# Dynamic Programming#2

# Longest Common Subsequences

# Subsequences

- Suppose we have a sequence $X = < x_1,x_2,\ldots\ldots\ldots,x_m>$ of elements over a finite set S. Then,
  - def: a sequence $Z = < z_1,z_2,\ldots\ldots..,z_k>$ over S is called a subsequence of X if and only if it can be obtained from X by deleting elements
  - i.e. obtained without changing the order of the remaining elements
  - i.e. this means there exist indices $i_1<i_2 <\ldots<i_k$ such that

    $z_a = x_{ia}$ for all a in the range $1<= a <= k$.

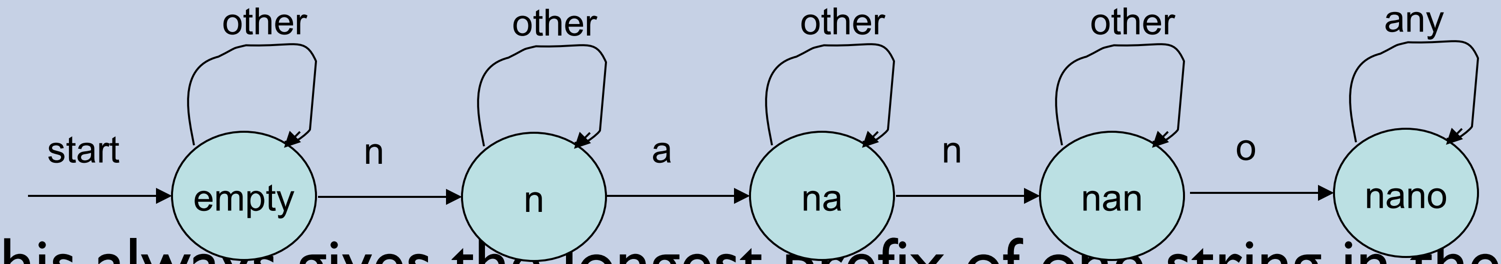
    - e.g. the following are all subsequences of "president": *pred, sdn, predent*
    - e.g. ABD is a subsequence of ABCDEF.

# Subsequences….

- Alternate definition
  - Given a sequence $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$ of elements over a finite set S, another sequence $Z = < z_1, z_2, \ldots\ldots.., z_k >$ is a subsequence of X if there exists a strictly increasing sequence $< z_1, z_2, \ldots\ldots.., z_k >$ of indices X such that for all j = 1,2,3,….k, we have $X_{ij} = z_{j.}$
    - e.g. Z = <B, C, D B> is a sequence of X = <A,B,C,B,D,A,B> with corresponding index sequence <2,3,5,7>.

# Determining Subsequences

- Given the two strings say "nano" and "nematode knowledge", how to find the whether one pattern appears in the other ?
- Write an algorithm based on the following




- This always gives the longest prefix of one string in the other…
- But, the issue is what if one pattern does not occur fully in the other….
  – The method in the automata above does not solve the problem, then.

# Common Subsequence

- Suppose that X and Y are two sequences over a set S.
- We say that Z is a common subsequence of X and Y if and only if
  - Z is a subsequence of X and Z is a subsequence of Y
  - e.g. given the subsequences viz.
    - P = cbabca, Q= bcabac and R= abcade
      - The set of CS of P and Q is **{Φ, a, aa, ab, aba, abc, ac, b, ba, baa, bab, baba, babc, bac, bb, bba, bbc, bc, bca, c, ca, caa, cab, caba, cabc, cac, cb, cba, cbac, cbc, cc}**
      - The set of CS of P and R is: **{ Φ, a, aa, ab, aba, abc, abca, ac, aca, b, ba, bc, bca, c, ca}**
- The longest amongst these is the Longest Common Subsequence

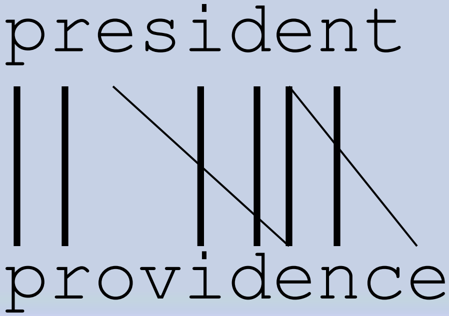
# Longest Common Subsequence (LCS)

- Problem: Given sequences x[1..m] and y[1..n], find a longest common subsequence of both.

Note “a” used here and not “the”

- e.g. x=ABCBDAB and y=BDCABA,
  - BCA is a common subsequence and
  - BCBA and BDAB are two LCSs

- e.g. x = president and y = providence
  - pree and den are common subsequence and
  - priden is an LCS.



# LCS : Another view

- Explore the Edit distance between S1 and S2
  - defined as the number of operations required to transform one of them into the other. e.g.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| S1: | a | b | c | d | a | c | e | |
| S2: | | b | a | d | c | a | b | e |

  - Length LCSS = 4
  - Edit Distance = 3(remove) +3(add) = 6
  - Which are removed ? Which are added ?

# Motivation & Applications

- Numerous Applications. Most striking ones……
- Biological Applications
  - compare two DNA sequences
    - a strand of DNA consists of string of molecules called bases
    - the possible bases are ADENINE, GUANINE, CYTOSINE, THYMINE (A,G,C,T)
      - thus a DNA sequence is defined over the alphabet (A,G,C,T)
    - e.g. for two different organisms, their DNA sequence may be
      - $S_1$ = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA and
      - $S_2$ = GTCGTTCGGAATGCCGTTGCTCTGTAAA

# Motivation & Applications….

- Biological Applications
  - The problem at hand is to determine how similar the two organisms are
    - i.e. compare two strings $S_1$ and $S_2$
  - Solution strategies
    - explore whether one is a substring of the other ? OR
    - the number of changes required to turn one string into the other
      - should be minimal OR
    - find a third strand $S_3$ s.t. the bases in $S_3$ appear in each of $S_1$ and $S_2$
      - preferably consecutively but at least in the same order….
      - the longer the strand $S_3$, more similar the two organisms are
      - This essentially is to determine the longest common subsequence in $S_1$ and $S_2$

# Motivation & Applications….

- File Comparisons
  - e.g. unix program "diff"
  - works by finding the LCS of the lines of the two files
    - i.e. anyline in the subsequence that would not have changed
- Screen redisplay
  - Used by text editors like "emacs"
    - Especially when used with slow dial-in terminals
- In passing, note that,
  - Longest common substring problem is different from LCS problem, that we attack here.

# LCS Solution Alternatives

- Various alternatives
  - Applying Brute force
  - Writing a recurrence equation & devising a recursive solution
  - Writing a recurrence equation & framing a dynamic programming solution

# Brute force solution

The picture can't be displayed.

- The basic approach
  - For every subsequence of x (of length m), check if it is a subsequence of y (of length n).
- Suppose one has N sequences of lengths $n_1, n_2, \ldots, n_N$.
- For a given sequence of length *m*, how many subsequences can there be ?
  - How do we solve for only two given sequences ?
- Analysis :
  - There are $2^m$ subsequences of x.
  - Each is to be checked against the subsequence y
    - this check takes $O(n)$ time, since we scan y for first element, and then scan for second element, etc.
  - The worst case running time is $O(n2^m)$.

# Brute force solution

- Thus,
  - Naive search
    - test each of the $2^{n1}$ subsequences of the first sequence against the other sequences to determine whether they are also subsequences of the remaining sequences or not
    - i.e. $2^{n1}$ $(n_2 + n_3 + n_4 + \ldots\ldots\ldots n_N)$ i.e. $O(2^{n1}\sum_{i>1} n_i)$

The picture can't be displayed.

# Alternative Solution Strategies

- Can we use divide-and-conquer to solve this problem?
- Can we use Greedy approach to solve this problem?
- We investigate these issues further…………

# Characterizing a LCS

- First question to be answered
  - Does LCs problem exhibit optimal substructure and overlapping subproblems properties ?
  - LCS indeed exhibits both these properties.
- Optimal substructure property
  - As compared to finding out LCS of strings of length m and n, it is surely easy to find the same from the two strings (m-1) and (n-1).
- Overlapping subproblems
  - the solution to main subproblem indeed depends on the solution to smaller instances
- We shall formalize these two with the help of a theorem, later……..

# Prefix of a sequence

- Motivation
  - The subproblems become simpler as the sequences become shorter.
  - Shorter sequences are conveniently described using *prefixes.*
- def:
  - For a given sequence $X = < x_1, x_2, \ldots, x_m >$, we define the $i^{th}$ prefix of the sequence X by $X_i$ as
    - $X_i = < x_1, x_2, \ldots, x_i >$ ………………..for i=0,1,2,3,….m
  - e.g. if the given sequence is X = “AABCDA” then,
    - $X_1$ = A, $X_2$ = AA, $X_3$ = AAB, $X_4$ = AABC, $X_5$ = AABCD,……..

# Revisiting Optimal Substructure

- Let $X = < x_1,x_2,\ldots,x_m>$ and $Y = < y_1,y_2,\ldots,y_n>$ be two sequences.
- Let $Z = < z_1,z_2,\ldots,z_k>$ is any LCS of X and Y.
- Then,
  - if $x_m = y_n$ then certainly $x_m = y_n = z_k$ and $Z_{k-1}$ is in $LCS(X_{m-1}, Y_{n-1})$
- Thus,
  - the problem size is reduced by one to solve it.
  - e.g. apply this approach to the two sequences GUAVA and GREATJAVA
    - The LCS is GAVA
- What is the conclusion ???

# Revisiting Optimal Substructure….

- $X_m$ and $Y_n$ end with $x_m = y_n$ and
  - LenLCS ….length of the longest common subsequence

| | | |
|---|---|---|
| $X_m$ | $x_1 \ x_2 \ \ldots \ x_{m-1}$ | $x_m$ |
| $Y_n$ | $y_1 \ y_2 \ \ldots \ y_{n-1}$ | $y_n = x_m$ |
| $Z_k$ | $z_1 \ z_2 \ldots z_{k-1}$ | $z_k = y_n = x_m$ |

- $Z_k$ is $Z_{k-1}$ followed by $z_k = y_n = x_m$ where $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$ and
- $LenLCS(m, n) = LenLCS(m-1, n-1) + 1$

# Revisiting Optimal Substructure….

- Again, let $X = < x_1, x_2, \ldots, x_m >$ and $Y = < y_1, y_2, \ldots, y_n >$ be two sequences.
- Let $Z = < z_1, z_2, \ldots, z_k >$ is any LCS of X and Y.
- Then,
  - What is the LCS if $x_m \neq y_n$ and $x_m \neq z_k$ ?
    - If $x_m \neq y_n$ then $x_m \neq z_k$ implies that Z is in LCS($X_{m-1}$ , Y)
  - What is the LCS if $x_m \neq y_n$ and $x_m \neq z_k$ ?
    - If $x_m \neq y_n$ then $y_n \neq z_k$ implies that Z is in LCS(X, $Y_{n-1}$)

# Revisiting Optimal Substructure….

- $X_m$ and $Y_n$ end with $x_m$ != $y_n$ and
  - LenLCS ….length of the longest common subsequence

| | | |
|---|---|---|
| $X_m$ | $x_1\ x_2\ \ldots\ x_{m-1}\ x_m$ | |
| $Y_n$ | $y_1\ y_2\ \ldots\ y_{n-1}$ | $y_n$ |
| $Z_k$ | $z_1\ z_2 \ldots z_{k-1}$ | $z_k$ !=$y_n$ |

| | | |
|---|---|---|
| $X_m$ | $x_1\ x_2\ \ldots\ x_{m-1}$ | $x_m$ |
| $Y_n$ | $y_1\ y_2\ \ldots\ y_{n-1}\ y_n$ | |
| $Z_k$ | $z_1\ z_2 \ldots z_{k-1}$ | $z_k$ !=$x_m$ |

- $z_k$ != $y_n$ …hence
- $Z_k$ is LCS of $X_m$ and $Y_{n-1}$

- $z_k$ != $x_m$ …hence
- $Z_k$ is LCS of $X_{m-1}$ and $Y_n$

- $LenLCS(m, n)$= max $\{LenLCS(m, n\text{-}1), LenLCS(m\text{-}1, n)\}$

# Revisiting Optimal Substructure....

- An Example
  - Let $X_n$ = ABCDEFG (n=7), $Y_m$ = BCDGK (m=5)
  - What could be the last element of the LCS end with ?
  - Two cases
    - If the LCS contains the last character as G
      - Then, $LCS(X_m, Y_n) = LCS(X_n, Y_{m-1})$
    - If the LCS does not end with the (last character) as G
      - Then, $LCS(X_m, Y_n) = LCS(X_{n-1}, Y_m)$
  - In the example above, if we remove K, then the LCS is easily determined.......

# Overlapping Subproblems

- This also exhibits overlapping subproblems property i.e. :
  - If $x_m = y_n$ then we solve the subproblem to find an element in LCS($X_{m-1}$ , $Y_{n-1}$ ) and append $x_m$
  - Then, *LenLCS*(*m*, *n*)=*LenLCS*(*m* -1, *n* -1) + 1

  - If $x_m <> y_n$ then we solve the two subproblems of finding elements in LCS($X_{m-1}$ , $Y_{n-1}$ ) and LCS($X_{m-1}$ , $Y_{n-1}$ ) and choose the longer one.
  - Then, *LenLCS*(*m*, *n*)= max {*LenLCS*(*m*, *n*-1), *LenLCS*(*m*-1, *n*)}

# Theorem: Optimal substructure of an LCS

- Let $X = < x_1, x_2, \ldots, x_m >$ and $Y = < y_1, y_2, \ldots, y_n >$ be two sequences and Let $Z = < z_1, z_2, \ldots, z_k >$ is any LCS of X and Y.
- Then,
  1. If $x_m = y_n$ then certainly $x_m = y_n = z_k$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$)
  2. If $x_m \neq y_n$ then $x_m \neq z_k$ implies that Z is an LCS of $X_{m-1}$ and Y
  3. If $x_m \neq y_n$ then $y_n \neq z_k$ implies that Z is an LCS of X and $Y_{n-1}$
- Proof:

# Devising the Recursive Solution

- Let X and Y be sequences.
- Let c[i,j] = LenLCS[i,j] be the length of an element in LCS($X_i$, $Y_j$).

c[i,j] =
LenLCS[i,j] =

| | |
|---|---|
| 0 | • if i=0 or j=0 |
| c[i-1,j-1]+1 | • if i,j>0 and $x_i = y_j$ |
| max(c[i,j-1],c[i-1,j]) | • if i,j>0 and $x_i$ <> $y_j$ |

# Devising the Recursive solution….

- Let
  - $X_i$ denote the $i^{th}$ *prefix* x[1..*i*] of x[1..*m*], and
  - $X_0$ denotes an empty prefix
  - The length of an LCS of $X_m$ and $Y_n$ be *LenLCS*(*m*, *n*)
  - Recursive formulation for computing *LenLCS*(*i*, *j*) is based on the facts observed earlier viz.
    - If $X_i$ and $Y_j$ end with the same character $x_i=y_j$,
      - the LCS must include the character. If it did not we could get a longer LCS by adding the common character.
    - If $X_i$ and $Y_j$ do not end with the same character
      - there are two possibilities:
        - ➢ either the LCS does not end with $x_i$,
        - ➢ or it does not end with $y_j$
    - Let $Z_k$ denote an LCS of $X_i$ and $Y_j$

# The recurrence equation

- Then, the recurrence relation is given by the following:

$$lenLCS(i,j) = \begin{cases} 0 & \text{if } i = 0, \text{ or } j = 0 \\ lenLCS(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_{\mathrm{i}} = y_{\mathrm{j}} \\ \max\{lenLCS(i-1, j), lenLCS(i, j-1)\} & \text{otherwise} \end{cases}$$

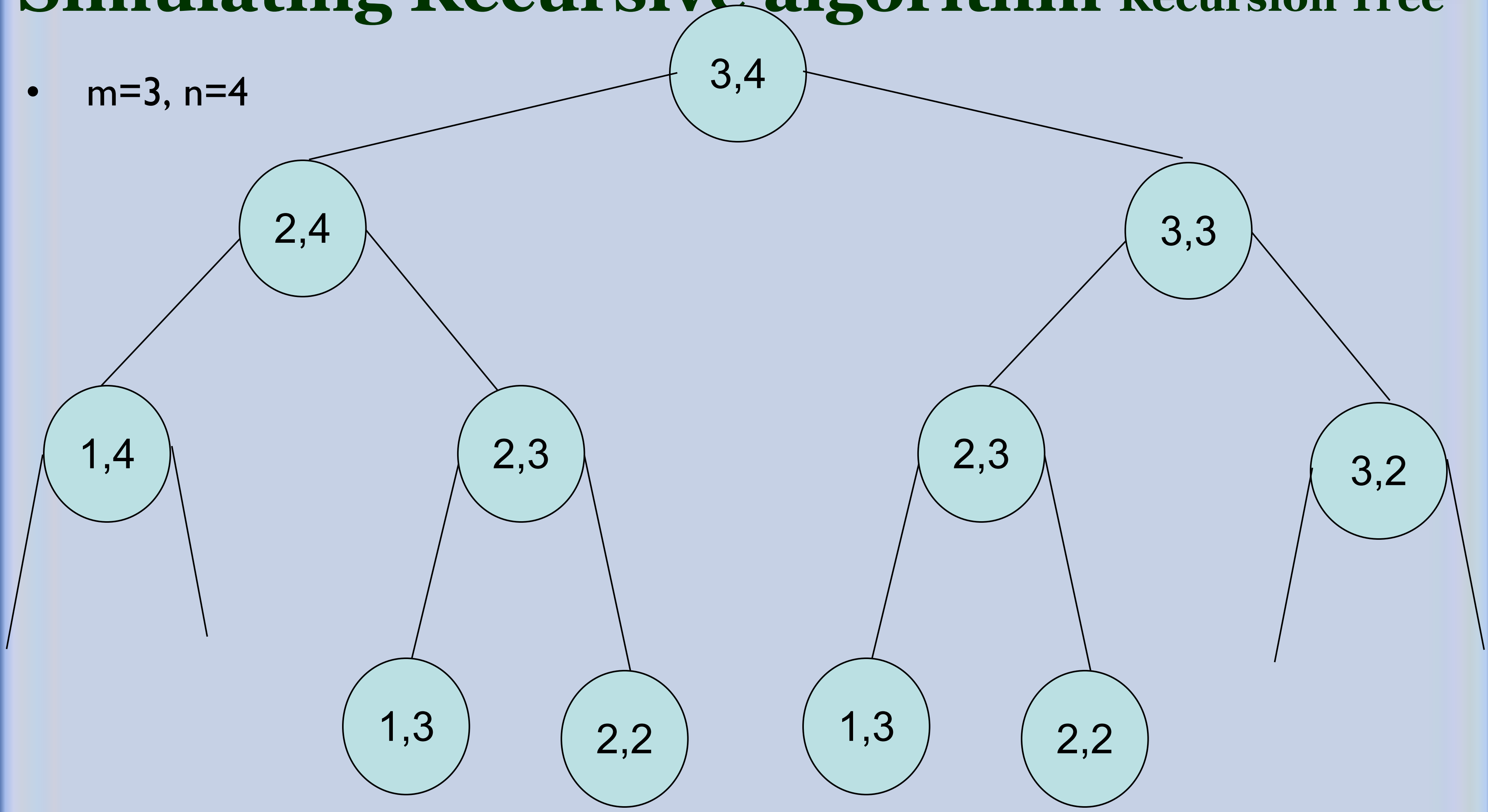
# Recursive algorithm for LCS

```
LCS (x, y, i, j)
1. If x[i] = y[j]
2.   then LenLCS[i.j] = LCS(x, y, i-1, j-1)+ 1
3.   else LenLCS[i.j]=
         max {LCS(x,y,i-1,j), LCS(x,y,i,j-1)}
```

- When does the worst case occur ?
  - When the two strings have no matching characters and so the last line always gets executed.
  - Thus, when $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.
  - So, if the bounds are the same (=n, say), the complexity is $O(2^n)$
  - Does it appear convincing with respect to other such algorithms discussed earlier in D&C?

# Simulating Recursive algorithm Recursion Tree

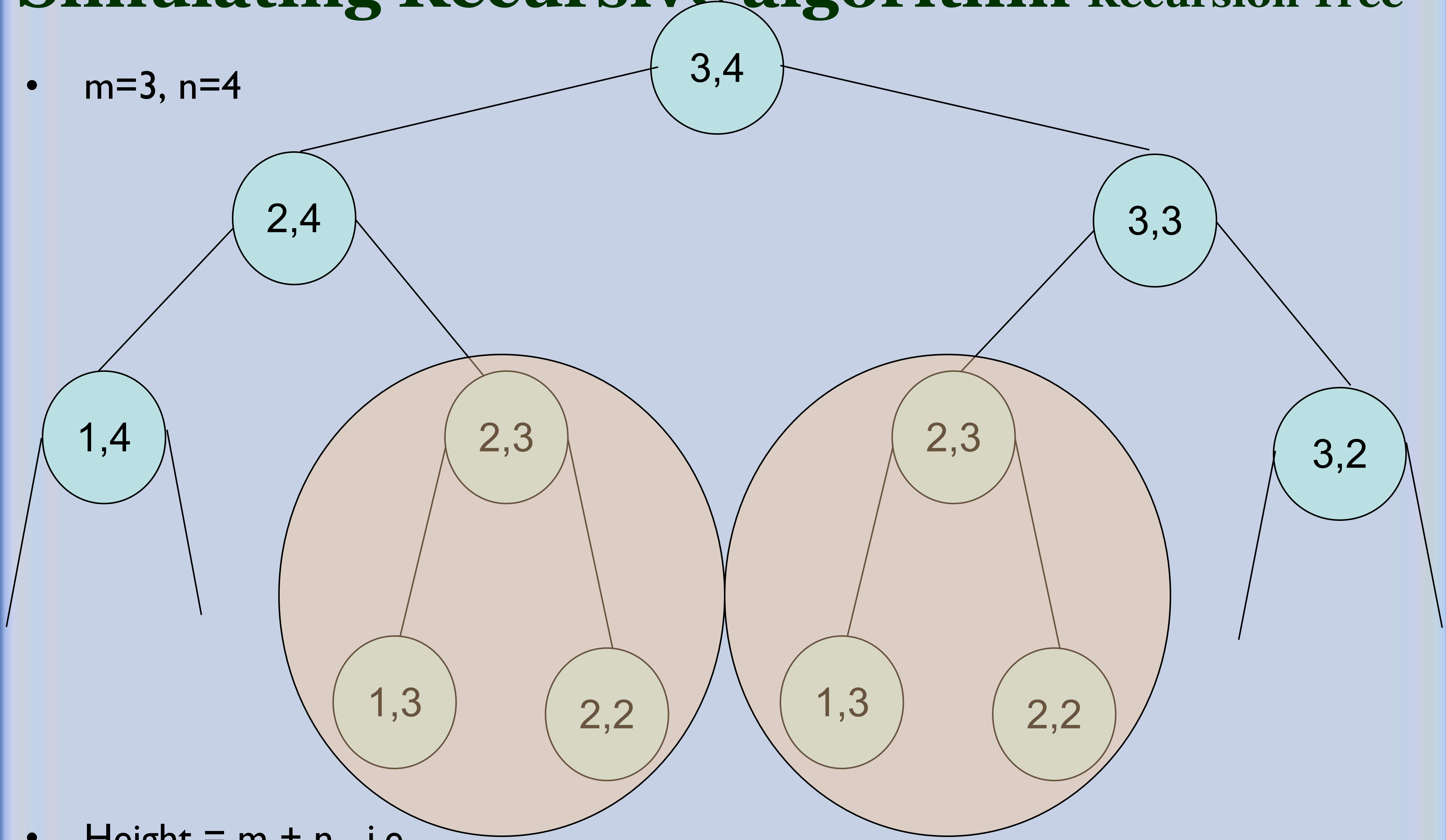
- m=3, n=4




- Height = m + n
  - but we're solving subproblems already solved

# Simulating Recursive algorithm Recursion Tree

- m=3, n=4




- Height = m + n i.e.
  - Work potentially exponential,

# Fact

- On the contrary, if the two sequences have lengths m and n
  - there are exactly (mn) subproblems
    - a pretty small number as compared to being exponential in m OR n.

# Improving the recursive approach: Memoization

- Memoization
  - after computing a solution to a subproblem, store it in a table.
  - In subsequent calls check the table to avoid redoing work.

```
LCS (x, y, i, j)
1. if LenLCS = NIL
2.  then if x[i] = y[j]
3.   then LenLCS[i.j]= LCS(x, y, i-1, j-1)+ 1
4.   else LenLCS[i.j]=
        max {LCS(x,y,i-1,j), LCS(x,y,i,j-1)}
```

# Improving the recursive approach: Memoization

- This is a top down memoization based approach
- Is more efficient
  - each call to the subproblem takes a constant time
  - we call the routine at most twice when we feel in the LenLCS array
  - there are mn entries….
    - hence, total time is O(mn)
- However, we may try for
  - a bottom-up iterative – true dynamic programming approach

# Dynamic Programming Solution

- To compute length of an element in LCS(X,Y) with |X| = m and |Y| = n, we
  - initialize first row and first column of the array c with 0.
  - calculate LenLCS[1,j] for 1 <= j <= n,

    LenLCS [2,j] for 1 <= j <= n

    LenLCS [3,j] for 1 <= j <= n

    … …
- Return LenLCS[m,n]
- How can we get an actual longest common subsequence?
  - Store in addition to the array c an array b pointing to the optimal subproblem chosen when computing LenLCS[i,j].

# Algorithm LCS-Length(X,Y)

RCS = ↖

```
m = X.Length
n = Y.Length
let b[1…m, 1…n] and LenLCS[0..m, 0..n] be new tables
for i = 1 to m LenLCS[i,0] = 0
for j = 1 to m LenLCS[j,0] = 0
for i = 1 to m
   for j = 1 to n
    if xi==yj
      LenLCS[i,j] = LenLCS[i-1,j-1] + 1
       b[i,j] = "RSA"
    elseif LenLCS[i-1,j] >= LenLCS[i,j-1]
       LenLCS[i,j]= LenLCS[i-1,j]
       b[i,j] = "↑"
    elseif LenLCS[i,j] = LenLCS[i,j-1]
       b[i,j] = " "
return LenLCS and b
```

# Algorithm LCS-Length(X,Y)…

- Whenever, we encounter a reverse slanting arrow
  - we realize that the algorithm has found an element of the LCS
  - however, we encounter the element of the LCS in the reverse order.
  - see the dry run……
- The complexity of the algorithm is O(m+n).

# Simulating LCSLength(X,Y)…

| | | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |
| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |

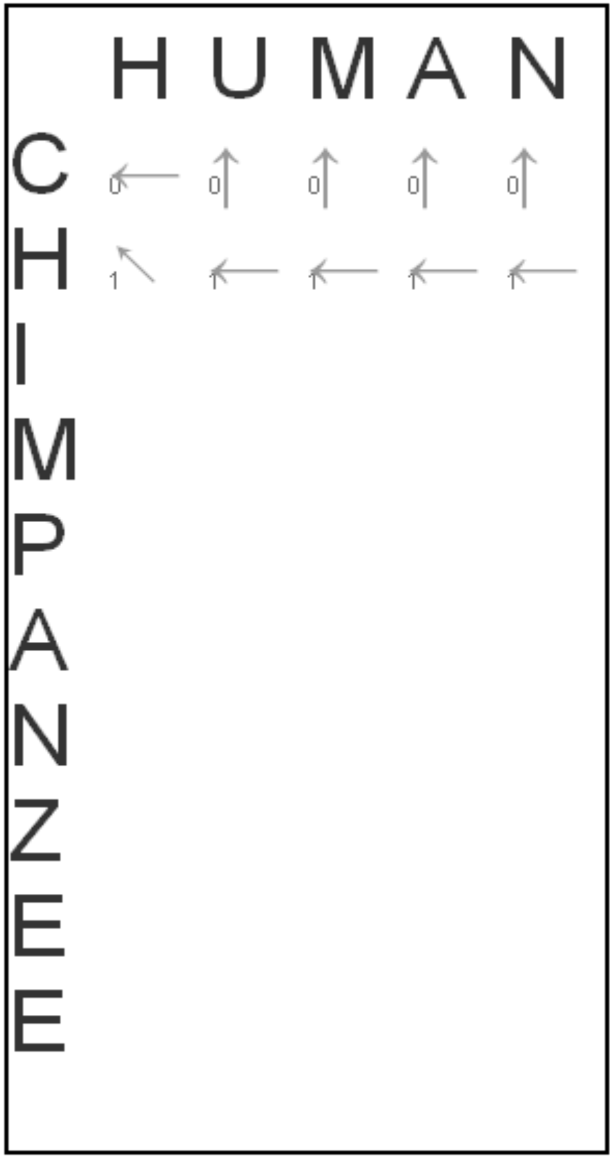
# Simulating LCSLength(X,Y)…

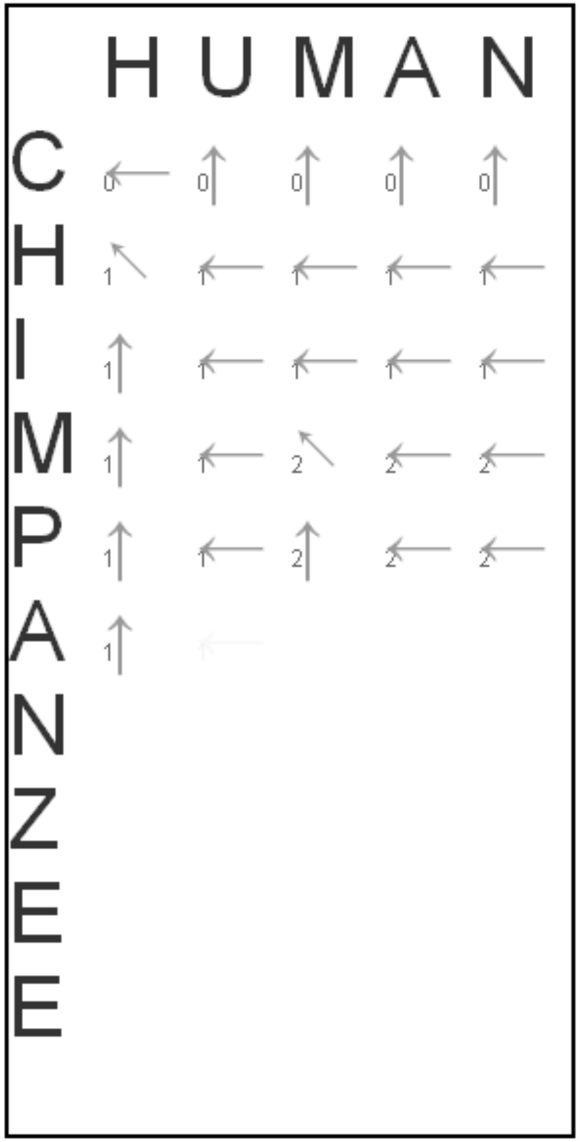
| | | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |
| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |

# Simulating LCSLength(X,Y)…

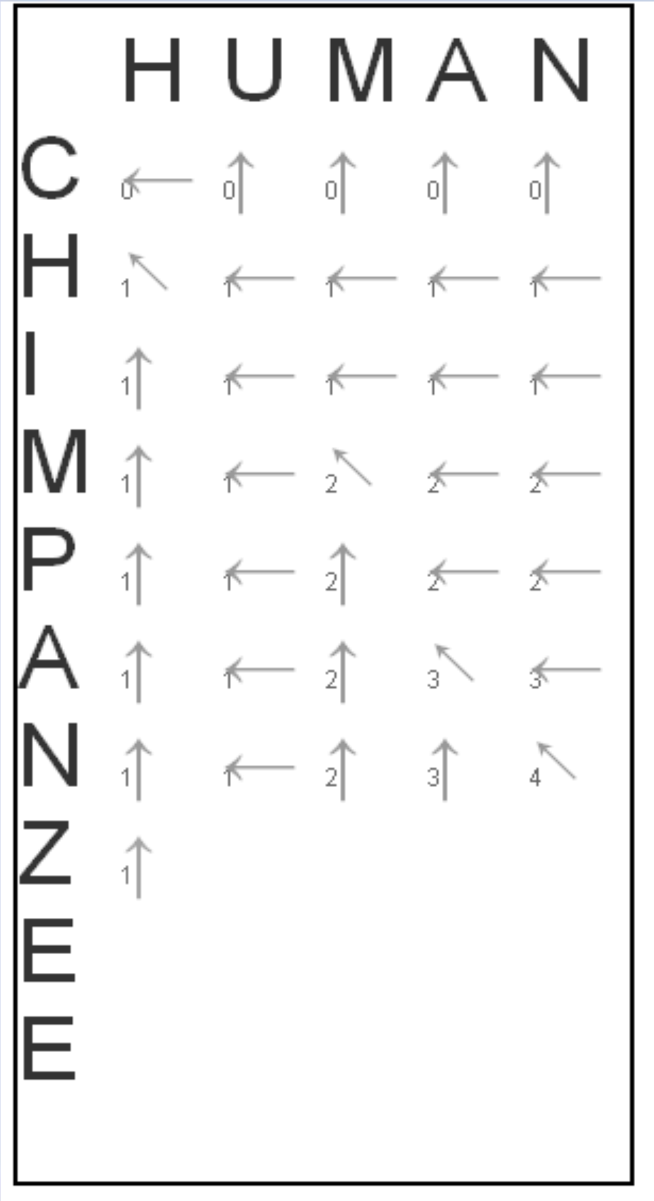
| | | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |
| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |

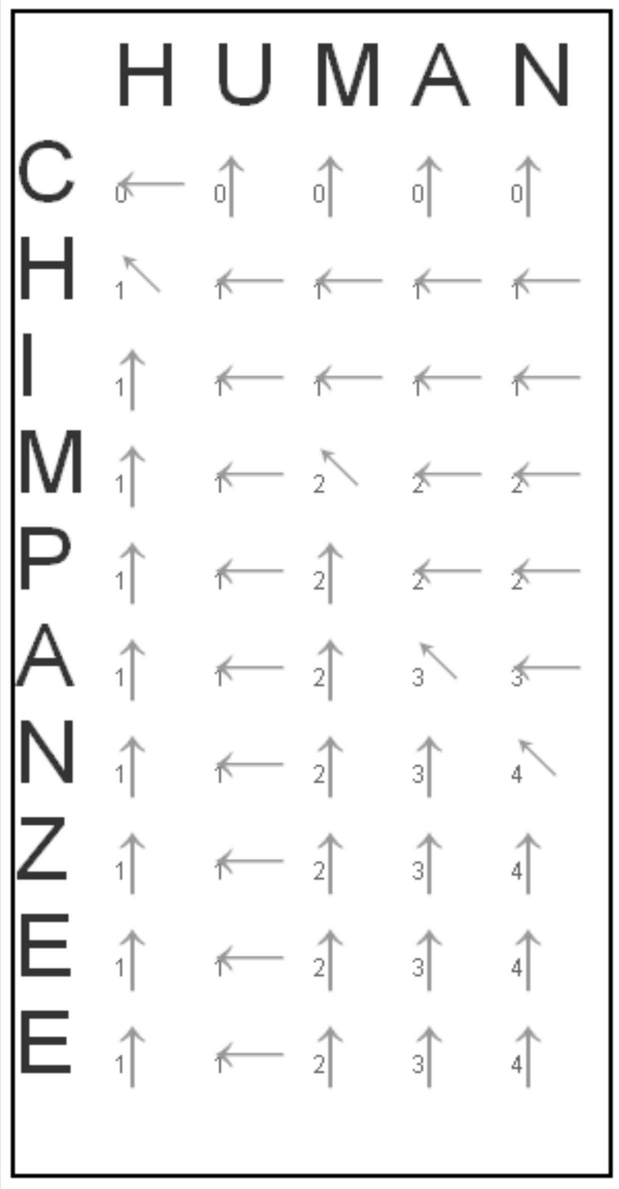
# Simulating LCSLength(X,Y)...

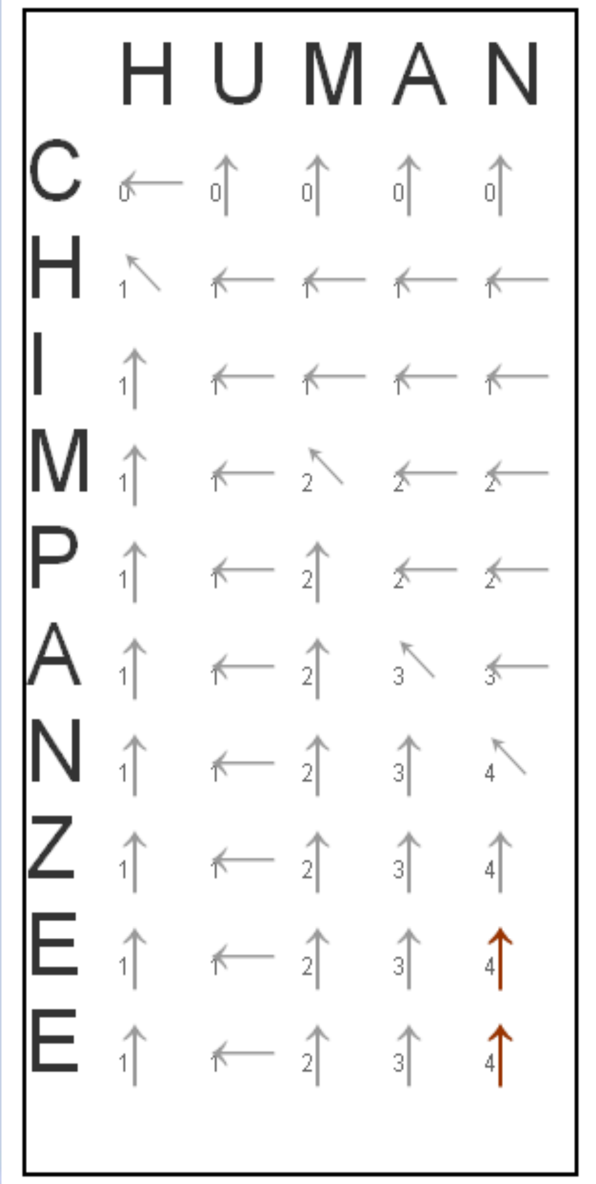



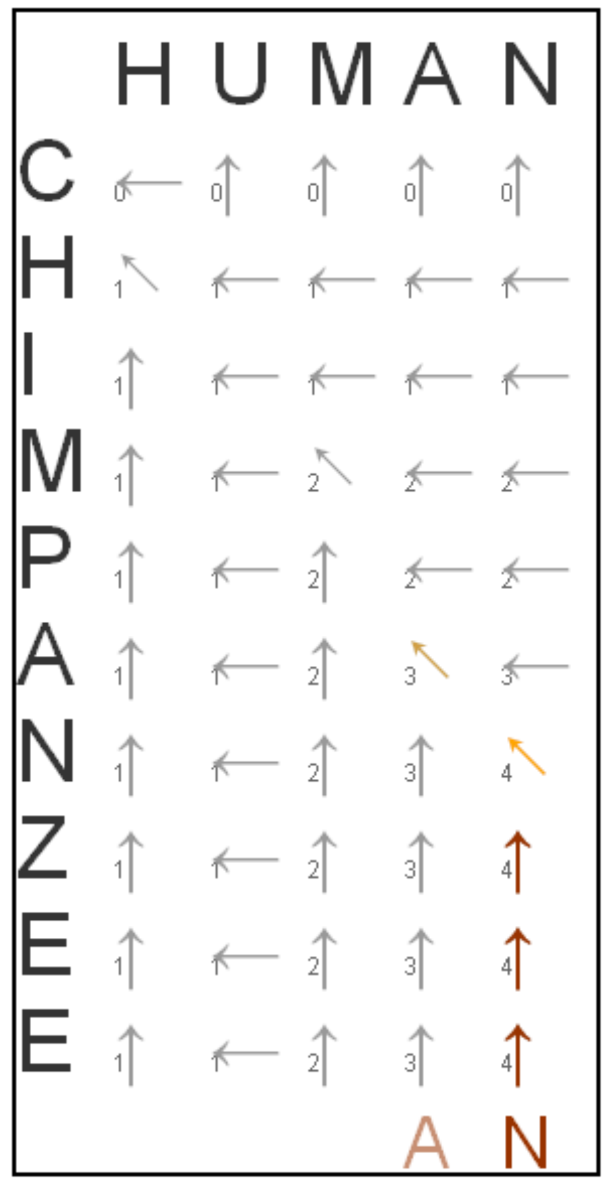
# Simulating LCSLength(X,Y)…

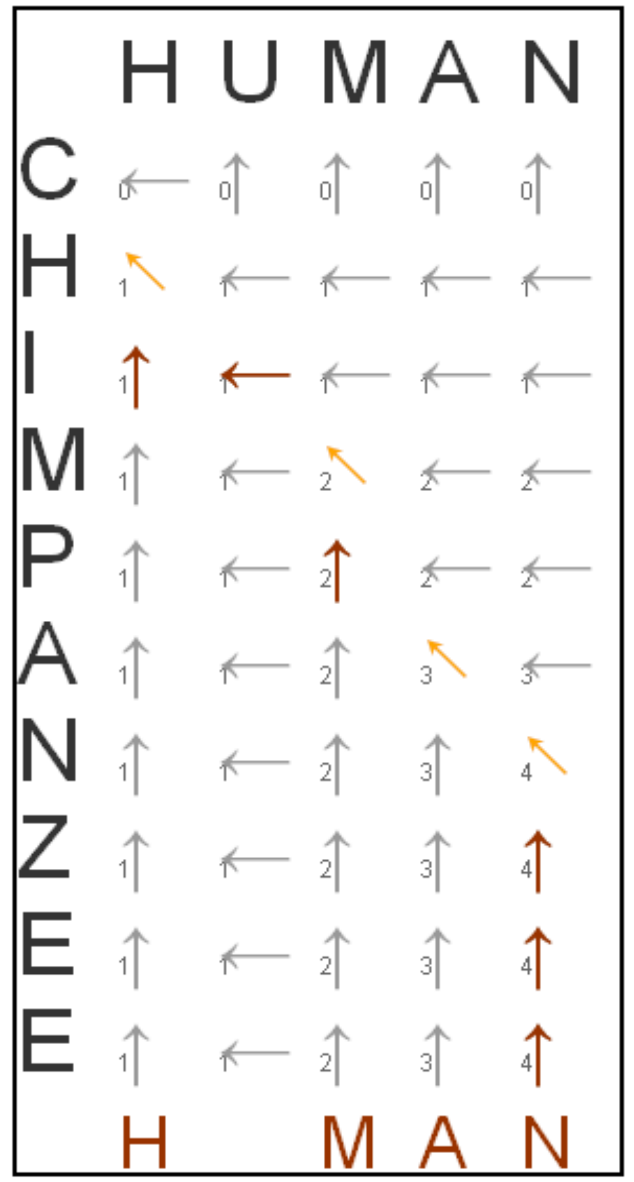


# Simulating LCSLength(X,Y)...



# Elements of Dynamic Programming

# : A Review

# Elements of Dynamic Programming

- Three key ingredients (some as in case of greedy)
  - Optimal substructure
  - Overlapping subproblems
  - memoization
- How do we map these three characteristics to any one of the problem that we have solved ?
  - ……….

# Optimal Substructure

- When does a problem exhibit optimal substructure ?
- When a problem exhibits optimal substructure…
  - either greedy approach or
  - dynamic programming approach

  may apply
- In dynamic programming
  - one builds the optimal solution to the main problem from optimal solutions to the subproblems
- Thus, the range of subproblems considered must be those included in the optimal solution.

# Dynamic Programming : Broad paradigm

- Three step process
  - Break the problem into smaller subproblems
  - solve these subproblems optimally
    - the subproblems are further solved following this three step process itself
  - use the discovered optimal solutions to the subproblems in framing the solution to the main problem.

# How to discover optimal substructure ?

- Follow the following steps to unearth a common pattern
  - designing the solution must involve making a choice
    - e.g. ……
  - this picked up choice must lead to one or more subproblems to be solved
  - identify which subproblem to solve and proceed along in the same manner
  - Prove that the solution works using contradiction

# Optimal Substructure…

- Optimal Substructure varies across problem domains in two ways
  - how many subproblems an optimal solution to the original problem uses
  - how many choices we have in determining which subproblems to use in an optimal solution.
- The running time, therefore, depends on
  - the total number of subproblems and
  - the number of choices to be investigated for each subproblem
- This can best be viewed from subproblem graph
  - as discussed in the Fibonacci problem….

# Top down OR Bottom up ?

- Dynamic programming uses optimal substructure usually in bottom-up fashion.
  - e.g………..
- However, at times we may also devise a top-down solution using dynamic programming
  - e.g…………

# Greedy vrs Dynamic

- Major difference
  - Greedy algorithms make choices quickly…one that is not an informed choice.
  - Dynamic algorithms make the choice of the subproblem to be solved only after evaluating the choices.

# Dynamic vrs. Divide and Conquer

- In dynamic programming,
  - typically the overlapping of the subproblems is exploited to an advantage whereas
  - in divide & conquer, the overlapping of the subproblems remains independent
- Whenever, the problem can be divided into typically equal sized subproblems,
  - use divide and conquer
  - use dynamic programming otherwise.

# All Pairs Shortest Paths

## Floyd Warshall's Algorithm

# All pairs shortest paths

- The problem of finding the shortest path between all pairs of vertices on a graph
  - is akin to making a table of all of the distances between all pairs of cities on a road map.
  - additionally, the route that gives rise to this shortest path also needs to be established.
  - one of many interesting problems that can be solved using graph algorithms.
  - there are a variety of solutions to this problem and
    - the algorithms that can be applied often depend on whether negative weighs are present in the graph.
  - one such algorithm is the Floyd-Warshall All-Pairs-Shortest-Path algorithm.

# All pairs shortest paths…

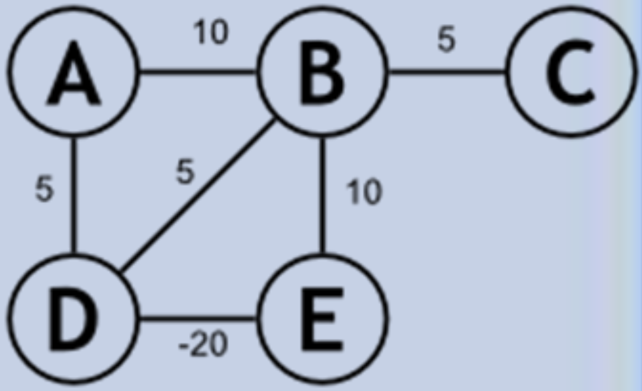
- A Variety of solutions to this problem exists
  - which algorithm to apply depend on the whether negative weighs are present in the graph.
- For example,
  - if no negative weights exists then it is possible to run Dijkstra's single-source shortest-path algorithm |V| times which has a run time of $O(|V|^3)$
  - when negative weights are present the Bellman-Ford algorithm could be run from each vertex.
  - another solution for graphs containing negative weights is known as the 'Slow-All-Pairs-Shortest-Path algorithm - has a run time of $O(|V|^4)$.

# Floyd-Warshall's Algorithm

- The Floyd-Warshall All-Pairs-Shortest-Path algorithm
  - uses a dynamic-programming methodology to solve the All-Pairs-Shortest-Path problem.
  - calculates the length of the shortest path between all nodes of a graph in $O(V^3)$ time.
  - negatively weighed edges may be present, however negatively weighted cycles cause problems with the algorithm.
  - it doesn't actually find the paths, it only finds their lengths.

# Negative weights and cycles : Difference

- e.g. consider a graph as shown




- There are several paths between A-E

  **Path 1**: A -> B -> E 20

  **Path 2**: A -> D -> E 25

  **Path 3**: A -> B -> D -> E 35

  **Path 4**: A -> D -> B -> E 20

- Look at the path *B -> E -> D -> B.*
  - The cost is *10 – 20 + 5 = -5.*
  - i.e. adding this loop to a path once lowers the cost of the path by *5*; and adding it twice would lower the cost by *2 * 5 = 10.*
  - Thus, negative cycle causes problems.

# Broad approach

- Step 1
  - Describe the structure of a shortest path
    - any subpath of a shortest path is a shortest path.
- Step 2
  - Give a recurrence for computing later values from earlier (bottom-up).
- Step 3
  - Give a high-level program (Computing the shortest-paths weights).
- Step 4
  - Constructing a shortest path.

# Step 1: Structure of a shortest path.

- Algorithm considers the "*intermediate*" vertices of a shortest path.
- Intermediate Vertices ?
- Given a directed graph
  - G = (V, E), |V| = n , V={1,2,….,n}
  - The vertices $v_2, v_3, v_4, \ldots. v_{K-1}$ are known as intermediate vertices of the path p = < $v_1, v_2, v_3, v_4, \ldots. v_k$>
- the algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1,2,….,k-1}.

# Step 1: Structure of a shortest path.

- Let $d_{ij}^{(k)}$ be the length of the shortest path from $i$ to $j$ such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.

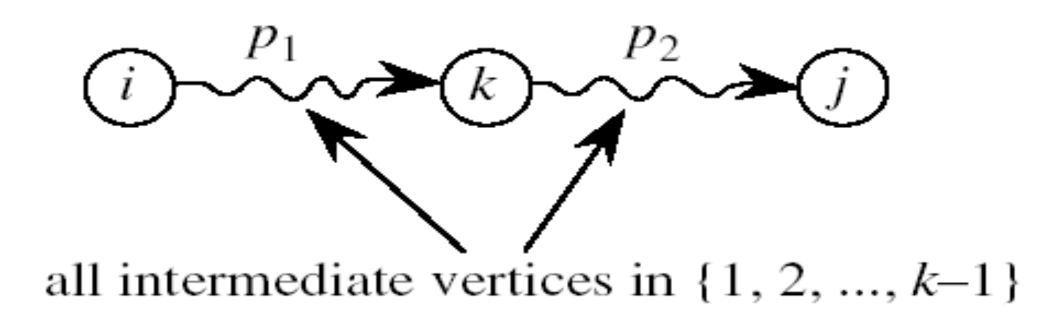
  $d_{ij}^{(0)}$ is set to be $w_{ij}$, i.e., no intermediate vertex. Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.

- Claim: $d_{ij}^{(n)}$ is the distance from $i$ to $j$. So our aim is to compute $D^{(n)}$.

- Subproblems: compute $D^{(k)}$ for $k = 0, 1, \cdots, n$.

# Step 1: Structure of a shortest path….

- Base condition: $d_{ij}^{(0)}$ = ?
  - $d_{ij}^{(0)} = w_{ij}$.
- For k>0:
  - Let p=<$v_i$, . . . , $v_j$> be a shortest path from vertex i to vertex j with all intermediate vertices in {1,2, …, k}.
  - If k is *not* an intermediate vertex of path p, then all intermediate vertices of path p are in {1,2, …, k-1}.
  - If k *is* an intermediate vertex, then p is composed of 2 shortest subpaths drawn from {1,2, …, k-1}.



# Step 2: Recursive Formulation for $d_{ij}^{(k)}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

- $d_{ij}^{(k)}$ = weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1,2,….,k}.

# Step 3: The Algorithm

FLOYD-WARSHALL$(W, n)$

$D^{(0)} \leftarrow W$

**for** $k \leftarrow 1$ **to** $n$

  **do for** $i \leftarrow 1$ **to** $n$

    **do for** $j \leftarrow 1$ **to** $n$

      **do** $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

**return** $D^{(n)}$

- W=(wij), where $w_{ij}$ = 0, if i = j

  = the weight of directed edge (i, j), if i != j and (i,j) ϵ E

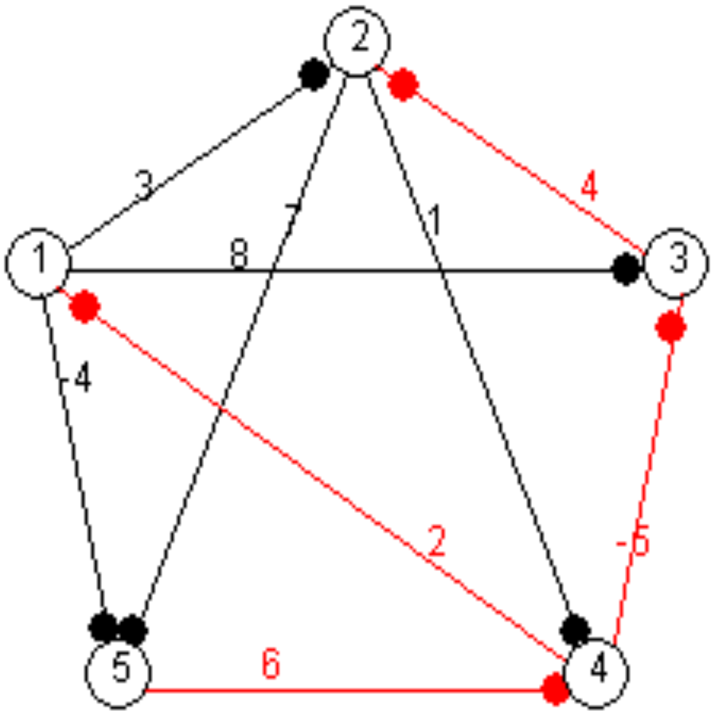
  = infinite, if i != j and (i, j) !ϵ E.

# Complexity of Floyd-Warshall algorithm.

- The running time of the algorithm is determined by the triply nested for loops.
- Because each execution of line-5 takes O(1) time, the algorithms runs in time $\Theta(n^3)$.
- Here code is tight, with no elaborate data structures, and so the constant hidden in the $\Theta$-notation is small.
- Thus this algorithm is quit practical for even moderate-sized input graphs.

# Running Floyd-Warshall algorithm

View Shortest path between: 1 to 2 Dist: 3 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 999 | -5 | 0 | 999 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

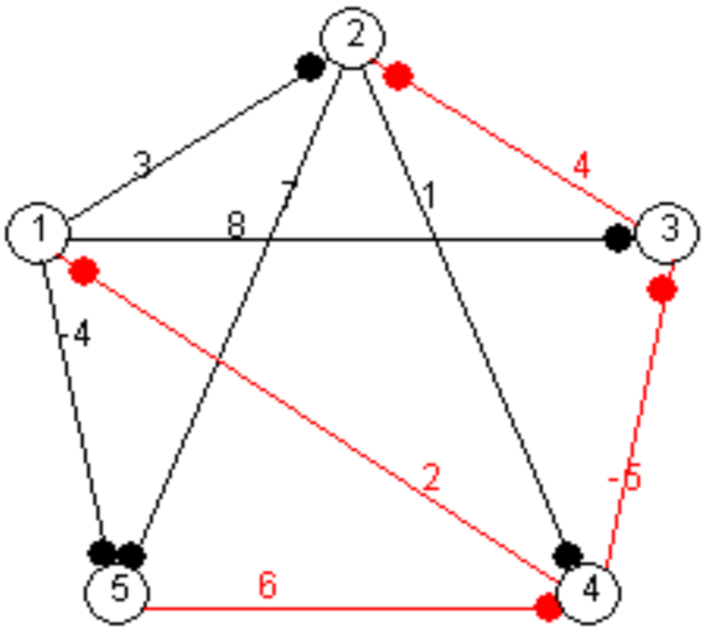
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 2 Dist: 3 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 5 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

☑ Single Step

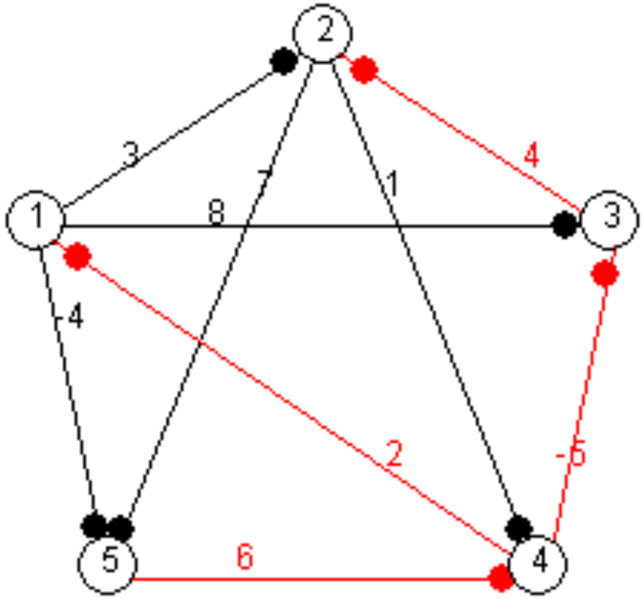
**Continue?**

Cycle complete using nodes 1 - 1 OK

**The Floyd**

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 2 Dist: 3 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 4 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 5 | 11 |
| 2 | 5 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

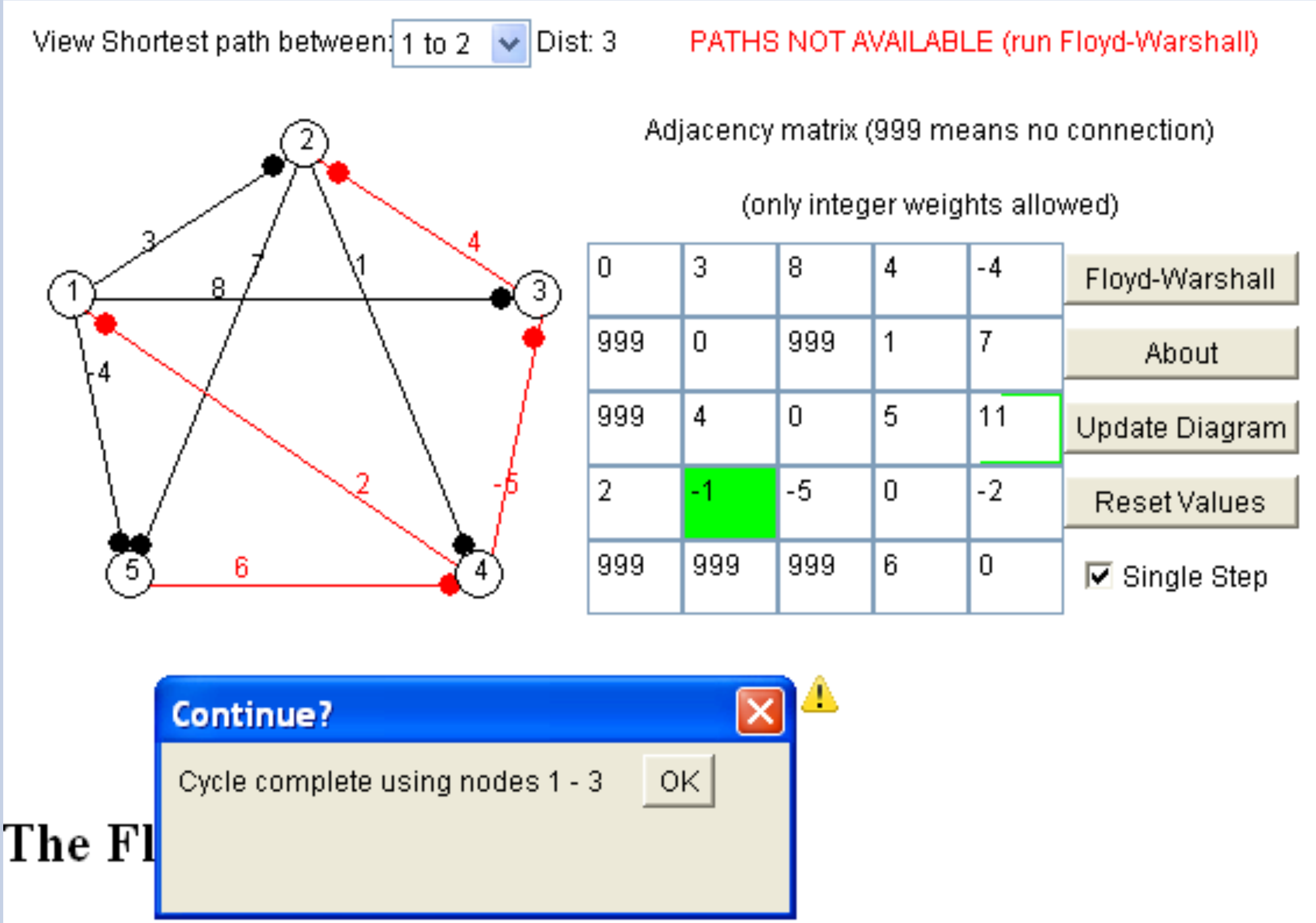
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 2 OK

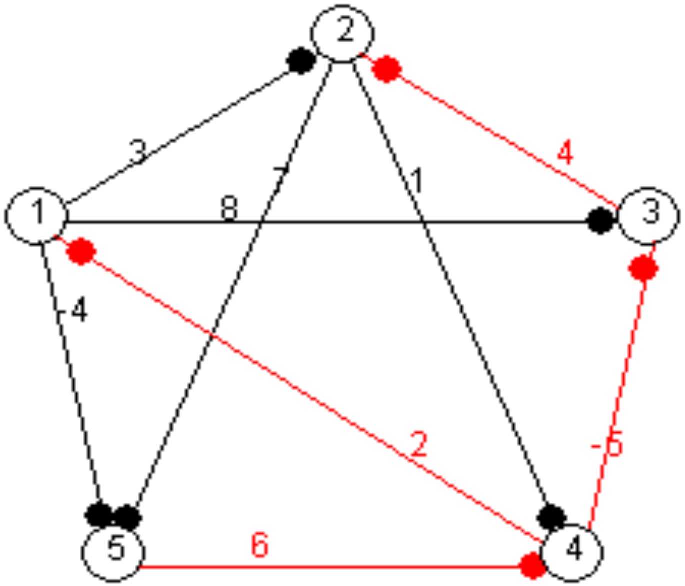
**The Floyd-**

# Running Floyd-Warshall algorithm…



# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 2 Dist: 3 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | -1 | 4 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

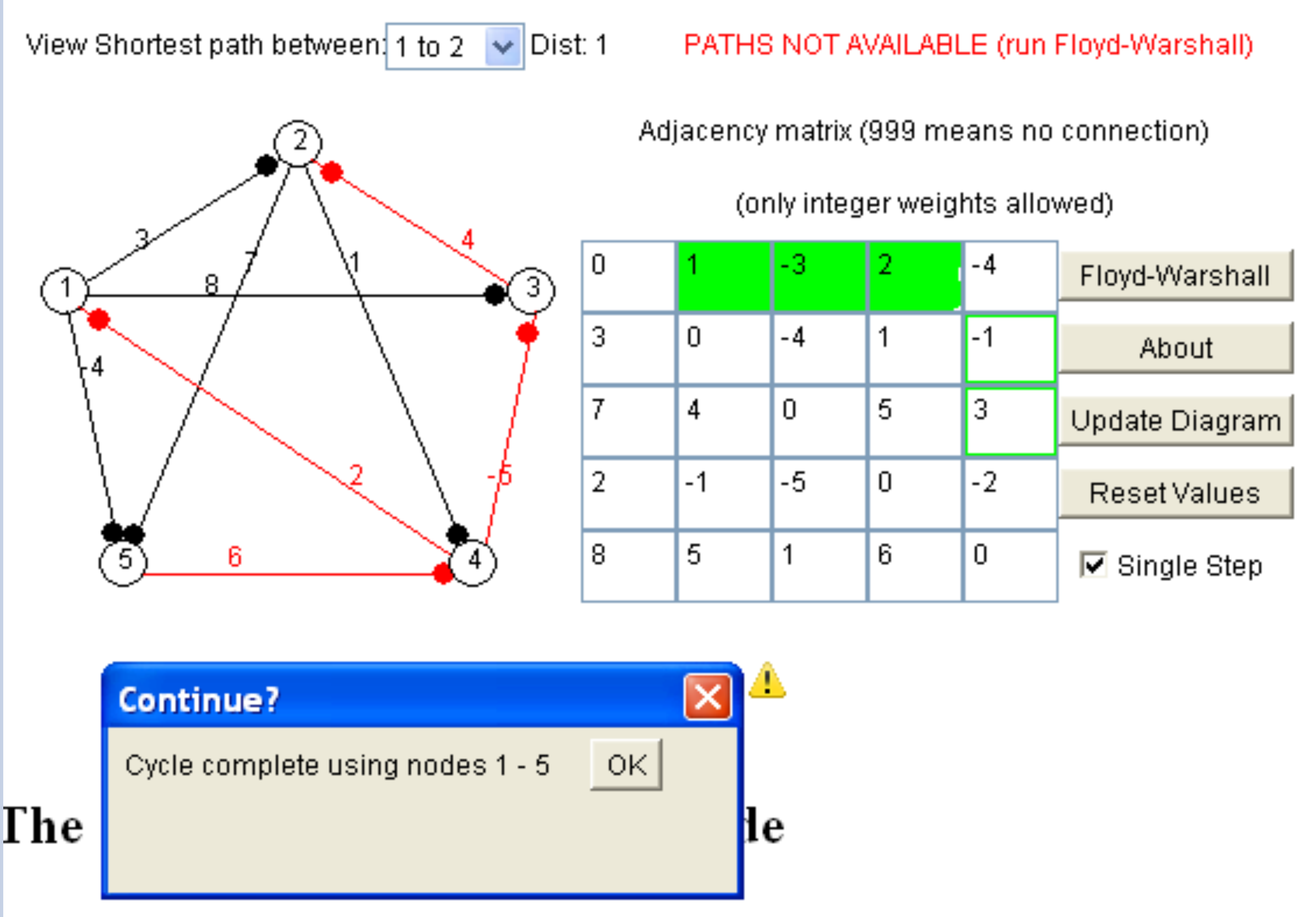
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 4 OK

**The Flo**

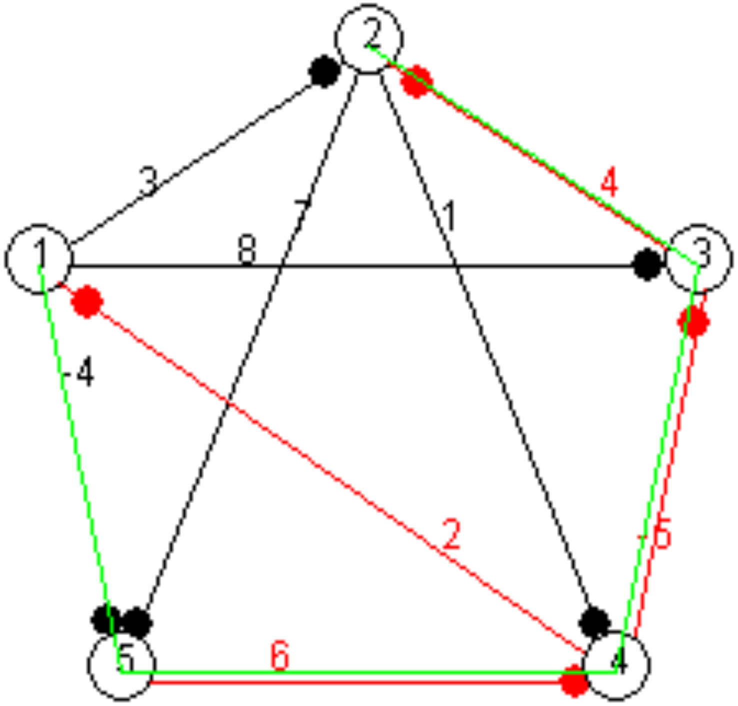
# Running Floyd-Warshall algorithm…




| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 2 Dist: 1 PATHS AVAILABLE




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

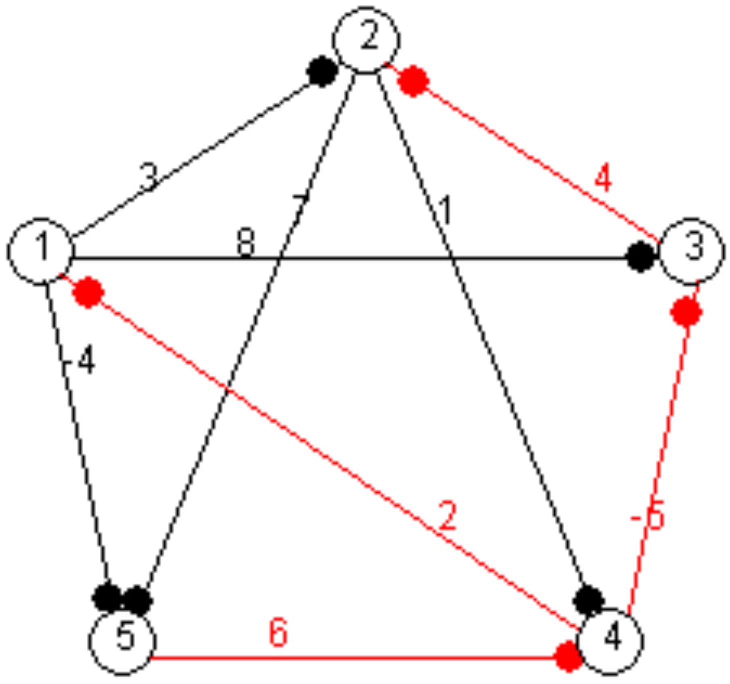
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 2 to 1 Dist: 999 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 999 | -5 | 0 | 999 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

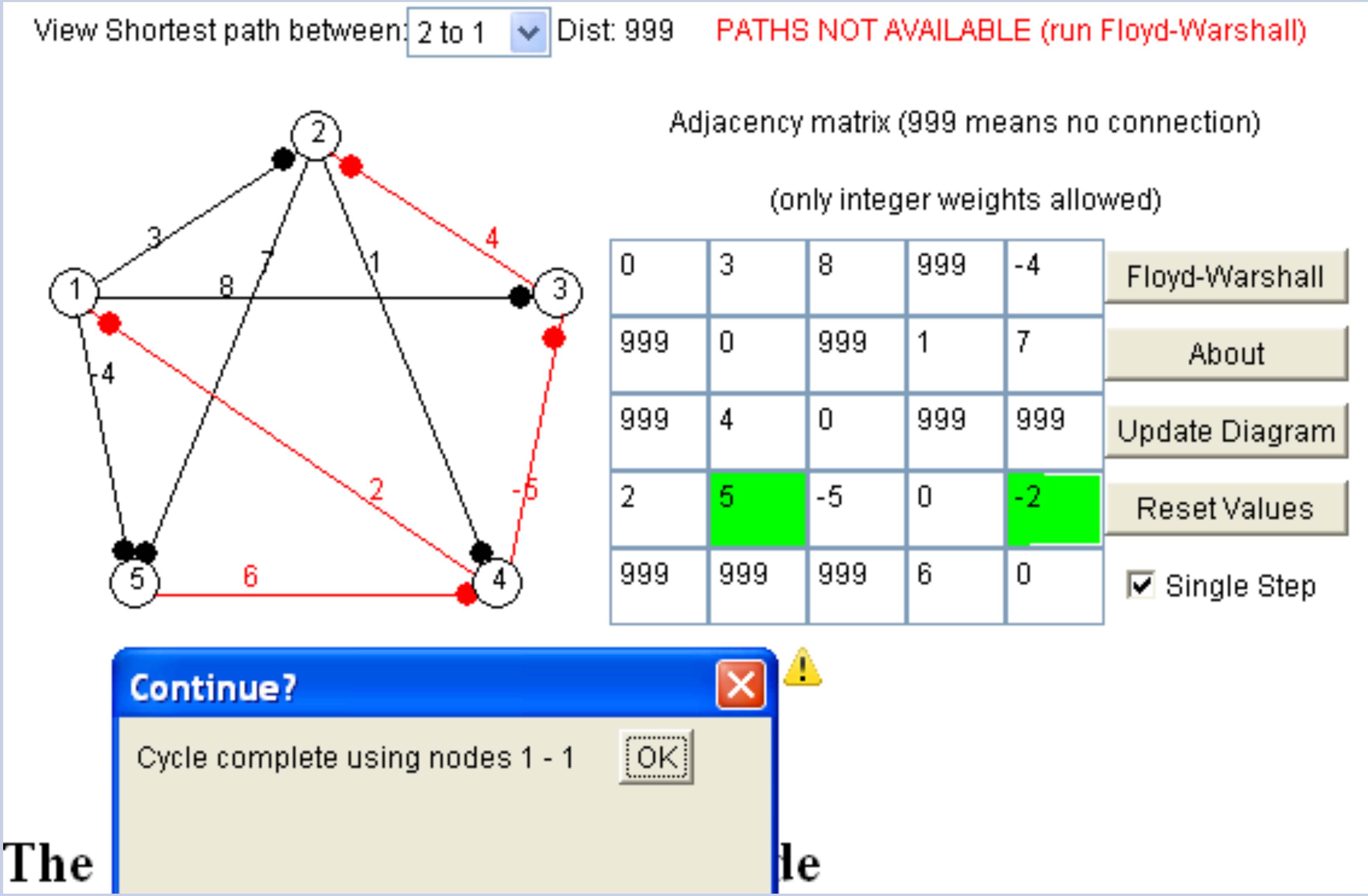
About

Update Diagram

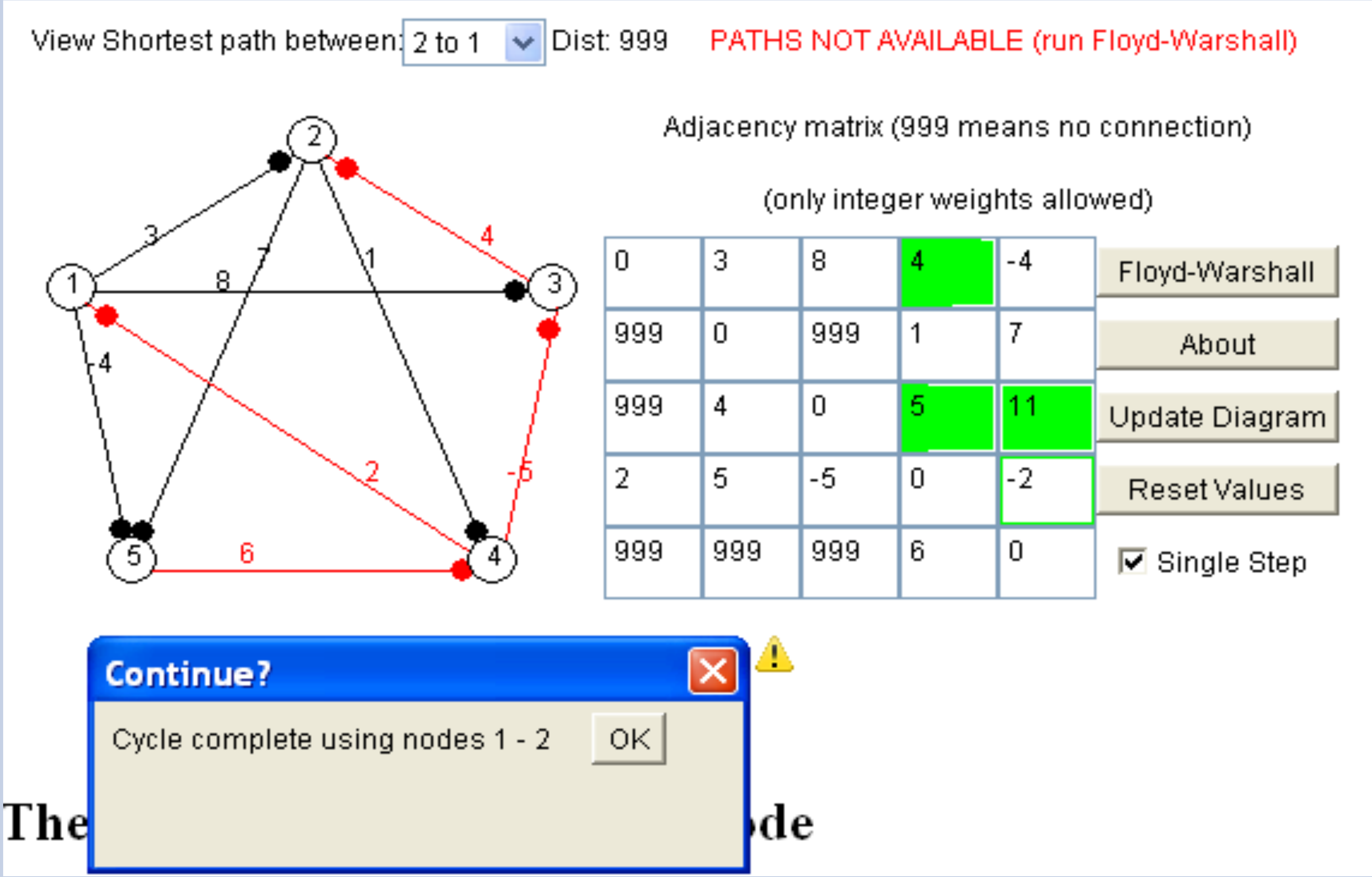
Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

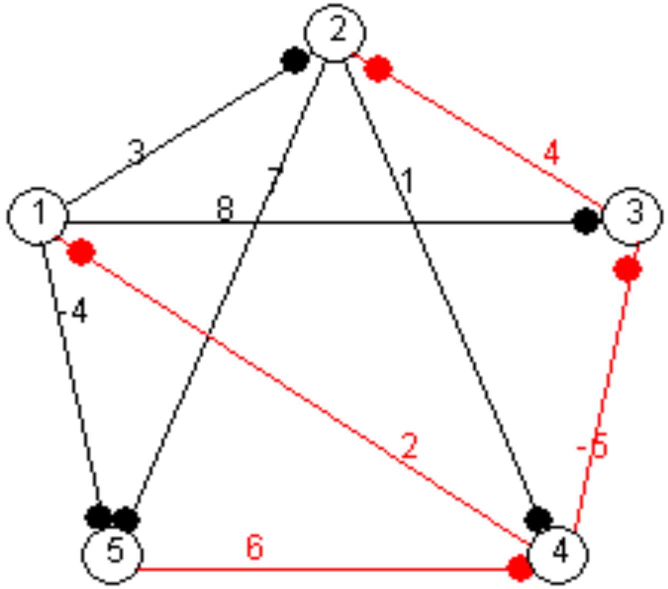


# Running Floyd-Warshall algorithm…



# Running Floyd-Warshall algorithm…

View Shortest path between: 2 to 1 Dist: 999 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 4 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 5 | 11 |
| 2 | -1 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

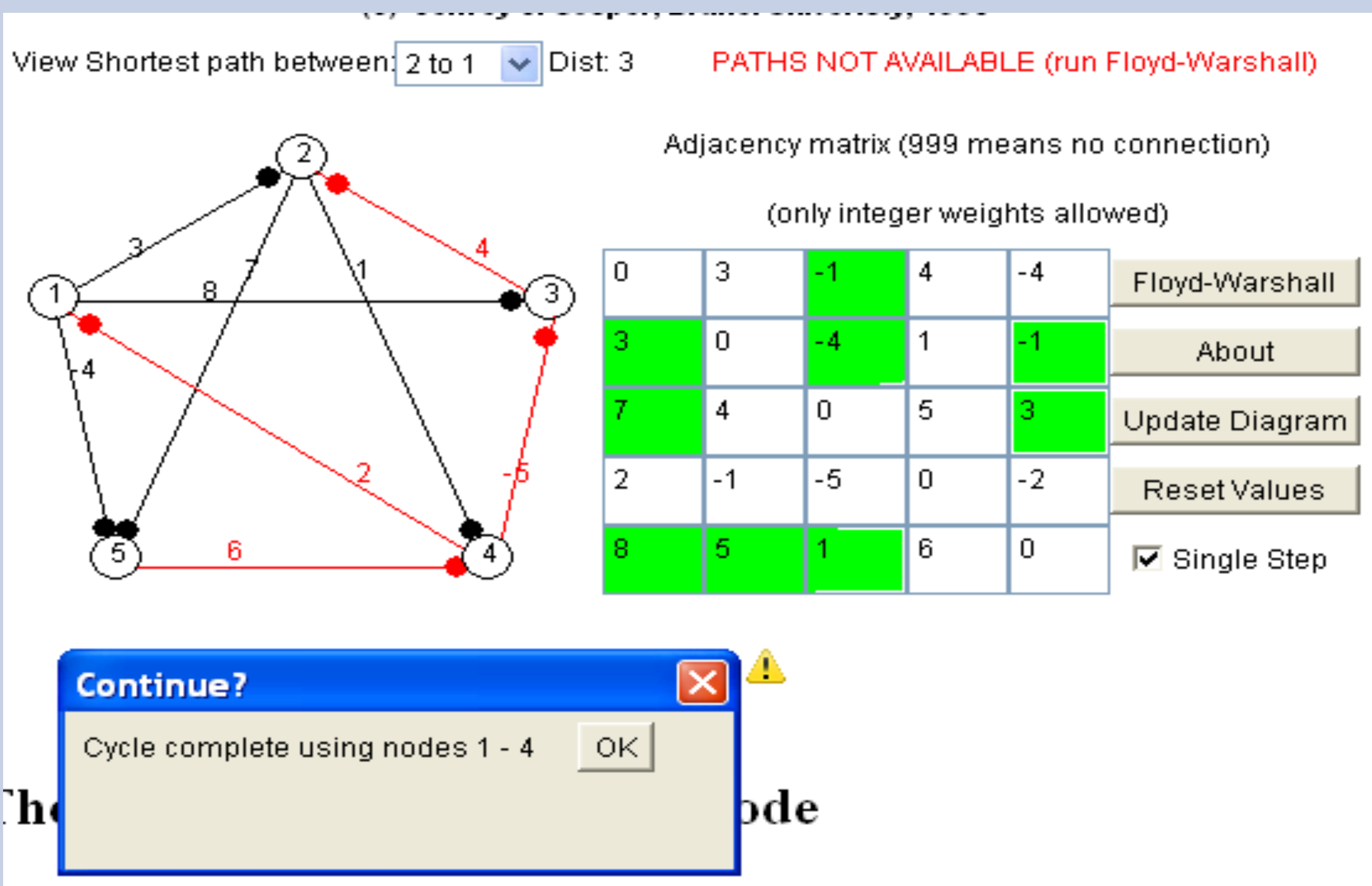
Single Step

Continue?

Cycle complete using nodes 1 - 3 OK

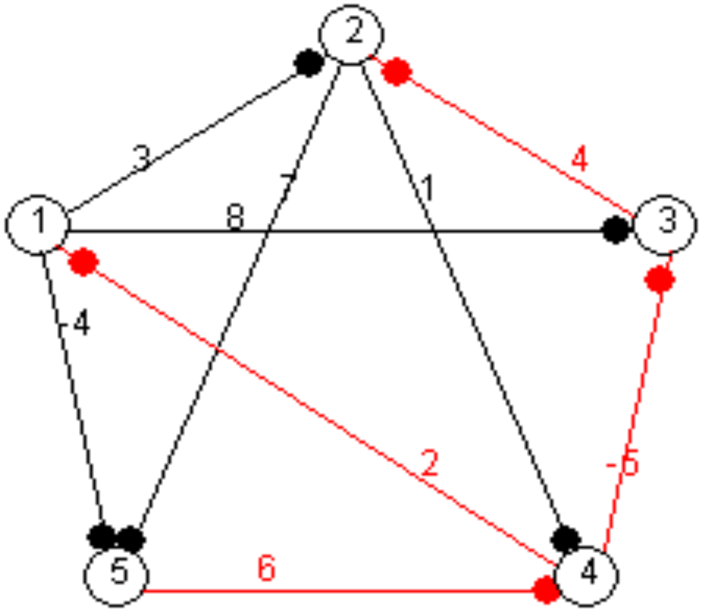
Th ode

# Running Floyd-Warshall algorithm…



# Running Floyd-Warshall algorithm…

View Shortest path between: 2 to 1 Dist: 3 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

☑ Single Step

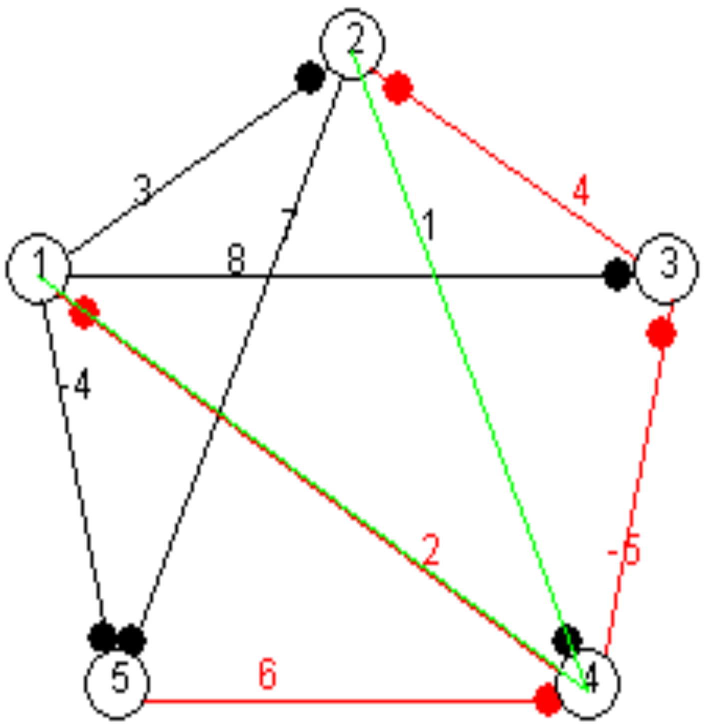
**Continue?**

Cycle complete using nodes 1 - 5 OK

**Th** **ode**

# Running Floyd-Warshall algorithm…

View Shortest path between: 2 to 1 Dist: 3 PATHS AVAILABLE




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

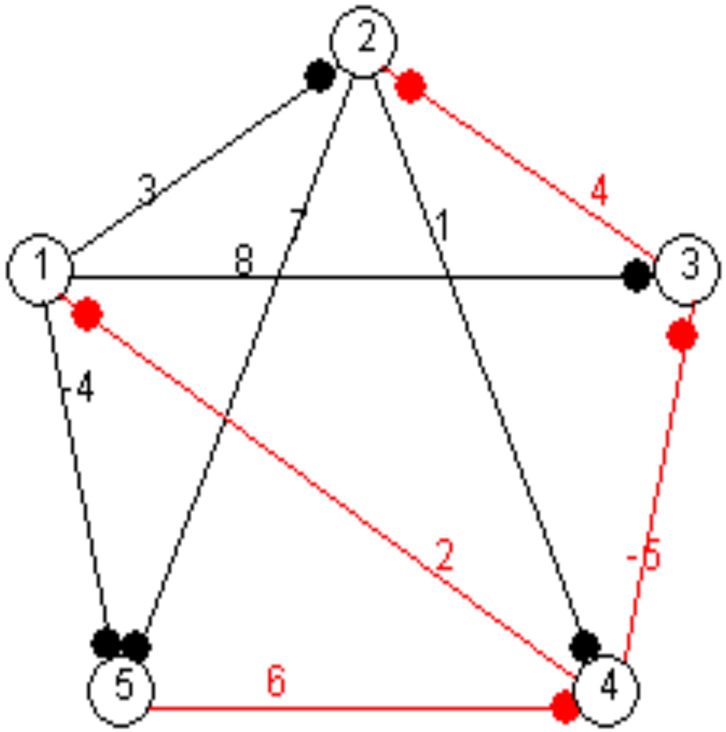
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5 Dist: -4 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 999 | -5 | 0 | 999 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

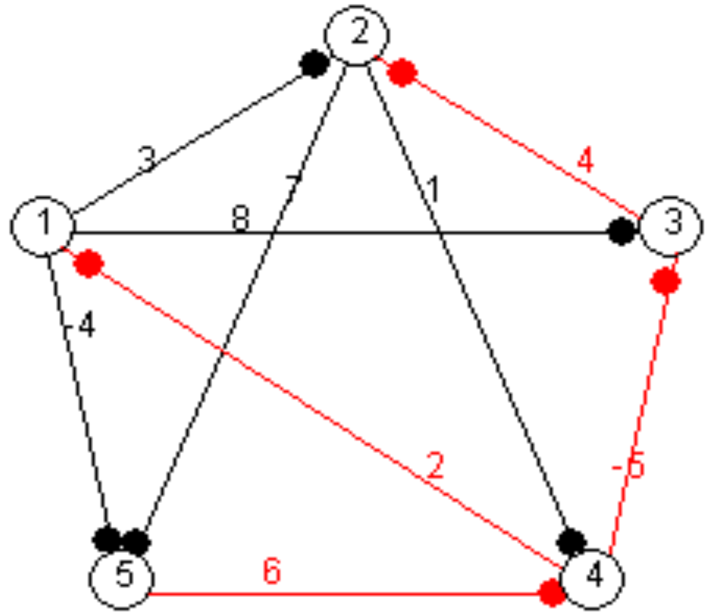
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5 Dist: -4 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 5 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

☑ Single Step

**Continue?**

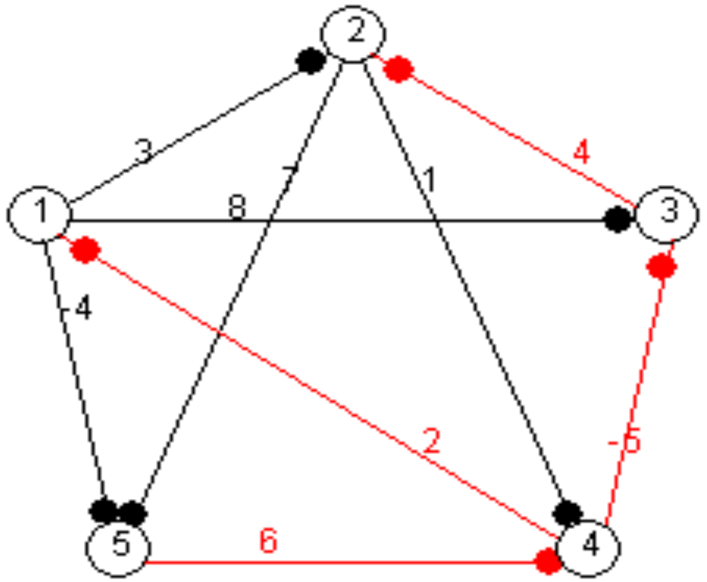
Cycle complete using nodes 1 - 1 OK

The

le

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5   Dist: -4   PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 4 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 5 | 11 |
| 2 | 5 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

☑ Single Step

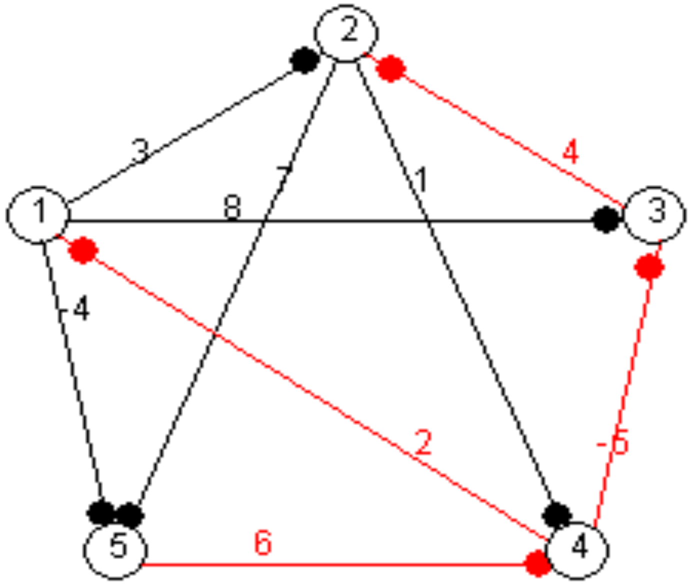
**Continue?**

Cycle complete using nodes 1 - 2   OK

Th … Code

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5 Dist: -4 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 4 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 5 | 11 |
| 2 | -1 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

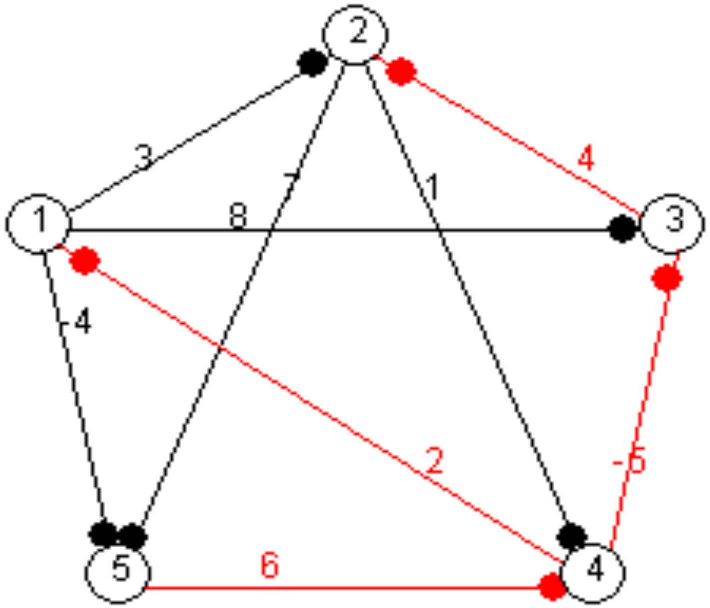
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 3 OK

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5 Dist: -4 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | -1 | 4 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

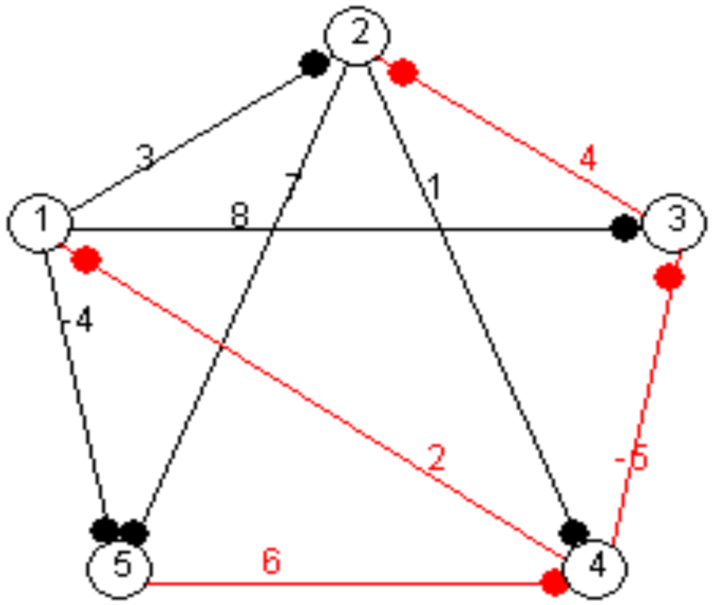
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 4 OK

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5 Dist: -4 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

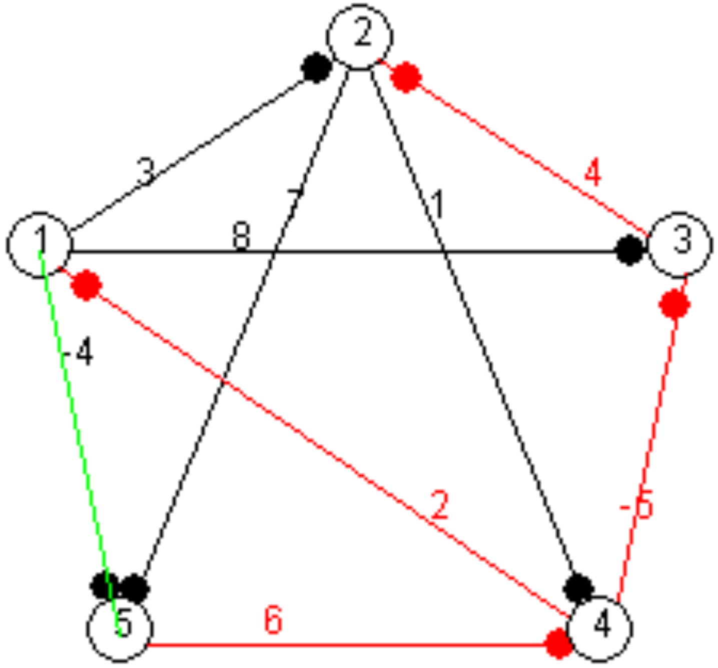
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 5 OK

# Running Floyd-Warshall algorithm…

View Shortest path between: 1 to 5 Dist: -4 PATHS AVAILABLE




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

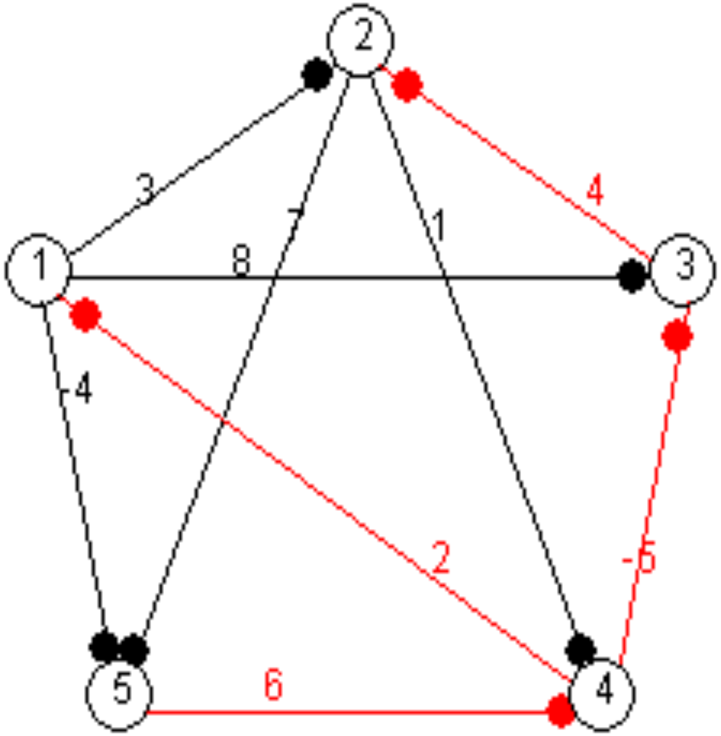
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 3 to 5 Dist: 999 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 999 | -5 | 0 | 999 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

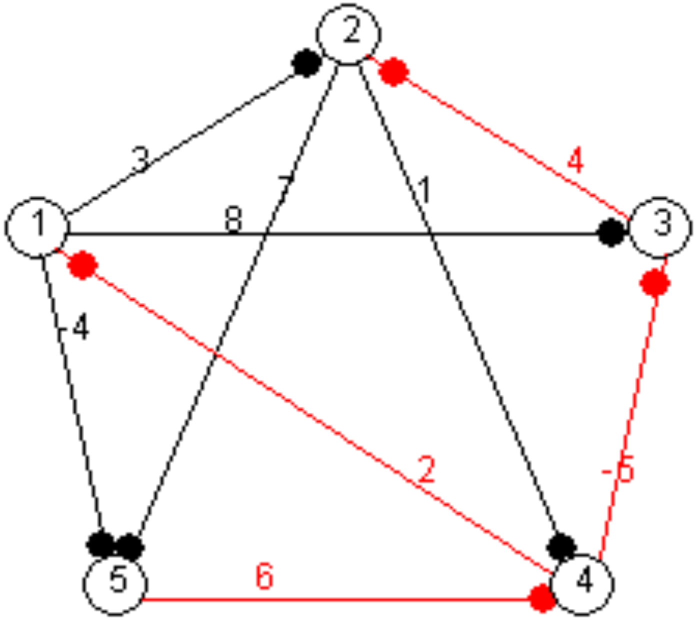
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 3 to 5 Dist: 999 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 5 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

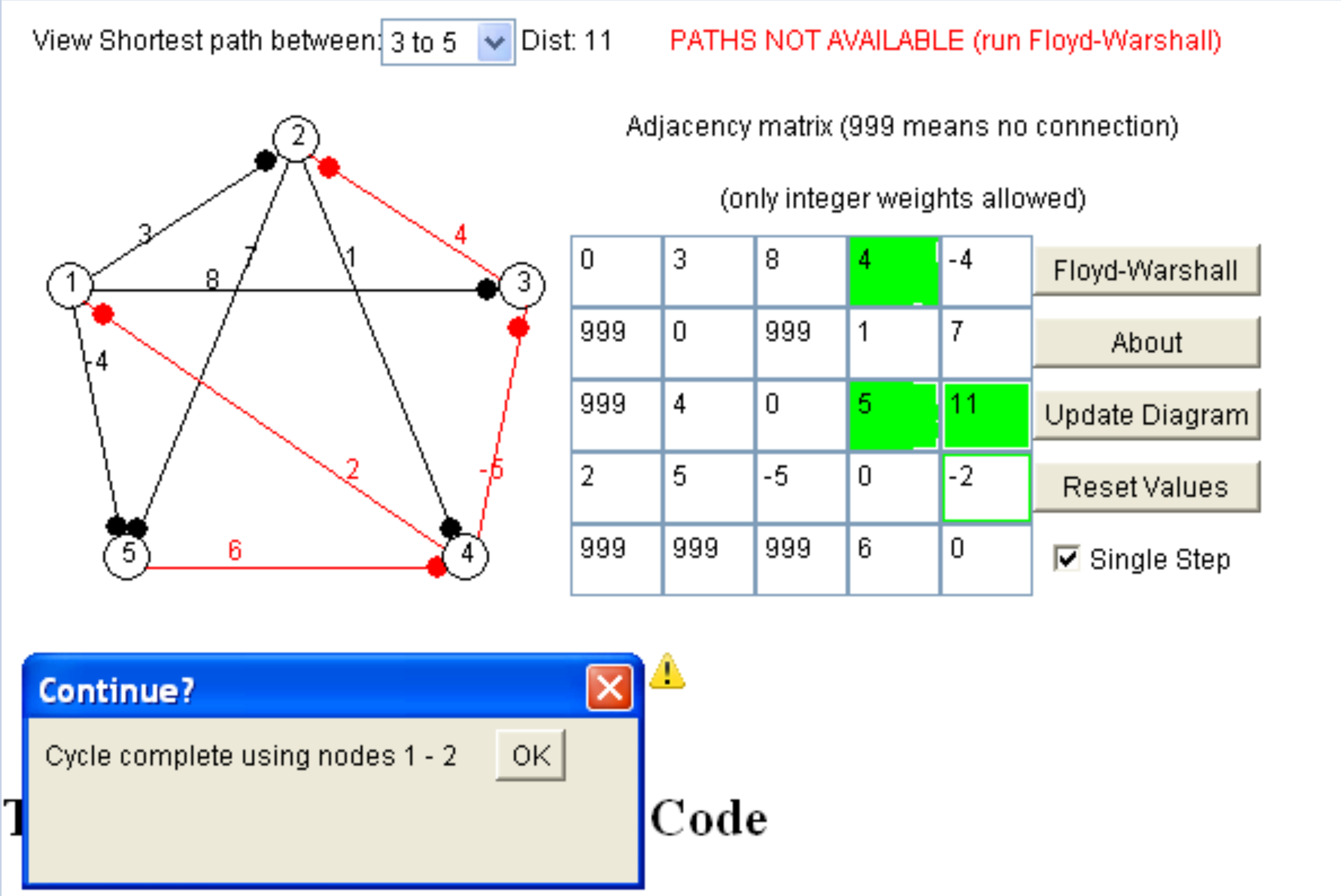
☑ Single Step

**Continue?**

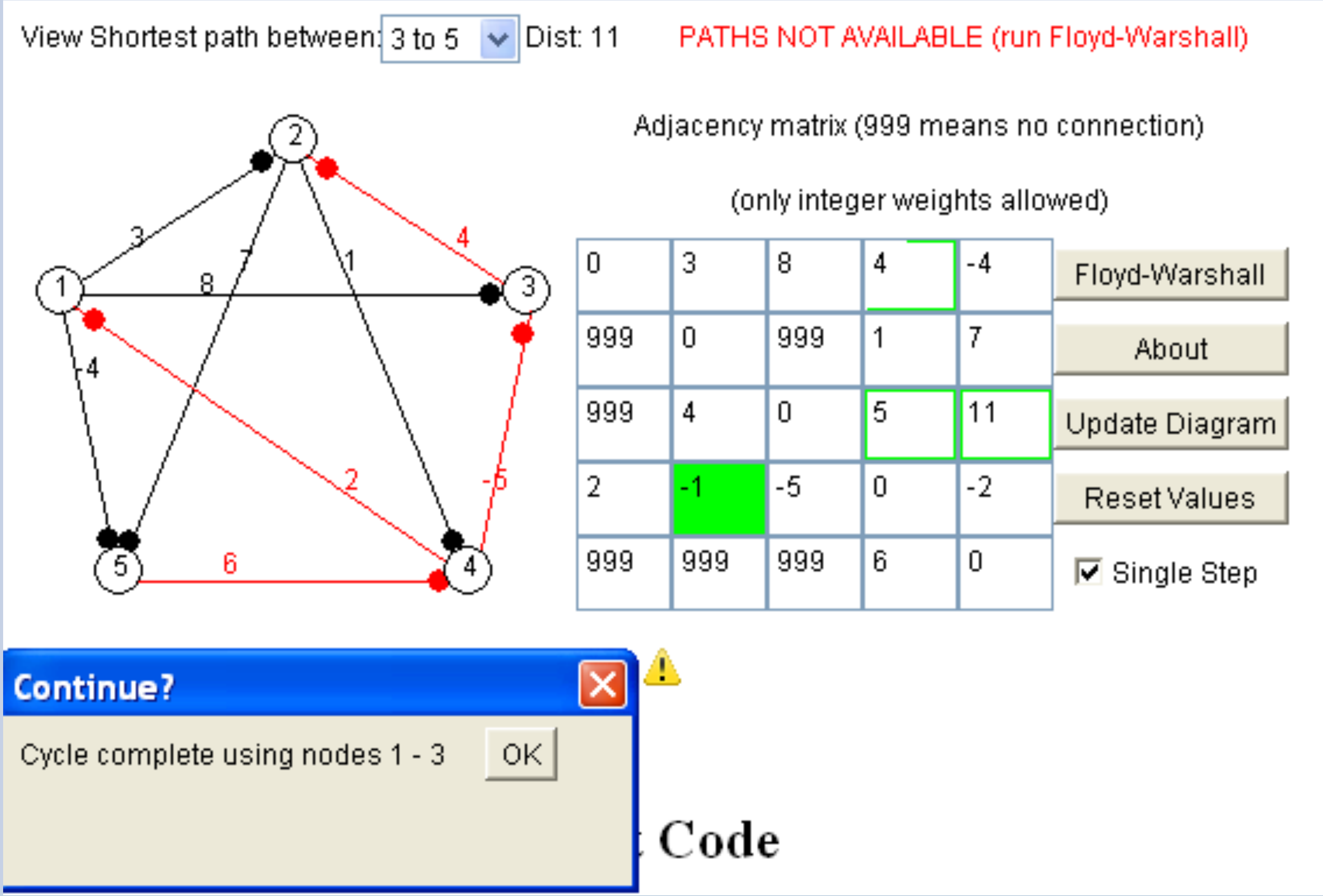
Cycle complete using nodes 1 - 1 OK

Code

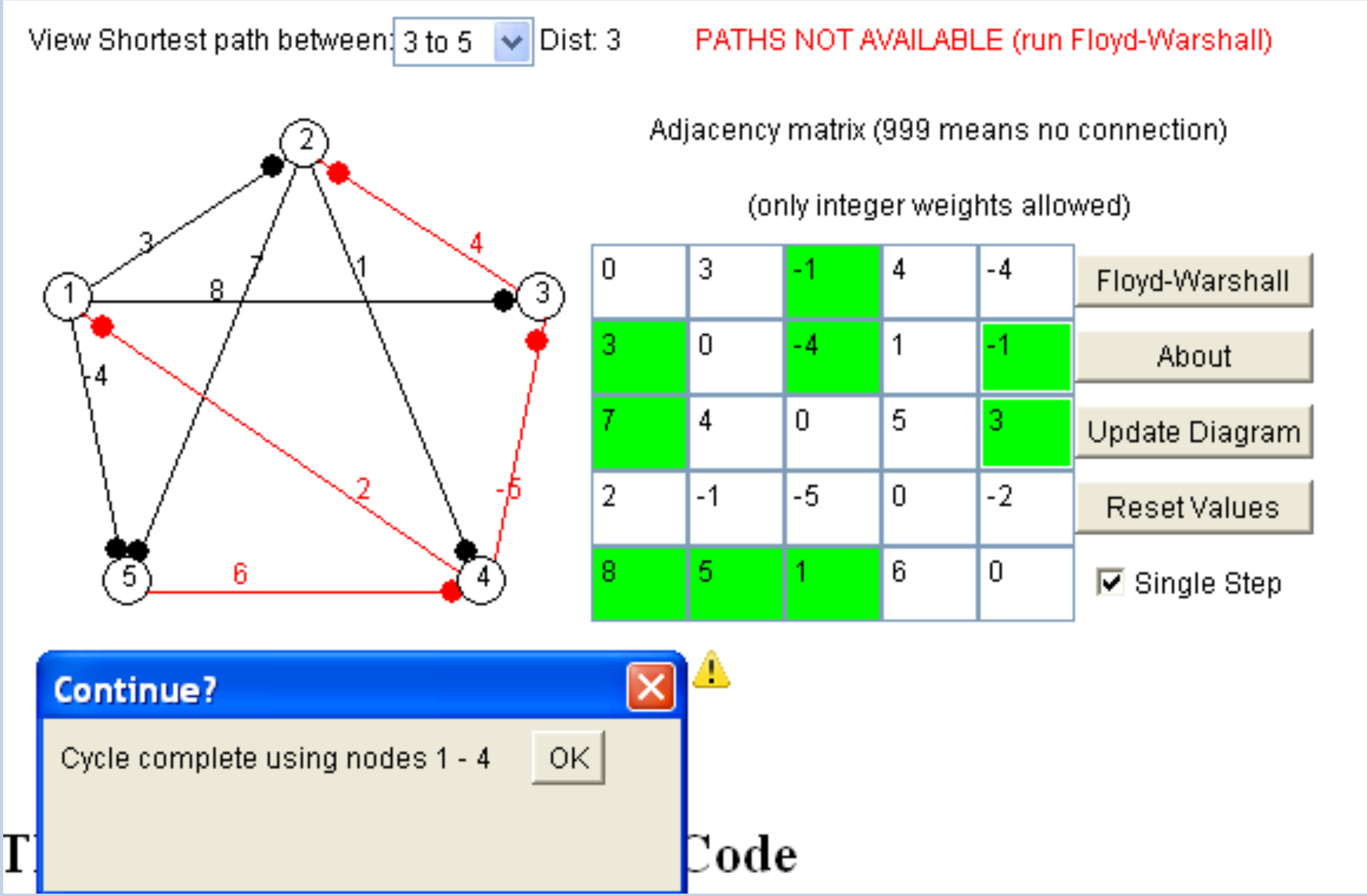
# Running Floyd-Warshall algorithm…



# Running Floyd-Warshall algorithm…



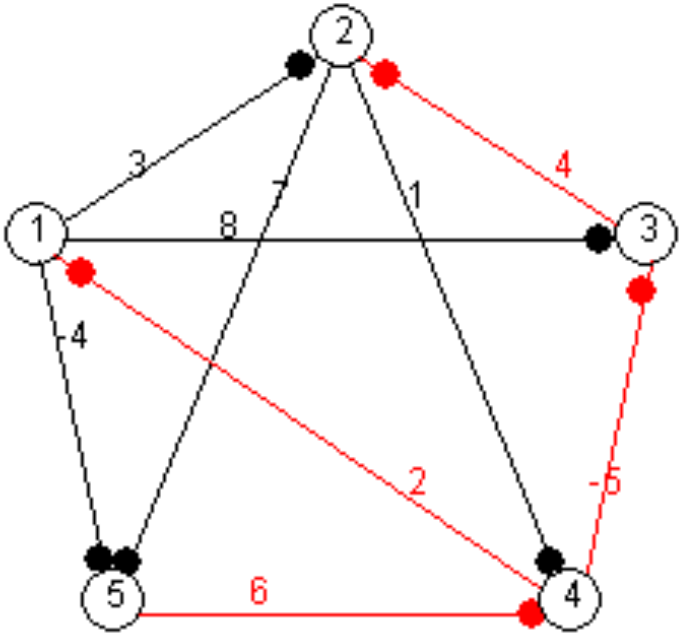
# Running Floyd-Warshall algorithm…




| | | | | |
|---|---|---|---|---|
| 0 | 3 | -1 | 4 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

# Running Floyd-Warshall algorithm…

View Shortest path between: 3 to 5 Dist: 3 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

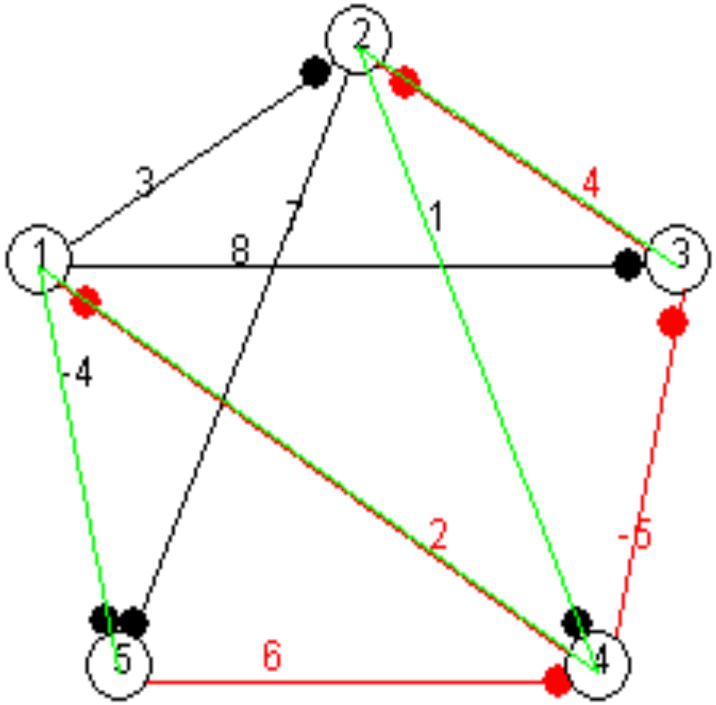
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 5 OK

# Running Floyd-Warshall algorithm…

View Shortest path between: 3 to 5 Dist: 3 PATHS AVAILABLE




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

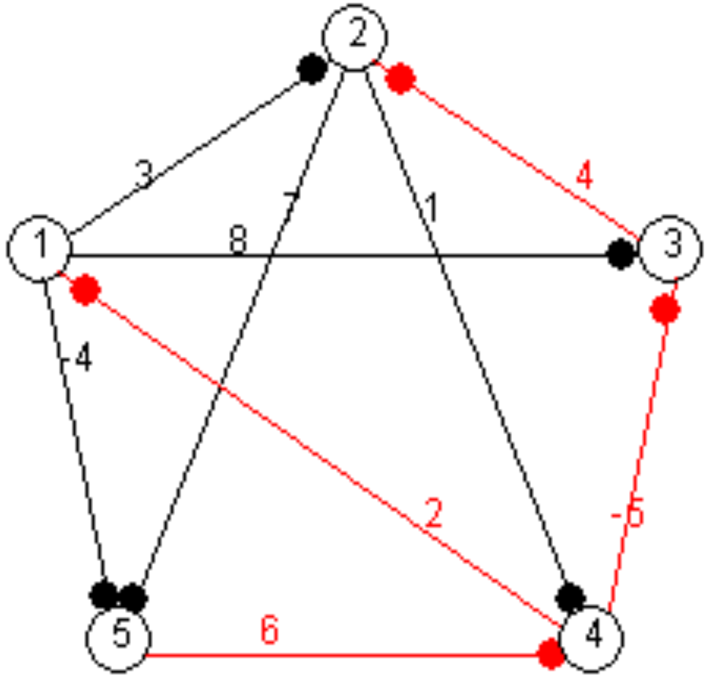
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 5 to 4 Dist: 6 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 999 | -5 | 0 | 999 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

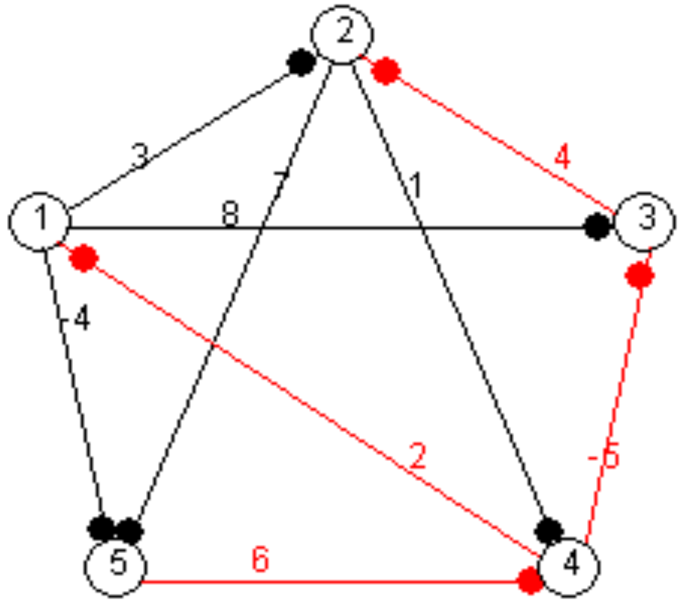
Update Diagram

Reset Values

☑ Single Step

# Running Floyd-Warshall algorithm…

View Shortest path between: 5 to 4 Dist: 6 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 999 | -4 |
| 999 | 0 | 999 | 1 | 7 |
| 999 | 4 | 0 | 999 | 999 |
| 2 | 5 | -5 | 0 | -2 |
| 999 | 999 | 999 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

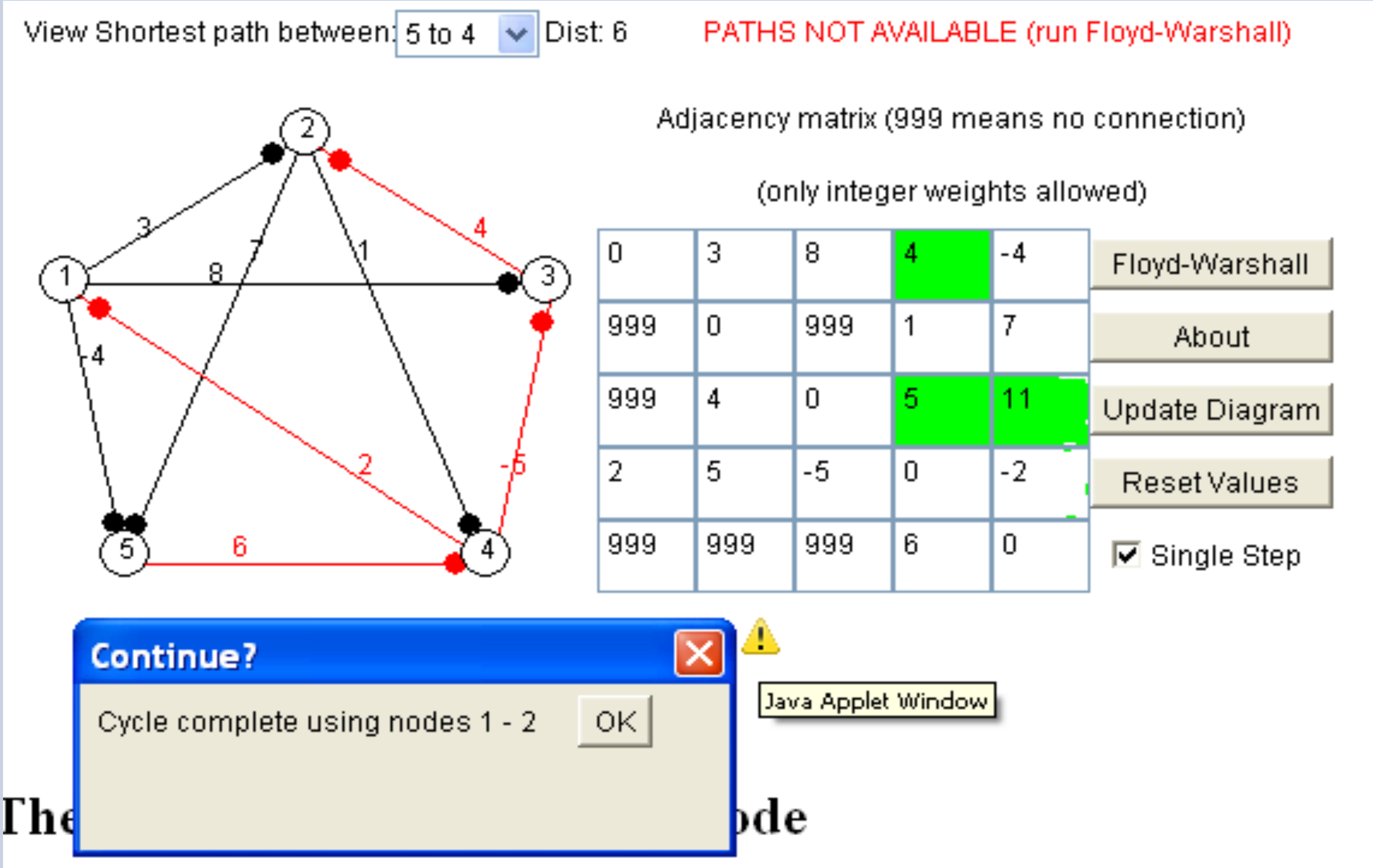
☑ Single Step

**Continue?**

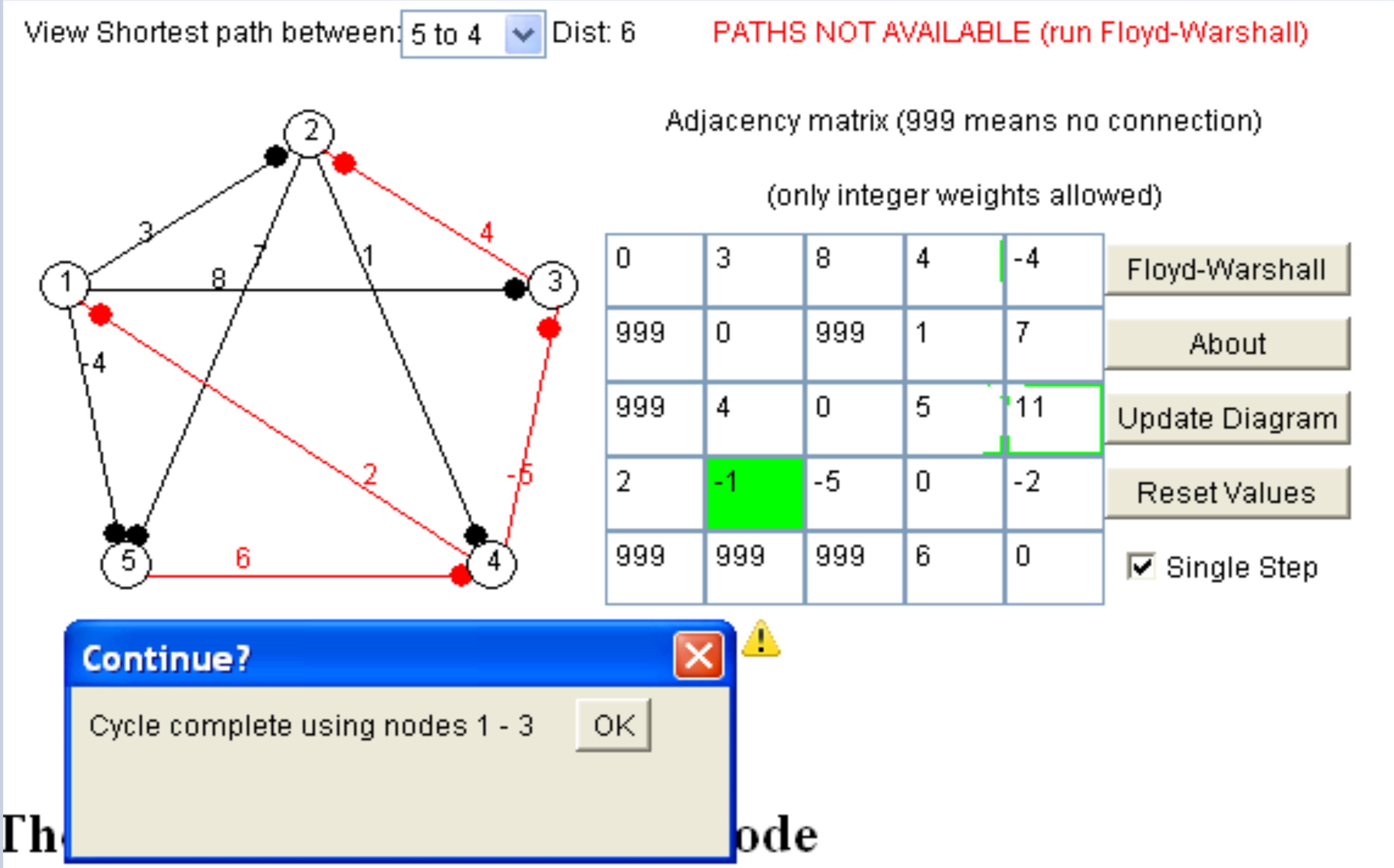
Cycle complete using nodes 1 - 1 OK

Code

# Running Floyd-Warshall algorithm…



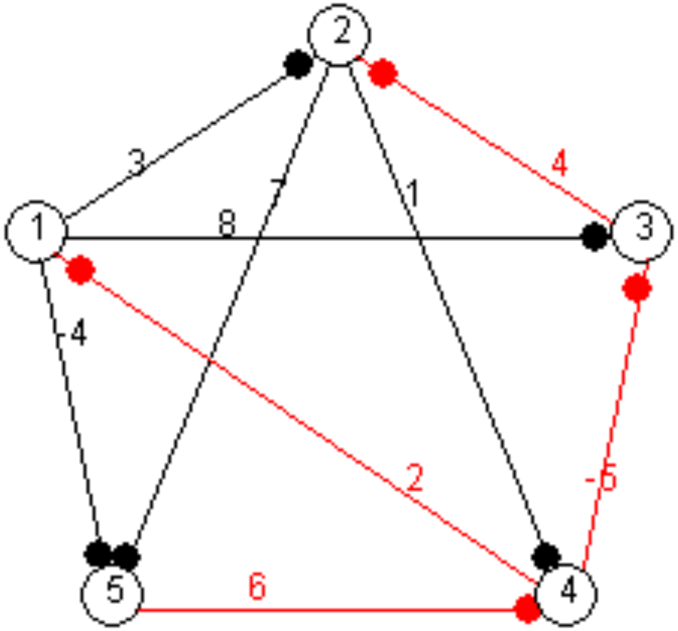
# Running Floyd-Warshall algorithm…




© Jeffery D Gosper

# Running Floyd-Warshall algorithm…

View Shortest path between: 5 to 4 Dist: 6 PATHS NOT AVAILABLE (run Floyd-Warshall)




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 3 | -1 | 4 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

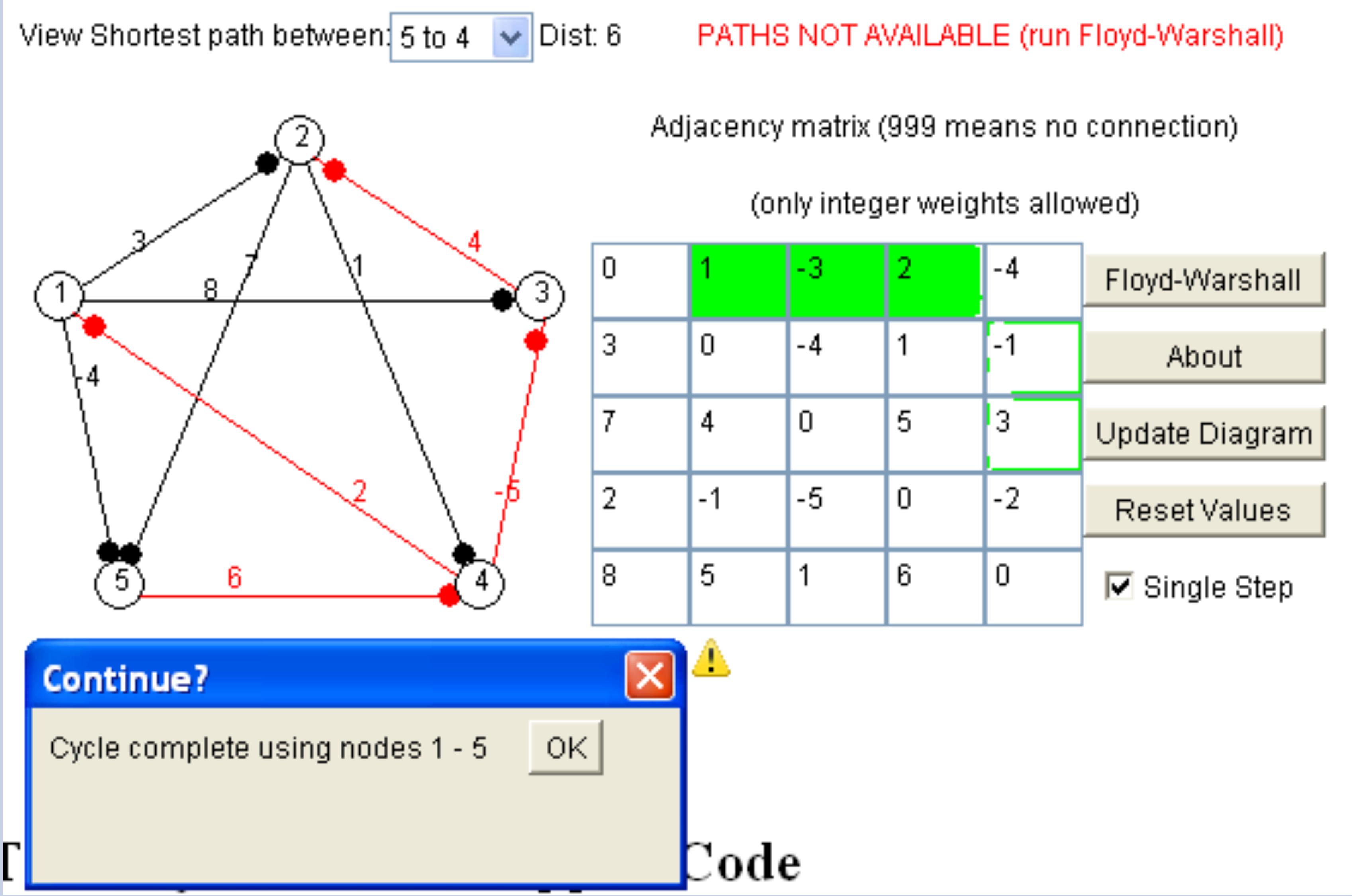
☑ Single Step

**Continue?**

Cycle complete using nodes 1 - 4 OK

[ Code

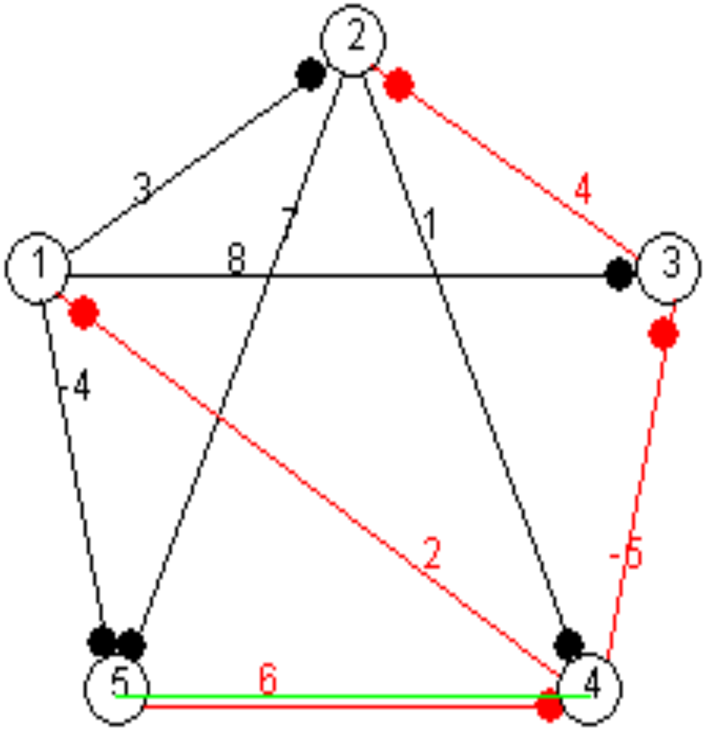
# Running Floyd-Warshall algorithm…




© Jeffery D Gosper

# Running Floyd-Warshall algorithm…

View Shortest path between: 5 to 4 Dist: 6 PATHS AVAILABLE




Adjacency matrix (999 means no connection)

(only integer weights allowed)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | -3 | 2 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

Floyd-Warshall

About

Update Diagram

Reset Values

☑ Single Step

# Summarizing Dynamic Programming

- The solution to a DP problem
  - is typically expressed as a minimum (or maximum) of possible alternate solutions.
  - If *r* represents the cost of a solution composed of subproblems $x_1, x_2, \ldots, x_l$, then *r* can be written as

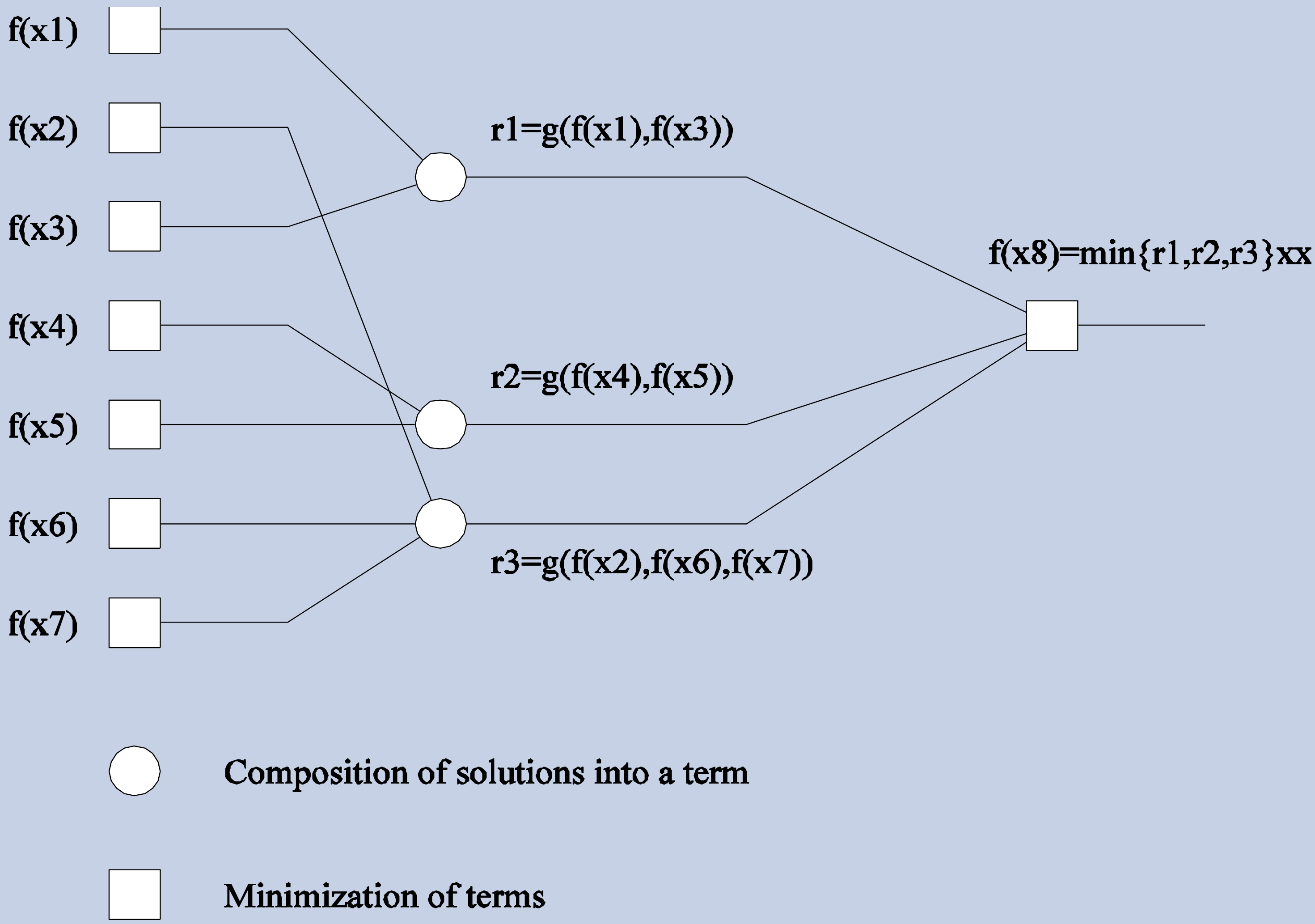
$$r = g(f(x_1), f(x_2), \ldots, f(x_l)).$$

  Here, *g* is the *composition function*.

- If the optimal solution to each problem
  - is determined by composing optimal solutions to the subproblems and
  - selecting the minimum (or maximum),

  the formulation is said to be a DP formulation.

# Dynamic Programming: Example




- The computation and composition of subproblem solutions to solve problem $f(x_8)$.

# Summarizing Dynamic Programming

- The recursive DP equation is also called
  - the *functional equation* or *optimization equation*.
  - the equation for the shortest path problem the composition function is $f(j) + c(j,x)$.
    - this contains a single recursive term ($f(j)$). Such a formulation is called monadic.
  - If the RHS has multiple recursive terms, the DP formulation is called polyadic.

# Summarizing Dynamic Programming

- The dependencies between subproblems can be expressed as a graph.
- If the graph can be levelized/linearized
  - (i.e., solutions to problems at a level depend only on solutions to problems at the previous level),
    - the formulation is called serial, else it is called non-serial.

# Summarizing Dynamic Programming

- Based on these two criteria, we can classify DP formulations into four categories
  - serial-monadic,
  - serial-polyadic,
  - non-serial-monadic,
  - non-serial-polyadic.
- This classification is useful since it identifies concurrency and dependencies that guide parallel formulations.

# Assignments

- APSP Problem : Floyd Warshall's algorithm
- Optimal Binary Search Trees
- Cutting the Iron Rod problem in CLRS