Ejercicio 14 de Programación Dinámica

Martín Gómez Abejón

Métodos Algorítmicos en Resolución de Problemas II

Enunciado: Dado un grafo dirigido $G = (V, A)$ con costes en las aristas, resolver mediante programación dinámica el problema del viajante de comercio. Se ha de dar el detalle de todos los pasos: recurrencia, solución iterativa, recuperación del ciclo hamiltoniano de mínimo coste, y coste del algoritmo en tiempo y en espacio.

Solución: podemos obtener un algoritmo mejor que el que recorre todas las permutaciones. Este algoritmo tiene coste $O(n^2 2^n)$, que es algo mejor que $O(n!)$ (recordemos que $\log(n^2 2^n) \in \Theta(n)$, mientras que $\log(n!) \in \Theta(n \log(n))$). El subproblema apropiado para el problema general es

Subproblema

Para todo subconjunto de ciudades $S \subseteq \{1, 2, \ldots, n\}$ que incluye al 1, y para todo $j \in S$, sea $DP(S, j)$ la longitud del camino más corto que visita a cada nodo en S exactamente una vez, empezando en el 1 y terminando en el j.

Podemos obtener el valor de esta función a través de programación dinámica. Hay dos casos base. Si $S = \{1\}$, entonces $DP(S, 1) = 0$. Por otro lado, si $|S| > 1$, entonces $DP(S, 1)$ es claramente ∞ ya que el camino descrito en el subproblema no puede empezar y terminar en el 1.

El resto de casos tienen una definición recursiva, considerando la penúltima ciudad por la que pasa el (uno de los) camino(s) más corto(s), i. Se cumple que $i \in S$, y el camino pasa por todos los vértices de S salvo por el j. La distancia del camino más corto es la longitud del camino que va desde 1 hasta i más la longitud de la arista de i a j, d_{ij}, y a priori no sabemos el valor de i por lo que comprobamos todos a través de llamadas recursivas, obteniendo así la siguiente fórmula recursiva.

$$DP(S, j) = \min_{i \in S \mid i \neq j} DP(S \setminus \{j\}, i) + d_{ij}$$

La inducción está bien definida, ya que los casos recursivos solo dependen de subconjuntos de cardinalidad estrictamente menor, por lo que se puede calcular el valor de $DP(S, j)$ recorriendo los conjuntos de manera ascendente en función del cardinal.

Para resolver el problema original, las llamadas iniciales necesarias desde un punto de vista recursivo son

$$DP(\{1, 2, \ldots, n\}, j) \forall j \in \{2, 3, \ldots, n\}.$$

Nótese que en el caso peor es necesario calcular todos los valores $DP(S, j)$ posibles para conocer sin lugar a dudas cuál es el valor de esas llamadas iniciales, ya que para cada conjunto S es necesario calcular los valores de DP para todos sus subconjuntos que contienen al 1 para calcular los valores en los que interviene S.

A partir de estas llamadas, se puede recuperar fácilmente el ciclo hamiltoniano viendo cada ciclo como un camino mínimo que va desde 1 hasta cierto vértice j pasando por cada vértice una sola vez. Una vez recorrido ese camino, hay que volver al vértice 1. Por lo tanto, el ciclo Hamiltoniano tiene longitud

$$\min_j DP(\{1, \ldots, n\}, j) + d_{j1}.$$

Para reconstruir la solución, en primer lugar calculamos el j que hace mínima la expresión anterior. Después, retrocedemos en las llamadas recursivas, obteniendo los sucesivos i que minimizan la expresión

$$DP(S, j) = \min_{i \in S \mid i \neq j} DP(S \setminus \{j\}, i) + d_{ij}$$

hasta llegar al conjunto $\{1\}$. El ciclo buscado se corresponde con los argumentos que minimizan los casos recursivos, en orden desde $\{1, 2, \ldots, n\}$ hasta $\{1\}$.

Hay como mucho $2^n \cdot n$ subproblemas, y cada uno se resuelve en tiempo $O(n)$, por lo que el coste en tiempo del algoritmo es $O(n^2 2^n)$ y en espacio es $O(n2^n)$. Usando argumentos de probabilidad (Teorema Central del Límite entre otros) se deduce que, para n suficientemente grande, una mayoría de los subconjuntos considerados tienen cardinalidad tan cercana en términos relativos a $n/2$, por lo que los costes calculados previamente son en orden exacto.

Hay varias formas de representar los subconjuntos en este problema. Una de ellas es utilizar estructuras como los números enteros para indizar los conjuntos en un vector. Así, si $n = 3$ tendríamos un vector de conjuntos con 4 elementos y la siguiente correspondencia.

Codificación	0	1	2	3
Conjunto	$\{1\}$	$\{1, 2\}$	$\{1, 3\}$	$\{1, 2, 3\}$

Hay otra forma de implementar el problema que nos permite rebajar mucho su complejidad si el grafo tiene muy pocas aristas (por ejemplo, con este sistema podríamos resolver el problema del viajante en una red de Metro). Se trata de considerar solo conjuntos para los cuales haya algún subproblema factible. Así, los cálculos funcionan aumentando la cardinalidad y considerando nuevos conjuntos de manera perezosa a través de una cola. Así, recorremos el árbol de conjuntos relevantes por niveles (es decir, aumentando la cardinalidad desde 1). Esta implementación es compleja si se implementa a bajo nivel y es necesario distinguir muchos casos, por lo que únicamente vamos a dar su pseudocódigo, implementando el recorrido exhaustivo con todo detalle.

El pseudocódigo de la estrategia perezosa es

```
cola q;
tablaDinamica DP;
for i ∈ conectadosCon({1}) :
    q ⟸ {1, i};
DP({1}, 1) = 0;
while q ≠ ∅ :
    q ⟹ S;
    for j ∈ S, j ≠ 1, S \ {j} ∈ DP :
        DP(S, j) = mín{C(S \ {j}, i) + d_ij | i ∈ S, i ≠ j, DP(S \ {j}, i) ∈ DP};
    for i ∈ conectadosCon(S) :
        q ⟸ S ∪ i;
return mín_j DP({1, ..., n}, j) + d_j1;
```

y el único valor añadido que tiene la implementación en detalle de este algoritmo es el cuidadoso manejo de llamadas recursivas de forma dinámica, así como la implementación eficiente de las clases y las funciones que intervienen en el algoritmo. Nótese que la función `conectadosCon` devuelve los vértices que están directamente conectados al conjunto S que no pertenecen a S. Además, al añadir conjuntos a la cola se marcan en `tablaDinamica` para evitar repeticiones.

El pseudocódigo de la versión no perezosa es

```
DP({1}, 1) = 0;
for s ∈ {2, ..., n} :
    for S ⊆ {1, ..., n} | |S| = s, 1 ∈ S :
        DP(S, 1) = ∞;
        for j ∈ S, j ≠ 1 :
            DP(S, j) = mín{C(S \ {j}, i) + d_ij | i ∈ S, i ≠ j};
return mín_j DP({1, ..., n}, j) + d_j1;
```

y a continuación presentamos una posible implementación, que utiliza índices desde 0.

```
int tam(lli i)
{
    int res = 1;
    while (i)
    {
    if (i%2){res++;}
    i>>=1;
    }
    return res;
}
vector<vector<lli>> procesar(
    const vector<vector<lli>>& grafo, int n)
{
    //Tratamos a \{INT_MAX,...,LONG_MAX\} como
    //el numero infinito
    vector<vector<lli>> DP((lli)1<<(n-1),
        vector<lli>(n,LONG_MAX/4));
    DP[0][0] = 0;
    for (int s = 2;s<=n;s++)
    {
        for (lli i = 0;i<DP.size();i++)
        {
            //Test de tamano
            if (tam(i)==s)
            {
                DP[i][0] = INT_MAX;
                for (int elem=1;elem<n;elem++)
                {
                    //Test de pertenencia
                    if ((1<<(elem-1))&i)
                    {
                        lli conjuntoPrevio = i &
                        (~(1<<(elem-1)));
                        for (int elemPrevio=0;elemPrevio<n;
                            elemPrevio++)
                        DP[i][elem] = min(DP[i][elem],
                            DP[conjuntoPrevio][elemPrevio]+
                            grafo[elem][elemPrevio]);
                    }
                }
            }
        }
    }
    return DP;
}
```

```
vector<int> obtenerCiclo(const vector<vector<lli>>& grafo,
    const vector<vector<lli>>& resProc, int n)
{
    lli conjuntoAct = (1<<(n))-1;
    int vertSalida = 0;
    int indSol = 0,solMenor = INT_MAX;
    vector<int> solucion;
    //Para cada vertice del recorrido optimo
    while (conjuntoAct)
    {
    for (int i=0;i<n;i++)
    {
        //Seleccionar el siguiente de manera optima
        if (i != vertSalida && ((conjuntoAct>>i)&1) &&
            resProc[conjuntoAct>>1][i]+
                grafo[vertSalida][i]<solMenor)
        {
            indSol = i;
            solMenor = resProc[conjuntoAct>>1][i]+
                grafo[vertSalida][i];
        }
    }
    solucion.push_back(indSol);vertSalida = indSol;
    solMenor = INT_MAX;conjuntoAct &= (~(1<<indSol));

    }
    return solucion;
}

int main()
{
    int n;cin >> n;
    vector<vector<lli>> grafo(n,vector<lli>(n,INT_MAX));
    int a1,a2,c;
    cin >> a1 >> a2 >> c;
    while (cin)
    {
        grafo[a1][a2]=c;
        grafo[a2][a1]=c;
        cin >> a1 >> a2 >> c;
    }
    auto resProc = procesar(grafo,n);
    vector<int> recorrido = obtenerCiclo(grafo,resProc,n);
    return 0;
}
```