20174179 杨小川 计科卓越班

（1）设gap代价为0.6，mismatch代价为1.0。求字符串 cafe 和 coffee的最小编辑距离

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | | C | O | F | F | E | E |
| 0.6 | 1.2 | 1.8 | 2.4 | 3.0 | 3.6 |
| C | 0.6 | 0 | 0.6 | 1.2 | 1.8 | 2.4 | 3.0 |
| A | 1.2 | 0.6 | 1.0 | 1.6 | 2.2 | 2.8 | 3.4 |
| F | 1.8 | 1.2 | 1.6 | 1.0 | 1.6 | 2.2 | 2.8 |
| E | 2.4 | 1.8 | 2.2 | 1.6 | 2.0 | 1.6 | 2.2 |

所以最小编辑距离为2.2

**#include**<iostream>

**#include**<cstdio>

**#include**<string>

**#define** N 2000

**using** **namespace** std;

**int** **minimum**(**int** a, **int** b, **int** c)

{

**int** temp = a < b ? a : b;

**return** temp < c ? temp : c;

}

**int** **main**()

{

string s\_change, s\_target;

cin >> s\_change;

cin >> s\_target;

**int** Edit\_dist[N][N] = {0};

**unsigned** **int** i,j;

**for** (i = 0; i <= s\_change.length(); i++)

Edit\_dist[i][0] = i;

**for** (i = 0; i <= s\_target.length(); i++)

Edit\_dist[0][i] = i;

**for** (i = 1; i <= s\_change.length(); i++)

{

**for** (j = 1; j <= s\_target.length(); j++)

{

**if** (s\_change[i-1] == s\_target[j-1])

Edit\_dist[i][j] = Edit\_dist[i-1][j-1];

**else**

{

Edit\_dist[i][j] = 1 + minimum(Edit\_dist[i-1][j],

Edit\_dist[i][j-1], Edit\_dist[i-1][j-1]);

}

}

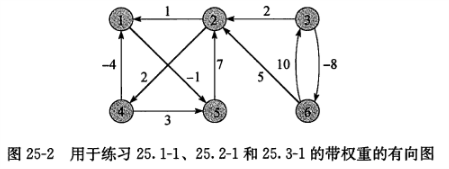
}

cout << Edit\_dist[s\_change.length()][s\_target.length()];

**return** 0;

}

（2）25.1-1 在图25-2所示的带权重的有向图上运行算法SLOW-ALL-PAIRS-SHORTEST-PATHS. 给出循环的每次迭代所计算出的矩阵.然后用算法FASTER-ALL-PAIRS-SHORTEST-PATHS重新做一遍。



SLOW-ALL-PAIRS-SHORTEST-PATHS：

L（1）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 99999 | 99999 | 99999 | -1 | 99999 |
| 1 | 0 | 99999 | 2 | 99999 | 99999 |
| 99999 | 2 | 0 | 99999 | 99999 | -8 |
| -4 | 99999 | 99999 | 0 | 3 | 99999 |
| 99999 | 7 | 99999 | 99999 | 0 | 99999 |
| 99999 | 5 | 10 | 99999 | 99999 | 0 |

L（2）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 6 | 99999 | 99999 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | 0 | 99999 |
| 3 | -3 | 0 | 4 | 99999 | -8 |
| -4 | 10 | 99999 | 0 | -5 | 99999 |
| 8 | 7 | 99999 | 9 | 0 | 99999 |
| 6 | 5 | 10 | 7 | 99999 | 0 |

L（3）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 6 | 99999 | 8 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | -3 | 99999 |
| -2 | -3 | 0 | -1 | 2 | -8 |
| -4 | 2 | 99999 | 0 | -5 | 99999 |
| 5 | 7 | 99999 | 9 | 0 | 99999 |
| 3 | 5 | 10 | 7 | 5 | 0 |

L（4）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 6 | 99999 | 8 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | -3 | 99999 |
| -5 | -3 | 0 | -1 | -3 | -8 |
| -4 | 2 | 99999 | 0 | -5 | 99999 |
| 5 | 7 | 99999 | 9 | 0 | 99999 |
| 3 | 5 | 10 | 7 | 2 | 0 |

L（5）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 6 | 99999 | 8 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | -3 | 99999 |
| -5 | -3 | 0 | -1 | -6 | -8 |
| -4 | 2 | 99999 | 0 | -5 | 99999 |
| 5 | 7 | 99999 | 9 | 0 | 99999 |
| 3 | 5 | 10 | 7 | 2 | 0 |

FASTER-ALL-PAIRS-SHORTEST-PATHS：

**#include** <iostream>

**#include** <iomanip>

**#define** UNV 99999

**using** **namespace** std;

**const** **int** N = 6;

**class** Graph

{

**public**:

**int** distance[N+1][N+1];

**int** vertices;

**Graph**(**int** n);

**Graph**(Graph &G);

};

**int** **MINIMUM**(**int** a, **int** b)

{

**return** a < b ? a : b;

}

**void** **Print**(Graph G)

{

**int** i, j;

**for**(i = 1; i <= N; i++)

{

**for**(j = 1; j <= N; j++)

cout<< right << setw(8) << G.distance[i][j];

cout<<**endl**;

}

cout<<**endl**;

}

Graph **EXTEND\_SHORTEST\_PATHS**(Graph L, Graph W)

{

**int** i, j, k, n = W.vertices;

Graph ret(n);

**for**(i = 1; i <= n; i++)

{

**for**(j = 1; j <= n; j++)

{

ret.distance[i][j] = UNV;

**for**(k = 1; k <= n; k++)

{

**if**(L.distance[i][k] != UNV && W.distance[k][j] != UNV)

ret.distance[i][j] = MINIMUM(ret.distance[i][j], L.distance[i][k] + W.distance[k][j]);

}

}

}

**return** ret;

}

Graph **SQUARE\_MATRIX\_MULTIPLY**(Graph A, Graph B)

{

**int** i, j, k, n = A.vertices;

Graph ret(n);

**for**(i = 1; i <= n; i++)

{

**for**(j = 1; j <= n; j++)

{

ret.distance[i][j] = 0;

**for**(k = 1; k <= n; k++)

{

ret.distance[i][j] = ret.distance[i][j] + A.distance[i][k] \* B.distance[k][j];

}

}

}

**return** ret;

}

Graph **SLOW\_ALL\_PAIRS\_SHORTEST\_PATHS**(Graph W)

{

**int** n = W.vertices, m;

Graph L(W);

**for**(m = 2; m < n; m++)

{

L = EXTEND\_SHORTEST\_PATHS(L, W);

Print(L);

}

**return** L;

}

Graph **FASTER\_ALL\_PAIRS\_SHORTEST\_PATHS**(Graph W)

{

**int** n = W.vertices, m = 1;

Graph L(W);

**while**(m < n - 1)

{

L = EXTEND\_SHORTEST\_PATHS(L, L);

m = 2 \* m;

Print(L);

}

**return** L;

}

**int** **main**()

{

**int** i, start, end, value;

Graph G(N);

G.distance[1][5] = -1;

G.distance[2][1] = 1;

G.distance[2][4] = 2;

G.distance[3][2] = 2;

G.distance[3][6] = -8;

G.distance[4][1] = -4;

G.distance[4][5] = 3;

G.distance[5][2] = 7;

G.distance[6][2] = 5;

G.distance[6][3] = 10;

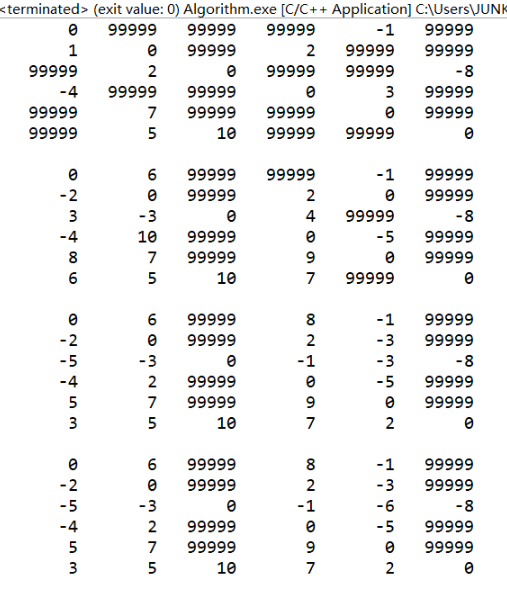
Print(G);

//SLOW\_ALL\_PAIRS\_SHORTEST\_PATHS(G);

FASTER\_ALL\_PAIRS\_SHORTEST\_PATHS(G);

**return** 0;

}



（3）25.2-1在图25-2所示的带权重的有向图上运行Floyd+Warshall算法，给出外层循环的每一次迭代所生成的矩阵D(k)。

D(1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 99999 | 99999 | 99999 | -1 | 99999 |
| 1 | 0 | 99999 | 2 | 0 | 99999 |
| 99999 | 2 | 0 | 99999 | 99999 | -8 |
| -4 | 99999 | 99999 | 0 | -5 | 99999 |
| 99999 | 7 | 99999 | 99999 | 0 | 99999 |
| 99999 | 5 | 10 | 99999 | 99999 | 0 |

D(2)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 99999 | 99999 | 99999 | -1 | 99999 |
| 1 | 0 | 99999 | 2 | 0 | 99999 |
| 3 | 2 | 0 | 4 | 2 | -8 |
| -4 | 99999 | 99999 | 0 | -5 | 99999 |
| 8 | 7 | 99999 | 9 | 0 | 99999 |
| 6 | 5 | 10 | 7 | 5 | 0 |

D(3)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 99999 | 99999 | 99999 | -1 | 99999 |
| 1 | 0 | 99999 | 2 | 0 | 99999 |
| 3 | 2 | 0 | 4 | 2 | -8 |
| -4 | 99999 | 99999 | 0 | -5 | 99999 |
| 8 | 7 | 99999 | 9 | 0 | 99999 |
| 6 | 5 | 10 | 7 | 5 | 0 |

D(1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 99999 | 99999 | 99999 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | -3 | 99999 |
| 0 | 2 | 0 | 4 | -1 | -8 |
| -4 | 99999 | 99999 | 0 | -5 | 99999 |
| 5 | 7 | 99999 | 9 | 0 | 99999 |
| 3 | 5 | 10 | 7 | 2 | 0 |

D(1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 6 | 99999 | 8 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | -3 | 99999 |
| 0 | 2 | 0 | 4 | -1 | -8 |
| -4 | 2 | 99999 | 0 | -5 | 99999 |
| 5 | 7 | 99999 | 9 | 0 | 99999 |
| 3 | 5 | 10 | 7 | 2 | 0 |

D(1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 6 | 99999 | 8 | -1 | 99999 |
| -2 | 0 | 99999 | 2 | -3 | 99999 |
| -5 | -3 | 0 | -1 | -6 | -8 |
| -4 | 2 | 99999 | 0 | -5 | 99999 |
| 5 | 7 | 99999 | 9 | 0 | 99999 |
| 3 | 5 | 10 | 7 | 2 | 0 |

**#include** <iostream>

**#include** <iomanip>

**#define** UNV 99999

**using** **namespace** std;

**const** **int** N = 6;

**class** Graph

{

**public**:

**int** distance[N+1][N+1];

**int** vertices;

**Graph**(**int** n):vertices(n)

{

**int** i, j;

**for**(i = 1; i <= N; i++)

{

**for**(j = 1; j <= N; j++)

**if**(i == j)

distance[i][j] = 0;

**else**

distance[i][j] = UNV;

}

}

**Graph**(Graph &G)

{

vertices = G.vertices;

**int** i, j;

**for**(i = 1; i <= N; i++)

{

**for**(j = 1; j <= N; j++)

distance[i][j] = G.distance[i][j];

}

}

};

**int** **MINIMUM**(**int** a, **int** b)

{

**return** a < b ? a : b;

}

**void** **Print**(Graph G)

{

**int** i, j;

**for**(i = 1; i <= N; i++)

{

**for**(j = 1; j <= N; j++)

cout<< right << setw(8) << G.distance[i][j];

cout<<**endl**;

}

cout<<**endl**;

}

Graph **EXTEND\_SHORTEST\_PATHS**(Graph L, Graph W)

{

**int** i, j, k, n = W.vertices;

Graph ret(n);

**for**(i = 1; i <= n; i++)

{

**for**(j = 1; j <= n; j++)

{

ret.distance[i][j] = UNV;

**for**(k = 1; k <= n; k++)

{

**if**(L.distance[i][k] != UNV && W.distance[k][j] != UNV)

ret.distance[i][j] = MINIMUM(ret.distance[i][j], L.distance[i][k] + W.distance[k][j]);

}

}

}

**return** ret;

}

**void** **FLOYD\_WARSHALL**(Graph G)

{

**int** i,j,k;

**int** temp;

**int** dist[N+1][N+1];

**int** path[N+1][N+1];

**for**(i=1; i <= G.vertices; i++)

{

**for**(j=1; j <= G.vertices; j++)

{

path[i][j] = j;

dist[i][j] = G.distance[i][j];

}

}

**for**(k=1; k <= G.vertices; k++)

{

**for**(i=1; i <= G.vertices; i++)

{

**for**(j=1; j <= G.vertices; j++)

{

**if**(k == 1)

{

**if**(i == j || G.distance[i][j] == UNV) path[i][j] = UNV;

**else** path[i][j] = i;

}

**else**

{

**if**(dist[i][j] > dist[i][k] + dist[k][j]) path[i][j] = path[k][j];

}

**if** (dist[i][k] == UNV || dist[k][j] == UNV)

temp = UNV;

**else**

temp = dist[i][k] + dist[k][j];

**if** (dist[i][j] > temp)

{

dist[i][j] = temp;

}

}

}

**for**(i = 1; i <= N; i++)

{

**for**(j = 1; j <= N; j++)

cout<< right << setw(8) << dist[i][j];

cout<<**endl**;

}

cout<<**endl**;

**for**(i = 1; i <= N; i++)

{

**for**(j = 1; j <= N; j++)

cout<< right << setw(8) << path[i][j];

cout<<**endl**;

}

cout<<**endl**;

}

}

**int** **main**()

{

Graph G(N);

G.distance[1][5] = -1;

G.distance[2][1] = 1;

G.distance[2][4] = 2;

G.distance[3][2] = 2;

G.distance[3][6] = -8;

G.distance[4][1] = -4;

G.distance[4][5] = 3;

G.distance[5][2] = 7;

G.distance[6][2] = 5;

G.distance[6][3] = 10;

Print(G);

//SLOW\_ALL\_PAIRS\_SHORTEST\_PATHS(G);

//FASTER\_ALL\_PAIRS\_SHORTEST\_PATHS(G);

FLOYD\_WARSHALL(G);

**return** 0;

}

（4）25.2-4如前所述，Floyd+Warshall算法的空间需求为θ(n3)，因为要计算dij(k)其中i,j,k=1，2,…，n.请证明下面所列出的去掉所有上标的算法是正确的，从而将Floyd-Warshall算法的空间需求降低到θ(n2)

FLOYD-WARSHALL

1 n=W.rows

2 D=W

3 for k=1 to n

4 for i=1 to n

5 for j=1 to n

6 dij = min(dij, dik+dkj)

7 return D

在有上标时，递推方程为，如果去掉上标，则我们需要计算：







包含{1,2,...,k-1}中所有顶点的最短路径，

而表示从i到j的包含{1,2,...,k}中的所有中间顶点的最短路径，显然中间节点是不可能取k的，否则在这条最短路径上会有一条环路。因此，所以可以去掉所有的上标，将Floyd-Warshall算法的空间需求降低到θ(n2)。