Greedy

PRÁCTICA 2 ALGORÍTMICA

2017

Jose Antonio Ruiz Millán — jantonioruiz@correo.ugr.es
Alejandro Rodríguez Muñoz — aromu@correo.ugr.es
Julio Antonio Fresneda García — juliofresnedag@correo.ugr.es
Adrián Peláez Vegas — adrianpelaez@correo.ugr.es

GRUPO B3

Greedy

Contenido

Introducción

Esta es la memoria de la practica 2 de la asignatura de algorítmica. En ella se describe la solución de los 4 ejercicios escogidos con la siguiente estructura: introducción al problema, esquema general greedy, descripción de la solución, código implementado y conclusión. El lenguaje escogido para implementar las soluciones es c++.

1. Contenedores de un barco

1.1. Introducción al problema

Tenemos un barco que puede almacenar K toneladas y un conjunto de contenedores que se pueden almacenar en dicho barco. Cada contenedor tendrá un peso determinado y la suma total de los pesos supera la capacidad total del barco(k).

El ejercicio nos pide dos cosas:

a) Implementar un algoritmo que maximice el número de toneladas a cargar en el barco.

b) Implementar un algoritmo que maximice el número de contenedores a cargar en el barco.

1.2 Esquema general Greedy

Maximizar el Nº de toneladas cargadas

-Conjunto de candidatos: contenedores disponibles para ser cargados.

-Candidatos ya usados: contenedores que ya han sido cargados en el barco.

-Función solución: Lista de candidatos tal que la diferencia entre el peso de estos y la capacidad total del barco no sea suficiente para albergar ningún otro contenedor de la lista de candidatos.

-**Criterio factible:** La suma del peso de los contenedores cargados es menor o igual que la capacidad del barco.

-Función selección: Seleccionamos el contenedor de mayor peso.

-Función objetivo: Maximizar el peso de los contenedores elegidos sin superar la capacidad del barco.

Maximizar el Nº de contenedores cargados

-Conjunto de candidatos: Contenedores disponibles para ser cargados.

-Candidatos ya usados: Contenedores que ya han sido cargados en el barco.

-Función solución: Lista de candidatos tal que la diferencia entre el peso de estos y la capacidad total del barco no sea suficiente para albergar ningún otro contenedor de la lista de candidatos.

-Criterio factible: La suma del peso de los contenedores cargados es menor o igual que la capacidad del barco.

-Función selección: Seleccionamos el contenedor de menor peso.

-Función objetivo: Maximizar el nº de contenedores elegidos sin superar la capacidad del barco.

1.3. Descripción de la solución

Para resolver este problema nos hemos abstraído de problemas superfluos como podría ser la ordenación del vector de contenedores (ordenado de mayor a menor peso cuando queremos maximizar el número de toneladas cargadas y de menor a mayor peso cuando queremos maximizar el número de contenedores cargados). Esto se resolvería pasándole al vector de contenedores un algoritmo de ordenación previo a los algoritmos de solución. En nuestro caso, ya guardamos los valores ordenados como nos conviene para su resolución. Una vez aclarado esto pasemos a ver la resolución en cada uno de los casos

Maximizar el Nº de toneladas cargadas

Hemos creado una función a la que le pasamos como parámetros el conjunto de contenedores, un conjunto vacío donde se guardara la solución y un entero para indicar la carga máxima del barco. El algoritmo se inicia con un bucle encargado de recorrer el conjunto de los contenedores. Si el peso del contenedor excede la carga soportada por el barco y siguen quedando contenedores por revisar, pasamos a comprobar el siguiente contenedor (que será de menor peso). Si no se cumple esta primera condición pasamos a comprobar otra que es que el peso del contenedor sea aceptable por el barco y sigan quedando contenedores. En este caso se añade el contenedor al conjunto solución, se ajusta el nuevo peso total del barco y se comprueba el siguiente. Si no se cumpliese esta última condición se terminaría el bucle ya que el peso total del barco está agotado o que no quedan contenedores que comprobar.

Maximizar el Nº de contenedores cargados

A esta función le pasamos como parámetros los mismos que a la anterior, ya que, es un problema semejante y solo hay cambia el valor que optimizar. Empieza con un bucle para ir recorriendo el conjunto de contenedores y mientras pueda soportar peso el barco seguirá introduciendo contenedores desde el menos hasta el más pesado. Así, al dar prioridad a los menos pesados, se consigue introducir más contenedores.

1.4. Código

Función maximizaToneladas:

```
26  void maximizaToneladas(vector < pair<int, int> > &v, vector < pair<int, int> > &s, int k)
27  {
28      bool continua = true;
29      int i = 0;
30
31      while(continua){
32          if(v[i].second > k && i < v.size()){
33              i++;
34          }
35          else if(k-v[i].second >= 0 && i < v.size()){
36              s.push_back(v[i]);
37              k -= v[i].second;
38              i++;
39          }
40          else{
41              continua = false;
42          }
43      }
44  }
```

Función maximizaContenedores:

```
9   void maximizaContenedores(vector < pair<int, int> > &v, vector < pair<int, int> > &s, int k)
10  {
11      bool continua = true;
12      int i = 0;
13
14      while(continua){
15          if(k-v[i].second >= 0){
16              s.push_back(v[i]);
17              k -= v[i].second;
18              i++;
19          }
20          else{
21              continua = false;
22          }
23      }
24  }
```

1.5. Conclusión

La resolución de este ejercicio no nos supuso un problema. Tras analizar el problema, pudimos hallar un modo de actuación rápido y lo implementamos aunque con esta metodología no de la solución óptima en uno de los casos.

Al poner diversos ejemplos se puede comprobar a simple vista que el algoritmo que se encarga de maximizar el número de contenedores es óptimo. Siempre dará la mejor solución porque al ir sumando desde el contenedor que pesa menos, en el momento que no quepa un contenedor sabrás que no podrán entrar ninguno de los que quedan al ser mayores. Si a esto sumamos que el tiempo de cómputo para la ejecución de este algoritmo es apenas inexistente, nos encontramos ante un algoritmo que cumple con creces su propósito.

Por contrapartida, el algoritmo encargado de maximizar el número de toneladas no devuelve siempre la solución óptima. Es fácil encontrar un ejemplo donde la suma de dos contenedores que sean menores puede sumar más que un contenedor mayor y dejar menos huecos en el barco. Aun así, debido a su mínimo tiempo de ejecución, este algoritmo puede ser útil en aquellos entornos donde el tiempo para encontrar una respuesta (aunque no sea la óptima) sea una restricción fuerte. En cambio, si el tiempo no es problema y necesitamos una respuesta optima lo recomendable seria recurrir a otros algoritmos que si la dan.

2. Minimizar el número de visitas al proveedor

2.1. Introducción al problema

Tenemos un granjero que debe tener un fertilizante. La cantidad de fertilizante la mediremos en días(R). No puede faltarle nunca fertilizante. El problema está en minimizar el número de veces que tiene que ir al proveedor para comprar, sabiendo que dicho proveedor no tiene un horario regular pero que conocemos a priori.

2.2. Esquema general greedy

-Conjunto de candidatos: Días que abre la tienda.

-Candidatos ya usados: Días que ha visitado la tienda.

-Función solución: Una lista de días en los que visita la tienda será solución siempre que la diferencia entre el último día que visita la tienda y el final del periodo de tiempo elegido sea menor que r (nº de días de utilización del fertilizante).

-Criterio factible: Que el tiempo entre el último día de visita a la tienda y el próximo día de apertura sea menor que r.

-Función selección: Día más lejano al último día que fue a la tienda sin sobrepasar r.

-Función objetivo: Visitar la tienda el menor número de días posible.

2.3. Descripción de la solución

Hemos creado un método llamado *visitas_proveedor_greedy* que devuelve una lista de enteros que corresponden a los números de día que el granjero tiene que comprar fertilizante. Se pasan 3 parámetros al método. *Tienda_abierta* que contiene los días en que está abierta la tienda para ir a comprar. *Dias_que_aguanto* que es un entero donde se almacena la cantidad de días para los que el granjero tiene fertilizante. Y *días_totales* que es otro entero donde se almacena la longitud del problema. Sirve para establecer un tiempo límite.

Una vez explicadas las variables pasemos al funcionamiento del algoritmo. Nada más entrar añade al conjunto de soluciones el primer día que hay en tienda_abierta porque el granjero empieza sin fertilizante y tiene que comprar. Después hay un bucle encargado de recorrer todos los días de tienda_abierta y dentro de este, otro bucle que en el que se comprueba los días cual es el día más lejano dentro del conjunto de días posibles antes de que se le termine. En caso de no poder aguantar con el fertilizante que le queda ira a comprar, aunque le quede bastante. Los días que tenga que ir se añadirán al conjunto de soluciones del problema.

2.4. Código

```
list<int> visitas_proveedor_greedy( const list<int> &tienda_abierta, int dias_que_aguanto, int dias_totales ){
  // Éste método devuelve una lista de enteros que corresponden a los números de día que el granjero tiene que comprar fertilizante
  // El algoritmo empieza desde el primer día que abre la tienda. (tienda_abierta contiene el 0)

  // tienda_abierta es una lista de enteros que corresponden a los números de día que la tienda abre.
  // dias_que_aguanto es un entero que corresponde con el número de días que el granjero aguanta sin fertilizante
  // días totales es el número total de días para el que cual se quiere programar las visitas_proveedor_greedy


  // False significa que ese día la tienda no abre. True que sí lo hace.

  // Primero inicializamos a false todos los días
  vector<bool> dias;
  for( int i = 0; i < dias_totales; i++ ){
    dias.push_back(false);
  }

  // Ponemos como true los días que la tienda sí abre.
  // Se presupone que la distancia entre los elementos true es menor o igual a dias_que_aguanto
  for( list<int>::const_iterator it = tienda_abierta.begin(); it != tienda_abierta.end(); ++it ){
    dias[*it] = true;
  }

  // Solucionamos el problema mediante un algoritmo greedy
  list<int> solucion;
  bool fin = false;
  int i = 0;
  int j;
  int dia_sol;
  solucion.push_back(i);

  while( !fin ){

    for( j = i+1; j <= dias_que_aguanto+i && j < dias_totales; j++ ){
      if( dias[j] == true ) dia_sol = j;
      if( j == dias_totales-1 ) fin = true;
    }

    i = dia_sol;
    solucion.push_back(dia_sol);
  }

  return solucion;
}
```

2.5. Conclusión

Este ejercicio nos supuso un problema al principio al no saber cómo plantearlo, pero una vez decididas las variables necesarias empezamos a comprenderlo mejor.

Al poner distintos ejemplos se puede ver con claridad que este algoritmo es óptimo ya que el problema se acaba reduciendo a escoger el día más lejano dentro de los posibles. De esta forma es imposible hallar una solución mejor con ningún algoritmo.

3. Recubrimiento de un grafo no dirigido

3.1. Introducción al problema

En este problema se nos da un grafo y se nos pide encontrar un conjunto de vértices mínimo desde los cuales se pueda llegar a todos los demás vértices del grafo. A esto se llama recubrimiento minimal del grafo.

El ejercicio nos pide dos cosas:

a) Diseñar un algoritmo para hallar el recubrimiento minimal de un grafo.

b) Diseñar un algoritmo para hallar el recubrimiento minimal de un grafo árbol.

3.2. Esquema general greedy

Grafos.

-Conjunto candidatos: Nodos.

-Candidatos ya usados: Nodos escogidos.

-Función solución: Tendremos una solución cuando tanto los nodos de la solución como los nodos adyacentes a éstos sean igual al total de nodos del grafo.

-Criterio factible: Cualquier conjunto de candidatos es factible.

-Función selección: Elegir el nodo, de entre los nodos candidatos, el nodo cuyo nº de vértices sea el mayor. En caso de empate, se coge aleatoriamente. Sólo se coge si aporta algo a la solución.

-Función objetivo: Minimizar el nº de nodos que expanden el grafo.

Árboles.

-Conjunto candidatos: Nodos.

-Candidatos ya usados: El nodo padre de las hojas.

-Función solución: Tendremos una solución cuando tanto los nodos de la solución como los nodos adyacentes a éstos sean igual al total de nodos del árbol.

Criterio factible: Cualquier conjunto de candidatos es factible.

-Función selección: Se elige, cada vez, un padre de un nodo hoja. Si tanto el padre como el "abuelo" solo tienen dos nodos adyacentes, los tres nodos se borran del grafo

-Función objetivo: Minimizar el nº de nodos que expanden el grafo.

3.3. Descripción de la solución

En ambos casos hemos optado por distintas representaciones para hacer el problema más. Dependiendo del problema, cada una de estas representaciones, tiene atributos distintos para adaptarse a nuestras necesidades.

Grafos

En este caso tenemos 3 listas. Una llamada *candidatos* donde se almacenarán todos los vértices del grafo y que están ordenados de mayor a menor por el número de aristas que tienen. Otra llamada *solución* en la que se almacenaran los vértices que sean una posible solución. Y una última llamada *cargados* donde se irán almacenando los vértices adyacentes a los vértices solución y que al concluir el problema tendrá todos los vértices que tenía *candidatos* al inicio.

Después de cargar los datos, lo primero que hacemos es un bucle que comprueba si la lista *cargados* está llena completa, ya que como hemos dicho antes, al estar llena quiere decir que todos los vértices sin visibles y termina el problema. Dentro, hemos puesto otro bucle encargado de recorrer la lista *candidatos*. En cada una de las iteraciones de este segundo bucle se comprueba si el vértice candidato añade nuevos vértices a la lista cargados. En caso afirmativo se añade dicho vértice a la lista solución y se comprueba el siguiente. En caso negativo no se añadiría a solución porque sus vértices adyacentes ya pueden ser accedidos desde otro vértice.

A este algoritmo le encontramos una mejora que consiste en ordenar la lista de candidatos de otra forma. Añadiendo al principio de la lista los nodos adyacentes a los nodos que tienen 1 sola arista. De este modo devuelve la solución óptima en más casos.

Árbol

Para este problema hemos creado una clase nodo propia que almacenara el valor de un vértice, un puntero a su padre, el número de aristas y el número de hijos que tiene. Además, tenemos dos listas. *Candidatos* que almacenará todos los vértices del árbol y *solución* en la que se guardaran los vértices que crean una posible solución. La lista de *candidatos* estará ordenada de menor a mayor por el número de hijos.

Después de llenar la lista con los nodos del árbol, comenzamos con un bucle que comprueba si la lista de candidatos esta vacía o si hemos llegado al nodo raíz. En caso afirmativo finalizaría el algoritmo. Dentro del bucle, se coge el primer nodo de *candidatos* y añadimos a su padre a la lista de soluciones. Acto seguido se pregunta dicho nodo padre si tiene más de un hijo. En caso negativo se eliminaría el nodo seleccionado y su padre por no tener ningún hijo más. También comprobamos si el nodo abuelo (padre del padre) tiene un solo hijo y en caso afirmativo también borramos al abuelo. Así nos aseguramos de tener que recorrer el mínimo numero de nodos. En el caso que el padre o el abuelo tuvieran más de un hijo, no se borrarían porque tendríamos que comprobarlos (podrían ser arboles no balanceados).

Repetiríamos el proceso anteriormente mencionado hasta que se dejasen de cumplir las dos condiciones del bucle.

3.4. Código

Grafo

```
66      while(cargados.size() < completo){
67        cargados.insert(candidatos.top().first);
68        int ant = cargados.size();
69        for(auto it = candidatos.top().second.begin(); it != candidatos.top().second.end(); ++it){
70          cargados.insert(*it);
71        }
72        int des = cargados.size();
73        if( ant != des ){
74          solucion.push_back(candidatos.top().first);
75        }
76        candidatos.pop();
77      }
78
79      cout << endl;
80      for(auto it = solucion.begin(); it != solucion.end(); ++it){
81        cout << *it << " ";
82      }
83      cout << endl;
84
85      return 0;
86    }
```

Grafo mejorado

```
list<Nodo> recubrimiento_minimo( list<Nodo> &grafo ){

  for( list<Nodo>::iterator it = grafo.begin(); it != grafo.end(); ++it ){
    if( it->adyacentes.size() == 1 ) {
      for( list<Nodo>::iterator it2 = grafo.begin(); it2 != grafo.end(); ++it2 ){
        if( *(it->adyacentes.begin() ) == it2->nombre) {  it2->prioridad = 100000000;
        cout << it2->nombre << " dopado" << it2->prioridad << endl;
        it2->prioridad = 10000;
      }
      }
    }

  }

  grafo.sort(comparar);


  for( list<Nodo>::iterator it = grafo.begin(); it != grafo.end(); ++it ){
    cout << it->nombre << "_" << it->prioridad << "___";
  }
  list<Nodo> solucion;
  int total = 0;

  list<Nodo>::iterator it = grafo.begin();
int tam = grafo.size();
  while( total < tam ){
    cout << total << "-" << tam << endl;
    solucion.push_back(*it);
    total+=(it->adyacentes.size());
    total++;


      for( list<char>::iterator itc = it->adyacentes.begin(); itc != it->adyacentes.end(); ++itc ){
        for( list<Nodo>::iterator itt = grafo.begin(); itt != grafo.end(); ++itt ){
          cout << *itc << " comparo con " << itt->nombre << endl;
          if( *itc == itt->nombre ){
            cout << "333" << *itc << "-" << itt->nombre << "--" << itt->prioridad << endl;
            itt->prioridad--;
          }
        }
      }


    grafo.erase(it);
    grafo.sort(comparar);
    it = grafo.begin();


  }

  return solucion;

}
```

Árbol

Clase nodo

```
7   class Nodo{
8   public:
9     Nodo(char v, Nodo *p, int h){
10      valor = v;
11      padre = p;
12      nhijos = h;
13    }
14    Nodo(const Nodo &n){
15      valor = n.getValor();
16      padre = n.getPadre();
17      nhijos = n.getHijos();
18    }
19    char getValor() const{
20      return valor;
21    }
22    Nodo* getPadre() const{
23      return padre;
24    }
25    int getHijos() const{
26      return nhijos;
27    }
28    void eliminarHijo(){
29      nhijos--;
30    }
31    bool operator <(const Nodo &otro) const{
32      return (this->nhijos <= otro.getHijos());
33    }
34    bool operator ==(const Nodo &otro) const{
35      return (valor == otro.valor && padre == otro.getPadre());
36    }
37  private:
38    char valor;
39    Nodo *padre;
40    int nlados;
41    int nhijos;
42  };
```

Algoritmo

```
70    while(candidatos.begin()->getPadre() != 0 && !candidatos.empty()){
71      solucion.insert(candidatos.begin()->getPadre()->getValor());
72      if( candidatos.begin()->getPadre()->getHijos() <= 1 ){
73        bool salir = false;
74        for( auto it = candidatos.begin(); it != candidatos.end() && !salir; ++it ){
75          if(candidatos.begin()->getPadre()->getValor() == it->getValor()){
76            p = it;
77            salir = true;
78          }
79        }
80        if(candidatos.begin()->getPadre() != 0 ){
81          if(candidatos.begin()->getPadre()->getPadre() != 0){
82            if(candidatos.begin()->getPadre()->getPadre()->getHijos() > 1){
83              candidatos.begin()->getPadre()->getPadre()->eliminarHijo();
84            }
85            else{
86              salir = false;
87              for( auto it = candidatos.begin(); it != candidatos.end() && !salir; ++it ){
88                if(candidatos.begin()->getPadre()->getPadre()->getValor() == it->getValor()){
89                  ab = it;
90                  salir = true;
91                }
92              }
93              candidatos.erase(ab);
94            }
95          }
96        }
97        candidatos.erase(p);
98      }
99      else{
100       candidatos.begin()->getPadre()->eliminarHijo();
101     }
102     candidatos.erase(candidatos.begin());
103     int tam = candidatos.size();
104     candidatos.swap(aux);
105     candidatos.clear();
106     for( int i = 0; i < tam; i++ ){
107       Nodo n = *(aux.begin());
108       aux.erase(aux.begin());
109       candidatos.insert(n);
110     }
111   }
```

3.5. Conclusión

Este ejercicio, sin duda, es el que más complicaciones nos ha dado. En principio no entendíamos del todo lo que pedía el problema, pero cuando lo comprendimos nos encontramos con una dificultad mayor porque lo que pareció un problema trivial, acabó teniendo más restricciones y comprobaciones de las que podíamos llegar a intuir en un principio. Tanto para el problema del grafo como el del árbol, implementamos varias

posibles soluciones que al final descartamos dejando solamente la que nos devolvía una solución más óptima. La mejora que encontramos del algoritmo muestra como en el enfoque greedy es muy importante ajustar bien las implementaciones porque el solo hecho de ordenar los elementos de otra forma consigue una mejora sustancial.

En el caso del árbol, podemos decir que el algoritmo devuelve el óptimo en todos los casos probados y no conseguimos encontrar un ejemplo que refute dicha afirmación. Para el grafo no se encontraba en todos los casos la solución óptima aunque hicimos varios ajustes para reducir lo máximo posible las soluciones no óptimas, quedándonos así, un algoritmo que cumplen bastante bien su cometido dentro de un enfoque greedy.

4. Reparaciones

4.1. Introducción al problema

En este problema se nos presenta un electricista que tiene una lista de trabajos que atender. Su empresa le paga según la satisfacción de los clientes y esta satisfacción es mayor cuanto menor sea el tiempo que tarda en atenderlos. Cada uno de los trabajos lleva asociado un tiempo. El problema pide maximizar la satisfacción de los clientes (minimizar el tiempo que se tarda en atenderlos) en caso de solo haber un electricista y para varios electricistas.

4.2. Esquema general greedy

-Conjunto de candidatos: Reparaciones que el electricista tiene pendientes.

-Candidatos ya usados: Reparaciones ya hechas.

-Función solución: Todas las reparaciones se hayan realizado.

-Criterio factible: Cualquier conjunto de candidatos es factible.

-Función selección: Reparación cuyo tiempo de realización sea el más corto.

-Función objetivo: Minimizar el tiempo de espera de los clientes.

4.3. Descripción de la solución

Para este ejercicio hemos creado una cola con prioridad (priority_queue) donde vamos volcando los trabajos que debe realizar el electricista. Esta cola, ordena de menor a mayor los trabajos según el tiempo que se tarda en realizarlos. En caso de no disponer de esta estructura de datos tan ventajosa tendríamos que ordenar previamente los trabajos con cualquier algoritmo de ordenación. Una vez llenada la cola, pasamos a ir sacando trabajos y atendiéndolos.

Para el caso de varios electricistas la única diferencia es que creamos una lista con todos los electricistas y puesto que ninguno tiene una ventaja sobre los demás, pueden ir cogiendo los trabajos ordenados indistintamente cualquiera de ellos. Cuando un electricista termina un trabajo queda libre para coger otro.

4.4. Código

Un solo electricista

```
    vector<int>::iterator it;
    cout << endl;
    cout << "----LISTA DE REPARACIONES PENDIENTES----" << endl;
    for(it = reparaciones.begin(); it != reparaciones.end(); ++it){
      cout << "Reparacion de duración "<< *it << endl;
    }
    cout << endl;
    for(it = reparaciones.begin(); it != reparaciones.end(); ++it){
      cola.push(*it);
    }
    cout << "----ORDEN EN EL QUE EL ELECTRICISTA ELIGE LAS REPARACIONES----" << endl;
    while(!cola.empty()){
      cout << "Electricista elige reparación de duración: " << cola.top() << endl;
      cola.pop();
    }



    return 0;
}
```

Varios electricistas

```
    cout << "----LISTA DE REPARACIONES PENDIENTES----" << endl;
    for(int i = 0; i < n_reparaciones; i++){
      cout << "Reparacion de duración "<< reparaciones[i] << endl;
    }
    cout << endl;
    for(int i = 0; i < n_reparaciones; i++){
      cola.push(reparaciones[i]);
    }


    while(!cola.empty()){
      for(int i = 0; i < n_electricistas && !cola.empty(); i++){
        electricista.push_back(cola.top());
        cout << "Electricista " << i << " elige reparación de duración " << electricista[i] << endl;
        cola.pop();
      }
      electricista.clear();
    }

    return 0;
}
```

4.5. Conclusión

Este ejercicio no nos supuso ningún problema ya que la única dificultad que tiene es la ordenación de los trabajos, cosa que subsanamos con una estructura de datos adecuada. Pudimos observar que se complicaría un poco más si los electricistas tuvieran distintas capacidades para resolver problemas más rápido o distintos conocimientos para resolver problemas de distintas áreas. De todas formas, eso también tendría fácil solución ordenando los electricistas según sus cualidades y dándole prioridad a los mejores. Por lo demás, la solución que hemos aportado al problema es optimal y no hemos podido encontrar ningún ejemplo en el que no devuelva la solución óptima.