**Método De La Ingeniería**

# **Identificación del problema:**

## **Descripción del contexto problemático**

Los grafos son una estructura de datos que tiene aristas y vértices, una arista puede tener o no un peso; conecta a un nodo con otro nodo o con el mismo. Gracias a estas características, un grafo puede representarse de diferentes maneras, como un árbol, como una figura geométrica o como un laberinto (Dependiendo la cantidad de aristas la representación puede ser una imagen 2D o 3D).

Precisamente los laberintos, han sido siempre los problemas más estudiados, tanto que hemos usado a los animales a resolver problemas con laberintos, un ejemplo de esto es que un ratón, sea lo suficientemente listo como para salir de laberinto en el menor tiempo posible y por el camino más corto.

## **Identificación del problema**

Se debe hacer que un ratón, que se encuentre en algún punto del laberinto, salga de este, pero lo debe hacer usando el camino más corto que le permita salir del laberinto.

## **Requerimientos funcionales**

R1 = Generar un laberinto.

R2 = Posicionar el ratón en algún punto del laberinto

R3 = Resolver el laberinto desde el punto donde se encuentra el laberinto hasta la salida más cercana.

# **Recopilación de la información necesaria**

## **Marco teórico:**

**Que es un laberinto:** Un laberinto (del [latín](https://es.wikipedia.org/wiki/Lat%C3%ADn) labyrinthus, y este del [griego](https://es.wikipedia.org/wiki/Idioma_griego) λαβύρινθος labýrinzos) es un lugar formado por calles y encrucijadas, intencionadamente complejo para confundir a quien se adentre en el mismo. La etimología de la palabra es dudosa, aunque parece provenir de [Asia Menor](https://es.wikipedia.org/wiki/Asia_Menor). (Wikipedia, 2018)

**Algoritmos de camino más corto:**

**Algoritmo de Bellman Ford**: El algoritmo de Bellman Ford se usa para encontrar las rutas más cortas desde el vértice de origen a todos los otros vértices en un gráfico ponderado. Depende del siguiente concepto: La ruta más corta contiene a lo sumo n-1 bordes, por eso la ruta más corta no puede tener un ciclo.

**El algoritmo de Dijkstra**: El algoritmo de Dijkstra tiene muchas variantes, pero la más común es encontrar las rutas más cortas desde el vértice de origen a todos los otros vértices en el gráfico.

**El algoritmo de Floyd – Warshall:** El algoritmo de Floyd-Warshall se usa para encontrar las rutas más cortas entre todos los pares de vértices en una gráfica, donde cada borde de la gráfica tiene un peso que es positivo o negativo. La mayor ventaja de usar este algoritmo es que todas las distancias más cortas entre 2 vértices podrían calcularse en O (V3), donde V es el número de vértices en una gráfica. (HackerEarthShortesPathAlgorithms, 2018)

# **Búsqueda de soluciones creativas**

1. Generar un laberinto aleatoriamente y colocar al ratón en una posición aleatoria.
2. Generar un laberinto paso a paso y que el ratón lo vaya solucionando.
3. Generar un laberinto aleatoriamente, pero con condiciones del usuario tales como: la posición de la salida, la posición del ratón y que tan grande será el laberinto.
4. Generar un laberinto semi aleatorio que cumpla las condiciones de que el grafo no sea un multígrafo, sea un grafo ponderado, no cíclico y completamente conexo.
5. Generar un laberinto aleatorio, y después usar un algoritmo de mínima expansión para no tener ningún inconveniente con la solución del problema
6. Tener una librería con laberintos predefinidos, tanto a nivel grafico como de estructura y que el usuario solo tenga que elegir el laberinto que quiere que el ratón resuelva y desde que punto hasta la salida

# **Transición de la formulación de ideas a los diseños preliminares**

## **Descarte de ideas no factibles**

Idea 1: Algo completamente aleatorio es difícil de controlar y es posible que no haya solución al laberinto o no cumpla con las especificaciones de un laberinto (la salida debe ser una hoja, la parte grafica del laberinto se puede dibujar en 2d.

Idea 2: Es muy lento a y al usuario no le parecerá un proyecto atractivo

Idea 6: Es una buena opción pero se limita en la cantidad de laberintos, y poner más laberintos sería llenar tontamente memoria secundaria o principal, cuando el ratón debe solucionar un laberinto a la vez, además buscar laberintos específicos con esas librerías es tedioso y llevaría un largo proceso de archivar.

## **Diseños preliminares**

* Seudocódigos necesarios

Código DFS

Complejidad Temporal O(V+E).

* DFS-iterative (G, s): //Where G is graph and s is source vertex
* let S be stack
* S.push( s ) //Inserting s in stack
* mark s as visited.
* while ( S is not empty):
* //Pop a vertex from stack to visit next
* v = S.top( )
* S.pop( )
* //Push all the neighbours of v in stack that are not visited
* for all neighbours w of v in Graph G:
* if w is not visited :
* S.push( w )
* mark w as visited
* DFS-recursive(G, s):
* mark s as visited
* for all neighbours w of s in Graph G:
* if w is not visited:
* DFS-recursive(G, w)

(HackerEarthDfs, 2018)

Código BFS

Complejidad Temporal  O(V + E).

BFS (G, s) //Where G is the graph and s is the source node

let Q be queue.

Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

mark s as visited.

while ( Q is not empty)

//Removing that vertex from queue,whose neighbour will be visited now

v = Q.dequeue( )

//processing all the neighbours of v

for all neighbours w of v in Graph G

if w is not visited

Q.enqueue( w ) //Stores w in Q to further visit its neighbour

mark w as visited.

(HackerEarthBfs, 2018)

Algoritmo de Bellman Ford

Complejidad Temporal O(V⋅E), en caso que E=V^2, O(E3).

vector <int> v [2000 + 10];

int dis [1000 + 10];

for(int i = 0; i < m + 2; i++){

v[i].clear();

dis[i] = 2e9;

}

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

scanf("%d%d%d", &from , &next , &weight);

v[i].push\_back(from);

v[i].push\_back(next);

v[i].push\_back(weight);

}

dis[0] = 0;

for(int i = 0; i < n - 1; i++){

int j = 0;

while(v[j].size() != 0){

if(dis[ v[j][0] ] + v[j][2] < dis[ v[j][1] ] ){

dis[ v[j][1] ] = dis[ v[j][0] ] + v[j][2];

}

j++;

}

}

Dijkstra's Algorithm

Complejidad temporal O(V2) si se usa una cola de prioridad O(V+ElogV).

#define SIZE 100000 + 1

vector < pair < int , int > > v [SIZE]; // each vertex has all the connected vertices with the edges weights

int dist [SIZE];

bool vis [SIZE];

void dijkstra(){

// set the vertices distances as infinity

memset(vis, false , sizeof vis); // set all vertex as unvisited

dist[1] = 0;

multiset < pair < int , int > > s; // multiset do the job as a min-priority queue

s.insert({0 , 1}); // insert the source node with distance = 0

while(!s.empty()){

pair <int , int> p = \*s.begin(); // pop the vertex with the minimum distance

s.erase(s.begin());

int x = p.s; int wei = p.f;

if( vis[x] ) continue; // check if the popped vertex is visited before

vis[x] = true;

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

int e = v[x][i].f; int w = v[x][i].s;

if(dist[x] + w < dist[e] ){ // check if the next vertex distance could be minimized

dist[e] = dist[x] + w;

s.insert({dist[e], e} ); // insert the next vertex with the updated distance

}

}

}

}

Floyd–Warshall's Algorithm

Complejidad temporal O(V^2)

for(int k = 1; k <= n; k++){

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

for(int j = 1; j <= n; j++){

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

}

}

}

(HackerEarthShortesPathAlgorithms, 2018)

* Interfaz

