A Faster Algorithm for Checking the Dynamic Controllability of Simple Temporal Networks with Uncertainty

Luke Hunsberger
Vassar College

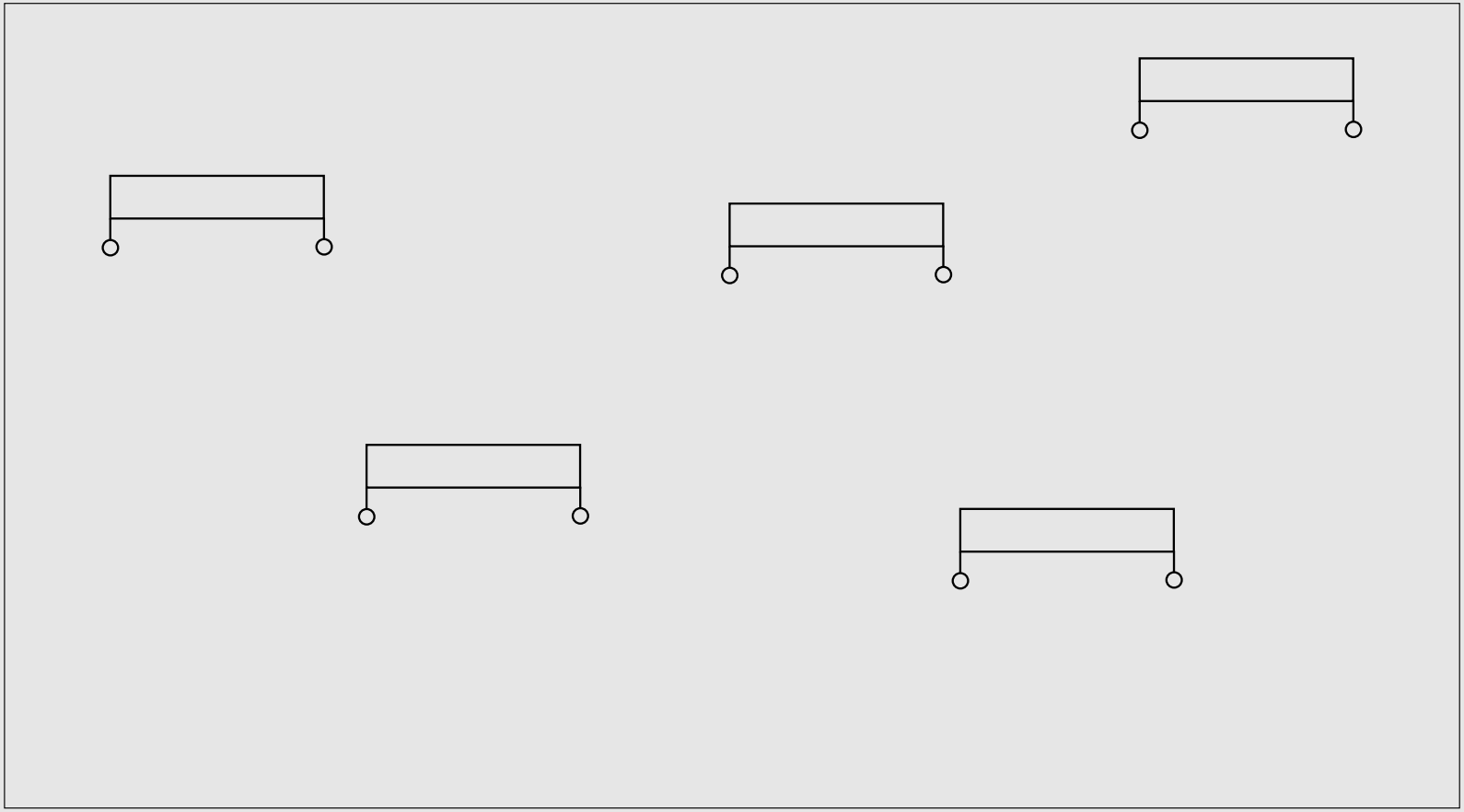
ICAART-2014
March 8, 2014

Outline

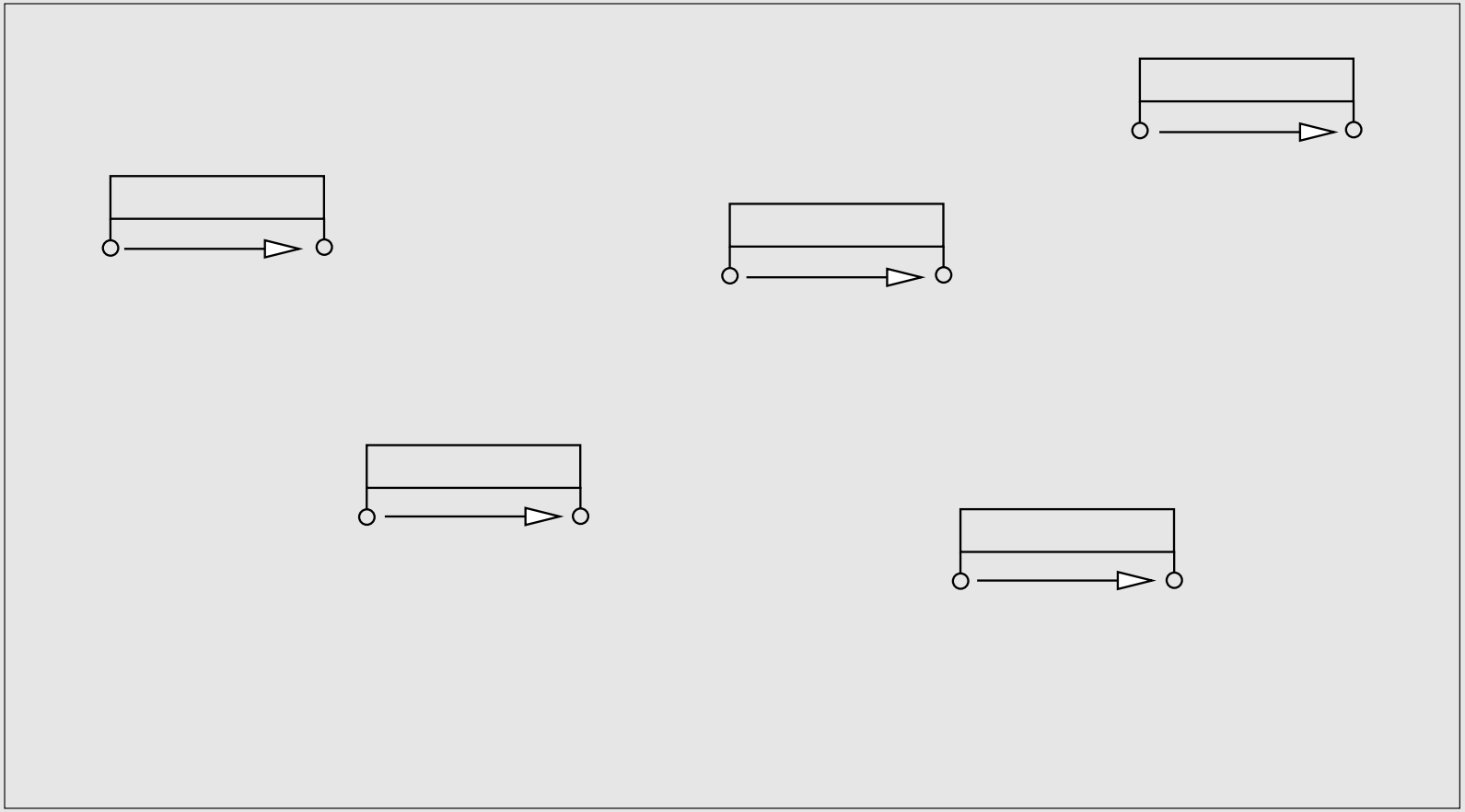
- STNUs and Dynamic Controllability
- New DC-Checking Algorithm
- Evaluation & Conclusions

STNUs and Dynamic Controllability

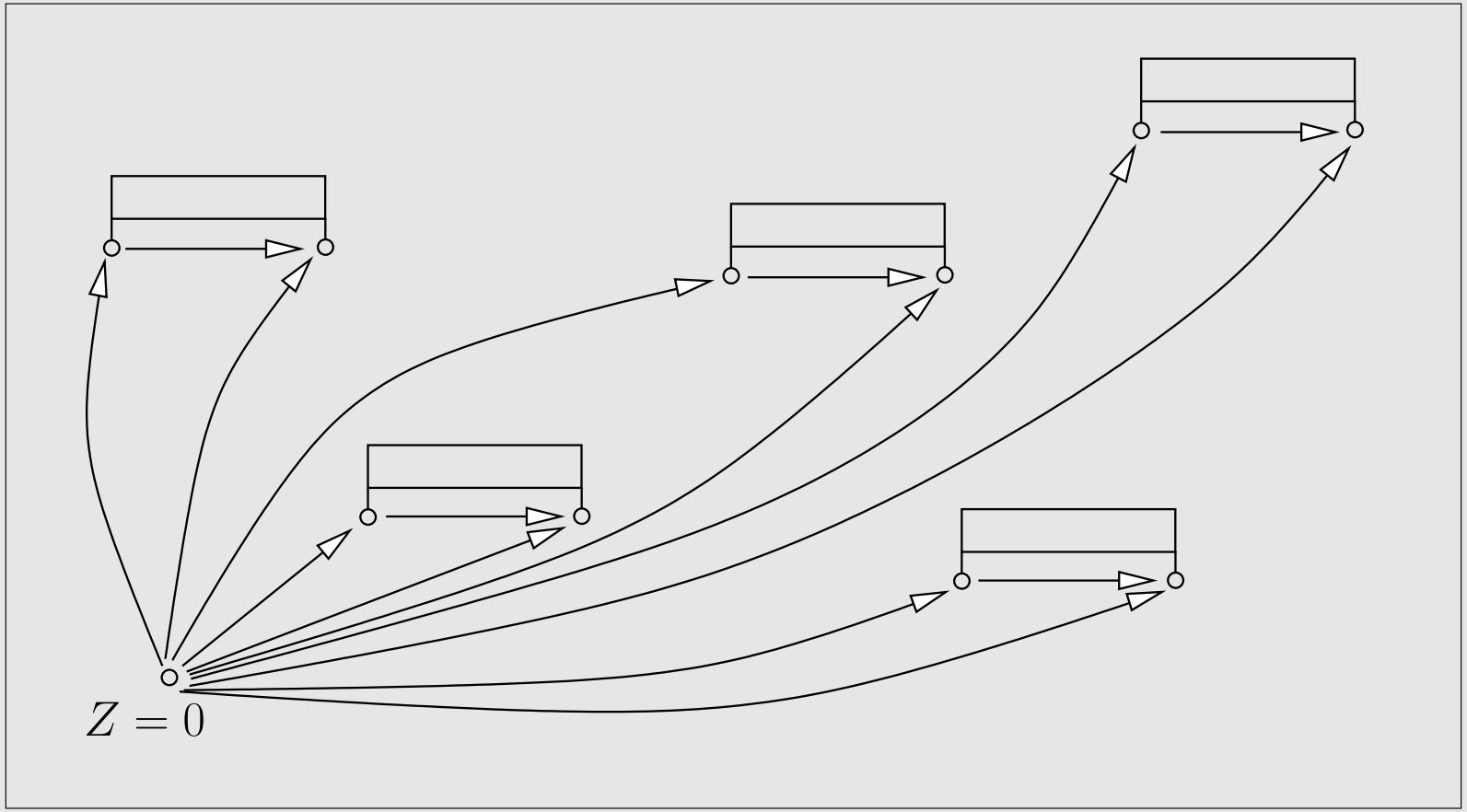
Actions



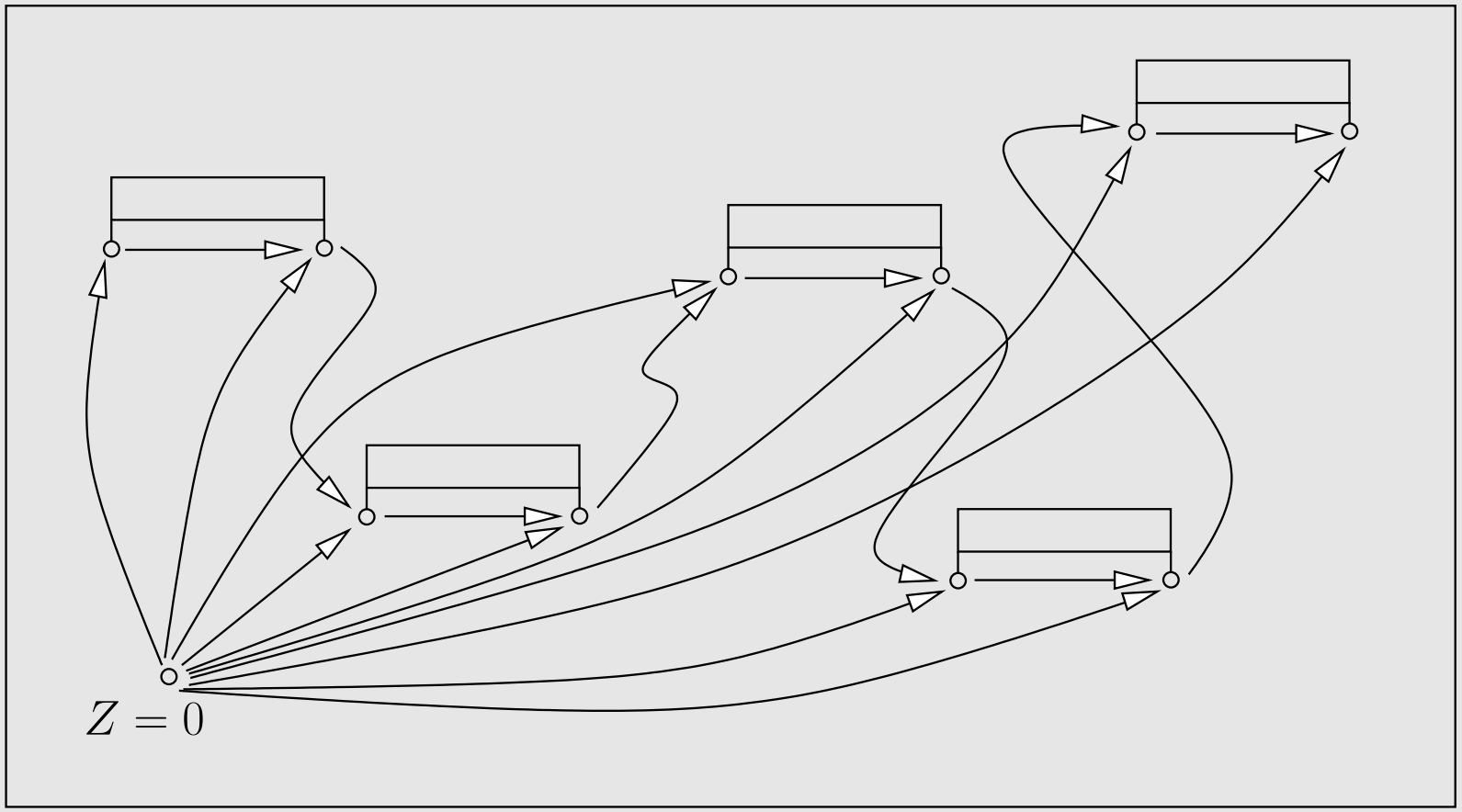
... plus Duration Constraints



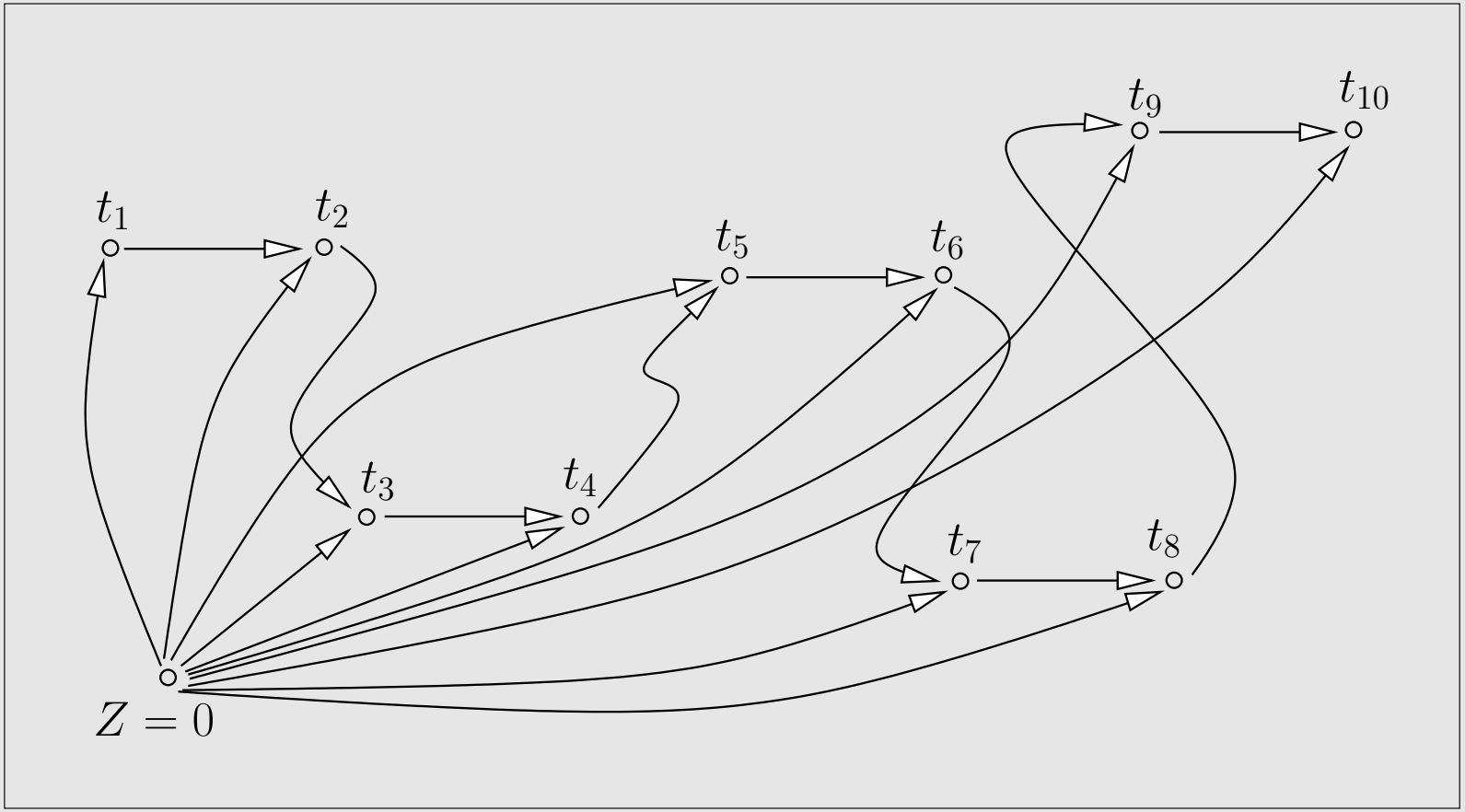
... plus Release and Deadline Constraints



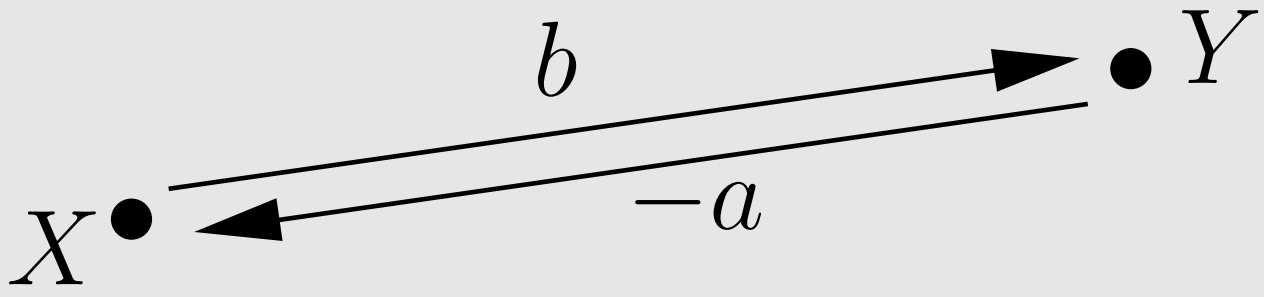
... plus Inter-Action Constraints



Simple Temporal Network (STN)

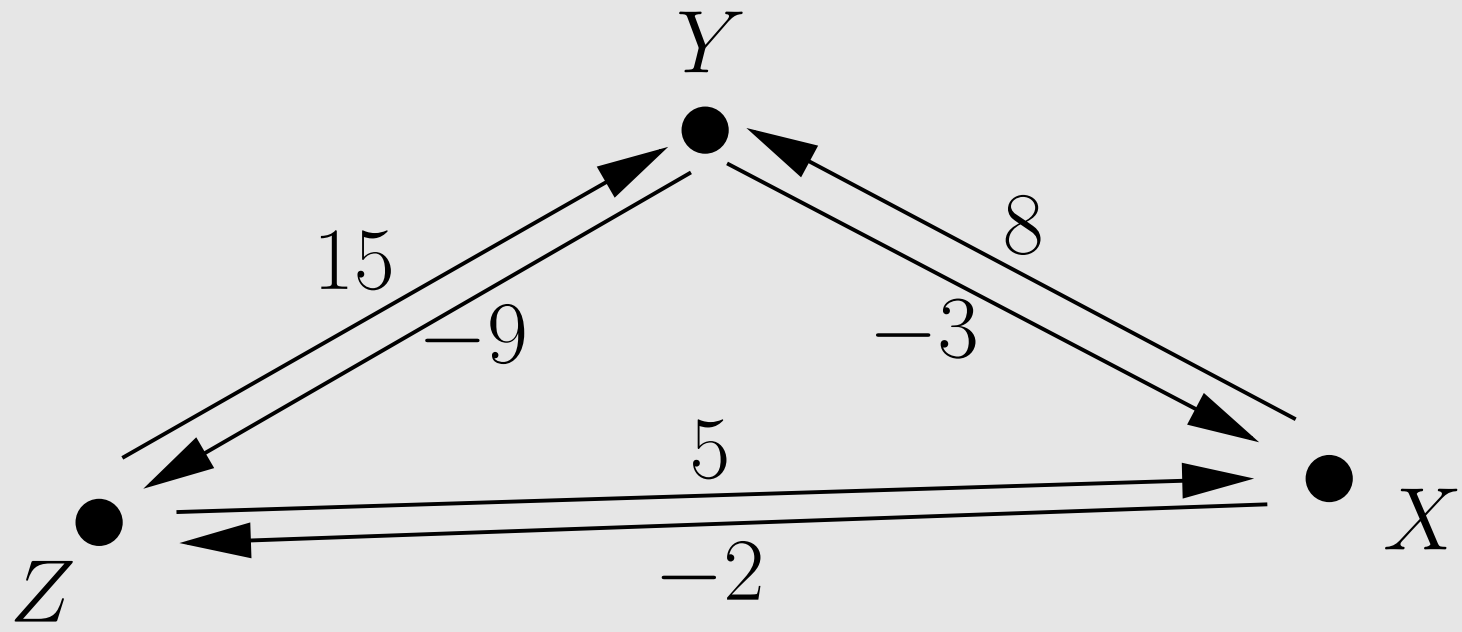


Simple Temporal Constraint



$$a \leq Y - X \leq b$$

Sample STN



Solution: $\{Z = 0,\ X = 4,\ Y = 10\}$

Fundamental Theorem for STNs

The following are equivalent:

- STN consistent
- Its graph has no negative loops
- Its *all-pairs, shortest-paths* matrix has zeroes down its main diagonal

Actions with Uncertain Durations

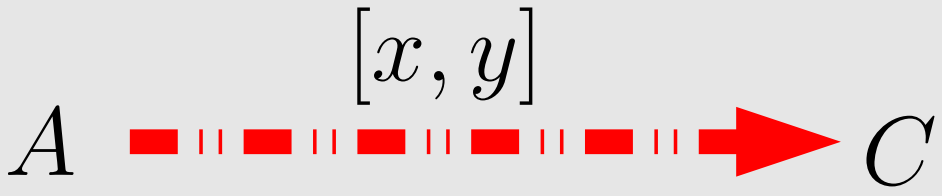
- Durations of some actions are uncertain, but within known bounds
  - Computer reboot takes 1–4 minutes
  - Driving to work takes 5–15 minutes
- Agent *initiates* action
- Agent *observes* duration in real time

STNU

Simple Temporal Network w/ Uncertainty

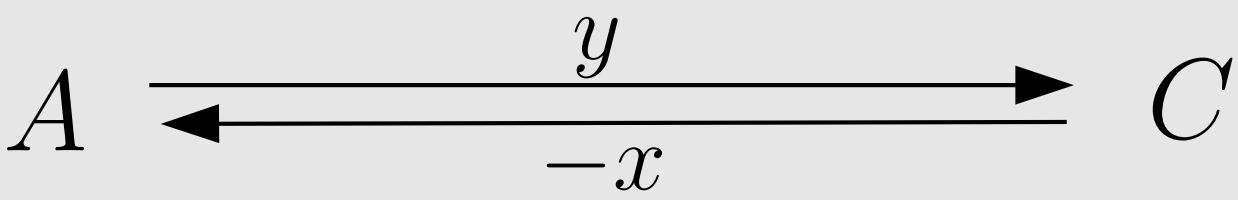
- Time Points
- Simple Temporal Constraints
- Contingent Links

Contingent Link

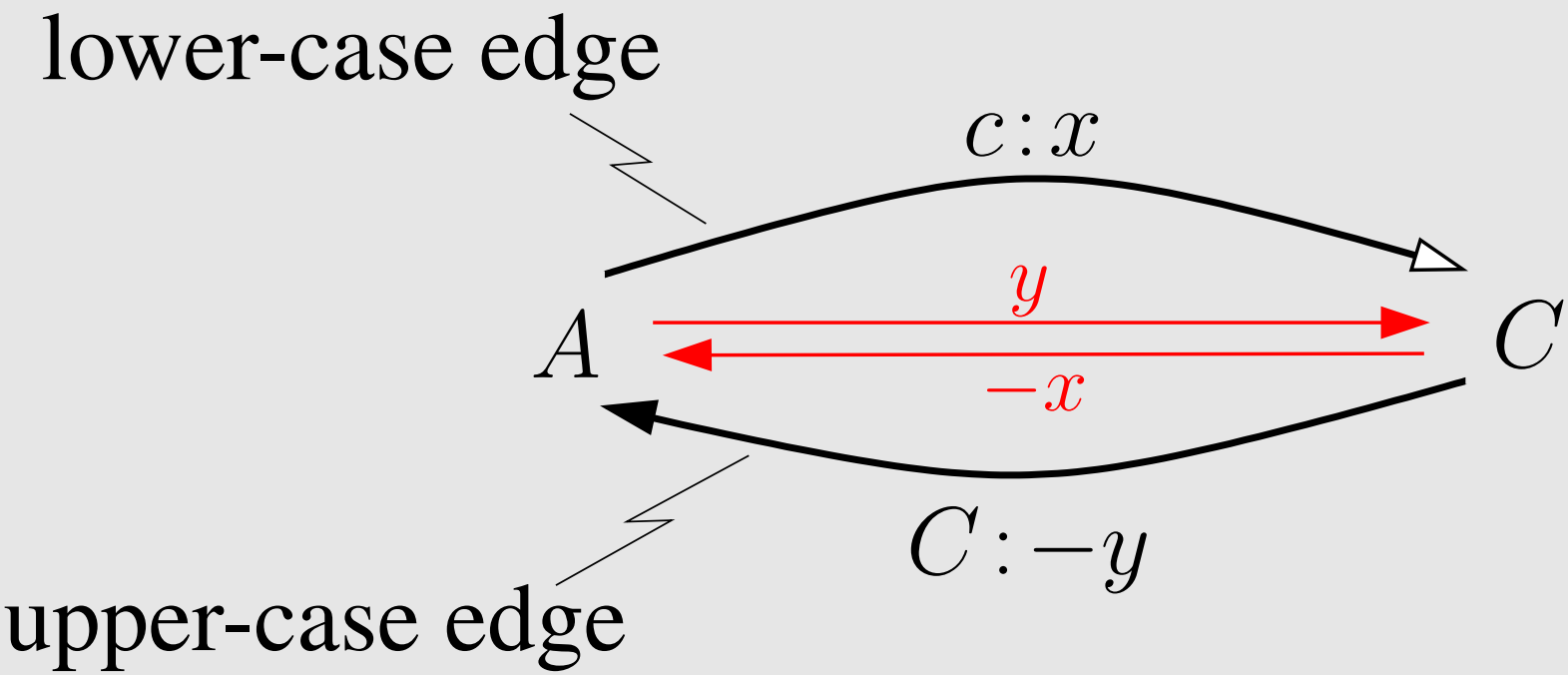


- A is the *activation* time-point
- C is the *contingent* time-point
- $0 < x < y < \infty$
- $C - A$ guaranteed to be within $[x, y]$

Contingent Link (continued)



Contingent Link (continued)



Dynamic Controllability (DC)

An STNU is *dynamically controllable* if there exists a *strategy* for executing the *non-contingent* time-points such that *all* of the constraints in the network will be satisfied—*no matter how the durations of the contingent links turn out.*

Dynamic Controllability (DC)

An STNU is *dynamically controllable* if there exists a *strategy* for executing the *non-contingent* time-points such that *all* of the constraints in the network will be satisfied—*no matter how the durations of the contingent links turn out.*

Dynamic Controllability (DC)

An STNU is *dynamically controllable* if there exists a *strategy* for executing the *non-contingent* time-points such that *all* of the constraints in the network will be satisfied—*no matter how the durations of the contingent links turn out.*

Dynamic Controllability (DC)

An STNU is *dynamically controllable* if there exists a *strategy* for executing the *non-contingent* time-points such that *all* of the constraints in the network will be satisfied—*no matter how the durations of the contingent links turn out.*

Dynamic Controllability (DC)

An STNU is *dynamically controllable* if there exists a *strategy* for executing the *non-contingent* time-points such that *all* of the constraints in the network will be satisfied—*no matter how the durations of the contingent links turn out.*

DC Checking vs. Execution

- DC Checking: Determine whether execution strategy *exists.*
- Execution: Generate execution strategy *on the fly.*

$\Rightarrow$ Information generated by DC-Checking algorithm is needed to generate execution strategy!

DC-Checking Algorithm for STNUs

- Fastest DC-checking algorithm, due to Morris (2006), is based on generation of new edges in STNU graph.
- Edge generation is more complex than in APSP due to presence of labeled edges:
  - Labeled edges: uncontr'ble possibilities
  - Ordinary edges: constraints to satisfy

Morris' DC-Checking Algorithm

- $O(N^4)$-time algorithm
- Uses edge-generation rules from Morris and Muscettola (2005), but focuses on *"reducing away lower-case edges"*
- After edge generation, only need to do an $O(N^3)$-time consistency check.

Important Result for STNUs

An STNU is dynamically controllable if and only if its graph has no *semi-reducible negative* (SRN) loops.

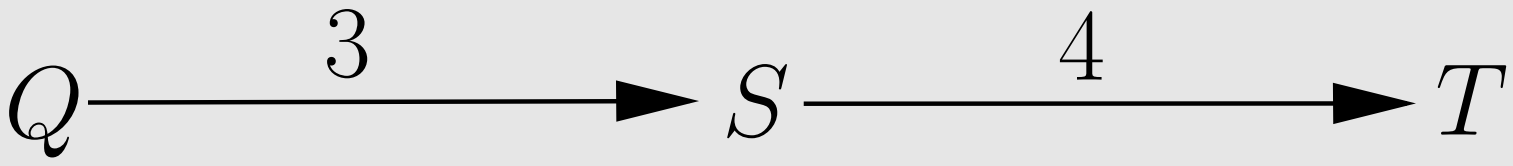
—Morris (2006)

Edge-Generation Rules

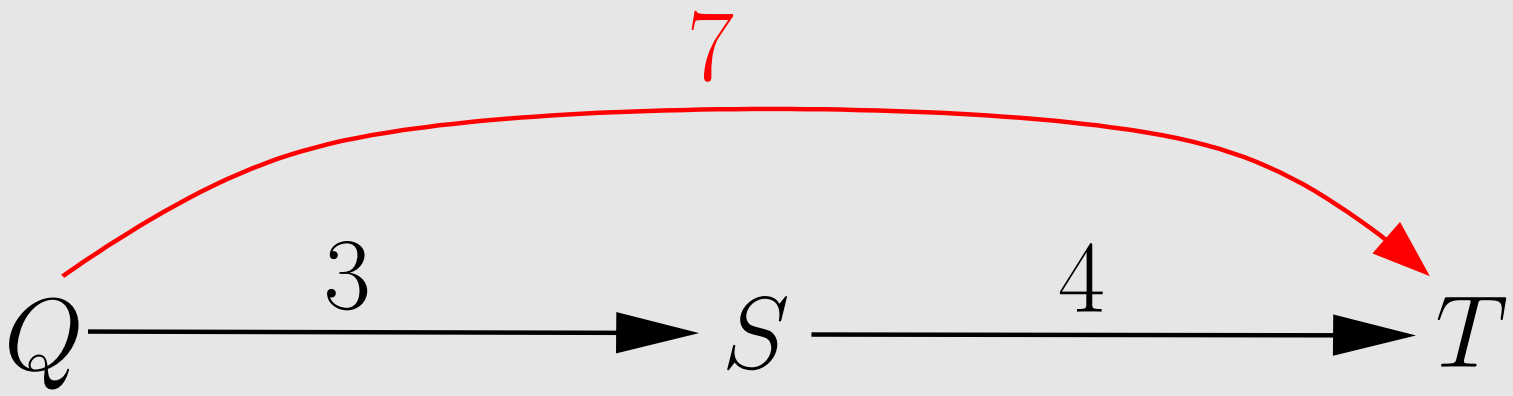
- *No Case* Rule
- *Upper-Case* Rule
- *Lower-Case* Rule
- *Cross-Case* Rule
- *Label-Removal* Rule

— *(Morris & Muscettola, 2005)*

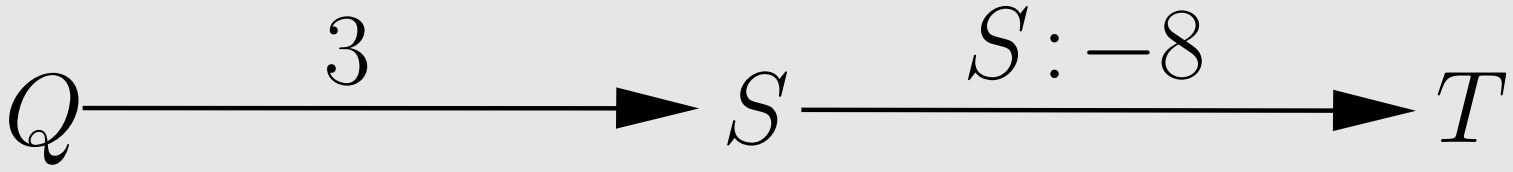
The No-Case Rule



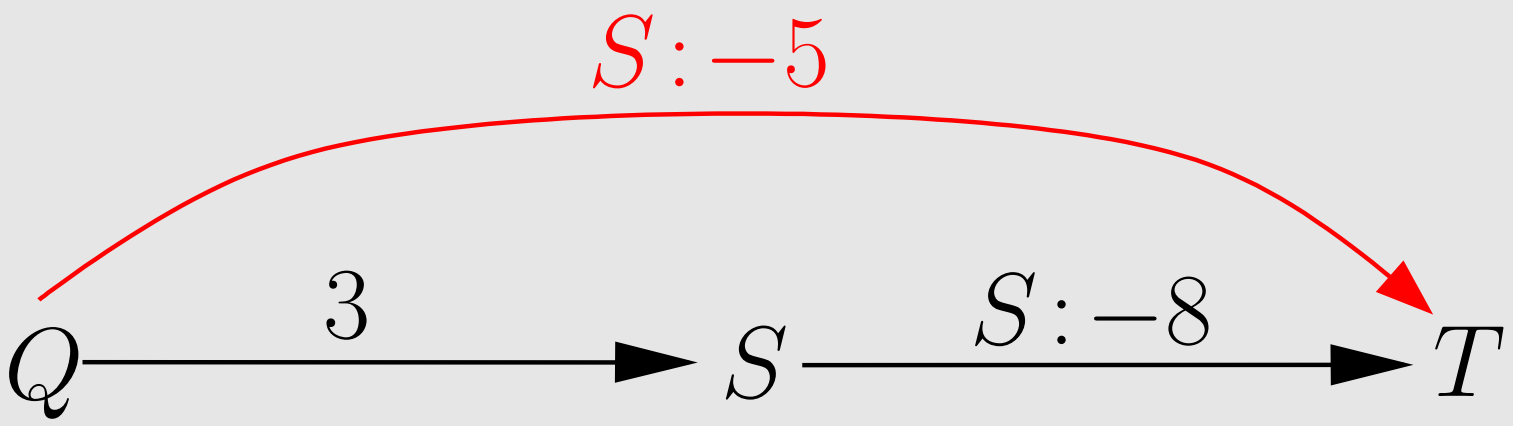
The No-Case Rule



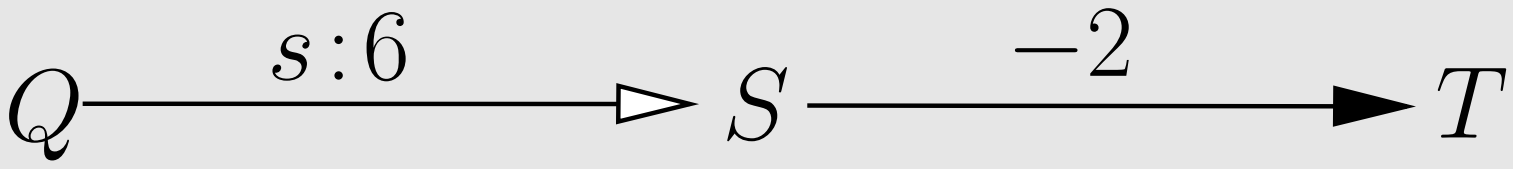
The Upper-Case Rule



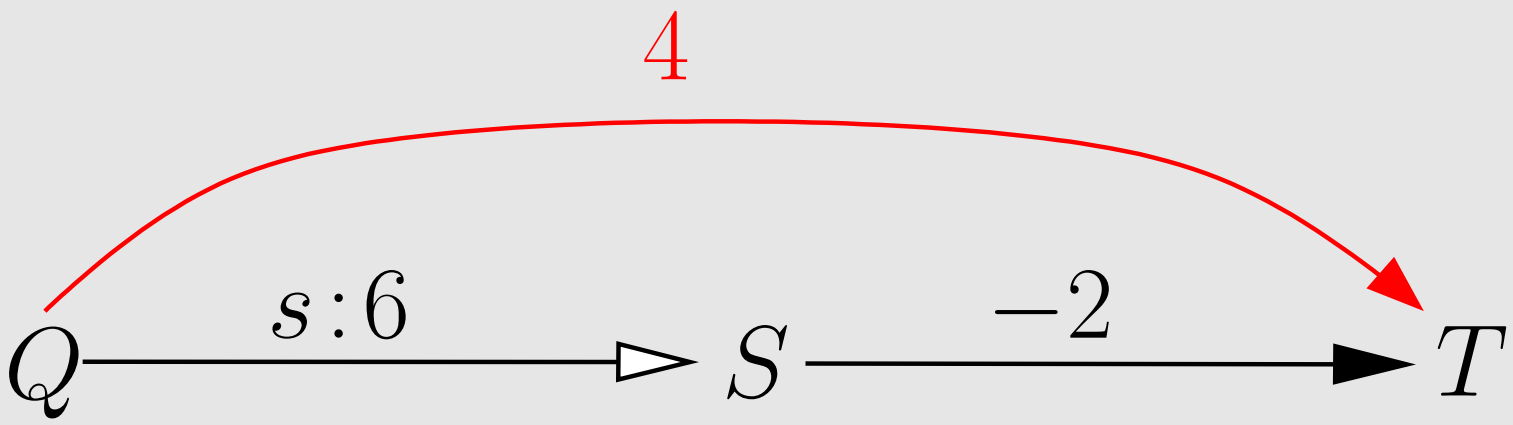
The Upper-Case Rule



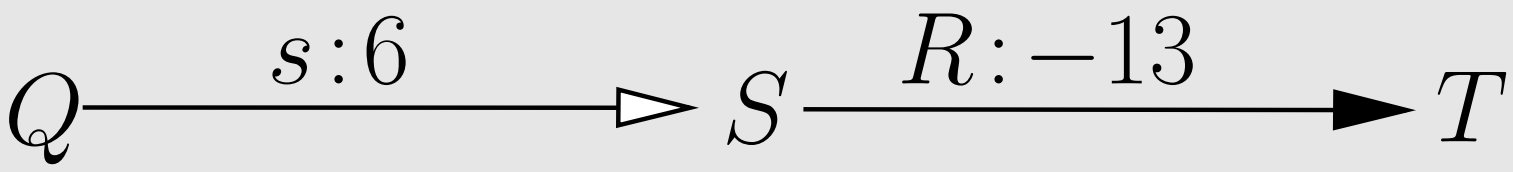
The Lower-Case Rule



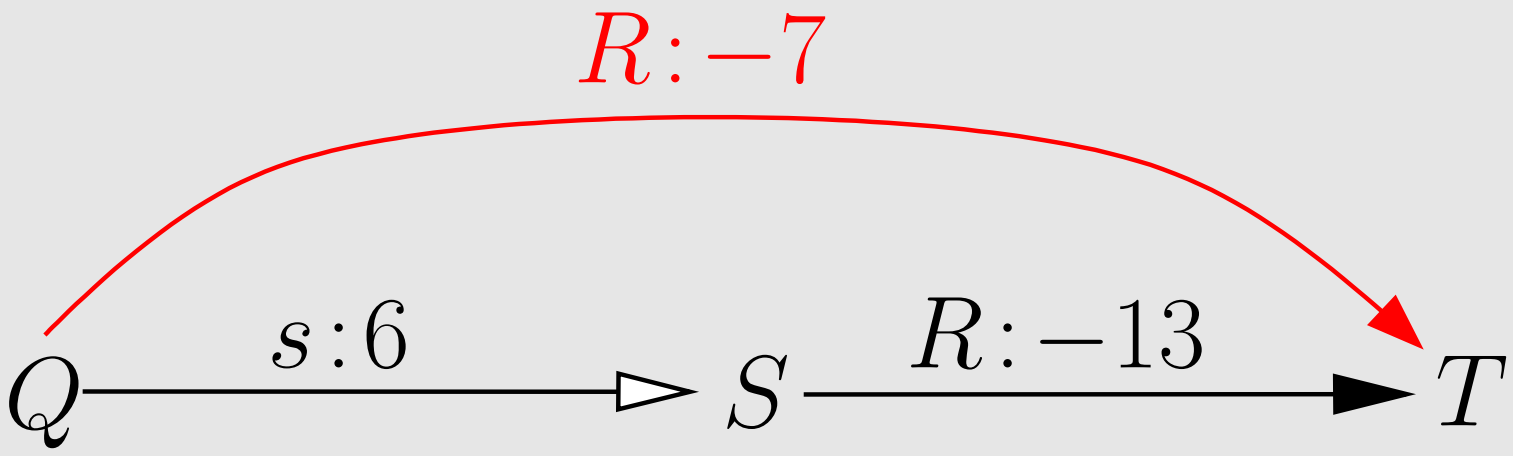
The Lower-Case Rule



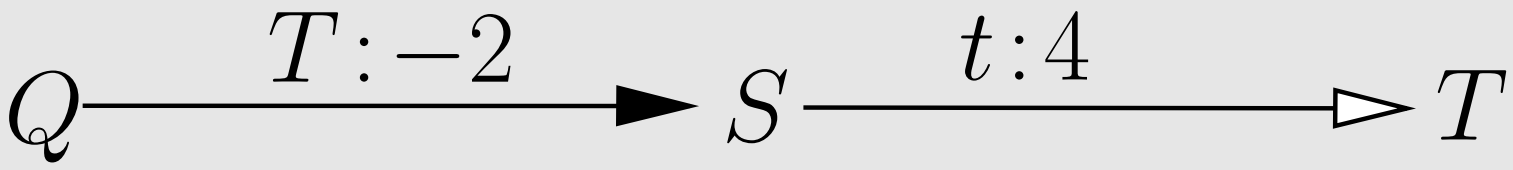
The Cross-Case Rule



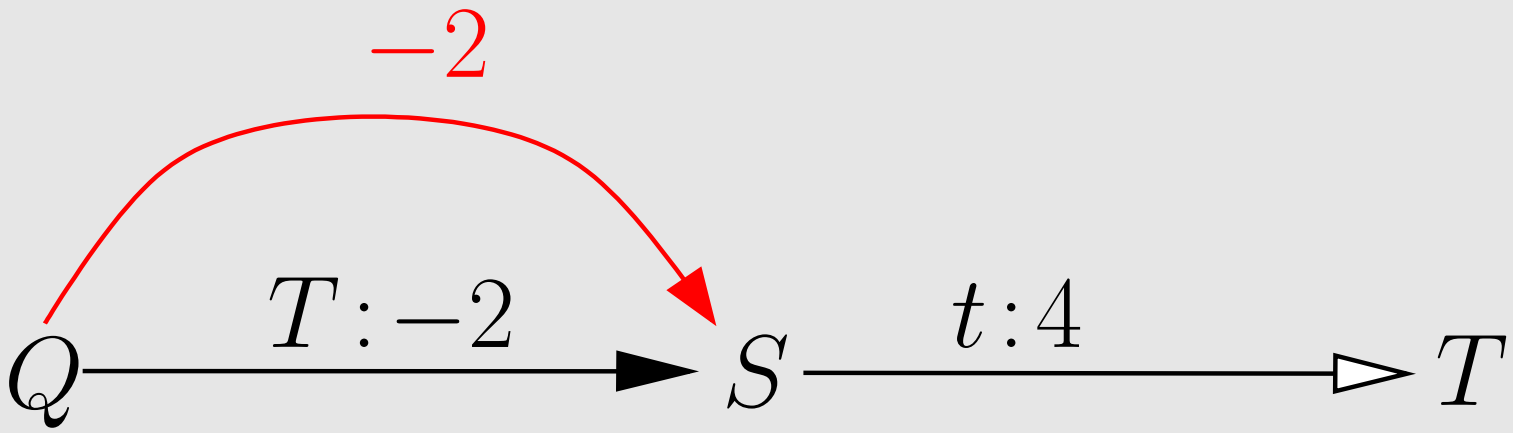
The Cross-Case Rule



The Label-Removal Rule



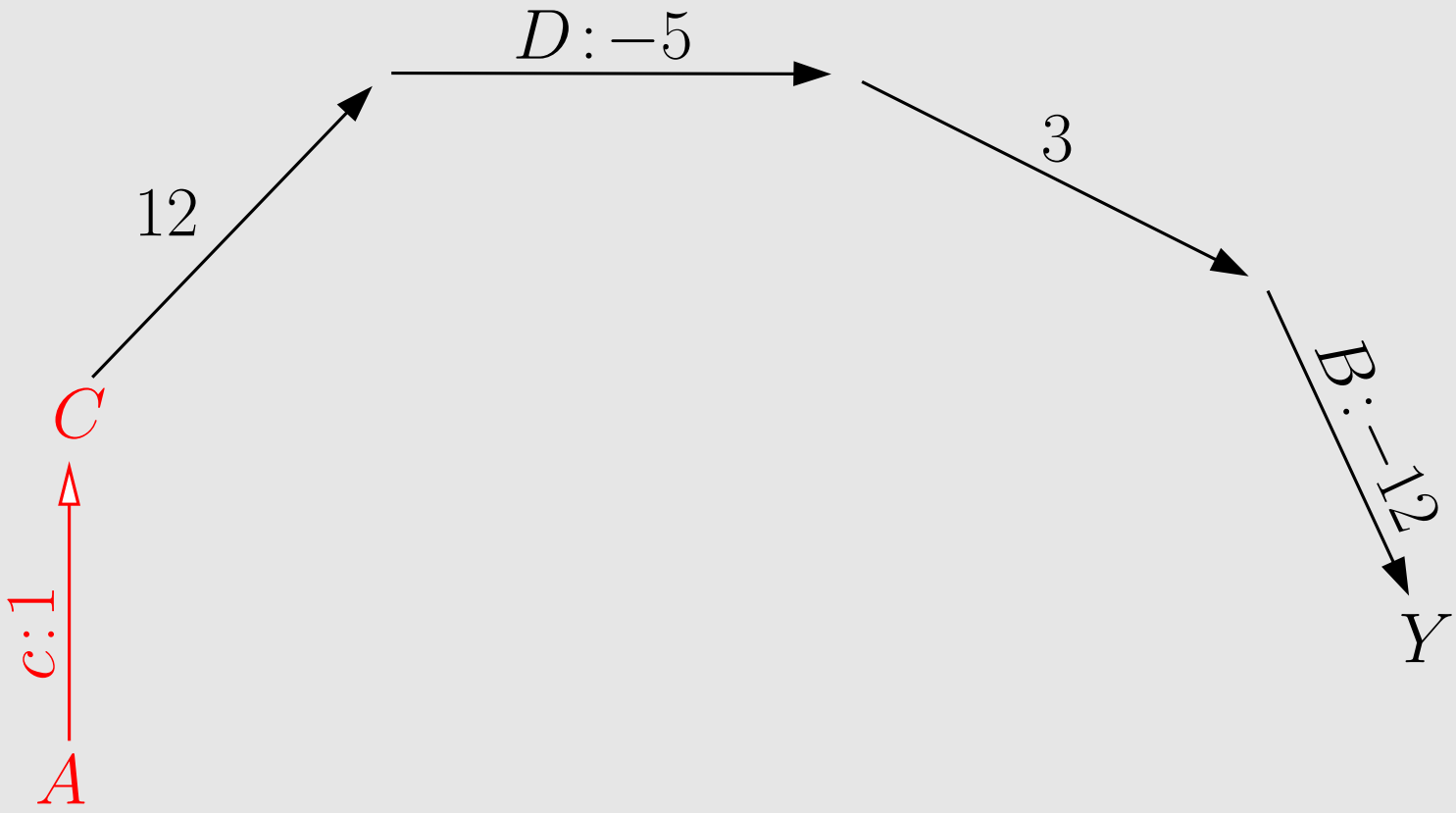
The Label-Removal Rule)



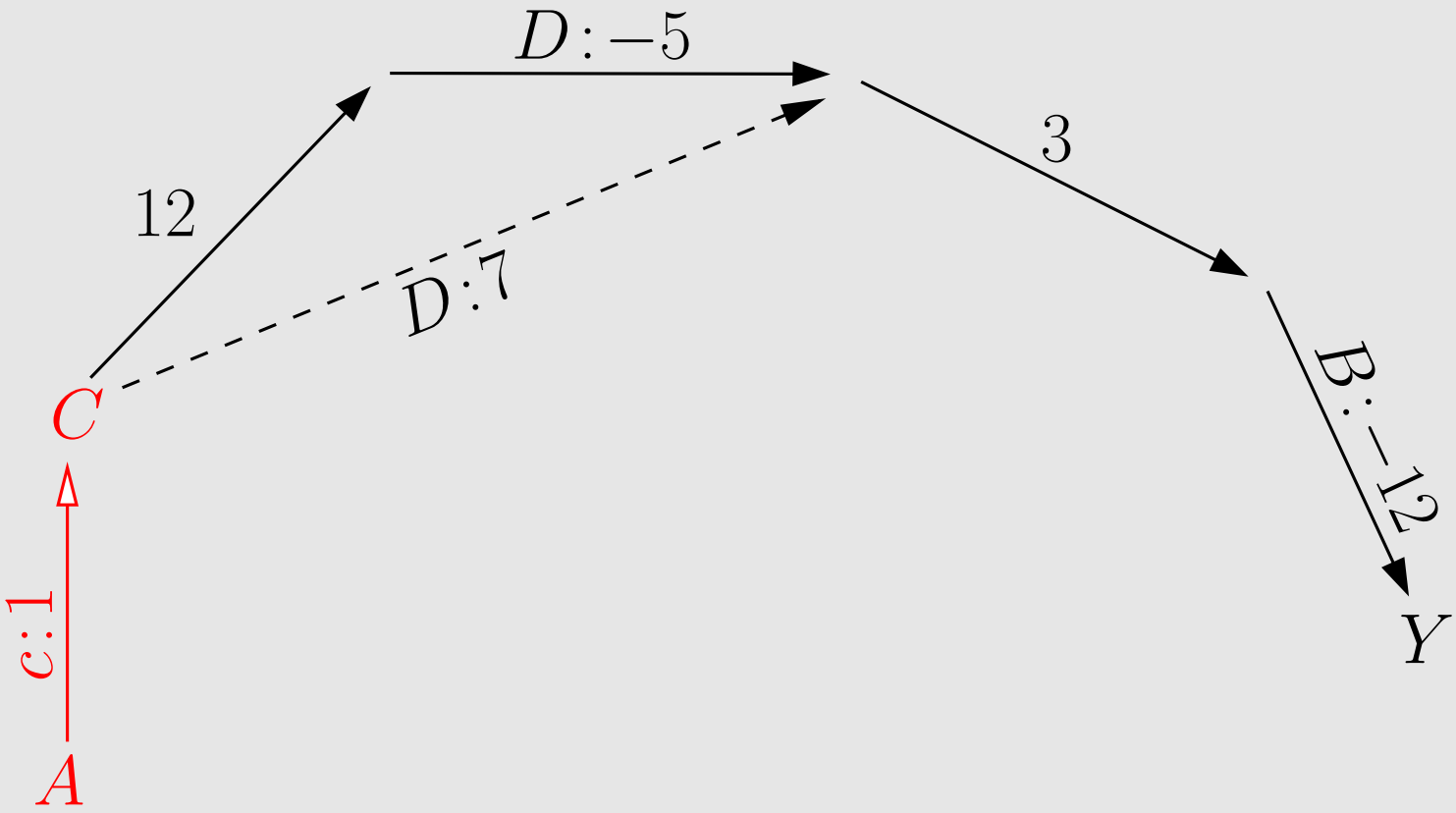
Edge Generation Properties

- Edge generation preserves path length
- Cannot generate new lower-case edges

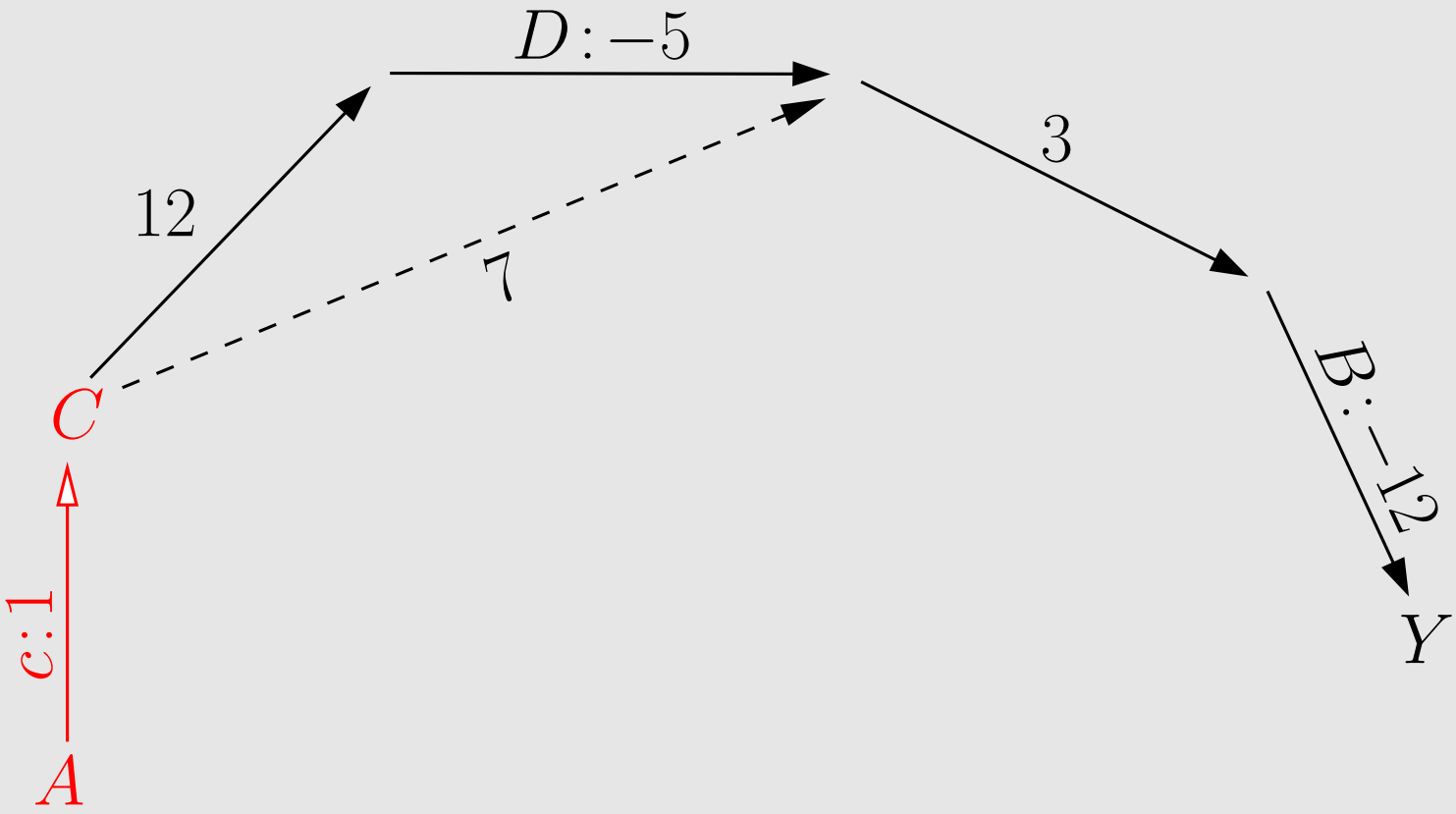
Reducing Away Lower-Case Edges



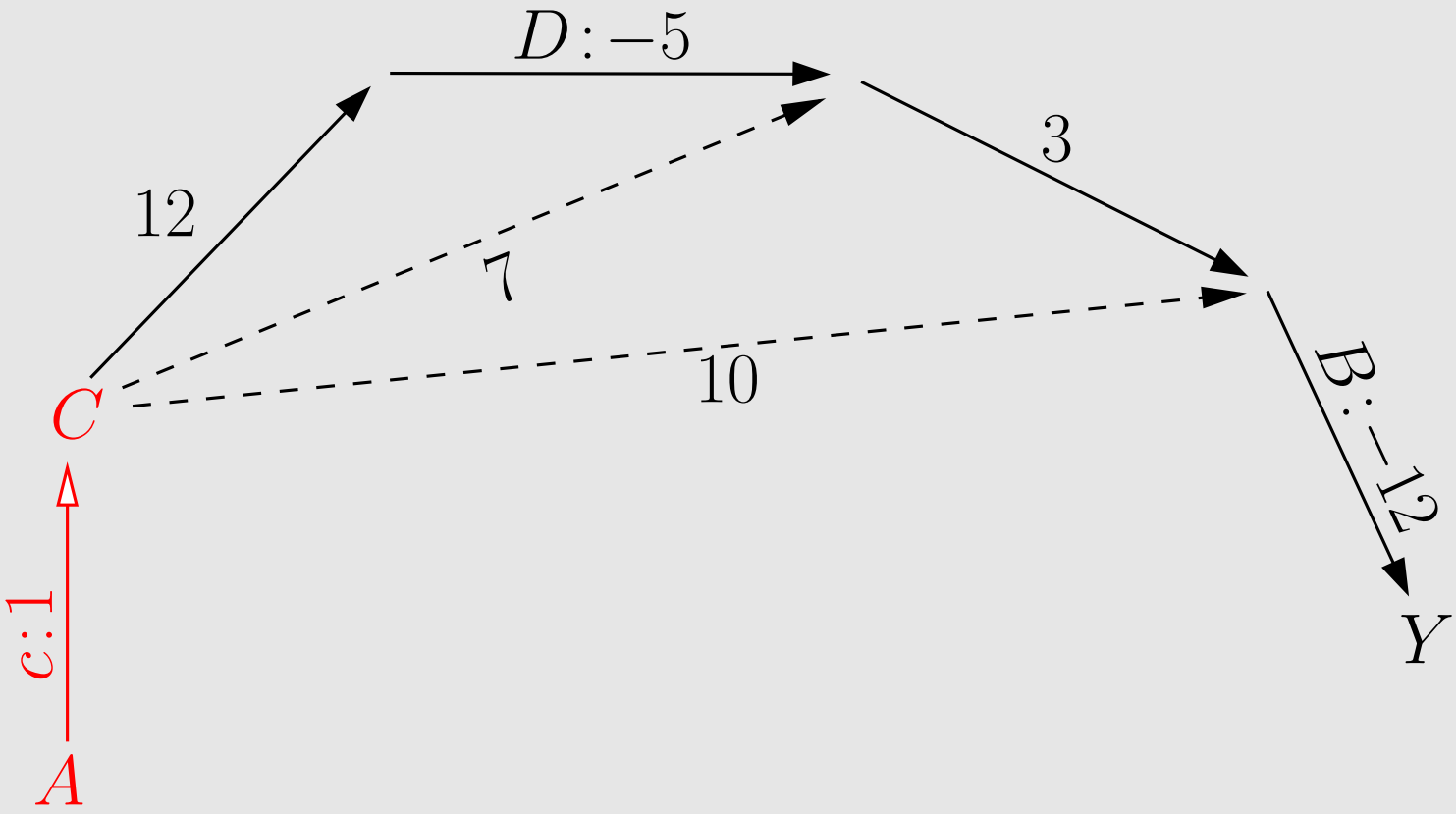
Reducing Away Lower-Case Edges



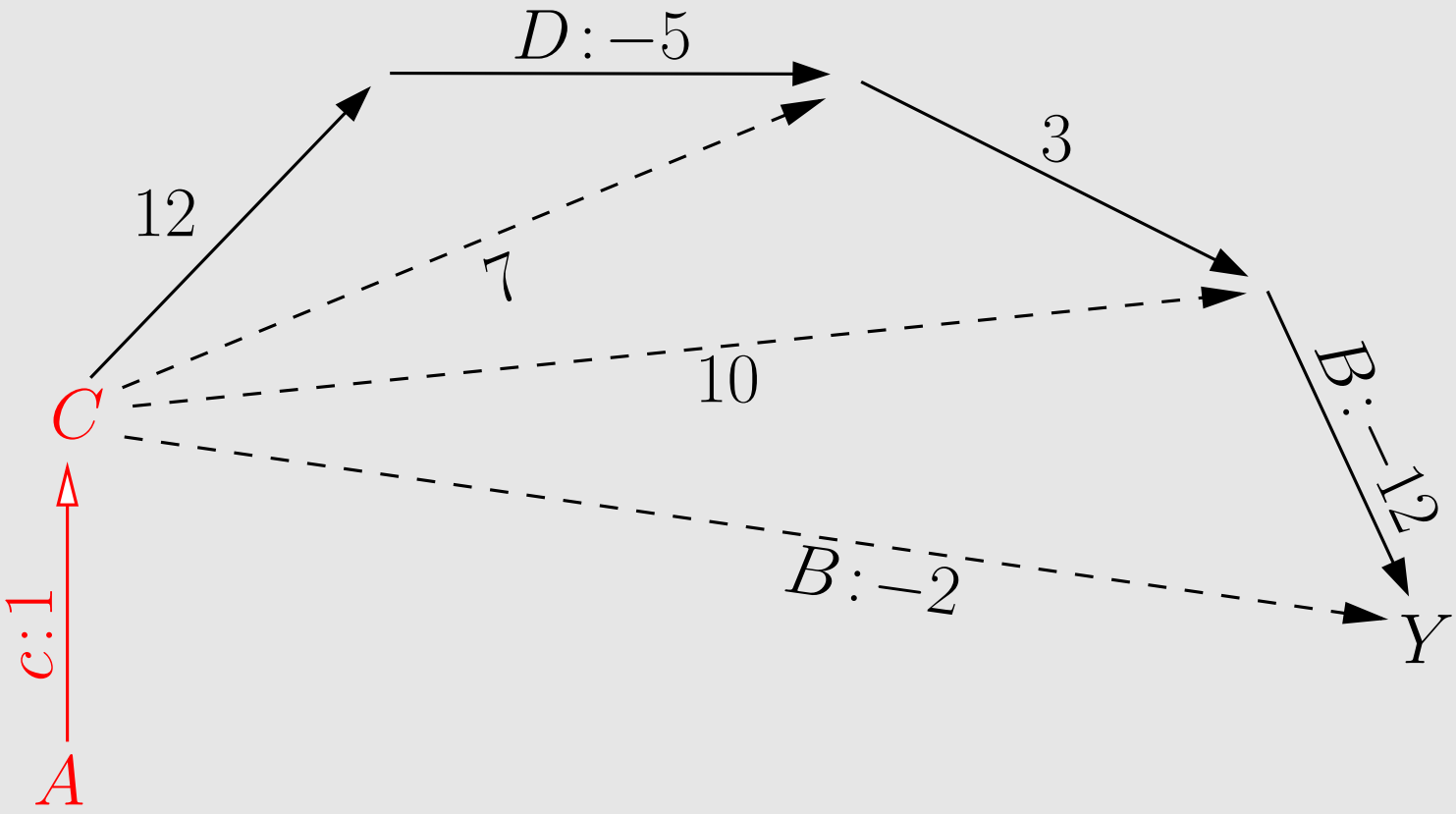
Reducing Away Lower-Case Edges



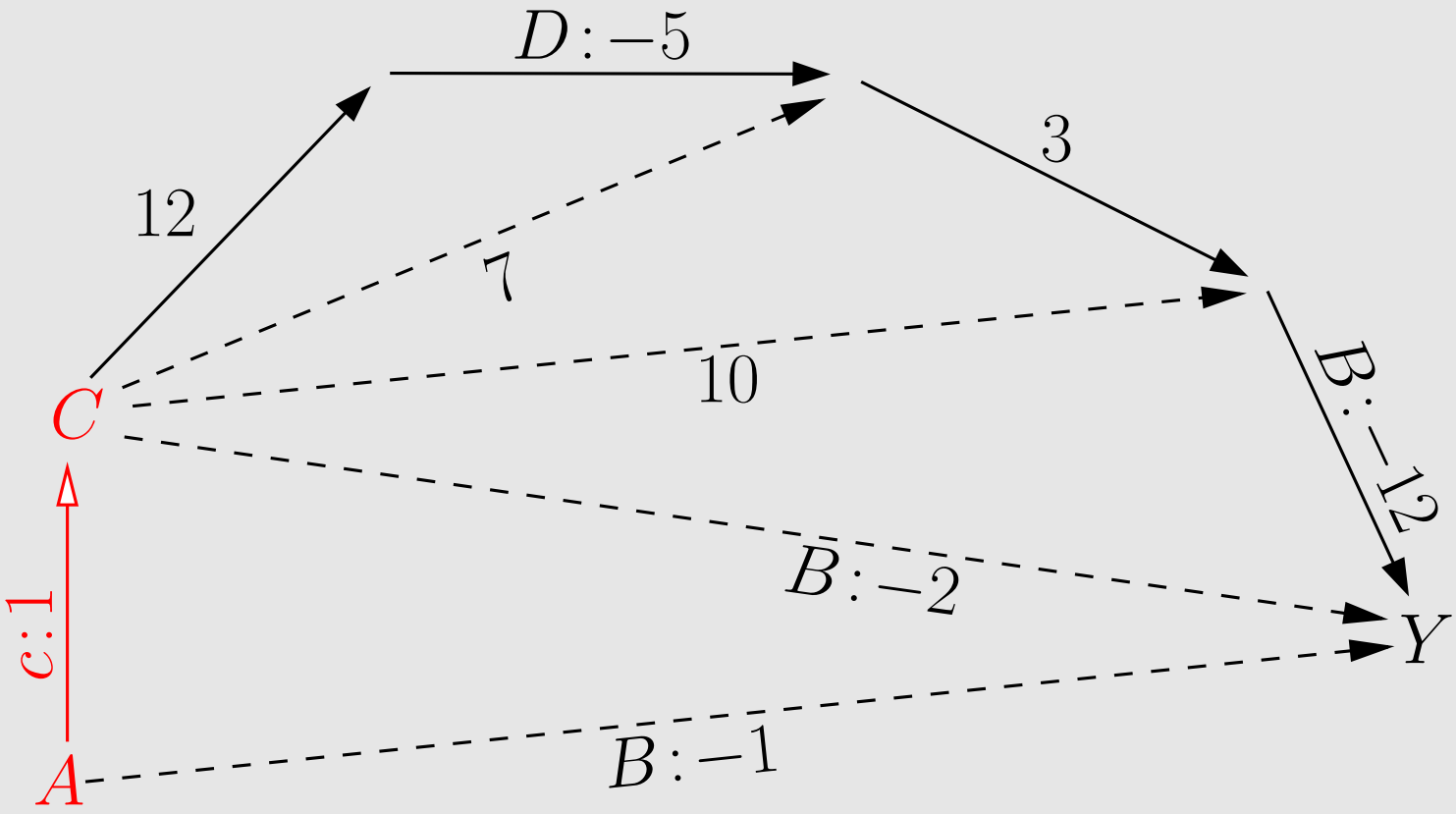
Reducing Away Lower-Case Edges



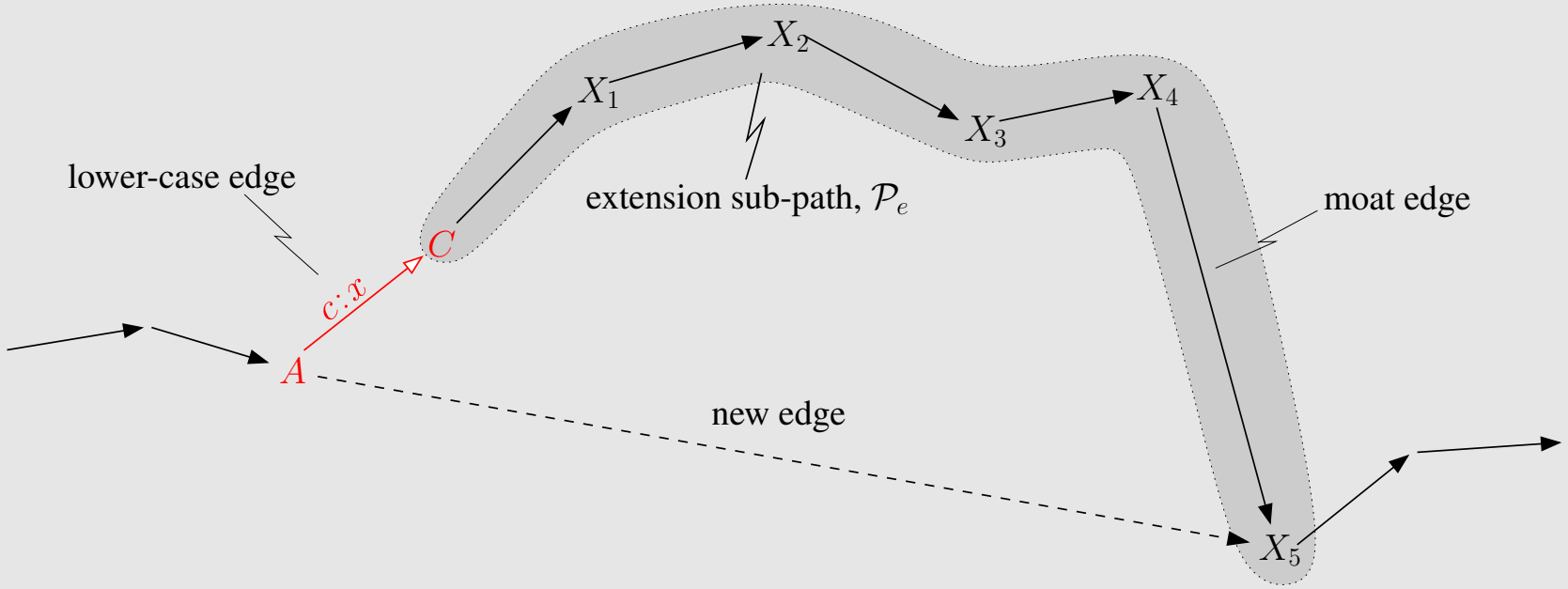
Reducing Away Lower-Case Edges



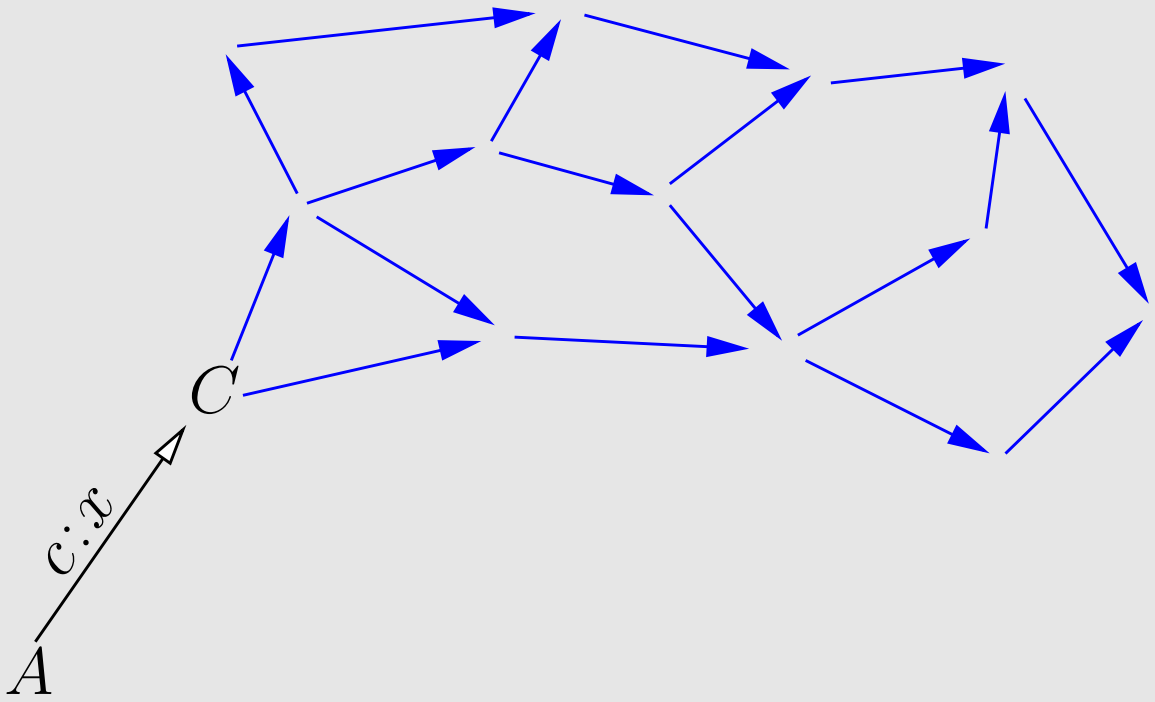
Reducing Away Lower-Case Edges



Extension Sub-Path for Lower-Case Edge

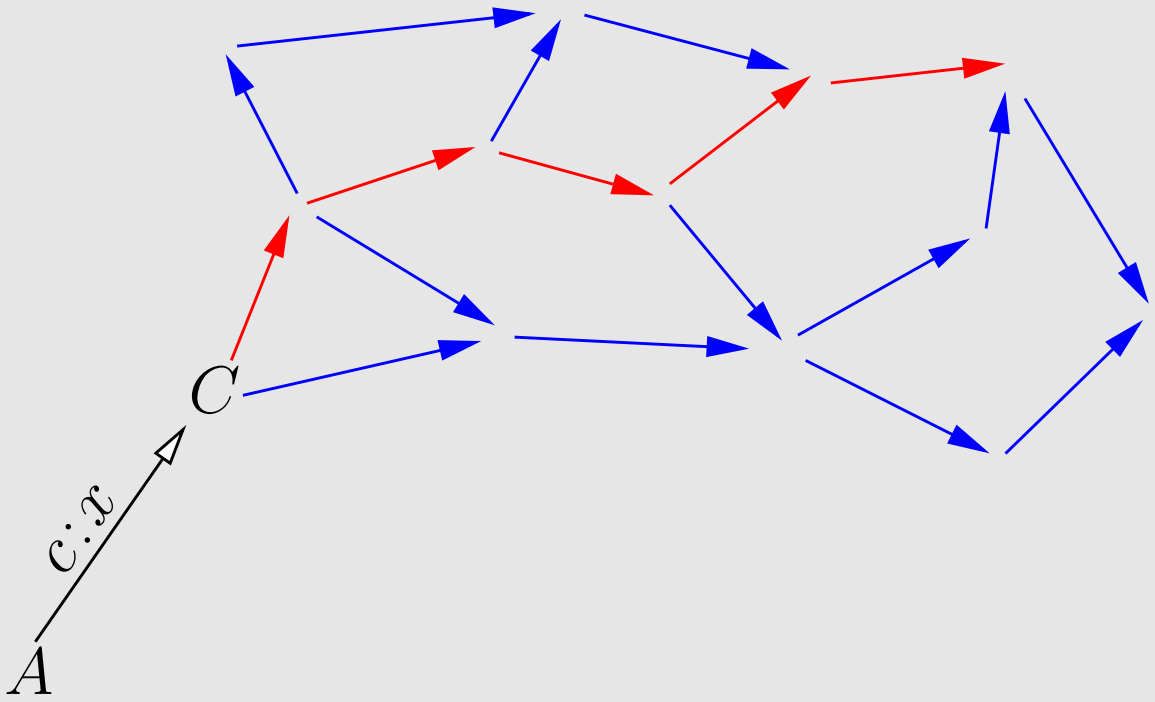


Central Process of Morris' Algorithm



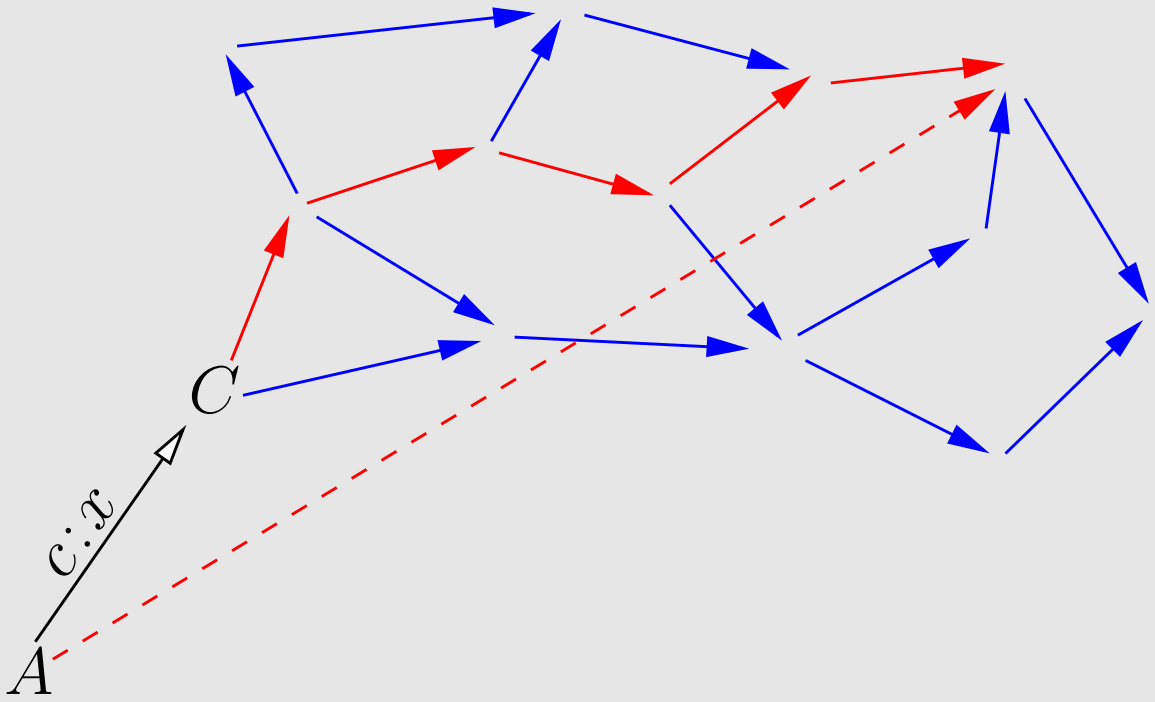
For a lower-case edge, $A \xrightarrow{c:x} C$, traverse paths emanating from C, searching for *extension sub-paths* to generate new edges.

Central Process of Morris' Algorithm



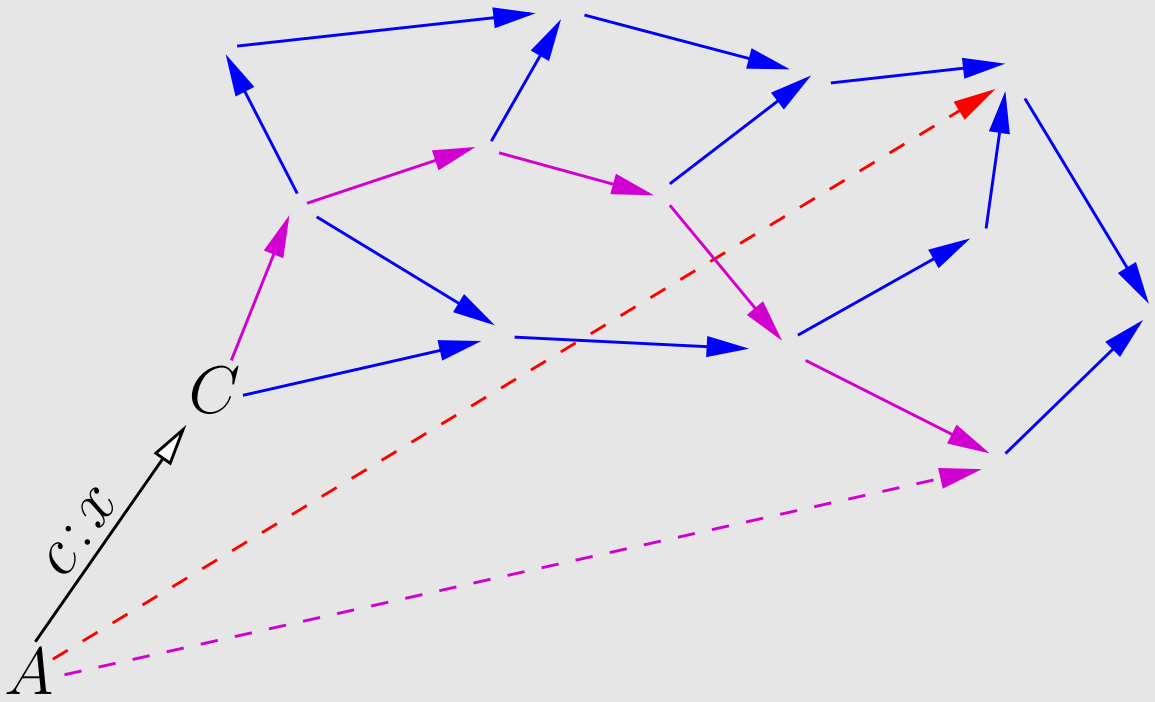
For a lower-case edge, $A \xrightarrow{c:x} C$, traverse paths emanating from C, searching for *extension sub-paths* to generate new edges.

Central Process of Morris' Algorithm



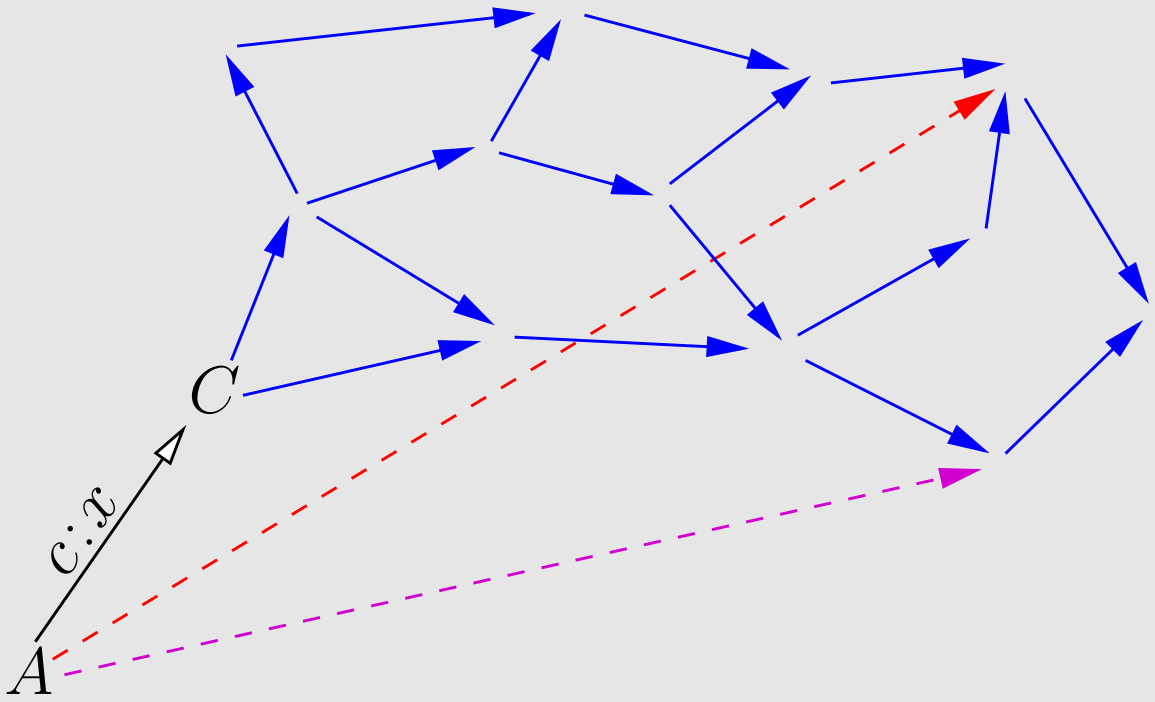
For a lower-case edge, $A \xrightarrow{c:x} C$, traverse paths emanating from C, searching for *extension sub-paths* to generate new edges.

Central Process of Morris' Algorithm



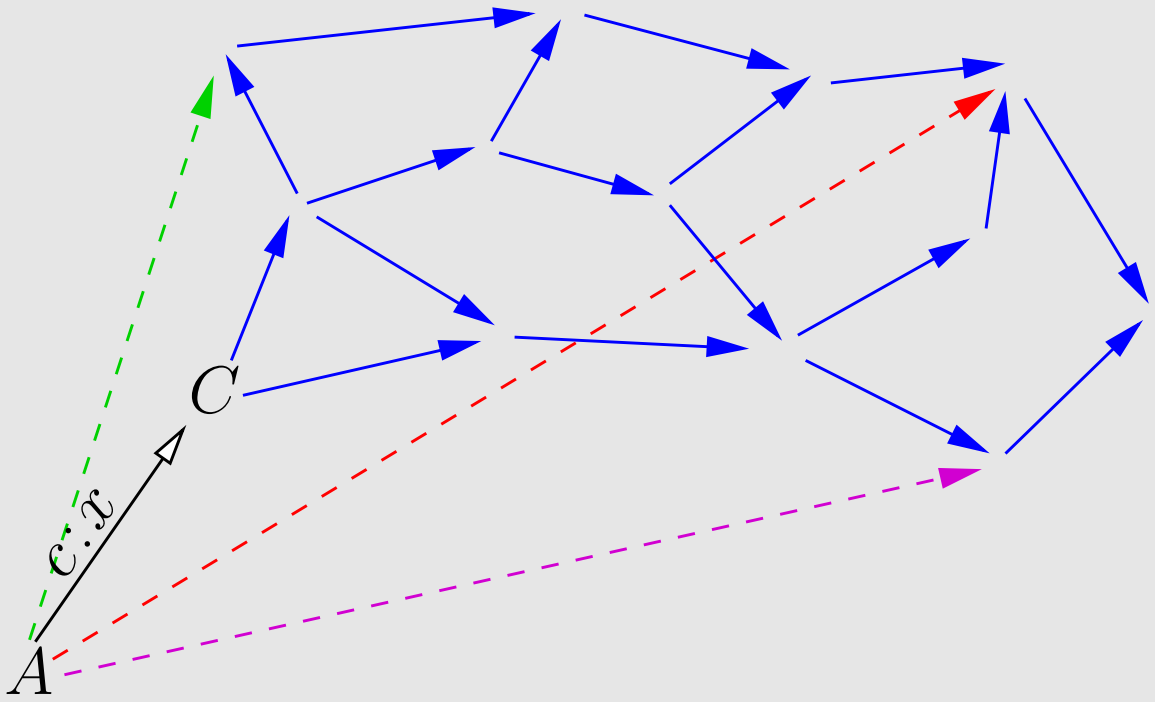
For a lower-case edge, $A \xrightarrow{c:x} C$, traverse paths emanating from C, searching for *extension sub-paths* to generate new edges.

Central Process of Morris' Algorithm



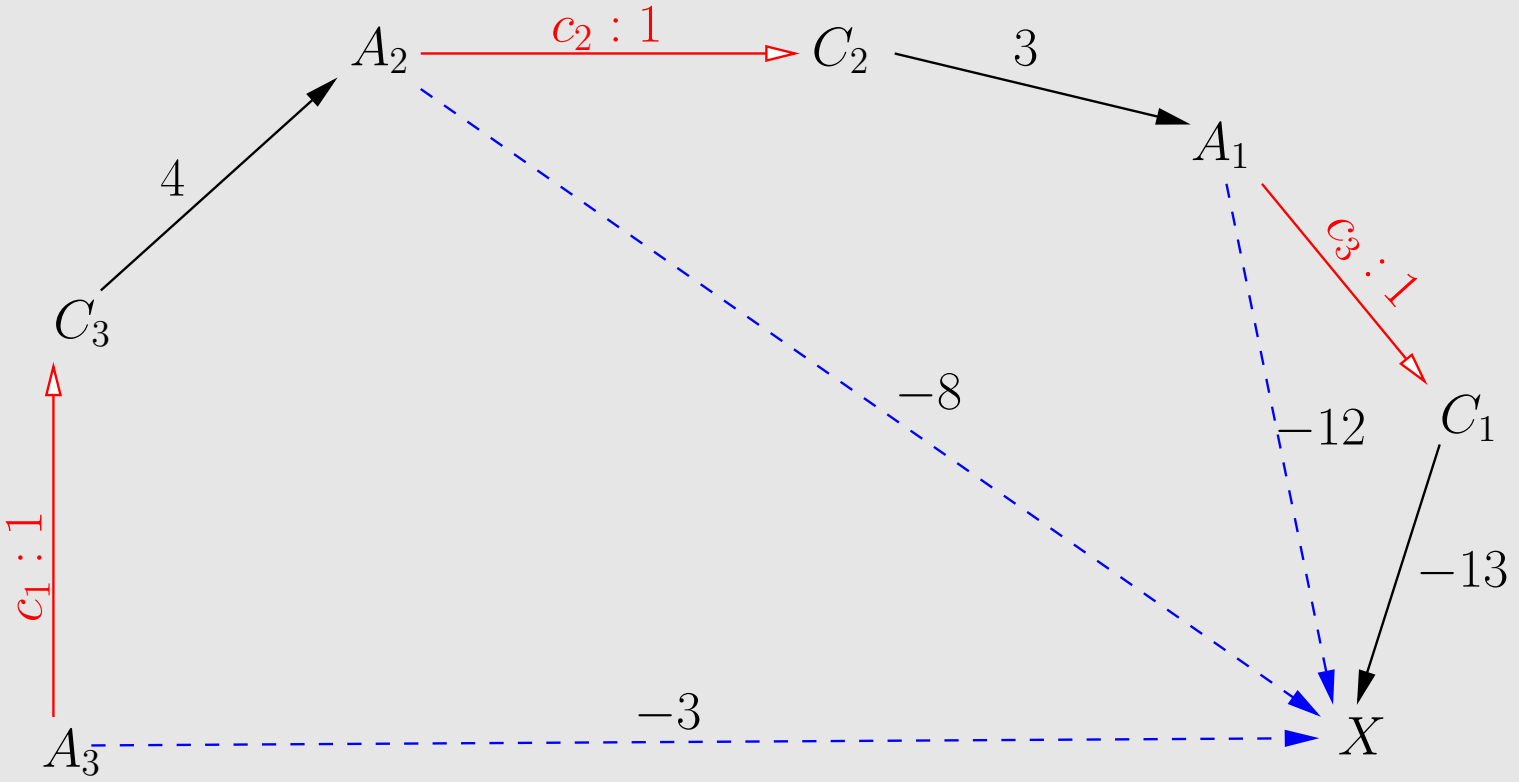
For a lower-case edge, $A \xrightarrow{c:x} C$, traverse paths emanating from C, searching for *extension sub-paths* to generate new edges.

Central Process of Morris' Algorithm



For a lower-case edge, $A \xrightarrow{c:x} C$, traverse paths emanating from C, searching for *extension sub-paths* to generate new edges.

Extension Sub-Paths can be Nested



Morris Theorem: Only need to consider nesting to depth of K.

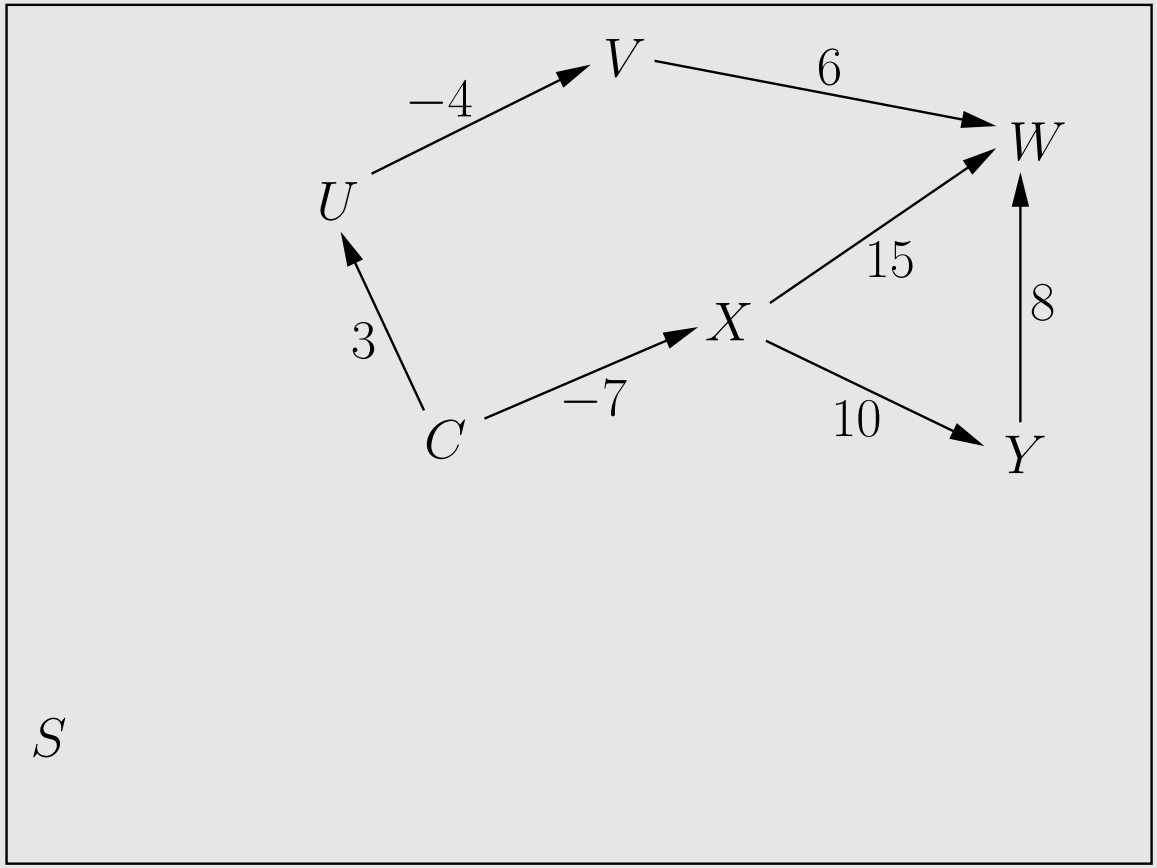
Morris' Algorithm

```
for i = 1 to K                          // Nesting Depth
  for j = 1 to K                        // Contingent Links
    Process j^th Contingent Link
do APSP
```

Morris' Algorithm

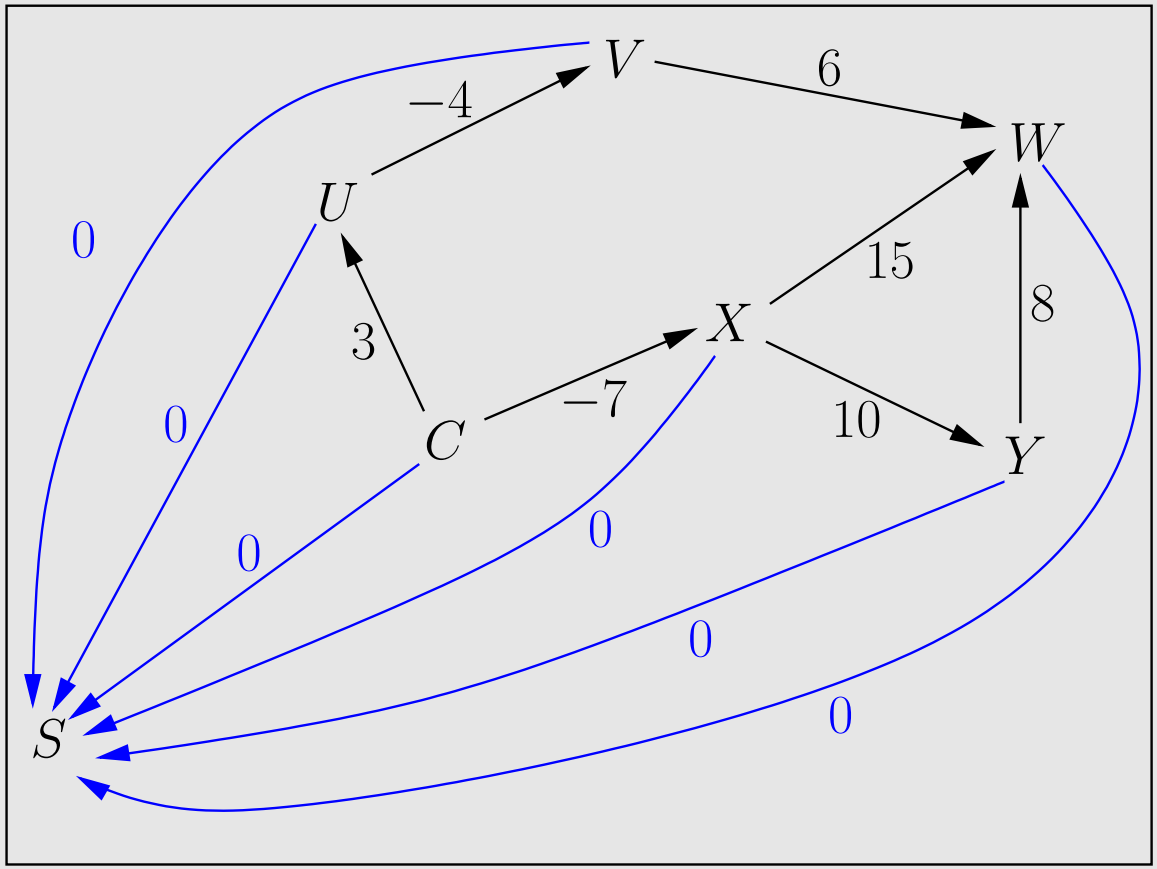
```
for i = 1 to K                          // Nesting Depth
BellmanFord                             // Generate Potential Function
  for j = 1 to K                        // Contingent Links
    Process j^th Contingent Link
do APSP
```

Enabling Dijkstra



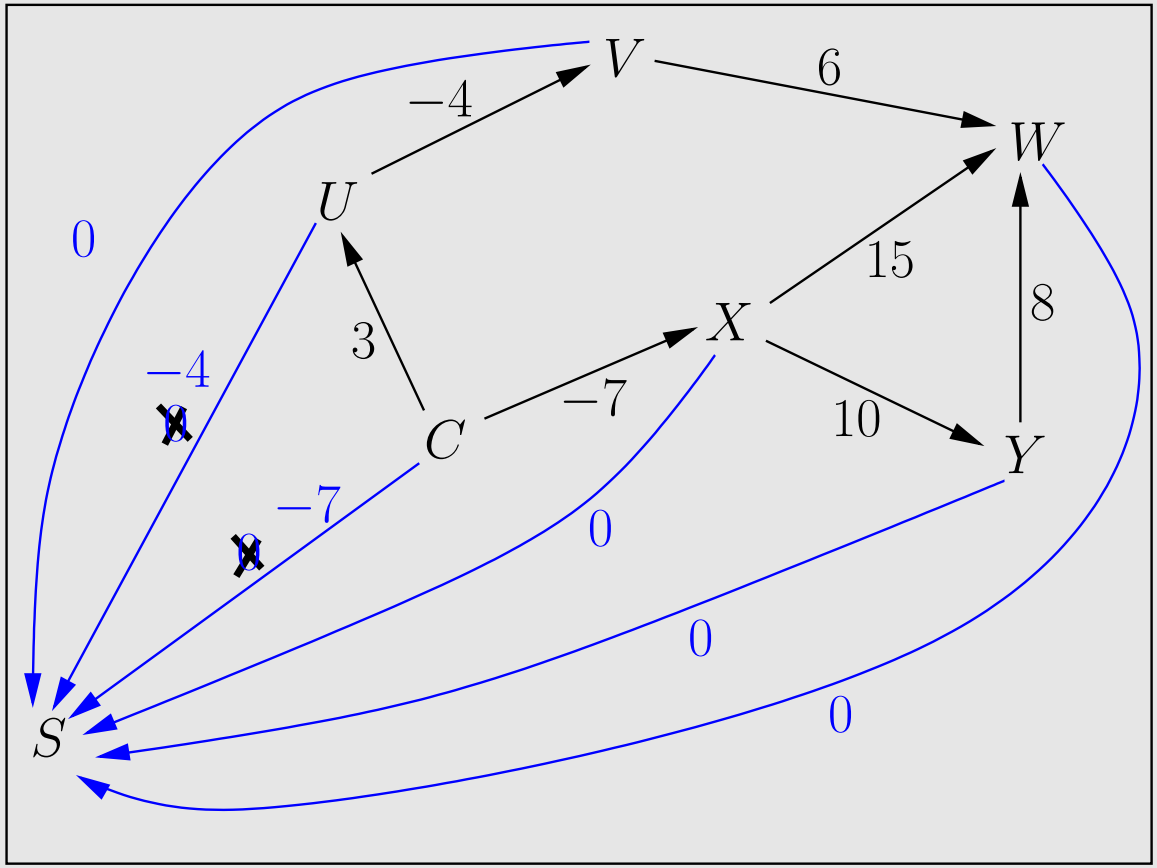
Goal: Traverse paths from C

Enabling Dijkstra



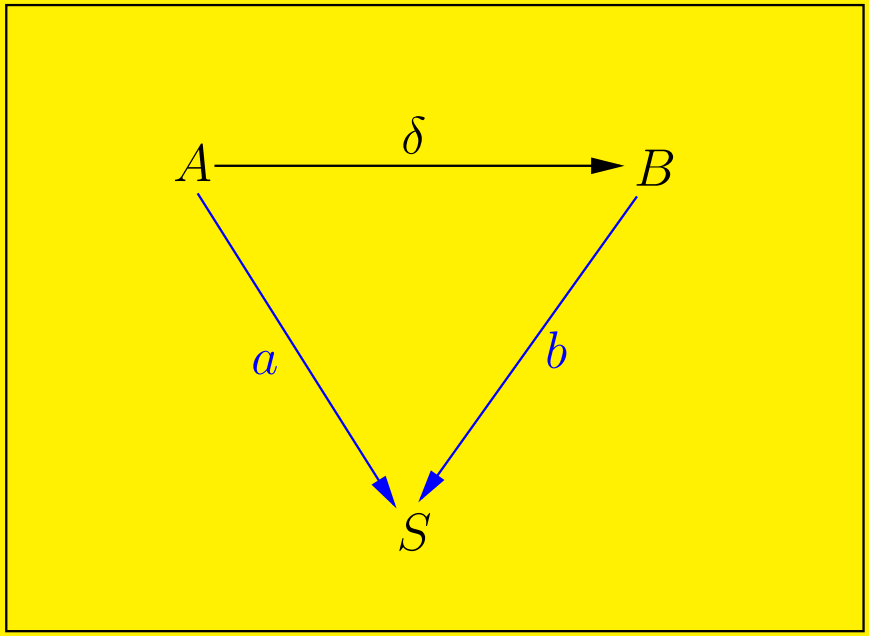
Add edges of length 0 to sink node S

Enabling Dijkstra



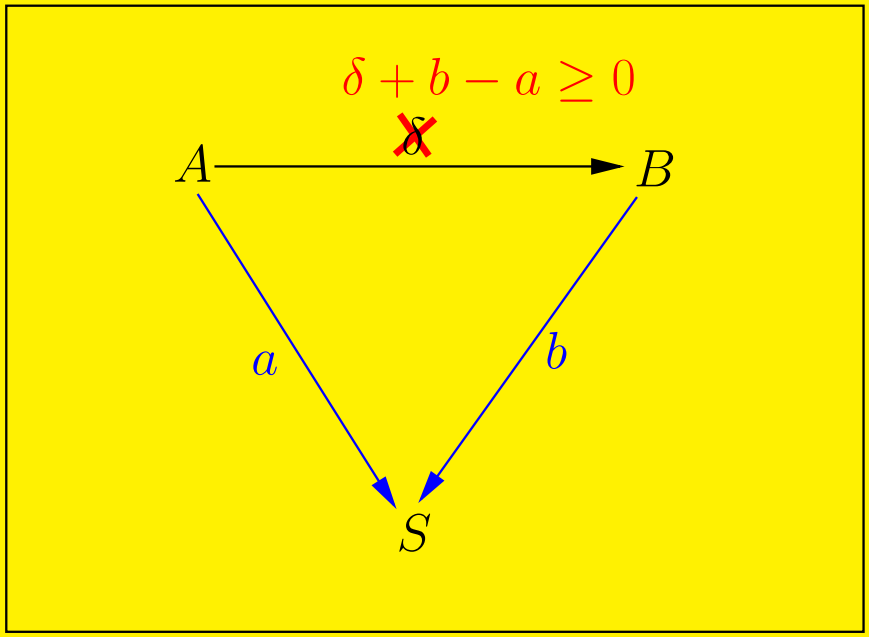
BellmanFord: compute shortest paths to S

Convert Edges to Non-Neg. Lengths



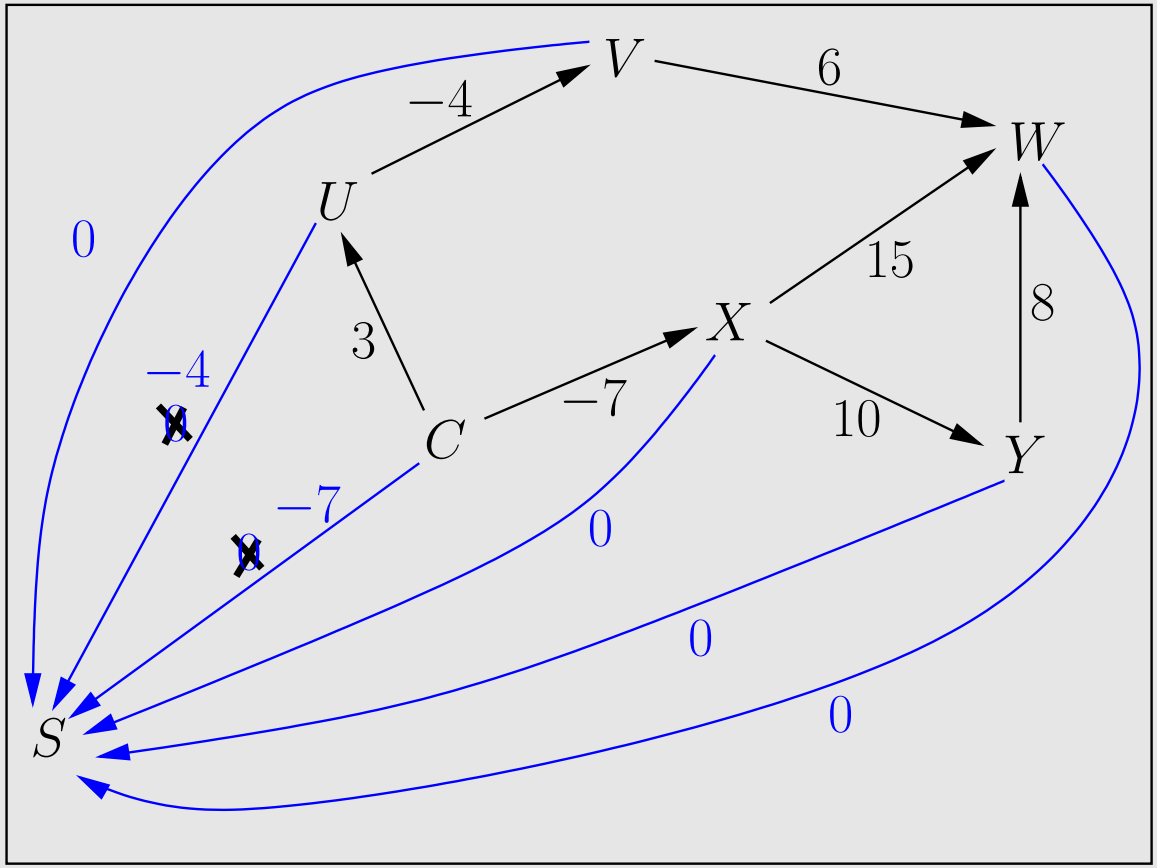
Given: shortest paths from A and B to S

Convert Edges to Non-Neg. Lengths



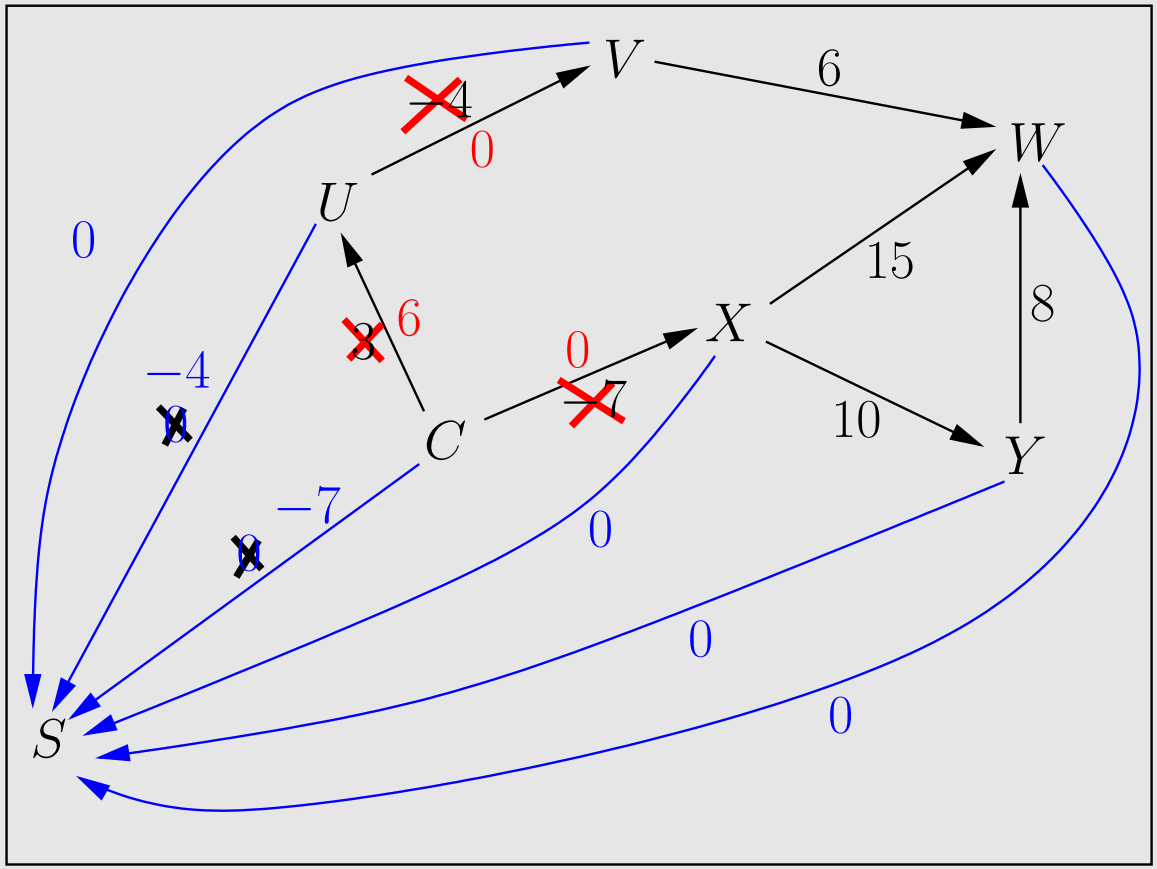
Note: $\delta + b \geq a$, since *dist*$(A, S) = a$

Enabling Dijkstra



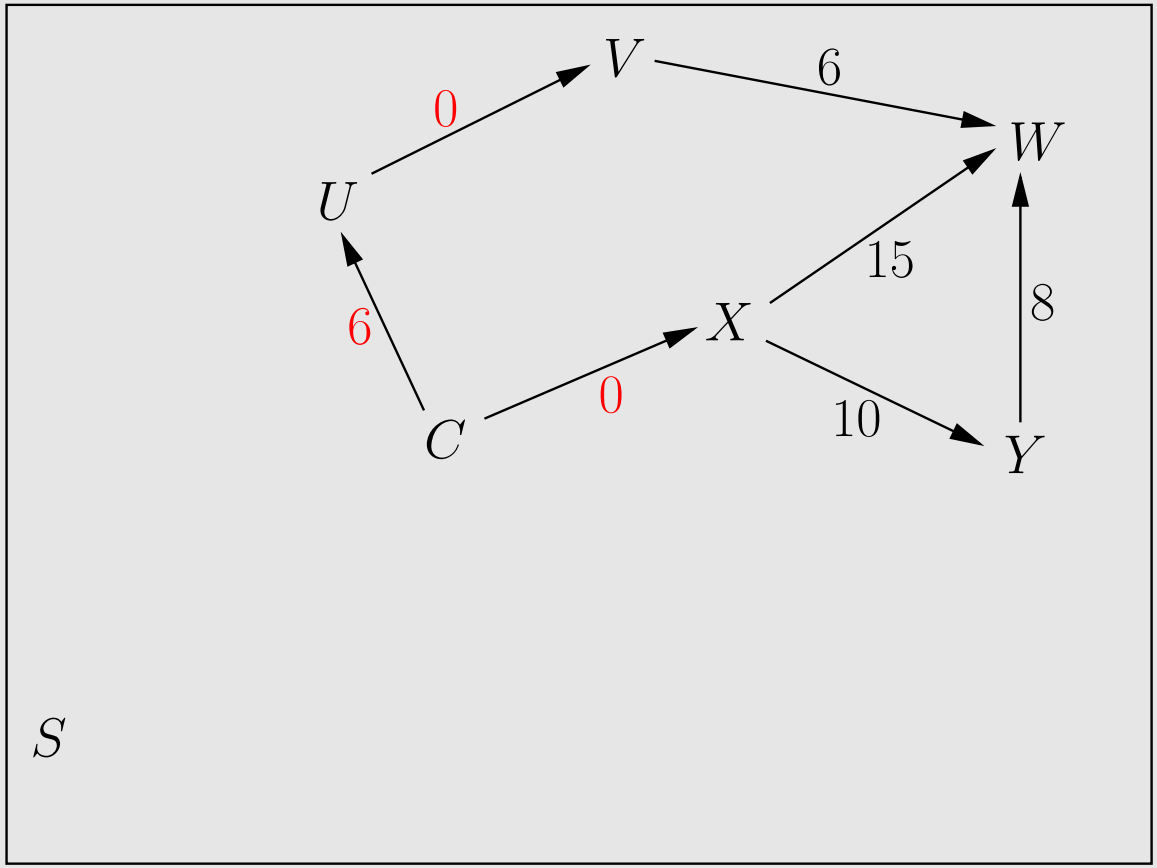
Compute shortest paths to S

Enabling Dijkstra



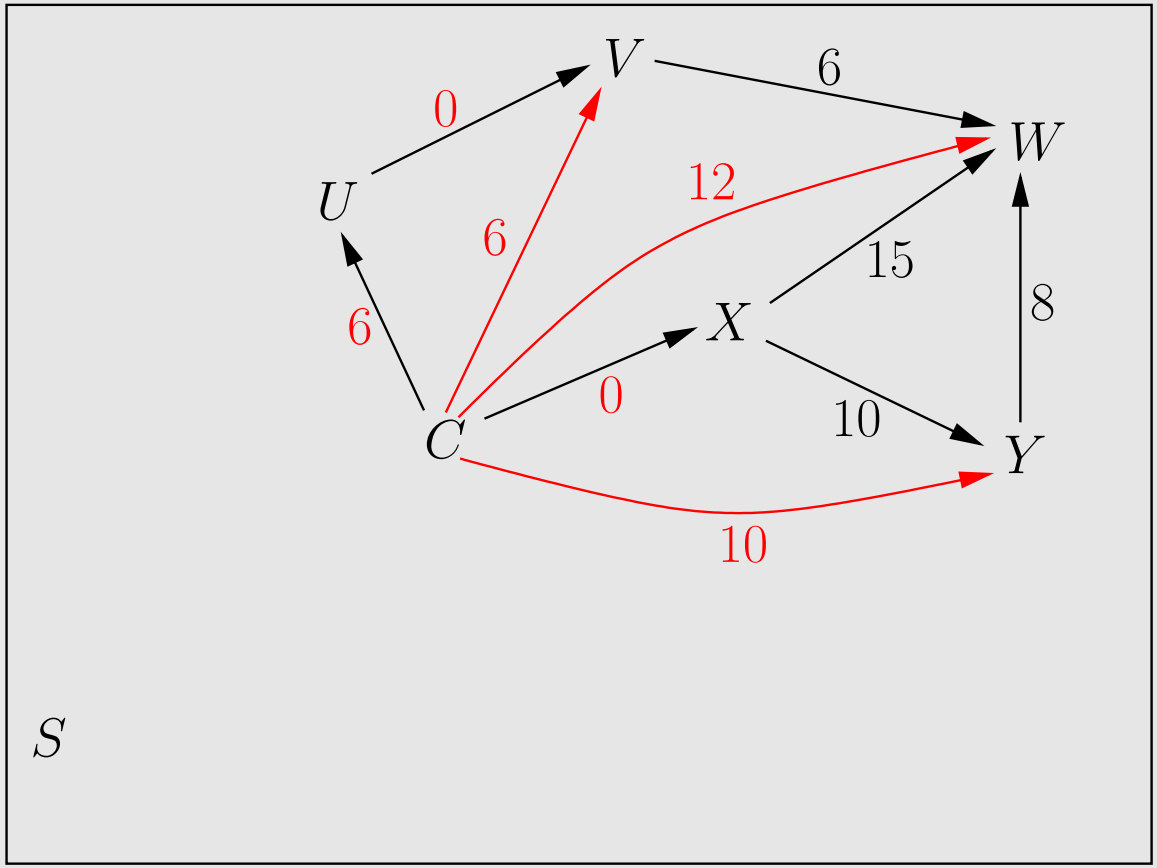
Convert orig'l. edges to non-neg. lengths

Enabling Dijkstra



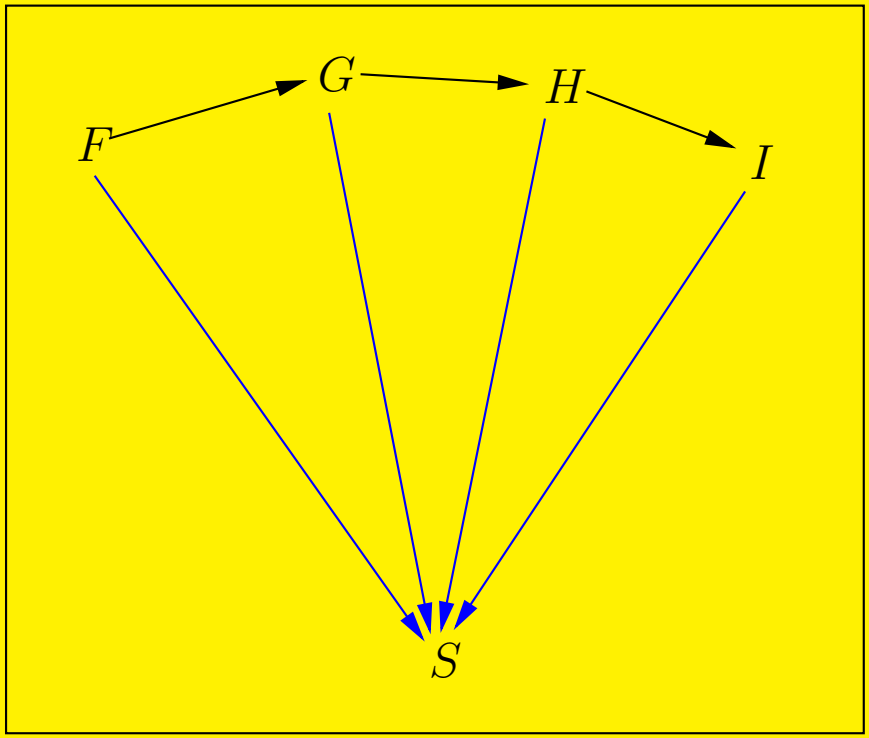
Use Dijkstra with C as source...

Enabling Dijkstra

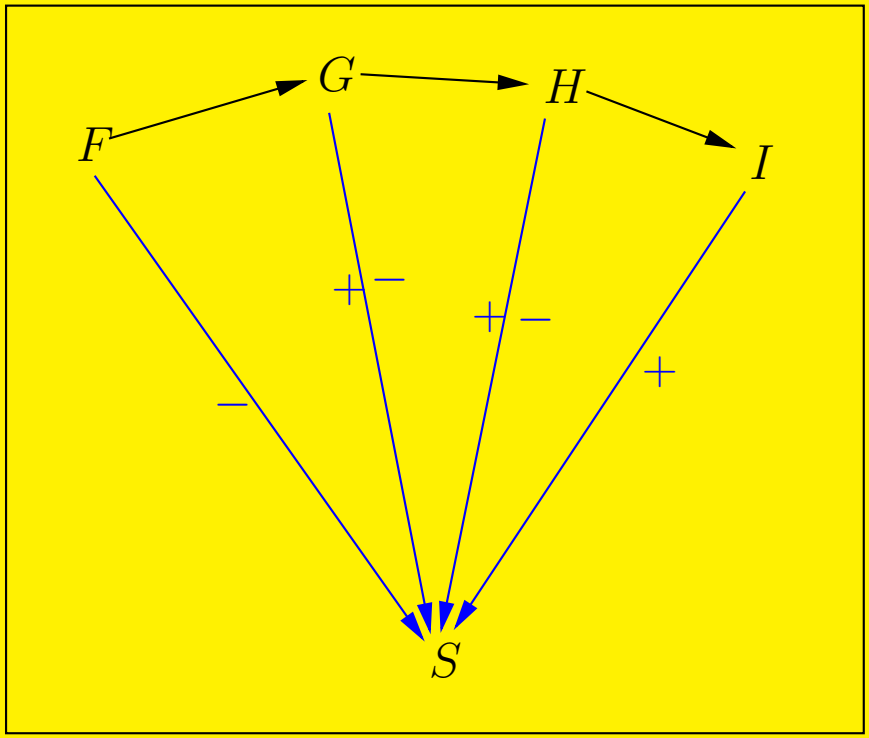


... to compute shortest paths from C

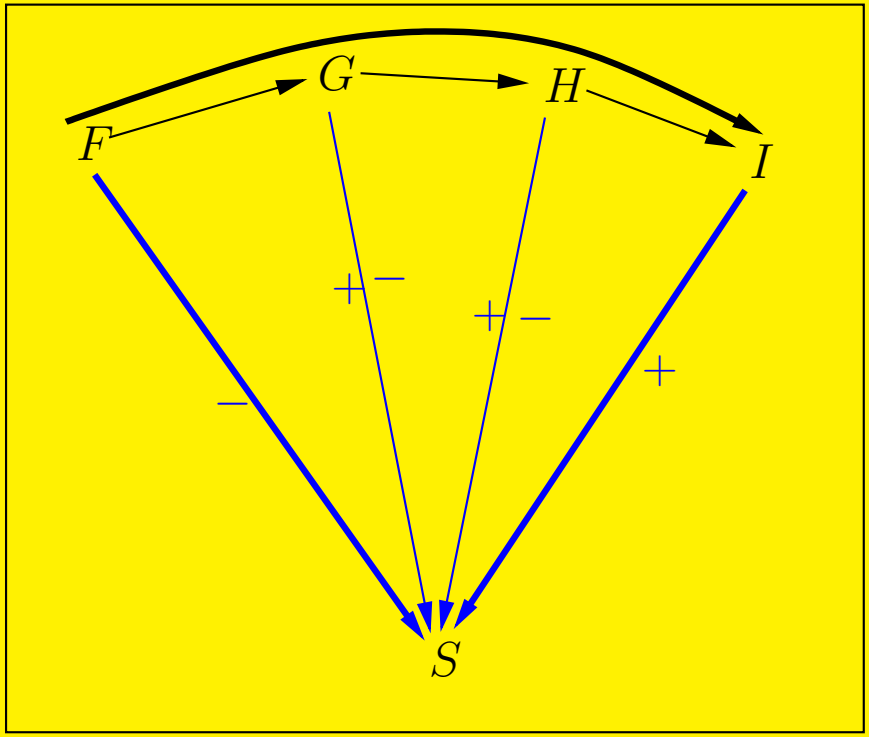
Convert Back



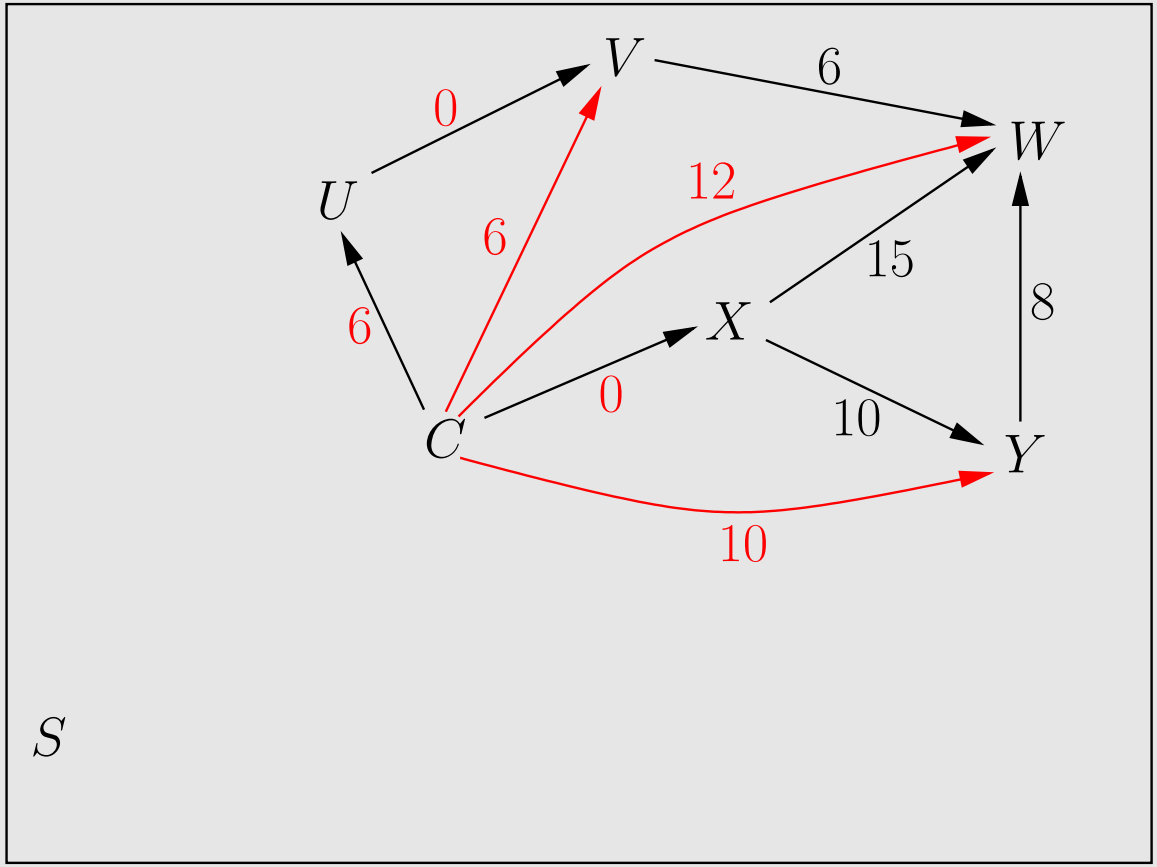
Convert Back



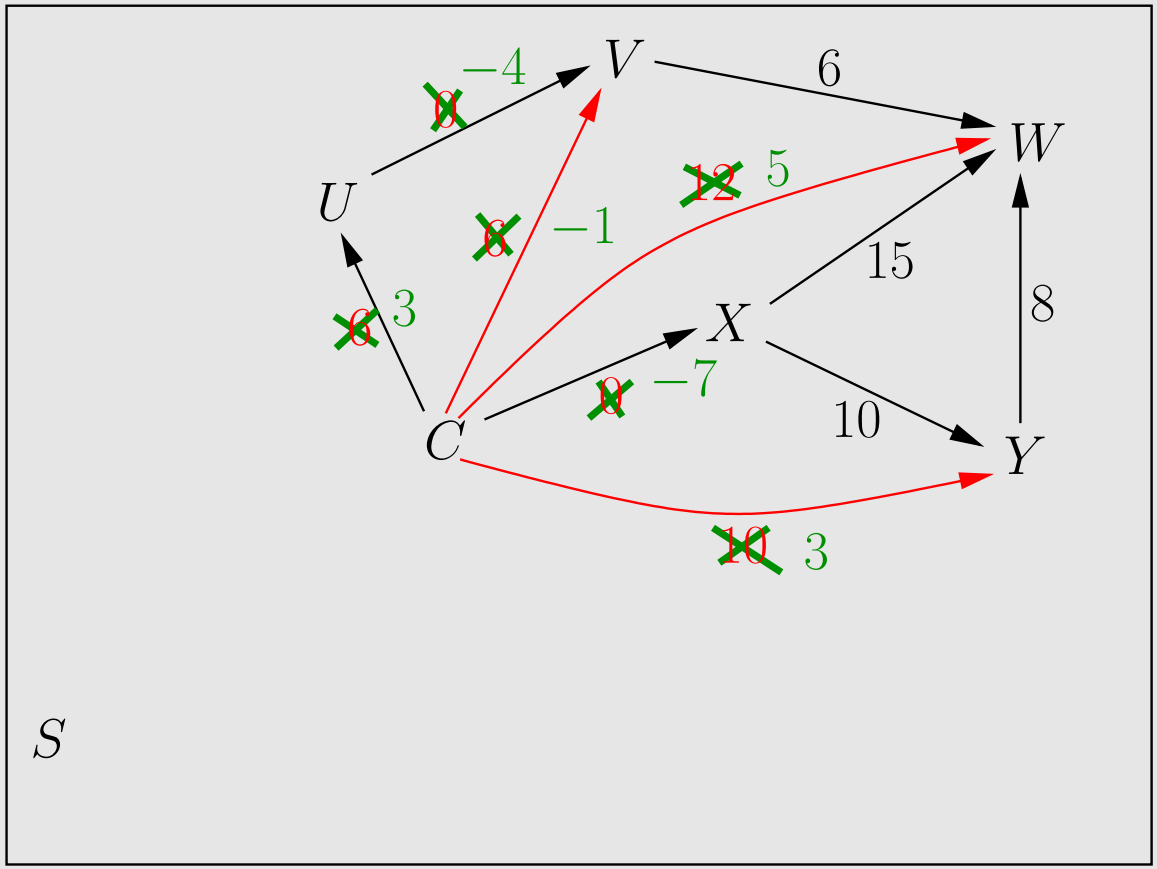
Convert Back



Enabling Dijkstra

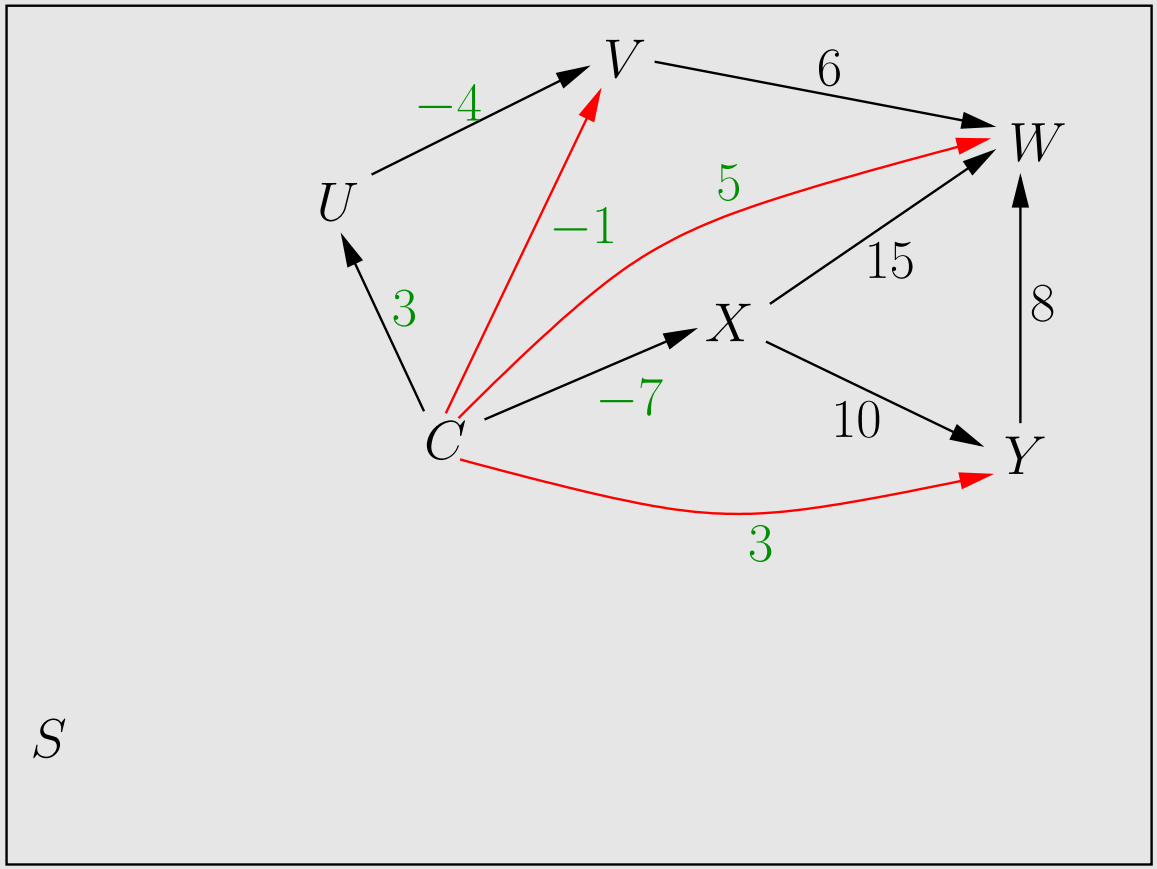


Enabling Dijkstra



Convert shortest path lengths to “normal”

Enabling Dijkstra



Convert shortest path lengths to “normal”

Morris' DC-Checking Algorihtm

INPUT: G, a graph for an STNU with K contingent links.

```
-1.  Gou := OU-graph for G.
 0.  Gx  := AllMax graph for G.
 1.  for i = 1, K:
 2.     result := BellmanFordSSSP(Gx).                    Outer Loop
 3.     if (result == inconsistent) return False.
 4.     else generatePotentialFunction(result).
 5.     newEdges := .
 6.     for j = 1, K:
 7.        Let C_j be the j^th contingent time-point.     Inner Loop
 8.        Traverse shortest allowable paths in G_ou emanating from C_j, searching for
           extension sub-paths that generate new edges. Add new edges to newEdges.
 9.     end for j = 1, K.
10.     if newEdges empty, return True.
11.     else insert newEdges into Gou and Gx.
12.  end for i = 1, K.
13.  result := BellmanFordSSSP(Gx).
14.  if (result == inconsistent) return False.
15.  else return True.
```

New DC-Checking Algorithm

Reflections on Morris' Algorithm

During each outer iteration, *one* potential function is used for K inner iterations (i.e., K traversals):

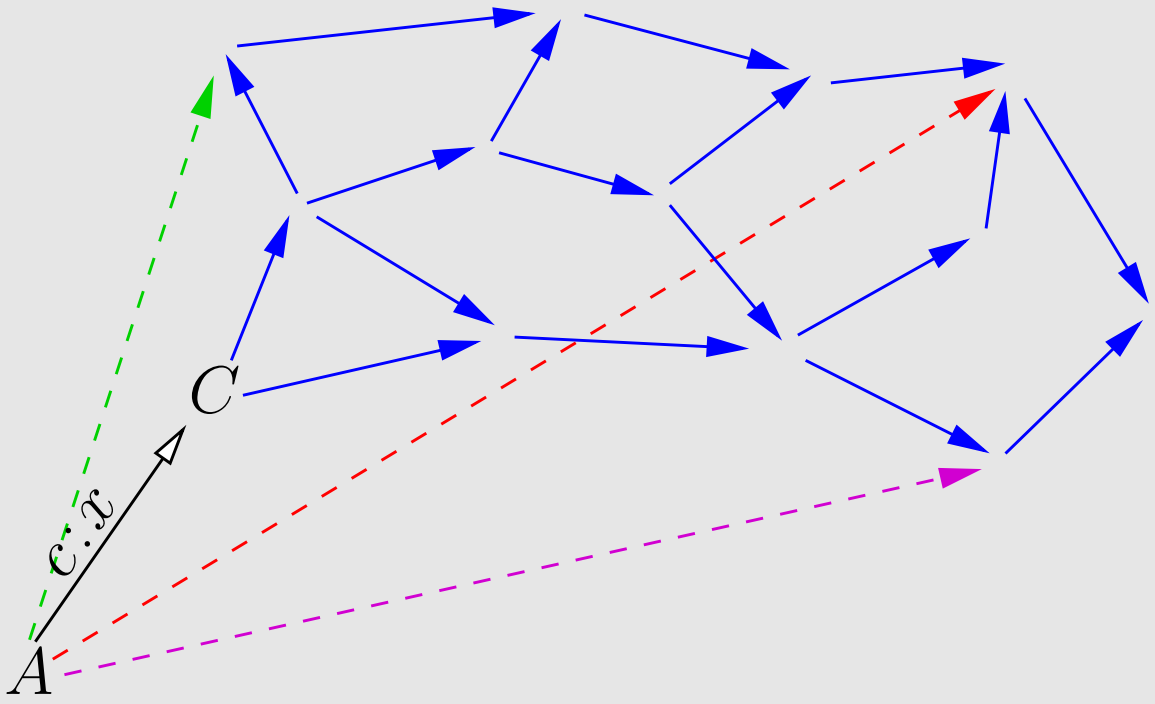
- Thus, new edges *cannot* be added to the network until *after* all K traversals are completed.
- Thus, the order in which the K traversals are done *cannot* matter.

Morris' DC-Checking Algorihtm

INPUT: G, a graph for an STNU with K contingent links.

```
-1.  Gou := OU-graph for G.
 0.  Gx  := AllMax graph for G.
 1.  for i = 1, K:
 2.     result := BellmanFordSSSP(Gx).                    Outer Loop
 3.     if (result == inconsistent) return False.
 4.     else generatePotentialFunction(result).
 5.     newEdges := .
 6.     for j = 1, K:
 7.        Let C_j be the j^th contingent time-point.     Inner Loop
 8.        Traverse shortest allowable paths in G_ou emanating from C_j, searching for
           extension sub-paths that generate new edges. Add new edges to newEdges.
 9.     end for j = 1, K.
10.     if newEdges empty, return True.
11.     else insert newEdges into Gou and Gx.
12.  end for i = 1, K.
13.  result := BellmanFordSSSP(Gx).
14.  if (result == inconsistent) return False.
15.  else return True.
```

Recall Central Process

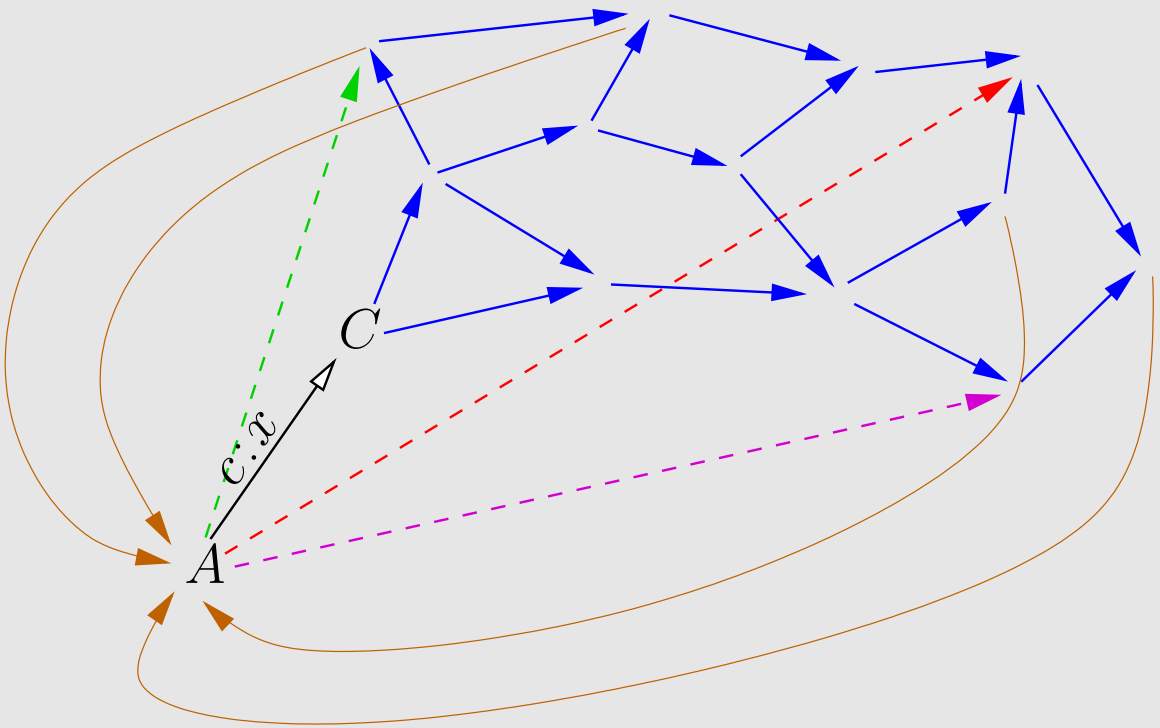


Key feature: New edges have A as their *source.*

New Approach

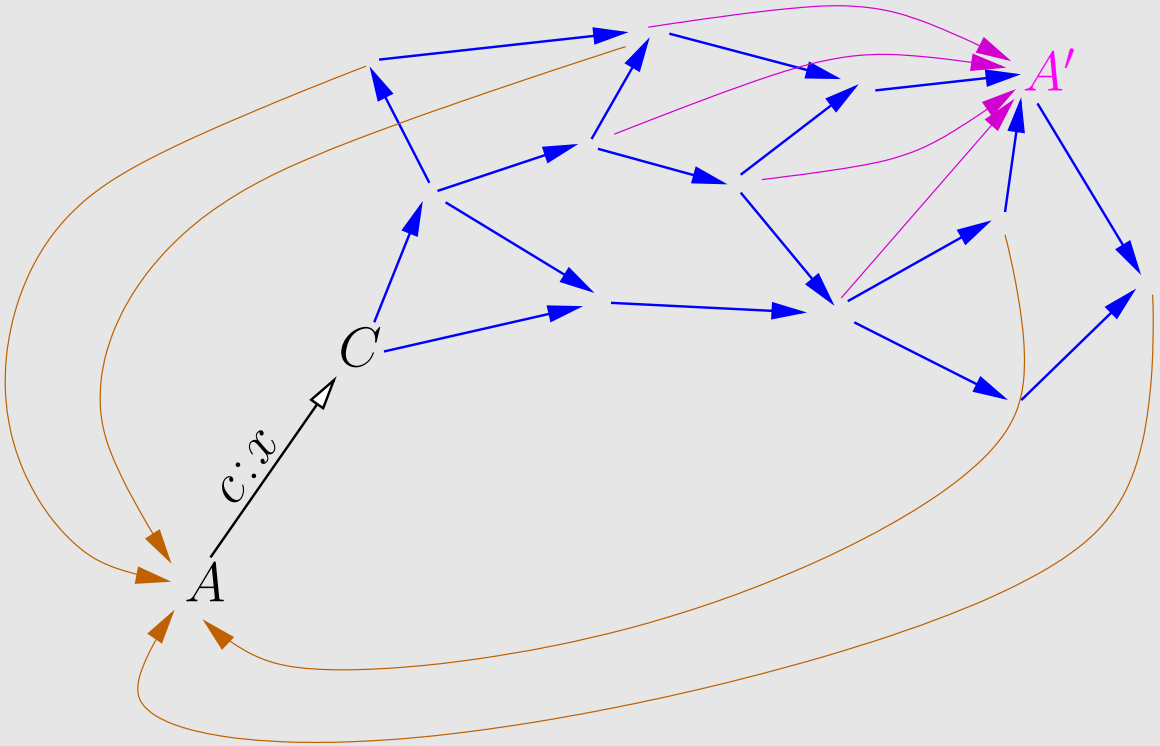
- Observation: When processing $A \xrightarrow{c:x} C$, new edges have A as their *source.*
- Observation: Such edges cannot change shortest paths having A as their *sink.*
- KEY IDEA: Use potential function based on shortest-paths with A as sink.
- KEY IDEA: Use Dijkstra to update potential function after each traversal!

Updating Potential Function



New edges cannot affect potential function

Updating Potential Function



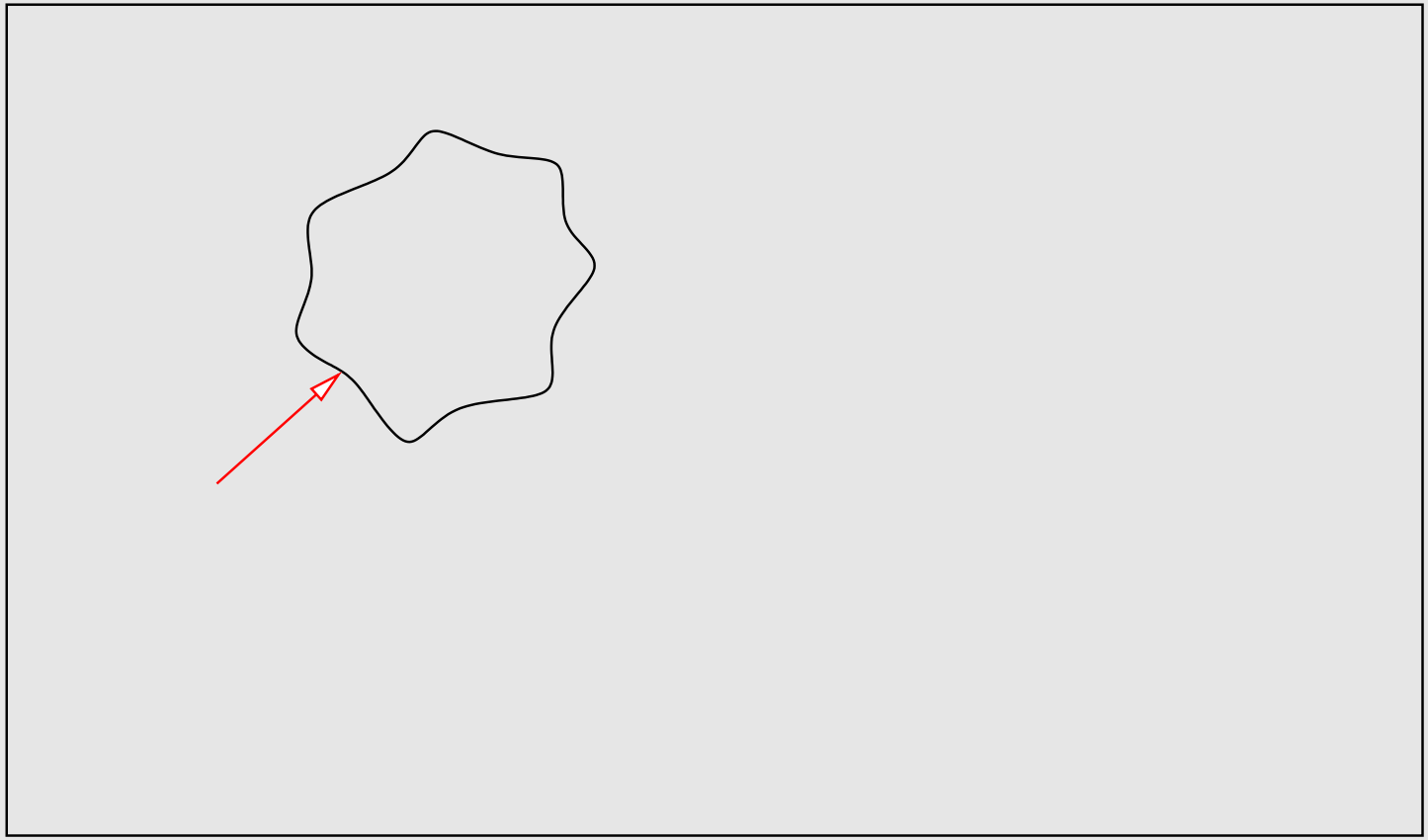
Use current potential function (sink $= A$) to generate new potential function (sink $= A'$)

New Approach

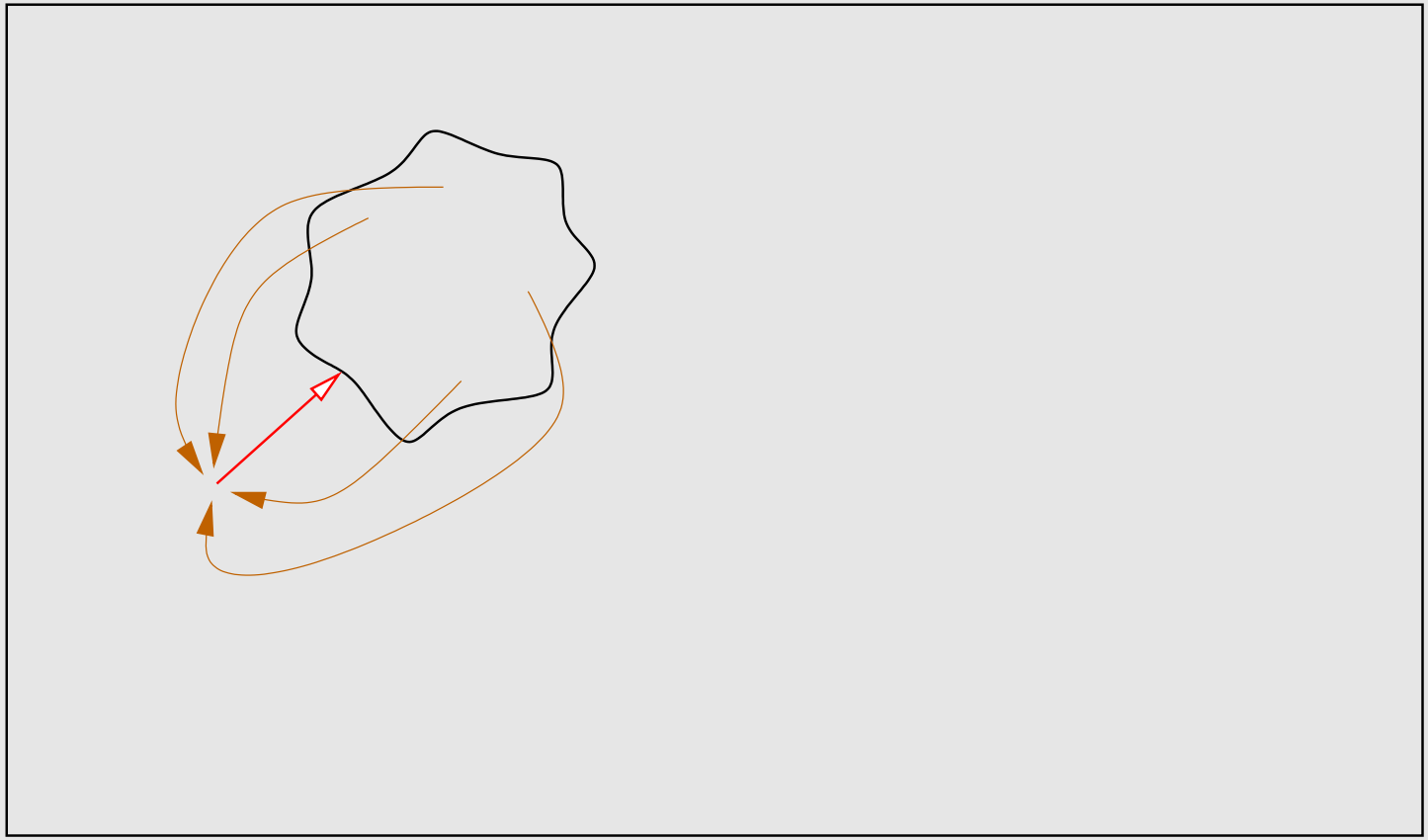
- Enables newly generated edges to be inserted into the network *immediately*
- Thus, the *order* in which lower-case edges are processed *matters!*
- Use heuristic to determine “good” order.

(Heuristic details are in the paper.)

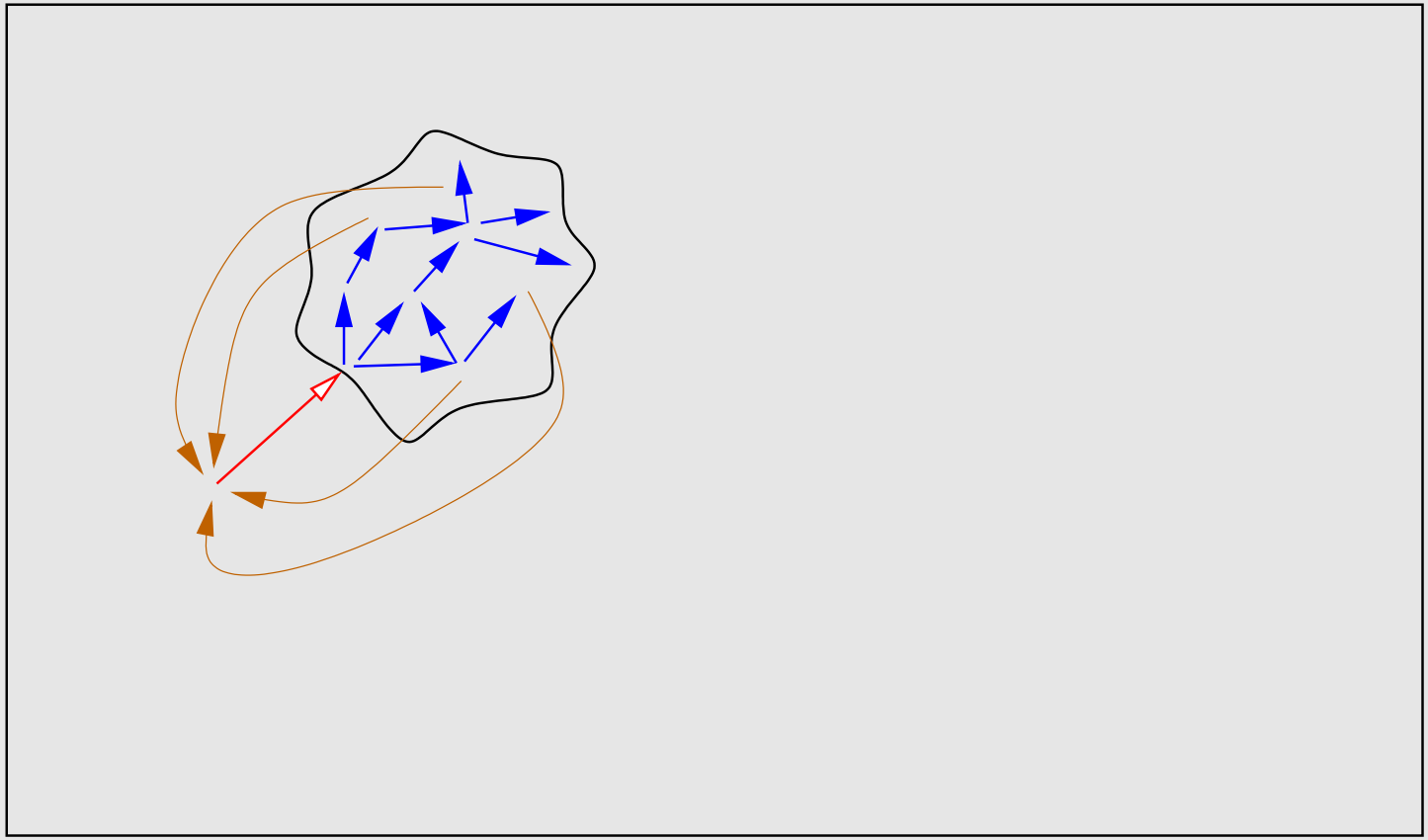
Rotating Dijkstra



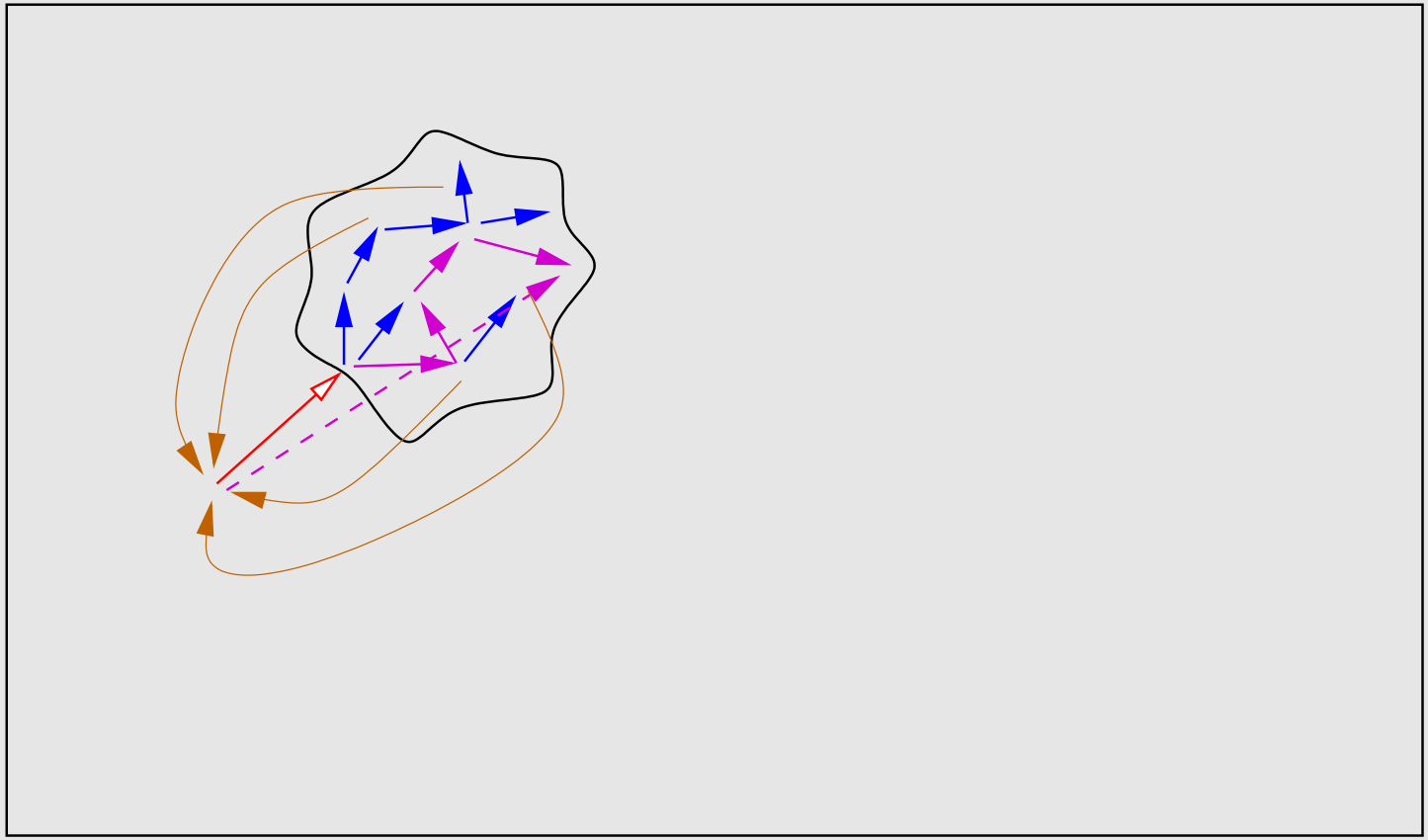
Rotating Dijkstra



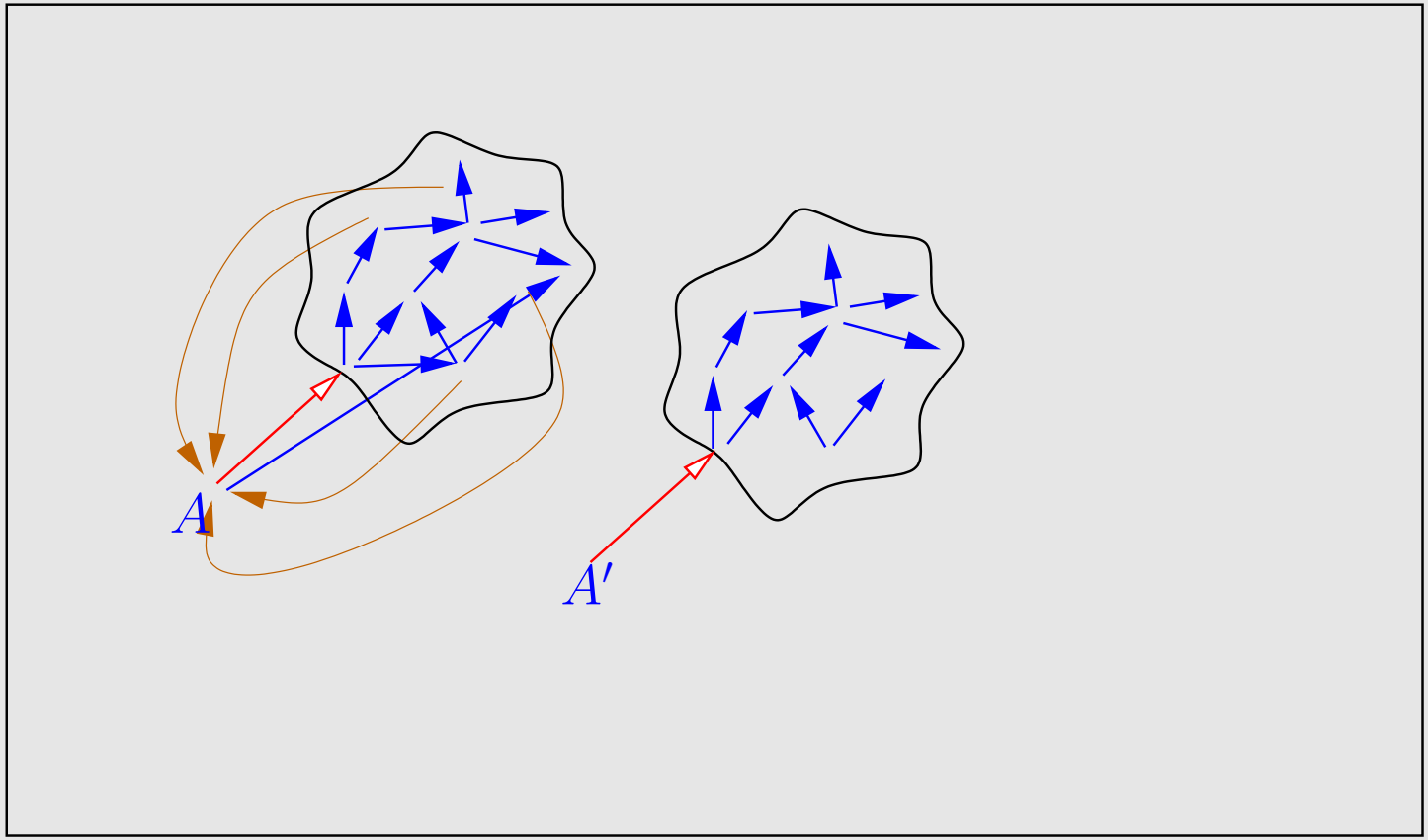
Rotating Dijkstra



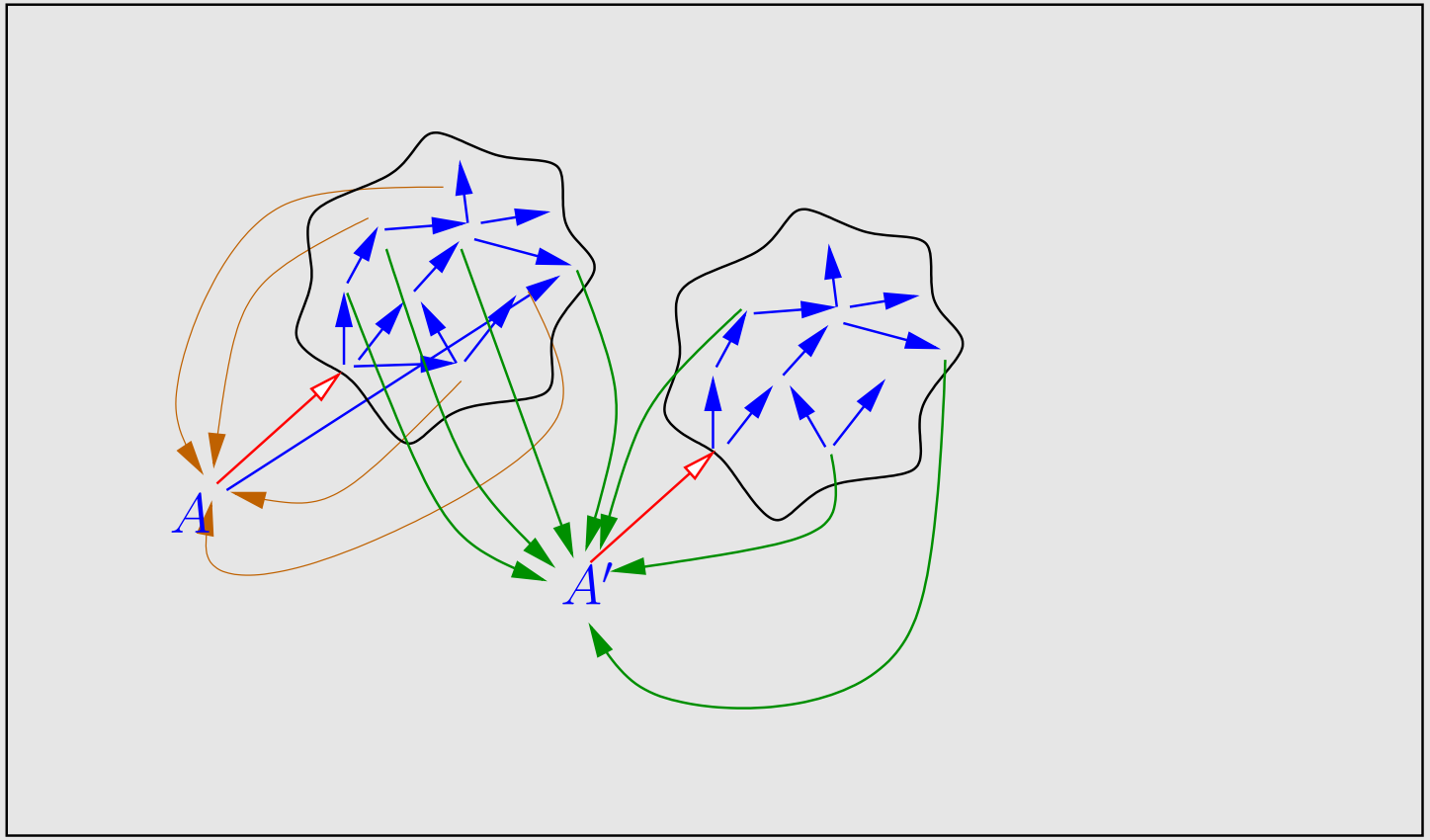
Rotating Dijkstra



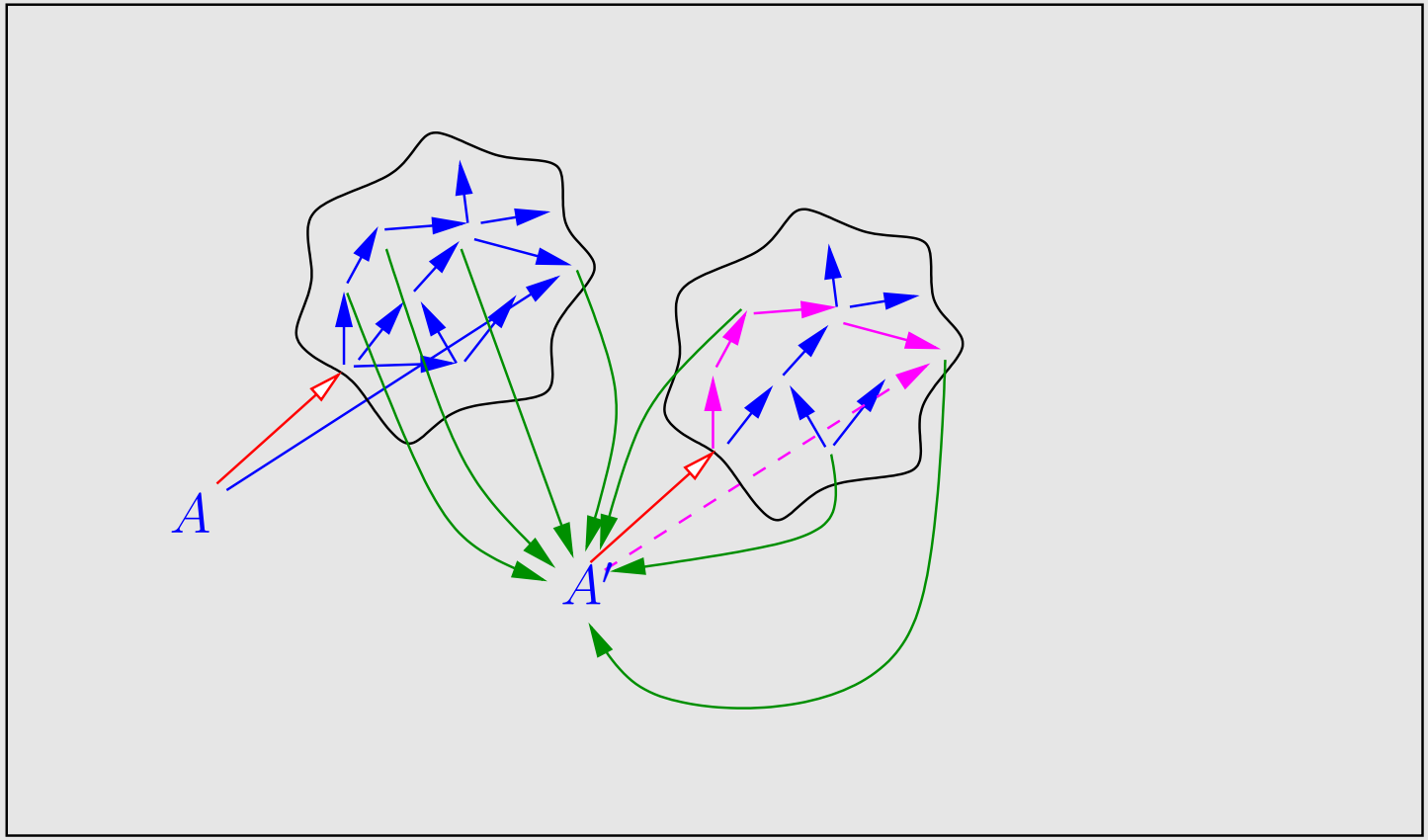
Rotating Dijkstra



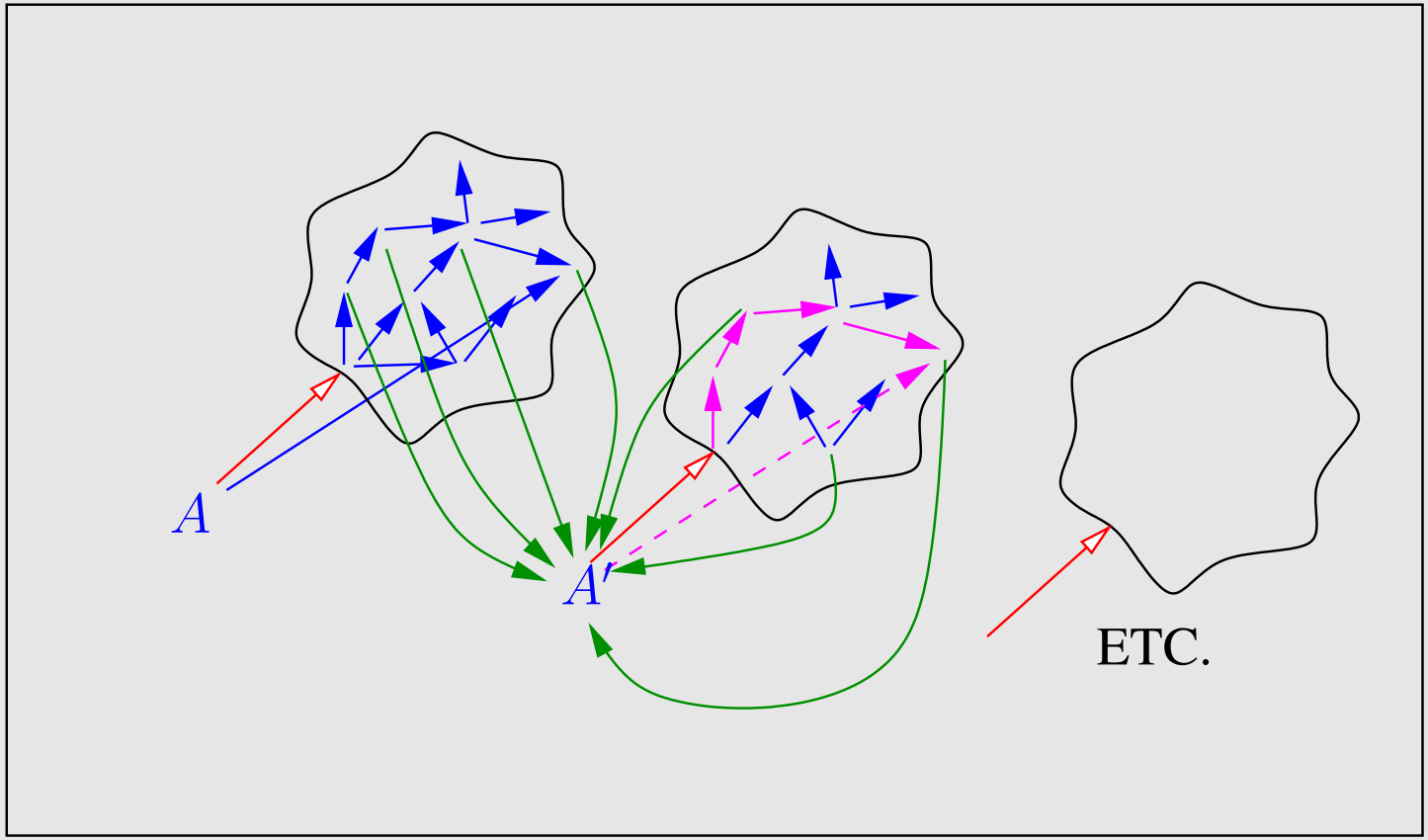
Rotating Dijkstra



Rotating Dijkstra



Rotating Dijkstra



New DC-Checking Algorihtm

```
     Gou := OU-graph for G;
0.   Gx  := AllMax graph for G;
1.   Dx  := Johnson(Gx);
2.   Order := Heuristic(Dx);
3.   GlobalIters := 0;
4.   LocalIters := 0;
5.   for i = 1, K:
6.     for j = 1, K:
7.       newEdges := {};
8.       Let (A_j, x_j, y_j, C_j) be the j^th contingent link according to Order.
9.       Use D_x(T, A_j) as potential function to transform edge-lengths in Gx and Gou.
10.      Traverse shortest allowable paths in G_ou emanating from C_j,
             searching for extension sub-paths. Add any new edges to newEdges.
11.      for each edge, A_j --δ--> X in newEdges:  if (δ < -D_x(X, A_j)) return False;
12.      GlobalIters++;
13.      if (GlobalIters ≥ K^2) return True;
14.      elseif newEdges empty:
15.        LocalIters++;
16.        if (LocalIters ≥ K) return True;
17.      else LocalIters := 0;
18.      run sinkDijkSinkPot(A',Aj), where A' is activation tp for next cont. link.
19.    end for j = 1, K.
20.  end for i = 1, K.
21.  return True.
```

Evaluation & Conclusions

Evaluation on *Magic Loops*

- Magic loops are one kind of worst case for DC-checking algorithms (Hunsberger 2013)
- Maximum nesting of extension sub-paths
- $2^K - 1$ occurrences of LC edges!
- Heuristic exploits *nesting order*
- New algorithm is an order of magnitude faster.

Evaluation on Random Networks

Test	N	K	E	D	RT ratio	It ratio
T0	58	18	295 ± 146	6.7 ± 0.6	$1.29 \pm .07$	$1.82 \pm .04$
T1	139	45	446 ± 78	11.5 ± 1.2	$1.47 \pm .03$	$2.12 \pm .05$
T2	139	45	321 ± 27	12.8 ± 1.9	$1.58 \pm .03$	$2.33 \pm .06$
T3	184	60	713 ± 219	14.7 ± 2.0	$1.63 \pm .04$	$2.36 \pm .08$
T4	229	75	569 ± 115	17.6 ± 3.5	$1.80 \pm .04$	$2.51 \pm .07$

N = number of time-points in network
K = number of contingent links
E = avg. number of edges
D = avg. max depth of nesting
RT ratio = avg. (MorrisTime/NewAlgTime)
It ratio = avg. (NumMorrisIters/NumNewAlgIters)

Conclusions

- New DC-checking algorithm outperforms state-of-the-art algorithm
- Uses novel *rotating Dijkstra* technique
- Uses heuristic to estimate/exploit nesting order

Future Work

- Exhaustive empirical evaluation
- Compare against incremental algorithms
- Explore further opportunities to exploit graphical structure
- Open problem: DC checking $\in O(N^3)$?