

Manuel Alberola Torres

74017527 P

ANÁLISIS Y DISEÑO DE ALGORITMOS

Branch & Bound

Práctica 8 de laboratorio

Entrega: Hasta el domingo 3 de Junio de 2018, 23:55h (a través de Moodle)

Asignación de coste mínimo IV

1 Estructuras de datos

- **1.1 Nodo:**

- Emplacements: Vector de enteros. Representa las ciudades, 0 donde no hay gasolinera 1 donde sí.
- k: Representa el nivel del árbol.
- g: Número de gasolineras asignadas. Lo utilizamos para evitar recorrer el vector emplacements.
- value: Valor obtenido con esta disposición de ciudades.
- optimistic: Valor de la cota optimista para ese nodo.

```
struct Node {  
    vector<int> emplacements;  
    unsigned k;  
    unsigned g; //Gasolineras asignadas  
    double value;  
    double optimistic;  
};
```

- **1.2 Lista de nodos vivos:** Cola de prioridad de nodos vivos. En ella almacenamos tanto los nodos prometedores como los factibles. Más adelante comentaremos la estrategia de ordenación.

He utilizado una cola de prioridad porque quiero una búsqueda dirigida hacia la tupla más prometedora

```
struct is_worse {  
    bool operator() (const Node& a, const Node& b) {  
        return a.g < b.g;  
    }  
};  
  
priority_queue<Node, vector<Node>, is_worse > NodosVivos;
```

2 Mecanismos de poda

2.1 Poda de tuplas no factibles: Entendemos por tuplas no factibles aquellas que no cumplen las condiciones del problema. Para controlarlas he creado una función que dado un nodo comprueba si las gasolineras de este superan las permitidas.

Antes de añadir un nodo a la lista de nodos compruebo si es un nodo permitido.

```
/*
 *Check if the combination is possible
 */
bool possible(Node nodo){ if (nodo.g > g){ return false; }else{ return true; }}
```

2.2 Poda de tuplas no prometedoras: Para evitar expandir nodos que no van a llevar a una solución hay que evitar meterlos en la cola. Para ello he implementado una función de **cota optimista** que devuelve el mejor valor posible de esa tupla. Aunque supere los valores permitidos.

La función supone que lo emplacements que faltan por completar sí llevan gasolinera. De forma que si el valor obtenido hasta ahora supera lo mejor que podría conseguir es un claro indicador de que no es una tupla prometedora y debemos podarla.

Antes de meter un nodo en la cola compruebo su cumple esta condición.

```
/*
 * Cota optimista empleada para poda
 * Devuelve el mejor valor que se puede conseguir. Puede ser imposible.
 * Si el mejor valor que se puede conseguir es peor que el mejor que
 * tenemos hasta el momento se poda el nodo.
 */
double optimistic(Node nodo){

    for(int i = nodo.k; i < nodo.emplacements.size(); i++){
        nodo.emplacements[i] = 1;
    }

    setSolution(nodo.emplacements);
    nodo.optimistic = greedy();
    return nodo.optimistic;
}
```

Como **cota pesimista** he empleado un algoritmo visto en clase, en concreto un voraz. Ordenamos los pueblos por número de habitantes y asignamos gasolineras hasta que se agotan las disponibles. El valor obtenido con el voraz pasa a ser el mejor resultado obtenido hasta el momento.

```
V_aux = V;

std::sort(V_aux.begin(), V_aux.end(), orderDesc);
std::sort(Index.begin(), Index.end(), orderIndex);

for (int i=g-1; i>=0; i--){
    S.push_back(Index[i]);
}

for (int i = S.size(); i<n; i++){
    T.push_back(Index[i]);
}

BEST = greedy();
```

Además de controlar las inserciones en la cola también hay que tener en cuenta las extracciones, ya que desde que insertas una tupla hasta que la extraes puede haber cambiado el mejor valor obtenido hasta la fecha. Para controlarlo he creado un bucle que extrae nodos hasta que encuentra uno cuya cota optimista sea mejor que la obtenida hasta ahora.

Si no es así desecho el nodo, ya que no tiene sentido expandir una tupla cuyo mejor posible resultado es un resultado peor que el que tengo.

```
while(!NodosVivos.empty()){
    do{
        Iterations ++;
        expandir = NodosVivos.top(); //Assign first node
        NodosVivos.pop(); //Extract first node
    }while (expandir.value>= BEST && !NodosVivos.empty());
```

4 Otras mejoras

A nivel de cálculo computacional, he tratado de evitar operaciones innecesarias que incrementaban notablemente el tiempo de cálculo. Algunas tan sencillas como una asignación.

Inicializando el vector a 0 evitamos hacer continuas asignaciones, reducimos las asignaciones a la mitad.

Así como el incremento de la variable g.

```
//Expand tree
Node hijo1 = expandir;
Node hijo2 = expandir;

//hijo1.emplacements[expandir.k] = 0;
hijo2.emplacements[expandir.k] = 1;

hijo1.k = expandir.k + 1;
hijo2.k = expandir.k + 1;

//hijo1.g = expandir.g + 0;
hijo2.g = expandir.g + 1;
```

5 Estrategia de recorrido del árbol

Estrategias de recorrido del árbol: He probado distintas estrategias, ordenando el árbol por nivel y por cota optimista pero el mejor resultado lo he obtenido con una ordenación por las gasolineras asignadas.

```
struct is_worse {
    bool operator() (const Node& a, const Node& b) {
        return a.g < b.g;
    }
};

priority_queue<Node, vector<Node>, is_worse > NodosVivos;
```

Estudio comparativo de estrategias de búsqueda

- Ordenando por nivel del árbol (k):

```
Terminal Archivo Editar Ver Buscar Terminal Ayuda
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 30a-12g.p
Bb: 6675.52
Emplacements: 5 6 10 13 14 15 16 17 19 20 21 24
CPU time (ms): 16723
Iterations of loop while: 11305090
manuel@manuel:~/Escritorio/ADA/Practica 9$
```

- Ordenando por número de gasolineras asignadas (g):


```
Terminal Archivo Editar Ver Buscar Terminal Ayuda
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 30a-12g.p
Bb: 6675.52
Emplacements: 5 6 10 13 14 15 16 17 19 20 21 24
CPU time (ms): 12461
Iterations of loop while: 8310195
manuel@manuel:~/Escritorio/ADA/Practica 9$
```

- Ordenando por cota optimista (optimistic):

```
Terminal Archivo Editar Ver Buscar Terminal Ayuda
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 30a-12g.p
Bb: 6675.52
Emplacements: 5 6 10 13 14 15 16 17 19 20 21 24
CPU time (ms): 12643
Iterations of loop while: 8310195
manuel@manuel:~/Escritorio/ADA/Practica 9$
```

6 Resultados obtenidos

El algoritmo es capaz de resolver todos los mapas en tiempos más que razonables. En esta imagen aparecen las ejecuciones de los mapas y sus distintos resultados.



```
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 07a-03g.p
Bb: 207.5
Emplacements: 3 4 5
CPU time (ms): 0
Iterations of loop while: 65
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 20a-10g.p
Bb: 6225.16
Emplacements: 4 5 6 7 9 13 14 15 17 19
CPU time (ms): 66
Iterations of loop while: 53782
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 30a-12g.p
Bb: 6675.52
Emplacements: 5 6 10 13 14 15 16 17 19 20 21 24
CPU time (ms): 16668
Iterations of loop while: 11305090
manuel@manuel:~/Escritorio/ADA/Practica 9$ ./mca -f 30a-25g.p
Bb: 758.106
Emplacements: 1 3 4 5 6 7 9 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 27 28 29
CPU time (ms): 10
Iterations of loop while: 4410
manuel@manuel:~/Escritorio/ADA/Practica 9$
```

La última ejecución, a pesar de ser de las más complicadas la resuelve casi al instante, esto se debe a la cota pesimista aportada por el voraz.

Al principio el algoritmo era incapaz de resolver el último mapa pero esta poda reduce en un 90% los nodos.