ANÁLISIS Y DISEÑO DE ALGORITMOS II

1er Cuatrimestre 2021

Trabajo Práctico Especial nº 3

Algoritmo de aproximación

Alumno: Bricio Vellaz Barbieri. Email: briciovellaz@gmail.com

1. Resumen

El siguiente informe pertenece al trabajo practico de laboratorio nº 3 de la cátedra Análisis y diseño de algoritmos II.

El objetivo es implementar un algoritmo de aproximación para resolver el problema de ubicación de estaciones de bombero en una ciudad, respetando las restricciones de complejidad algorítmica propuestas en el enunciado del problema.

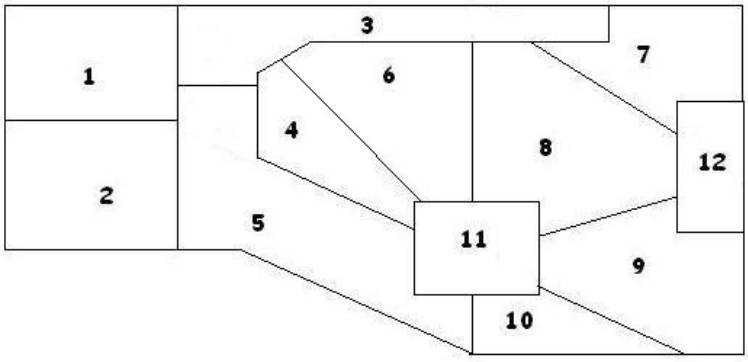
2. Objetivo y descripción del problema

Enunciado del problema:

En el planeamiento urbano de una ciudad, se desean ubicar estratégicamente las estaciones de bomberos. Cada estación tendrá la capacidad de cubrir las emergencias tanto de su barrio como de los barrios adyacentes, por ejemplo, una estación construida en el barrio 1 (ver Figura) podrá hacerse cargo de las emergencias de todo su vecindario, es decir, del barrio 1, barrio 2, barrio 3 y barrio 5.

Se necesitan construir tantas estaciones de bomberos como sea necesario para atender todos los barrios ante posibles emergencias, cuidando que todos los barrios estén atendidos por al menos una estación (una o más), minimizando el número de estaciones construidas.

Mapa de la ciudad a modo de ejemplo:



Este problema es del tipo set covering (cobertura de conjuntos) y se basa en la necesidad que tiene una ciudad de instalar nuevas estaciones de bomberos que cubran

todos los barrios de la ciudad. Cada estación puede cubrir su propio barrio y todos los barrios limítrofes. Un barrio puede estar cubierto por una o más estaciones.

Además, el algoritmo debe obtener soluciones válidas. Se considera una solución válida aquella que su complejidad no supere el orden polinómico.

3. Diseño

Para la implementación del algoritmo se modela la ciudad como un grafo, siendo cada barrio un nodo y las aristas sus conexiones. Para este fin se utiliza la clase grafo implementada en el primer trabajo práctico de laboratorio.

El algoritmo se basa en la técnica de programación Greedy y funciona de la siguiente manera:

1. Se crea una lista con todos los barrios restantes por cubrir (inicialmente contiene a todos los barrios), luego se crea una lista con todos los barrios y su cantidad de adyacentes ordenados de mayor a menor cantidad. Además, se crea un vector/lista/conjunto donde se guardarán los barrios para la solución final.

2. Se agrega a la solución el primer barrio de esa lista y se elimina del grafo, lista de vértices restantes y vector de barrios el barrio agregado a solución. También se eliminan todos sus adyacentes.

3. Se actualiza el vector barrios con la nueva cantidad de barrios restantes sin alcanzar y se vuelve al paso 2.

4. Mientras que la lista de barrios restantes no esté vacía, se repetirán los pasos 2 y 3.

5. Cuando no hay más vértices por cubrir, se muestra la solución y el algoritmo termina.

4. Implementación

En el primer paso se declaran los contenedores que se utilizaran:

```
vector<int> solucion;
list<int> vertices;
vector<pair<int,int>> barrios;
grafo.devolver_vertices(vertices);
```

Se crea también una lista de vértices para utilizar la función Devolver_Vertices que posee la clase Grafo. Esta lista se utilizará para almacenar los barrios sin cubrir.

Luego se llama a la función ActualizarAdyacentes, la cual se encarga de agregar en una lista cada barrio y su cantidad de adyacentes que aún queda por cubrir. Una vez agregados, ordena esa lista.

```
bool OrdenPorSeg(const pair<int,int> &a,
              const pair<int,int> &b)
{
    return (a.second > b.second);
}

void OrdenarDec(vector<pair<int,int>> &lista){
    sort(lista.begin(), lista.end(), OrdenPorSeg);
}

void ActualizarAdyacentes(vector<pair<int,int>> &barrios,Grafo<int> grafo,list<int> vertices){
            for(auto it=vertices.begin();it!=vertices.end();it++){
                barrios.push_back(make_pair(*it,grafo.cantidad_adyacentes(*it)));
            }

            OrdenarDec(barrios);
}
```

Luego comienzan los pasos 2 y 3.

Agregando en cada paso el barrio con más barrios vecinos sin cubrir. Cuando se agrega un nuevo barrio a la solución, se eliminan los barrios alcanzados para no ser tenidos en cuenta nuevamente y vuelve a llamar la función ActualizarAdyacentes.

```
   auto it=barrios.begin();
   while(!barrios.empty()){

        solucion.push_back(it->first);
        vertices.remove(it->first);
        list<pair<int,int>> adyacentes;
        grafo.devolver_adyacentes(it->first,adyacentes);
        grafo.eliminar_vertice(it->first);
        for(auto itAdy=adyacentes.begin();itAdy!=adyacentes.end();itAdy++){
            vertices.remove(itAdy->first);
            grafo.eliminar_vertice(itAdy->first);
        }
        barrios.clear();
        ActualizarAdyacentes(barrios,grafo,vertices);

   };
```

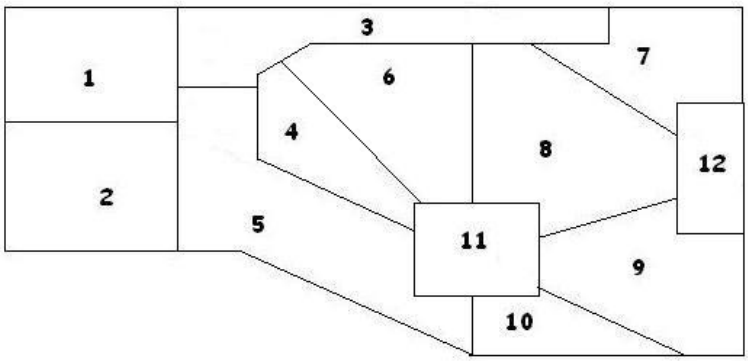
Una vez que no quedan barrios restantes por cubrir, el algoritmo termina e imprime la solución obtenida por pantalla.

El algoritmo completo sería de la forma:

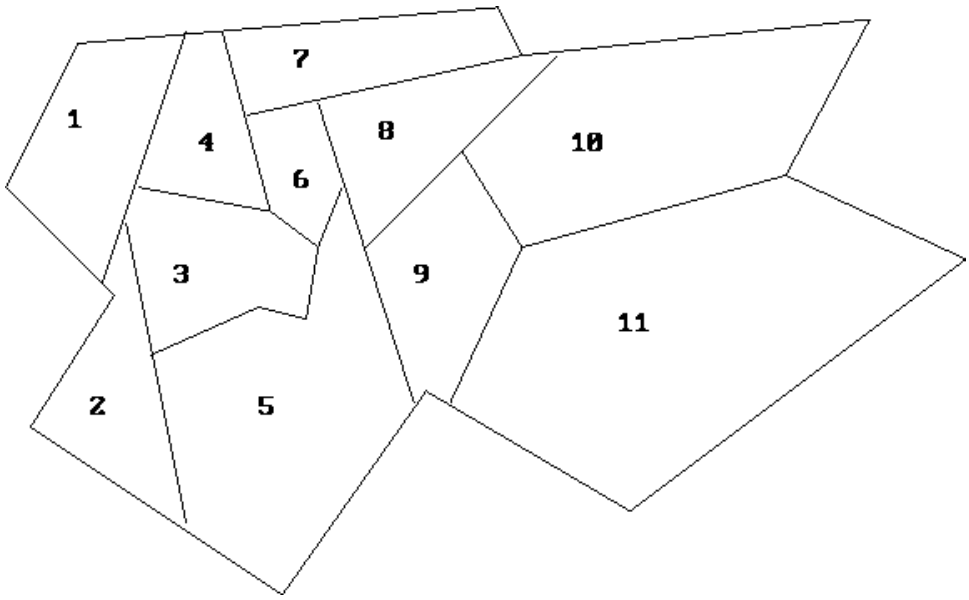
```
void CoberturaBomberos(Grafo<int> grafo){

    vector<int> solucion;
    list<int> vertices;
    vector<pair<int,int>> barrios;
    grafo.devolver_vertices(vertices);

    ActualizarAdyacentes(barrios,grafo,vertices);

    auto it=barrios.begin();
    while(!barrios.empty()){

        solucion.push_back(it->first);
        vertices.remove(it->first);
        list<pair<int,int>> adyacentes;
        grafo.devolver_adyacentes(it->first,adyacentes);
        grafo.eliminar_vertice(it->first);
        for(auto itAdy=adyacentes.begin();itAdy!=adyacentes.end();itAdy++){
            vertices.remove(itAdy->first);
            grafo.eliminar_vertice(itAdy->first);
        }
        barrios.clear();
        ActualizarAdyacentes(barrios,grafo,vertices);

    };

    cout<<"Solucion posible: ";
    for(auto it=solucion.begin();it!=solucion.end();it++){
        cout<< *it<<"  ";
    }
}
```

El algoritmo tiene una complejidad perteneciente a O(b^4) (siendo b la cantidad de barrios) ya que el ciclo while se repite b veces y dentro de él se tiene un ciclo for que se repite hasta b veces e invoca la función Eliminar_Vertice de la clase grafo, la cual tiene complejidad O(b^2)

5. Pruebas de ejecución



Para el ejemplo dado en el enunciado, el algoritmo devuelve como solución los vértices 3, 9 y 2. Siendo una solución de tamaño 3, que se aproxima al tamaño de la solución óptima para este ejemplo. La solución óptima son los barrios 5 y 8 y es una solución de tamaño 2.



Para este ejemplo, el algoritmo devuelve como solución los barrios 3, 9 y 7. Lo cual es una solución óptima debido a que 3 es la cantidad mínima para cubrir esta ciudad.

En conclusión, el algoritmo funciona de forma deseada al devolver una solución aproximada o la óptima en algunos casos.