MB	#19 AC 36ms/4.11MB	#20 AC 28ms/4.10MB																				
#21 AC 5ms/4.10MB	#22 AC 14ms/4.14MB	#23 TLE 3.20s/4.09MB																							

件名，每个文件名的长度最多为 MAXL。因此，这部分的时间复杂度是 $O(N * L * P)$ ，其中 N 是文件名的数量， L 是文件名的最大长度， P 是可能的字符集大小（在这个程序中是 MAXP）。

3. 回溯搜索 (search 函数)：对于每个正则表达式长度 len ，程序尝试添加三种类型的字符：'?'，'*' 和其他字符。对于每种类型的字符，它都会递归地调用 check 函数。最坏情况下，search 函数的递归深度可以达到 MAXL（文件名的最大长度），因此这部分的时间复杂度是 $O(3^L * N * L * P)$ ，其中 3^L 来自于对于每个位置，有三种选择（'?' 或 '*' 或其他字符）。

综合以上，整体时间复杂度大约是 $O(M + 3^L * N * L * P)$ 。

空间复杂度：

1. 文件名存储： M 是文件名的数量， L 是文件名的最大长度，存储文件名的空间复杂度是 $O(M * L)$ 。

2. 匹配表：match 数组是一个三维数组，其大小为 $O(L * N * P)$ 。

3. 字符频率表：cha 数组的大小为 $O(L * P)$ 。

4. 正则表达式存储：s 和 minmat 数组的大小为 $O(P)$

整体空间复杂度大约是 $O(M * L + L * N * P)$

(3) 构造了不同规模数据集并进行图像化演示：

构造了不同规模和大小的数据集，

数据大小为 10

```
输入文件：
10 I
T +
C -
ARUYS DL -
XJS -
CWPVAEYZ -
GWUSRETP -
ETF +
VMGKES +
YT +
NQUYPMJ +
NQUYPMJ +
匹配最多文件数：3
正则表达式为： *M*
程序运行时间： 0.007472 秒
```

数据大小为 100

```
QGWAODMI +
JQJL -
JPJV -
HCRST +
RD +
SOWGO +
SOWGO +
匹配最多文件数：5
正则表达式为： ??W*
程序运行时间： 0.114539 秒
```

数据大小为 250


```

TECJMDIN +
SBUUSMG -
QUQV +
KEBLSG -
UJAJCW +
OWAU +
E +
XB +
FACHHLCV +
TXA -
P +
IRQHER -
PDVAC +
PDVAC +
匹配最多文件数: 8
正则表达式为: ???A*
程序运行时间: 0.338483 秒

```

分析数据可知，满足时间复杂度 $O(M + 3^L * N * L * P)$ 的大致趋势。
这是生成数据的代码

```

import random
import string

def generate_data_with_signs(n, filename):
    # 随机生成长度不超过 8 的字符串
    def random_string():
        length = random.randint(1, 8) # 字符串长度在 1 到 8 之间
        return ''.join(random.choices(string.ascii_uppercase,
                                        k=length))

    # 生成数据
    data = [f"{n}"] # 第一行是数据规模
    for _ in range(n):
        string_part = random_string()
        sign = random.choice(["+", "-"])
        data.append(f"{string_part} {sign}")

    # 保存到文件
    with open(filename, "w") as file:
        file.write("\n".join(data))

    print(f"数据已保存到 {filename}")

# 示例：生成 5 行数据并保存到 data.txt
generate_data_with_signs(250, "./lab4/data.txt")

```

实验心得	<ol style="list-style-type: none"> 1. 针对问题较为复杂的问题，要首先明确问题的解决方向和解决框架，不能盲目地做，应该仔细地想结局问题的方法。 2. 回溯法在解决某些问题时的局限性，尤其是在面对具有高复杂度的算法时。在这个问题中，回溯法导致了较高的时间复杂度，这可能在处理大规模数据时成为瓶颈。 3. 认识到了问题规模对算法选择的影响。对于小规模问题，即使复杂度高的算法也能快速解决问题；但对于大规模问题，需要选择更高效的算法。 		
实验得分		助教签名	

附录：完整代码

```

#include "iostream"
#include "string"
#include "iomanip"
#include "fstream"
using namespace std;

/*
正则表达式可选字符集的排列顺序先为 '*'，'?'，操作文件名序列中出现的所有字符按其
频率递减的次序随后

大体思路：
正则表达式为 s，当前考察文件为 f

match(i, j) 为 s[1, i] 与 f[1, j] 匹配情况
如果 match(i-1, j-1) = 1, s[i] = '?'
    match(i-1, j-1) = 1, s[i] = f[j]
    match(i-1, k) = 1, s[i] = '*'
match(i, j) = 1
否则 match(i, j) = 0
*/

const int MAXN = 250; //最大文件数
const int MAXL = 8; //最大文件名长度
const int MAXP = 62; //大写字母 + 小写字母 + 数字
int minlen; //最优正则表达式长度
int maxmat; //最优正则表达式所能匹配的操作文件数
int curmat; //当前正则式所能匹配的操作文件数
string f[MAXN+1]; //操作文件 + 非操作文件
string k[MAXN+1]; //非操作文件
int n[2]; //n[0]为操作文件数, n[1]为非操作文件数
int p[MAXP]; //p[len-1] 存储 s[len]可选字符数
char s[MAXP]; //存储临时正则表达式

```

```

char minmat[MAXP]; //最优正则表达式
int match[MAXP][MAXN+1][MAXP]; //match[len][i][j]表示正则表达式的第len个字符与第i
个文件的第j个字符的匹配情况

struct Cha {
    char c; //字符
    int f; //字符出现的频率
};

Cha cha[MAXN][MAXP]; //cha[len][i]表示正则表达式长度为len时的字符排序情况

//对操作文件名中出现的字符按出现频率排序存储，以加快搜索进程
void save(char c, int len) {
    int i, j;
    for(i=1; i<=p[len]; i++)
        if(cha[len][i].c == c) {
            cha[len][i].f++;
            cha[len][0] = cha[len][i];
            j = i;
            while(cha[len][j-1].f < cha[len][0].f) { //将频率小于当前字符频率的字母
后移
                cha[len][j] = cha[len][j-1];
                j--;
            }
            cha[len][j] = cha[len][0]; //将当前字母放到适当位置
            return;
        }
    //如果p[len] = 0, 或者字符c第一次出现, 则增加一个字符
    cha[len][++p[len]].c = c;
    cha[len][p[len]].f = 1;
}

//计算当前匹配情况
bool check(int len) {
    int i, j, t, k = 0;
    curmat = 0;
    for(i=1; i<=n[0]; i++) { //遍历操作文件
        for(j=0; j<MAXP; j++)
            match[len][i][j] = 0; //初始化匹配表
        if(len==1 && s[1]=='*') //如果是*且正则长度为1, 匹配全部
            match[len][i][0] = 1;
        for(j=1; j<=f[i].length(); j++) { //操作文件的每个字符
            switch(s[len]) {
                case '*': //匹配任意长度的子串

```

```

        for(t=0; t<=j; t++)
            if(match[len-1][i][t]==1) {
                match[len][i][j] = 1; //正则表达式的第 len 个字符与第 i
个文件的第 j 个字符的匹配情况
                break;
            }
        break; //此处的 break 不能省略!
    case '?': //匹配单个字符
        match[len][i][j] = match[len-1][i][j-1];
        break;
    default:
        if(s[len]==f[i][j-1]) //精准字符匹配
            match[len][i][j] = match[len-1][i][j-1];
        break;
    }
}
//检查当前文件是否完全匹配
for(j=f[i].length(); j>=1; j--) {
    if(match[len][i][j] == 1) {
        k++;
        if(j == f[i].length()) //说明第 i 个文件与正则表达式匹配
            curmat++;
        break;
    }
}
}
//如果当前匹配情况比历史最佳情况查, 返回 false
if(k<maxmat || k==maxmat && len>=minlen)
    return false;
p[len] = 0; //更新字符频率
for(i=1; i<=n[0]; i++) //对与正则表达式匹配的文件中的字符重新排序, 以便正则式下
次扩展
    for(j=1; j<f[i].length(); j++)
        if(match[len][i][j]==1)
            save(f[i][j], len);
return true;
}

//判断是否匹配非操作文件
bool ok(int len) {
    int i, j, l, t;
    for(l=1; l<=len; l++) {
        for(i=n[0]+1; i<=n[0]+n[1]; i++) {
            for(j=0; j<MAXP; j++)

```

```

        match[l][i][j] = 0; //初始化匹配表
    if(s[l]=='*' && l==1)
        match[l][i][0] = 1;
    for(j=1; j<=f[i].length(); j++) {
        switch(s[l]) {
            case '*':
                for(t=0; t<=j; t++)
                    if(match[l-1][i][t]==1) {
                        match[l][i][j] = 1;
                        break;
                    }
                break;
            case '?':
                match[l][i][j] = match[l-1][i][j-1];
                break;
            default:
                if(s[l]==f[i][j-1])
                    match[l][i][j] = match[l-1][i][j-1];
                break;
        }
    }
}

//如果正则表达式匹配到非操作文件，返回 false
for(i=n[0]+1; i<=n[0]+n[1]; i++) //如果正则表达式与非操作文件匹配
    if(match[len][i][f[i].length()]==1)
        return false;
return true;
}

//求最优匹配的回溯法
//len 为当前正则表达式长度
//正则表达式可选字符集的排列顺序先为 '*'， '?'， 操作文件名序列中出现的所有字符按其频率递减的次序随后
void search(int len) {
    // 如果当前正则表达式更优(匹配更多文件名/有更短表达式)，更新最优解
    if((curmat>maxmat || (curmat==maxmat && len<minlen)) && ok(len)) {
        maxmat = curmat;
        minlen = len;
        for(int i=0; i<=minlen; i++)
            minmat[i] = s[i];
    }
    len++;
    //尝试添加?和*

```

```

    if(len==1 || s[len-1]!='*') {
        s[len] = '?';
        if(check(len))
            search(len);
        s[len] = '*';
        if(check(len))
            search(len);
    }
    // 尝试添加其他字符
    for(int i=1; i<=p[len-1]; i++) {
        s[len] = cha[len-1][i].c;
        if(check(len))
            search(len);
    }
}

int main() {
    ifstream fin("zz.txt");
    cout << "输入文件: \n";
    n[0] = 0;
    n[1] = 0;
    p[0] = 0;
    string str;
    char ch;
    while(!fin.eof()) { //读入
        fin >> str >> ch;

        if(ch == '+') {
            f[++n[0]] = str;
            cout << f[n[0]] << " " << ch << endl;
            save(f[n[0]][0], 0);
        } else {
            k[++n[1]] = str;
            cout << k[n[1]] << " " << ch << endl;
        }
    }
    for(int i=1; i<=n[1]; i++)
        f[n[0]+i] = k[i];
    memset(match, 0, sizeof(match));
    for(int i=1; i<=n[0]+n[1]; i++)
        match[0][i][0] = 1;
    maxmat = 0;
    minlen = 255;
}

```

```
search(0);  
cout << "匹配最多文件数: " << maxmat << endl;  
cout << "正则表达式为: ";  
for(int i=0; i<=minlen; i++)  
    cout << minmat[i];  
cout << endl;  
fin.close();  
return 0;  
}
```