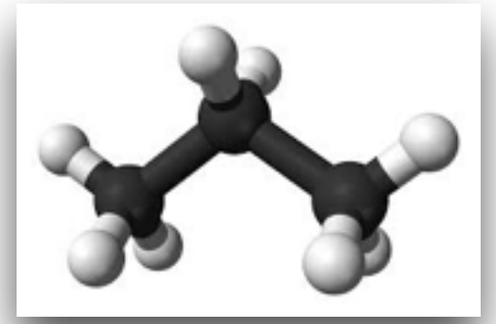


Propane

Programming Distributed Control Planes

Ryan Beckett (Princeton & MSR)
Ratul Mahajan (MSR)
Todd Millstein (UCLA)
Jitu Padhye (MSR)
David Walker (Princeton)

Configuring Networks is Error-Prone

Understanding BGP Misconfiguration

Ratul Mahajan David Wetherall Tom Anderson

{ratul,djw,tom}@cs.washington.edu
Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

ABSTRACT

It is well-known that simple, accidental BGP configuration errors can disrupt Internet connectivity. Yet little is known about the frequency of misconfiguration or its causes, except for the few spectacular incidents of widespread outages. In this paper, we present the first quantitative study of BGP misconfiguration. Over a three week period, we analyzed routing table advertisements from 23 vantage points across the Internet backbone to detect incidents of misconfiguration. For each incident we polled the ISP operators involved to verify whether it was a misconfiguration, and to learn the cause of the incident. We also actively probed the Internet to determine the impact of misconfiguration on connectivity.

Surprisingly, we find that configuration errors are pervasive, with 200-1200 prefixes (0.2-1.0% of the BGP table size) suffering from misconfiguration each day. Close to 3 in 4 of all new prefix advertisements were results of misconfiguration. Fortunately, the connectivity seen by end users is surprisingly robust to misconfigurations. While misconfigurations can substantially increase the update load on routers, only one in twenty five affects connectivity. While the causes of misconfiguration are diverse, we argue that most could be prevented through better router design.

Categories and Subject Descriptors

C.2.3 [**Communication Networks**]: Operations—*management*; C.4 [**Computer Systems**]: Performance—*reliability, availability, and serviceability*

General Terms

Human Factors, Management, Reliability

1. INTRODUCTION

As the Internet's inter-domain routing protocol, the Border Gateway Protocol (BGP) [34] is crucial to the overall reliability of the Internet. Faults in BGP implementations or mistakes in the way it is used have been known to disrupt large regions of the Internet. Recent studies have examined several kinds of BGP problems, including excessive churn due to implementation deficiencies [26], delayed convergence [24], persistent oscillations due to policy interactions [18, 40], and instability caused by the propagation of worms [9].

In this paper we examine another source of unreliability: the misconfiguration of the routers that speak BGP. We know from numerous studies of highly reliable systems, such as aircraft, bank databases, and the telephone network, that human operator error can account for 20-70% of system failures [3, 6, 15, 21]. These studies have shown that as systems become more reliable, the human factor becomes increasingly important to overall reliability. We would expect the same to be true of the Internet. There is substantial anecdotal evidence that BGP configuration errors do occur, with serious consequences. The canonical example is the AS7007 incident [31], in which AS7007 accidentally announced routes to most of the Internet and disrupted connectivity for over two hours. Despite the publicity over this event, serious misconfigurations continue to occur. In April 2001, AS3561 propagated more than 5000 improper route announcements from one of its downstream customers [12], again leading to global connectivity problems.

In this paper, we complement the anecdotal study of infrequent large scale events with a microscopic study of more frequent "near misses" – globally visible BGP misconfigurations that occur many times per day but do not necessarily disrupt connectivity. As with the study of airplane near collisions, we hope that our study of microscopic events can help improve the design of systems to avoid future larger scale problems. To the best of our knowledge, this is the first systematic study of globally visible BGP misconfigurations; it complements other studies that have examined backbone failures in general [25]. Our goal is to answer four questions:

- *How frequently do these misconfigurations occur?*
- *What is their impact on global connectivity and routing load?*
- *Why do the misconfigurations occur?*
- *What can be done to reduce their frequency and impact?*

We consider two broad classes of faults that propagate across the backbone and hence are visible from our measurement points: i) the accidental injection of routes into global BGP tables, including address space hijacks; and ii) the accidental export of routes in violation of an ISP's policy. We focus on misconfigurations that are globally visible because they arguably have the potential to cause wider disruption than those that do not propagate across the Internet We analyze the entire stream of BGP updates taken from 23 different vantage points around the Internet for a period of 21 days. We show that it is possible to identify misconfigurations with simple heuristics. To validate our results, we surveyed the ISP operators involved in each incident via email. We asked them whether the in-

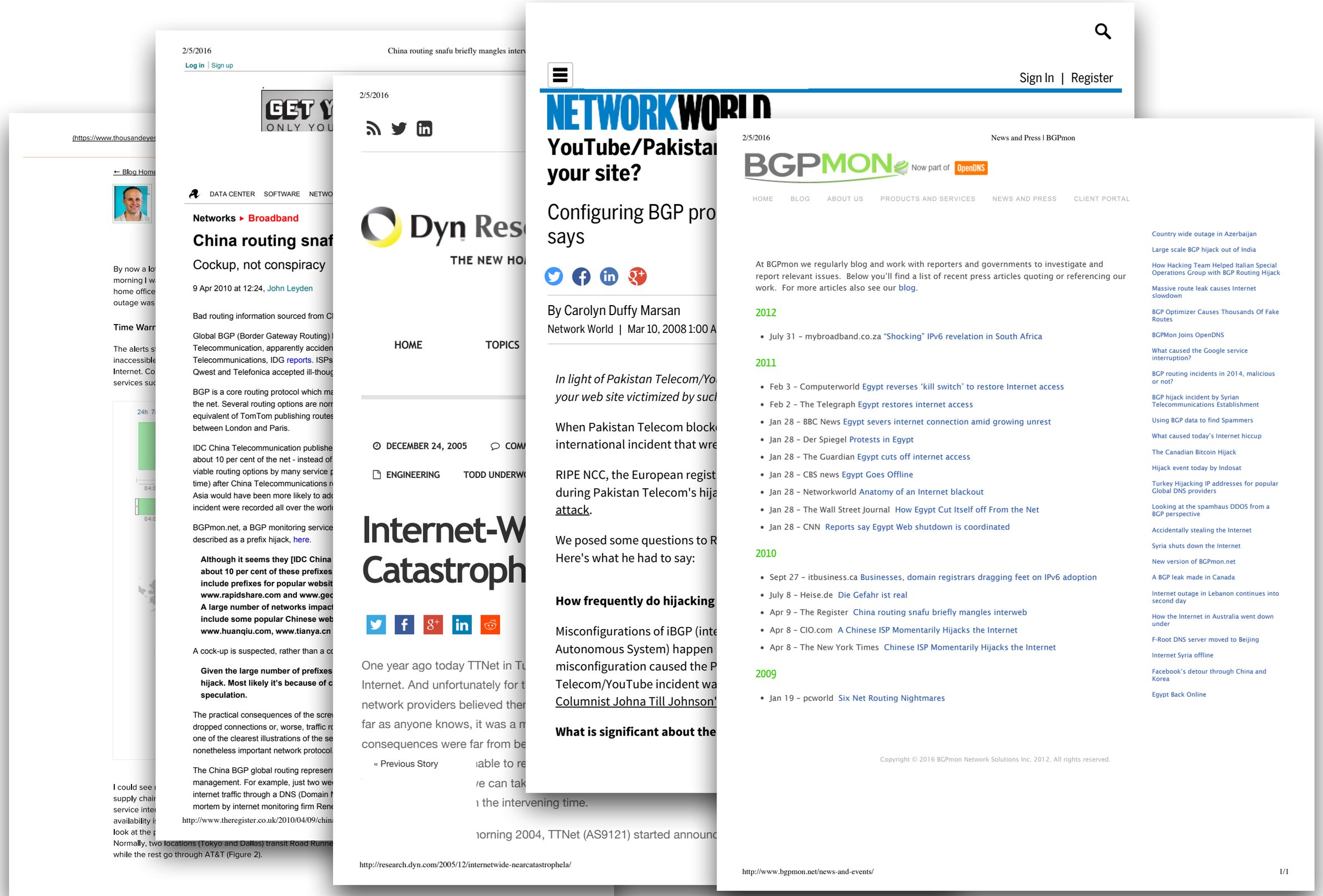
SIGCOMM'02, August 19-23, 2002, Pittsburgh, Pennsylvania, USA.

"Close to 3 in 4 of all new prefix advertisements were results of misconfiguration"

Causes:

- operator slips
- low-level details: filters, communities
- logical errors in plans
- misunderstanding of configuration semantics
- router bugs

Configuring Networks is Error-Prone



Objectives: Network-wide

- Prefer traffic to go through AT&T over Verizon
- Enforce isolation between two locations
- Don't use our network as transit between A and B
- Traffic must stay within national boundaries
- Don't leak internal services to the outside world
- Adhere to policies in the good times and the bad (failures)

Mechanisms: Device-by-Device

- Local decisions made independently on each device
- Several device-level actions together imply some higher-level behavior
- Failures interact with local decision-making algorithms in complex ways

Recent Developments in Network Programming

A new suite of SDN programming languages provide:

- Simpler, centralized programming models
- Network-wide abstractions like paths
- Compositional construction of complex policies from simpler parts
- Examples include:
  - Frenetic [ICFP '11], PANE [HotSDN '12], Fat Tire [HotSDN '13], Pyretic [NSDI '13], Merlin [CoNext '13], FlowLog [NSDI '14], NetKAT [POPL '14], …

But they aren't a panacea:

- They consider *intra*-domain routing but not *inter*-domain routing issues
- Programs don't express *preferences* ahead of time in case of failures
- Their implementations don't exploit existing network infrastructure
- One must still build controller infrastructure that is fault tolerant and scalable — issues often left undiscussed in academic projects

Propane:
Programming a Distributed Control Plane

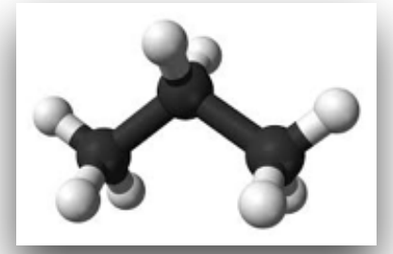
A language for expression of high-level operator objectives with:

- regular paths and relative preferences with fall-backs in case of failures
- uniform abstractions for intra- and inter-domain routing
- network-wide constraints that make some low-level errors impossible to express

And a compiler that bridges the gap between high-level objectives and low-level distributed control plane mechanisms:

- efficient algorithms to synthesize a set of policy-compliant BGP configs
  - reuse existing infrastructure
  - fault tolerant and scalable
- failure analysis guarantees *policy compliance* under all failures

This Talk



Two Examples (simplified but emblematic of the issues)

- Operator objectives and challenges
- Propane solutions

The Propane Compiler

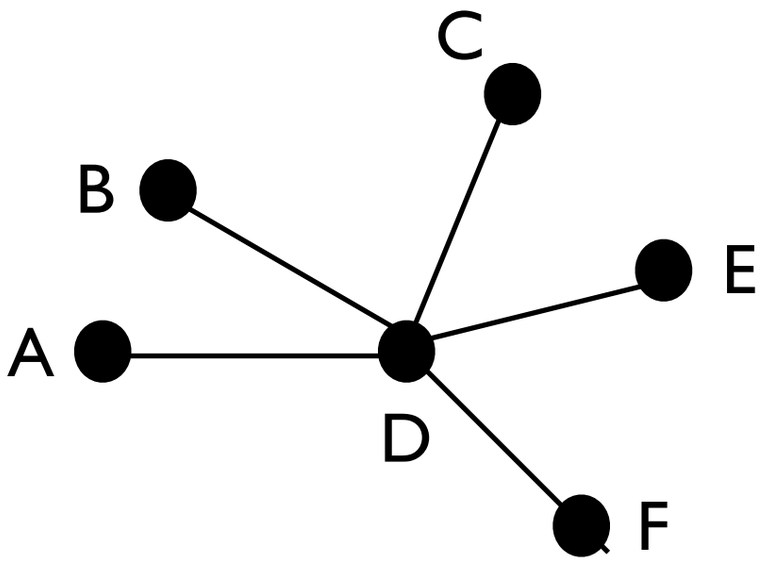
- Intermediate representations
- Compilation steps
- Failure analyses

Wrap-up

- Future work
- Conclusions

BGP Protocol

Prefix = 1.2.3.4/32

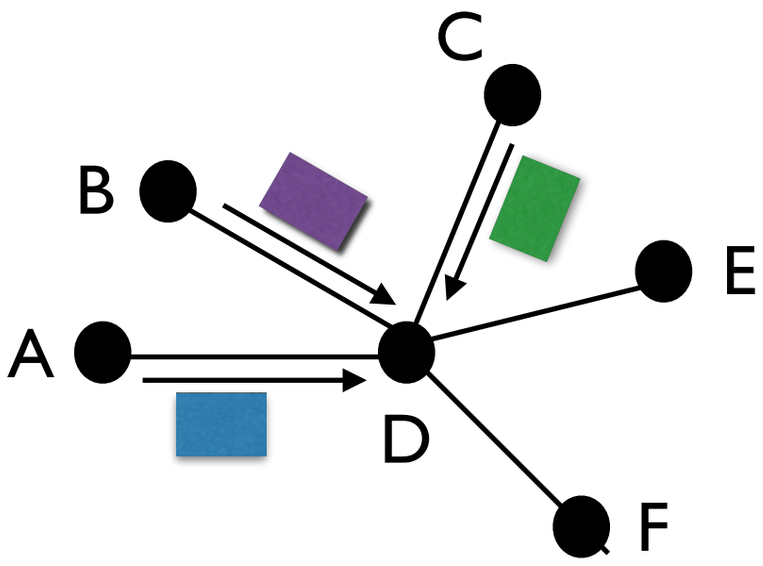


BGP Nodes:

- receive and filter announcements
  - modifying attributes
- decide on the best route per prefix using announcement attributes:
  - local preference, AS path length, …
- export and filter routes to neighbors

BGP Protocol

Prefix = 1.2.3.4/32



BGP Nodes:

- receive and filter announcements
  - modifying attributes
- decide on the best route per prefix using announcement attributes:
  - local preference, AS path length, …
- export and filter routes to neighbors

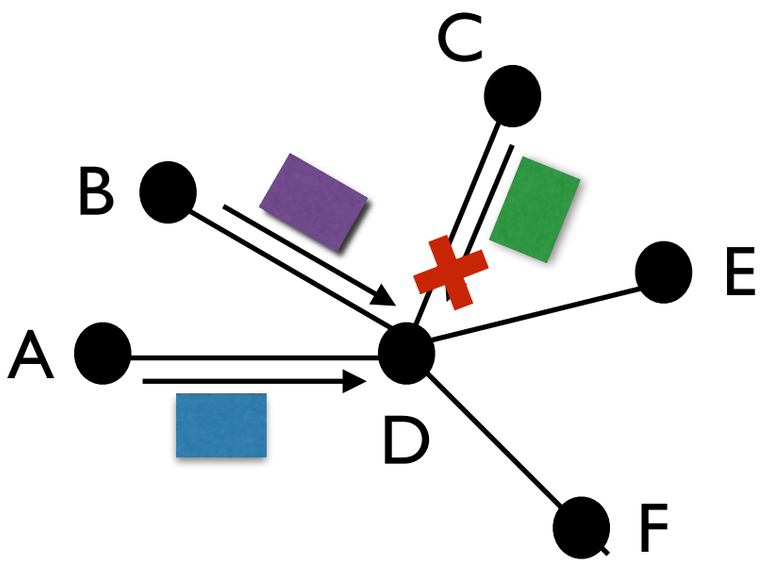
{Path=C X, Comm={}, … }

{Path=B Y X, Comm={6054:10}, … }

{Path=A X, Comm={6054:10}, … }

BGP Protocol

Prefix = 1.2.3.4/32



{Path=C X, Comm={}, … }

{Path=B Y X, Comm={6054:10}, … }

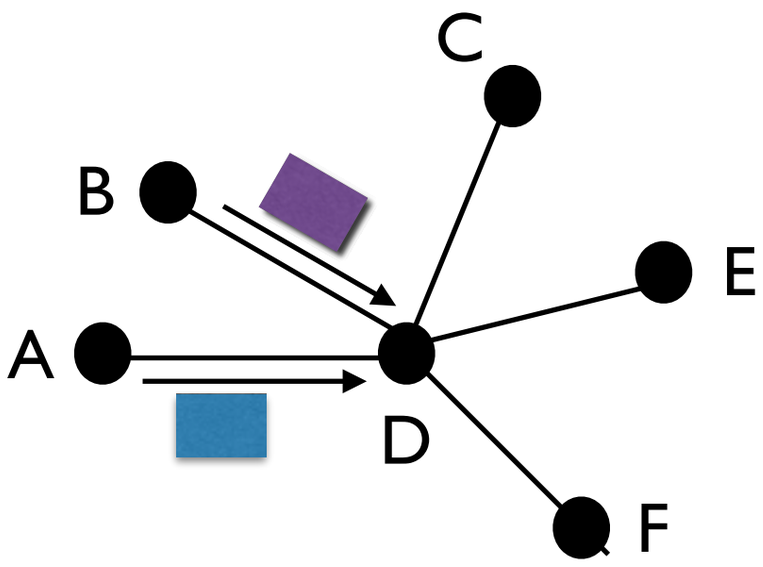
{Path=A X, Comm={6054:10}, … }

BGP Nodes:

- receive and filter announcements
  - modifying attributes
- decide on the best route per prefix using announcement attributes:
  - local preference, AS path length, …
- export and filter routes to neighbors

BGP Protocol

Prefix = 1.2.3.4/32



BGP Nodes:

- receive and filter announcements
  - modifying attributes
- decide on the best route per prefix using announcement attributes:
  - local preference, AS path length, …
- export and filter routes to neighbors

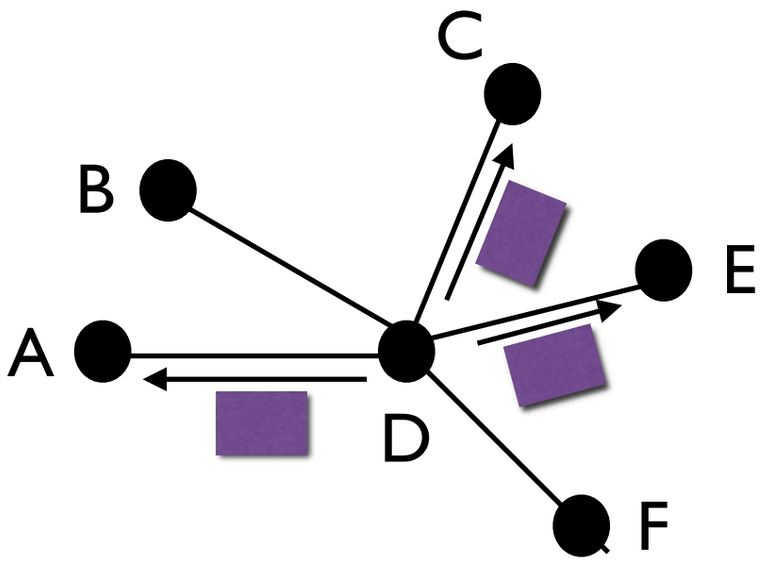
~~{Path=C X, Comm={}, … }~~

{Path=B Y X, Comm={6054:10}, … }

{Path=A X, Comm={6054:10}, … }

BGP Protocol

Prefix = 1.2.3.4/32

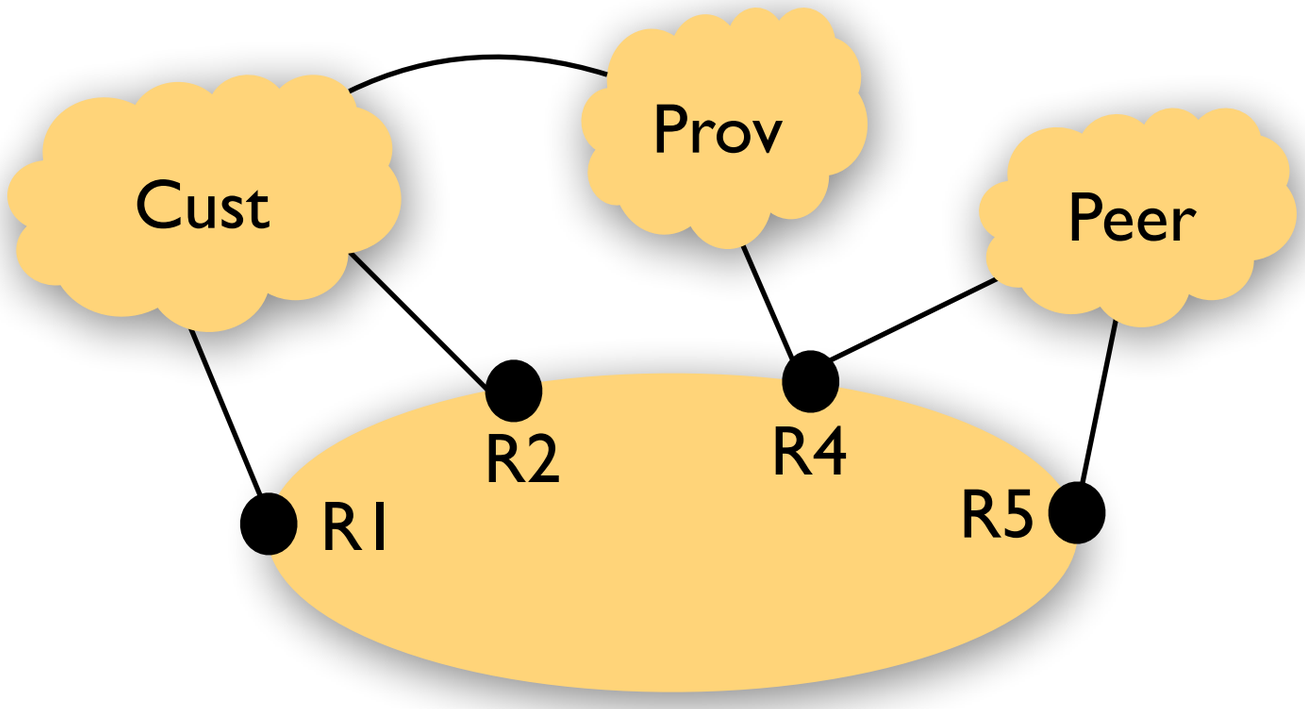


BGP Nodes:

- receive and filter announcements
  - modifying attributes
- decide on the best route per prefix using announcement attributes:
  - local preference, AS path length, …
- export and filter routes to neighbors

Example I: A Backbone Network

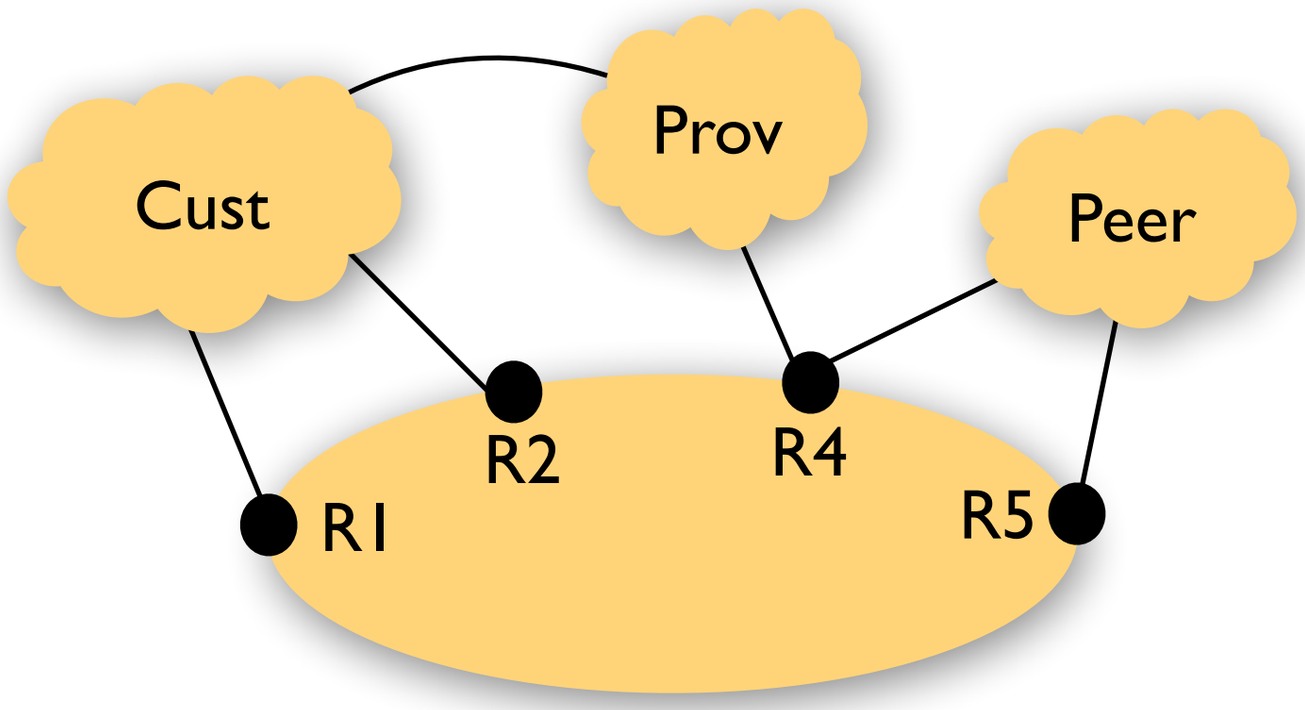
A Backbone Network



Goals:

- P1: Prefer to exit thru Cust > Peer > Prov
- P2: Disallow traffic between Prov and Peer
- P3: For Cust, prefer exit thru R1 > R2
- P4: Cust must be on path for its prefixes

A Backbone Network



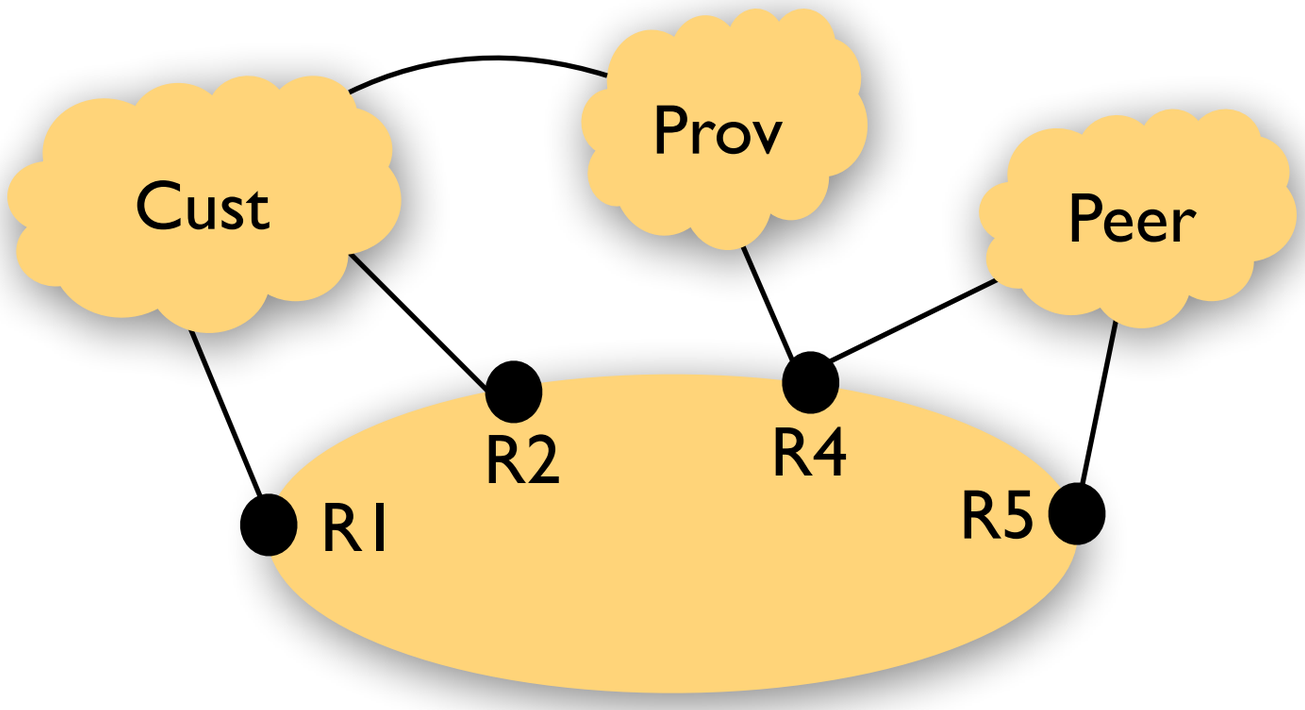
Goals:

- P1: Prefer to exit thru Cust > Peer > Prov
- P2: Disallow traffic between Prov and Peer
- P3: For Cust, prefer exit thru R1 > R2
- P4: Cust must be on path for its prefixes

Implementation Techniques:

- Compute and set local preferences consistently at all Peer-facing, Cust-facing, Prov-facing interfaces
- For Cust, ensure R2 local pref is lower than R1

A Backbone Network



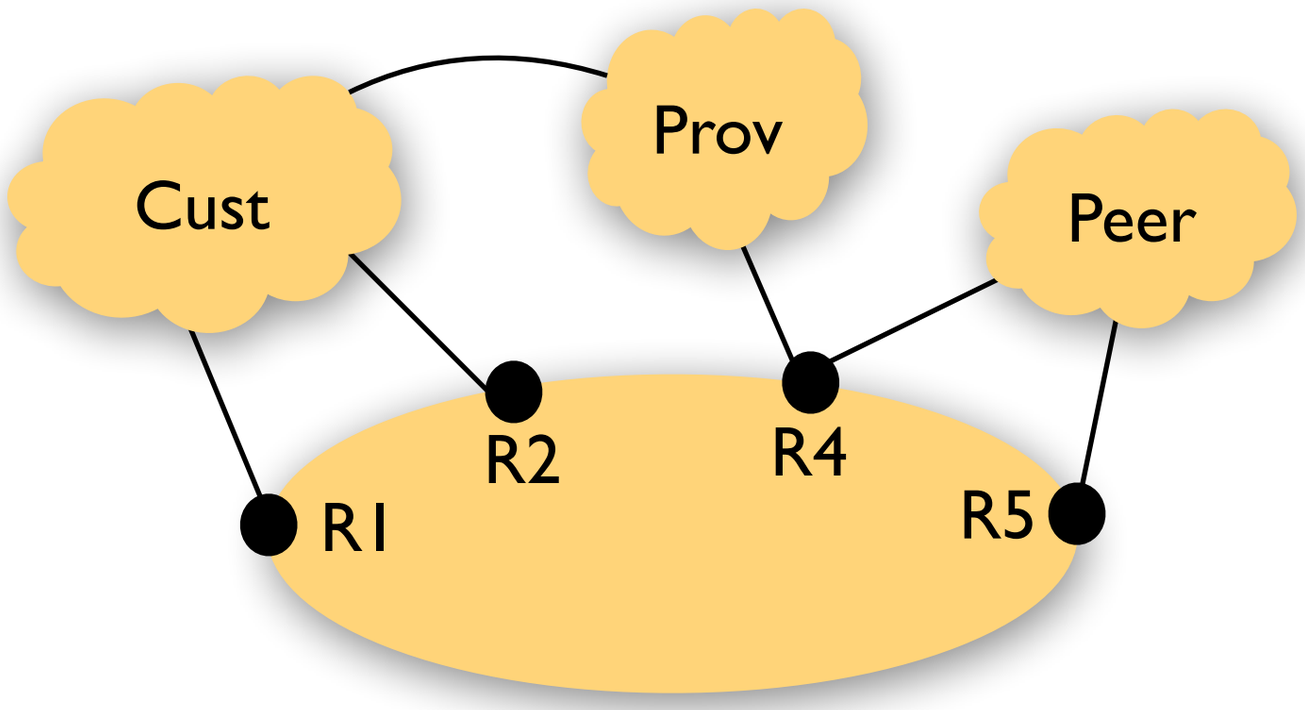
Goals:

- P1: Prefer to exit thru Cust > Peer > Prov
- P2: Disallow traffic between Prov and Peer
- P3: For Cust, prefer exit thru R1 > R2
- P4: Cust must be on path for its prefixes

Implementation Techniques:

- Use communities to mark the location an announcement entered the network

A Backbone Network



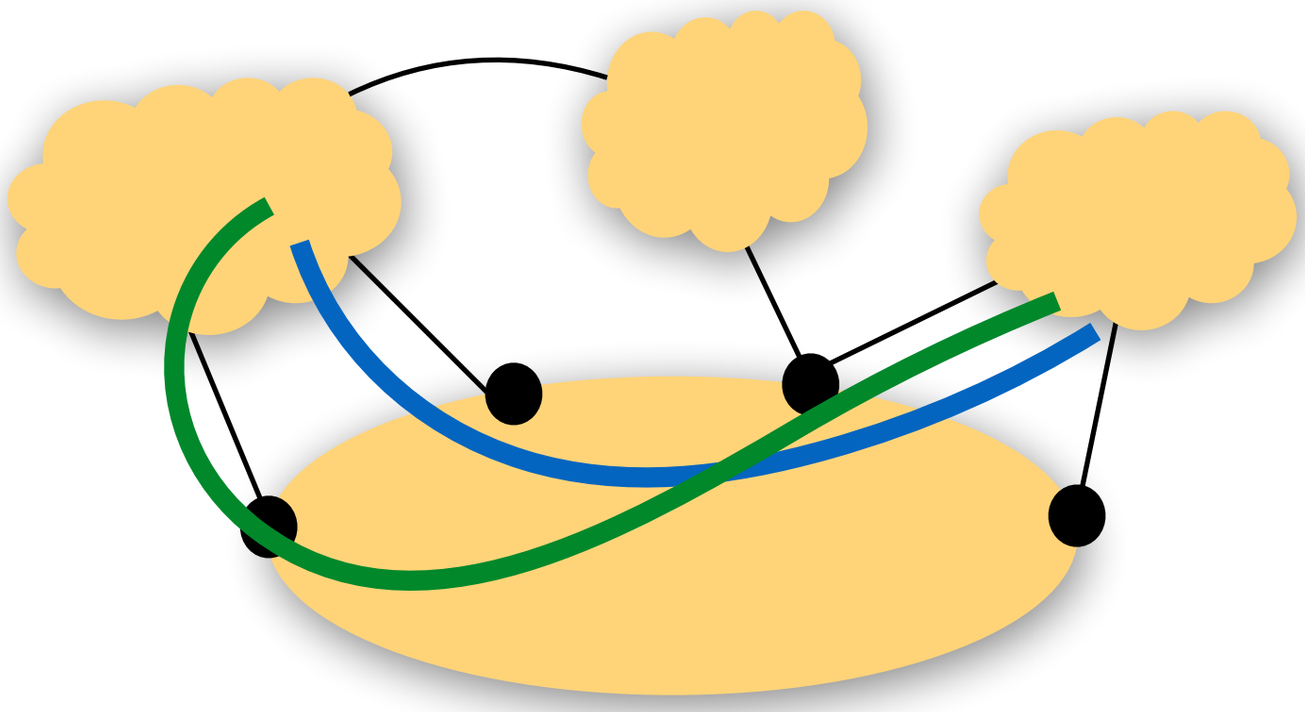
Goals:

- P1: Prefer to exit thru Cust > Peer > Prov
- P2: Disallow traffic between Prov and Peer
- P3: For Cust, prefer exit thru R1 > R2
- P4: Cust must be on path for its prefixes

Implementation Techniques:

- More filters to drop undesirable routes

Propane



A Propane *policy* associates *sets of ranked paths* with prefixes:

- Such paths define the desired flow of traffic
- Paths stretch from other networks, through our network, back out to neighbors
- Preferences between sets of paths express desired behavior when faults occur

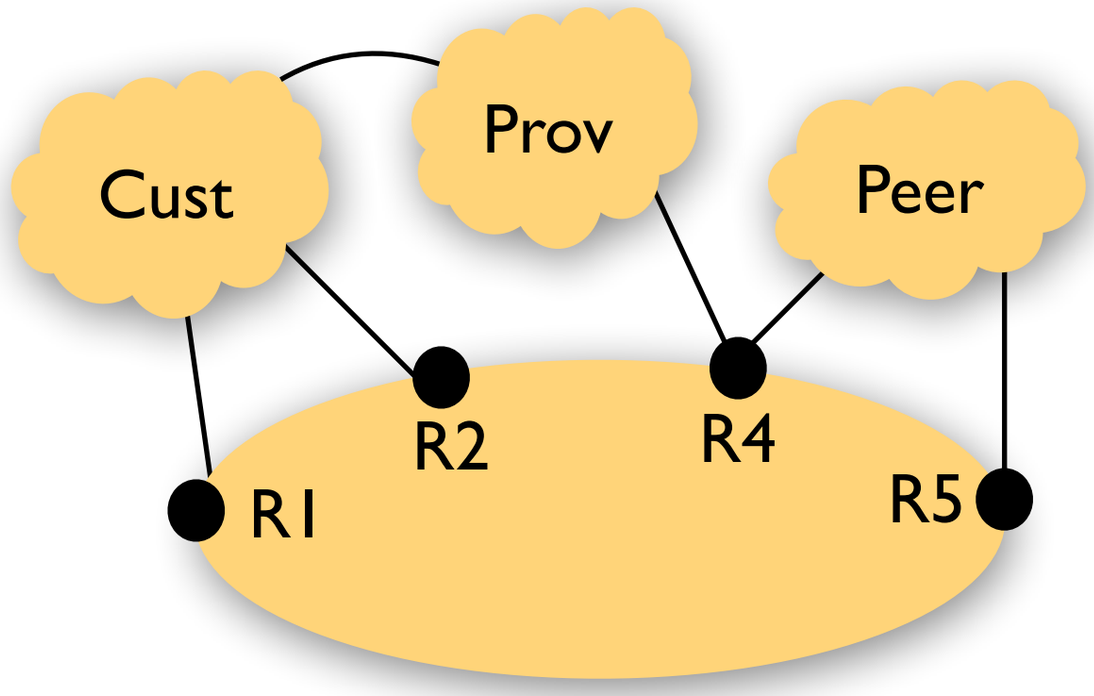
Propane

P1: Prefer to exit thru Cust > Peer > Prov

P2: Disallow traffic between Prov and Peer

P3: For Cust, prefer exit thru R1 > R2

P4: Cust must be on path for its prefixes




```
define Routing = {
  true  => exit(R1 >> R2 >> Peer >> Prov)
}
```

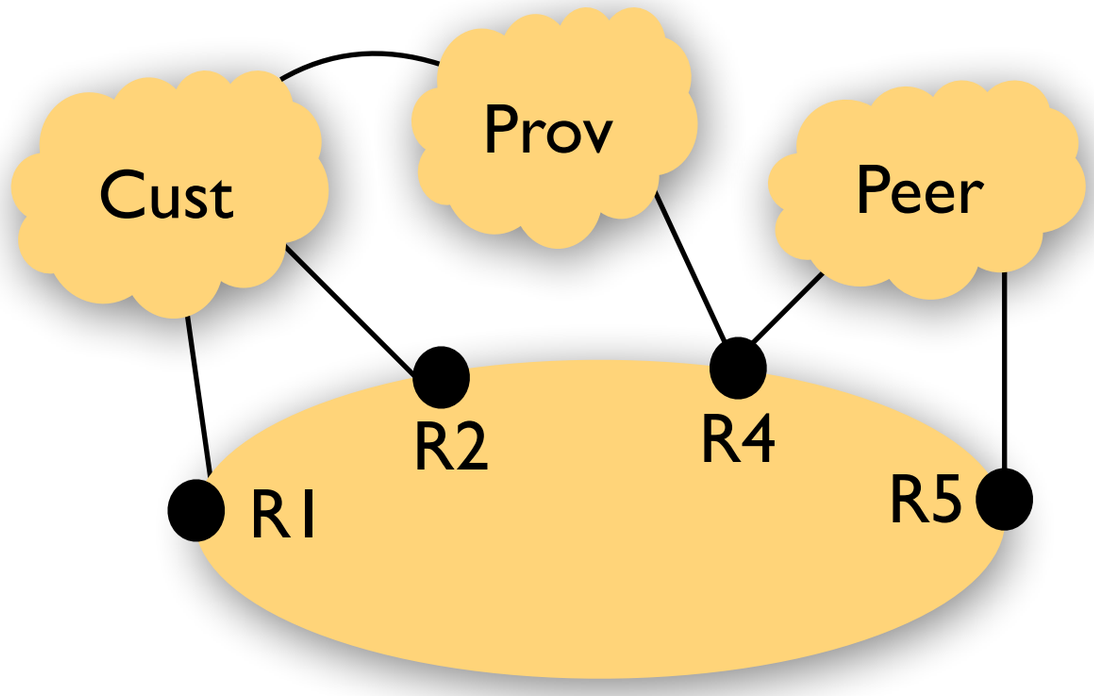
Propane

P1: Prefer to exit thru Cust > Peer > Prov

P2: Disallow traffic between Prov and Peer

P3: For Cust, prefer exit thru R1 > R2

P4: Cust must be on path for its prefixes




```
define Routing = {
  true  => exit(R1 >> R2 >> Peer >> Prov)
}

define Ownership = {
  PCust => later(Cust)
}
```

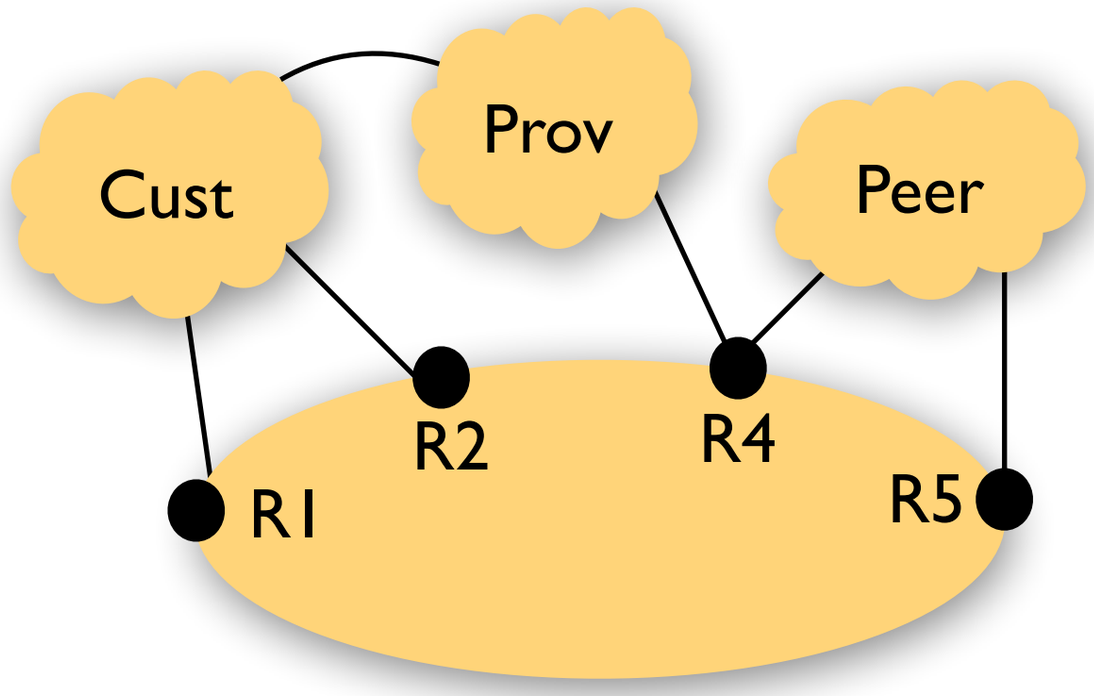
Propane

P1: Prefer to exit thru Cust > Peer > Prov

P2: Disallow traffic between Prov and Peer

P3: For Cust, prefer exit thru R1 > R2

P4: Cust must be on path for its prefixes




```
define transit(X) = enter(X) & exit(X)

define NoTransit = {
    true => !transit(Peer|Prov)
}
```

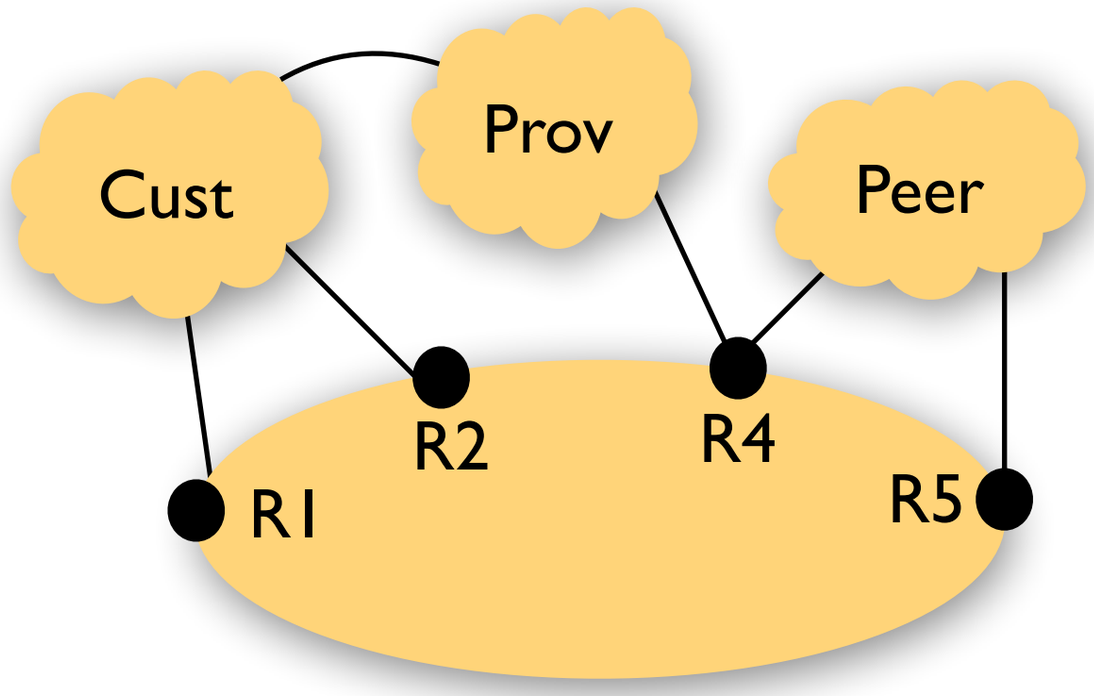
Propane

P1: Prefer to exit thru Cust > Peer > Prov

P2: Disallow traffic between Prov and Peer

P3: For Cust, prefer exit thru R1 > R2

P4: Cust must be on path for its prefixes




```
define transit(X) = enter(X) & exit(X)

define NoTransit = {
    true => !transit(Peer|Prov)
}

define Main = Routing & Ownership & NoTransit
```

Example II: A Data Center Network

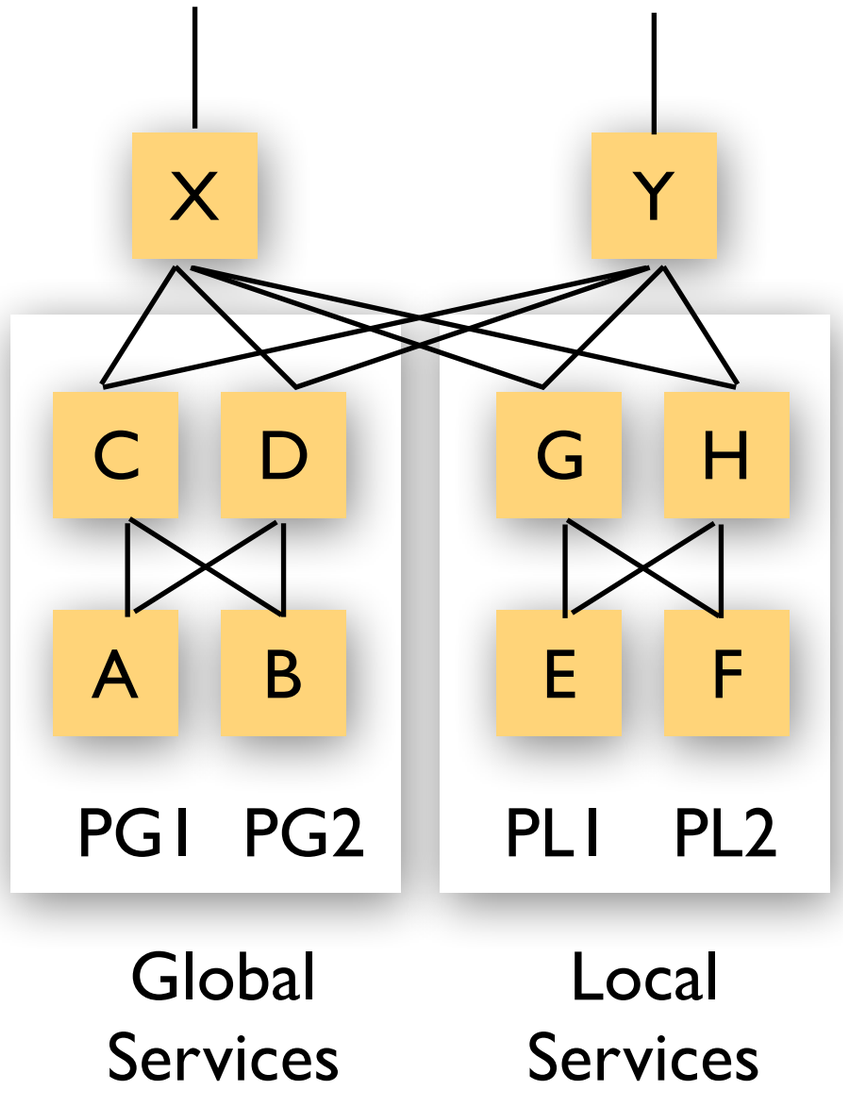
A Data Center Network

Goals:

- P1: Announce global services externally as the aggregate PG
- P2: Do not announce local services externally

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X, Y do not need to know which prefixes are local vs global
- aggregate to PG if announce is subset of PG



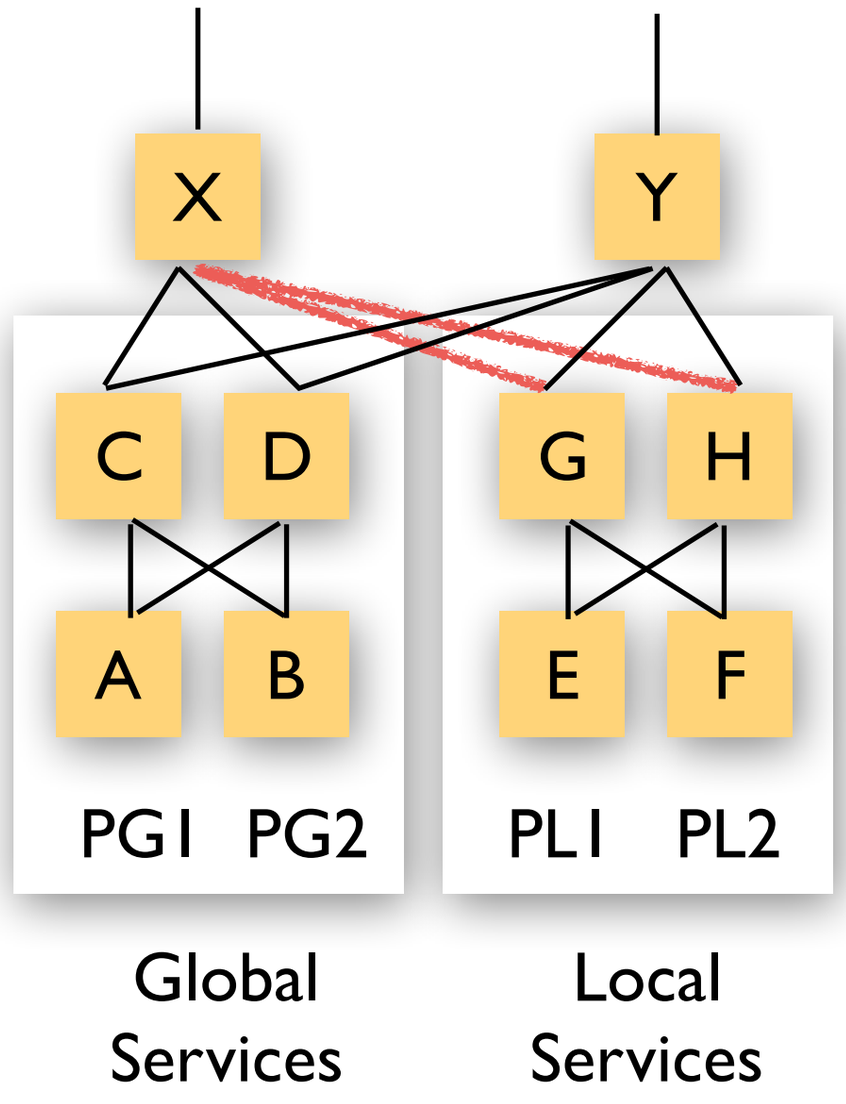
A Data Center Network

Goals:

- P1: Announce global services externally as the aggregate PG
- P2: Do not announce local services externally

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X, Y do not need to know which prefixes are local vs global
- aggregate to PG if announce is subset of PG



Consider X-G, X-H Failure:

- PL* announcements travel H-Y-D-X
- PL* announcements are then leaked
- Without failure, X selects direct H-X path, which is shorter

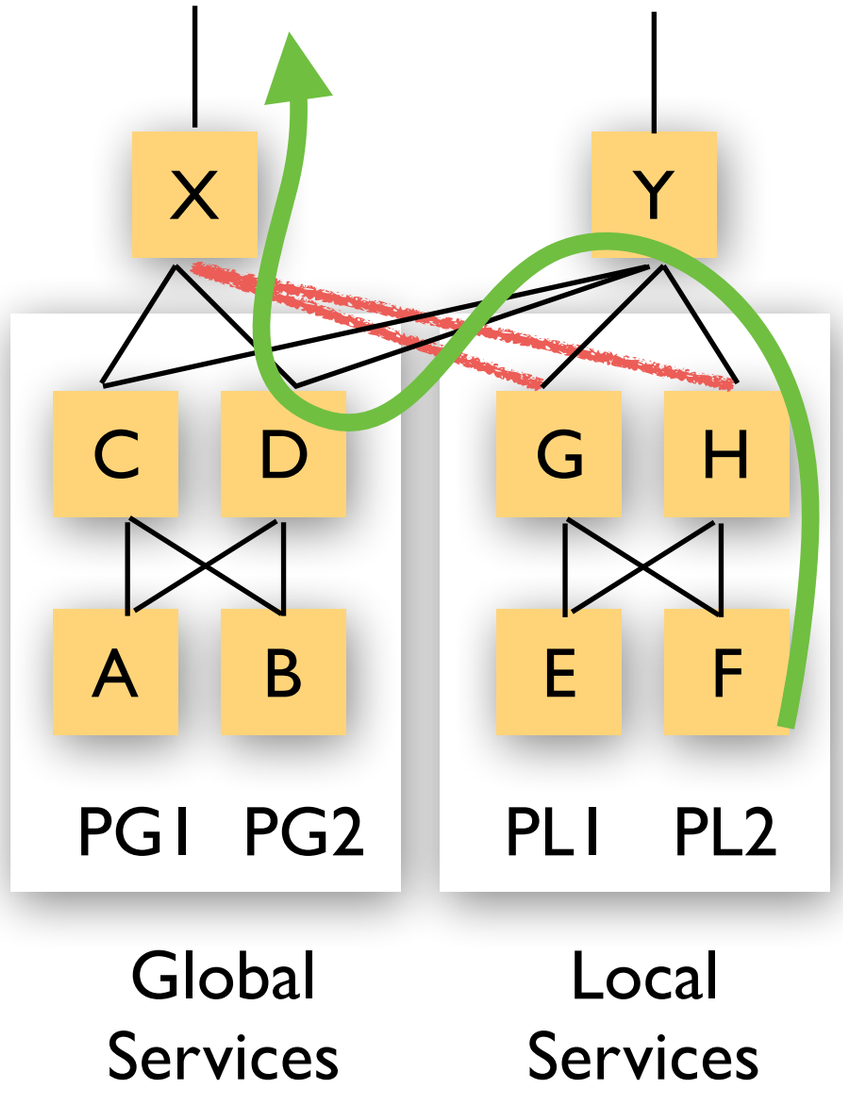
A Data Center Network

Goals:

- P1: Announce global services externally as the aggregate PG
- P2: Do not announce local services externally

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X, Y do not need to know which prefixes are local vs global
- aggregate to PG if announce is subset of PG



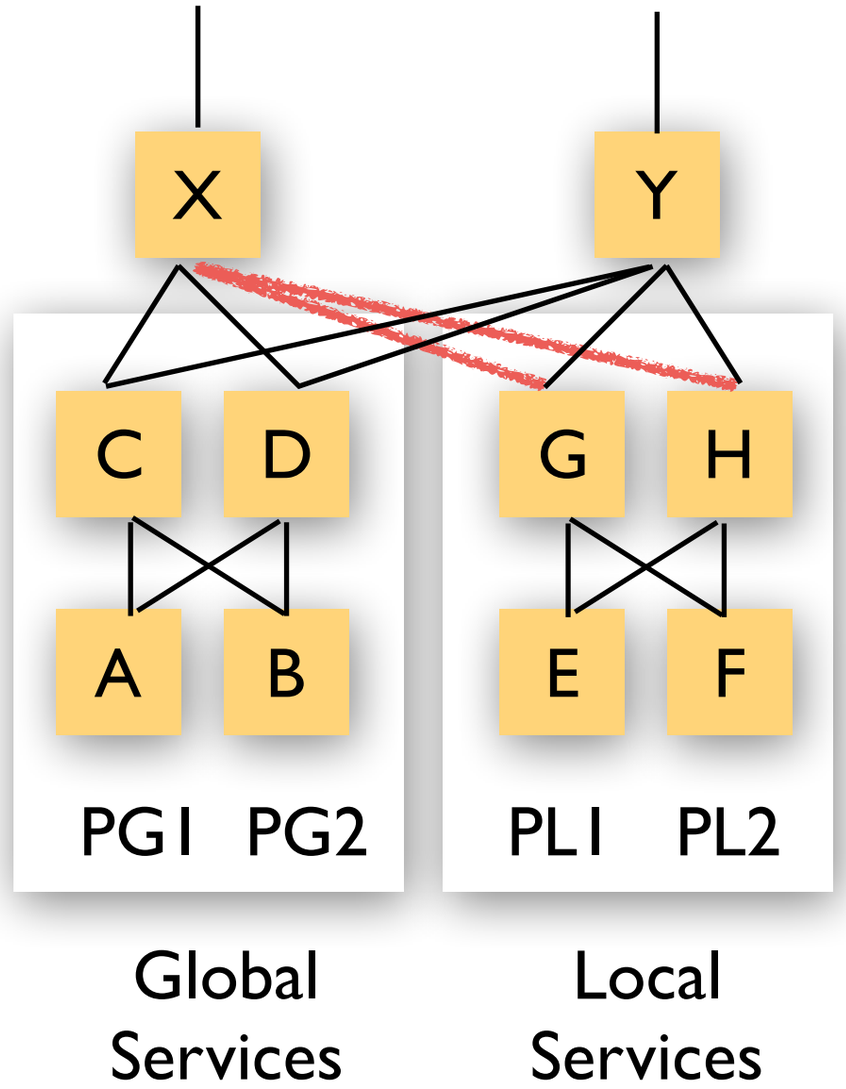
Consider X-G, X-H Failure:

- PL* announcements travel H-Y-D-X
- PL* announcements are then leaked

A Data Center Network

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X,Y do not need to know which prefixes are local vs global
- aggregate to PG if announce is subset of PG
- disallow "valley" paths

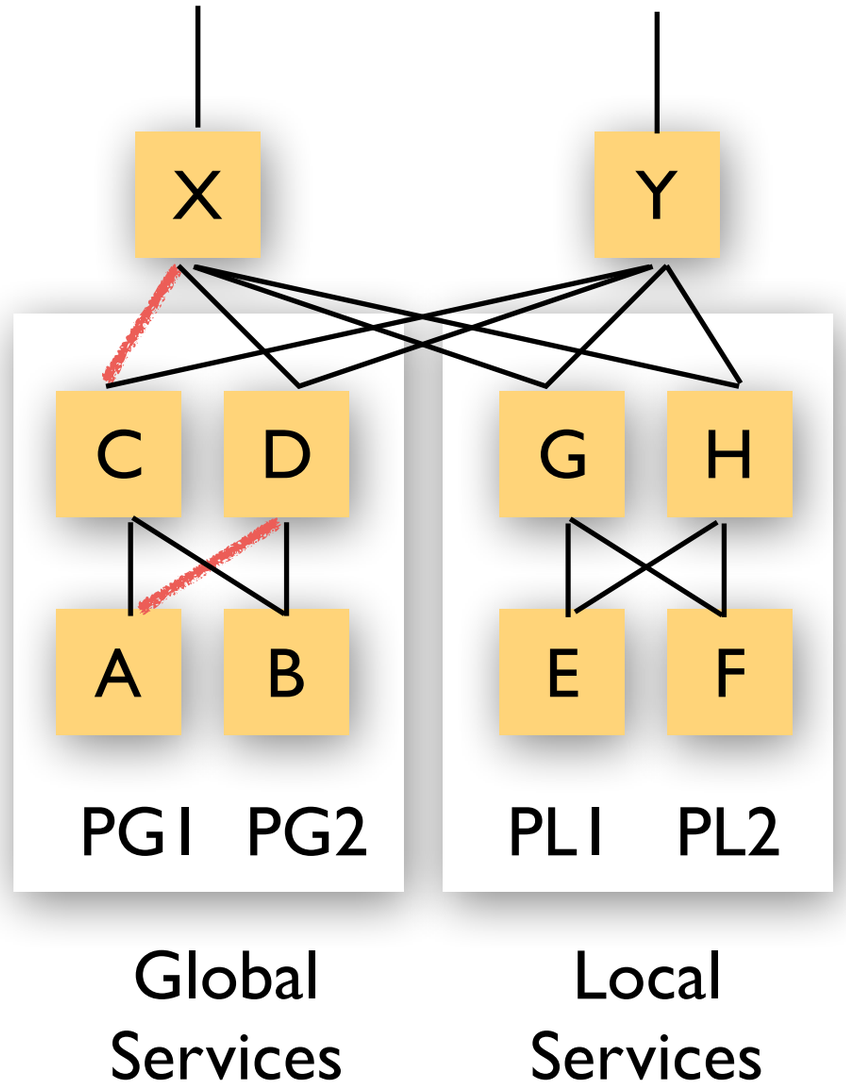


A Data Center Network

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X,Y do not need to know which prefixes are local vs global
  - aggregate to PG if announce is subset of PG
  - disallow "valley" paths

Consider D-A, X-C Failure:



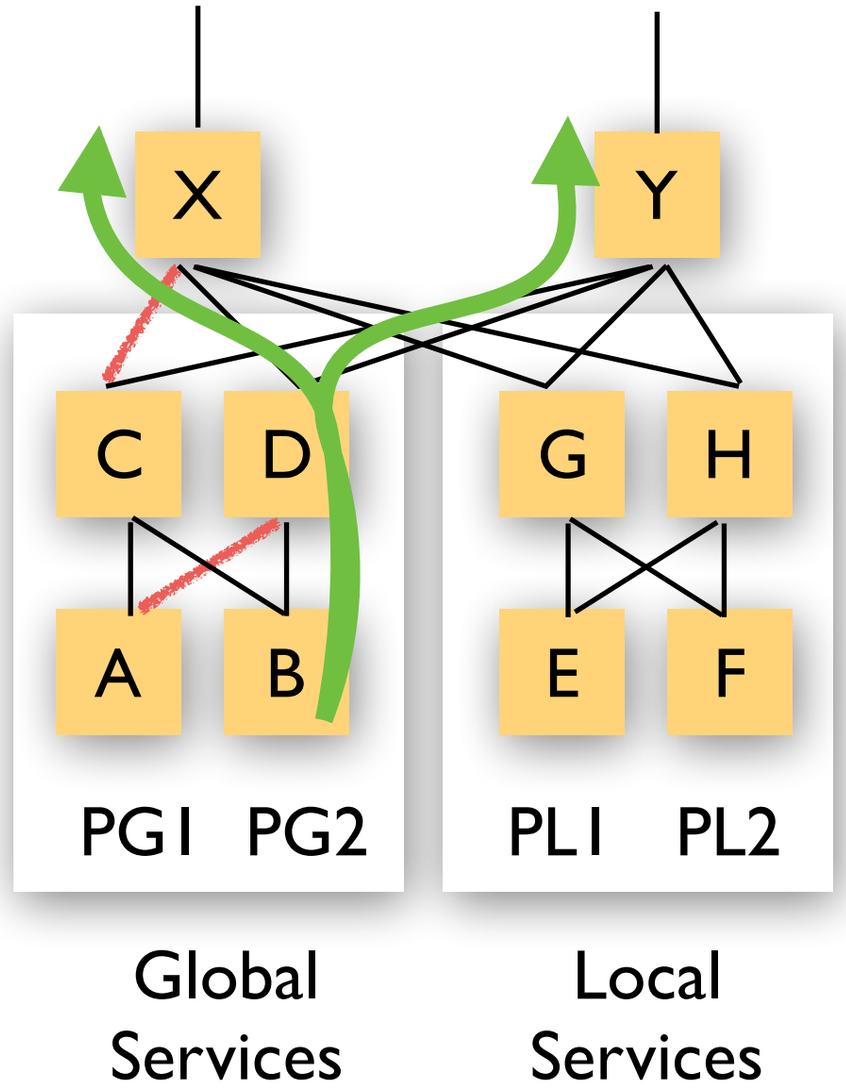
A Data Center Network

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X,Y do not need to know which prefixes are local vs global
  - aggregate to PG if announce is subset of PG
  - disallow "valley" paths

Consider D-A, X-C Failure:

- X and Y will hear PG2



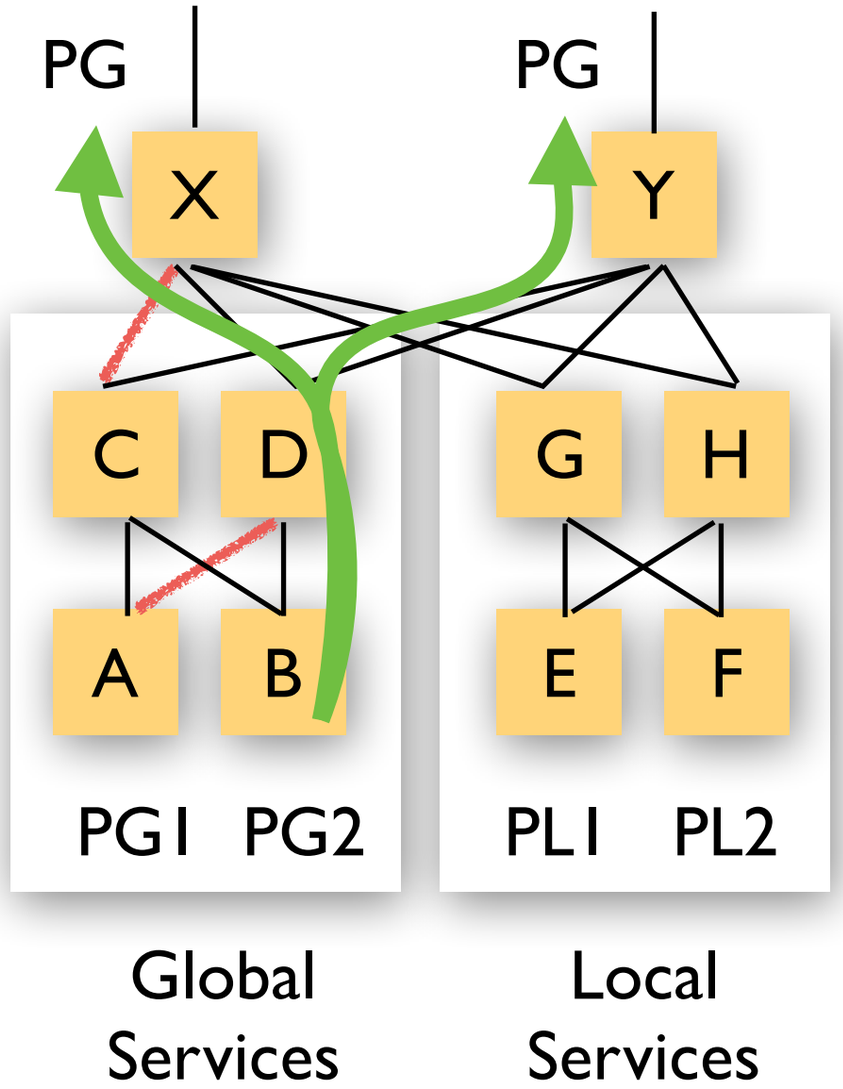
A Data Center Network

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X,Y do not need to know which prefixes are local vs global
  - aggregate to PG if announce is subset of PG
  - disallow "valley" paths

Consider D-A, X-C Failure:

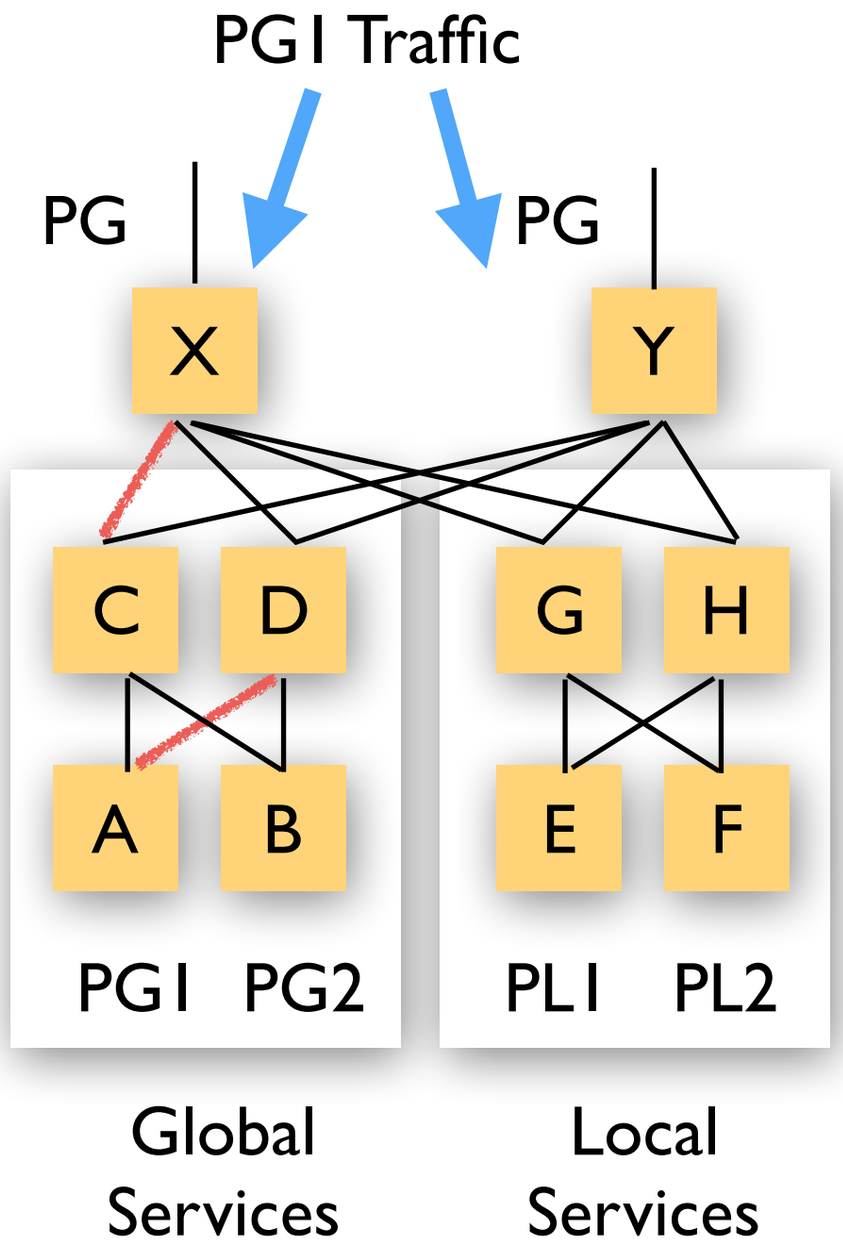
- X and Y will hear PG2
- X and Y will announce aggregate PG



A Data Center Network

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X,Y do not need to know which prefixes are local vs global
- aggregate to PG if announce is subset of PG
- disallow "valley" paths



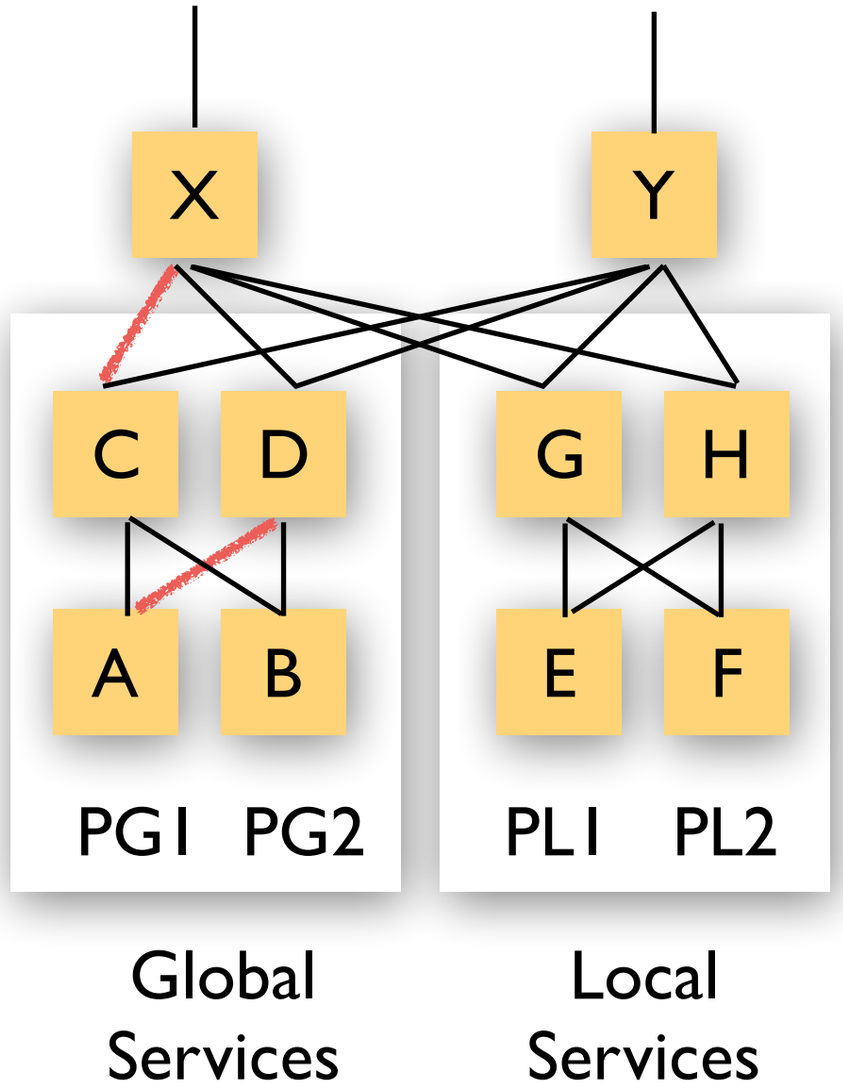
Consider D-A, X-C Failure:

- X and Y will hear PG2
- X and Y will announce aggregate PG
- But PG1 is inaccessible through X because there is no valley routing
- An aggregation-induced black hole is created [See Le et al, CoNext '11]

A Data Center Network

Implementation Techniques for X, Y:

- do export announce's from C, D outside
- do *not* export announce's from G,H outside
  - appeal: X,Y do not need to know which prefixes are local vs global
  - aggregate to PG if announce is subset of PG
  - disallow "valley" paths



Moral:

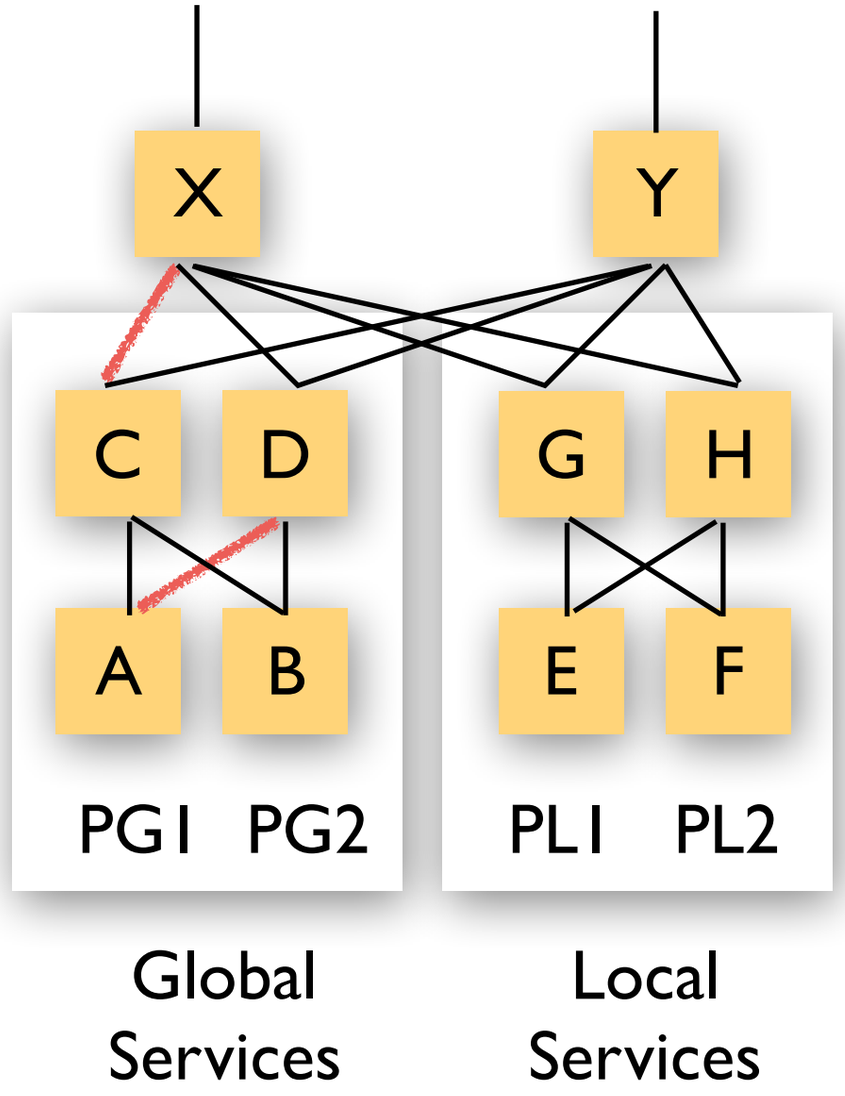
- Reasoning about the interactions between failures and the device-level configs is hard
- Not a good idea for humans to be doing that

A Data Center Network

P1: Traffic for prefixes for each TOR router must reach the router

P2: Do not announce local services externally

P3: Aggregation must be performed on global prefixes to reduce churn



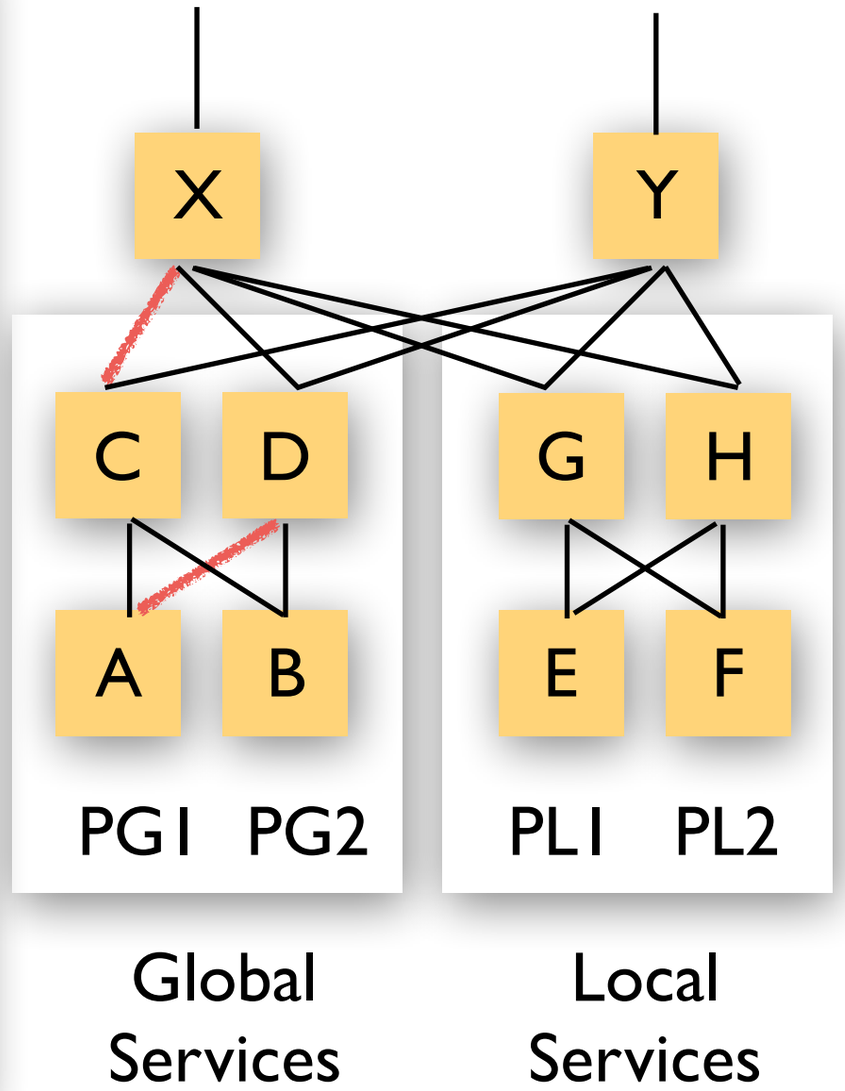
A Data Center Network

P1: Traffic for prefixes for each TOR router must reach the router

P2: Do not announce local services externally

P3: Aggregation must be performed on global prefixes to reduce churn

```
define Ownership = {
   PG1 => end(A)
   PG2 => end(B)
   PL1 => end(E)
   PL2 => end(F)
}
```



A Data Center Network

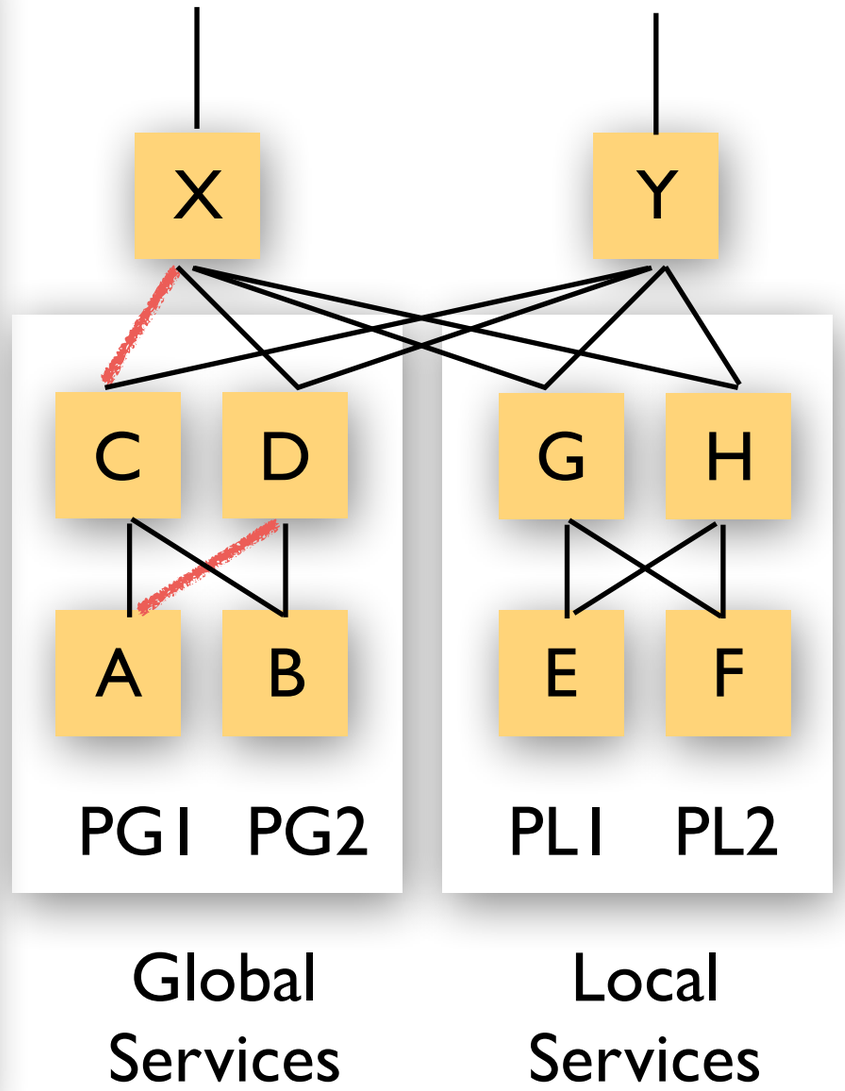
P1: Traffic for prefixes for each TOR router must reach the router

P2: Do not announce local services externally

P3: Aggregation must be performed on global prefixes to reduce churn

```
define Ownership = {
   PG1 => end(A)
   PG2 => end(B)
   PL1 => end(E)
   PL2 => end(F)
}

define Routing = {
   PL1 | PL2 => always(in)
}
```



A Data Center Network

P1: Traffic for prefixes for each TOR router must reach the router

P2: Do not announce local services externally

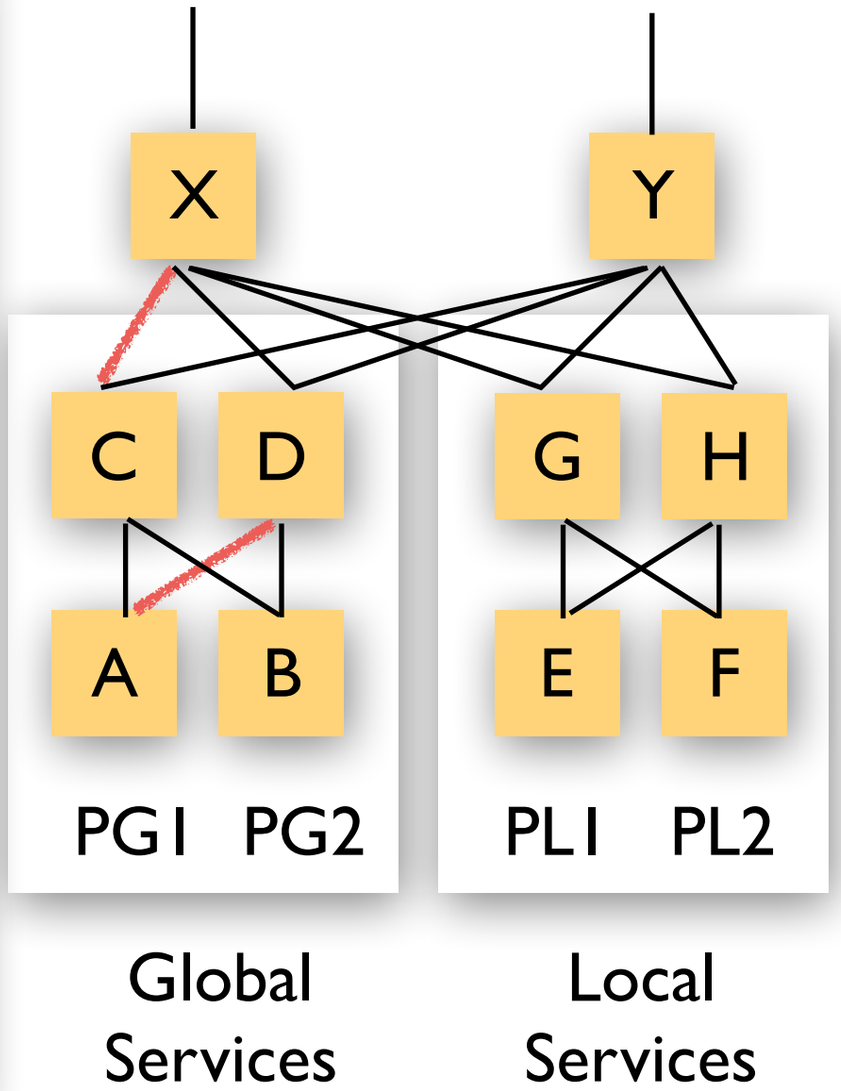
P3: Aggregation must be performed on global prefixes to reduce churn

```
define Ownership = {
   PG1 => end(A)
   PG2 => end(B)
   PL1 => end(E)
   PL2 => end(F)
}

define Routing = {
   PL1 | PL2 => always(in)
}

define Main =
   Ownership & Routing &
      agg(PG, in -> out)
```

compiler discovers lower bound on # failures required to induce a black hole



Propane Language Summary

Programmers express high-level objectives via:

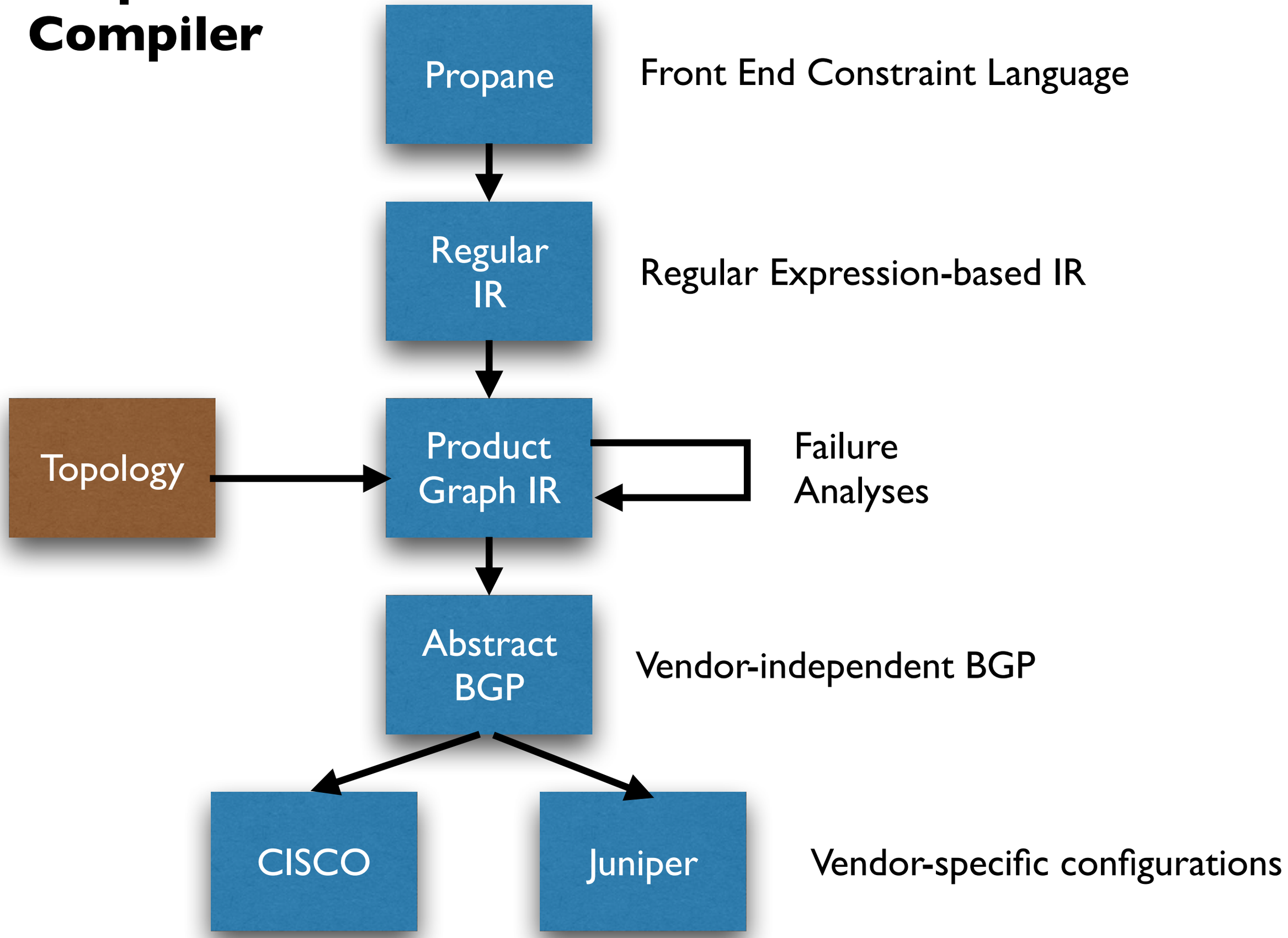
- Sets of inter- and intra-domain paths
- … defined by high-level abstractions: **enter**, **exit**, **always**, **end**, …
- … as well as preferences between those paths for failures

Programmers may also use:

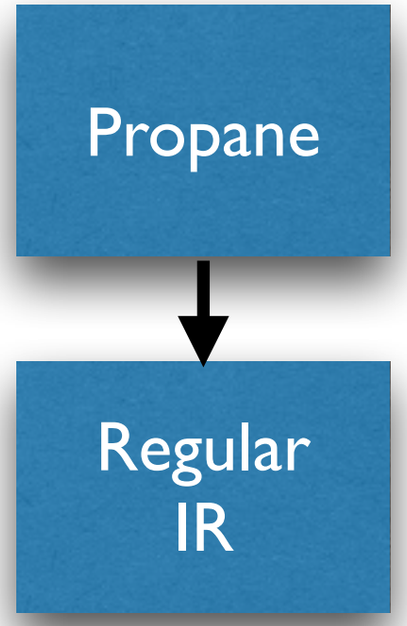
- a small number of control constraints (**agg**)
- … while counting on the compiler for failure analysis
- … and combine constraints together using natural logical combinators

Compiling Propane

Propane Compiler



Propane Regular IR



Expand constraints in to regular expressions. EG:

$$\mathbf{any} = \mathbf{out}^*.\mathbf{in}+.\mathbf{out}^*$$

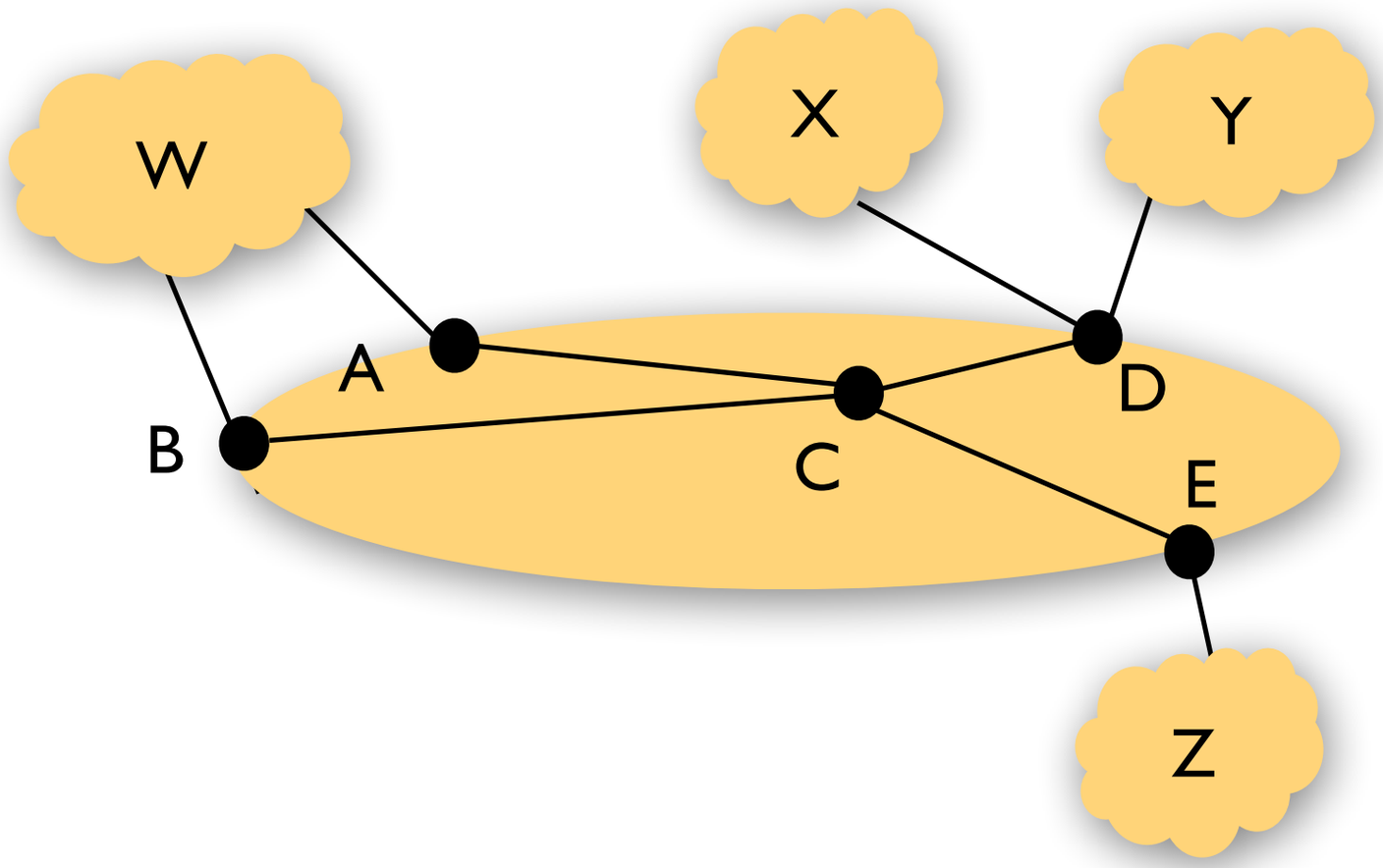
$$\mathbf{end}(X) = \mathbf{any} \cap (\Sigma^*.X)$$

$$\mathbf{exit}(X) = (\mathbf{out}^*.\mathbf{in}^*.(X \cap \mathbf{in}).\mathbf{out}+)\mathbf{U}$$
$$(\mathbf{out}^*.\mathbf{in}+.(X \cap \mathbf{out}).\mathbf{out}^*)$$

A few other simple transformations:

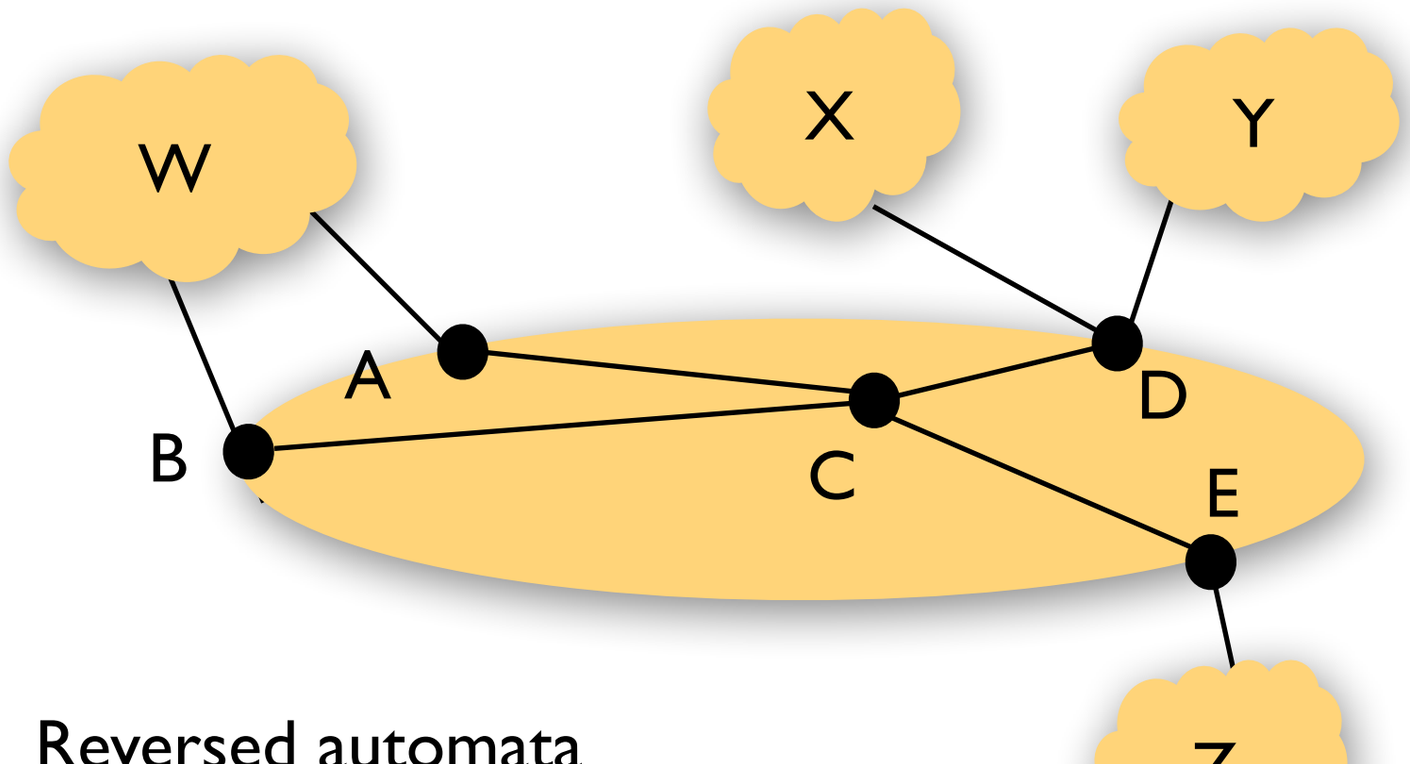
- conjunction of constraints ==> intersection of regular expressions
- conjunction of policies ==> prefix-by-prefix intersection
- nested preferences lifted: (x >> y) . z ==> (x.z) >> (y.z)

Compilation: An Idealized Example




```
(W.A.C.D.out) >> (W.B.in+.out)
```

Reversed Automata from Policies

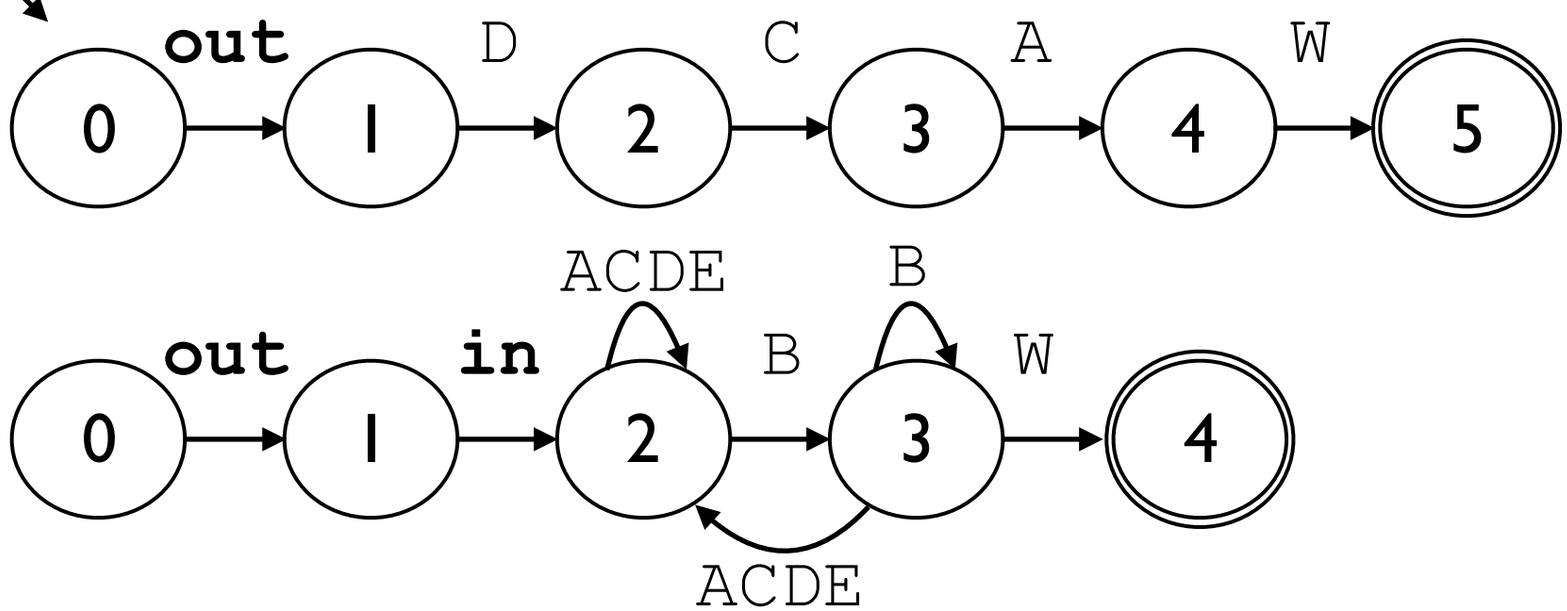


Reversed automata
tracks BGP message flow

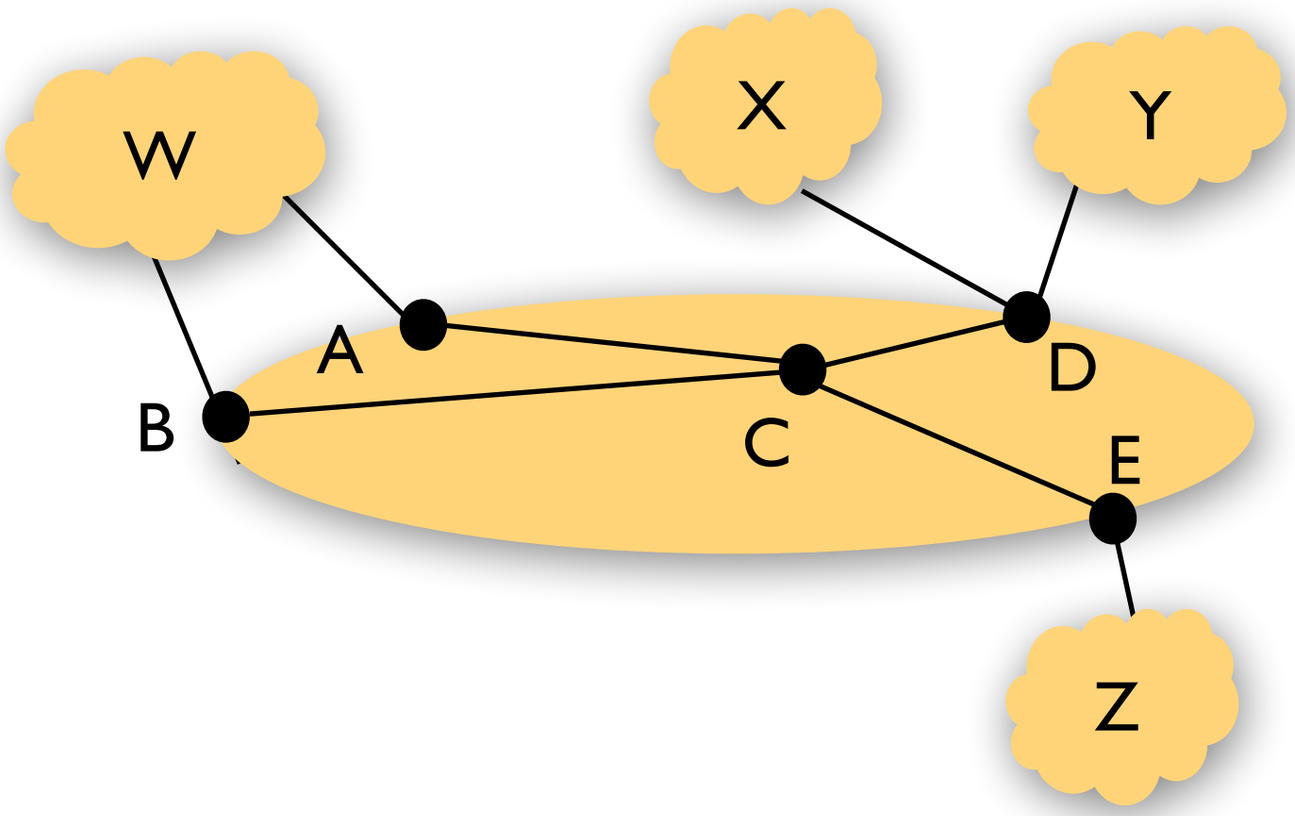
Policies

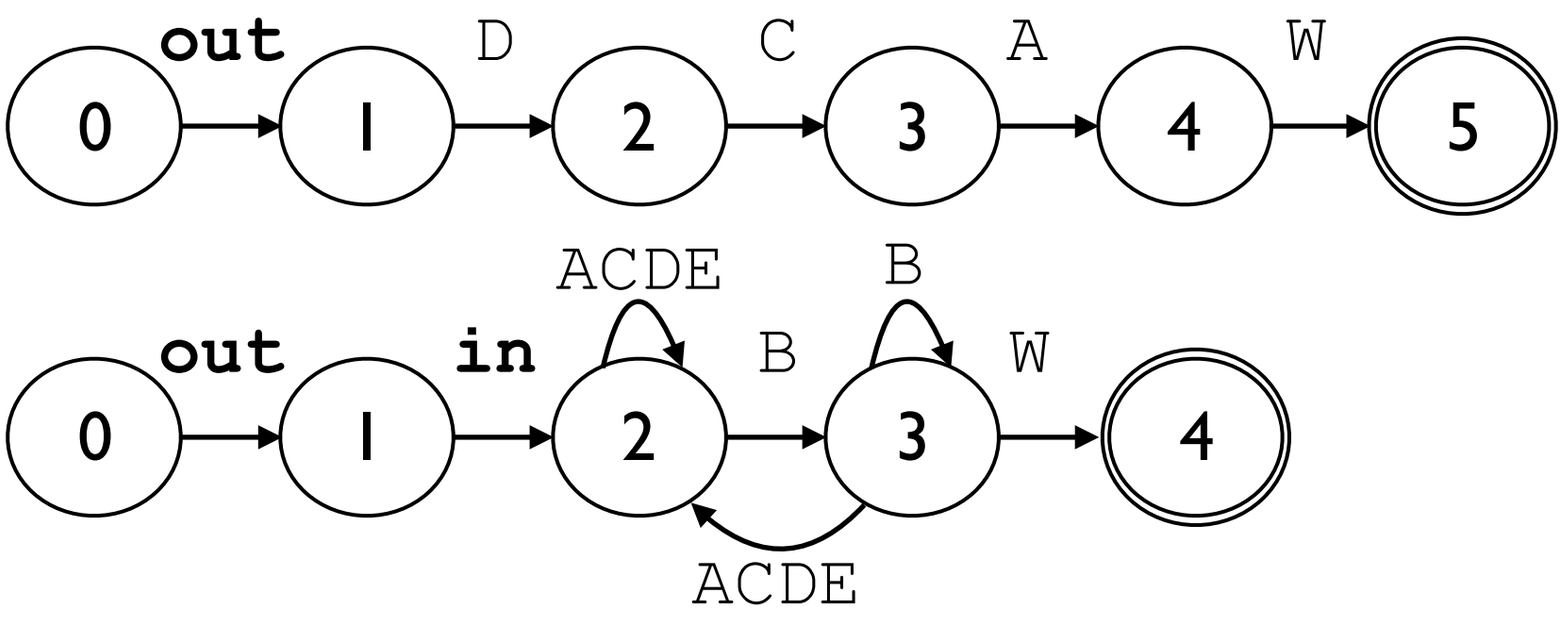
1. (W.A.C.D.**out**)

2. (W.B.**in**+.**out**)



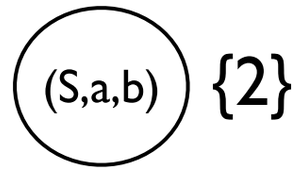
Constructing the Product Graph (PG)





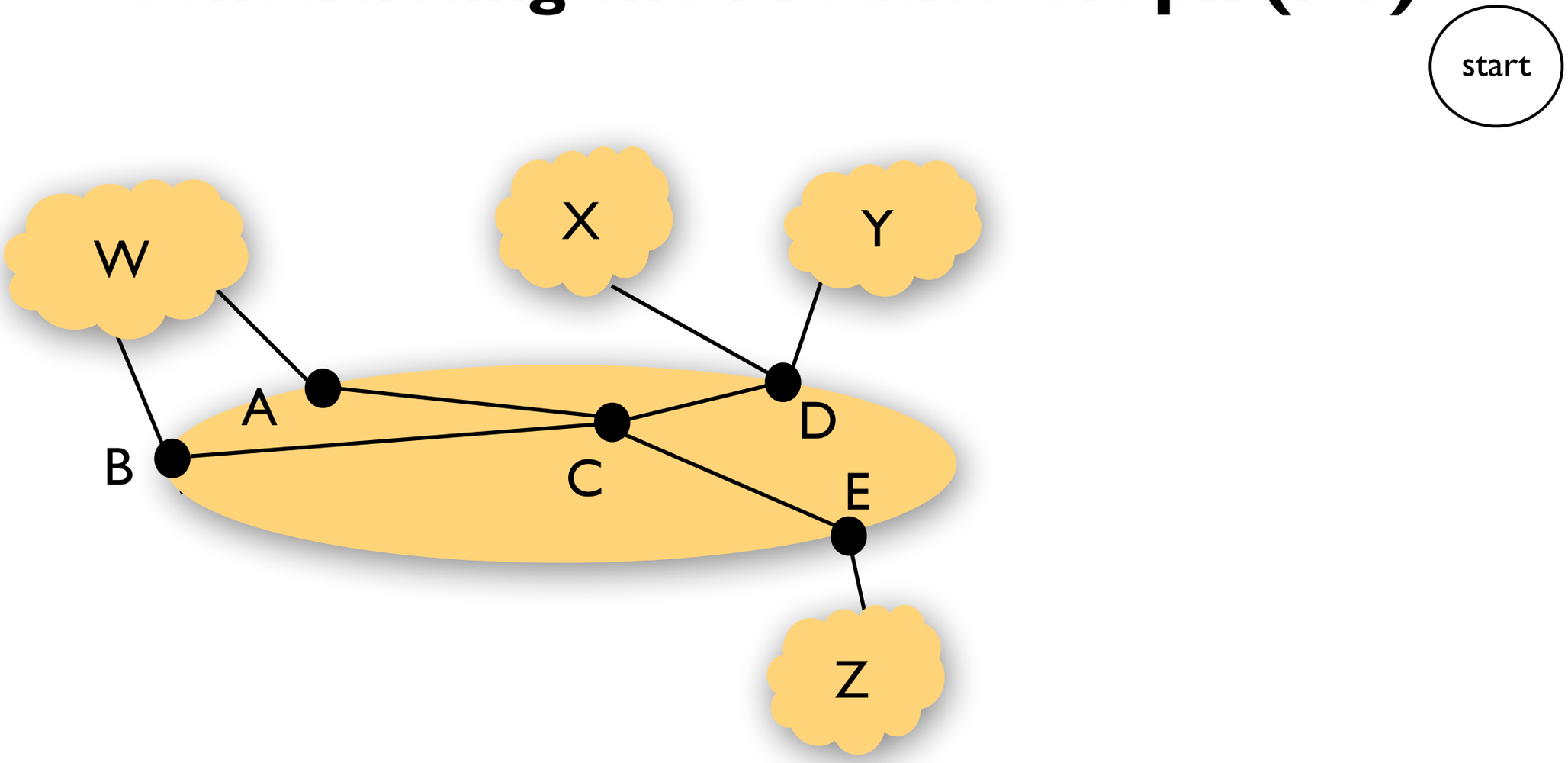
General Idea:
PG represents locations reachable in the topology while following the policy

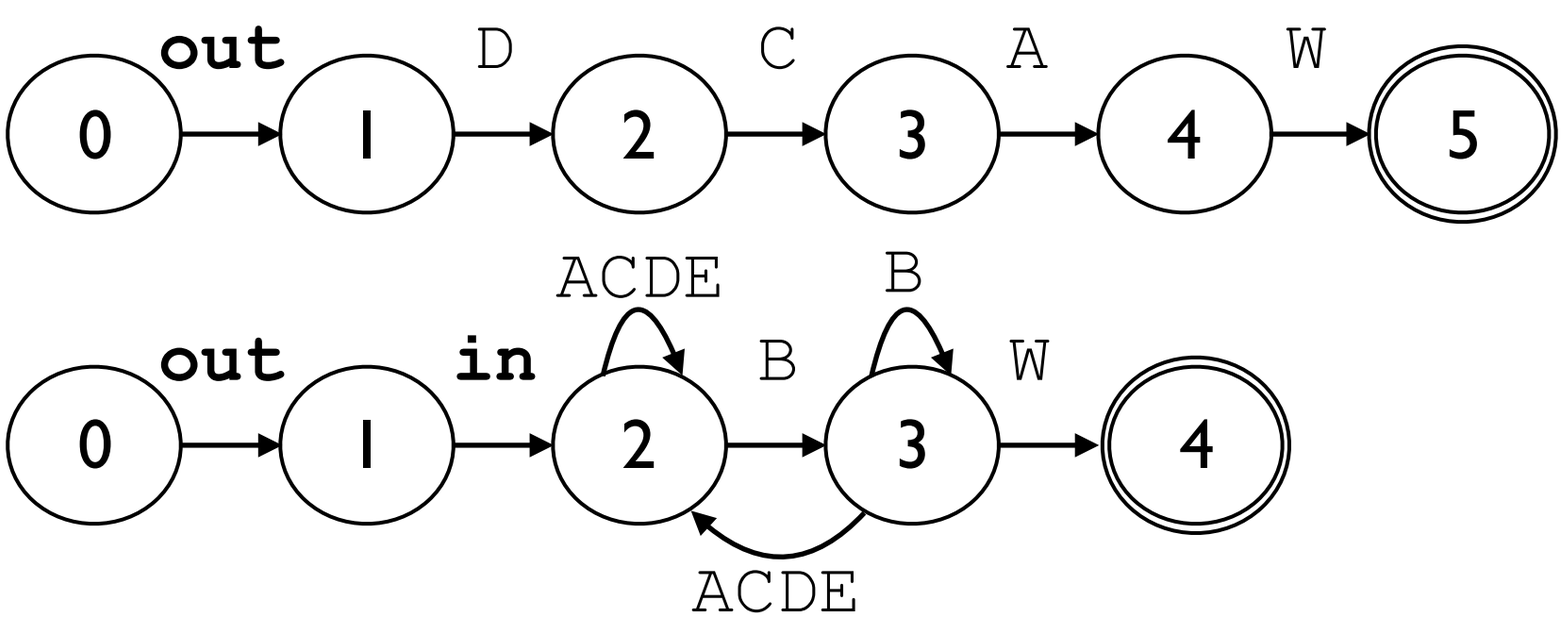
Each PG node contains:
- topology node (S)
- state of automaton 1 (a)
- state of automaton 2 (b)
- set of preferences achieved



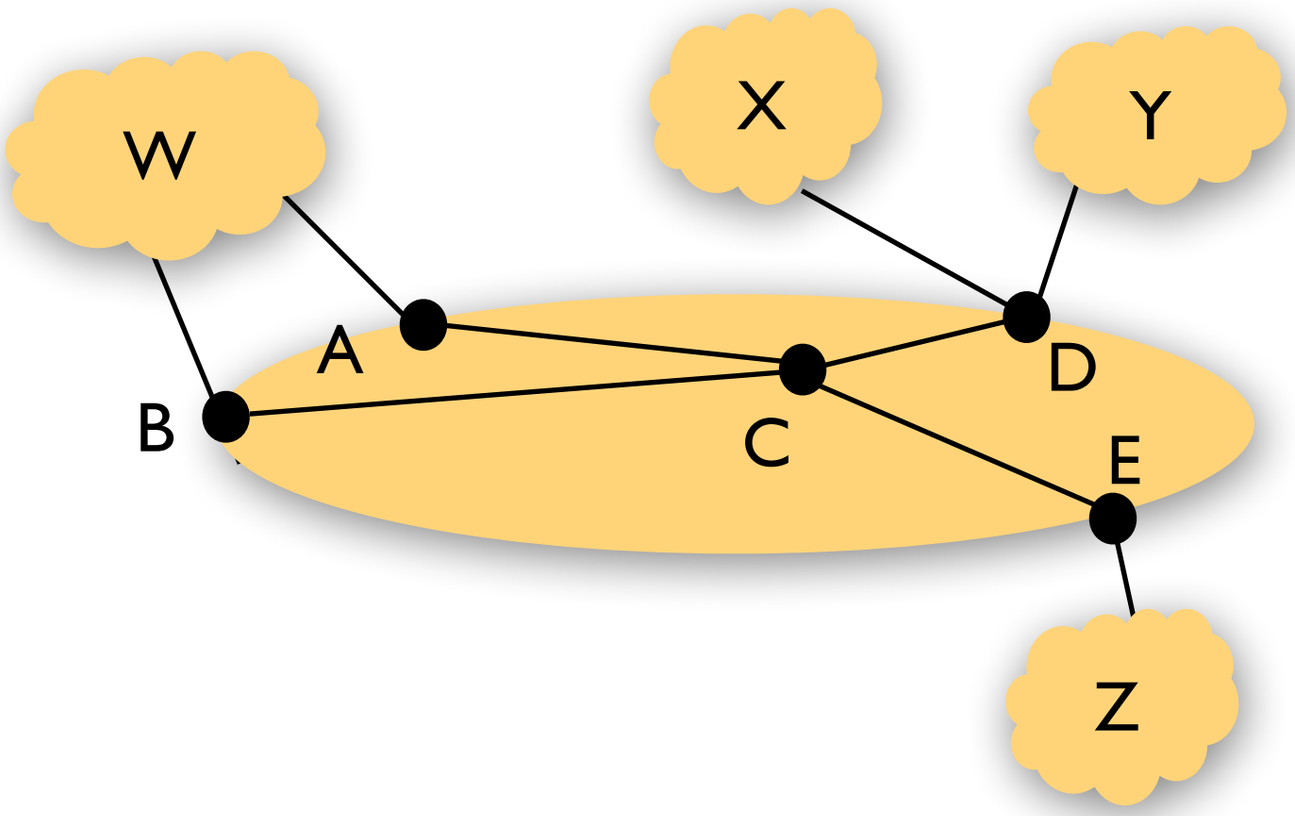
Two PG nodes are connected if topology nodes are connected and the automata make the specified transition

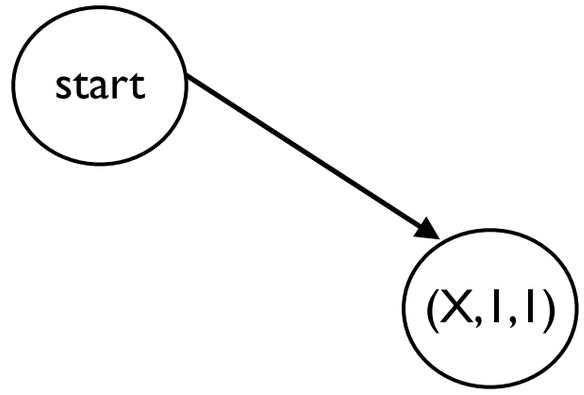
Constructing the Product Graph (PG)

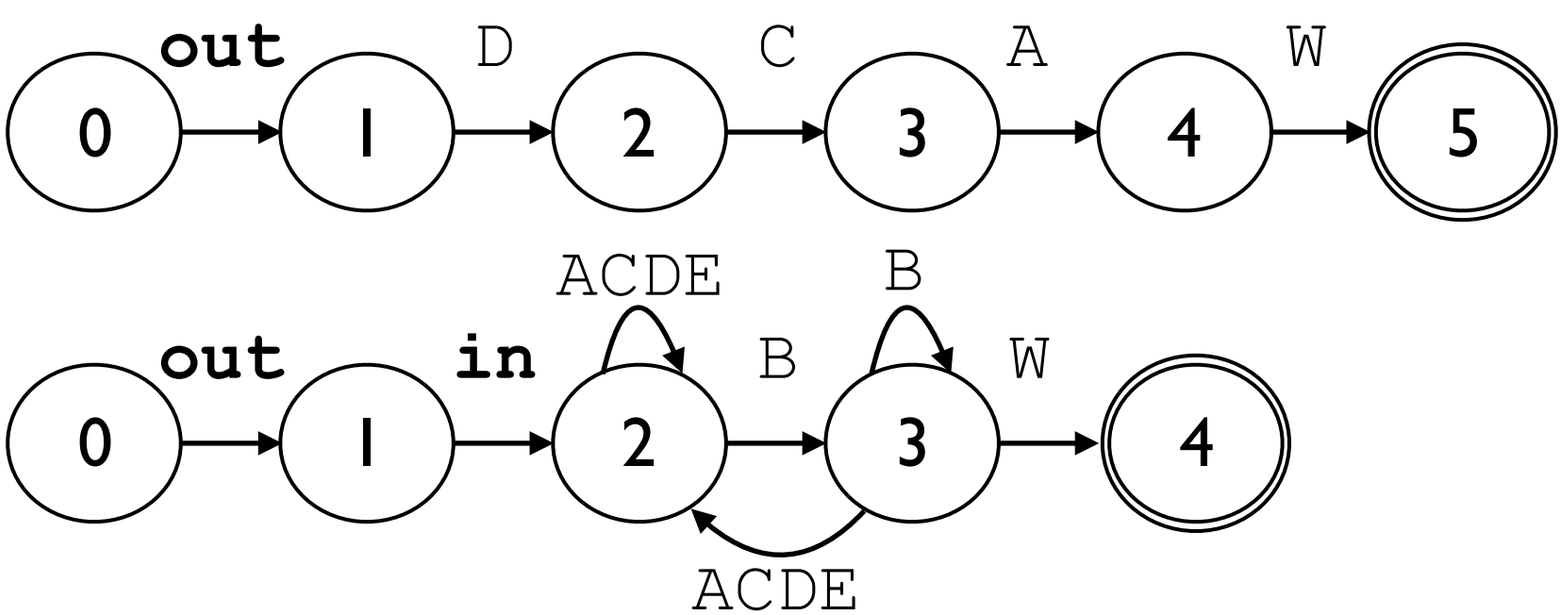




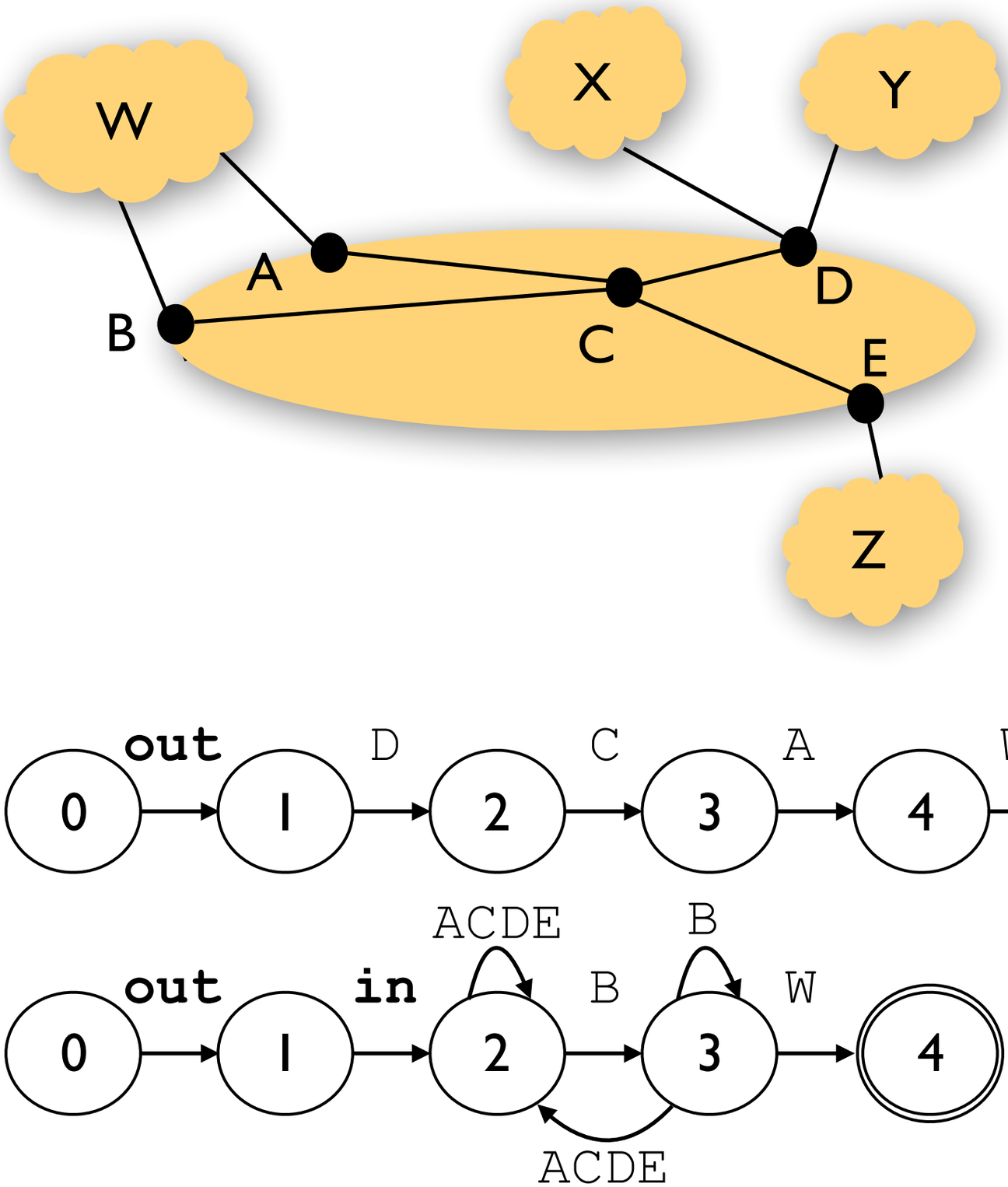
Constructing the Product Graph (PG)

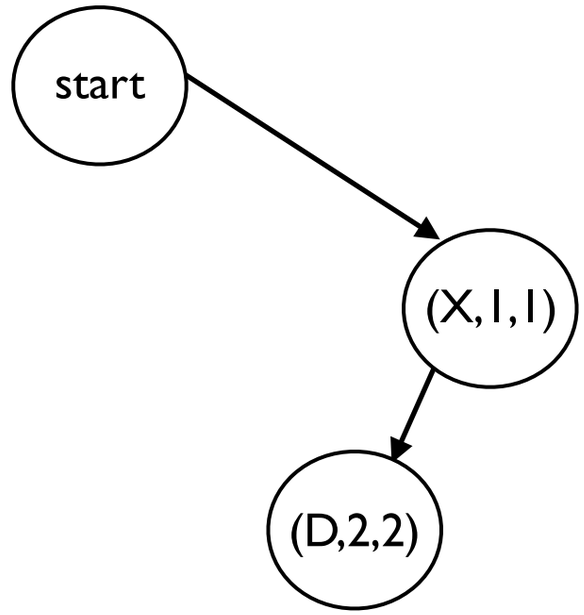




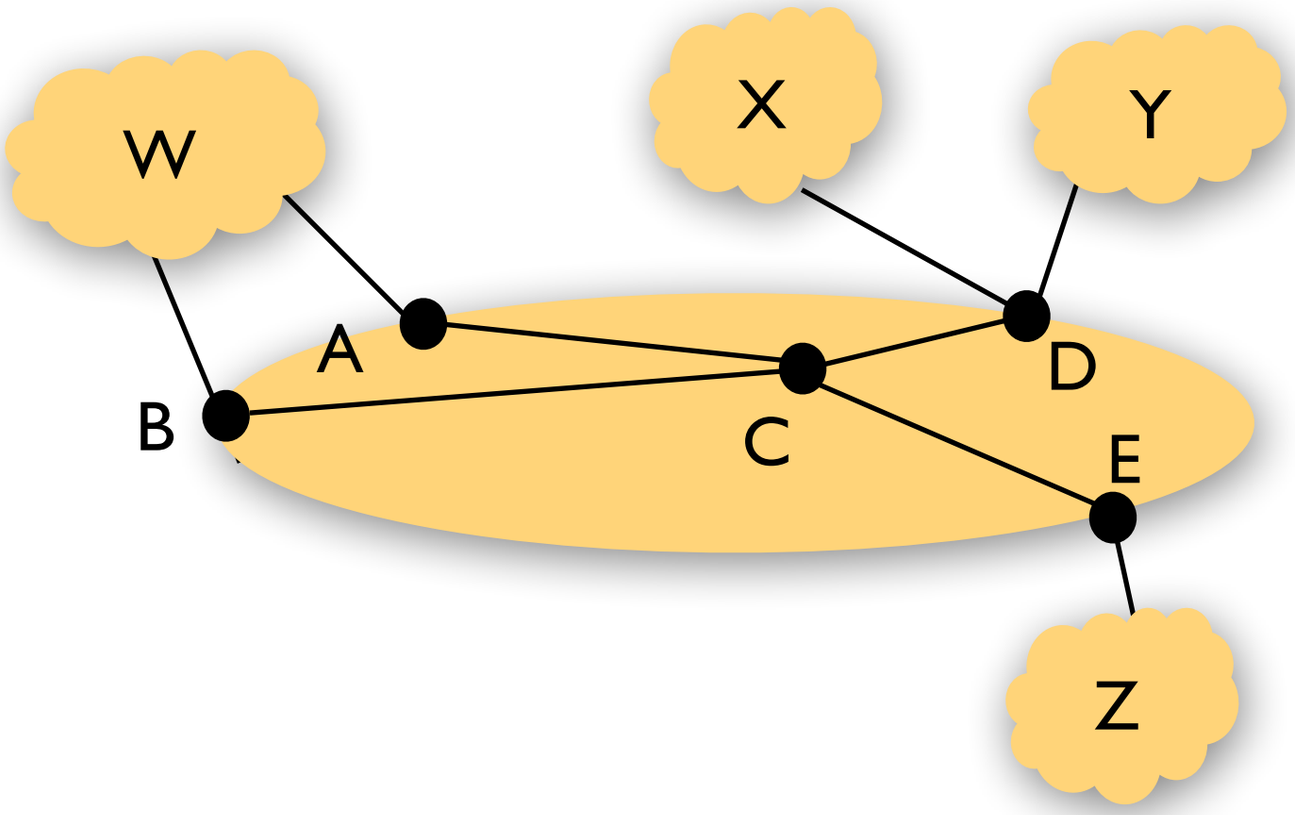


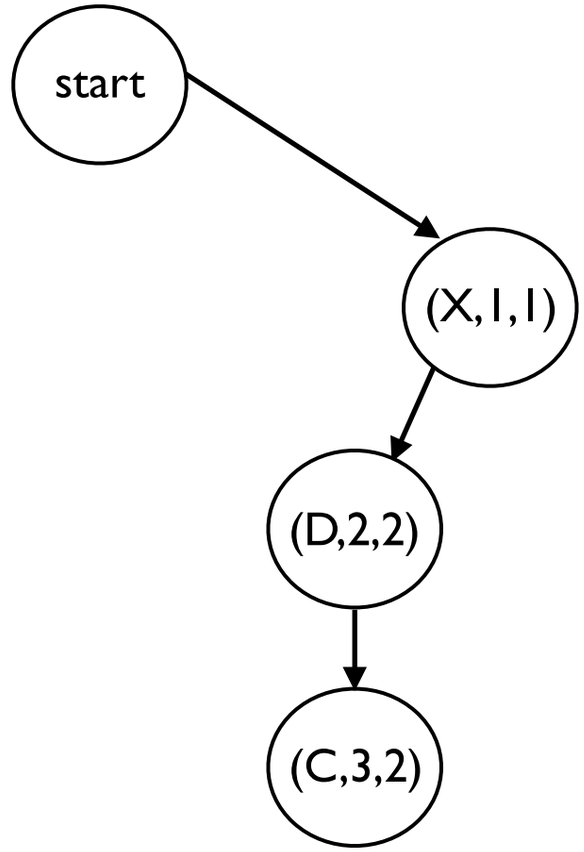
Constructing the Product Graph (PG)

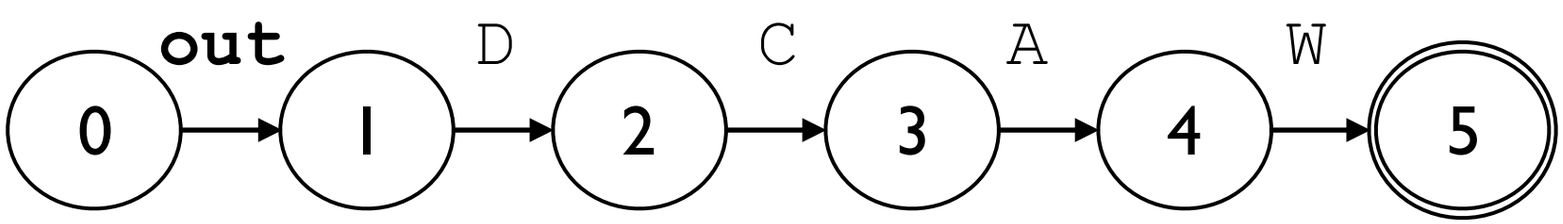


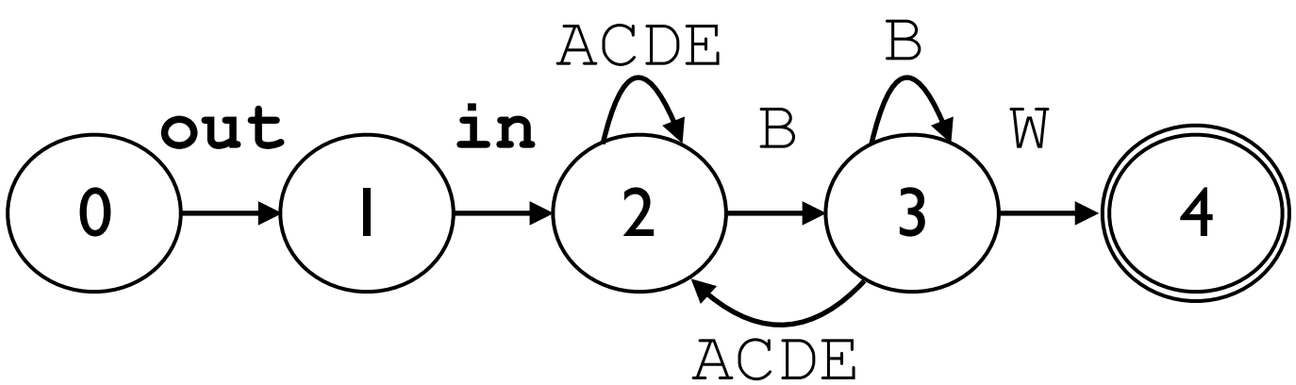


Constructing the Product Graph (PG)

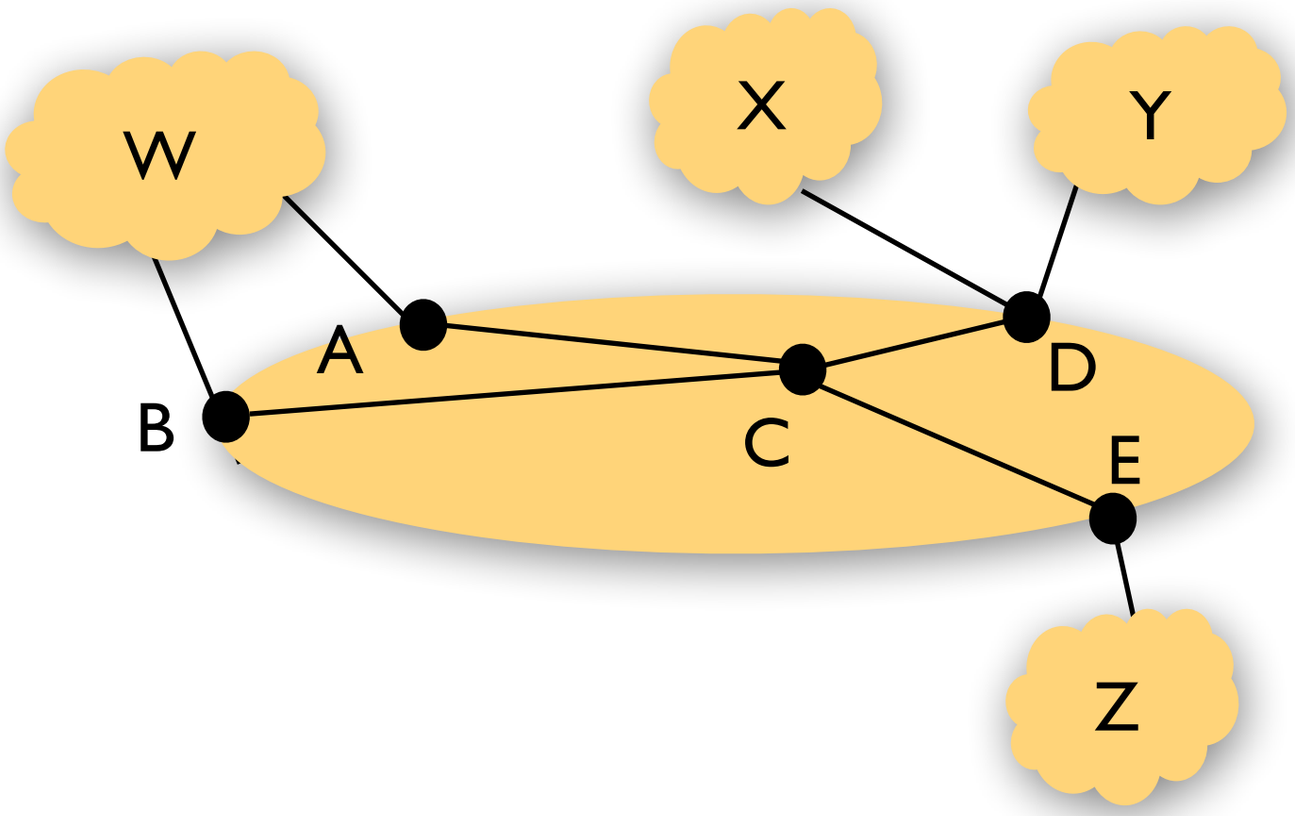


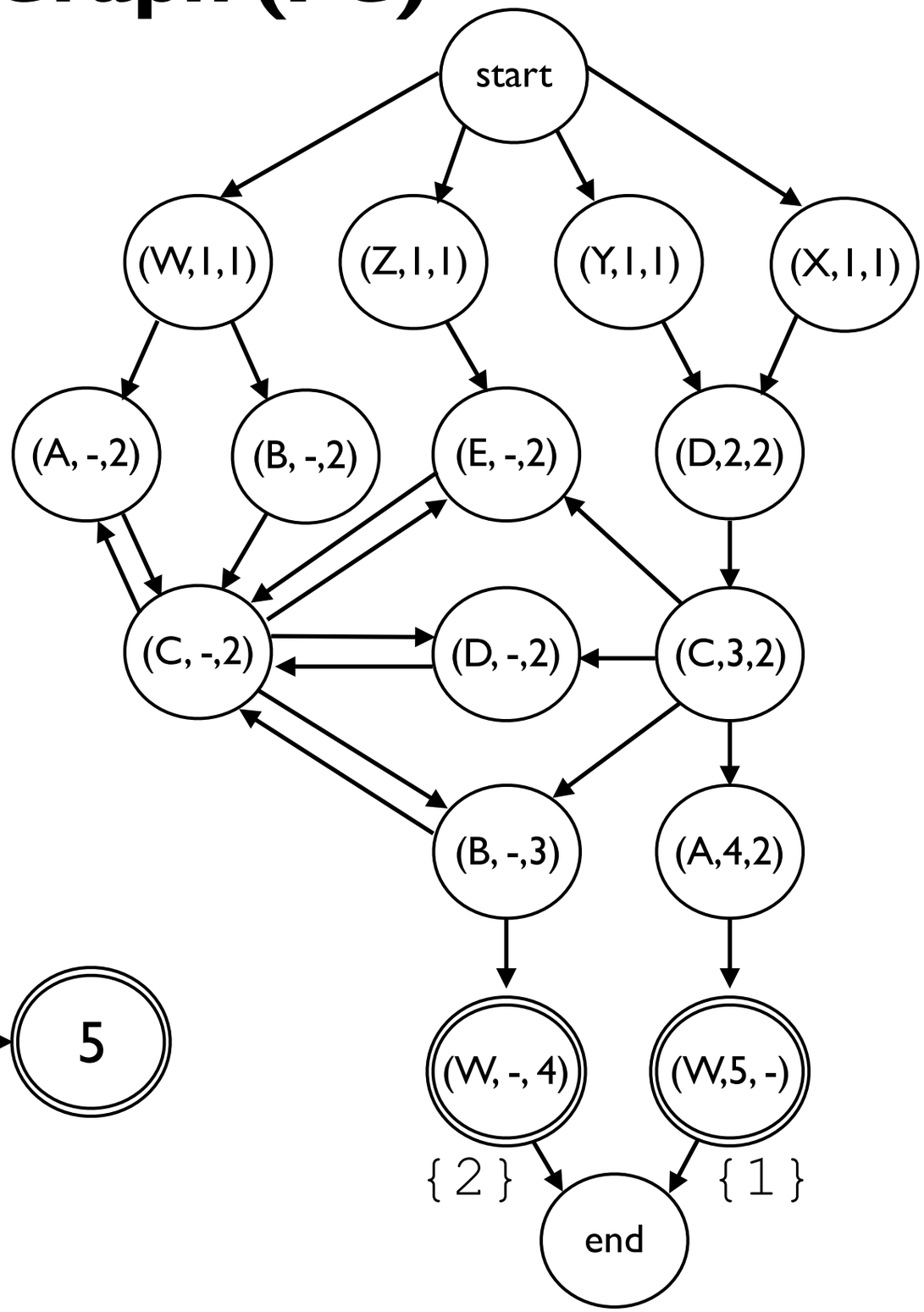


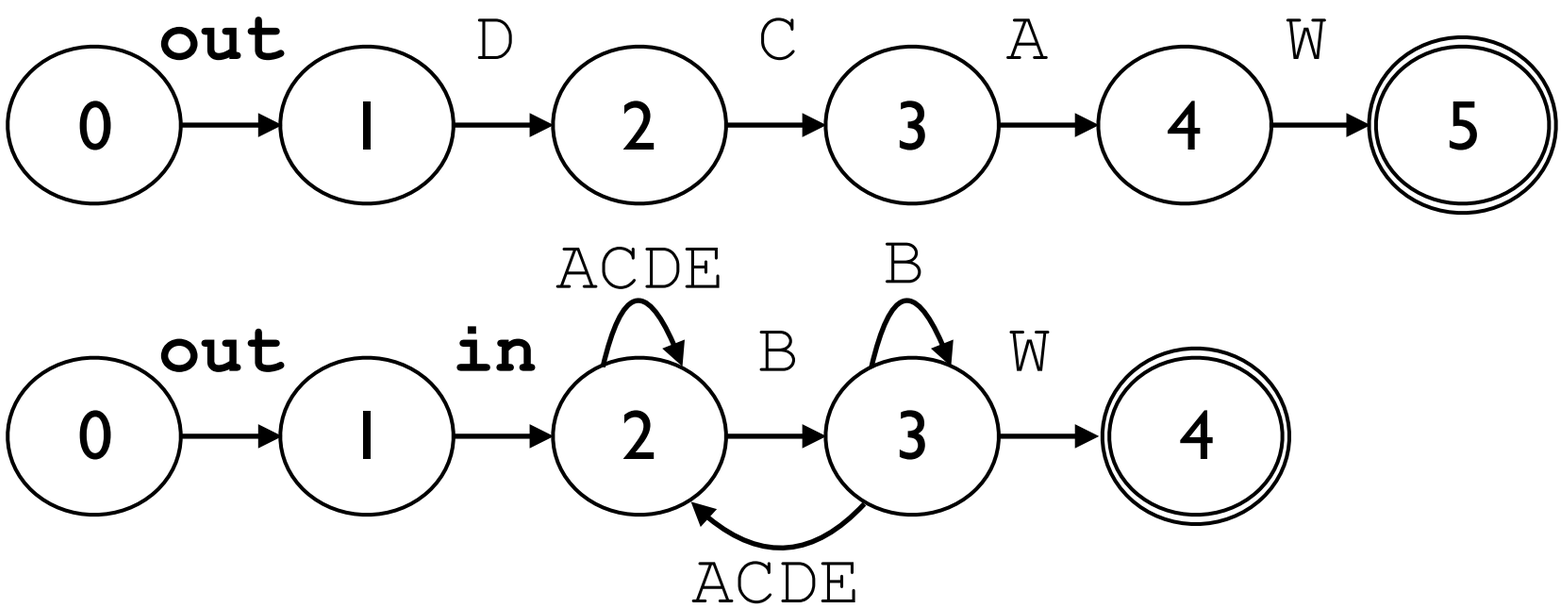




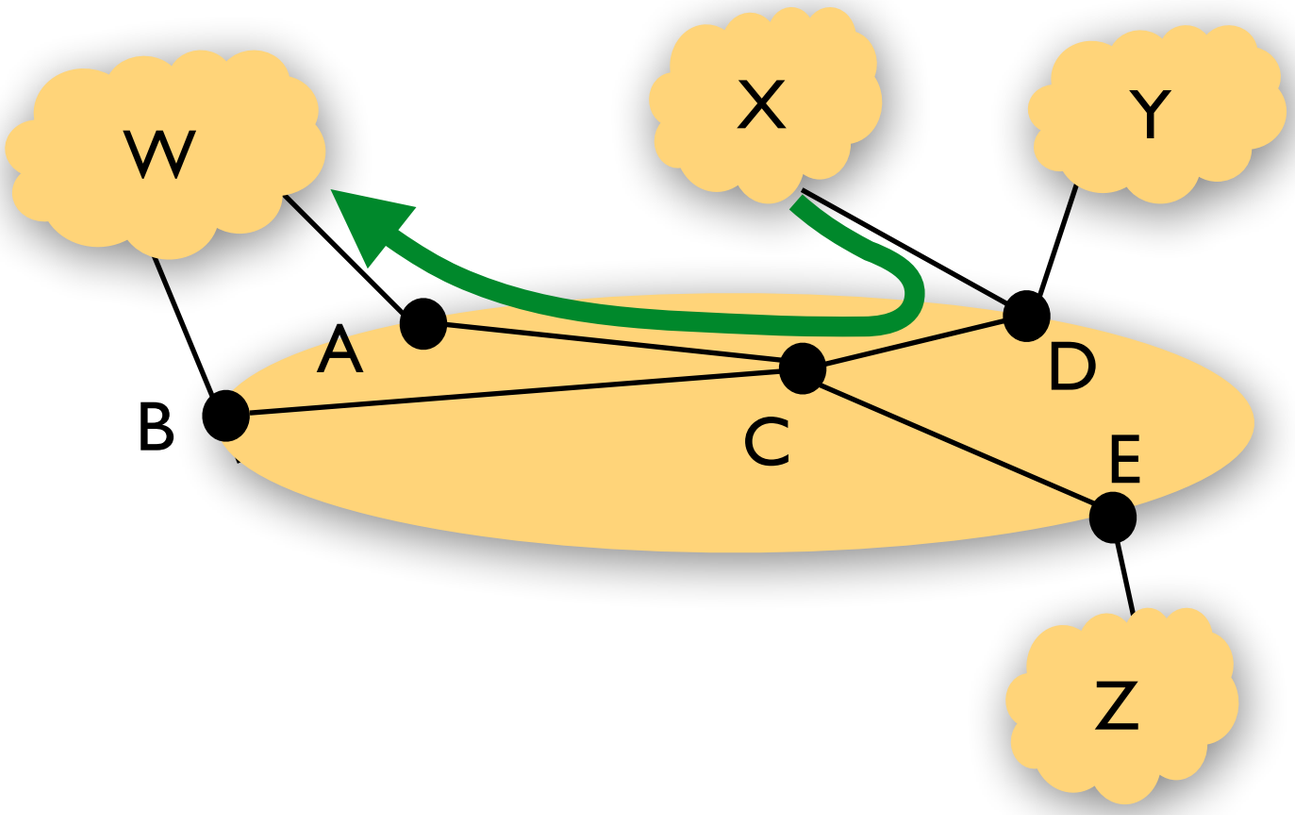
Constructing the Product Graph (PG)

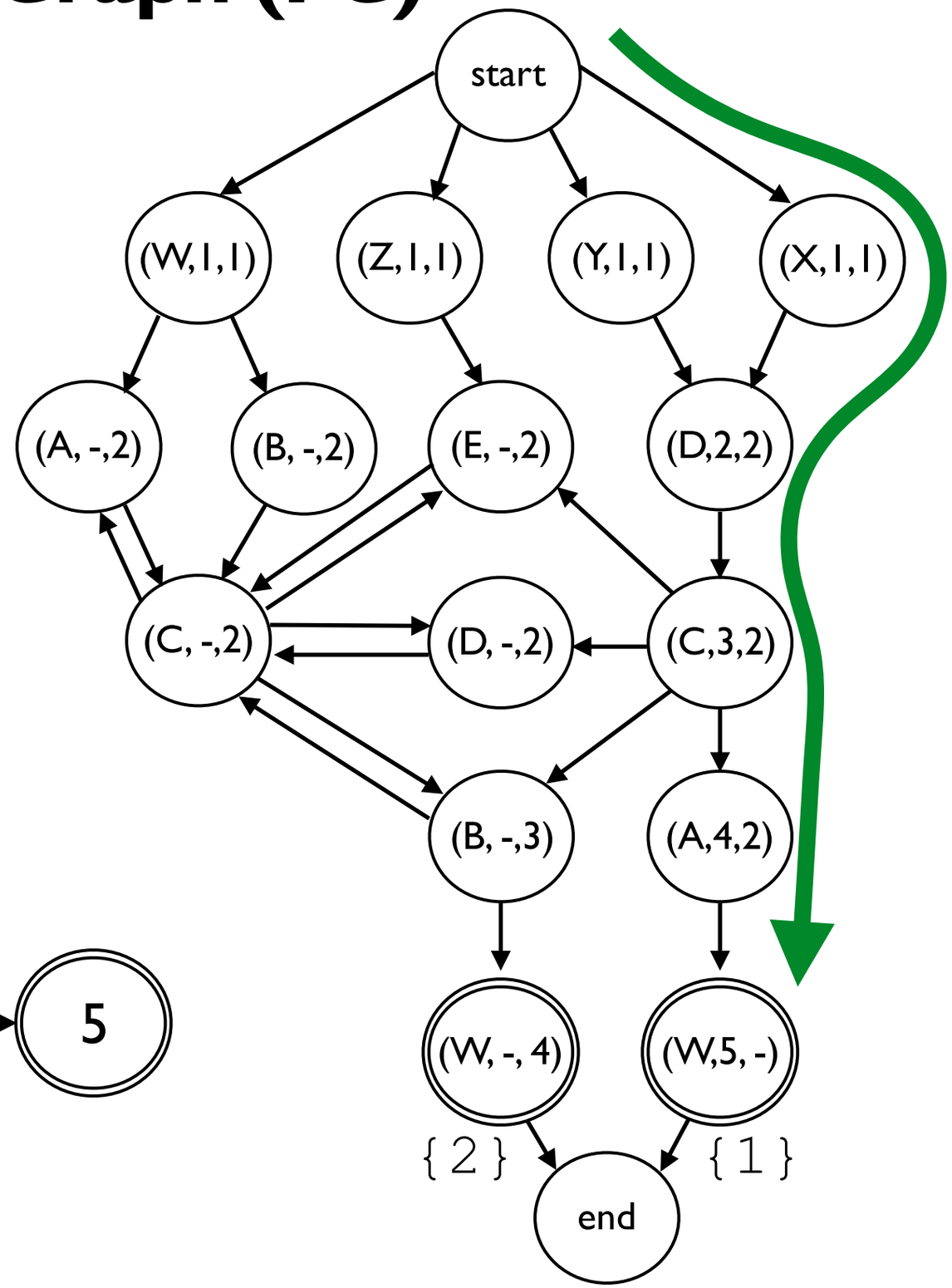


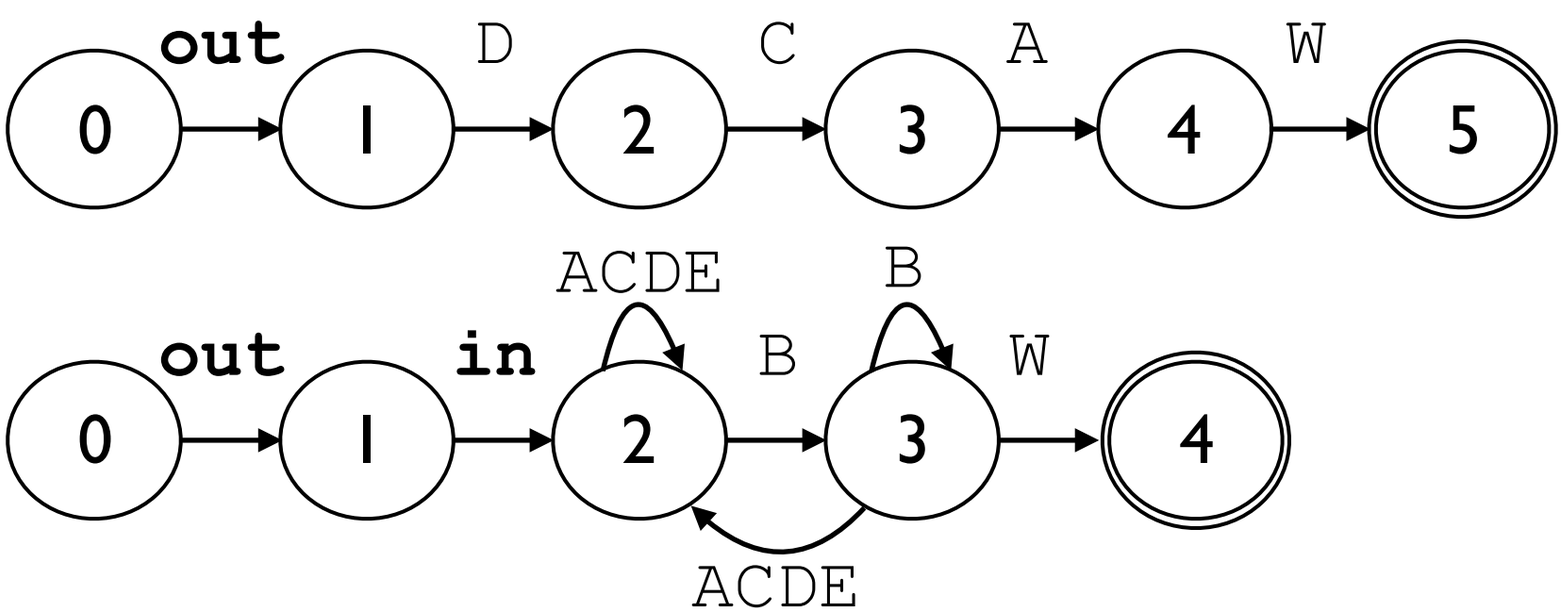




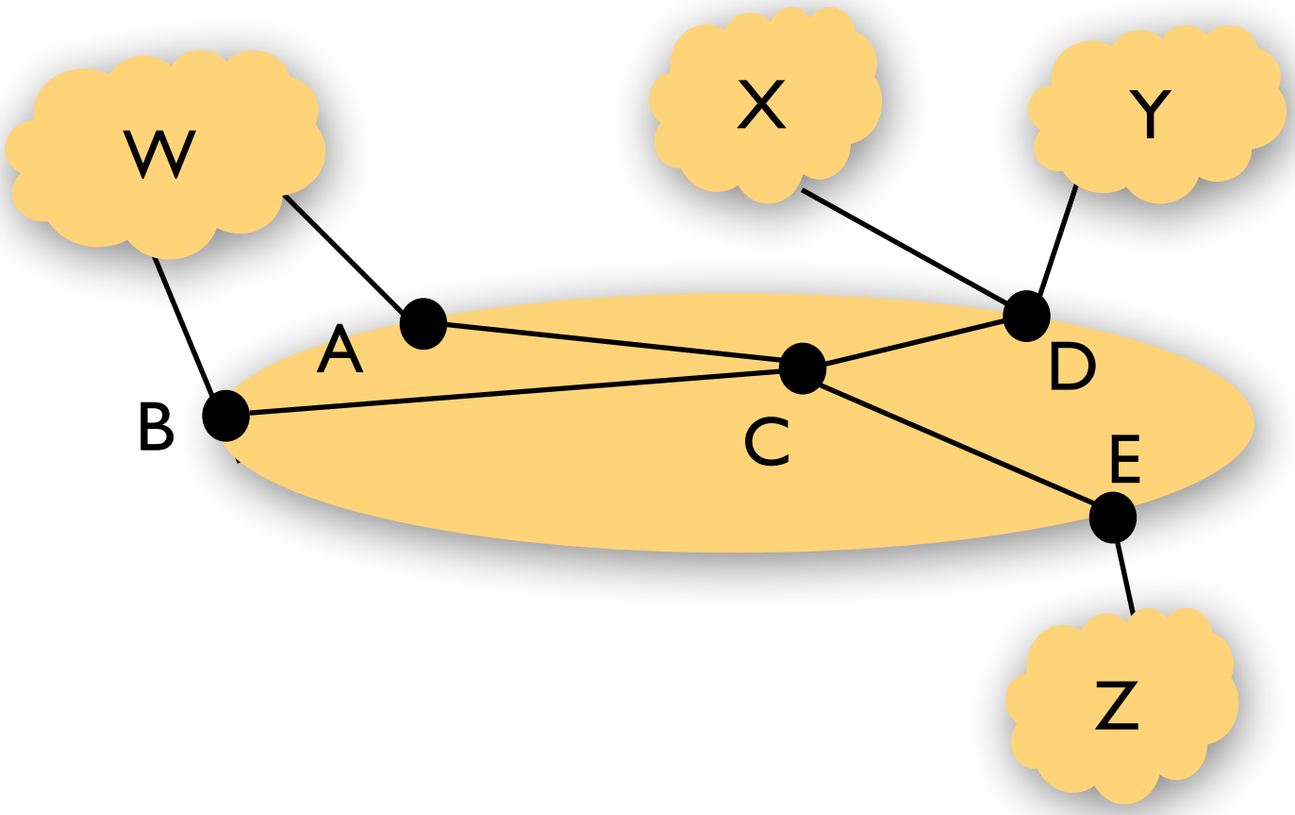
Constructing the Product Graph (PG)

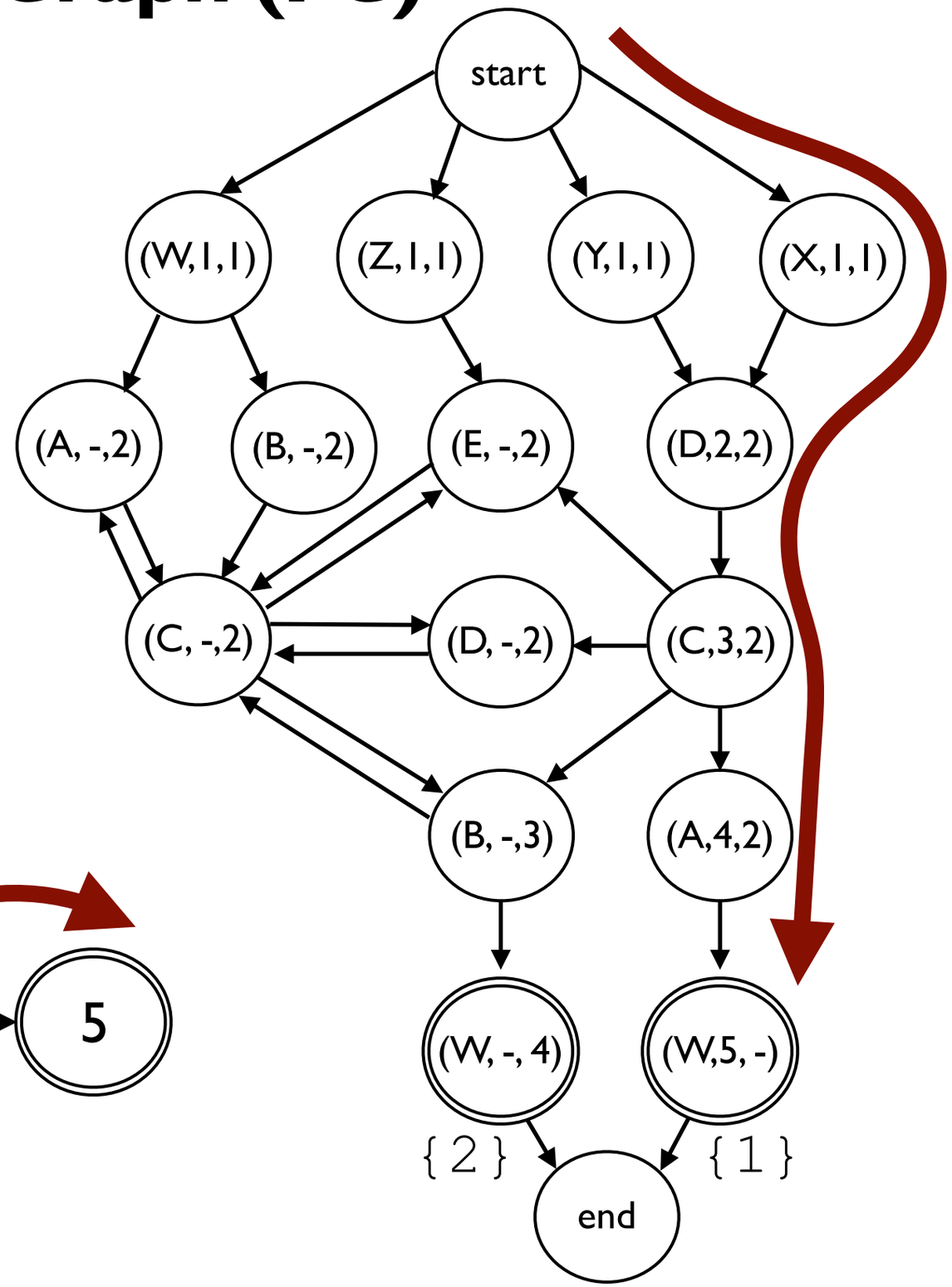


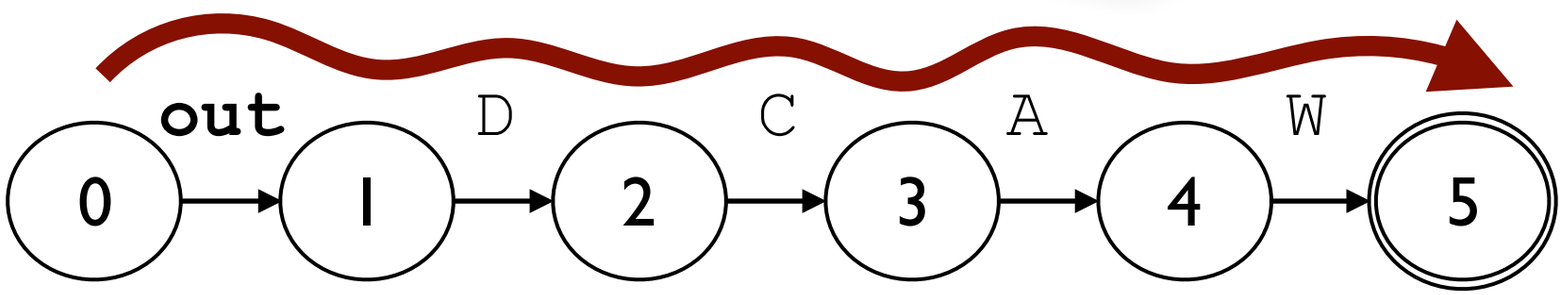


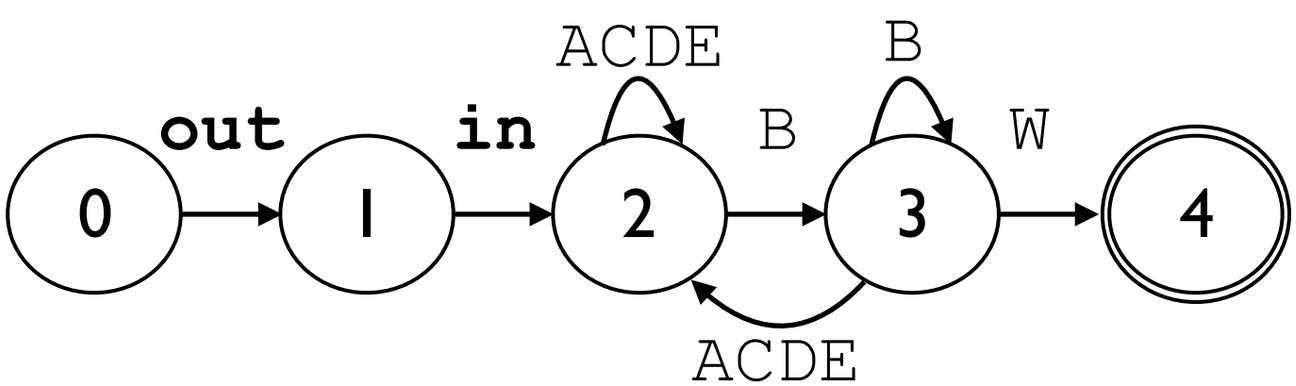


Constructing the Product Graph (PG)

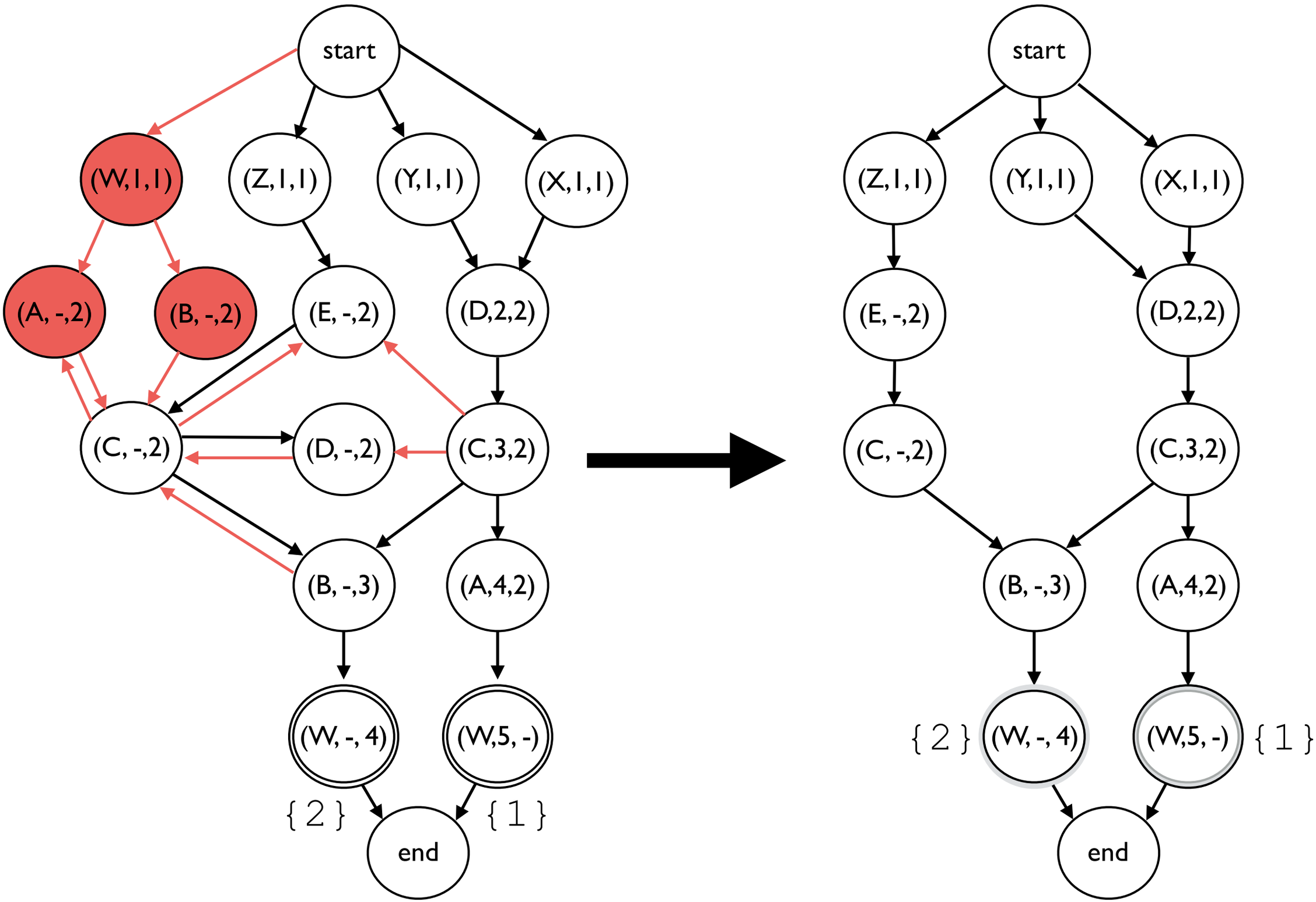




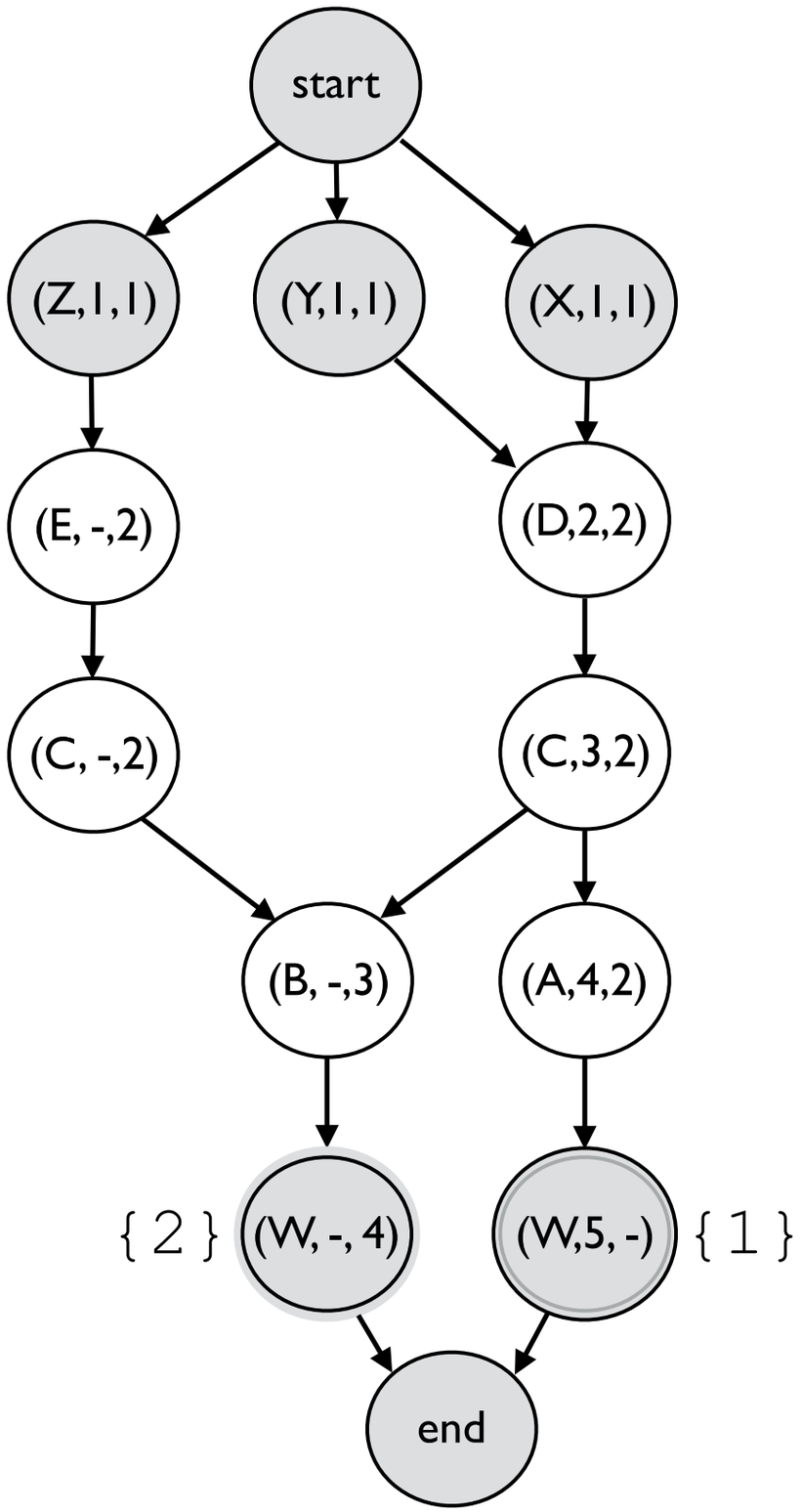




Product Graph Mimization



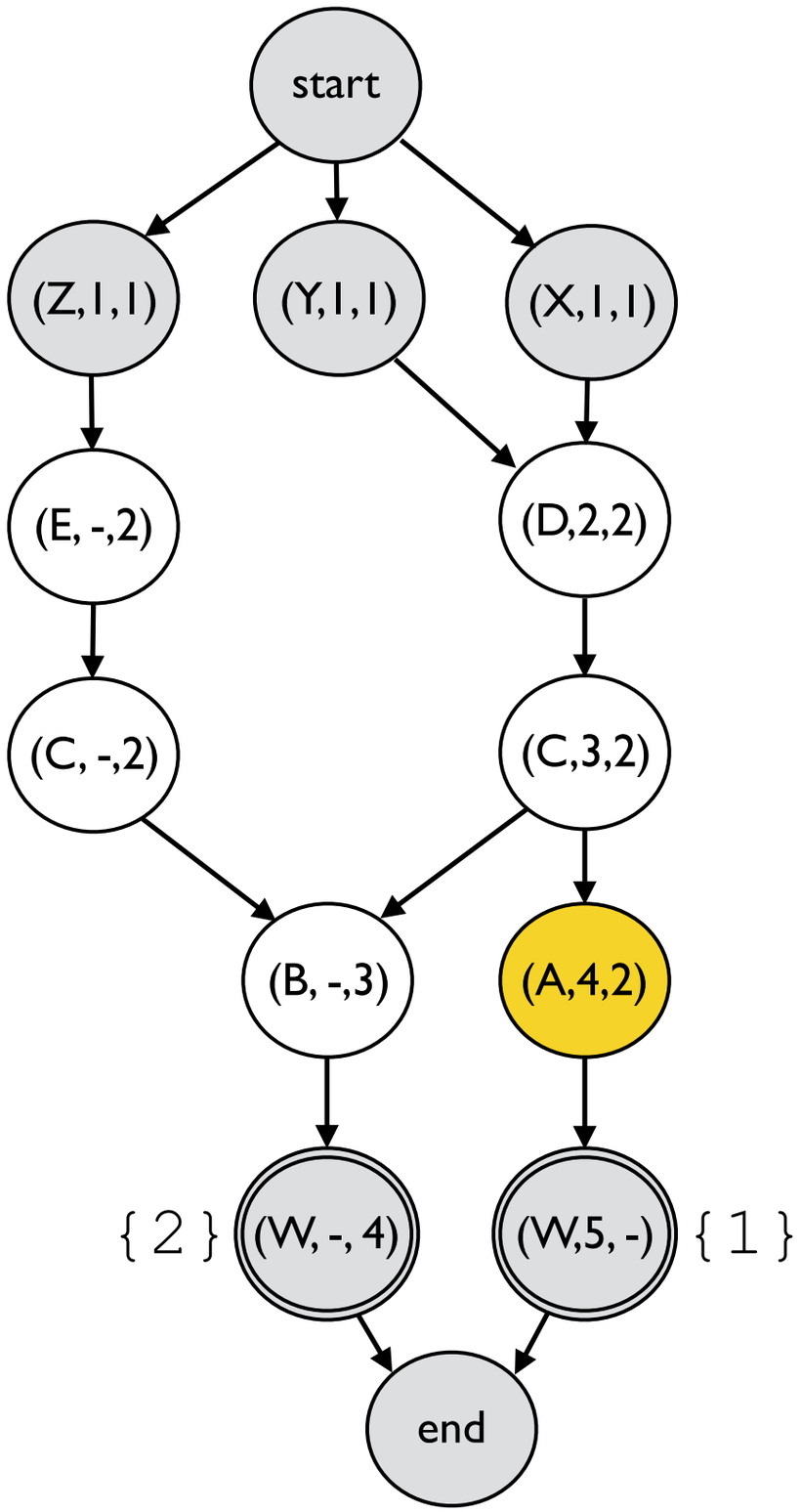
Compilation to BGP:



Idea:

- Filter import messages according to incoming PG edges.
- For each internal location, decide which announcements to prefer, forward messages along PG edges
- Use a community value to tag the state of the automata
- For each external location, do nothing

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),

comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,2),

comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

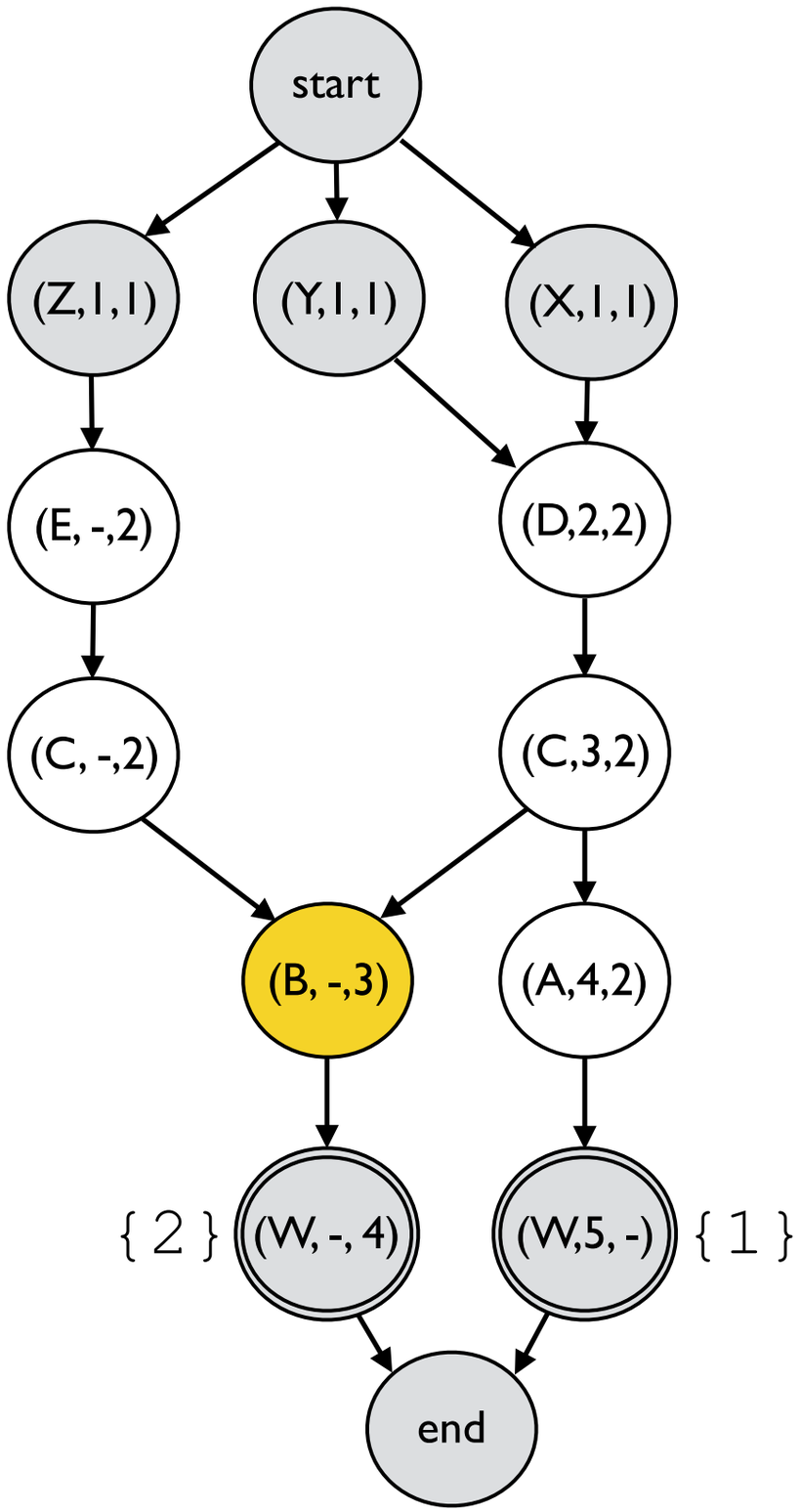
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

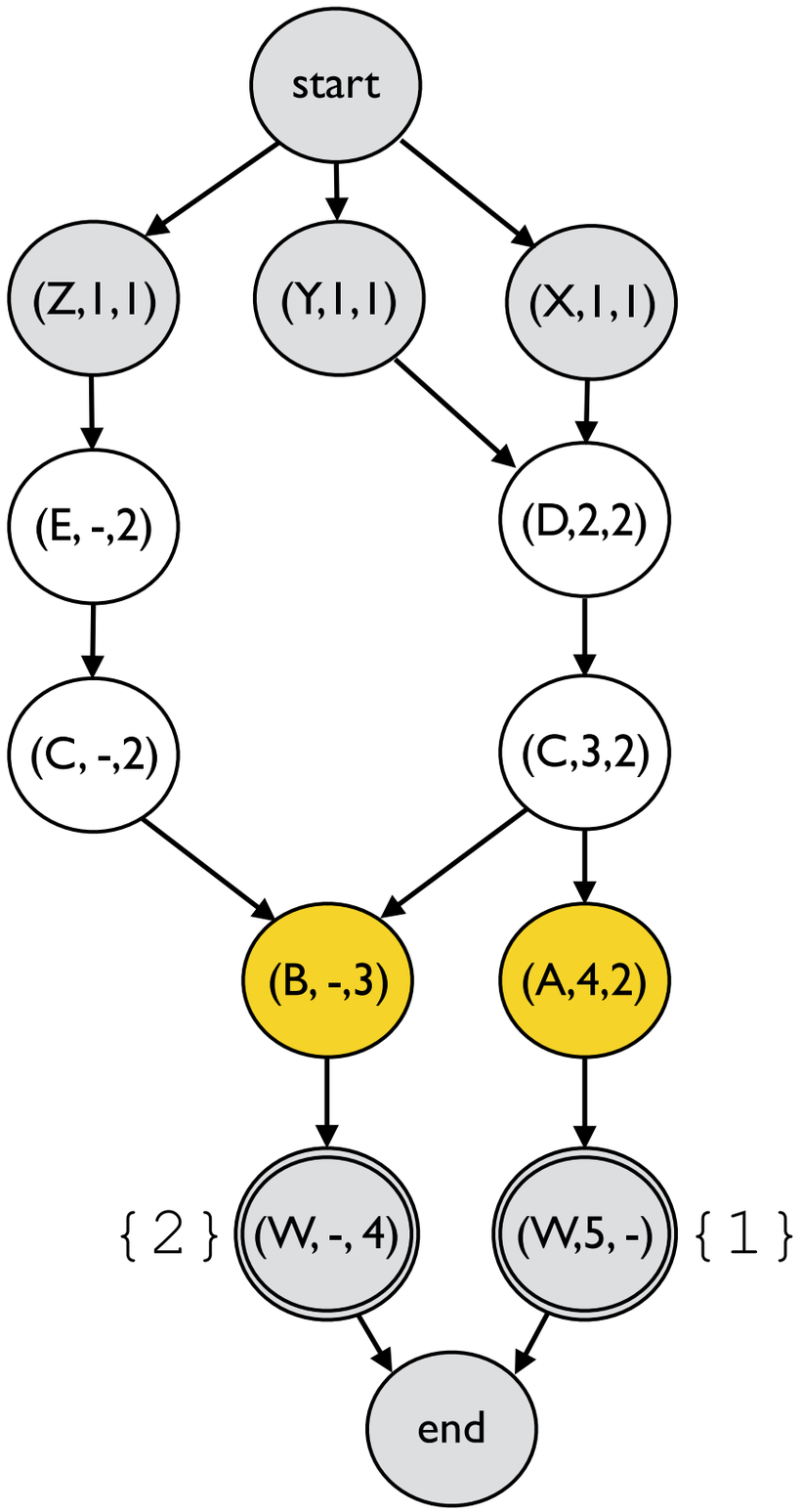
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

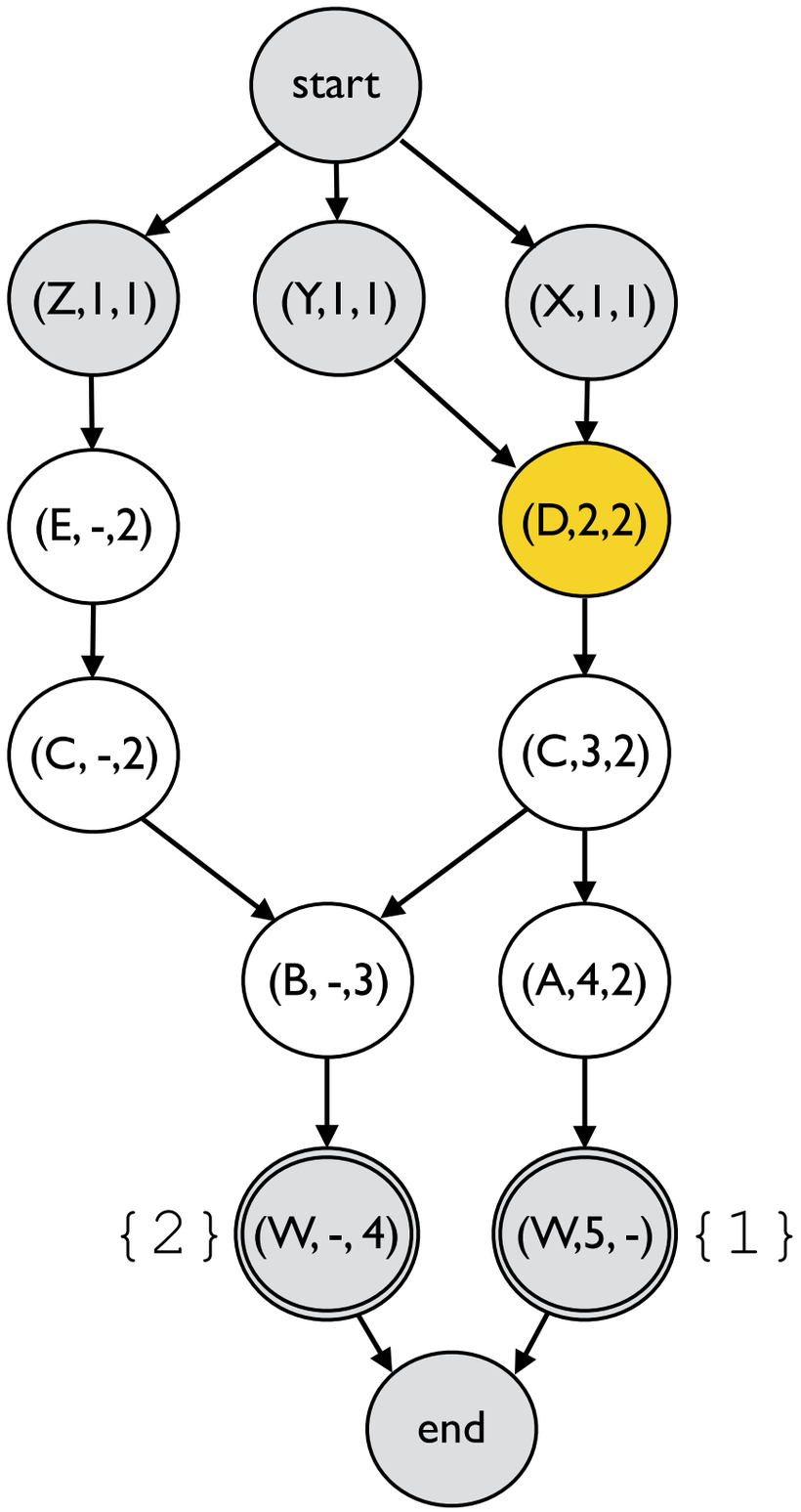
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

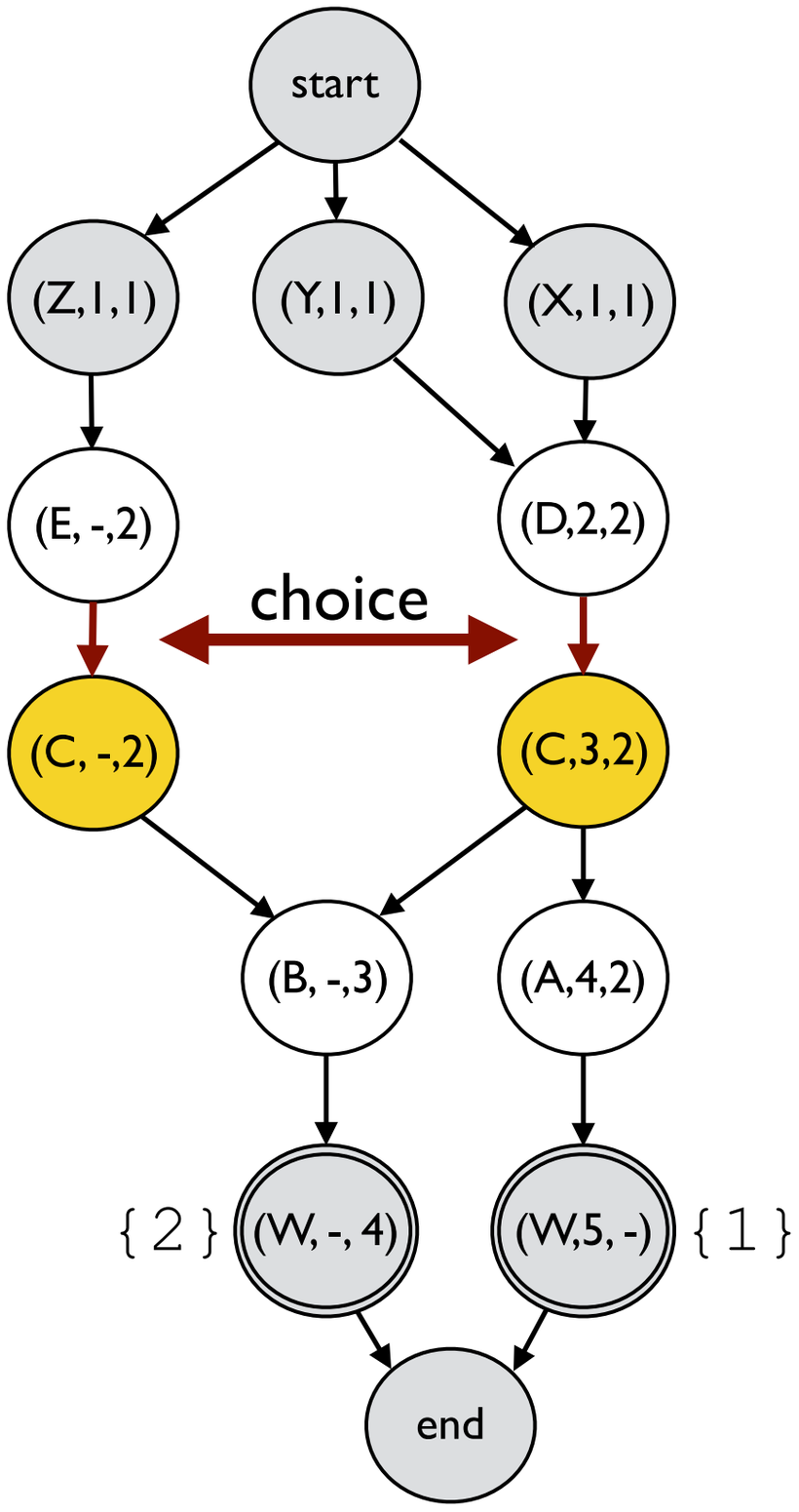
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),

comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),

comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

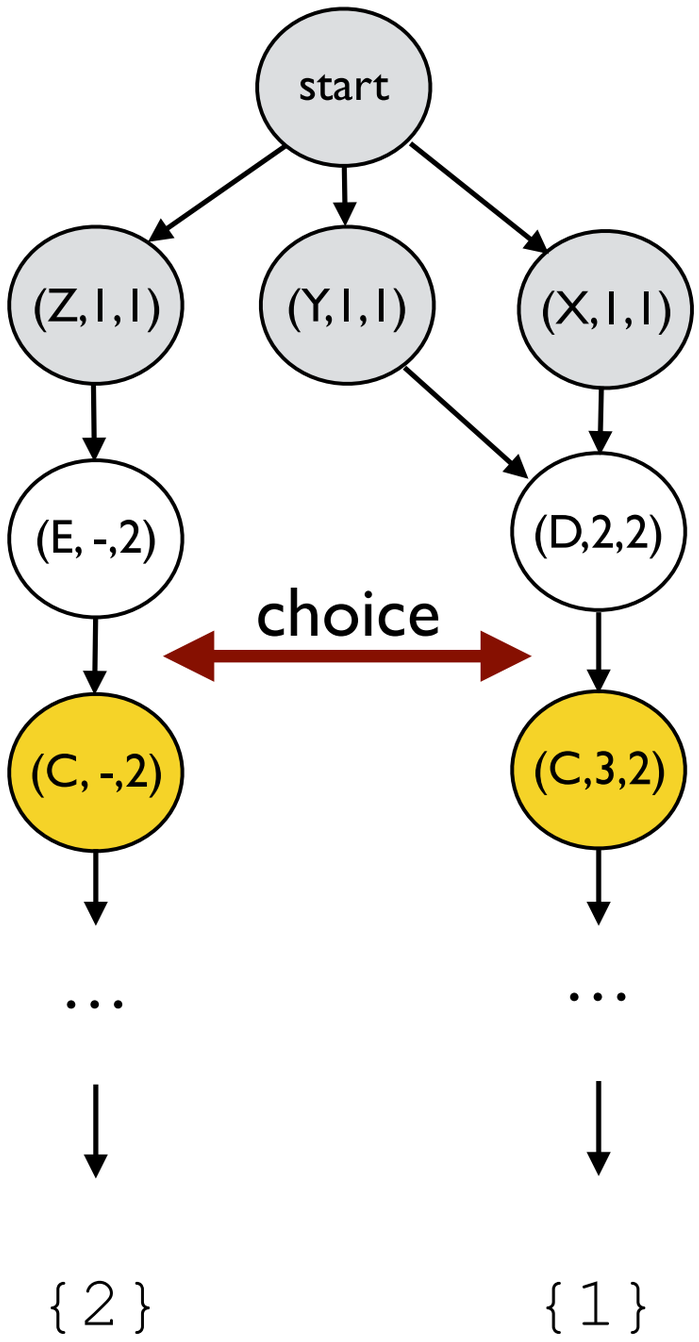
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),

comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),

comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

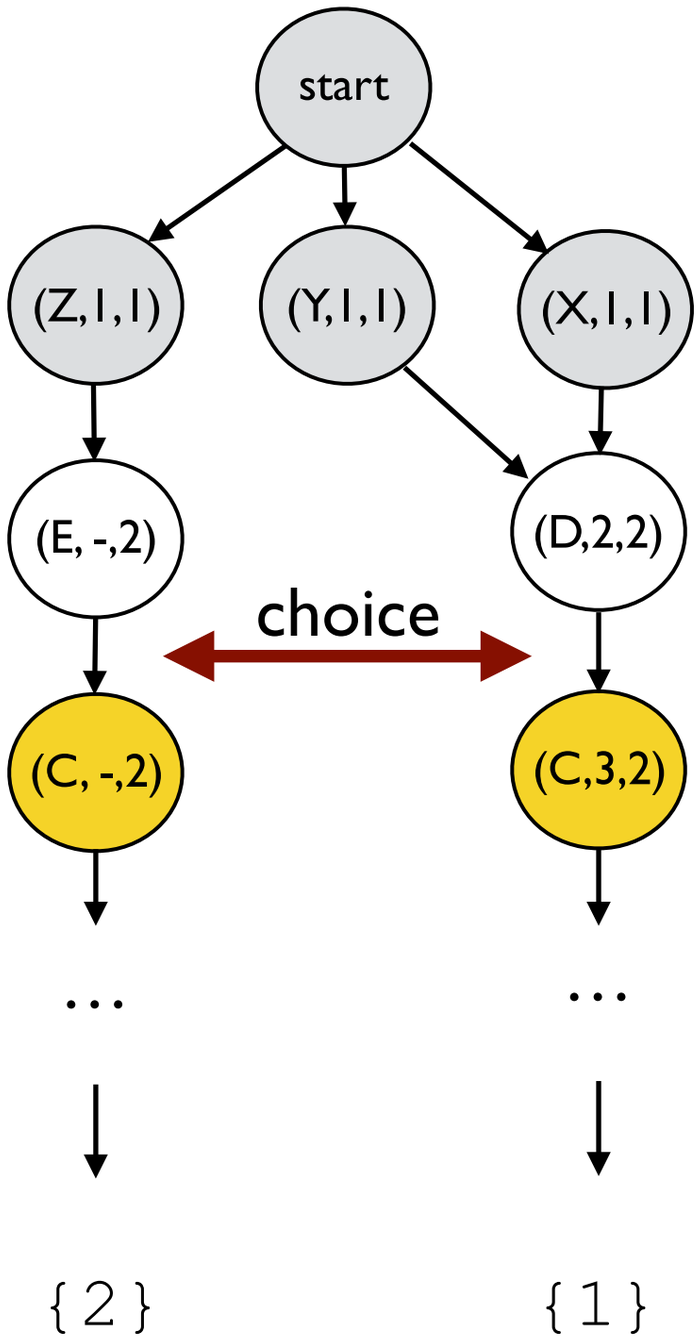
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Highest preference obtainable here

Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

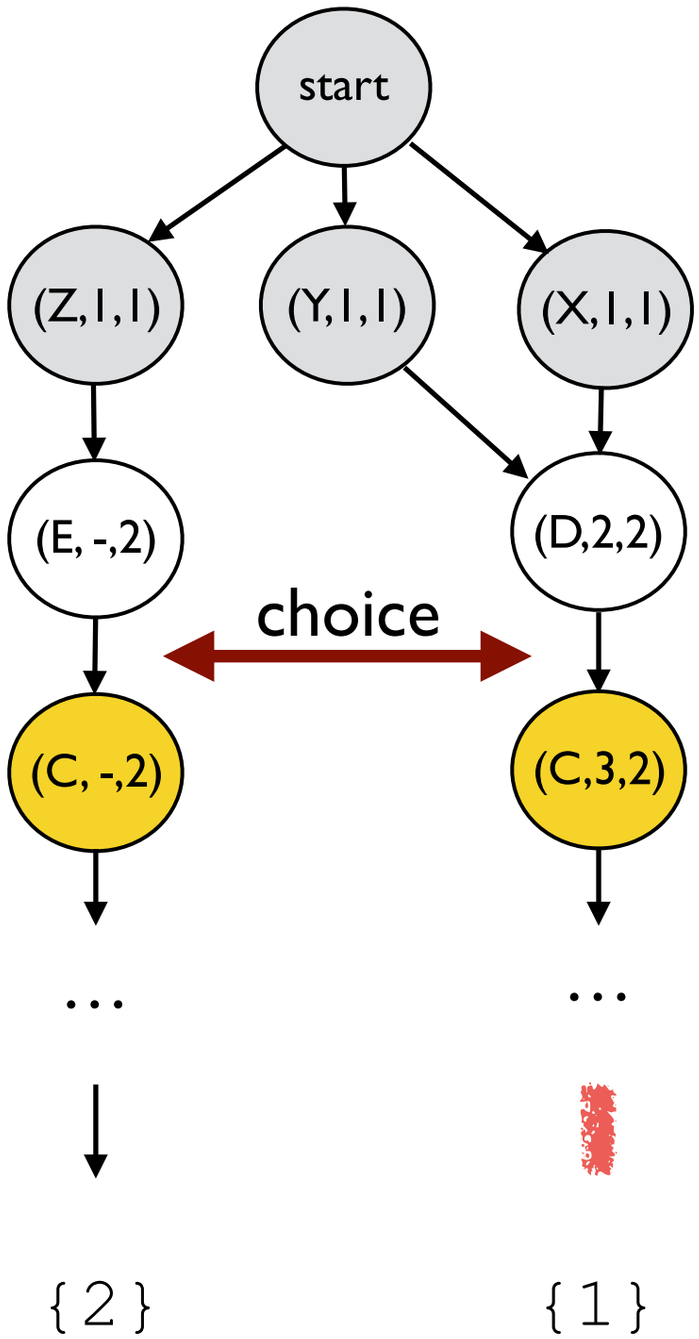
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



But there could be a failure!

Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

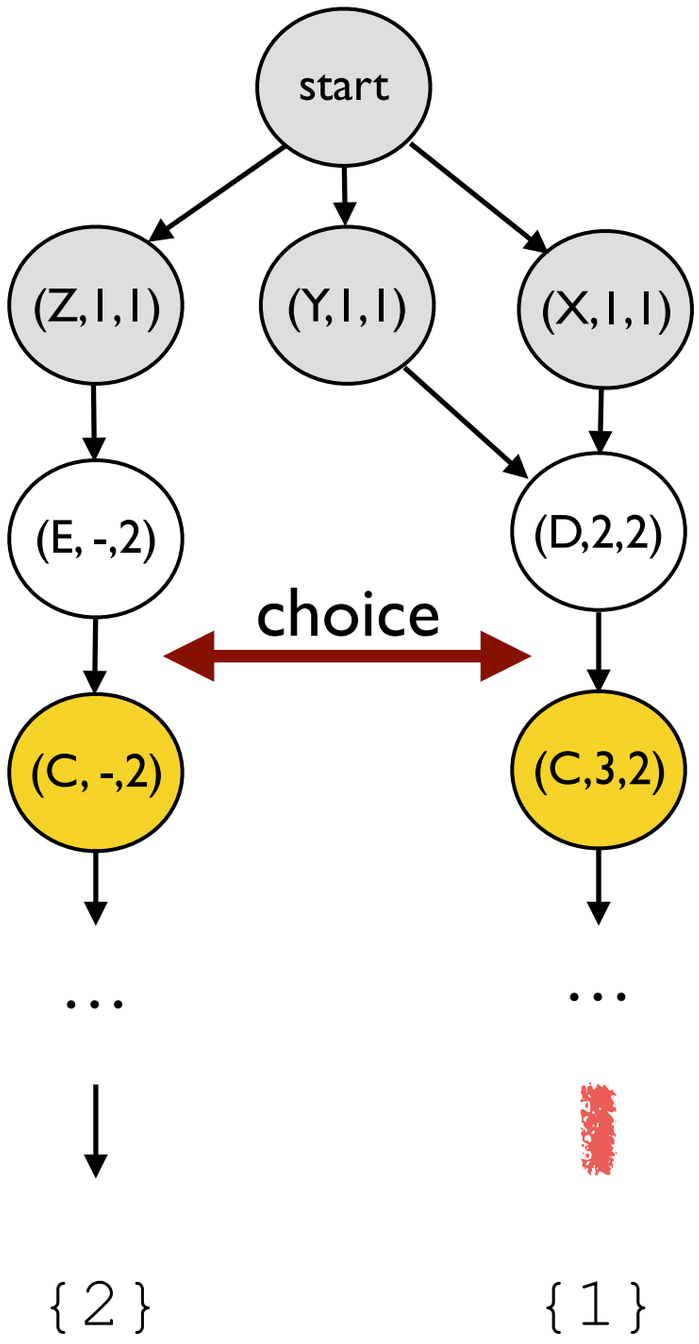
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Sometimes there is no best choice

Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

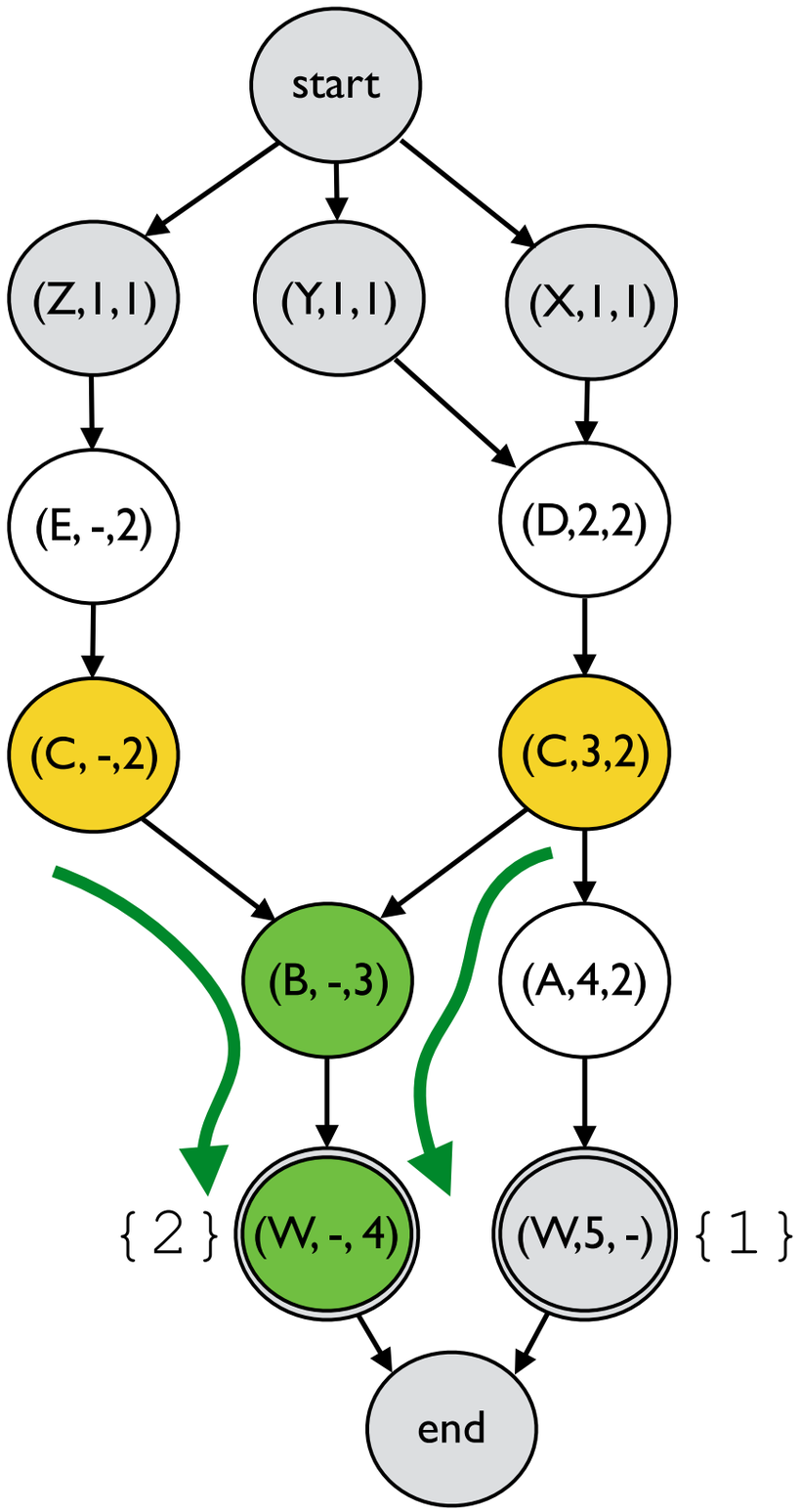
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

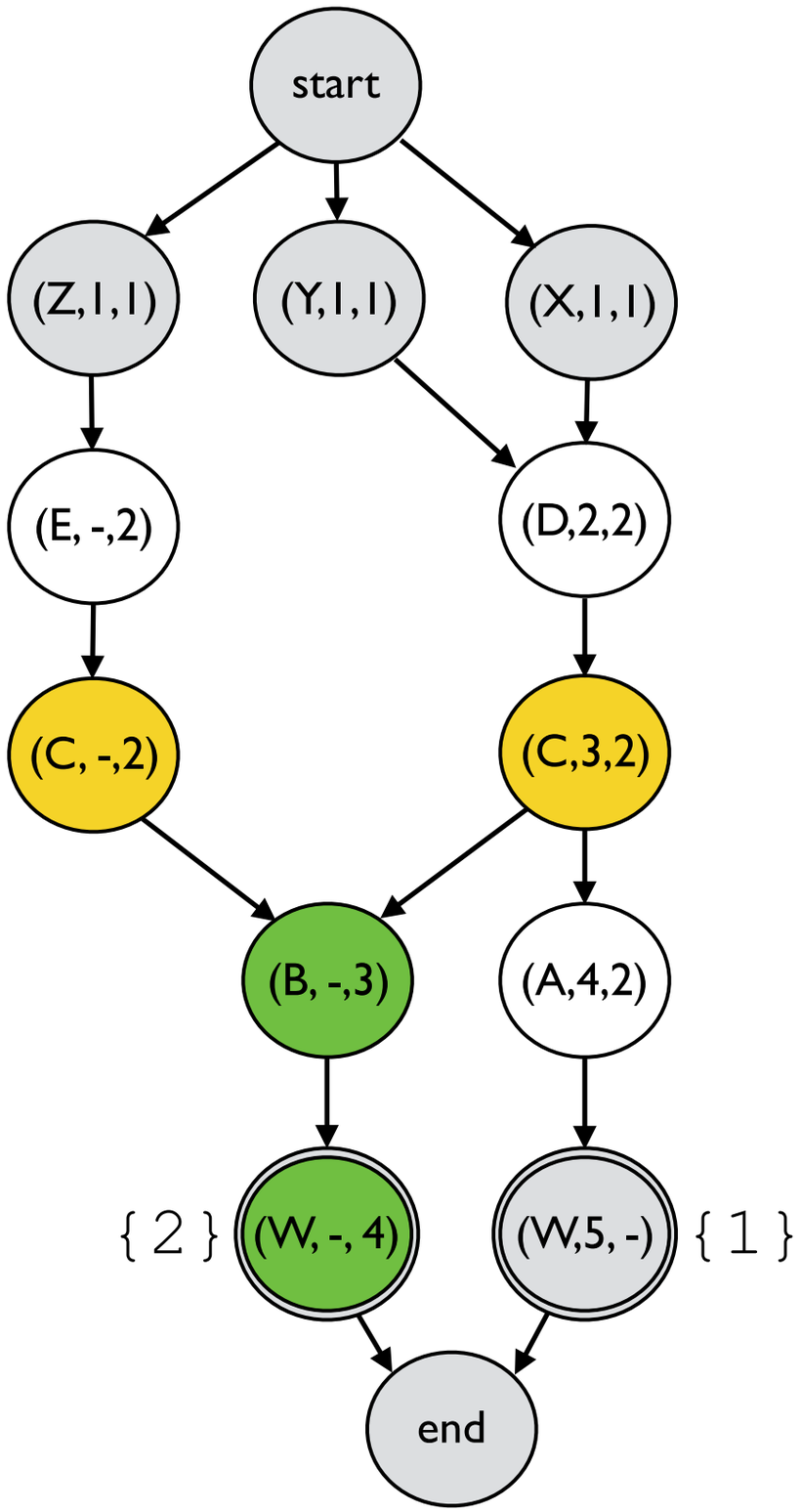
Router D

match regex=(X + Y)

export peer←C, comm←(2,2)

…

Compilation to BGP:



Router A

match peer=C comm=(3,2)

export peer←W, comm←(4,2),
comm← noexport, MED←80

Router B

match peer=C

export peer←W, comm←(-,3),
comm←noexport, MED←81

Router C

match[lp=99] peer=E, comm=(-,2)

export peer←B, comm←(-,2)

match[lp=100] peer=D, comm=(2,2)

export peer←A,B, comm←(3,2)

Router D

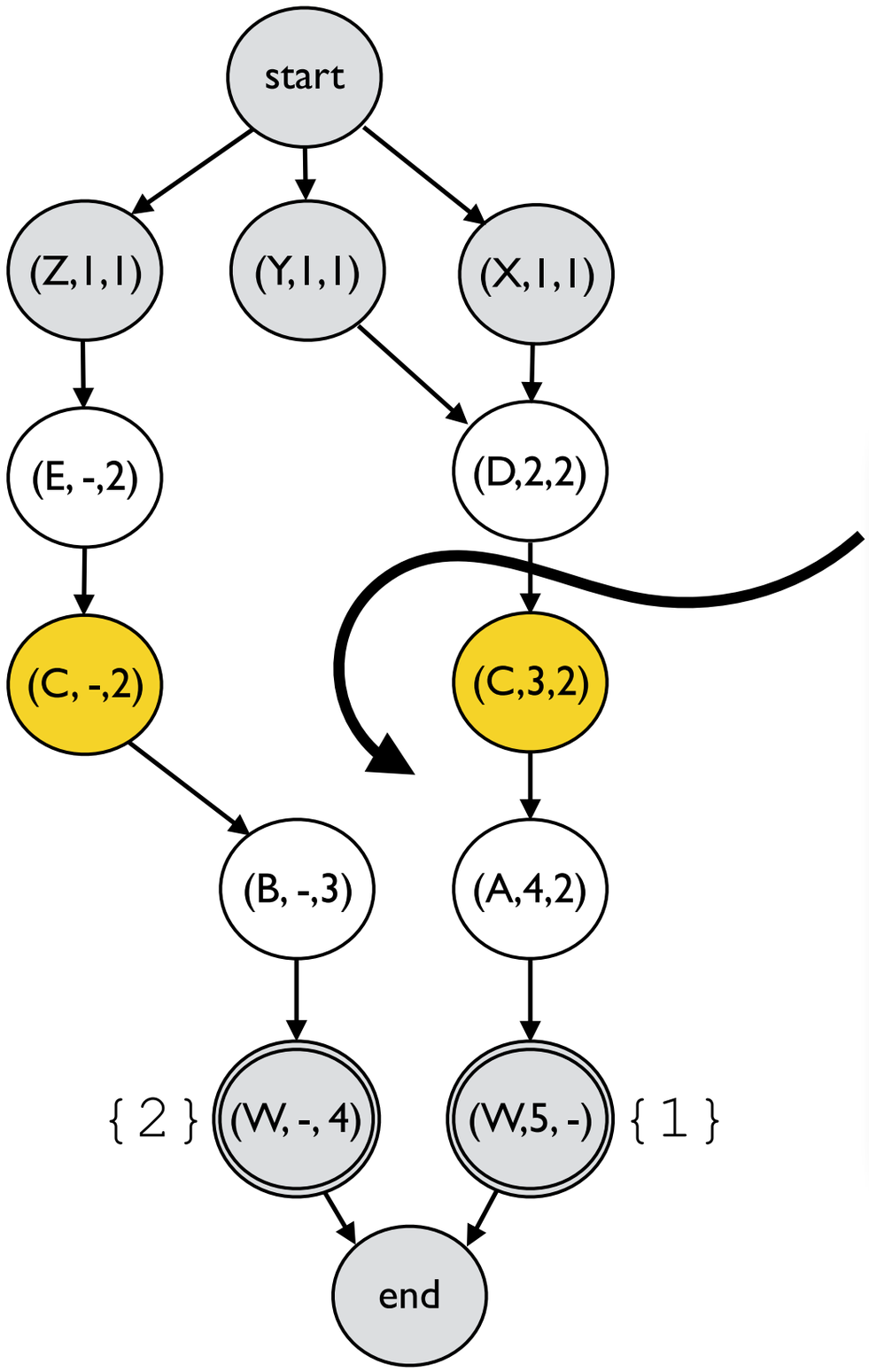
match regex=(X + Y)

export peer←C, comm←(2,2)

…

Prefer D's announce

Compilation to BGP:



If we remove this edge,
the policy becomes unimplementable.

If we choose D's announcement and forward to A, the A-W link may fail

If we choose E's announcement and forward to B, the B-W link may fail (or the A-W link may not fail and we will have a sub-optimal route)

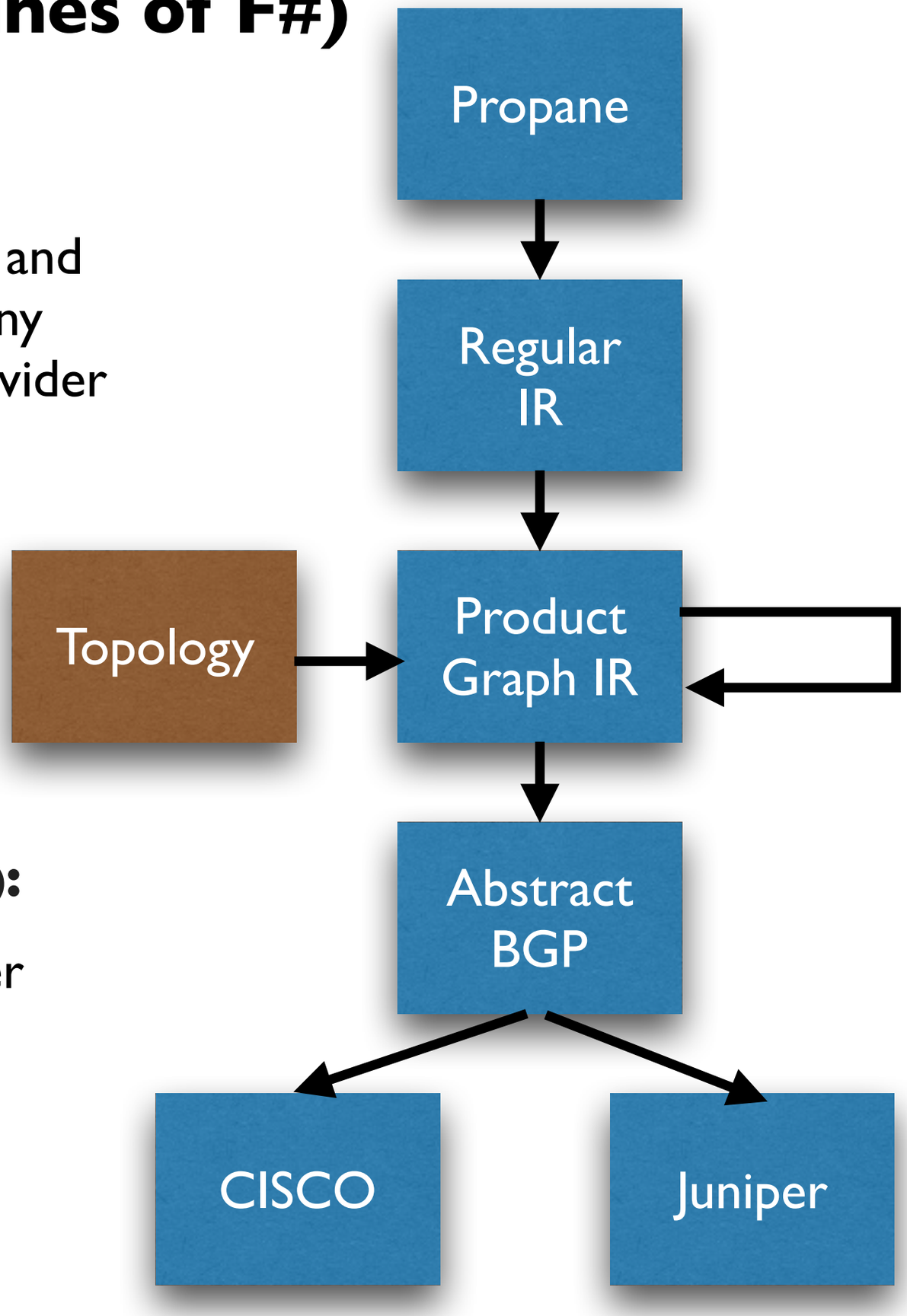
Implementation (5,500 lines of F#)

Benchmarks:

- data center policies (~1600 routers) and backbone policies (~200 routers, many peers/router) from a large cloud provider
- policy from English docs
- Ignoring prefix, customer group and ownership definitions:
  - 31 lines for data center
  - 43 lines for backbone

Scaling (8 core Windows machine):

- 10s/pfx (mean) for largest data center
- 45s/pfx (mean) for largest backbone
- 3 minutes total for the backbone
- 9 minutes total for the data center



Summary

Propane users express high-level objectives.

Objectives involve both intra- and inter-domain constraints.

Failure analysis guarantees strong safety properties.

Compiler bridges the gap between objectives and device-by-device control plane configurations.