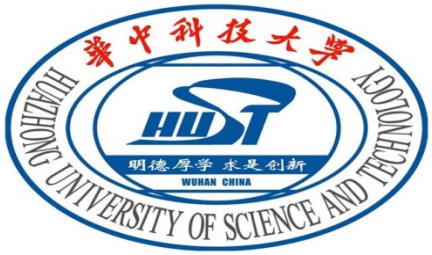
華中科技大學

《算法设计与分析实践》

总 结 报 告



专业班级：计算机科学与技术 2208 班

学　　号：U202215642

姓　　名：田清林

指导教师：王多强

完成日期：2024 年 1 月 7 日

计算机科学与技术学院

目 录

1. 完成情况

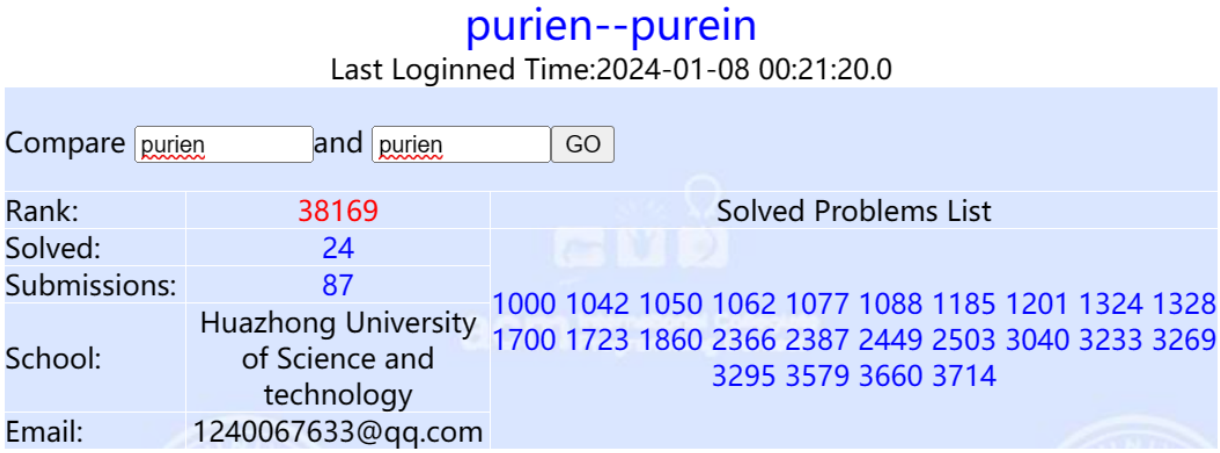


图 1 poj 解决题目截图

在给出的算法题目中，本人一共选择 24 道题，并 ac 了 24 道题，每个模块的题目均在两道以上，不存在已做而未 ac 的题目。

2. poj3579 Median 解题报告

2.1 题目分析

题意: 给出 n 个数 x1…xn, 求所有|xi - xj|中中位数的大小, 如果总数 m 为偶数, 则为位于(m+1)/2 处的数。

2.2 算法设计

使用分治策略。显然，将 n*(n-1)/2 个数算出再求出中位数极其消耗时间与空间，但可以看出其中位数一定在 0 到 n 个数的极差以内，因此这里选择在这个范围内进行二分查找，每次更新中位数的值及左右边界，直到刚好有 n(n+1)/4 个数在它的左边即可。

由于没有算出 n*(n-1)/2 个数，这里先将 n 个数升序排序后，定义一个 Greater 函数返回 mid>=|x[i]-x[j]|,即 mid+x[i]>=x[j]（j>=i）的个数,当刚好等于 n(n+1)/4 时得出答案，若小于 n(n+1)/4，则说明 mid 小了，向右区间搜索，若大于，向左区间搜索直到得出答案。

代码段 1 POJ3579

```
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<climits>
typedef long long ll;
using namespace std;
const int maxn = 100002;
int n;
ll    m;
int x[maxn];
ll Greater(int mid) {//比 x[i]+mid 大的个数
	ll sum = 0;
	for (int i = 0; i < n; i++) {
		sum += n - (lower_bound(x, x+n,x[i] + mid) - x);//lower_bound 返回第一个比 x[i]+mid 大的元素地址
	}
	return sum;
}
int main()
{
		int mid;
		while(scanf("%d",&n)!=EOF)
		{
			for(int i = 0 ; i < n; i++)
			{
				scanf("%d", &x[i]);
			}
```

```
                m = n*(n-1)/4;//差值的个数的一半
                sort(x,x+n);//升序排序
                int l = 0, r = x[n - 1] - x[0];
                while(r > l + 1)
                {
                        mid = (l+r)/2;
                        if (Greater(mid)<=m)//在差值中比 mid 大的个数太小（mid
太大）
                        r = mid;
            else l = mid;//mid 太小
                }
                mid = (l+r)/2;
                cout<<mid<<endl;
        }
        return 0;
}
```

2.3 性能分析

sort 函数使用快速排序，时间复杂度为 O（nlogn）；lower_bound 函数也使用二分搜索进行，时间复杂度为 O（logn），在 Greater 函数中执行 n 次，则 Greater 函数的时间复杂度为 O（nlogn），而在 0 到 xmax - xmin 二分搜索的平均执行次数约为 log(xmax-xmin)(取决于输入的极差而与 n 无关)，所以程序总时间复杂度为 O（n(logn)*(log(xmax-xmin))=O（nlogn）。

2.4 运行测试

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
24414808	purien	3579	Accepted	848K	594MS	G++	757B	2023-12-10 18:54:41

图 2 POJ3579 题目运行测试截图

3.poj1185 炮兵阵地解题报告

3.1 题目分析

司令部的将军们打算在 N*M 的网格地图上部署他们的炮兵部队。一个 N*M 的地图由 N 行 M 列组成，（N <= 100；M <= 10。）地图的每一格可能是山地（用 "H" 表示），也可能是平原（用"P"表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

图 3 POJ1185 题目附图

如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

3.2 算法设计

可以用动态规划算法进行，对于每一行，具有无后效性，即这一行的部署情况只与本身以及前两行的情况有关。

看似每行搜索需要搜索 2^m 个可行解，实际上满足题目要求的可行解很少，ac 数组用于存储可以在 2^m 中情况中满足题目所给互不冲突条件的状态个数，这样计算出的 count 不超过 60 种。

状态压缩：这里将每一行摆放炮兵的状态压缩为一个整数，从最低位开始，不能放置则设置为 1，可以放置炮兵设置为 0，这样会产生一个二进制数，由于 M<=10,大小不会超过 2^10-1。关于状态的判断，用位运算即可解决。

状态转移方程：设 dp[i][j][k],为填充到第 i 行，前两行状态为 state_j，state_k，时可以放的炮兵个数，设该行放后状态为 state_i，则状态转移方程为：
dp[i+1][state_i][state_j] = max(dp[i+1][state_i][state_j],dp[i][j][k]);

之后对于每一行可以枚举该行以及前两行，并判断它们之间是否由冲突以及他们是否与初状态（地形）有冲突，都没有冲突则记录该行放入炮兵的数目并存入数组 dp。依次 DP 下去即可求解。

代码段 2 POJ1185

```
#include<iostream>
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<climits>
#include<cstring>
#include<cmath>
using namespace std;
#define max(a,b) ((a)<(b)?(b):(a))
const int nmax = 101;
const int mmax = 10;
int n,m;
int dp[nmax][1030][1030];//填充第 i 行，前两列对
int ori[105];//第 i 行的初始状态
```

```
int ac[nmax];
int Count(int x)
{
	int cons = 0;
	while(x)
	{
		if(x & 1)cons++;
		x >>= 1;
	}
	return cons;
}
int main()
{
		int ans = 0;
		char ch;
		scanf("%d%d",&n,&m);
		for(int i = 0; i < n; i++)
		{
			for(int j = 0; j < m; j++)
			{
				scanf("%c",&ch);
				if(ch!='H'&&ch!='P')
				{
					j--;
					continue;
				}
				if(ch=='H')
				ori[i+2] += (1<<j);//第 i 行的初始状态设置，把 j 处设置为
1,意为不能放置,前两行设置为 0
```

```
                }
        }
        int count = 0;
        for(int i = 0; i < 1<<m; i++)//一行内的可行解
        {
                if(i&(i<<1))//隔一个不能放
                continue;
                if(i&(i<<2))//隔两个不能放
                continue;
                ac[count++] = i;
        }
        for(int i = 0; i < n; i++)//第 i 行
        for(int aci = 0; aci < count; aci++)//枚举可行解
        {
                if(ac[aci]&ori[i+2])//和初始格局有没有冲突，有就去掉
                continue;
                for(int ac1 = 0; ac1 < count; ac1++)//检查下一行间有没有冲突
                {
                        if(ac[ac1]&ori[i+1])continue;
                        if(ac[aci]&ac[ac1])
                        continue;
                        for(int ac2 = 0; ac2 < count; ac2++)
                        {
                                if(ac[ac2]&ori[i]) continue;
                                if(ac[aci]&ac[ac2]) continue;
                                if(ac[ac1]&ac[ac2]) continue;
                                //都通过则为可行解
                                dp[i+1][ac[aci]][ac[ac1]] =
max(dp[i+1][ac[aci]][ac[ac1]],dp[i][ac[ac1]][ac[ac2]]);
```

```
                                }
                                dp[i+1][ac[aci]][ac[ac1]]+=Count(ac[aci]);
                        }
                }
                for(int i = 0; i    < count; i++)
        {
                for(int j = 0; j    < count; j++)
                {
                        ans = max(ans,dp[n][ac[i]][ac[j]]);
                        //取最大的方案数
                }
        }
        printf("%d\n", ans);
        return 0;
}
```

3.3 性能分析

计算可行解时遍历了一行的所有状态，时间复杂度为 O（2^m），枚举到的可行状态数为 count（由于 m<=10 时，count 不超过 60），之后枚举 n 行状态并记录可行状态，每次枚举三行，嵌套三层循环，时间复杂度为 O（n*count^3），故程序总时间复杂度为 O（2^m + n*count^3）。

3.4 运行测试

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
24459870	purien	1185	Accepted	30856K	297MS	G++	1695B	2024-01-08 00:29:45
24427872	purien	1185	Accepted	34672K	297MS	G++	1673B	2023-12-18 16:12:13

图 4 POJ1185 题目运行测试截图

4. POJ1700 Crossing River 解题报告

4.1 题目分析

每个人过河都有自己的过河时间。有 n 个人想过河，但只有一只小船，最多只能装 2 个人。每一次过河，过河时间为用时最多的那人过河时间，如果还有人没有过河，那么过去一个用时最少的送回船。问 n 人过河最少要多少时间。

4.2 算法设计

用贪心算法。当剩余人数大于 2 时，船每次来回可以运送一个人，当剩余人数小于等于 2 时可以一次过去。由于过河时间为两人中用时最多的人的过河时间，显然，应该让过河时间最长的人先过河。分析一下最优决策，会发现当剩余人数小于等于 3 时，当剩余人数大于 3 时只有以下两种决策可能导出最优解：

1) 由当前渡河时间最短的一人带上渡河时间最长的一人，再由渡河时间最短的一人把船开回，用时为 tn+t1；

2) 由先由当前渡河时间最小的两人过河，把渡河时间次小者留在另一边，第二次由过河最慢和次慢者过河并由过河次快者把船送回，用时为 t[2]+t[1]+t[n]+t[2]。

比较两种方案，第一种方案运送两人的时间为 2*t[1]+t[n-1]+t[n]，第二种方案运送两人的时间为 t[1]+2*t[2]+t[n]。可以看出，当 2*t[2] >= t[n-1]+t[1]时，选择方案 1，当 2*t[2] < t[n-1]+t[1]，选择方案 2。

有了贪心选择的策略，之后就可以遍历数组累加得出答案了。

代码段 3 POJ1700

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<cstdio>
#include<cmath>
```

```
using namespace std;
const int nmax = 1001;
int num[nmax];
int main()
{
	int t,n,ans;
	cin>>t;
	while(t--)
	{
		cin>>n;
		for(int i = 0; i < n ; i++)
		scanf("%d",&num[i]);
		sort(num,num+n);
		ans = 0;
		//找到分界线,前后方案不同
		int k=n;
		for(int i = n-1; i>1;i--)
		{

	if(num[0]+num[i]>2*num[1]&&num[0]+num[i-1]<=2*num[1])
				{
					k = i;
					break;
				}
		}
		k++;//k 指向两个中最大的
		if((n-1-k)%2)
		k++;
		for(int i = n-1;i>1;)
```

```
                {
                        if(i>=k)
                        {
                        ans+=num[i]+2*num[1]+num[0];
                        i-=2;
                        }
                        else if(i<k)
                        {
                                ans+=num[i]+num[0];
                                i--;
                        }
                }
                if(n==1)
                ans+=num[0];
                else
                ans+=num[1];
                printf("%d\n",ans);
        }
        return 0;
}
```

4.3 性能分析

找到每个人过河适用于方案 1 还是 2 需要 O（n）的时间复杂度，模拟过河进行累加也是 O（n）的时间复杂度，故程序总时间复杂度为 O（n）。

4.4 运行测试

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
24421063	purien	1700	Accepted	724K	0MS	G++	732B	2023-12-14 00:30:47

图 5 POJ1700 题目运行测试截图

5. POJ 1077 Eight 解题报告

5.1 题目分析

题目大意：一个 3*3 的棋盘上有 8 个格子写着 1~8 的数字，还有一个空格，给定局面，要求给出一个空格的移动序列（由 u,d,l,r 组成，分别代表空格进行的上下左右移动），使得达到目标状态：

1 2 3

4 5 6

7 8 x

x 代表空格。

这是一个经典的搜索问题，为了提高程序运行效率使用 A*算法求解。

5.2 算法设计

5.2.1 A*算法

使用 A*算法，这是一种启发式搜索，搜索的每一步都利用估价函数：f(n)=g(n)+h(n) 对 OPEN 表中的节点排序。

估价函数“f(n)=g(n)+h(n)” 可简单理解为：从初始节点 S0 到目标节点 Sg 所需耗费的总代价。

这个总代价可分成两个部分：

① 从初始节点 S0 到中间节点 n(搜索的中间状态)，已经消耗的实际代价 g(n)；（在这个问题为从初始状态到中间状态的操作步数）

② 对从中间节点 n 到目标节点 Sg 的代价（在这个问题为从中间状态到目标状态的操作步数），由于实际代价在求解出答案前很难求出，所以用预测代价---h(n)代替。

可以看出，估计值\预测值的准确性会直接影响求解效率及能否求得合理的解。

可以证明，在 g(n)相对固定情况下，若任意估计值均小于等于真实值，也就是 h(n)≤h*(n) ，即可认定最终解就是问题的最优解。

本人选择的启发函数为各数码到目标状态对应数码的曼哈顿距离。由于每次空格移动只会使得某一个数码板在水平或垂直方向上的位置改变 1，所以 h(n)≤h*(n)，即估计值始终小于等于真实值，且对于目标结点，估计值等于真实值（此时真实值为 0）

算法步骤及过程如下：

1) 首先，将初始状态$\boldsymbol{S_0}$，根据选择启发函数的不同算出新状态的 g(n)与 h(n)值，放入 OPENLIST。
2) 每次取出 OPENLIST 中估价函数 f(n)=g(n)+h(n)最小的，并对其执行适当的操作符，产生新状态$\boldsymbol{S_n}$，并根据选择启发函数的不同算出新状态的 g(n)与 h(n)值放入 OPENLIST，已经执行过操作的状态放入 CLOSEDLIST。
3) 继续，直到产生目标状态$\boldsymbol{S_g}$为止。

以下为 A*算法求解主要函数 solve（）：

代码段 4 POJ1077 solve 函数

```
void solve()
{
        struct node temp;
        char t;
        int g,now;
        int c;//contor
        int cc;
        int x,y;
        int dis;
```

```
while(!q.empty())
{
    temp = q.top();
    q.pop();
    visit[temp.state] = true;
    char state[9];
    incontor(state,temp.state);
    cc = temp.state;
    now = temp.now;
    for(int i = 0; i < 4; i++)//四个方向左，右
    {
        x=now%3+dir[i][0];
        y=now/3+dir[i][1];
        if(x<0||x>2||y<0||y>2)
        continue;//越界
        swap(state[now+d[i]],state[now]);//交换位置
        c = Contor(state);//记录康托值
        swap(state[now+d[i]],state[now]);//换回来
        if(visit[c])
        continue;
        dis =
temp.f-Mdis(x,y,state[now+d[i]]-1)+Mdis(now%3,now/3,state[now+d[i]]-1)-linear_conflict(c
c)+linear_conflict(c);//dis 更新
        g = temp.g;
        if(state[now+d[i]]==now+d[i]+1)
        {
            g++;
        }else if(state[now+d[i]]==now+1)
        {
```

```
                                g--;
                        }
                        path[c].op = i;
                        path[c].k = cc;//上一个
                        if(c==tar)
                        {
                                int j = 0;
                                for(int i = c;i;i = path[i].k)
                                {
                                        p[j++] = di[path[i].op];
                                }
                                for(int i = j-2; i >= 0; i--)
                                {
                                        printf("%c", p[i]);
                                }
                                return ;
                        }
                        q.push({temp.dis+1,g,dis,now+d[i],c});
                }
        }
        printf("error");
        return ;
}
```

5.2.2 康托展开

康托展开是一个全排列到一个自然数的双射，常用于构建哈希表时的空间压缩。 康托展开的实质是计算当前排列在所有由小到大全排列中的顺序，因此是可逆的。

康托展开可以求解一个排列的序号，比如：12345 序号为 1 ，12354 序号为 2，按字典序增加编号递增，依次类推。

康托逆展开可以求解一个序号它对应的排列是什么。

以下为康托展开的公式：

$$X = a_n(n-1)! + a_{n-1}(n-2)! + \cdots + a_1 \cdot 0!$$

图 6 康托展开公式

将空格视为 0，数码板的状态可以看作 0 到 8 一共九个数字的排列，可以将该排列用康托展开转化为一个自然数（最大为 362880）存储，需要时再用逆康托展开转化为排列，这样可以有效降低存储空间，实现压缩存储。

以下为 contor 函数和逆康托函数 incontor（）的代码：

代码段 5 POJ1077 contor 与 incontor 函数

```
int fac[10]={1,1,2,6,24,120,720,5040,40320,362880};//康托展开用到的阶乘
int Contor(char *a) {//计算康托值
   int sum=0,small=0;
   int n = 9;
   for(int i=0;i<n;i++) {
      small=0;
      for(int j=i+1;j<n;j++) {
         if(a[j]<a[i]) {       //计算第 i 位右边比该数还要小的数的个数
            small++;
         }
      }
      sum+=small*fac[n-i-1];
      //例如 12345，i=0 时实际上对应着 5
      //顺数第 i 位实际上对应着第 n-i 位 ，即 a[n-i]=small,所以应该是 small* ((n-i)-1)!
```

```
        //所以应该是 small*fac[n-i-1]
    }
    return sum+1;
}
void incontor(char*res,int x) {
        x--;// 得到以 0 开始的排名
        int n = 9;// 保存数列答案
        int cnt;
        bool st[10];// 标记数组
        memset(st,0,sizeof st);
        for(int i = 0;i < n; ++i) {
            cnt = x/fac[n - i - 1];// 比 a[i]小且没有出现过的数的个数
            x %= fac[n - i - 1];// 更新 x
            for(int j = 1;j <= n; ++j) {// 找到 a[i]，从 1 开始向后找
                if(st[j]) continue;// 如果被标记过，就跳过
                if(!cnt) {// 如果 cnt == 0 说明当前数是 a[i]
                    st[j] = 1;//标记
                    res[i] = j-1;// 第 i 位是 j
                    break;
                }
                cnt--;// 如果当前不是 0，就继续往后找
            }
        }
        return;// 返回答案
}
```

5.2.3 可解性检查

对于八数码问题，可以按照行优先将棋盘改成一个 9 个数的数组，关于八数码问题是否可解的判断，可以计算出数码板 1 到 8（忽略空格 0）的逆序对数，若逆序对是偶数则可解，为奇数则不能解。

以下为 check 函数代码（检查是否无解，无解输出 true，否则输出 false）：

代码段 6 POJ1077 check 函数

```
bool check(char*s,int now)//检查是否有答案
{
	int cnt = 0;
	for(int i = 0; i < 9; i++)
	{
		if(!s[i])
		continue;
		for(int j = i+1; j < 9; j++)
		{
			if(!s[j])
			continue;
			if(s[i]>s[j])
			cnt++;
		}
	 }
	if(cnt%2)
	return false;
	return true;
}
```

5.2.4 程序主体

以下为主体程序，使用一个 9 元组描述棋盘状态，main 函数中进行读入，初始化，调用先前介绍的 solve、check、contor 和 incontor 函数。用 path 数组来存储求解路径，搜索到目标状态后输出路径。

代码段 7 POJ1077 程序主体

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<queue>
#include<cstdio>
using namespace std;
int d[4] = {-1,1,-3,3};
char di[4] = {'l','r','u','d'};
int dir[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
const int tar = 46234;//目标康托值
struct node
{
	int dis;//步骤数
	int g;//不在正确位置的数量
	int f;//启发函数值
	int now;//空格当前所处的位置
	int state;//用康托值表示当前状态
	bool operator<(const struct node b)const
	{
		return dis+f>b.dis+b.f;
	}
};
```

```
struct path
{
	int op;//存动作
	int k;//下一个编号，编号为 0 则为终点
}path[362881];
priority_queue<struct node> q;
void solve();
void swap(int &a,int &b)
{
	int t;
	t = a;
	a = b;
	b = t;
}
bool visit[362881];
char p[362881];
int Mdis(int x,int y,int i)
{
	return abs(x-i%3)+abs(y-i/3);
}
int main()
{
	char ch;
	char ori[9];//用一个 9 元组表示棋盘状态
	int count = 0;
	int now;
	for(int i = 0; i < 9;)
	{
```

```
            scanf("%c",&ch);
            if(ch<'9'&&ch>'0')
            {
                ori[i++] = ch-'0';
                if(ori[i-1]!=i)
                count++;
            }
            else if(ch=='x')
            {
            ori[i++] = 0; //0 表示空格
            now = i-1;
            }
    }
    if(!check(ori,now))
    {
            printf("unsolvable");
            return 0;
    }
    int dis = 0;
    for(int i = 0; i < 9;i++)
    {
            if(ori[i])
            dis += Mdis(i%3,i/3,ori[i]-1);
    }//算出最初的曼哈顿距离
    dis+=linear_conflict(Contor(ori));
    q.push({0,count,dis,now,Contor(ori)});
    solve();
    return 0;
```

```
}
```

5.3 性能分析

程序中使用优先队列 q（小顶堆），建堆用时 O（n）,每次入队出队时间复杂度为 O（logn）。

启发函数的更新上，曼哈顿距离可以优化为只在初始时计算曼哈顿整个图 n^2-1 个结点的曼哈顿距离之和，之后每次只计算改变位置的板的曼哈顿距离，从而将 O（n^2）的时间复杂度优化到了 O（1）。

contor 和逆 contor 函数中双层循环，都需要 O（n^2）的时间复杂度。

A*算法的时间复杂度取决于估价函数和搜索空间的大小。在最坏情况下，A*算法需要遍历整个搜索空间，因此时间复杂度为指数级别。但是在实际应用中，A*算法通常可以在较短的时间内找到最优解，估价函数的优劣影响了算法的搜索效率。

5.4 运行测试

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
24440247	purien	1077	Accepted	1944K	32MS	G++	4257B	2023-12-28 19:35:18

图 7 POJ1077 题目运行测试截图

6. 总结

6.1 实验总结

1) stl 容器与标准库函数的运用。在 POJ2503 Babelfish 中，我学习并运用了 map，在 POJ 1077 Eight 中学习并使用了 priority_queue，还使用了 lower_bound(),sort()等函数，调用这些使得我可以花更多时间在思考算法上而不是苦恼于部分算法的实现，使得我做算法题事半功倍。当然 stl 速度不如自己实现的快，所以在使用时也要慎重。

2) 图算法的模板。在攻克图相关的算法问题时，我在自己编程却出现 bug 时上网求助，却发现了对于一些知名的图算法做题时存在一些“模板”，虽然最后我解决问题靠的是自己修 bug 后的代码，但是在熟练算法的基础上使用模板也能帮助我们顺利的解题。

3) 读入方法的快读。在具有大量数据输入输出中，快读可以有效减少读入所需时间（相比直接 scanf 甚至是 cin）。

6.2 心得体会和建议

算法实践课作为算法设计与分析理论课的补充，两者相辅相成，共同促进了我算法能力的提高。课上学到的算法是我做算法题必要的理论支撑，而做这些算法题过程又加深了我对课上学到各种算法的认识，提高了我编写代码的能力。在课上觉得搞清楚的算法，在代码实现阶段又出现了很多问题，而且如何恰当地运用算法解决相似但不太一样的问题也是一个难点。不断地攻克这些难题，让我能力提升的同时也让我充满了成就感。

对于课程的建议方面，我感觉在做算法题的各个模块时明显感觉到了整体难度的差异，比如动态规划部分很多题目我尝试套用课上讲到的找最优子结构和状态转移方程，却发现我很难将问题剖析清楚，找网上的解答依然不太理解，而在完成贪心算法的内容时却每道题都完全由自己解决，基本没有遇到什么困难。不能否认算法本身难度就有差异，动态规划难度一般较大，但我认为如果在此时老

师能多加引导，比如给出一个推荐的做题顺序，给出一些相似题目练手，应该会更好地帮助算法基础较薄弱的同学入门动态规划。