Dynamic Programming - Graphs

Advanced Algorithms – Assign#4

Henri Medeiros dos Reis

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \times 2 \times 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 .$$

Analyze the running time of your algorithm.

b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

c. Include a dry run of your algorithm with some sample data and show your work.

d. Implement the algorithm in a programming language of your choice, and be sure to include screenshots of sample executions of your program in a test case report.

Submission:
Please upload your solution files to d2l by the due date.

Changes from floyd-warshall:

1-) the price change from a currency to itself is 1, no longers 0

2-) we want to maximize profit instead of minimize distance

3-) the weight is a multiplication instead of a addition

4-) want to return weather the initial start point can go thrugh other nodes on the graph and end up with a larger value

From Floyd-warshall we previously had

All Pairs Floyd-Warshall (G, w)

```
for s: 1 → n
  for t: 1 → n
    if st ∈ E, then D(0,s,t) = w(s,t)
    else, D(0,s,t) = ∞

for i: 1 → n
  for s: 1 → n
    for t: 1 → n
      sum = D(i,k) + D(k,j)
      if (sum < D(i,j))
        D(i,j) = sum
return D(n, ·, ·)
```

Making the changes gives

```
Arbitrage (G, w, starting)
    for s : 1 -> n
        for t : 1 -> n
            if st ∈ E, then D(0,s,t) = w(s,t)
            else , D(0,s,t) = 0   # needed change, since maximizing
                                    on no negative weights
    for k : 1 -> n
        for i : 1 -> n
            for j : 1 -> n
                if i ≠ k and j ≠ k # fix the colum and row
                  since diagonals may no longer be 1, causing cycles
                mul = D(i,k) D(k,j)
                if ( mul > D(i,j))
                    D(i,j) = mul
    if any element in diagonal > 1 return true
```

=> run time is $O(n^3)$, just like floyd-warshal

2-) in order to be able to print the sequence, it is needed to keep track where we are coming from, the previous step. So adding that change gives

```
Arbitrage_print (G, w, starting) {
    for s : 1 -> n
        for t : 1 -> n
            if st ∈ E
                D(0,s,t) = w(s,t)
                prev(s,t) = t
```

else , $D(0,s,t) = 0$ # needed change, since maximizing an no negative weights

```
for k : 1 -> n
    for i : 1 -> n
        for j : 1 -> n
            if i≠k and j≠k
            mul = D(i,k) D(k,j)
            if ( mul > D(i,j))
                D(i,j) = mul
                prev(i,j) = prev(i,k)
```

if any element in diagonal > 1 print ("there exists")
else return
=> recursion print on a diagonal that is large than 1
print_sol (starting, ending, prev)
}

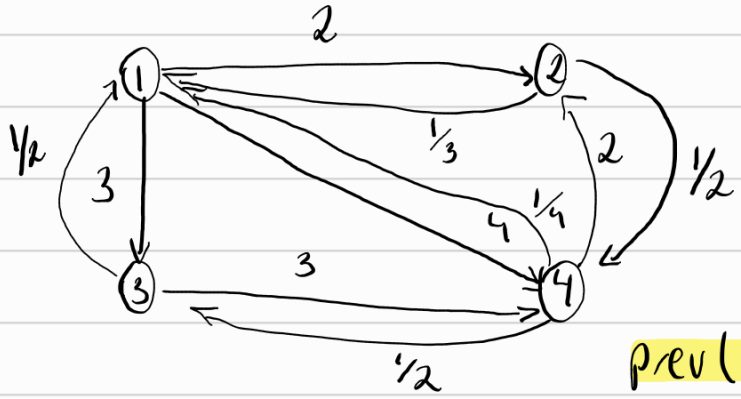
```
print_sol(starting, ending, my_matrix) {

    print(starting)
    if ( my_matrix(starting, ending) = ending)
        print(ending) ; return
    else
        middle = my_matrix(starting, ending)
        print_sol ( middle , ending, my_matrix)

}
```

=> $O(n^3)$ 3 nested for loops

3-) Dry run



prev(i,j) = prev(i,k)

/6

D_0 (i ↓, j →)

	V_1	V_2	V_3	V_4
V_1	1	2	3	4
V_2	1/3	1	0	1/2
V_3	1/2	0	1	3
V_4	1/4	2	1/2	1

Prev

	V_1	V_2	V_3	V_4
V_1	V_1	V_2	V_3	V_4
V_2	V_1	V_2	V_3	V_4
V_3	V_1	V_2	V_3	V_4
V_4	V_1	V_2	V_3	V_4

D_1

	V_1	V_2	V_3	V_4
V_1	1	2	3	4
V_2	1/3	1	1	4/3
V_3	1/2	1	3/2	3
V_4	1/4	2	3/4	1

Prev

	V_1	V_2	V_3	V_4
V_1	V_1	V_2	V_3	V_4
V_2	V_1	V_2	V_1	V_1
V_3	V_1	V_1	V_1	V_4
V_4	V_1	V_2	V_1	V_4

D_2

	V_1	V_2	V_3	V_4
V_1	1	2	3	4
V_2	1/3	1	1	4/3
V_3	1/2	1	3/2	3
V_4	2/3	2	2	8/3

Prev

	V_1	V_2	V_3	V_4
V_1	V_1	V_2	V_3	V_4
V_2	V_1	V_2	V_1	V_1
V_3	V_1	V_1	V_1	V_4
V_4	V_2	V_2	V_1	V_2

D_3

	V_1	V_2	V_3	V_4
V_1	3/2	3	3	9
V_2	1/2	1	1	3
V_3	1/2	1	3/2	3
V_4	1	2	2	6

prev

	V_1	V_2	V_3	V_4
V_1	V_3	V_3	V_3	V_3
V_2	V_1	V_2	V_1	V_1
V_3	V_1	V_1	V_1	V_4
V_4	V_1	V_2	V_1	V_1

D_4

	V_1	V_2	V_3	V_4
V_1	9	18	18	9
V_2	3	6	6	3
V_3	3	6	6	3
V_4	1	2	2	6

prev

	V_1	V_2	V_3	V_4
V_1	V_3	V_3	V_3	V_3
V_2	V_1	V_1	V_1	V_1
V_3	V_4	V_4	V_4	V_4
V_4	V_1	V_2	V_1	V_1

we can see the diagonals are large than 1, such as V_2, V_2

let's print

```
print_sol(v2, v2, prev)
    print(v2)                    V2
    middle = v1
print_sol(v1, v2, prev)
    print(v1)                    V1
    middle = v3
print_sol(v3, v2, prev)
    pri-(v3)                     V3
    middle = v4
print_sol(v4, v2, prev)
    print(v4)                    V4
    (v4, v2) = v2
    print(v2)                    V2
```

final answer ⇒ $V_2 \to V_1 \to V_3 \to V_4 \to V_2$

$\hookrightarrow 1/3 \quad \hookrightarrow 3 \quad \hookrightarrow 3 \quad \hookrightarrow 2 \quad = 6 > 1$