Graph Üzerinde Oyunlar

Graph üzerinde 2 kişi tarafından oynanan bir oyun var. Her nodedan çıkan edgelerle diğer hamleye geçebiliyorsunuz.

En genel durumu düşüneceğiz. Yani rastgele üretilmiş döngüler içeren tek yönlü bir graph. Amacımız, verilen bir başlangıç noktası için o noktada kimin avantajlı olduğunu bulmak(ya da beraberlik durumunu).

Bu problemi graphtaki her node için güzel bir şekilde çözebiliriz. Zaman karmaşıklığımız O(M) olur(M graphtaki edge sayısı).

Algoritmanın Tanımı

Dışarıya doğru edgei olmayan nodelardayken oyunun kesinlikle kaybedildiğini görebiliriz.
Bazı gözlemlerimiz var:

- Eğer A nodeundan kaybeden nodelardan herhangi birine bağlantı varsa A nodeu kazanan nodedur.
- Eğer bir nodedan sadece kazanan nodelara gidiş varsa o node kaybeden nodedur.
- İlk iki duruma uymayan nodelar beraberlik nodelarıdır.

Algoritmanın çalışma zamanı O(NM). Bütün nodelar için 1. ve 2. kuralı uygulamayı deneyeceğiz. Ve bu denemeleri artık graphta güncelleme yapamadığımız zamana kadar devam ettireceğiz. Fakat bu döngülerin işlem maliyeti çok fazla olabilir.

Algoritmayı ilk başta kazanan ya da kaybeden olduğu kesin olarak bilinen nodelardan başlatacağız. İşlemleri kolayca uygulayabilmemiz için bir nodedan başka nodea geçerken edgelerin yönlerini tersine düşüneceğiz. dfs, sadece kazanan ya da kaybeden olduğu kesin olan nodelara gidecek. Eğer dfs kaybeden bir nodedan ne olduğu belli olmayan bir nodea gitmek istiyorsa o nodeu kazanan nodea çevirip gitmelidir. Eğer dfs kazanan bir nodedan ne olduğu belli olmayan bir nodea gitek istiyorsa o nodedan gidilen(edgelerin normal yönünde) bütün nodeların durumlarını kontrol etmelidir. Bu kontrol O(1) işlemde yapılabilir. Her node için o nodedan kaç tane kazanan sayıda nodea edge olduğunu tutan bir sayaca ihtiyacımız var. dfs; kazanan bir nodedan ne olduğu belli olmayan bir nodea gitmek istediğinde o nodeun sayacını bir arttıracak, eğer nodeun sayacı o nodedan çıkan edgelrin sayısına eşit olursa artık o node kaybeden nodedur ve dfs o nodea gidebilir. Diğer durumda sadece sayacı arttırıp geçeceğiz.

Açıkça görülüyor ki her kaybeden ve kazanan node dfs tarafından sadece bir kez ziyaret ediliyor, ne olduğu belli olmayan nodelar ise ziyaret edilmiyor. Sonuç olarak bu algoritmanın zaman karmaşıklığı O(M).

Kodlama

Graphı g[] vector dizisinde, nodeların dışarı doğru olan edgelerinin sayısının da degree[] dizisinde tutuyoruz. Başlangıç nodelarımız da durumlarına göre win[] ve loose[] dizisinde işaretliler.

```
vector<int> g [100];
bool win [100];
bool loose [100];
bool used[100];
int degree[100];

void dfs (int v) {
	used[v] = true;
	for (vector<int>::iterator i = g[v].begin(); i != g[v].end(); ++i)
		if (!used[*i]) {
			if (loose[v])
				win[*i] = true;
			else if (--degree[*i] == 0)
				loose[*i] = true;
			else
				continue;
			dfs (*i);
		}
}
```

“Hırsız Polis” Problemi

Algoritma daha açık bir hâle geldi. Artık biraz daha somut bir problemle uğraşabiliriz. M*N tane hücreden oluşan bir alan var. Bunların bazılarına ulaşım yok. Bir polis ve hırsız var(başlangıç koordinatları veriliyor). Ve bazı kaçış noktları da mevcut. Eğer polis ile hırsız aynı hücrede bulunurlarsa oyunu polis kazanır. Eğer hırsız bir çıkış noktasına gelirse(o sırada bu noktada polis olmaması lazım) oyunu hırsız kazanır. Polis 8 yönde, hırsız ise 4 yönde(koordinat eksenerine paralel) hareket edebiliyor. Polis de hırsız da isterlerse herhangi bir turda yerlerinde kalabilirler. İlk hamleyi polis yapar ve sırayla oynarlar.

İlk olarak oyun alanının graphını kuracağız. Oyunun kurallarını şekillendirmeliyiz. Oyunun o anki durumunu polisin yeri olan P, hırsızın yeri olan T ve o an sıranın kimde olduğunu belirten Pstep ile belirtebiliriz. Böylece (P,T,Pstep) bir node belirtmiş olur

Sonraki adımda hangi nodeların kaybetme ya da kazanma nodeu olduğunu bulmalıyız. Buradaki önemli nokta, bir yerin kazanan ya da kaybeden olması durumunun Pstep’e göre(yani sıranın kimde olduğuna göre) değişebileceğidir. Sıranın poliste olduğu durumlar için eğer hırsızla polis aynı

yerdelerse o durum bir kazanma durumudur. Eğer hırsız herhangi bir çıkış noktasındaysa ve polis oraya o anda ulaşamıyorsa o durum kaybetme durumudur. Şimdi sıranın hırsızda olduğu durumları ele alalım. Hırsız bir çıkış noktasındaysa ve polis orada değilse bu hırsız için bir kazanma durumudur. Polisle hırsızın aynı yerde olduğu herhangi bir durum ise hırsız için kaybetme durumudur.

Kodlama:

```
struct state {
	char p, t;
	bool pstep;
};

vector<state> g [100][100][2];
// 1 = policeman coords; 2 = thief coords; 3 = 1 if policeman's step or 0 if thief's.
bool win [100][100][2];
bool loose [100][100][2];
bool used[100][100][2];
int degree[100][100][2];

void dfs (char p, char t, bool pstep) {
	used[p][t][pstep] = true;
	for (vector<state>::iterator i = g[p][t][pstep].begin(); i != g[p][t][pstep].end(); ++i)
		if (!used[i->p][i->t][i->pstep]) {
			if (loose[p][t][pstep])
				win[i->p][i->t][i->pstep] = true;
			else if (--degree[i->p][i->t][i->pstep] == 0)
				loose[i->p][i->t][i->pstep] = true;
			else
				continue;
			dfs (i->p, i->t, i->pstep);
		}
}


int main() {

	int n, m;
	cin >> n >> m;
	vector<string> a (n);
	for (int i=0; i<n; ++i)
		cin >> a[i];

	for (int p=0; p<n*m; ++p)
```

```
		for (int t=0; t<n*m; ++t)
			for (char pstep=0; pstep<=1; ++pstep) {
				int px = p/m, py = p%m, tx=t/m, ty=t%m;
				if (a[px][py]=='*' || a[tx][ty]=='*')  continue;

				bool & wwin = win[p][t][pstep];
				bool & lloose = loose[p][t][pstep];
				if (pstep)
					wwin = px==tx && py==ty,   lloose = !wwin && a[tx][ty] ==
'E';
				else
					wwin = a[tx][ty] == 'E',   lloose = !wwin && px==tx &&
py==ty;
				if (wwin || lloose)  continue;

				state st = { p, t, !pstep };
				g[p][t][pstep].push_back (st);
				st.pstep = pstep != 0;
				degree[p][t][pstep] = 1;

				const int dx[] = { -1, 0, 1, 0,   -1, -1, 1, 1 };
				const int dy[] = { 0, 1, 0, -1,   -1, 1, -1, 1 };
				for (int d=0; d<(pstep?8:4); ++d) {
					int ppx=px, ppy=py, ttx=tx, tty=ty;
					if (pstep)
						ppx += dx[d],  ppy += dy[d];
					else
						ttx += dx[d],  tty += dy[d];
					if (ppx>=0 && ppx<n && ppy>=0 && ppy<m &&
a[ppx][ppy]!='*' &&
						ttx>=0 && ttx<n && tty>=0 && tty<m &&
a[ttx][tty]!='*')
					{
						g[ppx*m+ppy][ttx*m+tty][!pstep].push_back (st);
						++degree[p][t][pstep];
					}
				}
			}

	for (int p=0; p<n*m; ++p)
		for (int t=0; t<n*m; ++t)
			for (char pstep=0; pstep<=1; ++pstep)
				if ((win[p][t][pstep] || loose[p][t][pstep]) && !used[p][t][pstep])
```

```
                                        dfs (p, t, pstep!=0);

        int p_st, t_st;
        for (int i=0; i<n; ++i)
                for (int j=0; j<m; ++j)
                        if (a[i][j] == 'C')
                                p_st = i*m+j;
                        else if (a[i][j] == 'T')
                                t_st = i*m+j;

        cout << (win[p_st][t_st][true] ? "WIN" : loose[p_st][t_st][true] ? "LOSS" : "DRAW");

}
```