Recent Advances in Temporal Networks for Planning and Scheduling

Luke Hunsberger
Vassar College
Poughkeepsie, NY USA
hunsberger@vassar.edu

ICAPS–2023 Tutorial
July 9, 2023

Acknowledgments

- Parts of this tutorial draw from an Invited Talk at the TIME-2021 Symposium on Temporal Representation and Reasoning by Luke Hunsberger (Vassar College) and Roberto Posenato (University of Verona).

- This tutorial was supported in part by NSF Award # 1909739:
  RI: Small: RUI: Automated Reasoning about Time
  – Methods and Analysis.

Outline I

Outline II

Simple Temporal Networks (STNs)

Introduction to STNs

Simple Temporal Networks

Overview

- Temporal networks are data structures for representing and reasoning about temporal constraints on activities.
- A Simple Temporal Network (STN) is the most basic kind of temporal network:
  1. An STN can accommodate such constraints as release times, deadlines, precedence constraints, and duration constraints.
     [Dechter et al., 1991]
  2. The fundamental computational tasks associated with STNs—including *checking consistency and managing execution*—can be done in polynomial time.
     [Dechter et al., 1991; Tsamardinos et al., 1998]
- STNs form the core of numerous more expressive kinds of temporal networks.

Simple Temporal Networks

Research and Applications

- STNs are used as a temporal reasoning tool for research and real-world applications.
- In September 2021, a search of the literature for *Simple Temporal Networks*[*] found:
  - > 1700 research articles in Google Scholar (having the subject in any part of the article);
  - 242 research articles in Scopus (having the subject in title or abstract)
- The most cited papers on STNs fall mainly within two areas:
  - *planning/scheduling for robots*
  - *industrial, business, and health-care management systems*

---

[*] The query string was `"simple temporal constraint network" OR "simple temporal network" OR "simple temporal problem" OR "simple temporal constraint problem" OR "simple temporal constraint networks" OR "simple temporal networks"`

Simple Temporal Networks

Applications in *planning/scheduling for robots*

Summary of articles with more than 150 citations in Google Scholar.

- Remote Agent (RA): on-board controller for Deep Space One, NASA's first New Millennium mission [Muscettola et al., 1998a]
    - Needs flexible plans, runs multiple parallel threads of planning and scheduling, uses fast constraint propagation algorithms.
    - Fast constraint propagation obtained using STNs and their local dispatchable property.
- Planners such as RAX-PS [Jonsson et al., 2000], MAPGEN [Ai-Chang et al., 2004; Bresina et al., 2005], KIRK [Kim et al., 2001], VHPOP [Younes and Simmons, 2003], EUROPA-2 [Frank and Jónsson, 2003], CRIKEY3 [Coles et al., 2008], OPTIC [Benton et al., 2012], IXTET-EXE [Lemai and Ingrand, 2004],...
- Control component for Autonomous Underwater Vehicles [McGann et al., 2008]

Simple Temporal Networks

Mars Exploration Rover (MER) [Bresina et al., 2005]

- In constraint–based planning, actions and states are described as holding over intervals of time.
- The temporal extent of an action or state is specified in terms of start and end times, represented by variables, connected by constraints.
- Typically, any partial plan, which is a set of activities connected by constraints, gives rise to a Simple Temporal Network, that admits a low–order checking algorithm.

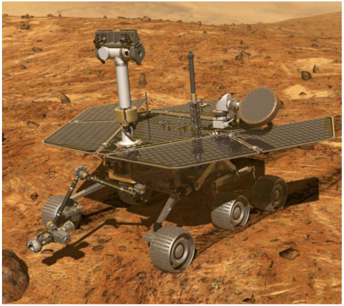
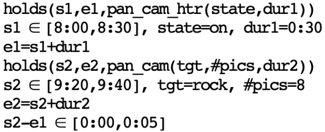


Figure 1: MER Rover



```
holds(s1,e1,pan_cam_htr(state,dur1))
s1 ∈ [8:00,8:30], state=on, dur1=0:30
e1=s1+dur1
holds(s2,e2,pan_cam(tgt,#pics,dur2))
s2 ∈ [9:20,9:40], tgt=rock, #pics=8
e2=s2+dur2
s2–e1 ∈ [0:00,0:05]
```

Typical plan with temporal constraints.

* Figures are from [Bresina et al., 2005]

Simple Temporal Networks

Applications in *industrial, business and health-care management systems*

Summary of articles with more than 60 citations in Google Scholar.

- Temporal reasoning in workflows [Bettini et al., 2002; Cesta et al., 2011; Combi and Posenato, 2009]
- Temporal reasoning in health-care systems [Anselma et al., 2006; Combi et al., 2009; Duftschmid et al., 2002; Zhou et al., 2006]
- Scheduling (in industrial processes) [Cesta et al., 2002; Ruml et al., 2005; Smith et al., 2007; Yoon and Lee, 2005]
- Chronicles on-line recognition [Ghallab, 1996]
- Dial-A-Ride Problem with Transfers [Masson et al., 2014]
- Image pose reconstruction [Dabral et al., 2018]
- IBM ILOG CP optimizer for scheduling [Laborie et al., 2018]
- ...

Simple Temporal Networks

The Dial–A–Ride Problem with Transfers [Masson et al., 2014] 1/2

- The Dial–A–Ride Problem with Transfers (DARPT): find set of minimum cost routes to satisfy a set of transportation requests.
    - Request = transporting a set of users from a set of pickup points to a set of delivery points.
    - Users associated with distinct requests can share a vehicle if its capacity not exceeded.
    - Max. ride time associated with each request.
    - Users can be transferred from one vehicle to another at intermediate points.
    - Goal: Minimize total distance traveled while staying within maximum ride time for each user.
- DARPT is NP–hard.

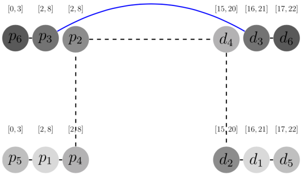


Fig. 1. A solution without transfer (DARP).

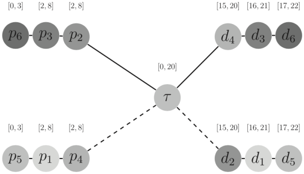


Fig. 2. A solution with transfer (DARPT).

p_i must go to d_i. 1 vehicle. Temporal ranges allowed for pickup/delivery. DARPT solution is 20% more efficient.

* Figures are from [Masson et al., 2014]

Simple Temporal Networks

The Dial–A–Ride Problem with Transfers [Masson et al., 2014] 2/2

- Solution algorithm based on Adaptive Large Neighborhood Search (ALNS)
  - It destroys and repairs a solution iteratively to improve it.
  - Heuristic operators for either destroying (removing requests from routes) or repairing (reinserting requests) the solution.
  - Each possible solution must satisfy all constraints (feasibility).
  - Evaluating route feasibility for the DARPT ⇔ proving the consistency of an STN.
- STNs have been employed because it is necessary to check a large number of possible solutions—which can be done very efficiently with STNs.

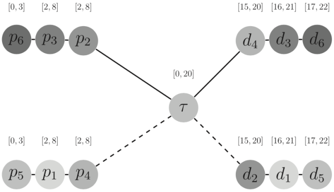


Fig. 2. A solution with transfer (DARPT).

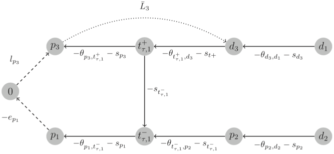


Fig. 5. A STP distance graph.

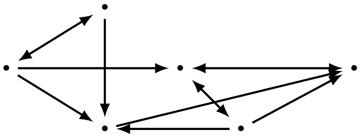
p_i must go to d_i. 1 vehicle. Ranges are allowed temporal range for pickup/delivery.

* Figures are from [Masson et al., 2014]

Simple Temporal Networks

Features and Benefits

- An STN has time–points and (simple) temporal constraints.
- STNs are expressive: can represent deadlines, release times, duration constraints, and inter–action constraints.
- STNs are flexible: Time–points can "float"; not "nailed down" until they are *executed*.
- Each STN has a graphical representation:



- Efficient algorithms exist for determining consistency, managing real–time execution, accommodating new constraints, etc.

STN Foundations

Simple Temporal Network

Definition [Dechter et al., 1991]

Definition 1 (Simple Temporal Network)

A *Simple Temporal Network (STN)* is a pair, $\mathcal{S} = (\mathcal{T}, \mathcal{C})$, where:

- $\mathcal{T}$ is a set of real–valued variables called *time–points*; and
- $\mathcal{C}$ is a set of binary constraints, each of the form:

$$Y - X \leq \delta$$

where $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$.

Simple Temporal Network

The *Zero* Time–Point, Z

- A special time–point, Z, whose value is fixed at `0`.
- Binary constraints involving Z are equivalent to unary constraints.

Example 1

$$X - Z \leq 7 \quad \Longleftrightarrow \quad X \leq 7$$

$$Z - X \leq -3 \quad \Longleftrightarrow \quad X \geq 3$$

Simple Temporal Network

Solutions, Consistency, Simple Temporal Problem

- A *solution* to an STN $\mathcal{S} = (\mathcal{T}, \mathcal{C})$ is a complete set of assignments to the time–points in $\mathcal{T}$:

  $$\{X_1 = w_1,\ X_2 = w_2,\ \ldots,\ X_n = w_n\}$$

  that together satisfy all of the constraints in $\mathcal{C}$.
- An STN with at least one solution is *consistent.*
- The problem of determining whether an STN is consistent is called the *Simple Temporal Problem* (STP).

Simple Temporal Network

STN for Travel Example

$$\mathcal{T} = \{Z, X_1, X_2, X_3, X_4\},\ \text{where}\ \ Z = \text{Noon, June 8}$$

$$\mathcal{C} = \left\{ \begin{array}{rcrl} Z - X_1 & \leq & -4 & \text{(Leave NYC after 4 p.m., June 8)} \\ X_4 - Z & \leq & 250 & \text{(Return NYC by 10 p.m., June 18)} \\ X_4 - X_1 & \leq & 168 & \text{(Gone no more than 7 days)} \\ X_2 - X_3 & \leq & -120 & \text{(In Rome at least 5 days)} \\ X_4 - X_3 & \leq & 7 & \text{(Return flight less than 7 hrs)} \end{array} \right.$$

Simple Temporal Network

Graph for an STN [Dechter et al., 1991]

The *graph* for an STN, $\mathcal{S} = (\mathcal{T}, \mathcal{C})$, is a graph, $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, where:

Time–points in $\mathcal{S}$ $\iff$ nodes in $\mathcal{G}$

Constraints in $\mathcal{C}$ $\iff$ edges in $\mathcal{E}$:

$$Y - X \leq \delta \qquad\qquad X \xrightarrow{\delta} Y$$

Simple Temporal Network

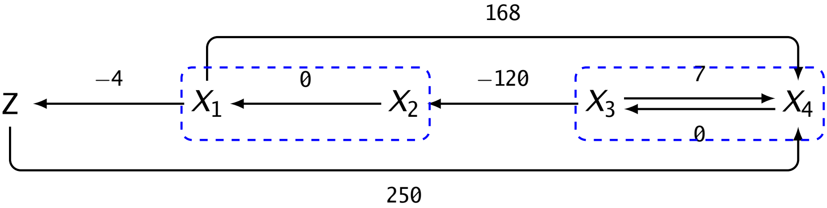
Graphical Representation

Constraint(s)	Edge(s)	Interval Notation
$Y - X \leq 7$	$X \xrightarrow{7} Y$	$X \xrightarrow{(-\infty, 7]} Y$
$X - Y \leq -3$ (equiv: $Y - X \geq 3$)	$X \xleftarrow[-3]{} Y$	$X \xrightarrow{[3, +\infty)} Y$
$3 \leq Y - X \leq 7$	$X \underset{-3}{\overset{7}{\rightleftarrows}} Y$	$X \xrightarrow{[3, 7]} Y$
$4 \leq X \leq 9$	$Z \underset{-4}{\overset{9}{\rightleftarrows}} X$	$Z \xrightarrow{[4, 9]} X$

Simple Temporal Network

Graph for Travel Example

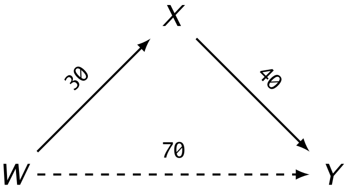
$$\left\{\begin{array}{rcrcrcr} Z - X_1 & \leq & -4, & \quad & X_4 - Z & \leq & 250 \\ X_4 - X_1 & \leq & 168, & & X_2 - X_3 & \leq & -120 \\ X_4 - X_3 & \leq & 7, & & X_1 - X_2 & \leq & 0 \\ X_3 - X_4 & \leq & 0 & & & & \end{array}\right\}$$



Simple Temporal Network
Implicit Constraints

Explicit constraints combine (propagate) to form implicit constraints:

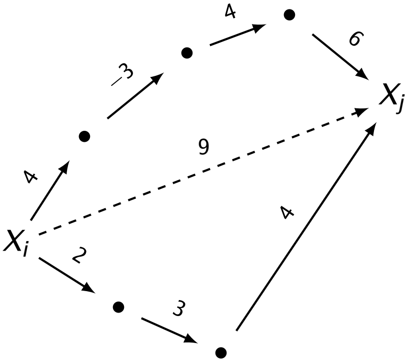
$$\begin{array}{rrcl} & X - W & \leq & 30 \\ + & Y - X & \leq & 40 \\ \hline & Y - W & \leq & 70 \end{array}$$



Simple Temporal Network

Chains of Constraints as Paths

- Chains of constraints correspond to **paths** in the graph.
- **Stronger** constraints correspond to **shorter** paths.



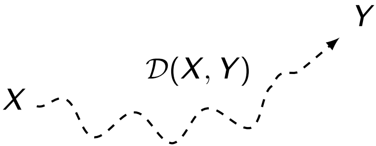
Simple Temporal Network

Distance Matrix [Dechter et al., 1991]

Definition 2 (Distance Matrix)

The *Distance Matrix* for an STN $\mathcal{S}$ is a matrix $\mathcal{D}$ defined by:

$$\mathcal{D}(X, Y) = \text{ Length of Shortest Path from } X \text{ to } Y \text{ in the graph for } \mathcal{S}$$

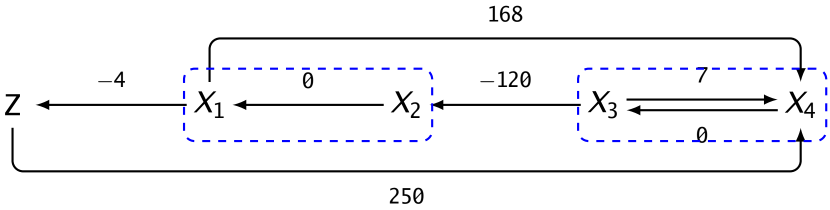


- The strongest implicit constraint on $Y - X$ in $\mathcal{S}$ is:

$$Y - X \leq \mathcal{D}(X, Y)$$

Simple Temporal Network

Distance Matrix for Travel Example



$\mathcal{D}$	Z	X_1	X_2	X_3	X_4
Z	0	130	130	250	250
X_1	−4	0	48	168	168
X_2	−4	0	0	168	168
X_3	−124	−120	−120	0	7
X_4	−124	−120	−120	0	0

Gray cells correspond to explicit edges.

Simple Temporal Network

Fundamental Theorem of STNs **[Dechter et al., 1991]**

For an STN $\mathcal{S}$, with graph $\mathcal{G}$, and distance matrix $\mathcal{D}$,
the following are equivalent:

- $\mathcal{S}$ is consistent
- $\mathcal{D}$ has non–negative values down its main diagonal
- $\mathcal{G}$ has no negative–length loops

Consistency–Checking Algorithms for STNs

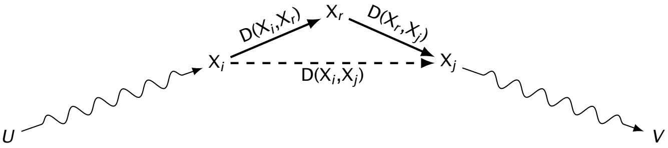
Sample Consistency–Checking Algorithms

- Floyd–Warshall (computes $\mathcal{D}$; generates solutions)
- Bellman–Ford SSSP (checks consistency; generates solution)
- Speed–ups to Bellman–Ford (up to 6 times faster)
- Dijkstra SSSP (only for non–neg. edges, but useful...)
- Johnson (uses BF and Dijkstra to compute $\mathcal{D}$)
- Directional and Partial Path Consistency (DPC and PPC)
- Incremental algorithms

Floyd–Warshall Algorithm

Floyd–Warshall Algorithm

Computes distance matrix $\mathcal{D}$ in $O(n^3)$ time [Floyd, 1962; Warshall, 1962]



```
Initialize D(_,_) using edge-weights
for r=1 to n
   for i=1 to n
      for j=1 to n
         D(X_i,X_j) := min{D(X_i,X_j), D(X_i,X_r) + D(X_r,X_j)}
return D
```

If a shortest path from U to V contains X_r as an interior point, then after the r^{th} round, that shortest path can ignore X_r.

Extracting Solutions from $\mathcal{D}$

[Dechter et al., 1991]

- For each $X \in \mathcal{T}$, its *time window* is: $[-\mathcal{D}(X, Z), \mathcal{D}(Z, X)]$
  - $-\mathcal{D}(X, Z)$ is a *lower–bound* for X because
    $$Z - X \leq \mathcal{D}(Z, X) \iff X \geq -\mathcal{D}(Z, X).$$
  - $\mathcal{D}(Z, X)$ is an *upper–bound* for X because
    $$X - Z \leq \mathcal{D}(X, Z) \iff X \leq \mathcal{D}(X, Z).$$
- Two easy–to–find solutions:
  - *Earliest–times* solution:
    $X_1 = -\mathcal{D}(X_1, Z),\ X_2 = -\mathcal{D}(X_2, Z), \ldots,\ X_n = -\mathcal{D}(X_n, Z)$
  - *Latest–times* solution:
    $X_1 = \mathcal{D}(Z, X_1),\ X_2 = \mathcal{D}(Z, X_2), \ldots,\ X_n = \mathcal{D}(Z, X_n)$.

GenSoln

Generating any and all solutions from $\mathcal{D}$ [Dechter et al., 1991]

Given any consistent STN graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ with distance matrix $\mathcal{D}$:

1. $\mathcal{U} := \mathcal{T}$ (currently unexecuted time–points)
2. For each $X \in \mathcal{T},\ \mathrm{TW}(X) = [-\mathcal{D}(X,Z), \mathcal{D}(Z,X)]$ (time windows)
3. Choose: Pick some $X \in \mathcal{U}$, and some $t \in \mathrm{TW}(X)$
4. Execute: set $X := t$, and remove X from $\mathcal{U}$
5. Propagate: Update time windows:

   For each $Y \in \mathcal{U}$: $\mathrm{TW}_Y := \mathrm{TW}_Y \cap [t - \mathcal{D}(Y,X), t + \mathcal{D}(X,Y)]$

   $$\boxed{\begin{array}{lllllll} \text{Upper: } Y - X \leq \mathcal{D}(X,Y) & \Longrightarrow & Y & \leq & X + \mathcal{D}(X,Y) & = & t + \mathcal{D}(X,Y) \\ \text{Lower: } X - Y \leq \mathcal{D}(Y,X) & \Longrightarrow & Y & \geq & X - \mathcal{D}(Y,X) & = & t - \mathcal{D}(Y,X) \end{array}}$$

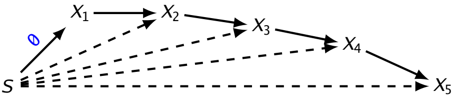
6. If $\mathcal{U}$ non–empty, go back to (3); else done.

Bellman–Ford Algorithm and Friends

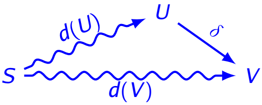
Bellman–Ford Algorithm: Version 1

An $O(mn)$–time SSSP[*] algorithm [Bellman, 1958; Ford and Fulkerson, 1962]

- Introduce new node $S \notin \mathcal{T}$ to use as a source node.
- Goal: Compute $d(X)$ = distance from S to X, for each $X \in \mathcal{T}$.
- Initialization: $d(X) = 0$ for each $X \in \mathcal{T}$.



```
for i=1 to (n-1),
   for each edge (U,$\delta$,V) in graph,
      d(V) := min{d(V), d(U) + $\delta$}
for each edge (U,$\delta$,V) in graph,
   if (d(V) > d(U) + $\delta$) return false
return true
```



⇒ After k^{th} iteration, will know length of every shortest path having at most k edges.

---

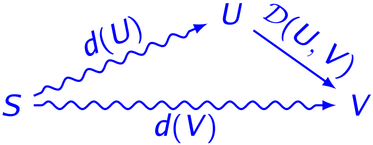
[*] SSSP = single–source, shortest–path

Extracting a Solution from Bellman–Ford

[Bellman, 1958; Ford and Fulkerson, 1962]

For a consistent STN $\mathcal{S}$, the distance function $d(X)$ computed by Bellman–Ford (Version 1) is a solution for $\mathcal{S}$.

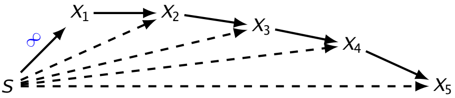
$$d(V) \leq d(U) + \mathcal{D}(U,V) \iff d(V) - d(U) \leq \mathcal{D}(U,V)$$



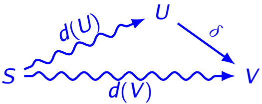
Bellman–Ford Algorithm: Version 2

An $O(mn)$–time SSSP* algorithm [Bellman, 1958; Ford and Fulkerson, 1962]

- Pick any node $S \in \mathcal{T}$ to use as the source node.
- Goal: Compute $d(X)$ = distance from S to X, for each $X \in \mathcal{T}$.
- Initialization: $d(X) = \infty$ for each $X \in \mathcal{T} \setminus \{S\}$.



```
for i=1 to (n-1),
   for each edge (U,δ,V) in graph,
      d(V) := min{d(V), d(U) + δ}
for each edge (U,δ,V) in graph,
   if (d(V) > d(U) + δ) return false
return true
```



⇒ After k^{th} iteration, will know length of every shortest path having at most k edges.

---

* SSSP = single–source, shortest–path

Speed–ups of Bellman–Ford

[Bannister and Eppstein, 2012; Yen, 1970]

- Stop early if no changes in preceding iteration.
- If no changes to $d(U)$ in preceding iteration, no need to check edges emanating from U in current iteration.
- Given an ordering/ranking of the time–points, *partition* the graph into two sub–graphs, $\mathcal{G}^+$ and $\mathcal{G}^-$, where:
  - $\mathcal{G}^+$ contains edges from lower–ranked to higher–ranked time–points, and
  - $\mathcal{G}^-$ contains edges from higher–ranked to lower–ranked time–points.
- During odd iterations, only propagate along edges in $\mathcal{G}^+$; during even iterations, propagate along edges in $\mathcal{G}^-$.
- Random ranking can make Bellman–Ford up to six times faster.

Speed–ups of Bellman–Ford (cont'd.)

[Bannister and Eppstein, 2012; Yen, 1970]

- Suppose source node is $X_0 \in \mathcal{T}$.
- Initialization: For each $X \in \mathcal{T}$, $d(X) = \infty$, except $d(X_0) = 0$.
- Suppose ranking is: $\{X_0, X_1, X_2, \ldots, X_n\}$.
- Just one iteration to compute lengths of all shortest paths in $\mathcal{G}^+$.

  Example: $X_0 \xrightarrow{-1} X_2 \xrightarrow{-3} X_4 \xrightarrow{2} X_7 \xrightarrow{-8} X_8 \xrightarrow{1} X_9$
- $k+1$ iterations to compute lengths of all shortest paths having at most k transitions between edges in $\mathcal{G}^+$ and edges in $\mathcal{G}^-$.

  Example: $X_0 \xrightarrow{1} X_1 \xrightarrow{-7} X_4 \xrightarrow{-1} X_3 \xrightarrow{3} X_2 \xrightarrow{1} X_9 \xrightarrow{2} X_{12}$

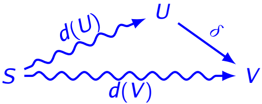
Dijkstra’s Algorithm

Dijkstra's Algorithm

SSSP algorithm for checking consistency in $O(m + n \log n)$ time [Dijkstra, 1959]

- Only works on STN graphs with *non–negative edges*
- For given source node $S \in \mathcal{T}$,
  computes $d(X) =$ distance from S to X, for all X.
- $O(m + n \log n)$ if using *Fibonacci heap* for priority queue

```
$d(X) := \infty$ for all $X$, but $d(S) := 0$
$\mathcal{Q} :=$ an empty priority queue
Insert $S$ into $\mathcal{Q}$ with priority 0
while $\mathcal{Q}$ non-empty,
   U := ExtractMinFrom($\mathcal{Q}$)
   for each successor edge $(U, \delta, V)$,
      $d(V) := \min\{d(V), d(U) + \delta\}$
return $d$
```



Johnson’s Algorithm

Johnson's Algorithm

Computes $\mathcal{D}$ in $O(mn + n^2 \log n)$ time [Johnson, 1977]

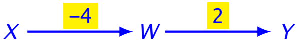
- Dijkstra's alg. only applies to graphs with non–negative edges.
- However, for *any* consistent STN $\mathcal{S}$:
  - Use Bellman–Ford to generate a solution f.
  - Use f as a potential function to re–weight edges in graph to non–negative values (next slide)
  - Then, for each X, use Dijkstra to compute one row of $\mathcal{D}$.
  - Easy to convert between original and non–negative weights (next slide).
- The result is the $O(mn + n^2 \log n)$–time Johnson's Algorithm.

Potential Functions & Re–weighted Graphs

- Given: $f : \mathcal{T} \to \mathbb{R}$, a solution for an STN $\mathcal{S} = (\mathcal{T}, \mathcal{C})$.
  - Then $f(Y) - f(X) \leq \delta$ for each constraint $(Y - X \leq \delta) \in \mathcal{C}$
  - In other words, $\mathtt{0} \leq f(X) + \delta - f(Y)$
  - Let $\mathcal{C}' = \{(X, \delta', Y) \mid (X, \delta, Y) \in \mathcal{C}\}$, where $\delta' = f(X) + \delta - f(Y)$
  - Then $\mathcal{S}' = (\mathcal{T}, \mathcal{C}')$ has only non–negative edges.
  - Therefore, can use Dijkstra's SSSP algorithm on $\mathcal{S}'$

$\Rightarrow$ Shortest paths in $\mathcal{S}$ correspond to shortest paths in $\mathcal{S}'$:

$$\mathcal{D}'(X, Y) = f(X) + \mathcal{D}(X, Y) - f(Y)$$



$$(f(X) + (-4) - f(W)) + (f(W) + 2 - f(Y)) = f(X) + (-4 + 2) - f(Y)$$

Johnson's Algorithm

[Johnson, 1977]

Given: an STN, $\mathcal{S} = (\mathcal{T}, \mathcal{C})$

Run Bellman–Ford to generate solution $f\colon \mathcal{T} \to \mathbb{R}$ for $\mathcal{S}$.

Let $\mathcal{S}' = (\mathcal{T}, \mathcal{C}')$ be re–weighted graph based on f:

$$\delta' = f(X) + \delta - f(Y) \geq \mathtt{0} \text{ for each } (X, \delta, Y) \in \mathcal{C}.$$

For each $X \in \mathcal{T}$, run Dijkstra on $\mathcal{S}'$ with X as source node

— computes $\mathcal{D}'(X, Y)$ for all $Y \in \mathcal{T}$.

Reverse the re–weighting to obtain $\mathcal{D}$ for $\mathcal{S}$:

$$\mathcal{D}(X, Y) = -f(X) + \mathcal{D}'(X, Y) + f(Y).$$

Complexity: $O(mn) + n * O(m + n \log n) = O(mn + n^2 \log n)$

Cython

Introduction to Cython

About Cython

- “Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex)."
- Cython “makes writing C extensions for Python as easy as Python itself."
- “Cython gives you the combined power of Python and C."
- Documentation, Tutorials, Examples available at `cython.org`
- Cython Tutorial [Behnel et al., 2009]

All quotes from `cython.org`

Installing Cython

If you currently have Python version 3.4 or greater:

```
pip3 install Cython
```

Using Cython

- Most source code goes into `*.pyx` files
- Function signatures and struct defs *can* go into `*.pxd` files
- Info about `*.pyx` files goes into a single `setup.py` file
- To Compile: `python3 setup.py build_ext --inplace`

- Generates `*.c` and `*.so` files enabling your modules to be imported into Python
- `cython -a myfile.pyx`

  Generates `html` file showing translation from Cython to C code.

Using Cython (cont'd.)

To speed up Cython code:

- Declare data types especially for arrays and array indices
- Use `numpy` arrays
- Use `csr_matrix` sparse matrices
- Use `malloc` and `free` to dynamically allocate and free memory

Getting Cython Code for this Tutorial

`All code for this tutorial is available at:`

`https://www.cs.vassar.edu/~hunsberg/icaps_2023_tutorial_code/`

`or`

`https://www.cs.vassar.edu/~hunsberg/icaps_2023_tutorial_code.zip`

Basic Cython Example: Version 1 (Pure Python)

Modified from [Behnel et al., 2009]

```
# File:  test1.pyx  --  SOURCE CODE

from math import sin as sin

#  INTEGRATE_SIN
# -------------------------------------------------------
#  Estimate integral of SIN from A to B using N divisions

def integrate_sin(a, b, N):
  s = 0
  dx = (b-a)/N
  for i in range(N):
    s += sin(a+i*dx)
  return s * dx
```

Basic Cython Example: Version 1 (Pure Python)

Modified from [Behnel et al., 2009]

```
# File: test1_setup.py  --  COMPILATION MANAGER

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules= [ Extension("test1", ["test1.pyx"]) ]

for e in ext_modules:
    e.cython_directives = {'language_level': "3"}

setup( name = 'Test1',
       cmdclass = {'build_ext': build_ext},
       ext_modules = ext_modules )
```

Basic Cython Example: Version 1 (Pure Python)

Compilation

```
icaps-2023-repo$ python3 test1_setup.py build_ext --inplace
/Users/hunsberger/Desktop/icaps-2023-repo/test1_setup.py:5: ...
running build_ext
cythoning test1.pyx to test1.c
building ’test1’ extension
clang -Wno-unused-result -Wsign-compare -Wunreachable-code ...
clang -bundle -undefined dynamic_lookup -arch arm64 -arch  ...

icaps-2023-repo$ ls test1.*
test1.c     test1.pyx     test1.cpython-310-darwin.so
```

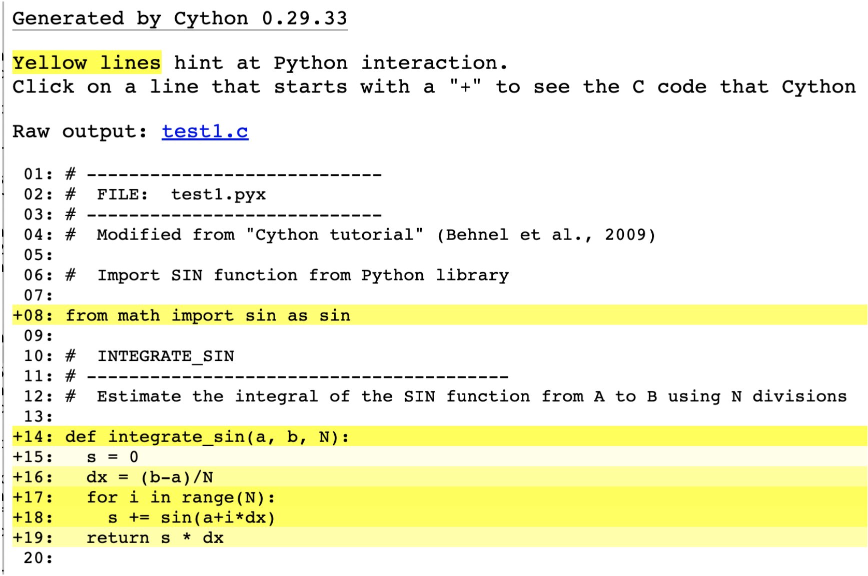
Basic Cython Example: Version 1 (Pure Python)

Importing Module into Python

```
icaps-2023-repo$ python3
Python 3.10.0 (v3.10.0:b494f5935c,  ...
Type "help", "copyright", "credits" ...

>>> import test1

>>> test1.integrate_sin(0, 1.57, 100)
0.991333151217814 7

>>> test1.integrate_sin(0, 6.28, 1000)
1.5074917580179498e-05
```

Basic Cython Example: Version 1 (Pure Python)

`html` **file generated by:** `cython -a test1.pyx`



Generated by Cython 0.29.33

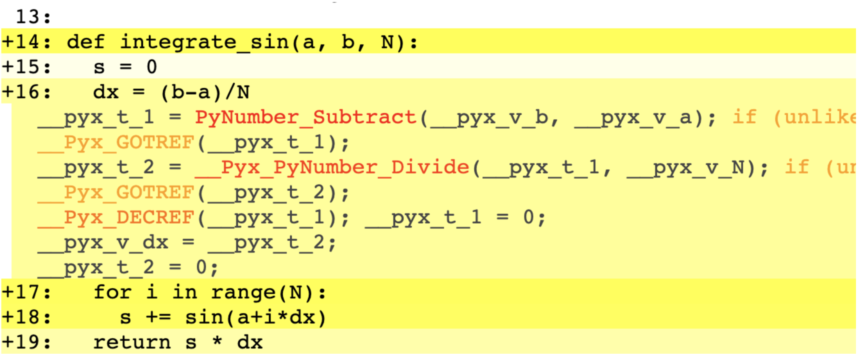
Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython

Raw output: test1.c

```
 01: # ----------------------------
 02: #  FILE:  test1.pyx
 03: # ----------------------------
 04: #  Modified from "Cython tutorial" (Behnel et al., 2009)
 05:
 06: #  Import SIN function from Python library
 07:
+08: from math import sin as sin
 09:
 10: #  INTEGRATE_SIN
 11: # ----------------------------------------
 12: #  Estimate the integral of the SIN function from A to B using N divisions
 13:
+14: def integrate_sin(a, b, N):
+15:   s = 0
+16:   dx = (b-a)/N
+17:   for i in range(N):
+18:     s += sin(a+i*dx)
+19:   return s * dx
 20:
```

Basic Cython Example: Version 1 (Pure Python)

`html` **file generated by:** `cython -a test1.pyx`



```
 13:
+14: def integrate_sin(a, b, N):
+15:   s = 0
+16:   dx = (b-a)/N
   __pyx_t_1 = PyNumber_Subtract(__pyx_v_b, __pyx_v_a); if (unlike
   __Pyx_GOTREF(__pyx_t_1);
   __pyx_t_2 = __Pyx_PyNumber_Divide(__pyx_t_1, __pyx_v_N); if (ur
   __Pyx_GOTREF(__pyx_t_2);
   __Pyx_DECREF(__pyx_t_1); __pyx_t_1 = 0;
   __pyx_v_dx = __pyx_t_2;
   __pyx_t_2 = 0;
+17:   for i in range(N):
+18:     s += sin(a+i*dx)
+19:   return s * dx
```

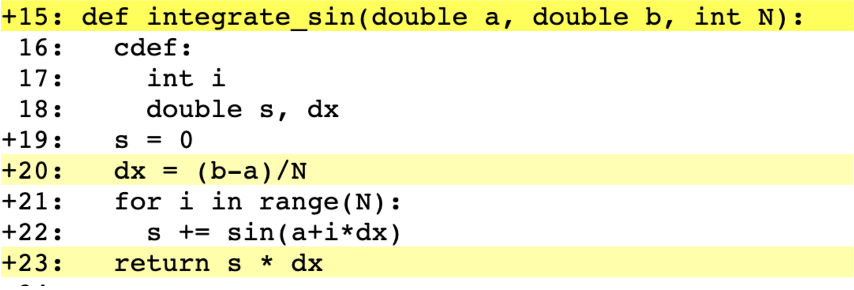
Basic Cython Example: Version 2

Modified from [Behnel et al., 2009]

```
#  File:  test2.pyx  --  SOURCE CODE

#  Import SIN function from C math library
from libc.math cimport sin

#  INTEGRATE_SIN
# ---------------------------------------------------------
#  Estimate integral of SIN from A to B using N divisions

def integrate_sin( double a, double b, int N ):
  cdef:
    int i
    double s, dx
  s = 0
  dx = (b-a)/N
  for i in range(N):
    s += sin(a+i*dx)
  return s * dx
```

Basic Cython Example: Version 2

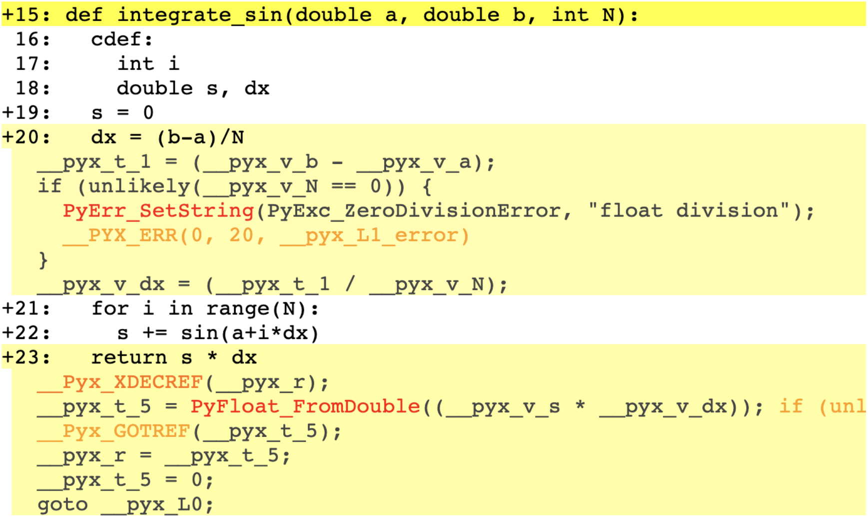
`html` **file generated by:** `cython -a test2.pyx`



```
+15: def integrate_sin(double a, double b, int N):
 16:   cdef:
 17:     int i
 18:     double s, dx
+19:   s = 0
+20:   dx = (b-a)/N
+21:   for i in range(N):
+22:     s += sin(a+i*dx)
+23:   return s * dx
```

Basic Cython Example: Version 2

`html` **file generated by:** `cython -a test2.pyx`



```
+15: def integrate_sin(double a, double b, int N):
 16:   cdef:
 17:     int i
 18:     double s, dx
+19:   s = 0
+20:   dx = (b-a)/N
  __pyx_t_1 = (__pyx_v_b - __pyx_v_a);
  if (unlikely(__pyx_v_N == 0)) {
    PyErr_SetString(PyExc_ZeroDivisionError, "float division");
    __PYX_ERR(0, 20, __pyx_L1_error)
  }
  __pyx_v_dx = (__pyx_t_1 / __pyx_v_N);
+21:   for i in range(N):
+22:     s += sin(a+i*dx)
+23:   return s * dx
  __Pyx_XDECREF(__pyx_r);
  __pyx_t_5 = PyFloat_FromDouble((__pyx_v_s * __pyx_v_dx)); if (unl
  __Pyx_GOTREF(__pyx_t_5);
  __pyx_r = __pyx_t_5;
  __pyx_t_5 = 0;
  goto __pyx_L0;
```

Cython Code for STNs

Representing STNs

First Attempt

- An STN graph is a pair $(\mathcal{T}, \mathcal{E})$ where:
  - $\mathcal{T}$ is a set of n time–points: $X_0, X_1, \ldots, X_{n-1}$
  - $\mathcal{E}$ is a set of m edges, each of the form: $X_i \xrightarrow{\delta} X_j$
- The time–points can be represented by numerical indices:

$$0,\ 1,\ \ldots,\ n-1$$

- The edges can be represented by a list:

$$(\ (X_{i_1}, \delta_1, X_{j_1}),\ (X_{i_2}, \delta_2, X_{j_2}),\ \ldots\ (X_{i_m}, \delta_m, X_{j_m})\ )$$

...but that doesn't allow fast access.

Representing STNs

Second Attempt

The edges in an STN can be represented by an *n*–by–*n* array/matrix:

	0	1	2	3	4
0	0	∞	∞	-2	5
1	∞	0	-7	∞	∞
2	-5	3	0	∞	8
3	∞	2	9	0	6
4	∞	-2	∞	∞	0

. . . but that wastes space if the graph is sparse.

For example, if: $m = 10 \ll 25 = n^2$.

. . . "no edge" represented by ∞.

. . . diagonal entries contain 0.

Compressed Sparse Row (CSR) Matrices

	0	1	2	3	4
0	0	∞	∞	-2	5
1	∞	0	-7	∞	∞
2	-5	3	0	∞	8
3	∞	2	9	0	6
4	∞	-2	∞	∞	0

$n = 5$ time–points

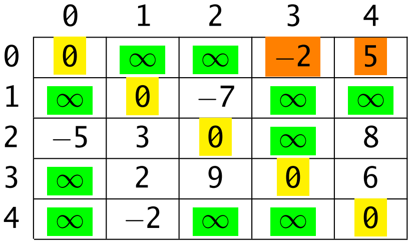
$m = 10$ edges

	0	1	2	3	4	5	6	7	8	9
`wts` :	-2	5	-7	-5	3	8	2	9	6	-2
`cols` :	3	4	2	0	1	4	1	2	4	1

`indptr` :	0	2	3	6	9	10

- `wts` and `cols` are m–vectors (one entry per edge).
- `indptr` is an $(n+1)$–vector.
- Content of `wts` and `cols` for row r starts at index `indptr[r]`

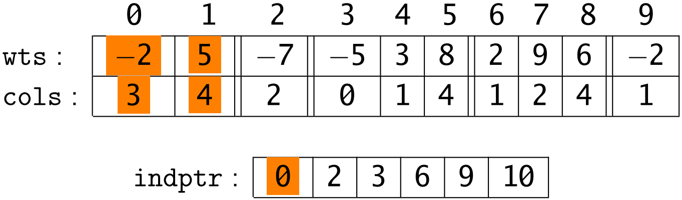
Compressed Sparse Row (CSR) Matrices



	0	1	2	3	4
0	0	∞	∞	−2	5
1	∞	0	−7	∞	∞
2	−5	3	0	∞	8
3	∞	2	9	0	6
4	∞	−2	∞	∞	0

$n = 5$ time–points

$m = 10$ edges



	0	1	2	3	4	5	6	7	8	9
`wts` :	−2	5	−7	−5	3	8	2	9	6	−2
`cols` :	3	4	2	0	1	4	1	2	4	1

`indptr` :	0	2	3	6	9	10

- `wts` and `cols` are m–vectors (one entry per edge).
- `indptr` is an $(n + 1)$–vector.
- Content of `wts` and `cols` for row r starts at index `indptr[r]`

Compressed Sparse Row (CSR) Matrices

	0	1	2	3	4
0	0	∞	∞	−2	5
1	∞	0	−7	∞	∞
2	−5	3	0	∞	8
3	∞	2	9	0	6
4	∞	−2	∞	∞	0

$n = 5$ time–points

$m = 10$ edges

	0	1	2	3	4	5	6	7	8	9
`wts` :	−2	5	−7	−5	3	8	2	9	6	−2
`cols` :	3	4	2	0	1	4	1	2	4	1

`indptr` :	0	2	3	6	9	10

- `wts` and `cols` are m–vectors (one entry per edge).
- `indptr` is an $(n + 1)$–vector.
- Content of `wts` and `cols` for row r starts at index `indptr[r]`

Compressed Sparse Row (CSR) Matrices

	0	1	2	3	4
0	0	∞	∞	-2	5
1	∞	0	-7	∞	∞
2	-5	3	0	∞	8
3	∞	2	9	0	6
4	∞	-2	∞	∞	0

$n = 5$ time–points

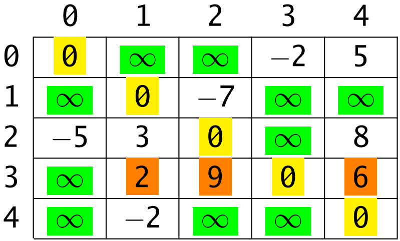
$m = 10$ edges

	0	1	2	3	4	5	6	7	8	9
`wts` :	-2	5	-7	-5	3	8	2	9	6	-2
`cols` :	3	4	2	0	1	4	1	2	4	1

`indptr` :	0	2	3	6	9	10

- `wts` and `cols` are m–vectors (one entry per edge).
- `indptr` is an $(n+1)$–vector.
- Content of `wts` and `cols` for row r starts at index `indptr[r]`

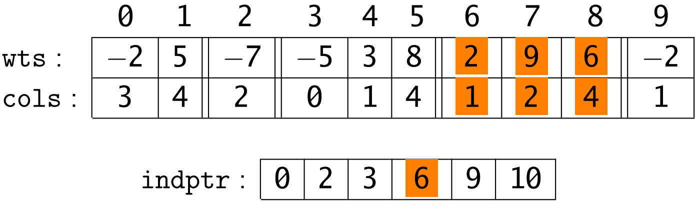
Compressed Sparse Row (CSR) Matrices



	0	1	2	3	4
0	0	∞	∞	-2	5
1	∞	0	-7	∞	∞
2	-5	3	0	∞	8
3	∞	2	9	0	6
4	∞	-2	∞	∞	0

$n = 5$ time–points

$m = 10$ edges



	0	1	2	3	4	5	6	7	8	9
`wts` :	-2	5	-7	-5	3	8	2	9	6	-2
`cols` :	3	4	2	0	1	4	1	2	4	1

`indptr` :	0	2	3	6	9	10

- `wts` and `cols` are m–vectors (one entry per edge).
- `indptr` is an $(n + 1)$–vector.
- Content of `wts` and `cols` for row r starts at index `indptr[r]`

Compressed Sparse Row (CSR) Matrices

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & \infty & \infty & -2 & 5 \\
1 & \infty & 0 & -7 & \infty & \infty \\
2 & -5 & 3 & 0 & \infty & 8 \\
3 & \infty & 2 & 9 & 0 & 6 \\
4 & \infty & -2 & \infty & \infty & 0
\end{array}$$

$n = 5$ time–points

$m = 10$ edges

	0	1	2	3	4	5	6	7	8	9
`wts` :	−2	5	−7	−5	3	8	2	9	6	−2
`cols` :	3	4	2	0	1	4	1	2	4	1

`indptr` :	0	2	3	6	9	10

- `wts` and `cols` are m–vectors (one entry per edge).
- `indptr` is an $(n+1)$–vector.
- Content of `wts` and `cols` for row r starts at index `indptr[r]`

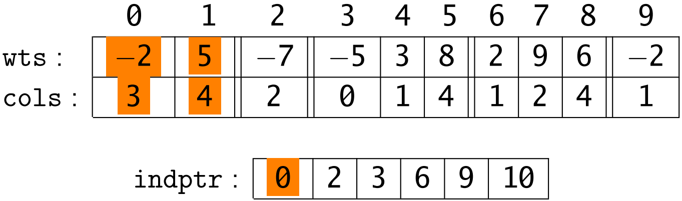
Iterating over Edges in a CSR Matrix

	0	1	2	3	4	5	6	7	8	9
wts :	-2	5	-7	-5	3	8	2	9	6	-2
cols :	3	4	2	0	1	4	1	2	4	1

indptr :	0	2	3	6	9	10

```
for row in range(n):
  for indy in range(indptr[row], indptr[row+1]):
    print(f"EDGE: ({row}, {wts[indy]}, {cols[indy]})")
```

Iterating over Edges in a CSR Matrix

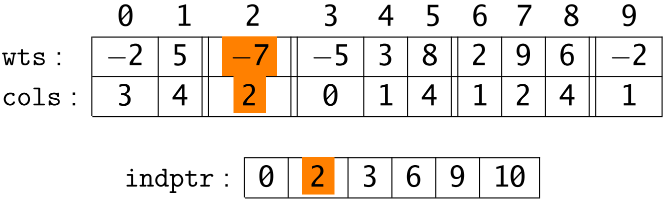


	0	1	2	3	4	5	6	7	8	9
wts :	−2	5	−7	−5	3	8	2	9	6	−2
cols :	3	4	2	0	1	4	1	2	4	1

indptr :	0	2	3	6	9	10

```
for row in range(n):
  for indy in range(indptr[row], indptr[row+1]):
    print(f"EDGE: ({row}, {wts[indy]}, {cols[indy]})")
```

Iterating over Edges in a CSR Matrix

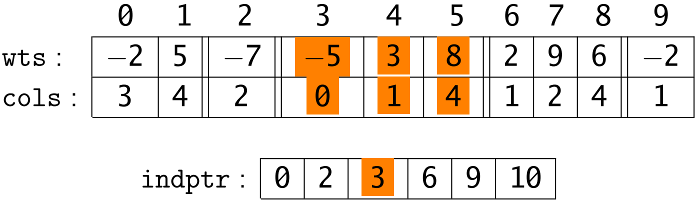


	0	1	2	3	4	5	6	7	8	9
wts :	−2	5	−7	−5	3	8	2	9	6	−2
cols :	3	4	2	0	1	4	1	2	4	1

indptr :	0	2	3	6	9	10

```
for row in range(n):
  for indy in range(indptr[row], indptr[row+1]):
    print(f"EDGE: ({row}, {wts[indy]}, {cols[indy]})")
```

Iterating over Edges in a CSR Matrix

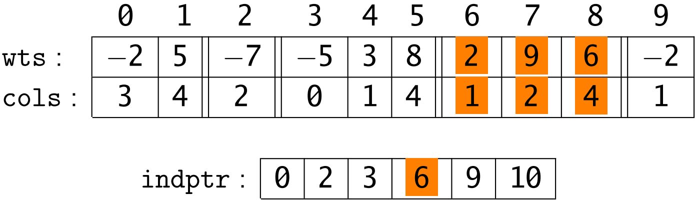


	0	1	2	3	4	5	6	7	8	9
wts :	−2	5	−7	−5	3	8	2	9	6	−2
cols :	3	4	2	0	1	4	1	2	4	1

indptr :	0	2	3	6	9	10

```
for row in range(n):
  for indy in range(indptr[row], indptr[row+1]):
    print(f"EDGE: ({row}, {wts[indy]}, {cols[indy]})")
```

Iterating over Edges in a CSR Matrix

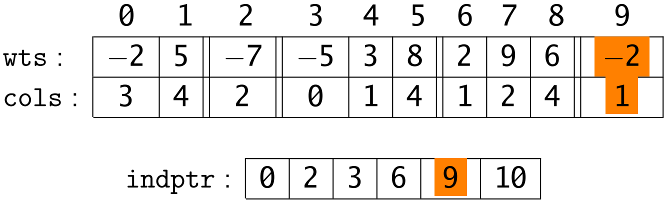


	0	1	2	3	4	5	6	7	8	9
wts :	-2	5	-7	-5	3	8	2	9	6	-2
cols :	3	4	2	0	1	4	1	2	4	1

indptr :	0	2	3	6	9	10

```
for row in range(n):
  for indy in range(indptr[row], indptr[row+1]):
    print(f"EDGE: ({row}, {wts[indy]}, {cols[indy]})")
```

Iterating over Edges in a CSR Matrix



	0	1	2	3	4	5	6	7	8	9
wts :	-2	5	-7	-5	3	8	2	9	6	-2
cols :	3	4	2	0	1	4	1	2	4	1

indptr :	0	2	3	6	9	10

```
for row in range(n):
  for indy in range(indptr[row], indptr[row+1]):
    print(f"EDGE: ({row}, {wts[indy]}, {cols[indy]})")
```

Compressed Sparse Row (CSR) Matrices

- A CSR matrix only represents the edges that are present.
- For an n–by–n matrix with m entries, a CSR matrix uses three vectors that, in the literature, are called:
  - `data`: an m–vector (we use for weights)
  - `indices`: an m–vector (we use for column indices)
  - `indptr`: an $(n+1)$–vector
- For each row $\mathtt{r} \in \{0, 1, \ldots, n-1\}$,

  For each index $\mathtt{i} \in [\mathtt{indptr}[\mathtt{r}], \mathtt{indptr}[\mathtt{r}+1]]$,

  There is an edge: $\mathtt{r} \xrightarrow{\mathtt{data}[\mathtt{i}]} \mathtt{indices}[\mathtt{i}]$.
- Can use `csr_matrix` from `scipy.sparse`, but:
  - Their algorithms assume that implicit entries are 0, whereas we need ∞ in off–diagonal entries.

Modules in STN Library

File Name	Description
`min_bin_heaps.pyx`	Minimum Binary Heaps
`fib_heaps.pyx`	Fibonacci Heaps
`bellman_ford.pyx`	Bellman–Ford Alg.
`dist_mat.pyx`	Distance–matrix Algs.
`lifo.pyx`	Last–in, first–out queues
`pred_graphs.pyx`	Predecessor graphs
`tarjan_scc.pyx`	Tarjan's SCC Alg.
`rigid_components.pyx`	Compute rigid components
`rev_post_order.pyx`	Reverse post–order
`disp_new.pyx`	Dispatchability Algs.

- Only def functions can be imported into Python session.
- But cdef functions can be imported/used by other Cython modules if their signatures are included in the corresponding `*.pxd` file.
- Similarly, cdef Cython structs and enums that are defined in `*.pxd` files can be used by other Cython modules.

Example: `min_bin_heaps.pyx/pxd`

Implementation of Minimum Binary Heaps

- `min_bin_heaps.pxd` file provides:
  - Definition for `enum State`
  - Definitions for `struct Node` and `struct MinBinHeap`
  - Signatures for Cython functions: `init_heap`, `clear_heap`, `free_heap`, `is_empty`, `insert_node`, `decrease_key`, `get_status`, `insert_or_decrease_key`, and `extract_min_node`.
- `min_bin_heaps.pyx` file provides:
  - Definitions for all of the above (exportable) Cython functions.
  - Definitions of private Cython functions: `swapper`, `init_node`, `print_node`, `get_status` and `min_heapify`.
  - Definition of a Python–importable function: `test_mbh`.

Testing the `min_bin_heaps` Module

The `test_mbh` function generates `num` random numbers, inserts them into the queue, and then extracts them in order of priority.

```
>>> mbh.test_mbh(5)
node: tp:1, pri: 5.0 , loc:-1, state:ALREADY_POPPED
node: tp:3, pri: 6.0 , loc:-1, state:ALREADY_POPPED
node: tp:4, pri: 21.0 , loc:-1, state:ALREADY_POPPED
node: tp:0, pri: 22.0 , loc:-1, state:ALREADY_POPPED
node: tp:2, pri: 48.0 , loc:-1, state:ALREADY_POPPED
--- MBH Test Done! ---
```

Testing the `dist_mat` Module

Algorithms for computing the distance matrix

```
>>> import stn_helpers as sh           # See Python file:  stn_helpers.py
>>> import dist_mat as dm
>>> gr = sh.gen_rand_csr_matrix(5,12)    # Generate random STN with n=5, m=12
>>> sh.show_square_csr_matrix(gr)        # Display the STN edges as square matrix
______  ______   18.00   34.00  ______

-47.00  ______  ______  -20.00  -21.00

______   64.00  ______   25.00   10.00

 -5.00  ______   -4.00  ______  ______

______  ______    7.00    3.00  ______

>>> (retval, disty) = dm.fw(5, gr)       # Call Floyd-Warshall
>>> disty                                # Show the distance matrix
array([[  0.,  82.,   18. ,   31. ,   28.],
       [ -47. ,   0.,  -29.,  -20. ,  -21. ],
       [  8.,  64. ,    0.,   13. ,   10. ],
       [  -5. , 60.,  -4. ,     0.,    6.],
       [ -2.,  63.,   -1. ,    3. ,    0.]])
```

$2 \xrightarrow{10} 4 \xrightarrow{3} 3$

$4 \xrightarrow{3} 3 \xrightarrow{-4} 2$

Real–Time Execution and Dispatchability for STNs

Motivating Dispatchability

Motivating Dispatchability

- Concern: Solution fixed in advance has no flexibility.
- Goal: Preserve flexibility by postponing execution decisions until needed in real time—without incurring heavy computational cost.

A *dispatchable* STN:

- Preserves maximum flexibility
- Supports generating solutions in real time
- Requires only *local* propagation during execution

Background: *GenSoln*

An Algorithm for Generating an STN Solution [Dechter et al., 1991]

Given any consistent STN graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$:

1. $\mathcal{U} := \mathcal{T}$ (currently unexecuted time–points)
2. For each $X \in \mathcal{T},\ \mathrm{TW}(X) = [-\mathcal{D}(X,Z), \mathcal{D}(Z,X)]$ (time windows)
3. Choose: Pick some $X \in \mathcal{U}$, and some $t \in \mathrm{TW}(X)$
4. Execute: set $X := t$, and remove X from $\mathcal{U}$
5. Propagate: Update time windows:

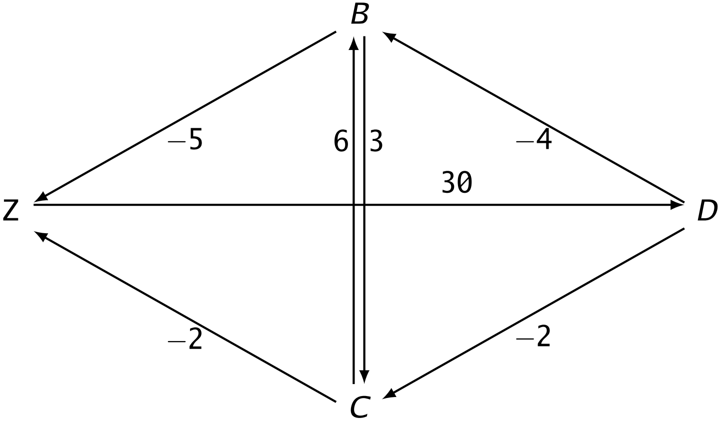
   For each $Y \in \mathcal{U}$: $\mathrm{TW}_Y := \mathrm{TW}_Y \cap [t - \mathcal{D}(Y,X), t + \mathcal{D}(X,Y)]$

   $$\boxed{\begin{array}{lccccc} \text{Upper: } Y - X \leq \mathcal{D}(X,Y) & \Longrightarrow & Y & \leq & X + \mathcal{D}(X,Y) & = & t + \mathcal{D}(X,Y) \\ \text{Lower: } X - Y \leq \mathcal{D}(Y,X) & \Longrightarrow & Y & \geq & X - \mathcal{D}(Y,X) & = & t - \mathcal{D}(Y,X) \end{array}}$$

6. If $\mathcal{U}$ non–empty, go back to (3); else done.

Executing an STN in Real Time

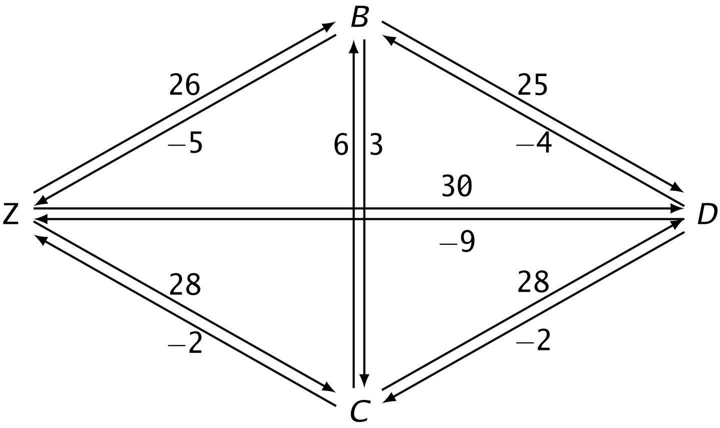
Initial Attempt: Using *GenSoln*



Executing an STN in Real Time

Initial Attempt: Using *GenSoln*

Compute $\mathcal{D}$ (equiv., compute APSP graph)

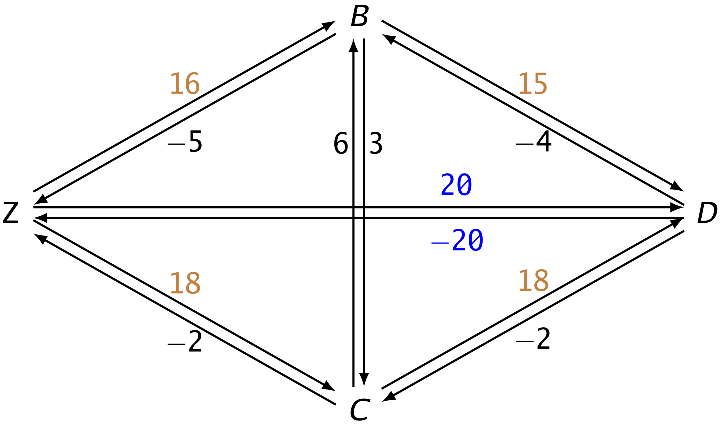


Time Windows: $Z \in [0, 0]$, $B \in [5, 26]$, $C \in [2, 28]$, $D \in [9, 30]$

Executing an STN in Real Time

Initial Attempt: Using *GenSoln*

Arbitrarily choose $D = 20 \in [9, 30]$; then update $\mathcal{D}$:



Updated Time Windows: $B \in [5, 16]$, $C \in [2, 18]$

Whoops! Can't go back in time to execute $B \leq 16$ or $C \leq 18$!

Executing an STN in Real Time

Initial attempt: Using *GenSoln*

- *GenSoln* is great for finding solutions for consistent STNs —*in advance.*
- *GenSoln* is *not* reliable for real–time exeuction.
  ⇒ Can't go backward in time!
- Also: Updating $\mathcal{D}$ after each variable assignment is expensive.
- Another lesson: Shouldn't execute a time–point like D until all of its *outgoing negative edges* point at *already–executed time–points* (i.e., until D is *enabled*).

Real–Time Execution (RTE) Algorithm

[Muscettola et al., 1998b]

Goal: Preserve flexibility while requiring minimal computation.

⓪ For each $X \in \mathcal{T},\ \mathrm{TW}(X) = [0, \infty)$ (time windows)

① $t := 0$ (curr. time); $\mathcal{U} := \mathcal{T}$ (unexecuted); $\mathbf{E} := \{Z\}$ (enabled)

② Choose: Remove any $X \in \mathbf{E}$ such that t is in X's time window;

③ Execute: set $X := t$, and remove X from $\mathcal{U}$;

④ Propagate: propagate $X = t$ to X's *immediate neighbors*;

⑤ Update: update $\mathbf{E}$ to include all $Y \in \mathcal{U}$ for which *no* negative edges emanating from Y have a destination in $\mathcal{U}$;

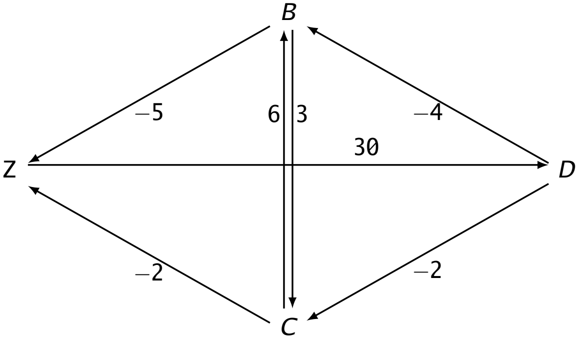
⑥ Wait: wait until t has advanced to some time between $\min\{lb(W) \mid W \in \mathbf{E}\}$ and $\min\{ub(W) \mid W \in \mathbf{E}\}$;

⑦ If $\mathcal{U}$ non–empty, go back to (2); else done.

Executing an STN in Real Time

Second Attempt: Using *RTE* Algorithm

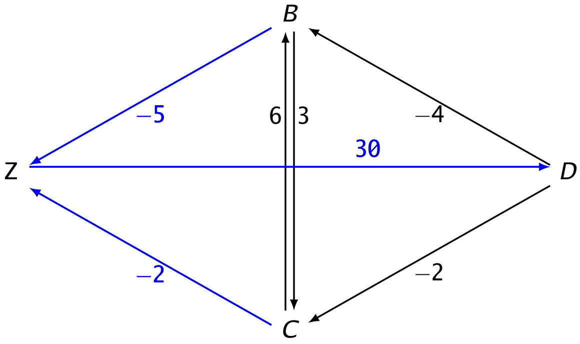
Initially, $\mathbf{E} = \{Z\}$ and $t = 0 \in \mathrm{TW}(Z) = [0, \infty)$



Executing an STN in Real Time

Second Attempt: Using *RTE* Algorithm

Initially, $\mathbf{E} = \{Z\}$ and $t = 0 \in \mathrm{TW}(Z) = [0, \infty)$

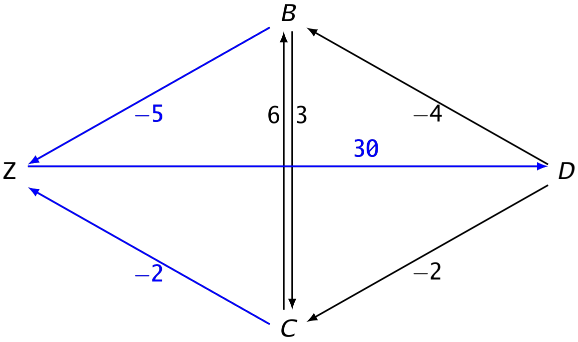


After executing $Z := 0$, $\mathbf{E} = \{B, C\}$ and
$\mathrm{TW}(B) = [5, \infty),\ \mathrm{TW}(C) = [2, \infty),\ \mathrm{TW}(D) = [0, 30]$

Executing an STN in Real Time

Second Attempt: Using *RTE* Algorithm

Initially, $\mathbf{E} = \{Z\}$ and $t = 0 \in \mathrm{TW}(Z) = [0, \infty)$



After executing $Z := 0$, $\mathbf{E} = \{B, C\}$ and

$$\mathrm{TW}(B) = [5, \infty),\ \mathrm{TW}(C) = [2, \infty),\ \mathrm{TW}(D) = [0, 30]$$

So, according to RTE, can wait until $t = 100$, then choose $B := 100$

Dispatchable STN

Definition [Muscettola et al., 1998b]

An STN $\mathcal{S}$ is *dispatchable* if the RTE Algorithm necessarily successfully executes $\mathcal{S}$ *in real time*.

Dispatchable STN

Definition [Muscettola et al., 1998b]

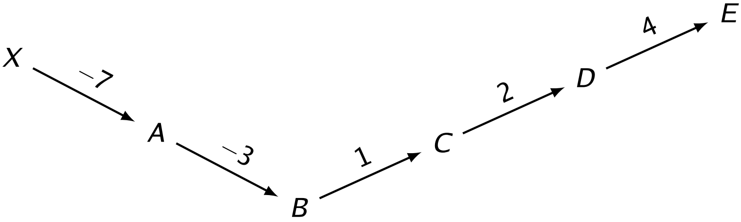
An STN $\mathcal{S}$ is *dispatchable* if the RTE Algorithm necessarily successfully executes $\mathcal{S}$ *in real time*.

(The graph in the preceding example was not dispatchable!)

Equivalent Characterization of Dispatchability

[Morris, 2016]

- Morris found a graphical characterization of dispatchability in terms of *vee–paths*.
- A *vee–path* consists of zero or more negative edges followed by zero or more non–negative edges.
- Theorem: An STN is dispatchable iff for every $X, Y \in \mathcal{T}$, if there is a path from X to Y in $\mathcal{G}$, then there is a *shortest* path from X to Y that is a *vee–path*.



Dispatchability Algorithms for STNs

- The APSP graph is always dispatchable, but has $O(n^2)$ edges.
- Dispatchability algorithms determine which edges from the APSP graph are needed to ensure dispatchability.
  - $O(n^3)$–time *edge–filtering* algorithm [Muscettola et al., 1998b]
    Start with APSP graph, then remove *dominated* edges.
  - $O(mn + n^2 \log n)$–time algorithm [Tsamardinos et al., 1998]
    Accumulate *undominated* edges without building APSP graph.

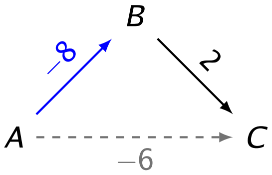
Filtering Algorithm for STN Dispatchability

[Muscettola et al., 1998b]

Dominated Edges – Part 1

Dominated Negative Edges

A negative edge AC is dominated by a *negative* edge AB if $\mathcal{D}(A, B) + \mathcal{D}(B, C) = \mathcal{D}(A, B)$:

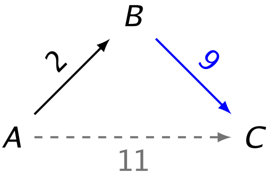


- AB and AC have the *same source* node: A.
- During execution, it is not necessary to propagate (backward) along dominated negative edges (e.g., AC).

Dominated Edges – Part 2

Dominated Non–Negative Edges

A non–negative edge AC is dominated by a *non–negative* edge BC if $\mathcal{D}(A, B) + \mathcal{D}(B, C) = \mathcal{D}(A, B)$:

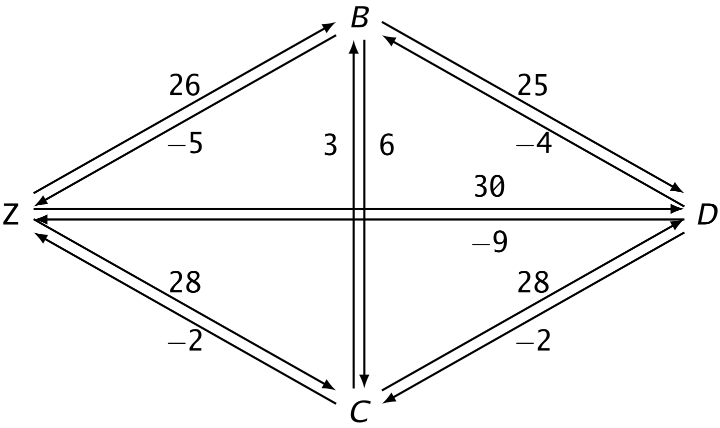


- BC and AC have the *same destination* node: C.
- During execution, it is not necessary to propagate (forward) along dominated non–negative edges (e.g., AC).

Edge–Filtering Algorithm

Example

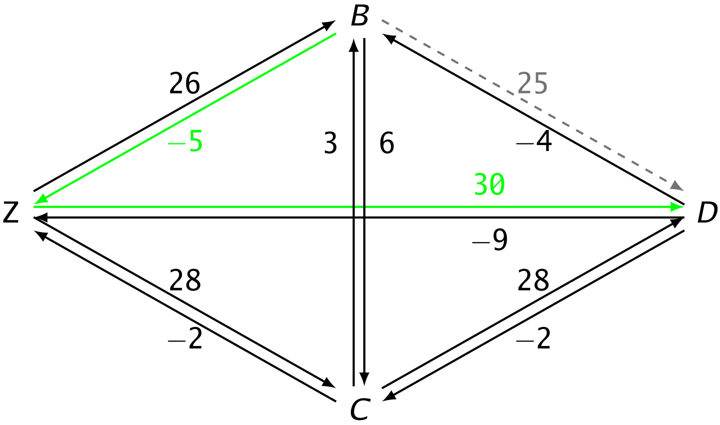
Start with Distance Matrix



Edge–Filtering Algorithm

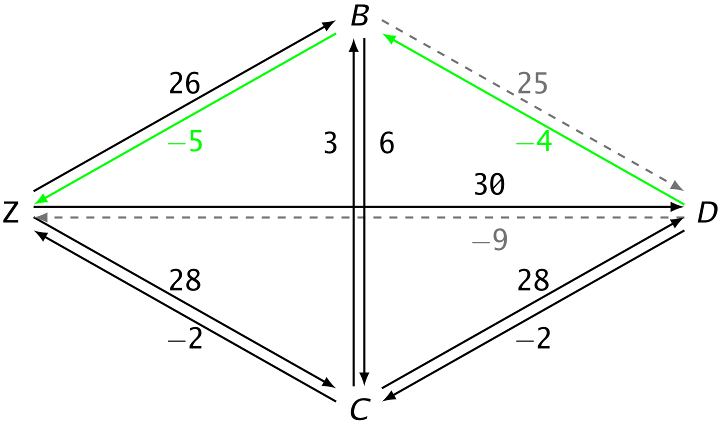
Example

Remove "dominated" edges:



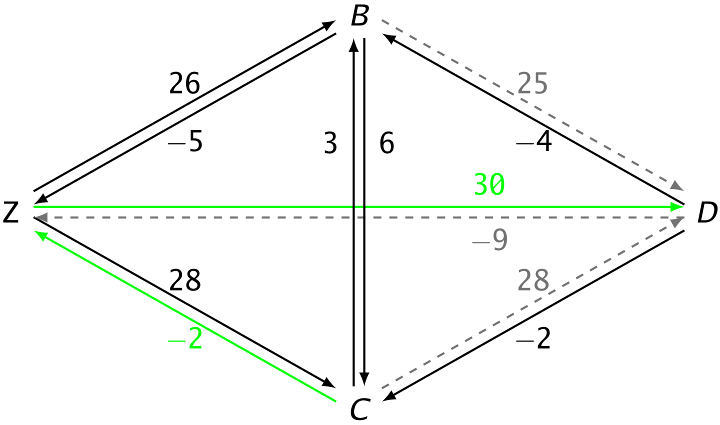
Edge–Filtering Algorithm
Example

Remove “dominated” edges:



Edge–Filtering Algorithm
Example

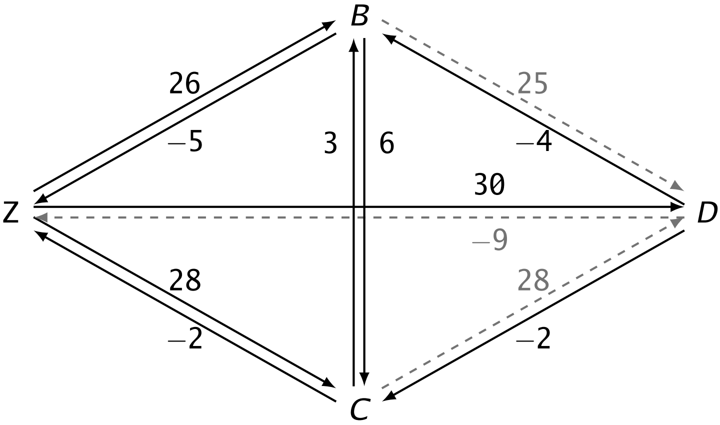
Remove "dominated" edges:



Edge–Filtering Algorithm

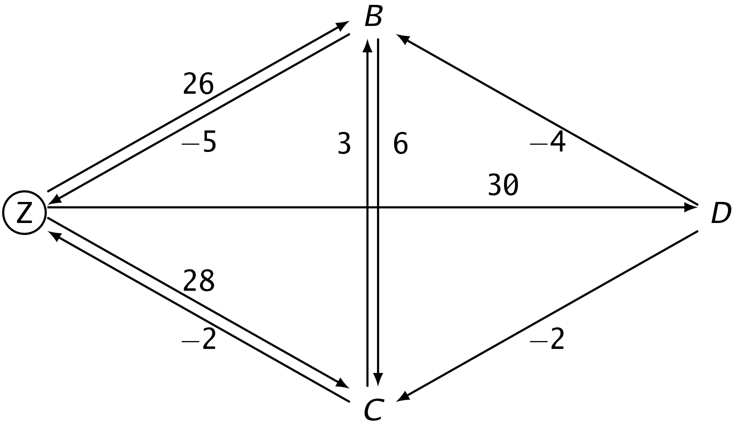
Example

Remove “dominated” edges:



Running the RTE Alg. on the Dispatchable STN

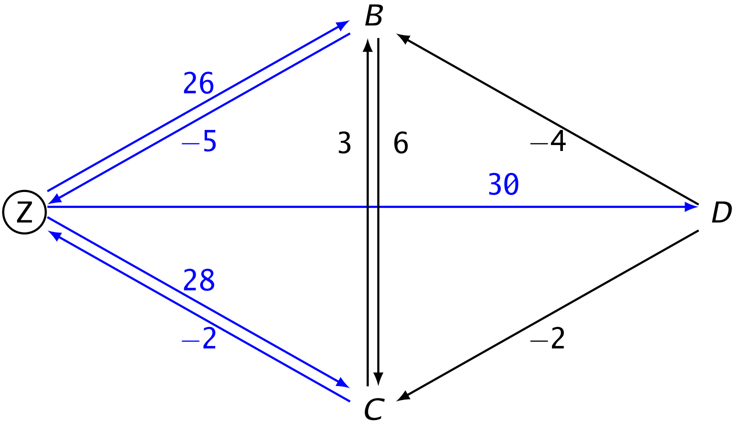
Initially: $t = 0$, $\mathcal{X} = \{\}$, $\mathbf{E} = \{Z\}$.



Remove Z from **E**. Set $Z = 0$. Add Z to $\mathcal{X}$.

Running the RTE Alg. on the Dispatchable STN (ctd.)

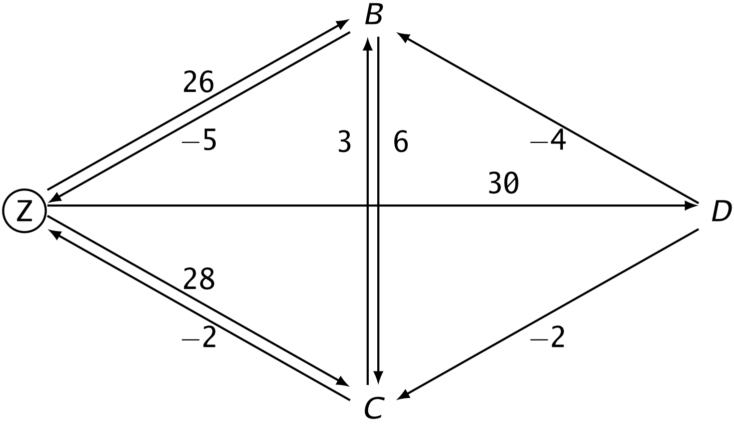
Propagate Z = 0 to neighbors;



$\mathcal{X} = \{Z\}$, $\mathbf{E} = \{B, C\}$; $B \in [5, 26]$, $C \in [2, 28]$, $D \in [0, 30]$.

Running the RTE Alg. on the Dispatchable STN (ctd.)

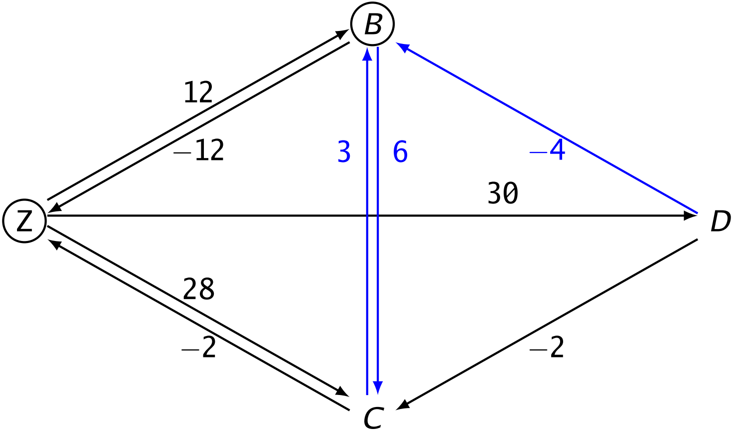
$\mathcal{X} = \{Z\}$, $\mathbf{E} = \{B, C\}$; Bounds: $B \in [5, 26]$, $C \in [2, 28]$.



Let t advance to 12; Pick B from $\mathbf{E}$; Set $B = 12$.

Running the RTE Alg. on the Dispatchable STN (ctd.)

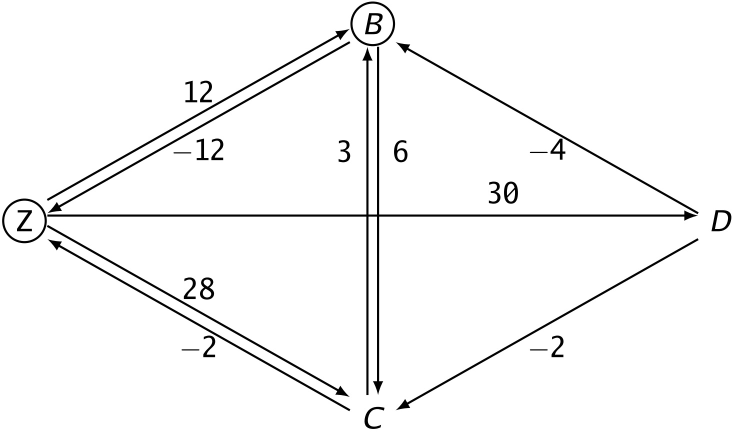
Propagate $B = 12$ to neighbors



$\mathcal{X} = \{Z, B\}$, $t = 12$, $\mathbf{E} = \{C\}$, $C \in [12, 18]$, $D \in [16, 30]$

Running the RTE Alg. on the Dispatchable STN (ctd.)

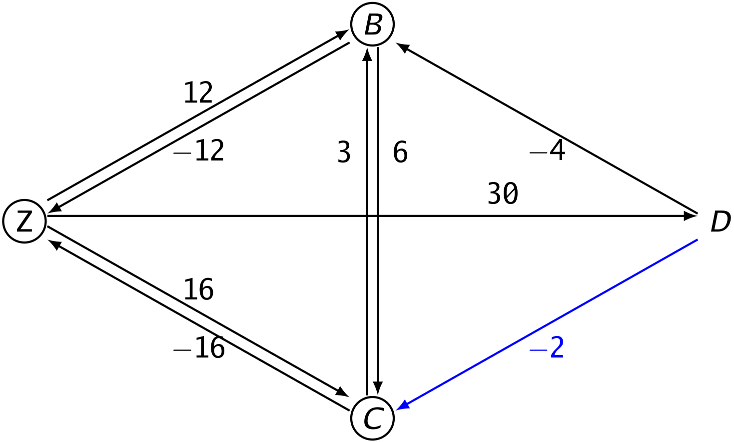
$\mathcal{X} = \{Z, B\}$, $t = 12$, $\mathbf{E} = \{C\}$, $C \in [12, 18], D \in [16, 30]$



Let t advance to 16, pick C from **E**, set $C = 16$.

Running the RTE Alg. on the Dispatchable STN (ctd.)

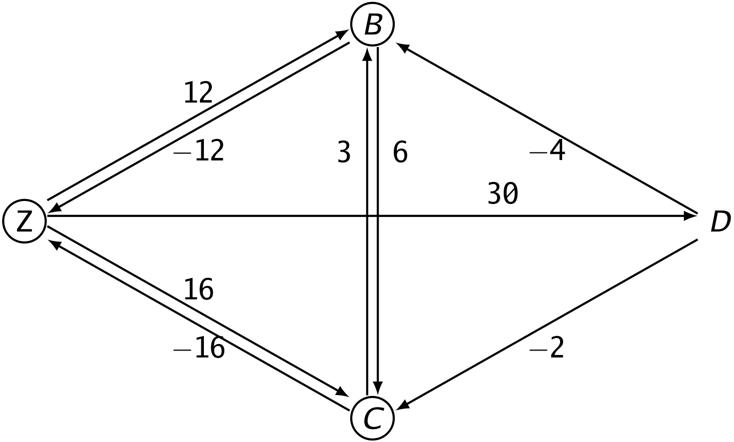
Propagate $C = 16$ to C's only remaining neighbor, D.



$\mathcal{X} = \{Z, B, C\}$, $t = 16$, $\mathbf{E} = \{D\}$, $D \in [18, 30]$

Running the RTE Alg. on the Dispatchable STN (ctd.)

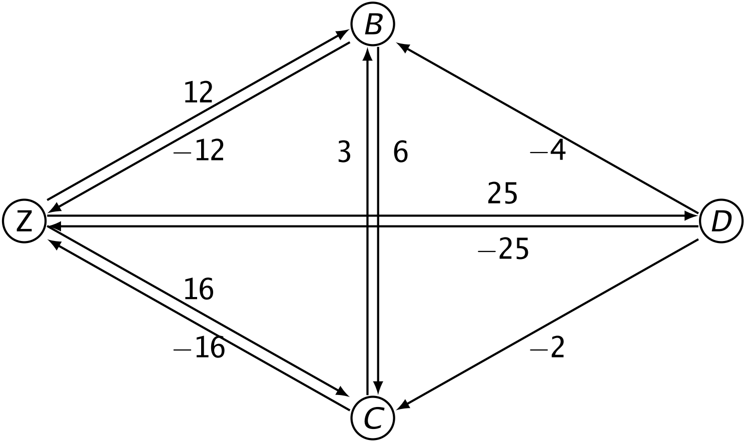
$\mathcal{X} = \{Z, B, C\}$, $t = 16$, $\mathbf{E} = \{D\}$, $D \in [18, 30]$



Let t advance to 25, pick D from $\mathbf{E}$, set $D = 25$.

Running the RTE Alg. on the Dispatchable STN (ctd.)

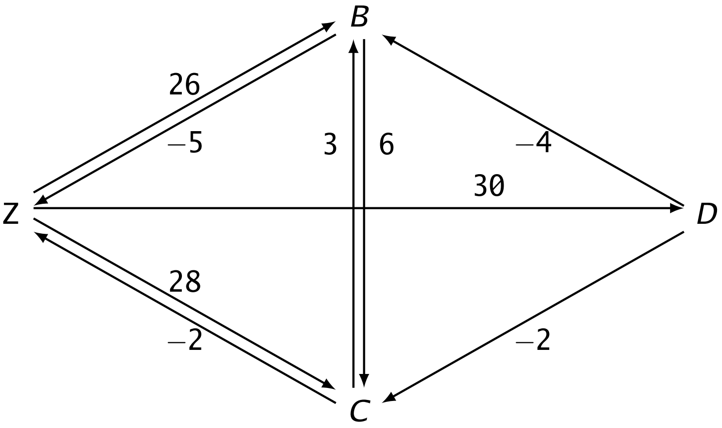
$\mathcal{X} = \{Z, B, C, D\}$, $t = 25$, $\mathbf{E} = \{\}$



Solution: $Z = 0, B = 12, C = 16, D = 25$.

Running the RTE Alg. on the Dispatchable STN (ctd.)

Easy to check that $Z = 0, C = 20, B = 23, D = 28$ can also be generated by the RTE algorithm.



More Efficient STN Dispatchability Algorithm

[Tsamardinos et al., 1998]

More Efficient Dispatchability Algorithm for STNs

[Tsamardinos et al., 1998]

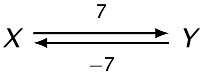
Given any STN graph $\mathcal{G}$:

- First, find and collapse any rigid components in $\mathcal{G}$.
- Second, for each $X \in \mathcal{T}$, accumulate undominated edges by:
  - Constructing predecessor graph $\mathcal{P}_X$ rooted at X
  - Exploring $\mathcal{P}_X$ in reverse post–order
    — while keeping track of certain information.
- Return all accumulated undominated edges.

$\Rightarrow$ Does not require constructing APSP (equiv., computing $\mathcal{D}$)

Collapsing Rigid Components

- X and Y are rigidly connected iff $\mathcal{D}(X,Y) + \mathcal{D}(Y,X) = 0$.

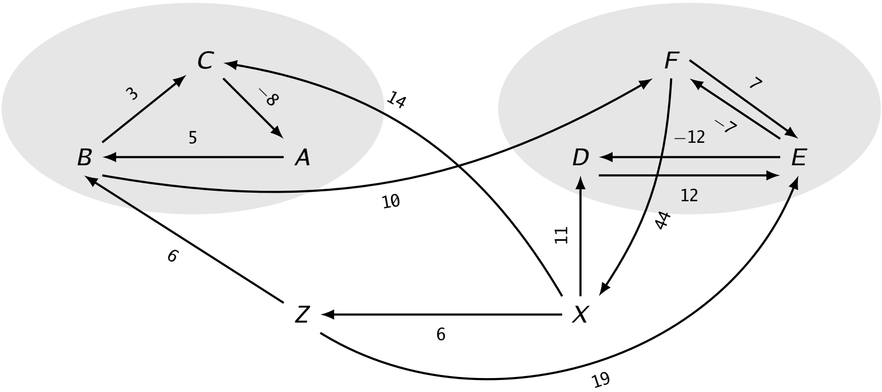
Example: 

(i.e., $Y = X + 7$)

- All time-points along a cycle of length 0 are rigidly connected.
- Being rigidly connected is an equivalence relation.
- A rigid component (RC) contains all of the time-points that are rigidly connected to one another.
- Any time-point in an RC can serve as a representative for the RC.
- Edges incident to time-points in an RC can be redirected to the RC's representative time-point.
- Afterward, each RC can effectively be collapsed to its representative (while preserving the offset information to other time-points in the RC).

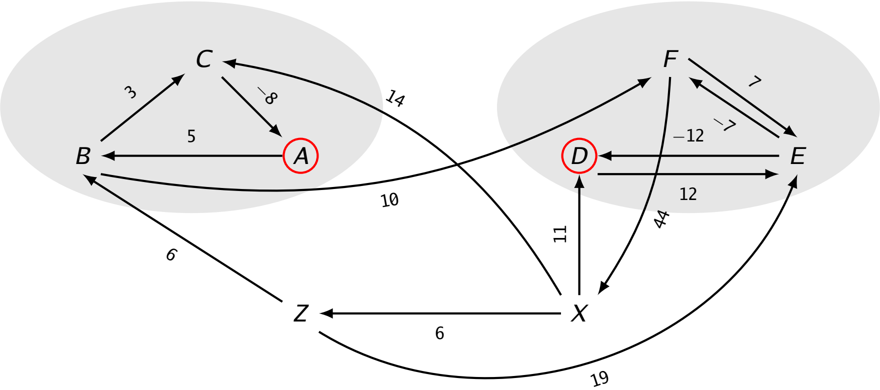
Collapsing Rigid Components

Example



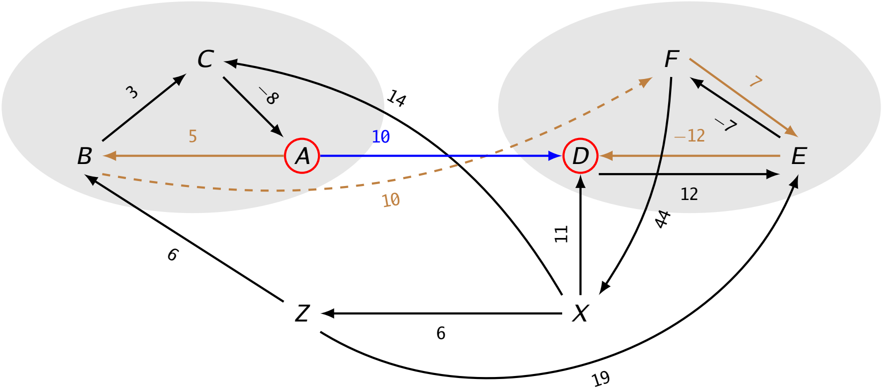
Collapsing Rigid Components

Example



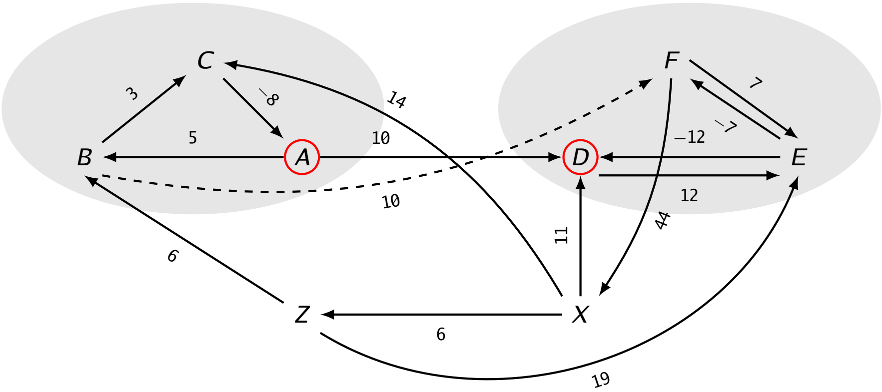
Collapsing Rigid Components

Example



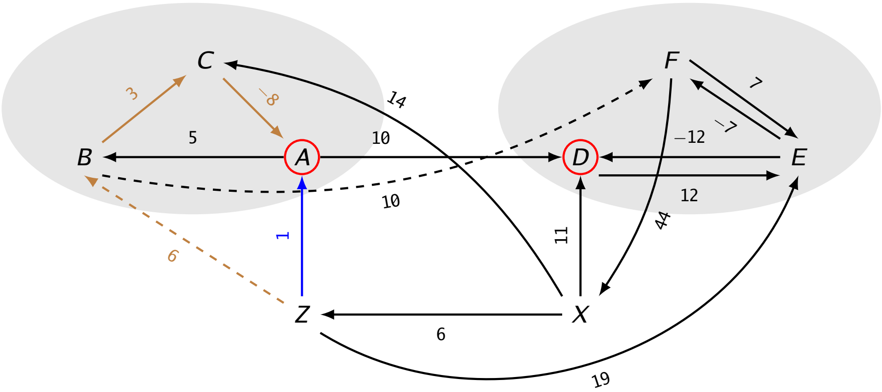
Collapsing Rigid Components

Example



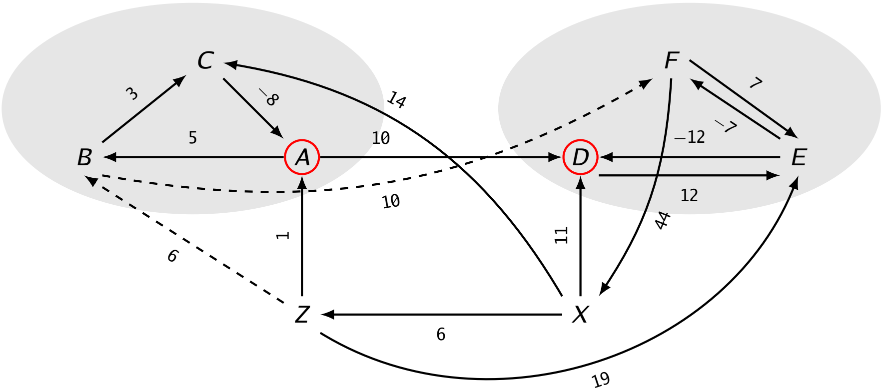
Collapsing Rigid Components

Example



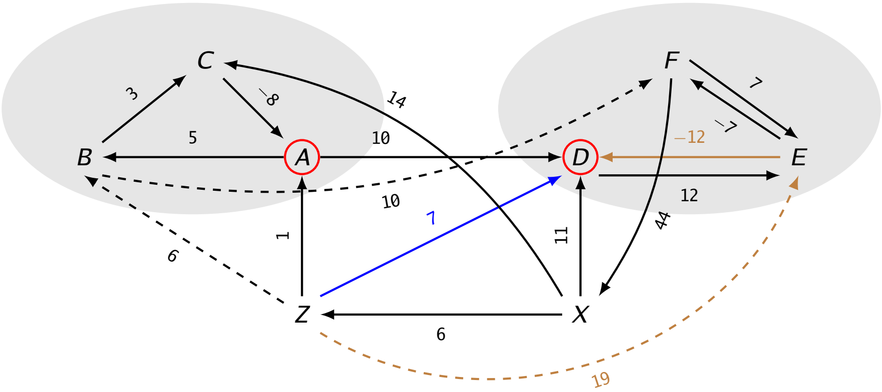
Collapsing Rigid Components

Example



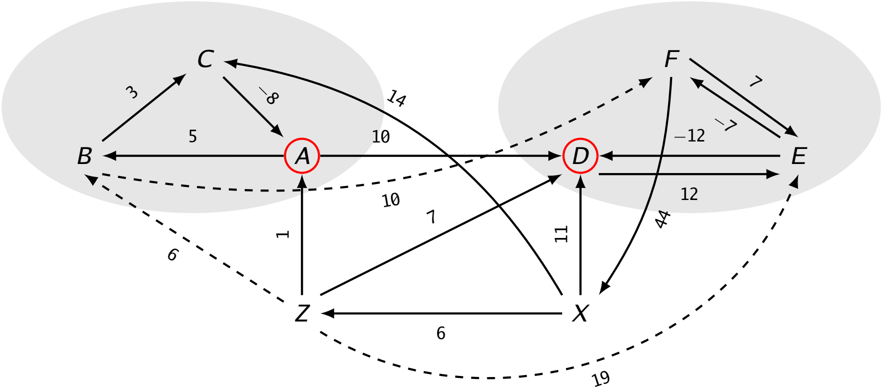
Collapsing Rigid Components

Example



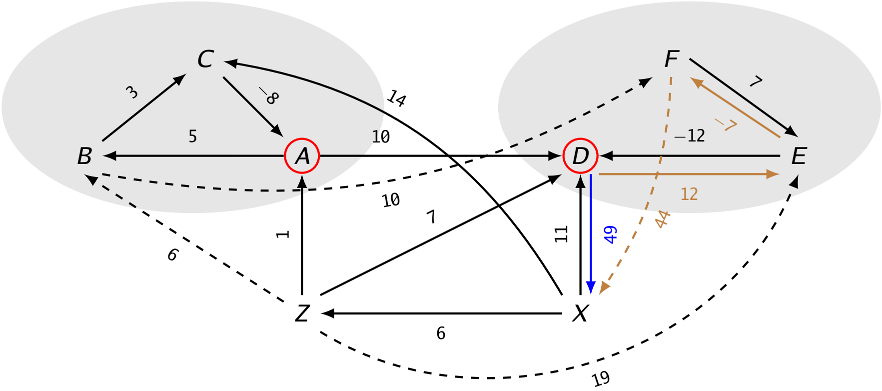
Collapsing Rigid Components

Example



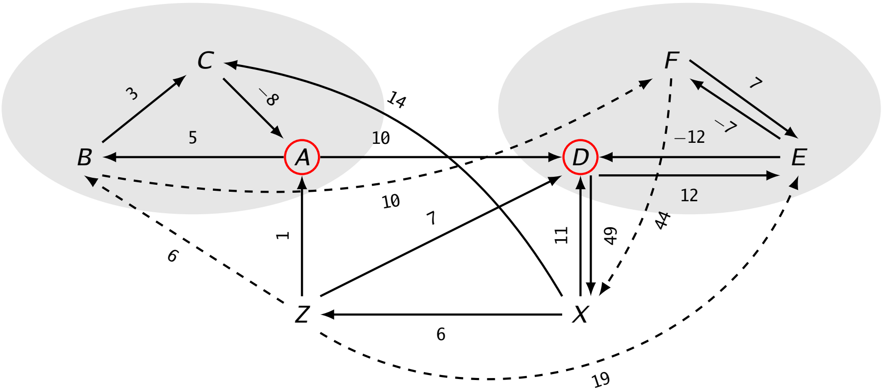
Collapsing Rigid Components

Example



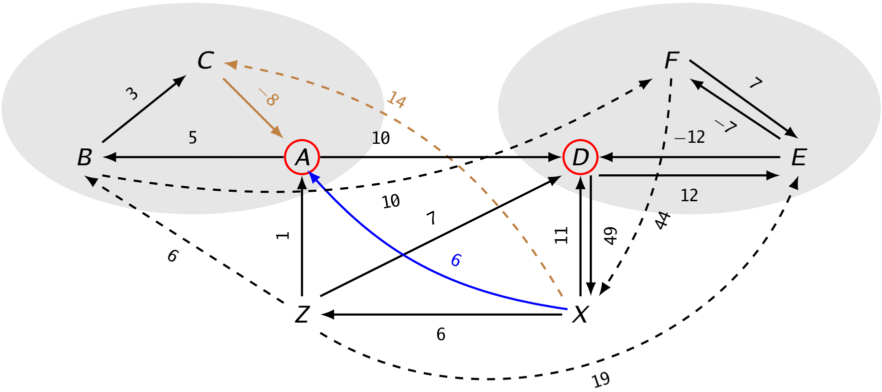
Collapsing Rigid Components

Example



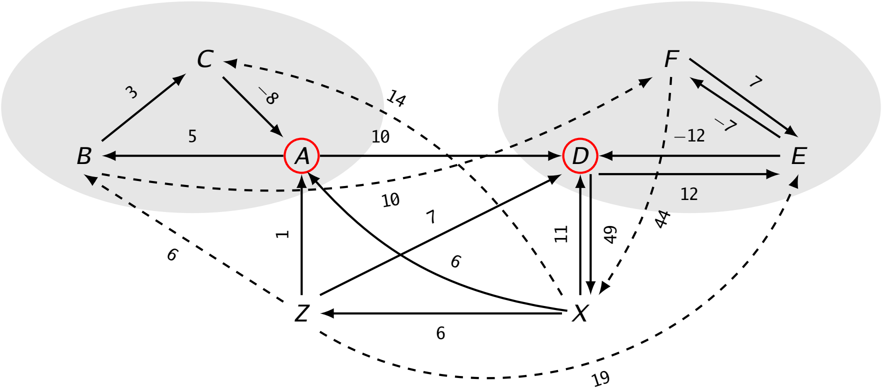
Collapsing Rigid Components

Example



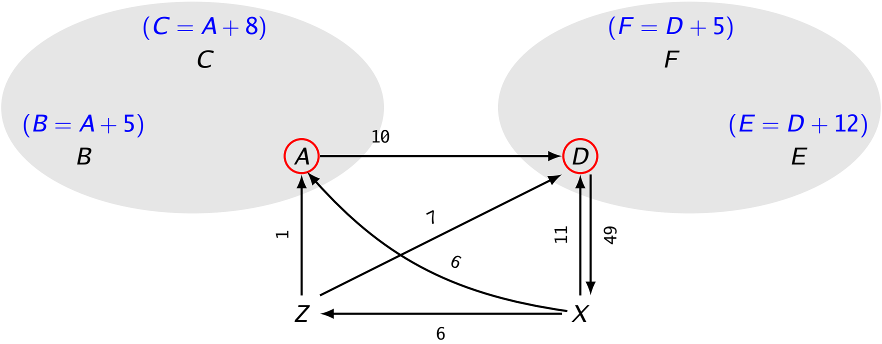
Collapsing Rigid Components

Example



Collapsing Rigid Components

Example

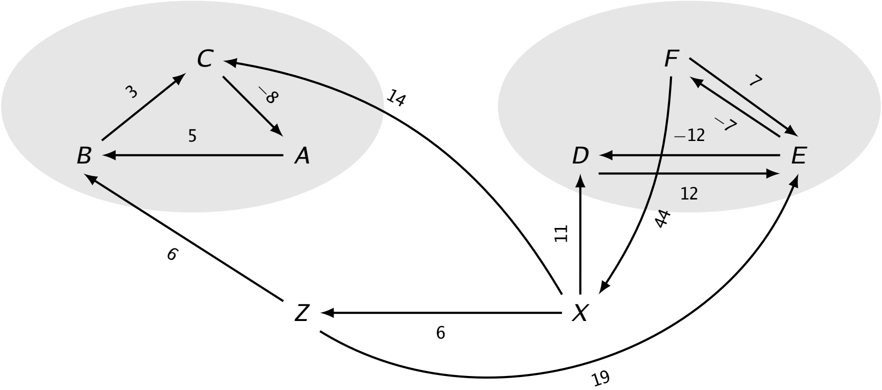


Finding Rigid Components

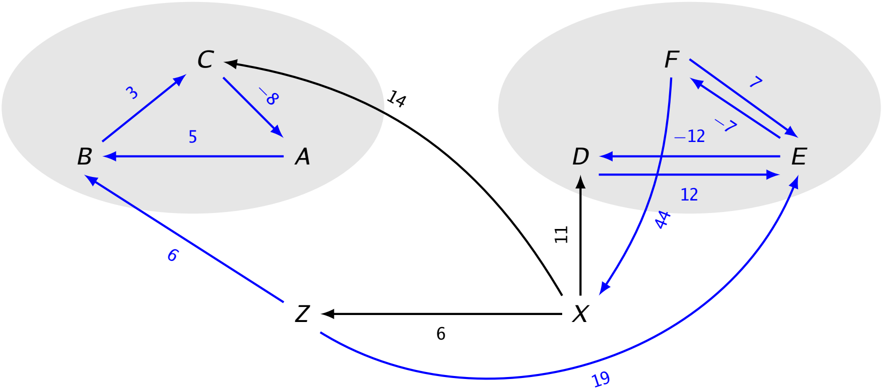
Given any consistent STN graph $\mathcal{G}$, with solution f:

- Construct predecessor graph $\mathcal{P}_Z$, using Z as source node.
  - $\mathcal{P}_Z$ contains all edges in $\mathcal{G}$ lying on shortest paths from Z.
  - $\mathcal{P}_Z$ can be constructed using Dijkstra, with f as a potential function to re-weight edges to be non-negative.
- Rigid components in $\mathcal{G}$ correspond to strongly connected components (SCCs) in $\mathcal{P}_Z$. (Cycles in $\mathcal{P}_Z$ must have length 0.)
- Use Tarjan's SCC algorithm to detect SCCs in $\mathcal{P}_Z$.

Finding Rigid Components



Finding Rigid Components



Predessor graph's edges in blue

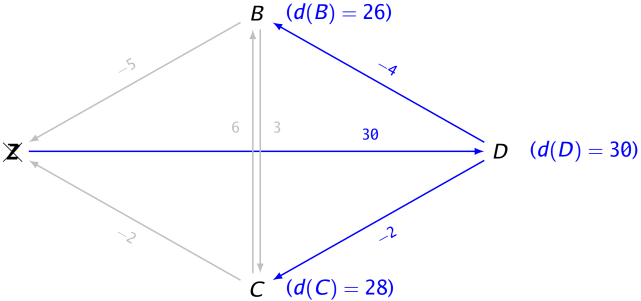
Accumulating Undominated Edges

Without Computing the Distance Matrix

- For each $X \in \mathcal{T}$:
  - Compute pred graph $\mathcal{P}_X$ with X as source, generating distance function $d(Y) =$ distance from X to Y for each $Y \in \mathcal{T}$.
  - Explore $\mathcal{P}_X$ in reverse post–order, along the way updating the following information for each $Y \in \mathcal{T}$:
    - Seen an ancestor W of Y in $\mathcal{P}_X$ with $d(W) < 0$?
    - $\min\{d(W) \mid W \text{ is anc of } Y\}$.
  - When processing Y:
    - If $d(Y) < 0$ and have not seen an ancestor W of Y with $d(W) < 0$, then accumulate (undominated) edge $(X, d(Y), Y)$.
    - If $d(Y) \geq 0$ and $\min\{d(W) \mid W \text{ is anc of } Y\} > d(Y)$, then accumulate (undominated) edge $(X, d(Y), Y)$.
    - In either case, for each outgoing edge (Y, δ, V) in $\mathcal{P}_X$, update info for V regarding its ancestor Y.

Accumulating Undominated Edges

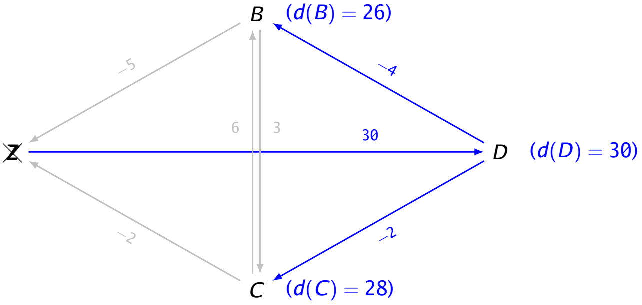
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	*Z*	*B*	*C*	*D*	*Z*	*B*	*C*	*D*
–	–	–	No	No	No	–	∞	∞	∞

Accumulating Undominated Edges

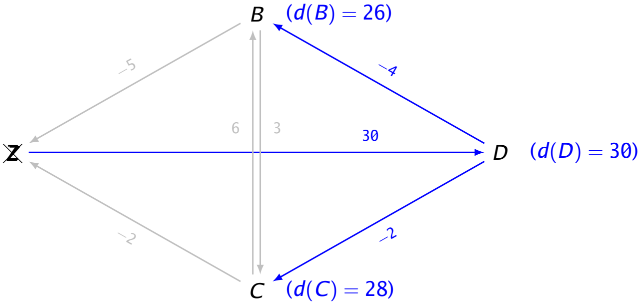
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	–	No	No	No	–	∞	∞	∞
D	$d(D) < minAncDist(D)$								

Accumulating Undominated Edges

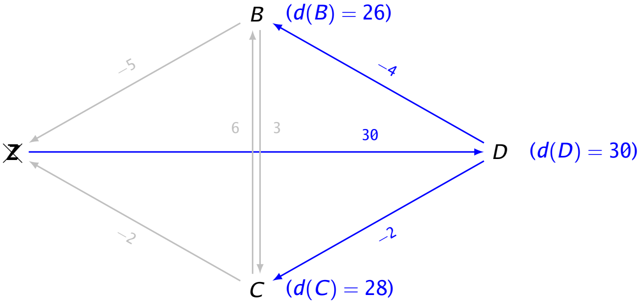
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
-	-	-	No	No	No	-	∞	∞	∞
D	$(Z, 30, D)$	-	No	No	-	-	30	30	-

Accumulating Undominated Edges

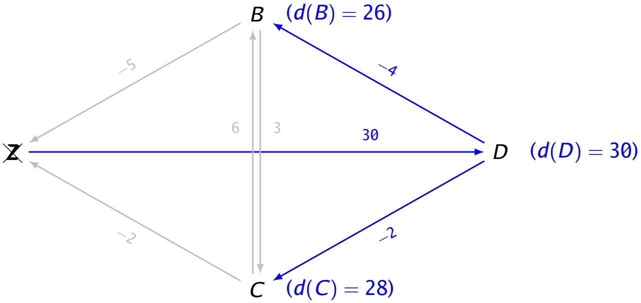
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
-	-	-	No	No	No	-	∞	∞	∞
D	$(Z, 30, D)$	-	No	No	-	-	30	30	-
B	$d(B) < minAncDist(B)$								

Accumulating Undominated Edges

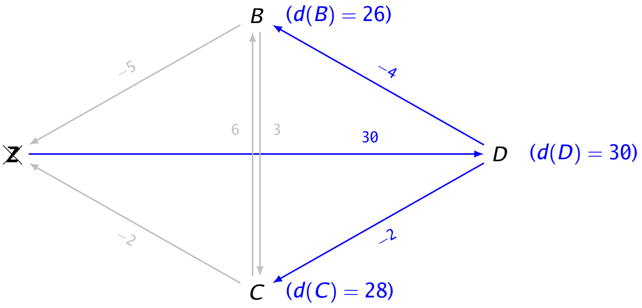
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
-	-	-	No	No	No	-	∞	∞	∞
D	$(Z, 30, D)$	-	No	No	-	-	30	30	-
B	$(Z, 26, B)$	-	-	No	-	-	-	30	-

Accumulating Undominated Edges

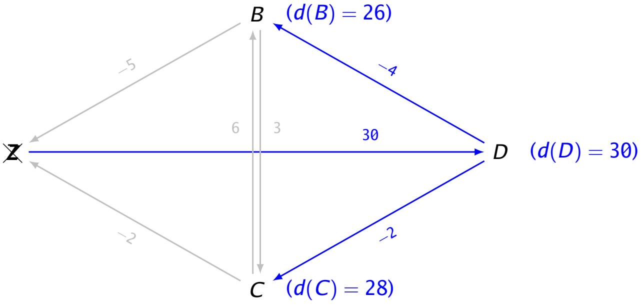
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
-	-	-	No	No	No	-	∞	∞	∞
D	$(Z, 30, D)$	-	No	No	-	-	30	30	-
B	$(Z, 26, B)$	-	-	No	-	-	-	30	-
C	$d(C) < minAncDist(C)$								

Accumulating Undominated Edges

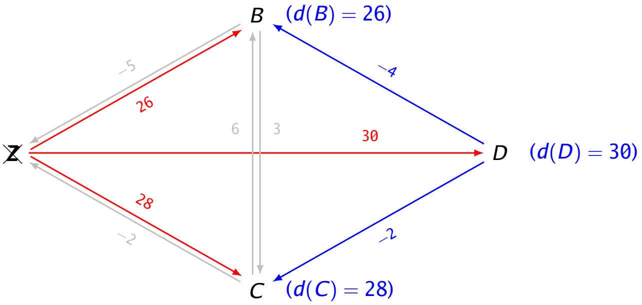
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	–	No	No	No	–	∞	∞	∞
D	$(Z, 30, D)$	–	No	No	–	–	30	30	–
B	$(Z, 26, B)$	–	–	No	–	–	–	30	–
C	$(Z, 28, C)$	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

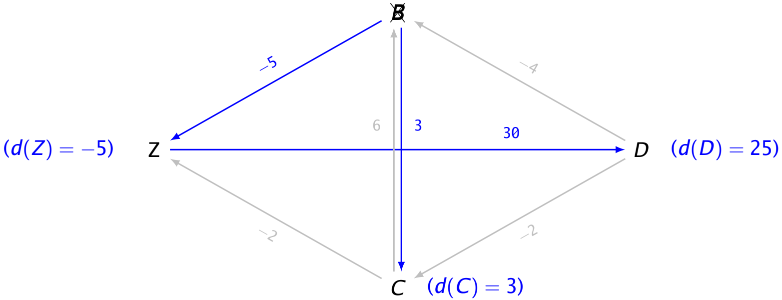
Example: Exploring predecessor graph $\mathcal{P}_Z$ in rev. post-order: $\cancel{Z}, D, B, C$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	–	No	No	No	–	∞	∞	∞
D	$(Z, 30, D)$	–	No	No	–	–	30	30	–
B	$(Z, 26, B)$	–	–	No	–	–	–	30	–
C	$(Z, 28, C)$	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

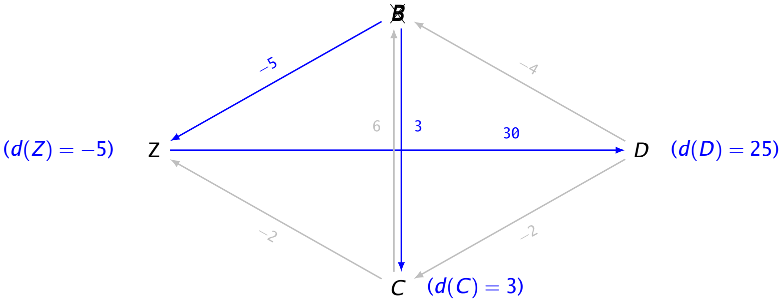
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\cancel{B}, C, Z, D$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞

Accumulating Undominated Edges

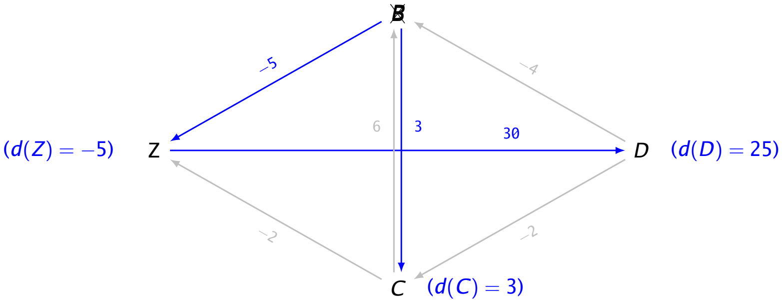
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\cancel{B}, C, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc* Z	B	C	D	*minAncDist* Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$d(C) < \mathit{minAncDist}(C)$								

Accumulating Undominated Edges

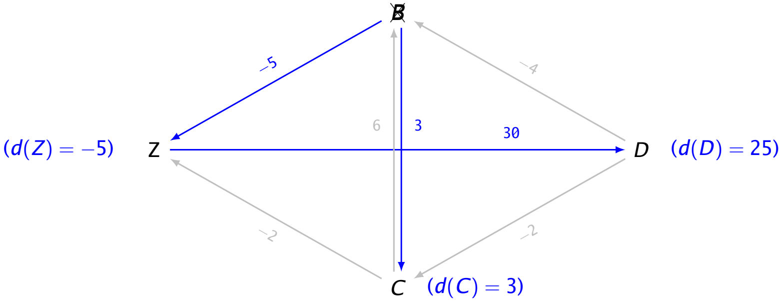
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\xcancel{B}, C, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$(B, 3, C)$	No	–	–	No	∞	–	–	∞

Accumulating Undominated Edges

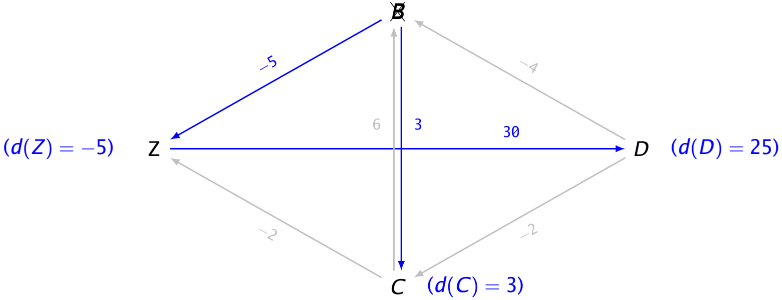
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\cancel{B}, C, Z, D$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$(B, 3, C)$	No	–	–	No	∞	–	–	∞
Z	$hasNegAnc(Z) = \text{No}$								

Accumulating Undominated Edges

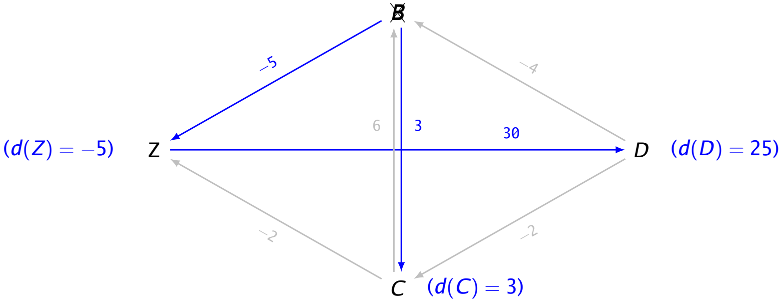
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\cancel{B}, C, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc* Z	B	C	D	*minAncDist* Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$(B, 3, C)$	No	–	–	No	∞	–	–	∞
Z	$(B, -5, Z)$	–	–	–	No	–	–	–	-5

Accumulating Undominated Edges

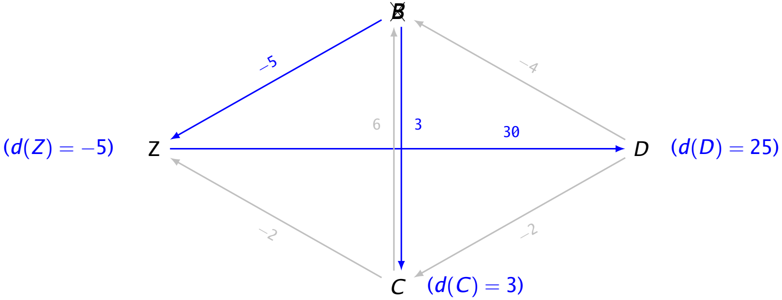
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post–order: $\cancel{B}, C, Z, D$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$(B, 3, C)$	No	–	–	No	∞	–	–	∞
Z	$(B, -5, Z)$	–	–	–	No	–	–	–	-5
D	$d(D) \geq minAncDist(D)$								

Accumulating Undominated Edges

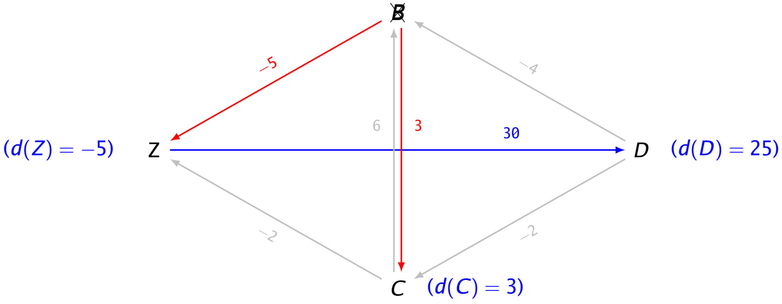
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\cancel{B}, C, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc* Z	B	C	D	*minAncDist* Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$(B, 3, C)$	No	–	–	No	∞	–	–	∞
Z	$(B, -5, Z)$	–	–	–	No	–	–	–	-5
D	–	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

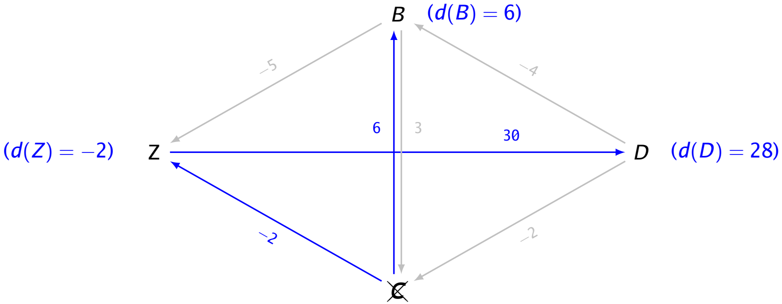
Example: Exploring predecessor graph $\mathcal{P}_B$ in rev. post-order: $\cancel{B}, C, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	–	No	No	∞	–	∞	∞
C	$(B, 3, C)$	No	–	–	No	∞	–	–	∞
Z	$(B, -5, Z)$	–	–	–	No	–	–	–	-5
D	–	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

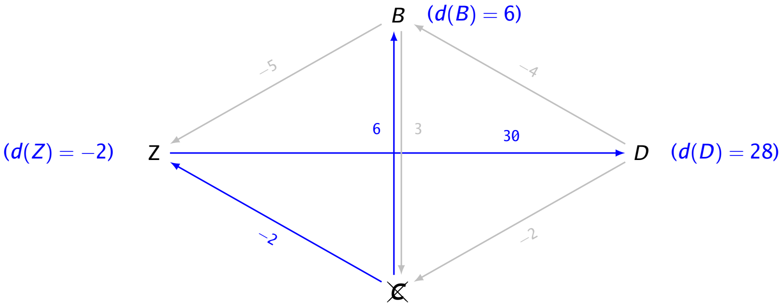
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\cancel{C}, B, Z, D$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞

Accumulating Undominated Edges

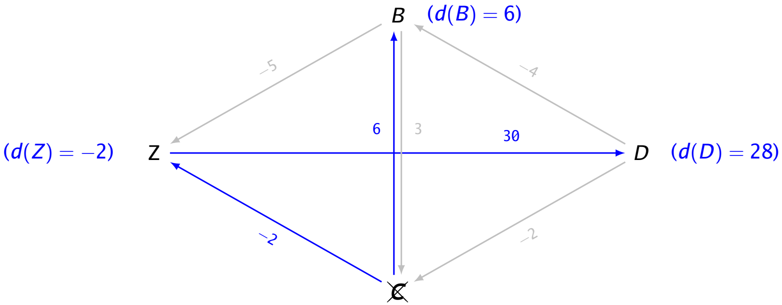
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\xcancel{C}, B, Z, D$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$d(B) < minAncDist(B)$								

Accumulating Undominated Edges

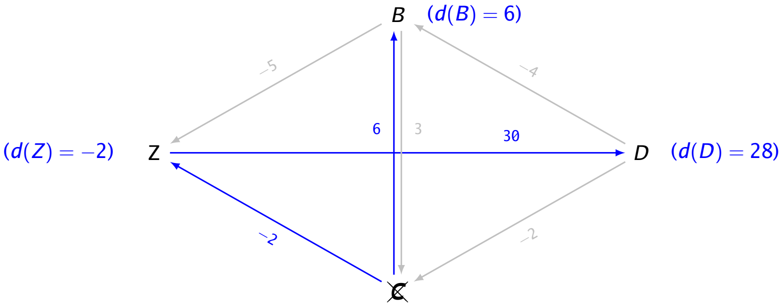
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post–order: $\cancel{C}, B, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$(C, 6, B)$	No	–	–	No	∞	–	–	∞

Accumulating Undominated Edges

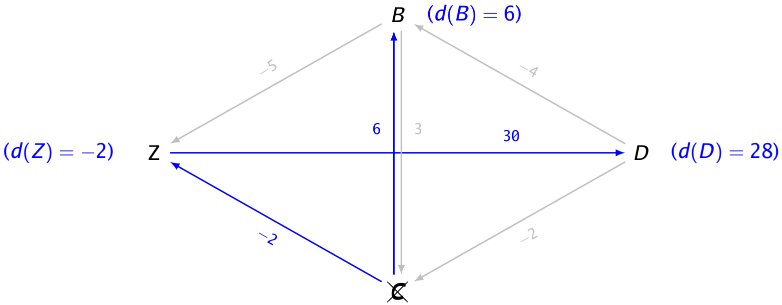
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\xcancel{C}, B, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$(C, 6, B)$	No	–	–	No	∞	–	–	∞
Z	$hasNegAnc(Z) = \text{No}$								

Accumulating Undominated Edges

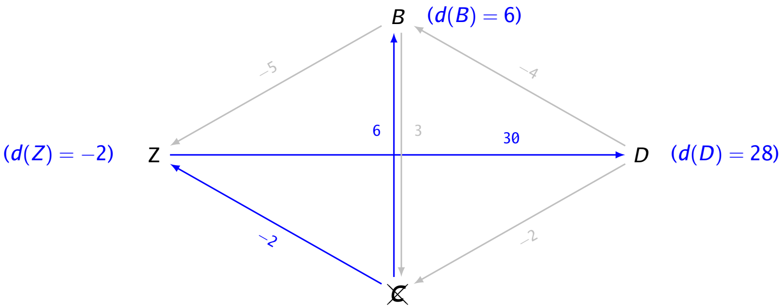
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\cancel{C}, B, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$(C, 6, B)$	No	–	–	No	∞	–	–	∞
Z	$(C, -2, Z)$	–	–	–	Yes	–	–	–	-2

Accumulating Undominated Edges

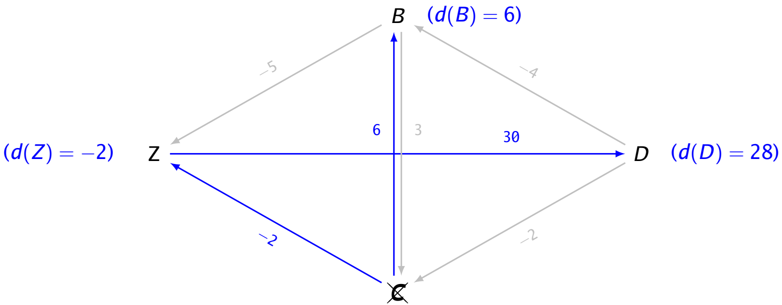
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\xcancel{C}, B, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$(C, 6, B)$	No	–	–	No	∞	–	–	∞
Z	$(C, -2, Z)$	–	–	–	Yes	–	–	–	-2
D	$d(D) \geq minAncDist(D)$								

Accumulating Undominated Edges

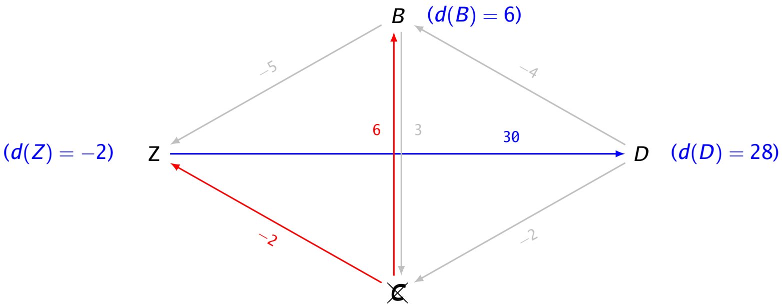
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\cancel{C}, B, Z, D$



Curr. TP	Accum. Edges	*hasNegAnc* Z	B	C	D	*minAncDist* Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$(C, 6, B)$	No	–	–	No	∞	–	–	∞
Z	$(C, -2, Z)$	–	–	–	Yes	–	–	–	-2
D	–	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

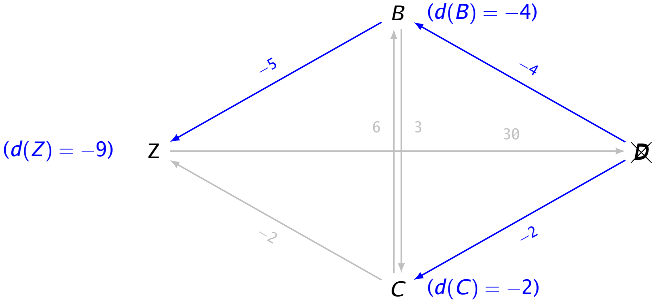
Example: Exploring predecessor graph $\mathcal{P}_C$ in rev. post-order: $\cancel{C}, B, Z, D$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	–	No	∞	∞	–	∞
B	$(C, 6, B)$	No	–	–	No	∞	–	–	∞
Z	$(C, -2, Z)$	–	–	–	Yes	–	–	–	-2
D	–	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

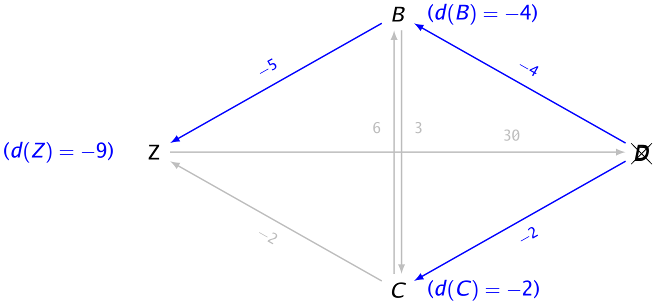
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–

Accumulating Undominated Edges

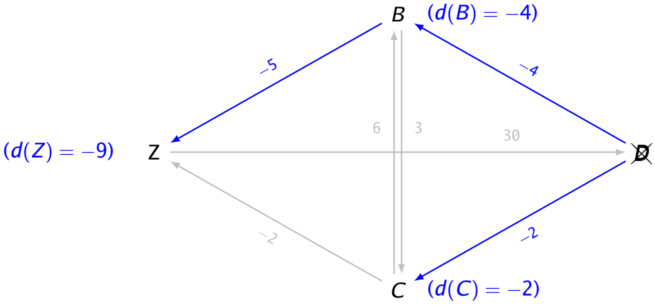
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



Curr. TP	Accum. Edges	hasNegAnc				minAncDist			
		Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–
C	$hasNegAnc(C) = \text{No}$								

Accumulating Undominated Edges

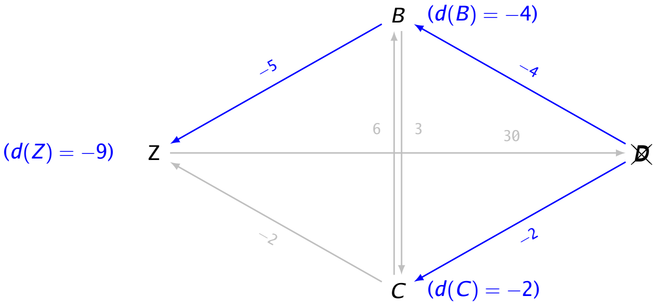
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–
C	$(D, -2, C)$	No	No	–	–	∞	∞	–	–

Accumulating Undominated Edges

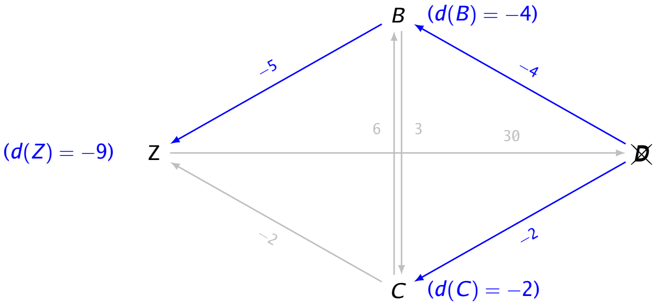
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–
C	$(D, -2, C)$	No	No	–	–	∞	∞	–	–
B	$hasNegAnc(B) = \text{No}$								

Accumulating Undominated Edges

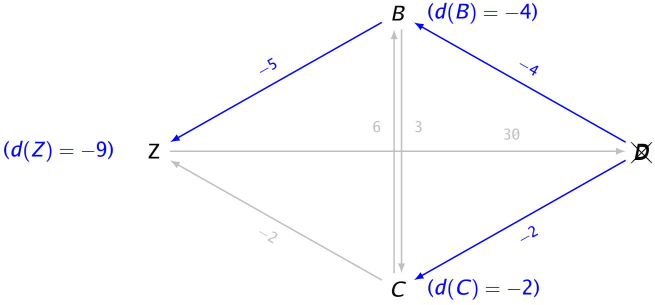
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–
C	$(D, -2, C)$	No	No	–	–	∞	∞	–	–
B	$(D, -4, B)$	Yes	–	–	–	-4	–	–	–

Accumulating Undominated Edges

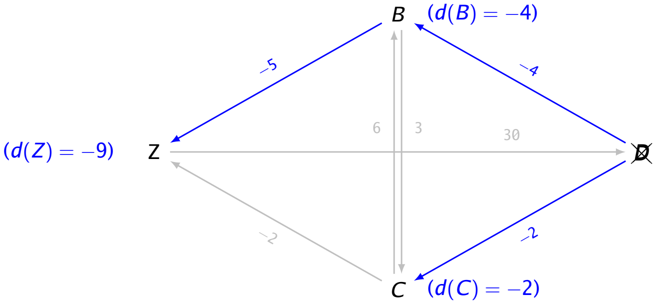
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



		hasNegAnc				*minAncDist*			
Curr. TP	Accum. Edges	Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–
C	$(D, -2, C)$	No	No	–	–	∞	∞	–	–
B	$(D, -4, B)$	Yes	–	–	–	-4	–	–	–
Z	$hasNegAnc(Z) = \text{Yes}$								

Accumulating Undominated Edges

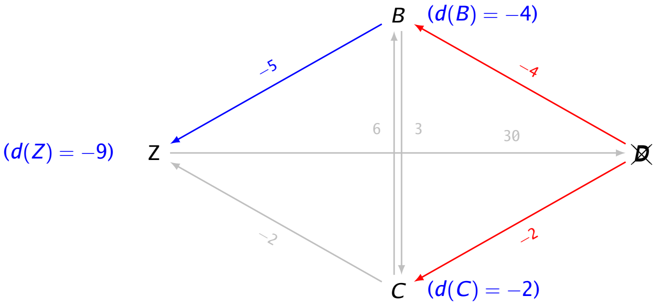
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



Curr. TP	Accum. Edges	*hasNegAnc*				*minAncDist*			
		Z	B	C	D	Z	B	C	D
–	–	No	No	No	–	∞	∞	∞	–
C	$(D, -2, C)$	No	No	–	–	∞	∞	–	–
B	$(D, -4, B)$	Yes	–	–	–	-4	–	–	–
Z	–	–	–	–	–	–	–	–	–

Accumulating Undominated Edges

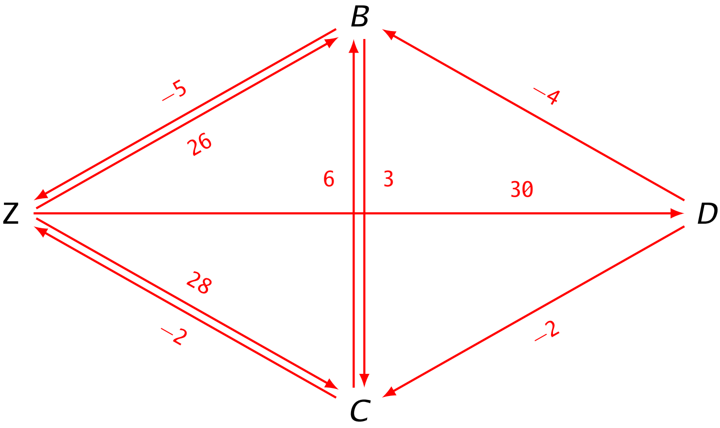
Example: Exploring predecessor graph $\mathcal{P}_D$ in rev. post-order: $\cancel{D}, C, B, Z$



Curr. TP	Accum. Edges	*hasNegAnc* Z	*hasNegAnc* B	*hasNegAnc* C	*hasNegAnc* D	*minAncDist* Z	*minAncDist* B	*minAncDist* C	*minAncDist* D
–	–	No	No	No	–	∞	∞	∞	–
C	$(D, -2, C)$	No	No	–	–	∞	∞	–	–
B	$(D, -4, B)$	Yes	–	–	–	-4	–	–	–
Z	–	–	–	–	–	–	–	–	–

Accumulation of Undominated Edges

Example: Total Accumulated Edges



Book on STNs — Coming soon!

Simple Temporal Networks with Uncertainty (STNUs)

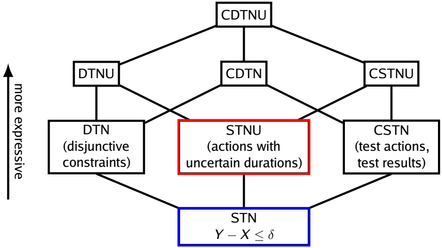
Simple Temporal Networks with Uncertainty

Motivation

- You may control when an action starts, but not how long it lasts:
  — taxi ride, bus ride, baseball game, medical procedure.
- Although their durations may be uncertain, they are often within known bounds.
- Such actions can be represented by *contingent links* in a temporal network . . .

STN with Uncertainty (STNU)

Definition [Morris et al., 2001]



An STNU is a triple, $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$, where:

- $(\mathcal{T}, \mathcal{C})$ is an STN
- $\mathcal{L}$ is a set of contingent links, each of the form (A, x, y, C):
  - A is the activation time–point.
  - C is the contingent time–point.
  - Duration bounded: $C - A \in [x, y]$ but *uncontrollable*

Notation: $n = |\mathcal{T}|,\ m = |\mathcal{C}|,\ k = |\mathcal{L}|$.

STNU Graph

[Morris and Muscettola, 2005]

Each STNU has a graphical form where:

- Nodes and "ordinary" edges as in an STN graph

$$Y - X \in [3, 7] \quad \Longleftrightarrow \quad X \overset{7}{\underset{-3}{\rightleftarrows}} Y$$

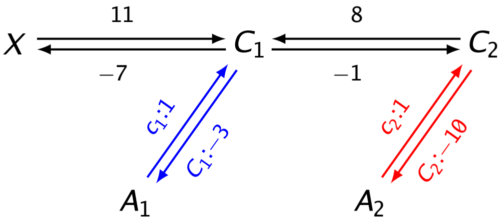
- Contingent Links $\Longleftrightarrow$ Labeled Edges

$$(A, 3, 7, C) \quad \Longleftrightarrow \quad A \overset{c:3}{\underset{C:-7}{\rightleftarrows}} C$$

- The lower-case (LC) edge, $A \xrightarrow{c:3} C$, represents the uncontrollable possibility that $C - A$ might equal 3.

- The upper-case (UC) edge, $A \xleftarrow{C:-7} C$, represents the uncontrollable possibility that $C - A$ might equal 7.

Sample STNU Graph



- Contingent links: $C_1 - A_1 \in [1, 3]$ and $C_2 - A_2 \in [1, 10]$
- Agent only controls execution of A_1, A_2 and X.

STNU Notation

[Hunsberger, 2015b]

For a given STNU graph $\mathcal{G}$:

- LO edges: the lower-case or ordinary edges
- OU edges: the ordinary or upper-case edges

STNU Notation

[Hunsberger, 2015b]

For a given STNU graph $\mathcal{G}$:

- LO edges: the lower–case or ordinary edges
- OU edges: the ordinary or upper–case edges
- LO graph: STNU graph comprising the LO edges
- OU graph: STNU graph comprising the OU edges

STNU Notation

[Hunsberger, 2015b]

For a given STNU graph $\mathcal{G}$:

- LO edges: the lower-case or ordinary edges
- OU edges: the ordinary or upper-case edges
- LO graph: STNU graph comprising the LO edges
- OU graph: STNU graph comprising the OU edges

Ignoring any alphabetic labels, the LO graph and OU graphs may be viewed as STNs.

Dynamic Controllability (DC)

[Morris et al., 2001] [Hunsberger, 2009]

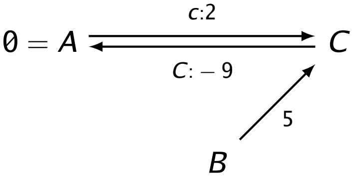
An STNU is *dynamically controllable* (DC) if:

- there exists a *dynamic strategy* . . .
- for executing the *non–contingent* time–points . . .
- such that *all* of the constraints will be satisfied . . .
- *no matter how the contingent durations turn out.*

A dynamic strategy can *react* to contingent executions.

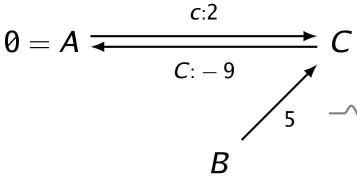
STN with Uncertainty

STNU Example #1



STN with Uncertainty

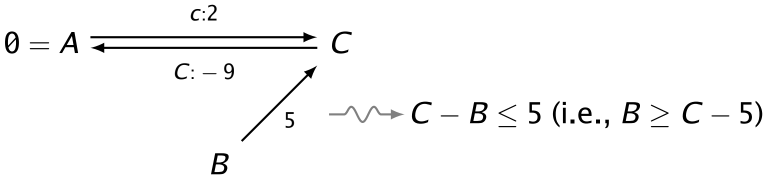
STNU Example #1



$\rightsquigarrow C - B \leq 5$ (i.e., $B \geq C - 5$)

STN with Uncertainty

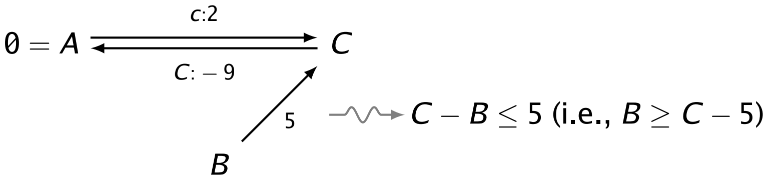
STNU Example #1



- If C executes at time 9, then $B \geq C - 5$ iff $B \geq 4$.

STN with Uncertainty

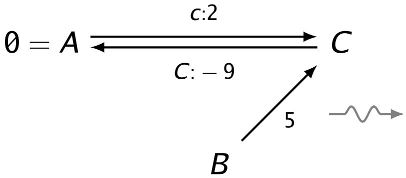
STNU Example #1



- If C executes at time 9, then $B \geq C - 5$ iff $B \geq 4$.
- Strategy cannot know in advance when C will execute, so B must wait until time 4

STN with Uncertainty

STNU Example #1

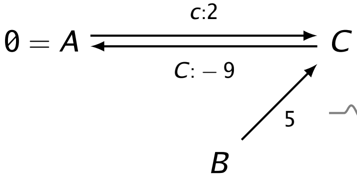


$C - B \leq 5$ (i.e., $B \geq C - 5$)

- If C executes at time 9, then $B \geq C - 5 \;\text{ iff }\; B \geq 4$.
- Strategy cannot know in advance when C will execute, so B must wait until time 4
  —unless C executes early (e.g., at time 2).
  In that case, B could then execute immediately.

STN with Uncertainty

STNU Example #1



$\rightsquigarrow C - B \leq 5$ (i.e., $B \geq C - 5$)

- If C executes at time 9, then $B \geq C - 5$ iff $B \geq 4$.
- Strategy cannot know in advance when C will execute, so B must wait until time 4

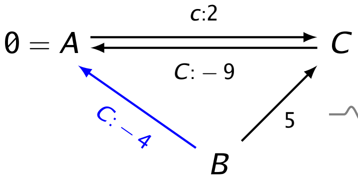
  —unless C executes early (e.g., at time 2).

  In that case, B could then execute immediately.
- Conclusion: To ensure that $C - B \leq 5$ is satisfied:

  As long as C unexecuted, B must wait until time 4.

STN with Uncertainty

STNU Example #1

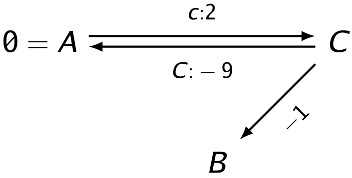


$\rightsquigarrow C - B \leq 5$ (i.e., $B \geq C - 5$)

- If C executes at time 9, then $B \geq C - 5$ iff $B \geq 4$.
- Strategy cannot know in advance when C will execute, so B must wait until time 4
  —unless C executes early (e.g., at time 2).
  In that case, B could then execute immediately.
- Conclusion: To ensure that $C - B \leq 5$ is satisfied:
  As long as C unexecuted, B must wait until time 4.
- $A \xleftarrow{C:-4} B$ is an example of a *wait* constraint.

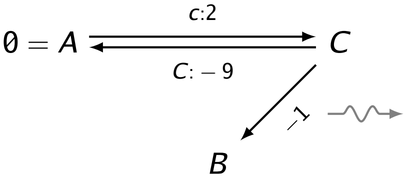
STN with Uncertainty

STNU Example #2



STN with Uncertainty

STNU Example #2

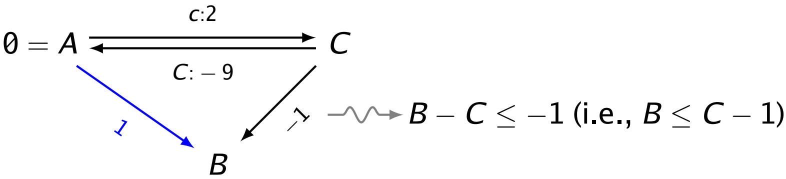


$B - C \leq -1$ (i.e., $B \leq C - 1$)

- If C executes at time 2, then $B \leq C - 1$ iff $B \leq 1$.

STN with Uncertainty

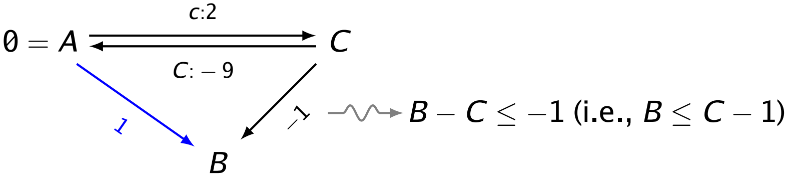
STNU Example #2



- If C executes at time 2, then $B \leq C - 1 \text{ iff } B \leq 1$.
- Strategy cannot know in advance whether C will execute early, so B must execute before time 1.

STN with Uncertainty

STNU Example #2



$\rightsquigarrow B - C \leq -1$ (i.e., $B \leq C - 1$)

- If C executes at time 2, then $B \leq C - 1$ iff $B \leq 1$.
- Strategy cannot know in advance whether C will execute early, so B must execute before time 1. No exceptions!

Computational Problems associated with STNUs

- DC Checking: How to check whether an STNU is DC?
- Dispatchability: How to efficiently execute an STNU in real time?

DC–Checking Algorithms for STNUs

Recent Approaches to DC–Checking for STNUs

- Based on constraint–propagation/edge–generation rules
- Focus on reducing away/bypassing “problem" edges
- Some use potential functions (as in Johnson’s algorithm) to guide exploration of shortest paths in related STN graphs.

Authors	Morris [2006]	Morris [2014]	Cairo et al. [2018]
Problem Edges	LC edges	Neg. OU edges	UC edges
Prop. Along	OU edges	LO edges	LO edges
Prop. Rules	MM05*	MM05*	RUL^{-}
Prop. Dir’n.	Fwd	Bkwd	Bkwd
Pot. Func.?	Yes	No	Yes
Complexity	$O(n^4)$	$O(n^3)$	$O(mn + k^2 n + kn \log n)$

* [Morris and Muscettola, 2005]

The MM05 Propagation Rules

[Morris and Muscettola, 2005]

Rule	Graphical Representation	Applicability Conditions
No Case (NC)	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, dashed, bend right, "v+w"'] & Y \arrow[r, "w"] & W \end{tikzcd}$	(none)
Upper Case (UC)	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, dashed, bend right, "C{:}v-w"'] & Y \arrow[r, "C{:}-w"] & A \end{tikzcd}$	(none)
Lower Case (LC)	$\begin{tikzcd} A \arrow[r, "c{:}x"] \arrow[rr, dashed, bend right, "x+w"'] & C \arrow[r, "w"] & U \end{tikzcd}$	$w < 0$
Cross Case (CC)	$\begin{tikzcd} A \arrow[r, "c{:}x"] \arrow[rr, dashed, bend right, "C_2{:}x+w"'] & C \arrow[r, "C_2{:}w"] & A_2 \end{tikzcd}$	$C_2 \not\equiv C$, $w < 0$
Label Removal (LR)	$\begin{tikzcd} X \arrow[r, "C{:}w"] \arrow[r, dashed, shift right, "w"'] & A \arrow[r, "c{:}x"] & C \end{tikzcd}$	$w \geq -x$

The RUL$^-$ Propagation Rules

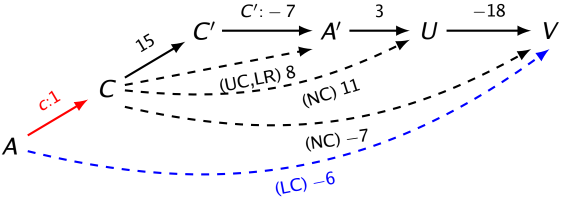
[Cairo et al., 2018]

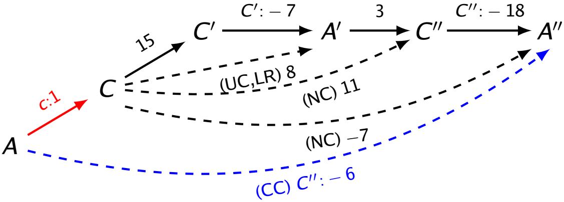
Rule	Graphical representation	Applicability Conditions
R$^-$	$\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dashed, bend right, "v+w"'] & Q \arrow[r, "w"] & {\color{red}C} \end{tikzcd}$	$(A, x, y, C) \in \mathcal{L},\ {\color{blue}w < y - x},\ Q \in \mathcal{T}_X$
L$^-$	$\begin{tikzcd} A' \arrow[r, "c' : x'"] \arrow[rr, dashed, bend right, "x+w"'] & C' \arrow[r, "w"] & {\color{red}C} \end{tikzcd}$	$(A, x, y, C) \in \mathcal{L},\ {\color{blue}w < y - x},\ C' \not\equiv C$
U$^-$	$\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dashed, bend right, "\max\{v-y, -x\}"'] & {\color{red}C} \arrow[r, red, "{\color{red}C\colon -y}"] & {\color{red}A} \end{tikzcd}$	$(A, x, y, C) \in \mathcal{L}$

Morris' $O(n^4)$–time DC–Checking Algorithm

[Morris, 2006]

Starting from LC edges, propagate forward along OU edges, using MM05 rules, aiming to generate bypass edges.

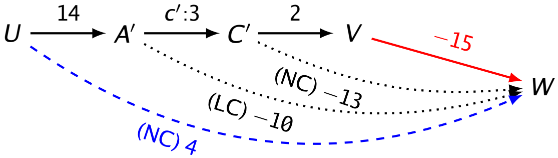


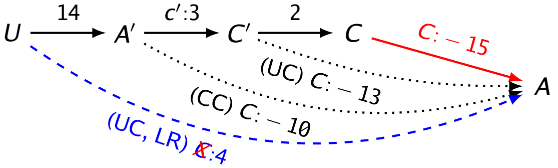


Morris' $O(n^3)$–time DC–Checking Algorithm

[Morris, 2014]

Starting from negative OU edges, propagate backward along non–negative LO edges, also using MM05 rules, aiming to generate OU bypass edges.



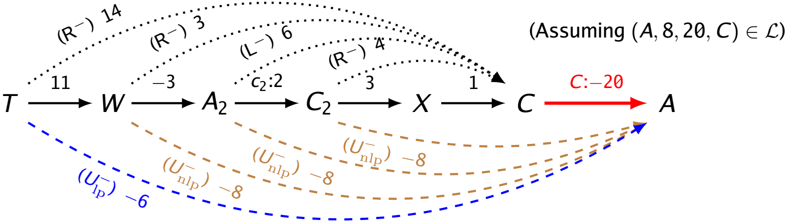


Same idea applies to multiple negative edges incoming to a single node.

The RUL$^-$ DC–Checking Algorithm

An $O(mn + k^2 n + kn \log n)$–time algorithm, [Cairo et al., 2018]

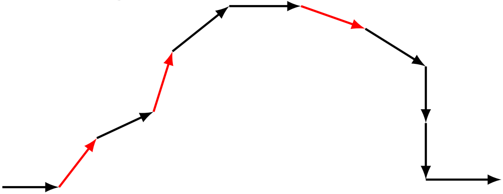
Starting from UC edges, propagate backward along LO edges, using the RUL$^-$ rules, aiming to generate ordinary bypass edges.



Morris' 2006 $O(n^4)$–time DC–checking algorithm

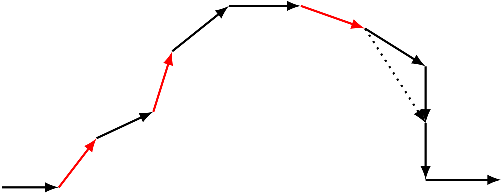
Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
- 
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

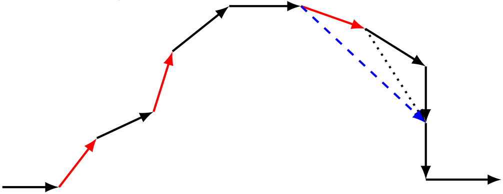
Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
- 
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
-

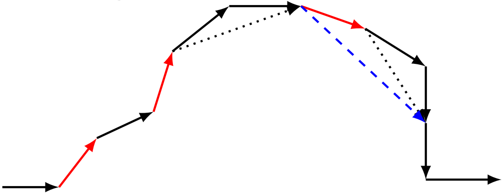


- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

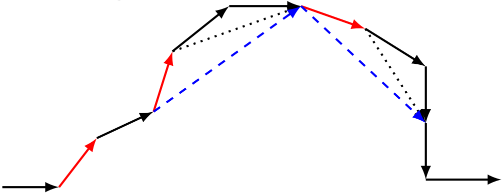
- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.



-
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

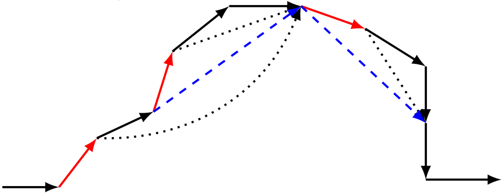
Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
- 
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.

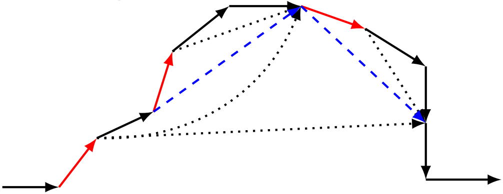


-
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

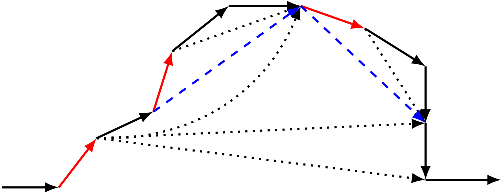
- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
-



- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

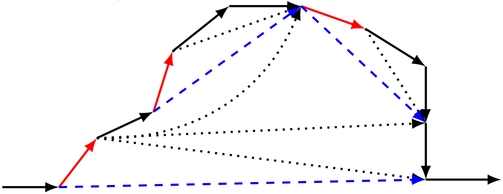
Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
- 
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

Morris' $O(n^4)$–time DC–Checking Algorithm

Overview

- Focus: Generate OU edges that bypass LC edges.
- From each LC edge, $A \xrightarrow{c:x} C$, propagate forward along OU edges, looking for opportunities to generate new OU edges that bypass the LC edge.
- 
- To guide exploration of shortest paths in the OU graph, use a potential function generated by Bellman–Ford.
- If all of the LC edges in a given path $\mathcal{P}$ can be bypassed by OU edges, then $\mathcal{P}$ is called semi–reducible.
- Input STNU is DC iff no semi–reducible negative cycles.

The MM05 Propagation Rules

[Morris and Muscettola, 2005]

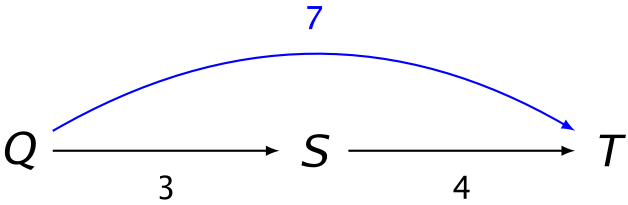
Rule	**Graphical Representation**	**Applicability Conditions**
No Case (NC)	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, dashed, bend right, "v+w"'] & Y \arrow[r, "w"] & W \end{tikzcd}$	(none)
Upper Case (UC)	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, dashed, bend right, "C{:}v-w"'] & Y \arrow[r, "C{:}-w"] & A \end{tikzcd}$	(none)
Lower Case (LC)	$\begin{tikzcd} A \arrow[r, "c{:}x"] \arrow[rr, dashed, bend right, "x+w"'] & C \arrow[r, "w"] & U \end{tikzcd}$	$w < 0$
Cross Case (CC)	$\begin{tikzcd} A \arrow[r, "c{:}x"] \arrow[rr, dashed, bend right, "C_2{:}x+w"'] & C \arrow[r, "C_2{:}w"] & A_2 \end{tikzcd}$	$C_2 \not\equiv C$, $w < 0$
Label Removal (LR)	$\begin{tikzcd} X \arrow[r, "C{:}w"] \arrow[r, dashed, shift right, "w"'] & A \arrow[r, "c{:}x"] & C \end{tikzcd}$	$w \geq -x$

The MM05 Propagation Rules

The No-Case (NC) Rule

The MM05 Propagation Rules

The No-Case (NC) Rule

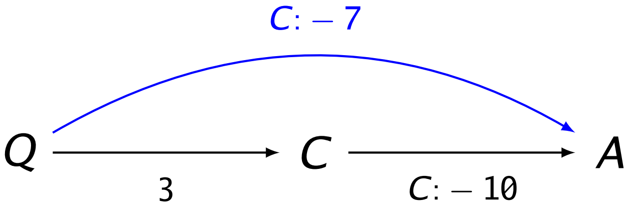


The MM05 Propagation Rules

The Upper–Case (UC) Rule

The MM05 Propagation Rules

The Upper-Case (UC) Rule



The generated edge is a wait constraint:

As long as *C* remains unexecuted, *B* must wait until 7 after *A*.

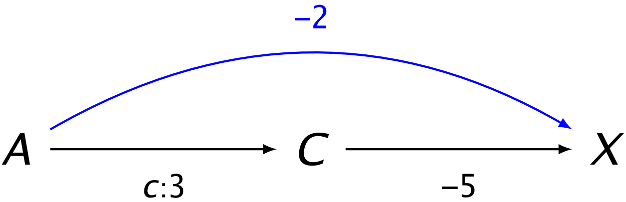
The MM05 Propgation Rules

The Lower-Case (LC) Rule

(Applies since $-5 < 0$)

The MM05 Propgation Rules

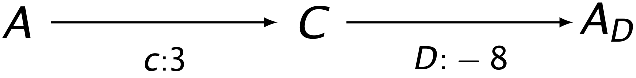
The Lower-Case (LC) Rule



(Applies since $-5 < 0$)

The MM05 Propagation Rules

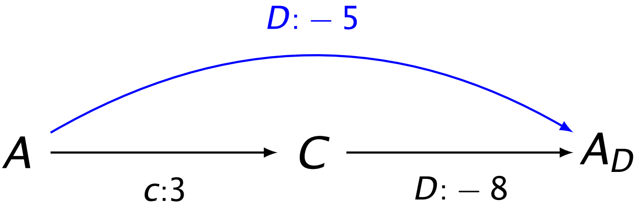
The Cross-Case (CC) Rule



(Applies since $-8 < 0$ and $C \neq D$)

The MM05 Propagation Rules

The Cross-Case (CC) Rule



(Applies since $-8 < 0$ and $C \not\equiv D$)

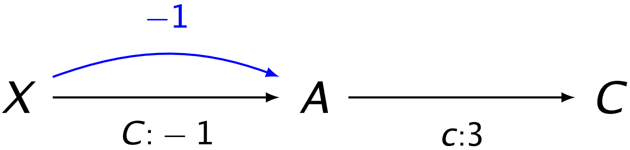
Originally:	C must wait until 8 after A_D unless D executes early.
But:	C is contingent! Therefore, to ensure that C "waits":
Generated:	A must wait until 5 after A_D unless D executes early.

The MM05 Propagation Rules

The Label-Removal (LR) Rule

The MM05 Propagation Rules

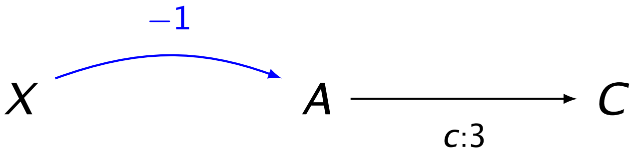
The Label-Removal (LR) Rule



(Applies since $-1 \geq -3$)

The MM05 Propagation Rules

The Label-Removal (LR) Rule



X must wait at least 1 after A unless C executes early.

— But C cannot execute before time 1!

So X must wait no matter what!

The MM05 Propgation Rules

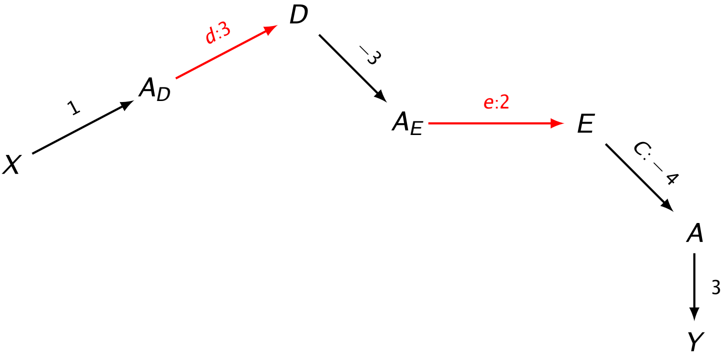
Important Property

All of the MM05 edge-generation rules are length preserving!

Morris' $O(n^4)$–time DC–checking Algorithm

Sample Semi–Reducible Paths [Morris, 2006]

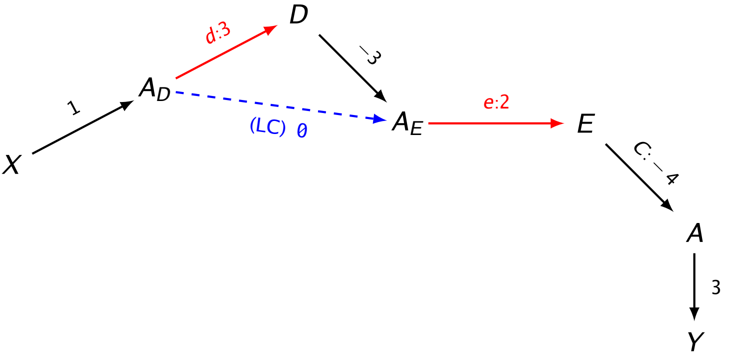
A path is *semi–reducible* if it can be transformed into a path with no *lower–case* edges.



Morris' $O(n^4)$–time DC–checking Algorithm

Sample Semi–Reducible Paths [Morris, 2006]

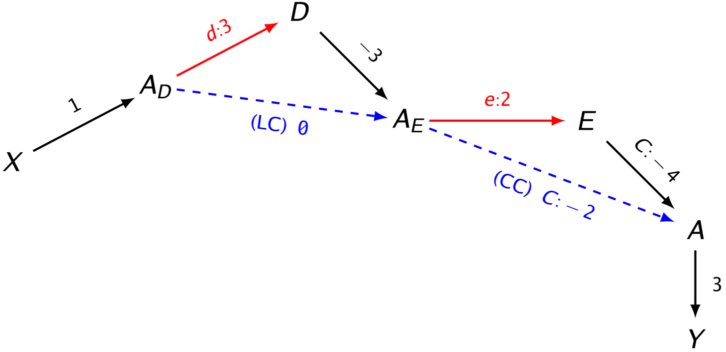
A path is *semi–reducible* if it can be transformed into a path with no *lower–case* edges.



Morris' $O(n^4)$–time DC–checking Algorithm

Sample Semi–Reducible Paths [Morris, 2006]

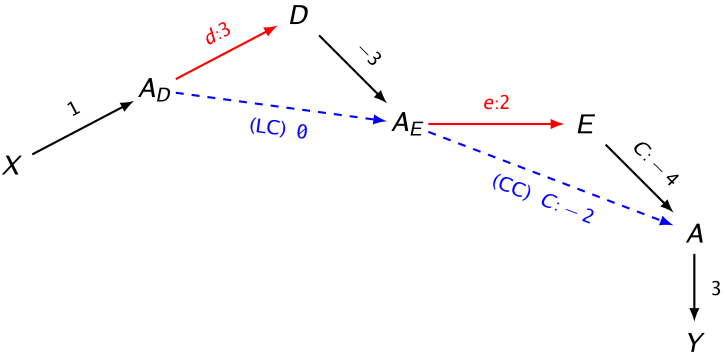
A path is *semi–reducible* if it can be transformed into a path with no *lower–case* edges.



Morris' $O(n^4)$–time DC–checking Algorithm

Sample Semi–Reducible Paths [Morris, 2006]

A path is *semi–reducible* if it can be transformed into a path with no *lower–case* edges.

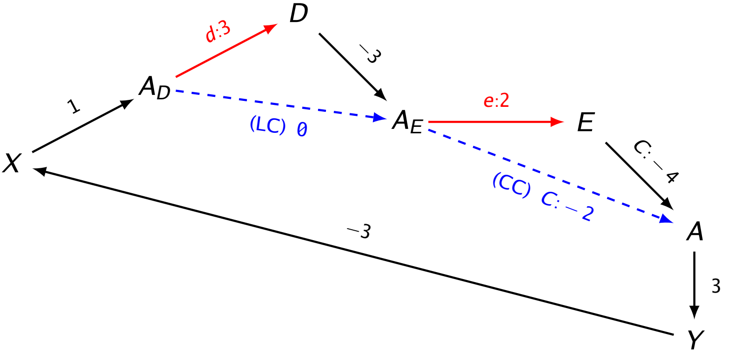


The original path from X to Y is semi–reducible.

Morris' $O(n^4)$–time DC–checking Algorithm

Sample Semi–Reducible Paths [Morris, 2006]

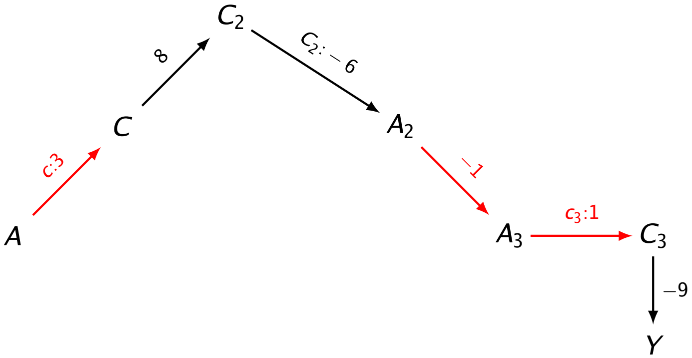
A path is *semi–reducible* if it can be transformed into a path with no *lower–case* edges.



With edge from Y to X, it becomes
a semi–reducible negative (SRN) cycle!

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

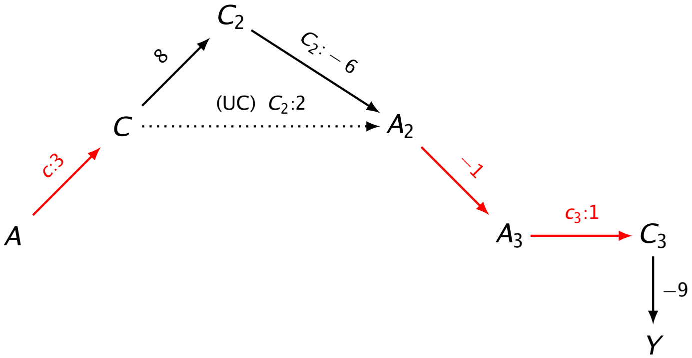
Based on "canonical reduction" of semi-reducible paths



Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

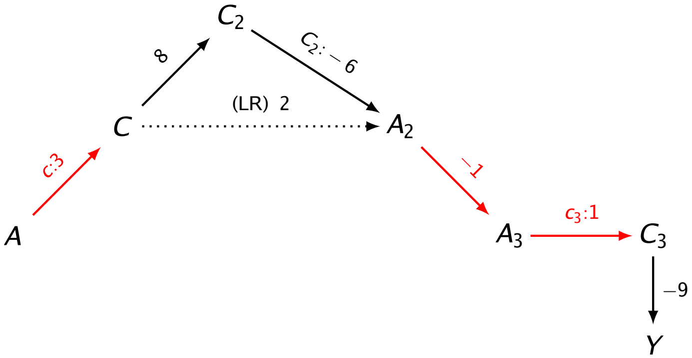
Based on "canonical reduction" of semi-reducible paths



Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

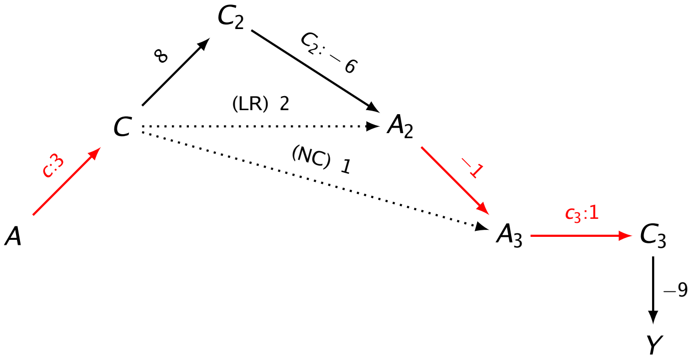
Based on "canonical reduction" of semi-reducible paths



Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

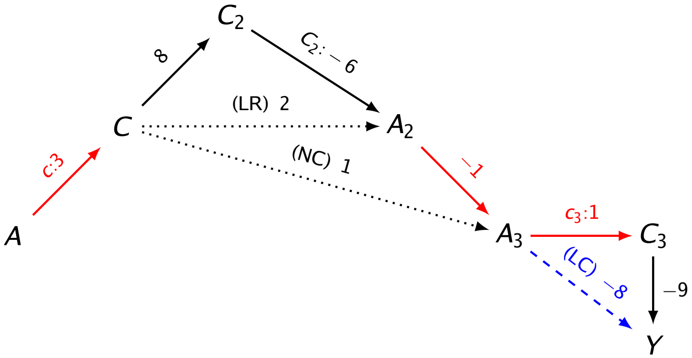
Based on "canonical reduction" of semi-reducible paths



Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

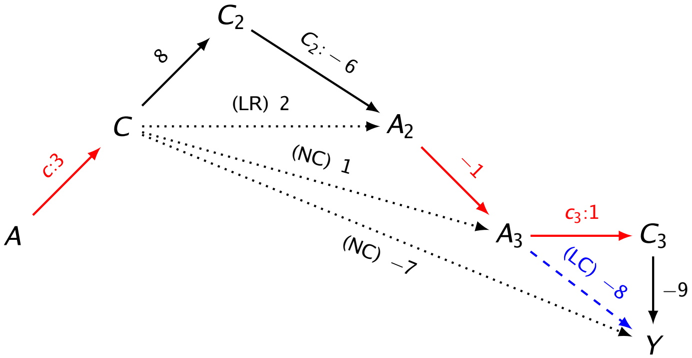
Based on "canonical reduction" of semi-reducible paths



Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

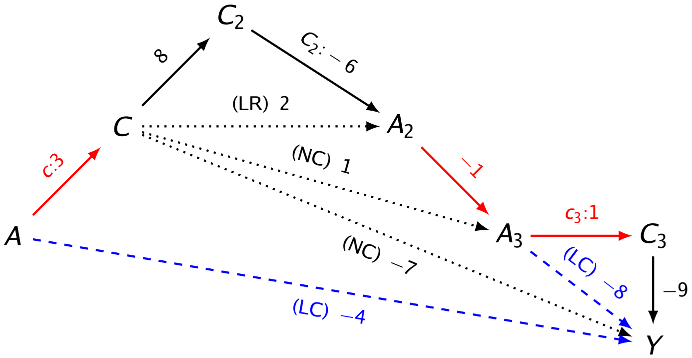
Based on "canonical reduction" of semi-reducible paths



Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Based on "canonical reduction" of semi-reducible paths



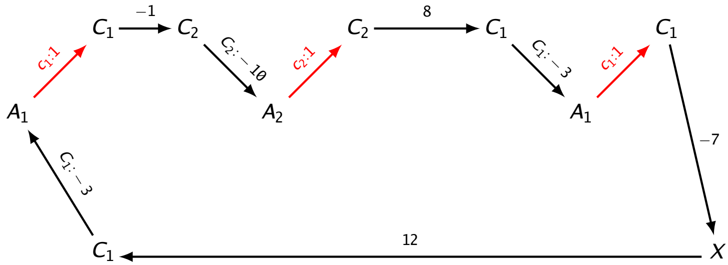
Key idea: Propagate forward along OU paths emanating from C attempting to "reduce away" the lower-case edge $A \xrightarrow{c:3} C$.

Morris' 2006 $O(n^4)$ DC–Checking Algorithm

```
for i = 1 to k,
    Run Bellman–Ford on OU graph to get potential function
    for each contingent link $(A, x, y, C)$,
        Do Dijkstra–like propagation using $C$ as source,
            following paths in the re–weighted OU graph.
    Insert all new edges from this round.
Run Bellman–Ford on OU graph to verify consistency.
```

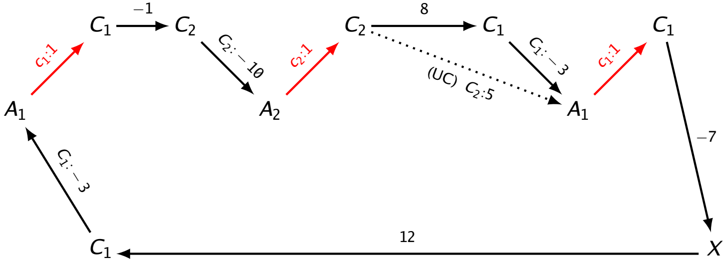
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



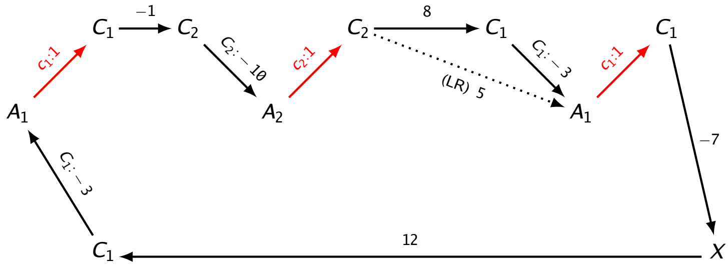
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



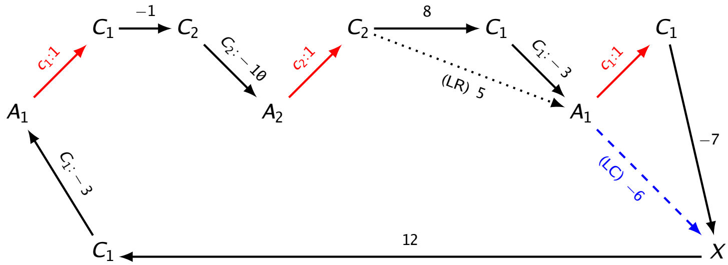
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



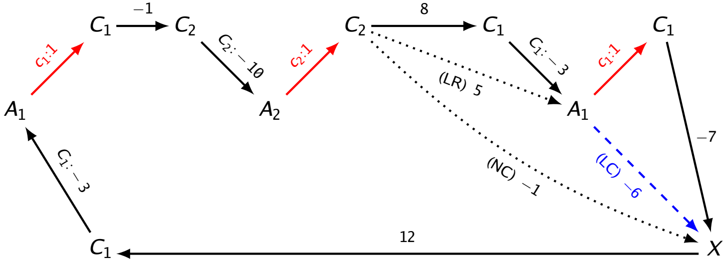
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



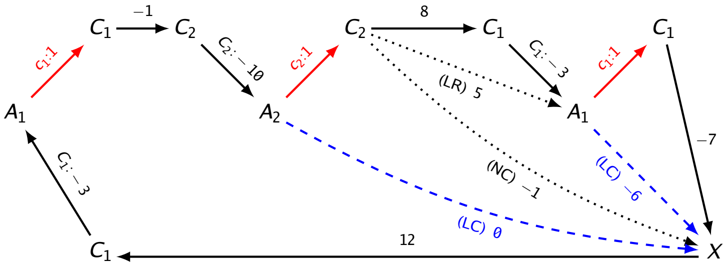
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



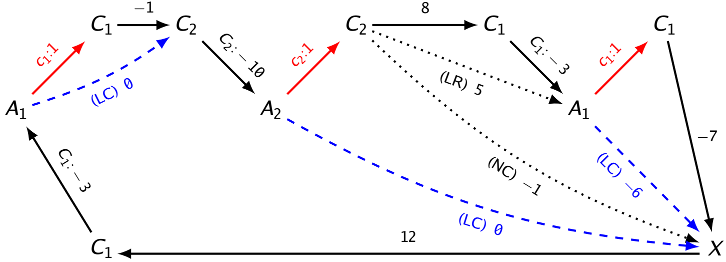
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



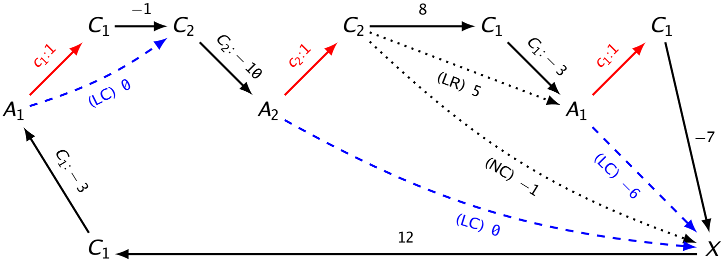
Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



Morris' 2006 $O(n^4)$ DC–Checking Algorithm

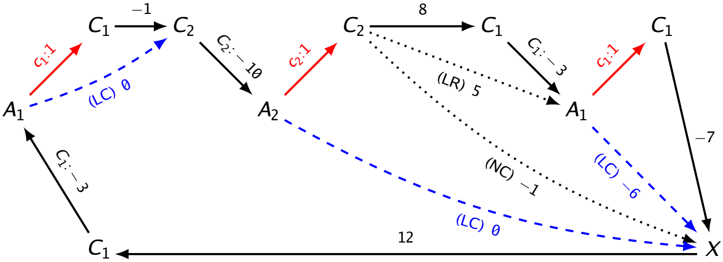
Finding a Semi–Reducible Negative Loop



- Negative cycle in OU graph! Therefore, STNU not DC.
  (All edges in OU graph must be satisfied in AllMax projection.)

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

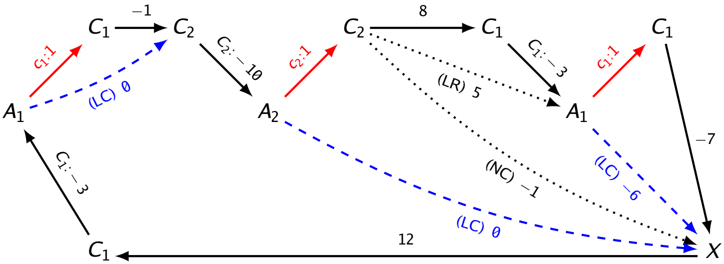
Finding a Semi-Reducible Negative Loop



- Nesting of edge generation implies that the order in which LC edges are processed matters!

Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Finding a Semi-Reducible Negative Loop



- Nesting of edge generation implies that the order in which LC edges are processed matters!
- If A_2C_2 is processed first, then need two rounds;
  If A_1C_1 is processed first, then need only one round.
  If heuristic guesses good nesting order, $O(n^4) \rightarrow O(n^3)$.
  [Hunsberger, 2013]

Morris' 2014 $O(n^3)$ DC–checking algorithm

Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

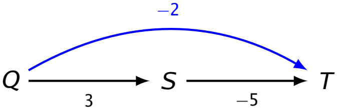
NC Rule:

Morris' 2014 $O(n^3)$ DC–checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi–reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non–neg. edges can be "absorbed" by neg. edges.

NC Rule:

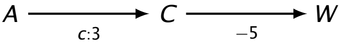


Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

LC Rule:

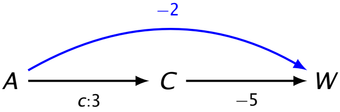


Morris' 2014 $O(n^3)$ DC–checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi–reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non–neg. edges can be "absorbed" by neg. edges.

LC Rule:

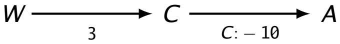


Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

UC Rule:

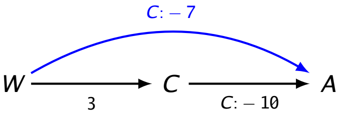


Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

UC Rule:

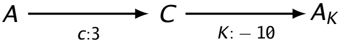


Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

CC Rule:

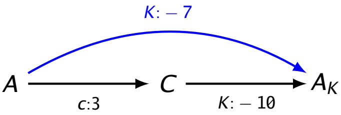


Morris' 2014 $O(n^3)$ DC-checking algorithm

Key insights

- When the 2006 algorithm propagates forward along paths in the OU graph, nesting of edge generation can cause problems
  — unless you process LC edges in a "good" order.
- A semi-reducible negative cycle can always be reduced to a cycle consisting solely of negative OU edges since preceding non-neg. edges can be "absorbed" by neg. edges.

CC Rule:



Morris' 2014 $O(n^3)$ DC-checking algorithm

Overview

- Morris' 2014 algorithm uses the same edge-generation rules as the 2006 algorithm, but:
    - starts from negative OU edges
    - propagates backward along non-neg. LO edges, absorbing them
    - looks for opportunities to bypass negative OU edges with non-negative OU edges.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Overview

- Morris' 2014 algorithm uses the same edge-generation rules as the 2006 algorithm, but:
    - starts from negative OU edges
    - propagates backward along non-neg. LO edges, absorbing them
    - looks for opportunities to bypass negative OU edges with non-negative OU edges.
- Propagating backward automatically resolves the nesting issue!

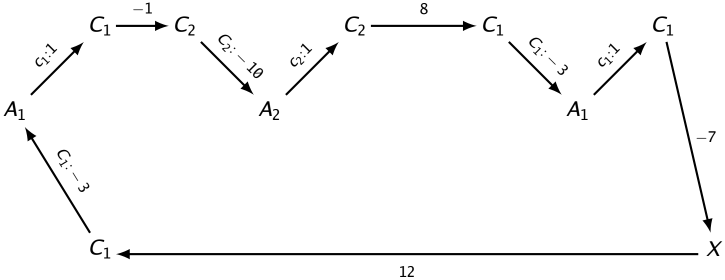
Morris' 2014 $O(n^3)$ DC–checking algorithm

Overview

- Morris' 2014 algorithm uses the same edge–generation rules as the 2006 algorithm, but:
  - starts from negative OU edges
  - propagates backward along non–neg. LO edges, absorbing them
  - looks for opportunities to bypass negative OU edges with non–negative OU edges.
- Propagating backward automatically resolves the nesting issue!
- Since back–propagation is only done along non–negative LO edges, don't need for a potential function!

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



Starting from a negative edge ...

Propagate backward along *non-negative* edges ...

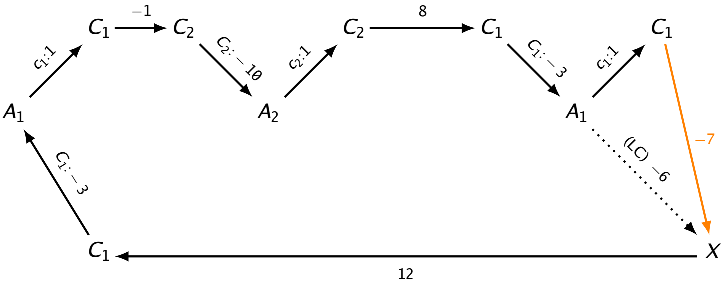
If path-length goes non-negative,
generate a non-negative bypass edge.

If ever encounter another negative edge...

Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



Starting from a negative edge ...

Propagate backward along *non-negative* edges ...

If path-length goes non-negative,

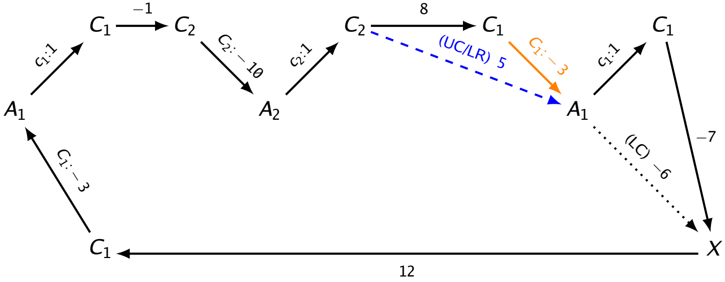
generate a non-negative bypass edge.

If ever encounter another negative edge...

Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle

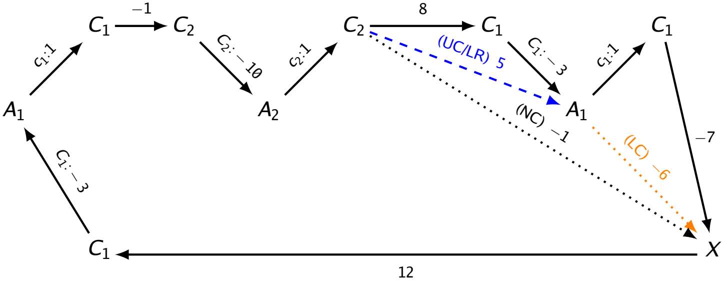


Starting from a negative edge ...

- Propagate backward along *non-negative* edges ...
  - If path-length goes non-negative,
    generate a non-negative bypass edge.
- If ever encounter another negative edge...
  - Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC–checking algorithm

Finding a Semi–Reducible Negative Cycle



Starting from a negative edge ...

Propagate backward along *non–negative* edges ...

If path–length goes non–negative,

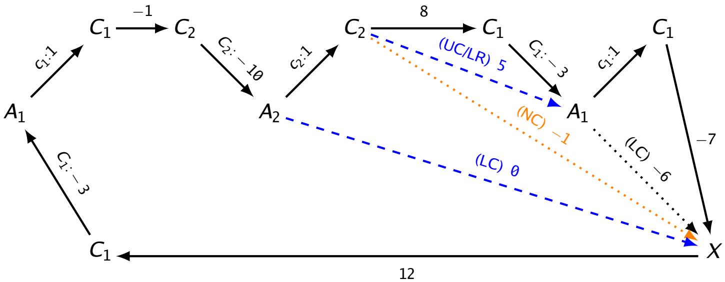
generate a non–negative bypass edge.

If ever encounter another negative edge...

Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle

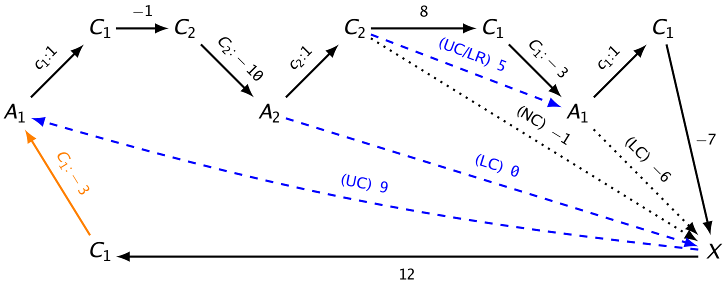


Starting from a negative edge ...

- Propagate backward along *non-negative* edges ...
  - If path-length goes non-negative,
    - generate a non-negative bypass edge.
- If ever encounter another negative edge...
  - Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



Starting from a negative edge . . .

Propagate backward along *non-negative* edges . . .

If path-length goes non-negative,

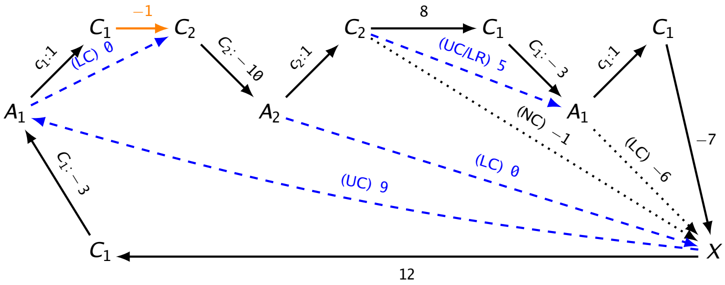
generate a non-negative bypass edge.

If ever encounter another negative edge. . .

Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle

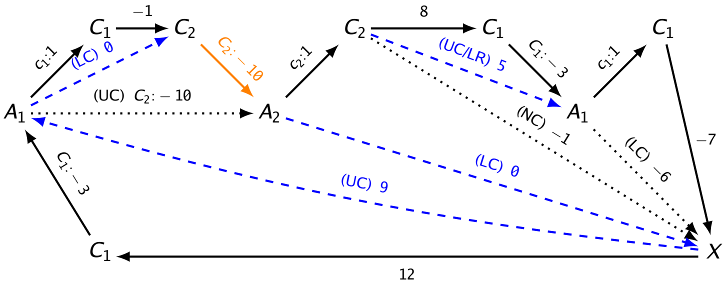


Starting from a negative edge . . .

- Propagate backward along *non-negative* edges . . .
  - If path-length goes non-negative,
    generate a non-negative bypass edge.
- If ever encounter another negative edge. . .
  - Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



Starting from a negative edge ...

Propagate backward along *non-negative* edges ...

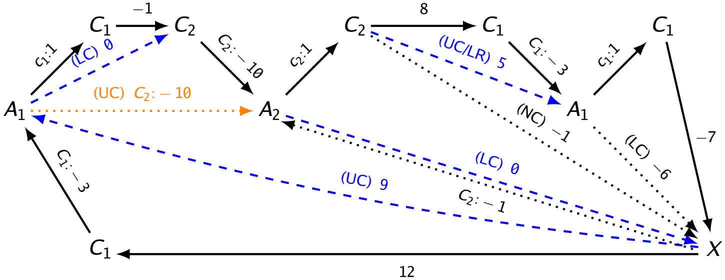
If path-length goes non-negative,
generate a non-negative bypass edge.

If ever encounter another negative edge...

Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



Starting from a negative edge . . .

Propagate backward along *non-negative* edges . . .

If path-length goes non-negative,

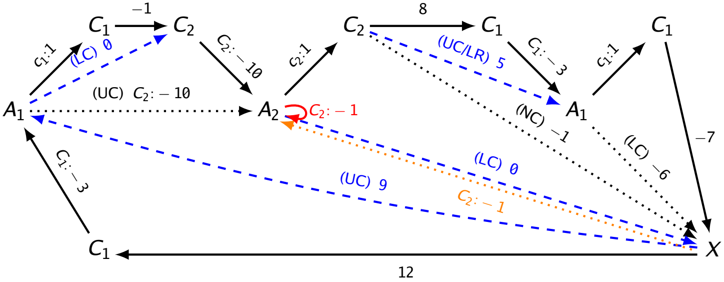
generate a non-negative bypass edge.

If ever encounter another negative edge. . .

Recursively process the interrupting edge.

Morris' 2014 $O(n^3)$ DC–checking algorithm

Finding a Semi–Reducible Negative Cycle



Starting from a negative edge ...

Propagate backward along *non–negative* edges ...

If path–length goes non–negative,

generate a non–negative bypass edge.

If ever encounter another negative edge...

Recursively process the interrupting edge.

Morris' $O(n^3)$–time DC–Checking Algorithm

Pseudocode

```
for each negative node X:
  if (DCbackprop(X) = false) return false
else return true
```

Morris' 2014 $O(n^3)$ DC-checking Algorithm

Pseudocode – Detect Negative Cycle or Redundant Call

```
DCbackprop(source)
  if ancestor call with same source:  return false;
  if prior terminated call with source:  return true;

  distance(source) = 0;
  for each node x other than source:  distance(x) = infinity;
  PriorityQueue queue = empty;
  for each Edge(n,wt,source) in InEdges(source) do
    distance(n) = wt;
    insert n into queue;

  while queue not empty:

    ... body of while loop ...

  return true;
```

Morris' 2014 $O(n^3)$ DC-checking Algorithm

Pseudocode – Initialize Priority Queue

```
DCbackprop(source)
  if ancestor call with same source:     return false;
  if prior terminated call with source:  return true;

  distance(source) = 0;
  for each node x other than source:  distance(x) = infinity;
  PriorityQueue queue = empty;
  for each Edge(n,wt,source) in InEdges(source) do
   distance(n) = wt;
   insert n into queue;

  while queue not empty:

    ... body of while loop ...

  return true;
```

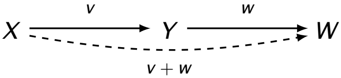
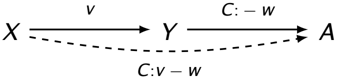
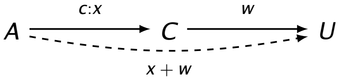
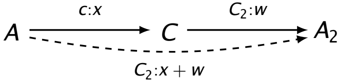
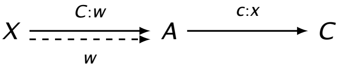
Morris' 2014 $O(n^3)$ DC–checking Algorithm

Pseudocode – The Main Loop

```
DCbackprop(source)
  ... detect neg cycle or redundant call ...
  ... init priority queue ...

  while queue not empty:
    pop Node u from queue;
    if distance(u) >= 0:
      add Edge(u,distance(u),source) to graph;      // Bypass Edge!
    else:
      if (u is negative node):
        if (DCbackprop(u) = false):  return false;  // Recursive Call!
      for each e = Edge(n,wt,u) in InEdges(u):
        if (wt >= 0) and (e is suitable):
          if distance(u) + wt < distance(v)    // One-step back-prop along
            distance(v) = distance(u) + wt;    // non-negative edges
            insert v into queue;
  return true;
```

The RUL$^-$ DC-checking Algorithm

Recall the MM05 Propagation Rules

Rule	**Graphical Representation**	**Applicability Conditions**
No Case (NC)		(none)
Upper Case (UC)		(none)
Lower Case (LC)		$w < 0$
Cross Case (CC)		$C_2 \not\equiv C,\ w < 0$
Label Removal (LR)		$w \geq -x$

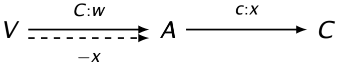
Recall the MM05 Propagation Rules

Rule	**Graphical Representation**	**Applicability Conditions**
No Case (NC)	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, bend right, dashed, "v+w"'] & Y \arrow[r, "w"] & W \end{tikzcd}$	(none)
Upper Case (UC)	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, bend right, dashed, "C{:}v-w"'] & Y \arrow[r, "C{:}-w"] & A \end{tikzcd}$	(none)
Lower Case (LC)	$\begin{tikzcd} A \arrow[r, "c{:}x"] \arrow[rr, bend right, dashed, "x+w"'] & C \arrow[r, "w"] & U \end{tikzcd}$	$w < 0$
Cross Case (CC)	$\begin{tikzcd} A \arrow[r, "c{:}x"] \arrow[rr, bend right, dashed, "C_2{:}x+w"'] & C \arrow[r, "C_2{:}w"] & A_2 \end{tikzcd}$	$C_2 \not\equiv C,\ w < 0$
Label Removal (LR)	$\begin{tikzcd} X \arrow[r, shift left, "C{:}w"] \arrow[r, shift right, dashed, "w"'] & A \arrow[r, "c{:}x"] & C \end{tikzcd}$	$w \geq -x$

All of these rules are length preserving!

General Unordered Reduction (GUR) Rule

From [Morris et al., 2001]

	Graphical Representation	Applicability Condition
In General:		$w < -x$ (eqiuv., $-w > x$)

General Unordered Reduction (GUR) Rule

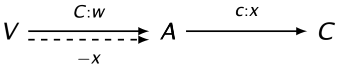
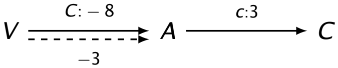
From [Morris et al., 2001]

	Graphical Representation	Applicability Condition
In General:	$V \underset{-x}{\overset{\xrightarrow{C:w}}{\dashrightarrow}} A \xrightarrow{c:x} C$	$w < -x$ (eqiuv., $-w > x$)
Example:	$V \underset{-3}{\overset{\xrightarrow{C:-8}}{\dashrightarrow}} A \xrightarrow{c:3} C$	$-8 < -3$ (eqiuv., $8 > 3$)

- Given: While C unexecuted, V must wait at least 8 after A.
- But C cannot execute sooner than 3 after A.
- Therefore, in every situation, V must wait at least 3 after A.

General Unordered Reduction (GUR) Rule

From [Morris et al., 2001]

	Graphical Representation	Applicability Condition
In General:	 $V \xrightarrow[-x]{C:w} A \xrightarrow{c:x} C$	$w < -x$ (eqiuv., $-w > x$)
Example:	 $V \xrightarrow[-3]{C:-8} A \xrightarrow{c:3} C$	$-8 < -3$ (eqiuv., $8 > 3$)

- Given: While C unexecuted, V must wait at least 8 after A.
- But C cannot execute sooner than 3 after A.
- Therefore, in every situation, V must wait at least 3 after A.

The GUR rule is *not* length–preserving.

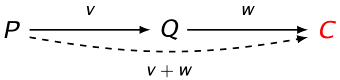
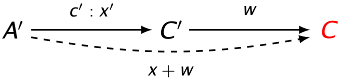
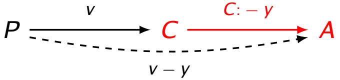
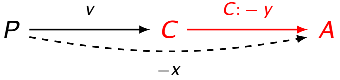
The RUL$^-$ Propagation Rules
[Cairo et al., 2018]

Rule	Graphical representation	Applicability Conditions
R$^-$	$\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dashed, bend right, "v+w"'] & Q \arrow[r, "w"] & C \end{tikzcd}$	$(A, x, y, C) \in \mathcal{L},\ w < y - x,\ Q \in \mathcal{T}_X$
L$^-$	$\begin{tikzcd} A' \arrow[r, "c' : x'"] \arrow[rr, dashed, bend right, "x+w"'] & C' \arrow[r, "w"] & C \end{tikzcd}$	$(A, x, y, C) \in \mathcal{L},\ w < y - x,\ C' \not\equiv C$
U$^-$	$\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dashed, bend right, "\max\{v-y, -x\}"'] & C \arrow[r, "C: -y"] & A \end{tikzcd}$	$(A, x, y, C) \in \mathcal{L}$

- R$^-$ $\approx$ NC, L$^-$ $\approx$ LC, U$^-$ $\approx$ (UC + LR + GUR).
- RUL$^-$ rules only generate *ordinary* edges.
- RUL$^-$ rules never generate wait edges, so no need for CC rule.
- Propagations always involve 1–2 contingent time-points ($k < n$).

The RUL$^-$ Propagation Rules — Take 2

[Hunsberger and Posenato, 2022]

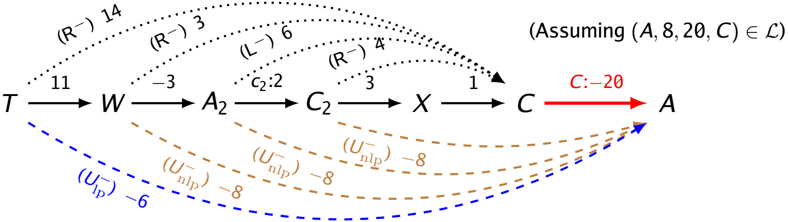
Rule	Graphical representation	Applicability Conditions
R$^-$		$(A, x, y, C) \in \mathcal{L},\ w < y - x,\ Q \in \mathcal{T}_X$
L$^-$		$(A, x, y, C) \in \mathcal{L},\ w < y - x,\ C' \not\equiv C$
U^-_{lp}		$(A, x, y, C) \in \mathcal{L},\ v - y \geq -x$
U^-_{nlp}		$(A, x, y, C) \in \mathcal{L},\ v - y < -x$

Only the U^-_{nlp} rule is not length preserving.

The RUL$^-$ DC-Checking Algorithm

An $O(mn + k^2 n + kn \log n)$–time algorithm, [Cairo et al., 2018]

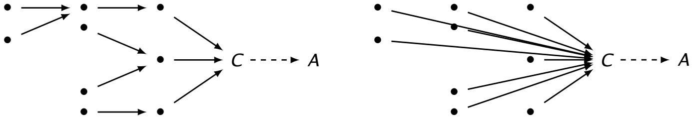
Starting from UC edges, propagate backward along LO edges, using the RUL$^-$ rules, aiming to generate ordinary bypass edges.



The RUL$^-$ DC–Checking Algorithm

Processing a UC edge, $C \xrightarrow{C:\,-y} A$

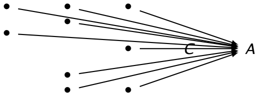
- Phase 1:
  - (a) Back–prop from C along LO edges.
  - (b) Use L$^-$ and R$^-$ to generate new edges terminating at C.



(a) Back–prop from C

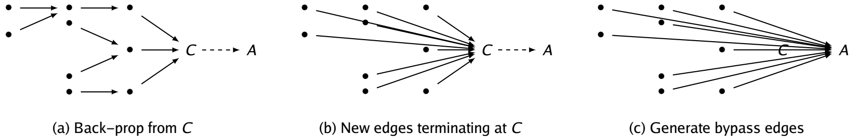
(b) Generate new edges terminating at C

- Phase 2: For each new edge XC generated during Phase 1, apply U^-_{lp} or U^-_{nlp} to XC and CA to generate ordinary bypass edge.



RUL$^-$ DC-Checking Algorithm

Overview (continued)

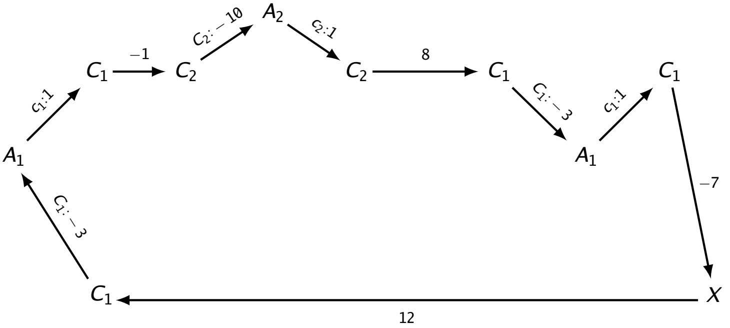


(a) Back-prop from C (b) New edges terminating at C (c) Generate bypass edges

- Backward propagation along LO edges in Phase 1 guided by Dijkstra, using a potential function to re-weight the LO graph.
- After inserting new bypasses edges in Phase 2, incrementally update the potential function.
- If Phase 1 back-prop from C encounters another UC edge $C'A'$ that has not yet been processed, then interrupt Phase 1 back-prop from C, and instead process $C'A'$.
- Re-start Phase 1 back-prop from C only after all interrupting UC edges have been processed.
- Push-down stacks used to ensure that each UC edge processed at most twice; and to detect negative cycles of interruptions.

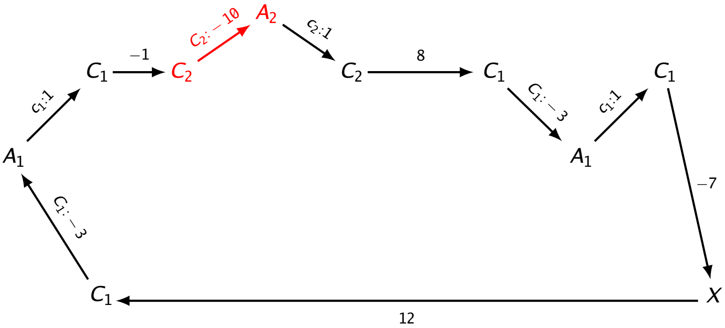
RUL^{-} DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



RUL$^-$ DC-checking algorithm

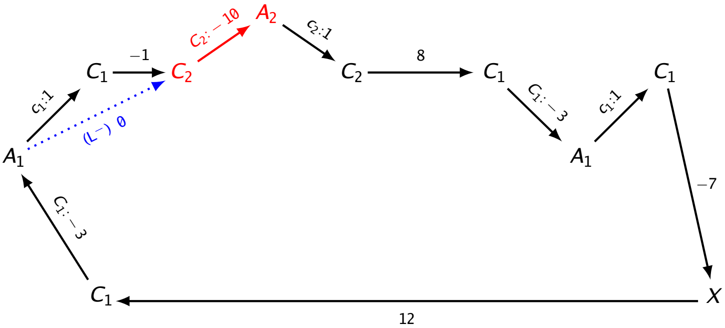
Finding a Semi-Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.

RUL$^-$ DC-checking algorithm

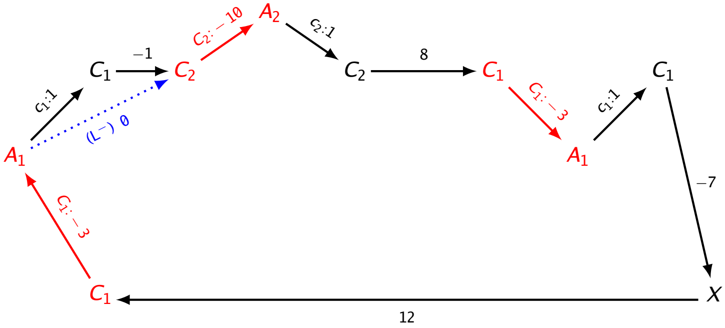
Finding a Semi-Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
    - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.

RUL$^-$ DC-checking algorithm

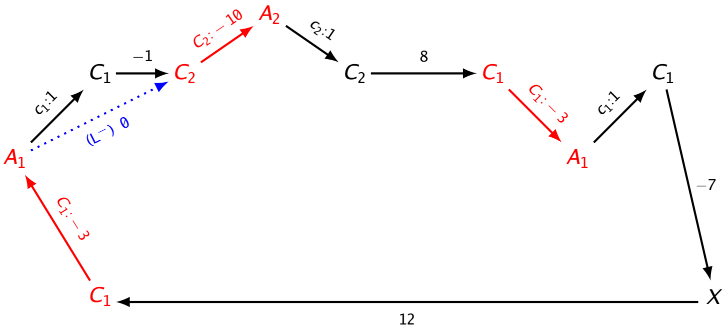
Finding a Semi-Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back-prop from C_2 using L$^-$ and R$^-$.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:-3} A_1$.

RUL$^-$ DC–checking algorithm

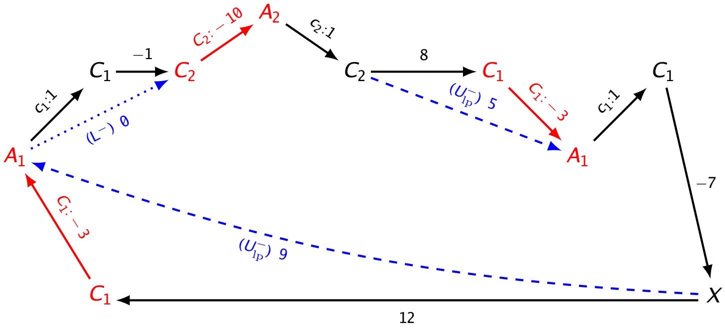
Finding a Semi–Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:\,-3} A_1$.
    - Phase 1: Back–prop from C_1 using L$^-$ and R$^-$.
      No new edges since $12 \geq 2 = y - x$ and $8 \geq 2 = y - x$.

RUL$^-$ DC–checking algorithm

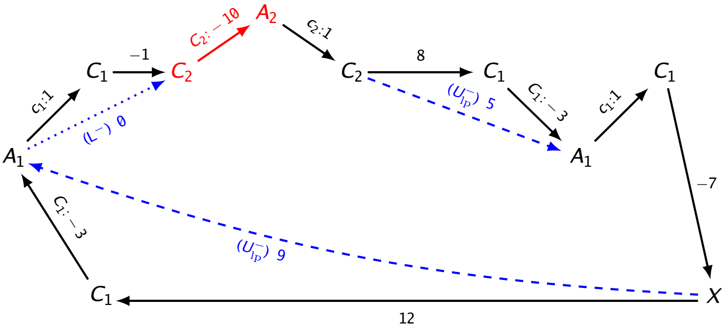
Finding a Semi–Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:\,-3} A_1$.
    - Phase 1: Back–prop from C_1 using L$^-$ and R$^-$.
      No new edges since $12 \geq 2 = y - x$ and $8 \geq 2 = y - x$.
    - Phase 2: Generate bypass edges using U$^-$.

RUL$^-$ DC-checking algorithm

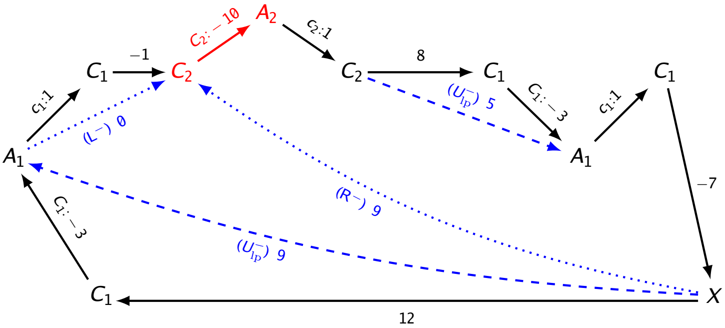
Finding a Semi-Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back-prop from C_2 using L$^-$ and R$^-$.
  - Phase 1 (again): Back-prop from C_2 using L$^-$ and R$^-$.

RUL$^-$ DC-checking algorithm

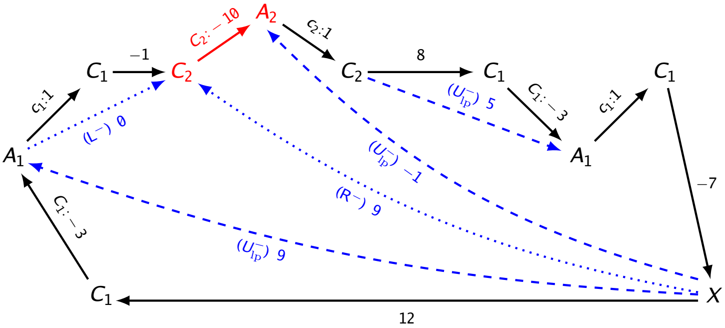
Finding a Semi-Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back-prop from C_2 using L$^-$ and R$^-$.
  - Phase 1 (again): Back-prop from C_2 using L$^-$ and R$^-$.

RUL$^-$ DC-checking algorithm

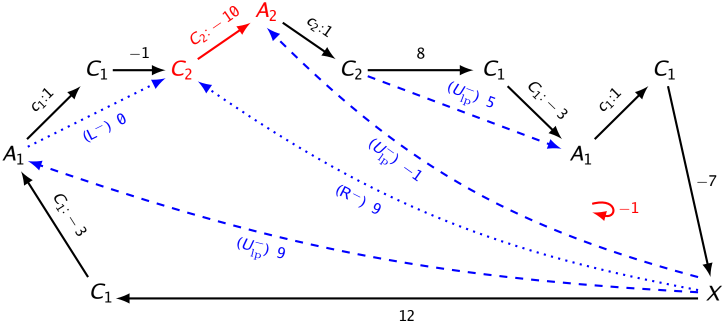
Finding a Semi-Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back-prop from C_2 using L$^-$ and R$^-$.
  - Phase 1 (again): Back-prop from C_2 using L$^-$ and R$^-$.
  - Phase 2: Generate bypass edge using $U^-_{\rm lp}$.

RUL$^-$ DC–checking algorithm

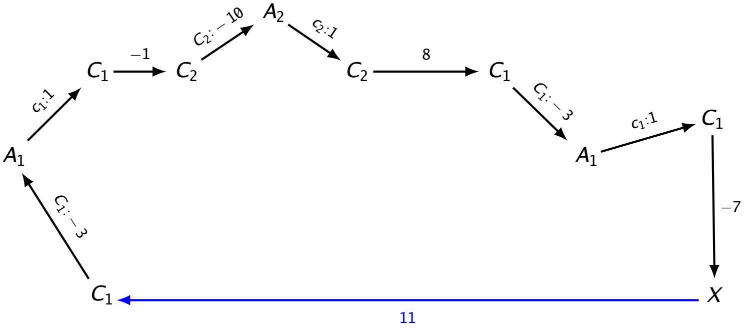
Finding a Semi–Reducible Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2 \colon -10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Phase 1 (again): Back–prop from C_2 using L$^-$ and R$^-$.
  - Phase 2: Generate bypass edge using U^-_{lp}.
  - Negative cycle detected when incrementally updating potential function on LO edges.

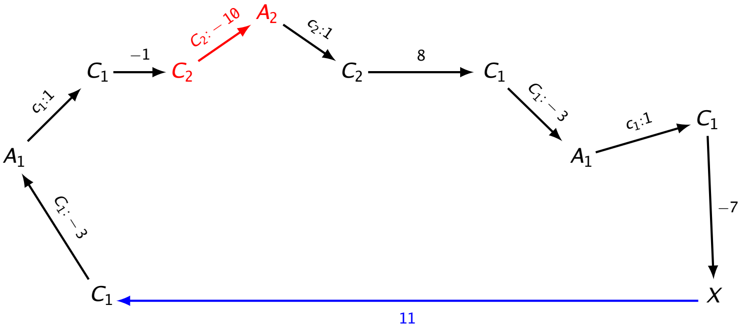
RUL$^-$ DC-checking algorithm

Finding a Slightly Different Negative Cycle



RUL$^-$ DC-checking algorithm

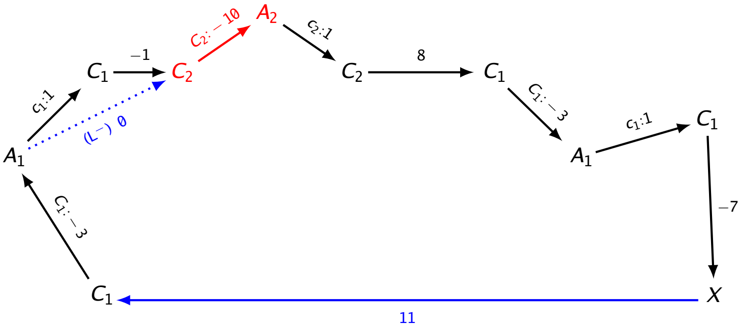
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.

RUL^{-} DC–checking algorithm

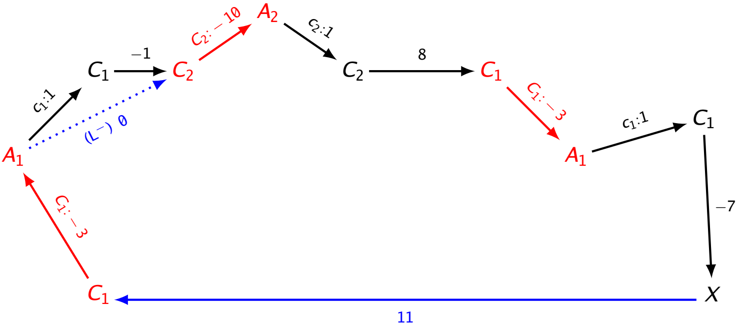
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L^{-} and R^{-}.

RUL$^-$ DC–checking algorithm

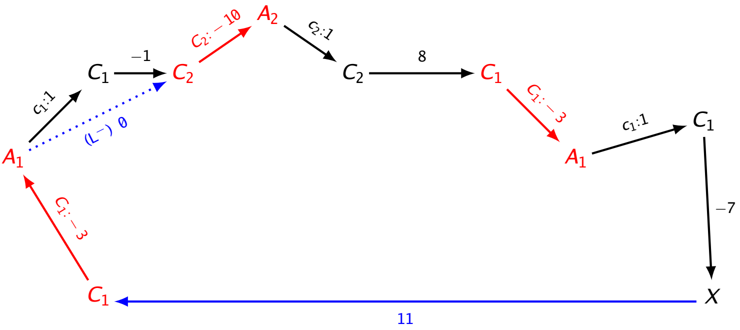
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:-3} A_1$.

RUL$^-$ DC–checking algorithm

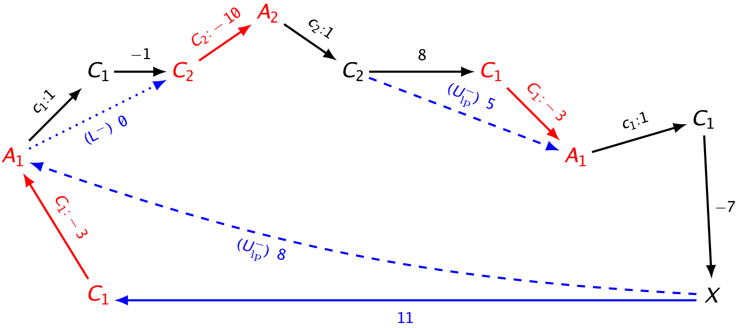
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:\,-3} A_1$.
    - Phase 1: Back–prop from C_1 using L$^-$ and R$^-$. No new edges since $11 \geq 2 = y - x$ and $8 \geq 2 = y - x$.

RUL$^-$ DC–checking algorithm

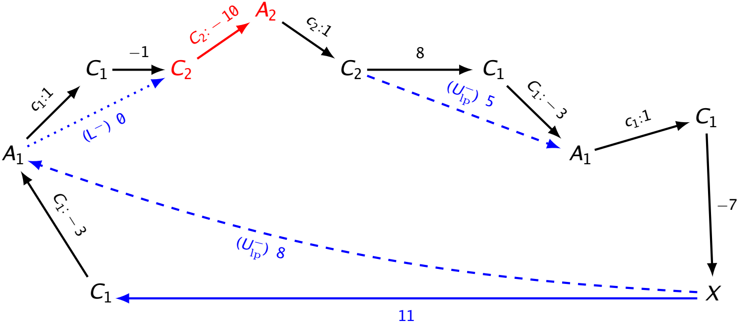
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:-3} A_1$.
    - Phase 1: Back–prop from C_1 using L$^-$ and R$^-$.
      No new edges since $11 \geq 2 = y - x$ and $8 \geq 2 = y - x$.
    - Phase 2: Generate bypass edges using U$^-$.

RUL⁻ DC–checking algorithm

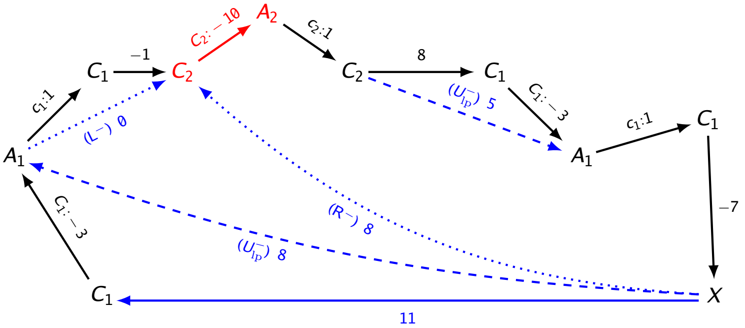
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L⁻ and R⁻.
  - Phase 1 (again): Back–prop from C_2 using L⁻ and R⁻.

RUL^{-} DC–checking algorithm

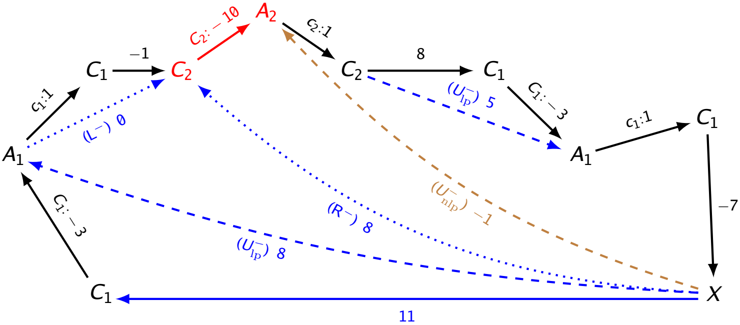
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
    - Phase 1: Back–prop from C_2 using L^{-} and R^{-}.
    - Phase 1 (again): Back–prop from C_2 using L^{-} and R^{-}. Other Phase 1 edges not shown.

RUL$^-$ DC–checking algorithm

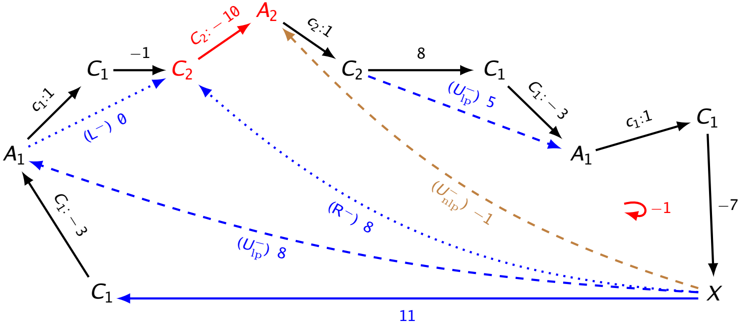
Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
    - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
    - Phase 2: Generate bypass edge using U^-_{nlp}
      Other such edges not shown.

RUL$^-$ DC–checking algorithm

Finding a Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Phase 2: Generate bypass edge using U^-_{nlp} Other such edges not shown.
  - Negative cycle in LO graph (found when trying to update potential function over LO edges).

RUL$^-$ DC-checking Algorithm

Summary

- Complexity: $O(mn + k^2 n + kn \log n)$ time
- Faster than Morris' `2014` $O(n^3)$ algorithm on sparse graphs (e.g., in cases where Morris' algorithm generates $O(n^2)$ edges).

The RUL2021 Algorithm

The RUL2021 DC-Checking Algorithm

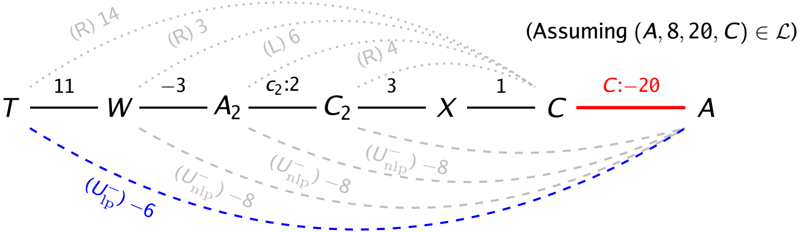
[Hunsberger and Posenato, 2022]

- Combines techniques from prior algorithms with novel ideas.
  - Like RUL$^-$:
    - Back-propagates from upper-case edges
    - Uses potential function to enable Dijkstra
    - Uses the *length-preserving* rules from RUL$^-$
  - Unlike RUL$^-$:
    - Does not use *non-length-preserving* U^-_{nlp} rule
    - Inserts *dramatically* fewer edges
    - Uses *some* forward propagation (like Morris06), but only to detect certain (rarely encountered) negative cycles
    - Implemented recursively (like Morris14)
    - Deals more efficiently with interruptions
- Same worst-case $O(mn + k^2 n + kn \log n)$ time as RUL$^-$

  — but an order of magnitude faster in practice!

RUL2021 Algorithm: $O(mn + k^2 n + kn \log n)$ time

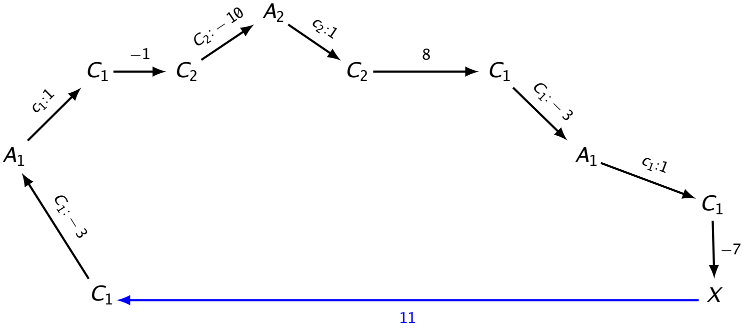
Propagate *backward* from *upper-case edges*, using RUL$^-$ rules (but not U^-_{nlp}), aiming to generate *bypass edges*.



- Computes—but does not insert!—dotted edges
- Does not compute or insert gray dashed edges
- Only inserts blue bypass edges.

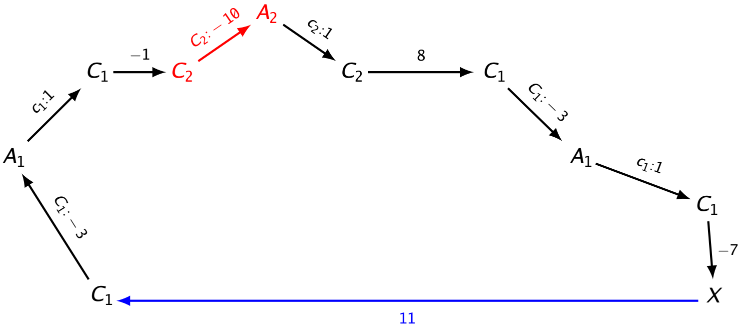
RUL2021 DC-checking algorithm

Finding the Slightly Different Negative Cycle



RUL2021 DC-checking algorithm

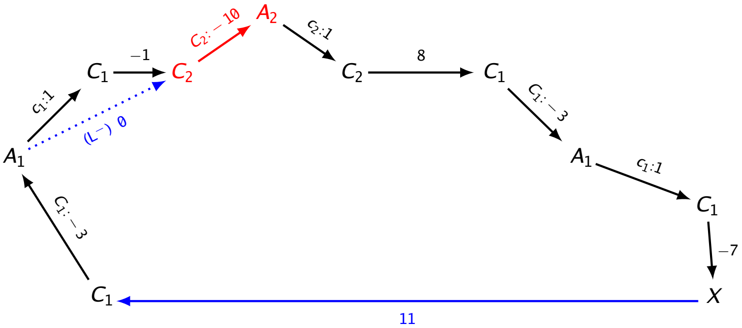
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.

RUL2021 DC–checking algorithm

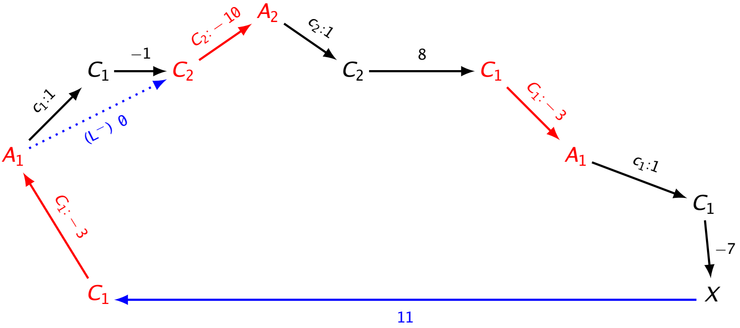
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
    - Phase 1: Back–prop from C_2 using L^- and R^-.

RUL2021 DC–checking algorithm

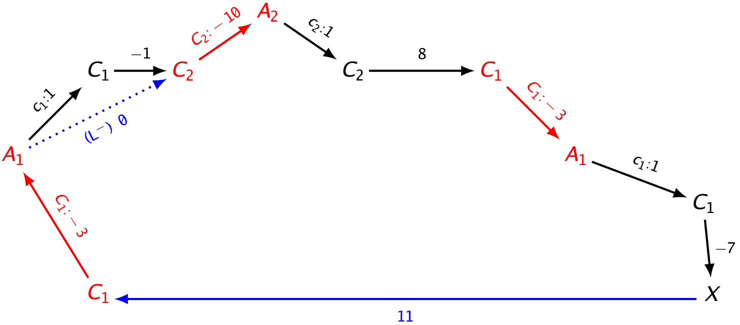
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L^- and R^-.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:-3} A_1$.

RUL2021 DC-checking algorithm

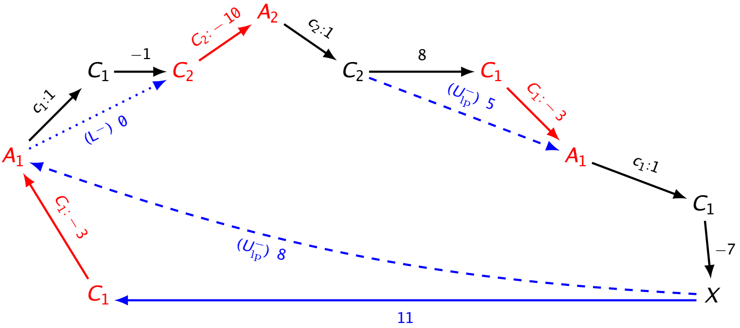
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L^- and R^-.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:-3} A_1$.
    - Phase 1: Back–prop from C_1 using L^- and R^-. No new edges since $11 \geq 2 = y - x$ and $8 \geq 2 = y - x$.

RUL2021 DC-checking algorithm

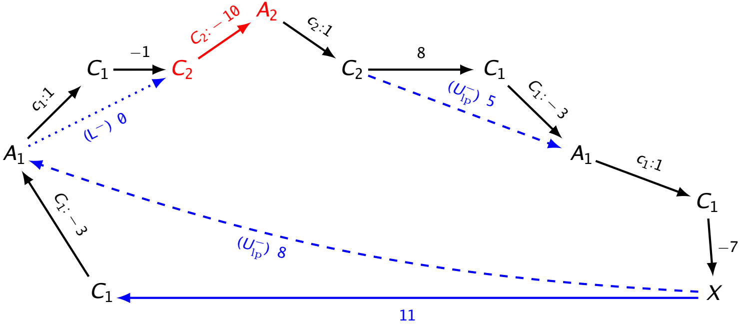
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L^- and R^-.
  - Interrupting UC edge: $C_1 \xrightarrow{C_1:-3} A_1$.
    - Phase 1: Back–prop from C_1 using L^- and R^-.
      No new edges since $11 \geq 2 = y - x$ and $8 \geq 2 = y - x$.
    - Phase 2: Generate bypass edges using U^-.

RUL2021 DC–checking algorithm

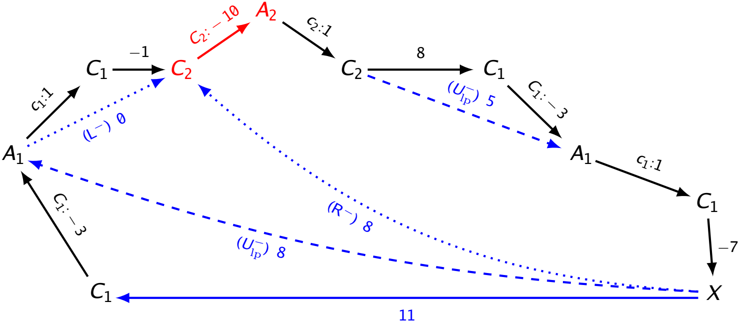
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2\colon -10} A_2$.
    - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
    - Phase 1 (again): Back–prop from C_2 using L$^-$ and R$^-$.

RUL2021 DC-checking algorithm

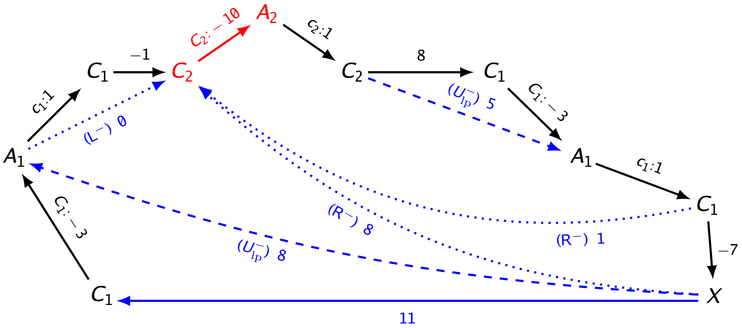
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
    - Phase 1: Back–prop from C_2 using L^- and R^-.
    - Phase 1 (again): Back–prop from C_2 using L^- and R^-.

RUL2021 DC–checking algorithm

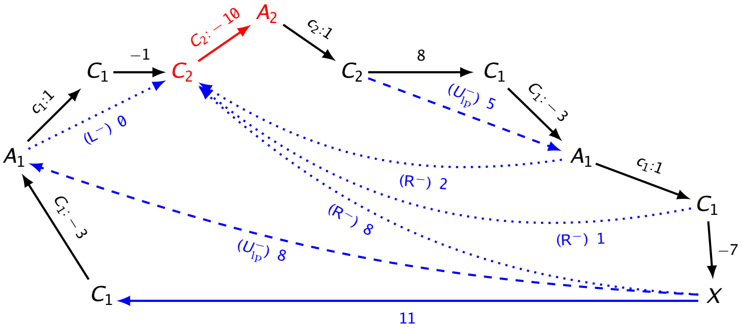
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back–prop from C_2 using L⁻ and R⁻.
  - Phase 1 (again): Back–prop from C_2 using L⁻ and R⁻.

RUL2021 DC-checking algorithm

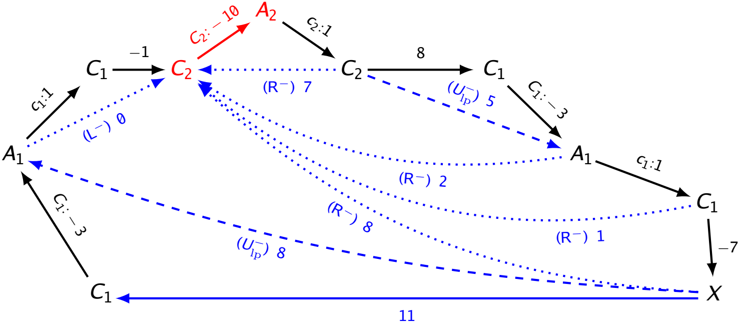
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L^- and R^-.
  - Phase 1 (again): Back–prop from C_2 using L^- and R^-.

RUL2021 DC-checking algorithm

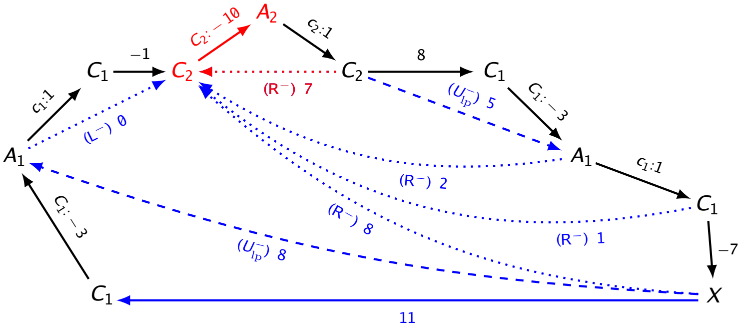
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L^- and R^-.
  - Phase 1 (again): Back–prop from C_2 using L^- and R^-.

RUL2021 DC-checking algorithm

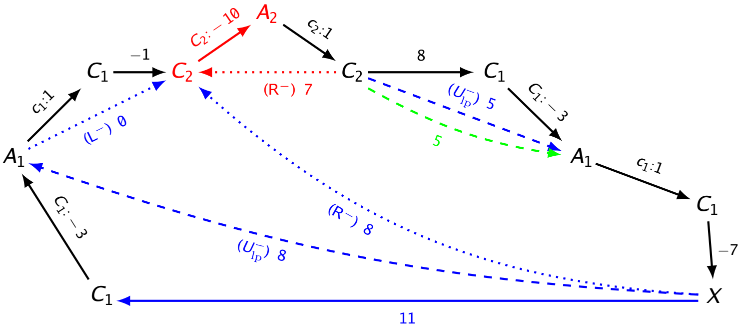
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L⁻ and R⁻.
  - Phase 1 (again): Back–prop from C_2 using L⁻ and R⁻.
  - Found C_2 to C_2 cycle of length $7 < \Delta_2 = 10 - 1 = 9$!
  - Must propagate forward!

RUL2021 DC-checking algorithm

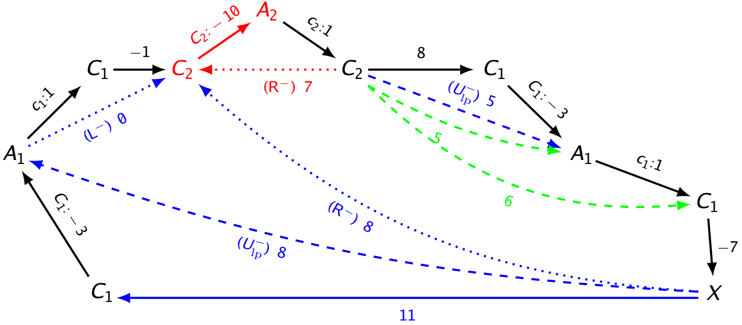
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:-10} A_2$.
  - Phase 1: Back-prop from C_2 using L^- and R^-.
  - Forward propagation from C_2 along LO edges

RUL2021 DC–checking algorithm

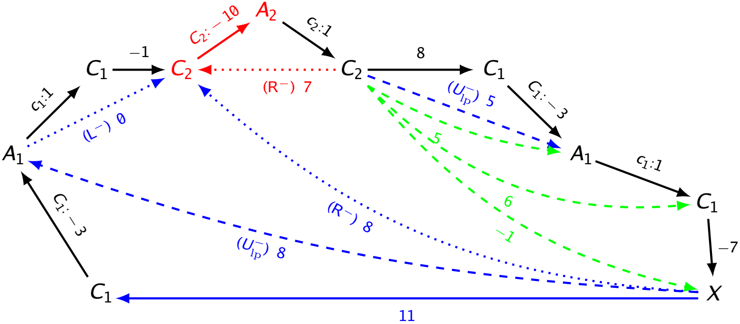
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
    - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
    - Forward propagation from C_2 along LO edges

RUL2021 DC-checking algorithm

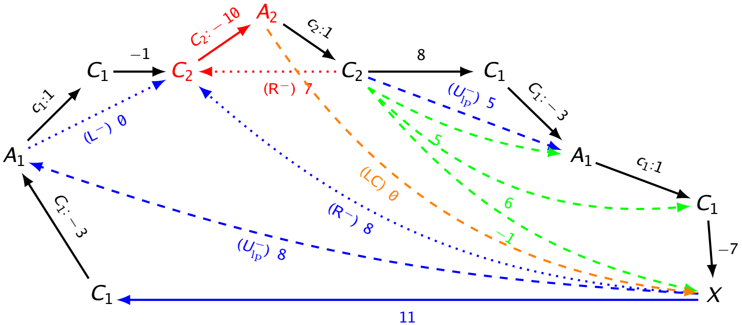
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
    - Phase 1: Back–prop from C_2 using L⁻ and R⁻.
    - Forward propagation from C_2 along LO edges

RUL2021 DC-checking algorithm

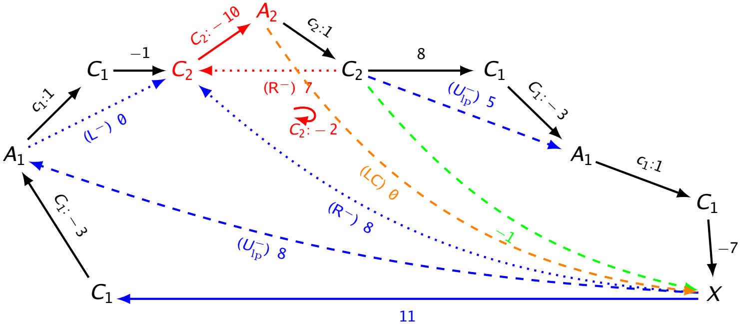
Finding the Slightly Different Negative Cycle



- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back–prop from C_2 using L$^-$ and R$^-$.
  - Forward propagation from C_2 along LO edges
  - Found negative length path from C_2 to X!
    — Indicates negative cycle in OU graph!

RUL2021 DC-checking algorithm

Finding the Slightly Different Negative Cycle

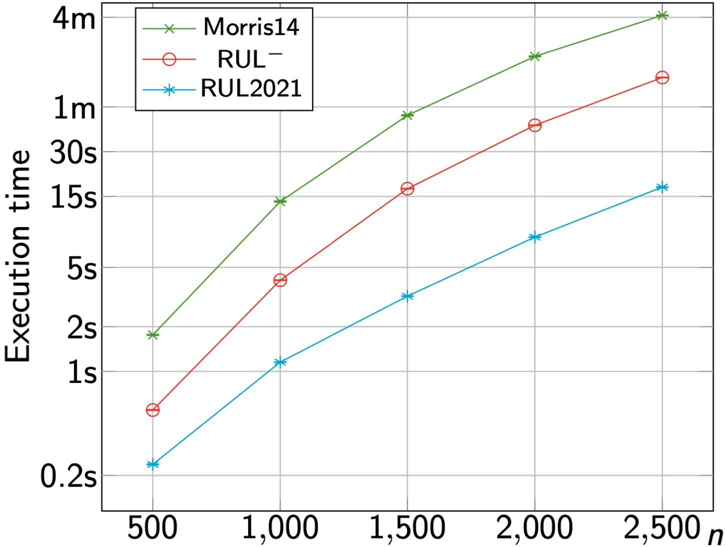


- Process UC edge, $C_2 \xrightarrow{C_2:\,-10} A_2$.
  - Phase 1: Back-prop from C_2 using L$^-$ and R$^-$.
  - Forward propagation from C_2 along LO edges
  - Found negative length path from C_2 to X!

    — Indicates negative cycle in OU graph!

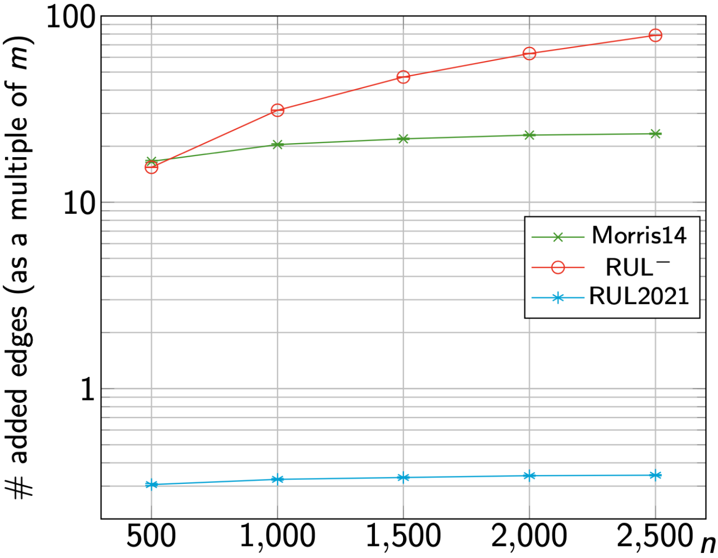
Empirical Evaluation

Execution Time vs. Number of Nodes, *n*



Empirical Evaluation

Number of Added Edges (as multiple of *m*) vs. Number of Nodes, *n*



Related Work

- Magic Loops: Worst-case indivisible semi-reducible negative cycles with applications to DC-Checking for STNUs

  [Hunsberger, 2013, 2014a,b, 2015a]

- Using Timed Game Automata (TGAs) to synthesize execution strategies for STNUs

  [Cimatti et al., 2014b]

- Semantics for STNUs and real-time execution decisions

  [Hunsberger, 2009]

Dispatchability for STNUs

Motivating Dispatchability for STNUs

Recall RTE Algorithm for STNs

[Muscettola et al., 1998b]

Goal: Preserve flexibility while requiring minimal computation.

0. For each $X \in \mathcal{T}$, $\mathrm{TW}(X) = [0, \infty)$ (time windows)
1. $t := 0$ (curr. time); $\mathcal{U} := \mathcal{T}$ (unexecuted); **E** $:= \{Z\}$ (enabled)
2. Choose: Remove any $X \in$ **E** such that t is in X's time window;
3. Execute: set $X := t$, and remove X from $\mathcal{U}$;
4. Propagate: propagate $X = t$ to X's *immediate neighbors*;
5. Update: update **E** to include all $Y \in \mathcal{U}$ for which *no* negative edges emanating from Y have a destination in $\mathcal{U}$;
6. Wait: wait until t has advanced to some time between $\min\{lb(W) \mid W \in \mathbf{E}\}$ and $\min\{ub(W) \mid W \in \mathbf{E}\}$;
7. If $\mathcal{U}$ non-empty, go back to (2); else done.

RTE Algorithm for STNUs

Must incorporate WAIT constraints

- Initialize data (e.g., time windows, enabled TPs, etc.)
- While some TPs not yet executed:
  - ⋆ Generate real–time execution decision (RTED)
  - ⋆ Observe outcome: some TP executed
  - ⋆ Update data

RTED: Hunsberger [2009]

RTE Algorithm for STNUs

Initialize Data

Given STNU graph: $((\mathcal{T}_x \cup \mathcal{T}_c), \mathcal{E}_o, \mathcal{E}_{lc}, \mathcal{E}_{uc}, \mathcal{E}_{ucg})$:

- For each $X \in \mathcal{T}_x$:
  - $\star$ $\texttt{TW}(X) = [\texttt{lb}(X), \texttt{ub}(X)] = (-\inf, \inf)$ (Time Windows)
  - $\star$ $\texttt{ActWaits}(X) = \emptyset$ (Activated Waits)
- $\mathcal{U}_x = \mathcal{T}_x$ (Unexecuted Executable TPs)
- $\mathcal{U}_c = \mathcal{T}_c$ (Unexecuted Contingent TPs)
- $\texttt{EnabledTPs} = \{Z\}$ (Enabled Executable TPs)
- $\texttt{now} = 0$ (Current time)

RTE Algorithm for STNUs

Activated Wait Constraints

- Suppose there is a UC edge: $X \xrightarrow{C:-9} A$.
- This represents a wait constraint:

  While C unexecuted, X must wait at least 9 after A.

- If A not yet executed, then X cannot be enabled.
- If A executed (say, $A = 5$), then the wait is activated:

  While C unexecuted, X must wait until time 14.

  $\texttt{ActWaits}(X) = \{(14, C)\}$ (Activated Wait)

- If C executes before time 14, the wait disappears:

  $\texttt{ActWaits}(X) = \emptyset$

  Perhaps X can now be enabled ...

- Multiple activated waits:

  $\texttt{ActWaits}(X) = \{(14, C), (18, C_2), (21, C_5)\}$

RTE Algorithm for STNUs

Enabled Time–Points

An executable time–point X is not enabled for execution if:

- There is a negative edge from X to some unexecuted TP Y; or
- X has one or more unactivated waits; or
- X has one or more activated waits.

RTE Algorithm for STNUs

Compute Next Real–Time Execution Decision, RTED [Hunsberger, 2009]

- If $\texttt{EnabledTPs} = \emptyset$ then $\texttt{RTED} = \texttt{Wait}$
- For each $X \in \texttt{EnabledTPs}$:
  - $\texttt{lb}_\texttt{w}(X) = \max\{w \mid \exists(w, C) \in \texttt{ActWaits}(X)\}$ (Max wait for X)
  - $\texttt{glb}(X) = \max\{\texttt{lb}(X), \texttt{lb}_\texttt{w}(X)\}$ (Overall lower bound for X)
- $t_L = \min\{\texttt{glb}(X) \mid X \in \texttt{EnabledTPs}\}$ (Soonest next execution)
- $t_L^* = \max\{\texttt{now}, t_L\}$ (After `now`!)
- $t_U = \min\{\texttt{ub}(X) \mid X \in \texttt{EnabledTPs}\}$ (Latest next execution)
- $\texttt{possWin} = [t_L^*, t_U]$ (Range for next execution)
- t = pick any time from `possWin`
- X = any TP from `EnabledTPs` for which $t \in [\texttt{glb}(X), \texttt{ub}(X)]$
- $\texttt{RTED} = (t, X)$ ("If nothing happens before time t, set $X = t$")

RTE Algorithm for STNUs

Observe Outcome and Update Info

Case 1: A contingent TP C executed at some $\rho <= t$.

- ⋆ Delete all waits labeled by C from relevant `ActWaits` sets
- ⋆ Update time–windows for neighboring TPs (as for STNs)
- ⋆ Update `EnabledTPs` (because of any deleted waits or incoming negative edges to C)
- ⋆ $\texttt{now} = \rho$

RTE Algorithm for STNUs

Observe Outcome and Update Info

Case 2: Nothing happened before time t.

- ⋆ Execute X at time t
- ⋆ Update time–windows for neighboring TPs (as for STNs)
- ⋆ If X is an activation TP:

  Then for each UC edge $Y \xrightarrow{C:-w} X$

  insert $(t + w, C)$ into $\texttt{ActWaits}(Y)$.
- ⋆ Update `EnabledTPs` (due to any incoming neg edges to X)
- ⋆ $\texttt{now} = t$

RTE Algorithm for STNUs

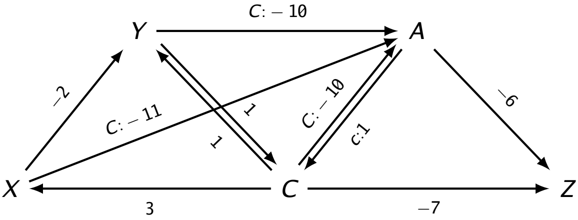
Observe Outcome and Update Info

Case 3: A contingent time–point C
and an executable time–point X both execute at time t.

- ⋆ Combine updates from Cases 2 and 3.

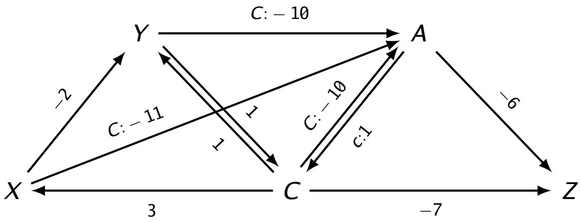
RTE Algorithm for STNUs

Example 1



RTE Algorithm for STNUs

Example 1

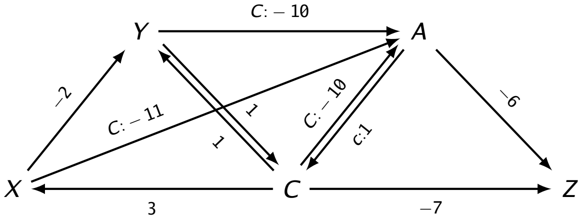


Initialization:

- $\mathcal{U}_X = \{Z, A, X, Y\}$, $\texttt{EnabledTPs} = \{Z\}$, $\texttt{now} = 0$.

RTE Algorithm for STNUs

Example 1

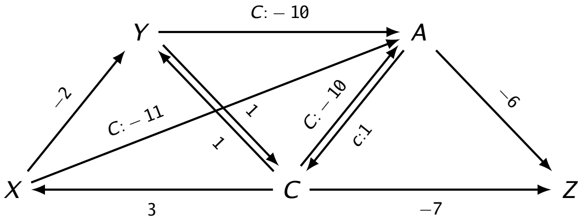


Iteration 1: Compute RTED

- $\mathcal{U}_X = \{Z, A, X, Y\}$, $\texttt{EnabledTPs} = \{Z\}$, $\texttt{now} = 0$.
- $\texttt{possWin} = [0, \inf)$
- $\texttt{RTED} = (0, Z)$

RTE Algorithm for STNUs

Example 1



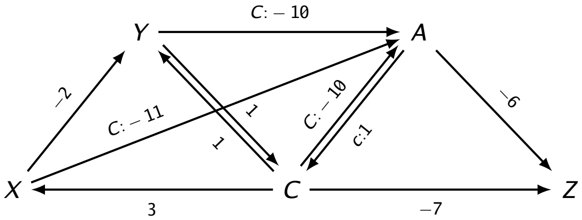
Iteration 1: Observe outcome

- Nothing happens before time 0
- So execute Z at time 0

Executions: $Z = 0$

RTE Algorithm for STNUs

Example 1



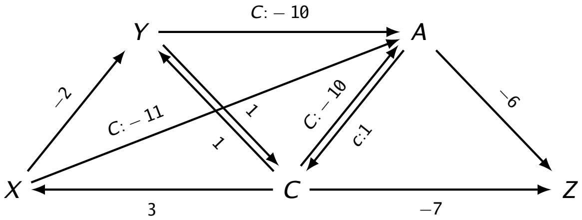
Iteration 1: Update info

- Update `EnabledTPs`: $\texttt{EnabledTPs} = \{A\}$

Executions: $Z = 0$

RTE Algorithm for STNUs

Example 1



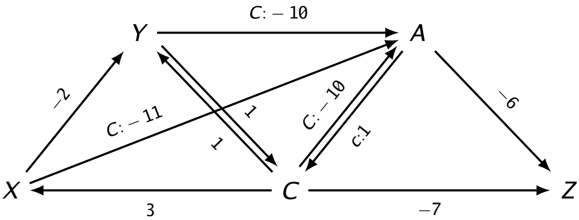
Iteration 2: Compute RTED

- $\mathcal{U}_X = \{A, X, Y\}$, $\texttt{EnabledTPs} = \{A\}$, $\texttt{now} = 0$.
- $\texttt{TW}(A) = [6, \inf)$, $\texttt{possWin} = [6, \inf)$, $\texttt{RTED} = (9, A)$

Executions: $Z = 0$

RTE Algorithm for STNUs

Example 1



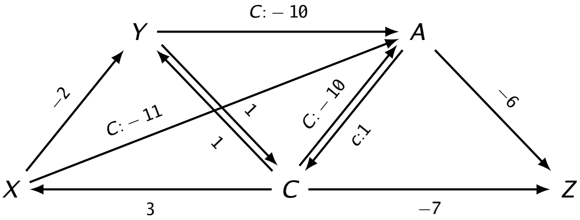
Iteration 2: Observe outcome

- Nothing happens before time 9
- So execute *A* at time 9

Executions: $Z = 0$, $A = 9$

RTE Algorithm for STNUs

Example 1



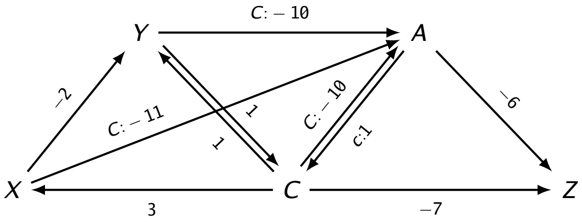
Iteration 2:   Update info

- Update time–windows: —
- $\texttt{ActWaits}(X) = \{(20, C)\}$, $\texttt{ActWaits}(Y) = \{(19, C)\}$
- $\mathcal{U}_x = \{X, Y\}$, $\texttt{EnabledTPs} = \emptyset$, $\texttt{now} = 9$.

Executions: $Z = 0$, $A = 9$

RTE Algorithm for STNUs

Example 1



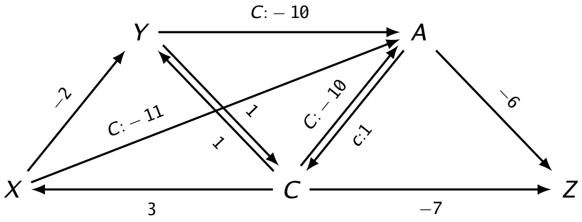
Iteration 3: Compute RTED

- $\mathcal{U}_x = \{X, Y\}$, $\texttt{EnabledTPs} = \emptyset$, $\texttt{now} = 9$.
- Since $\texttt{EnabledTPs} = \emptyset$, $\texttt{RTED} = \texttt{Wait}$

Executions: $Z = \texttt{0}$, $A = 9$

RTE Algorithm for STNUs

Example 1



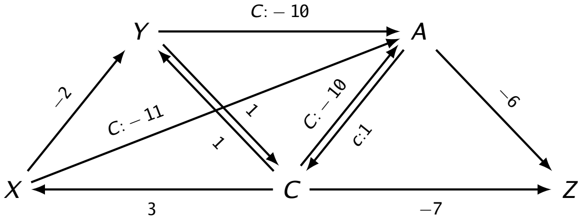
Iteration 3: Observe outcome

- Observe C executing at time 15.

Executions: $Z = 0$, $A = 9$, $C = 15$

RTE Algorithm for STNUs

Example 1



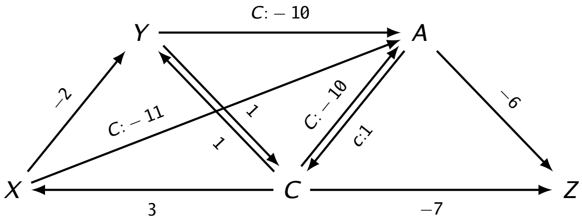
Iteration 3: Update info

- Update time–windows: $\mathtt{TW}(Y) = (-\inf, 16]$, $\mathtt{TW}(X) = (-\inf, 18]$
- Delete waits: $\mathtt{ActWaits}(X) = \emptyset$, $\mathtt{ActWaits}(Y) = \emptyset$
- $\mathcal{U}_x = \{X, Y\}$, $\mathtt{EnabledTPs} = \{Y\}$, $\mathtt{now} = 15$

Executions: $Z = \mathtt{0}$, $A = 9$, $C = 15$

RTE Algorithm for STNUs

Example 1



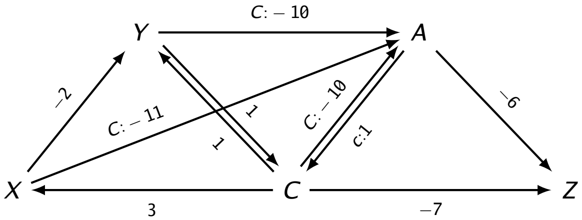
Iteration 4: Compute RTED

- $\texttt{EnabledTPs} = \{Y\}$, $\texttt{TW}(Y) = (-\inf, 16]$, $\texttt{now} = 15$
- $\texttt{possWin} = [15, 16]$, $\texttt{RTED} = (16, Y)$

Executions: $Z = 0$, $A = 9$, $C = 15$

RTE Algorithm for STNUs

Example 1



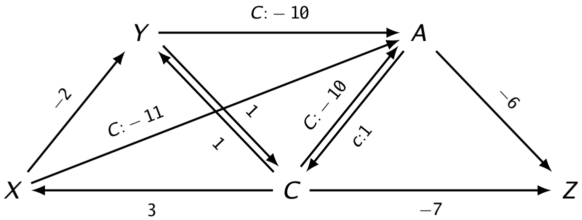
Iteration 4: Observe outcome

- $\texttt{EnabledTPs} = \{Y\}$, $\texttt{TW}(Y) = (-\inf, 16]$, $\texttt{now} = 15$
- $\texttt{possWin} = [15, 16]$, $\texttt{RTED} = (16, Y)$
- Execute Y at time 16

Executions: $Z = 0$, $A = 9$, $C = 15$, $Y = 16$

RTE Algorithm for STNUs

Example 1



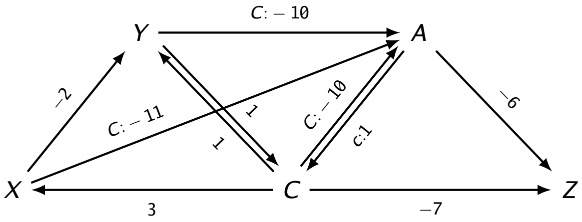
Iteration 4: Update info

- EnabledTPs $= \{X\}$, TW$(X) = [16, 18]$, now $= 16$

Executions: $Z = 0$, $A = 9$, $C = 15$, $Y = 16$

RTE Algorithm for STNUs

Example 1



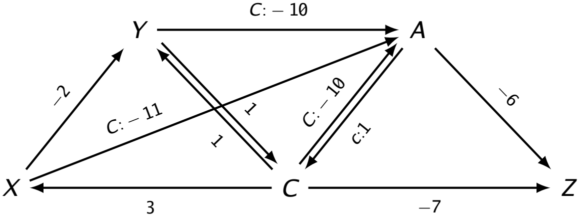
Iteration 5: Compute RTED

- $\mathtt{RTED} = (17, X)$

Executions: $Z = \mathtt{0}$, $A = 9$, $C = 15$, $Y = 16$

RTE Algorithm for STNUs

Example 1



Iteration 5: Observe outcome

- Execute X at 17

Executions: $Z = 0$, $A = 9$, $C = 15$, $Y = 16$, $X = 17$

Dispatchability of STNUs

- Morris [2014, 2016] formally analyzed STNU dispatchability
- THEOREM:

  An STNU is dispatchable if and only if
  all of its projections are dispatchable (as STNs).

Dispatchability of STNUs

- Morris [2014, 2016] formally analyzed STNU dispatchability
- THEOREM:

  An STNU is dispatchable if and only if
  all of its projections are dispatchable (as STNs).
- A projection of an STNU is the STN that results from fixing all of its contingent links to allowable values.

Dispatchability of STNUs

- Morris [2014, 2016] formally analyzed STNU dispatchability
- THEOREM:

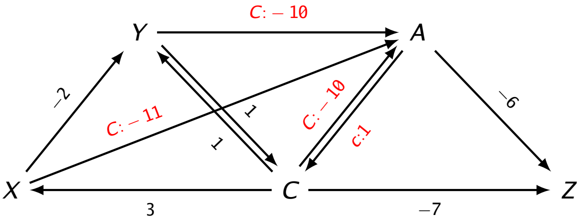
  An STNU is dispatchable if and only if
  all of its projections are dispatchable (as STNs).
- A projection of an STNU is the STN that results from fixing all of its contingent links to allowable values.
- If there are *k* contingent links, then there is a *k*–dimensional space of all the projections.

Dispatchability of STNUs

- Morris [2014, 2016] formally analyzed STNU dispatchability
- THEOREM:

  An STNU is dispatchable if and only if
  all of its projections are dispatchable (as STNs).

The sample STNU:

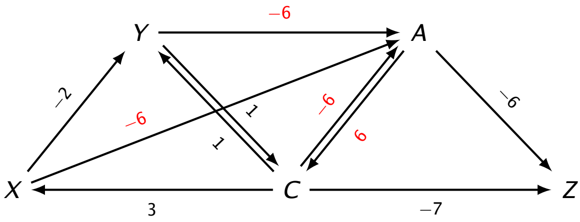


Dispatchability of STNUs

- Morris [2014, 2016] formally analyzed STNU dispatchability
- THEOREM:

  An STNU is dispatchable if and only if
  all of its projections are dispatchable (as STNs).

The projection of that STNU where $C - A = 6$:

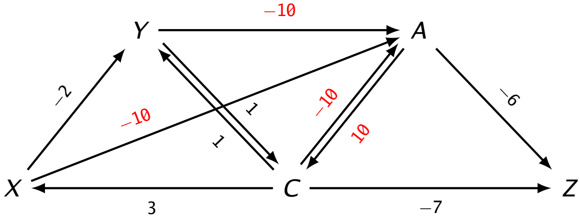


Dispatchability of STNUs

- Morris [2014, 2016] formally analyzed STNU dispatchability
- THEOREM:

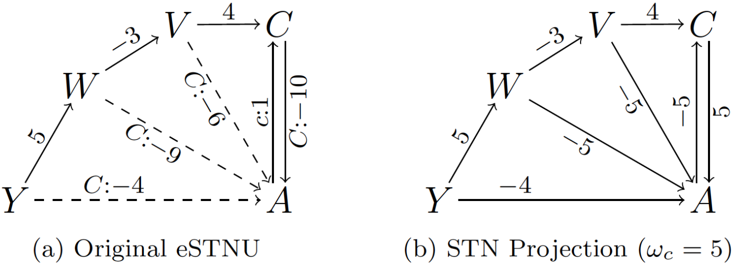
  An STNU is dispatchable if and only if
  all of its projections are dispatchable (as STNs).

The projection of that STNU where $C - A = 10$:



STNU Projection

One more example



(a) Original eSTNU

(b) STN Projection ($\omega_c = 5$)

Real–Time Execution of Dispatchable STNUs

STNU dispatchable iff every projection is

- We don't know in advance which projection will occur.
- However, for any projection, running the STNU version of the RTE algorithm (without knowing what the projection is) is equivalent to running the STN version of the RTE algorithm on that projection, where the durations of the contingent time–points are chosen to match that projection.

Converting STNUs into Dispatchable Form

- Recall that not all consistent STNs are dispatchable.
- But consistent STNs that are vee–path complete (VPC) are dispatchable.
- Similarly, not all DC STNUs are dispatchable.
- But there are two recent algorithms for transforming DC STNUs into equivalent dispatchable forms.

Morris' 2014 Algorithm for STNU Dispatchability

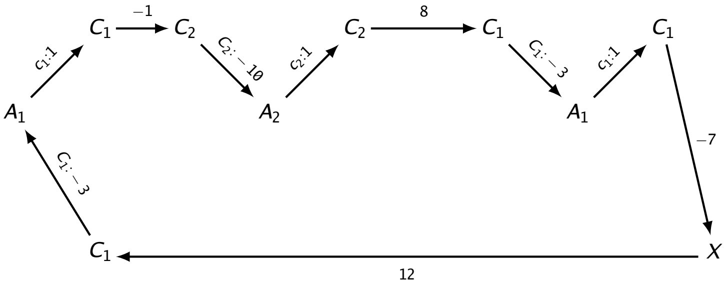
Morris' 2014 Algorithm

Easy to modify to generate dispatchable STNU

- Morris' 2014 DC-checking algorithm does not typically generate a dispatchable STNU.
- In particular, as it back-propagates from negative edges, it only inserts non-negative bypass edges.
- Simply modifying it to insert the negative edges it traverses along the way ensures that the output STNU will be dispatchable.

Morris' 2014 $O(n^3)$ DC-checking algorithm

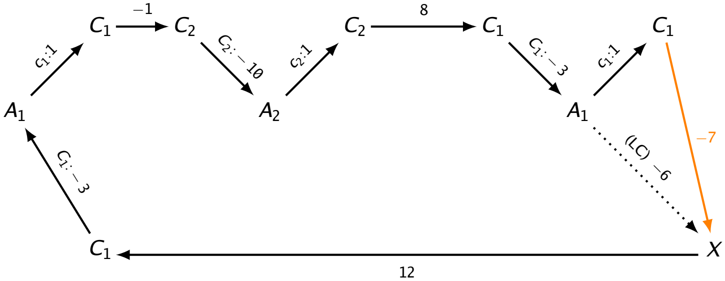
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non-vee-paths it traverses.

Morris' 2014 $O(n^3)$ DC–checking algorithm

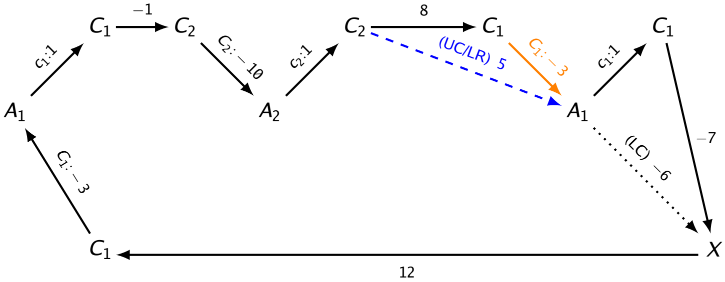
Finding a Semi–Reducible Negative Cycle



- Generates bypass edges for all non–vee–paths it traverses.
- Ensures a vee–path between every connected pair of TPs.

Morris' 2014 $O(n^3)$ DC-checking algorithm

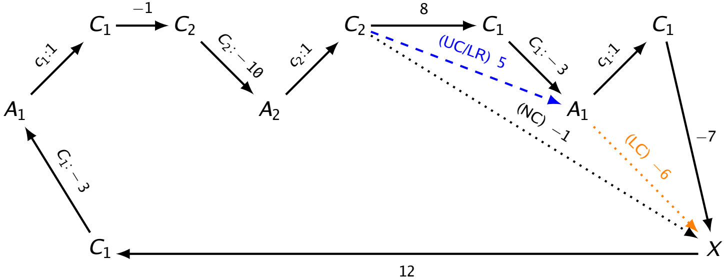
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non-vee-paths it traverses.
- Ensures a vee-path between every connected pair of TPs.
- However, it can add too many edges...

Morris' 2014 $O(n^3)$ DC-checking algorithm

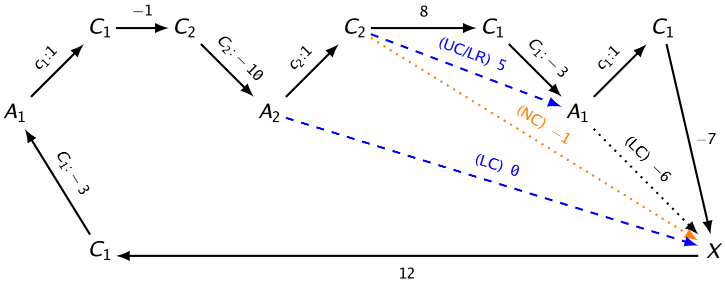
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non-vee-paths it traverses.
- Ensures a vee-path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we've seen.

Morris' 2014 $O(n^3)$ DC-checking algorithm

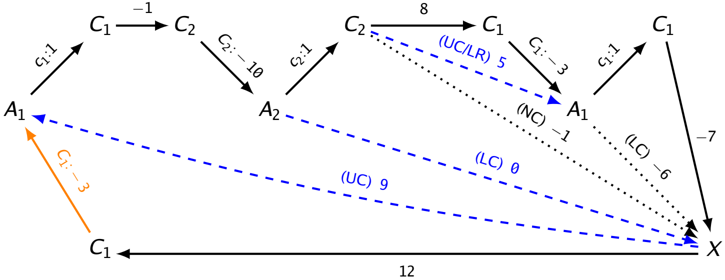
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non-vee-paths it traverses.
- Ensures a vee-path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we've seen.

Morris' 2014 $O(n^3)$ DC-checking algorithm

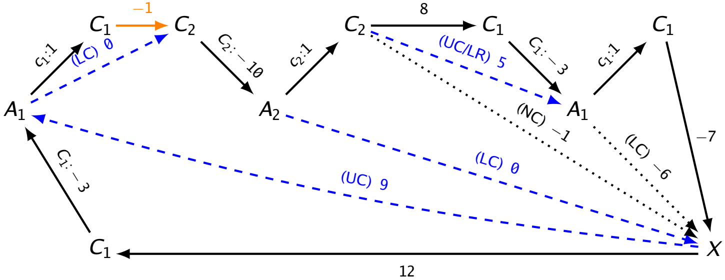
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non–vee–paths it traverses.
- Ensures a vee–path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we've seen.

Morris' 2014 $O(n^3)$ DC–checking algorithm

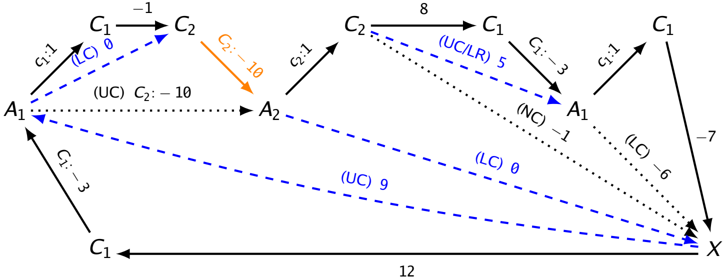
Finding a Semi–Reducible Negative Cycle



- Generates bypass edges for all non–vee–paths it traverses.
- Ensures a vee–path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we've seen.

Morris' 2014 $O(n^3)$ DC-checking algorithm

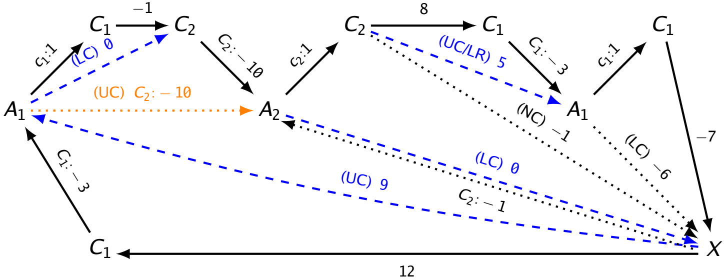
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non-vee-paths it traverses.
- Ensures a vee-path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we've seen.

Morris’ 2014 $O(n^3)$ DC-checking algorithm

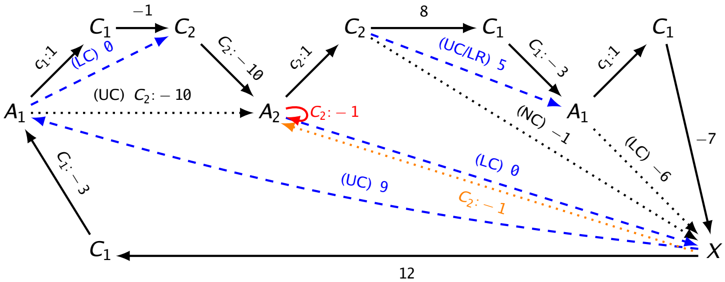
Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non–vee–paths it traverses.
- Ensures a vee–path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we’ve seen.

Morris' 2014 $O(n^3)$ DC-checking algorithm

Finding a Semi-Reducible Negative Cycle



- Generates bypass edges for all non-vee-paths it traverses.
- Ensures a vee-path between every connected pair of TPs.
- However, it can add too many edges...
- For example, running it on an STN will generate far more edges than the STN dispatchability algorithms we've seen.

Faster STNU Dispatchability Alg.

[Hunsberger and Posenato, 2023]

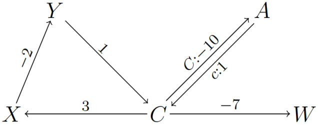
A New Dispatchability Alg. for STNUs: $\mathrm{FD}_{\mathrm{STNU}}$

[Hunsberger and Posenato, 2023]

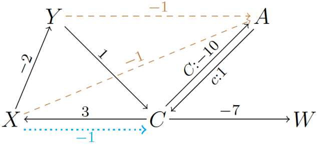
The $\mathrm{FD}_{\mathrm{STNU}}$ algorithm has three phases:

1. Run the `RUL2021` DC-checking algorithm, propagating backward from UC edges aiming to bypass UC edges with ordinary edges and waits.
2. Propagate forward from each LC edge, but only propagating along LO edges, aiming to bypass LC edges with ordinary edges.
3. Run the STN dispatchability algorithm on the ordinary subgraph to make that ordinary subgraph dispatchable as an STN.

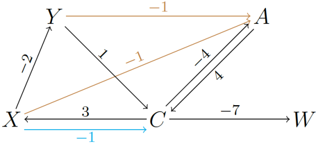
Comparing RUL^{-} and FD_{STNU}



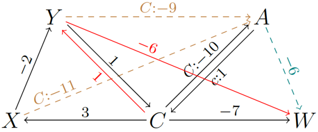
(a) Sample DC STNU



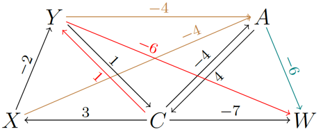
(b) RUL^{-} output



(c) STN Projection, $\omega_c = 4$



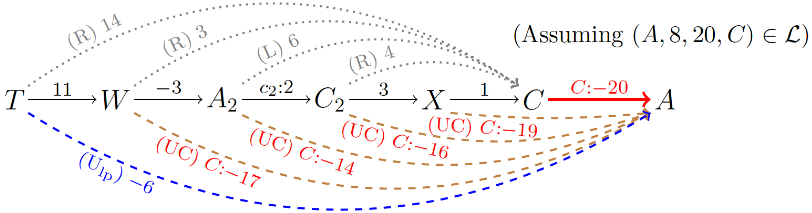
(d) $\mathrm{FD}_{\mathrm{STNU}}$ output



(e) STN Projection, $\omega_c = 4$

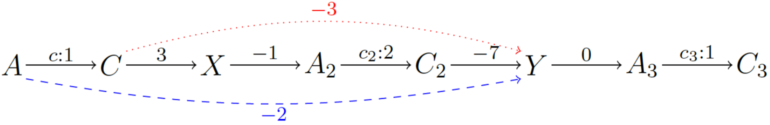
The FD$_{\text{STNU}}$ Algorithm

Phase One: Propagate Backward from UC Edges – RUL2021



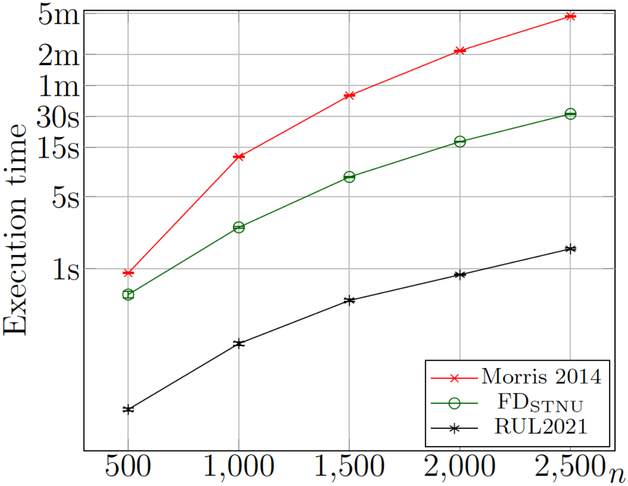
The FD$_{\mathrm{STNU}}$ Algorithm

Phase Two: Propagate Forward from LC Edges – but only along LO edges



Empirical Comparison

FD$_{\text{STNU}}$ vs. Morris14 vs. RUL2021 (only DC)



Related Work

- Managing execution of DC STNUs

  [Hunsberger, 2010, 2015b]

- Using Timed Game Automata (TGAs) to synthesize execution strategies for STNUs

  [Cimatti et al., 2014b]

Conditional Simple Temporal Networks (CSTNs)

Conditional STNs

Motivation

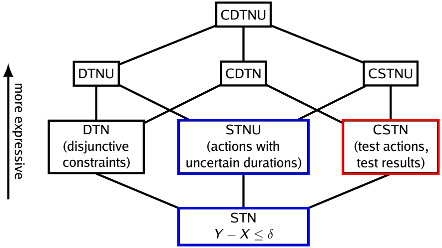
- Many actions generate information (e.g., medical tests, opening a box, monitoring traffic).
- The generated information is generally not known in advance, but discovered in real time.
- Some actions only make sense in certain scenarios (e.g., don't give drug if test result is negative).
- An execution strategy could be more flexible if it could react dynamically to generated information.

Conditional STNs

Motivation (ctd.)

- Many businesses using *workflow management systems* to automate manufacturing processes.
- Hospitals can use workflows to represent possible treatment pathways for a patient.
- CSTNs can serve as the temporal foundation for workflow management systems.

Conditional STNs (CSTNs)*



- Time–points and constraints as in STNs
- Observation time–points generate truth values for propositional letters
- Time–points and constraints labeled by conjunctions of propositional letters

* [Tsamardinos et al., 2003]

Conditional STNs

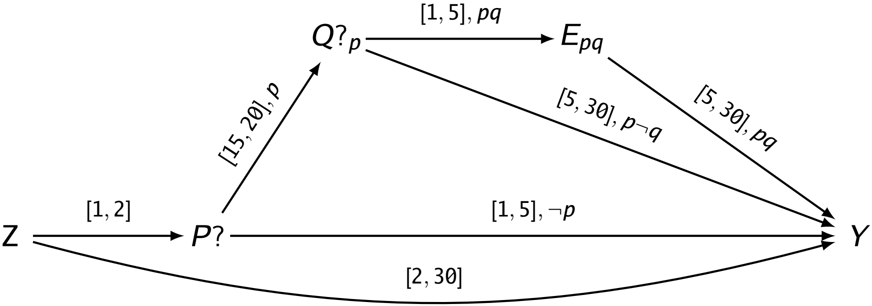
Propositional Labels

- Propositional letters: $p, q, r, s, t, \ldots$
- Each p has corresponding observation time–point, $P?$.
  Executing $P?$ generates truth value for p
- Label: conjunction of literals (e.g., $p(\neg q)r$)
- A scenario specifies values for *all* letters
  (e.g., $p = true$, $q = false$, $r = true$).
- The real scenario is only revealed incrementally.
- Time–points and constraints* can be labeled;
  they only apply in scenarios where their labels are true.

* [Hunsberger et al., 2015]

Conditional STNs

Sample CSTN



$P?$ and $Q?$ represent tests for a patient.
$Q?$ is a *child* of $P?$: only executed in scenarios where $p = true$.

Conditional STNs

Dynamic Consistency

- Dynamic Execution Strategy: execution decisions can react to observations.
- A CSTN is *dynamically consistent* (DC) if there exists a dynamic execution strategy that guarantees that all *relevant* constraints will be satisfied no matter which scenario is incrementally revealed over time.

Conditional STNs

Approaches to DC Checking

- Convert to Disjunctive Temporal Problem

  [Tsamardinos et al., 2003]
- Convert to controller–synth. problem for Timed Game Automaton

  [Cimatti et al., 2014a]
- Convert to Hyper Temporal Network consistency problem

  [Comin and Rizzi, 2015]
- Propagate labeled constraints

  [Hunsberger and Posenato, 2018b, 2020; Hunsberger et al., 2015]

Conditional STNs

DC Checking via Propagation

- Propagate *labeled* constraints
  — Motivated by related work on STNs with choice [Conrad and Williams, 2011]
- Introduce new kind of literals and labels:
  Q–literals (e.g., $p?$) and *Q–labels* (e.g., $p\neg q(r?)s$)
- Analysis of *negative q–loops* and *negative q–stars*

Conditional STNs

Labeled Constraints

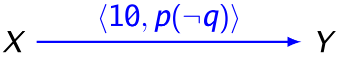
$$X \xrightarrow{\langle \delta, \alpha \rangle} Y$$

$Y - X \leq \delta$ must hold in every scenario where α is true.

(If $\alpha = \boxdot$, then $Y - X \leq \delta$ must hold in all scenarios.)

Conditional STNs

Labeled Constraints



$Y - X \leq 10$ must hold in every scenario where $p(\neg q)$ is true.

Conditional STNs

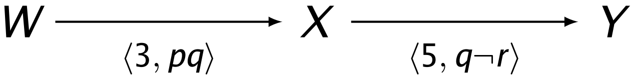
Propagation Rules for CSTNs

Labeled Propagation:	LP and qLP
Label Modification:	R_0 and qR_0
Label "Spreading":	R_3^* and qR_3^*

(The "q" rules propagate q-labeled constraints.)

Conditional STNs

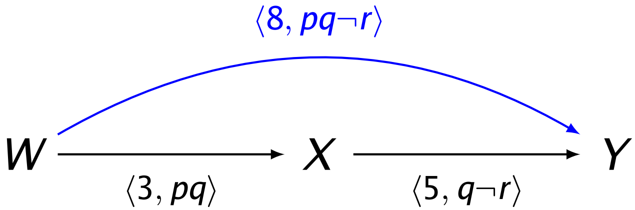
The LP Rule



Labels of two pre–existing edges are conjoined;
The resulting label must be consistent.

Conditional STNs

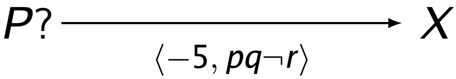
The LP Rule



Labels of two pre–existing edges are conjoined;
The resulting label must be consistent.

Conditional STNs

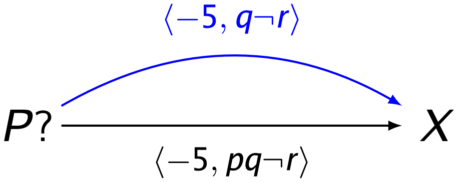
The R_0 Rule



Edge weight must be negative;
Any occurrence of *p* (or $\neg p$) removed from label.

Conditional STNs

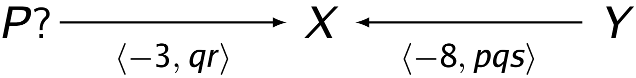
The R_0 Rule



Edge weight must be negative;
Any occurrence of p (or $\neg p$) removed from label.

Conditional STNs

The R_3^* Rule



Pre–existing labels must be consistent;

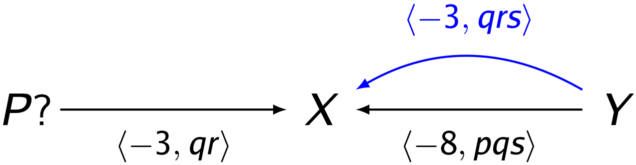
Generated label is conjunction of pre–existing labels
— minus any occurrence of p (or $\neg p$);

Lefthand weight must be negative;

Generated weight is max of pre–existing weights.

Conditional STNs

The R_3^* Rule



Pre–existing labels must be consistent;

Generated label is conjunction of pre–existing labels
— minus any occurrence of p (or $\neg p$);

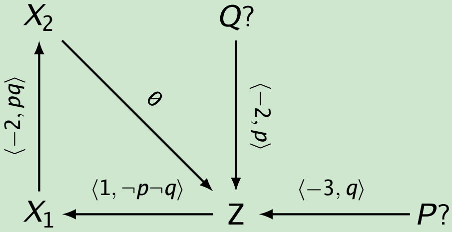
Lefthand weight must be negative;

Generated weight is max of pre–existing weights.

Conditional STNs

Example: Non–DC Instance

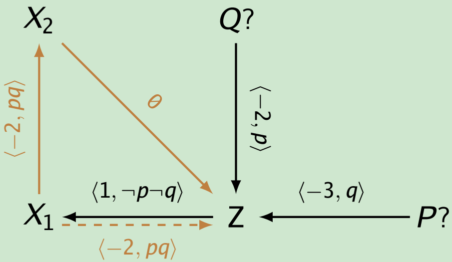
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

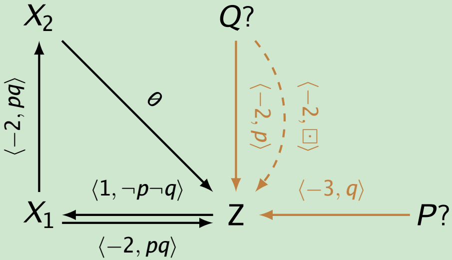
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

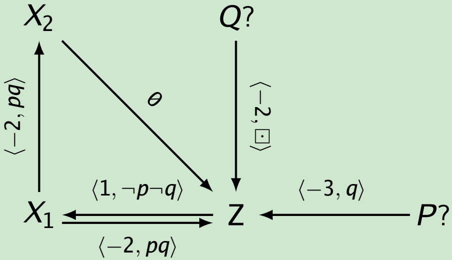
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

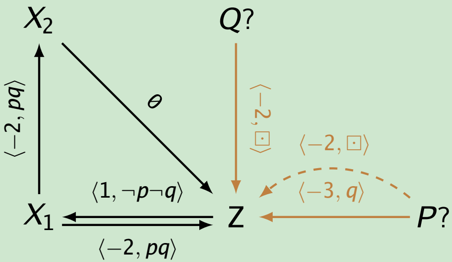
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

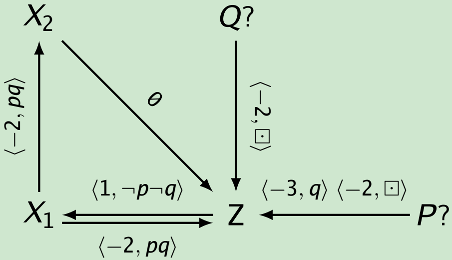
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

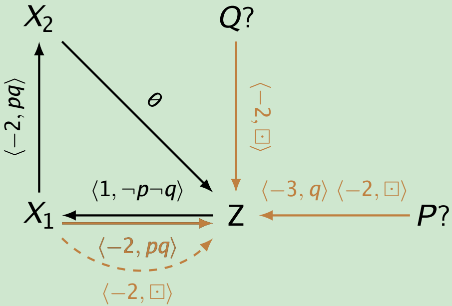
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

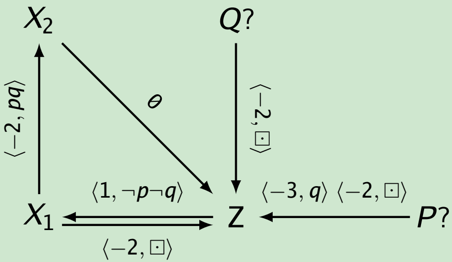
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

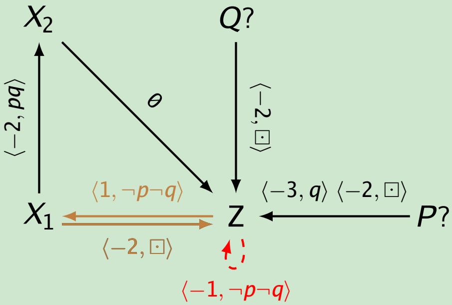
Example 2 (Non–DC Instance)



Conditional STNs

Example: Non–DC Instance

Example 2 (Non–DC Instance)

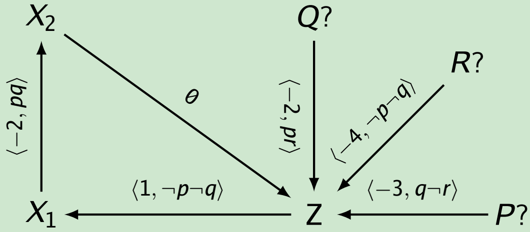


There is a scenario, $\neg p \neg q$, in which there exists a negative loop!
Therefore, the CSTN is not DC!

Conditional STNs
Propagating Q–Labels

- Propagating along consistent labels is insufficient

Example 3 (Non–DC instance, but LP, R_0 and R_3^* not enough)

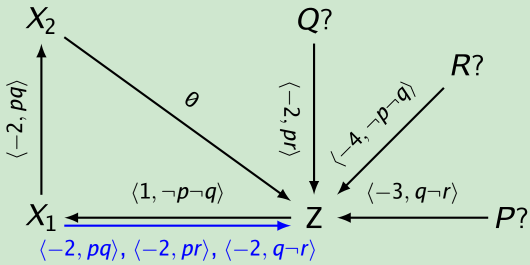


Conditional STNs

Propagating Q–Labels

- Propagating along consistent labels is insufficient

Example 3 (Non–DC instance, but LP, R_0 and R_3^* not enough)



Conditional STNs

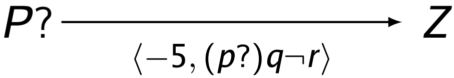
Propagating Q-Labels

- Propagating along consistent labels is insufficient
- *Q-labels*: contain literals such as $p?$.
- A constraint labeled by $p?$ must hold as long as p's value is unknown (i.e., as long as $P?$ remains unexecuted).
- Conjunction operation generalized to cover q-labels:
  $p \wedge \neg p \equiv p?$; $p \wedge p? \equiv p?$; $\neg p \wedge p? \equiv p?$; etc.
- Q-labels only needed on *lower-bound* constraints*
  (i.e., edges pointing at Z).

* [Hunsberger and Posenato, 2018b]

Conditional STNs

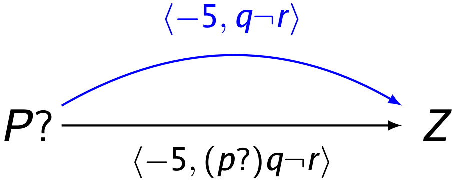
The qR_0 Rule



- Edge must terminate at Z;
- Edge weight must be negative;
- Any occurrence of p (or $\neg p$ or $p?$) removed from label.

Conditional STNs

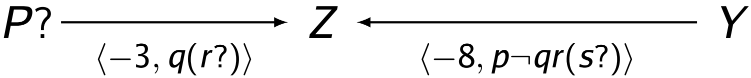
The qR$_\emptyset$ Rule



- Edge must terminate at Z;
- Edge weight must be negative;
- Any occurrence of p (or $\neg p$ or $p?$) removed from label.

Conditional STNs

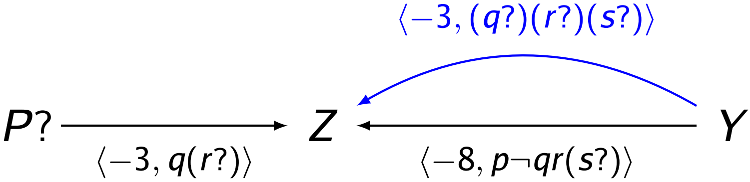
The $\mathbf{qR}_3^*$ Rule



- Labels need not be consistent;
- Lefthand weight must be negative;
- Generated weight is max of pre-existing weights.

Conditional STNs

The qR_3^* Rule



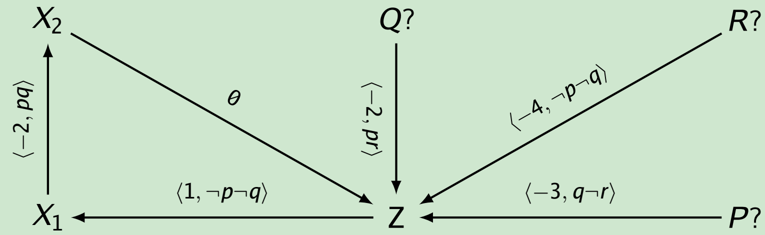
- Labels need not be consistent;
- Lefthand weight must be negative;
- Generated weight is max of pre-existing weights.

Conditional STNs

Propagating Q–Labels

- Propagating along q–labels *is* sufficient!

Example 4 (Non–DC instance confirmed by rules LP, qR_0 and qR_3^*)

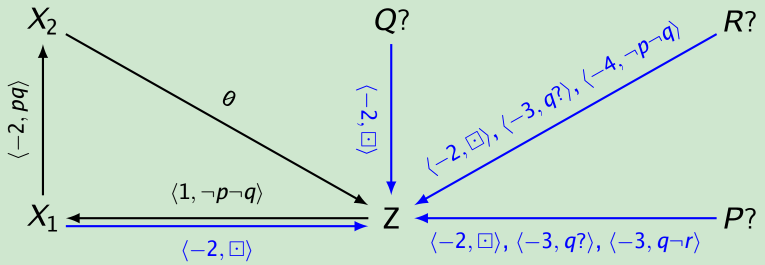


Conditional STNs

Propagating Q–Labels

- Propagating along q–labels *is* sufficient!

Example 4 (Non–DC instance confirmed by rules LP, qR$_0$ and qR$_3^*$)



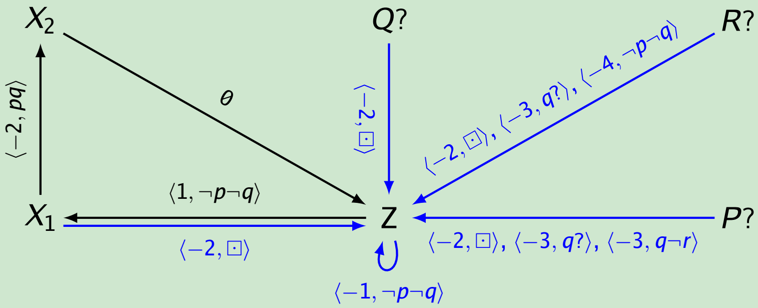
- Incidentally, the blue edges form a “negative q–star”.

Conditional STNs

Propagating Q–Labels

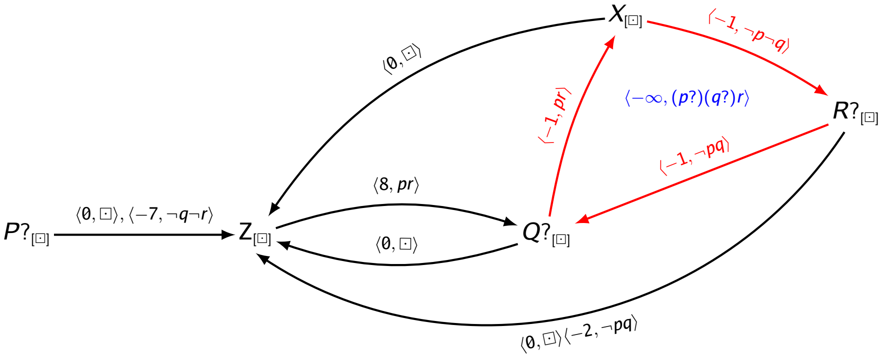
- Propagating along q–labels *is* sufficient!

Example 4 (Non–DC instance confirmed by rules LP, qR_0 and qR_3^*)



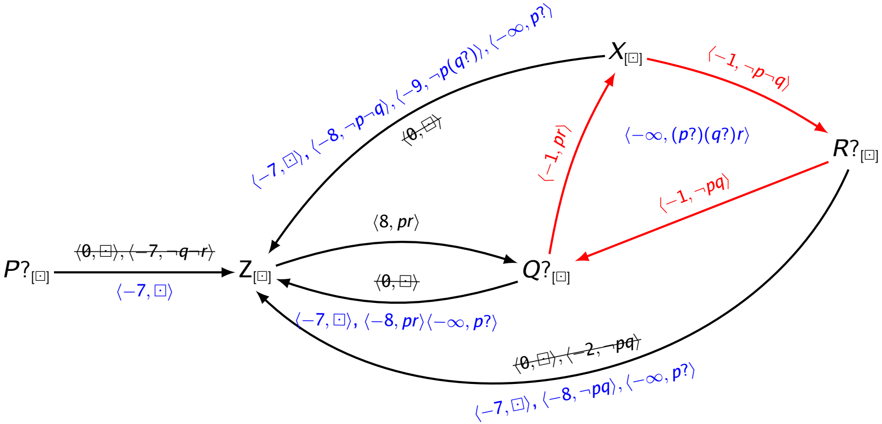
Conditional STNs

Negative Q–Loop Example



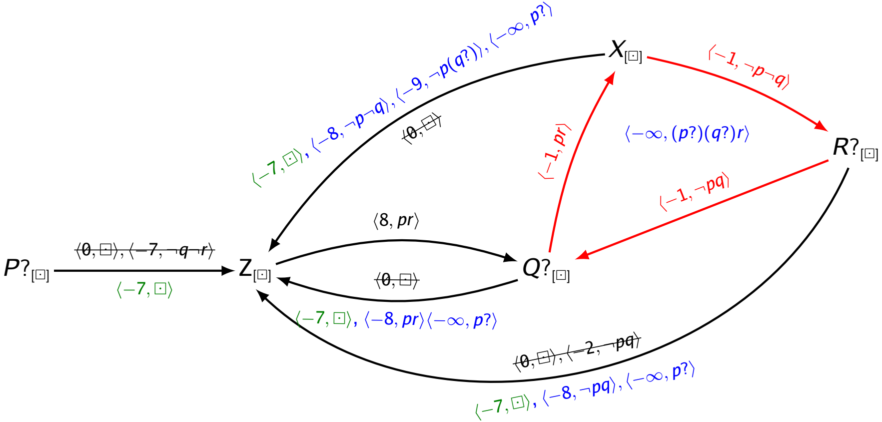
Conditional STNs

Negative Q–Loop Example



Conditional STNs

Negative Q–Loop Example



The *Spreading Lemma*

The minimum lower–bound constraint $\langle -7, \boxdot \rangle$
has spread to *all* unexecuted time–points. [Hunsberger et al., 2015]

Conditional STNs

DC–Checking Algorithm for CSTNs

- The DC–Checking Algorithm exhaustively propagates constraints using LP, qLP , and qR_3^*.
- Returns `NO` if any negative self–loop with a consistent label is ever found; otherwise returns `YES`.
- In positive cases, constructs *earliest–first* strategy, which is viable due to the Spreading Lemma.
- Although exponential–time in the worst case, shown to be practical across a variety of sample networks.

[Hunsberger and Posenato, 2018b; Hunsberger et al., 2015]

Conditional STNs

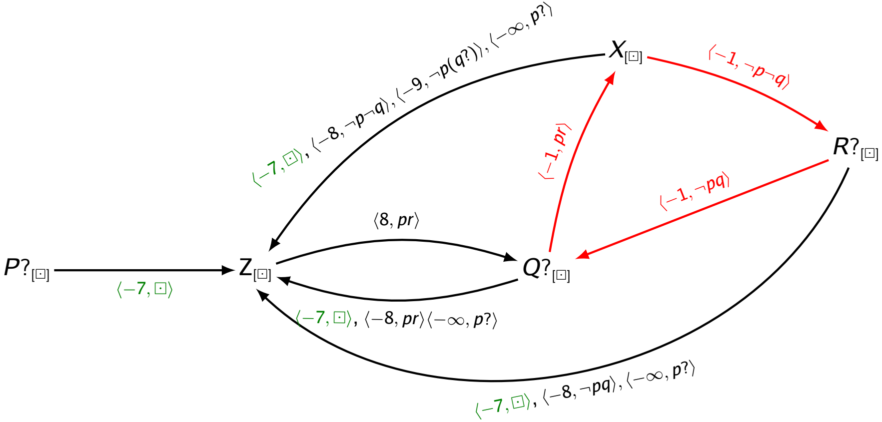
The *Earliest–First* Strategy

- Keep track of *current partial scenario* (CPS), π.
  Initially $\pi = \boxdot$.
- After each execution event, compute *effective lower bound* (ELB) for each as–yet–unexecuted time–point.
- $ELB(X, \pi)$ restricts attention to lower bounds for X whose labels are applicable to π.
- Next time–point to execute is the one with the min. *ELB* value.

Conditional STNs

Sample Execution

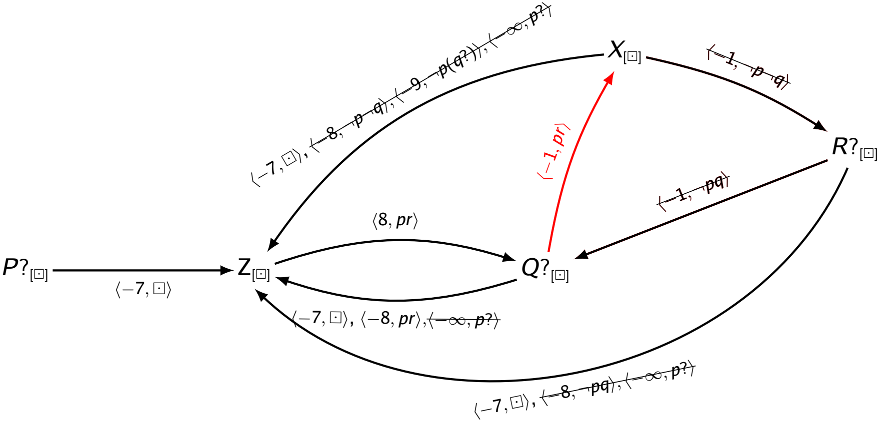
$\pi = \boxdot, \quad Z = 0, \quad ELB(P?, \boxdot) = -7; \quad \text{execute } P? = 7.$



Conditional STNs

Sample Execution

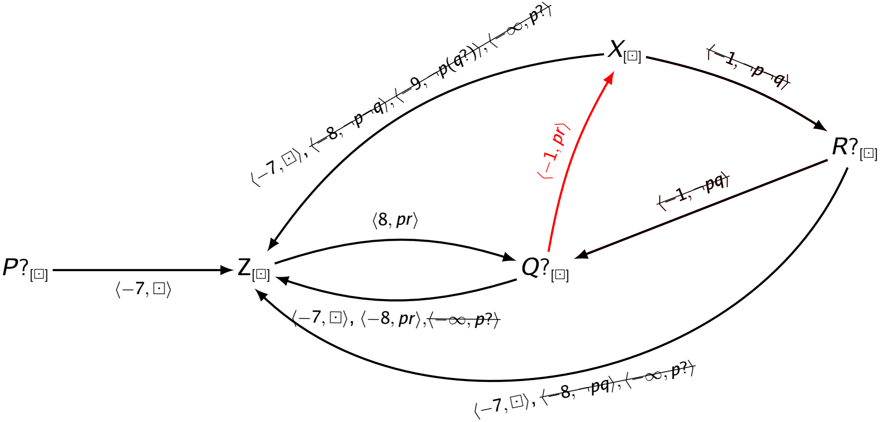
Suppose $p = true$. $\pi = p$; $ELB(X, p) = 7 = ELB(R?, p)$.
So execute $X = 7$ and $R? = 7$.



Conditional STNs

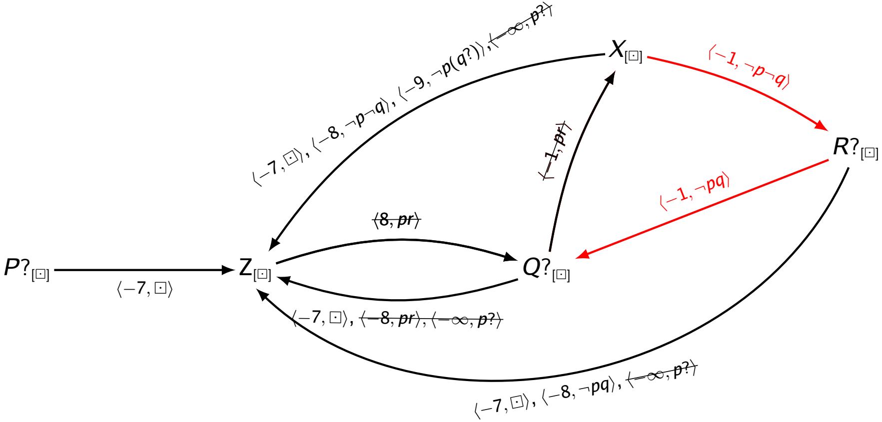
Sample Execution

Suppose $r = true$. $\pi = pr$; $ELB(Q?, p) = 8$.
So execute $Q? = 8$.



Conditional STNs
Alternative Execution

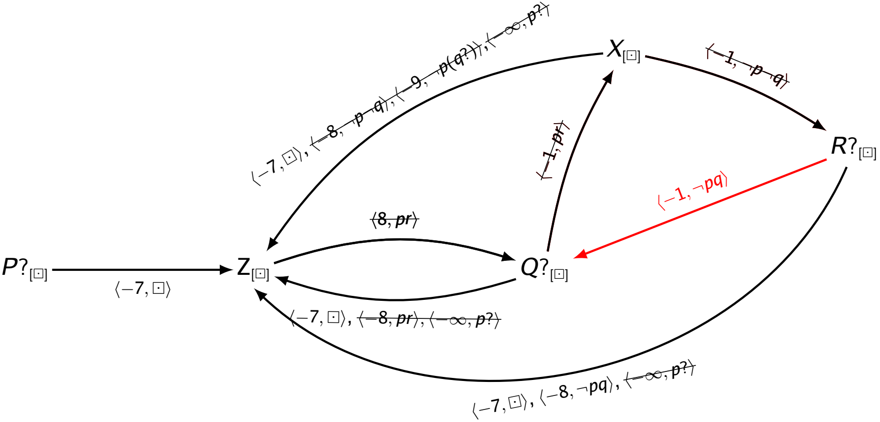
Suppose $p = false$. $\pi = \neg p$; $ELB(Q?, \neg p) = 7$
So execute $Q? = 7$.



Conditional STNs

Alternative Execution

Suppose $q = true$. $\pi = \neg pq$; $ELB(X, \neg pq) = 7$.
So execute $X = 7$. Afterward, execute $R? = 8$.



Conditional STNs

Related Work

- ϵ–dynamic consistency requires bounded reaction time $\epsilon > 0$ [Comin and Rizzi, 2015].
- Propagation–based ϵ–DC checking algorithm [Hunsberger and Posenato, 2016].
- Semantics of instantaneous reactivity for CSTNs [Cairo et al., 2016].
- Streamlined CSTNs [Cairo et al., 2017].

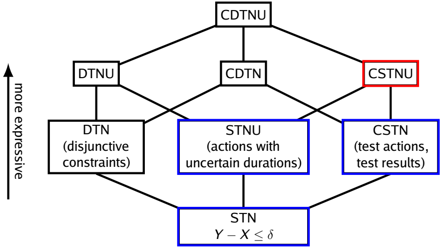
Conditional STNs
CSTN Summary

- Theory of dynamic consistency for CSTNs very solid:
  - instantaneous vs. non-instantaneous reactivity
  - bounded reaction time.
- Several proposed DC-checking algorithms: all exponential

  —but propagation-based algorithm shows promise.
- More work to do on flexible execution.

Conditional STNs with Uncertainty

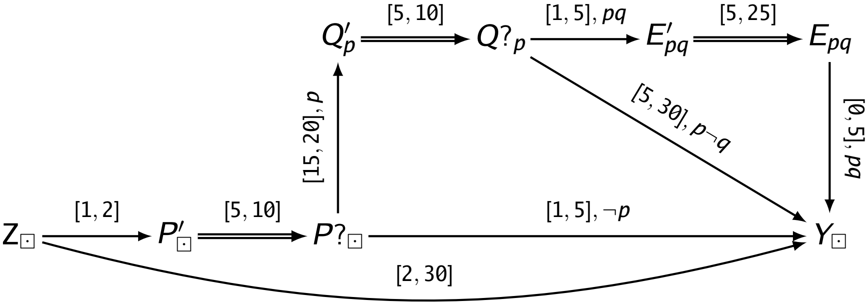
CSTNUs



- A *Conditional Simple Temporal Network with Uncertainty* (CSTNU) combines contingent links from STNUs and observation time-points from CSTNs.

Conditional STNs with Uncertainty

Sample CSTNU



- Contingent links $(P', [5, 10], P?)$ and $(Q', [5, 10], Q?)$ represent tests for a patient.
- Contingent link $(E', [5, 25], E)$ represents the emergency therapy.
- Contingent links have no propositional labels,
  but the labels on their endpoints must be the same.

Conditional STNs with Uncertainty (CSTNUs)

Dynamic Controllability

- Dynamic Execution Strategy: execution decisions may react to observations and contingent durations.
- A CSTNU is *dynamically controllable* if there exists a dynamic execution strategy that guarantees that all *relevant* constraints will be satisfied no matter which scenario is incrementally revealed over time, and no matter how the contingent durations turn out.

Conditional STNs with Uncertainty (CSTNUs)

DC-Checking for CSTNUs

- Convert to Timed Game Automaton
  [Cimatti et al., 2014a]
- Propagate labeled constraints
  [Hunsberger and Posenato, 2018a]

Conditional STNs with Uncertainty (CSTNUs)

DC Checking via Propagation

- Propagate *labeled* constraints as done for CSTN
- Propagate also upper-case and lower-case (contingent) edges as done for STNU considering labeled constraints

Conditional STNs with Uncertainty (CSTNUs)

DC Checking via Propagation

- Propagate *labeled* constraints as done for CSTN
- Propagate also upper–case and lower–case (contingent) edges as done for STNU considering labeled constraints

The mixing of these two kind of propagations requires extending the STNU–concept of upper–case values!

Conditional STNs with Uncertainty (CSTNUs)

DC Checking via Propagation

Generalizing Upper–Case Labels

- Given contingent time–points $C_1, C_2, \ldots, C_k$, their names are called Upper Case (UC) alphabetic–letters (a–letters).
- An *UC alphabetic label (a–label)* is a set of a–letters:
  - is empty, notated as $\diamond$; or
  - contains *one or more* UC a–letters, notated as $C_{i_1} \ldots C_{i_m}$.
- For any UC a–labels $\aleph, \aleph'$, their *conjunction* is given by their union (i.e., $\aleph\aleph' = \aleph \cup \aleph'$).

Conditional STNs with Uncertainty (CSTNUs)

DC Checking via Propagation

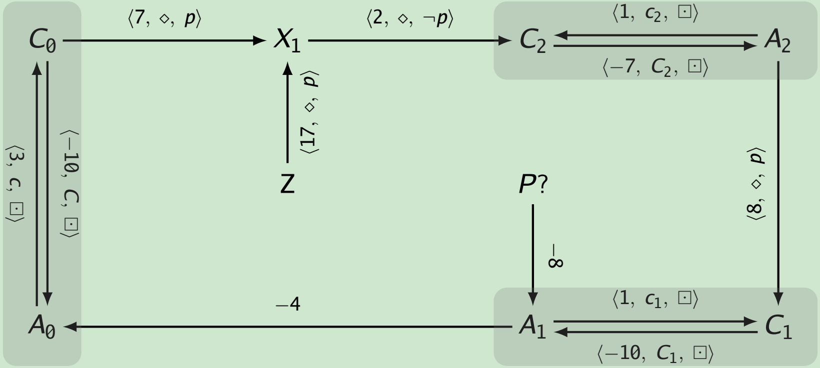
Generalizing labeled values

- Each edge is annotated by a triple, called a *labeled value*
- A *labeled value* is a triple, $\langle \delta,\ \aleph,\ \alpha \rangle$, where:
  - $\delta \in \mathbb{R}$
  - $\aleph$ is an a−label
  - α is a propositional label (from CSTN)

Conditional STNs with Uncertainty (CSTNUs)

DC Checking via Propagation

Example 5 (A CSTNU represented using labeled values)



Edge value v is a shorthand for $\langle v, \diamond, \boxdot\rangle$

Conditional STNs with Uncertainty (CSTNUs)

Propagation Rules for CSTNUs

Forward Upper Case Propagation: z!

Labeled Extended Propagation: zLP/Nc/Uc

Cross Case and Lower Case Propagation: zLc/Cc

Label Removal: zLR

Label Modification: zqR_0

Label “Spreading”: zqR_3^*

Conditional STNs with Uncertainty (CSTNUs)

The z! rule

The *z*! rule can generate edges with multiple UC letters.

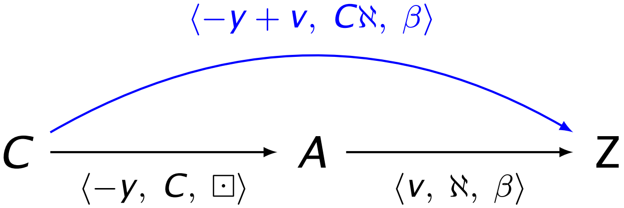
Conditions:

- $-y + v < 0$
- β does not contain unknown literals.

Conditional STNs with Uncertainty (CSTNUs)

The z! rule

The *z*! rule can generate edges with multiple UC letters.

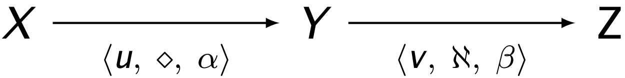


Conditions:

- $-y + v < 0$
- β does not contain unknown literals.

Conditional STNs with Uncertainty (CSTNUs)

The zLP/Nc/Uc Rule

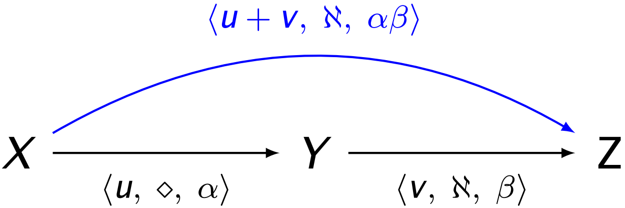


Conditions:

- $u + v < 0$
- α and β must be consistent.

Conditional STNs with Uncertainty (CSTNUs)

The zLP/Nc/Uc Rule



Conditions:

- $u + v < 0$
- α and β must be consistent.

Conditional STNs with Uncertainty (CSTNUs)

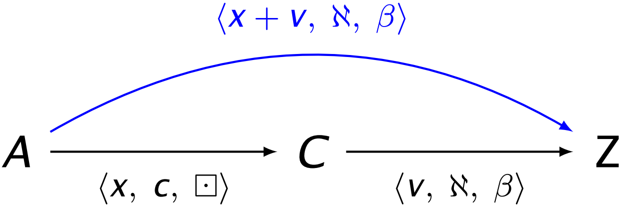
The zLc/Cc rule

Conditions:

- $x + v < 0$
- $C \notin \aleph$
- β does not contain unknown literals.

Conditional STNs with Uncertainty (CSTNUs)

The zLc/Cc rule

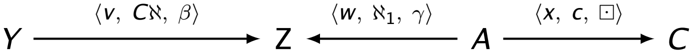


Conditions:

- $x + v < 0$
- $C \notin \aleph$
- β does not contain unknown literals.

Conditional STNs with Uncertainty (CSTNUs)

The zLR rule

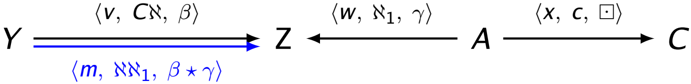


Conditions:

- $C \notin \aleph\aleph_1$
- β, γ can contain unknown literals.

Conditional STNs with Uncertainty (CSTNUs)

The zLR rule



where $m = \max\{v, w - x\}$.

Conditions:

- $C \notin \aleph\aleph_1$
- β, γ can contain unknown literals.

Conditional STNs with Uncertainty (CSTNUs)

The zqR$_0$ rule

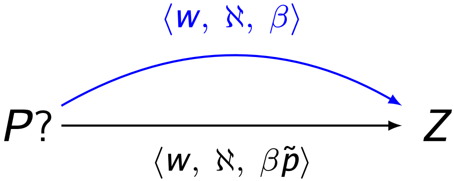
$$P? \xrightarrow[\langle w,\ \aleph,\ \beta\tilde{p}\rangle]{} Z$$

Conditions:

- $w < 0$
- β can contain unknown literals
- $\tilde{p} \in \{p, \neg p, p?\}$

Conditional STNs with Uncertainty (CSTNUs)

The zqR$_0$ rule

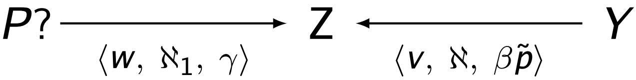


Conditions:

- $w < 0$
- β can contain unknown literals
- $\tilde{p} \in \{p, \neg p, p?\}$

Conditional STNs

The zqR$_3^*$ rule



$P?$

$\langle w,\ \aleph_1,\ \gamma \rangle$

Z

$\langle v,\ \aleph,\ \beta\tilde{p} \rangle$

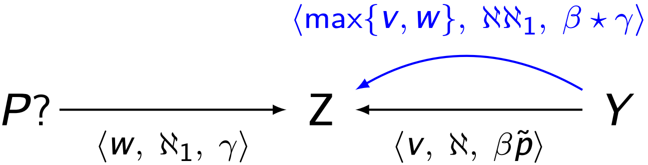
Y

Conditions:

- β, γ can contain unknown literals
- $\tilde{p} \in \{p, \neg p, p?\}$

Conditional STNs

The zqR_3^* rule



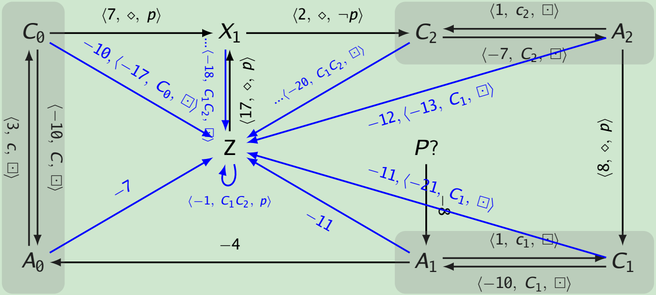
Conditions:

- β, γ can contain unknown literals
- $\tilde{p} \in \{p, \neg p, p?\}$

Conditional STNs with Uncertainty (CSTNUs)

DC Checking via Propagation

Example 6 (A snapshot of CSTNU DC–checking algorithm)



- This CSTNU is not DC: if C_0 occurs at its minimum, while C_1 and C_2 at their maximum, then X cannot be set satisfying all constraints.
- After 123 propagations, the CSTNU contains an explicit negative loop at Z.
- The blue constraints are some of those determined by the algorithm before the negative loop is discovered.

Conditional STNs with Uncertainty (CSTNUs)

DC–Checking Algorithm for CSTNUs

- The DC–Checking Algorithm does exhaustive propagation
- Returns `NO` if any negative loop with a consistent label is ever found; otherwise, returns `YES`.
- In positive cases, constructs *earliest–first* strategy, which is viable due to the spreading lemma for CSTNUs.
- The algorithm has exponential–time complexity in the worst case.
- Currently we are working on some rule optimizations for making it as practical for a variety of sample networks as the CSTN DC–checking algorithm.

Conditional STNs with Uncertainty (CSTNUs)

CSTNU Summary

- Theory of dynamic controllability for CSTNUs has a solid foundation.
- Two competing DC-checking algorithms*, both exponential.
- Propagation-based algorithms show promise, but require further investigation.
- Alternatives to earliest-first strategy?

* [Hunsberger and Posenato, 2018a]

CDTNUs

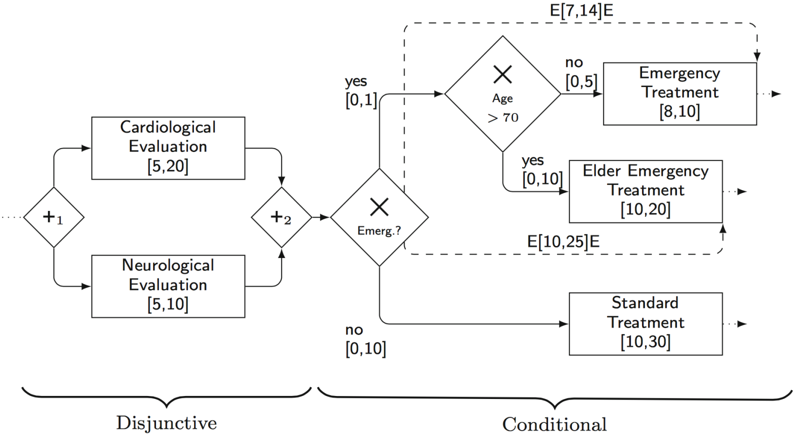
Adding Disjunction to CSTNUs

- A *Conditional Disjunctive Temporal Network with Uncertainty* (CDTNU) augments a CSTNU to include disjunctive constraints.
- Possible to convert the DC-checking problem for CDTNUs into a *controller-synthesis* problem for *Timed Game Automata* (TGAs)*.
- Highlights connections between temporal networks and TGAs, but algorithm not yet practical.

* [Cimatti et al., 2016]

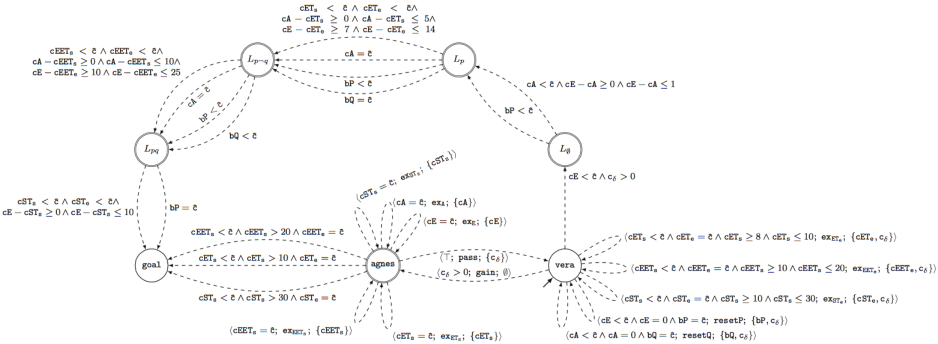
CDTNUs

Sample Workflow



CDTNUs

TGA Encoding of Workflow



References I

M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J.C.-J. Hsu, A. Jonsson, B. Kanefsky, P. Morris, Kanna Rajan, J. Yglesias, B.G. Chafin, W.C. Dias, and P.F. Maldague. Mapgen: mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8-12, 2004.

Luca Anselma, Paolo Terenziani, Stefania Montani, and Alessio Bottrighi. Towards a comprehensive treatment of repetitions, periodicity and temporal constraints in clinical guidelines. *Artificial Intelligence in Medicine*, 38(2):171-195, October 2006. ISSN 0933-3657. doi: 10.1016/j.artmed.2006.03.007.

Michael J. Bannister and David Eppstein. Randomized Speedup of the Bellman-Ford Algorithm. In *9th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 41-47, 2012. ISBN 978-1-61197-213-9. doi: 10.1137/1.9781611973020.6.

Stefan Behnel, Robert W. Bradshaw, and Dag Sverre Seljebotn. Cython tutorial. In *Proceedings of the 8th Python in Science Conference (SciPy 2009)*, 2009.

Richard Bellman. On a routing problem. *Q. Appl. Math.*, 16:87-90, 1958.

J. Benton, Amanda Coles, and Andrew Coles. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 2-10, 2012. URL `https://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699/4708`.

Claudio Bettini, Xiaoyang Sean Wang, and Sushil Jajodia. Temporal reasoning in workflow systems. *Dist. & Paral. Data.*, 11(3):269-306, 2002. doi: 10.1023/A:1014048800604.

References II

John L Bresina, Ari K Jónsson, Paul H Morris, and Kanna Rajan. Activity Planning for the Mars Exploration Rovers. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, page n.n., 2005. URL `https://www.aaai.org/Papers/ICAPS/2005/ICAPS05-005.pdf`.

Massimo Cairo, Carlo Comin, and Romeo Rizzi. Instantaneous Reaction–Time in Dynamic–Consistency Checking of Conditional Simple Temporal Networks. In *23rd Int. Symp. on Temporal Representation and Reasoning (TIME–2016)*, pages 80-89, 2016. doi: 10.1109/TIME.2016.16.

Massimo Cairo, Luke Hunsberger, Roberto Posenato, and Romeo Rizzi. A Streamlined Model of Conditional Simple Temporal Networks – Semantics and Equivalence Results. In *24th Int. Symp. on Temporal Representation and Reasoning (TIME–2017)*, volume 90 of *LIPIcs*, pages 10:1-10:19, 2017. ISBN 978–3–95977–052–1. doi: 10.4230/LIPIcs.TIME.2017.10.

Massimo Cairo, Luke Hunsberger, and Romeo Rizzi. Faster Dynamic Controllablity Checking for Simple Temporal Networks with Uncertainty. In *25th International Symposium on Temporal Representation and Reasoning (TIME–2018)*, volume 120 of *LIPIcs*, pages 8:1-8:16, 2018. doi: 10.4230/LIPIcs.TIME.2018.8.

Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. A Constraint–Based Method for Project Scheduling with Time Windows. *Journal of Heuristics*, 8(1):109-136, January 2002. ISSN 1572–9397. doi: 10.1023/A:1013617802515.

References III

Amedeo Cesta, Gabriella Cortellessa, Riccardo Rasconi, Federico Pecora, Massimiliano Scopelliti, and Lorenza Tiberio. Monitoring elderly people with the ROBOCARE domestic environment: Interaction synthesis and user evaluation. In *Comput. Intell.*, volume 27, pages 60-82, 2011. doi: 10.1111/j.1467-8640.2010.00372.x.

Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, Roberto Posenato, and Marco Roveri. Sound and complete algorithms for checking the dynamic controllability of temporal networks with uncertainty, disjunction and observation. In Andrea Cesta, Carlo Combi, and Francois Laroussinie, editors, *21st Int. Symp. on Temporal Representation and Reasoning (TIME–2014)*, pages 27–36, 2014a. doi: 10.1109/TIME.2014.21.

Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, and Marco Roveri. Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In *28th National Conf. on Artificial Intelligence (AAAI–2014)*, 2014b.

Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, Roberto Posenato, and Marco Roveri. Dynamic controllability via timed game automata. *Acta Informatica*, 53(6-8):681–722, 2016. ISSN 1432-0525. doi: 10.1007/s00236-016-0257-2.

Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with Problems Requiring Temporal Coordination. In *Proceedings of the Twenty–Third AAAI Conference on Artificial Intelligence*, pages 892-897, 2008. URL `https://www.aaai.org/Papers/AAAI/2008/AAAI08-142.pdf`.

References IV

Carlo Combi and Roberto Posenato. Controllability in temporal conceptual workflow schemata. In *Business Process Management, 7th International Conference, BPM 2009*, volume 5701 of *LNCS*, pages 64-79. Springer, 2009. ISBN 3-642-03847-6. doi: 10.1007/978-3-642-03848-8_6.

Carlo Combi, Matteo Gozzi, Barbara Oliboni, Jose M. Juarez, and Roque Marin. Temporal similarity measures for querying clinical workflows. *Artif. Intell. Med.*, 46(1):37-54, 2009. ISSN 09333657. doi: 10.1016/j.artmed.2008.07.013.

Carlo Comin and Romeo Rizzi. Dynamic consistency of conditional simple temporal networks via mean payoff games: a singly-exponential time dc-checking. In *22st Int. Symp. on Temporal Representation and Reasoning (TIME 2015)*, pages 19-28, 2015. doi: 10.1109/TIME.2015.18.

Patrick R. Conrad and Brian C. Williams. Drake: An efficient executive for temporal plans with choice. *J. of Artificial Intelligence Research (JAIR)*, 42:607-659, 2011. URL `http://dx.doi.org/10.1613/jair.3478`.

Rishabh Dabral, Anurag Mundhada, Uday Kusupati, Safeer Afaque, Abhishek Sharma, and Arjun Jain. Learning 3D Human Pose from Structure and Motion. In *Computer Vision–ECCV*, pages 679-696, 2018. ISBN 978-3-030-01240-3. doi: 10.1007/978-3-030-01240-3_41.

Rina Dechter, Itay Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49 (1-3):61-95, 1991. doi: 10.1016/0004-3702(91)90006-6.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 (1):269-271, 1959. ISSN 0945-3245. doi: 10.1007/BF01386390.

References V

Georg Duftschmid, Silvia Miksch, and Walter Gall. Verification of temporal scheduling constraints in clinical practice guidelines. *Artif. Intell. Med.*, 25(2):93-121, 2002. ISSN 09333657. doi: 10.1016/S0933-3657(02)00011-8.

Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962. ISSN 0001-0782. doi: 10.1145/367766.368168.

Lester R. Ford and D. R. Fulkerson. *Flows in networks*, volume 59. Princeton University Press, 1962.

Jeremy Frank and Ari Jónsson. Constraint–Based Attribute and Interval Planning. *Constraints*, 8(4):339-364, 2003. ISSN 13837133. doi: 10.1023/A:1025842019552.

Malik Ghallab. On chronicles: Representation, on–line recognition and learning. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 597-606. Morgan Kaufmann, 1996.

Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *16th International Symposium on Temporal Representation and Reasoning (TIME–2009)*, pages 155-162, 2009. doi: 10.1109/TIME.2009.25.

Luke Hunsberger. A fast incremental algorithm for managing the execution of dynamically controllable temporal networks. In Nicolas Markey and Jef Wijsen, editors, *17th International Symposium on Temporal Representation and Reasoning (TIME–2010)*, pages 121-128, 2010. doi: 10.1109/TIME.2010.16.

References VI

Luke Hunsberger. Magic Loops in Simple Temporal Networks with Uncertainty-Exploiting Structure to Speed Up Dynamic Controllability Checking. In *5th International Conference on Agents and Artificial Intelligence (ICAART-2013)*, volume 2, pages 157-170, 2013.

Luke Hunsberger. A faster algorithm for checking the dynamic controllability of simple temporal networks with uncertainty. In *6th International Conference on Agents and Artificial Intelligence (ICAART-2014)*, 2014a.

Luke Hunsberger. Magic Loops and the Dynamic Controllability of Simple Temporal Networks with Uncertainty. In Joaquim Filipe and Ana Fred, editors, *Agents and Artificial Intelligence*, volume 449 of *Communications in Computer and Information Science (CCIS)*, pages 332-350, 2014b. doi: 10.1007/978-3-662-44440-5_20.

Luke Hunsberger. New techniques for checking dynamic controllability of simple temporal networks with uncertainty. In B. Duval, J. van den Herik, S. Loiseau, and J. Filipe, editors, *6th International Conference on Agents and Artificial Intelligence (ICAART-2014), Revised Selected Papers*, volume 8946 of *LNCS*, pages 170-193. 2015a.

Luke Hunsberger. Efficient execution of dynamically controllable simple temporal networks with uncertainty. *Acta Informatica*, 53(2):89-147, 2015b. doi: 10.1007/s00236-015-0227-0.

Luke Hunsberger and Roberto Posenato. Checking the dynamic consistency of conditional temporal networks with bounded reaction times. In Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *26th Int. Conf. on Automated Planning and Scheduling, ICAPS 2016*, pages 175-183, 2016. URL `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13108`.

References VII

Luke Hunsberger and Roberto Posenato. Sound–and–Complete Algorithms for Checking the Dynamic Controllability of Conditional Simple Temporal Networks with Uncertainty. In *25th Int. Symp. on Temporal Representation and Reasoning (TIME–2018)*, volume 120 of *LIPIcs*, pages 14:1-14:17, 2018a. doi: 10.4230/LIPIcs.TIME.2018.14.

Luke Hunsberger and Roberto Posenato. Simpler and Faster Algorithm for Checking the Dynamic Consistency of Conditional Simple Temporal Networks. In *26th Int. Joint Conf. on Artificial Intelligence, (IJCAI–2018)*, pages 1324-1330, 2018b. doi: 10.24963/ijcai.2018/184.

Luke Hunsberger and Roberto Posenato. Faster Dynamic–Consistency Checking for Conditional Simple Temporal Networks. In *30th Int. Conf. on Automated Planning and Scheduling, ICAPS 2020*, volume 30, pages 152-160, 2020. URL `https://www.aaai.org/ojs/index.php/ICAPS/article/view/6656`.

Luke Hunsberger and Roberto Posenato. Speeding up the RUL^- Dynamic–Controllability–Checking Algorithm for Simple Temporal Networks with Uncertainty. In *36th AAAI Conference on Artificial Intelligence (AAAI–22)*, volume 36–9, pages 9776-9785. AAAI Pres, 2022. doi: 10.1609/aaai.v36i9.21213.

Luke Hunsberger and Roberto Posenato. A Faster Algorithm for Converting Simple Temporal Networks with Uncertainty into Dispatchable Form. *Information and Computation*, 293 (105063):1-21, 2023. ISSN 0890–5401. doi: 10.1016/j.ic.2023.105063.

References VIII

Luke Hunsberger, Roberto Posenato, and Carlo Combi. A Sound–and–Complete Propagation–based Algorithm for Checking the Dynamic Consistency of Conditional Simple Temporal Networks. In Fabio Grandi, Martin Lange, and Alessio Lomuscio, editors, *22nd Int. Symp. on Temporal Representation and Reasoning (TIME–2015)*, pages 4-18, 2015. doi: 10.1109/TIME.2015.26.

Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. of ACM*, 24 (1):1-13, 1977. ISSN 0004–5411. doi: 10.1145/321992.321993.

Ari K Jonsson, Paul H Morris, Nicola Muscettola, and Kanna Rajan. Planning in Interplanetary Space: Theory and Practice. In *Proceeding of AIPS 2000*, page 10, 2000. ISBN 1–57735–111–8. URL `https://www.aaai.org/Papers/AIPS/2000/AIPS00-019.pdf`.

Phil Kim, Brian C. Williams, and Mark Abramson. Executing reactive, model–based programs through graph–based temporal planning. In *18th Int. Joint Conf. on Artificial Intelligence (IJCAI–2001)*, pages 487-493, 2001.

Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210-250, April 2018. ISSN 1572–9354. doi: 10.1007/s10601-018-9281-x.

Solange Lemai and Felix Ingrand. Interleaving Temporal Planning and Execution in Robotics Domains. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 617-622, 2004. URL `https://www.aaai.org/Papers/AAAI/2004/AAAI04-098.pdf`.

References IX

Renaud Masson, Fabien Lehuédé, and Olivier Péton. The Dial–A–Ride Problem with Transfers. *Computers & Operations Research*, 41:12-23, 2014. ISSN 0305–0548. doi: 10.1016/j.cor.2013.07.020.

Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. A deliberative architecture for AUV control. In *Proc. – IEEE Int. Conf. Robot. Autom.*, pages 1049-1054, 2008. ISBN 9781424416479. doi: 10.1109/ROBOT.2008.4543343.

Paul Morris. A Structural Characterization of Temporal Dynamic Controllability. In *Principles and Practice of Constraint Programming (CP–2006)*, volume 4204, pages 375-389, 2006. doi: 10.1007/11889205_28.

Paul Morris. Dynamic controllability and dispatchability relationships. In *CPAIOR 2014*, volume 8451 of *LNCS*, pages 464-479. Springer, 2014. doi: 10.1007/978-3-319-07046-9_33.

Paul Morris. The Mathematics of Dispatchability Revisited. In *26th International Conference on Automated Planning and Scheduling (ICAPS–2016)*, pages 244-252, 2016.

Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *IJCAI 2001: Proc. of the 17th international joint conference on Artificial intelligence*, volume 1, pages 494-499, 2001.

Paul H. Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *20th National Conference on Artificial Intelligence (AAAI–2005)*, pages 1193-1198, 2005. URL `https://www.aaai.org/Papers/AAAI/2005/AAAI05-189.pdf`.

References X

N. Muscettola, P.P. Nayak, B. Pell, and B.C. Williams. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1–2):5-47, 1998a. doi: 10.1016/s0004-3702(98)00068-x.

Nicola Muscettola, Paul H. Morris, and Ioannis Tsamardinos. Reformulating Temporal Plans for Efficient Execution. In *6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR–1998)*, pages 444-452, 1998b.

Wheeler Ruml, Minh B. Do, and Markus P.J. Fromherz. On–line planning and scheduling for high–speed manufacturing. In *ICAPS 2005 – Proc. 15th Int. Conf. Autom. Plan. Sched.*, pages 30-39, 2005. ISBN 1577352203.

Stephen F. Smith, Anthony Gallagher, and Terry Zimmerman. Distributed management of flexible times schedules. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 1-8. ACM, May 2007. ISBN 978–81–904262–7–5. doi: 10.1145/1329125.1329215.

Ioannis Tsamardinos, Nicola Muscettola, and Paul Morris. Fast Transformation of Temporal Plans for Efficient Execution. In *15th National Conf. on Artificial Intelligence (AAAI–1998)*, pages 254-261, 1998.

Ioannis Tsamardinos, Thierry Vidal, and Martha E. Pollack. CTP: A new constraint–based formalism for conditional, temporal planning. *Constraints*, 8:365-388, 2003. doi: 10.1023/A:1025894003623.

Stephen Warshall. A Theorem on Boolean Matrices. *J. of the ACM*, 9(1):11-12, 1962.

References XI

Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970. ISSN 0033-569X. doi: 10.1090/qam/253822.

Hyun Joong Yoon and Doo Yong Lee. Online scheduling of integrated single-wafer processing tools with temporal constraints. *IEEE Trans. Semicond. Manuf.*, 18(3):390–398, August 2005. ISSN 08946507. doi: 10.1109/TSM.2005.852103.

H. L. S. Younes and R. G. Simmons. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20:405–430, December 2003. ISSN 1076-9757. doi: 10.1613/jair.1136.

Li Zhou, Genevieve B. Melton, Simon Parsons, and George Hripcsak. A temporal constraint structure for extracting temporal information from clinical narrative. *Journal of Biomedical Informatics*, 39(4):424–439, 2006. ISSN 1532-0464. doi: 10.1016/j.jbi.2005.07.002. URL `https://www.sciencedirect.com/science/article/pii/S1532046405000730`.