GRAFOS

Matias Ramos
UTN - FRSF

Training Camp

2021

Contenido

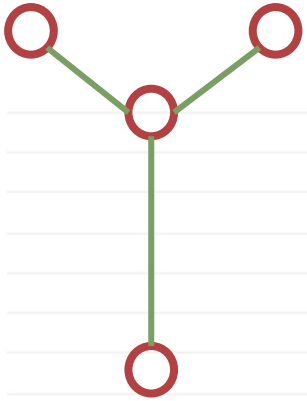
1. Conceptos básicos.
2. Recorriendo un grafo:
    - BFS
    - DFS
3. Más conceptos:
    - Componentes Conexas
    - Grafo Bipartito
    - DAG - Orden Topologico
4. Camino Mínimo:
    - Dijkstra
    - Bellman-Ford
    - Floyd-Warshall
5. Modelado.
6. Arbol Generador Minimo (Kruskal)

01

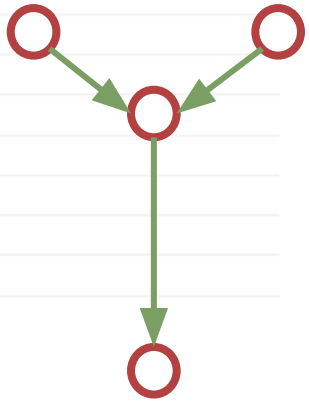
Conceptos Básicos

Grafo

Un grafo es un conjunto de objetos llamados **vértices o nodos**, unidos por enlaces llamados **aristas o arcos**.



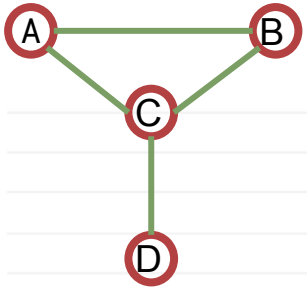
Grafo No Dirigido



Grafo Dirigido

Representación

La **matriz de adyacencia** es una matriz de $n \times n$ donde n es la cantidad de nodos del grafo, que en la posición(i,j) tiene un 1 (o true) si hay una arista entre los nodos i y j, o 0 (o false) en caso contrario.
Uso de memoria: $O(n^2)$



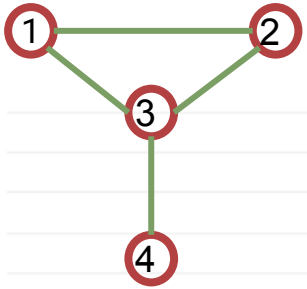
	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	1	1	0	1
D	0	0	1	0

Implementación

```
int n,m;
cin>>n>>m;
int graph[n][n];
for(int i=0;i<m;i++){
     int a,b;
     cin>>a>>b;
     graph[a][b] = 1;
     graph[b][a] = 1;
}
```

Representación

La lista de adyacencia es un vector de vectores de enteros, que en el i-ésimo vector tiene el número *j* si hay una arista entre los nodos *i* y *j*. Uso de memoria: O(m), donde *m* es la cantidad de aristas.



1	2,3
2	1,3
3	1,2,4
4	3

Implementación

```
vector<int> graph[10000];
// voy a tener n<=10000
int n,m;

int main(){
     cin>>n>>m;
     for(int i=0;i<m;i++){
          int a,b;
          cin>>a>>b;
          graph[a].push_back(b);
          graph[b].push_back(a);
     }
}
```

Implementación

```
vector<vector<int>> graph;
int n,m;

int main(){
     cin>>n>>m;
     graph.resize(n);
     for(int i=0;i<m;i++){
          int a,b;
          cin>>a>>b;
          graph[a].push_back(b);
          graph[b].push_back(a);
     }
}
```

02

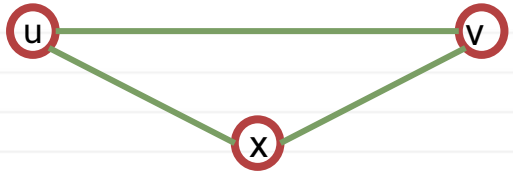
Recorriendo un Grafo

Vecino y Grado

Dada una arista que conecta dos vértices *u* y *v*, decimos que *u* es **vecino** de *v* (y que *v* es vecino de *u*). Además, a la cantidad de vecinos de *u* se le llama el **grado** de u.

Distancia

Definimos la distancia de *u* a *v* como el menor *n* tal que hay un camino de largo *n* de *u* a *v*.



02

BFS

BFS

Búsqueda en Anchura.

Es un algoritmo para recorrer el grafo iniciando en un nodo *v*.

- Inicialmente se procesa *v*, que tiene distancia a si mismo 0, por lo que seteamos $d_v = 0$. Además es el unico a distancia 0.
- Los vecinos de *v* tienen si o si distancia 1, por lo que seteamos $d_y = 1$ para todo *y* vecino de *v*. Ademas, estos son los unicos a distancia 1.
- Ahora tomamos los nodos a distancia 1, y para cada uno nos fijamos en sus vecinos cuya distancia aún no calculamos.Estos son los nodos a distancia 2.
- Asi sucesivamente, para saber cuales son los nodos a distancia *k + 1*, tomamos los vecinos de los nodos a distancia *k* que no hayan sido visitados aun.

Ejemplo

visualgo

Implementación

```
queue<int> q;
bool visited[N];
int distance[N];
```

Implementación

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
     int s = q.front(); q.pop();
     // process node s
     for (auto u : adj[s]) {
          if (visited[u]) continue;
          visited[u] = true;
          distance[u] = distance[s]+1;
          q.push(u);
     }
}
```

BFS

BFS nos deja saber la distancia mínima de un nodo v a los otros.

- Cada nodo lo procesamos una sola vez, porque cuando lo agregamos a la queue, también inicializamos su distancia, por lo que no volverá a ser agregado.
- Como cada nodo es procesado una sola vez, entonces cada arista es procesada una sola vez (o una vez en cada sentido para no-dirigidos).
- Entonces, la complejidad de BFS es O(m), donde m es la cantidad de aristas.

02

DFS

DFS

Búsqueda en profundidad.

Es un algoritmo para recorrer el grafo iniciando en un nodo v.

- Arrancamos de cierto nodo y lo marcamos como visitado.
- Luego, evaluamos sus vecinos de a uno, y cada vez que encontramos uno que no este marcado como visitado, seguimos procesando a partir de el. Este procedimiento es recursivo, por lo que se puede implementar así.
- Cuando no quedan vecinos por visitar salgo para atrás (vuelvo en la recursión).
- Cuando no se conoce el máximo del stack de recursión que usa nuestro juez, no es mala idea implementar el DFS con nuestro propio stack aunque sea mas codigo.

Ejemplo

visualgo

Implementación

```
vector adj[N];
bool visited[N];
```

Implementación

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s]) {
        dfs(u);
    }
}
```

DFS vs BFS

- Un análisis similar al que hicimos para BFS concluye que DFS también es $O(m)$.
- BFS y DFS son similares: ambos procesan todos los nodos alcanzables desde cierto nodo, pero lo hacen en distinto orden.

- Muchos problemas salen con ambas técnicas, por lo que podemos usar la que nos quede más cómodo.
- Algunos problemas solo se resuelven con una de ellas.
- BFS: distancias minimas.
- DFS: algunos problemas que tienen que ver con la estructura del grafo, como por ejemplo, encontrar puentes y puntos de articulación.

03

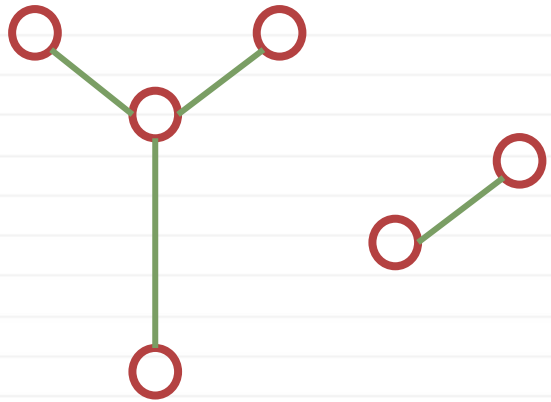
Más Conceptos

Componentes Conexas

- Decimos que dos vértices *u* y *v* están **conectados** si hay un camino (secuencia de aristas con vértice en común) que los une.

- Para todo grafo **no-dirigido**, podemos particionar el conjunto de nodos en varios subconjuntos, tales que dos nodos están conectados si y sólo si pertenecen al mismo subconjunto.

- Estos subconjuntos se denominan **componentes conexas** del grafo.



Problema

Tenemos el mapa de una edificio, y queremos saber cuántas habitaciones existen. Cada posición es piso o pared.
Solo se puede caminar a la izquierda, derecha, arriba o abajo.

INPUT

```
########
#..#...#
####.#.#
#..#...#
########
```

https://cses.fi/problemset/task/1192

Problema

- Hay que contar las componentes conexas!
- Podemos utilizar el input como la representación del grafo

INPUT

```
########
#..#...#
####.#.#
#..#...#
########
```

```
int comp = 0;
for(int i = 0; i < n; i++){
    if(!visited[i]){
        comp++;
        dfs(i);
    }
}
```

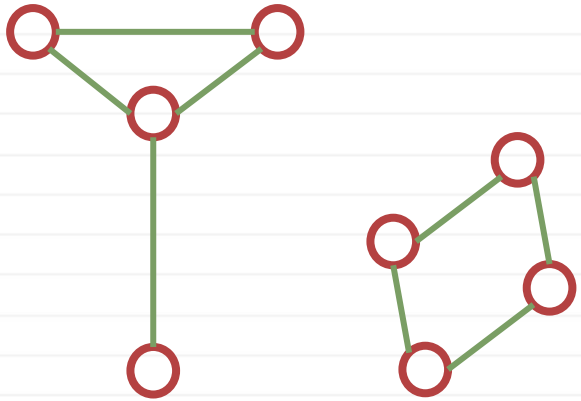
```
int comp = 0;
for(int i = 0; i < n; i++){
for(int j = 0; j < m; j++){
    if(!graph[i][j]!='#'){
        comp++;
        dfs(i);
    }
}}
```

03

Grafo Bipartito

Grafo Bipartito

- Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vertices de distinto color.
- Un grafo es bipartito si y solo si no tiene **ciclos** impares.



Problema

En una clase hay n alumnos, y m amistades entre esos alumnos, queremos armar dos equipos de manera tal que no exista una amistad entre dos miembros del mismo equipo.
Si es imposible realizar esto, mostrar IMPOSSIBLE

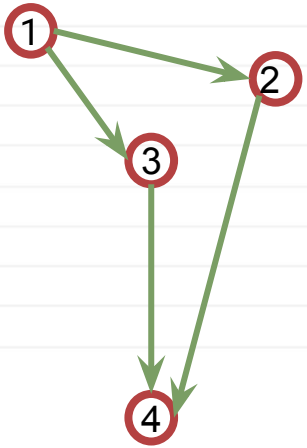
building teams

03

DAG y TopoSort

DAG

Grafo Dirigido Aciclico (no tiene ciclos).



Orden Topologico

Dado un DAG, un orden topológico es un ordenamiento de los nodos de modo que para cada arista $x \rightarrow y$, x viene antes que y en el orden.

Posibles órdenes topológicos:

- 1, 2, 3, 4
- 1, 3, 2, 4

Implementación TopoSort

- Obtenemos el grado de entrada de cada nodo y metemos los de grado 0 en una cola.
- Mientras que la cola no este vacia, tomamos el nodo del frente y "lo eliminamos" del grafo, actualizando el grado de entrada de sus vecinos y metiendo a la cola los que pasen a tener grado 0.

Complejidad O(V+E)

Implementación

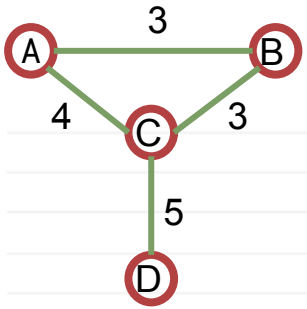
```
vector<int> topSort(vector<vector<int>> g, vector<int> inGrade)
{
     int n = inGrade.size();
     vector<int> topSorted;
     queue<int> q;
     for(int i = 0; i < n; i++) if(inGrade[i] == 0) q.push(i);
     while(!q.empty()) {
          int node = q.front(); q.pop();
          topSorted.push_back(node);
          for(int y : g[node]) if(--inGrade[y] == 0) q.push(y);
     }
     //Si hay ciclos retornar un vector vacio
     if(topSorted.size() < n) topSorted.clear();
     return topSorted;
}
```

04

Camino Mínimo

Aristas con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un **grafo ponderado** es un grafo en el cual las aristas tienen peso. Los pesos de los ejes pueden ser números.
- Los grafos ponderados se pueden representar con lista de adyacencia, guardando además del nodo destino, el peso de la arista (usando un par)



A	{B,3},{C,4}
B	{A,3},{C,3}
C	{A,4},{B,3},{D,5}
D	{C,5}

Importante

BFS ya no nos sirve para conocer camino mínimo teniendo grafos ponderados

04

Dijkstra

Dijkstra

Dado un nodo *v*, el algoritmo de **Dijkstra** calcula la **distancia mínima** a todos los demás nodos en un grafo ponderado (*sin aristas negativas*)

- *v* tiene distancia a si mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).
- Vamos procesando los nodos uno por uno: En cada paso elegimos el nodo *x* que esta a **menor** distancia de v (entre los que no procesamos).
- Para cada **vecino** *y* de *x*, actualizamos su distancia, en caso de que el camino que pasa por *x* justo antes de ir a *y* tenga menor costo que d_y (es decir si $d_x + c < d_y$, donde *c* es el peso de la arista(*x*, *y*)).

Ejemplo

visualgo

Dijkstra

- La complejidad es $O(|V|^2)$ porque en cada uno de los $|V|$ pasos tenemos que buscar el nodo que esta a menor distancia, y cada arista se considera una sola vez.
- Se puede implementar en $O(|E|log(|E|))$, usando una cola de prioridad para elegir el nodo a menor distancia (suele ser necesario usar esta versión en las competencias).

- Dijkstra no funciona si algunas aristas tienen pesos negativos, porque cuando procesamos un nodo, no podemos estar seguros de que su distancia es efectivamente la que calculamos hasta ese punto.
- Ademas, cuando hay pesos negativos, puede que la distancia entre algún par de nodos sea $-\infty$, en caso de que exista algún ciclo negativo.

04

Bellman Ford

Bellman Ford

Dado un nodo (v), el algoritmo de Bellman-Ford calcula la distancia mínima a todos los demás nodos en un grafo ponderado, incluso si hay aristas negativas.

- Asumimos que no hay ciclos negativos.
- *v* tiene distancia a si mismo 0, por lo que seteamos $d_v = 0$.
- Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).
- Iteramos *n-1* veces, en cada iteración intentamos **relajar** la distancia a los nodos usando todas las aristas.
- Se puede demostrar que luego de la i-ésima iteración, ya se han encontrado todos los caminos mínimos que usan *i* o menos aristas.
- Hacemos 1 iteración más, si la distancia a algún nodo disminuye, el grafo contiene un ciclo negativo.

Ejemplo

visualgo

Implementación

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
     for (auto e : edges) {
          int a, b, w;
          tie(a, b, w) = e;
          distance[b] = min(distance[b], distance[a]+w);
     }
}
```

Complejidad *O(n*aristas)*

04

Floyd-Warshall

Floyd Warshall

El algoritmo de **Floyd-Warshall** calcula las distancia entre cada par de nodos, lo hace usando **programación dinámica.**

- Inicialmente, $d_{x,y}$ (distancia de x a y) es el peso de la arista de x a y si esta existe, ∞ si no existe, o 0 si $x = y$.
- Procesamos los nodos uno por uno, y mantenemos como invariante que las distancias son las mínimas que se pueden lograr pasando solo por los nodos que procesamos.
- Cuando procesamos el nodo k, para cada par de nodos i, j actualizamos su distancia como el mínimo entre lo que ya había antes y la distancia si pasamos por k (es decir, $d_{i,k} + d_{k,j}$).

Implementación

```
void floyd(vector<vector<int> > &d){
    // d -> matriz inicial de distancias (diapo anterior)
    for(int k=0;k<n;k++)
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

Complejidad $O(n^3)$

- En el caso de pesos negativos, podemos saber si x está en un ciclo negativo, fijandonos si ocurre $d_{x,x} < 0$.
- La distancia de x a y es $-\infty$ si hay un nodo z que está en un ciclo negativo y podemos ir de x a z y de z a y.

05

Modelado

Problema

Tenemos un laberinto, donde hay una persona S, un cubo B, y un objetivo T.
El cubo solo se puede empujar.
Queremos saber si es posible llevar al cubo al objetivo T, y si esto es posible, hay que mostrar la menor cantidad de pasos posibles qué haría la persona, y la menor cantidad de empujes al cubo

INPUT

```
##########
#T##.....#
#.#.#..####
#....B....#
#.######..#
#.....S...#
##########
```

https://www.urionlinejudge.com.br/judge/en/problems/view/2056

Soluciones?

Quizás la primero que uno pensaría es hacer diferentes caminos mínimos, desde S a B, y de B a T. Pero esto no tiene en cuenta los empujes.

Lo que hay que hacer es modelar un grafo de estados!

INPUT

```
##########
#T##.....#
#.#.#..####
#....B....#
#.######..#
#.....S...#
##########
```

Modelado

Podemos plantear nodos con la posición de S y B. Vamos a tener un nodo inicial (S,B) y queremos llegar a un nodo final donde B == T, y no interesa donde este S.

Cada nodo solo podra tener como máximo, 4 nodos adyacentes, las 4 direcciones en que nos podemos mover.

Y hay que tener en cuenta, que si S está al lado de B, lo puede mover.

INPUT

```
###########
#T##......#
#.#.#..####
#....B....#
#.######..#
#.....S...#
###########
```

Y por último correr Dijkstra en este nuevo grafo planteado

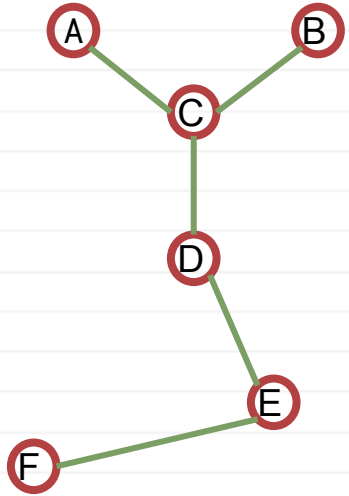
Para muchos problemas puede ser útil tratar de **modelar** un grafo, de tal forma que lo que pide el problema se transforme en una "consulta" sobre ese grafo. La forma de hacer este modelado suele ser con un **vértice por cada estado posible** distinto que puede haber en el problema y **una arista por cada transición posible entre dos estados**.

06

Arbol Generador Minimo

Arboles

- Un **árbol** es un grafo conexo sin ciclos.
- Una particularidad de los árboles es que siempre $|E| = |V| - 1$.



06

Union-Find

Union Find

Union-Find es una estructura de datos que mantiene las componentes conexas de un grafo.

Las operaciones del Union-Find se pueden realizar de manera muy eficiente (practicamente O(1)).

Operaciones soportadas

- *find(x)*: Devuelve un id de la componente conexa en la que está el nodo x. Se puede usar para ver si dos nodos están en la misma componente chequeando find(x) == find(y).
- *union(x, y)*: Agrega una arista entre los nodos *x* e *y*. Es decir, junta las componentes de *x* e *y* en la misma componente.

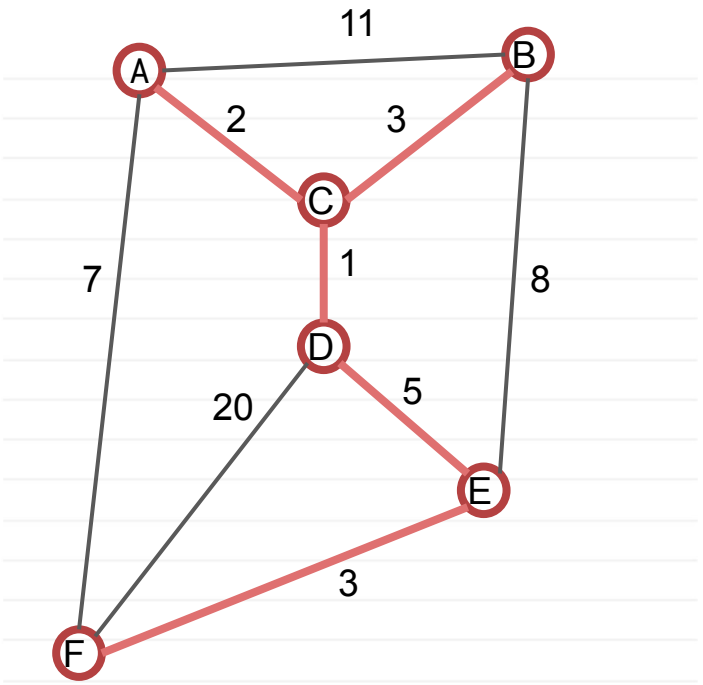
06

Arbol Generador Minimo

Arbol Generador Minimo

Dado un grafo conexo G, un **árbol generador** de G es un árbol que tiene todos los nodos de G y cuyas aristas también son aristas de G.

Si las aristas de G tienen pesos, un árbol generador mınimo de G es uno que cumple que la suma de sus aristas es lo mas chica posible.



T = 14

06

Kruskal

El algoritmo de Kruskal calcula el árbol generador mínimo como sigue:

- Mantengo un union-find con las componentes conexas determinadas por las aristas que agregue al árbol.
- Armó un arreglo con todas las aristas. Las ordeno por peso de menor a mayor.
- Recorro cada arista: Si los nodos que conecta estan en la misma componente, entonces no hago nada. Si estan en distintas, agregó la arista al árbol y hago unión de los dos nodos en el union-find.
- La complejidad es lineal, excepto en la parte de ordenar las aristas. Entonces en total es $O(|E|log(|E|))$
- Para Kruskal, no hace falta representar el grafo como lista de adyacencia, sino simplemente como un arreglo con las aristas.

Ejemplo

visualgo

Implementación

```
int kruskal(vector<pair<int, pair<int, int>>> edges, int n) {
    //edges: lista de aristas en la forma {peso, {nodo1, nodo2}}
    sort(edges.begin(),edges.end()); // ordena por peso
    init_uf(n);
    int u = 0, res = 0;
    for(auto p: edges){
        int c = p.first, x = p.second.first, y = p.second.second;
        x = find(x); y = find(y);
        if(x == y) continue; // los nodos ya estan conectados
        res += c;
        u++;
        join(x,y);
        if(u == n - 1) // completamos el arbol?
            return res;
    }
    return -1; // si llegamos hasta aca entonces no es conexo
}
```

07

Como sigo?

Recomendación Personal

Competitive Programmer's Handbook

+

CSES Problem Set

Graph Algorithms

Counting Rooms	8815 / 9416	–
Labyrinth	5514 / 6814	–
Building Roads	7088 / 7413	–
Message Route	6084 / 6362	–
Building Teams	5874 / 6153	–
Round Trip	4738 / 5191	–
Monsters	2627 / 3355	–
Shortest Routes I	4828 / 5398	–
Shortest Routes II	4144 / 4472	–
High Score	2401 / 3347	–
Flight Discount	2814 / 3310	–
Cycle Finding	2324 / 2746	–
Flight Routes	1975 / 2175	–
Round Trip II	2277 / 2711	–
Course Schedule	3133 / 3258	–
Longest Flight Route	2038 / 2710	–
Game Routes	2470 / 2604	–
Investigation	1637 / 1788	–
Planets Queries I	1640 / 1753	–
Planets Queries II	708 / 852	–
Planets Cycles	980 / 1096	–
Road Reparation	2001 / 2078	–
Road Construction	2000 / 2065	–
Flight Routes Check	1875 / 2064	–
Planets and Kingdoms	1607 / 1655	–
Giant Pizza	751 / 856	–
Coin Collector	1084 / 1193	–

Gracias!

mramos@frsf.utn.edu.ar

Suerte en el Contest de hoy!