CSEP 521
Applied Algorithms

Richard Anderson

Lecture 7

Dynamic Programming

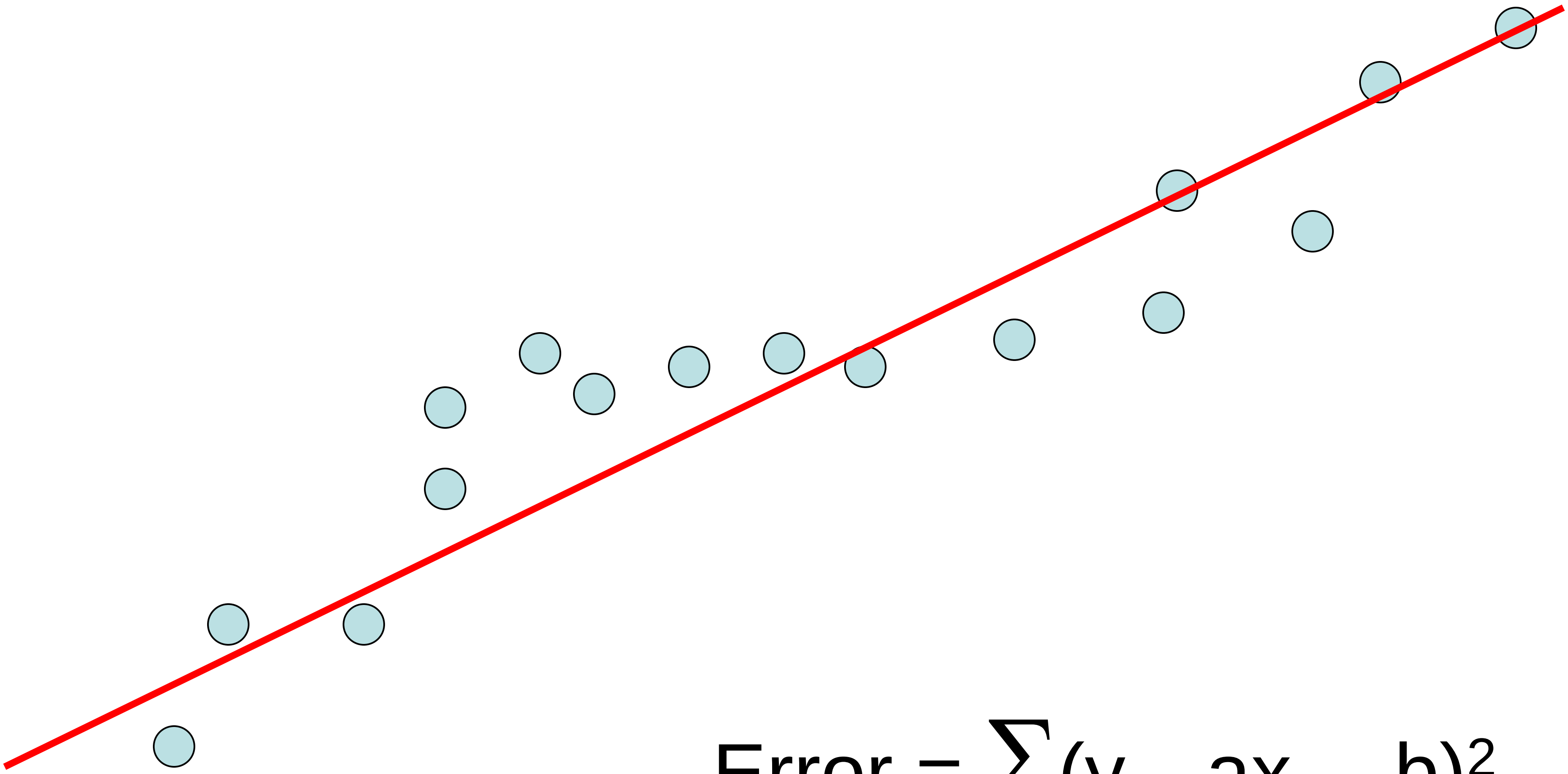
Announcements

- Reading for this week
  - 6.1-6.8

Review from last week

Weighted Interval Scheduling

Optimal linear interpolation



$$\text{Error} = \sum (y_i - ax_i - b)^2$$

Subset Sum Problem

- Let $w_1,\ldots,w_n = \{6, 8, 9, 11, 13, 16, 18, 24\}$
- Find a subset that has as large a sum as possible, without exceeding 50

Counting electoral votes

Dynamic Programming Examples

- Examples
  - Optimal Billboard Placement
    - Text, Solved Exercise, Pg 307
  - Linebreaking with hyphenation
    - Compare with HW problem 6, Pg 317
  - String approximation
    - Text, Solved Exercise, Page 309

Billboard Placement

- Maximize income in placing billboards
  - $b_i = (p_i, v_i)$, v_i: value of placing billboard at position p_i
- Constraint:
  - At most one billboard every five miles
- Example
  - {(6,5), (8,6), (12, 5), (14, 1)}

Design a Dynamic Programming Algorithm for Billboard Placement

- Compute Opt[1], Opt[2], . . ., Opt[n]
- What is Opt[k]?

Input $b_1, \ldots, b_n$, where $b_i = (p_i, v_i)$, position and value of billboard i

Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from sub problems?

Input $b_1, \ldots, b_n$, where bi = (p_i, v_i), position and value of billboard i

Solution

```
j = 0;               // j is five miles behind the current position
                      // the last valid location for a billboard, if one placed at P[k]
for k := 1 to n
          while (P[ j ] < P[ k ] – 5)
                    j := j + 1;
          j := j – 1;
          Opt[ k]  = Max(Opt[ k-1] , V[ k ] + Opt[ j ]);
```

Optimal line breaking and hyphenation

- Problem: break lines and insert hyphens to make lines as balanced as possible
- Typographical considerations:
  - Avoid excessive white space
  - Limit number of hyphens
  - Avoid widows and orphans
  - Etc.

Penalty Function

- Pen(i, j) – penalty of starting a line a position i, and ending at position j

Opt-i-mal line break-ing and hyph-en-a-tion is com-put-ed with dy-nam-ic pro-gram-ming

- Key technical idea
  - Number the breaks between words/syllables

String approximation

- Given a string S, and a library of strings B = $\{b_1, \ldots b_m\}$, construct an approximation of the string S by using copies of strings in B.

B = {abab, bbbaaa, ccbb, ccaacc}

S = abaccbbbaabbccbbccaabab

Formal Model

- Strings from B assigned to non-overlapping positions of S
- Strings from B may be used multiple times
- Cost of δ for unmatched character in S
- Cost of γ for mismatched character in S
  - MisMatch(i, j) – number of mismatched characters of b_j, when aligned starting with position i in s.

Design a Dynamic Programming Algorithm for String Approximation

- Compute Opt[1], Opt[2], . . ., Opt[n]
- What is Opt[k]?

Target string $S = s_1s_2\ldots s_n$
Library of strings $B = \{b_{1,}\ldots,b_m\}$
MisMatch(i,j) = number of mismatched characters with b_j when aligned starting at position i of S.

$Opt[k] = fun(Opt[0],\ldots,Opt[k-1])$

- How is the solution determined from sub problems?

Target string $S = s_1s_2\ldots s_n$
Library of strings $B = \{b_1,\ldots,b_m\}$
$MisMatch(i,j)$ = number of mismatched characters with b_j when aligned starting at position i of S.

Solution

```
for i := 1 to n
        Opt[k] = Opt[k-1] + δ;
        for j := 1 to |B|
                p = i – len(b_j);
                Opt[k] = min(Opt[k],  Opt[p-1] + γ MisMatch(p, j));
```

Longest Common Subsequence

Longest Common Subsequence

- $C=c_1 \ldots c_g$ is a subsequence of $A=a_1 \ldots a_m$ if C can be obtained by removing elements from A (but retaining order)
- LCS(A, B): A maximum length sequence that is a subsequence of both A and B

`ocurranec`

`occurrence`

`attacggct`

`tacgacca`

Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

KRUSTYTHECLOWN

String Alignment Problem

- Align sequences with gaps

```
CAT TGA  AT

CAGAT AGGA
```

- Charge δ_x if character x is unmatched
- Charge γ_{xy} if character x is matched to character y

Note: the problem is often expressed as a minimization problem,
with $\gamma_{xx} = 0$ and $\delta_x > 0$

LCS Optimization

- $A = a_1 a_2 \ldots a_m$
- $B = b_1 b_2 \ldots b_n$

- Opt[j, k] is the length of LCS($a_1 a_2 \ldots a_j$, $b_1 b_2 \ldots b_k$)

Optimization recurrence

If $a_j = b_k$, Opt[j,k] = 1 + Opt[j-1, k-1]

If $a_j \neq b_k$, Opt[j,k] = max(Opt[j-1,k], Opt[j,k-1])

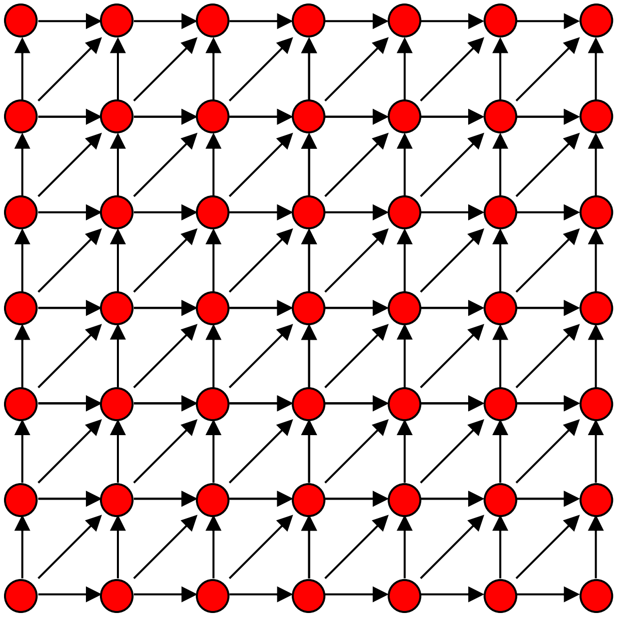
Give the Optimization Recurrence for the String Alignment Problem

- Charge δ_x if character x is unmatched
- Charge γ_{xy} if character x is matched to character y

Opt[j, k] =

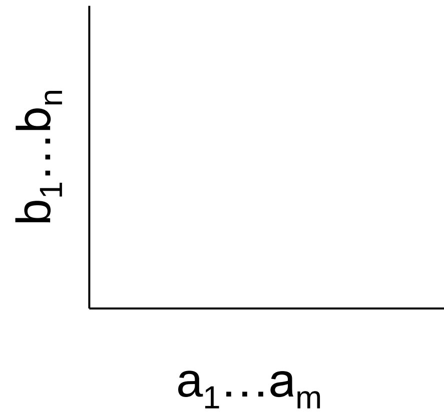
Let $a_j = x$ and $b_k = y$
Express as minimization

Dynamic Programming Computation



Code to compute Opt[j,k]

Storing the path information




```
A[1..m],  B[1..n]
for i := 1 to m     Opt[i, 0] := 0;
for j := 1 to n     Opt[0,j] := 0;
Opt[0,0] := 0;
for i := 1 to m
            for j := 1 to n
                        if A[i] = B[j]  {  Opt[i,j] := 1 + Opt[i-1,j-1];  Best[i,j] := Diag; }
                        else if Opt[i-1, j] >= Opt[i, j-1]
                                    {  Opt[i, j] := Opt[i-1, j], Best[i,j] := Left; }
                        else        {  Opt[i, j] := Opt[i, j-1], Best[i,j] := Down; }
```

How good is this algorithm?

- Is it feasible to compute the LCS of two strings of length 100,000 on a standard desktop PC? Why or why not.

Observations about the Algorithm

- The computation can be done in O(m+n) space if we only need one column of the Opt values or Best Values

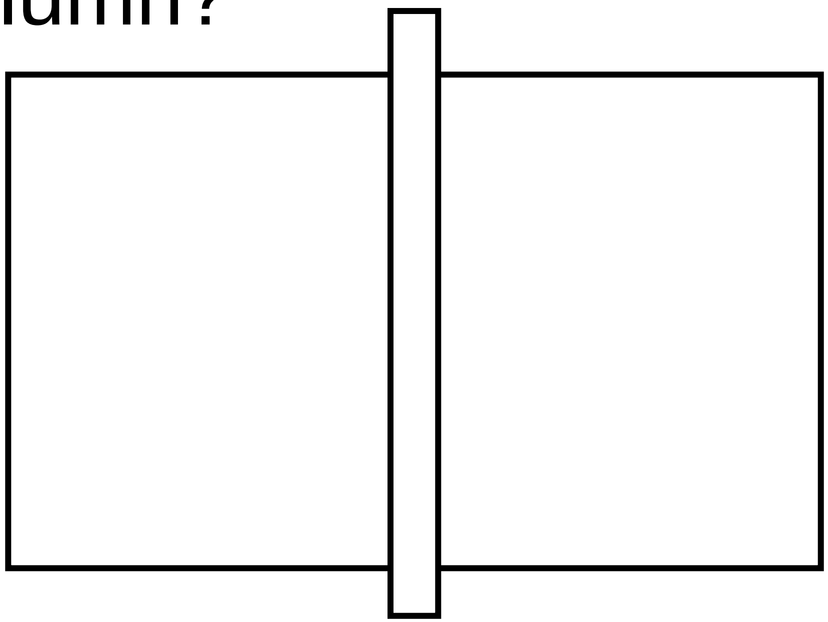
- The algorithm can be run from either end of the strings

Computing LCS in O(nm) time and O(n+m) space

- Divide and conquer algorithm
- Recomputing values used to save space

Divide and Conquer Algorithm

- Where does the best path cross the middle column?



- For a fixed i, and for each j, compute the LCS that has a_i matched with b_j

Constrained LCS

- $LCS_{i,j}(A,B)$: The LCS such that
  - $a_1,\ldots,a_i$ paired with elements of $b_1,\ldots,b_j$
  - $a_{i+1},\ldots a_m$ paired with elements of $b_{j+1},\ldots,b_n$

- $LCS_{4,3}$(abbacbb, cbbaa)

A = RRSSRTTRTS

B=RTSRRSTST

Compute $LCS_{5,0}(A,B)$, $LCS_{5,1}(A,B)$,…,$LCS_{5,9}(A,B)$

A = RRSSRTTRTS

B=RTSRRRSTST

Compute $LCS_{5,0}(A,B)$, $LCS_{5,1}(A,B)$,…,$LCS_{5,9}(A,B)$

j	left	right
0	0	4
1	1	4
2	1	3
3	2	3
4	3	3
5	3	2
6	3	2
7	3	1
8	4	1
9	4	0

Computing the middle column

- From the left, compute $LCS(a_1 \ldots a_{m/2}, b_1 \ldots b_j)$
- From the right, compute $LCS(a_{m/2+1} \ldots a_m, b_{j+1} \ldots b_n)$
- Add values for corresponding j's

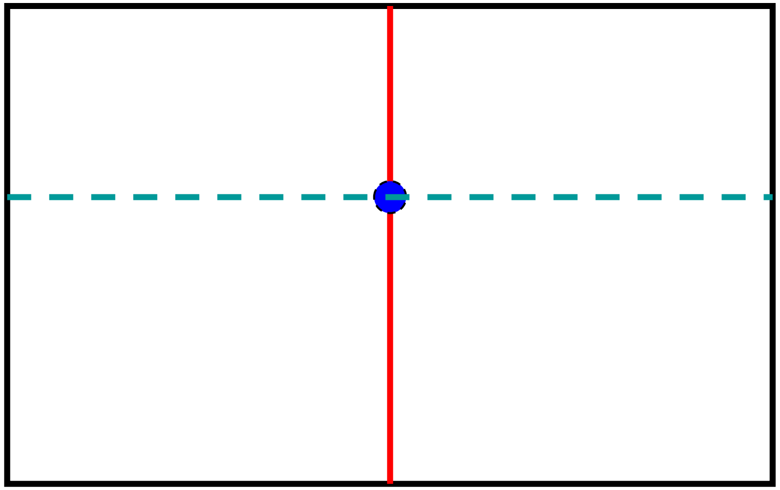
- Note – this is space efficient

Divide and Conquer

- $A = a_1, \ldots, a_m$ $\quad\quad\quad\quad$ $B = b_1, \ldots, b_n$
- Find j such that
  - LCS($a_1 \ldots a_{m/2}$, $b_1 \ldots b_j$) and
  - LCS($a_{m/2+1} \ldots a_m$, $b_{j+1} \ldots b_n$) yield optimal solution
- Recurse

Algorithm Analysis

- $T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm$



Prove by induction that $T(m,n) \le 2cmn$

Memory Efficient LCS Summary

- We can afford O(nm) time, but we can't afford O(nm) space
- If we only want to compute the length of the LCS, we can easily reduce space to O(n+m)
- Avoid storing the value by recomputing values
  - Divide and conquer used to reduce problem sizes

Shortest Paths with Dynamic Programming

Shortest Path Problem

- Dijkstra's Single Source Shortest Paths Algorithm
  - O(mlog n) time, positive cost edges
- General case – handling negative edges
- If there exists a negative cost cycle, the shortest path is not defined
- Bellman-Ford Algorithm
  - O(mn) time for graphs with negative cost edges

Lemma

- If a graph has no negative cost cycles, then the shortest paths are simple paths

- Shortest paths have at most n-1 edges

Shortest paths with a fixed number of edges

- Find the shortest path from v to w with exactly k edges

Express as a recurrence

- $Opt_k(w) = \min_x [Opt_{k-1}(x) + c_{xw}]$
- $Opt_0(w) = 0$ if $v=w$ and infinity otherwise

Algorithm, Version 1

```
foreach w
        M[0, w] = infinity;
M[0, v] = 0;
for i = 1 to n-1
        foreach w
                M[i, w] = min_x(M[i-1,x] + cost[x,w]);
```

Algorithm, Version 2

```
foreach w
        M[0, w] = infinity;
M[0, v] = 0;
for i = 1 to n-1
        foreach w
                M[i, w] = min(M[i-1, w], min_x(M[i-1,x] + cost[x,w]))
```

Algorithm, Version 3

```
foreach w
        M[w] = infinity;
M[v] = 0;
for i = 1 to n-1
        foreach w
                M[w] = min(M[w], min_x(M[x] + cost[x,w]))
```

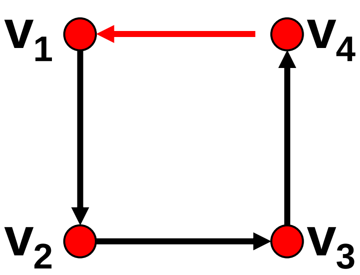
Correctness Proof for Algorithm 3

- Key lemma – at the end of iteration i, for all w, M[w] <= M[i, w];

- Reconstructing the path:
  - Set P[w] = x, whenever M[w] is updated from vertex x

If the pointer graph has a cycle, then the graph has a negative cost cycle

- If P[w] = x then M[w] >= M[x] + cost(x,w)
  - Equal when w is updated
  - M[x] could be reduced after update
- Let v_1, v_2,…v_k be a cycle in the pointer graph with (v_k,v_1) the last edge added
  - Just before the update
    - $M[v_j] >= M[v_{j+1}] + cost(v_{j+1}, v_j)$ for $j < k$
    - $M[v_k] > M[v_1] + cost(v_1, v_k)$
  - Adding everything up
    - $0 > cost(v_1,v_2) + cost(v_2,v_3) + \dots + cost(v_k, v_1)$

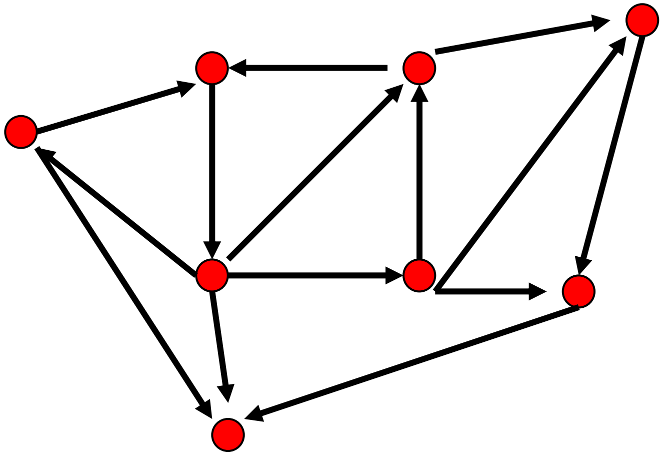


Negative Cycles

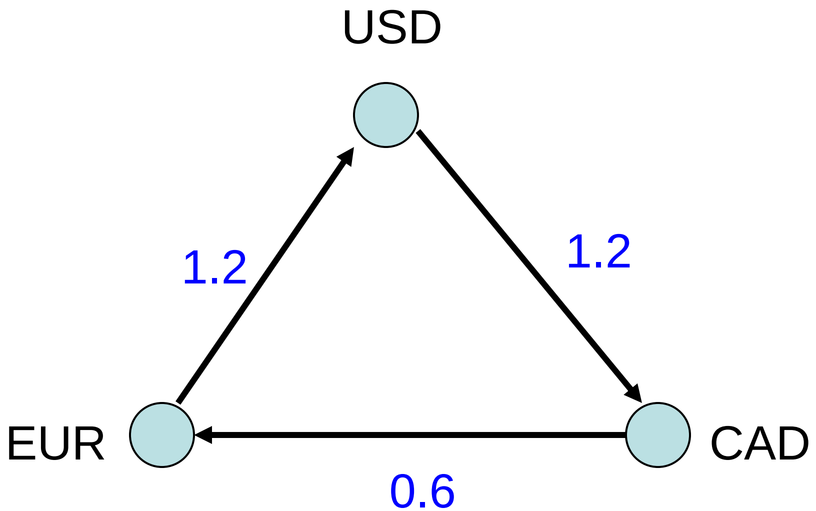
- If the pointer graph has a cycle, then the graph has a negative cycle
- Therefore: if the graph has no negative cycles, then the pointer graph has no negative cycles

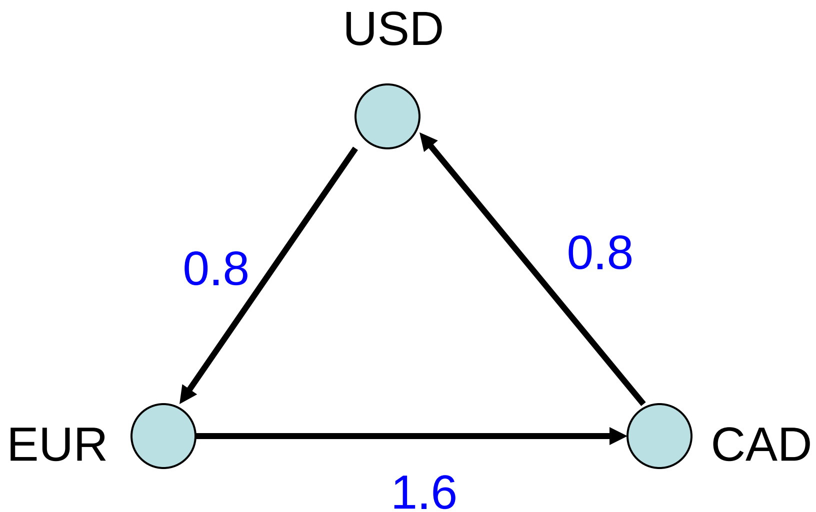
Finding negative cost cycles

- What if you want to find negative cost cycles?



Foreign Exchange Arbitrage





	USD	EUR	CAD
USD	------	0.8	1.2
EUR	1.2	------	1.6
CAD	0.8	0.6	-----