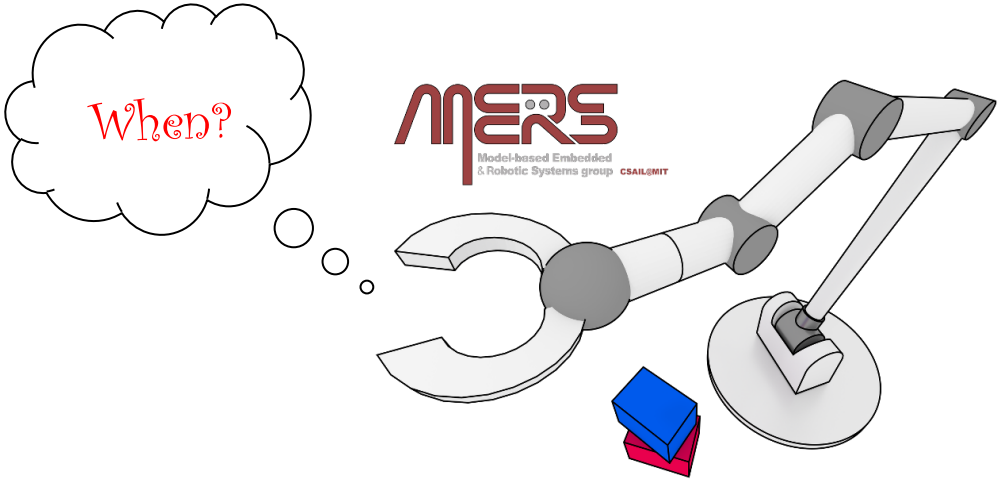
Non-determistic Programs with Flexible Time



Brian Williams

Feb 5, 2020

Contributions:

Patrick Conrad

Simon Fang

Steve Levine

Paul Morris

Nicola Muscettola

Pedro Santana

Julie Shah

John Stedl

Andrew Wang

Brian Williams

Yuening Zhang

Assignments

Homework:

- Pset 0: Jupyter Intro + Mini-research due next Monday 2/10 @ 11:59 pm
- Remember to tell us your (non-MIT) GitHub username! https://forms.gle/J1tRatm3yQRdY4dW6

References:

Today:

- Dechter, R., I. Meiri, J. Pearl, “Temporal Constraint Networks,” Artificial Intelligence, 49, pp. 61-95,1991.
- Muscettola, N., P. Morris and I. Tsamardinos, “Reformulating Temporal Plans for Efficient Execution.” Intl Conf. on Knowledge Representation and Reasoning (KRR), 1998

Next Lecture:

- Vidal, T and Fargier, H., "Handling contingency in temporal constraint networks: from consistency to controllability." Journal of Exp. & Theo. AI 11.1 (1999): 23-45.
- Fang, C., Yu, P., and Williams, B., “Chance-constrained Probabilistic Simple Temporal Problems.” AAAI, 2014.
- Wang, A. and Williams, B., “Chance-constrained Scheduling via Conflict-directed Risk Allocation.” AAAI, 2015.

How do we program robots to be robust?

Programming cognitive robots

- Its not about writing Python/C/C++ programs for specific tasks. (see your advanced lectures).

- Its about a vision of how we write programs for robots, where
  - we set goals,
  - robots decide how to act in context.

- Its about giving robots the ability to execute our goal-directed programs **robustly**.

A little more context…

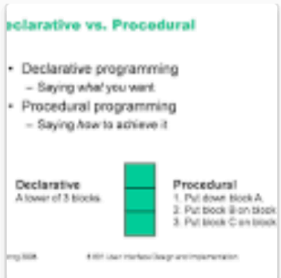
Programming paradigms

Object-oriented prog…

Functional programming

Procedural programming

Imperative programming



Declarative programming

Logic programming

- When programming computers we think of paradigms.
- What are the programming paradigms for robots?

Programming languages

Languages are classified by paradigms

- Haskell is a functional programming language
- Many support multiple paradigms - C++, Python, Java

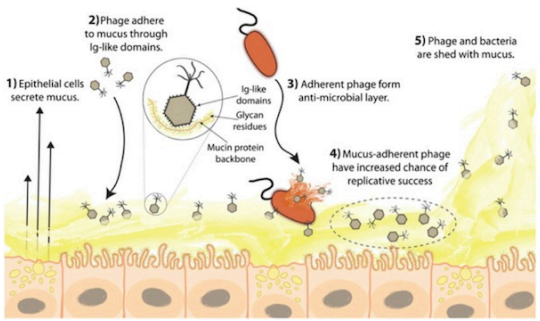
Our Reactive Model-based Programming Language (RMPL) supports model-based autonomy:

- Multiple robot programming paradigms
- goal-directed,
- decision-theoretic,
- risk-bounded,
- . . .

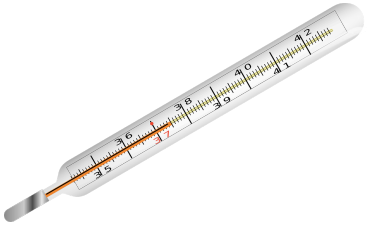
Inspiration: Human defense system

(as viewed by computer scientists)

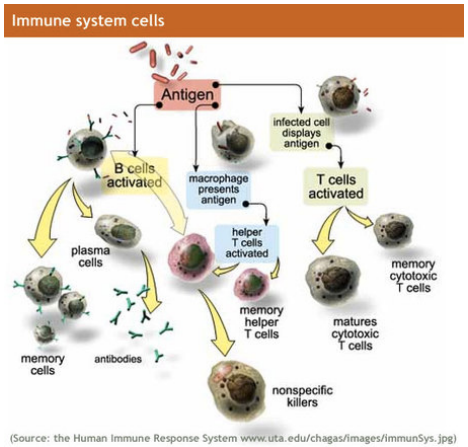
- Multiple layers of defense
- Physical barriers
- Inhospitable environments
- Immune system
- Tolerance
- Death



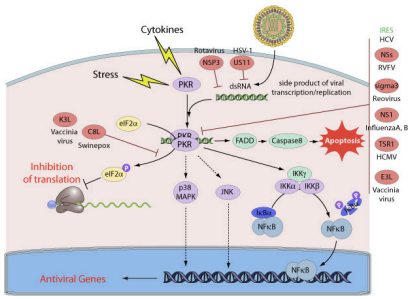
Skin



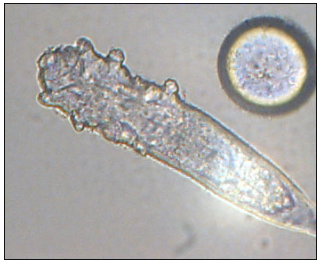
High fever



Immune system



Self-inhibition



Drug-like molecules

Robot Programs Should Facilitate Layers of Defense

Idea: A language executive that achieves robustness through decision layers that are:

Suspicious
Monitor states and goals.

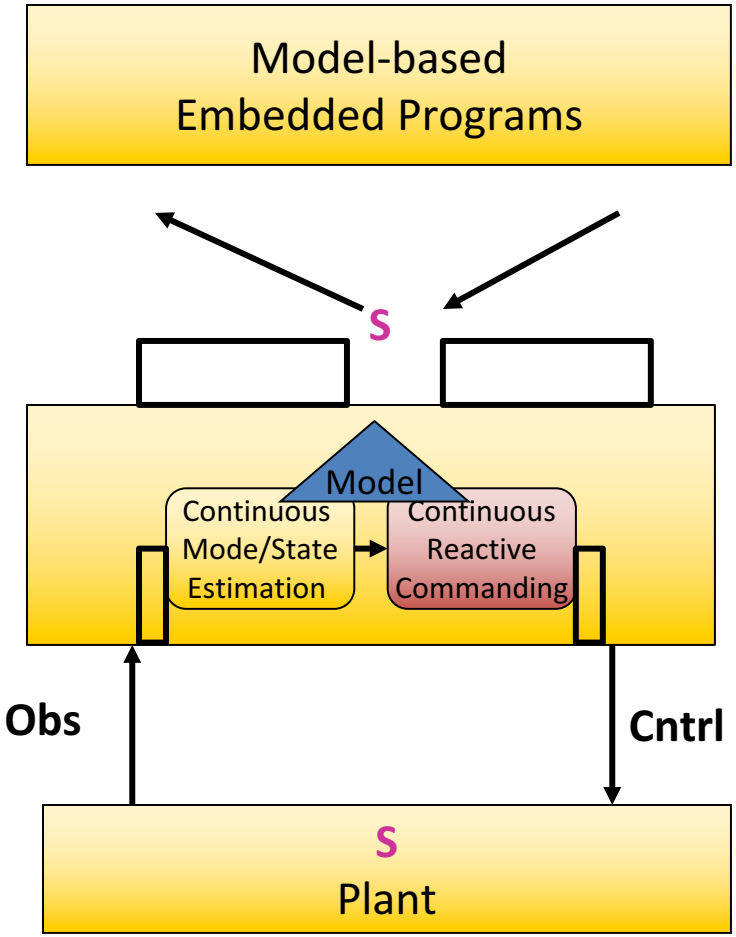
Adapt to disturbance
Adjust timing
Select contingencies

State Aware
Plan to achieve goals states.

Precise
Achieve continuous goal states.

Collaborate
Executes programs, revise goals and plan with humans.

Manage Risk
Executes programs in uncertain environments with bounded risk.



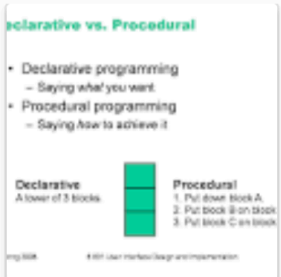
Programming paradigms

Non-deterministic programming

Decision-theoretic programming

State-based programming

Risk-bounded programming



Self-monitoring programs

Concurrent Constraint programming

- When programming computers we think of paradigms.
- What are the programming paradigms for robots?

➢ Our answer: Model-based autonomy
➢ Embodied in the Reactive Model-based Programming Language (RMPL)

Programming languages

Syntax

- In Backus-Naur form (BNF)

Semantics

- How program executes
  - Language executive

```
expression ::= atom | list
atom       ::= number | symbol
number     ::= [+-]?['0'-'9']+
symbol     ::= ['A'-'Z''a'-'z'].*
list       ::= '(' expression* ')'
```

Fragment of Lisp Syntax in BNF

RMPL includes combinators for each paradigm, with associated syntax and semantics.

Language executives

- Systems that execute programs
  - Compilers, interpreters, and execution kernels

- For RMPL, we have model-based executives,
  - which perform decision-making using models.
  - Built using the Enterprise architecture.

- Enterprise coordinates decision-making algorithms, while RMPL hides details of decision-making from the programmer.

- Enterprise has a set of architectural principles and decision-making algorithms in support of each paradigm.

Navigating through this course

- For each lecture,
  we encourage you to identify the associated
  - programming paradigm,
  - language and
  - executive.

- Languages build on top of each other:
  Try to compare vertically each language and its paradigm.

- Try to understand how each decision algorithm enables
  an executive that supports a particular paradigm.

Outline

- Non-deterministic programs and flexible execution
- Describing temporal plans
- Offline scheduling
- Online scheduling and dispatch

Outline

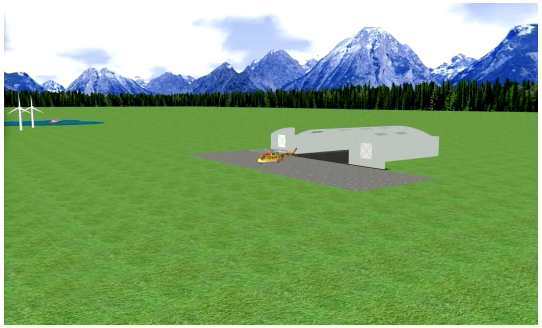
- **Non-deterministic programs and flexible execution**
- Describing temporal plans
- Offline scheduling
- Online scheduling and dispatch

Programs Supporting Flexible Execution

- Theme 1: Goal-directed autonomy

Programs Supporting Flexible Execution

- Theme 1: Goal-directed autonomy



Imperative programs

```
{sequence
    uav1.takeoff();
    uav1.fly_base1_to_lake1();
    uav1.load_water(lake1);
    uav1.fly_lake1_to_fire1();
    uav1.drop_water_high_altitute(fire1);
    uav1.fly_fire1_to_lake1();
    ……
}
```

➢ state-space programs

```
{sequence
    (fire1 == out);
    (fire2 == out);
    (uav1.flying == no &&
     uav1.location == base_location)
 }
```

Programs Supporting Flexible Execution

- Theme 1: Goal-directed autonomy
- We will **build** our language in **steps**, starting with imperative programs.

Start

Imperative programs

```
{sequence
    uav1.takeoff();
    uav1.fly_base1_to_lake1();
    uav1.load_water(lake1);
    uav1.fly_lake1_to_fire1();
    uav1.drop_water_high_altitute(fire1);
    uav1.fly_fire1_to_lake1();
    ……
}
```

Programs Supporting Flexible Execution

- Theme 1: Goal-directed autonomy
- We will build our language in steps, starting with imperative programs
- **Step 1: Flexible Execution** is key to robustness
  - **Program provides** choices, which **executive makes** at run time.

Start

Imperative programs

```
{sequence
    uav1.takeoff();
    uav1.fly_base1_to_lake1();
    uav1.load_water(lake1);
    uav1.fly_lake1_to_fire1();
    uav1.drop_water_high_altitute(fire1);
    uav1.fly_fire1_to_lake1();
    ……
}
```

Step 1

Concurrent programs with flexible time and choice

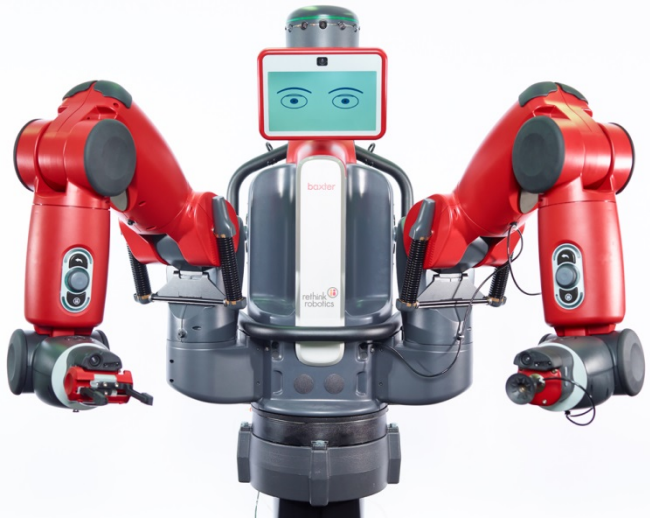
```
{parallel
  [7,15] Rover2.goto(p3);
  {choose
    [with reward 15]
     {sequence
        [2,5] Rover1.imageTargets1();
        [5,10] Rover1.goto(p3);
     };
    [with reward 10]
     {sequence
        [5,10] Rover1.goto(p3);
        [2,5] Rover1.imageTargets1();
     }
  }
}
```

Timed Reactive Systems

- Traditional imperative programs execute sequentially, line by line.

Sequential imperative program

Left arm goes to x_1,y_1;
Right arm goes to x_1,y_1;
Left arm takes the cup from right arm;
Left arm goes to x_3,y_3;
…
Right arm pour water;



How do we coordinate the behavior of two arms of a robot?
– Concurrency

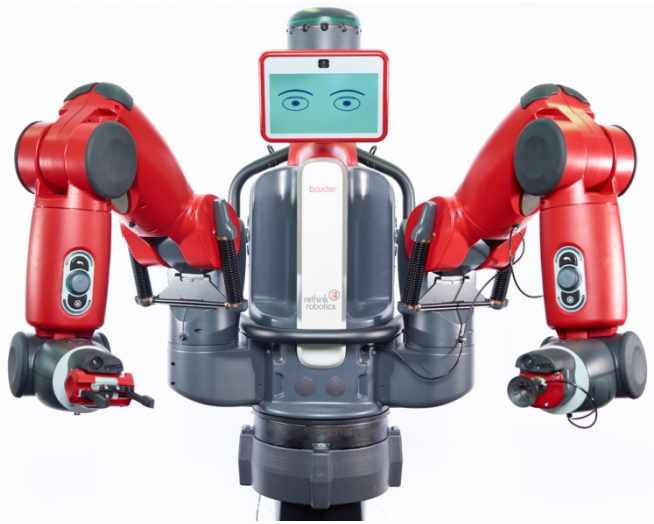
How do we ensure the left arm moves away fast enough, for the right arm to pour?
– Real-time

Timed Reactive Systems

- Robots are real-time concurrent systems.
- Timed concurrent programming languages specify:
  - execution time.
  - concurrent behavior.

Timed concurrent programs:

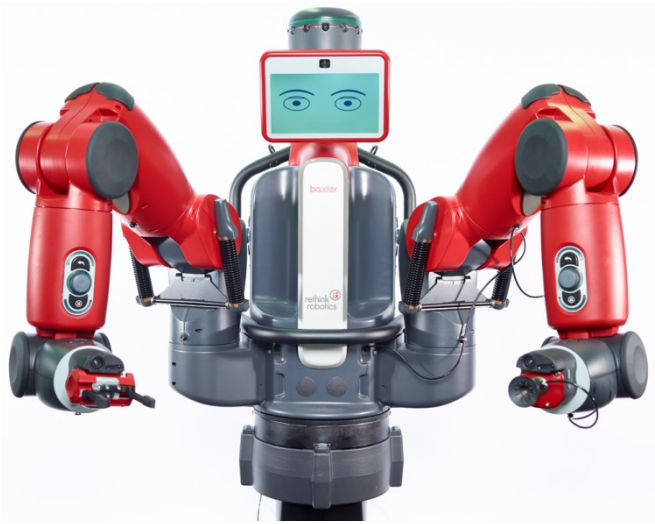
```
{ parallel
 { sequential
      00:00 Left arm go to x1,y1;
      00:10 Left arm take the cup from right arm;
      00:15 Left arm go to x3,y3;
   {
}
 { sequential
      00:00 Right arm go to x1,y1;
      00:25 Right arm pour water;
   }
}
```



Timed Reactive Systems

Timed concurrent programs

- Execution is easy, because time is exact.



Timed concurrent programs:

```
{ parallel
 { sequential
      00:00 Left arm goes to x1,y1;
      00:10 Left arm takes the cup from right arm;
      00:15 Left arm goes to x3,y3;
   {
}
 { sequential
      00:00 Right arm goes to x1,y1;
      00:25 Right arm pours water;
   }
}
```

Timed Reactive Systems

Timed concurrent programs

- Execution is easy, because time is exact.
- But:
  - **Programmer** needs to **decide** execution time.
  - Execution **may easily fail**, if an action is **not finished** in time.
  - Execution **is wasteful**, if **too much margin** given.
  - ➢ **Not robust**.

Timed concurrent programs:

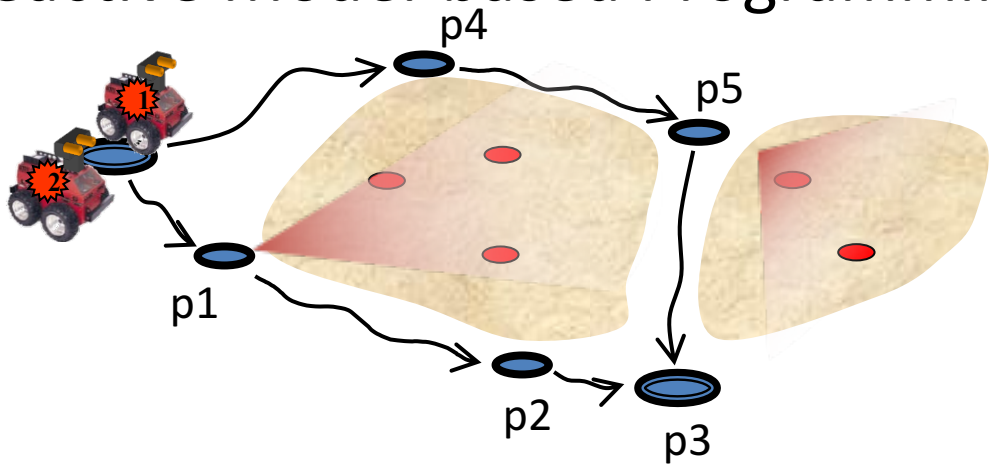
```
{ parallel
 { sequential
      00:00 Left arm goes to x1,y1;
      00:10 Left arm takes the cup from right arm;
      00:15 Left arm goes to x3,y3;
   {
}
 { sequential
      00:00 Right arm goes to x1,y1;
      00:25 Right arm pours water;
   }
}
```

Flexible Time

Idea: Increase robustness by exploiting flexibility in temporal requirements to dynamically adapt.

- Program specifies requirements on timing, not a solution:
  - "Engage spacecraft boosters 2-4 minutes after launch, terminating before reaching orbit."
- Executive schedules on the fly.

Programs with Flexible Time: Reactive Model-based Programming Language




```
Program A ::=
    prim_action(args)    |
    [lb, ub] A   |
    {Sequence A1; A2; ... } |
    {Parallel A1; A2; ... }
```

RMPL Syntax

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
     00:00 Rover1.goto(p5);
     00:30 Rover1.imageTargets();
     00:00 Rover1.goto(p3);
  },
  {Sequence
     00:20 Rover2.goto(p1);
     00:50 Rover2.imageTargets();
     00:00 Rover2.goto(p3);
  }
}
```

→

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
    [5,10] Rover1.goto(p5);
    [2,5]  Rover1.imageTargets();
    [5,10] Rover1.goto(p3);
  },
  {Sequence
    [5,10] Rover2.goto(p1);
    [5,10] Rover2.imageTargets();
    [5,10] Rover2.goto(p3);
  }
}
```

Fixed Time

Dispatch action "Go to x_3,y_3 " at 00:40.

00:00 Go to x_1,y_1
00:20 Land at x_2,y_2
00:40 Go to x_3,y_3
…
04:10 Pour water

"I am not available now.
Since I need to refuel first."

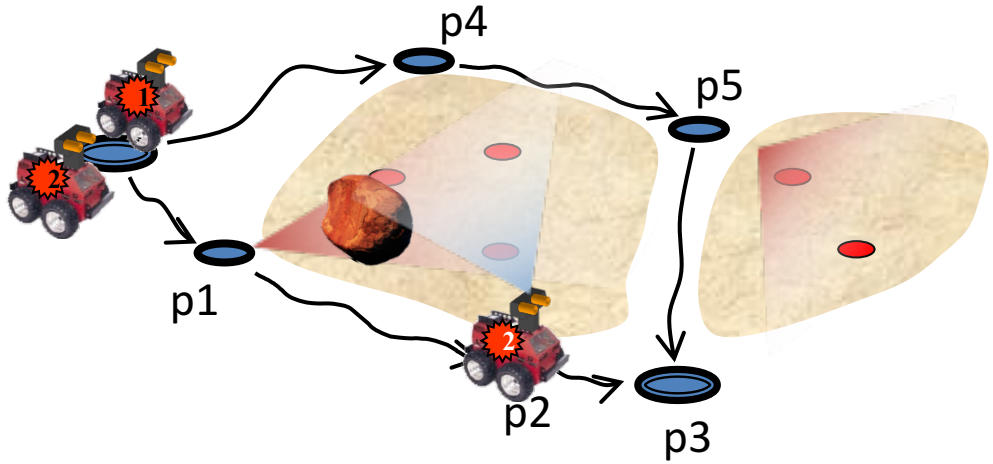
Flexible Time

After finishing action "Land at x_2,y_2,"
dispatch action "Go to x_3,y_3,"
which can take between 0 to 20 minutes.

[0,10] Go to x_1,y_1
[10,20] Land at x_2,y_2
[0,20] Go to x_3,y_3
…
[20,30] Pour water

"I will be ready.
Since I have another 20 minutes,
which is enough time to refuel."

Programs with Choice:
Non-deterministic and Decision-theoretic Programs




```
Program A ::=
    prim_action(args)    |
    [lb, ub] A   |
    {Sequence A1; A2; ... } |
    {Parallel A1; A2; ... }     |
    {Choose [with reward R1] A1;
            [with reward R2] A2; . . .}
```

RMPL Syntax

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
    [5,10] Rover1.goto(p5);
    [2,5]   Rover1.imageTargets();
    [5,10] Rover1.goto(p3);
  },
  {Sequence
    [5,10] Rover2.goto(p1);
    [5,10] Rover2.imageTargets();
    [5,10] Rover2.goto(p3);
  }
}
```

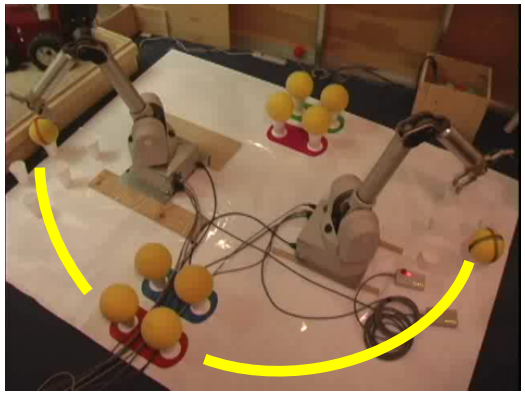
```
Choose{
    [with reward 15]
    [5,10] Rover2.imageTargets();

    [with reward 10]
    {Sequence
        [5,10] Rover2.goto(p2);
        [5,10] Rover2.imageTargets();
    };
}
```

What if we want more flexibility in our program?

Flexible Execution in Action

Robots should adapt to each other to perform cooperative tasks

Shah and Williams, ICAPS 09



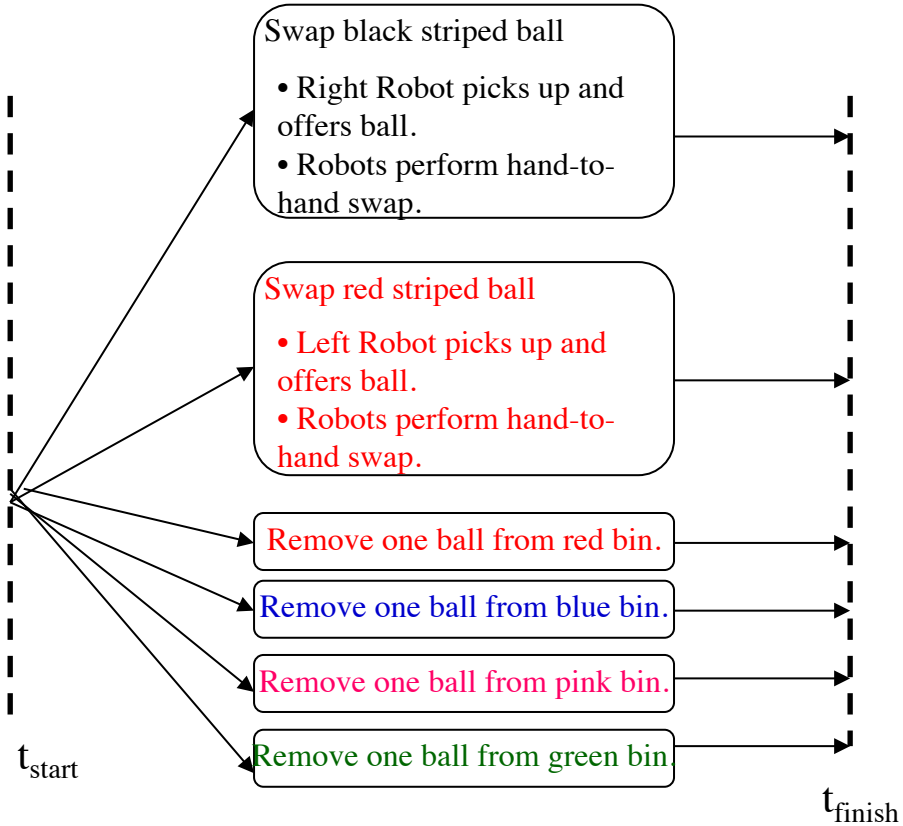
Agents choose and schedule activities.

(Someone) Remove one ball from red bin.

Remove one ball from red bin.

OR

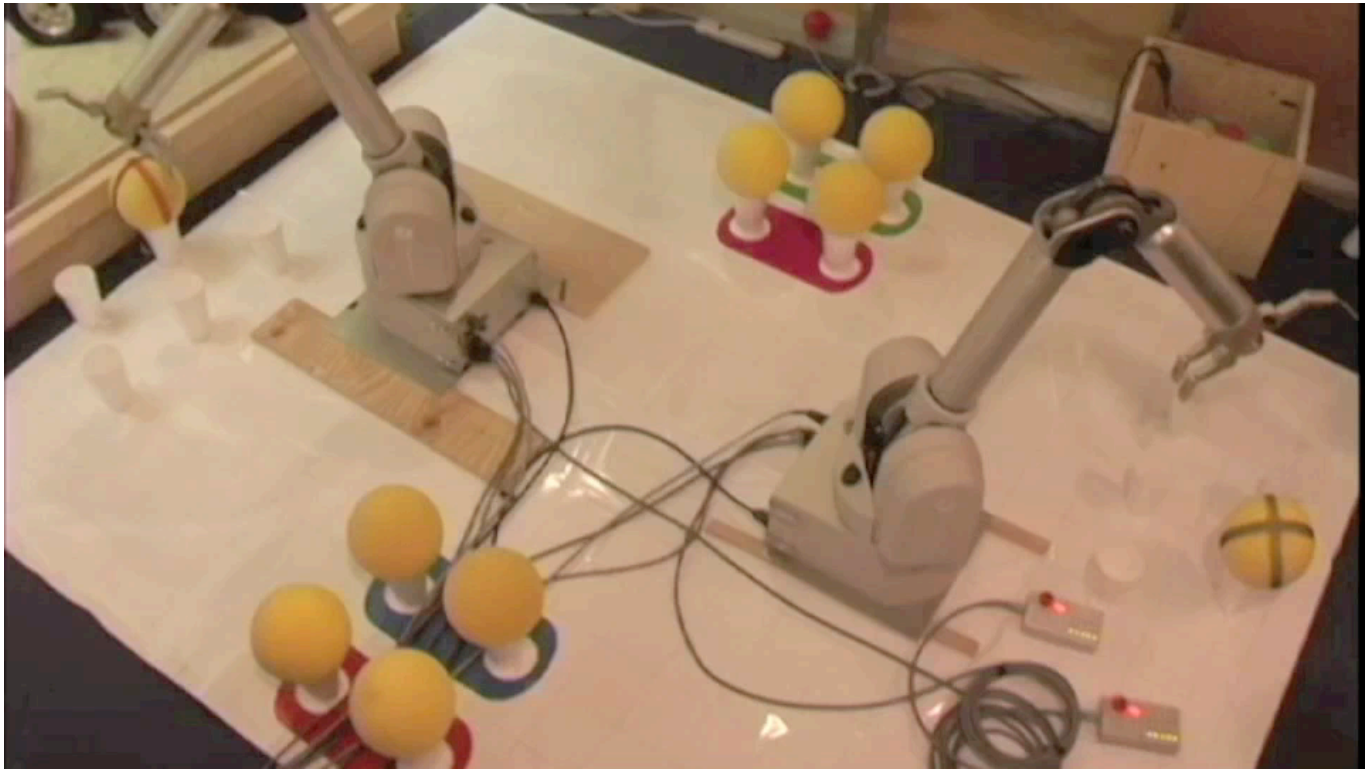
L[32,39] V R[42,55]



Flexible Execution in Action

- Equal Partners
- Off-nominal
- Partner adapts in response to teammate's failure

Kim, Williams, Abramson IJCAI 2001;
Shah and Williams ICAPS 2009;
Conrad, Shah and Williams ICAPS 2010;



Non-deterministic programs can express different types of choices

When:

- Alternative orderings of actions

With What:

- Alternative resource assignments

Who:

- Alternative task assignments

How:

- Alternative methods for completing a task

All decisions made during Execution

Lectures on non-deterministic timed programs and flexible execution

When:

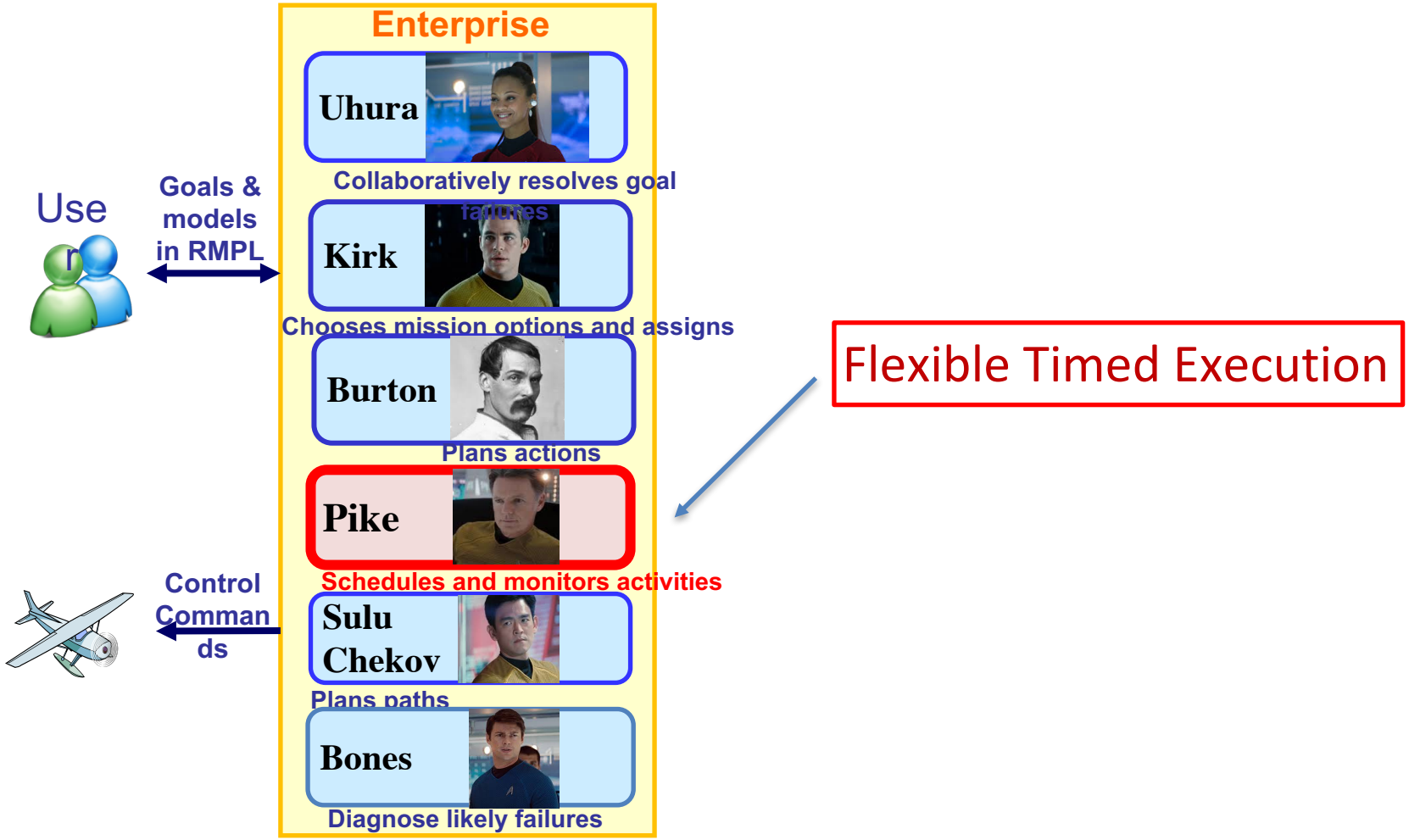
- Time of execution is flexible but bounded within an interval.
  - Today's lecture

- Risk-bounded execution bounds likelihood of failure, given model of environment uncertainties.
  - Later lecture

With What, Who, How:

- Non-deterministic programs specify discrete choices, made on the fly.
  - Later lecture

Recall: A single "cognitive system" language (RMPL) and executive (Enterprise).

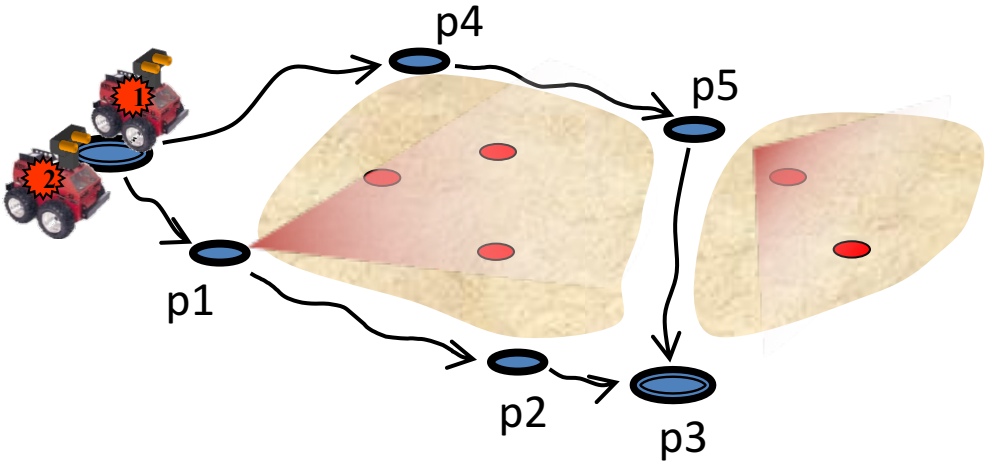


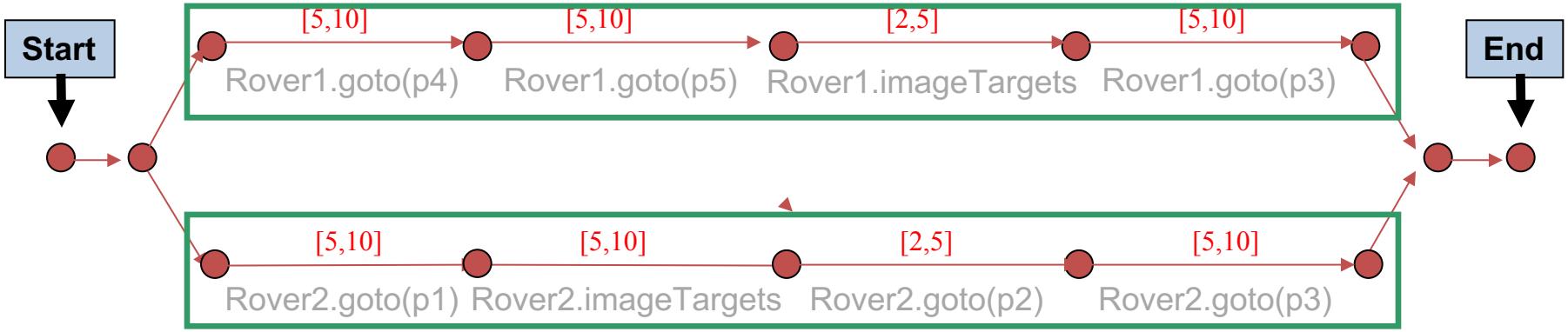
Outline

- Non-deterministic programs and flexible execution
- **Describing temporal plans**
- Offline scheduling
- Online scheduling and dispatch

Execution of Timed Model-based Programs

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
    [5,10] Rover1.goto(p4);
    [5,10] Rover1.goto(p5);
    [2,5] Rover1.imageTargets();
    [5,10] Rover1.goto(p3);
  },
  {Sequence
    [5,10] Rover2.goto(p1);
    [5,10]Rover2.imageTargets()
    [2,5] Rover2.goto(p2);
    [5,10] Rover2.goto(p3);
  }
}
```





[Williams et al 01] RMPL

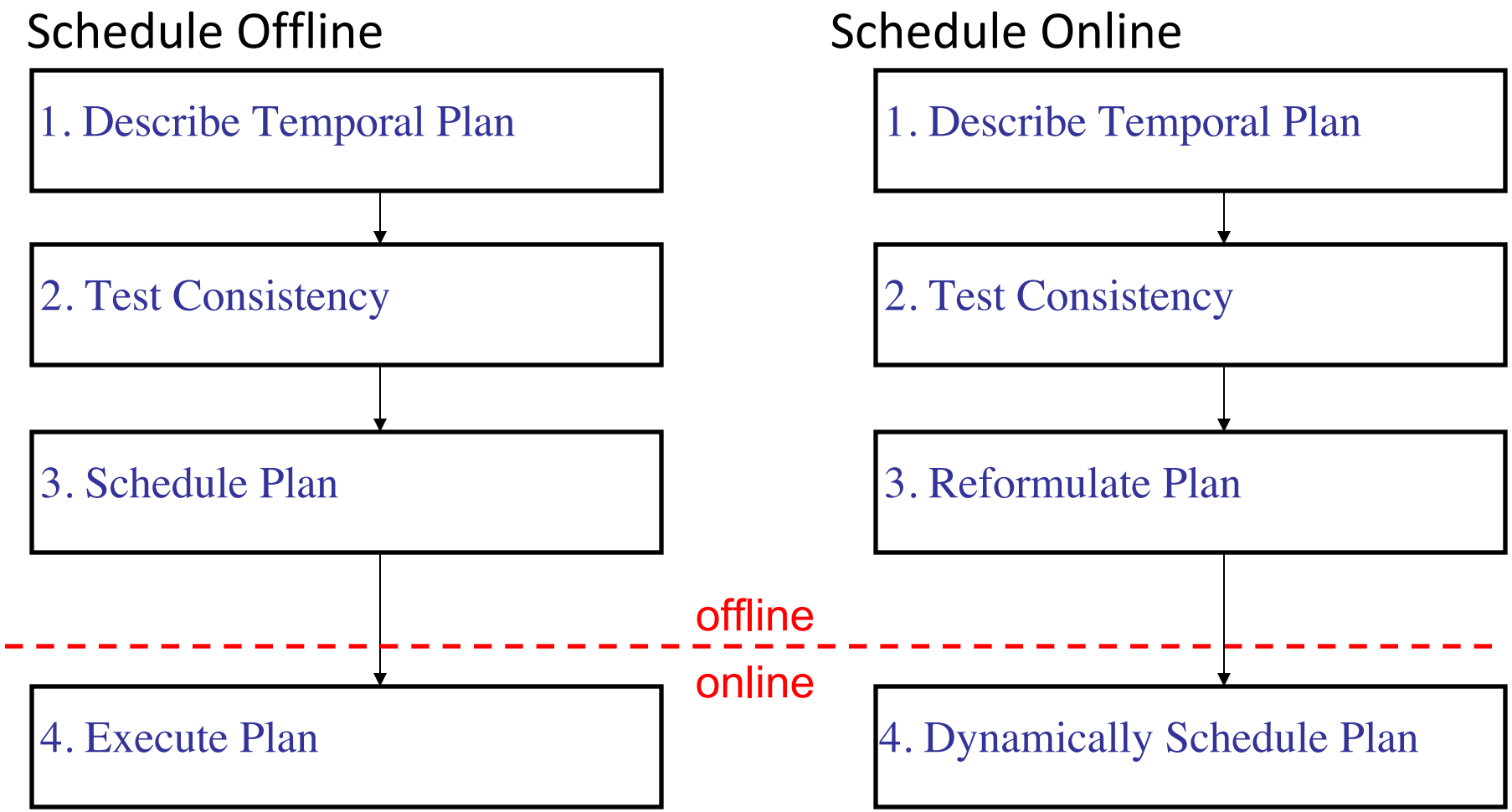
[Muscettola, Morris, Tsamardinos,KR 98] Execution

Execution of Timed Model-based Programs

- A smarter executive that makes decisions dynamically, including what, how, who and when. Focus of today
- Agents adapt to temporal disturbances in a coordinated manner by scheduling the start of activities on the fly.
- What does the executive do?
  - Compile the timed program into a temporal plan internally.
  - Schedule activities by reasoning over the temporal plan.

[Muscettola, Morris, Tsamardinos,KR 98]

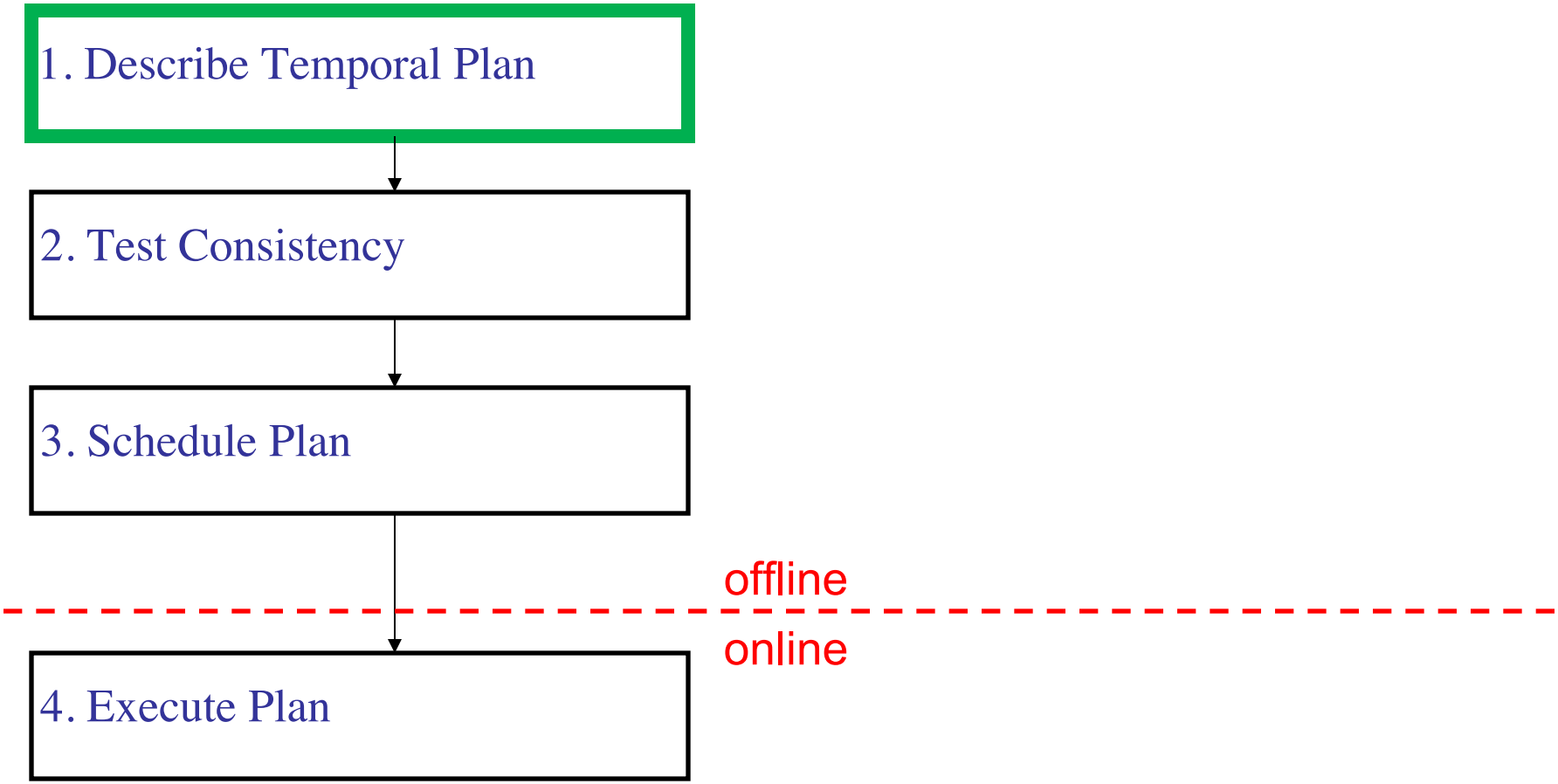
To Execute a Temporal Plan



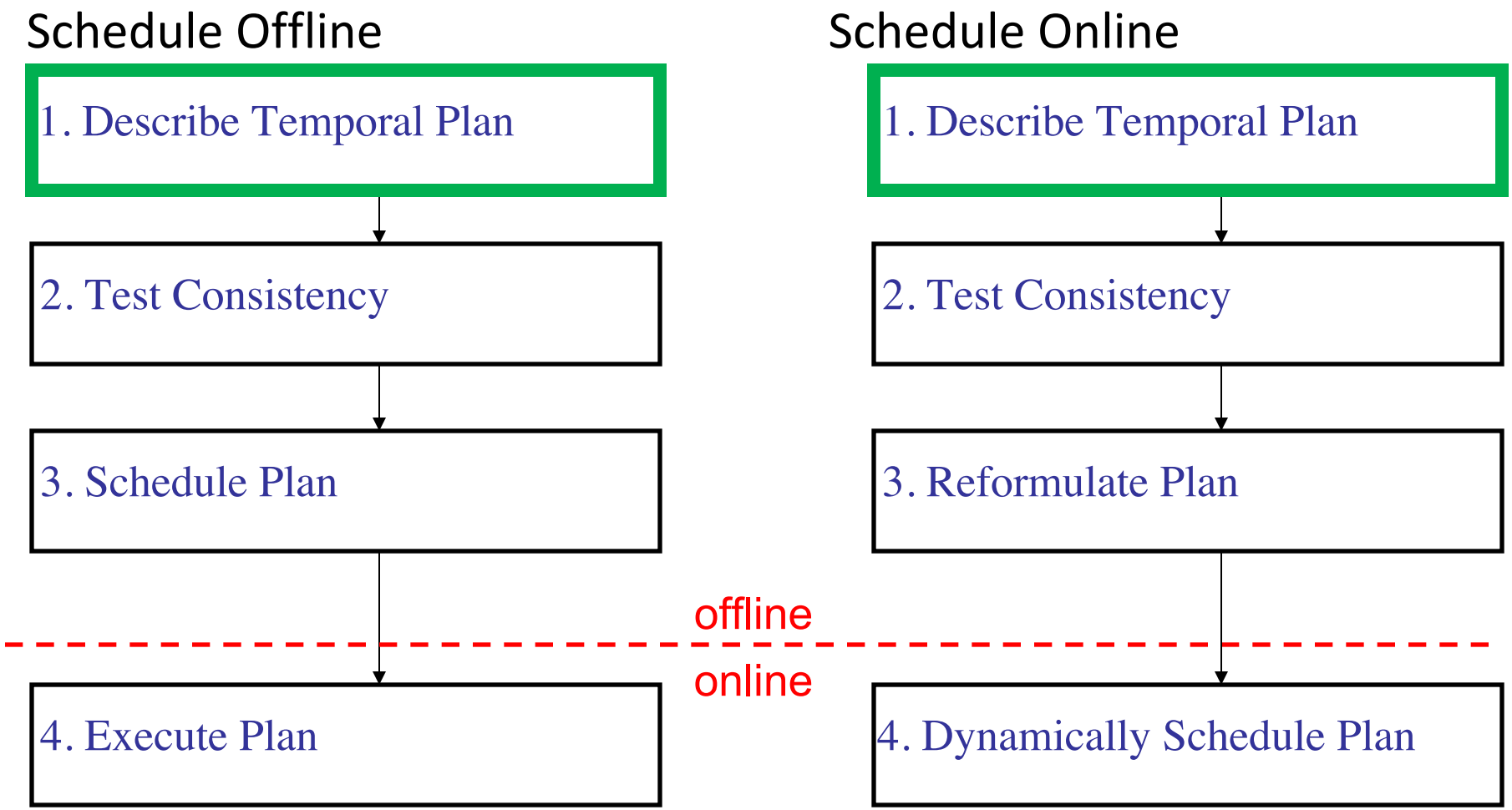
To Execute a Temporal Plan

Schedule Offline

Schedule Online

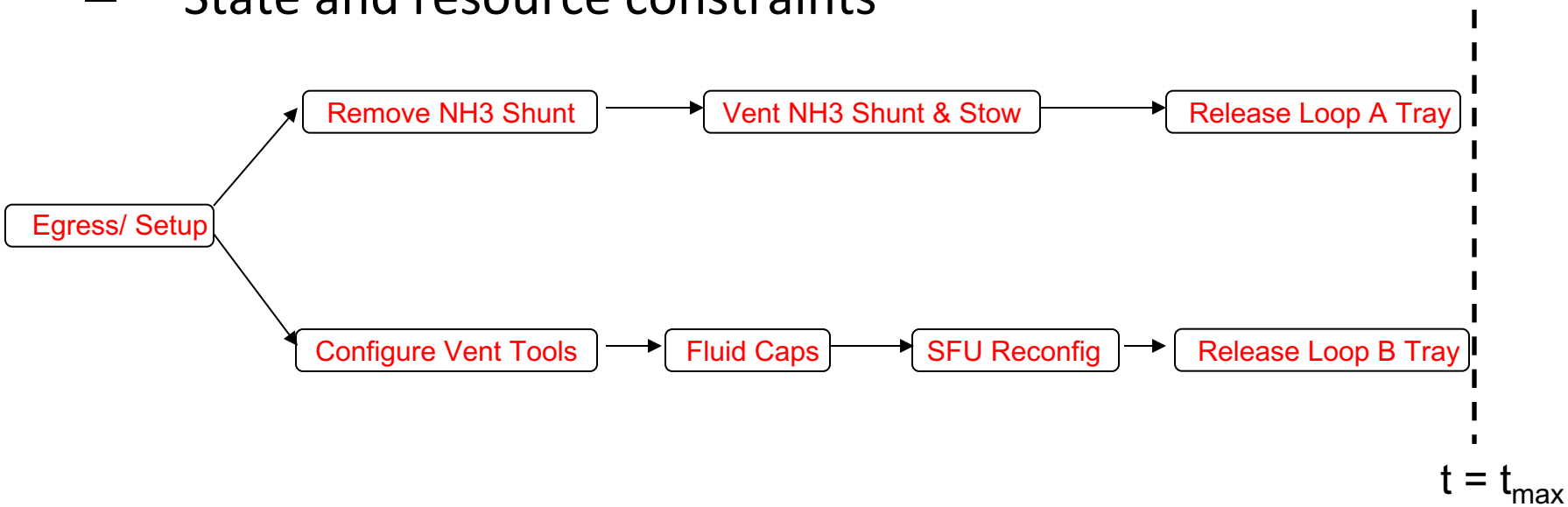


To Execute a Temporal Plan



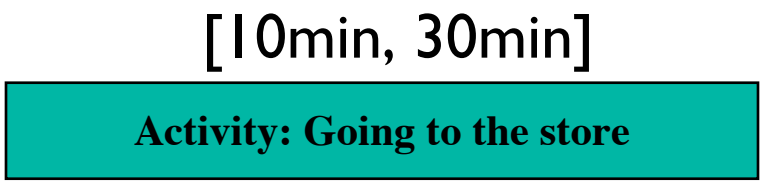
Describe Temporal Plans

- Activities to perform
  - procedure calls, plan operators (PDDL), . . .
- Relationships among activities
  - **temporal constraints**
  - State and resource constraints

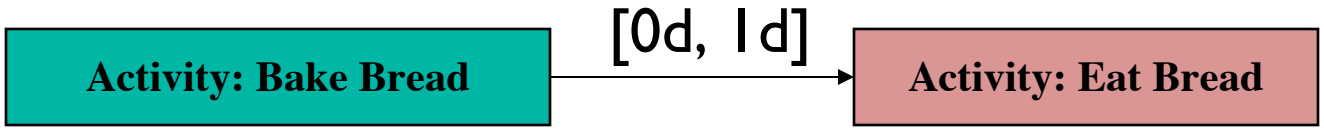


Metric Temporal Relations

- Going to the store takes at least 10 min and at most 30 min.

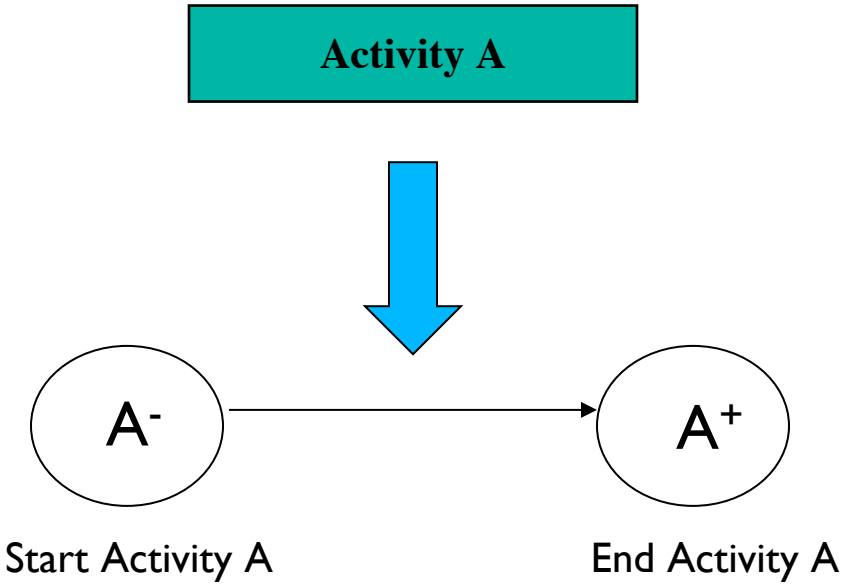


- Bread should be eaten within one day of baking.



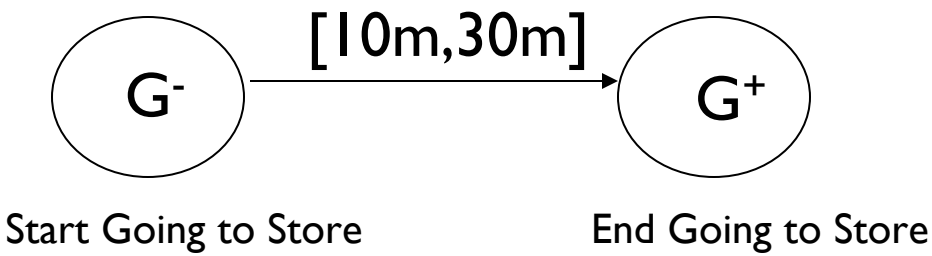
Could also be Qualitative: Before, After, During, Overlaps, Ends Together, ….

Simplify by reducing interval relations to relations on time points (*events*).



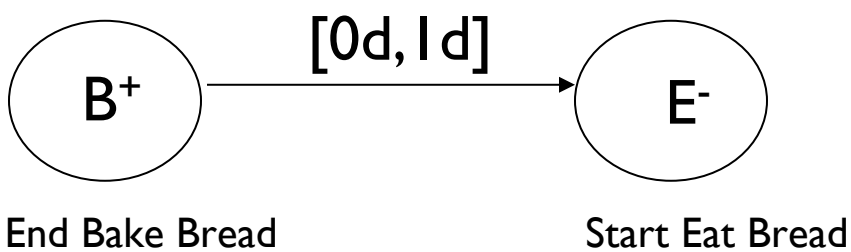
Metric Temporal Relations

- Going to the store takes at least 10 min and at most 30 min.



$10 \leq (G^{+} - G^{-}) \leq 30$

- Bread should be eaten within one day of baking.



$0 \leq (E^{-} - B^{+}) \leq 1$

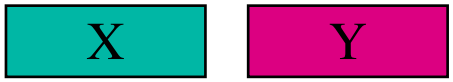
Called a
simple temporal constraint

Qualitative Temporal Relations

as common linguistic expressions

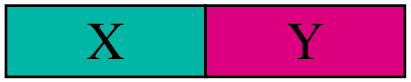
[Allen 83]

X before Y



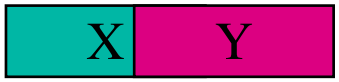
Y after X

X meets Y



Y met-by X

X overlaps Y



Y overlapped-by X

X during Y

Y contains X

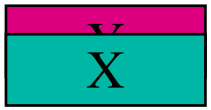
X starts Y

Y started-by X

X finishes Y

Y finished-by X

X equals Y



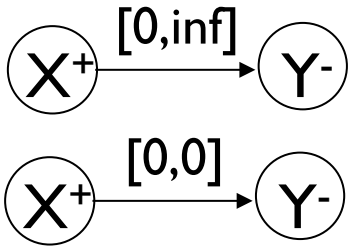
Y equals X

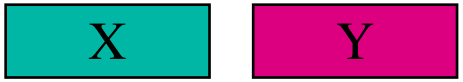
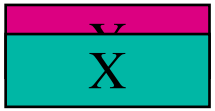
X disjoint Y

Qualitative Temporal Relations

as timepoint inequalities

[Villain & Kautz; Simmons]



X before Y		$X^+ < Y^-$
X meets Y		$X^+ = Y^-$
X overlaps Y		$Y^- < X^+$ and $X^- < Y^+$
X during Y		$Y^- < X^-$ and $X^+ < Y^+$
X starts Y		$X^- = Y^-$ and $X^+ < Y^+$
X finishes Y		$X^- < Y^-$ and $X^+ = Y^+$
X equals Y		$X^- = Y^-$ and $X^+ = Y^+$
X disjoint Y		$X^+ < Y^-$ or $Y^+ < X^-$

Temporal Relations Described by …

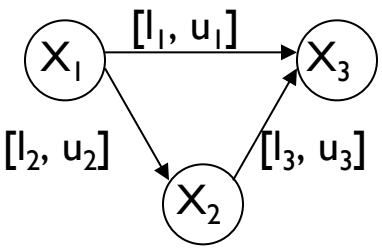
Simple Temporal Network (STN) <X, C>

[Dechter, Meiri, Pearl 91]

- Variables $X_1,\ldots X_n$, representing time points with real-valued domains.

- Binary (simple temporal) constraints C :

$$(X_k - X_i) \in [a_{ik}, b_{ik}]$$



  – called *links*.

Sufficient to represent:
- simple metric constraints
- all Allen relations but 1…

Can't represent:
- Allen's disjoint relation

Temporal Relations Described by …

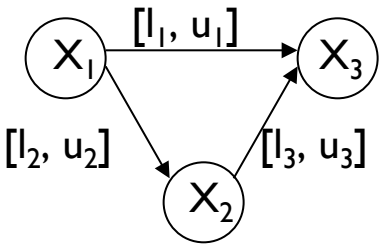
A Simple Temporal Network (STN) <X, C> [Dechter, Meiri, Pearl 91]

- Variables $X_1,\ldots X_n$, representing time points with real-valued domains.

- Binary (simple temporal) constraints C :

$$(X_k - X_i) \in [a_{ik}, b_{ik}]$$

  – called *links*.



Visualization of Simple Temporal Network Consistency

- http://bicycle.csail.mit.edu/stn/

- Thanks to Steve Levine!

To Execute a Temporal Plan

Schedule Offline

1. Describe Temporal Plan

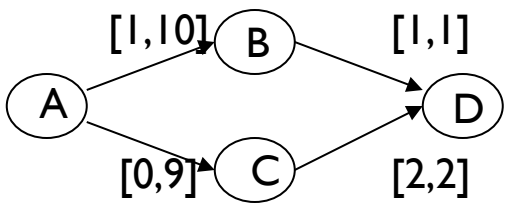
2. Test Consistency

3. Schedule Plan

offline

online

4. Execute Plan

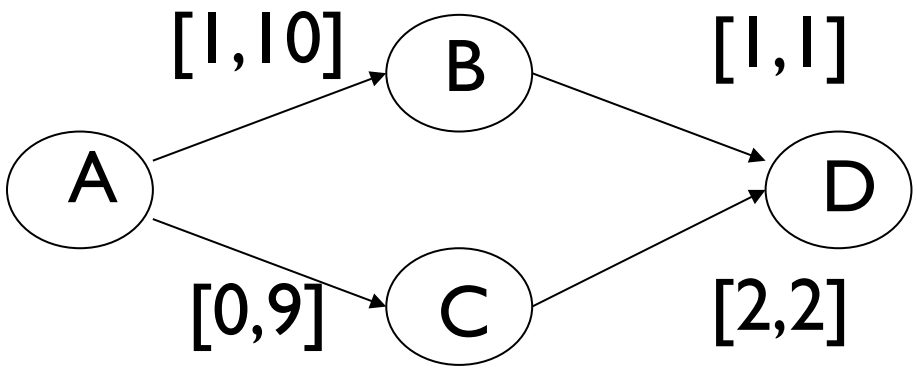


Outline

- Non-deterministic programs and flexible execution
- Describing temporal plans
- **Offline scheduling**
  - Exposing implicit constraints
  - Checking consistency
  - Offline scheduling
- Online scheduling and dispatch

Consistency of an STN

Input: STN <X, C> where C_j = < <X_k, X_i>, <a_j, b_j> >

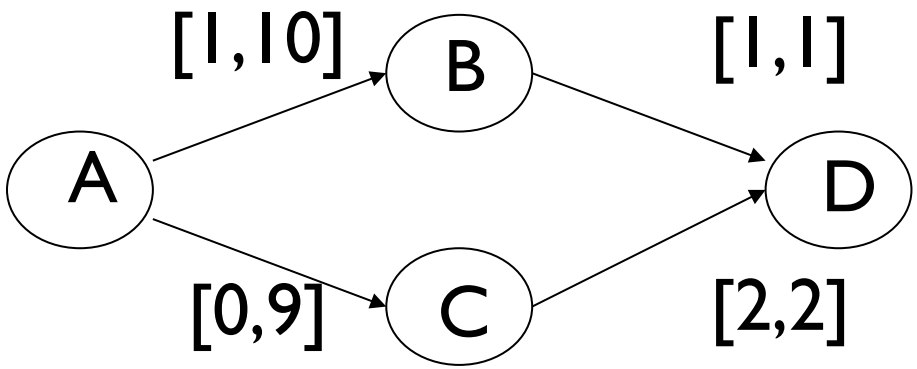


STN is ***consistent*** iff there exists an assignment to time points X satisfying C.

Conflict: Subset of C that is inconsistent.

Schedule of an STN

Input: STN <X, C> where C_j = < <X_k, X_i>, <a_j, b_j> >



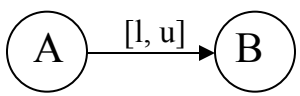
Schedule is an assignment to time points X consistent with constraints.

How do we find a schedule?

- Idea: Transform STN
  1. Transform to a ***distance graph***
  2. Find implied constraints
     by applying common graph algorithms
     (e.g., shortest path – Dijkstra, Floyd-Warshall, Johnson).
  3. Use implied constraints to extract schedule.

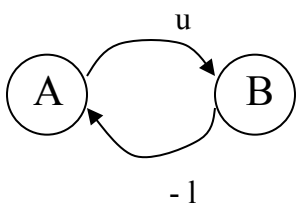
Map Simple Temporal Net to Distance Graph

Simple Temporal Network



$l < B - A < u$

Distance Graph



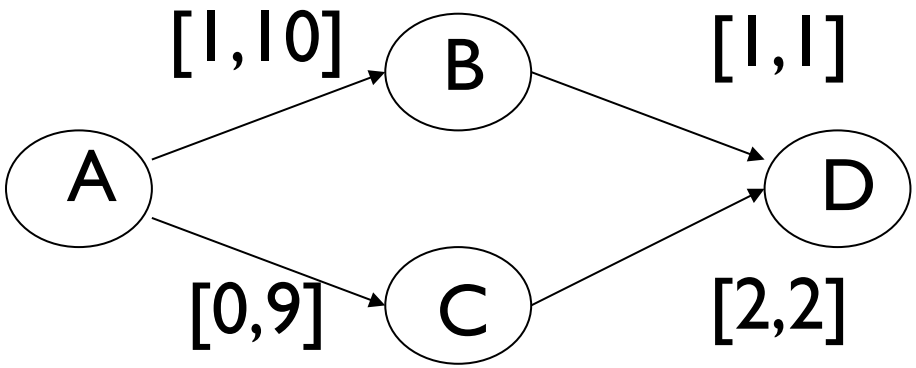
$B - A < u$

$l < B - A$ $\quad A - B < -l$

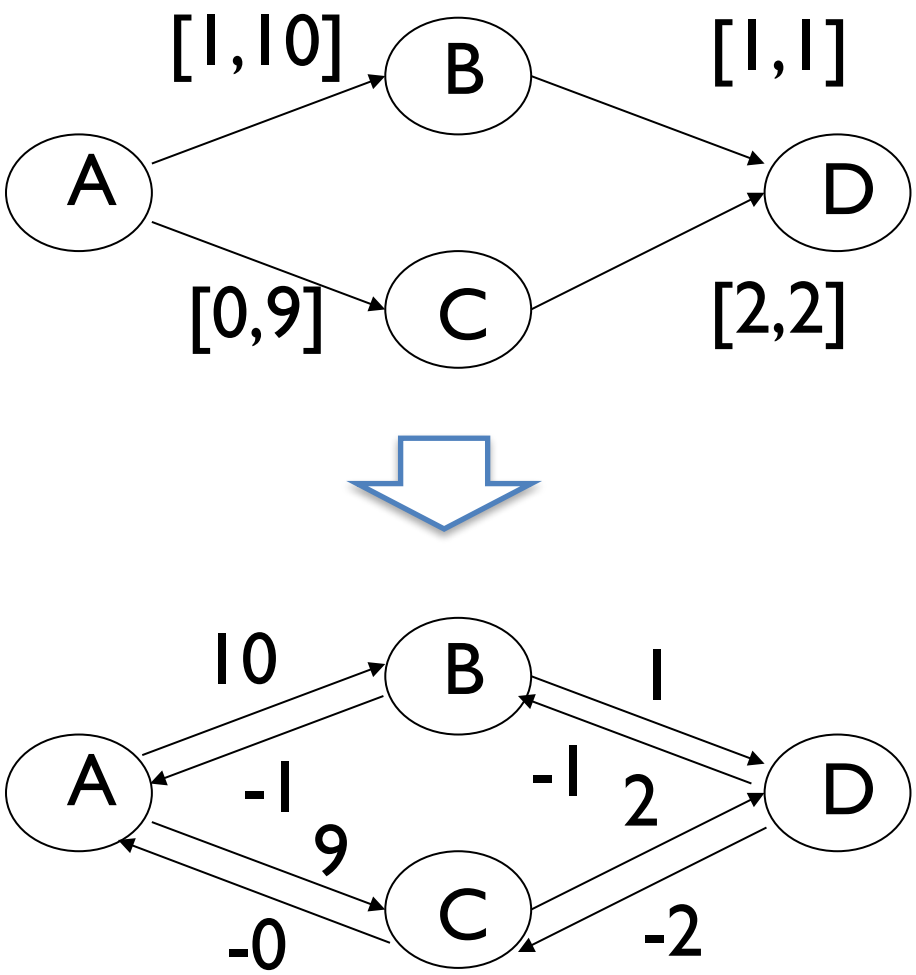
- Upper bound mapped to outgoing, non-negative arc.
- Lower bound mapped to incoming, negative arc.

[Dechter, Meiri, Pearl 91]

Transformation to distance graph



Transformation to distance graph



A framework for offline scheduling

Initialize "execution" window to [-∞, ∞] for each event

while unexecuted events exist:

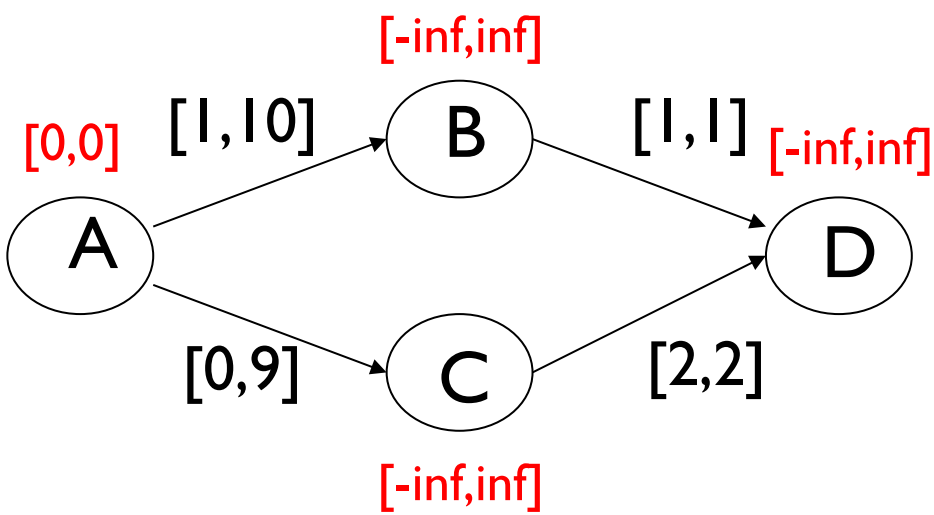
x_i = pick any unexecuted event

t_i = pick any time in x_i's execution window

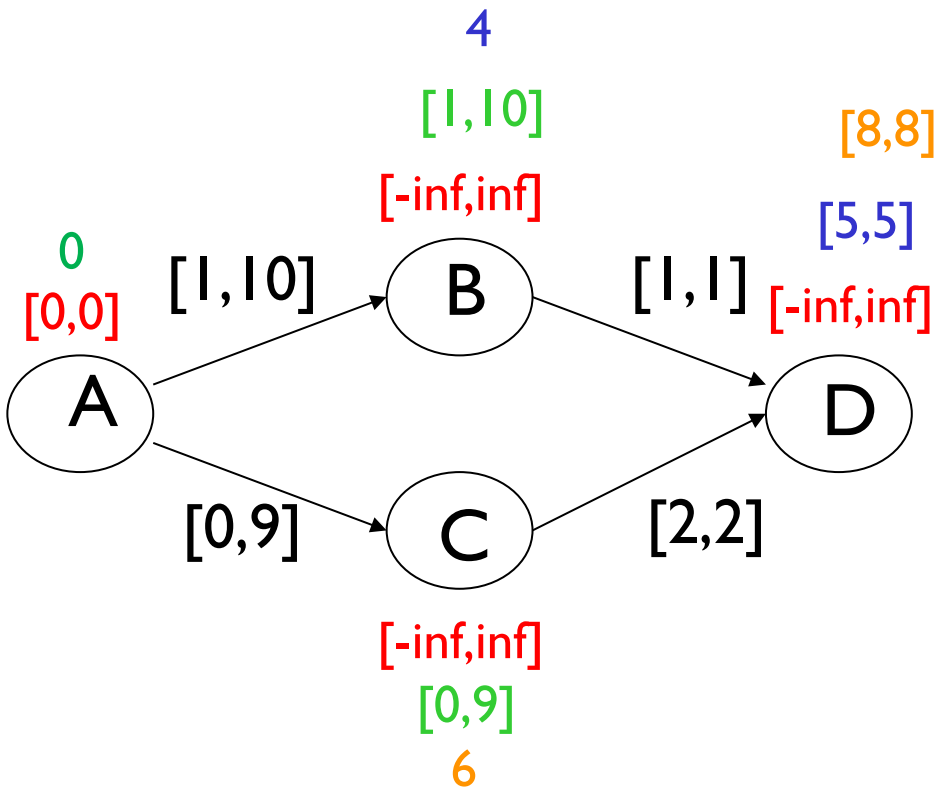
Propagate to all x_i's neighbors & update their windows

Applying the framework directly doesn't work

- (Board: A=0,B=4,C=6)



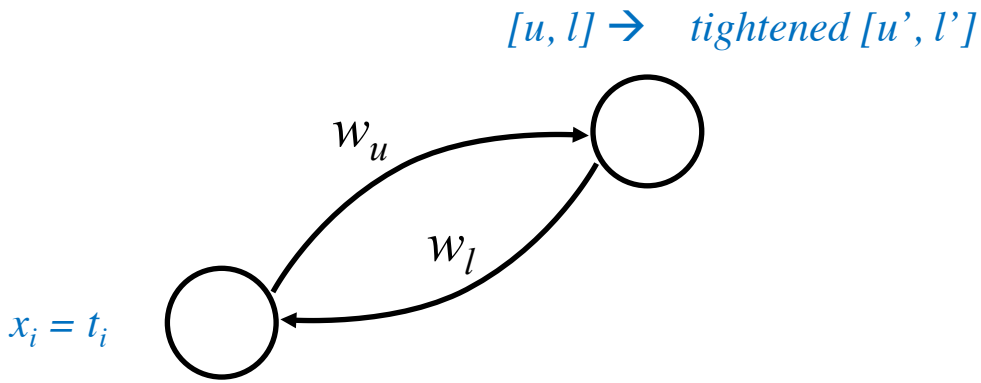
Applying the framework directly doesn't work



Propagating to neighbors

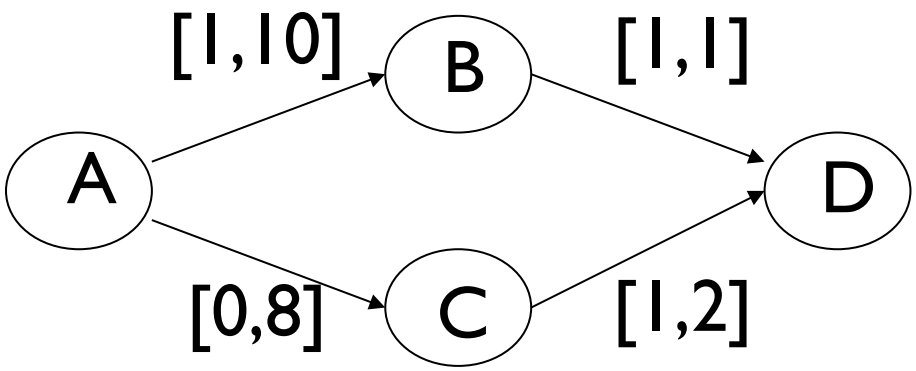
Tighten neighbor's execution windows:

- outgoing edges to neighbor: $u' = \min(u, t_i + w_u)$
- incoming edges from neighbor: $l' = \max(l, t_i - w_l)$

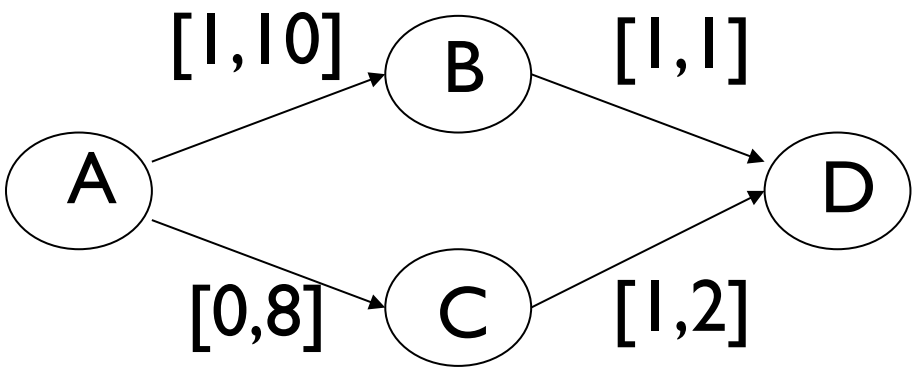


Next: Expose Implicit Constraints

- D-A=[?,?]



Expose Implicit Constraints

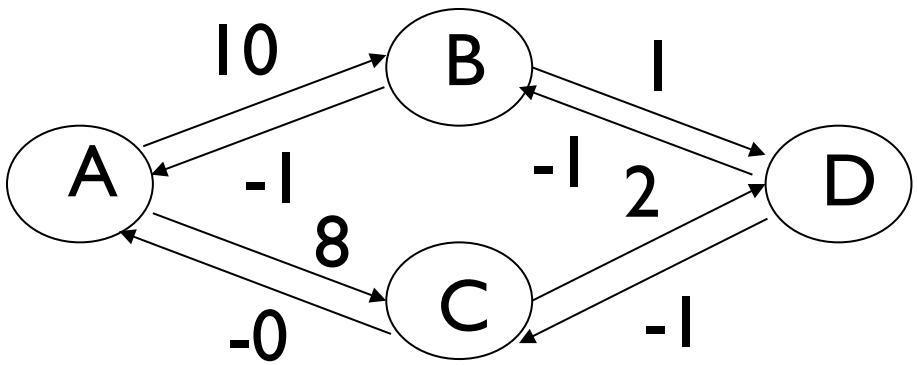


- T_{AD} in [1,10] + [1,1] = [2,11]
- T_{AD} in [0,8] + [1,2] = [1,10]

- T_{AD} in [2,11] intersect [1,10] = [2,10]

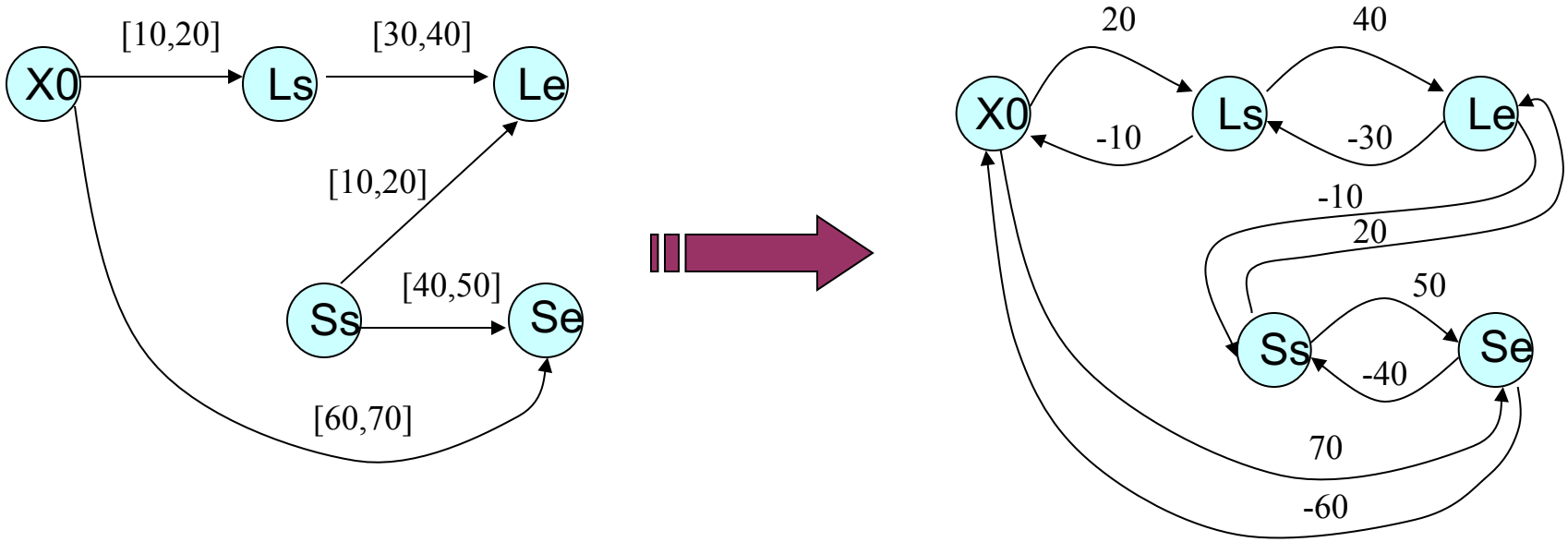
Exposing Implicit Constraints as Shortest Paths

- D-A=[?,?]



Exposing Implicit Constraints

- (Board: Le – X0 in [?,?]) [40, 50]



Decomposable form

- An STN or distance graph is ***decomposable*** if:
  - It can be properly scheduled via local propagation to neighbors only.
  - ➢ No search!
- Requires all implicit constraints be explicit.

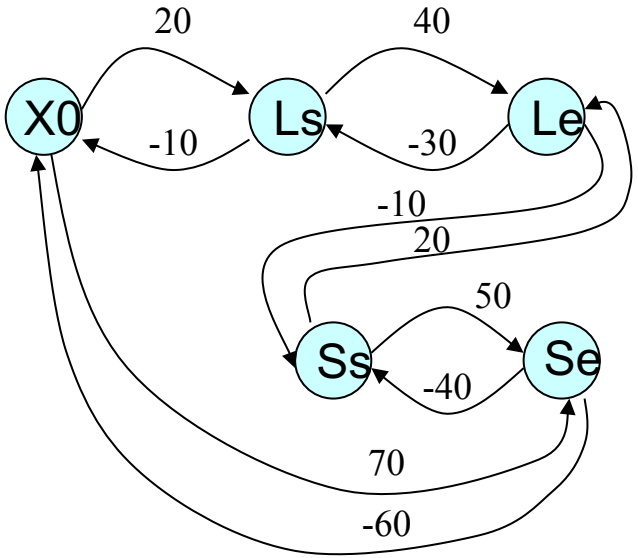
Generate Decomposable Form using APSP Graph

	X_0	Ls	Le	Ss	Se
X_0	0	20	50	30	70
Ls	-10	0	40	20	60
Le	-40	-30	0	-10	30
Ss	-20	-10	20	0	50
Se	-60	-50	-20	40	0

+ Latest Times

- Earliest Times

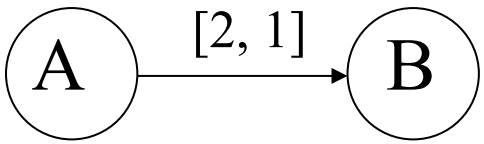
APSP d-graph



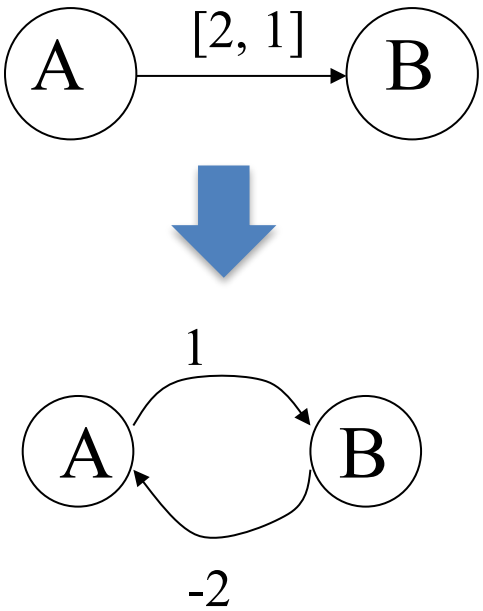
- Ls in [10, 20]
- Le in [40, 50]
- Ss in [20, 30]
- Se in [60, 70]

Checking Consistency

- (Board: A-A in [?,?])



Inconsistent when distance graph has negative cycle

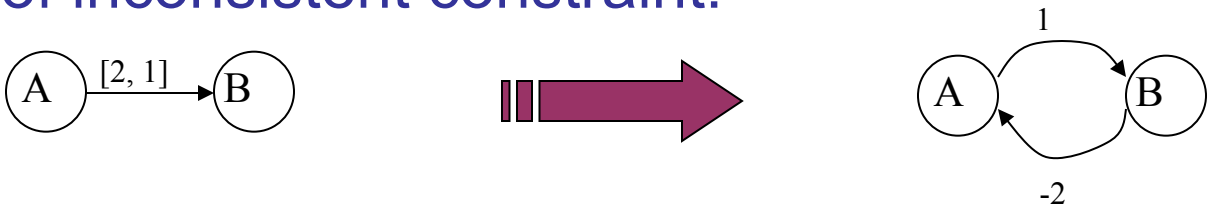


Distance Graph

Check Distance Graph Consistency

- Consistent iff distance graph has no negative cycles.
- Detect by computing shortest path from one event to all others.
  - Single Source Shortest Path (SSSP).
  - Event must reach all others.

Example of inconsistent constraint:



Simple Temporal Network

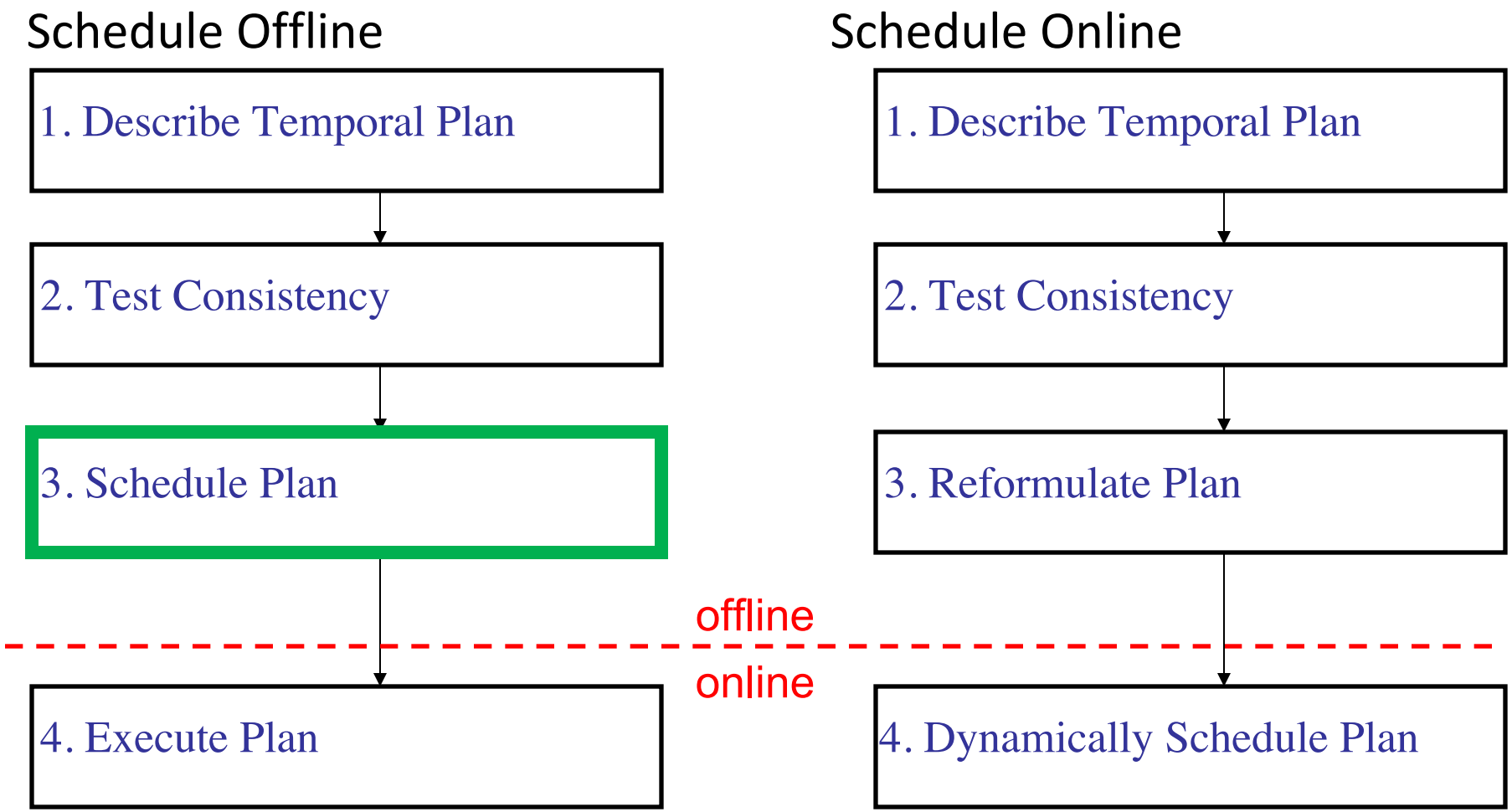
Distance Graph

Summary Thus Far

- We can schedule quickly using a simple, local-propagation algorithm if . . .
  - we expose implicit constraints.
- Expose implicit constraints by computing all-pairs shortest paths (APSP) of distance graph.
  - Puts network in ***decomposable form***.
- If negative cycle in APSP, then inconsistent.

2019 LECTURE 3 PICKED UP HERE

To Execute a Temporal Plan



Algorithm: offline scheduling

Compute decomposable form (i.e., APSP)

Initialize execution window to [-∞, ∞] for each event.

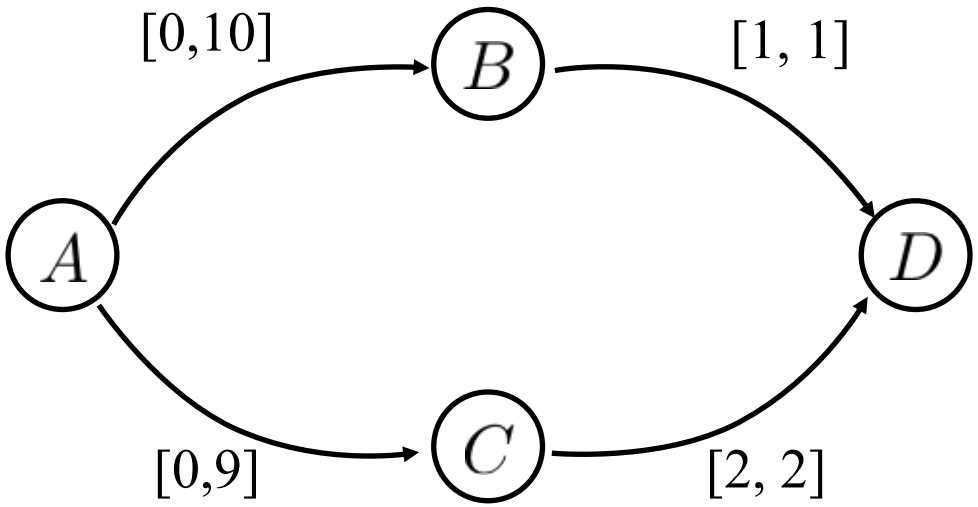
While unscheduled events:

x_i = pick any unscheduled event

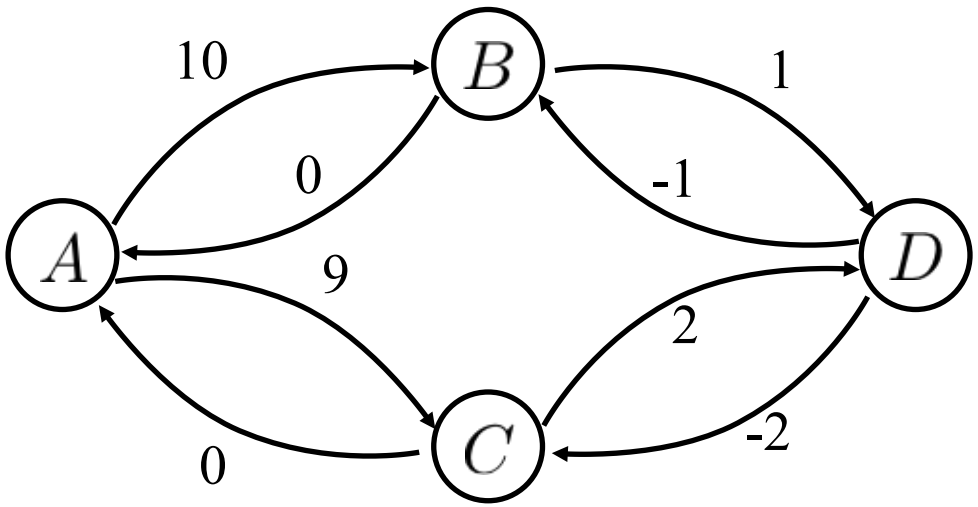
t_i = pick any time in x_i's execution window

Propagate to all x_i's neighbors & update their windows

The original STN



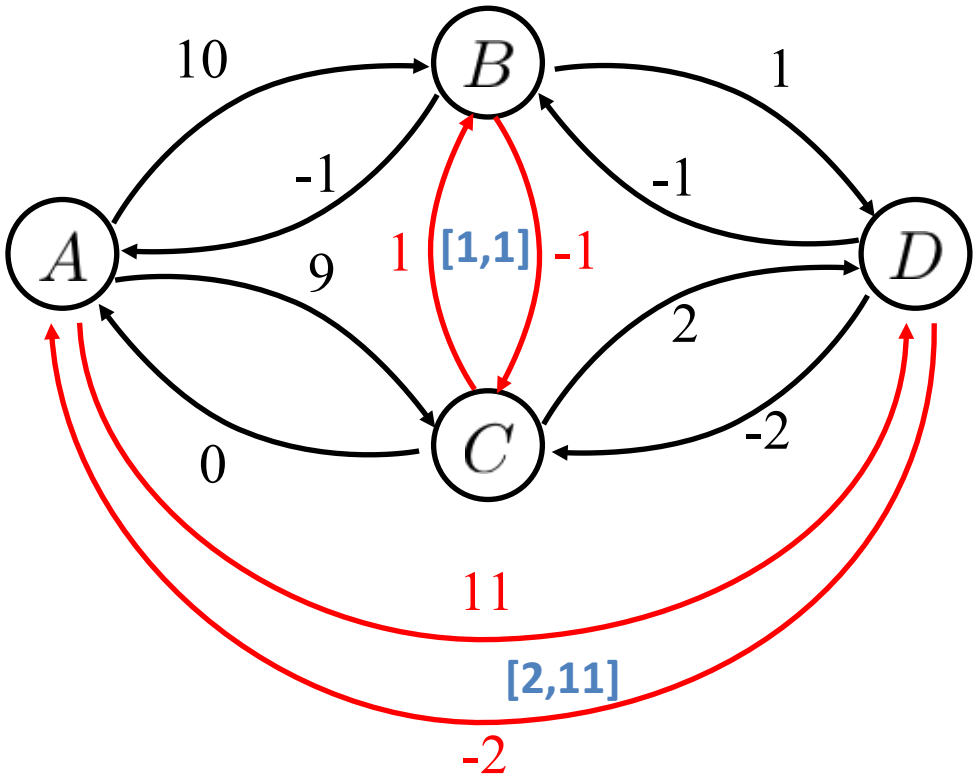
Distance graph transformation



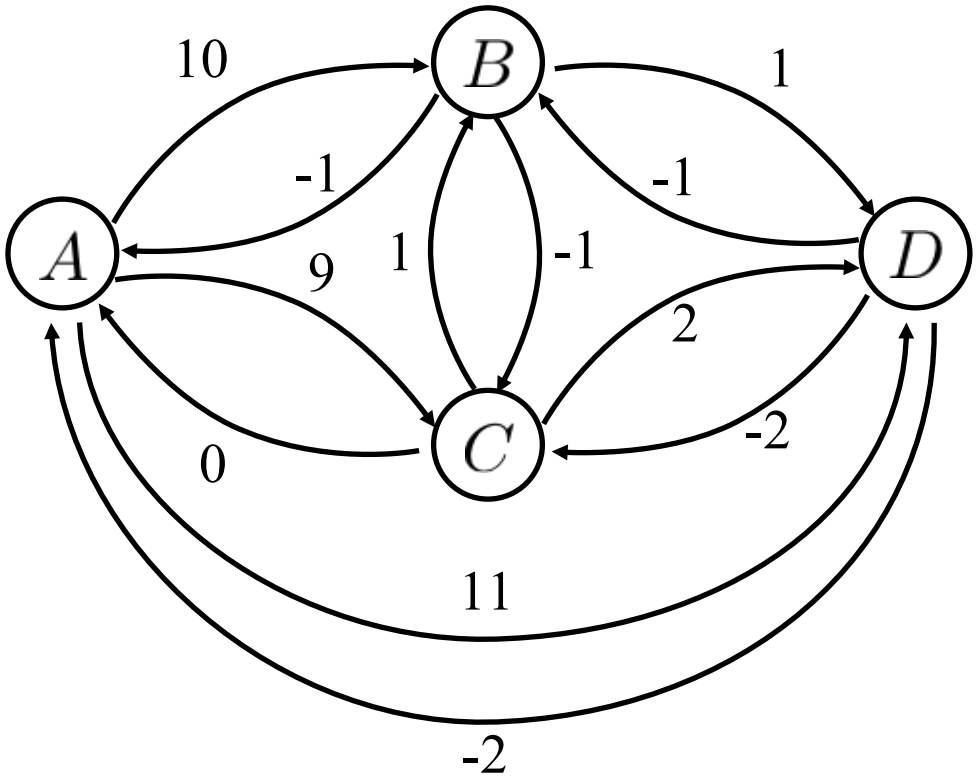
Check for negative cycle

Find implicit constraints using All Pairs Shortest Path

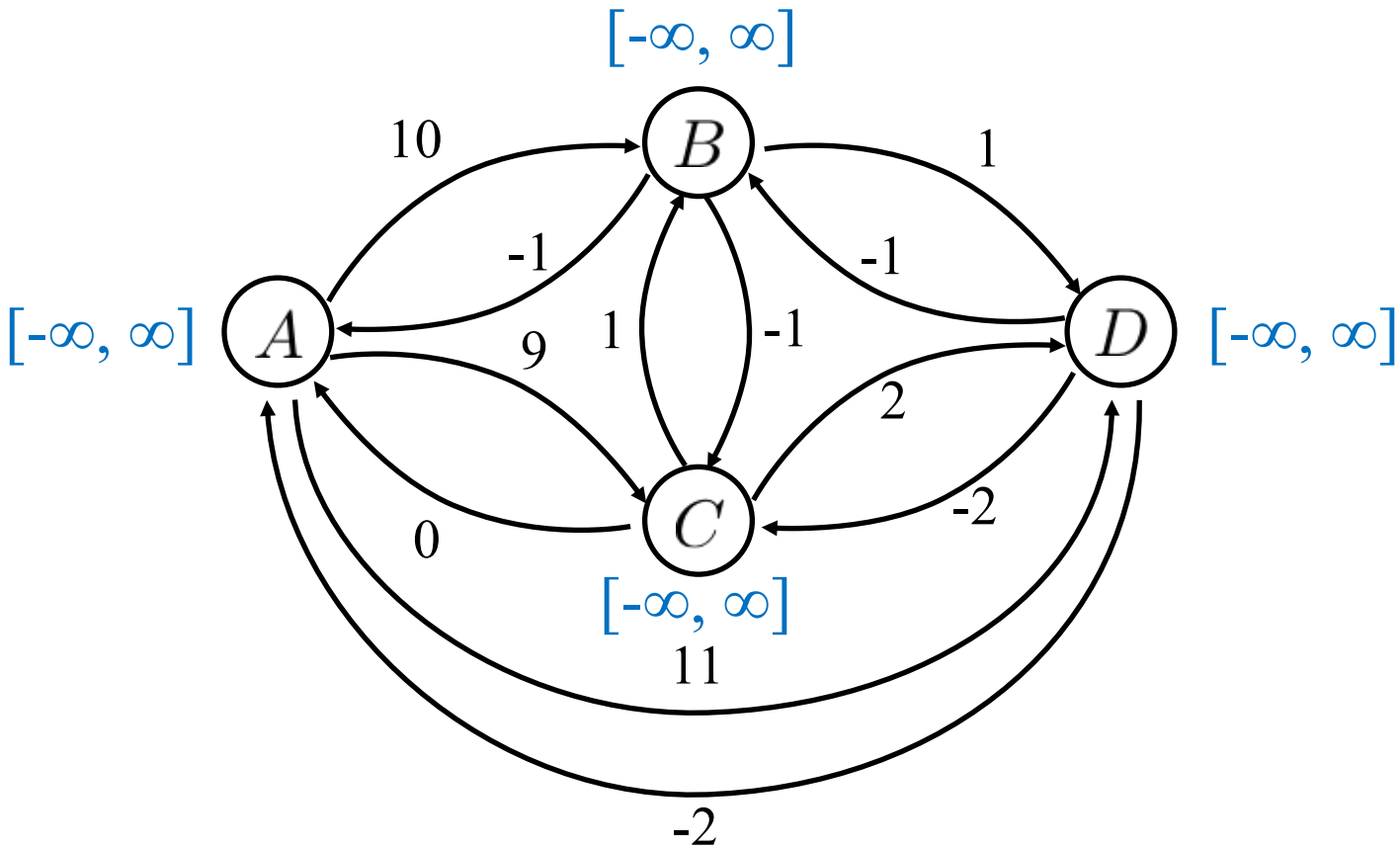
	A	B	C	D
A	0	10	9	11
B	-1	0	-1	1
C	0	1	0	2
D	-2	-1	-2	0



Computing a schedule

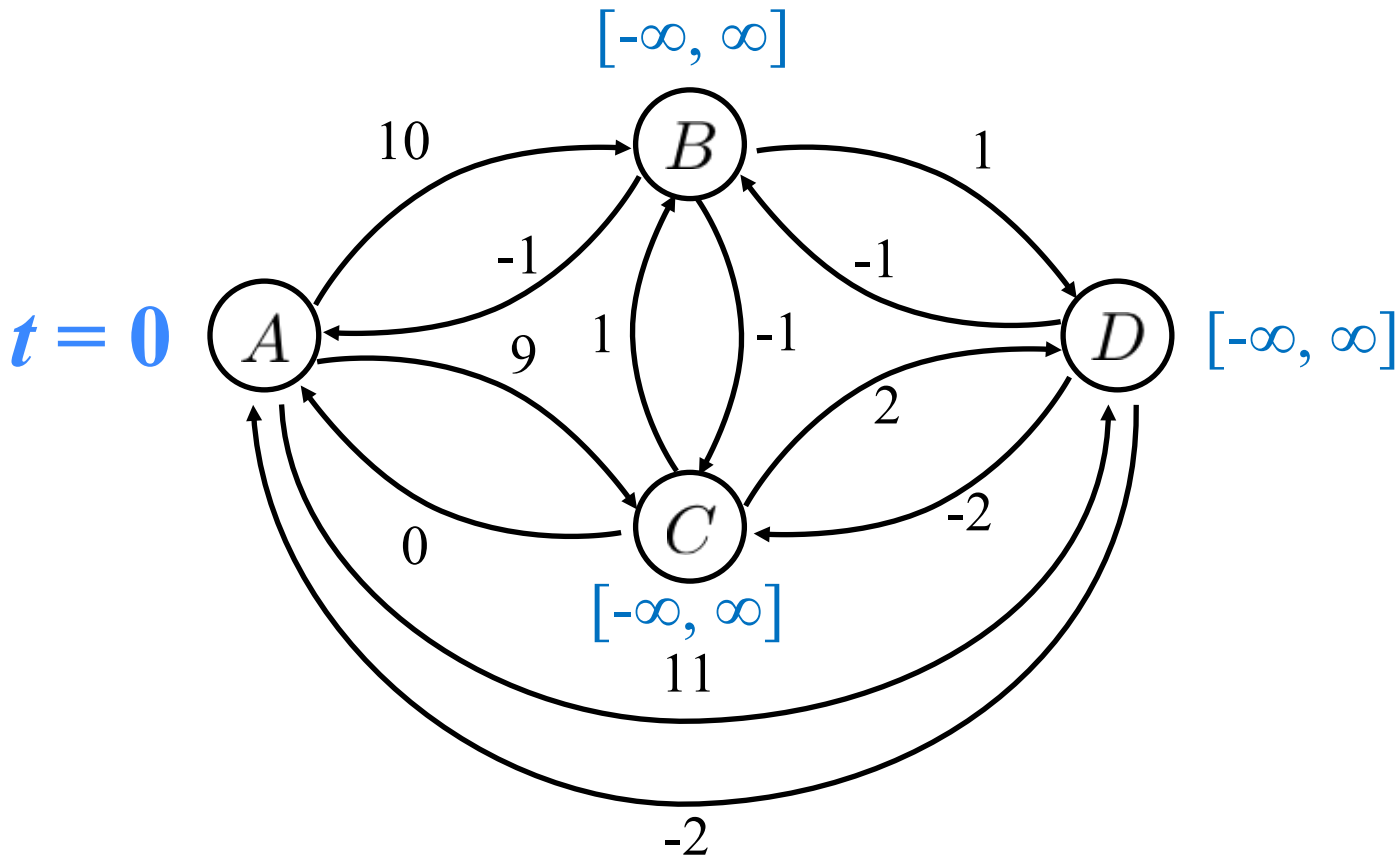


Computing a schedule



Initialize execution windows for each event in the plan

Computing a schedule

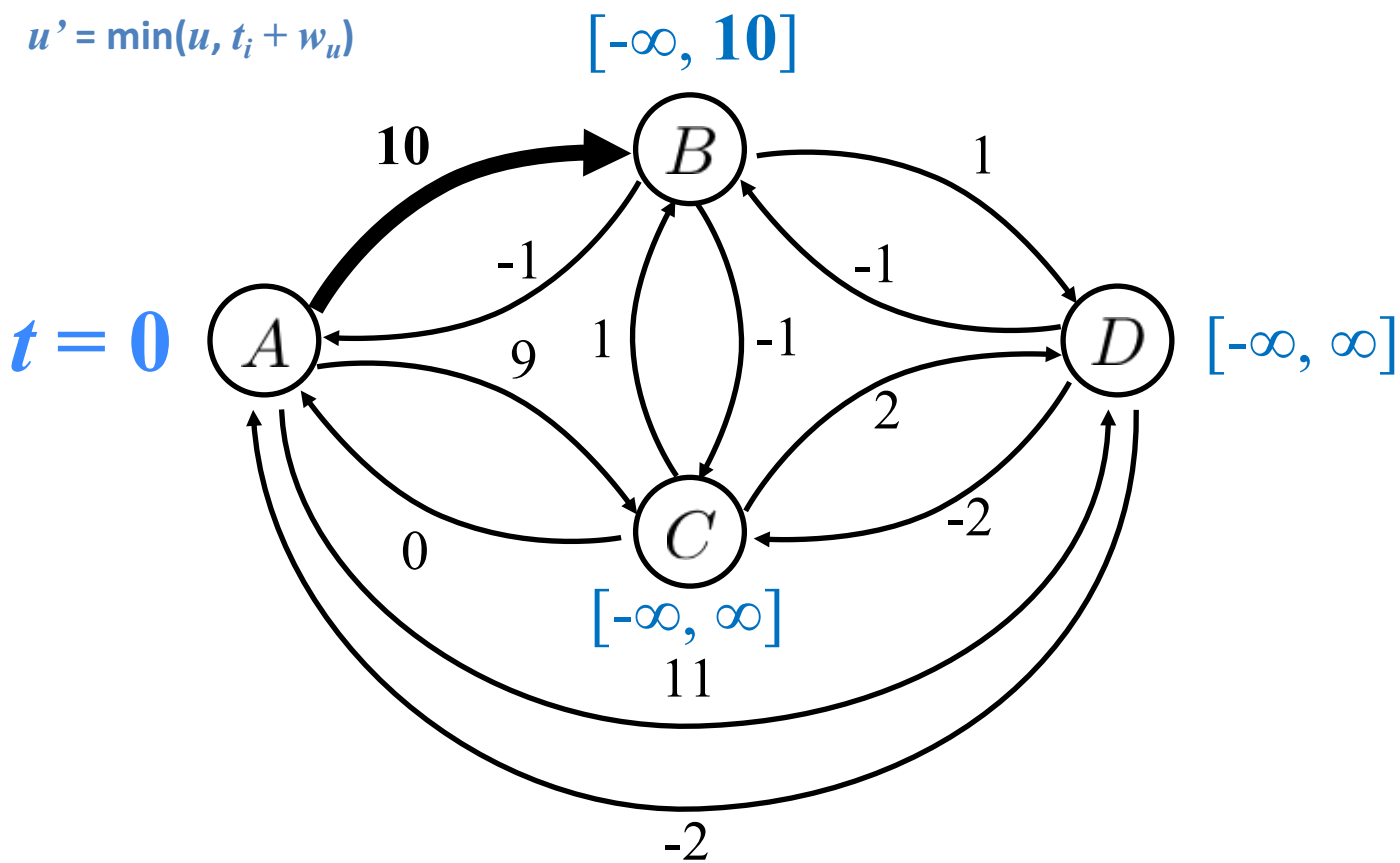


Assign the first event

Computing a schedule

Forward : outgoing edges to neighbor

$u' = \min(u, t_i + w_u)$

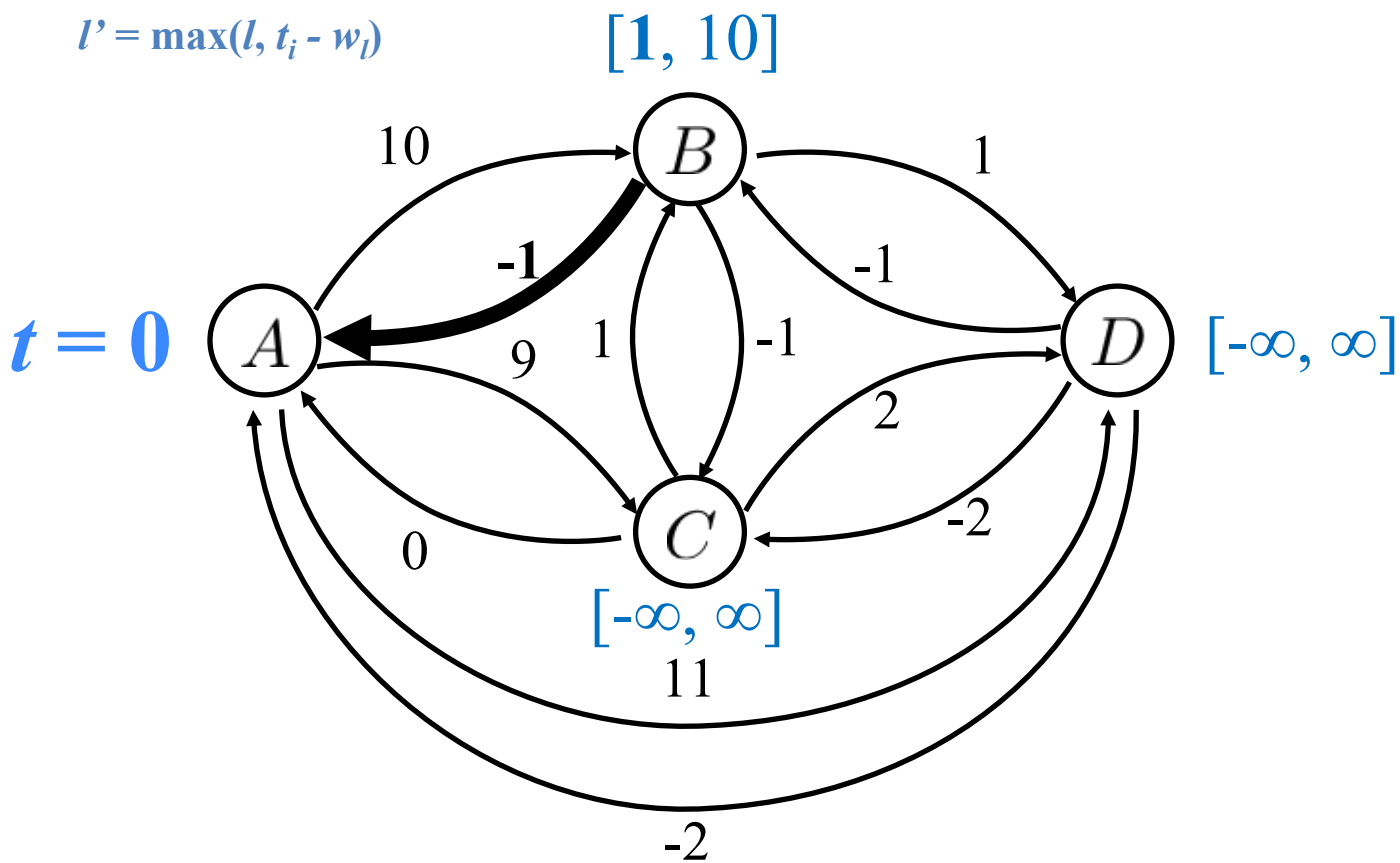


Propagate new time bounds to neighbors

Computing a schedule

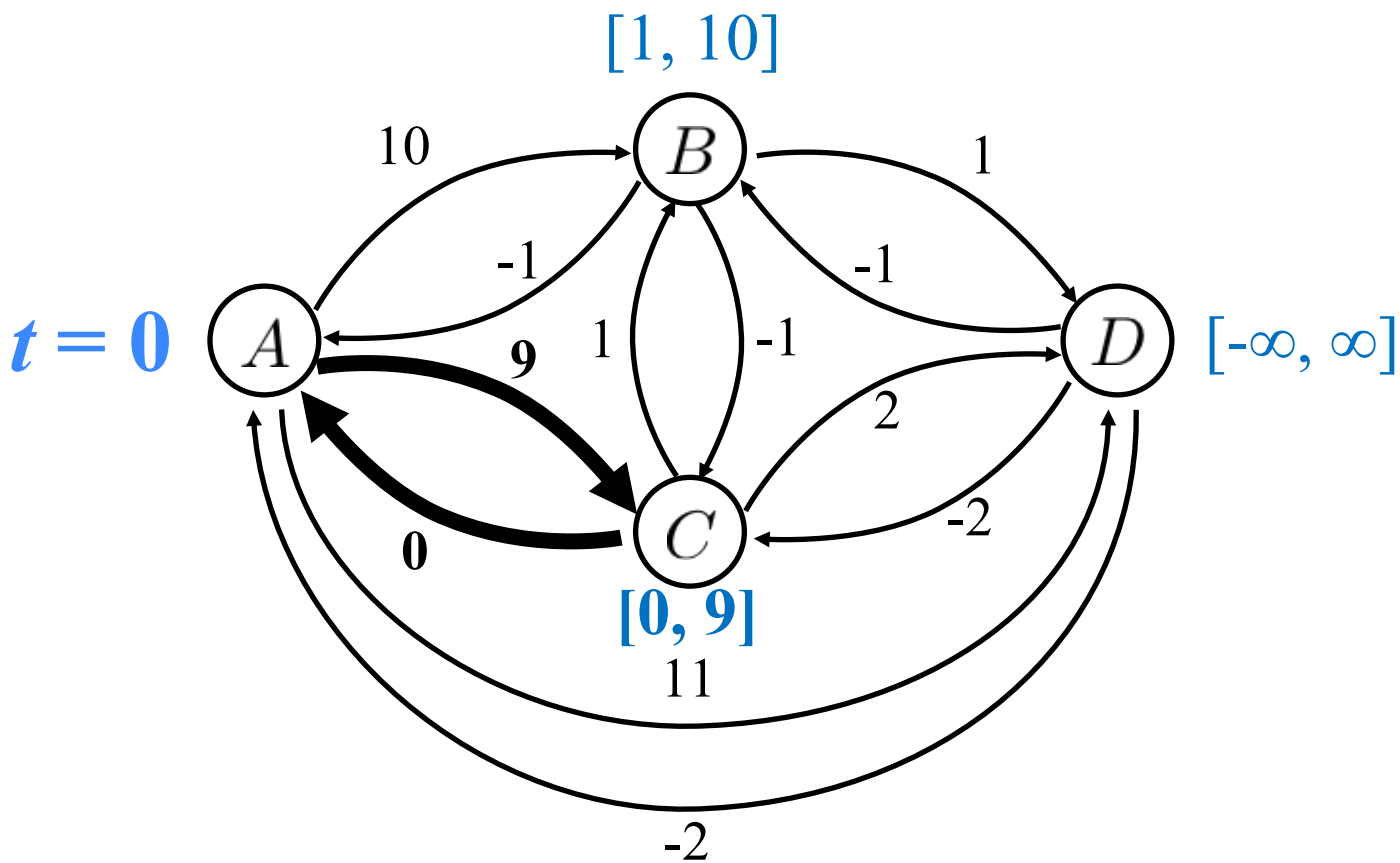
Backward: incoming edges from neighbor

$l' = \max(l, t_i - w_l)$



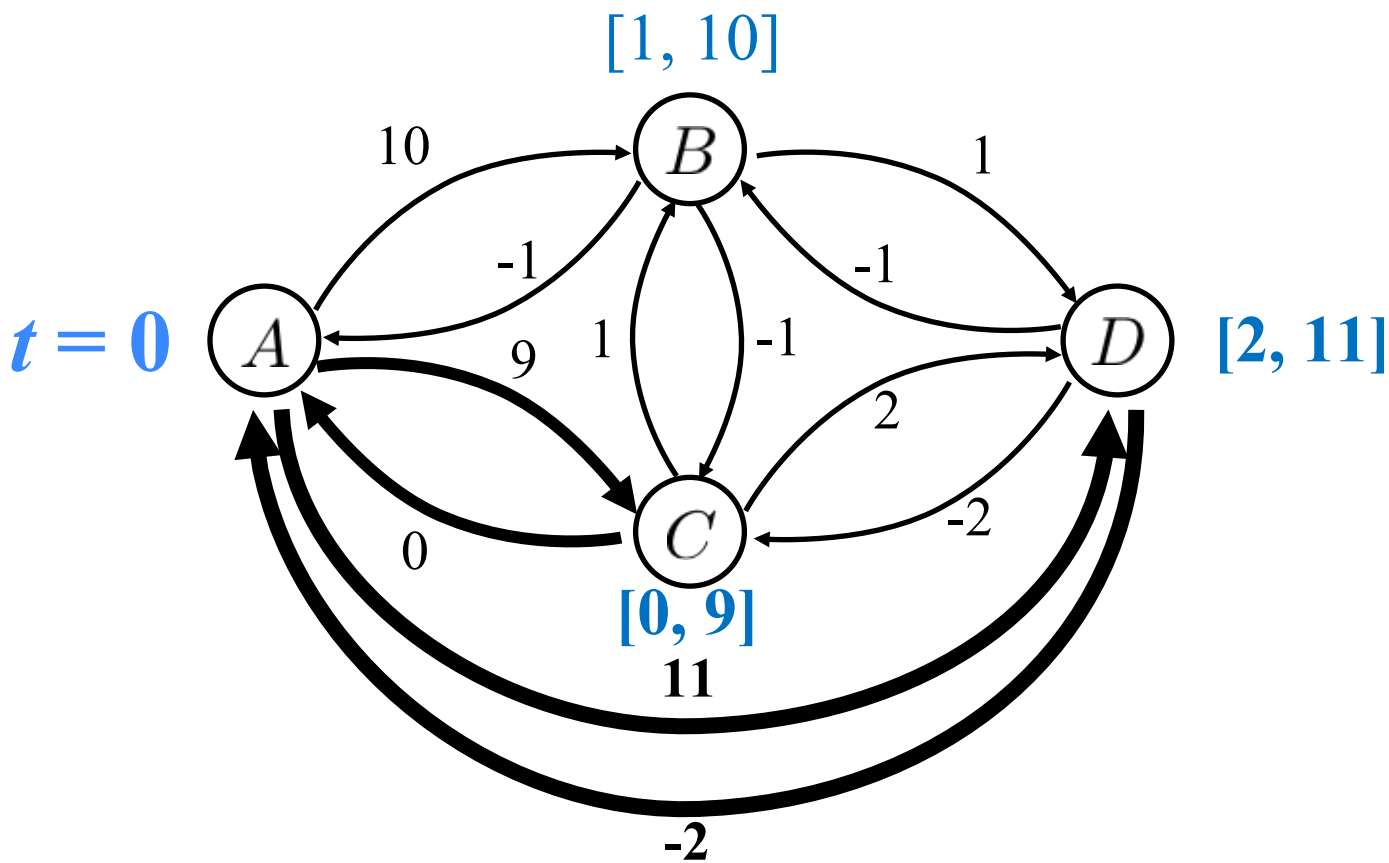
Propagate new time bounds to neighbors

Computing a schedule



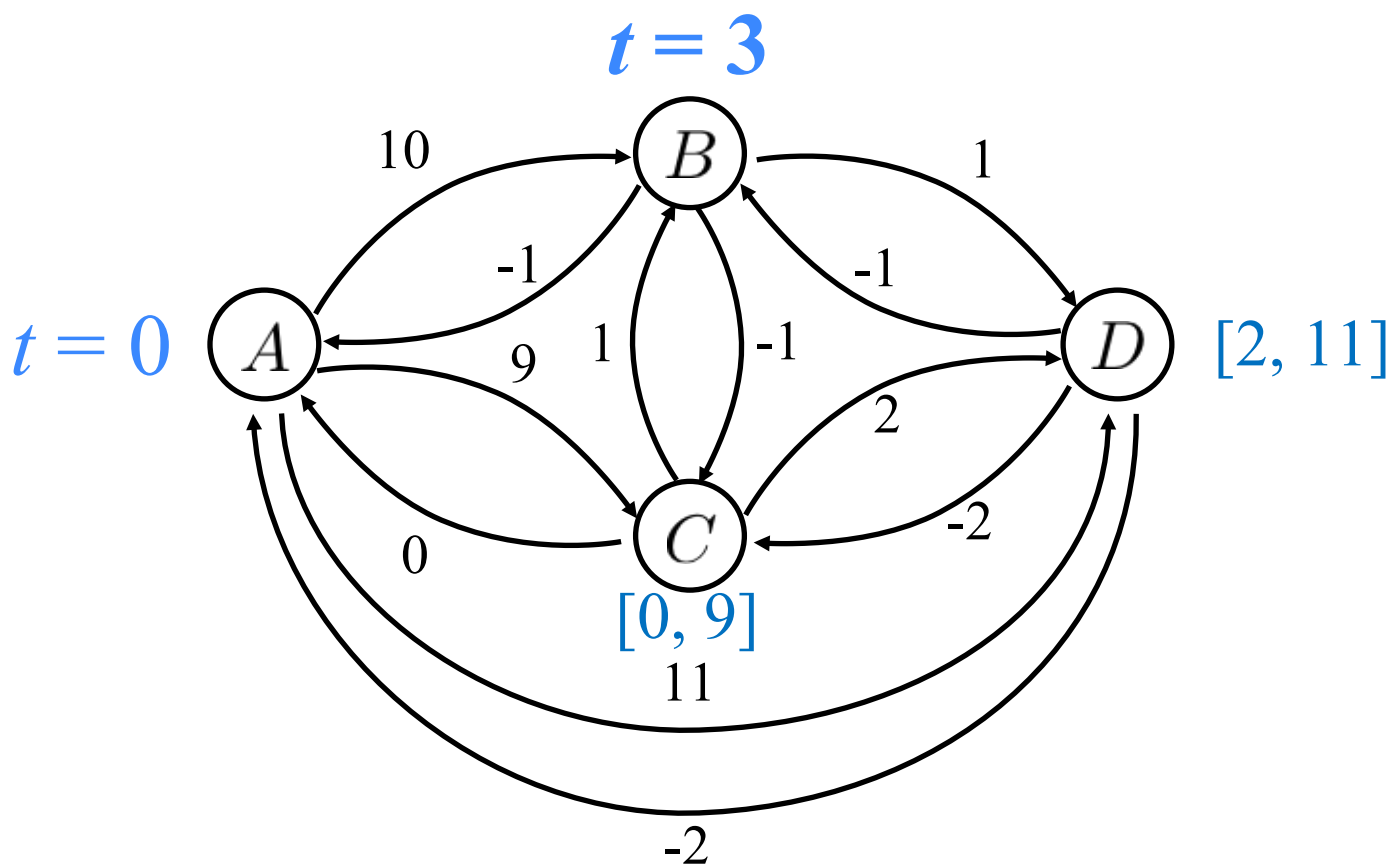
Propagate new time bounds to neighbors

Computing a schedule



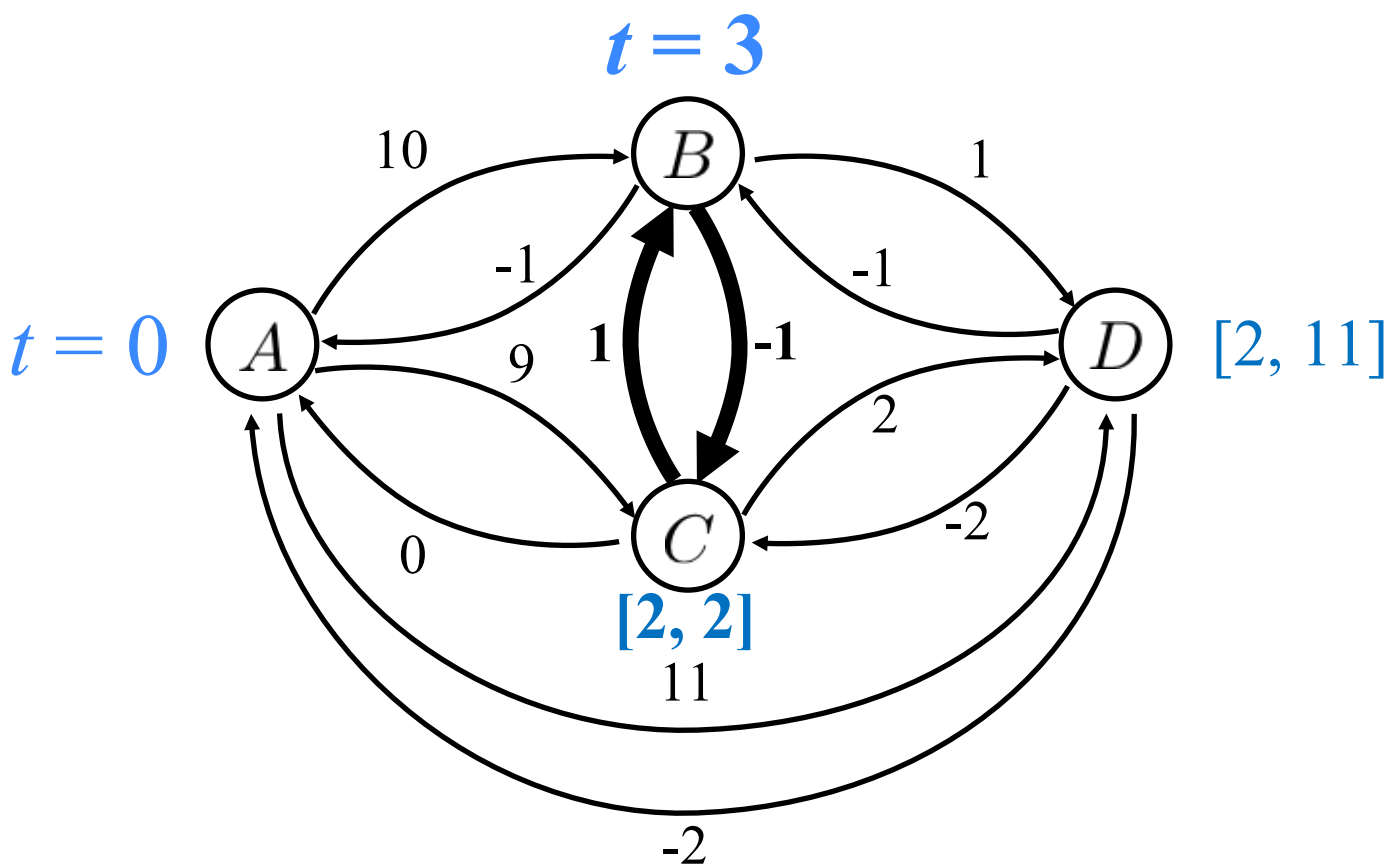
Propagate new time bounds to neighbors

Computing a schedule



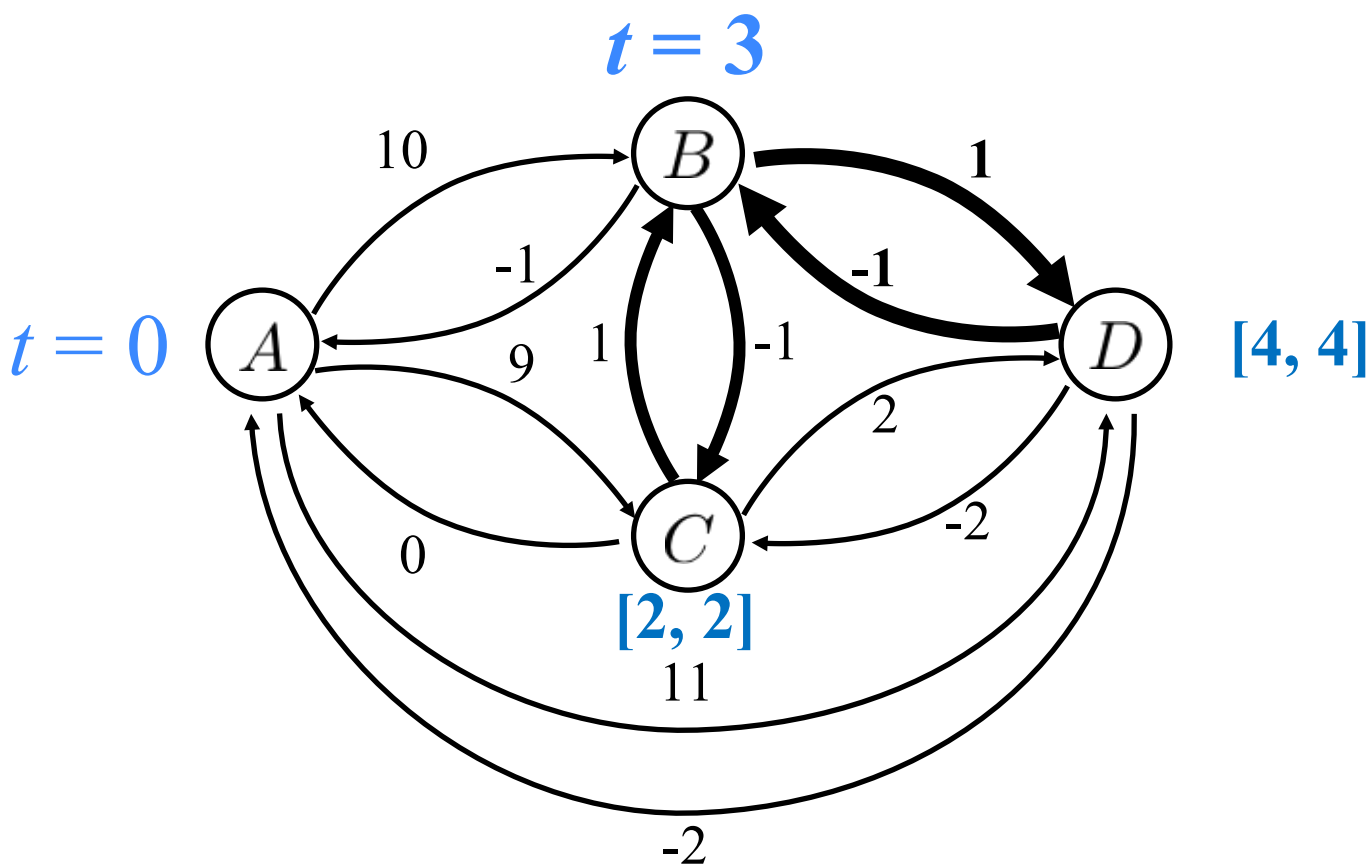
Arbitrarily pick another event and assign it...

Computing a schedule



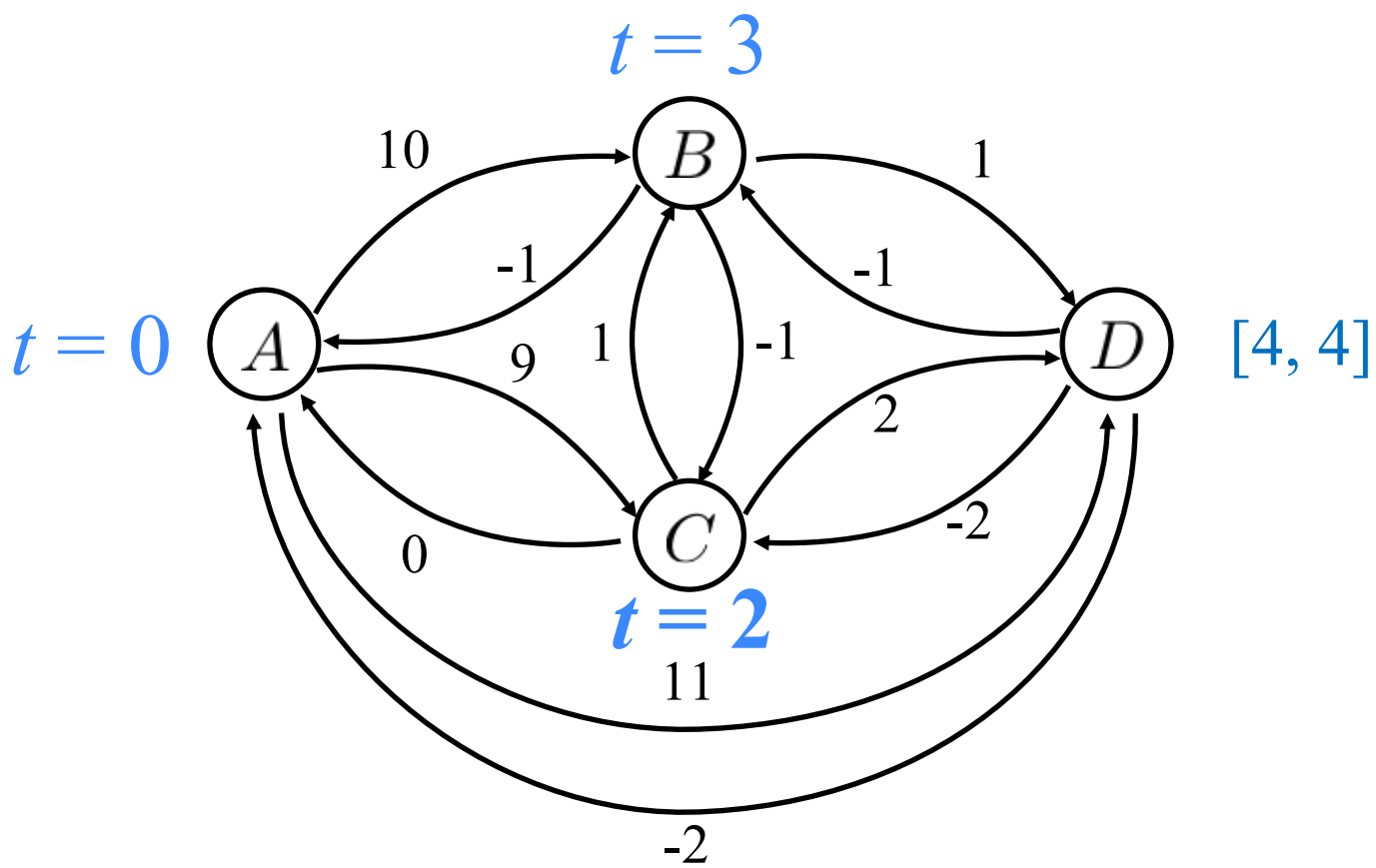
Propagate new time bounds to neighbors

Computing a schedule



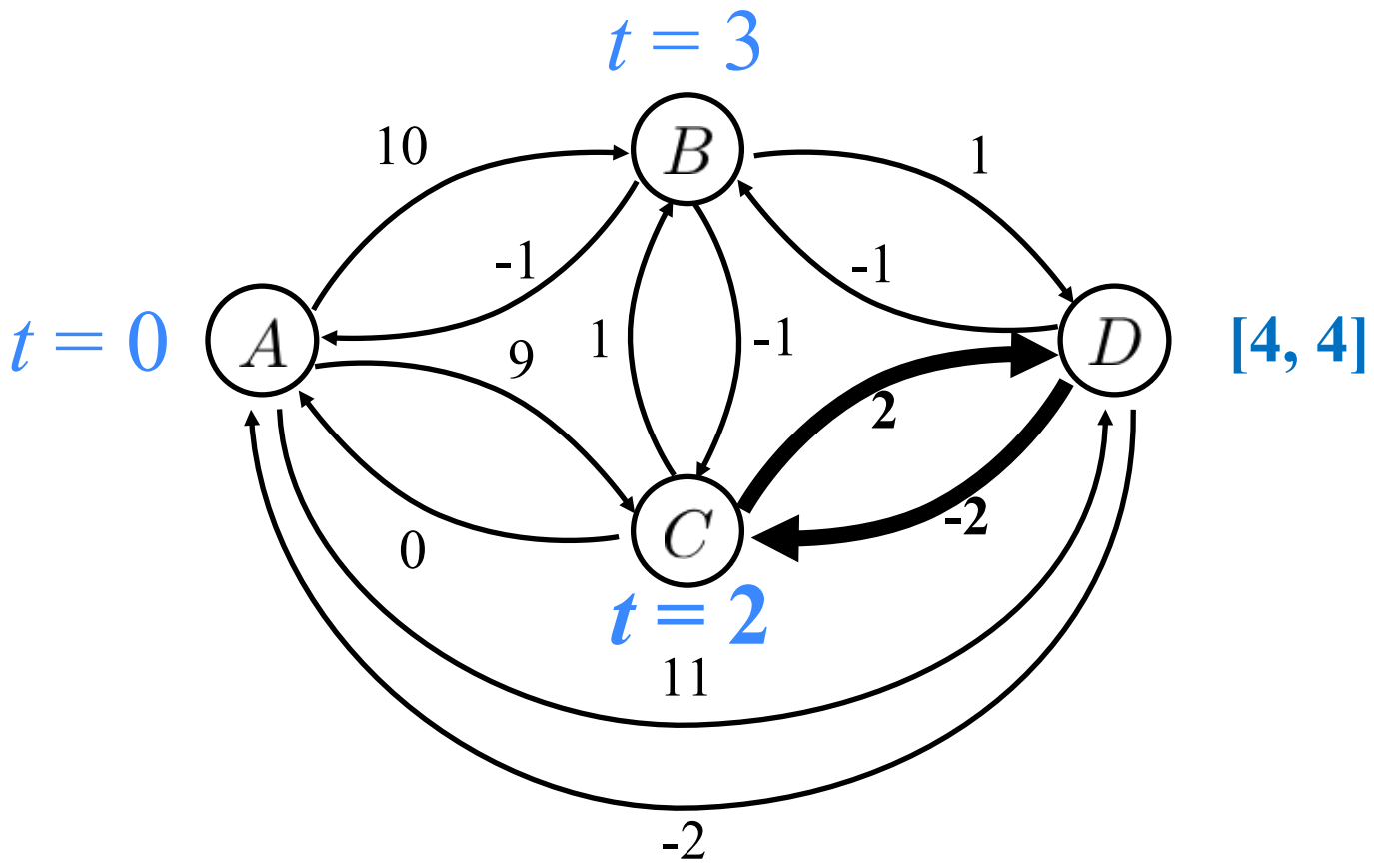
Propagate new time bounds to neighbors

Computing a schedule



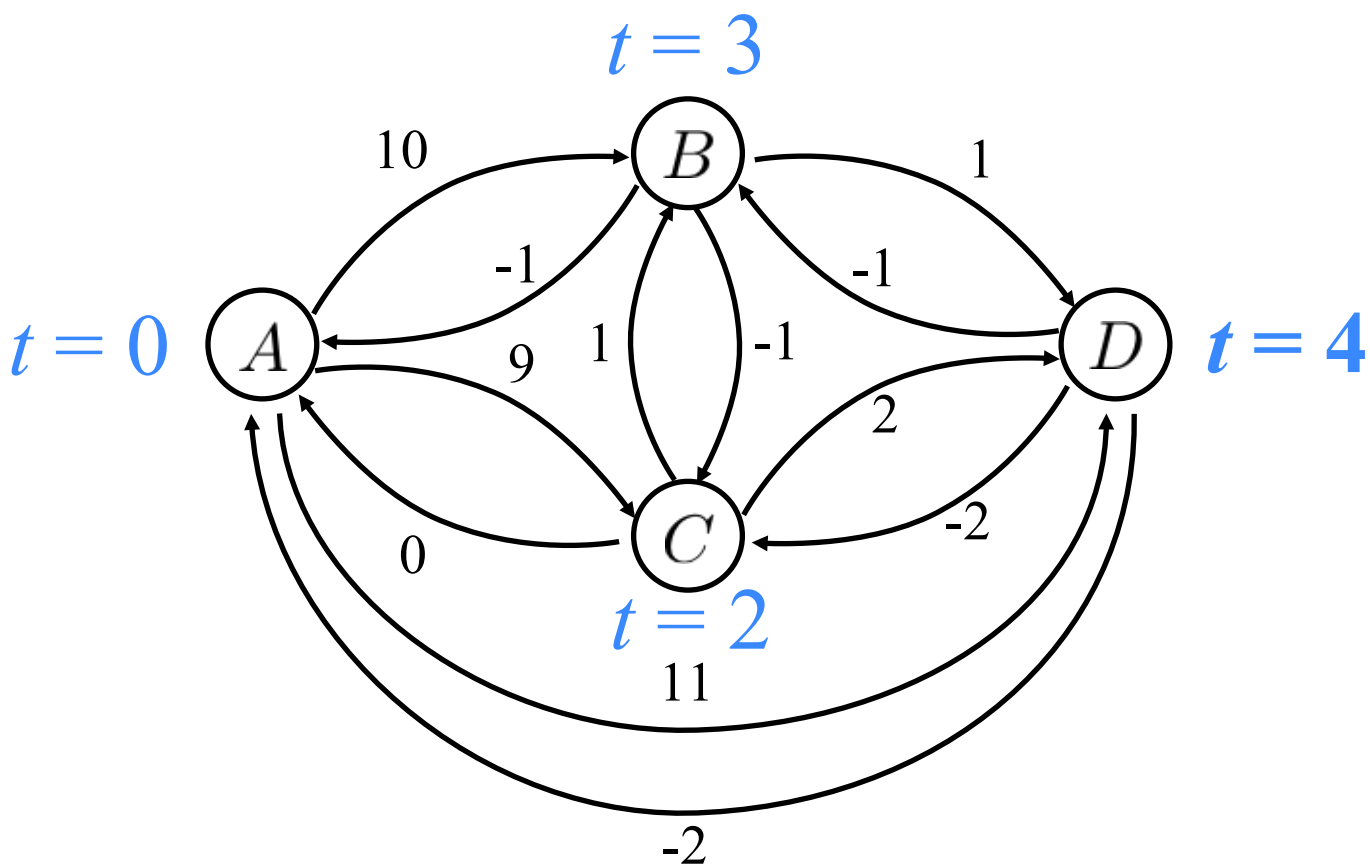
Pick another event and assign it

Computing a schedule



Propagate to neighbors

Computing a schedule



Assign the final event

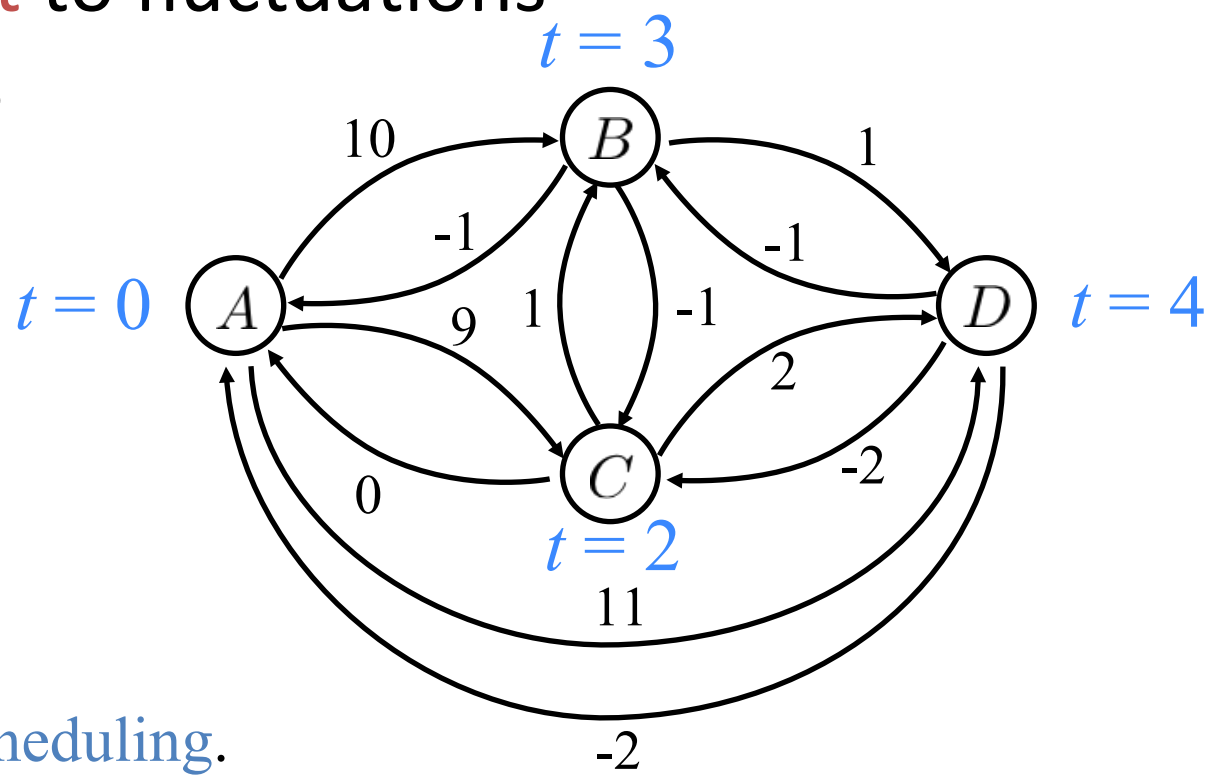
Issue: Precomputed schedules are not robust to fluctuations

Suppose C is delayed?

$t_C = 2.00001$

$t_C = 5$

$t_C = 100$



Idea: Dynamic scheduling.

- Schedule events "on the fly," after observing past event times.

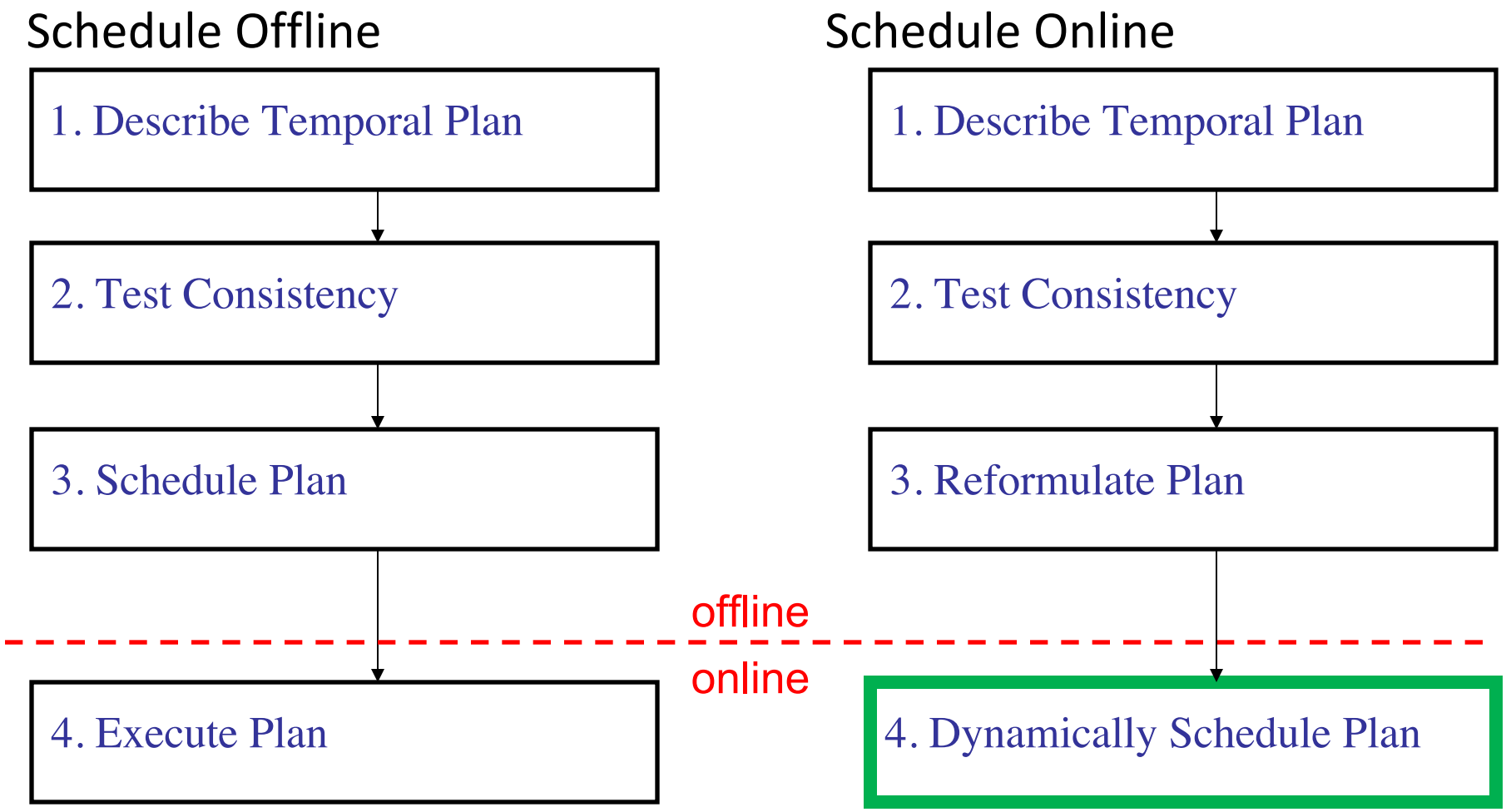
➢ Increases robustness to fluctuations.

➢ Enabled by **flexible temporal constraints**!

Outline

- Non-deterministic programs and flexible execution
- Describing temporal plans
- Offline scheduling
- **Online scheduling and dispatch**
  - Dispatching activities
  - Reformulating for faster online execution (Appendix)

To Execute a Temporal Plan



How do we schedule online?

An incorrect approach:

- Apply offline scheduling algorithm online.
  - **Wait** until **current time** is in an event's **execution window** ("**active**").
  - **Assign active event** the current time and propagate.

Simple (incorrect!) online scheduler

Compute decomposable form (i.e., APSP)

Initialize execution window to [-∞, ∞] for each event

while unexecuted events:

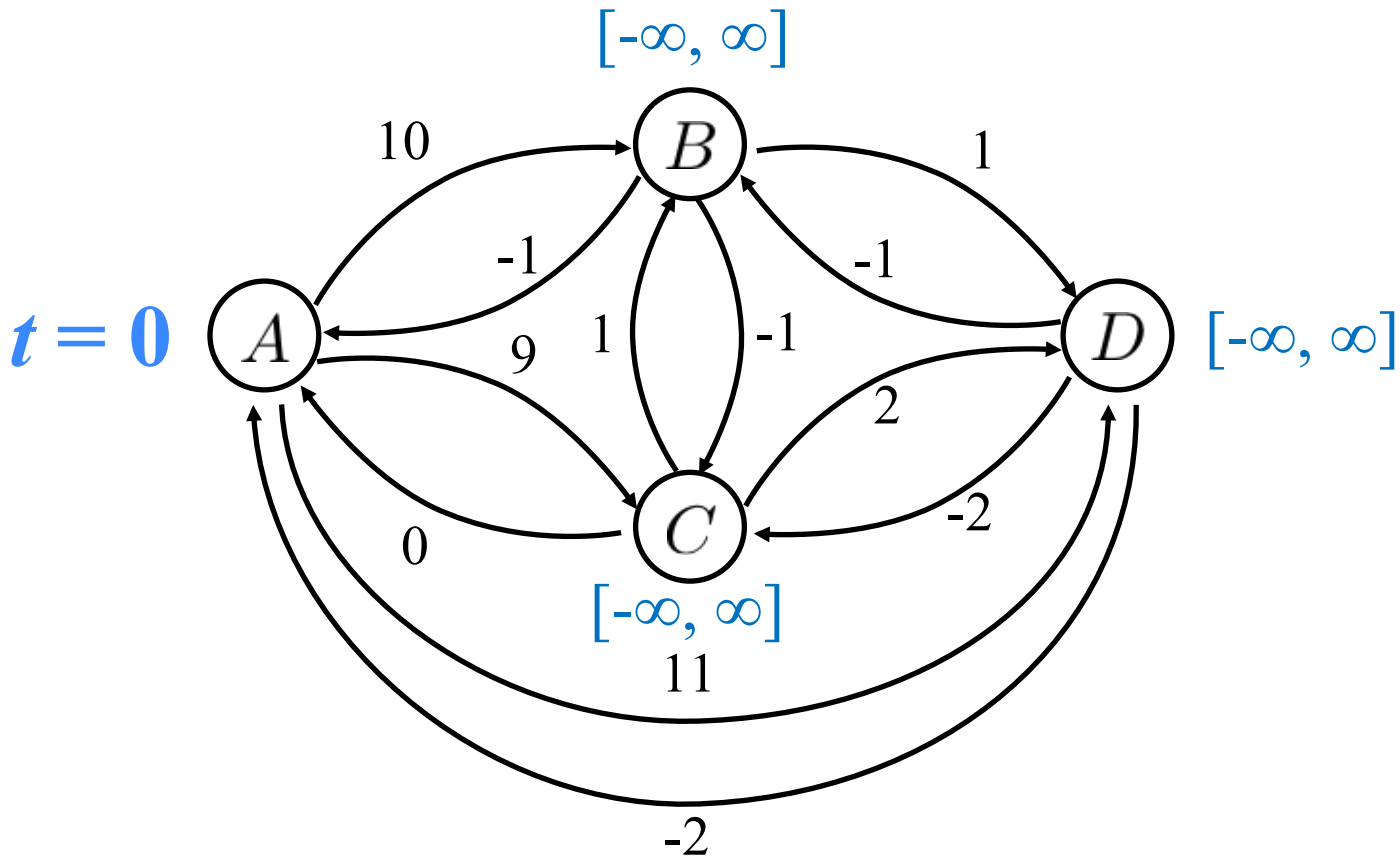
x_i = pick any unexecuted event

such that current time in window

t_i = **now**

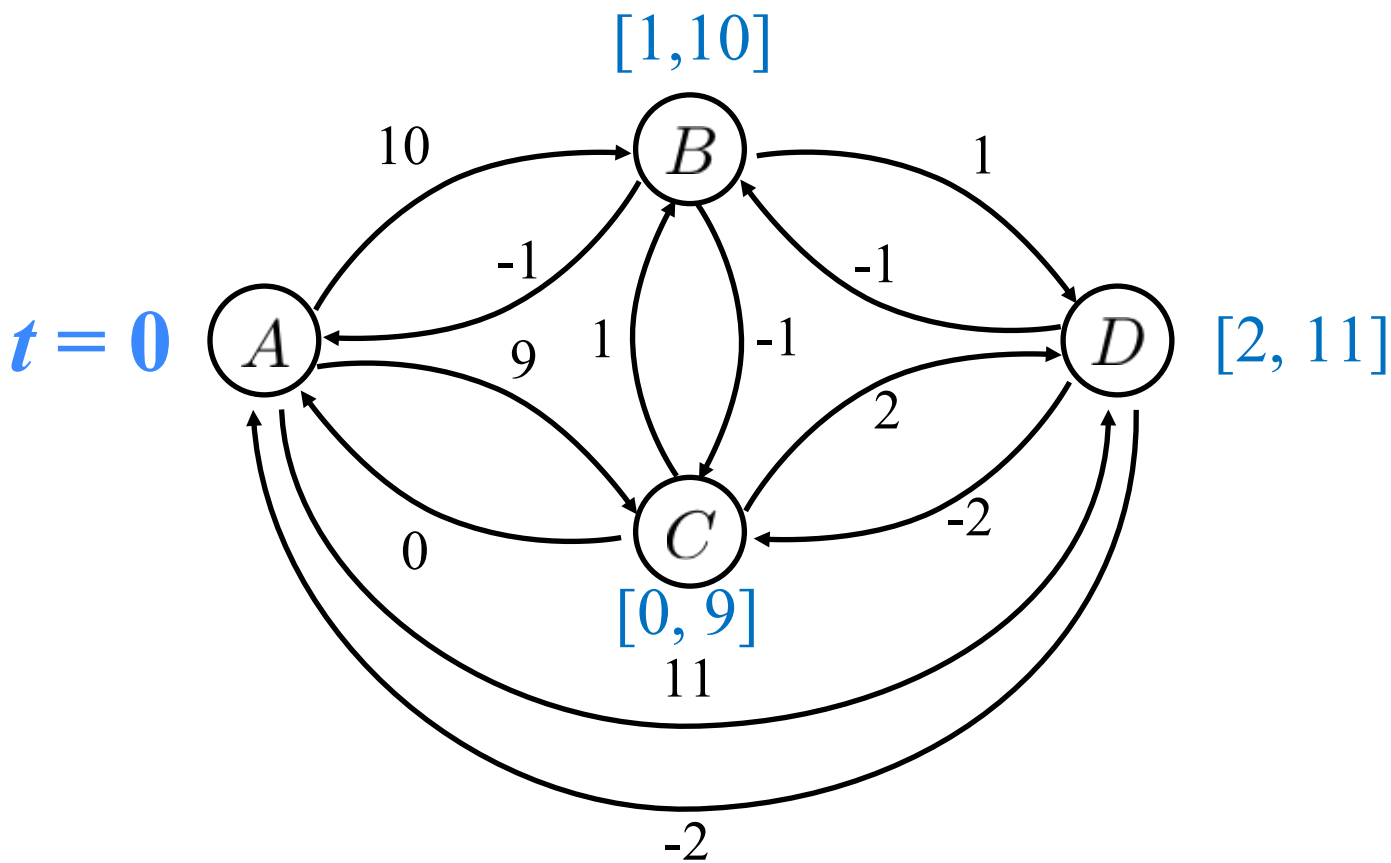
Propagate to all x_i's neighbors & update their windows

Simple (incorrect!) online scheduling



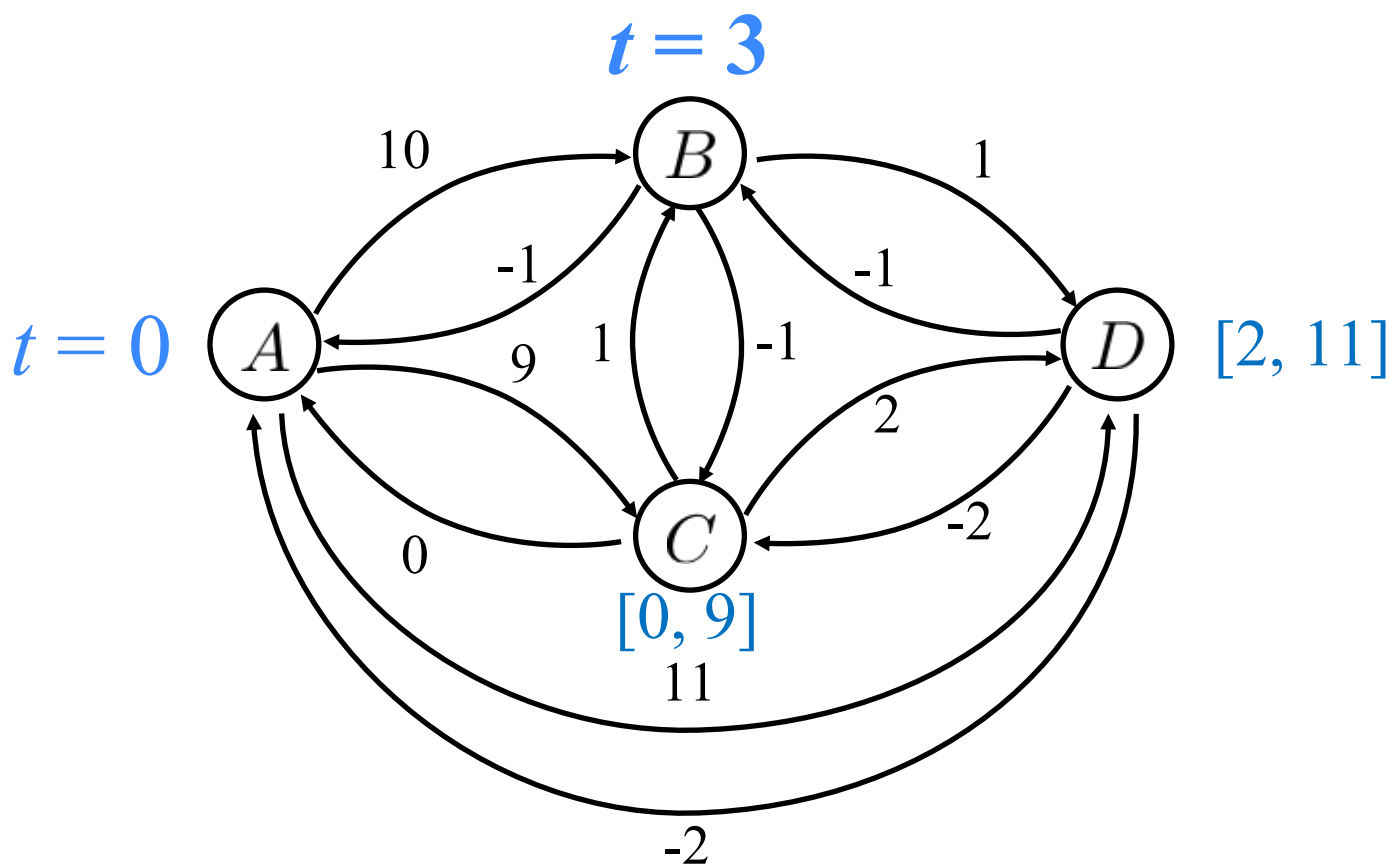
Arbitrarily pick next event…

Simple (incorrect!) online scheduling



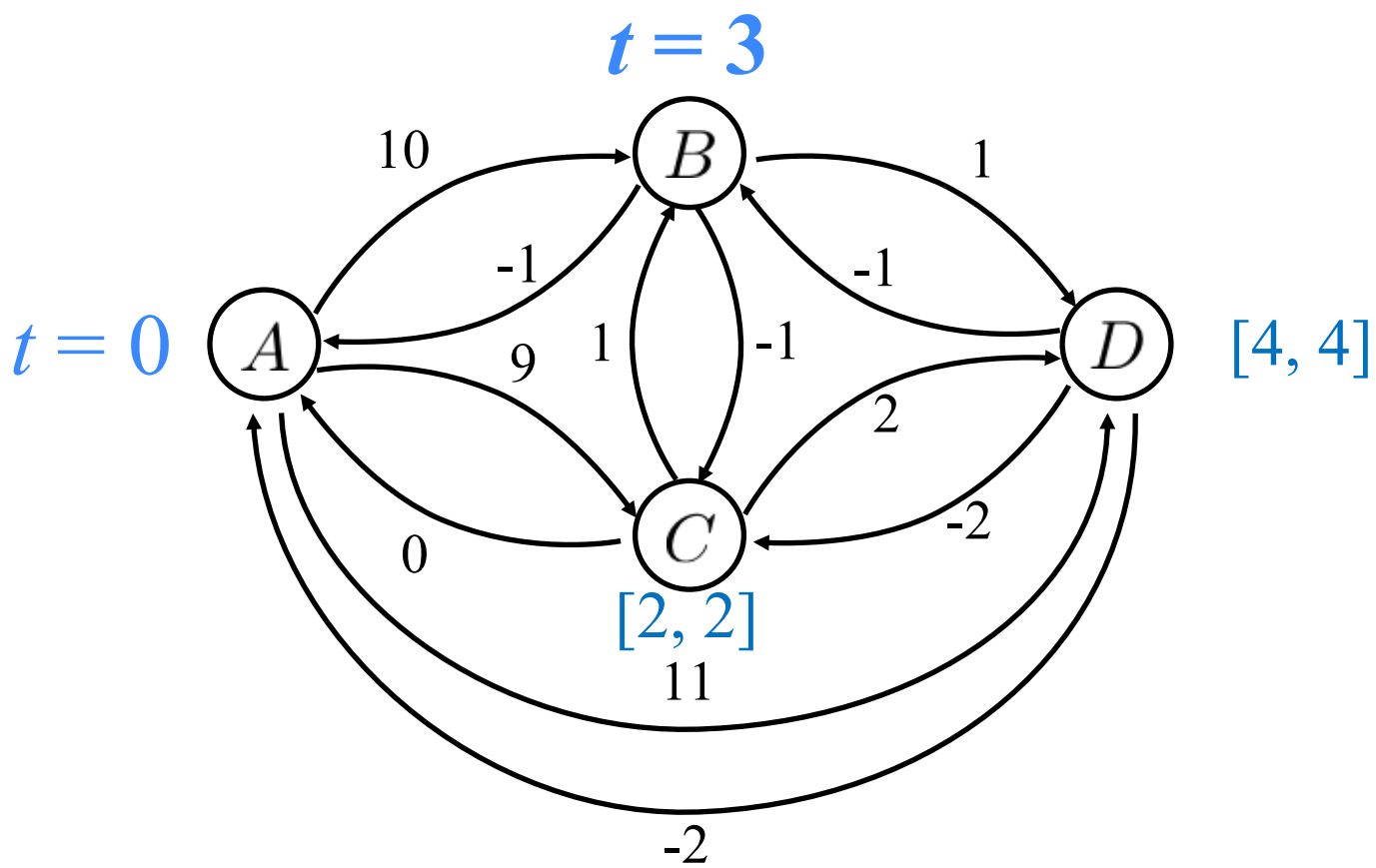
Arbitrarily pick next event…

Simple (incorrect!) online scheduling



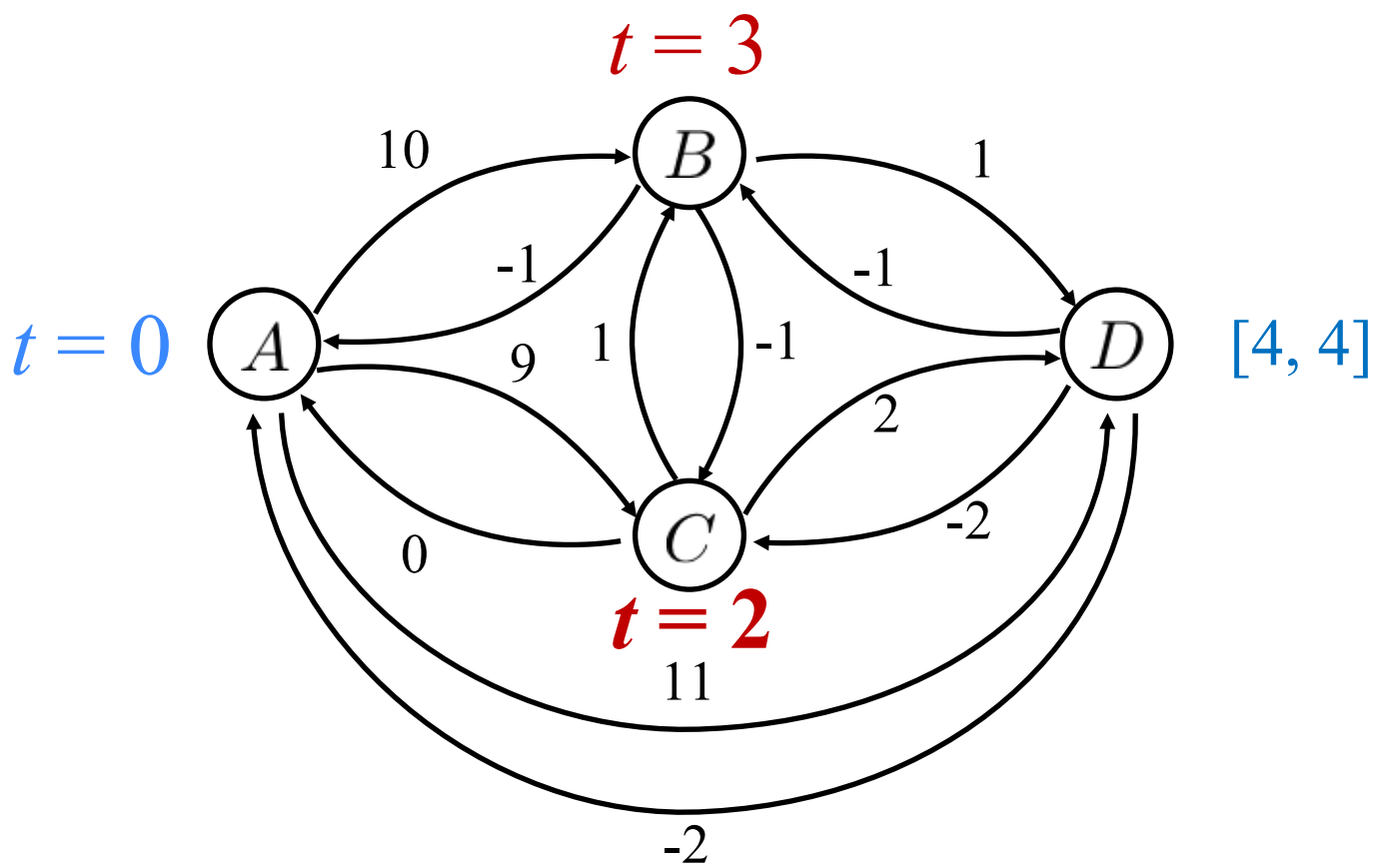
Arbitrarily pick next event…

Simple (incorrect!) online scheduling



Arbitrarily pick next event…

Simple (incorrect!) online scheduling



...but wait! We just ***assigned a past time!***

Issue: Time Always Marches Forward

- Offline: Can assign in any order.
- Online: Must assign monotonically increasing times.

How can online scheduling achieve this?

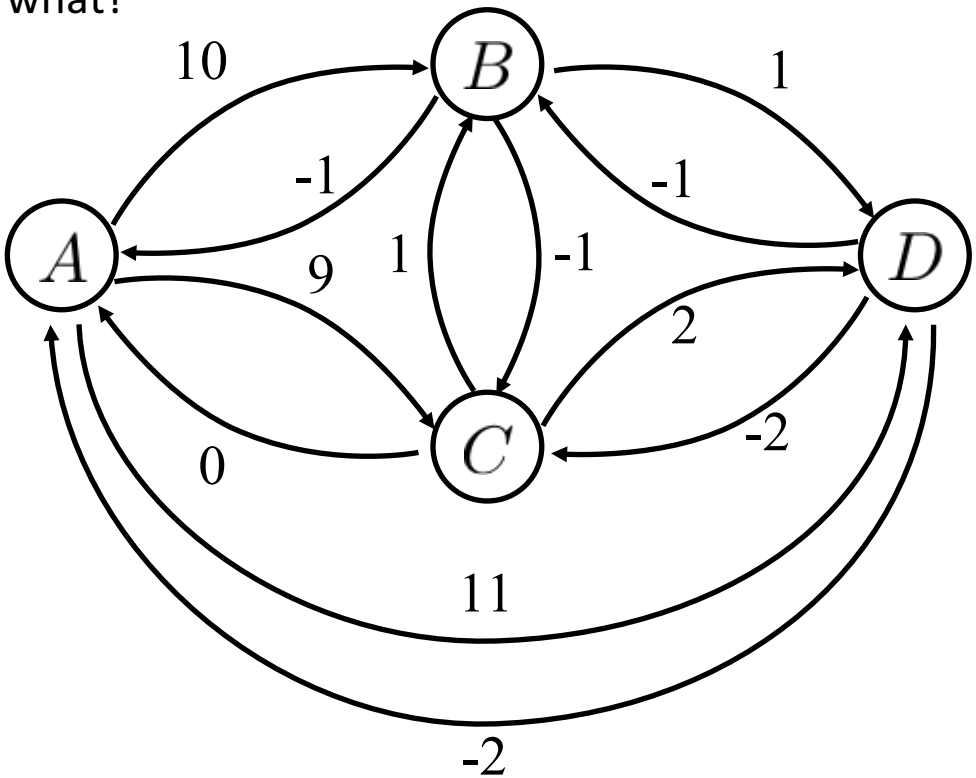
Solution:

1. Infer when one event **must precede** another.

2. "**Enable**" an event to be scheduled only **after** predecessors are scheduled.

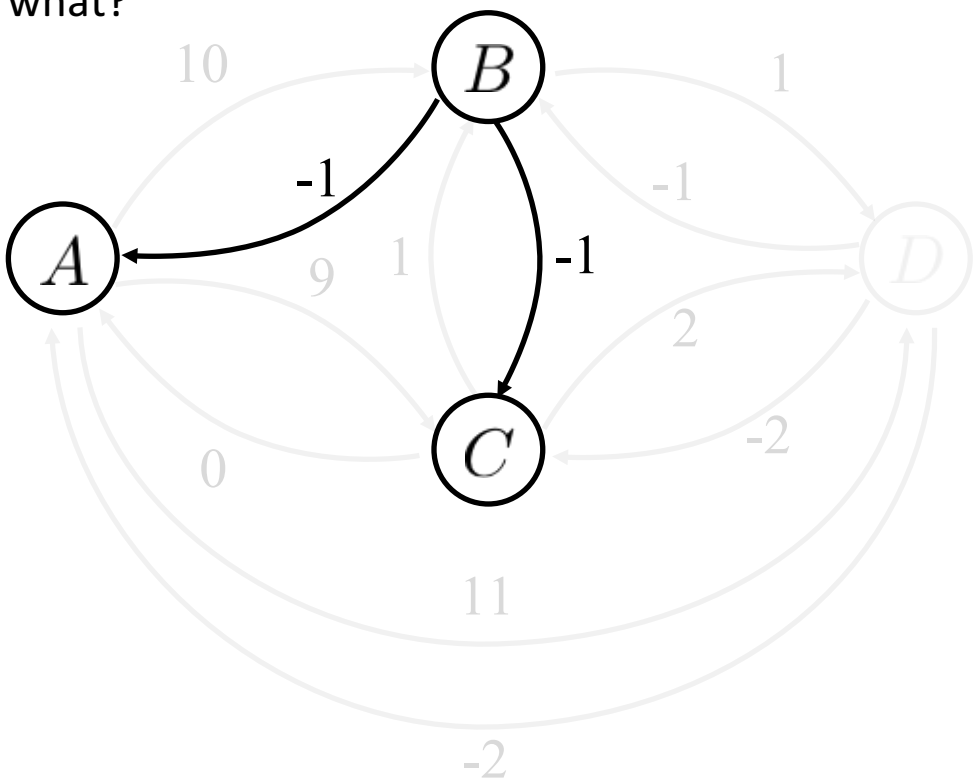
Idea: Constrain order of dispatched events

- Negative edges from APSP dictate ordering constraints
  - B follows what?



Idea: Constrain order of dispatched events

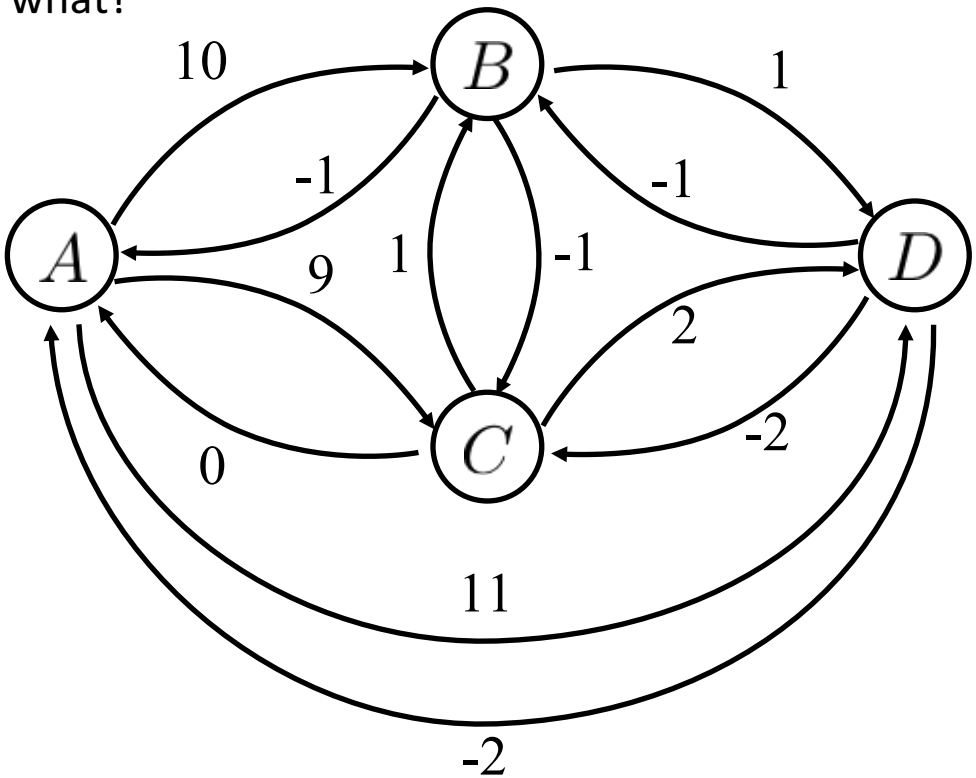
- Negative edges from APSP dictate ordering constraints
  - B follows what?



B must occur after both A and C!

Idea: Constrain order of dispatched events

- Negative edges from APSP dictate ordering constraints
  - D follows what?



D must occur after A, B and C!

Idea: Constrain order of dispatched events

- An event e is ***enabled*** if all its **predecessors** have been "dispatched" (scheduled).
  - **A "predecessor" is the target of a negative edge from e.**

- Online algorithm only dispatches events that are **enabled**.
  - An event is ***active*** if
    - a) enabled
    - b) current time in event's execution window.

Corrected online scheduler

Compute decomposable form (i.e., APSP)

Initialize execution window to [-∞, ∞] for each event

E ← {events with no predecessors} # set of enabled events

S ← {} # set of executed events

while unexecuted events:

Wait until some event x_i in E is active

t_i = now # dispatch x_i now at t_i

Propagate to all x_i's neighbors & update their windows

Add x_i to S

Add to *E* any now-enabled events

Running online scheduler

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events

Running online scheduler

$A, C < B \qquad A, B, C < D$

$E = ?$

$S = \{\}$

Compute decomposable form (i.e., APSP)

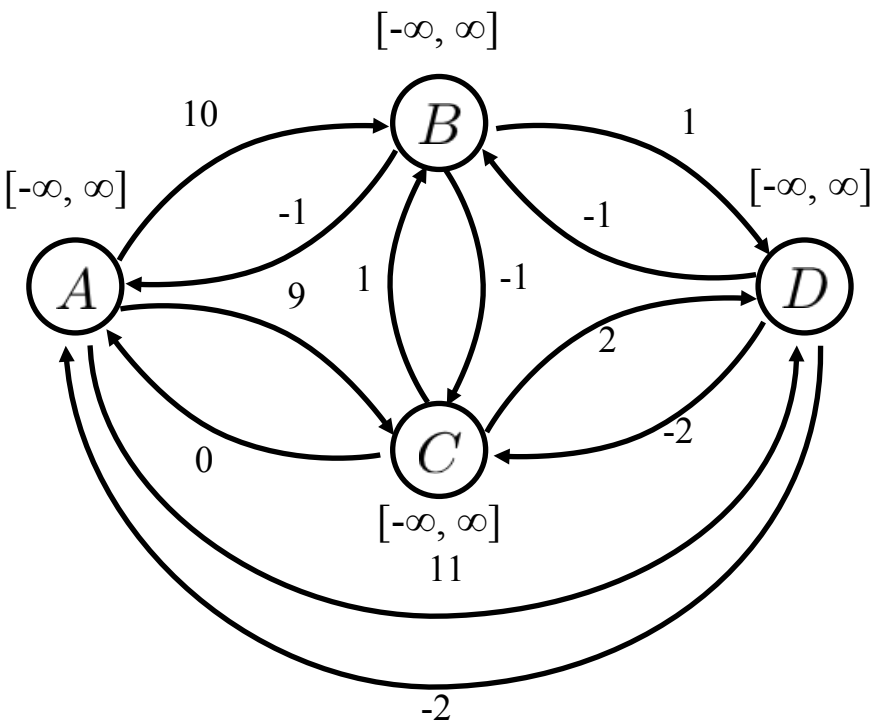
Initialize execution windows to [-∞, ∞]

$E \leftarrow$ {events with **no predecessors**}

$S \leftarrow \{\}$

while unexecuted events:

- Wait until some event x_i in E is **active**
- t_i = now
- Propagate to x_i's neighbors
- Add x_i to S
- Add to E any **now-enabled** events

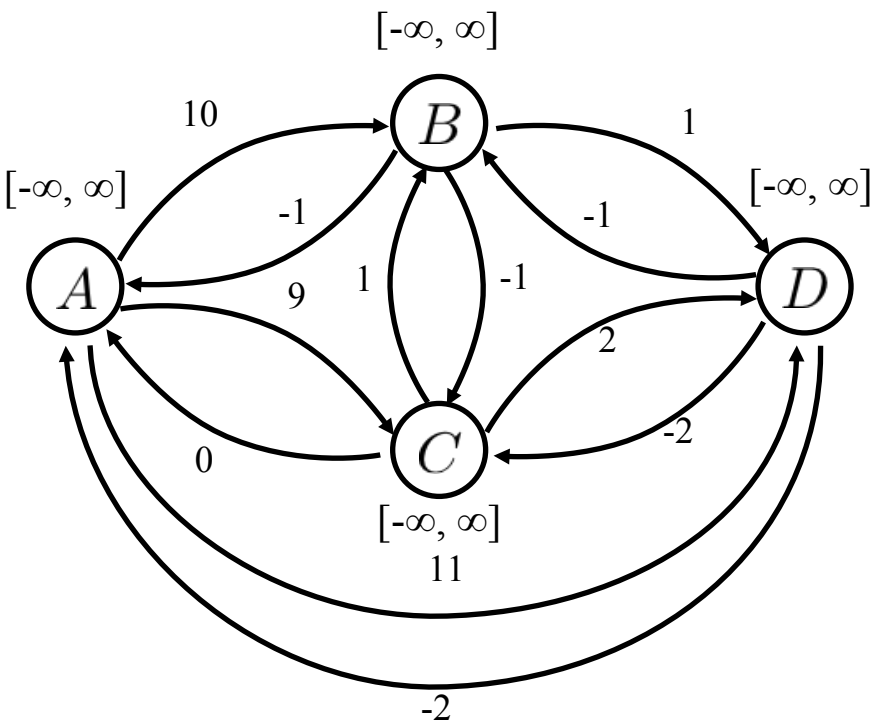


Running online scheduler

$A, C < B \qquad A, B, C < D$

$E = \{A, C\}$
$S = \{\}$

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
$E \leftarrow$ {events with no predecessors}
$S \leftarrow \{\}$
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events



A, C initially in *E*
– have no negative, outgoing edges

Running online scheduler

$A, C < B \qquad A, B, C < D$

$E = \{A, C\}$

$S = \{\}$

Compute decomposable form (i.e., APSP)

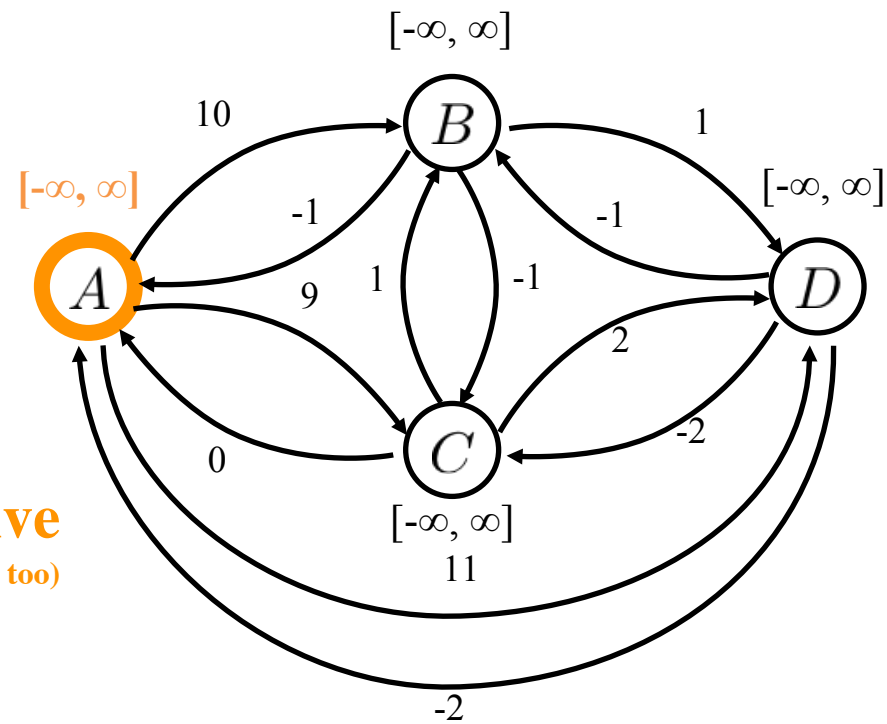
Initialize execution windows to [-∞, ∞]

$E \leftarrow$ {events with no predecessors}

$S \leftarrow$ {}

while unexecuted events:

- **Wait until some event x_i in E is active**
- t_i = now
- Propagate to x_i's neighbors
- Add x_i to S
- Add to E any now-enabled events



For t=0, A is Active

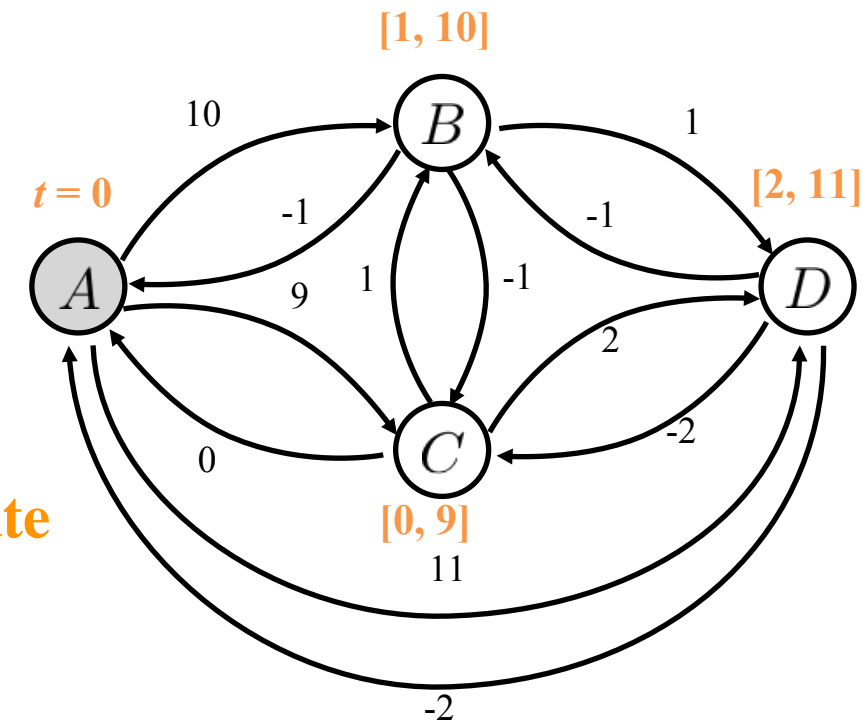
(could have chosen C too)

Running online scheduler

$A, C < B \qquad A, B, C < D$

$E = \{C\}$
$S = \{A\}$

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events



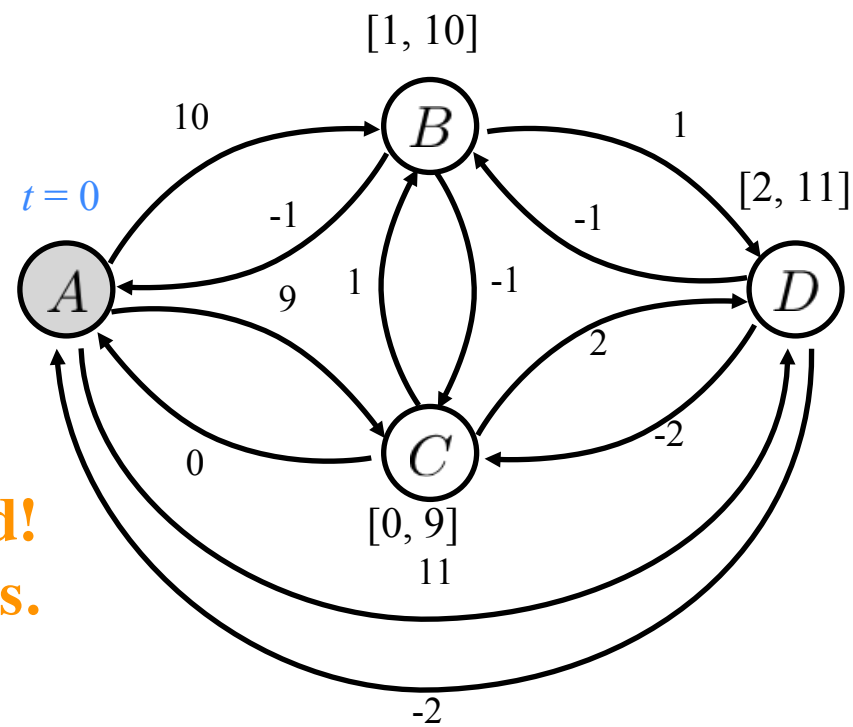
Dispatch A and propagate

Running online scheduler

A, C < B **A, B, C < D**

$E = \{C\}$
$S = \{A\}$

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events



B, D not yet enabled!
but C still is.

Running online scheduler

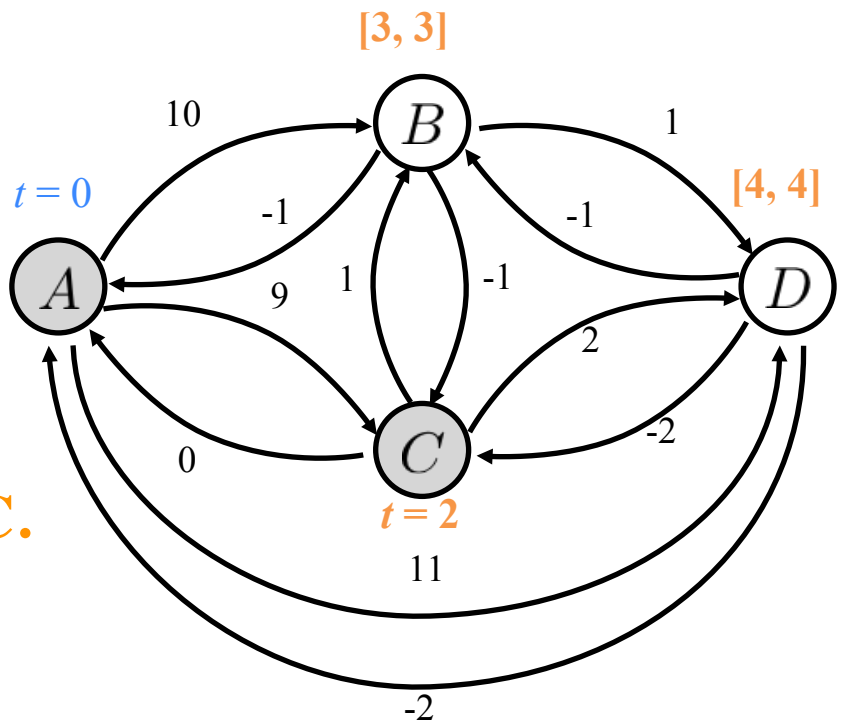
$A, C < B \qquad A, B, C < D$

$E = \{\}$
$S = \{A, C\}$

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:

- **Wait until some event x_i in E is active**
- **t_i = now**
- **Propagate to x_i's neighbors**
- **Add x_i to S**
- Add to E any now-enabled events

Dispatch & propagate C.

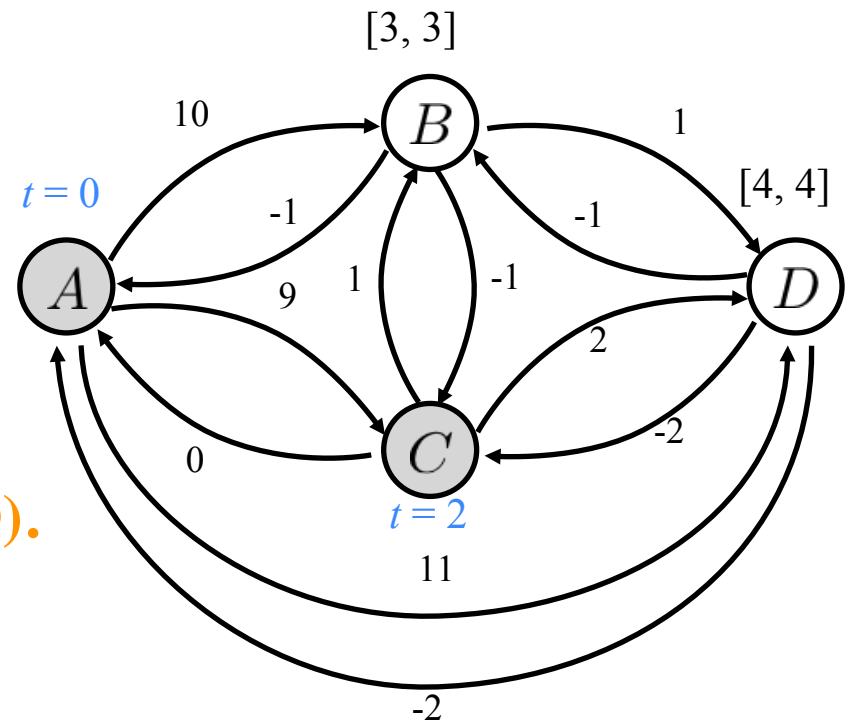


Running online scheduler

A, C < B **A, B, C < D**

$E = \{B\}$
$S = \{A, C\}$

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events



B is now enabled (but still not D).

Running online scheduler

$A, C < B \qquad A, B, C < D$

$E = \{\}$

$S = \{A, C, B\}$

Compute decomposable form (i.e., APSP)

Initialize execution windows to [-∞, ∞]

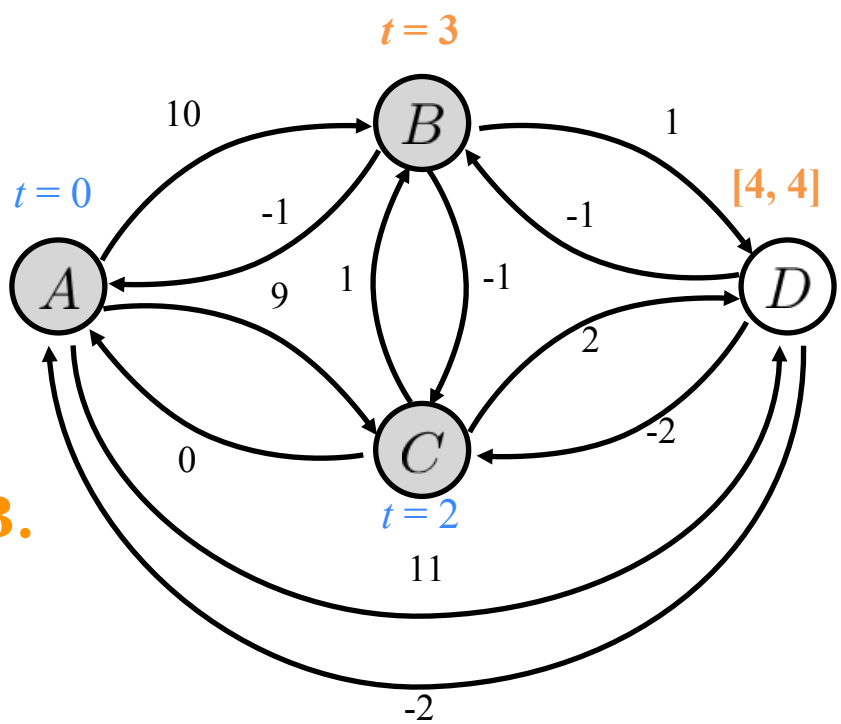
$E \leftarrow$ {events with no predecessors}

$S \leftarrow$ {}

while unexecuted events:

- **Wait until some event x_i in E is active**
- **t_i = now**
- **Propagate to x_i's neighbors**
- **Add x_i to S**
- Add to E any now-enabled events

Dispatch & propagate B.



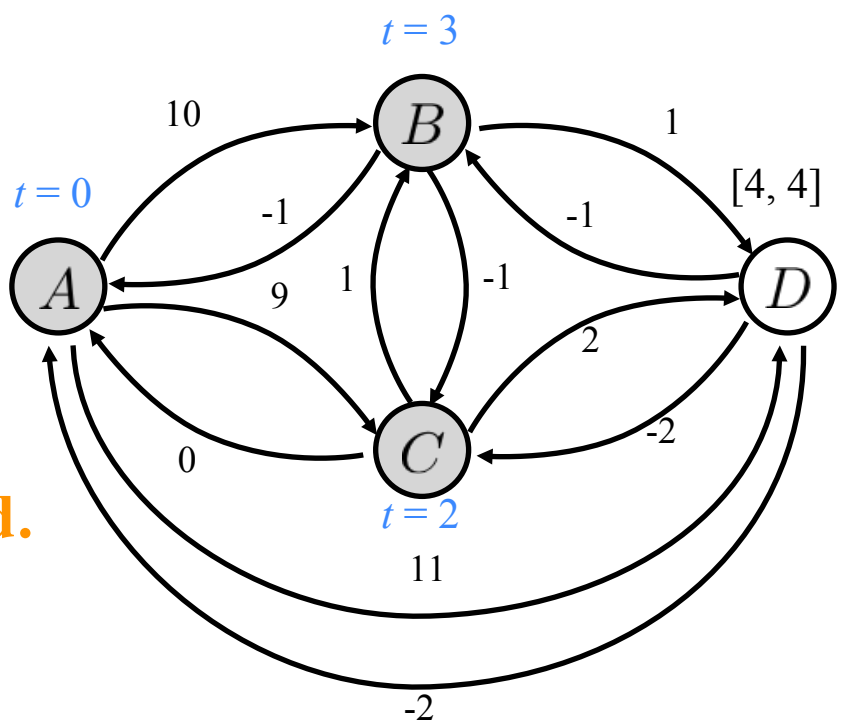
Running online scheduler

A, C < B **A, B, C < D**

E = {D}
S = {A, C, B}

Compute decomposable form (i.e., APSP)
Initialize execution windows to [-∞, ∞]
E ← {events with no predecessors}
S ← {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events

D is finally enabled.



Running online scheduler

$A, C < B \qquad A, B, C < D$

$E = \{\}$

$S = \{A, C, B, D\}$

Compute decomposable form (i.e., APSP)

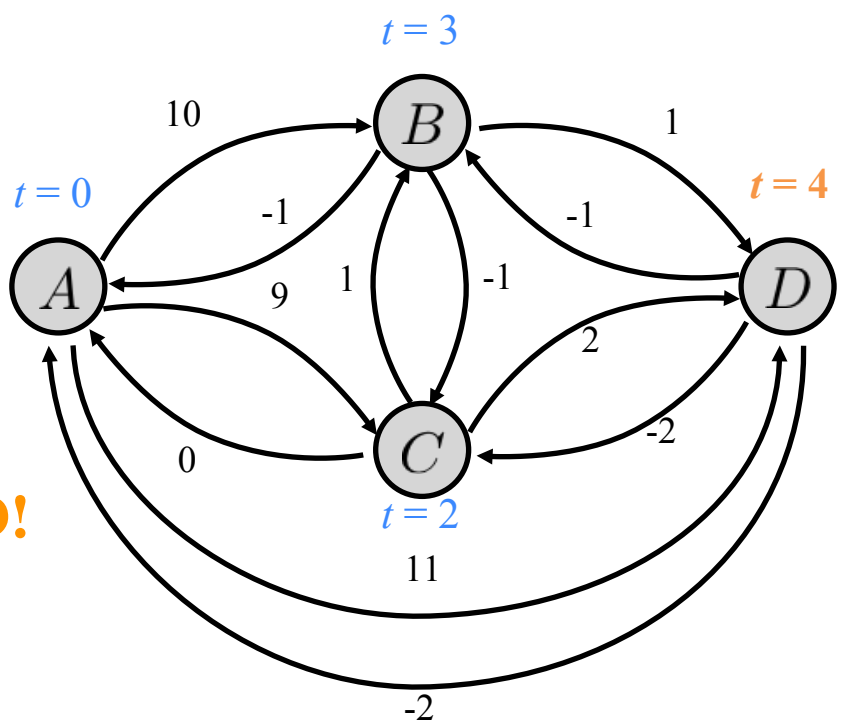
Initialize execution windows to [-∞, ∞]

$E \leftarrow$ {events with no predecessors}

$S \leftarrow$ {}

while unexecuted events:

- **Wait until some event x_i in E is active**
- **t_i = now**
- **Propagate to x_i's neighbors**
- **Add x_i to S**
- Add to E any now-enabled events



Finish up by dispatching D!

Online scheduling algorithm remarks

- By considering predecessors, we guarantee that events are assigned monotonically increasing times.

- Online scheduler responds to fluctuations that do not violate feasibility wrt events already scheduled.

(**Note:** must be run on a decomposable / APSP graph!)

Online scheduler efficiency

- Consider an STN with n edges.
  - How many edges in APSP distance graph?

Compute decomposable form (i.e., APSP)
Initialize execution windows to $[-\infty, \infty]$
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events

Online scheduler efficiency

- Consider an STN with n edges.
  - How many edges in APSP distance graph? n^2.

Compute decomposable form (i.e., APSP)
Initialize execution windows to $[-\infty, \infty]$
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events

Online scheduler efficiency

- Consider an STN with n events.
  - How many edges in APSP distance graph? n^2.
  - How many neighbors to propagate to during each step?

Compute decomposable form (i.e., APSP)
Initialize execution windows to $[-\infty, \infty]$
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events

Online scheduler efficiency

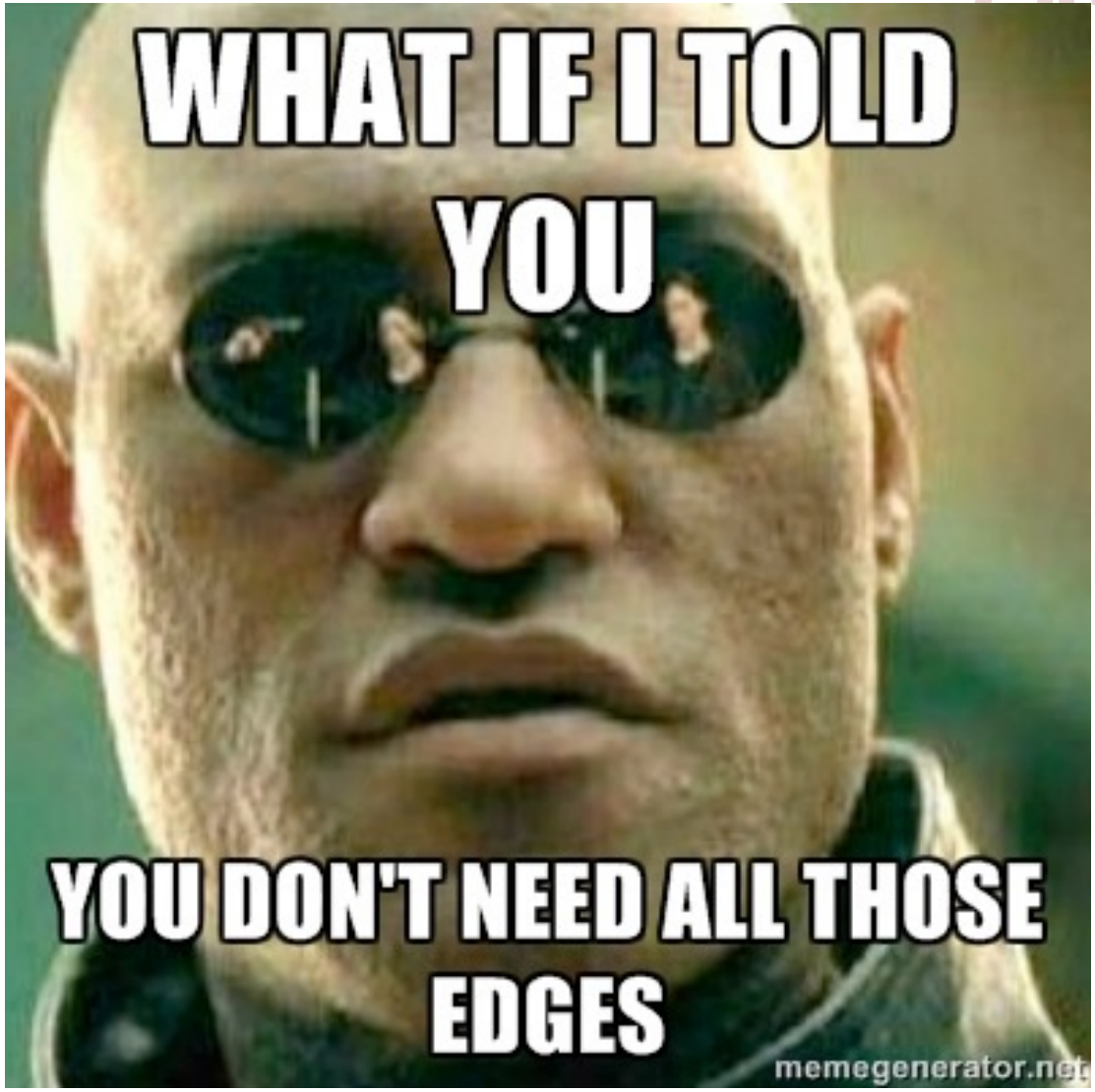
- Consider an STN with n events.
  - How many edges in APSP distance graph? n^2.
  - How many neighbors to propagate to during each step? n.

Compute decomposable form (i.e., APSP)
Initialize execution windows to $[-\infty, \infty]$
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events

Online scheduler efficiency

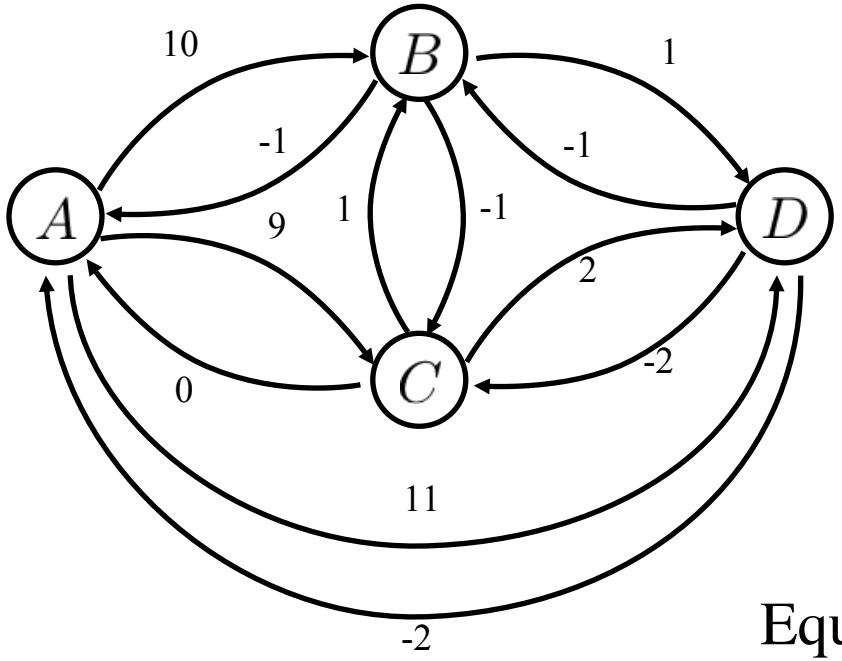
- Consider an STN with n events.
  - How many edges in APSP distance graph? n^2
  - How many neighbors to propagate to during each step? n
  - **For large STNs – network large, propagation is slow. Want to reduce.**

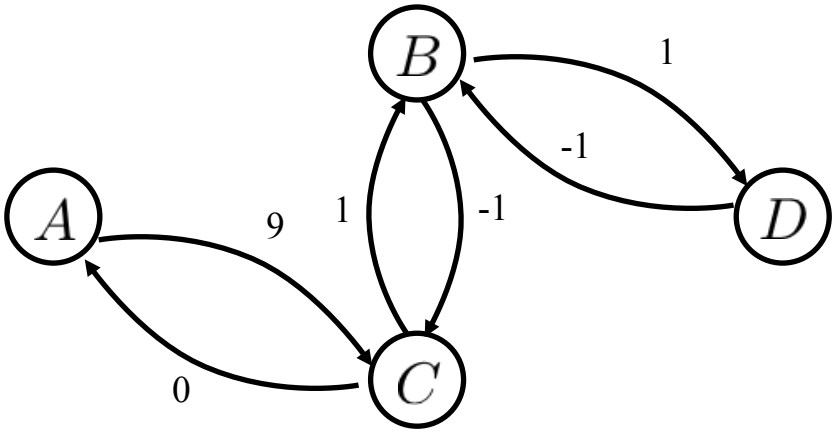
Compute decomposable form (i.e., APSP)
Initialize execution windows to $[-\infty, \infty]$
$E \leftarrow$ {events with no predecessors}
$S \leftarrow$ {}
while unexecuted events:
    Wait until some event x_i in E is active
    t_i = now
    Propagate to x_i's neighbors
    Add x_i to S
    Add to E any now-enabled events



See Appendix!

You don’t need all those edges!





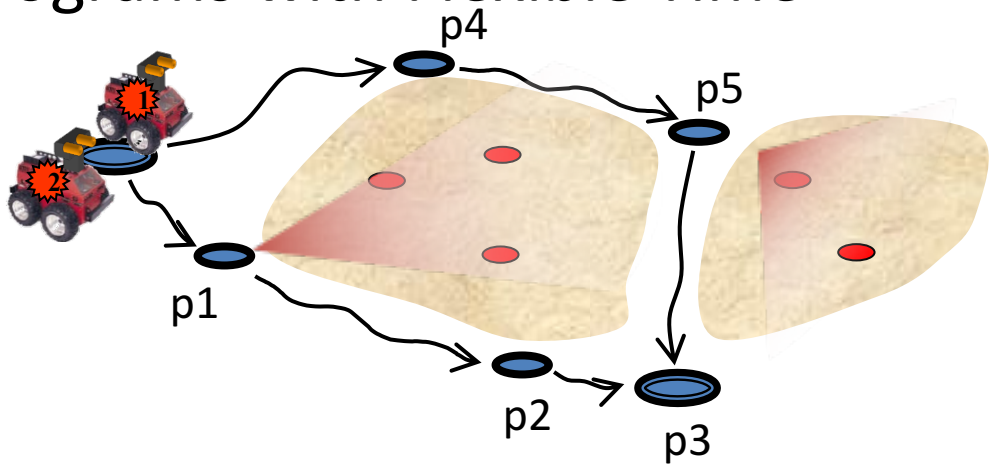
Equivalent ***minimal dispatchable network***

Summary of Online Scheduling

- To schedule online,
  times must **monotonically increase**
  → use enablement conditions.
- Scheduling online allows greater **robustness**
  to fluctuations.

- Propagation costs can be large for large graphs.
- Reduce by using domination to trim edges
  (see Appendix).

To Improve Robustness: Dynamically Executing Programs with Flexible Time




```
Program A ::=
    prim_action(args)    |
    [lb, ub] A   |
    {Sequence A1; A2; ... } |
    {Parallel A1; A2; ... }
```

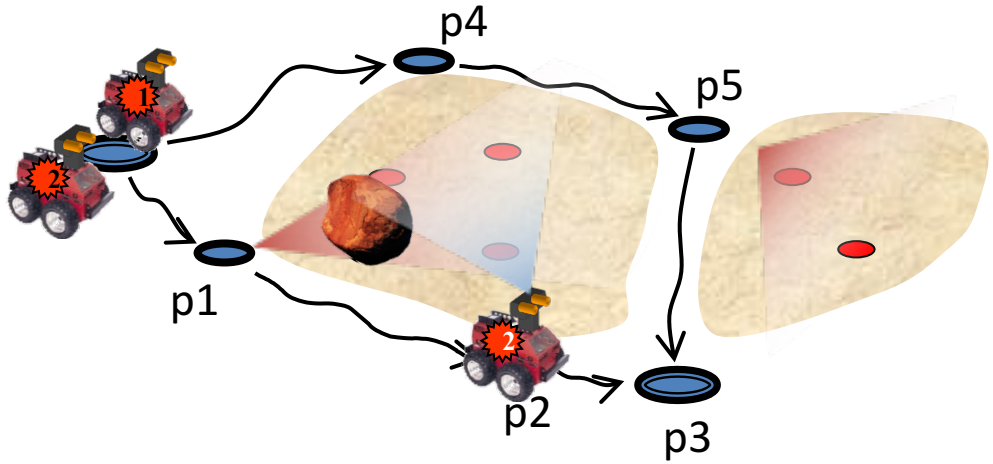
RMPL Syntax

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
     00:00 Rover1.goto(p5);
     00:30 Rover1.imageTargets();
     00:00 Rover1.goto(p3);
  },
  {Sequence
     00:20 Rover2.goto(p1);
     00:50 Rover2.imageTargets();
     00:00 Rover2.goto(p3);
  }
}
```

→

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
    [5,10] Rover1.goto(p5);
    [2,5]   Rover1.imageTargets();
    [5,10] Rover1.goto(p3);
  },
  {Sequence
    [5,10] Rover2.goto(p1);
    [5,10] Rover2.imageTargets();
    [5,10] Rover2.goto(p3);
  }
}
```

We Expand Flexibility via Programs with Added Choices: Non-deterministic and Decision-theoretic Programs




```
Program A ::=
    prim_action(args)    |
    [lb, ub] A   |
    {Sequence A1; A2; ... } |
    {Parallel A1; A2; ... }     |
    {Choose [with reward R1] A1;
            [with reward R2] A2; . . .}
```

RMPL Syntax

```
imageScienceTargets(Rover1, Rover2)
{Parallel
  {Sequence
    [5,10] Rover1.goto(p5);
    [2,5]   Rover1.imageTargets();
    [5,10] Rover1.goto(p3);
  },
  {Sequence
    [5,10] Rover2.goto(p1);
    [5,10] Rover2.imageTargets();
    [5,10] Rover2.goto(p3);
  }
}
```

→

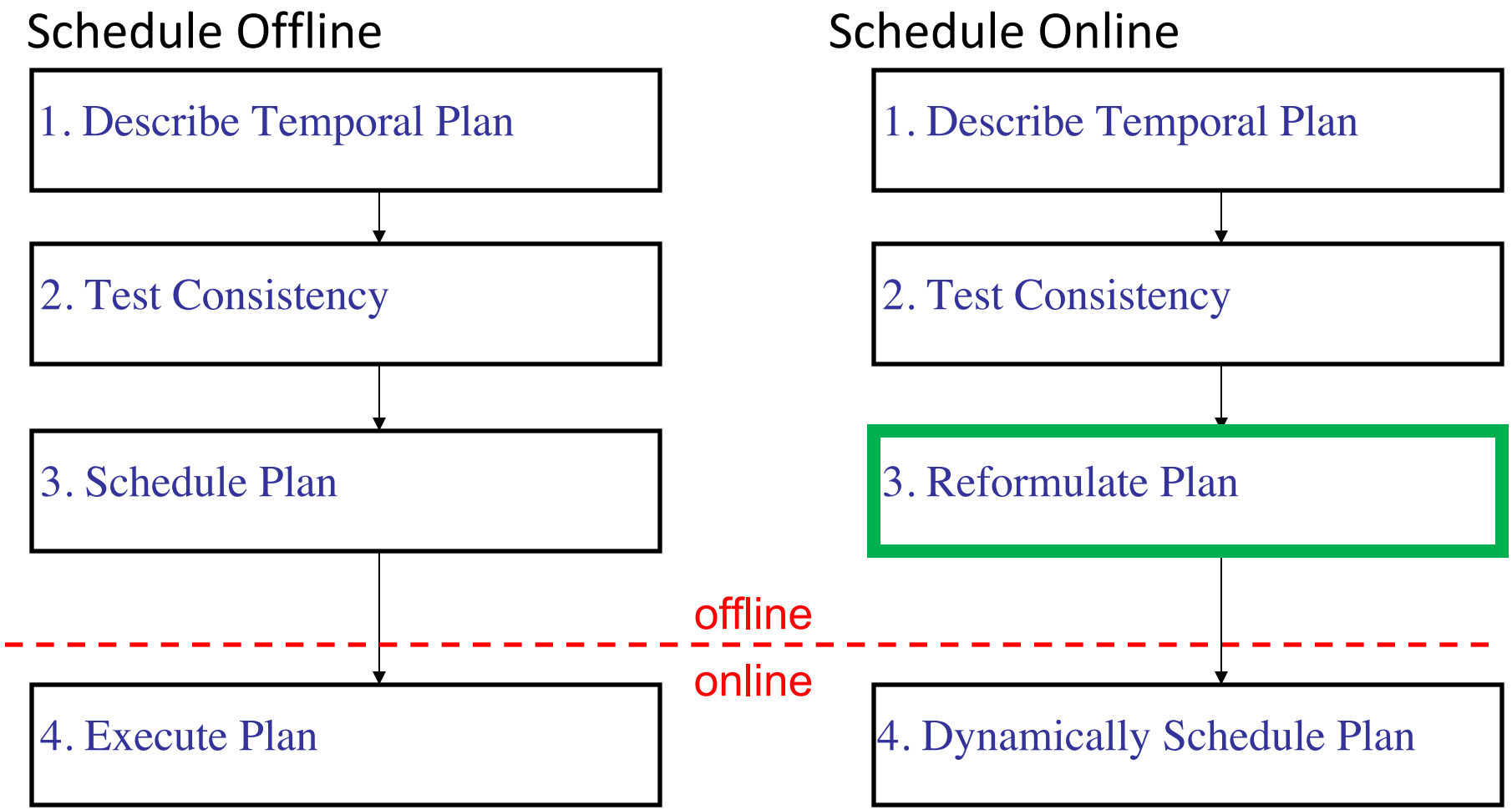
```
Choose{
    [with reward 15]
    [5,10] Rover2.imageTargets();

    [with reward 10]
    {Sequence
        [5,10] Rover2.goto(p2);
        [5,10] Rover2.imageTargets();
    };
}
```

What if we want more flexibility in our program?

Minimal Dispatchable Form and Summary

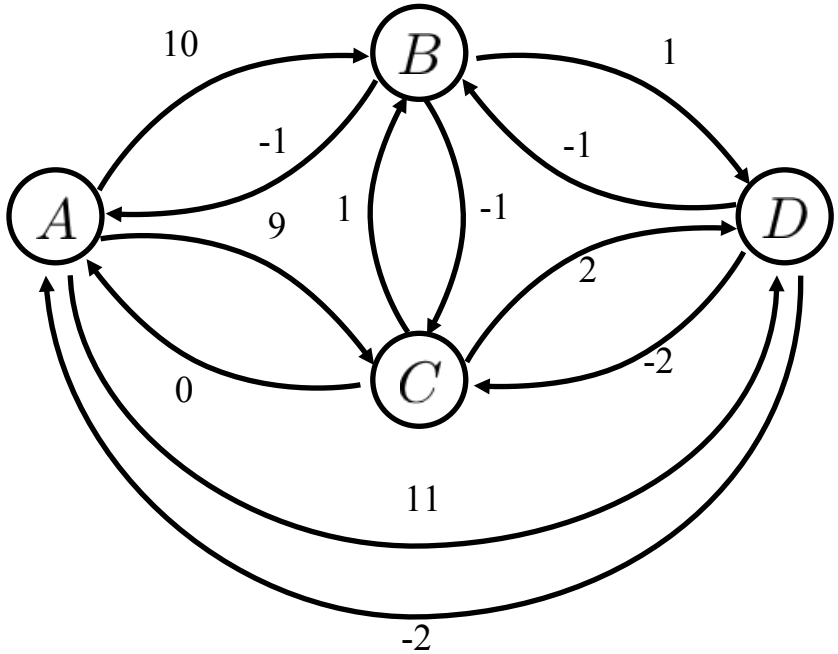
APPENDIX

To Execute a Temporal Plan



You don't need all those edges!

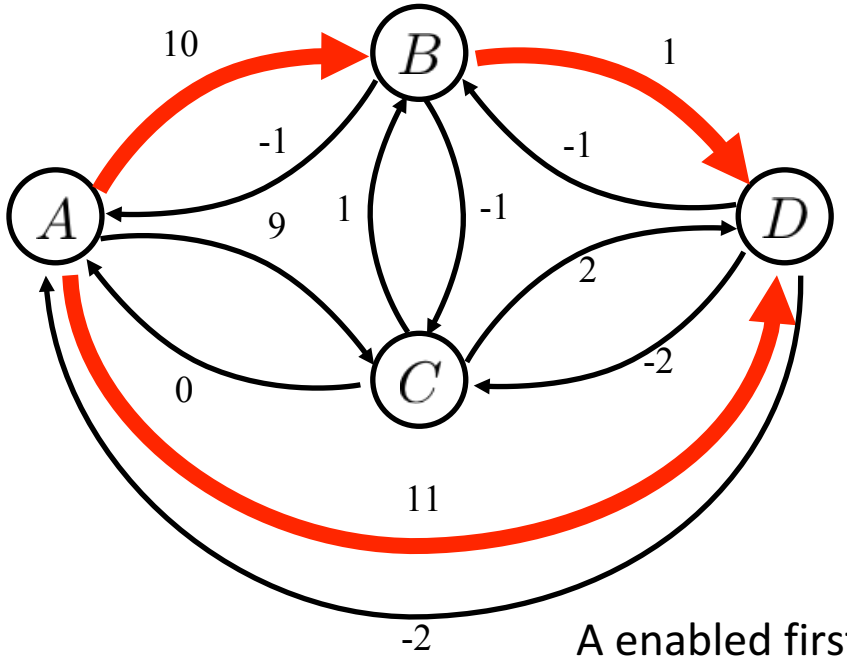
Let's consider
a specific triangle of edges.



You don't need all those edges!

Let's consider
a specific triangle of edges.

Do we really need
the bottom edge?



BD: $D < B + 1$

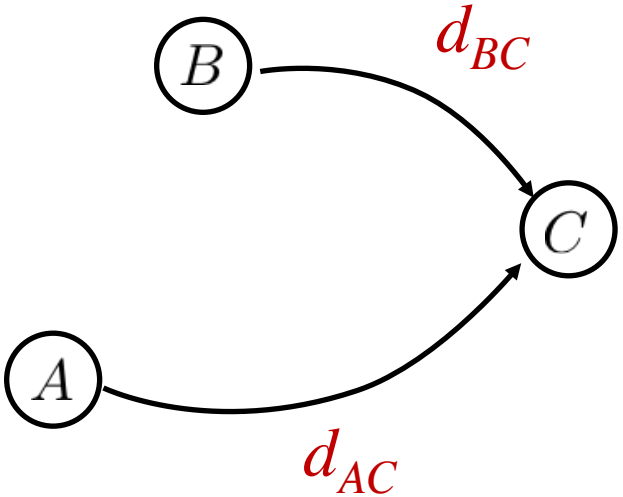
AB: $B < A + 10$

AD: $D < A + 11$

<u>A enabled first:</u> AB enforced by propagation
<u>B enabled second:</u> BD enforced by propagation
→ AB, BD implies AD by triangle inequality

<u>Can delete AD</u>

Upper dominating edges
- trim after APSP



If $d_{BC}, d_{AC} \geq 0$ and

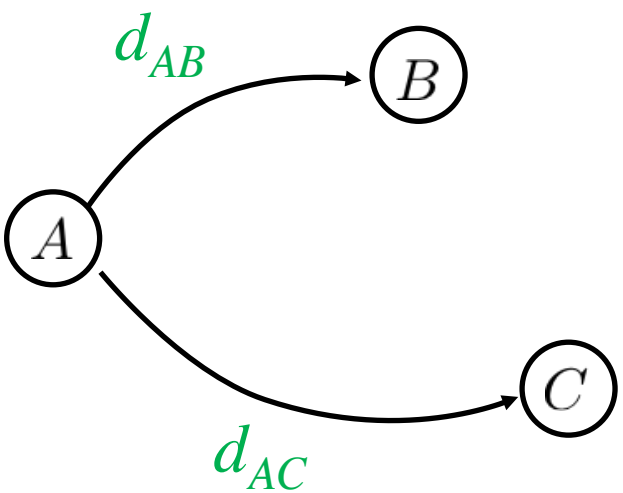
$d_{AB} + d_{BC} = d_{AC}$

then BC ***dominates*** AC

Remove AC

(Proof omitted - based on triangle rule property of APSP. Please see notes / reading for more info)

Lower dominating edges
- trim after APSP



If $d_{AB}, d_{AC} < 0$ and

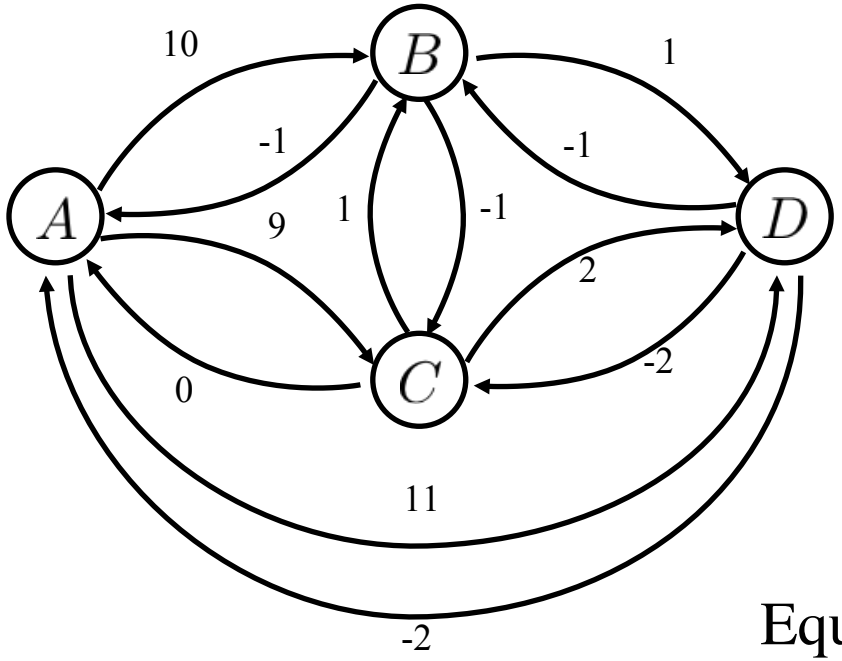
$d_{AB} + d_{BC} = d_{AC}$

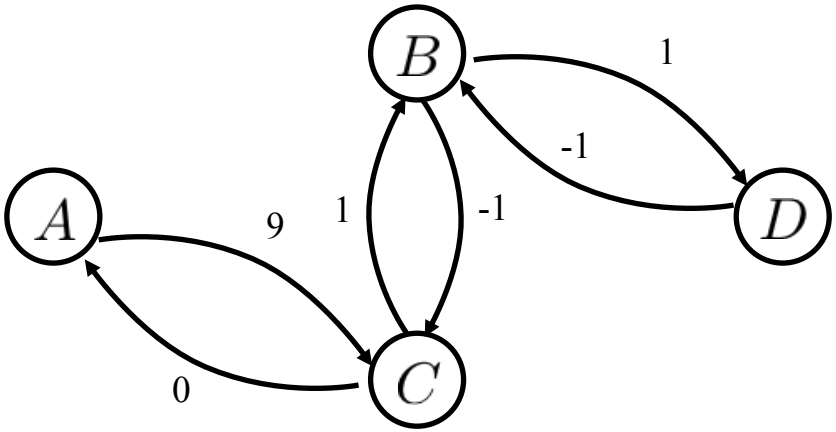
then AB ***dominates*** AC

Remove AC

(Proof omitted - based on triangle rule property of APSP. Please see notes / reading for more info)

You don’t need all those edges!





Equivalent ***minimal dispatchable network***

To Execute a Temporal Plan

Schedule Offline

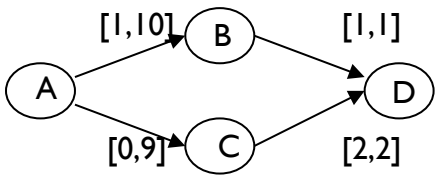
1. Describe Temporal Plan
2. Test Consistency
3. Schedule Plan

offline

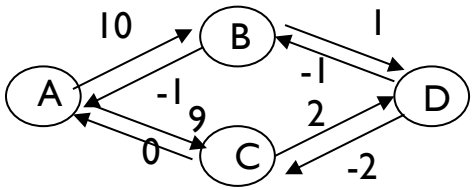
online

4. Execute Plan

STN



D Graph



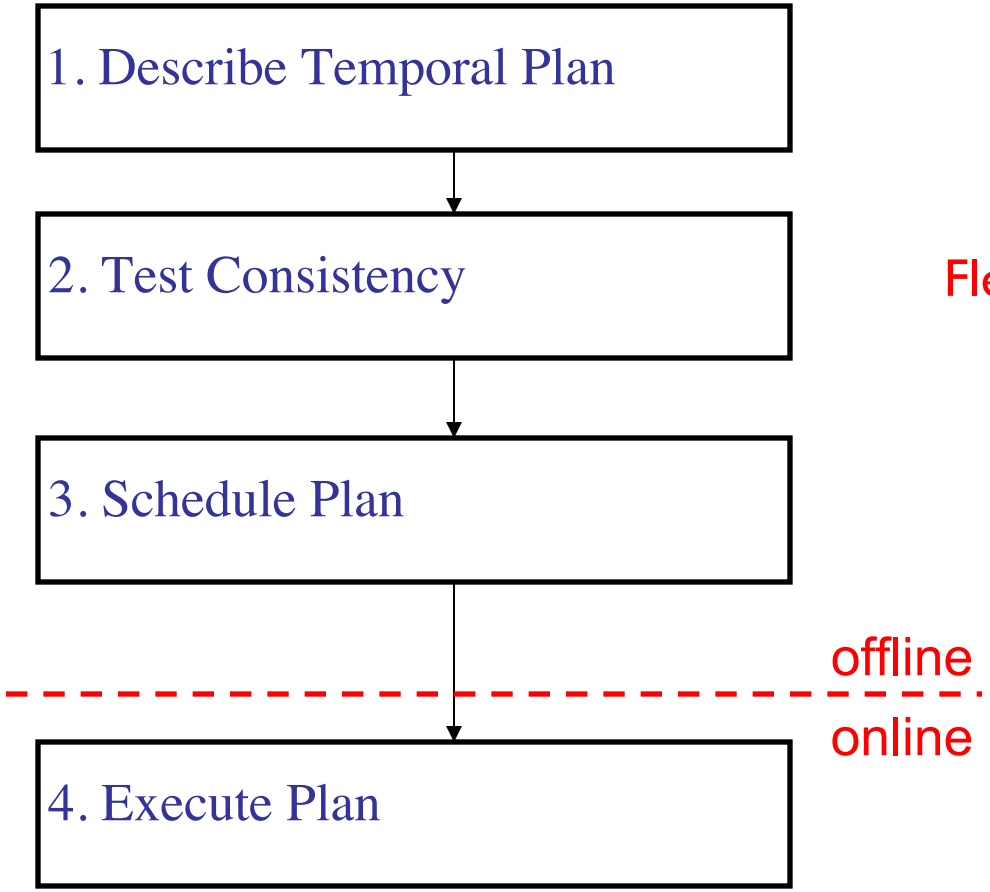
Detect negative loops (SSSP).

APSP + Decomposition.

[Dechter, Meiri, Pearl 91]

To Execute a Temporal Plan

Schedule Offline



Problem: delays and fluctuations in task duration can cause plan failure.

Observation: temporal constraints leave room to adapt.

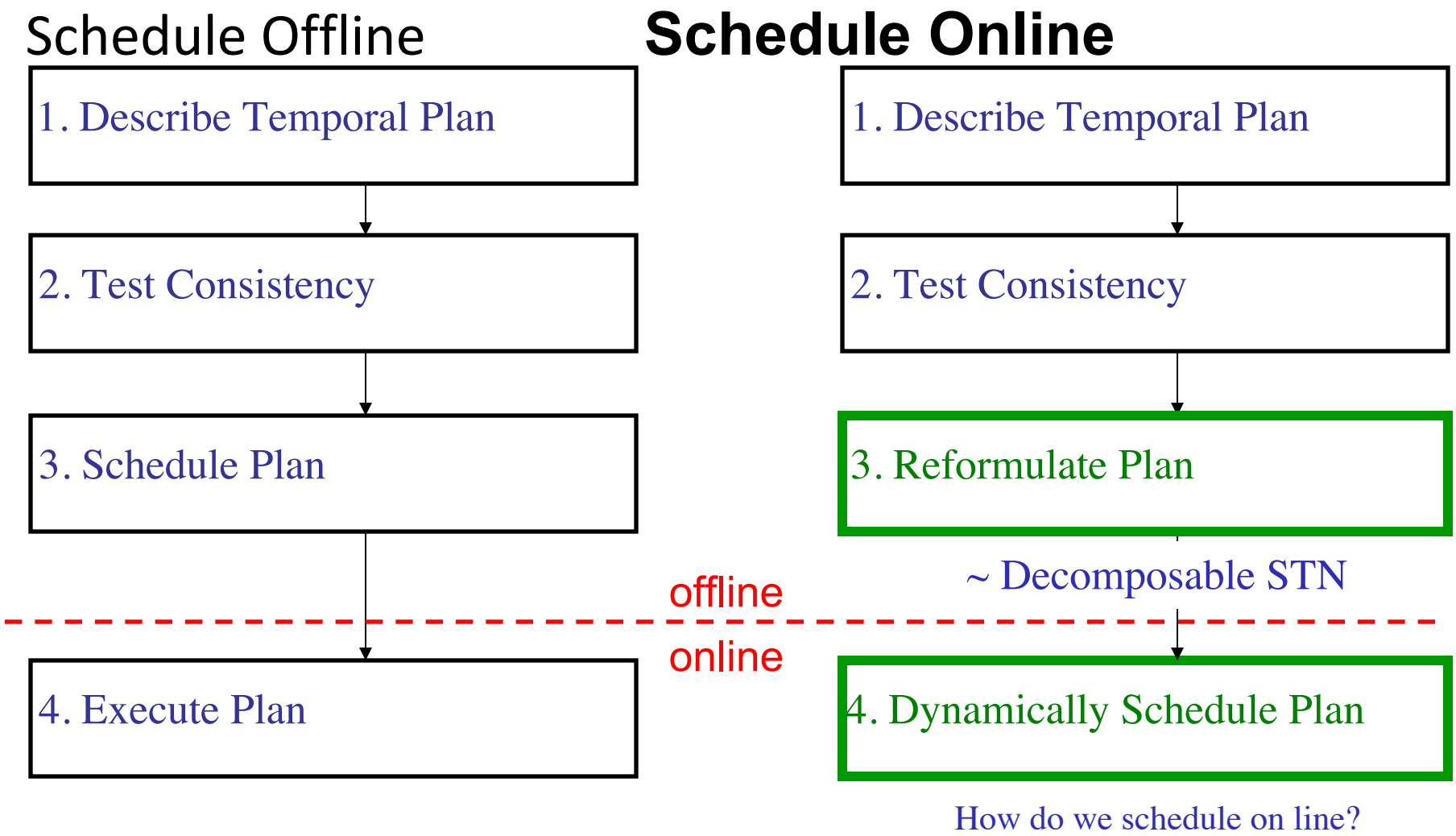
Flexible Execution adapts through dynamic scheduling:

Assign time to event when executed.

- Guarantee that all constraints will be satisfied.
- Schedule with low latency through pre-compilation.

[Muscettola, Morris, Tsmardinos KR98]

To Execute a Temporal Plan



To Execute a Temporal Plan

Schedule Online

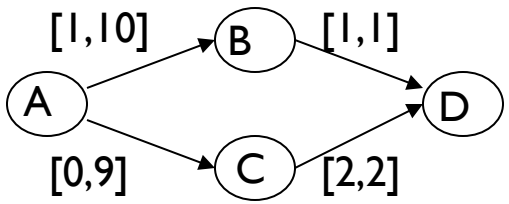
1. Describe Temporal Plan
2. Test Consistency
3. Reformulate Plan

offline

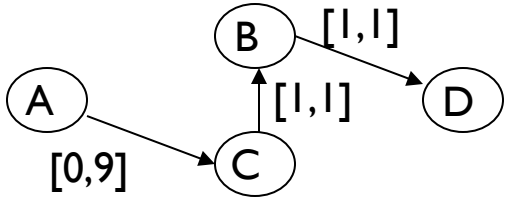
online

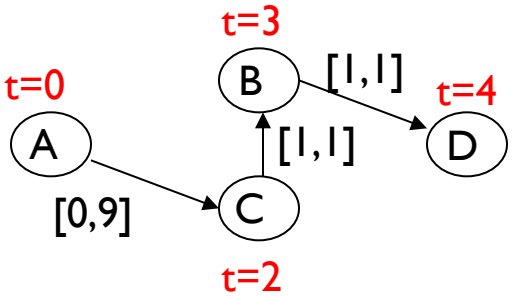
4. Dynamically Execute Plan

[Muscettola, Morris, Tsamardinos KR98]



To minimize latency remove redundant edges.





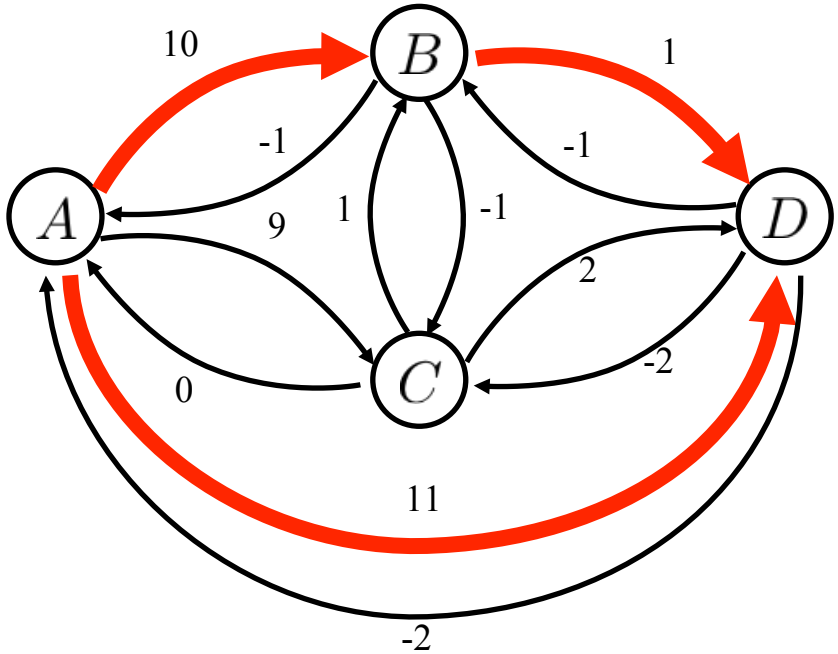
Details of edge trimming
based on dominance

APPENDIX CONTINUED

You don't need all those edges!

Let's consider a specific triangle of edges.

Do we really need the bottom edge?



BD: D < B + 1

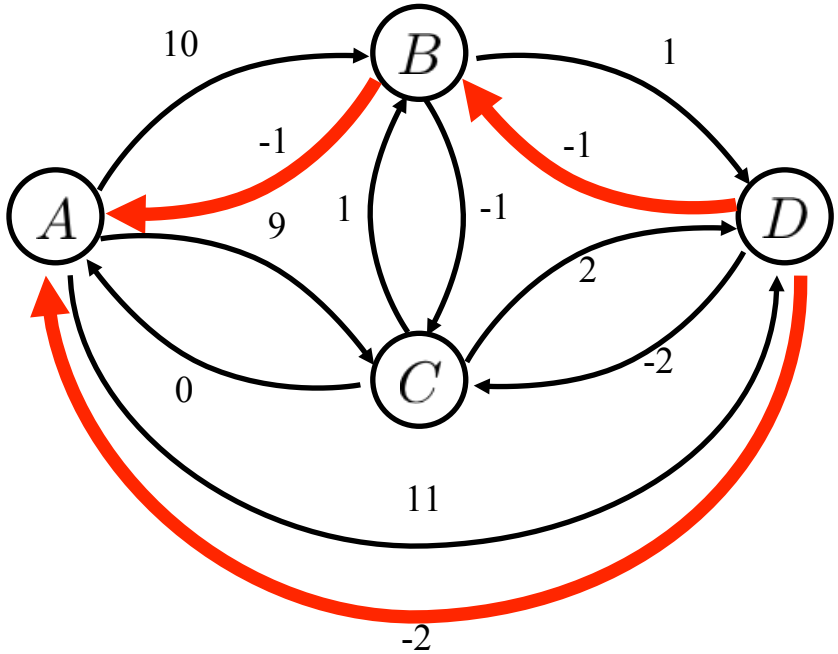
AB: B < A + 10

AD: D < A + 11

You don't need all those edges!

Let's consider a different triangle of edges.

Do we really need the bottom edge?

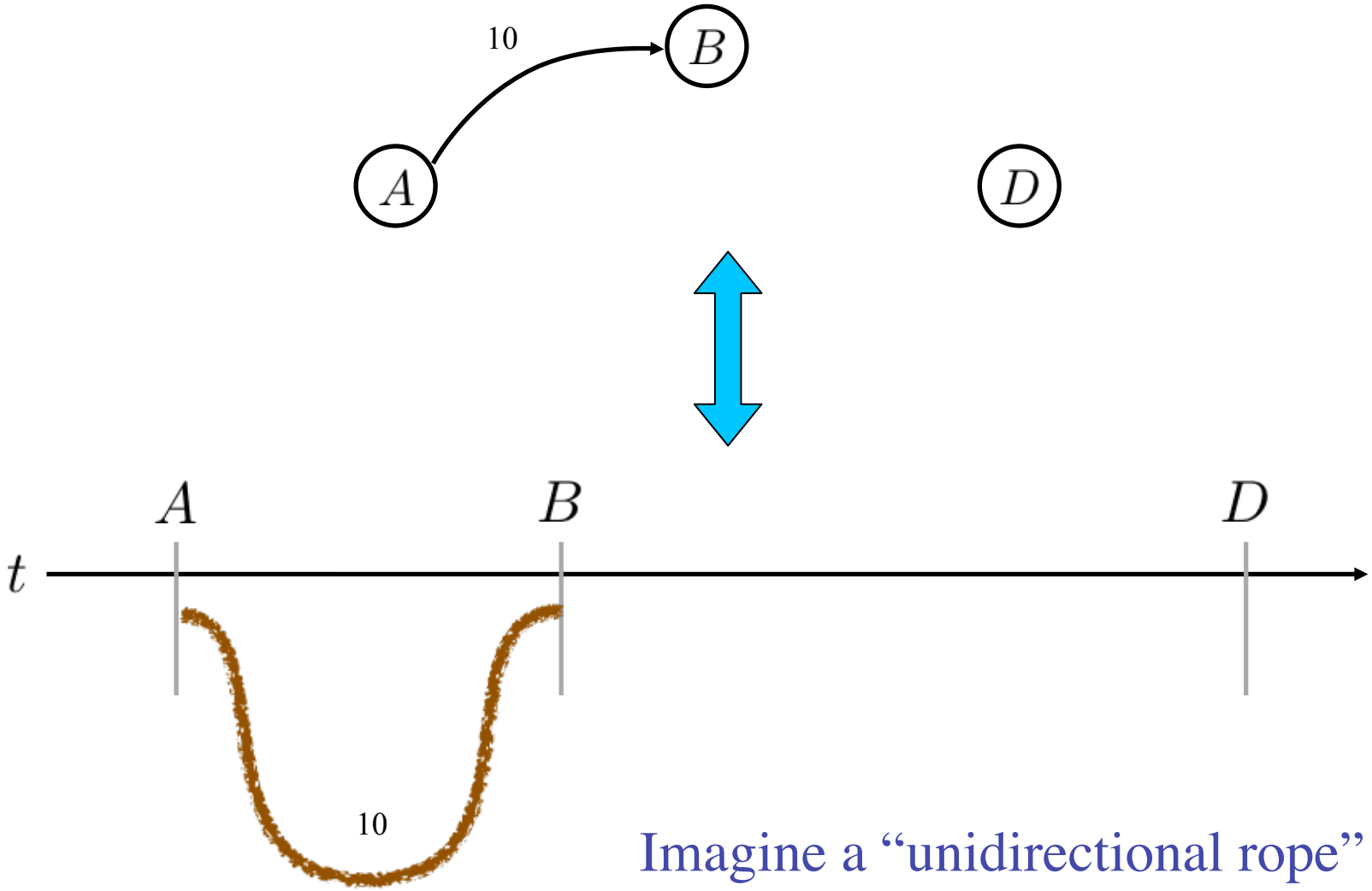


DB: D > B + 1

BA: B > A + 1

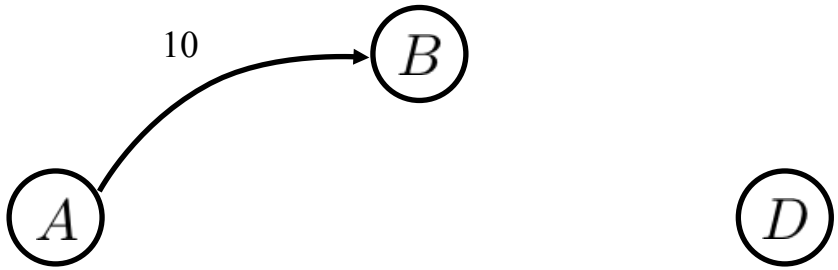
DA: D > A + 2

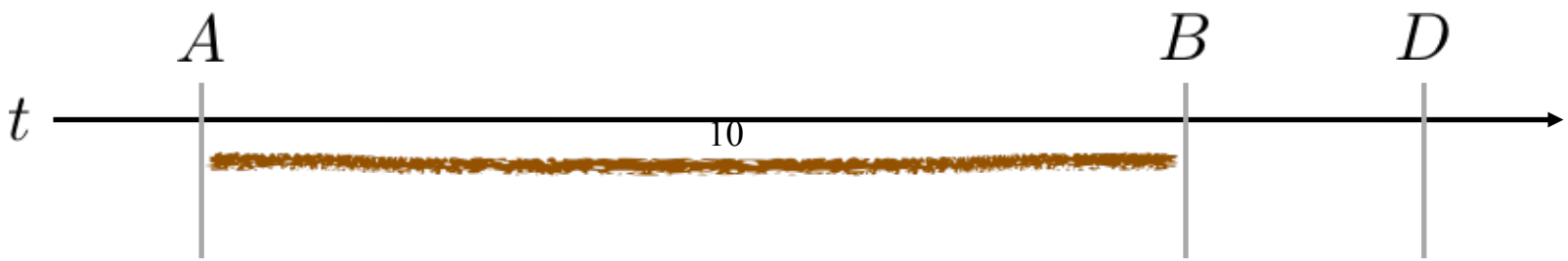
Rope analogy



Imagine a "unidirectional rope" of length 10 constraining sliders on a track.

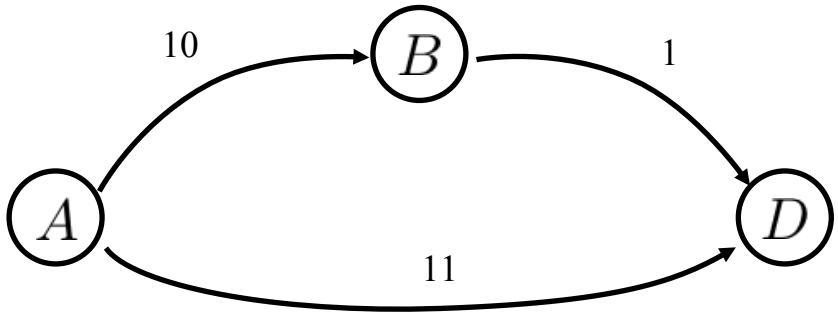
Rope analogy

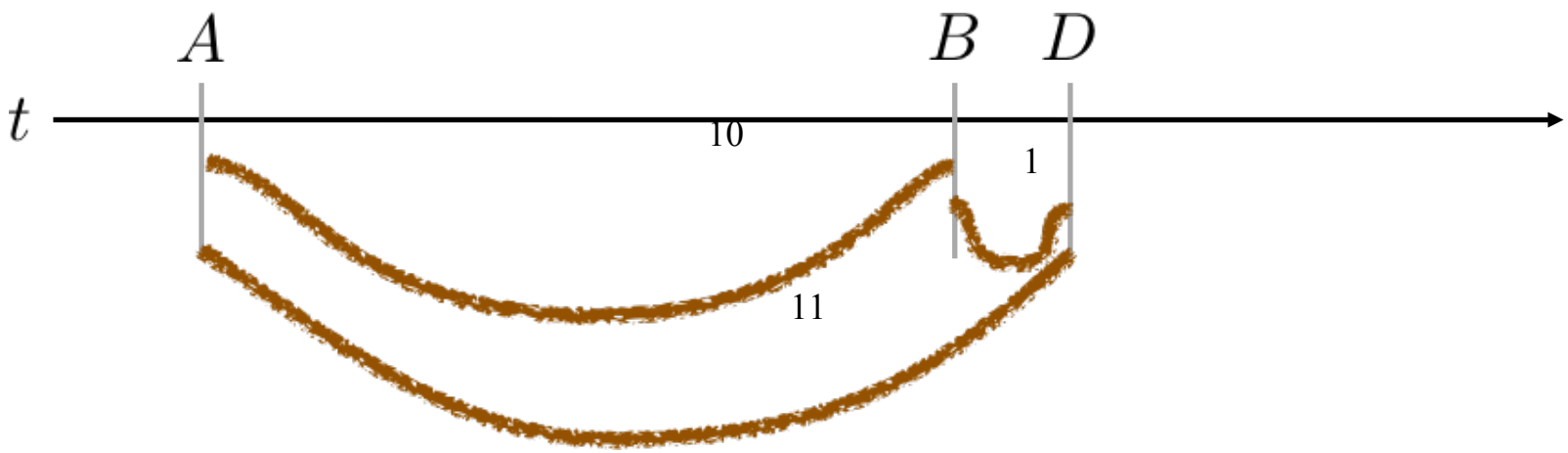




Imagine a "unidirectional rope" of length 10 constraining sliders on a track.

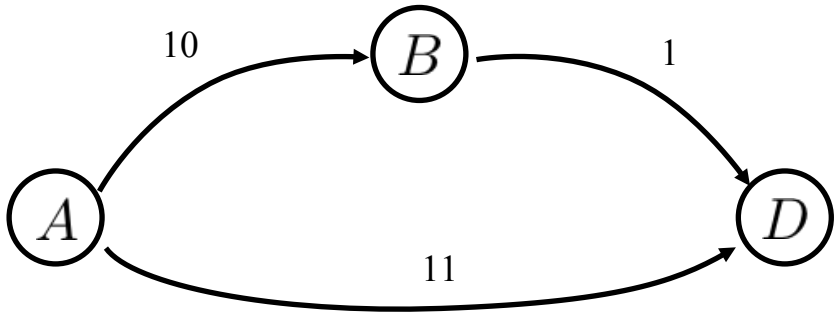
Rope analogy



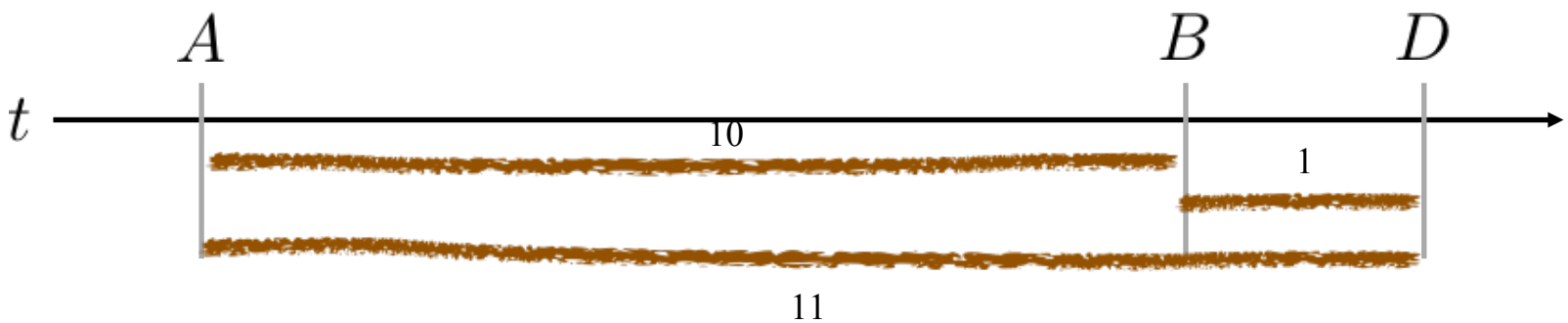


Now add in ropes for other constraints

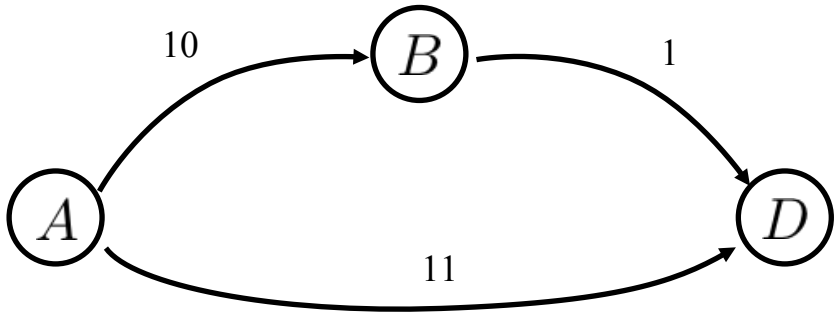
Rope analogy



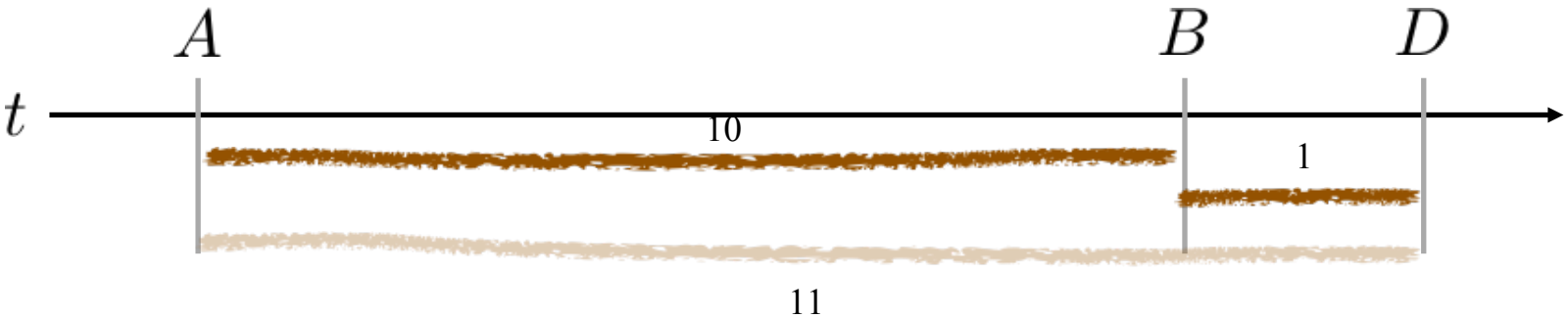
Imagine pulling A and D as tight as possible.



Rope analogy

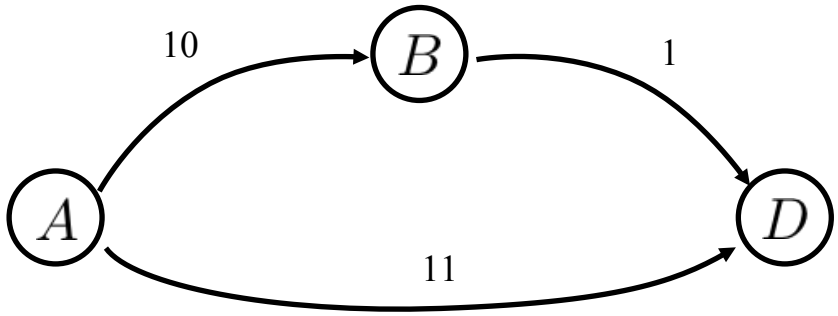


Can we remove rope AD
without changing behavior?

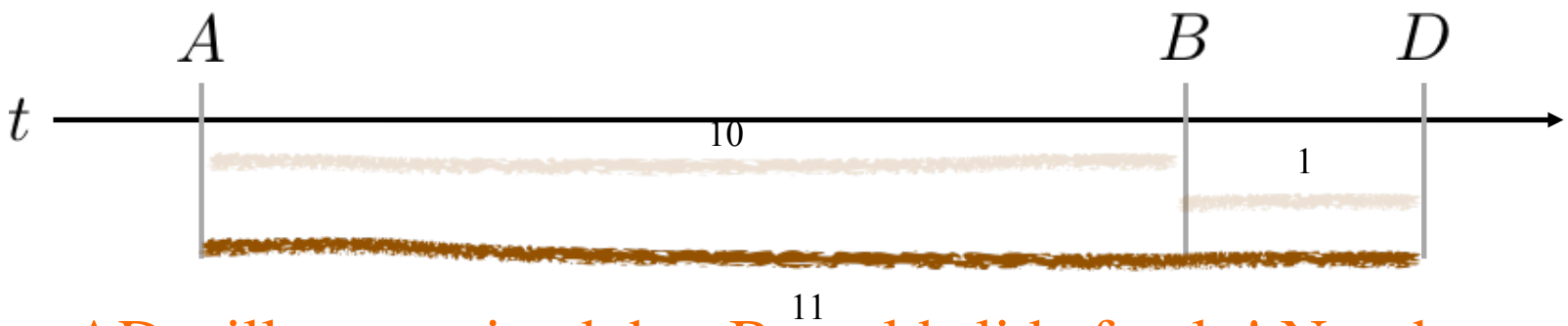


Yes! Same possible positions for A, B, D.

Rope analogy



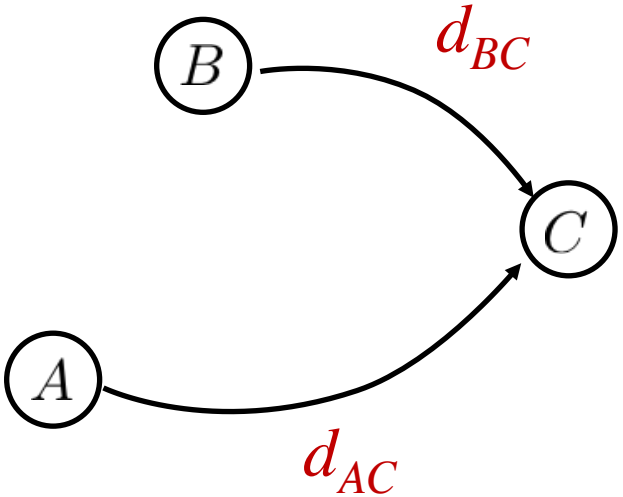
Can we remove ropes AB or BD without changing behavior?



No. AD still constrained, but B could slide freely! Not the same.
Collectively, AB and BD entails AD.

Not AD and BD entails BD . . .

Upper dominating edges
- trim after APSP



If $d_{BC}, d_{AC} \geq 0$ and

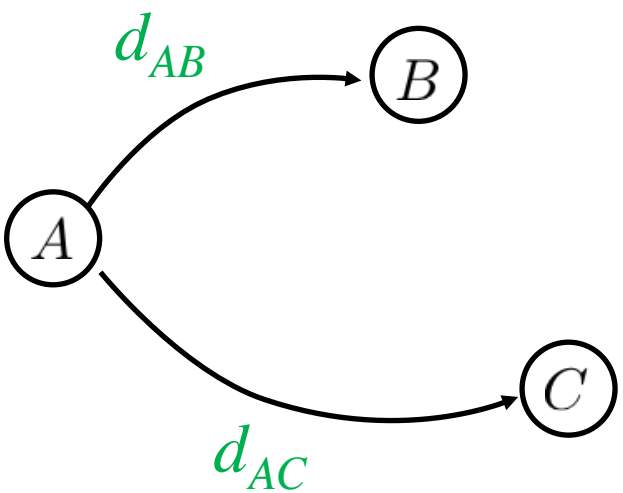
$d_{AB} + d_{BC} = d_{AC}$

then BC ***dominates*** AC

Remove AC

(Proof omitted - based on triangle rule property of APSP. Please see notes / reading for more info)

Lower dominating edges
- trim after APSP



If $d_{AB}, d_{AC} < 0$ and

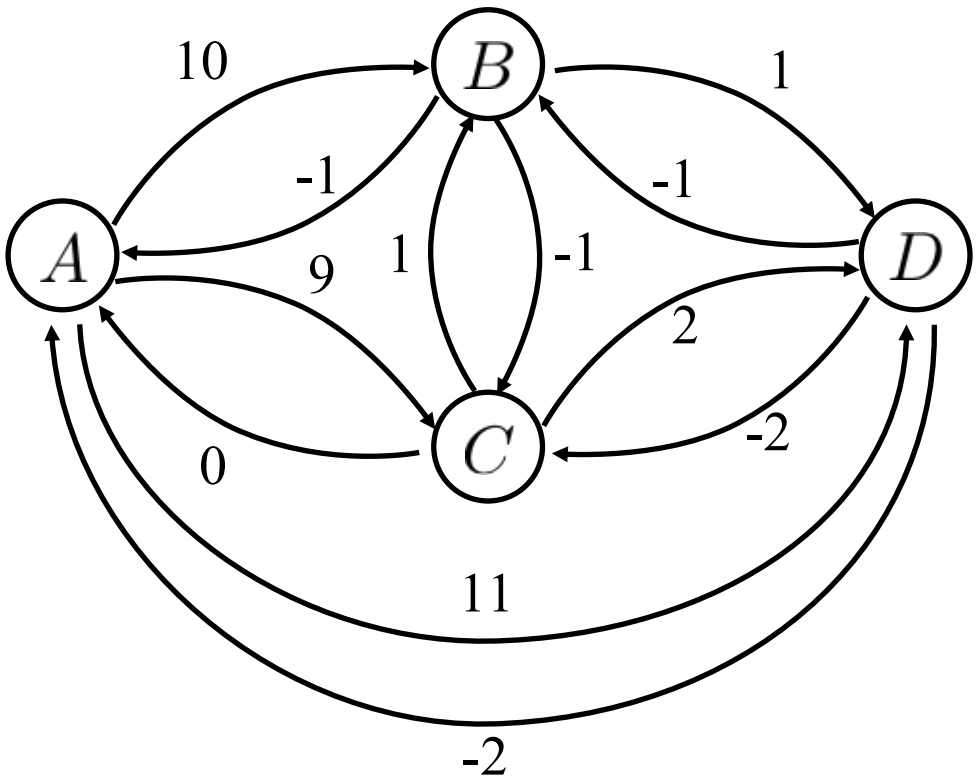
$d_{AB} + d_{BC} = d_{AC}$

then AB ***dominates*** AC

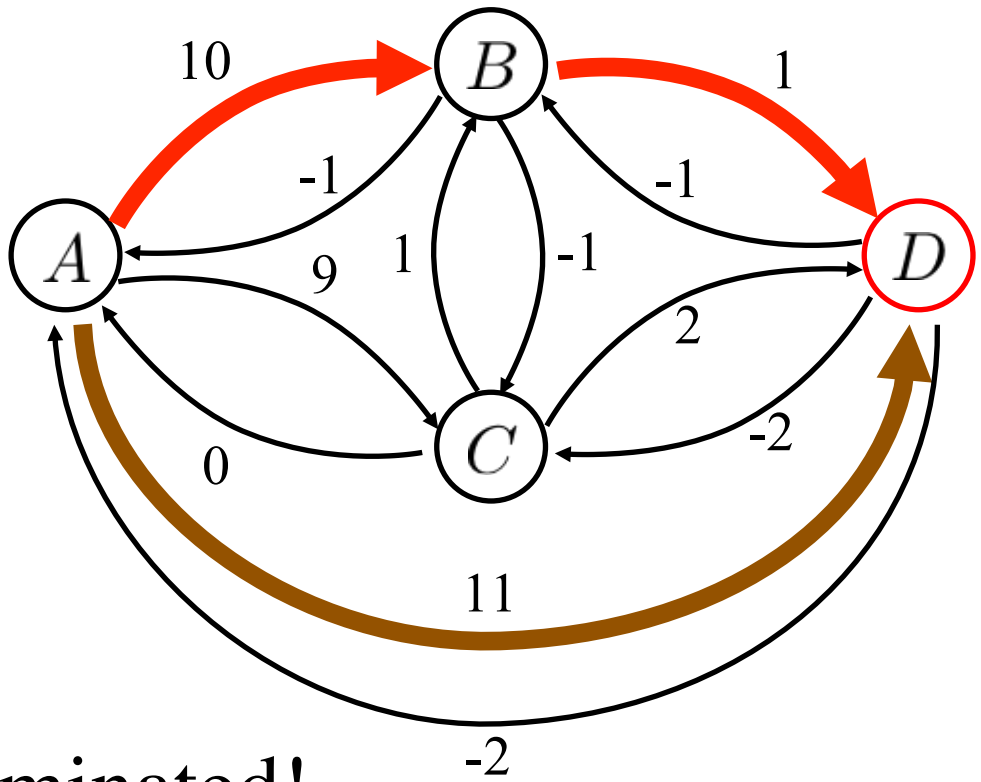
Remove AC

(Proof omitted - based on triangle rule property of APSP. Please see notes / reading for more info)

Dominance example

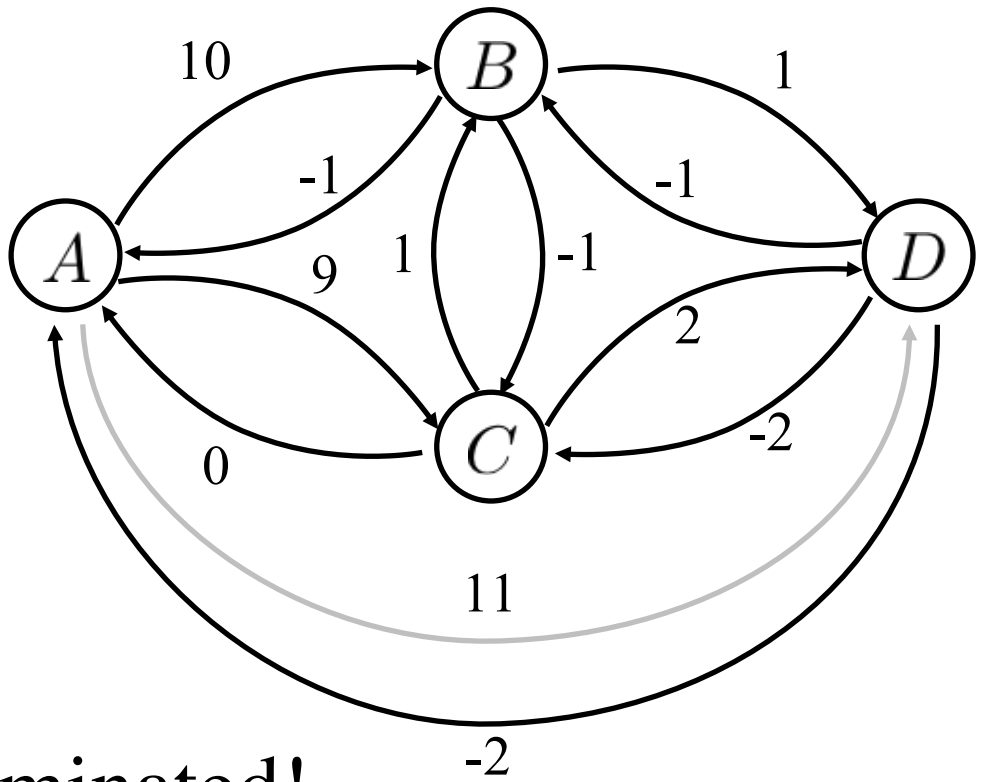


Dominance example



Upper dominated!

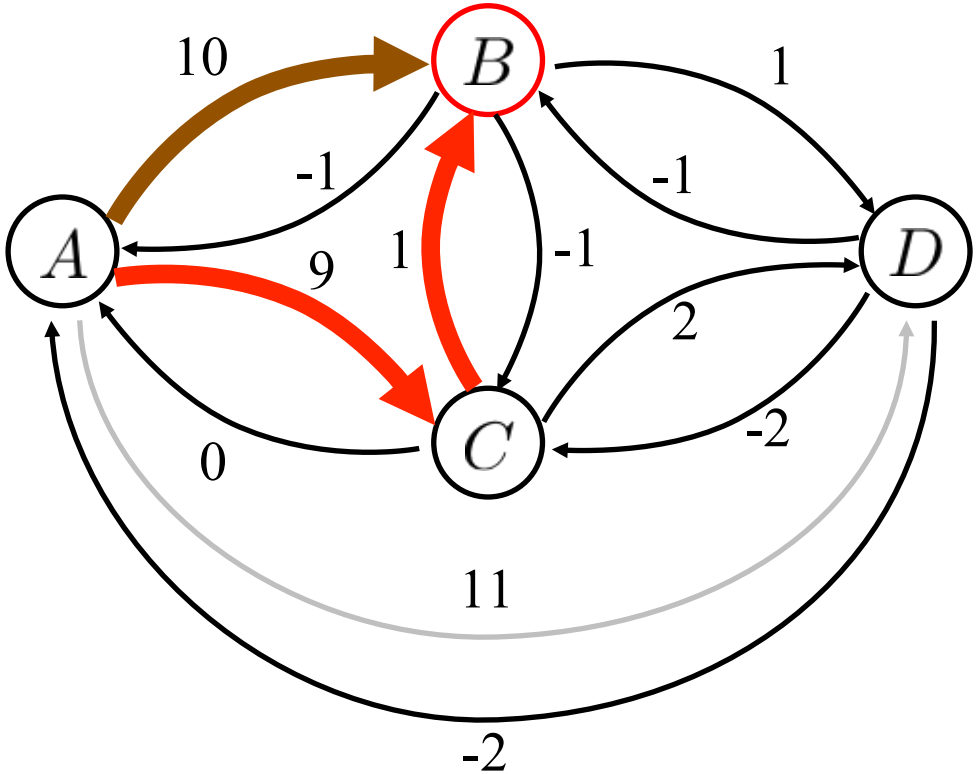
Dominance example



Upper dominated!

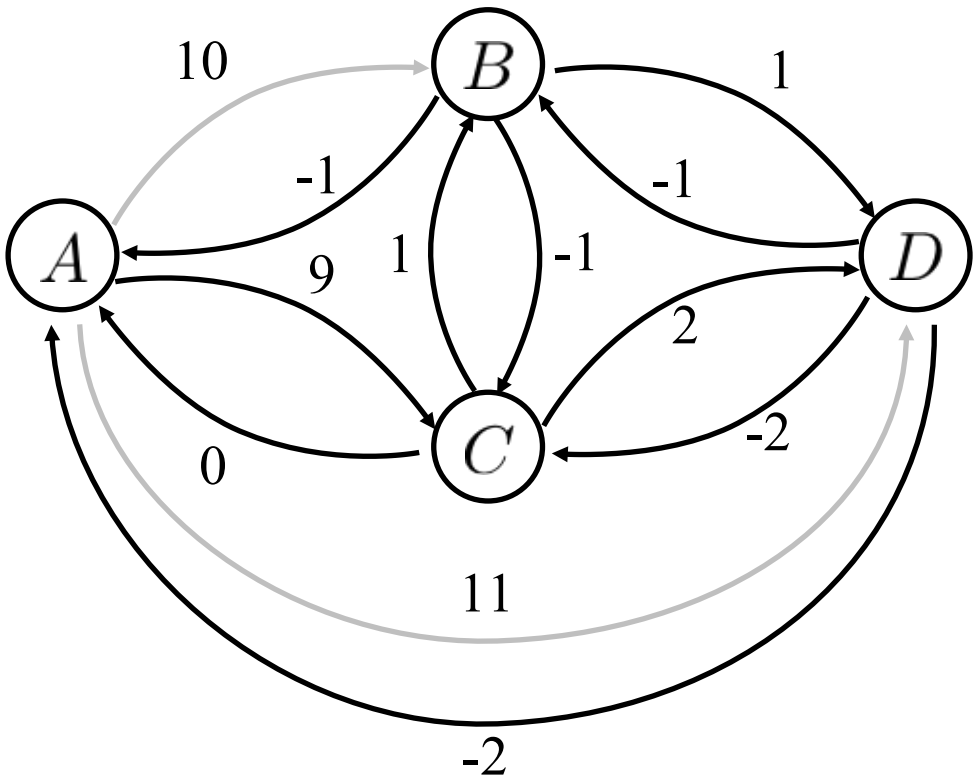
Dominance example

Upper dominated!

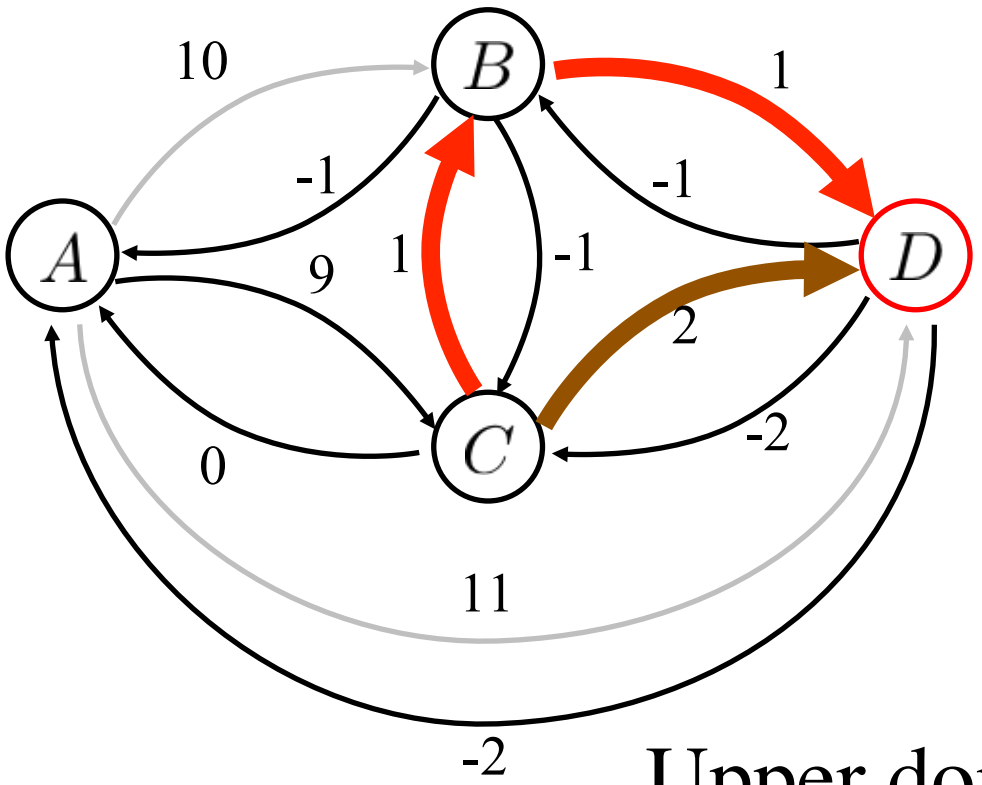


Dominance example

Upper dominated!

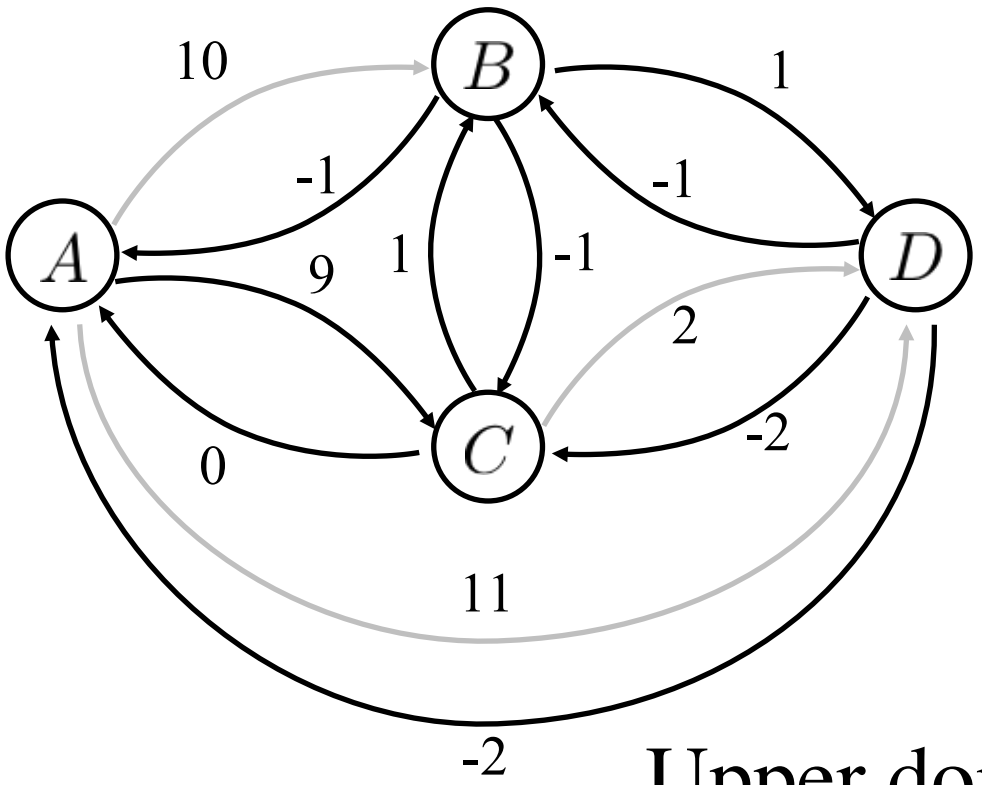


Dominance example



Upper dominated!

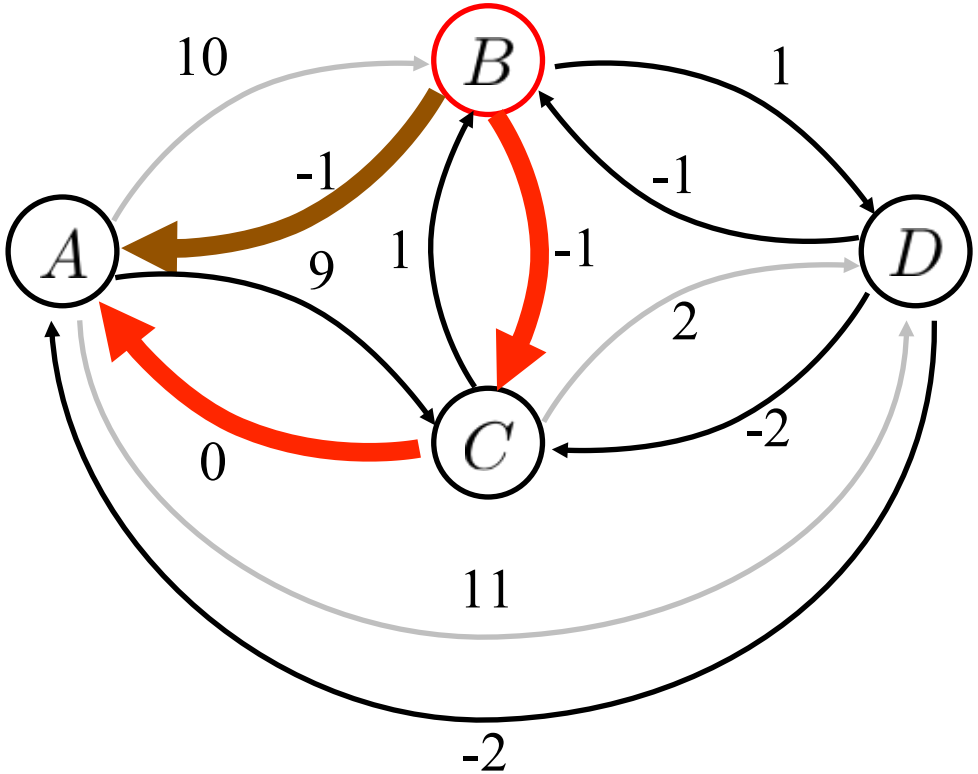
Dominance example



Upper dominated!

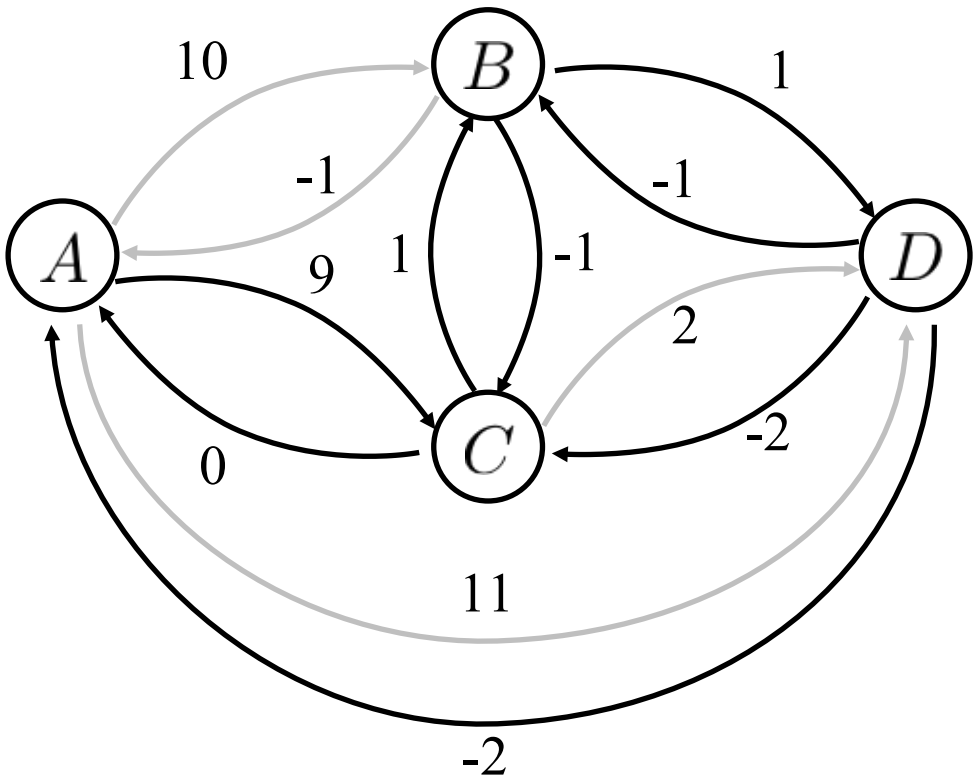
Dominance example

Lower dominated!

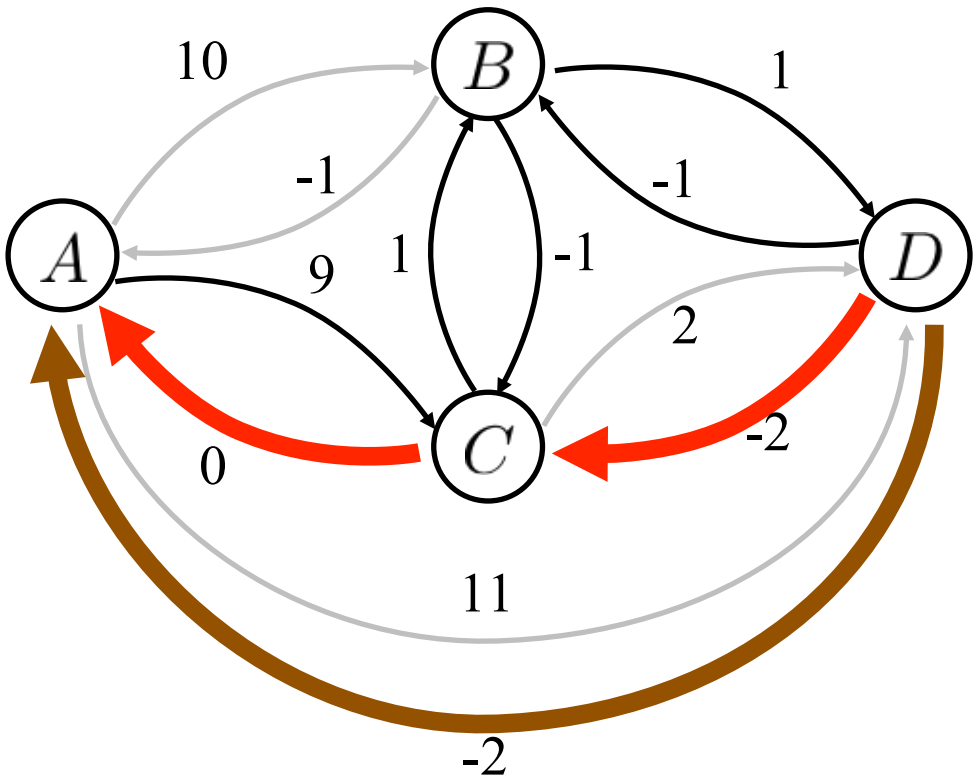


Dominance example

Lower dominated!

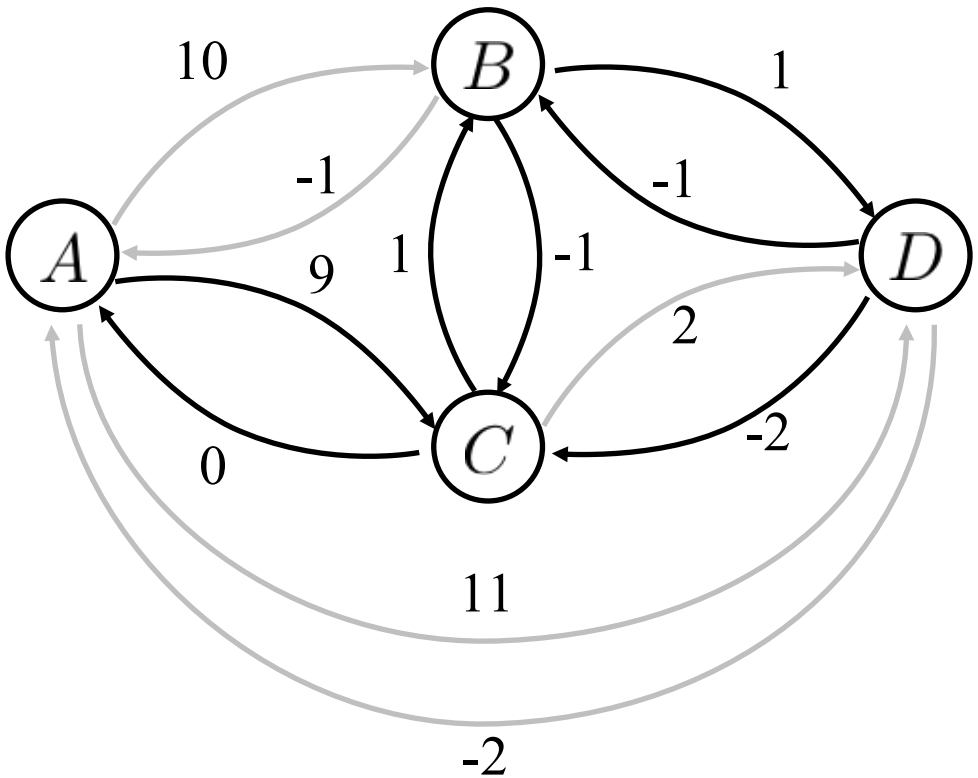


Dominance example



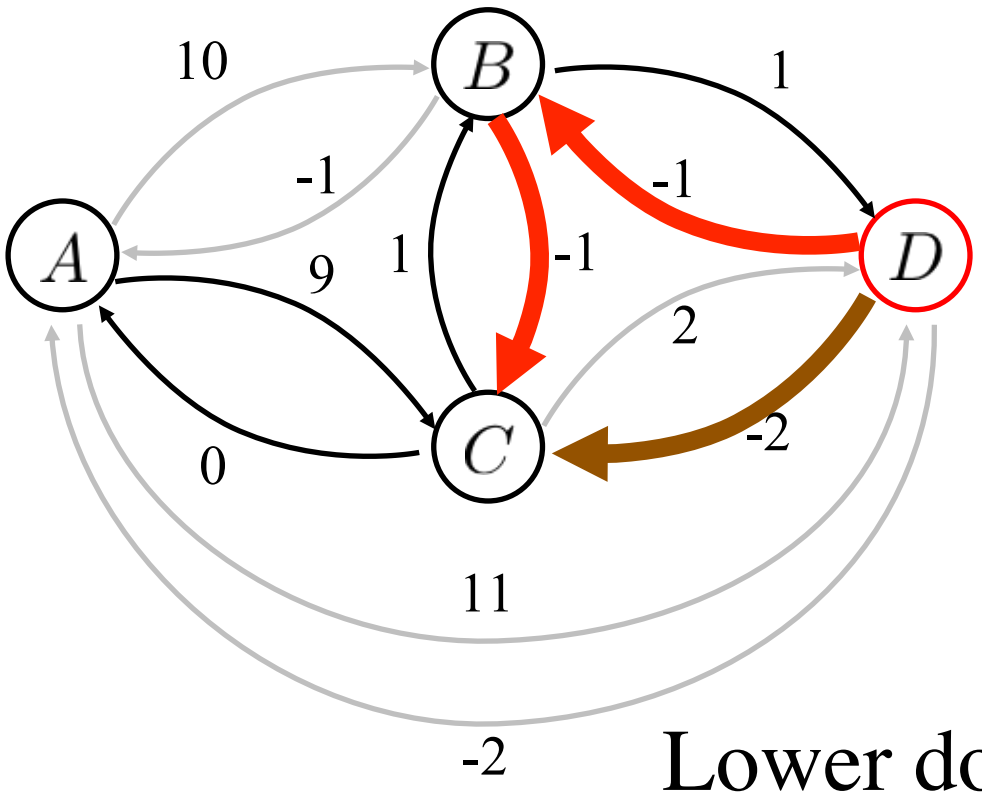
Lower dominated!

Dominance example



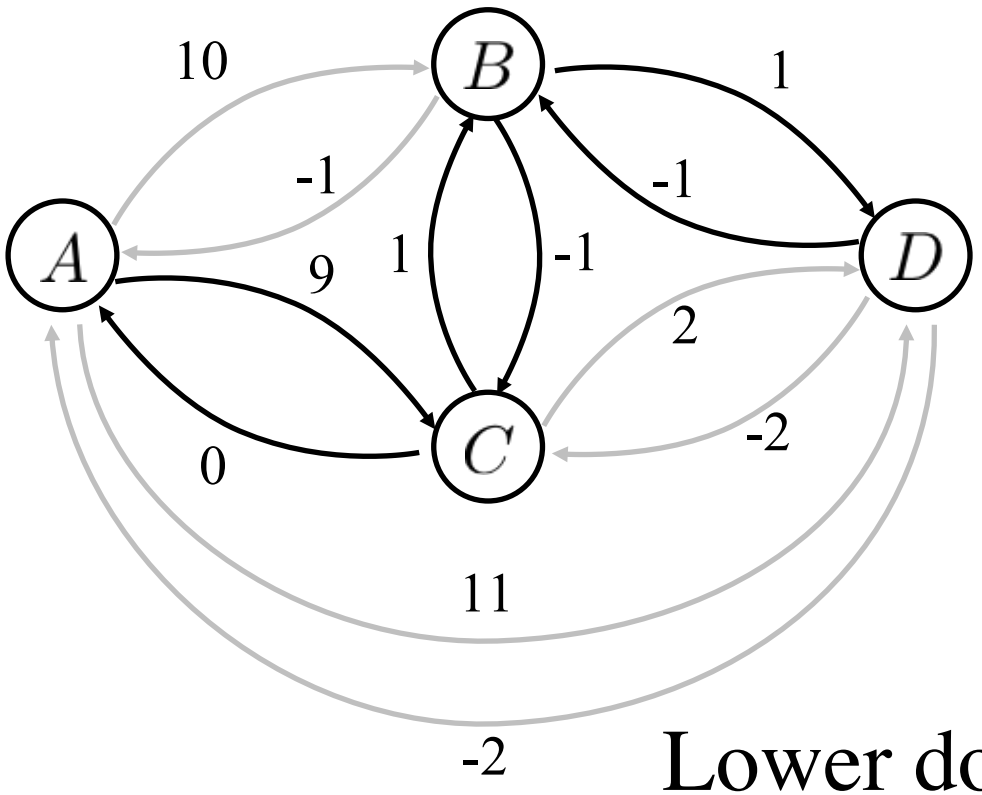
Lower dominated!

Dominance example



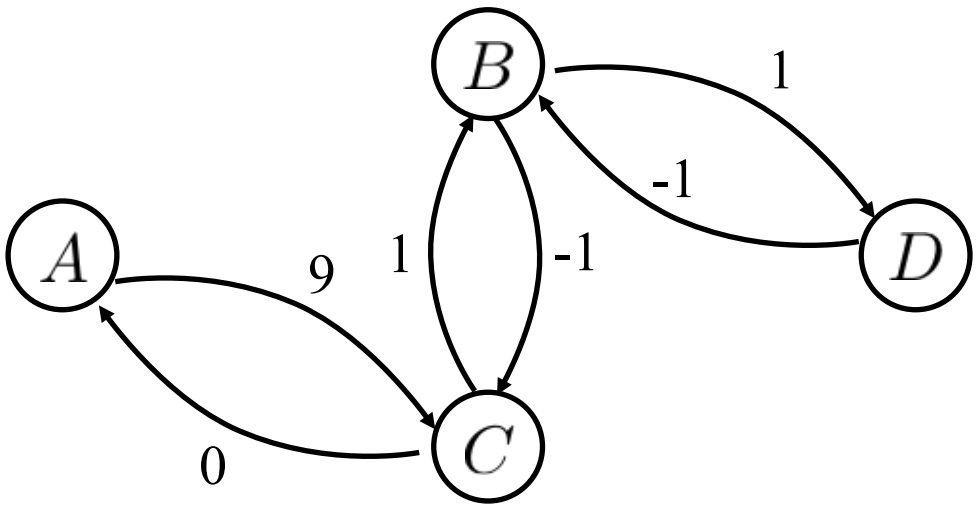
Lower dominated!

Dominance example

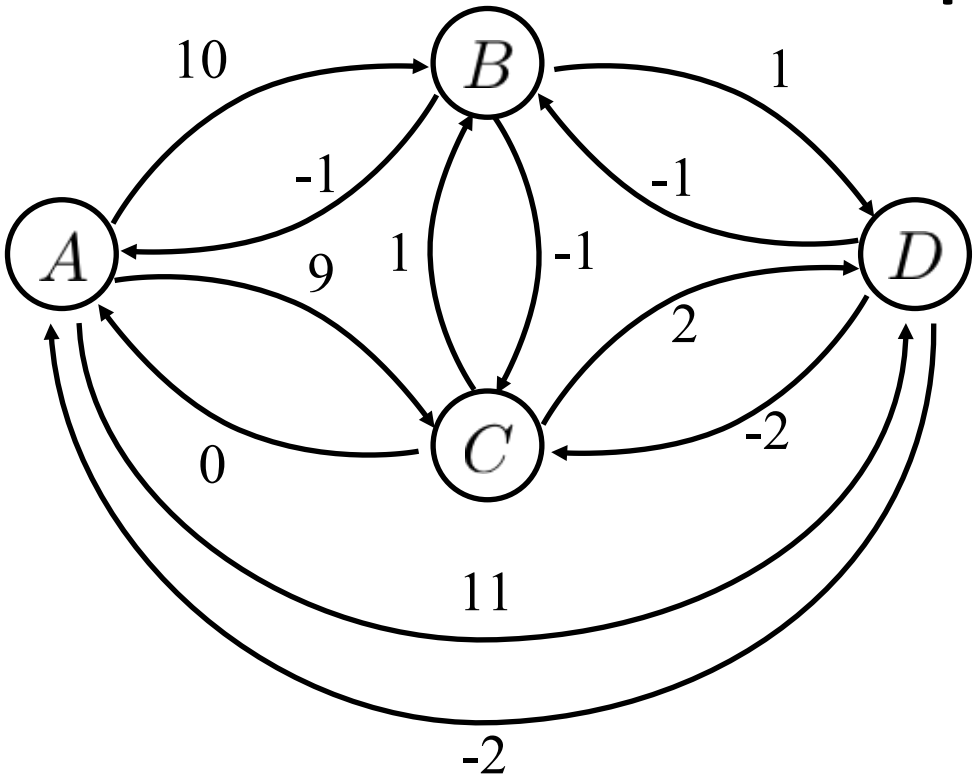


Lower dominated!

Dominance example

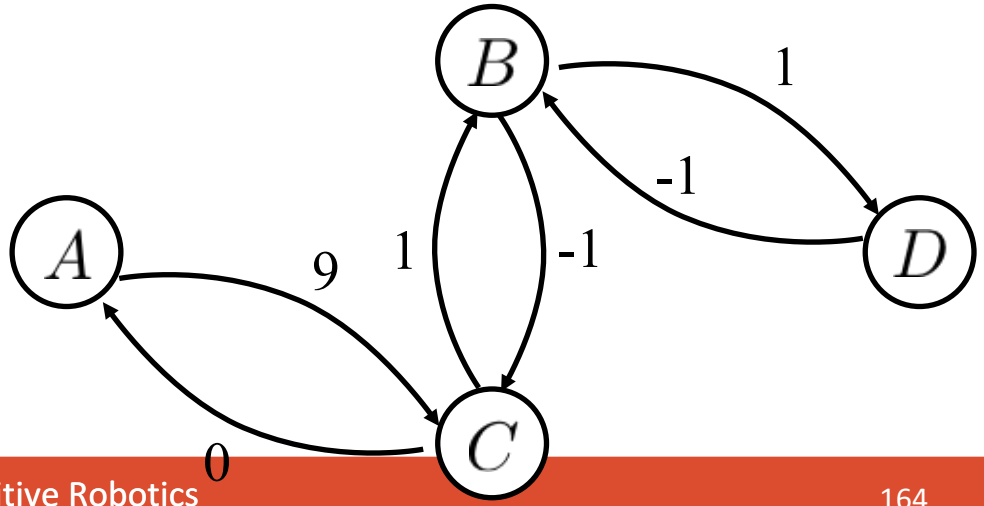


Dominance example

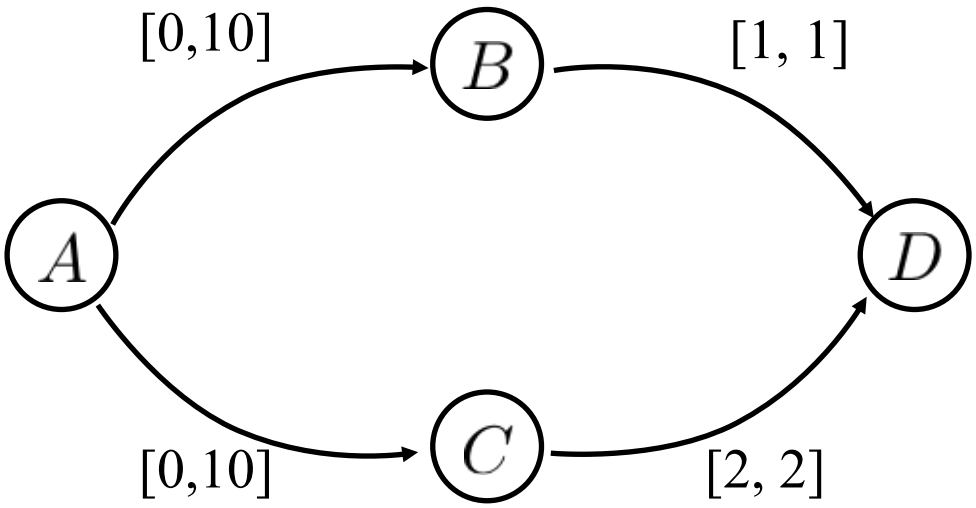


Original APSP distance graph

... now in ***minimal dispatchable form!***

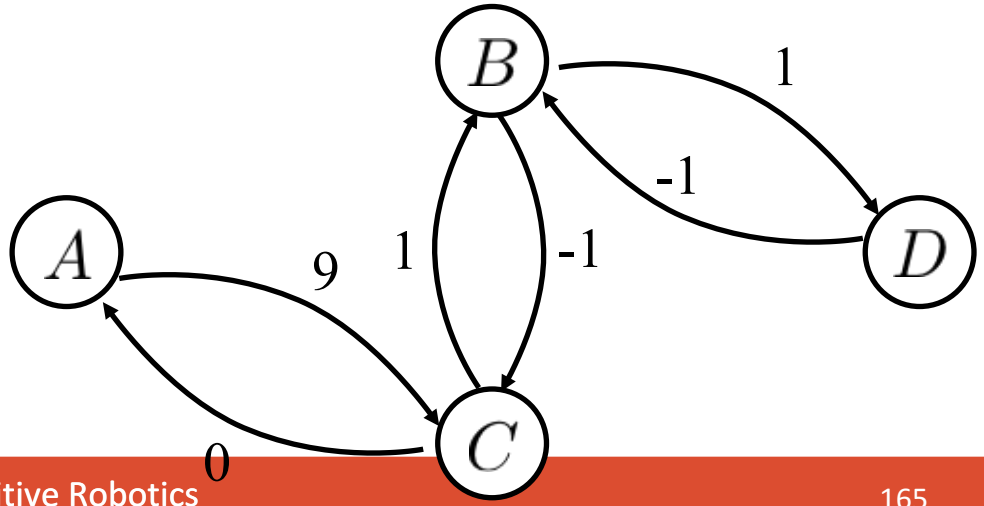


Dominance example

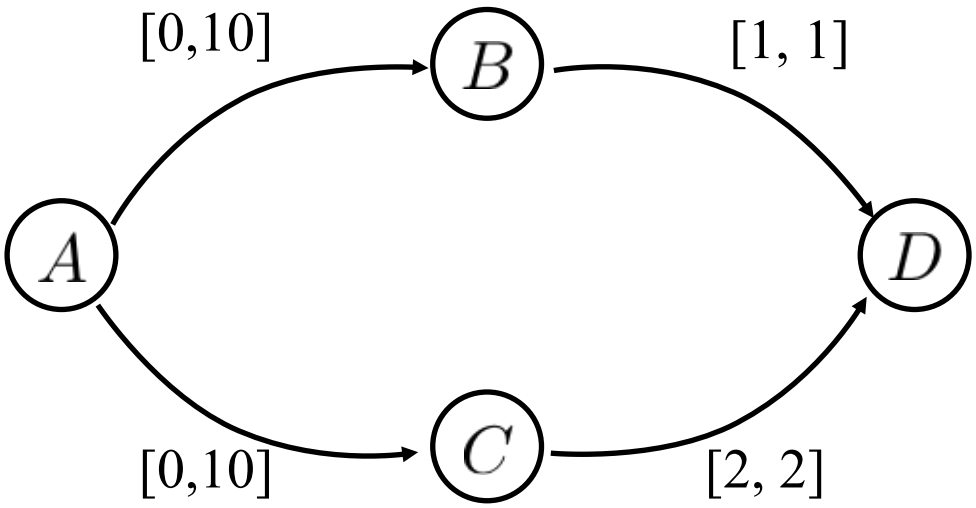


Original STN

... now in ***minimal dispatchable form!***

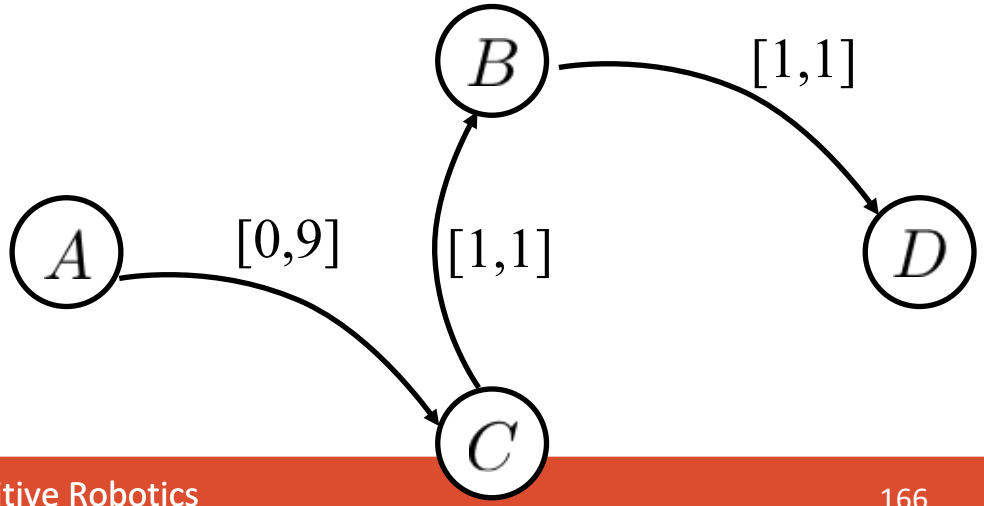


Dominance example



Original STN

... now in ***minimal dispatchable form!***



FilteringAlgorithm(G)

Input: A dispatchable APSP-graph G

Output: A minimal dispatchable graph

```
for each pair of intersecting edges in G
  if both dominate each other
    if neither is marked
      arbitrarily mark one for elimination
    end if
  else if one dominates the other
    mark dominated edge for elimination
  end if
end for
remove all marked edges from graph
return G
```

Avoiding Intermediate Graph Explosion

Problem:

- All pairs shortest path graph consumes $O(n^2)$ space.
- APSP result only used as an intermediate – not needed after minimal dispatchable graph obtained.

Solution:

- Interleave process of APSP construction with edge elimination.
  - Never need to build whole APSP graph.

[Tsarmardinos 1998]