**图论**

**MST**

**最小生成树**

**Kruskal**

9.克鲁斯卡尔算法

/\*

|Kruskal算法|

|适用于 稀疏图 求最小生成树|

|16/11/05ztx thanks to wangqiqi|

\*/

/\*

第一步：点、边、加入vector，把所有边按从小到大排序

第二步：并查集部分 + 下面的code

\*/

void Kruskal() {

ans = 0;

for (int i = 0; i<len; i++) {

if (Find(edge[i].a) != Find(edge[i].b)) {

Union(edge[i].a, edge[i].b);

ans += edge[i].len;

}

}

}

**Prim**

10.普里姆算法

/\*

|Prim算法|

|适用于 稠密图 求最小生成树|

|堆优化版，时间复杂度：O(elgn)|

|16/11/05ztx, thanks to chaixiaojun|

\*/

struct node {

int v, len;

node(int v = 0, int len = 0) :v(v), len(len) {}

bool operator < (const node &a)const { // 加入队列的元素自动按距离从小到大排序

return len> a.len;

}

};

vector<node> G[maxn];

int vis[maxn];

int dis[maxn];

void init() {

for (int i = 0; i<maxn; i++) {

G[i].clear();

dis[i] = INF;

vis[i] = false;

}

}

int Prim(int s) {

priority\_queue<node>Q; // 定义优先队列

int ans = 0;

Q.push(node(s,0)); // 起点加入队列

while (!Q.empty()) {

node now = Q.top(); Q.pop(); // 取出距离最小的点

int v = now.v;

if (vis[v]) continue; // 同一个节点，可能会推入2次或2次以上队列，这样第一个被标记后，剩下的需要直接跳过。

vis[v] = true; // 标记一下

ans += now.len;

for (int i = 0; i<G[v].size(); i++) { // 开始更新

int v2 = G[v][i].v;

int len = G[v][i].len;

if (!vis[v2] && dis[v2] > len) {

dis[v2] = len;

Q.push(node(v2, dis[v2])); // 更新的点加入队列并排序

}

}

}

return ans;

}

**Bellman-Ford**

**单源最短路**

**Dijkstra**

11.迪杰斯特拉算法

/\*

|Dijkstra算法|

|适用于边权为正的有向图或者无向图|

|求从单个源点出发，到所有节点的最短路|

|优化版：时间复杂度 O(elbn)|

|16/11/05ztx, thanks to chaixiaojun|

\*/

struct node {

int v, len;

node(int v = 0, int len = 0) :v(v), len(len) {}

bool operator < (const node &a)const { // 距离从小到大排序

return len > a.len;

}

};

vector<node>G[maxn];

bool vis[maxn];

int dis[maxn];

void init() {

for (int i = 0; i<maxn; i++) {

G[i].clear();

vis[i] = false;

dis[i] = INF;

}

}

int dijkstra(int s, int e) {

priority\_queue<node>Q;

Q.push(node(s, 0)); // 加入队列并排序

dis[s] = 0;

while (!Q.empty()) {

node now = Q.top(); // 取出当前最小的

Q.pop();

int v = now.v;

if (vis[v]) continue; // 如果标记过了, 直接continue

vis[v] = true;

for (int i = 0; i<G[v].size(); i++) { // 更新

int v2 = G[v][i].v;

int len = G[v][i].len;

if (!vis[v2] && dis[v2] > dis[v] + len) {

dis[v2] = dis[v] + len;

Q.push(node(v2, dis[v2]));

}

}

}

return dis[e];

}

**SPFA**

12.最短路径快速算法（Shortest Path Faster Algorithm）

/\*

|SPFA算法|

|队列优化|

|可处理负环|

\*/

vector<node> G[maxn];

bool inqueue[maxn];

int dist[maxn];

void Init()

{

for(int i = 0 ; i < maxn ; ++i){

G[i].clear();

dist[i] = INF;

}

}

int SPFA(int s,int e)

{

int v1,v2,weight;

queue<int> Q;

memset(inqueue,false,sizeof(inqueue)); // 标记是否在队列中

memset(cnt,0,sizeof(cnt)); // 加入队列的次数

dist[s] = 0;

Q.push(s); // 起点加入队列

inqueue[s] = true; // 标记

while(!Q.empty()){

v1 = Q.front();

Q.pop();

inqueue[v1] = false; // 取消标记

for(int i = 0 ; i < G[v1].size() ; ++i){ // 搜索v1的链表

v2 = G[v1][i].vex;

weight = G[v1][i].weight;

if(dist[v2] > dist[v1] + weight){ // 松弛操作

dist[v2] = dist[v1] + weight;

if(inqueue[v2] == false){ // 再次加入队列

inqueue[v2] = true;

//cnt[v2]++; // 判负环

//if(cnt[v2] > n) return -1;

Q.push(v2);

} } }

}

return dist[e];

}

/\*

不断的将s的邻接点加入队列，取出不断的进行松弛操作，直到队列为空

如果一个结点被加入队列超过n-1次，那么显然图中有负环

\*/

**Floyd-Warshall**

13.弗洛伊德算法

/\*

|Floyd算法|

|任意点对最短路算法|

|求图中任意两点的最短距离的算法|

\*/

for (int i = 0; i < n; i++) { // 初始化为0

for (int j = 0; j < n; j++)

scanf("%lf", &dis[i][j]);

}

for (int k = 0; k < n; k++) {

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

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

dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);

}

}

}

**二分图**

14.染色法

/\*

|交叉染色法判断二分图|

|16/11/05ztx|

\*/

int bipartite(int s) {

int u, v;

queue<int>Q;

color[s] = 1;

Q.push(s);

while (!Q.empty()) {

u = Q.front();

Q.pop();

for (int i = 0; i < G[u].size(); i++) {

v = G[u][i];

if (color[v] == 0) {

color[v] = -color[u];

Q.push(v);

}

else if (color[v] == color[u])

return 0;

}

}

return 1;

}

15..匈牙利算法

/\*

|求解最大匹配问题|

|递归实现|

|16/11/05ztx|

\*/

vector<int>G[maxn];

bool inpath[maxn]; // 标记

int match[maxn]; // 记录匹配对象

void init()

{

memset(match, -1, sizeof(match));

for (int i = 0; i < maxn; ++i) {

G[i].clear();

}

}

bool findpath(int k) {

for (int i = 0; i < G[k].size(); ++i) {

int v = G[k][i];

if (!inpath[v]) {

inpath[v] = true;

if (match[v] == -1 || findpath(match[v])) { // 递归

match[v] = k; // 即匹配对象是“k妹子”的

return true;

}

}

}

return false;

}

void hungary() {

int cnt = 0;

for (int i = 1; i <= m; i++) { // m为需要匹配的“妹子”数

memset(inpath, false, sizeof(inpath)); // 每次都要初始化

if (findpath(i)) cnt++;

}

cout << cnt << endl;

}

/\*

|求解最大匹配问题|

|dfs实现|

|16/11/05ztx|

\*/

int v1, v2;

bool Map[501][501];

bool visit[501];

int link[501];

int result;

bool dfs(int x) {

for (int y = 1; y <= v2; ++y) {

if (Map[x][y] && !visit[y]) {

visit[y] = true;

if (link[y] == 0 || dfs(link[y])) {

link[y] = x;

return true;

} } }

return false;

}

void Search() {

for (int x = 1; x <= v1; x++) {

memset(visit,false,sizeof(visit));

if (dfs(x))

result++;

}

}

**动态规划**

**背包**

16.17.18背包问题

/\*

|01背包|

|完全背包|

|多重背包|

|16/11/05ztx|

\*/

// 01背包：

void bag01(int cost,int weight) {

for(i = v; i >= cost; --i)

dp[i] = max(dp[i], dp[i-cost]+weight);

}

// 完全背包：

void complete(int cost, int weight) {

for(i = cost ; i <= v; ++i)

dp[i] = max(dp[i], dp[i - cost] + weight);

}

// 多重背包：

void multiply(int cost, int weight, int amount) {

if(cost \* amount >= v)

complete(cost, weight);

else{

k = 1;

while (k < amount){

bag01(k \* cost, k \* weight);

amount -= k;

k += k;

}

bag01(cost \* amount, weight \* amount);

}

}

// other

int dp[1000000];

int c[55], m[110];

int sum;

void CompletePack(int c) {

for (int v = c; v <= sum / 2; ++v){

dp[v] = max(dp[v], dp[v - c] + c);

}

}

void ZeroOnePack(int c) {

for (int v = sum / 2; v >= c; --v) {

dp[v] = max(dp[v], dp[v - c] + c);

}

}

void multiplePack(int c, int m） {

if (m \* c > sum / 2)

CompletePack(c);

else{

int k = 1;

while (k < m){

ZeroOnePack(k \* c);

m -= k;

k <<= 1;

}

if (m != 0){

ZeroOnePack(m \* c);

}

}

}

**LIS**

19.最长上升子序列

/\*

|最长上升子序列|

|状态转移|

|16/11/05ztx|

\*/

/\*

状态转移dp[i] = max{ 1.dp[j] + 1 }; j<i; a[j]<a[i];

d[i]是以i结尾的最长上升子序列

与i之前的 每个a[j]<a[i]的 j的位置的最长上升子序列+1后的值比较

\*/

void solve(){ // 参考挑战程序设计入门经典;

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

dp[i] = 1;

for(int j = 0; j < i; ++j){

if(a[j] < a[i]){

dp[i] = max(dp[i], dp[j] + 1);

} } }

}

/\*

优化方法：

dp[i]表示长度为i+1的上升子序列的最末尾元素

找到第一个比dp末尾大的来代替

\*/

void solve() {

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

dp[i] = INF;

}

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

\*lower\_bound(dp, dp + n, a[i]) = a[i]; // 返回一个指针

}

printf("%d\n", \*lower\_bound(dp, dp + n, INF) - dp;

}

/\*

函数lower\_bound()返回一个 iterator 它指向在[first,last)标记的有序序列中可以插入value，而不会破坏容器顺序的第一个位置，而这个位置标记了一个不小于value的值。

\*/

**LCS**

20.最长公共子序列

/\*

|求最长公共子序列|

|递推形式|

|16/11/05ztx|

\*/

void solve() {

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

for (int j = 0; j < m; ++j) {

if (s1[i] == s2[j]) {

dp[i + 1][j + 1] = dp[i][j] + 1;

}else {

dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);

} } }

}