目录

[KM 1](#_Toc419150117)

[Hopcroft 2](#_Toc419150118)

[一般图最大匹配\_片段 2](#_Toc419150119)

[费用流 3](#_Toc419150120)

[无向图最小割 4](#_Toc419150121)

[有向图最小生成树 5](#_Toc419150122)

[最大团搜索算法 5](#_Toc419150123)

[极大团的计数 6](#_Toc419150124)

[弦图的完美消除序列 6](#_Toc419150125)

[Manacher 7](#_Toc419150126)

[ExtKMP 7](#_Toc419150127)

[SA 7](#_Toc419150128)

[字符串的最小表示 8](#_Toc419150129)

[后缀自动机 8](#_Toc419150130)

[DLX 8](#_Toc419150131)

[综合 10](#_Toc419150132)

set nocompatible sta hls wrap ruler cindent nu nobackup noswapfile autoindent ts=4 noet sts=4 sw=4

syntax on

autocmd FileType c,cpp nmap <F8> <ESC>:w <CR><ESC>:!g++ % -o %< <CR>

autocmd FileType c,cpp nmap <F9> :!time ./%< <./%<.in <CR>

autocmd FileType c,cpp nmap <F10> :!time ./%< <CR>

nmap <F2> :vs %<.in <CR>

## KM

**const** **int** maxn=**200**;**const** **int** oo=**0x7fffffff**;

**int** w[maxn][maxn],x[maxn],y[maxn],px[maxn],py[maxn],sy[maxn],slack[maxn];

**int** par[maxn];**int** n;**int** pa[**200**][**2**],pb[**200**][**2**],n0,m0,na,nb;**char** s[**200**][**200**];

**void** adjust(**int** v){ sy[v]=py[v]; **if** (px[sy[v]]!=-**2**) adjust(px[sy[v]]);}

**bool** find(**int** v){**for** (**int** i=**0**;i<n;i++)

**if** (py[i]==-**1**){

**if** (slack[i]>x[v]+y[i]-w[v][i]){

slack[i]=x[v]+y[i]-w[v][i]; par[i]=v;}

**if** (x[v]+y[i]==w[v][i]){

py[i]=v; **if** (sy[i]==-**1**){adjust(i); **return** **1**;}

**if** (px[sy[i]]!=-**1**) **continue**; px[sy[i]]=i;

**if** (find(sy[i])) **return** **1**;

}}**return** **0**;}

**int** km(){**int** i,j,m;

**for** (i=**0**;i<n;i++) sy[i]=-**1**,y[i]=**0**;

**for** (i=**0**;i<n;i++) {x[i]=**0**; **for** (j=**0**;j<n;j++) x[i]=max(x[i],w[i][j]);}

**bool** flag;

**for** (i=**0**;i<n;i++){

**for** (j=**0**;j<n;j++) px[j]=py[j]=-**1**,slack[j]=oo;

px[i]=-**2**; **if** (find(i)) **continue**; flag=**false**;

**for** (;!flag;){

m=oo; **for** (j=**0**;j<n;j++) **if** (py[j]==-**1**) m=min(m,slack[j]);

**for** (j=**0**;j<n;j++){

**if** (px[j]!=-**1**) x[j]-=m;

**if** (py[j]!=-**1**) y[j]+=m;

**else** slack[j]-=m;}

**for** (j=**0**;j<n;j++){

**if** (py[j]==-**1**&&!slack[j]){

py[j]=par[j];

**if** (sy[j]==-**1**){ adjust(j); flag=**true**; **break**;}

px[sy[j]]=j; **if** (find(sy[j])){flag=**true**;**break**;}

}}}}

**int** ans=**0**; **for** (i=**0**;i<n;i++) ans+=w[sy[i]][i];**return** ans;}

## Hopcroft

**int** n, m, match = 0; queue<**int**> Q;

**int** mx[Maxn], my[Maxn], dx[Maxn], dy[Maxn], dis, visit[Maxn];

**int** ux[Maxn], uy[Maxn], px[Maxn], py[Maxn], pv[Maxn];

**void** adde(**int** u, **int** v) {}

**bool** searchPath() {

**int** i, j; dis = MOD; **for**(i = 0; i < n; i++) dx[i] = -1;

**for**(j = n; j < n + m; j++) dy[j] = -1; **while**(!Q.empty()) Q.pop();

**for**(**int** i = 0; i < n; i++) **if**(-1 == mx[i]) Q.push(i);

**int** u, v;

**while**(!Q.empty()) {

u = Q.front(); Q.pop(); **if**(dx[u] > dis) **break**;

**for**(**int** j = last[u]; j != -1; j = e[j].next) {

v = e[j].v; **if**(-1 != dy[v]) **continue**; dy[v] = dx[u] + 1;

**if**(-1 == my[v]) dis = dy[v];

**else** {dx[my[v]] = dy[v] + 1; Q.push(my[v]);} } }

return dis != MOD; }

**bool** dfs(**int** u){ **int** v;

**for**(**int** j = last[u]; j != -1; j = e[j].next) {

v = e[j].v; **if**(visit[v] || dx[u] + 1 != dy[v]) **continue**;

**if**(dy[v] == dis && my[v] != -1) **continue**; visit[v] = **true**;

**if**(-1 == my[v] || dfs(my[v])){my[v] = u; mx[u] = v;return **true**;} } return **false**; }

**int** solve(){**int** i, j; match = 0;

**for**(i = 0; i < n; i++) mx[i] = -1;

**for**(j = n; j < n + m; j++) my[j] = -1;

**while**(searchPath()){

**for**(j = n; j < n + m; j++) visit[j] = **false**;

**for**(**int** i = 0; i < n; i++)**if**(-1 == mx[i] && dfs(i)) match++;}

return match; }

## 一般图最大匹配

**int** n, head, tail, Start, Finish, Q[Maxn], mark[Maxn], InBlossom[Maxn], inqueue[Maxn];

**int** match[Maxn];//表示哪个点匹配了哪个点 **int** father[Maxn]; //这个是增广路径的father

**int** base[Maxn];//该点属于哪朵花 **bool** mp[Maxn][Maxn]; //邻接关系

**void** BlossomContract(**int** x, **int** y) {

memset(mark, **false**, sizeof(mark));

memset(InBlossom, **false**, sizeof(InBlossom));

#define pre father[match[i]]

**int** lca, i;

**for**( i = x; i; i = pre) {i = base[i]; mark[i] = **true**; }

**for**(i = y; i; i = pre) {i = base[i]; //寻找lca

**if**(mark[i]) {lca = i; **break**; } }

**for**(i = x; base[i] != lca; i = pre) {

**if**(base[pre] != lca) father[pre] = match[i];

//对于BFS树中的父边是匹配边的点, father向后跳

InBlossom[base[i]] = **true**; InBlossom[base[match[i]]] = **true**; }

**for**(i = y; base[i] != lca; i = pre) {

**if**(base[pre] != lca) father[pre] = match[i]; // 同理

InBlossom[base[i]] = **true**; InBlossom[base[match[i]]] = **true**; }

#undef pre

**if**(base[x] != lca) father[x] = y; **if**(base[y] != lca) father[y] = x;

**for**(i = 1; i <= n; i++) **if**(InBlossom[base[i]]) { base[i] = lca;

**if**(!inqueue[i]) {Q[tail++] = i; inqueue[i] = **true**; } } }

**void** Change() { **int** x, y, z = Finish;

**while**(z) { y = father[z]; x = match[y];

match[y] = z; match[z] = y; z = x; } }

**void** FindAugmentPath() { **int** i; memset(father, 0, sizeof(father));

memset(inqueue, **false**, sizeof(inqueue));

**for**(i = 1; i <= n; i++) base[i] = i;

head = tail = 0; Q[tail++] = Start; inqueue[Start] = 1;

**while**(head < tail) { **int** x = Q[head++];

**for**(**int** y = 1; y <= n; y++)

**if**(mp[x][y] && base[x] != base[y] && match[x] != y) {

**if**(Start == y || match[y] && father[match[y]])

BlossomContract(x, y);

**else** **if**(!father[y]) { father[y] = x;

**if**(match[y]) { Q[tail++] = match[y];

inqueue[match[y]] = **true**; }

**else** {Finish = y; Change(); return; }

} } }

}

**void** Edmonds() { memset(match, 0, sizeof(match));

**for**(Start = 1; Start <= n; Start++)

**if**(match[Start] == 0) FindAugmentPath(); }

**void** output() { memset(mark, **false**, sizeof(mark));

**int** i, cnt = 0;

**for**(i = 1; i <= n; i++) **if**(match[i]) cnt++;

/\* pr**int**f("%d\n", cnt); //输出匹配关系

**for**(**int** i = 1; i <= n; i++) {

**if**(!mark[i] && match[i]) {

mark[i] = **true**; mark[match[i]] = **true**;

pr**int**f("%d %d\n", i, match[i]); }

} //\*/

**if**(cnt < n)pr**int**f("NO\n");**else** pr**int**f("YES\n"); }

## 费用流

**typedef** **int** ValueType;

**const** ValueTyep MOD = 0x3f3f3f3f3f3f3f3fLL;

ValueType flow, cost, value, dist[Maxn];

**int** visit[Maxn], src, des;

deque<**int**> Q;

**void** adde(**int** u, **int** v, ValueType c, ValueType w) {}

ValueType Aug(**int** u, ValueType m) {

**if**(u == des) {

cost += value \* m; flow += m;

return m; }

visit[u] = **true**; **int** j, v;

ValueType l = m, c, w, del;

**for**(j = last[u]; j != -1; j = e[j].next) {

v = e[j].v; c = e[j].c; w = e[j].w;

**if**(c && !w && !visit[v]) {

del = Aug(v, l < c ? l : c);

e[j].c -= del; e[j ^ 1].c += del; l -= del;

**if**(!l) return m; } }

return m - l; }

**bool** Modlabel(**int** src, **int** des, **int** n) {

**int** i, j, u, v; ValueType c, w, del;

memset(dist, 0x3f, sizeof(dist[0])\*(n + 3));

**while**(!Q.empty()) Q.pop\_back();

dist[src] = 0; Q.push\_back(src);

**while**(!Q.empty()) {

u = Q.front(); Q.pop\_front();

**for**(j = last[u]; j != -1; j = e[j].next) {

v = e[j].v; c = e[j].c; w = e[j].w;

**if**(c && (del = dist[u] + w) < dist[v]) {

dist[v] = del;

**if**(Q.empty() || del <= dist[Q.front()]) Q.push\_front(v);

**else** Q.push\_back(v); } } }

**for**(i = 0; i < n; i++) {

**for**(j = last[i]; j != -1; j = e[j].next)

e[j].w -= dist[e[j].v] - dist[i];

}

value += dist[des];

return dist[des] < MOD; }

**void** zkw(**int** src, **int** des, **int** n) {

value = cost = flow = 0;

**while**(Modlabel(src, des, n)){

do {

memset(visit, 0, sizeof(visit[0]) \* (n + 3));

}**while**(Aug(src, MOD)); } }

## 无向图最小割

//K连通块计数, 注意节点下标0~n-1

**typedef** **int** ValueType;

ValueType edge[Maxn][Maxn], g[Maxn][Maxn], minCut, maxi;

**int** n, m, k, S, T, top, sta[Maxn], comb[Maxn], node[Maxn];

vector<**int**> parta, partb, belong[Maxn];

ValueType Search (**int** n) {

**int** i, j, u, vis[Maxn]; ValueType wet[Maxn], minCut = 0, maxi;

**int** temp = -1, top = 0; S = -1, T = -1;

memset(vis, 0, sizeof(vis)); memset(wet, 0, sizeof(wet));

**for** (i=0; i< n; i++) { maxi = -MOD;

**for** (j = 0; j < n; j++) { u = node[j];

**if** (!comb[u] && !vis[u] && wet[u] > maxi) {

temp = u; maxi = wet[u]; } }

sta[top++] = temp; vis[temp] = **true**;

**if** (i == n - 1) minCut = maxi;

**for** (j = 0; j < n; j++) { u = node[j];

**if** (!comb[u] && !vis[u]) wet[u] += edge[temp][u]; } }

S = sta[top - 2]; T = sta[top - 1];

**for** (i = 0; i < top; i++) node[i] = sta[i]; return minCut; }

ValueType StoerWagner (vector<**int**> & li) {

**int** i, j, k, cur, n = li.SZ, u, v, used[Maxn];

ValueType ans = MOD; memset(comb, 0, sizeof(comb));

**for** (i = 0; i < n; i++){node[i] = i; belong[i].clear();belong[i].PB(i);}

**for** (i = 1; i < n; i++) {k = n - i + 1; cur = Search(k);

**if** (cur < ans) { ans = cur; **for**(j = 0; j < n; j++) used[j] = 0;

**for**(j = 0; j < belong[T].SZ; j++) used[belong[T][j]] = 1; }

**for**(j = 0; j < belong[T].SZ; j++) belong[S].PB(belong[T][j]);

**if** (ans == 0) **break**; comb[T] = **true**;

**for** (j = 0; j < n; j++) { **if** (j == S) **continue**;

**if** (!comb[j]) {edge[S][j] += edge[T][j];

edge[j][S] += edge[j][T]; } } }

parta.clear(); partb.clear();

**for**(j = 0; j < n; j++) {**if**(used[j]) parta.PB(li[j]);

**else** partb.PB(li[j]); } return ans; }

**int** dfs(vector<**int**> &li) {**int** n = li.SZ, i, j;

**for**(i = 0; i < n; i++) **for**(j = 0; j < n; j++)

edge[i][j] = g[li[i]][li[j]];

**int** cur = StoerWagner(li); **if**(cur >= k) return 1;

vector<**int**> a(parta), b(partb); return dfs(a) + dfs(b); }

## 有向图最小生成树

/\* O(VE),根不固定，添加一个根节点与所有点连无穷大的边！

\* 如果求出比2\*MOD大, 则不连通; 根和虚拟根相连的结点

\* 根据pre的信息能构造出这棵树！

\* 注意结点必须从0~n-1\*/

**typedef** **int** mytype; mytype inv[Maxn];

**int** visit[Maxn], pre[Maxn], belong[Maxn], ROOT;

mytype dirtree(**int** n, **int** m, **int** root) {

mytype sum = 0; **int** i, j, k, u, v;

**while** (1) {

**for** (i = 0; i < n; i++) {

inv[i] = MOD; pre[i] = -1; belong[i] = -1; visit[i] = -1; }

inv[root] = 0;

**for** (i = 0; i < m; i++) {//除原点外，找每个点的最小入边

u = e[i].u; v = e[i].v;

**if** (u != v) {

**if** (e[i].w < inv[v]) {

inv[v] = e[i].w; pre[v] = u;

**if**(u == root) ROOT = i; //记录根所在的边,输出根时利用ROOT-m计算是原图哪个结点

} } }

**for** (i = 0; i < n; i++) **if** (inv[i] == MOD) return -1;

**int** num = 0;

**for** (i = 0; i < n; i++) { //找圈，收缩圈

**if** (visit[i] == -1) {

j = i;

**for**(j = i; j != -1 && visit[j] == -1 && j != root; j = pre[j]) visit[j] = i;

**if** (j != -1 && visit[j] == i) {

**for** (k = pre[j]; k != j; k = pre[k]) belong[k] = num;

belong[j] = num ++ ; } }

sum += inv[i]; }

**if** (num == 0) return sum;

**for** (i = 0; i < n; i++)

**if** (belong[i] == -1) belong[i] = num ++ ;

**for** (i = 0; i < m; i++) { //重新构图

e[i].w = e[i].w - inv[e[i].v]; e[i].v = belong[e[i].v];

e[i].u = belong[e[i].u]; }

n = num; root = belong[root]; } }

## 最大团搜索算法

**Int** g[][]为图的邻接矩阵。 MC(V)表示点集V的最大团

令Si={vi, vi+**1**, ..., vn}, mc[i]表示MC(Si). 倒着算mc[i]，那么显然MC(V)=mc[**1**]

此外有mc[i]=mc[i+**1**] **or** mc[i]=mc[i+**1**]+**1**

**void** init(){

**int** i, j;**for** (i=**1**; i<=n; ++i) **for** (j=**1**; j<=n; ++j) scanf(**"%d"**, &g[i][j]);

}

**void** dfs(**int** size){

**int** i, j, k;

**if** (len[size]==**0**) { **if** (size>ans) { ans=size; found=**true**;} **return**;}

**for** (k=**0**; k<len[size] && !found; ++k) {

**if** (size+len[size]-k<=ans) **break**;

i=list[size][k]; **if** (size+mc[i]<=ans) **break**;

**for** (j=k+**1**, len[size+**1**]=**0**; j<len[size]; ++j)

**if** (g[i][list[size][j]]) list[size+**1**][len[size+**1**]++]=list[size][j];

dfs(size+**1**);}}

**void** work(){

**int** i, j; mc[n]=ans=**1**;

**for** (i=n-**1**; i; --i) { found=**false**; len[**1**]=**0**;

**for** (j=i+**1**; j<=n; ++j) **if** (g[i][j]) list[**1**][len[**1**]++]=j;

dfs(**1**); mc[i]=ans;}}

## 极大团的计数

**Bool** g[][] 为图的邻接矩阵，图点的标号由1至n。

**void** dfs(**int** size){

**int** i, j, k, t, cnt, best = **0**; **bool** bb;

**if** (ne[size]==ce[size]){**if** (ce[size]==**0**) ++ans;**return**;}

**for** (t=**0**, i=**1**; i<=ne[size]; ++i) {

**for** (cnt=**0**, j=ne[size]+**1**; j<=ce[size]; ++j)

**if** (!g[list[size][i]][list[size][j]]) ++cnt;

**if** (t==**0** || cnt<best) t=i, best=cnt;

}

**if** (t && best<=**0**) **return**;

**for** (k=ne[size]+**1**; k<=ce[size]; ++k) {

**if** (t>**0**){

**for** (i=k; i<=ce[size]; ++i)

**if** (!g[list[size][t]][list[size][i]]) **break**;

swap(list[size][k], list[size][i]);

}

i=list[size][k]; ne[size+**1**]=ce[size+**1**]=**0**;

**for** (j=**1**; j<k; ++j)**if** (g[i][list[size][j]])

list[size+**1**][++ne[size+**1**]]=list[size][j];

**for** (ce[size+**1**]=ne[size+**1**], j=k+**1**; j<=ce[size]; ++j)

**if** (g[i][list[size][j]]) list[size+**1**][++ce[size+**1**]]=list[size][j];

dfs(size+**1**); ++ne[size]; --best;

**for** (j=k+**1**, cnt=**0**; j<=ce[size]; ++j) **if** (!g[i][list[size][j]]) ++cnt;

**if** (t==**0** || cnt<best) t=k, best=cnt;

**if** (t && best<=**0**) **break**;

}}

**void** work(){

**int** i; ne[**0**]=**0**; ce[**0**]=**0**; **for** (i=**1**; i<=n; ++i) list[**0**][++ce[**0**]]=i;

ans=**0**; dfs(**0**);}

## 弦图的完美消除序列

最大势算法:简单的弦图判定,先求完美消除序列L,再利用L判断是否弦图

**int** adj[Maxn][Maxn], n, m, L[Maxn], cnt[Maxn], visit[Maxn], mpL[Maxn];

priority\_queue<PII> que;

//利用MSC最大势算法求完美消除序列L,无合法序列返回**false**

**int** getList() { **int** i, j, k, u, v, w;

**for**(i = 1; i <= n; i++) cnt[i] = 0, visit[i] = 0;

**while**(!que.empty()) que.pop(); que.push(MP(0, 1)); k = n;

**while**(!que.empty()) {u = que.top().BB; w = que.top().AA; que.pop();

**if**(w != cnt[u]) **continue**; visit[u] = 1; mpL[u] = k; L[k--] = u;

**for**(v = 1; v <= n; v++) **if**(!visit[v] && adj[u][v]) {

cnt[v]++; que.push(MP(cnt[v], v)); } }

**if**(k < 1) return **true**; **else** return **false**; }

//利用完美消除序列判断是否弦图

**int** check() {**int** i, j, k, u, v, w;

**for**(i = n - 1; i >= 1; i--) {u = L[i]; k = -1;

**for**(j = i + 1; j <= n; j++) { v = L[j];

**if**(adj[u][v]) {k = v; **break**; } }

**if**(k != -1) **for**( j++; j <= n; j++) {

v = L[j]; **if**(adj[u][v] && !adj[k][v]) return **false**; }

} return **true**; }

1.团数 ≤ 色数

2.最大独立集数 ≤ 最小团覆盖数

3.任何一个弦图都至少有一个单纯点，不是完全图的弦图至少有两个不相邻的单纯点。

4.设第i个点在弦图的完美消除序列第p(i)个。令N(v) = {w | w与v相邻且p(w) > p(v)}弦图的极大团一定是v∪N(v)的形式。

5.弦图最多有n个极大团。

6.设next(v) 表示N(v)中最前的点。令w\*表示所有满足A∈B的w中最后的一个点。判断v∪N(v)是否为极大团,只需判断是否存在一个w，满足Next(w) = v且|N(v)| + 1 ≤ |N(w)|即可。

7.最小染色：完美消除序列从后往前依次给每个点染色，给每个点染上可以染的最小的颜色。//团数=色数

8.最大独立集：完美消除序列从前往后能选就选。

9.最小团覆盖：设最大独立集为{p1 , p2 , …, pt}，则{p1∪N(p1), …, pt∪N(pt)}为最小团覆盖。 //最大独立集数 = 最小团覆盖数!!!

## Manacher

//s为原串, str为插入$和#的串, 读入s后, 调用init(s, str, len),

//最后调用Manacher(str,p,len), 求解遍历p数组求最大值, 注意输出ans-1

最长回文子串对应原串T中的位置:l = (i - p[i])/2; r = (i + p[i])/2 - 2;

**int** len, p[Maxn]; char s[Maxn], str[Maxn];

**void** init(char s[], char str[], **int**& len) {

**int** i, j, k; str[0] = '$'; str[1] = '#';

**for** (i = 0; i < len; i++) { str[i \* 2 + 2] = s[i];

str[i \* 2 + 3] = '#'; }

len = len \* 2 + 2; s[len] = 0; }

**void** Manacher (char str[], **int** p[], **int** len) {

**int** i, mx = 0, id; **for** (i = len; i < Maxn; i++) str[i] = 0;

**for** (i = 1; i < len; i++) {

**if** ( mx > i ) p[i] = min ( p[2 \* id - i], p[id] + id - i );

**else** p[i] = 1;

**for** (; str[i + p[i]] == str[i - p[i]]; p[i]++);

**if** ( p[i] + i > mx ) {mx = p[i] + i; id = i;} } }

## ExtKMP

char S[Maxn], T[Maxn]; **int** next[Maxn], B[Maxn];

**void** preExKmp(char T[], **int** LT, **int** next[]) {

**int** i, ind = 0, k = 1; next[0] = LT;

**while**(ind + 1 < LT && T[ind + 1] == T[ind]) ind++;

next[1] = ind;

**for**(i = 2; i < LT; i++) {

**if**(i <= k + next[k] - 1 && next[i - k] + i < k + next[k])

next[i] = next[i - k];

**else** { ind = max(0, k + next[k] - i);

**while**(ind + i < LT && T[ind + i] == T[ind]) ind++;

next[i] = ind; k = i; }

} }

**void** exKmp(char S[], **int** LS, char T[], **int** LT, **int** next[], **int** B[]) {

**int** i, ind = 0, k = 0; preExKmp(T, LT, next);

**while**(ind < LS && ind < LT && T[ind] == S[ind]) ind++;

B[0] = ind;

**for**(i = 1; i < LS; i++) {

**int** p = k + B[k] - 1, L = next[i - k];

**if**((i - 1) + L < p) B[i] = L;

**else** { ind = max(0, p - i + 1);

**while**(ind + i < LS && ind < LT && S[ind + i] == T[ind]) ind++;

B[i] = ind; k = i; }

} }

## SA

//论文模板, 使用时注意num[]有效位为0~n-1, 但是需要将num[n]=0, 否则RE;另外, 对于模板的处理将空串也处理了,作为rank最小的串, 因此有效串为0~n共, n-1个, 在调用da()函数时, 需要调用da(num, n + 1, m); 对于sa[], rank[]和height[]数组都将空串考虑在内, 作为rank最小的后缀!

//调用da(num, len+1, m);//m为字符个数略大

**int** len, num[Maxn], sa[Maxn], rank[Maxn], height[Maxn]; //num待处理的串

**int** wa[Maxn], wb[Maxn], wv[Maxn], wd[Maxn];

//sa[1~n]value(0~n-1); rank[0..n-1]value(1..n); height[2..n]

**int** cmp(**int** \*r, **int** a, **int** b, **int** x) {

return r[a] == r[b] && r[a + x] == r[b + x];}

**void** da(**int** \*r, **int** n, **int** m) {//倍增 r为待匹配数组 n为总长度+1 m为字符范围

**int** i, j, k, p, \*x = wa, \*y = wb, \*t;

**for**(i = 0; i < m; i++) wd[i] = 0;

**for**(i = 0; i < n; i++) wd[x[i] = r[i]]++;

**for**(i = 1; i < m; i++) wd[i] += wd[i - 1];

**for**(i = n - 1; i >= 0; i--) sa[--wd[x[i]]] = i;

**for**(j = 1, p = 1; p < n; j <<= 1, m = p) {

**for**(p = 0, i = n - j; i < n; i++) y[p++] = i;

**for**(i = 0; i < n; i++) **if**(sa[i] >= j) y[p++] = sa[i] - j;

**for**(i = 0; i < n; i++) wv[i] = x[y[i]];

**for**(i = 0; i < m; i++) wd[i] = 0;

**for**(i = 0; i < n; i++) wd[wv[i]]++;

**for**(i = 1; i < m; i++) wd[i] += wd[i - 1];

**for**(i = n - 1; i >= 0; i--) sa[--wd[wv[i]]] = y[i];

**for**(t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i < n; i++)

x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++; }

**for**(i = 0, k = 0; i < n; i++) rank[sa[i]] = i;

**for**(i = 0; i < n - 1; height[rank[i++]] = k)

**for**(k ? k-- : 0, j = sa[rank[i] - 1]; r[i + k] == r[j + k]; k++); }

## 字符串的最小表示

**int** MinRep (char S[], **int** L) {**int** i = 0, j = 1, k = 0, t;

**while** (i < L && j < L && k < L) { //找不到比它还小的或者完全匹配

t = S[ (i + k) % L] - S[ (j + k) % L];

//t = s[(i + k) >= L ? i + k - L : i + k] - s[(j + k) >= L ? j + k - L : j + k];

**if** (t == 0) k++;//相等的话,检测长度加1

**else** {//大于的话,s[i]为首的肯定不是最小表示,最大表示就改<

**if** (t > 0) i += k + 1; **else** j += k + 1; **if** (i == j) j++; k = 0;}

} return min (i, j);}

## 后缀自动机

**struct** State **{**

**static** vector **<**State**\*>** states**;**

**int** id**,** length**;**

State **\***parent**;**

State**\*** go**[**C**];**

State**(int** length**)** **:** id**((int)**states**.**size**()),** length**(**length**),** parent**(NULL)** **{**

memset**(**go**,** **NULL,** **sizeof(**go**));**

states**.**push\_back**(this);**

**}**

State**\*** extend**(**State**\*** start**,** **int** token**)** **{**

State **\***p **=** **this;**

State **\***np **=** **new** State**(**length **+** 1**);**

**while** **(**p **&&** **!**p**->**go**[**token**])** **{**

p**->**go**[**token**]** **=** np**;**

p **=** p**->**parent**;**

**}**

**if** **(!**p**)** **{**

np**->**parent **=** start**;**

**}** **else** **{**

State **\***q **=** p**->**go**[**token**];**

**if** **(**p**->**length **+** 1 **==** q**->**length**)** **{**

np**->**parent **=** q**;**

**}** **else** **{**

State **\***nq **=** **new** State**(**p**->**length **+** 1**);**

memcpy**(**nq**->**go**,** q**->**go**,** **sizeof(**q**->**go**));**

nq**->**parent **=** q**->**parent**;**

np**->**parent **=** q**->**parent **=** nq**;**

**while** **(**p **&&** p**->**go**[**token**]** **==** q**)** **{**

p**->**go**[**token**]** **=** nq**;**

p **=** p**->**parent**;**

**}**

**}**

**}**

**return** np**;**

**}**

**};**

## DLX

**struct** DLX{

**struct** Node{ Node \*L, \*R, \*U, \*D; **int** col, row;

} \*head, \*row[Maxn], \*col[Maxm], node[Maxn \* Maxm];

**int** colsum[Maxm], cnt;

/\* dancing link 精确覆盖问题 可以添加迭代加深优化：

\* 1）枚举深度h; \* 2）若当前深度+predeep > h return **false** \*/

/\* **int** predeep() {

**bool** vis[Maxm]; **int** ret = 0; memset(vis, 0, sizeof(vis));

**for** (Node \*p = head->R; p != head; p = p->R)

**if** (!vis[p->col]) { ret ++ ; vis[p->col] ++ ;

**for** (Node \*q = p->D; q != p; q = p->D)

**for** (Node \*r = q->R; r != q; r = r->R)

vis[r->col] = **true**;

} return ret; } //\*/

**void** init(**int** mat[][Maxm], **int** n, **int** m) {

cnt = 0; head = &node[cnt ++ ];

memset(colsum, 0, sizeof (colsum) );

**for**(**int** i = 1; i <= n; i ++ ) row[i] = &node[cnt ++ ];

**for**(**int** j = 1; j <= m; j ++ ) col[j] = &node[cnt ++ ];

head->D = row[1], row[1]->U = head;

head->R = col[1], col[1]->L = head;

head->U = row[n], row[n]->D = head;

head->L = col[m], col[m]->R = head;

head->row = head->col = 0;

**for**(**int** i = 1; i <= n; i ++ ) {

**if** (i != n) row[i]->D = row[i + 1];

**if** (i != 1) row[i]->U = row[i - 1];

row[i]->L = row[i]->R = row[i];

row[i]->row = i, row[i]->col = 0;}

**for**(**int** i = 1; i <= m; i ++ ) {

**if** (i != m) col[i]->R = col[i + 1];

**if** (i != 1) col[i]->L = col[i - 1];

col[i]->U = col[i]->D = col[i]; col[i]->col = i, col[i]->row = 0;}

**for**(**int** i = n; i > 0; i -- ) **for**(**int** j = m; j > 0; j -- )

**if**(mat[i][j]) { Node \*p = &node[cnt ++ ];

p->R = row[i]->R, row[i]->R->L = p;

p->L = row[i], row[i]->R = p;

p->D = col[j]->D, col[j]->D->U = p;

p->U = col[j], col[j]->D = p;

p->row = i; p->col = j; colsum[j] ++ ; }

}

**void** remove(Node \*c) {

c->L->R = c->R; c->R->L = c->L;

**for**(Node \*p = c->D; p != c; p = p->D)

**for**(Node \*q = p->R; q != p; q = q->R) {

q->U->D = q->D;q->D->U = q->U;colsum[q->col] -- ;}}

**void** resume(Node \*c) {

**for**(Node \*p = c->U; p != c; p = p->U)

**for**(Node \*q = p->L; q != p; q = q->L) {

q->U->D = q; q->D->U = q; colsum[q->col] ++ ;}

col[c->col]->L->R = col[c->col]; col[c->col]->R->L = col[c->col]; }

**int** dfs(**int** deep) {

**if**(head->R == head) return deep; Node \*p, \*q = head->R;

**for**(p = head->R; p != head; p = p->R)

**if**(colsum[p->col] < colsum[q->col]) q = p;

remove(q);

**for**(p = q->D; p != q; p = p->D) {

**for**(Node\* r = p->R; r != p; r = r->R)

**if** (r->col != 0) remove (col[r->col]);

/\*可修改区域\*/ans[deep] = p->row;/\*------\*/

**int** sta = dfs (deep + 1); **if**(sta) return sta;

**for**(Node\* r = p->L; r != p; r = r->L)

**if**(r->col != 0) resume (col[r->col]);

}

resume(q); return **false**;

}

///\*可重复覆盖

**void** remove(Node \*c) {

**for**(Node \* p = c->D; row[p->row] != row[c->row]; p = p->D)

p->R->L = p->L; p->L->R = p->R; }

**void** resume(Node \*c) {

**for**(Node \* p = c->U; row[p->row] != row[c->row]; p = p->U)

p->L->R = p->R->L = p; }

**int** dfs(**int** deep) {

**if**(head->R == head) return deep <= K;

**if**(deep + predeep() > K) return **false**;

Node \*p, \*q = head->R, \*r;

**for**(p = head->R; p != head; p = p->R)

**if**(colsum[p->col] < colsum[q->col]) q = p;

**for**(p = q->D; p != q; p = p->D) { remove(p);

**for**(r = p->R; r != p; r = r->R) **if**(r->col != 0) remove(r);

/\*可修改区域\*/ans[deep] = p->row;/\*------\*/

**int** sta = dfs(deep + 1); **if**(sta) return sta;

**for**(r = p->L; r != p; r = r->L) **if**(r->col != 0) resume(r);

resume(p); }

return **false**; }

//可重复覆盖\*/

} dlx;

## 综合

**网络流拓展**:

1.无源汇上下界可行流:

添加附加源汇S,T 对于某边 (u,v) 在新网络中连边S->v容量B[u,v],u->T容量B[u,v],u->v容量C[u,v]-B[u,v].最后,一样也是求一下新网络的最大流,判断从附加源点的边,是否都满流即可.求具体的解:根据最前面提出的强制转换方式,边(u,v)的最终解中的实际流量即为g[u,v]+B[u,v]

2.有源汇上下界可行流

从汇点到源点连一条上限为INF,下限为0的边.按照1.无源汇的上下界可行流一样做即可.改成无源汇后,求的可行流是类似环的,流量即T->S边上的流量.这样做显然使S,T也变得流量平衡了.

3.有源汇的上下界最大流

方法一 :2.有源汇上下界可行流中,从汇点到源点的边改为连一条上限为INF,下限为x的边.因为显然x>ans即MIN(T->S )> MAX(S->T),会使求新网络的无源汇可行流无解的（S,T流量怎样都不能平衡）而x<=ans会有解.所以满足二分性质,二分x,最大的x使得新网络有解的即是所求答案原图最大流.

方法二:从汇点T到源点S连一条上限为INF,下限为0的边,变成无源汇的网络.照求无源汇可行流的方法(如1)，建附加源点S'与汇点T',求一遍S'->T‘的最大流.再把从汇点T到源点S的这条边拆掉.求一次从S到T的最大流即可.(关于S',T'的边好像可以不拆?)(这样一定满足流量平衡?)表示这方法我也没有怎么理解.

4.有源汇的上下界最小流

方法一:2.有源汇上下界可行流中,从汇点到源点的边改为连一条上限为x,下限为0的边.与3同理,二分上限,最小的x使新网络无源汇可行流有解,即是所求答案原图最小流.