UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 5: Programación Dinámica

Doble Grado en Ingeniería Informática y Matemáticas

Algorítmica

Autores y correos:

Jaime Corzo Galdó: jaimecrz04@correo.ugr.es

Marta Zhao Ladrón de Guevara Cano: mzladron72@correo.ugr.es

Sara Martín Rodríguez: smarrod02@correo.ugr.es

27 de septiembre de 2024

Índice

1. Autores

El trabajo se ha repartido de manera igualitaria pues cada uno ha realizado un tercio del trabajo.

2. Objetivos

El objetivo de esta práctica es comprender y asimilar y el funcionamiento de la técnica de resolución de problemas "Programación Dinámica".

3. Definición del problema

El problema a resolver utilizando la técnica de programación dinámica es el ya conocido Viajante de Comercio cuyo enunciado es el siguiente: Un agente comercial tiene que visitar n ciudades. Se proporciona una matriz cuadrada D, de dimensiones n×n tal que $d_{i,j}$ indica el coste de viajar desde la ciudad i a la j. El agente debe realizar un recorrido visitando todas las ciudades y finalizando en la misma ciudad desde la que partió. Además sólo puede visitar cada cuidad una sola vez. El coste global de un recorrido dado es la suma de los costes de todos los traslados. El objetivo es encontrar un recorrido que cumpla las restricciones establecidas y tenga el menor coste posible.

3.1. Datos de entrada

Los datos de entrada de este problema son las distancias que hay entre cada una de las ciudades, suponiendo que no tiene por qué existir un camino, en tal caso se supone que la distancia entre ambas ciudades es infinito. Estos datos vendrán dispuestos en una matriz cuadrada en la que su diagonal principal estará llena de ceros.

3.2. Datos de salida

Teniendo en cuenta la definición del algoritmo y que el viajante de comercio tiene que pasar por todas las ciudades sin repetir, la solución será una permutación sin repetición, que se devolverá en un vector.

4. Enfoque a la programación dinámica

Para ver que dicho problema puede resolverse utilizando la programación dinámica hay que comprobar dos condiciones: que haya subestructuras optimales y que haya superposición de problemas. La existencia de subproblemas optimales nos dice que la solución óptima a un problema puede ser definida en función de soluciones óptimas a subprobemas de tamaño menor. Por otro lado, el solapamiento entre subproblemas quiere decir que al usar la implementación recursiva, un mismo problema se acaba resolviendo más de una vez, lo cuál es ineficiente, pero si esos datos son las

almacenados en una tabla hash, cada vez que se vuelvan a necesitar, tan solo hará falta buscarlos en la tabla.

Dentro de la programación y dada la naturaleza del problema se va a hacer uso de la técnica de programación dinámica basada en la memorización, implementando un algoritmo recursivo.

5. Algoritmo diseñado

5.1. Descripción del algoritmo

Antes de mostrar el pseudocódigo a utilizar se van a nombrar las diferentes estructuras de datos y variables que se necesitan y usan para la resolución:

1. Matriz de costos C: indica las distancias entre cada par de ciudades, no necesariamente simétrica.
2. Tabla de memoria hash H: va a ir guardando los distintos subproblemas que se vayan calculando, la clave será el problema a resolver y el valor que almacene el resultado de dicho problema.
3. Vector de candidatos S: conjunto de las ciudades por las que hay que ir pasando
4. Índice i: indica la posición
5. Vector solución *camino_min*: conjunto de las ciudades que hasta el vértice i obtienen el recorrido mínimo.
6. Índice f: ciudad que se fija de partida

Denotaremos por $g(i, S, \&camino\_min, f)$ a la función que nos da la el conjunto de vértices visitados y la longitud de este desde i hasta f, la ciudad inicial, que pase exactamente una vez por cada vértice de S. Esta función se define se define como:

- $g(i, S, camino\_min, f)$ $=\min(C_{ij} + g(j, S - \{j\}, camino\_min))$ con $j \in S$, si $i \neq f,\ S \neq \emptyset\ i \notin S$
- $g(i, \emptyset, camino\_min, f) = C_{if}, i = 2, 3, ..., n$, si $S \neq \emptyset$

5.2. Pseudocódigo del algoritmo

El pseudocódigo del algoritmo es el siguiente:

Pseudocódigo función g:

Pseudocódigo función g

```
vector<int>vector_min
camino_min.push_back(f)

funcion g(i,S, &camino_min,f) devuelve double
varibles min, distancia, pos_min
camino_min.push_back(f)

si S = ∅
     entonces devuelve C[i,f]
else
     si hash.find([i][S]) != hash.end()
          return hash[i][S]
     else
          min = ∞
          for j en S
               distancia = C[i][j] + g(j, S - {j})
               si distancia < min
                    min= distancia
                    hash[i][S] = min
                    pos_min = j
          camino_min.push_back(min)
          return min
```

Este algoritmo implementa la función $g(i, S, camino\_min, f)$ definida anteriormente y que será recursiva. Esta devuelve tal y como se ha mencionado anteriormente la longitud del camino mínimo partiendo desde la ciudad f.

En primer lugar, se añade la ciudad inicial al vector solución y, a continuación, se comprueba el conjunto S.

- Si S está vacío, significa que no hay más vértices por recorrer, luego se devuelve directamente la distancia de la matriz de costes entre el vértice i y el vértice inicial f.
- Si S no está vacío, esto quiere decir que tenemos que calcular cual es camino más corto desde i hasta f. Primero, comprobamos si esta operación ya se ha realizado anteriormente, en cuyo caso estará guardada en la tabla hash y podemos devolver directamente el resultado obtenido. De no ser así, pasamos a calcular el camino mínimo y su longitud desde ese vértice llamando recursivamente al la función g para cada uno de los vértices de S.

5.2.1. Demostración de la validez del algoritmo

La demostración de la validez del algoritmo se va a basar el principio de optimalidad de Bellman que dice lo siguiente: *Una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones debe ser también óptima respecto al subproblema que resuelve"*. Es decir, la solución óptima a cualquier caso no trivial de un problema es una combinación de soluciones

óptimas de algunos de los subcasos. Vamos a ver que si la secuencia de decisiones $x1..xn$ es óptima donde n es el número de vértices candidatos, entonces $x1..xn-1$ es óptima para $n-1$ vértices candidatos.

La demostración se hará por inducción:
Veamos en primer lugar, el caso base. Si $S = \emptyset$; es decir, cuando el número de vértices candidatos es 0, la longitud del camino mínimo desde i hasta 1 es $g(i, \emptyset) = C_{ij}$ que evidentemente es óptimo.
Suponemos ahora que se verifica para un número de vértices candidatos igual a $n-2$ es óptimo: **Hipótesis de inducción**.
Veamos ahora que con un número de vértices candidatos igual a $n-1$ se tiene que $g(1, S-\{1\})$ es óptimo: **Tesis de inducción**.
Lo vamos a hacer mediante una reducción al absurdo: Si no fuese así existe E una solución óptima, lo que implica que $E < g(1, S-\{1\})$. También tenemos que va a existir una posición $i \in S-\{1\}$ que sea la segunda ciudad que en esta solución se visite desde 1. Luego la solución es $C_{1i}+$ coste de ir desde i a 1 recorriendo $S-\{1,i\}$ pero como la solución es óptima debe de ser igual a $g(i, S-\{1i\})$(hipótesis de inducción) luego tenemos que:

1. Como $g(i, S-\{1i\})$ es óptima, al añadir el punto i tenemos $g(1, S-\{1\}) < C_{1,i} + g(1, S-\{1,i\})$
2. Como $g(i, S-\{1i\})$=min$(C_{ij} + g(j, S-\{1,j\}))$ con $j \in S-\{1\}$ tenemos que $g(1, S-\{1\}) >= C_{1i} + g(j, S-\{1,i\})$

Evidentemente esto es absurdo luego concluye nuestra demostración.

■

Para concluir la demostración de la optimalidad de este algoritmo faltaría demostrar que se producen superposiciones de problemas, otra de las razones por lo que es útil utilizar la técnica de programación dinámica para solucionar el problema.

Hemos definido $g(i, S, camino\_min, f)$ =min$(C_{ij} + g(j, S-\{j\}, camino_min))$ con $j \in S$, si $i \neq f,\ S \neq \emptyset\ i \notin S$.
Fijamos los índices $i, j, k, \psi \in \mathbb{N}$ y sabemos que f es el vértice de la ciudad que tomamos como inicial. Supongamos que el número de elementos del conjunto S es mayor o igual que 5 y que $i, j, \psi, k \in S$ con $i \neq j \neq \psi \neq k$.
Comenzamos calculando la función g para el índice ψ: $g(f, S \setminus \{f\}) = C_{f,\psi} + g(\psi, S \setminus \{f, \psi\})$, entonces tendremos que calcular:

- $g(i, S \setminus \{f, \psi, i\})$ donde tendremos que calcular $g(j, S \setminus \{f, \psi, i, j\})$ y posteriormente $g(k, S \setminus \{f, \psi, i, j, k\})$ pues tenemos que pasar por todos los vértices.
- $g(j, S \setminus \{f, \psi, j\})$ donde tendremos que calcular $g(i, S \setminus \{f, \psi, j, i\})$ y posteriormente $g(k, S \setminus \{f, \psi, j, i, k\})$ pues tenemos que pasar por todos los vértices.

Entonces vemos claramente que se repiten problemas que calcular por lo que tiene sentido guardarlo dichos resultados para ahorrar tiempo de ejecución.

■

5.2.2. Eficiencia del algoritmo

Vamos a calcular la función g para cada uno de los vértices de S excepto para el vértice f que fijamos, luego n-1, en la que tenemos dos opciones:

1. Si $S = \emptyset$ solo se realiza una consulta a la matriz de costes C en tiempo constante, luego tenemos una eficiencia de $O(1)$
2. Si $S \neq \emptyset$, volvemos a encontrarnos dos casuísticas:
   - Si la operación ya se ha realizado anteriormente solo consultamos su resultado en la tabla hash que normalmente almacenaremos en un unordered_map pues en casos medios la consulta de un elemento se realiza en tiempo constante. Luego la eficiencia es $O(1)$
   - Si la operación no se ha realizado, para n-2 vértices (todos menos f e i) se realiza una llamada recursiva y una serie de comprobaciones para comprobar que el resultado obtenido es el mínimo, todas ellas en tiempo constante que nos da una eficiencia de $\Omega(n2^n)$

Luego la eficiencia es de $\Omega(n2^n)$ que es mejor que $\Omega(n!)$ que resultaría de usar la fuerza bruta.

6. Conclusiones

El desarrollo de esta práctica nos ha permitido ver que la técnica de Programación dinámica es una buena técnica para resolver el problema del Viajante de Comercio pues nos da una solución óptima al contrario de lo que ocurría cuando intentábamos aplicar la técnica Greedy. Además, esta supone una ventaja frente a la técnica Divide y Vencerás, los subproblemas que obtenemos están encajados y podemos reducir el número de cálculos a realizar. Sin embargo, también se comprueba que esta técnica supone un mayor consumo de recurso, sobre todo de memoria para guardar los cálculos realizados.