Adrián Berenguer Agulló, 74445262N

Memoria de la práctica final

1. Estructura de datos

1.1. Nodo

El nodo ha sido definido con 4 valores:

En primer lugar un typedef Sol con el vector de las puertas, cuya estructura se basa en 0 (no es puerta de enlace) y 1 (es puerta de enlace). Se guarda para cada uno de los nodos de manera que se pueda calcular el tráfico estimado.

El segundo valor es un entero en el cual se guarda el índice (k) de la iteración en la que se encuentra el nodo. Se guarda de manera que se va iterando en base a este iterador.

El tercer valor es un entero que actúa como contador de las puertas que se han colocado hasta el momento. Se guarda para podar en caso por ejemplo que ya hayamos puesto m puertas, lo demás son 0 y terminamos.

El último valor es un double en el que se guarda el valor de la cota optimista del nodo que más tarde se utilizará para aumentar el número de nodos que fueron prometedores y se descartaron finalmente cuando 'afloraron'.

Captura de la declaración del tipo Sol para el vector y el tipo tuple para el nodo.

```
199     typedef vector<int> Sol;
200     typedef tuple<Sol, int, int, double> Node;
```

1.2. Lista de nodos vivos

Se ha utilizado una cola de prioridad, priority_queue (heap máximo) para ir almacenando cada uno de los nodos con la estructura anteriormente mencionada a medida que se iteraba en el bucle while. De esta manera guardamos la lista de nodos vivos que son prometedores.

Captura en la que se muestra la declaración de la cola de prioridad, la forma en la que se introduce un nodo (línea 208) y la forma en la que se accede a leer sus datos y se elimina el primer nodo de ella (líneas 211-216).

```
202     priority_queue<Node> pq;
203     Node nodo;
204     double ind, cota;
205     int contador;
206     best_val = met_greedy_2(m, n, capacidades, caminos); //Cota pesim
207
208     pq.emplace( Sol(capacidades.size()), 0, 0, numeric_limits<double>
209
210     while( !pq.empty() ) {
211         nodo = pq.top();
212         vec = get<0>(nodo);
213         ind = get<1>(nodo);
214         contador = get<2>(nodo);
215         cota = get<3>(nodo);
216         pq.pop();
```

Para extraer el nodo más prometedor se ha comenzado dando valor a la variable que guarda el resultado mediante una cota pesimista con el algoritmo voraz. Desde ese momento tenemos en el mejor valor un resultado que es factible y puede o no ser el óptimo. En base a él iteramos en busca de nodos que sean mejores que este. Para ello, se ha utilizado una cota optimista basada en pensar que el número de nodos restantes son puerta de enlace. En caso de que la solución sea peor (más alta) que la mejor hasta el momento, se descarta el nodo. En caso contrario, seguimos explorando este nodo.

2. Mecanismos de poda

2.1. Poda de nodos no factibles

La poda de nodos no factibles se ha realizado en base al contador que tenemos en cada nodo de las puertas de enlace que han sido colocadas, de manera que, si este contador es mayor que m, significa que ya tenemos todas las puertas de enlace colocadas y no es necesario seguir explorando, por tanto, descartamos el nodo por no ser factible.

```
}
if(contador <= m) { //Nodos explorados
    explorados++;
    double newBest = cota_optimista(m, n, capacidade
```

2.2. Poda de nodos no prometedores

La poda de nodos no prometedores se basa en la cota optimista de poner, en los nodos restantes, puerta de enlace en todos. De esta manera, si la solución de poner puerta de enlace en los nodos restantes, sin pararse a pensar en las puertas de enlace que quedan por colocar, es mayor que la mejor solución hasta el momento, no seguimos explorando este nodo y lo descartamos por no ser prometedor.

```
    double newBest = cota_optimista(m, n, capacidades
    if(newBest < best_val) {
        pq.emplace( vec, ind+1, contador, newBest);
    }
```

3. Cotas pesimistas y optimistas

3.1. Cota pesimista inicial (inicialización)

La cota pesimista inicial utilizada es la del algoritmo voraz (greedy) con la heurística de comenzar con todos los nodos puertas de enlace e ir quitando, uno a uno, los nodos que menos hagan aumentar el tráfico estimado. Este algoritmo obtiene un resultado factible, que puede o no ser el óptimo.

```
int contador;
best_val = met_greedy_2(m, n, capacidades, caminos); //Cota pesim
```

3.2. Cota pesimista del resto de nodos

En este ejercicio no se han usado cotas pesimistas para el resto de nodos.

3.3. Cota optimista

La cota optimista se basa, como se ha explicado anteriormente, en colocar en los x nodos restantes puertas de enlace en todos y, si no se obtiene un resultado mejor que el que se tiene hasta ahora, se descarta el nodo, ya que un resultado mejor que ese es imposible porque si todos los nodos son puerta de enlace, el tráfico es 0.

```
188  //Cota optimista basada en poner 1 en los nodos restantes
189  double cota_optimista(int m, int n, vector<double> capacidades, vecto
190
191      for(unsigned i = indice; i < puertas.size(); i++) {
192          puertas[i] = 1;
193      }
194      double result = tablaBuena(m, n, capacidades, caminos, puertas);/
195      return result;
196  }
197
```

4. Otros medios empleados para acelerar la búsqueda

Para acelerar la búsqueda se han usado dos medios añadidos, que son:

- En caso de tener m puertas ya colocadas, ya no tengo que colocar puertas, por tanto puedo rellenar el vector de puertas con 0 y calcular el tráfico, en caso de obtener un mejor valor que el actual, lo guardo y continúo con la siguiente iteración del bucle while, de esta manera, no se exploran tantos nodos que no se necesiten.

```
243          }
244          if(contador == m) { //Mejora -> si ya tengo m puertas colocad
245              for(unsigned x = ind; x < capacidades.size(); x++) {
246                  vec[x] = 0;
247              }
248              double aux = tablaBuena(m, n, capacidades, caminos, vec);
249              if(aux < best_val) {
250                  best_val = aux;
251                  tablaFinal = vec;
252              }
253              continue ;
254          }
```

- En caso de que falten x nodos por colocar y x puertas, se colocan x puertas en los x nodos, calculo el tráfico y termino, de manera que se ahorran iteraciones y nodos que no son necesarios.

```
233         if(ind >= n - m && m - contador == n - ind) { //Mejora -> Si
234             for(unsigned x = ind; x < capacidades.size(); x++) {
235                 vec[x] = 1;
236             }
237             double aux = tablaBuena(m, n, capacidades, caminos, vec);
238             if(aux < best_val) {
239                 best_val = aux;
240                 tablaFinal = vec;
241             }
242             continue ;
243         }
```

5. Estudio comparativo de distintas estrategias de búsqueda

Después de realizar la ejecución del programa con los algoritmos de backtracking y ramificación y poda, podemos realizar una comparativa de los resultados obtenidos.
En medidas generales, podemos observar como el algoritmo de backtracking resuelve en menor tiempo el algoritmo y visita un número menor de nodos.
Tabla comparativa con resultados de tiempo y nodos visitados, explorados, nodos hoja, nodos no factibles y nodos no prometedores de ramificación y poda y backtracking.

Ramificación y poda

Prb	Tie	Vis	Ex	Ho	Nf	Np
7n3m	0.6	68	68	0	0	2
10n2m	1.8	88	88	0	0	10
10n5m	3.2	252	252	0	0	82
20n5m	344.1	21532	21532	0	0	4584
20n	111	744	744	0	0	281

Backtracking

Prb	Tie	Vis	Ex	Ho	Nf	Np
7n3m	0.35	66	66	0	0	4
10n2m	0.65	88	88	0	0	10
10n5m	1.36	252	252	0	0	82
20n5m	72.78	20170	20170	0	0	4418
20n	221	736	736	0	0	278

10m	5.2	68	68			56
20n15m	67.36	4036	4036	0	0	1662
30n12m	321108.71	12834686	12834686	0	0	4251768

10m	.61	90	90			72
20n15m	16	4036	4036	0	0	1662
30n12m	55207.88	12409336	12409336	0	0	4122976

En definitiva, podemos concluir que el algoritmo voraz, obtiene la solución en menos tiempo. Este algoritmo es posible que tenga una gran complejidad temporal y es bastante mejorable de la forma en que se ha hecho. El algoritmo de ramificación y poda podría hacerse también una poda más exhaustiva para definir cuál es más rápido finalmente.

6. Tiempos de ejecución

- Fichero 07n03m.p: 0.60 ms.
- Fichero 10n02m.p: 1.80 ms.
- Fichero 10n05m.p: 3.20 ms.
- Fichero 20n05m.p: 344.71 ms.
- Fichero 20n10m.p: 1151.20 ms.
- Fichero 20n15m.p: 67.36 ms.
- Fichero 30n12m.p: 321108.71 ms. (5 minutos y 10 segundos…)
- Fichero 30n26m.p: 120.64 ms.
- Fichero 40n05m.p: 28374.95 ms.
- Fichero 40n10m.p: ?
- Fichero 40n20m.p: ?
- Fichero 40n25m.p: ?
- Fichero 50n05m.p: ?
- Fichero 50n07m.p: ?