# Práctica 3: Algoritmos Voraces (Greedy)

Irene Huertas González
Javier Alcántara García
César Muñoz Reinoso

# Índice

# Objetivo de la practica

El objetivo de esta práctica es apreciar la utilidad de los métodos voraces para resolver problemas de forma muy eficiente obteniendo soluciones óptimas en algunos casos y aproximaciones en otros.
Para ello se nos ha pedido que desarrollemos dos problema usando algoritmos voraces, además en el primero veremos distintos enfoques que nos darás distintas soluciones válidas pero distintas entre ellas.

# Problema comun: El problema del viajante de comercio

**Enunciado**

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto departida, de forma tal que la distancia recorrida sea mínima. Mas formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

# TSP - Enfoque basado en Cercanía

**¿En qué consiste?** Dada una ciudad inicial, se agrega como siguiente aquella que aun no se haya visitado que esté más cercana a esta. Esto se repite hasta visitarlas todas.
La distancia entre una ciudad y otra se calcula con la fórmula de la distancia euclídea.

# TSP - Enfoque basado en Cercanía

**Implementación**

- Tenemos una matriz en la cual se han metido los valores de las coordenadas de las ciudades leídas desde el fichero .tsp
- Creamos la matriz que contiene las distancias que hay de una ciudad a otra. Explicación: el elemento de la posicion (1,3) es la distancia que hay desde la ciudad 1 a la ciudad 3, que es igual al elemento que encontramos en la posición (3,1). La diagonal principal de la matriz está compuesta de ceros ya que contiene la distancia entre una ciudad consigo misma.

# TSP - Enfoque basado en Cercanía

```
int distancias[dimension][dimension] = {0}; //Matriz con la distancia entre ciudades

for (int i = 0; i < dimension; i++){
  for(int j = 0; j < dimension; j++){
    distancias[i][j] = distancia_euclidea(matriz[j][0], matriz[i][0],
                                          matriz[j][1], matriz[i][1]);
  }
}
```

- Se hace llamamiento a la función que calcula la distancia euclídea entre dos ciudades.

# TSP - Enfoque basado en Cercanía

- La función recibe como parámetros las coordenadas x e y de dos ciudades distintas, calcula la raiz cuadrada de la suma de las diferencias al cuadrado y aproxima el resultado al entero más próximo

```
int distancia_euclidea(int x2, int x1, int y2, int y1){
  double d = sqrt(((x2-x1)*(x2-x1)) + ((y2-y1)*(y2-y1)));
  return round(d);
}
```

# TSP - Enfoque basado en Cercanía

- Seleccionamos como ciudad inicial la primera que hayamos leído.
- Nos vamos a la fila de la matriz de distancias que se corresponde con el número de la ciudad (ej: ciudad 1 = primera fila).
- Recorremos la fila, buscamos el menor entero (la distancia más pequeña a otra ciudad) y nos quedamos con el número de columna
- Comprobamos en el vector de orden que no hemos estado ya allí y la insertamos
- saltamos a la fila de la matriz distancias correspondiente a la ciudad seleccionada y volvemos a realizar los mismos pasos hasta recorrerlas todas.
- Al final del todo se vuelve a meter la primera ciudad para cerrar el ciclo

# TSP - Enfoque basado en Cercanía

```
orden[0] = ciudad; //Se pone como primera ciudad la primera


for(int i=1; i < dimension; i++){
  min=9000;
  for(int j = 0; j < dimension; j++){
    if((distancias[ciudad-1][j] != 0) && (distancias[ciudad-1][j] < min)){
      for(int k=0; k < dimension; k++){
        if (j+1 == orden[k])
          repe = true;
      }
      if(!repe){
        min = distancias[ciudad-1][j];
        ciudad_min = j;
      }
      else
      repe = false;
    }
  }

  orden[i] = ciudad_min+1;
  ciudad = orden[i] -1;
}

orden[dimension]=1; //Cerramos el circulo
```

# TSP - Enfoque basado en Inserción

**¿En qué consiste?**

Primero creamos una matriz con la distancias entre las ciudades, utilizando el algoritmo de Euclides. Comenzamos seleccionando la ciudad inicial como la 1. Recorremos la fila 1 y seleccionamos la ciudad a menor distancia (columnas). Cuando hacemos esto saltamos a la fila de la ciudad seleccionada y volvemos a realizar este procedimiento. Tenemos en cuenta que no podemos introducir ciudades repetidas. Cuando haya introducido de la lista todas las ciudades sin repeticiones, añadimos a la distancia final el camino entre la última ciudad y la primera, e insertamos de nuevo la ciudad 1 (única que se repite).

```
list<int> TSP(const vector<vector<int> > &matriz,
const map<int, pair<double, double> >  &m, int &distancia_final) {
    int n=1;
    int s=1;
    int w=1;

    list<int> salida;
    map<int,pair<double,double> >::const_iterator it  = m.begin();
    list<int> ciudades_restantes;
    ciudades_restantes.push_back(1);

    //En cada par estan las coordenadas de la primera ciudad
    pair<double,double> pn=(*it).second, ps=(*it).second, pw=(*it).second;
    it++;

    //Escogemos las ciudades mas al norte, sur y oeste
    for(it; it != m.end(); it++) {
        ciudades_restantes.push_back((*it).first);
        pair<double,double> p=(*it).second;
        int ciudad_actual=(*it).first;

        //Tiene que haber elses ya que debe escoger ciudades distintas (la de mas al norte
        // y mas al oeste no pueden ser la misma)
        if(p.first<pw.first) { //Elige la de mas al oeste
            pw=p;
            w=ciudad_actual;
        }
        else if(p.second>pn.second || (n==w && n==1)) { //La de mas al norte
            pn=p;
            n=ciudad_actual;
        }
        else if(p.second<ps.second || (s==w && s==1)) { //La de mas al sur
            ps=p;
            s=ciudad_actual;
        }
    }
```

```
    //n,s,w forman el triangulo mas grande
    //suponemos que empezamos en n, una de las ciudades del triangulo

    //Mete las tres en la solucion
    salida.push_back(n);
    salida.push_back(s);
    salida.push_back(w);

    //Las elimina de las reestantes
    ciudades_restantes.remove(n);
    ciudades_restantes.remove(s);
    ciudades_restantes.remove(w);

    while(ciudades_restantes.size() > 0) {
        pair<int,list<int>::iterator> pres;
        pres = ElegirCiudad(salida, matriz, ciudades_restantes);
        InsertarCiudad(salida, pres.first, pres.second);
    }

    salida.push_back(*(salida.begin())); //volvemos a la primera ciudad
    distancia_final = DistanciaTotal(salida, matriz);

    return salida;
}
```

# TSP - Enfoque basado en Arista

**¿En qué consiste?**
Creamos un mapa formado por la distancia entre dos ciudades y un par con esas dos ciudades. Vamos seleccionando las distancias menores entre dos ciudades y añadiendo estas al conjunto de seleccionados. Para poder insertar una nueva ciudad, esta no puede tener más de 2 aristas ni crear un ciclo. Cómo el camino resultante de este algoritmo no es cerrado, tenemos que recorrer el camino y añadir la distancia entre las 2 ciudades que únicamente tienen una arista.

```
vector<int> TSP(multimap<int, pair<int,int> > & aristas_par_ciudades, int & distancia) {
    vector<int> salida;
    list<pair<int,int> > aux;
    multimap<int , pair<int,int> >::iterator it = aristas_par_ciudades.begin();

    //Mete la primera arista
    aux.push_back((*it).second);
    //añade su distancia
    distancia+=(*it).first;
    it++;
    while(it != aristas_par_ciudades.end()) {
        //Si no hay ciclos y la nueva arista no provoca que un vertice tenga mas de dos aristas
        //se mete en la lista
        if(CiudadFactible(aux, (*it).second) && !ExistenCiclos(aux, (*it).second)) {
            aux.push_back((*it).second);
            distancia += (*it).first;
        }
        it++;
    }
    //Se cierra el ciclo (conectar ultimo con el primero)
    CerrarCiclo(aux, aristas_par_ciudades, distancia);
    //Se obtiene el camino creado
    salida = TransformaACamino(aux);

    return salida;
}
```

# TSP - Comparación de las tres estrategias

**Visualización recorridos**

Ejemplo de ejecución del fichero att48.tsp

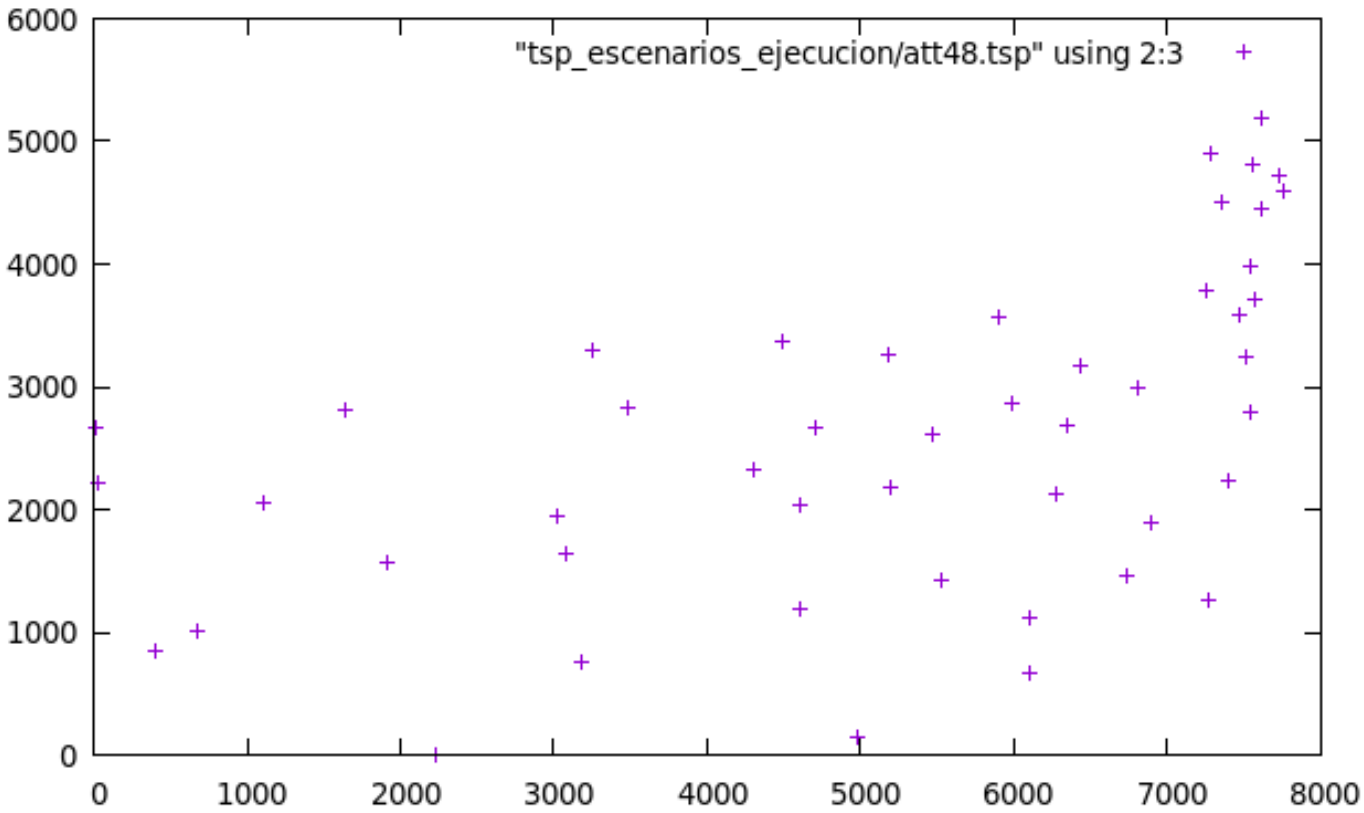
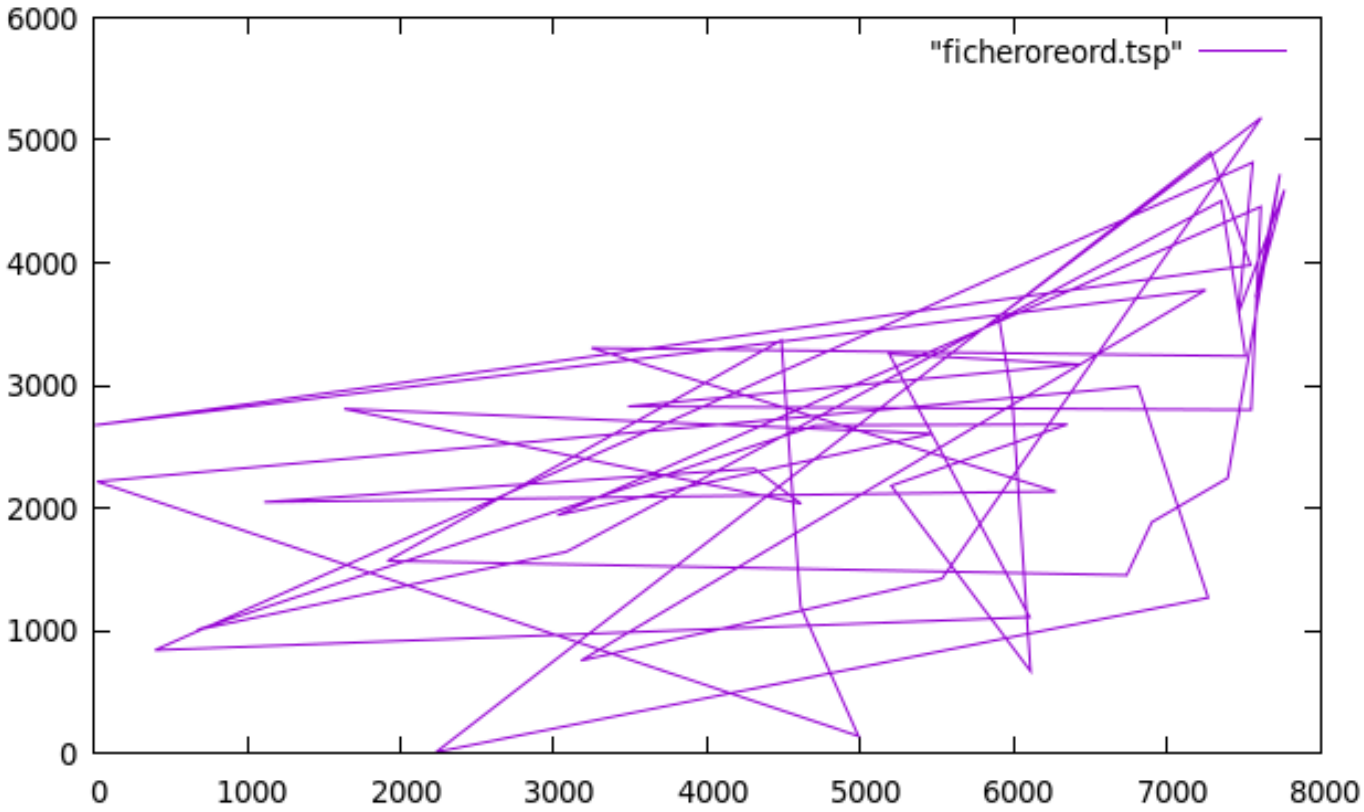



Figura: Disposición de las ciudades en el plano

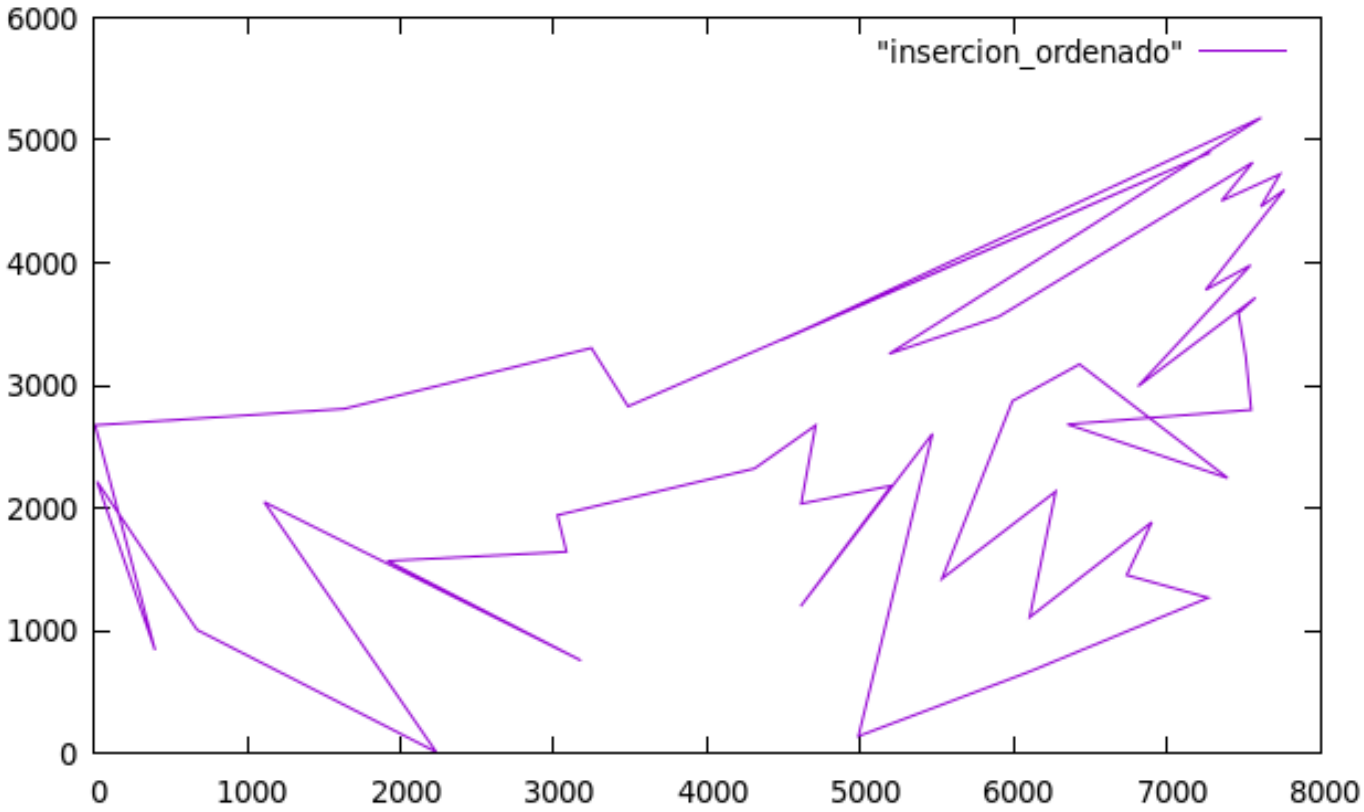
# TSP - Comparación de las tres estrategias

Recorrido enfoque basado en cercanía



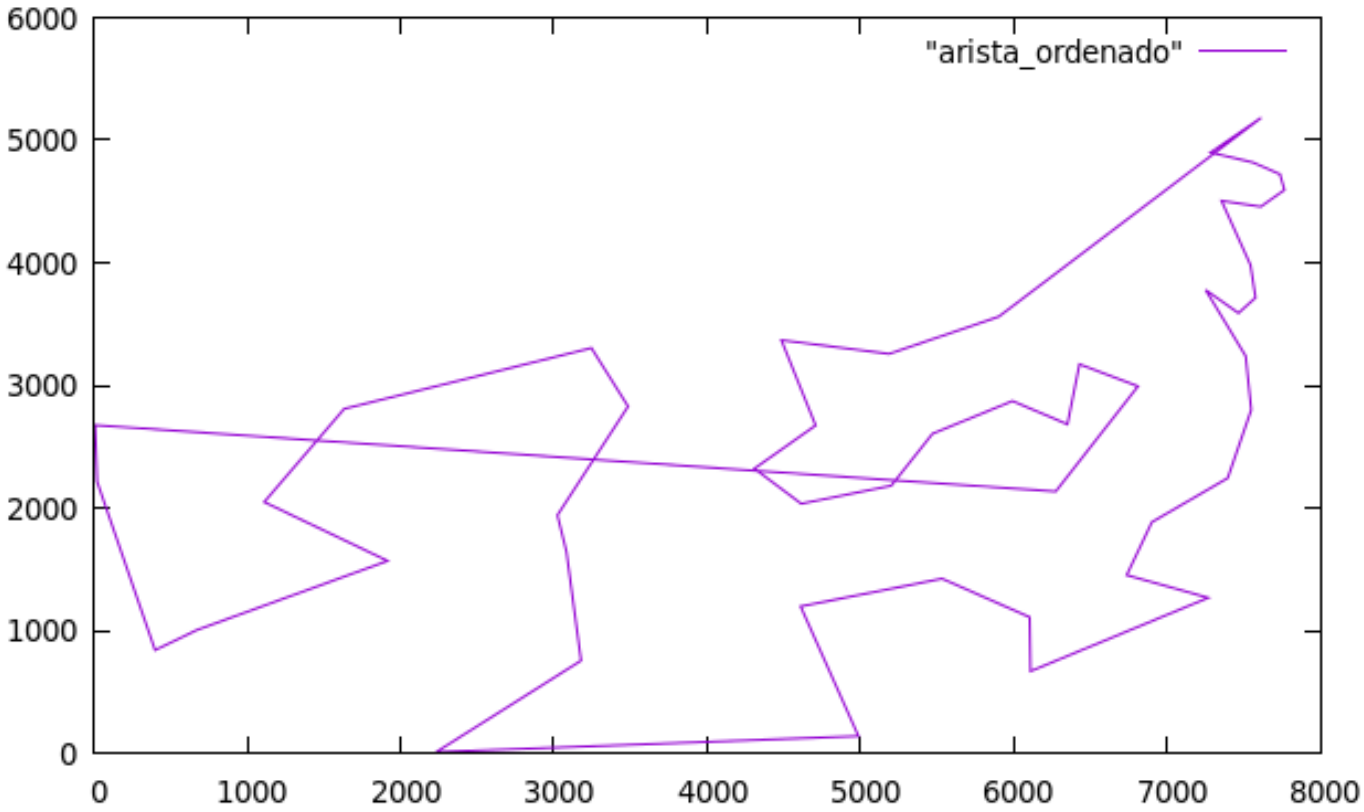
# TSP - Comparación de las tres estrategias

Recorrido enfoque basado en inserción



# TSP - Comparación de las tres estrategias

Recorrido enfoque basado en arista



# TSP - Comparación de las tres estrategias

**Longitud Ciclos**

Hemos calculado la longitud de los ciclos previos y nos han salido los siguientes valores:

| | |
|---|---|
| Cercanía | 161208 |
| Inserción | 64134 |
| Arista | 48229 |

Por lo tanto podemos deducir que el mejor enfoque en este caso es el de Arista porque nos sale una distancia recorrida más pequeña.

# Problema asignado: Asignación de tareas

**Enunciado** Tenemos que completar un conjunto de n tareas con plazos limite. Cada una de las tareas consume la misma cantidad de tiempo (una unidad) y, en un instante determinado, podemos realizar únicamente una tarea. La tarea i tiene como plazo limite d i y produce un beneficio gi (g i > 0) solo si la tarea se realiza en un instante de tiempo t <= d i . Diseñe un algoritmo voraz que nos permita seleccionar el conjunto de tareas que nos asegure el mayor beneficio posible.

# Problema asignado: Asignación de tareas

**Algoritmo** Se ejecutan varias funciones en un bucle while

```
while(!func_sol(v, n)){
  int obj = func_seleccion(v, sol, asig, n);
  if (!func_fact(sol,n)){
    sol[asig-1] = 0;
    asig--;
  }
}
```

# Problema asignado: Asignación de tareas

**Algoritmo:** El algoritmo de selección de tarea:

```
int func_seleccion(int **v, int *sol, int &asig, int n){
  int objetivo = 0;
  int menor,mayor;
  menor = v[0][1];
  mayor = v[0][2];

  for(int i = 1 ; i < n ;i++){
//  Igual duracion pero con mayor beneficio
    if (v[i][1] == menor && v[i][1] > asig){
      if (v[i][2] > mayor){
        if(menor == asig)
          v[objetivo][0] = -1;
        menor = v[i][1];
        objetivo= i;
        mayor = v[i][2];
      }
      else if (v[i][1] <= asig){
          v[i][0] = -1;


      }
    }// Menor duracion y dentro cronologicamente
```

# Problema asignado: Asignación de tareas

```
    else if (v[i][1] < menor && v[i][1] > asig){
        menor = v[i][1];
        objetivo= i;
        mayor = v[i][2];
    }// Anterior usado y usamos este
    else if (v[objetivo][0] == -1 && v[i][0] != -1 && v[i][1] > asig){
      menor = v[i][1];
      mayor = v[i][2];
      objetivo = i;
    }
  }

  sol[asig] = v[objetivo][2];
  asig++;

  v[objetivo][0] = -1;


  return objetivo;
}
```