# Inductive logic programming: an introduction and recent advances

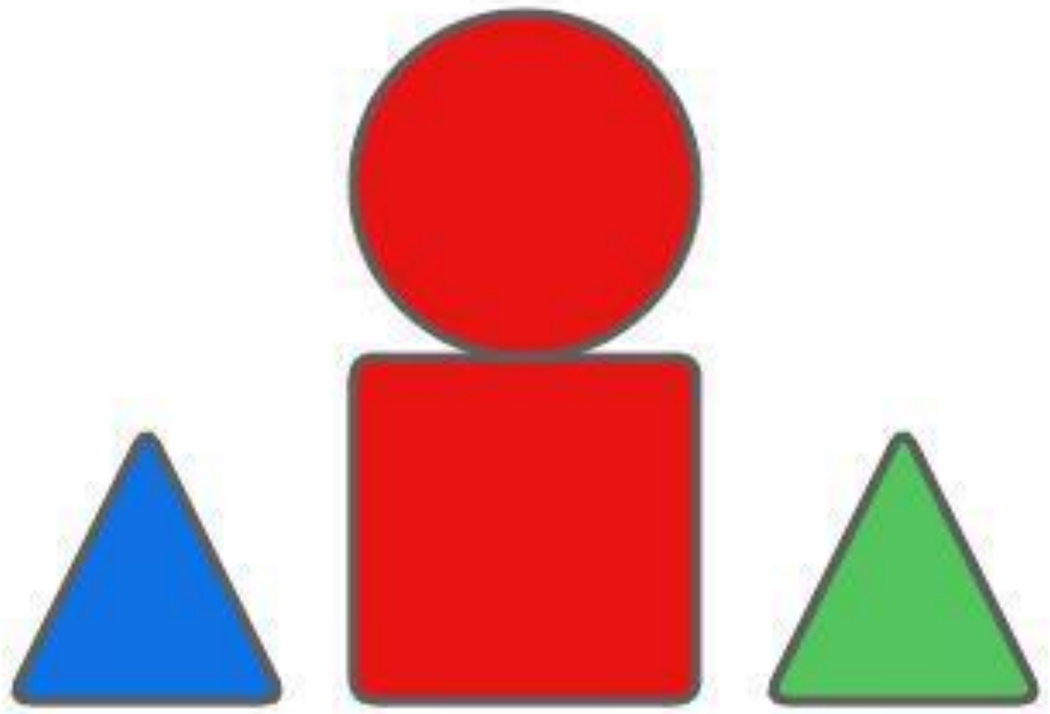
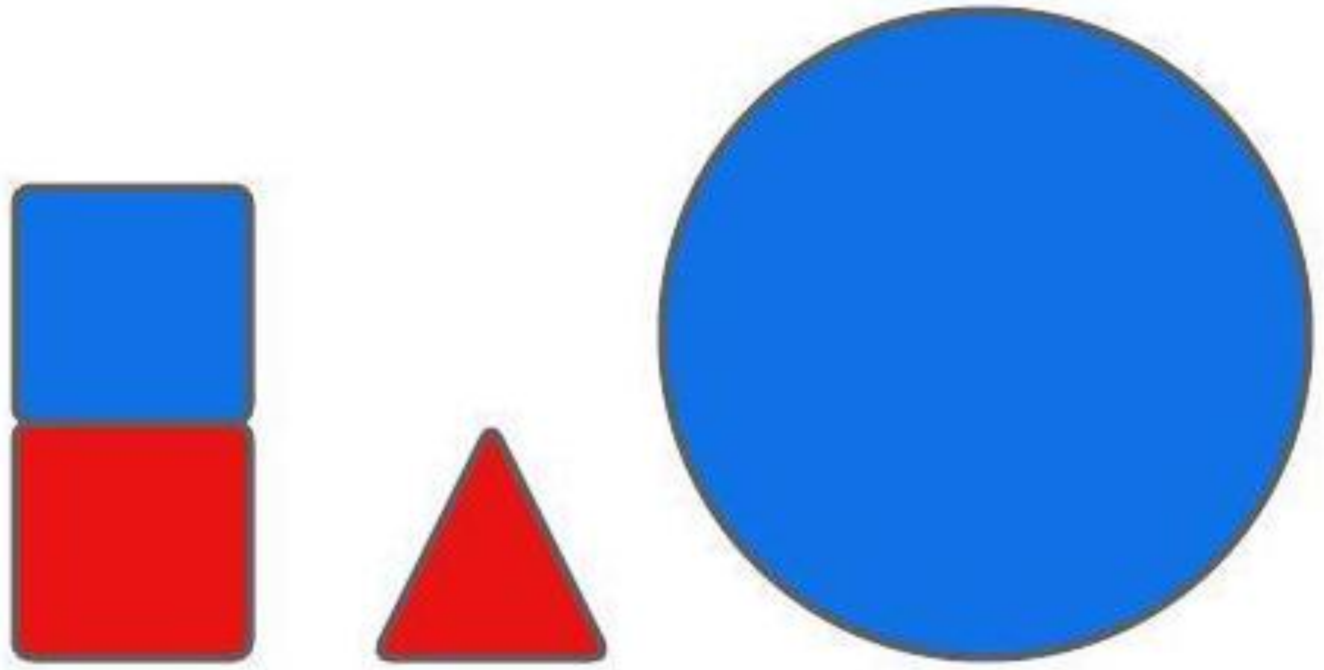
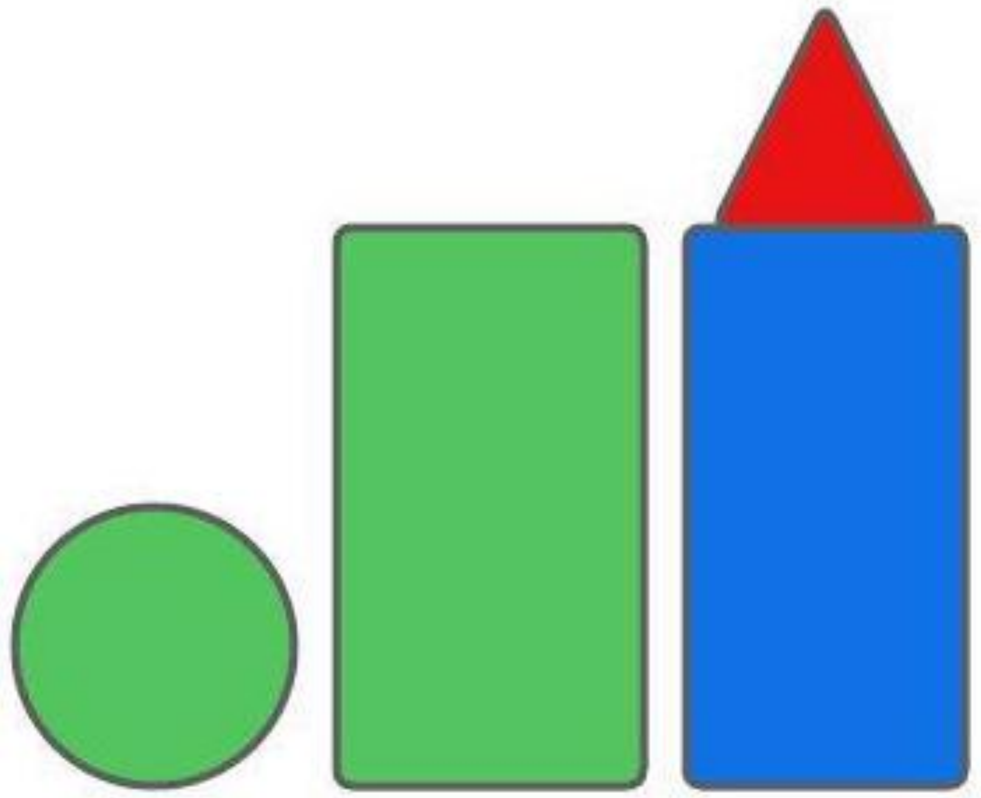
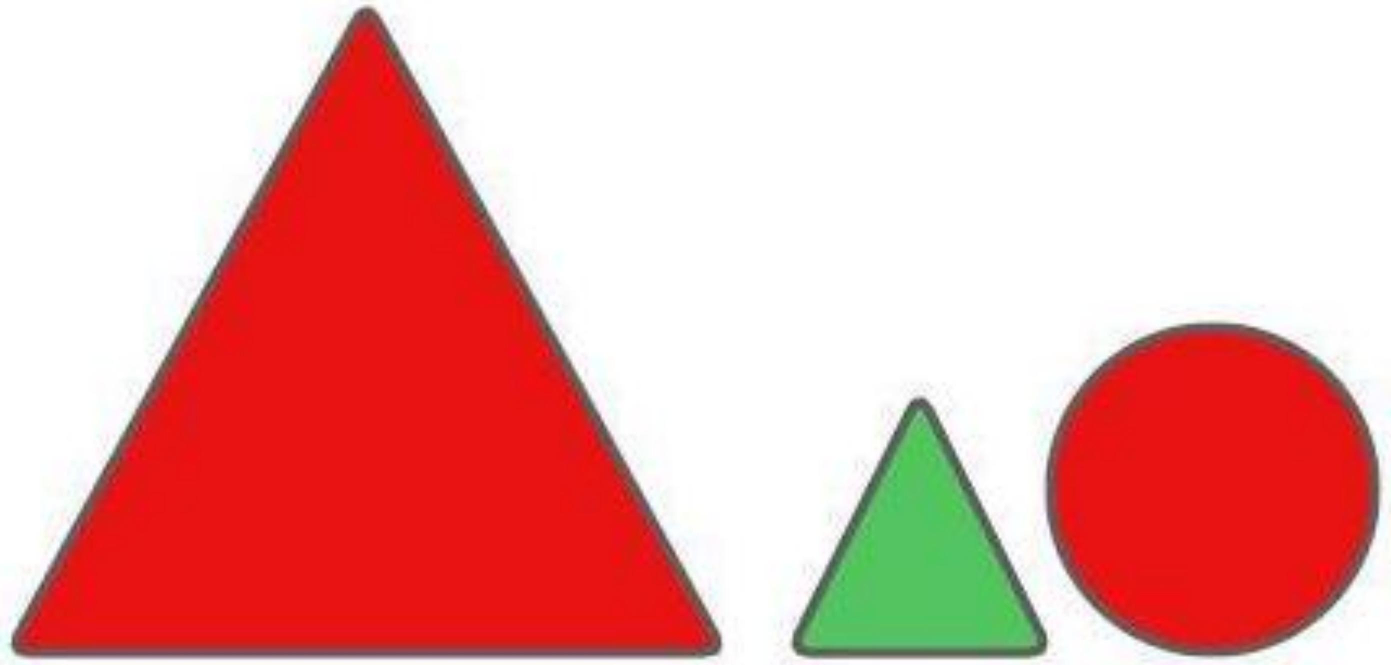
AAAI '23 Tutorial

# Part I: Introduction

# Part I: Introduction

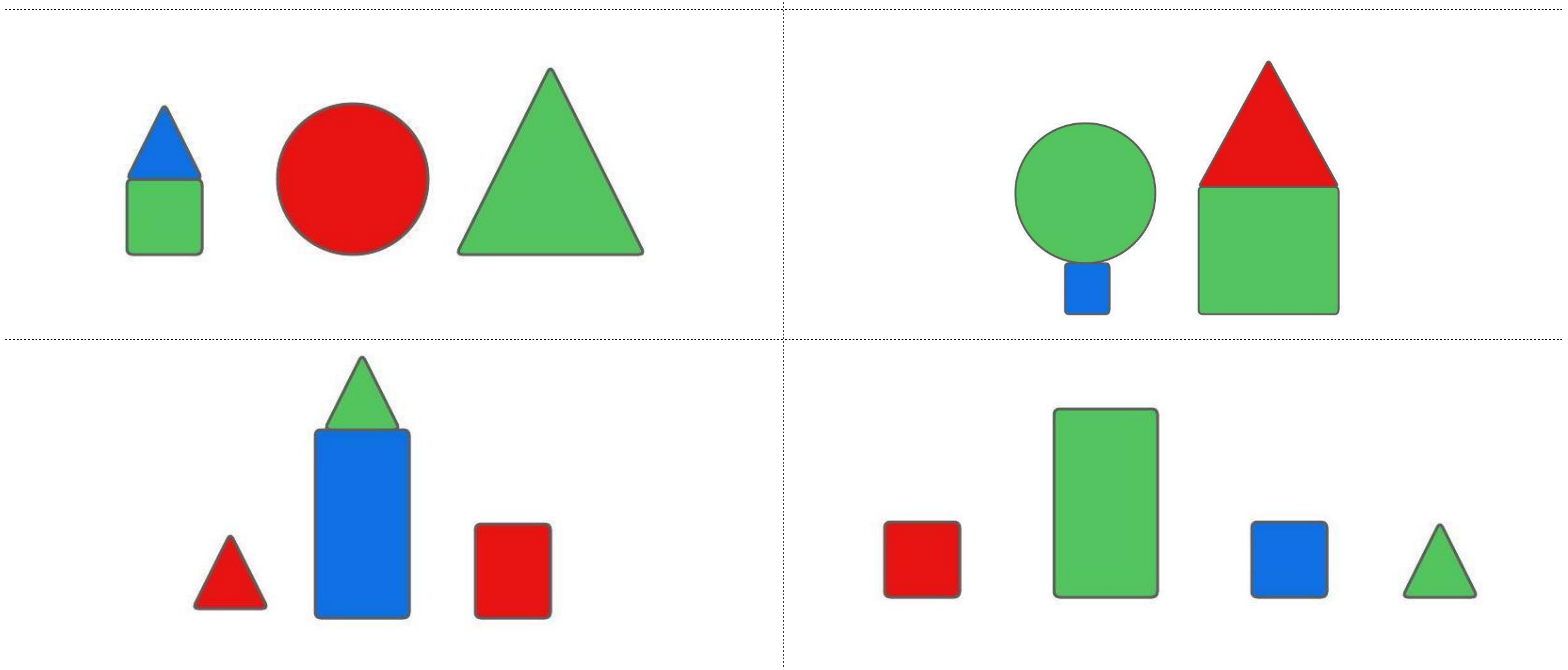
Motivation

# Let’s play a game

| Positive | Negative |
| --- | --- |
|  |  |
|  |  |

*There is one object of each color*

Positive | Negative



*There are two objects in contact with one small and not blue*

# Let’s play a game

| Input | Output |
| --- | --- |
| inductive | gxkvewfpk |
| logic | ekiqn |
| programming | ipkooctiqtr |

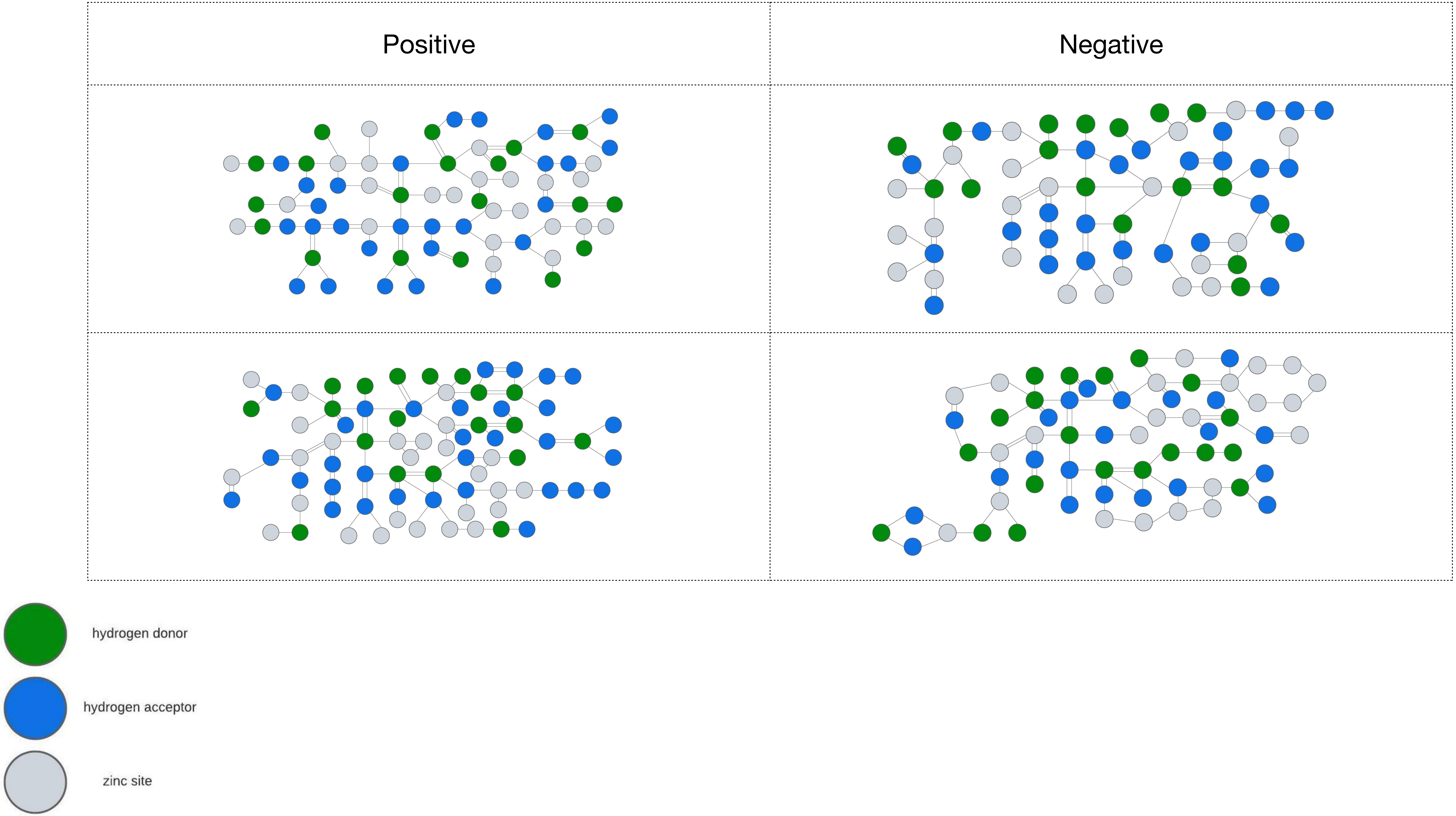
*Add two to each element and reverse*

Let’s play a game


Positive
Negative
hydrogen donor
hydrogen acceptor
zinc site

*There is a hydrogen receptor connected to two zinc sites with single bonds*

Let’s play a game

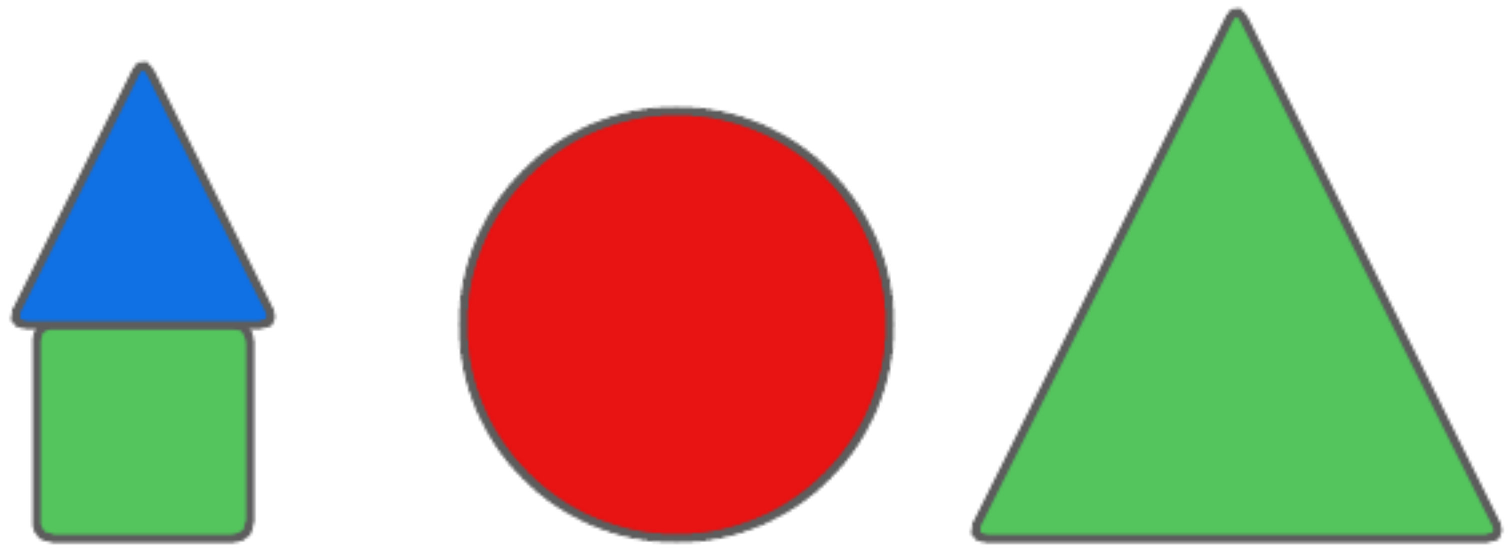

Positive
Negative
hydrogen donor
hydrogen acceptor
zinc site

Let’s use ML on these problems

# What do we need?

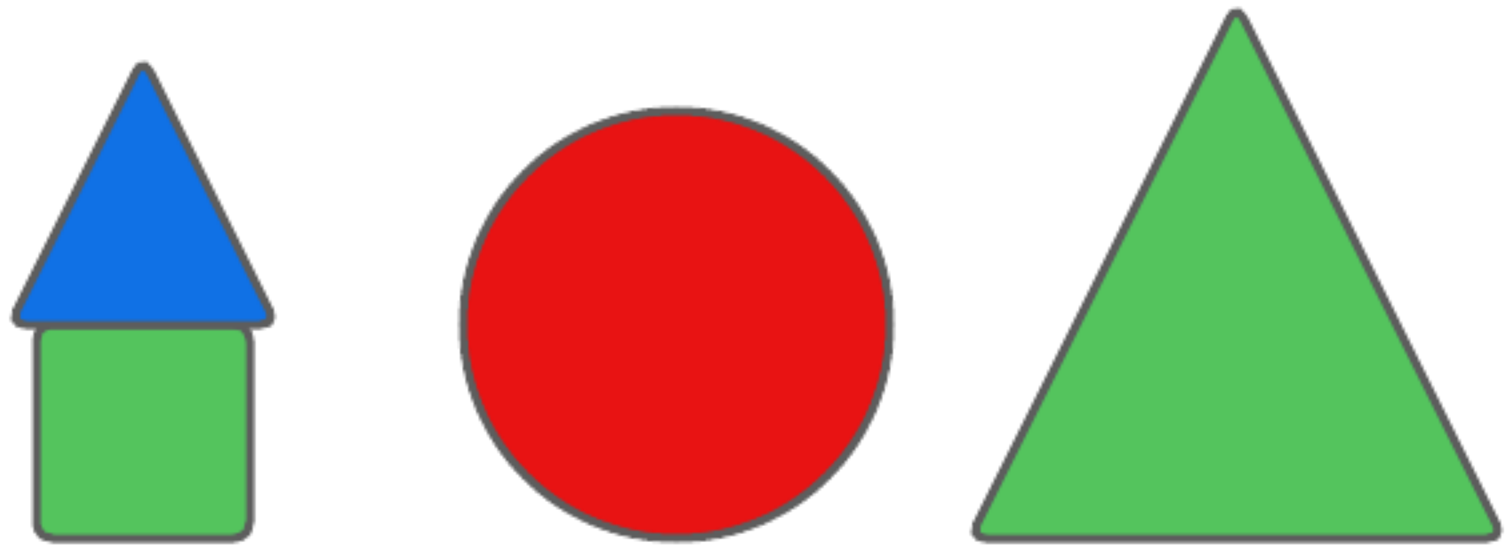
Learn from small a number of examples

# Playing Zendo with ML



Features

# Playing Zendo with ML



Features

red blue green

rectangle triangle square circle

medium large small

contact_p1 contact_p2

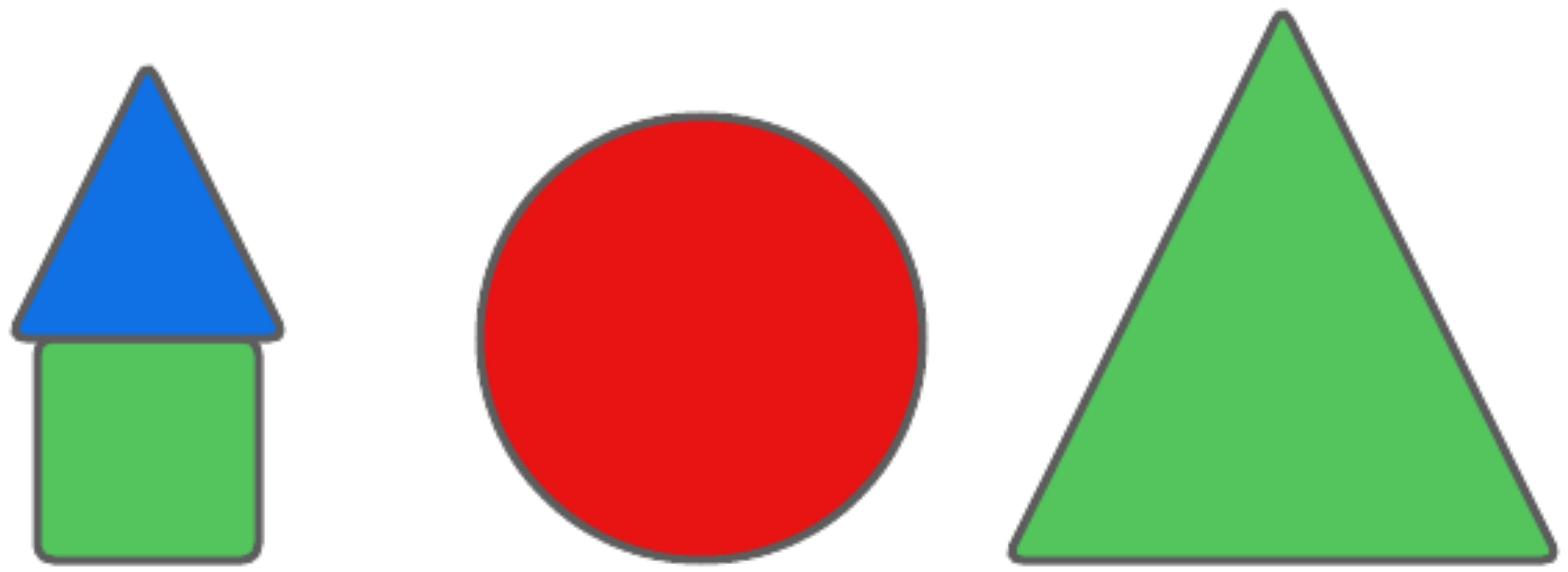
contact_p3 contact_p4

x_pos y_pos

right_of_p1 left_of_p1 ...

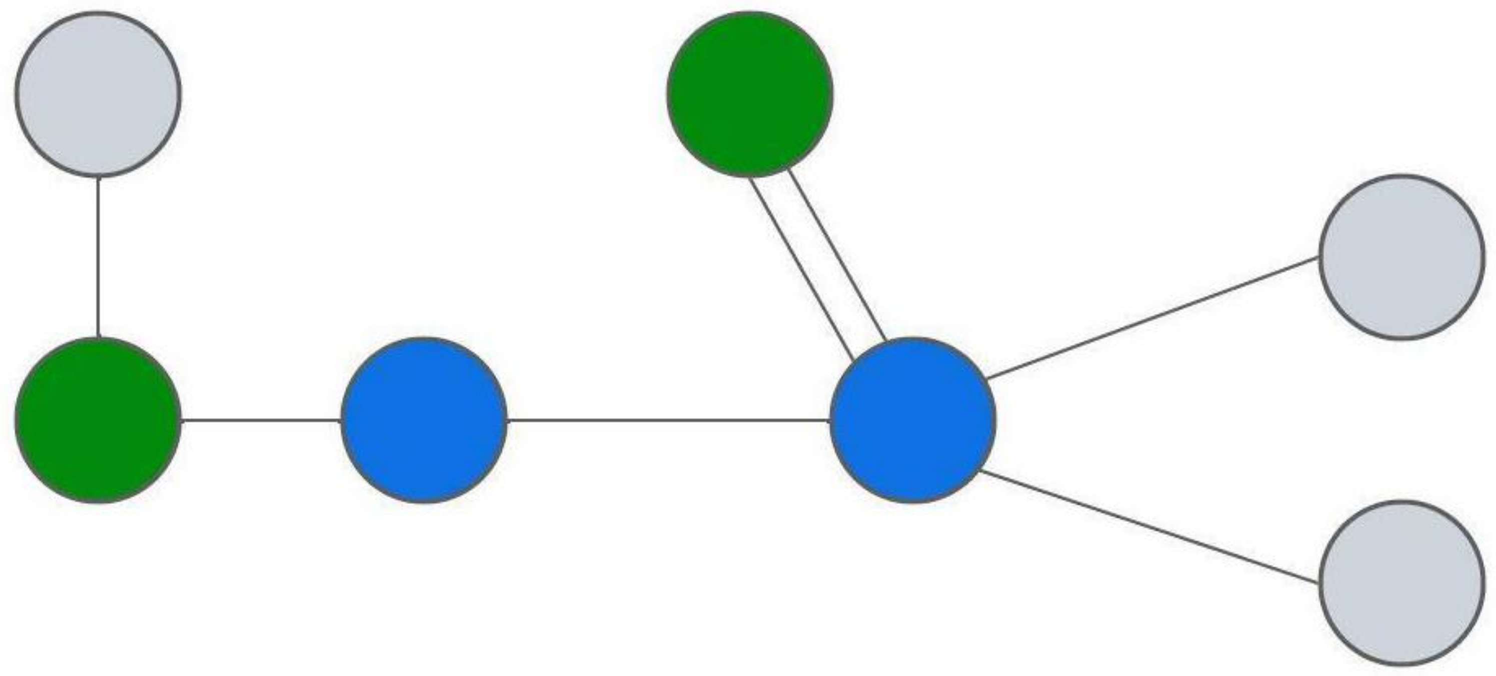
# Playing Zendo with ML

| | red | green | blue | triangle | rectangle | square | circle | contact_p1 | contact_p2 | contact_p3 | contact_p4 | small | medium | large |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| piece1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| piece2 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| piece3 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| piece4 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |



Learn explainable solutions

# Understanding networks with ML

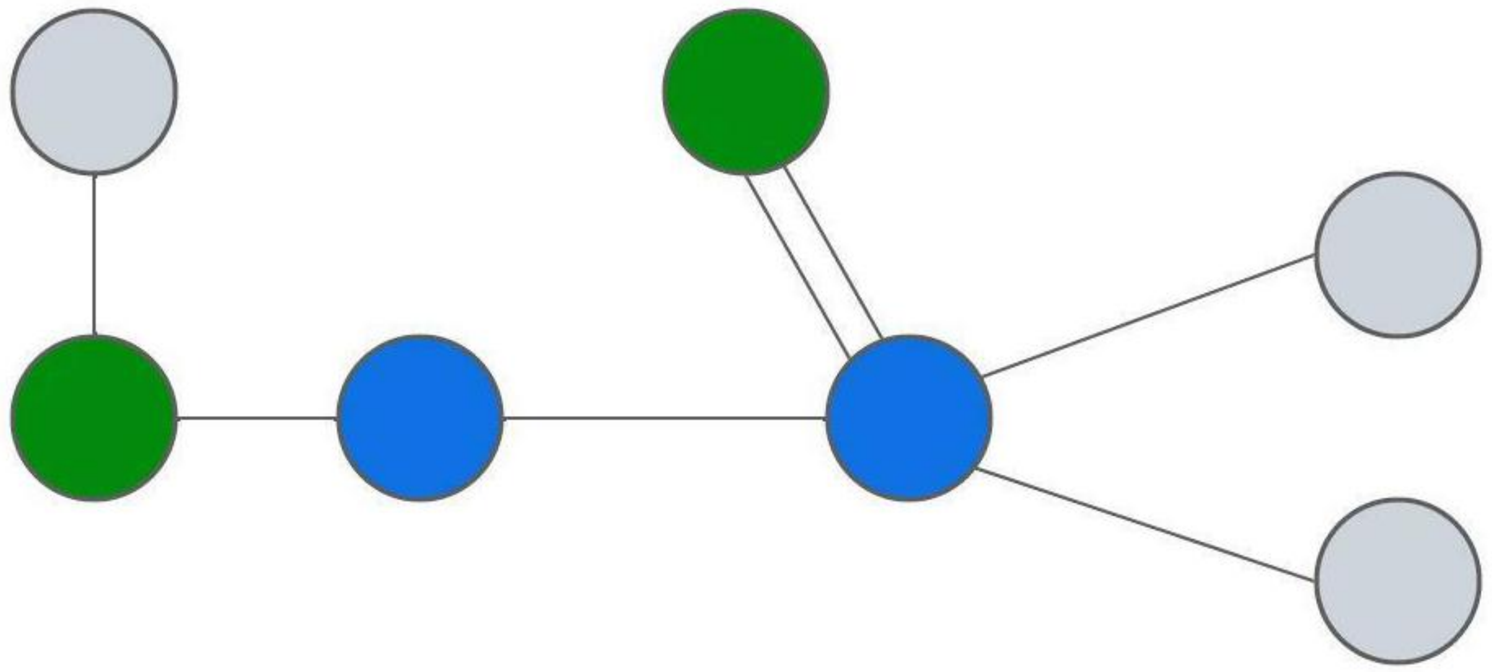


Features

hacc hdonor
zincsite
singlebond_a1 singlebond_a2
singlebond_a1 doublebond_a1
doublebond_a2 doublebond_a3
distance_a1 distance_a2
distance_a3...

# Understanding networks with ML

| | hacc | hdonor | zincsite | singlebond_a1 | singlebond_a2 | singlebond_a3 | doublebond_a1 | doublebond_a2 | doublebond_a3 |
|---|---|---|---|---|---|---|---|---|---|
| a1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| a2 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| a3 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| a4 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |



Learn from highly relational data

# Breaking the cipher with ML

| Input | Output |
| --- | --- |
| inductive | gxkvewfpk |
| logic | ekiqn |
| programming | ipkooctiqtr |

Features

input_1_a input_1_b input_1_c
input_2_a input_2_b input_2_c
input_3_a input_3_b input_3_c
…

# Breaking the cipher with ML

| | input_1_a | input_1_b | input_1_c | input_1_i | input_1_j | input_1_k | input_1_l | input_1_m | input_1_p |
|---|---|---|---|---|---|---|---|---|---|
| inductive | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| logic | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| programmin g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

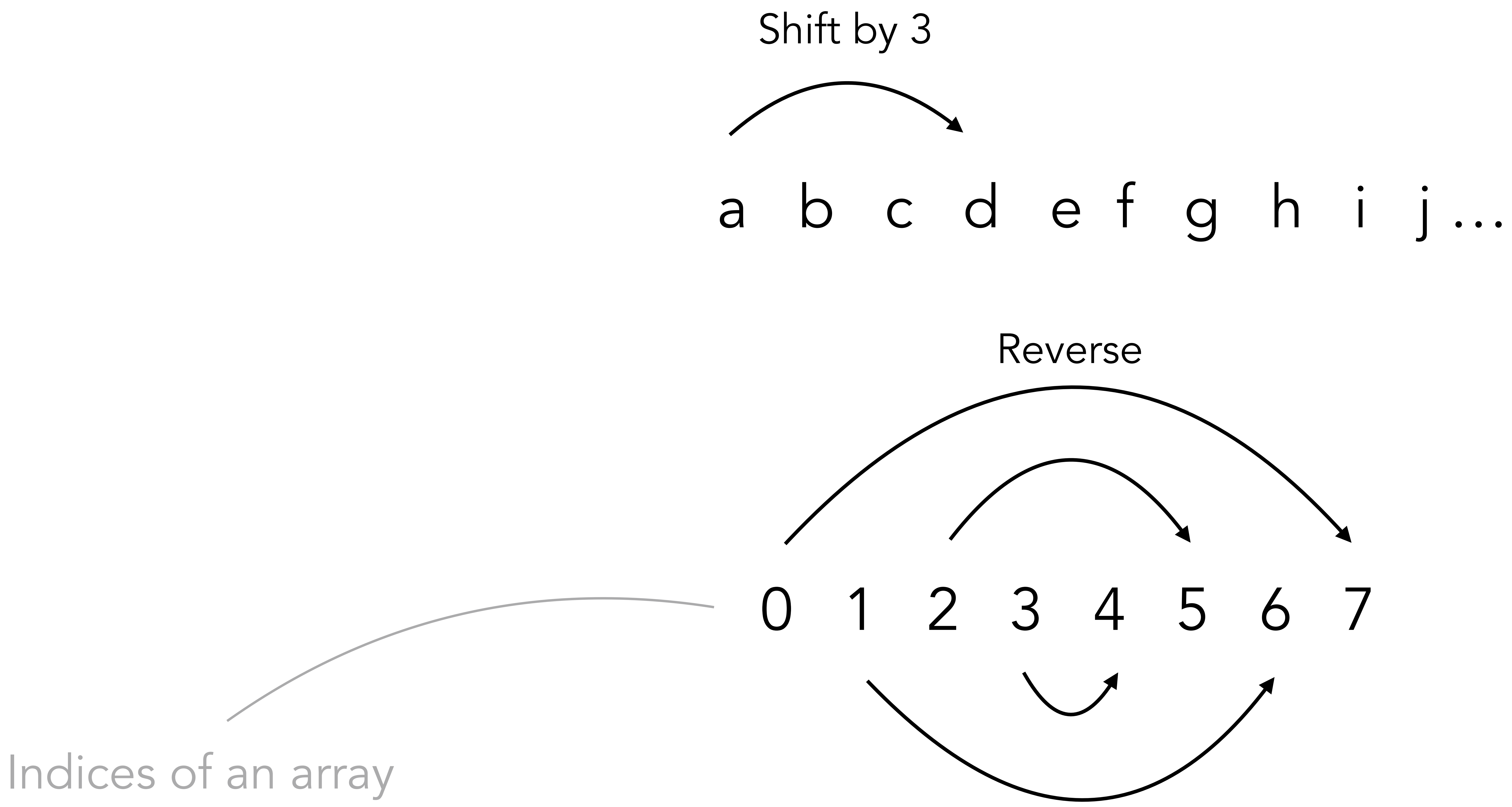
| Input | Output |
|---|---|
| inductive | gxkvewfpk |
| logic | ekiqn |
| programming | ipkooctiqtr |

# Breaking the cipher with ML

Shift by 3

a b c d e f g h i j …

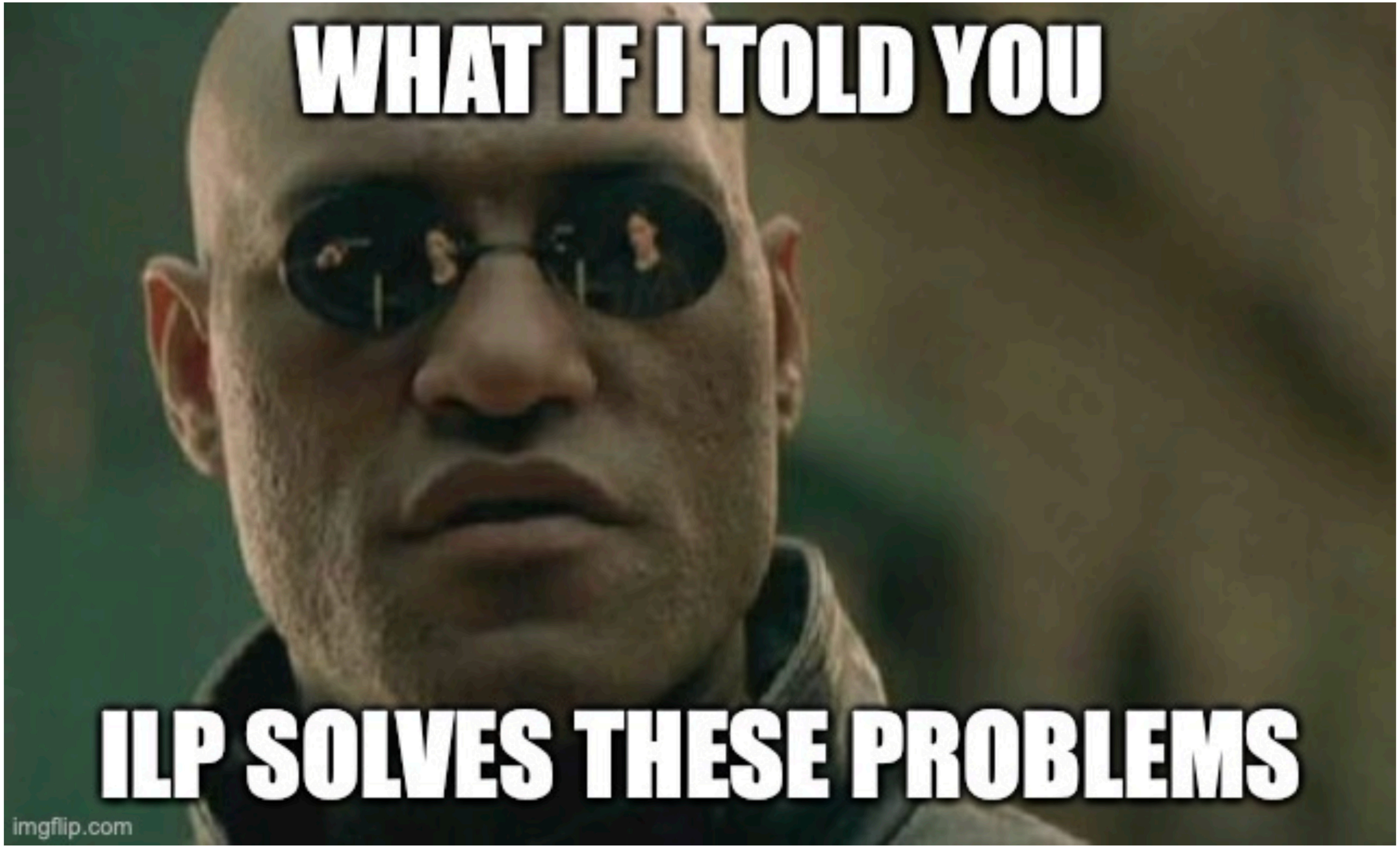
# Breaking the cipher with ML



Learn from small a number of examples

Explainable solutions

Learn from highly relational data


WHAT IF I TOLD YOU
ILP SOLVES THESE PROBLEMS
imgflip.com

# What is ILP good at?

Learn from small a number of examples ✓

Learn from small a number of examples ✓

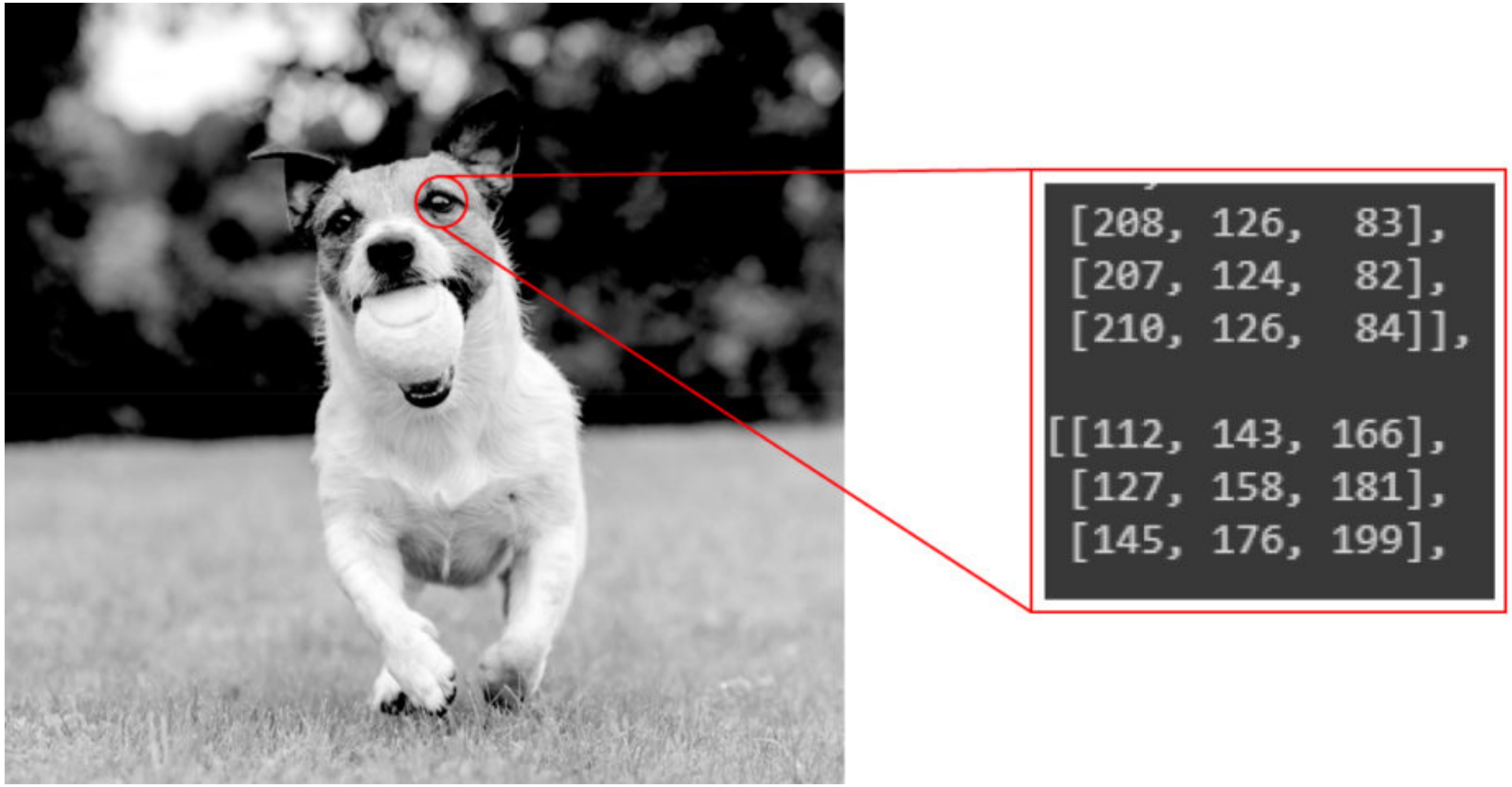
Explainable solutions ✓

Learn from small a number of examples ✓

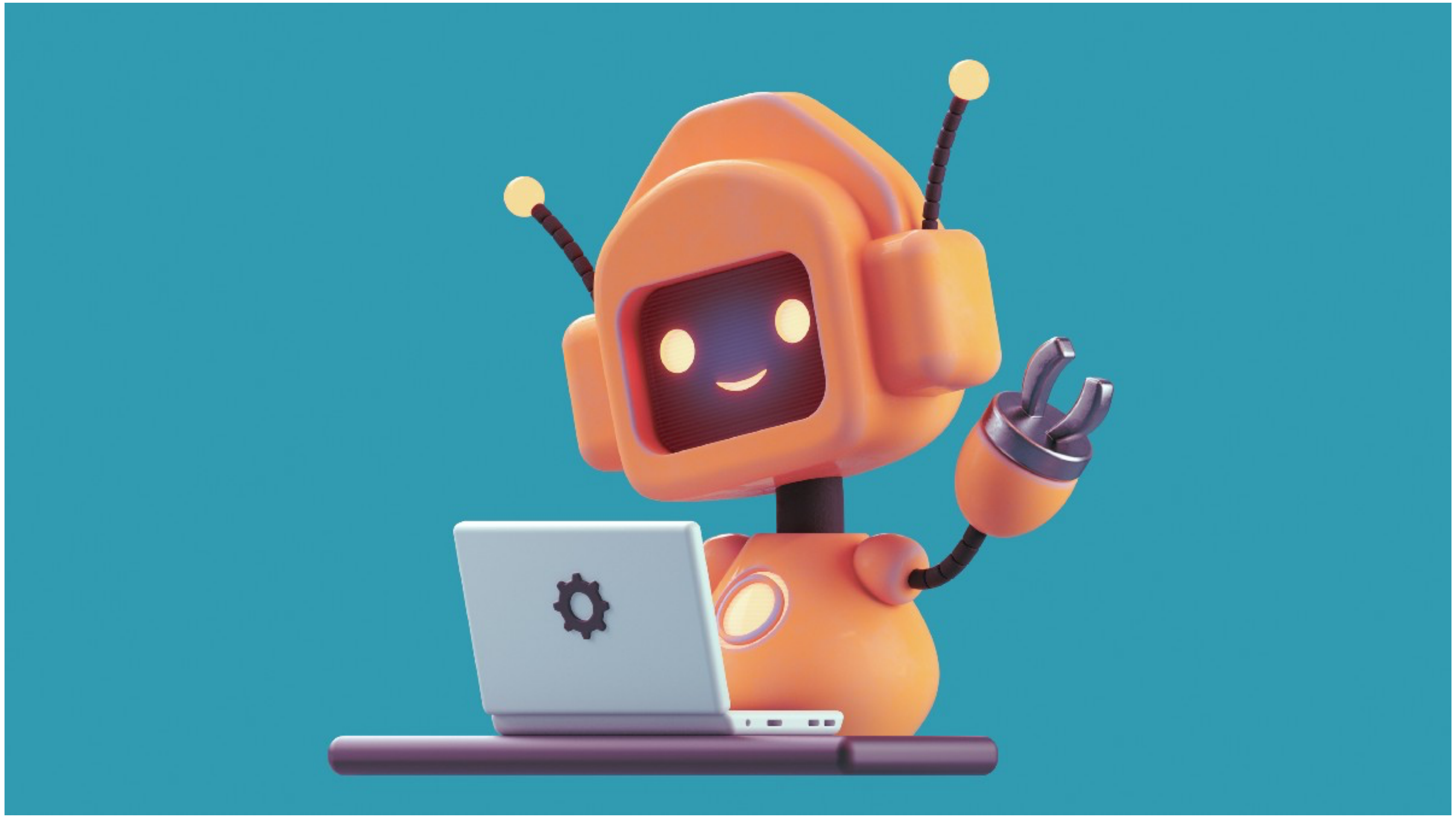
Explainable solutions ✓

Learn from highly relational data ✓

# ILP is not a silver bullet


[208, 126,  83],
[207, 124,  82],
[210, 126,  84]],
[[112, 143, 166],
[127, 158, 181],
[145, 176, 199],


Google DeepMind
Challenge Match
8 - 15 March 2016
AlphaGo
AlphaGo
Lee Sedol
Google DeepMind
Challenge Match
8 - 15 March 2016



# Goal of this tutorial

Developing intuition about ILP and its possibilities

# Goal of this tutorial

For technical details, check the accompanying publication

**Inductive Logic Programming At 30: A New Introduction**

**Andrew Cropper** ANDREW.CROPPER@CS.OX.AC.UK
*University of Oxford*

**Sebastijan Dumančić** S.DUMANCIC@TUDELFT.NL
*TU Delft*

**Abstract**

Inductive logic programming (ILP) is a form of machine learning. The goal of ILP is to induce a hypothesis (a set of logical rules) that generalises training examples. As ILP turns 30, we provide a new introduction to the field. We introduce the necessary logical notation and the main learning settings; describe the building blocks of an ILP system; compare several systems on several dimensions; describe four systems (Aleph, TILDE, ASPAL, and Metagol); highlight key application areas; and, finally, summarise current limitations and directions for future research.

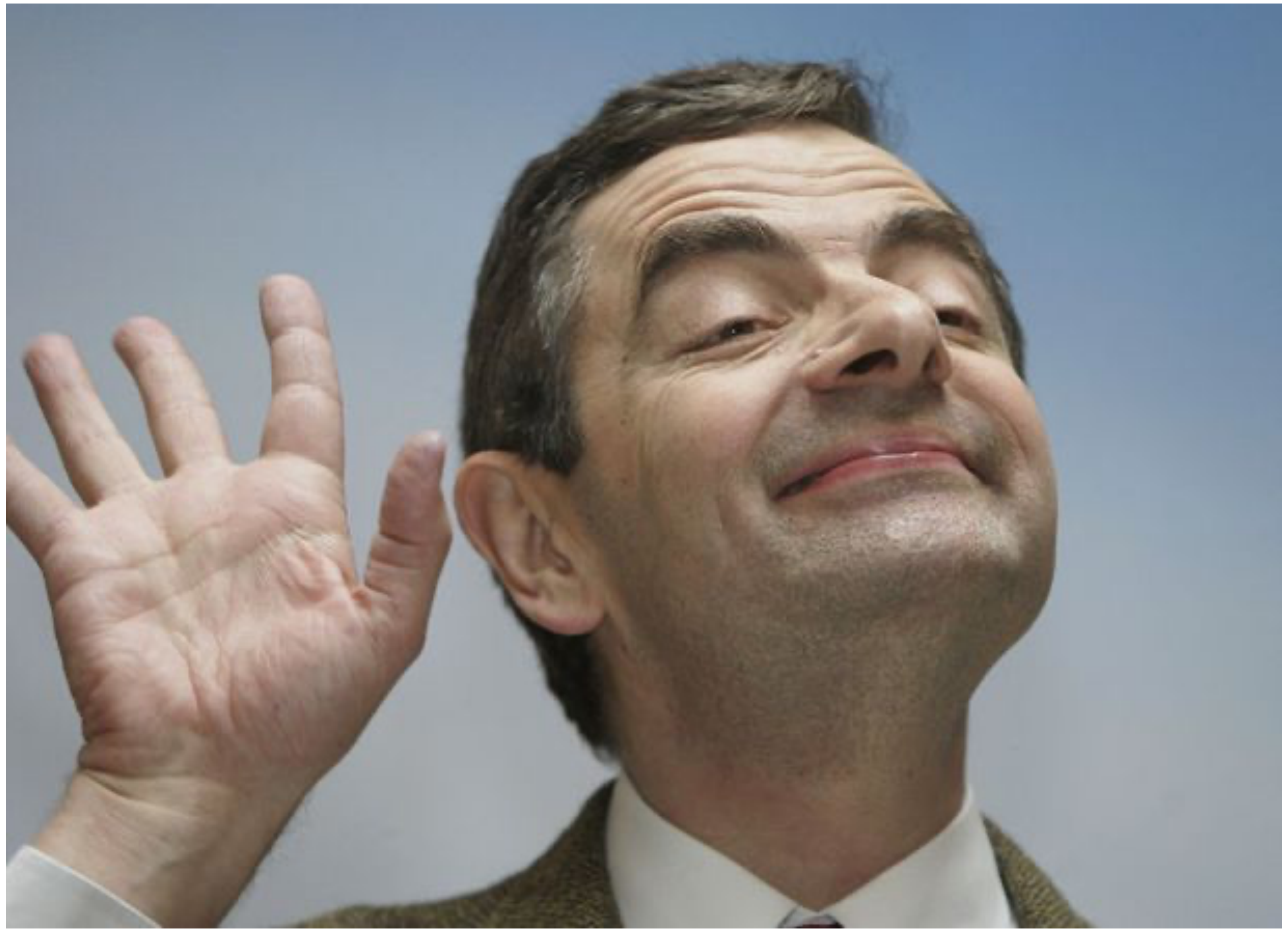
# Outline

1. Logic: What and why?

2. Building an ILP system

3. Features and applications

4. Challenges and opportunities



**Please ask questions and interrupt!**

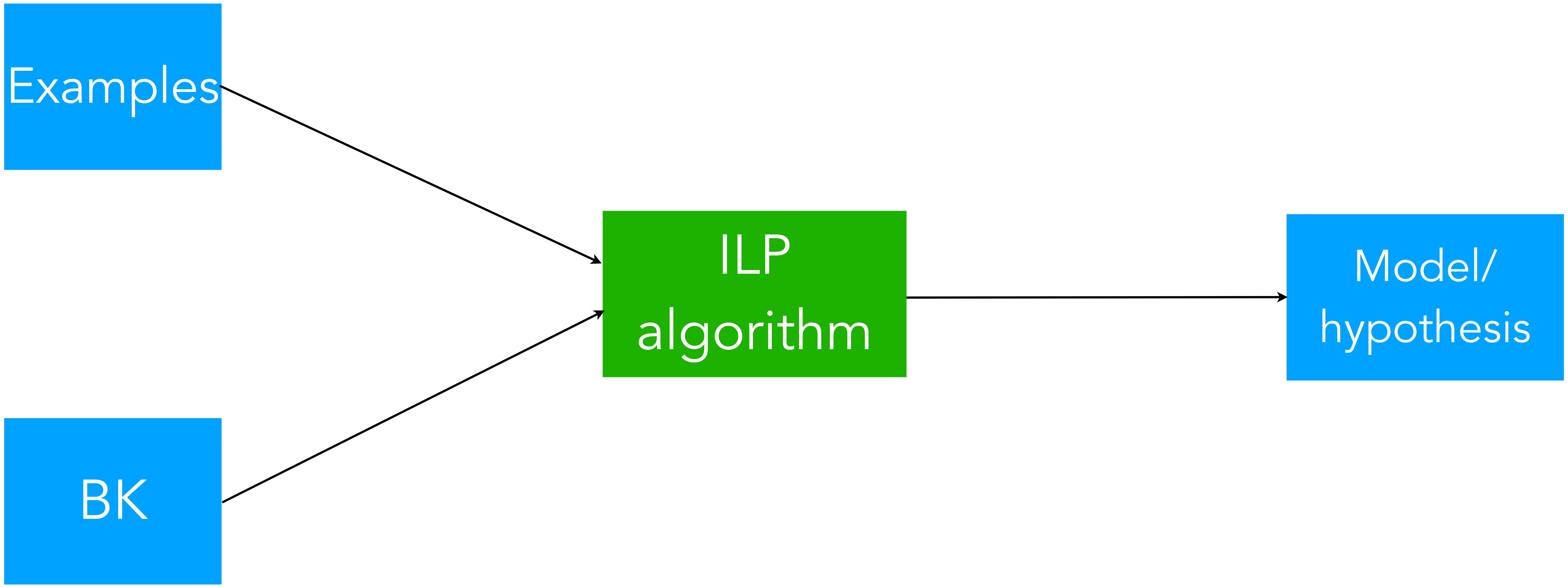
# Part I: Introduction

## What is ILP?

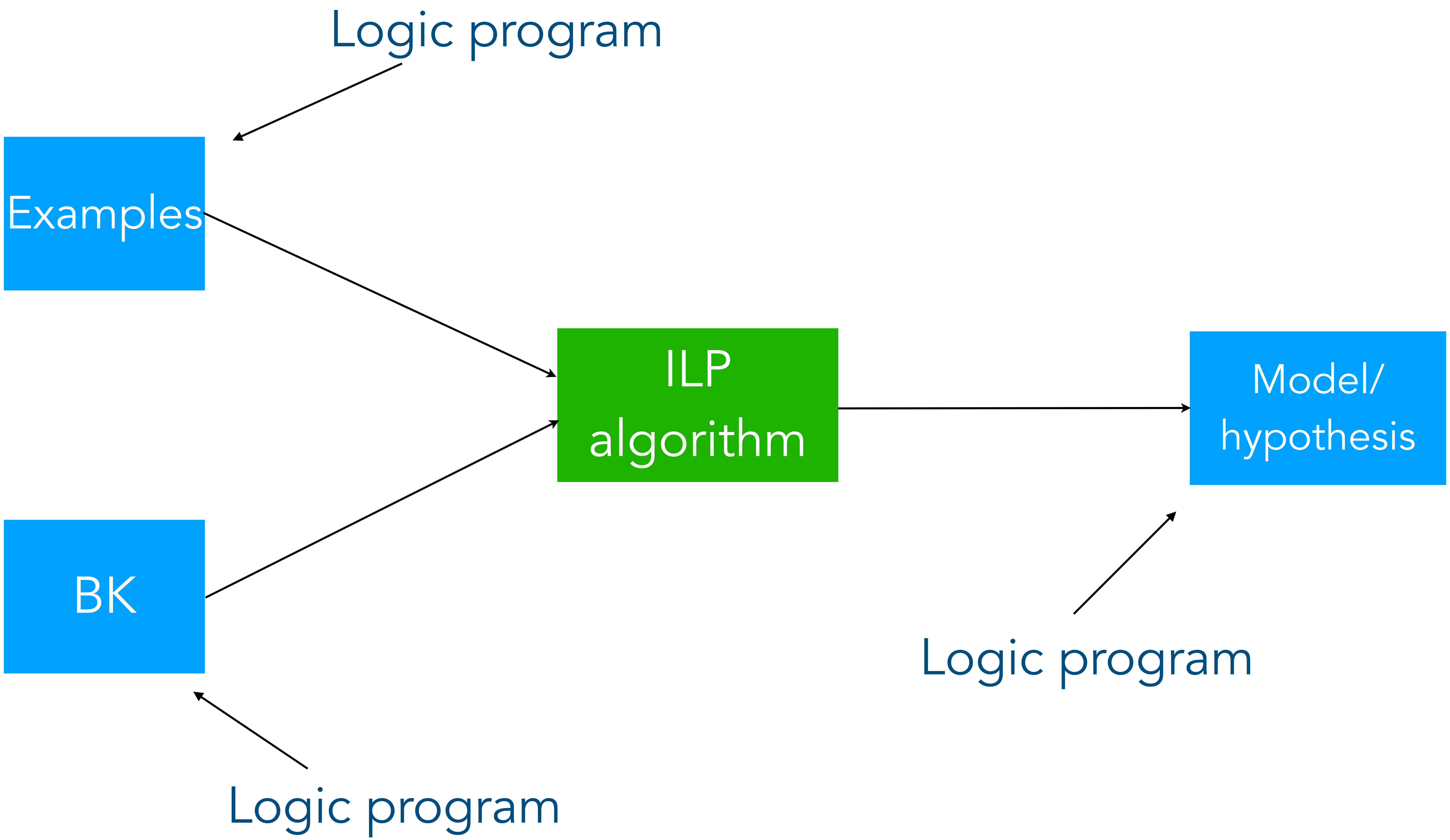
ML + logic

# ML

# ILP



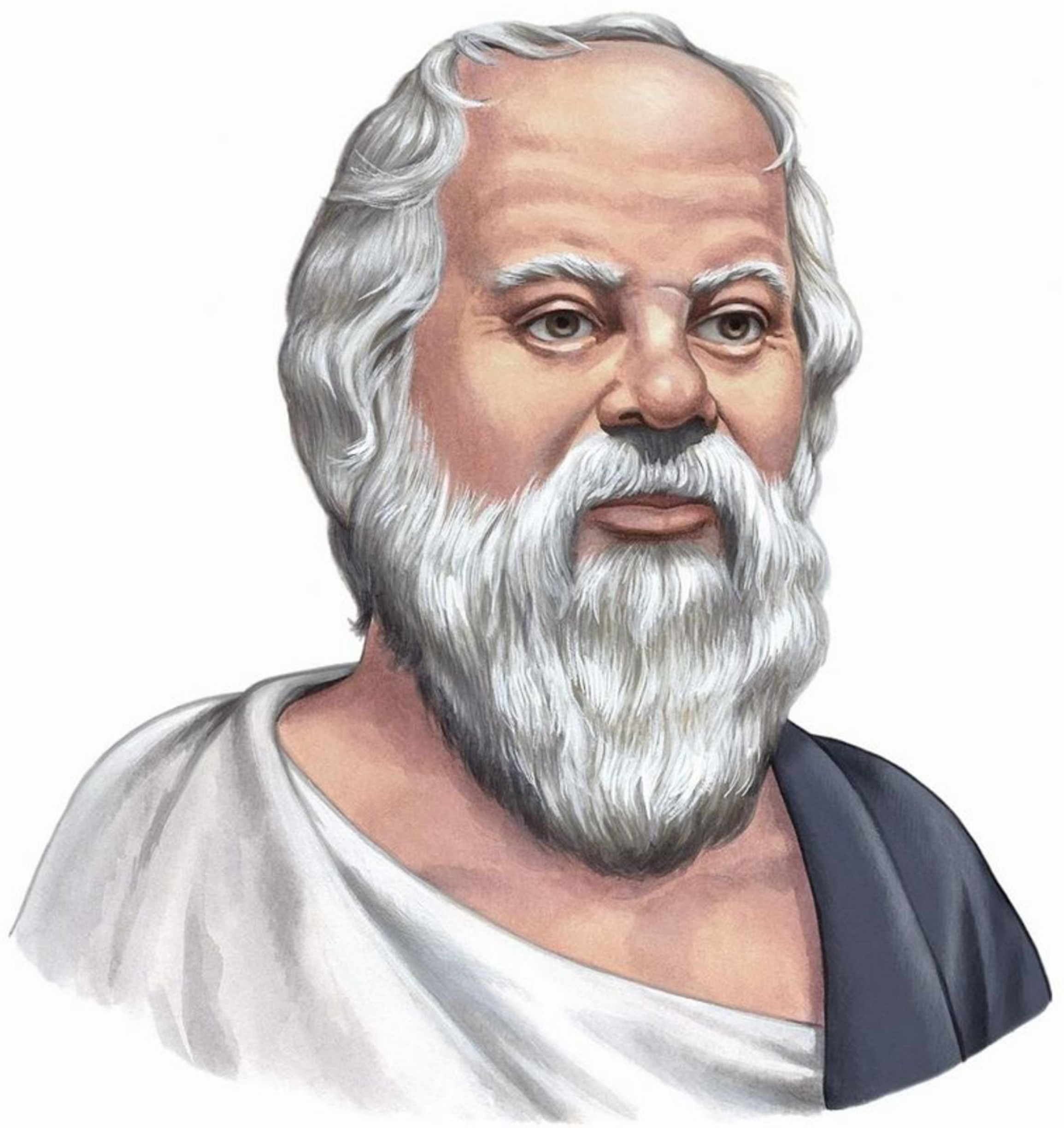
# ILP


Logic program
Examples
ILP algorithm
Model/ hypothesis
BK
Logic program
Logic program

# Program synthesis

# Logic refresher

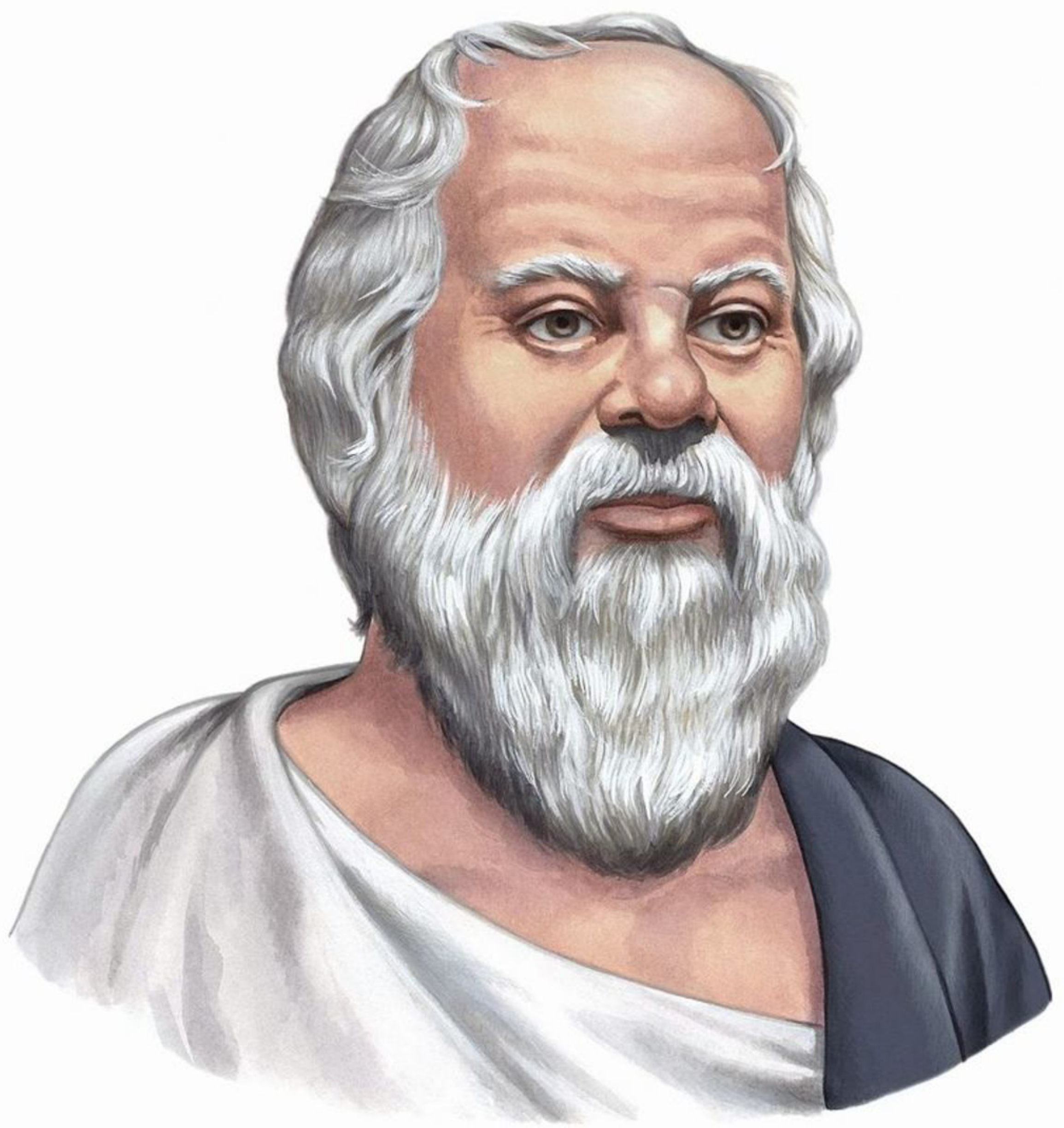
*Socrates is a man.*
*All men are mortal.*



*Socrates is a man.*
*All men are mortal.*

— — — — — — — — — — — — — — — — —

*Therefore, Socrates is mortal.*



*Socrates is a man.*
*All men are mortal.*
— — — — — — — — — — — — — — — —
*Therefore, Socrates is mortal.*

man(socrates).
$\forall \mathbf{A}$ man($\mathbf{A}$) $\rightarrow$ mortal($\mathbf{A}$).

atom

*Socrates is a man.*
*All men are mortal.*
*— — — — — — — — — — — — — — —*
*Therefore, Socrates is mortal.*

man(socrates).
∀**A** man(**A**) → mortal(**A**).

*rule*

*Socrates is a man.*
*All men are mortal.*
*— — — — — — — — — — — — — — — —*
*Therefore, Socrates is mortal.*

man(socrates).
$\forall \mathbf{A}$ man($\mathbf{A}$) → mortal($\mathbf{A}$).

*then this side is true*

*if this side is true*

| | |
|---|---|
| *Socrates is a man.* | man(socrates). |
| *All men are mortal.* | $\forall \mathbf{A}$ man($\mathbf{A}$) $\rightarrow$ mortal($\mathbf{A}$). |
| — — — — — — — — — — — — — — — — — | — — — — — — — — — — — — — — — — — |
| *Therefore, Socrates is mortal.* | mortal(socrates). |

$$\forall \mathbf{A}\ \mathrm{man}(\mathbf{A}) \rightarrow \mathrm{mortal}(\mathbf{A}).$$

$$\forall \mathbf{A}\ \mathrm{man}(\mathbf{A}) \rightarrow \mathrm{mortal}(\mathbf{A}).$$

$$\Downarrow$$

$$\mathrm{man}(\mathbf{A}) \rightarrow \mathrm{mortal}(\mathbf{A}).$$

*variables are all universally quantified*

$$\forall \mathbf{A}\ \mathrm{man}(\mathbf{A}) \rightarrow \mathrm{mortal}(\mathbf{A}).$$

$$\Downarrow$$

$$\mathrm{man}(\mathbf{A}) \rightarrow \mathrm{mortal}(\mathbf{A}).$$

$$\Downarrow$$

*flip the implication arrow direction*

$$\mathrm{mortal}(\mathbf{A}) \leftarrow \mathrm{man}(\mathbf{A}).$$

$\forall \mathbf{A}$ man($\mathbf{A}$) $\rightarrow$ mortal($\mathbf{A}$).

$\Downarrow$

man($\mathbf{A}$) $\rightarrow$ mortal($\mathbf{A}$).

$\Downarrow$

mortal($\mathbf{A}$) $\leftarrow$ man($\mathbf{A}$).

$\Downarrow$

*replace the arrow with :-* mortal(A):- man(A).

$\forall \mathbf{A}\ man(\mathbf{A}) \rightarrow mortal(\mathbf{A}).$

$\downdownarrows$

$man(\mathbf{A}) \rightarrow mortal(\mathbf{A}).$

$\downdownarrows$

$mortal(\mathbf{A}) \leftarrow man(\mathbf{A}).$

$\downdownarrows$

```
mortal(A):- man(A).
```

*valid Prolog / Datalog / ASP rule*

$\forall \mathbf{A}. \forall \mathbf{B}\ \text{knows}(\mathbf{A},\mathbf{B}) \wedge \text{rich}(\mathbf{B}) \wedge \text{famous}(\mathbf{B}) \rightarrow \text{happy}(\mathbf{A}).$

$$\forall \mathbf{A}.\forall \mathbf{B}\ \text{knows}(\mathbf{A},\mathbf{B}) \wedge \text{rich}(\mathbf{B}) \wedge \text{famous}(\mathbf{B}) \rightarrow \text{happy}(\mathbf{A}).$$

$$\Downarrow$$

$$\text{knows}(\mathbf{A},\mathbf{B}) \wedge \text{rich}(\mathbf{B}) \wedge \text{famous}(\mathbf{B}) \rightarrow \text{happy}(\mathbf{A}).$$

$$\forall \mathbf{A}. \forall \mathbf{B}\ \text{knows}(\mathbf{A},\mathbf{B}) \wedge \text{rich}(\mathbf{B}) \wedge \text{famous}(\mathbf{B}) \rightarrow \text{happy}(\mathbf{A}).$$

$$\downdownarrows$$

$$\text{knows}(\mathbf{A},\mathbf{B}) \wedge \text{rich}(\mathbf{B}) \wedge \text{famous}(\mathbf{B}) \rightarrow \text{happy}(\mathbf{A}).$$

$$\downdownarrows$$

$$\text{happy}(\mathbf{A}) \leftarrow \text{knows}(\mathbf{A},\mathbf{B}) \wedge \text{rich}(\mathbf{B}) \wedge \text{famous}(\mathbf{B}).$$

$$\forall \mathbf{A}.\forall \mathbf{B}\ \mathrm{knows}(\mathbf{A},\mathbf{B}) \wedge \mathrm{rich}(\mathbf{B}) \wedge \mathrm{famous}(\mathbf{B}) \rightarrow \mathrm{happy}(\mathbf{A}).$$

$$\Downarrow$$

$$\mathrm{knows}(\mathbf{A},\mathbf{B}) \wedge \mathrm{rich}(\mathbf{B}) \wedge \mathrm{famous}(\mathbf{B}) \rightarrow \mathrm{happy}(\mathbf{A}).$$

$$\Downarrow$$

$$\mathrm{happy}(\mathbf{A}) \leftarrow \mathrm{knows}(\mathbf{A},\mathbf{B}) \wedge \mathrm{rich}(\mathbf{B}) \wedge \mathrm{famous}(\mathbf{B}).$$

$$\Downarrow$$

```
happy(A):- knows(A,B), rich(B), famous(B).
```

What does this have to do with programming?

# Logic programs

```
empty([]).
head([H|_],H).
tail([_|T],T).
```

# Logic programs

```
empty([]).
head([H|_],H).
tail([_|T],T).
```

```
?- head([h,e,l,l,o],X).
X = h.

?- tail([h,e,l,l,o],X).
X = [e, l, l, o].
```

# Logic programs

```
empty([]).
head([H|_],H).
tail([_|T],T).
```

```
?- tail([h,e,l,l,o],[c,a,t]).
false.
```

# Logic programs

```
empty([]).
head([H|_],H).
tail([_|T],T).
```

```
[?- tail(X,[c,a,t]).
X = [_9930, c, a, t].
```

# Logic programs

```
length([],0).
length([H|T],N2):-
    length(T,N1),
    N2 is N1+1.
```

# Logic programs

```
length([],0).
length([H|T],N2):-
    length(T,N1),
    N2 is N1+1.
```

```
?- length([c,a,t],X).
X = 3.
```

# Logic programs

```
length([],0).
length([H|T],N2):-
    length(T,N1),
    N2 is N1+1.
```

```
?- length(X,4).
X = [_6240, _6246, _6252, _6258].
```

Any questions?

# Why logic programs?

Relational

Declarative

Interpretable

Universal

# Relational data


Edgware Road
Marylebone
St John's Wood
Euston
Edgware Road
Baker Street
Great Portland Street
Euston Square
Warren Street
Regent's Park
Goodge Street
Russell Square
Bayswater
Bond Street
Oxford Circus
Marble Arch
Notting Hill Gate
Tottenham Court Road
Holborn
Covent Garden
Lancaster Gate
Queensway
Green Park
Leicester Square
Hyde Park Corner
Piccadilly Circus
High Street Kensington
Knightsbridge
Charing Cross
Gloucester Road
Victoria
Westminster
Embankment
Earl's Court
South Kensington
Sloane Square
St James's Park
Temple


```
edge(oxford_circus, bond_street).
edge(oxford_circus, piccadilly_circus).
edge(south_kensington, gloucester_road).
```


```
connected(S1,S2):- edge(S1,S2).
connected(S1,S2):- edge(S1,S3), connected(S3,S2).
```

# Declarative

Say what you what to happen, not how it should happen

```
zendo(A):- piece(A,C),contact(C,B),size(B,E),
    small(E),color(B,D),not_blue(D).
```

Can execute/evaluate the rule in any order. If any literal fails, the whole rule fails.

```
zendo(A):- piece(A,C),contact(C,B),size(B,E),
        small(E),color(B,D),not_blue(D).
zendo(A):- piece(A,C),contact(C,B),size(B,E),
        small(E),color(B,D),not_red(D).
```

If any rule succeeds, the whole program succeeds.

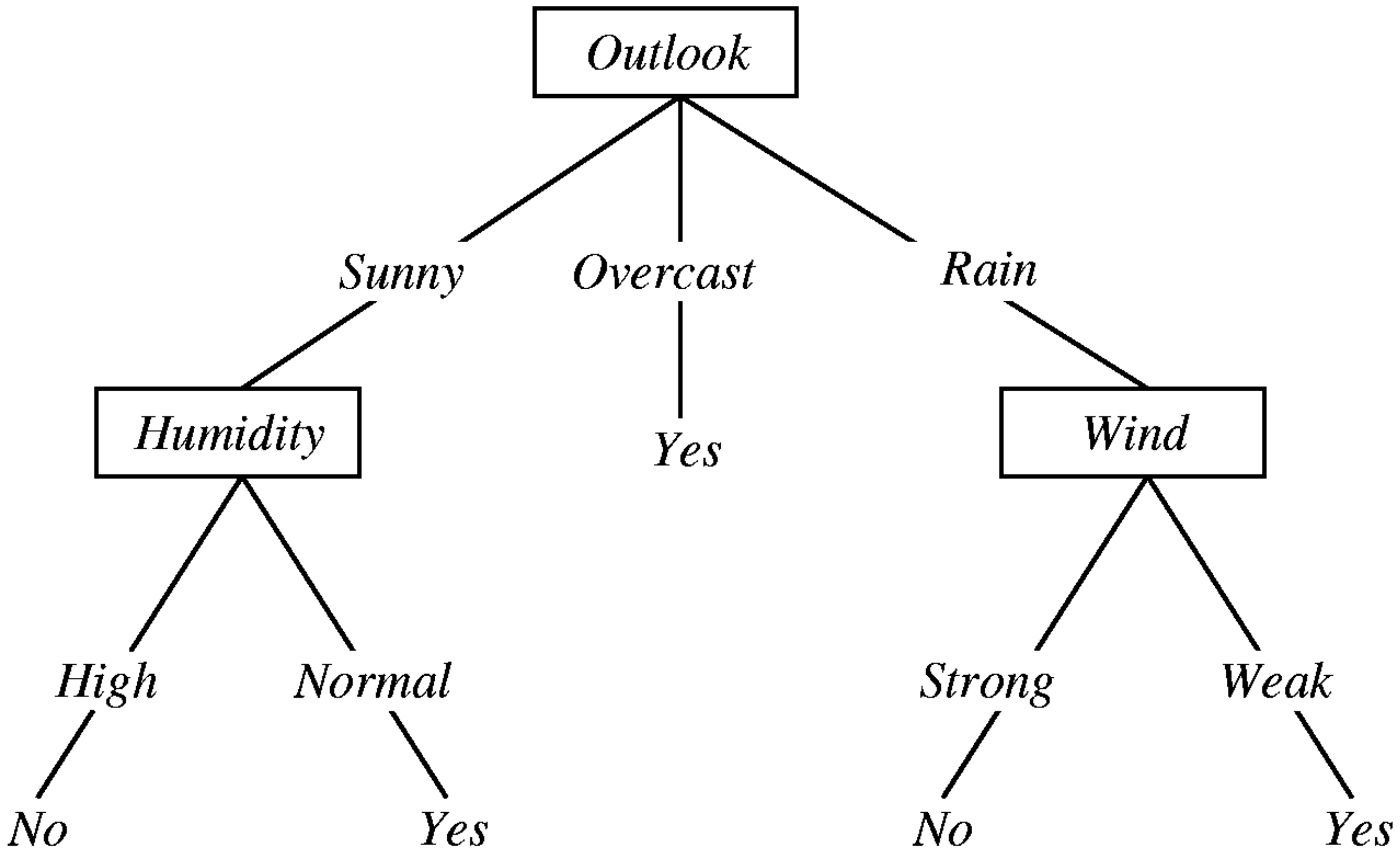
# Interpretable

```
zendo(A):- piece(A,C),contact(C,B),size(B,E),
        small(E),color(B,D),not_blue(D).
zendo(A):- piece(A,C),contact(C,B),size(B,E),
        small(E),color(B,D),not_red(D).
```

You can understand this program without having to take a course in logic programming!

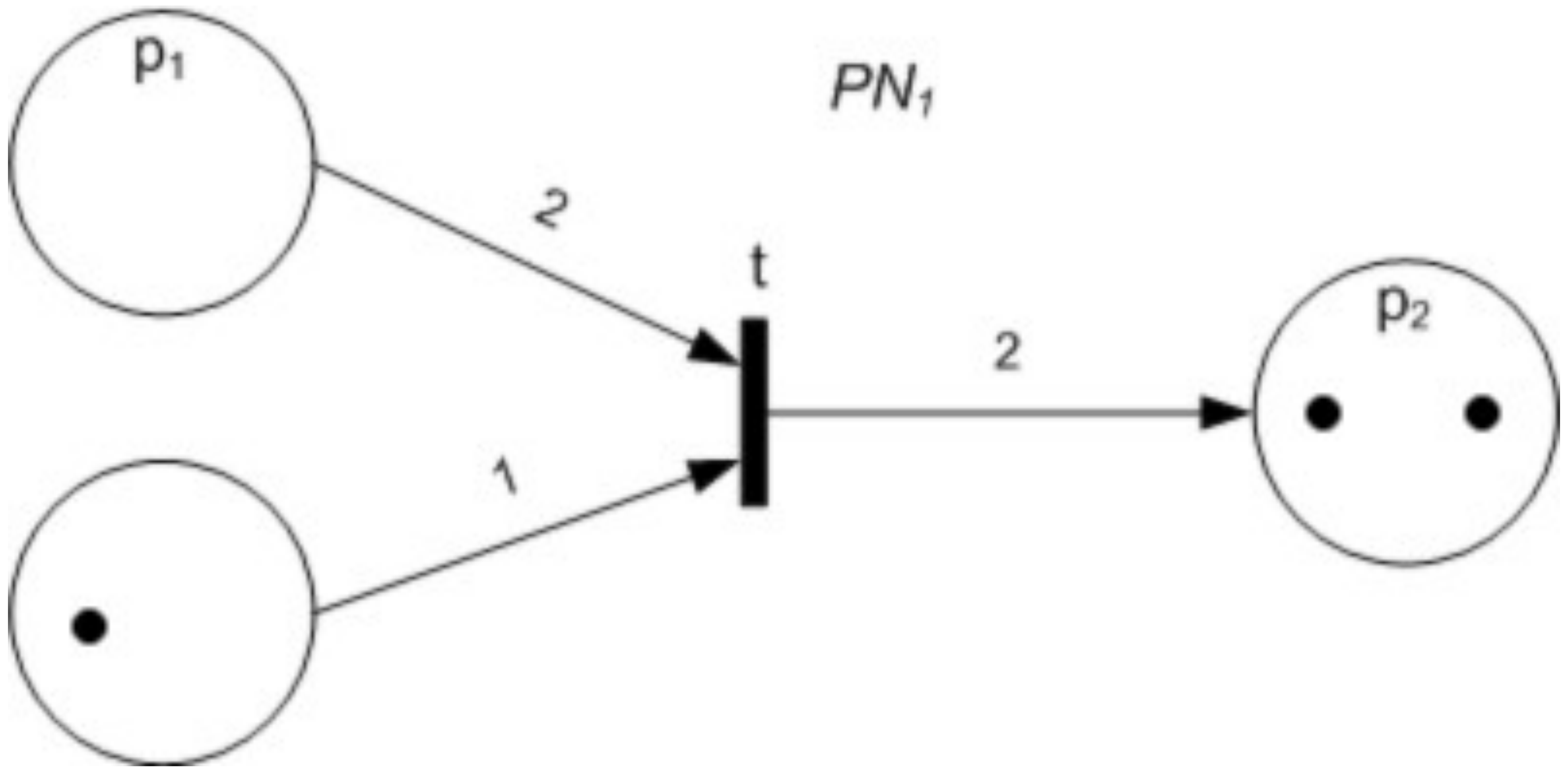
# Universal

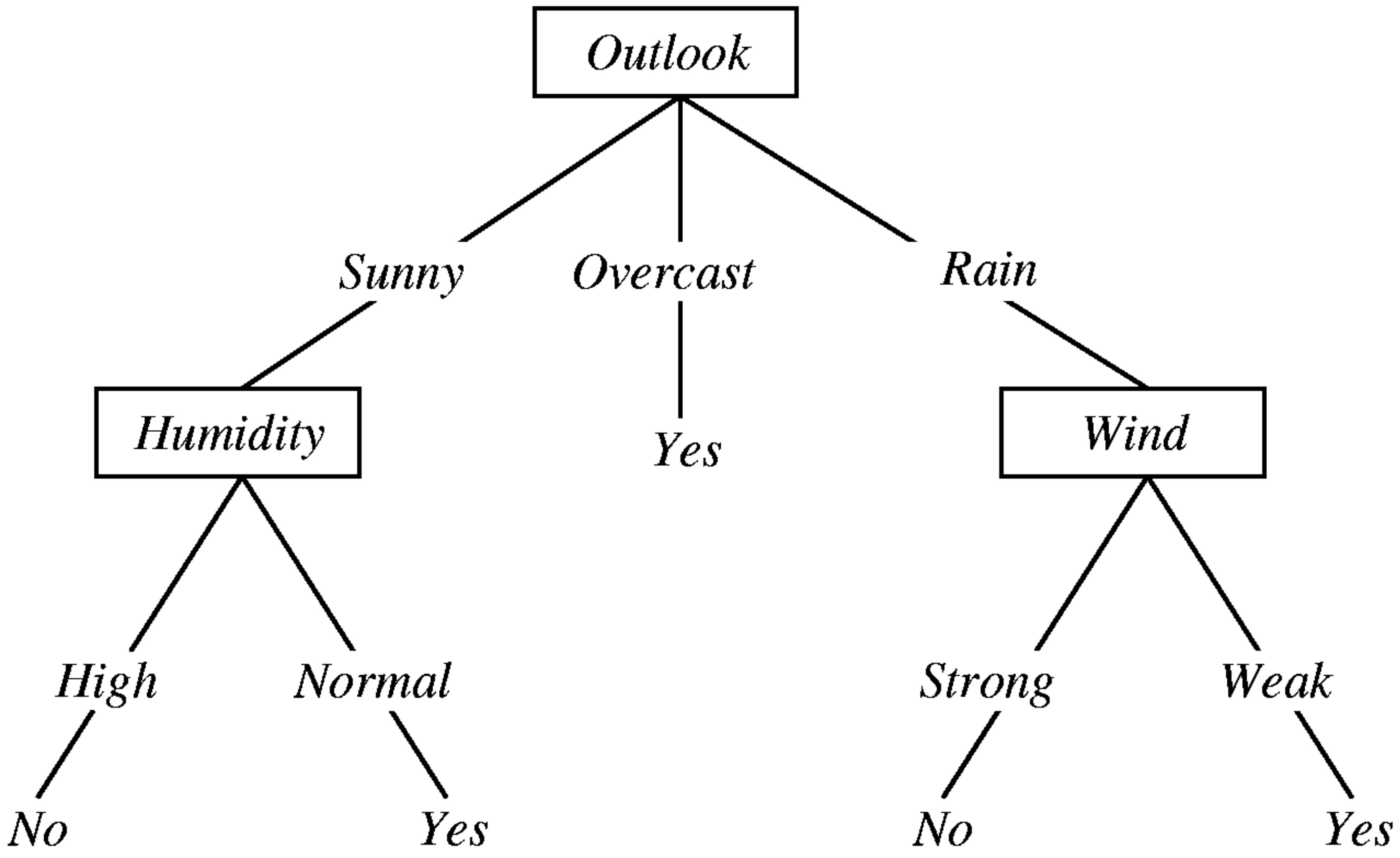
# Universal



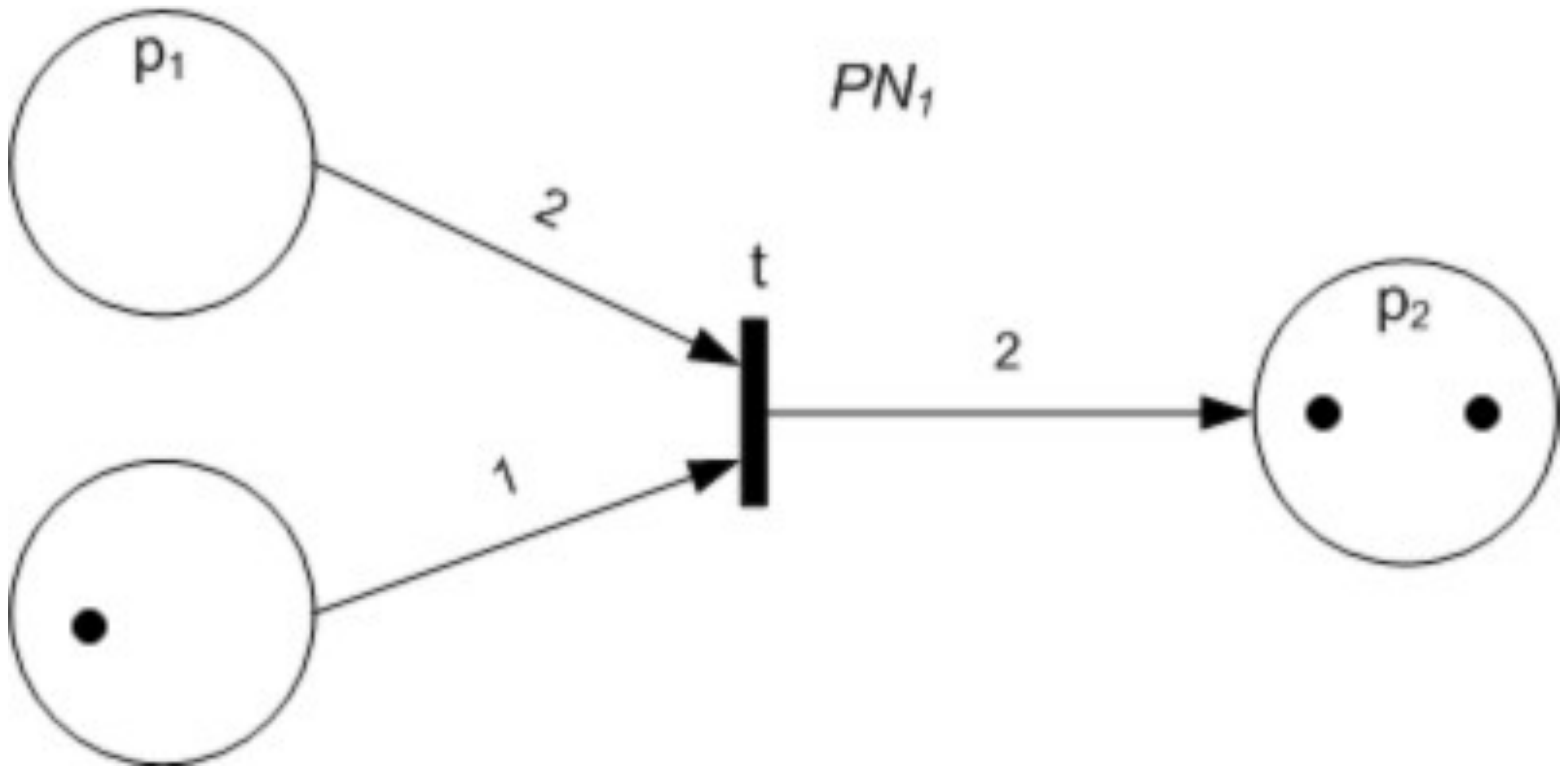
Outlook
Sunny
Overcast
Rain
Humidity
Yes
Wind
High
Normal
Strong
Weak
No
Yes
No
Yes

# Universal

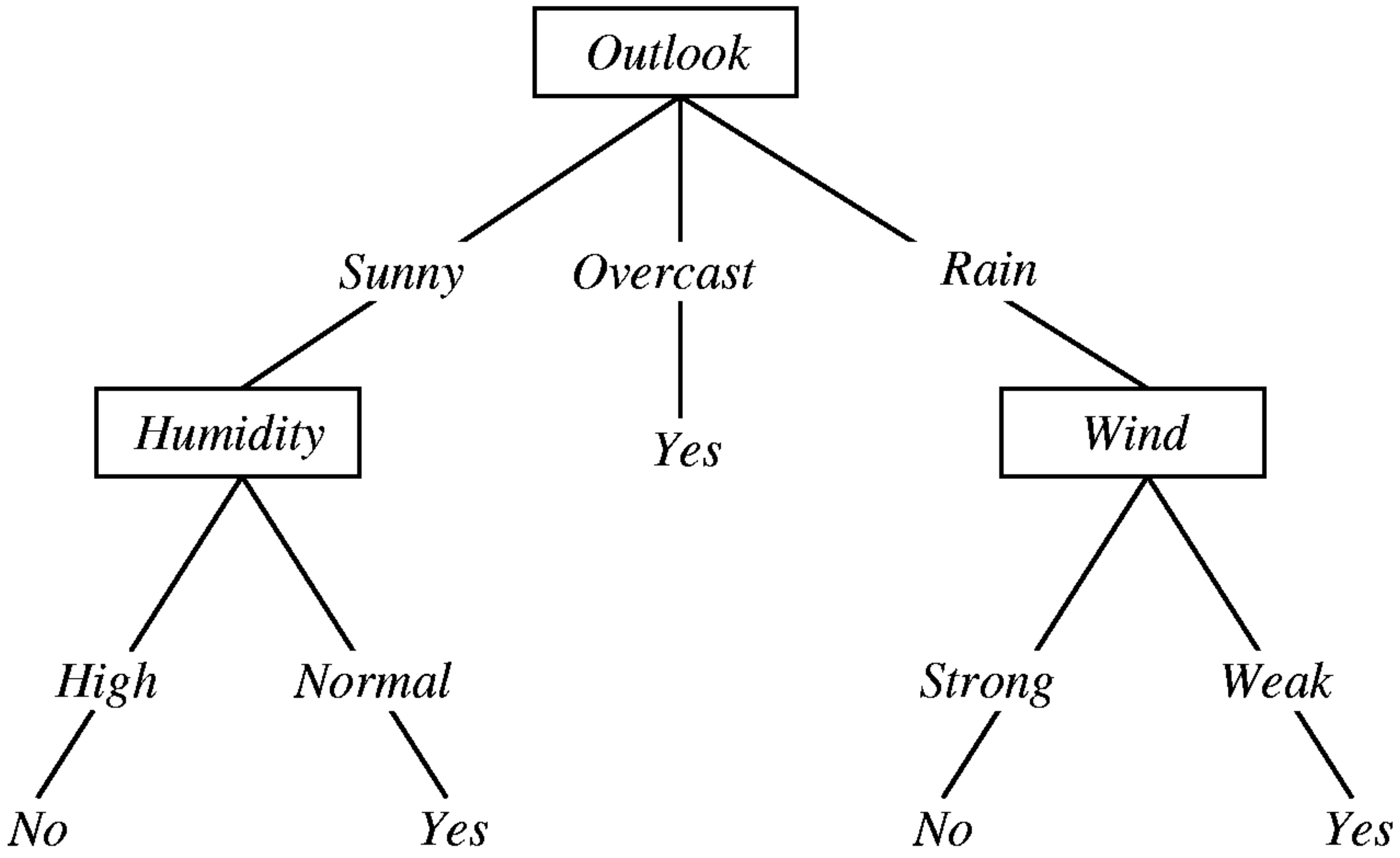

p1
PN1
2
t
p2
1
2


Outlook
Sunny
Overcast
Rain
Humidity
Yes
Wind
High
Normal
Strong
Weak
No
Yes
No
Yes

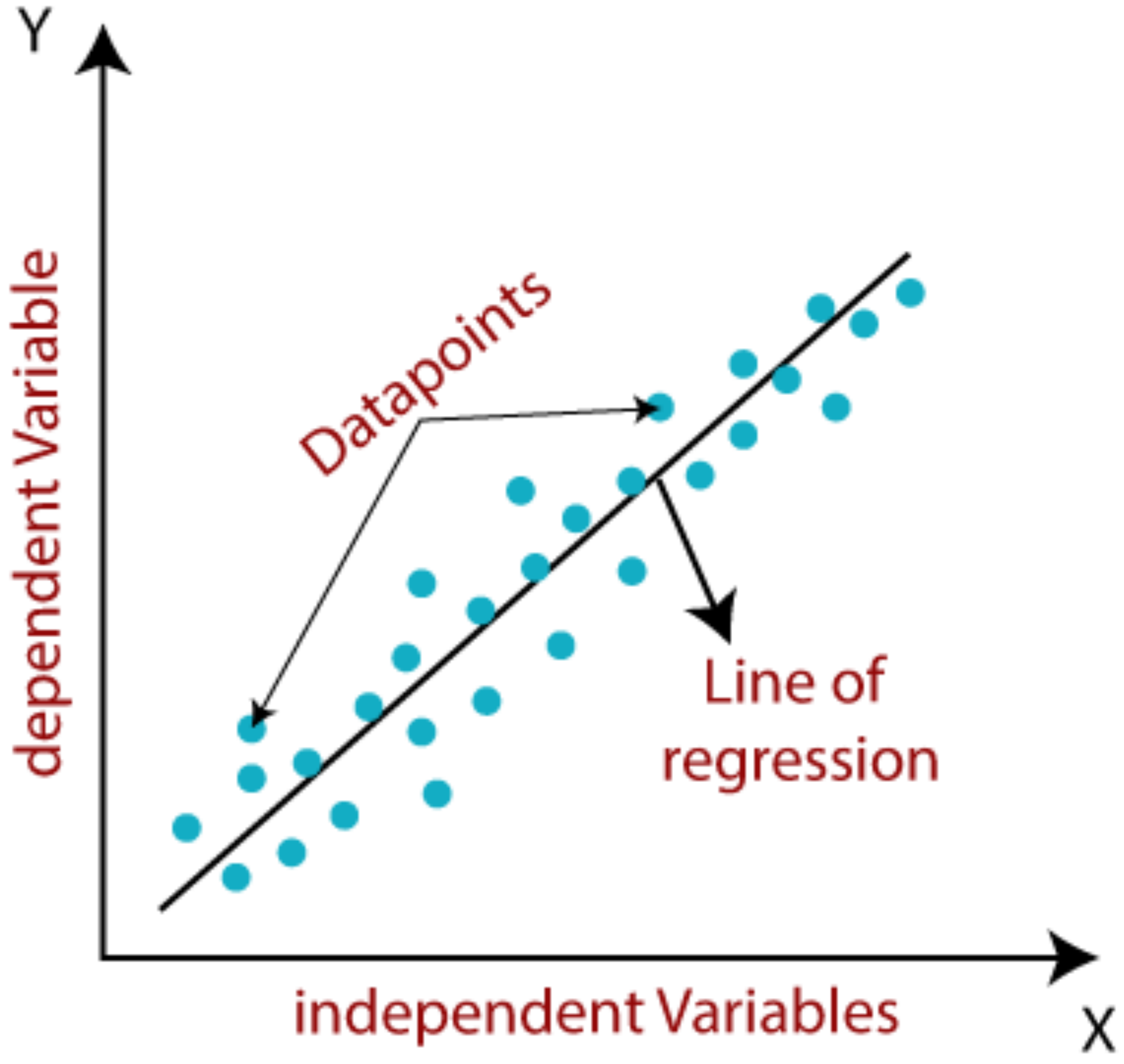
# Universal



p1
PN1
2
t
1
2
p2

Outlook
Sunny
Overcast
Rain
Humidity
Yes
Wind
High
Normal
Strong
Weak
No
Yes
No
Yes

Y
dependent Variable
Datapoints
Line of regression
independent Variables
X

# Why not logic programs?

Less control

Few people use them

Iffy software

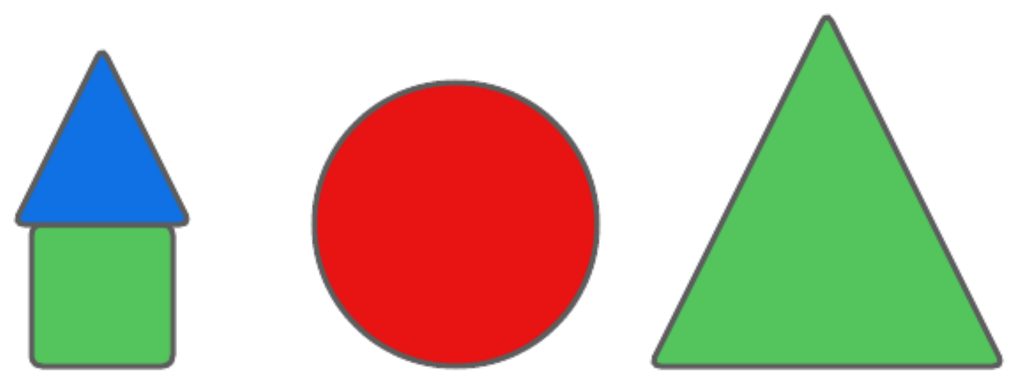
# Questions?

# Break time

# Part I: Introduction

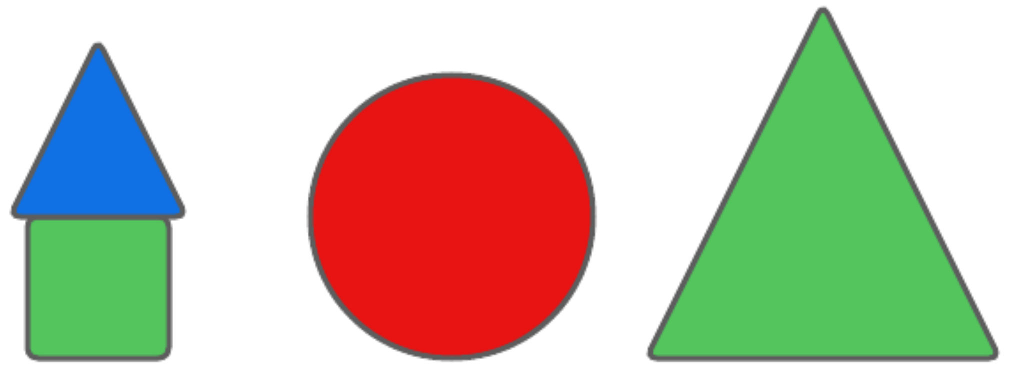
## What is ILP?

# Zendo in DT



# Zendo in DT

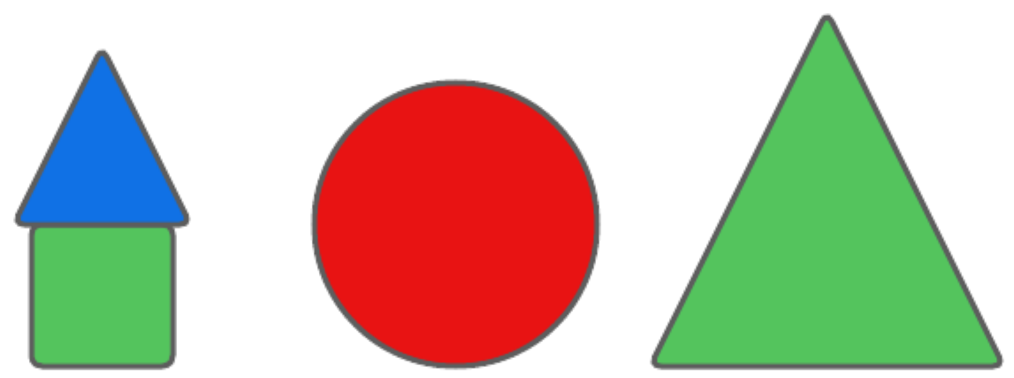
p1_notblue



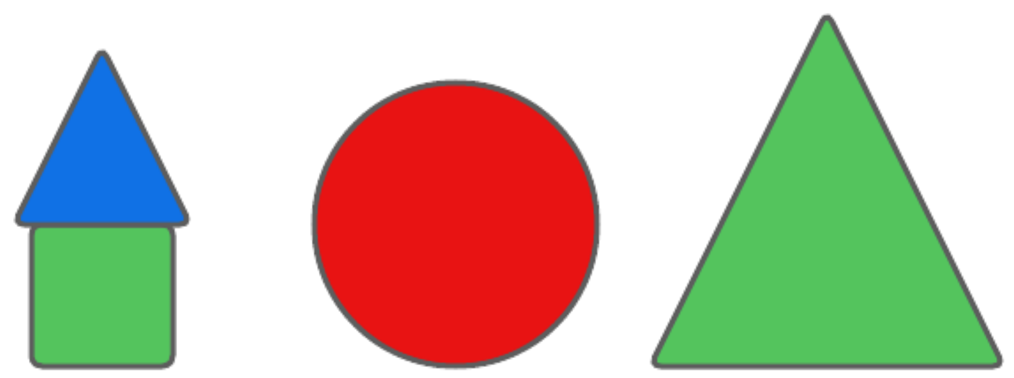
# Zendo in DT

yes

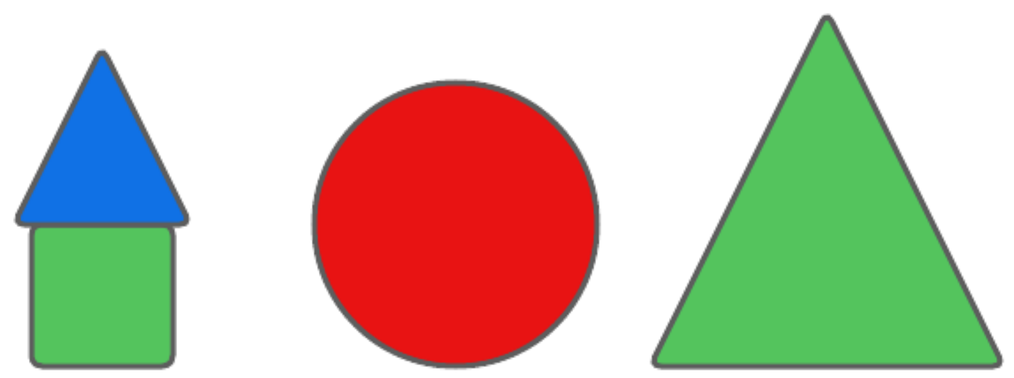
p1_notblue



# Zendo in DT

# Zendo in DT



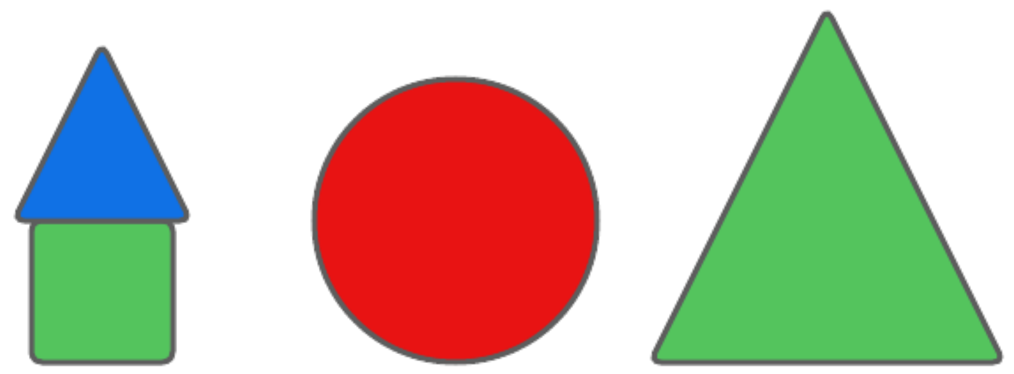
p1_notblue

yes

no

yes

# Zendo in DT



p1_notblue

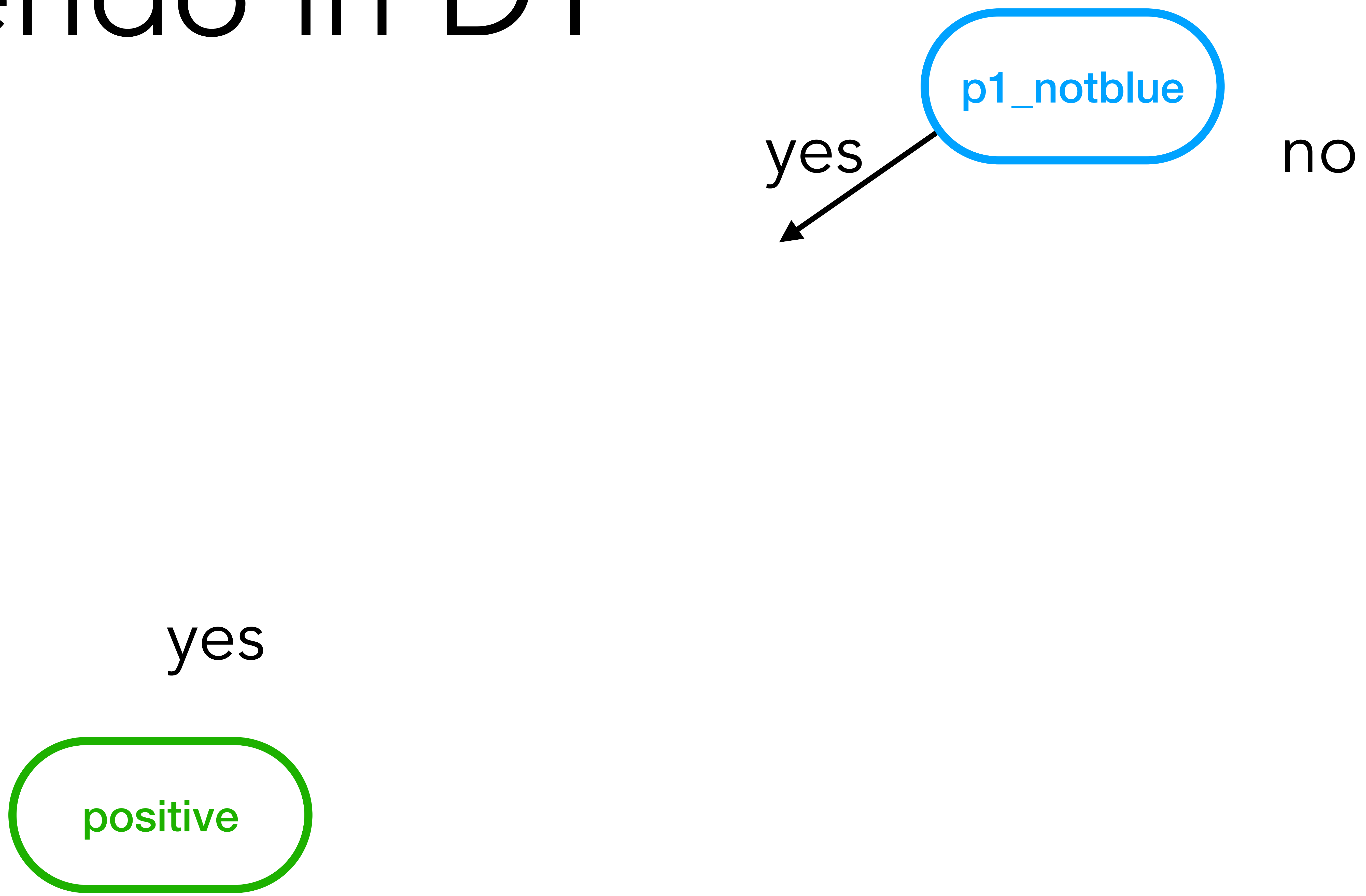
yes

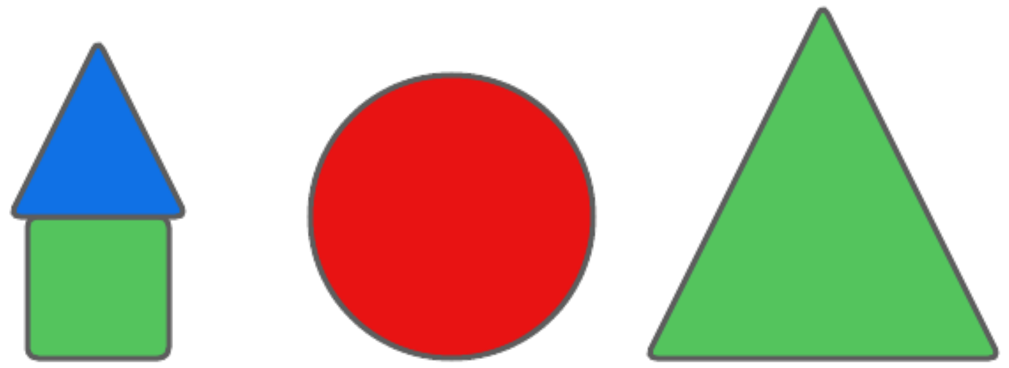
no

yes

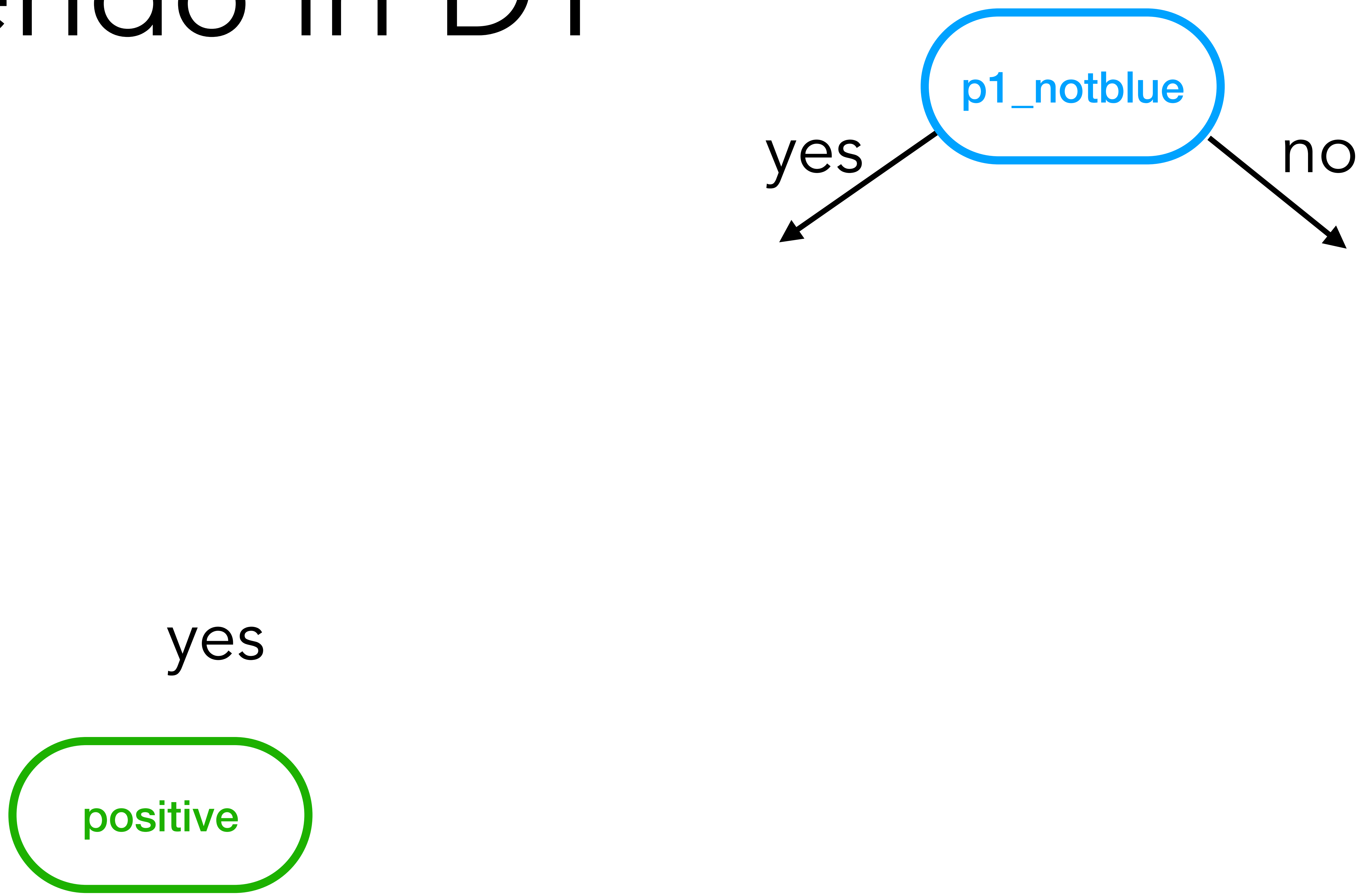
positive

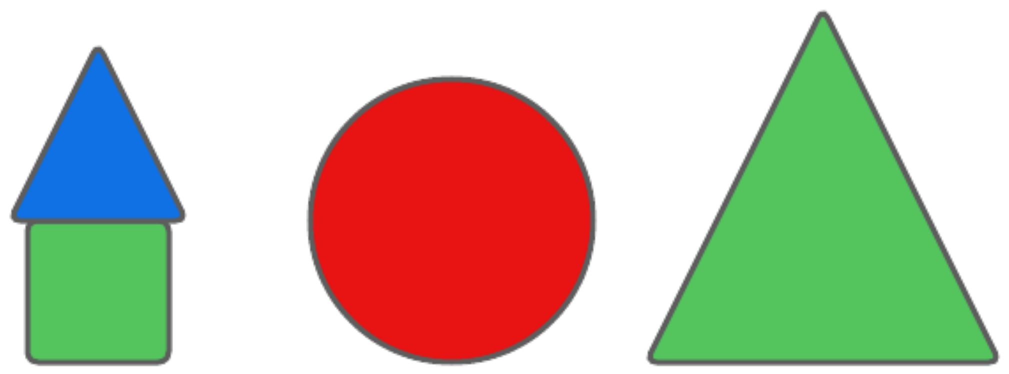
# Zendo in DT



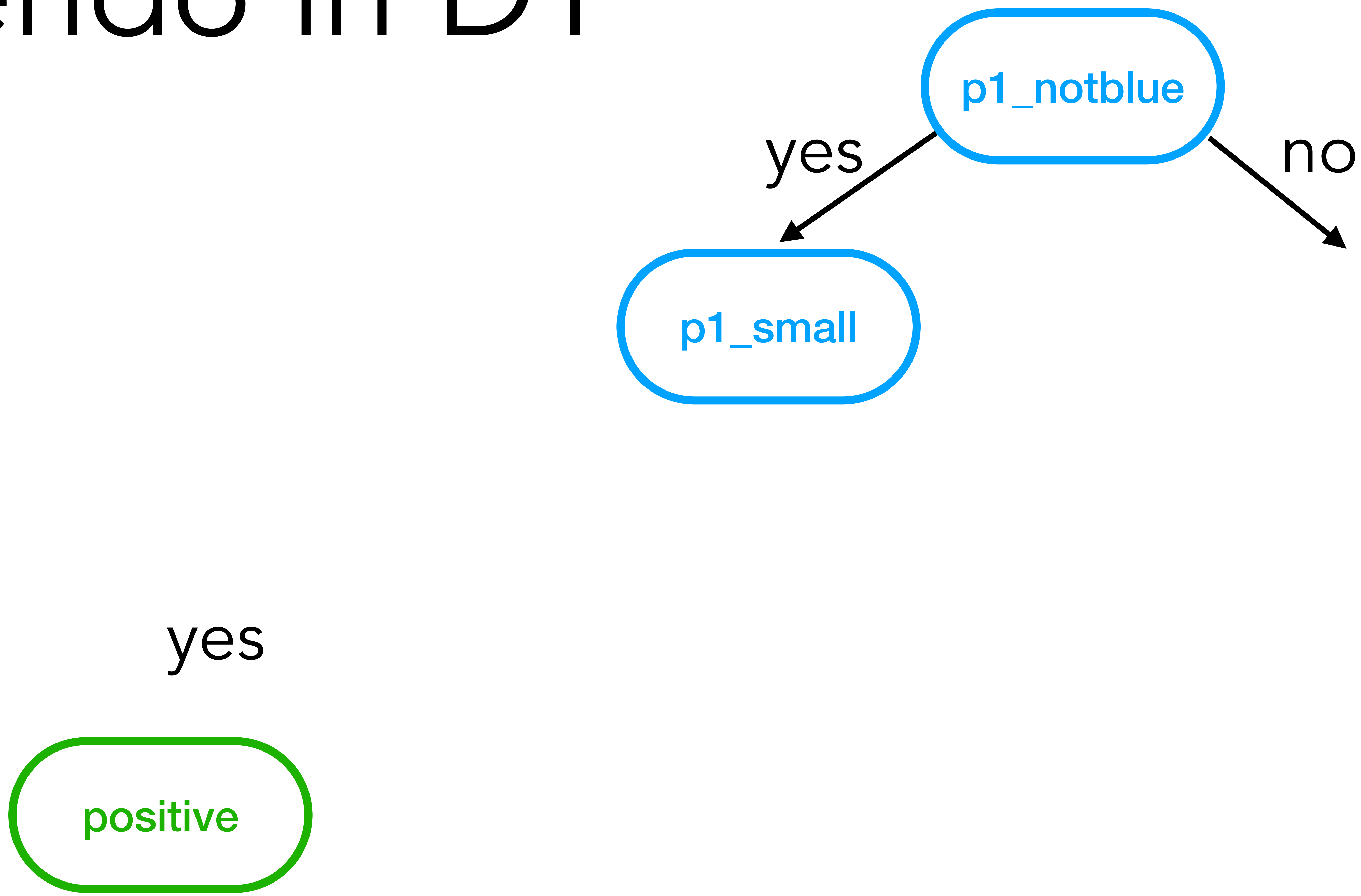



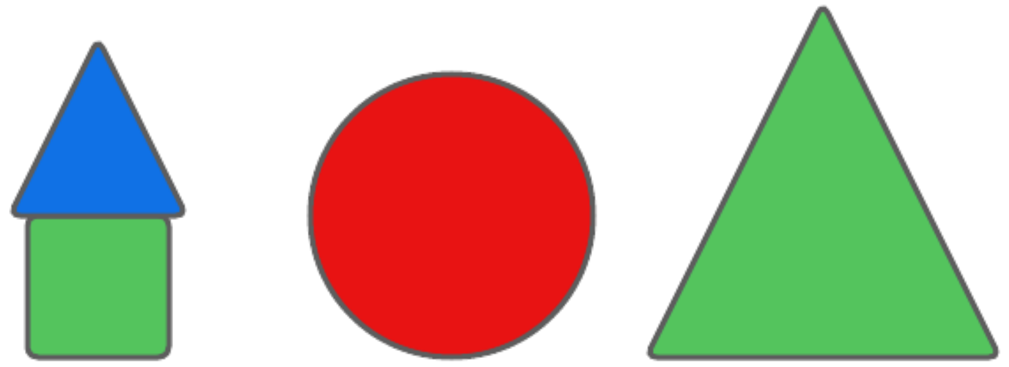
# Zendo in DT



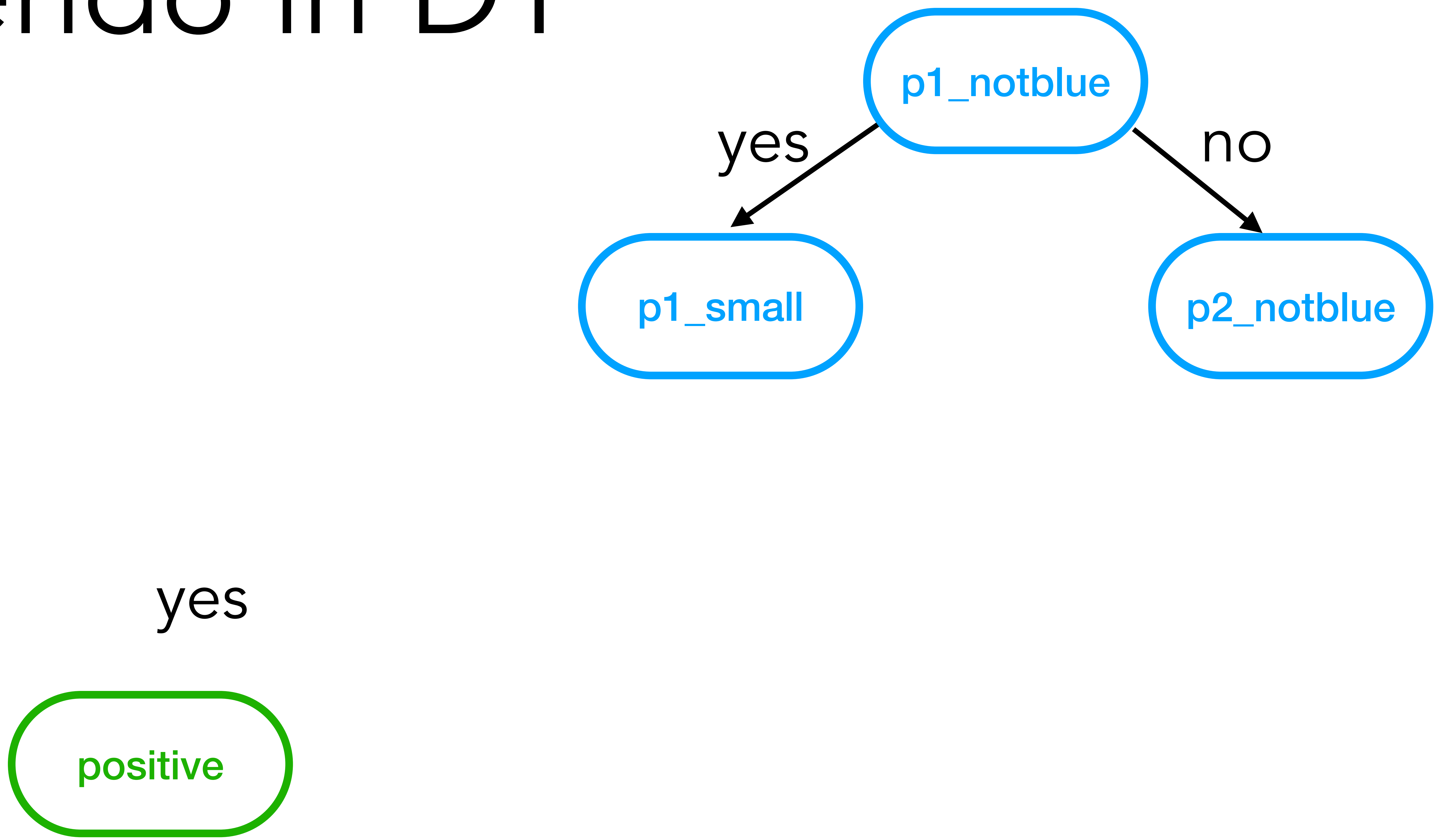



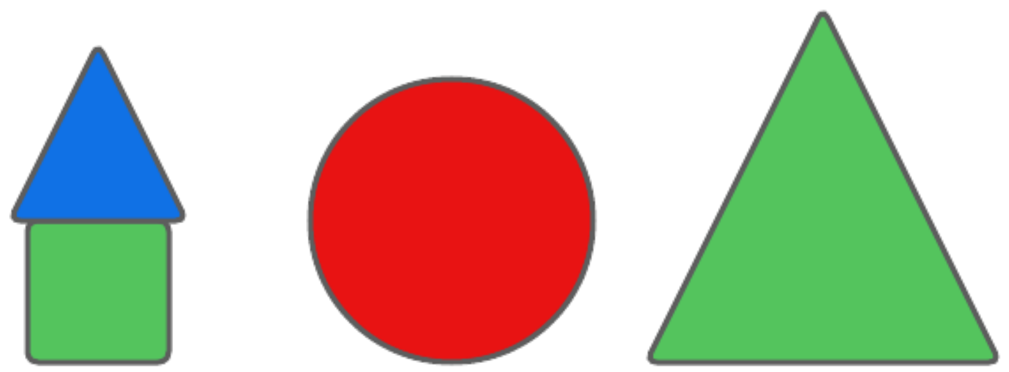
# Zendo in DT



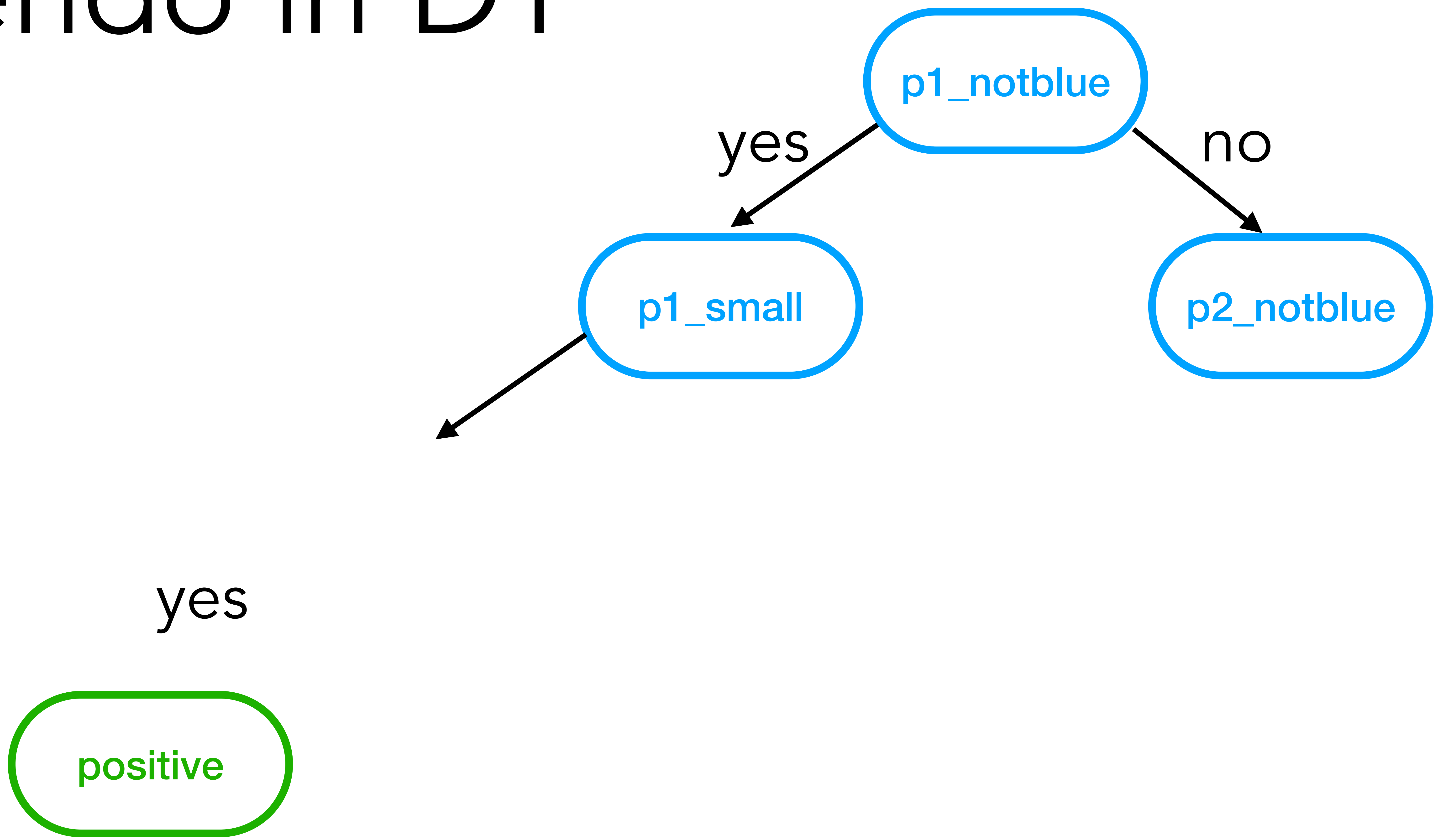



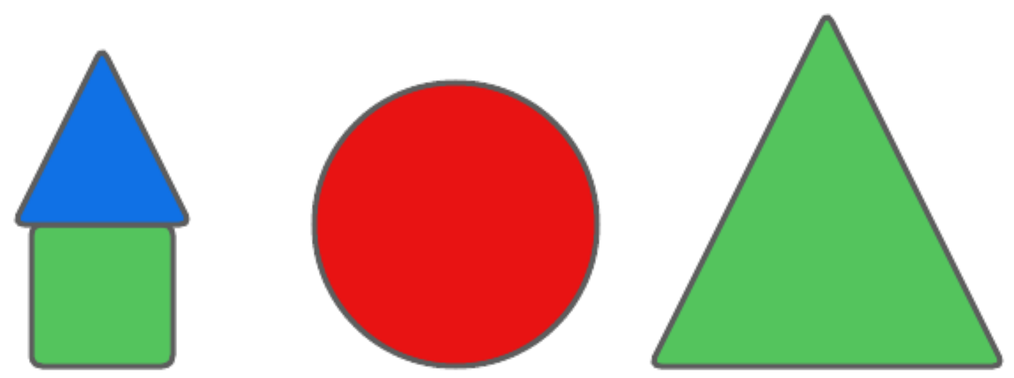
# Zendo in DT



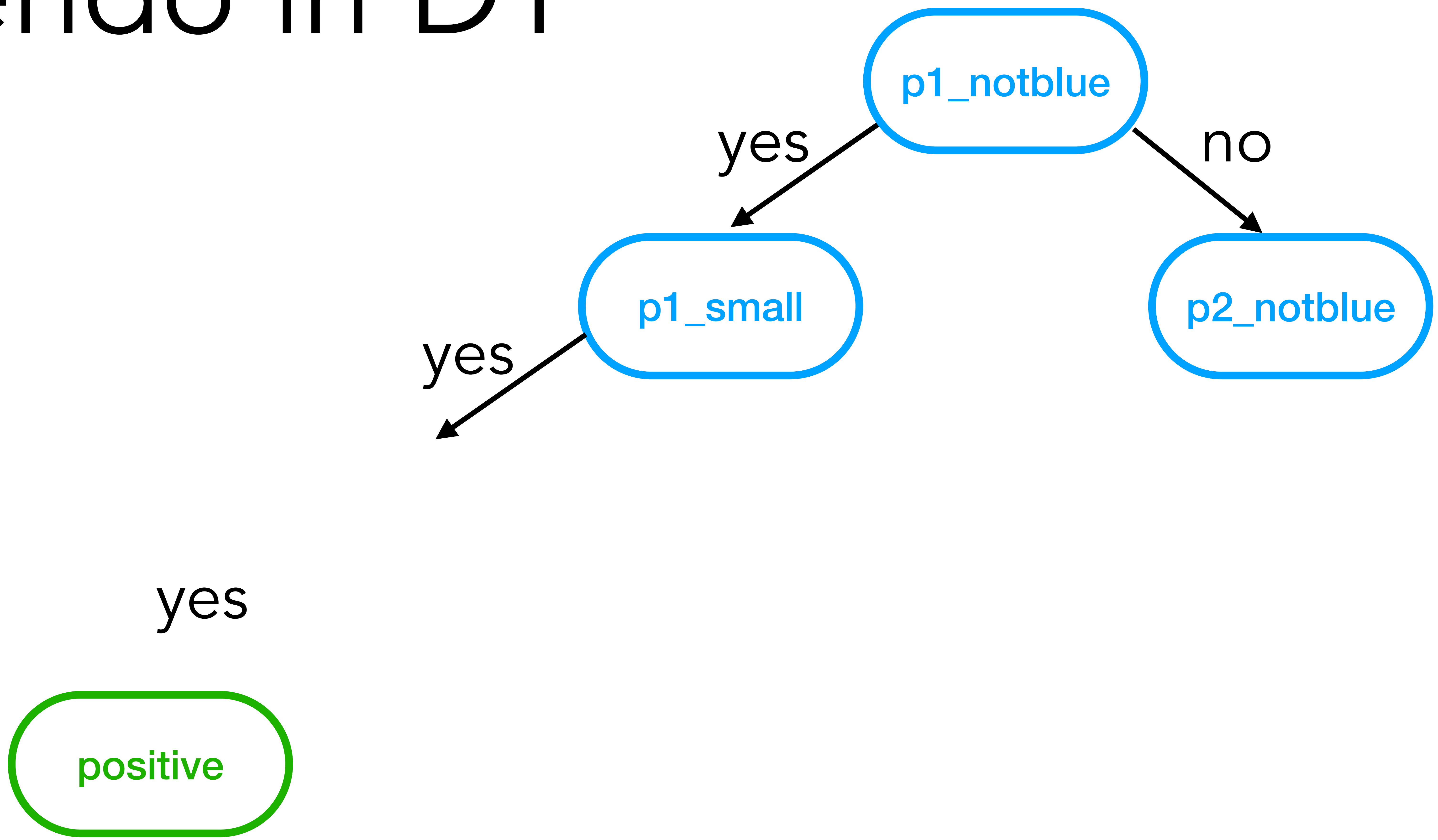



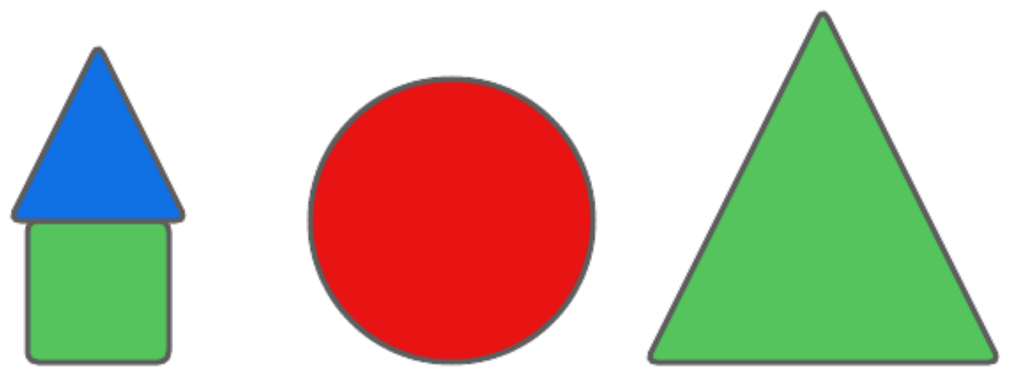
# Zendo in DT



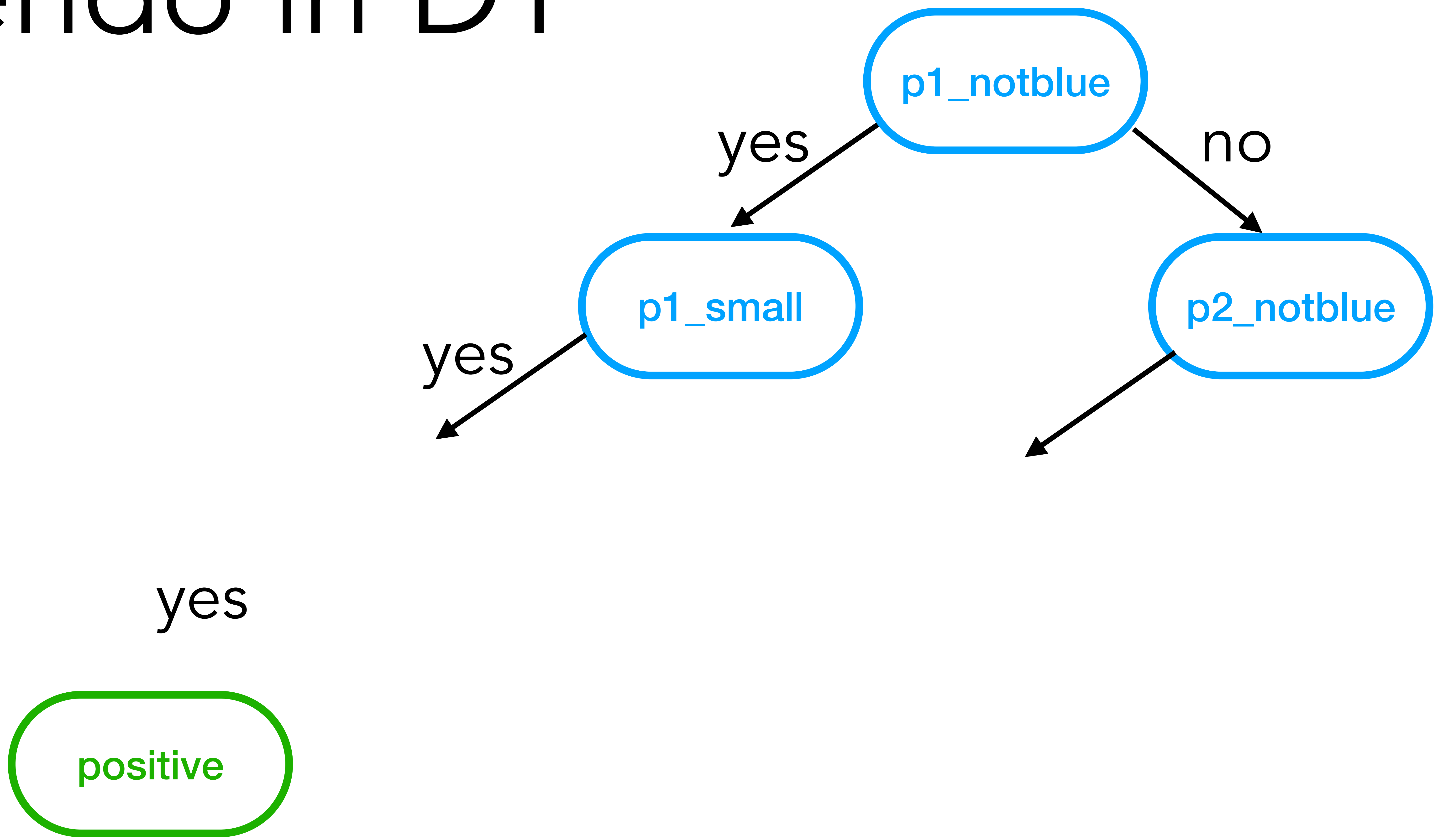



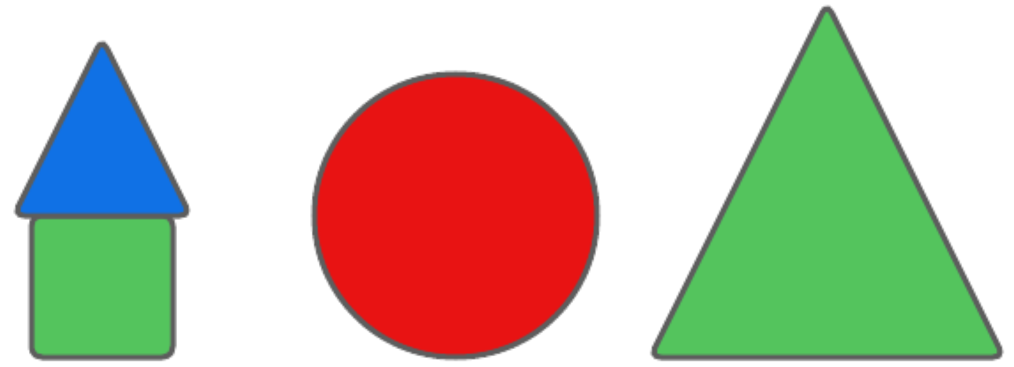
# Zendo in DT



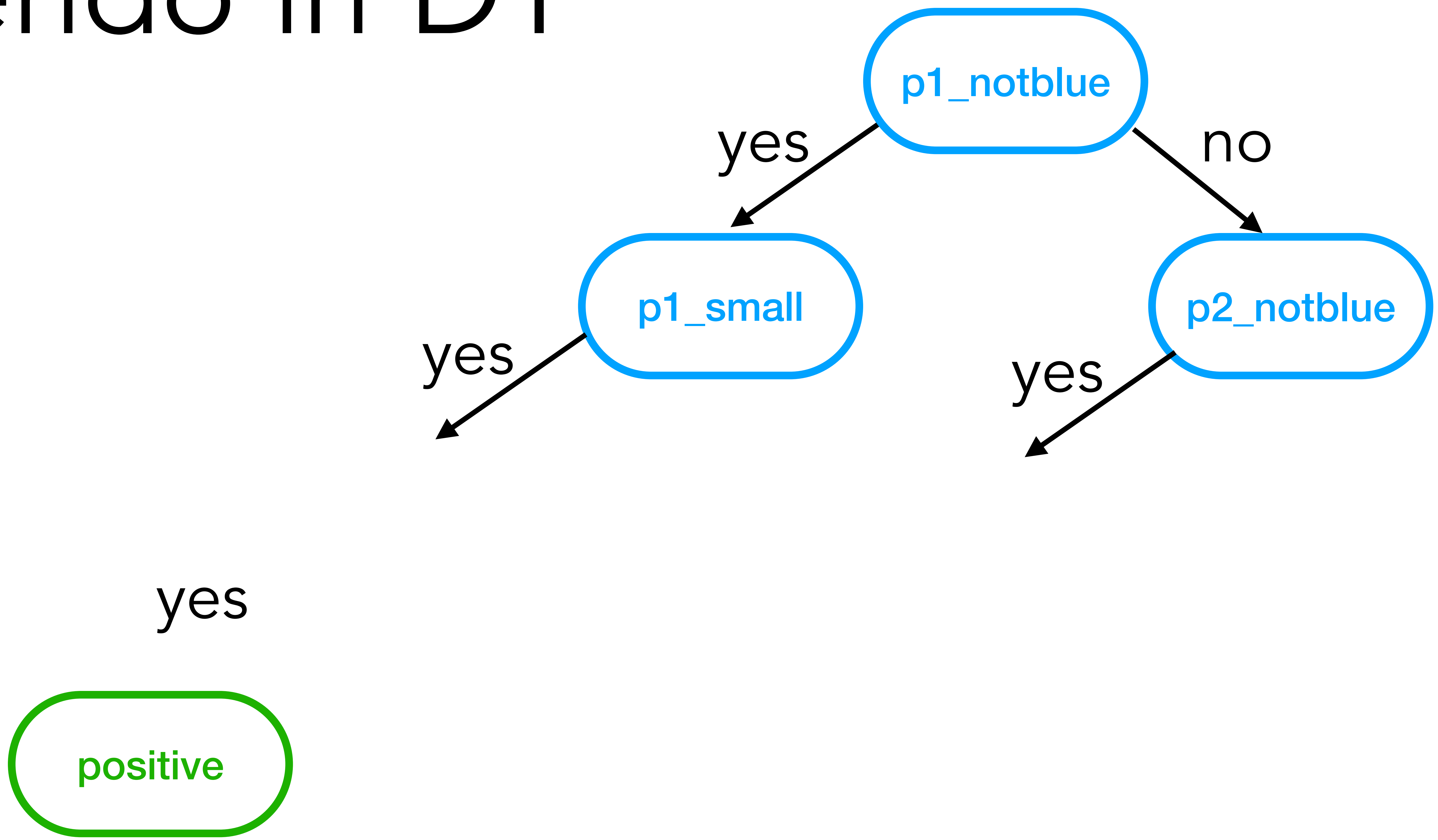



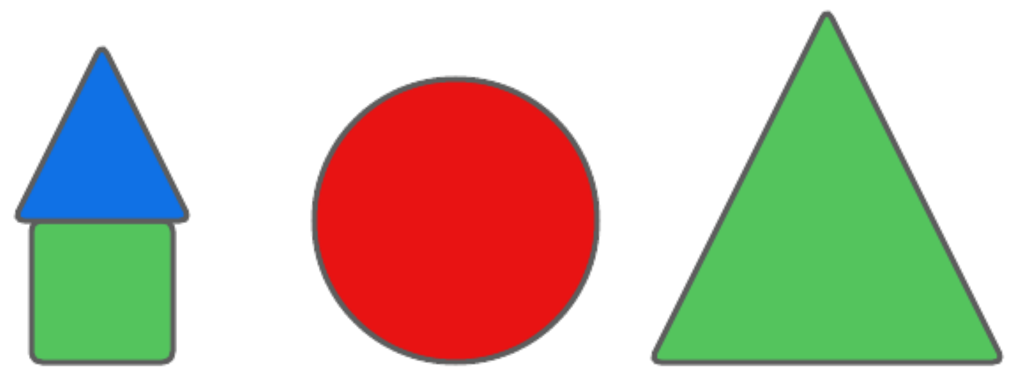
# Zendo in DT



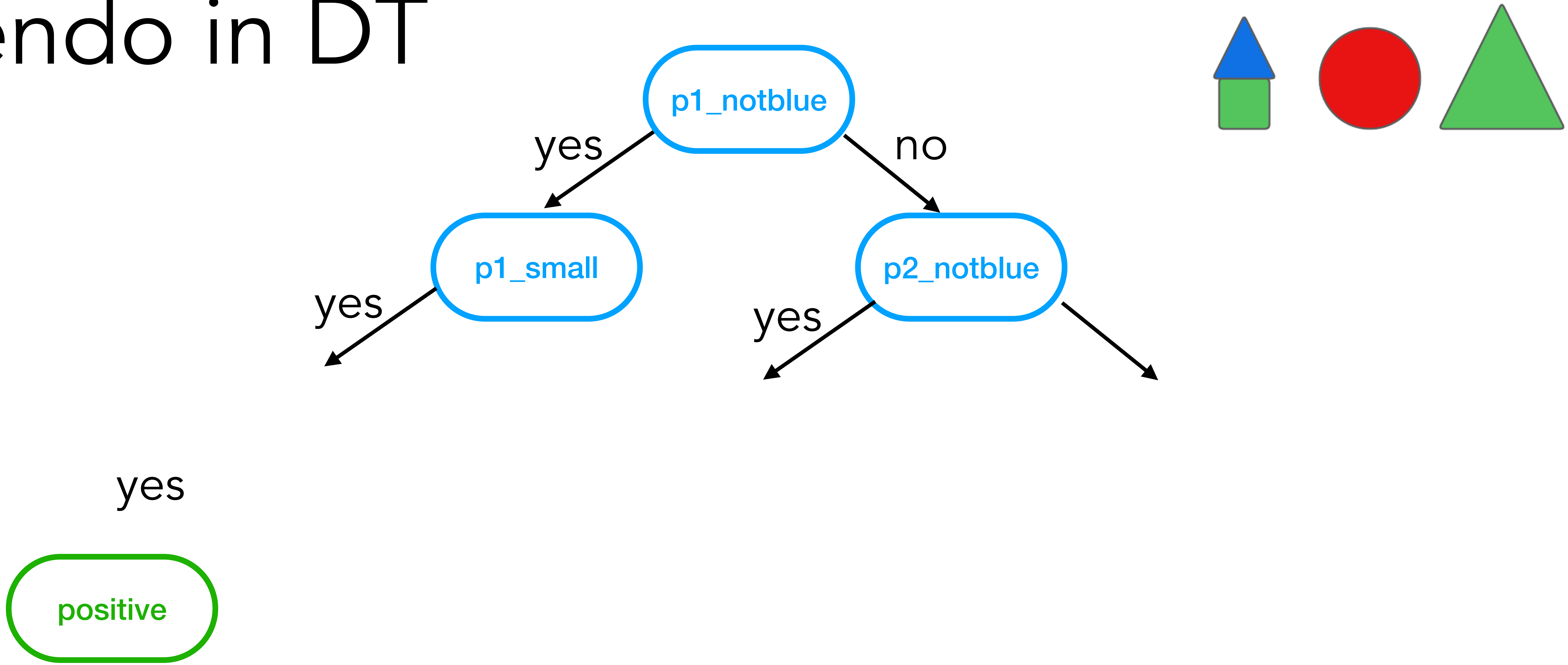



# Zendo in DT

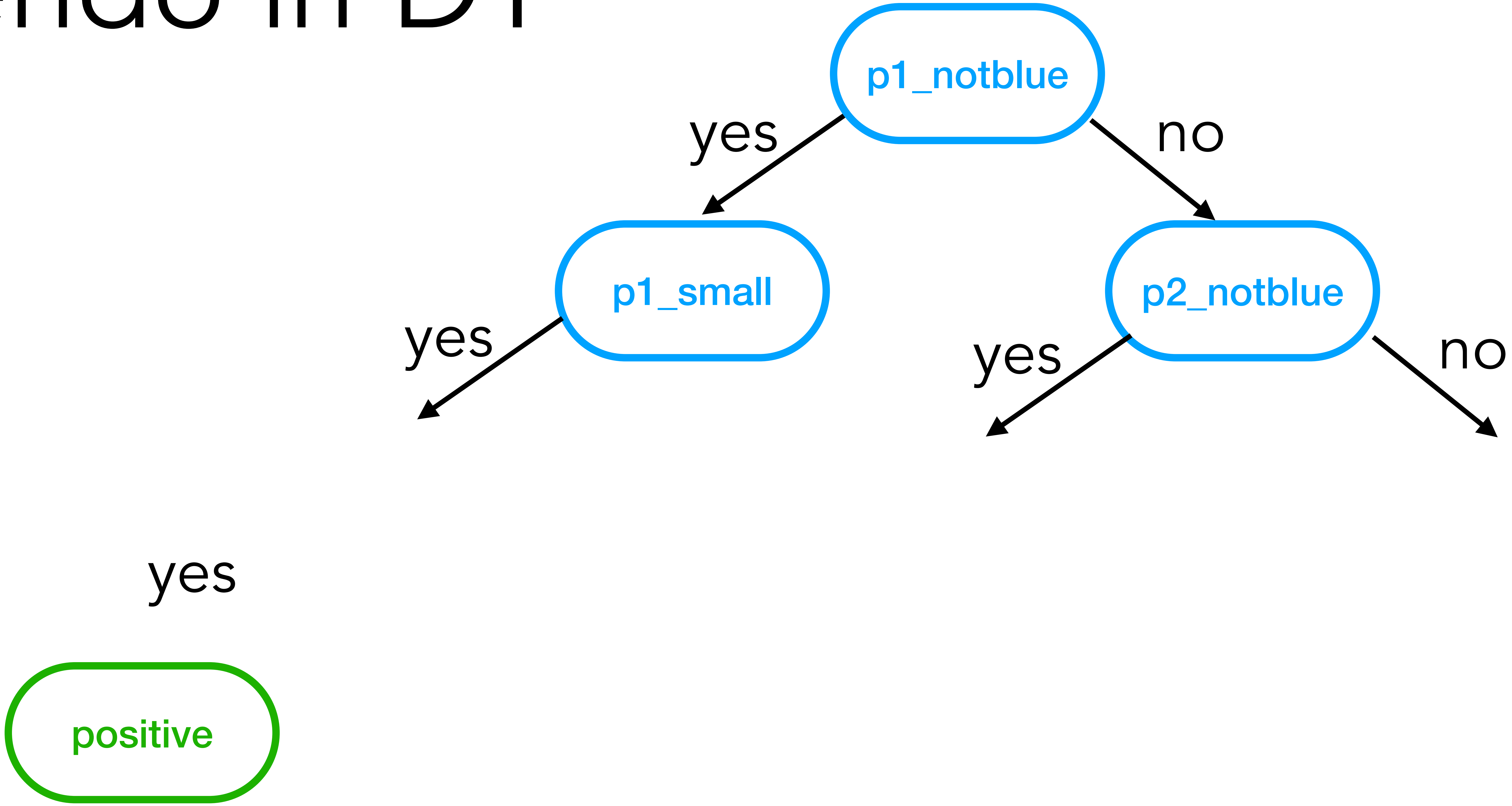


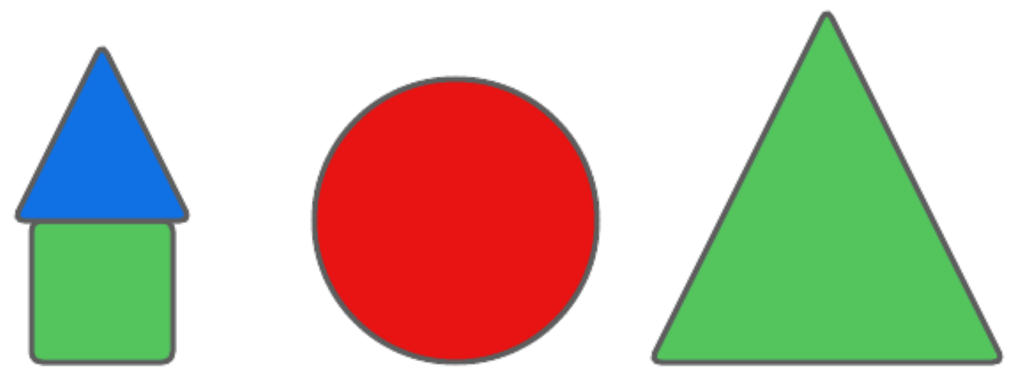



# Zendo in DT

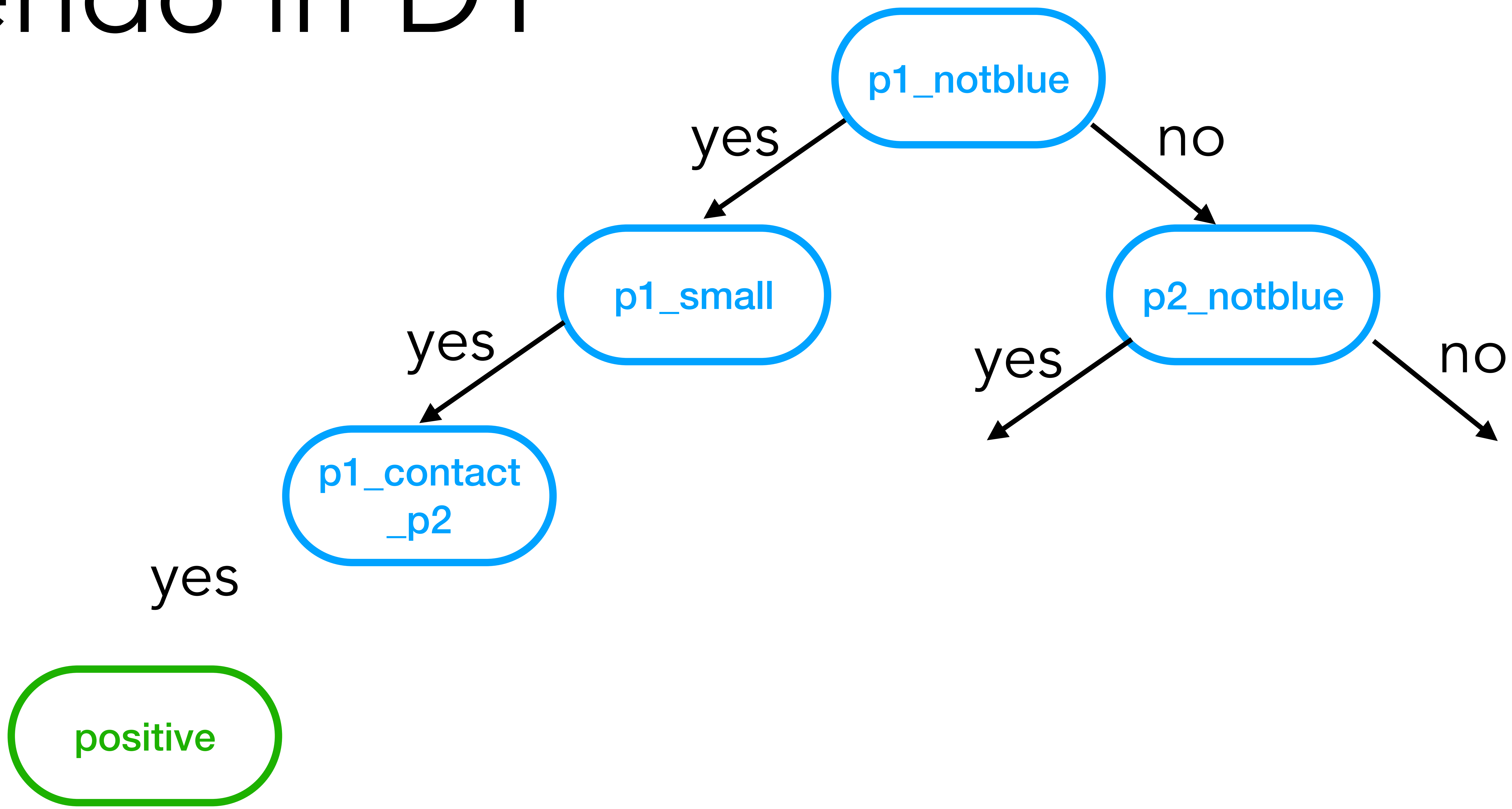


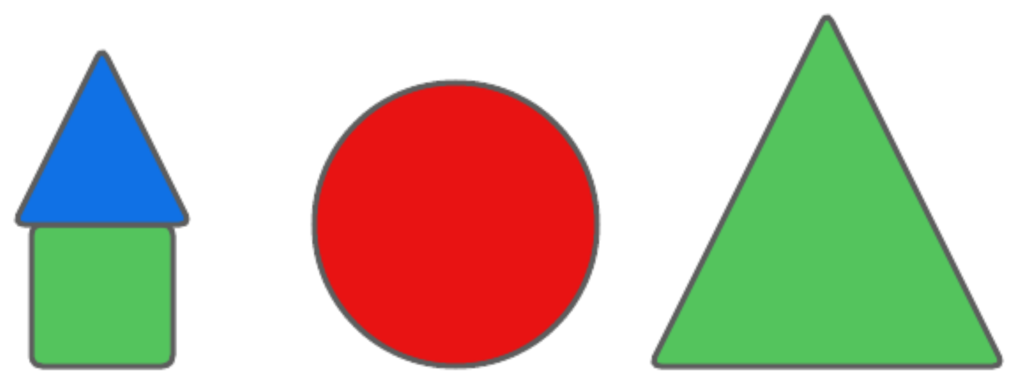
# Zendo in DT



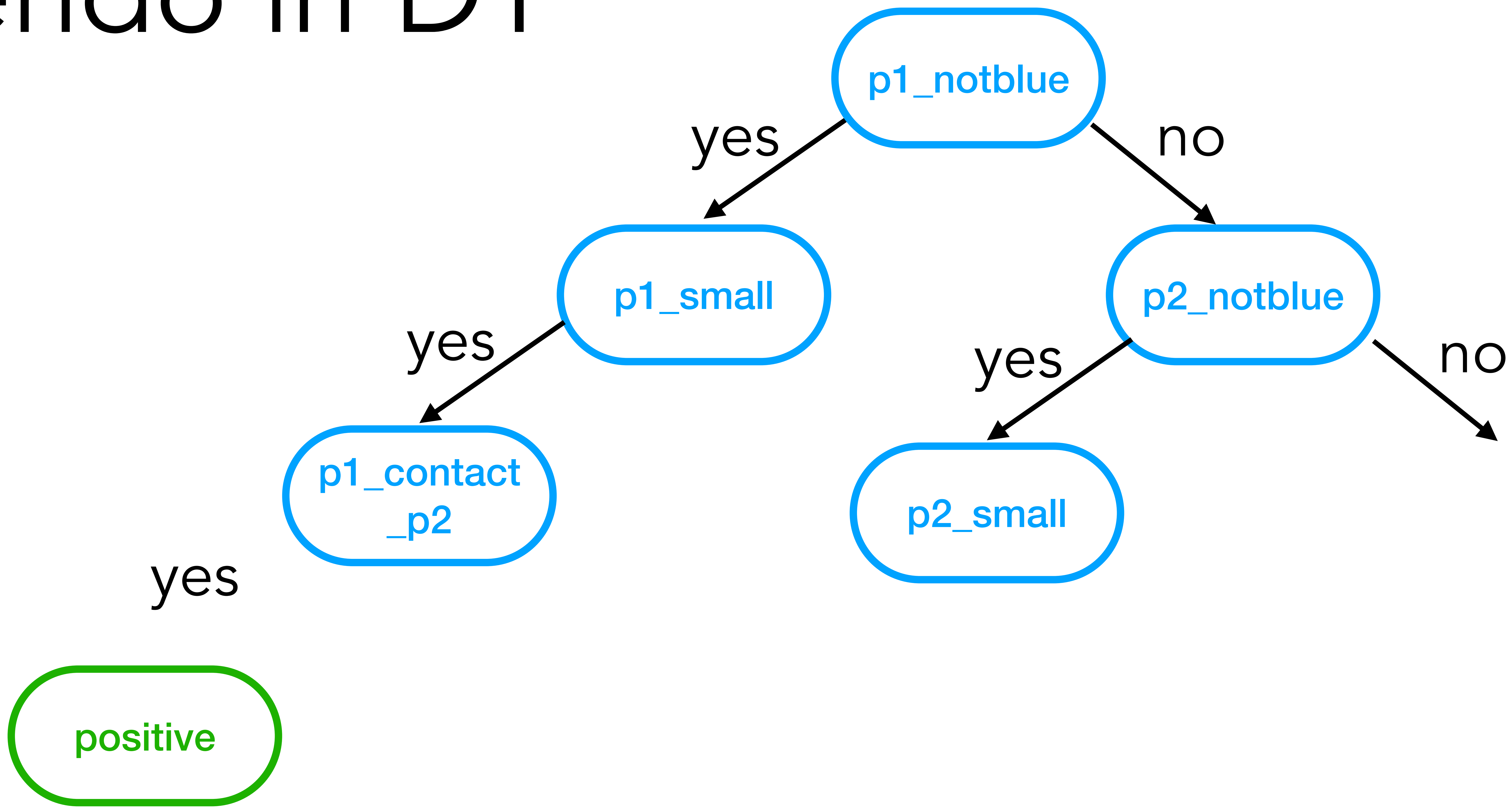



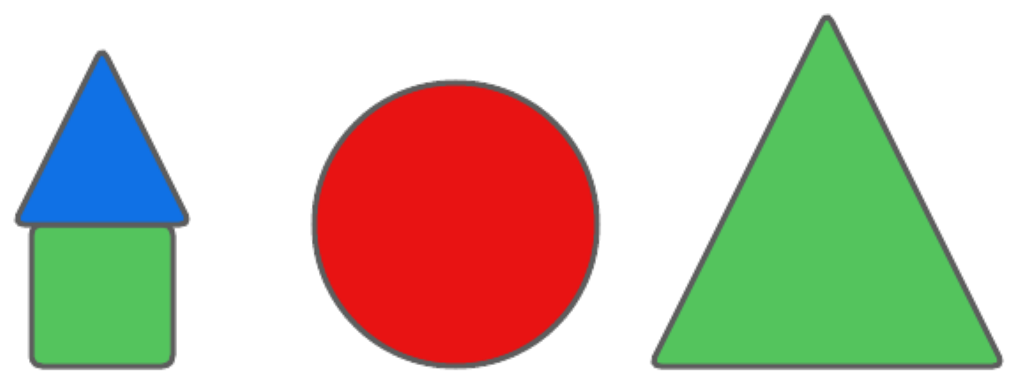
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
yes
positive

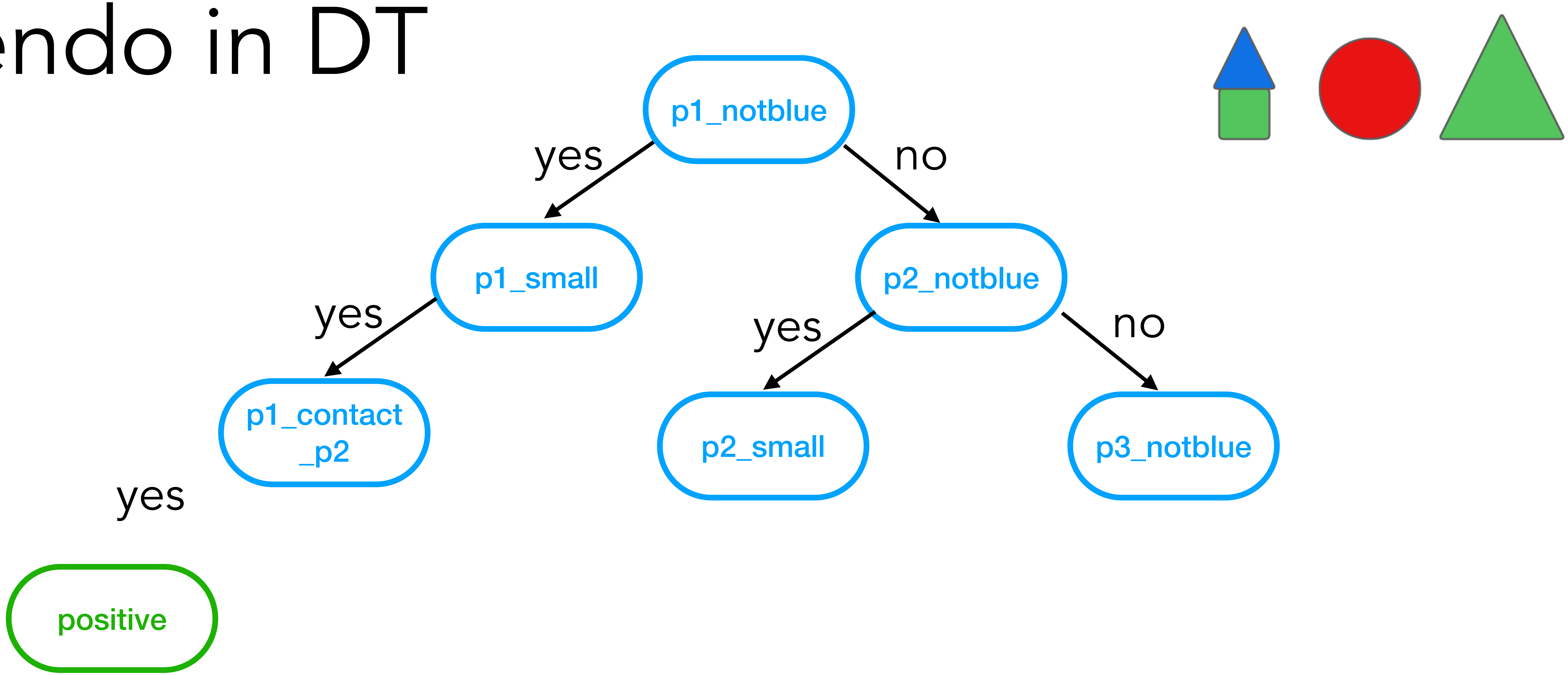


# Zendo in DT

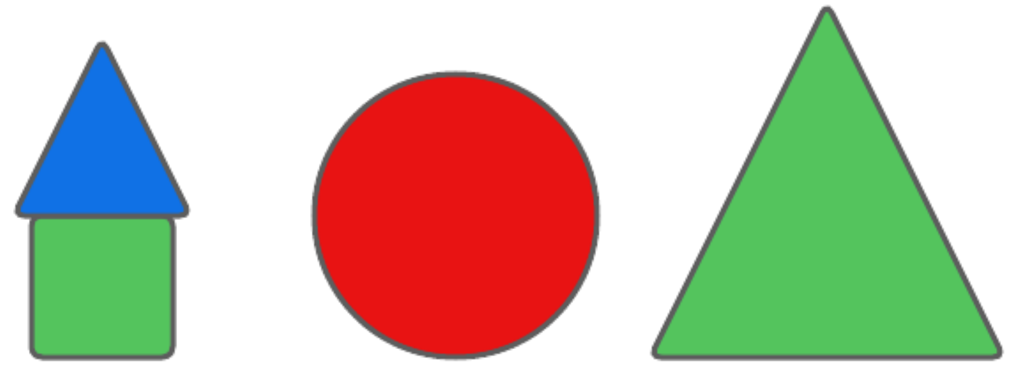

p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
p1_contact
_p2
p2_small
yes
positive

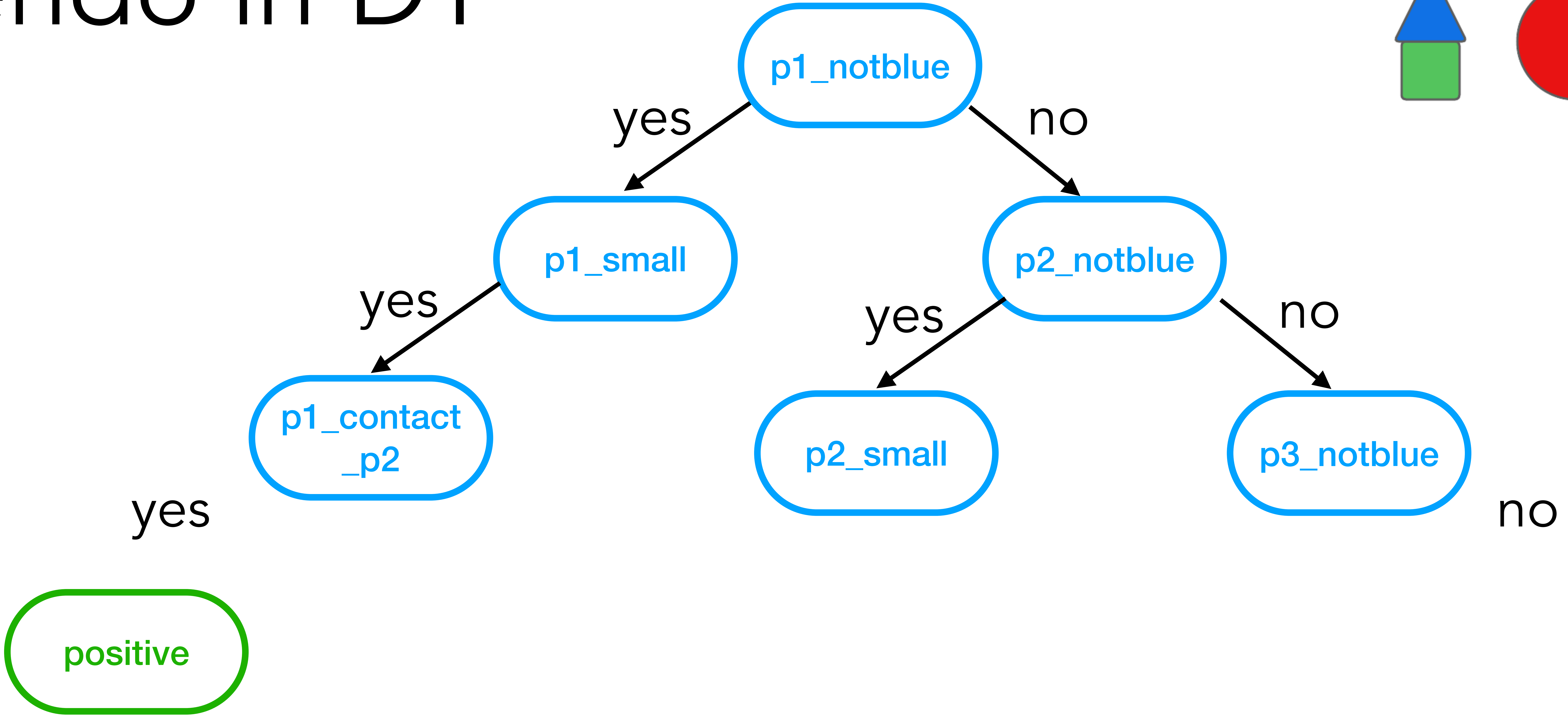


# Zendo in DT

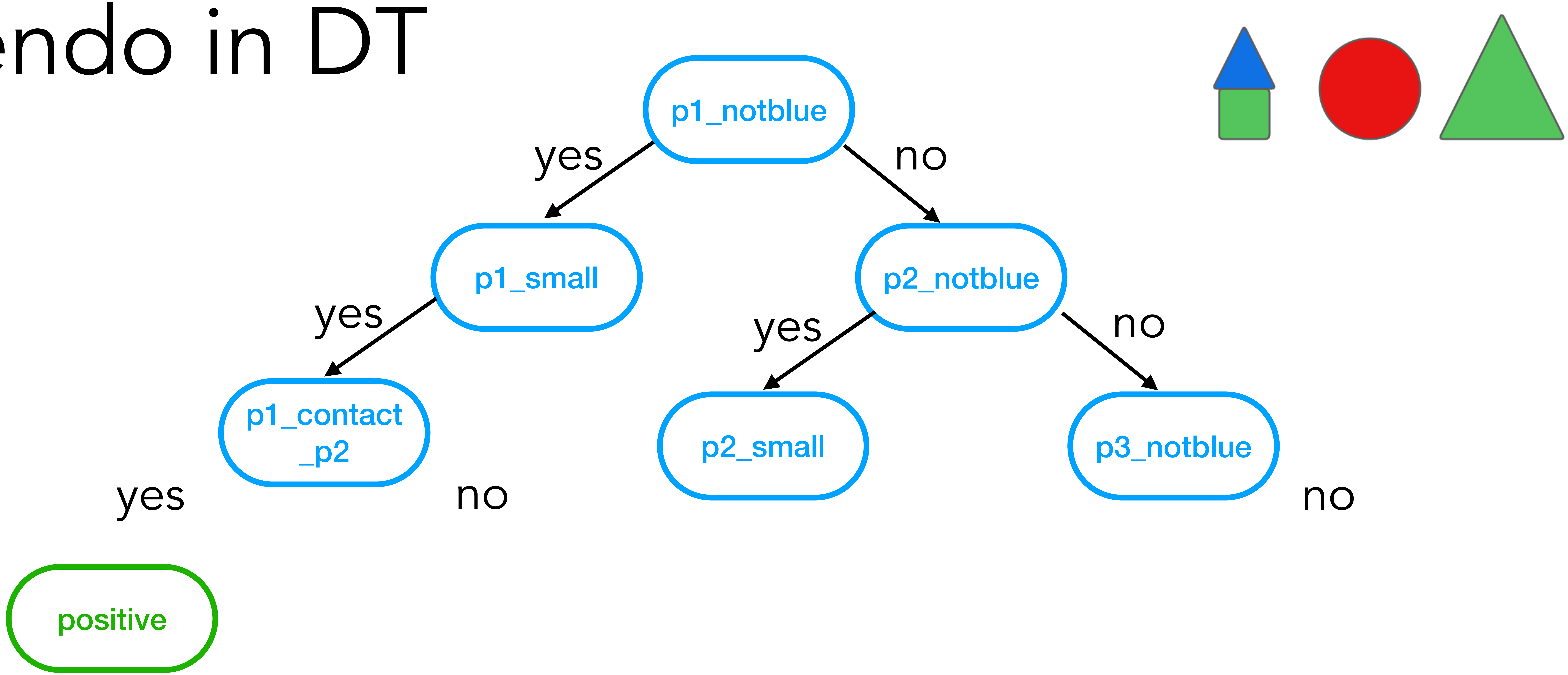

p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
positive

# Zendo in DT

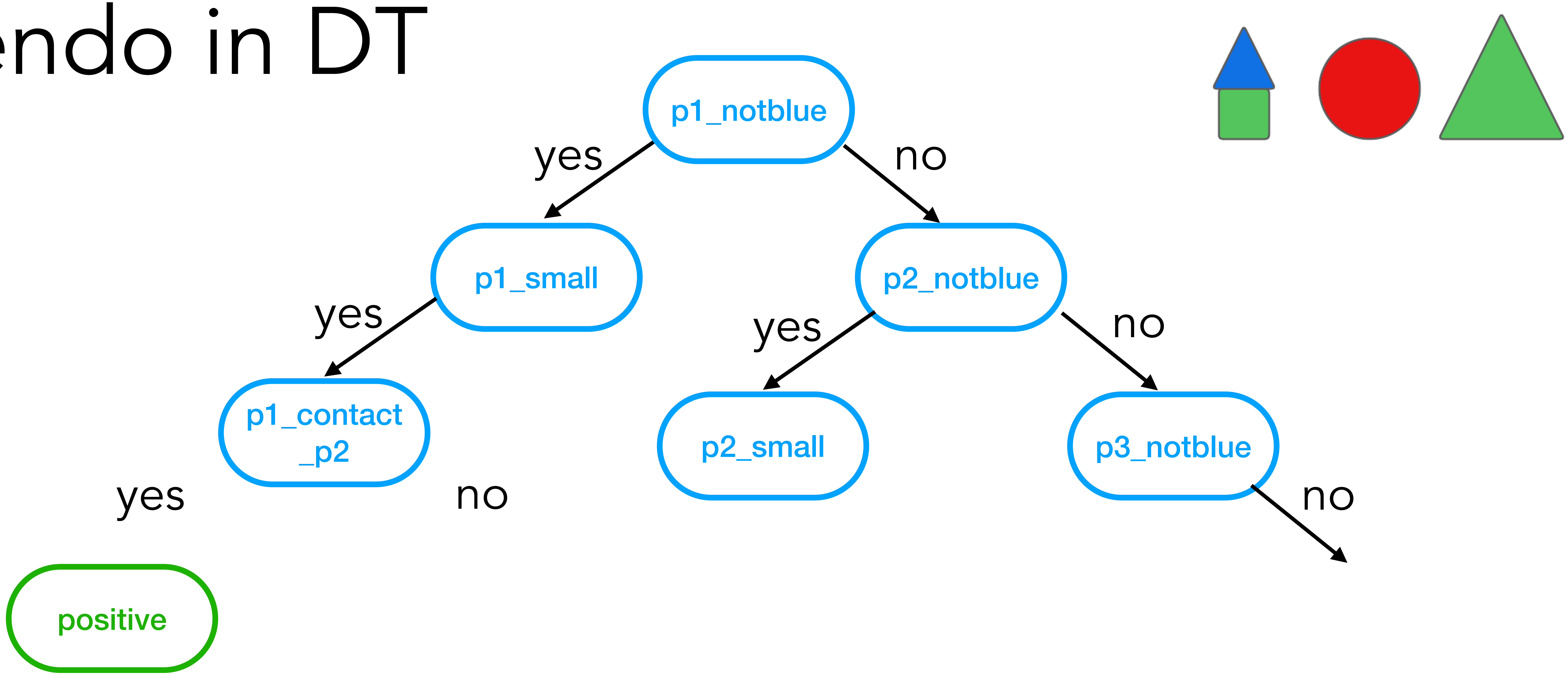



p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
positive

# Zendo in DT

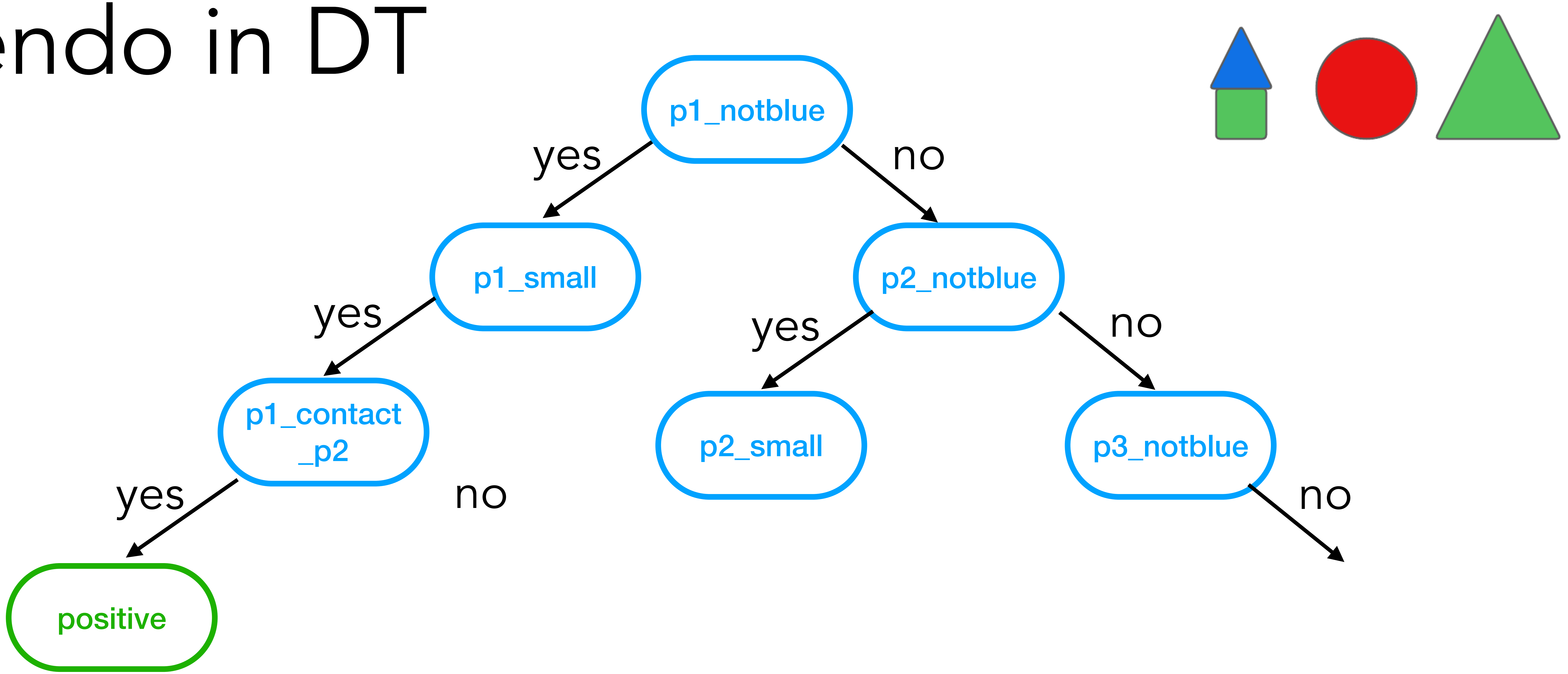

p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact _p2
p2_small
p3_notblue
yes
no
no
positive

# Zendo in DT

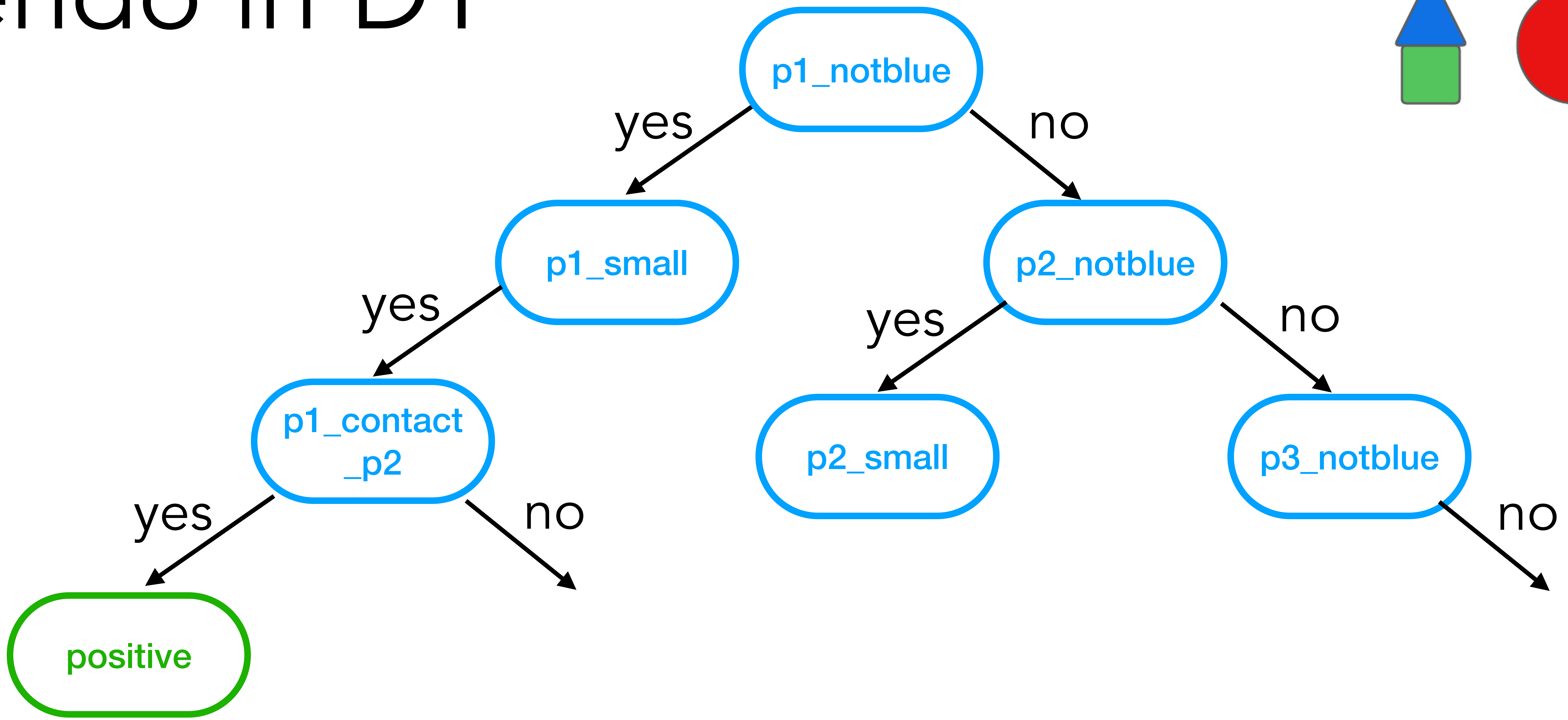


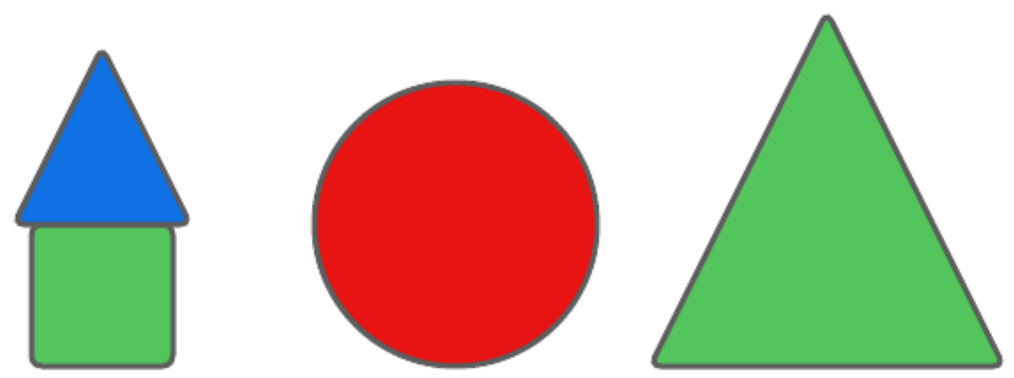
p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
no
positive

# Zendo in DT

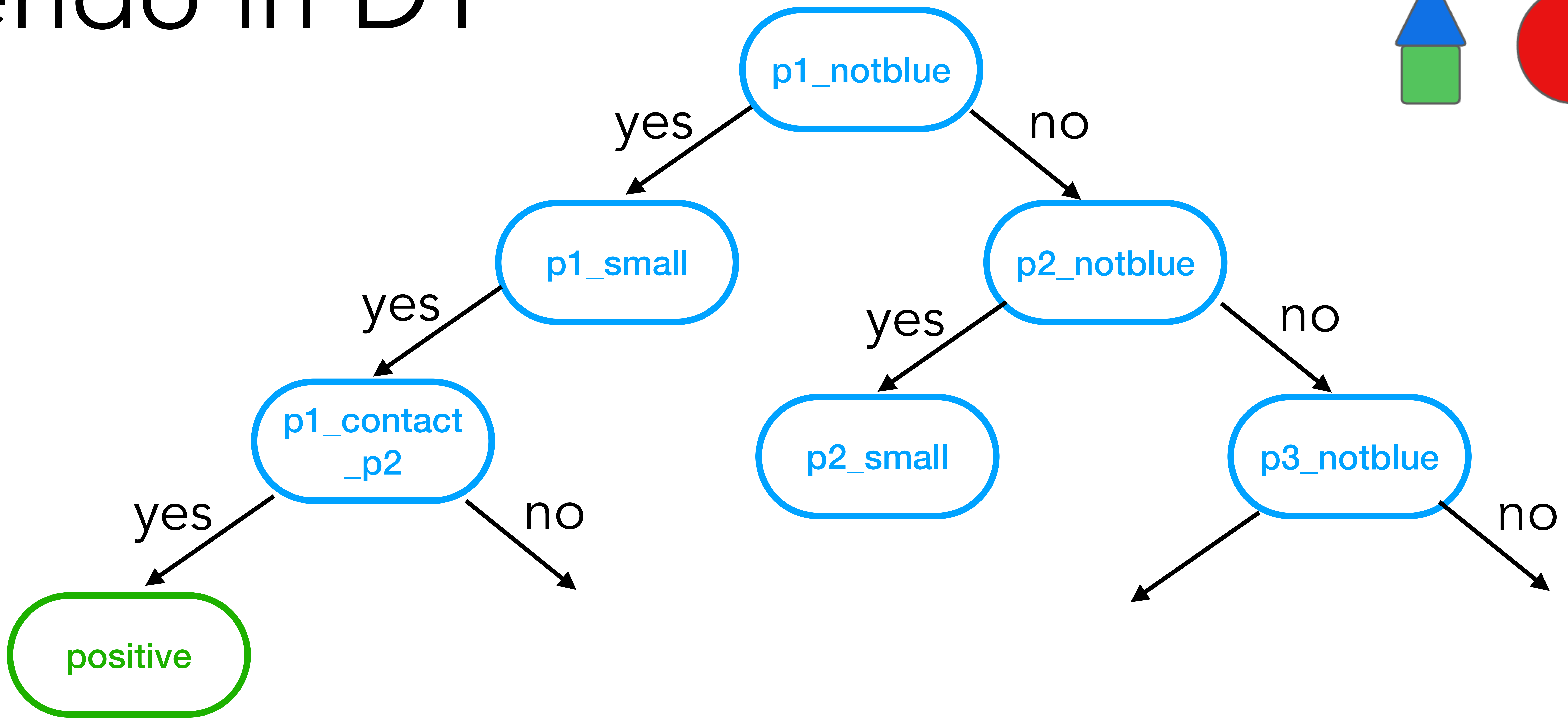

p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
no
positive

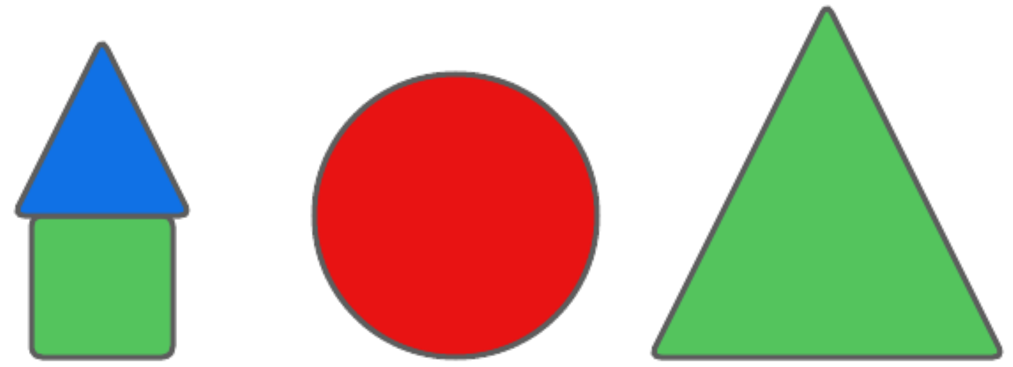
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
no
positive

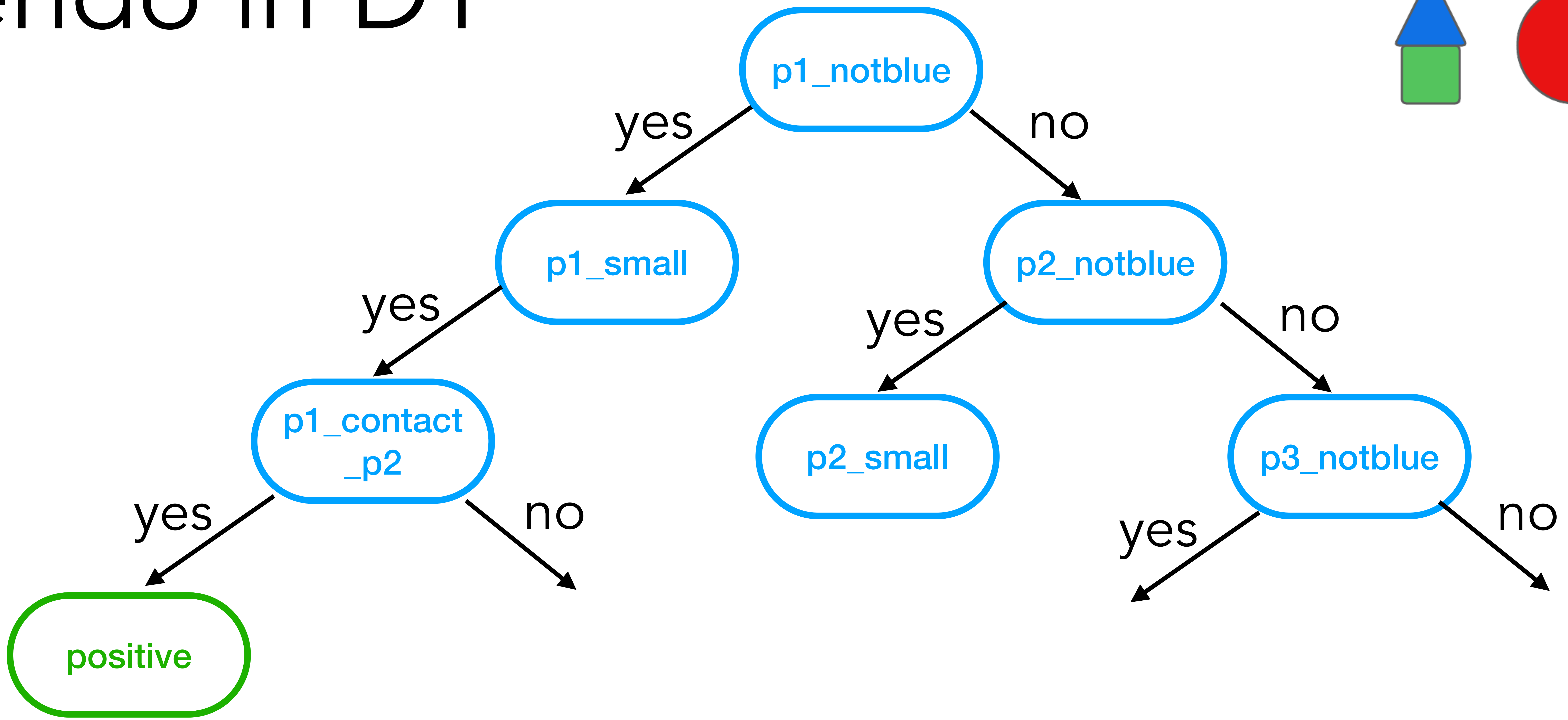


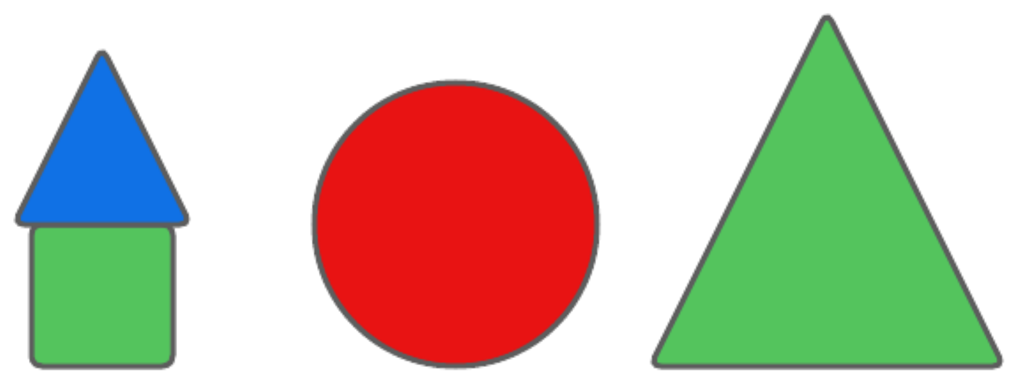
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
p1_contact
_p2
p2_small
p3_notblue
yes
no
no
positive

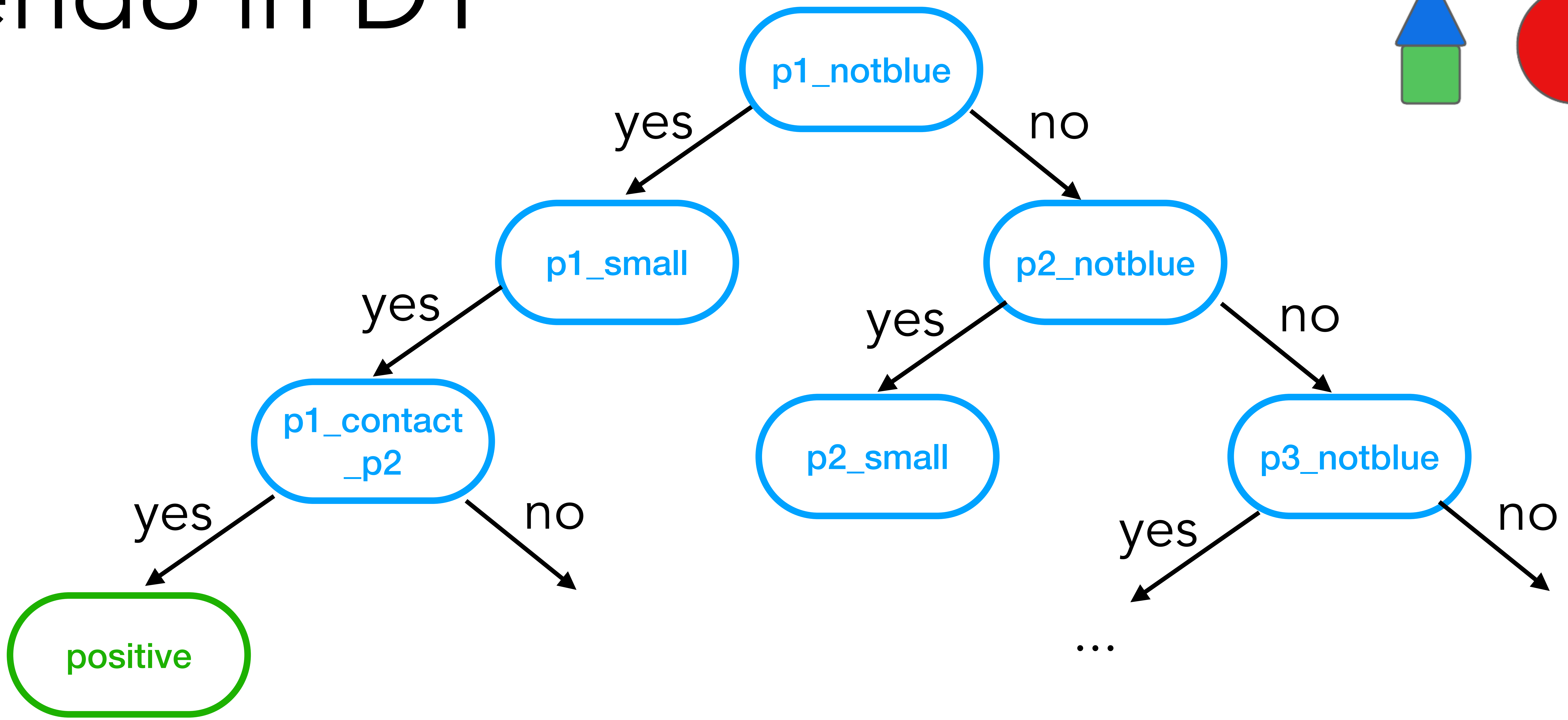


# Zendo in DT

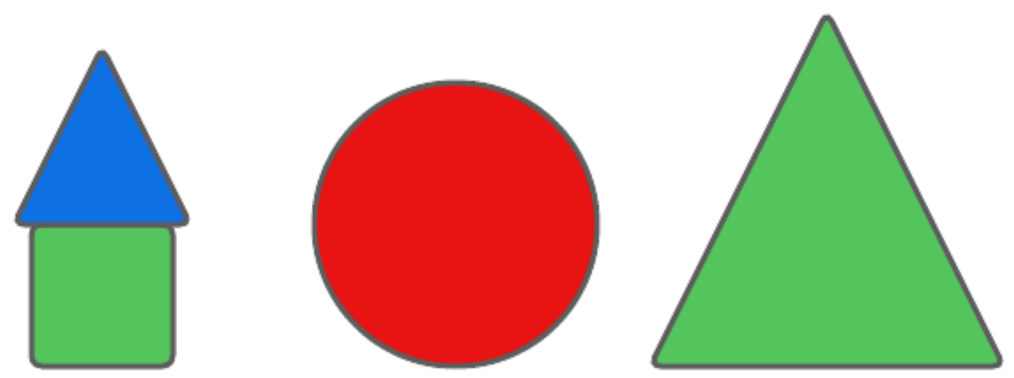

p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive



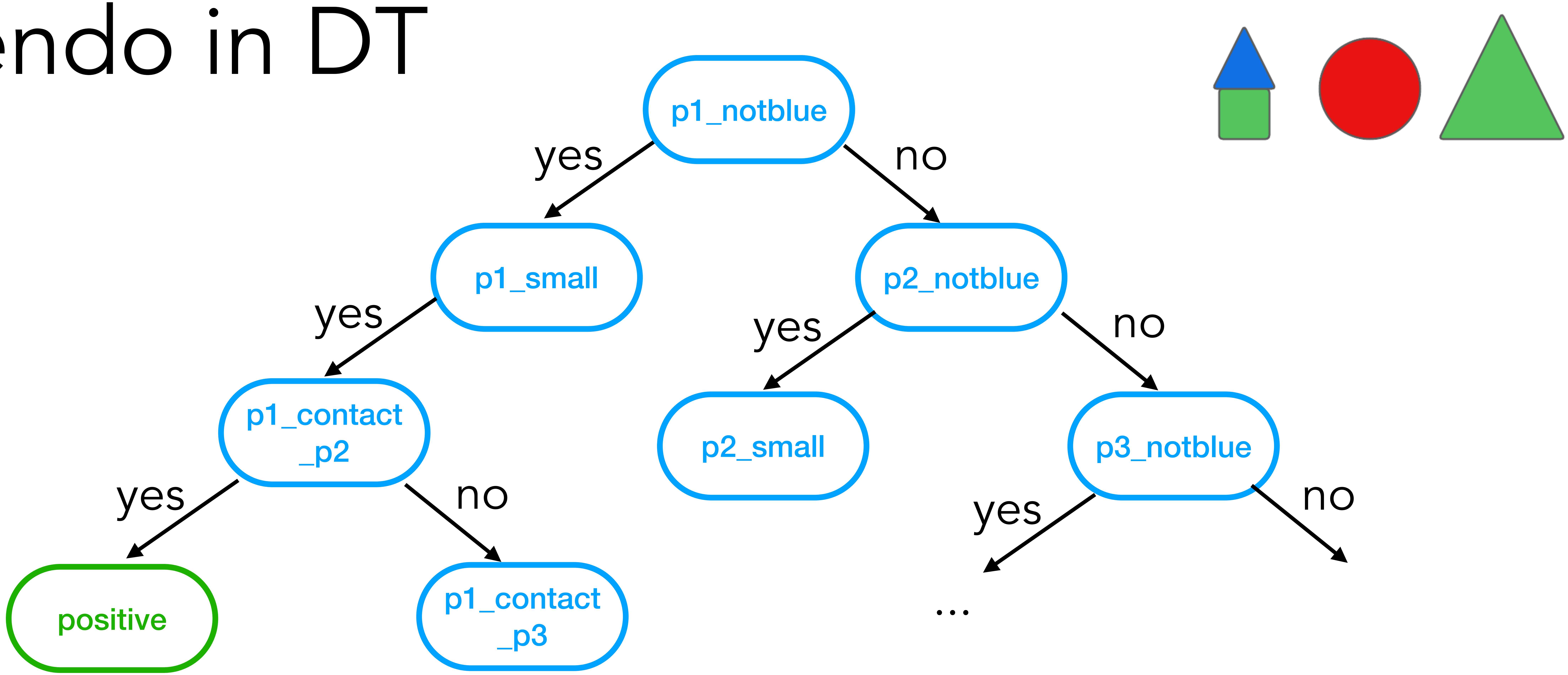
# Zendo in DT



p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact _p2
p2_small
p3_notblue
yes
no
yes
no
positive
...

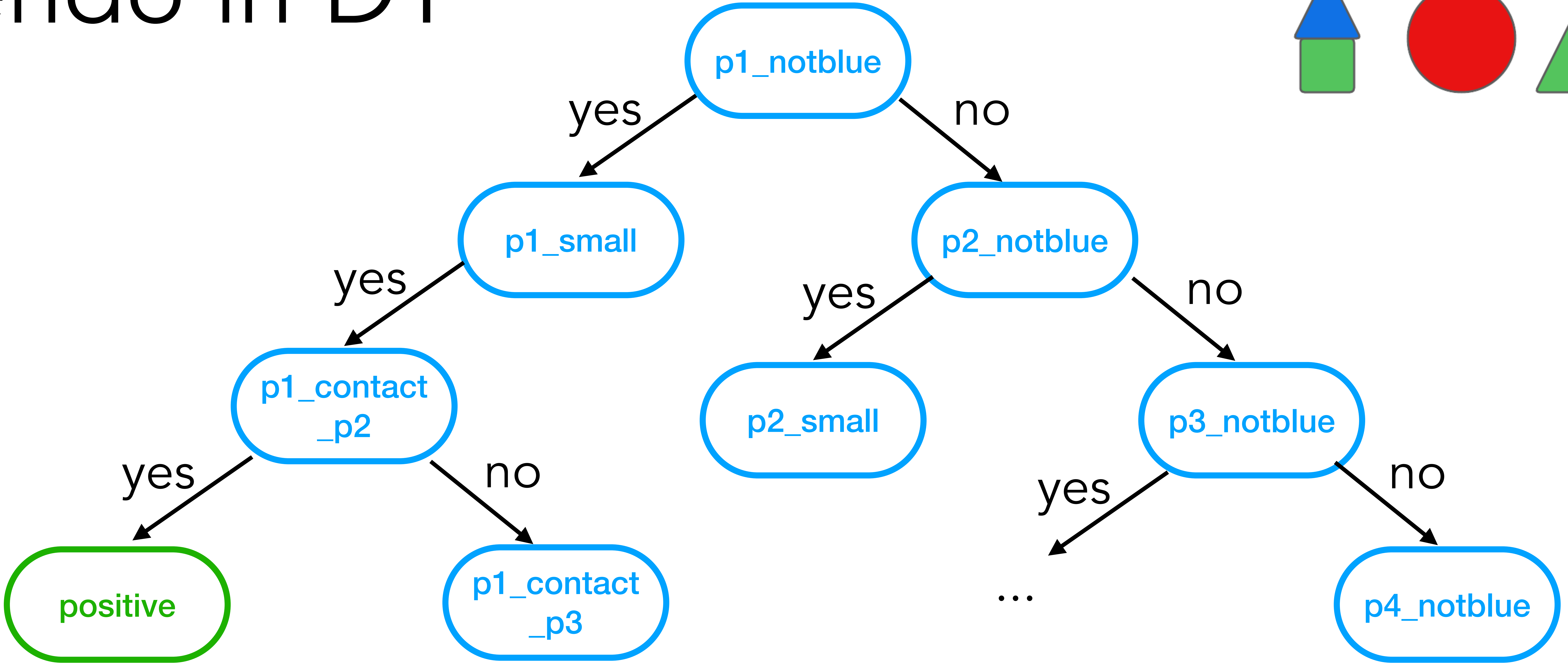



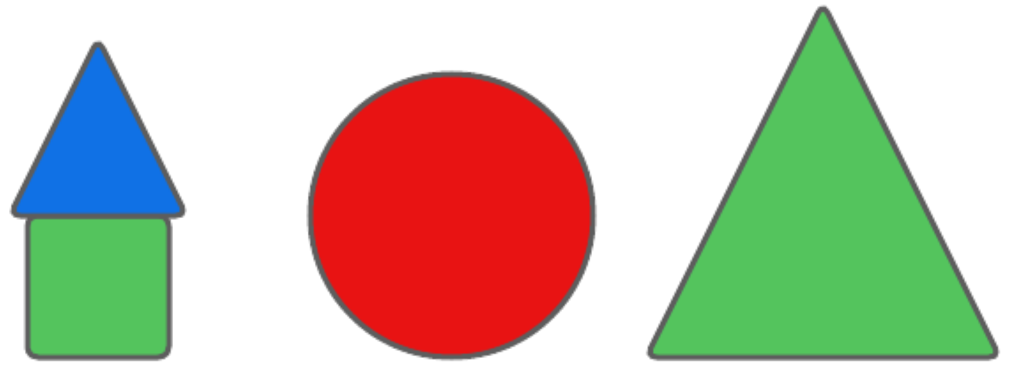
# Zendo in DT



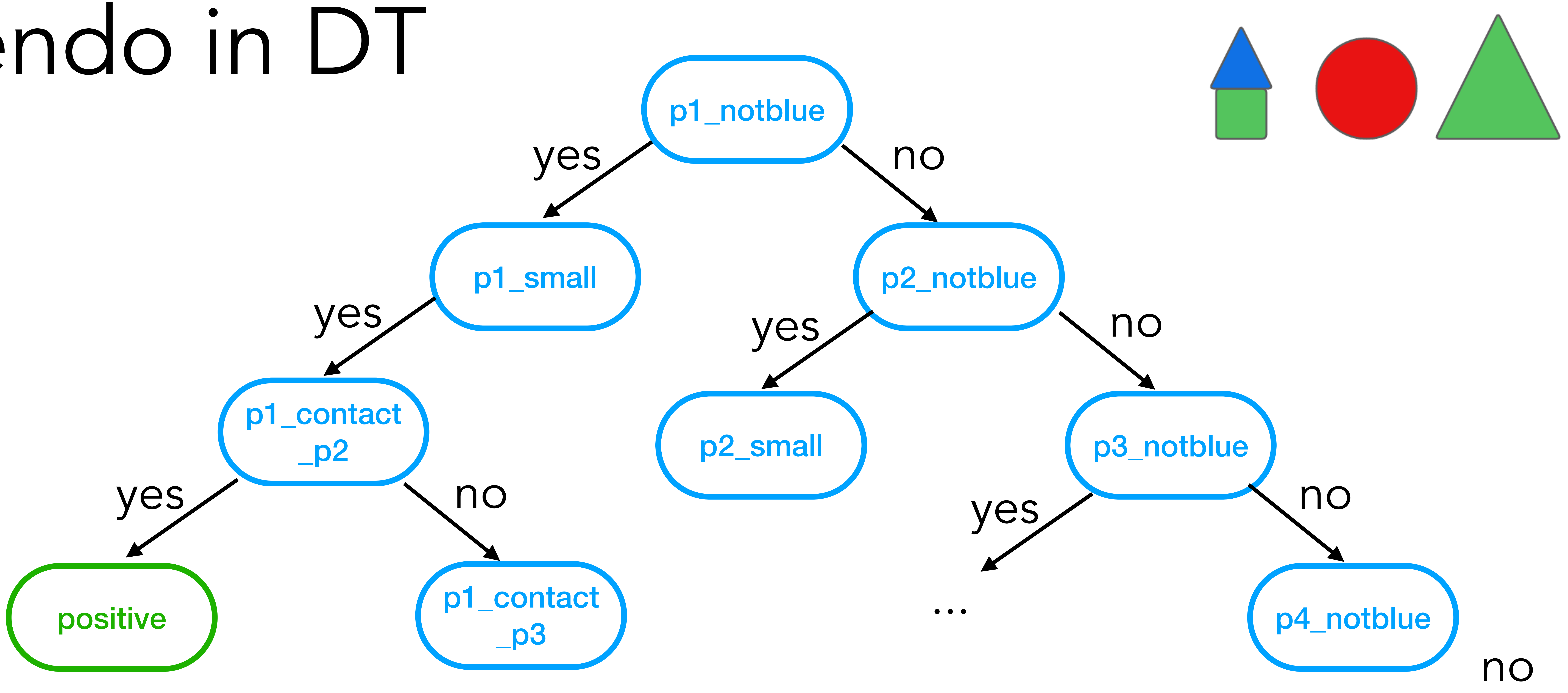
p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...

# Zendo in DT

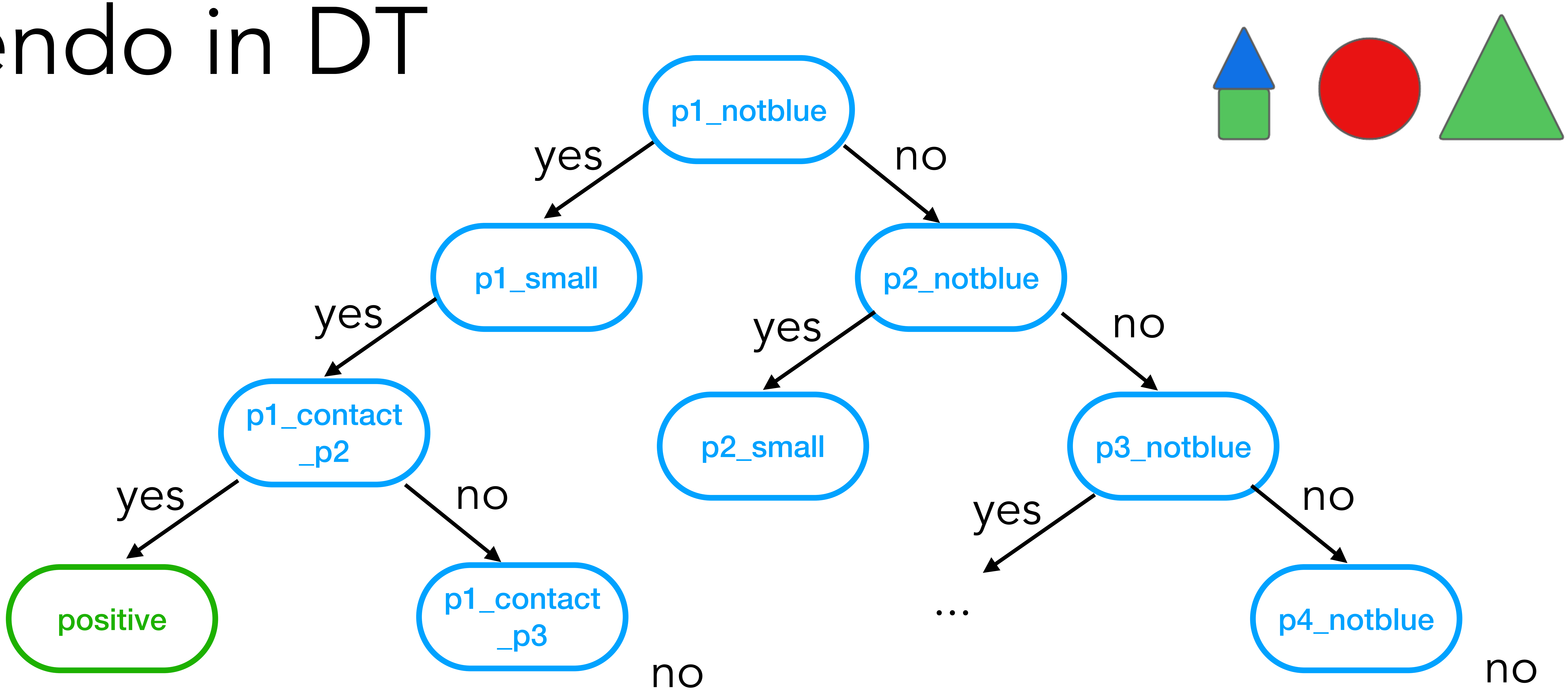

p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact _p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact _p3
...
p4_notblue



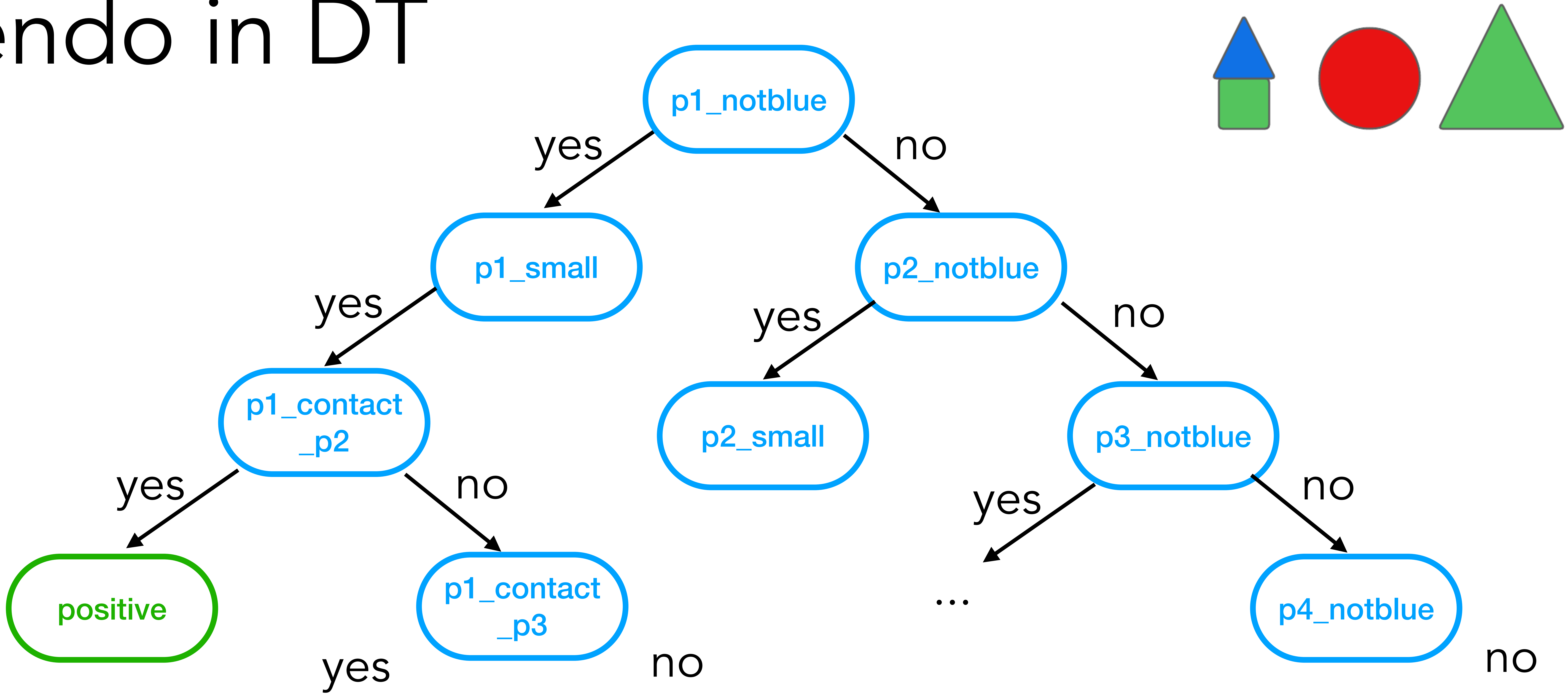
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
p1_contact _p2
yes
positive
no
p1_contact _p3
yes
p2_small
no
p3_notblue
yes
...
no
p4_notblue
no

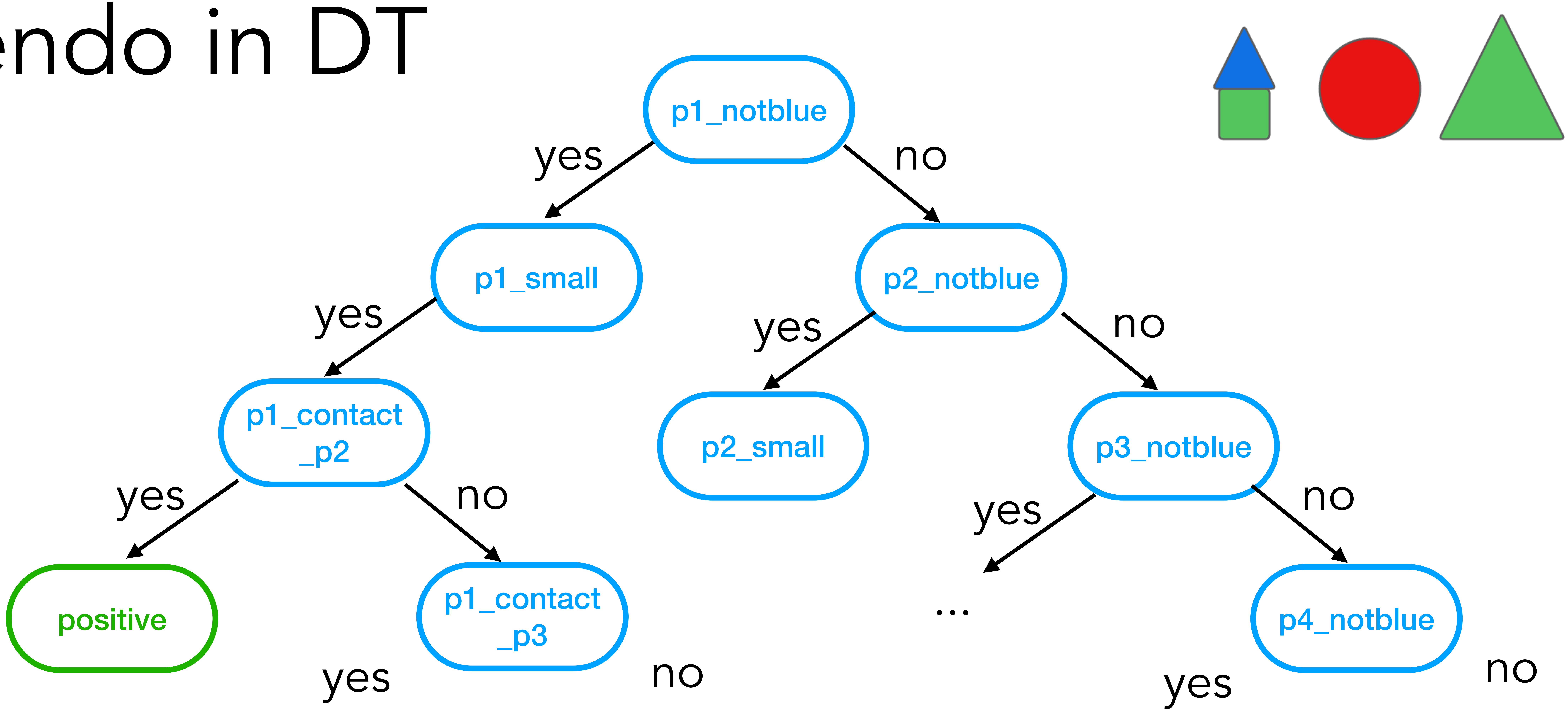
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...
p4_notblue
no
no

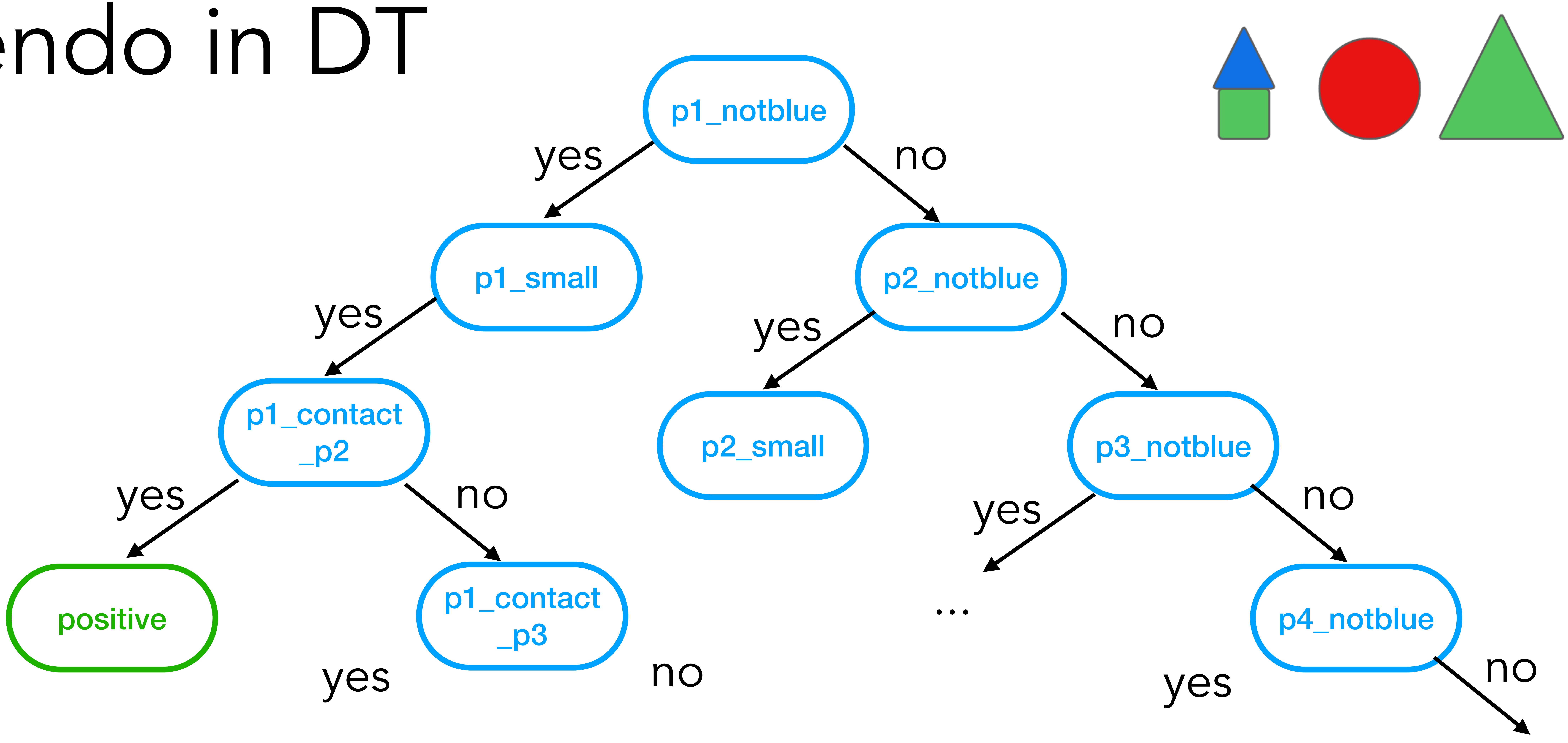
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...
p4_notblue
yes
no
no

# Zendo in DT

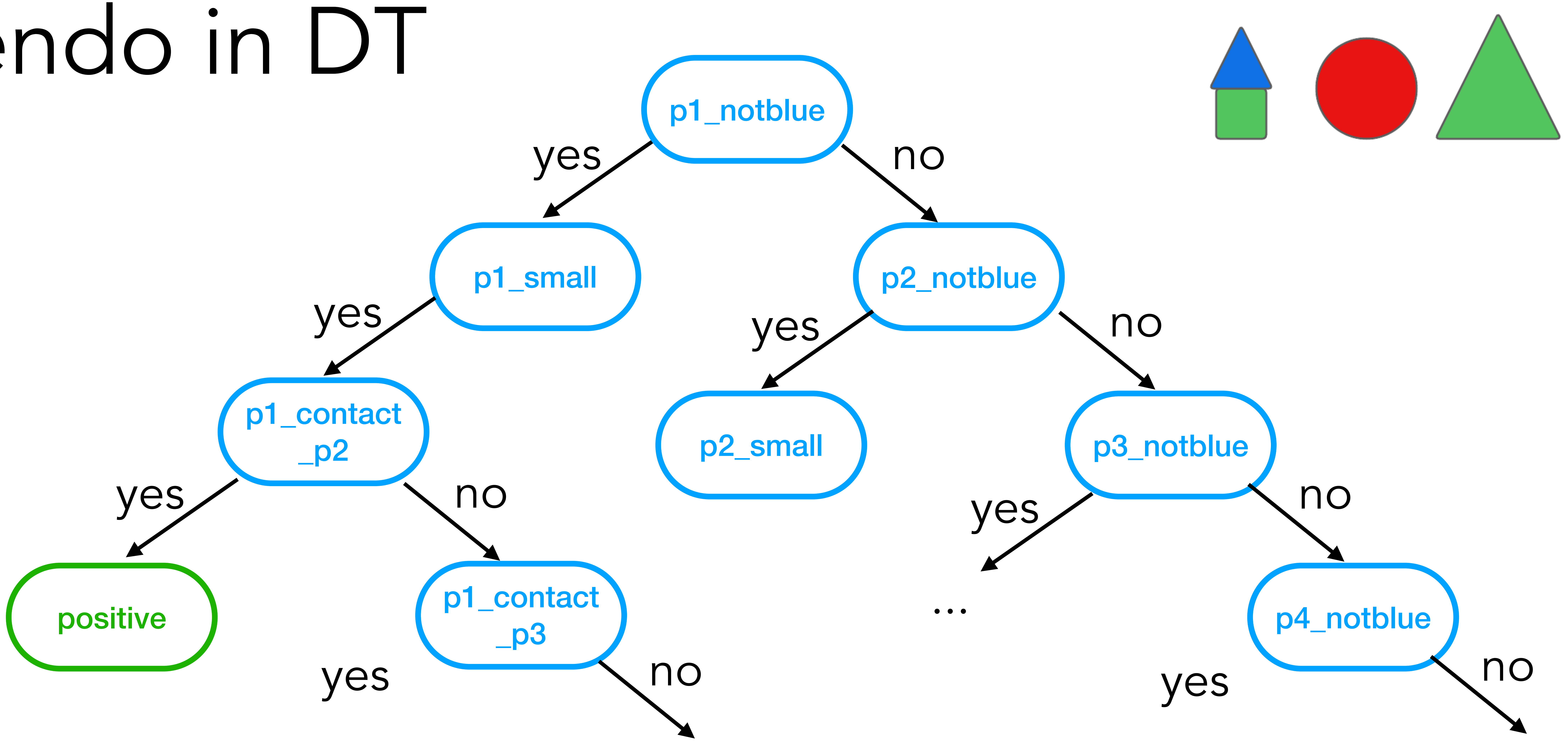

p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
no
p1_contact _p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact _p3
...
p4_notblue
yes
no
yes
no

# Zendo in DT

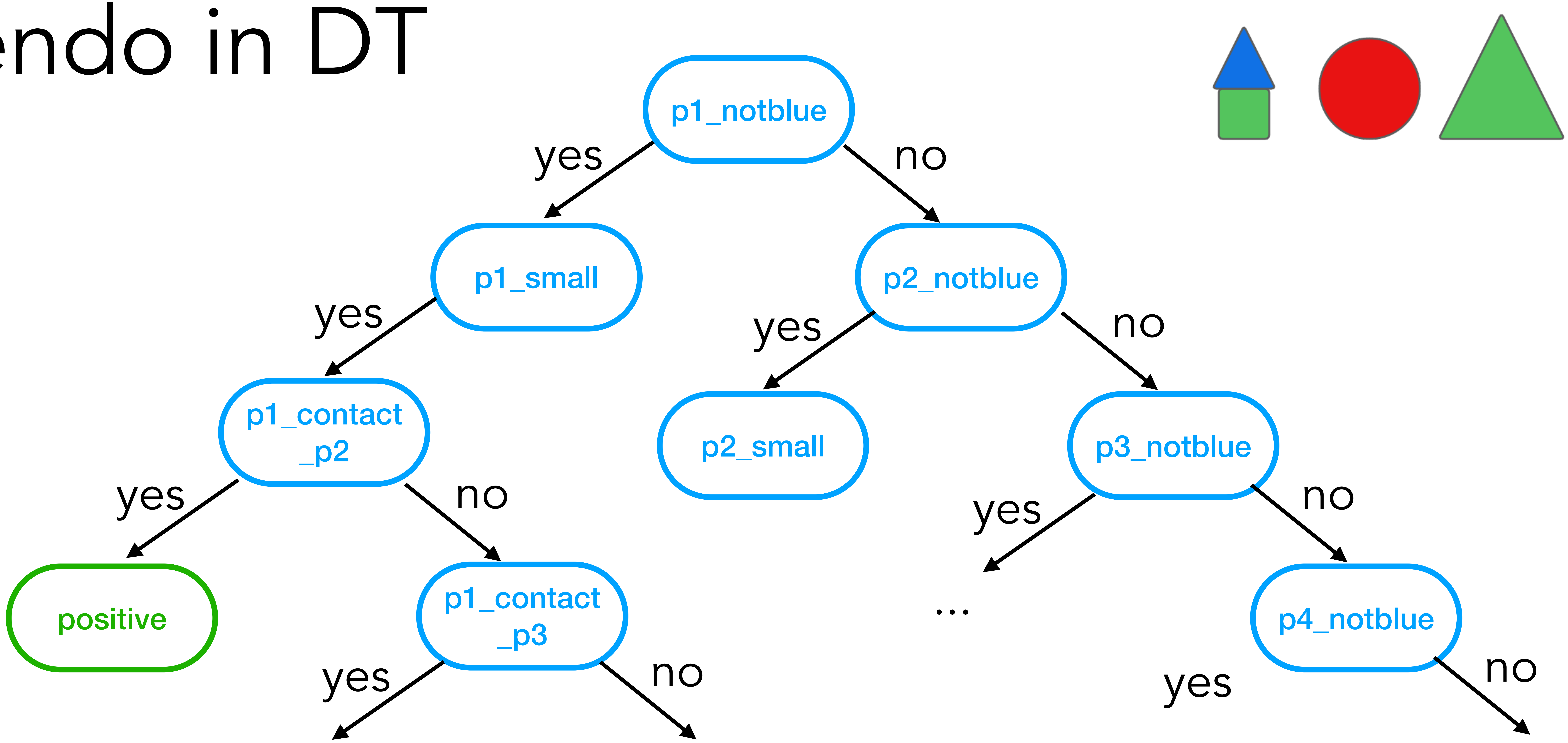


p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...
p4_notblue
yes
no
yes
no

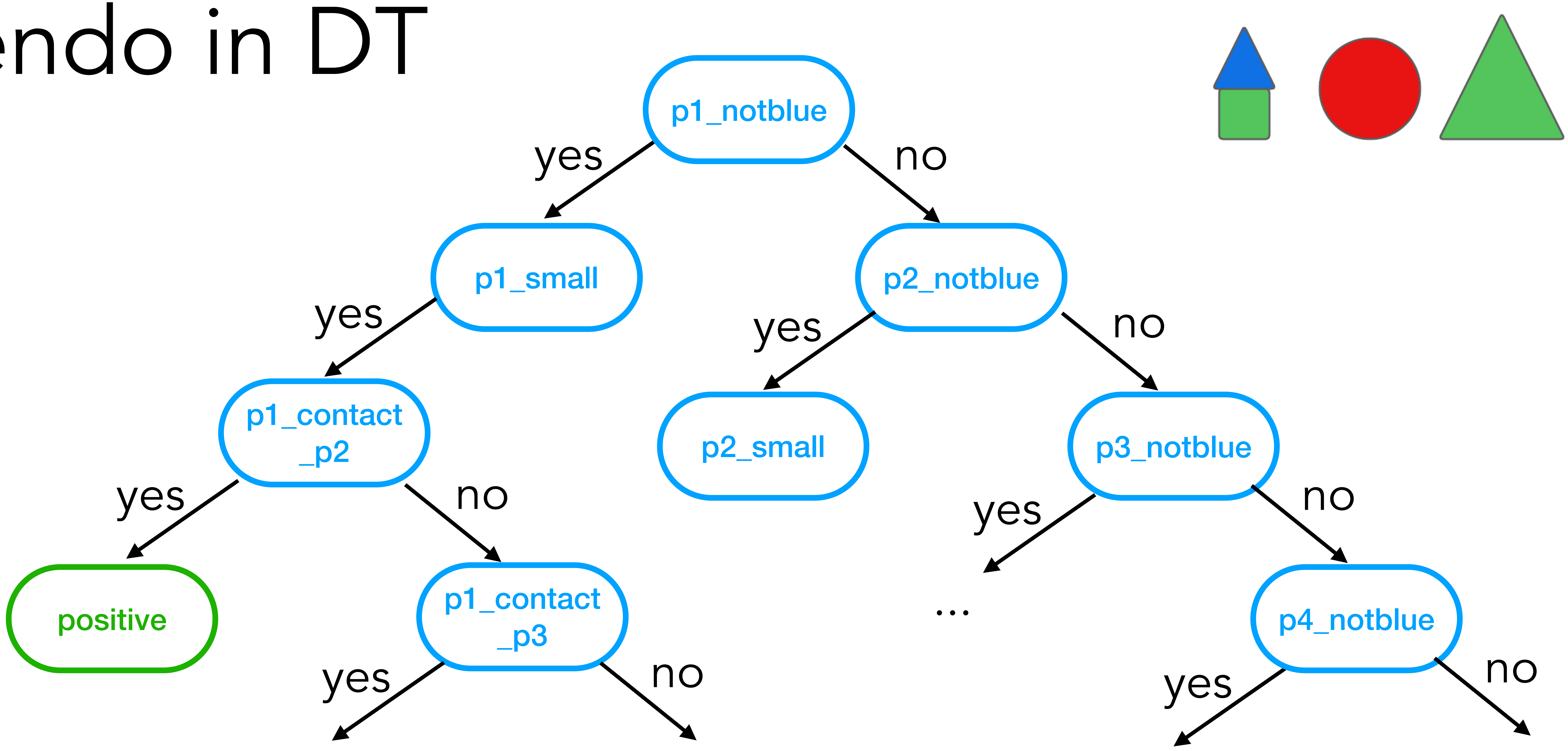
# Zendo in DT


p1_notblue
yes
no
p1_small
p2_notblue
yes
no
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...
p4_notblue
yes
no
yes
no

# Zendo in DT

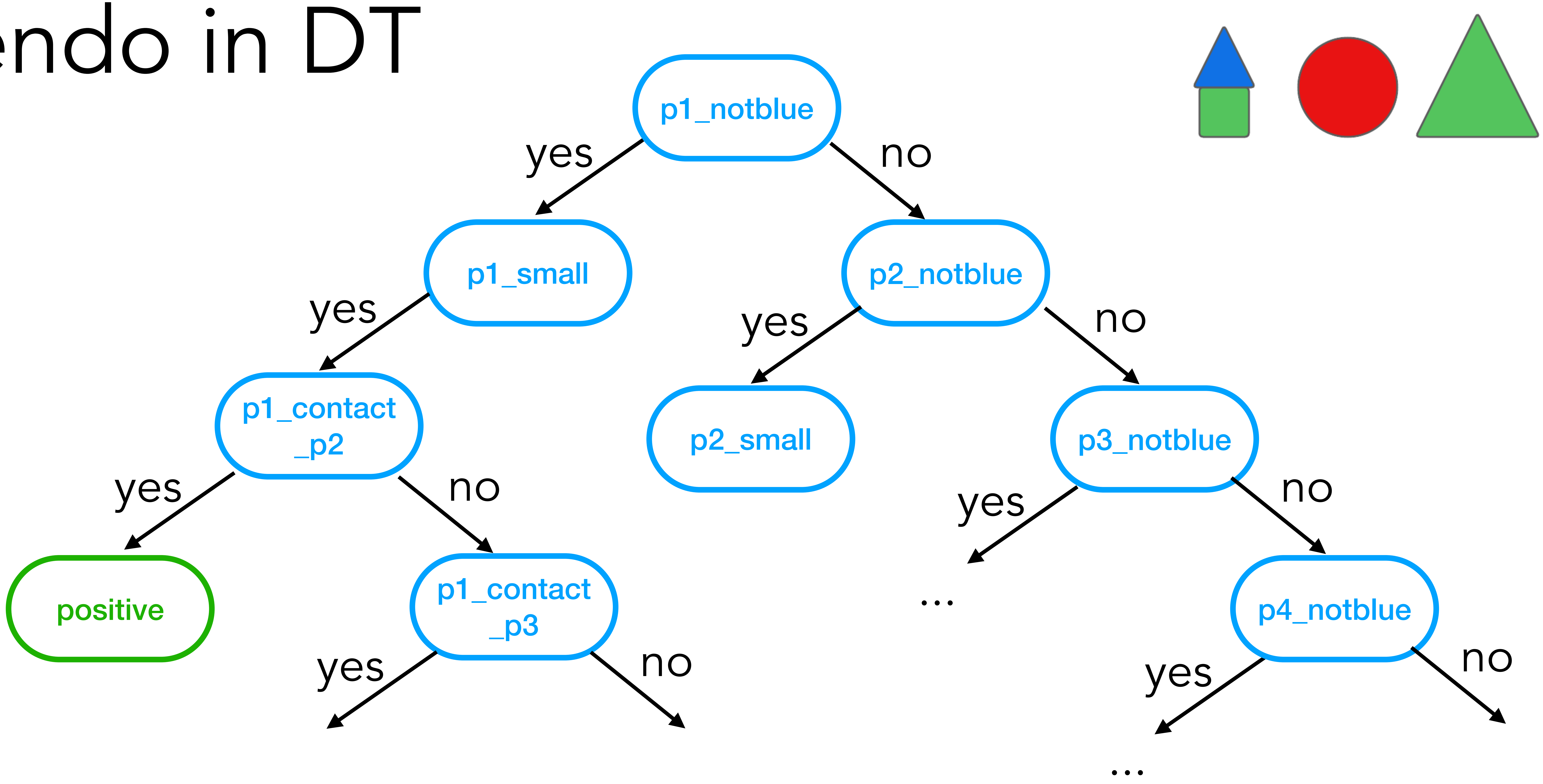

p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact
_p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...
p4_notblue
yes
no
yes
no

# Zendo in DT

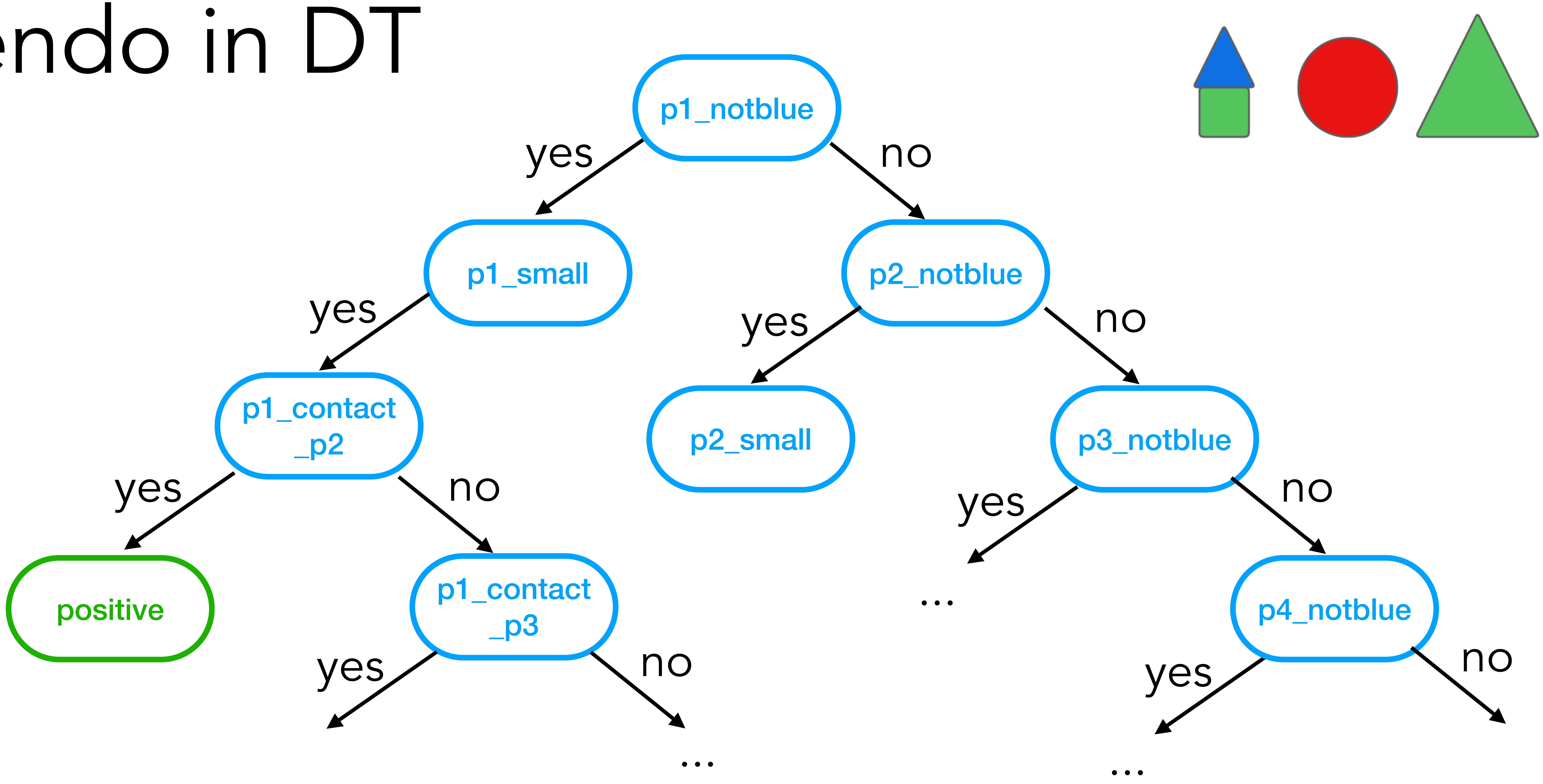


p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact _p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact _p3
...
p4_notblue
yes
no
yes
no

# Zendo in DT

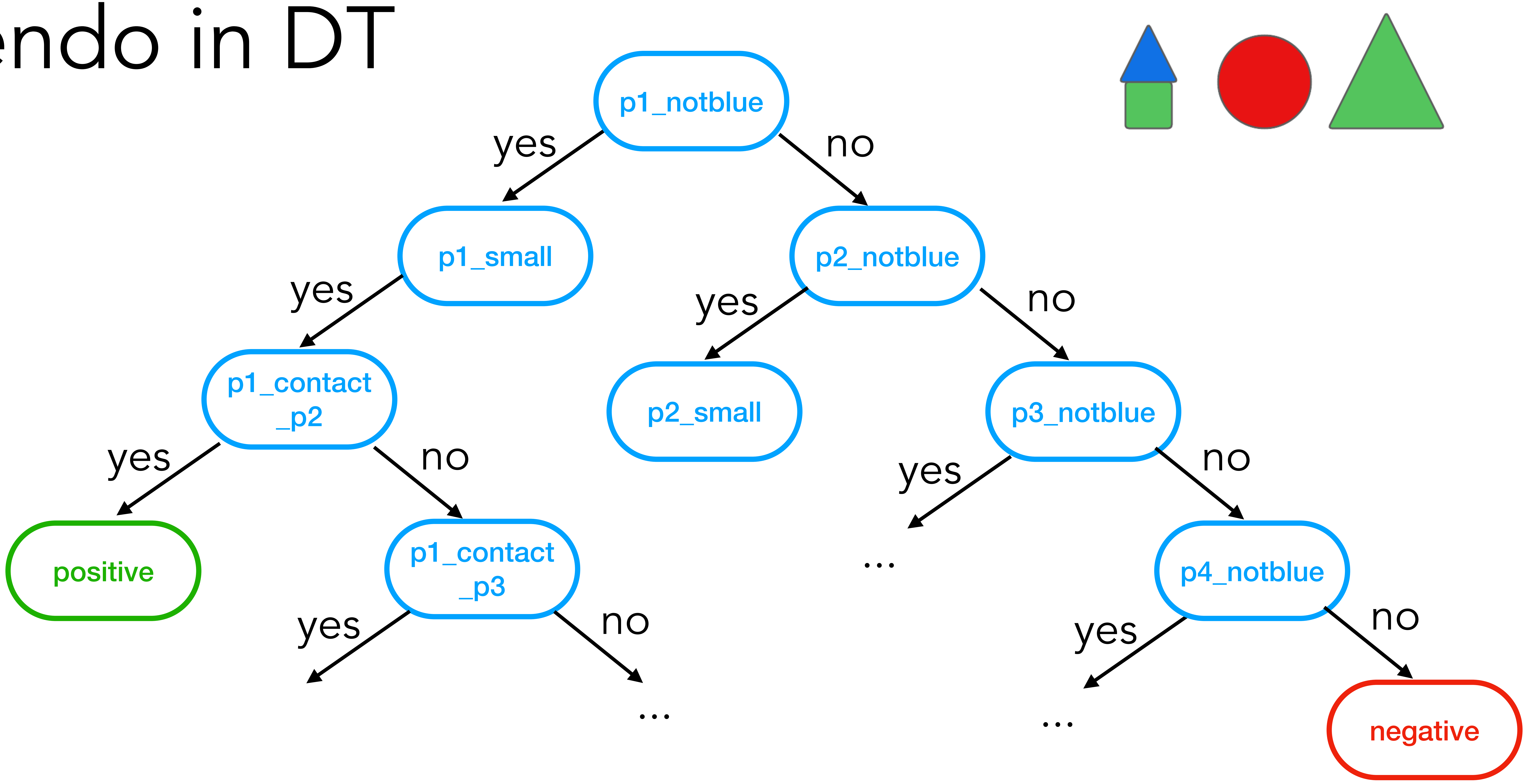

p1_notblue
yes
no
p1_small
p2_notblue
yes
p1_contact
_p2
p2_small
no
p3_notblue
yes
no
yes
positive
p1_contact
_p3
...
no
p4_notblue
yes
no
yes
no
...

# Zendo in DT

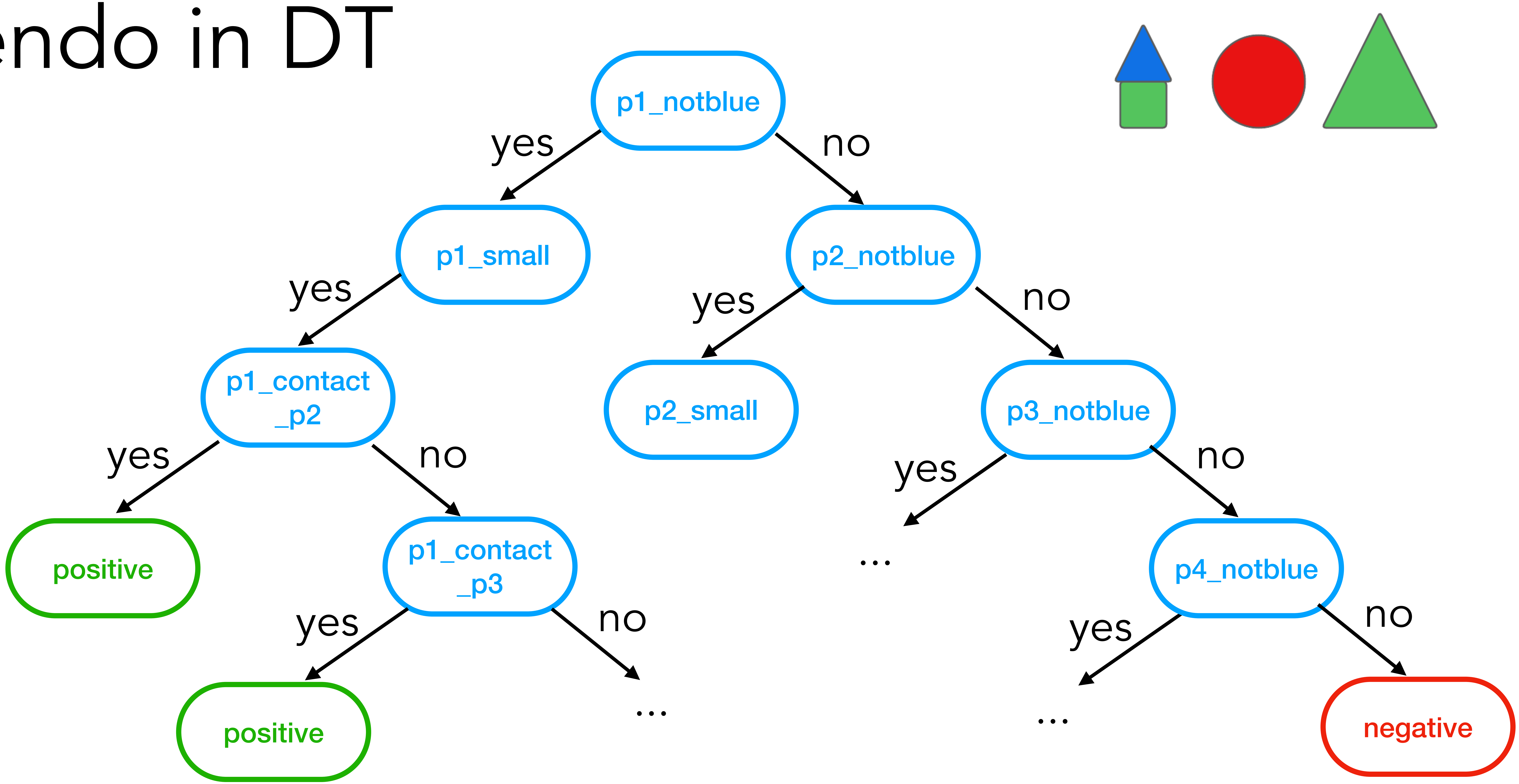


p1_notblue
yes
no
p1_small
p2_notblue
yes
yes
no
p1_contact _p2
p2_small
p3_notblue
yes
no
yes
no
positive
p1_contact _p3
...
p4_notblue
yes
no
yes
no
...
...

# Zendo in DT

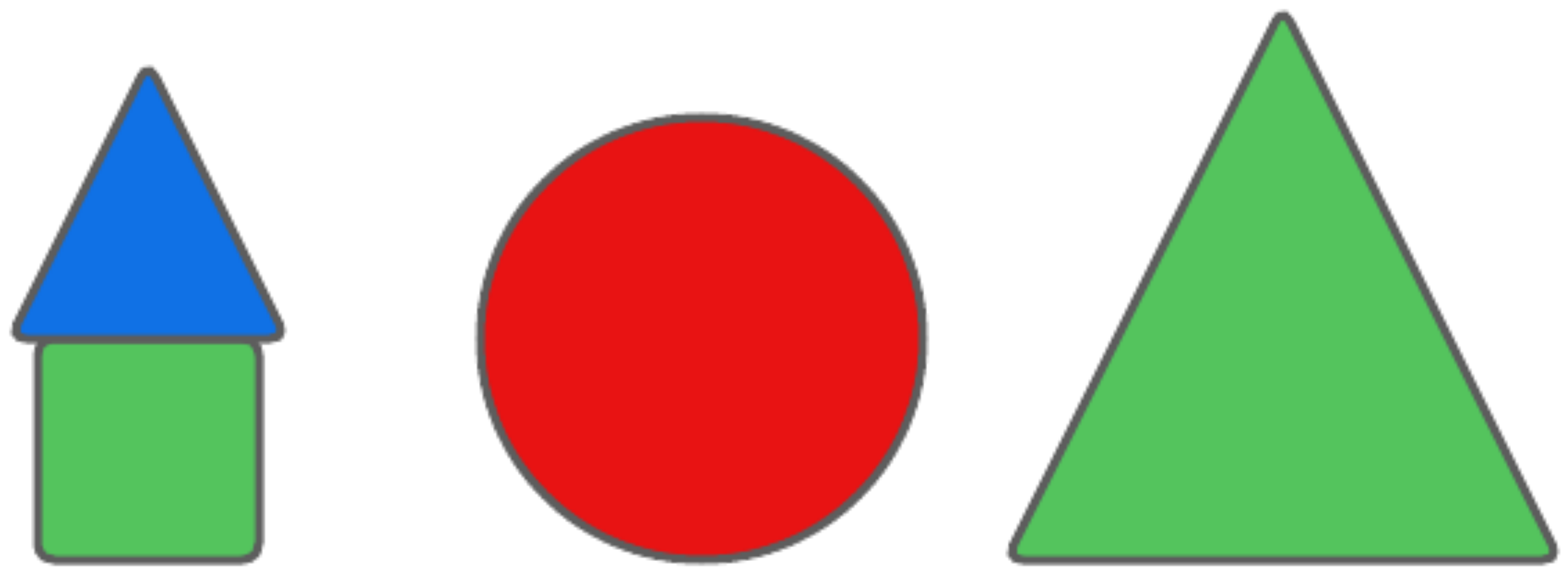

p1_notblue
yes
no
p1_small
p2_notblue
yes
p1_contact
_p2
p2_small
no
p3_notblue
yes
no
yes
no
positive
p1_contact
_p3
...
p4_notblue
yes
no
yes
no
...
...
negative

# Zendo in DT

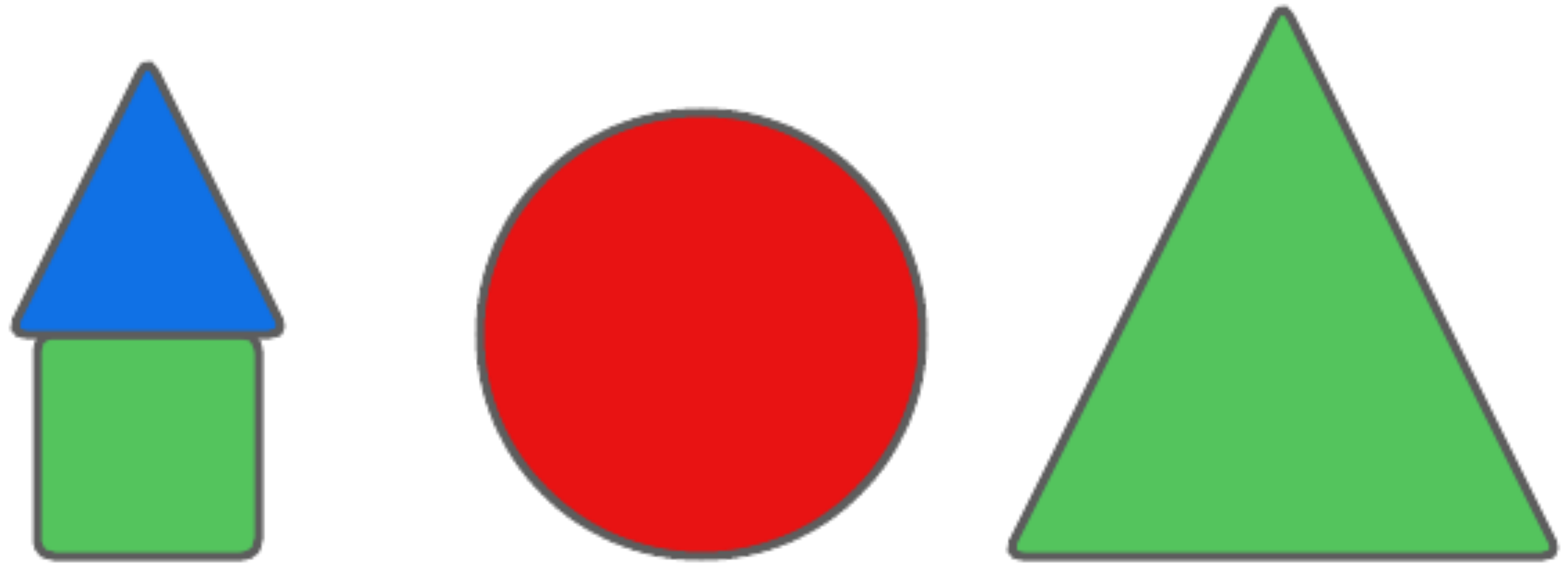


p1_notblue
yes
no
p1_small
p2_notblue
yes
p1_contact
_p2
p2_small
no
p3_notblue
yes
no
yes
positive
p1_contact
_p3
yes
...
no
p4_notblue
yes
no
no
positive
...
...
negative

# Zendo in ILP

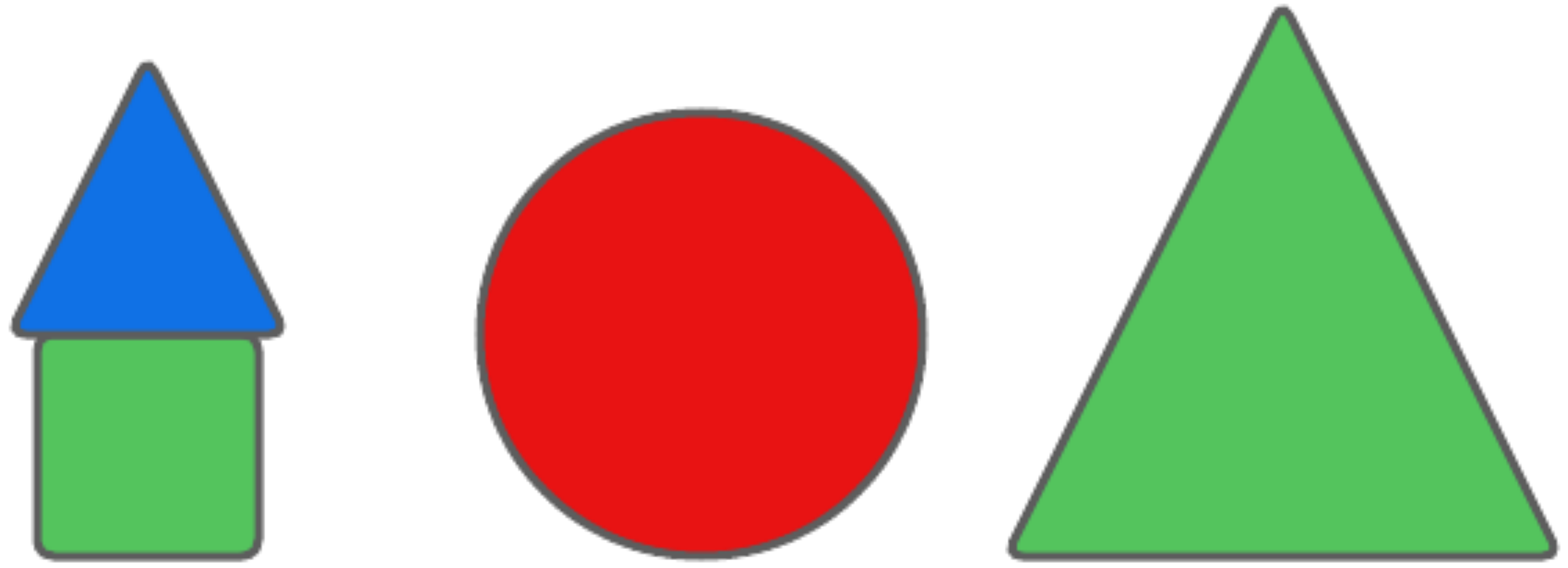


# Zendo in ILP

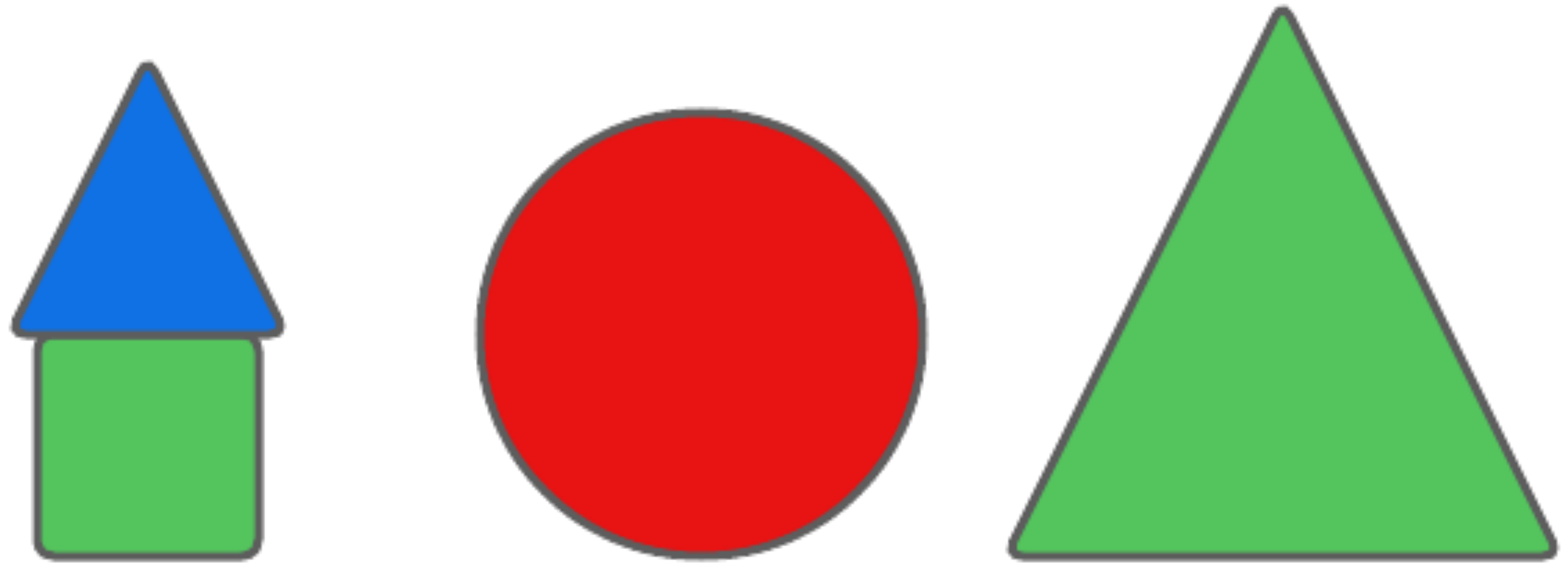


```
% positive example
pos(zendo(structure1)).

% background knowledge
piece(structure1, p1).
piece(structure1, p2).
green(p1).
blue(p2).
small(p1).
small(p2).
contact(p1,p2).
x_pos(p1,1).
x_pos(p2,1).
```

# Zendo in ILP



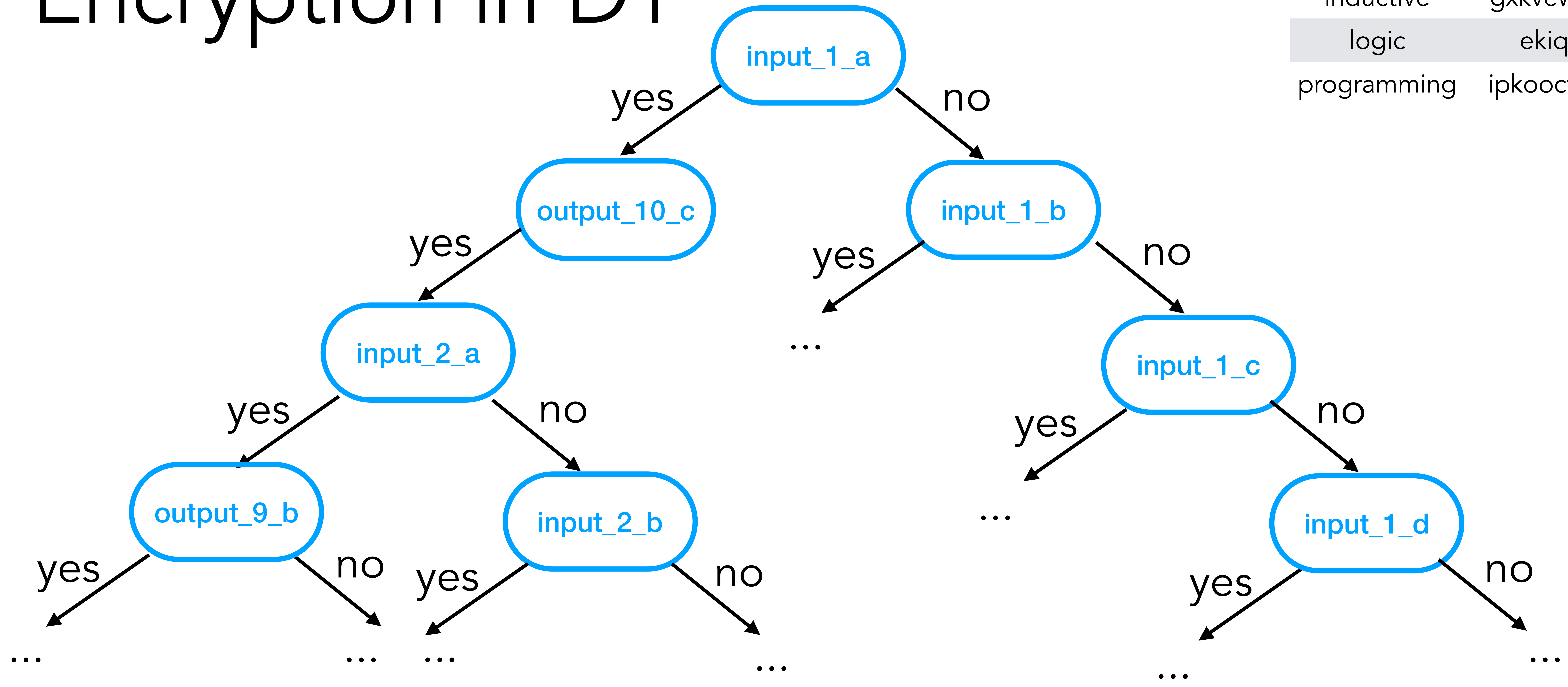
# Zendo in ILP



```
zendo(A):-
    piece(A,C),
    contact(C,B),
    size(B,E),
    small(E),
    color(B,D),
    not_blue(D).
```

# Encryption in DT

| Input | Output |
|---|---|
| inductive | gxkvewfpk |
| logic | ekiqn |
| programming | ipkooctiqtr |



# Encryption in ILP

# Encryption in ILP

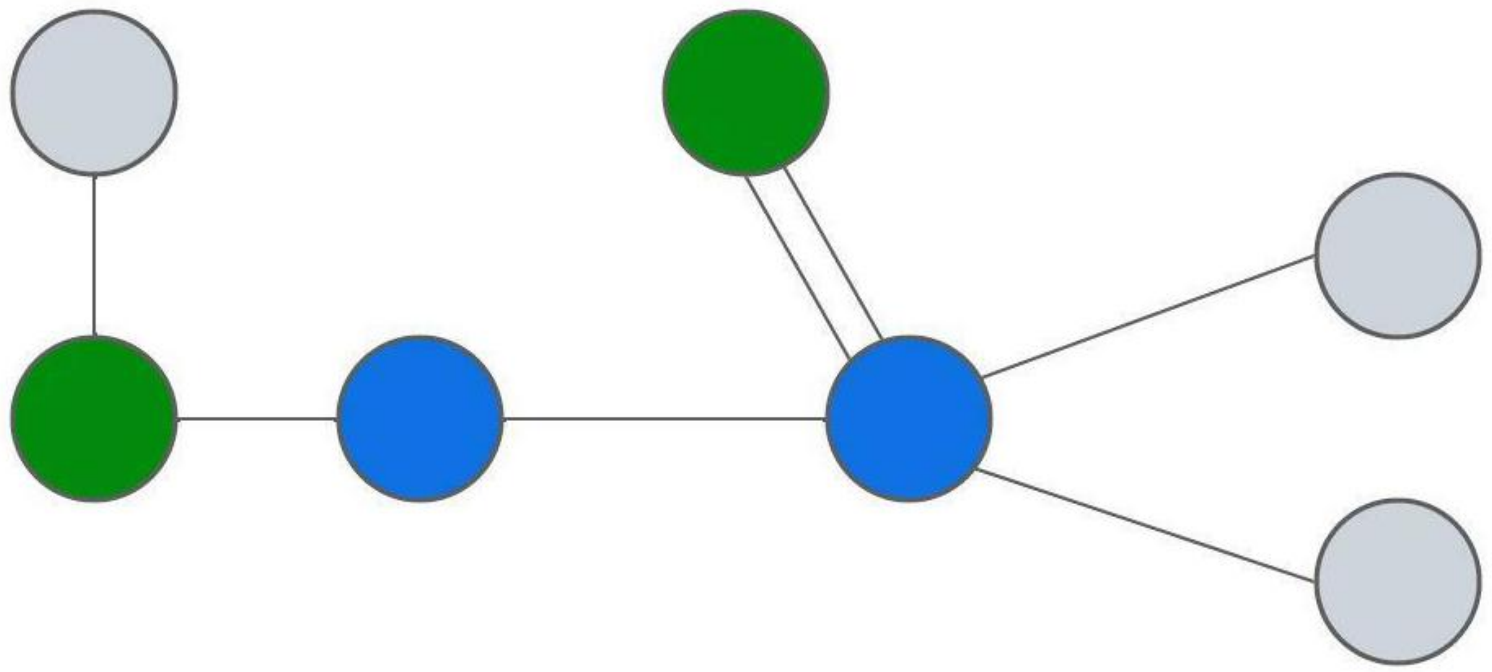
```
% positive examples
pos(f([i,n,d,u,c,t,i,v,e],[g,x,k,v,e,w,f,p,k])).
pos(f([l,o,g,i,c],[e,k,i,q,n])).
pos(f([p,r,o,g,r,a,m,m,i,n,g],[i,p,k,o,o,c,t,i,q,t,r])).

% background knowledge
head([H|_], H).
tail([_|T], T).
empty([]).
succ(A,B) :- B is A+1.
ord(a,97).
ord(b,98).
inttochar(97,a).
inttochar(98,b).
...
```

# Encryption in ILP

| Input | Output |
|---|---|
| inductive | gxkvewfpk |
| logic | ekiqn |
| programming | ipkooctiqtr |

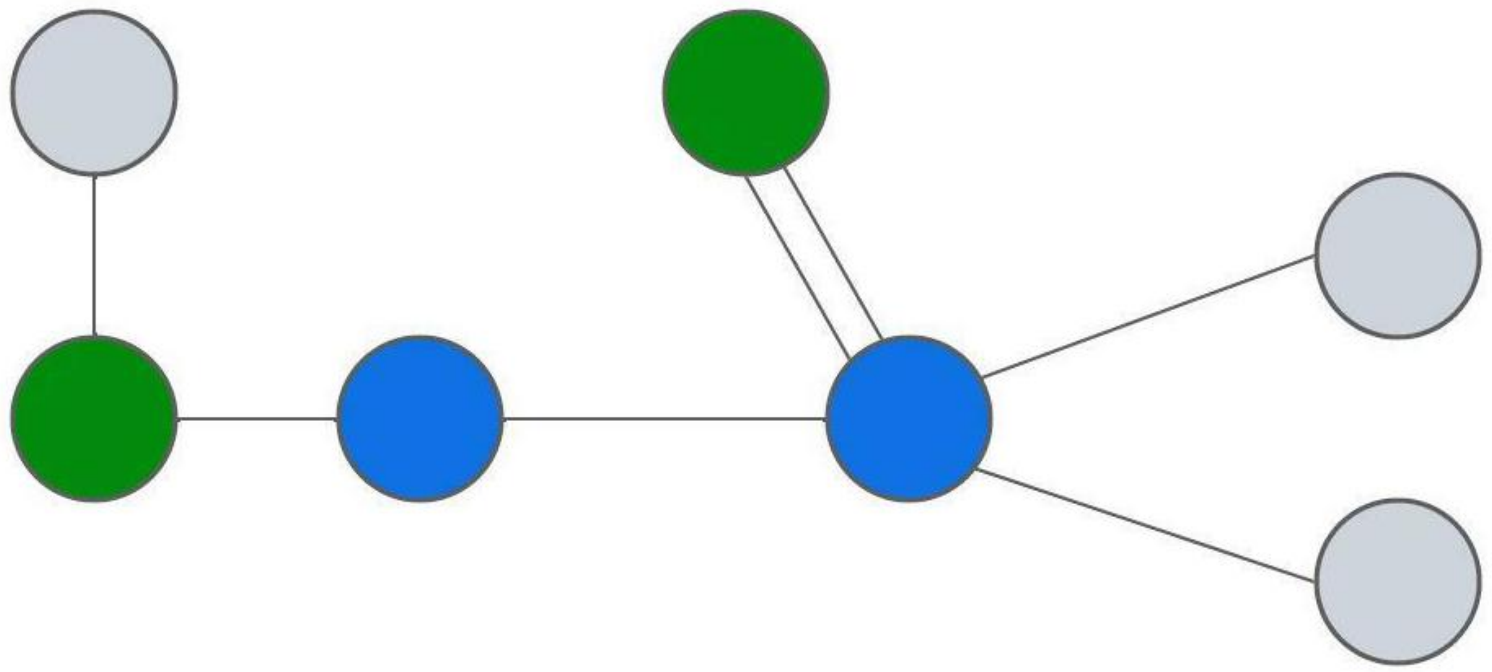
```
encryption(A,B):-
  map(A,C,inv_1),
  reverse(C,B).
inv_1(A,B):-
  ord(A,E),
  succ(E,C),
  succ(C,D),
  inttochar(D,B).
```

# Networks in DT



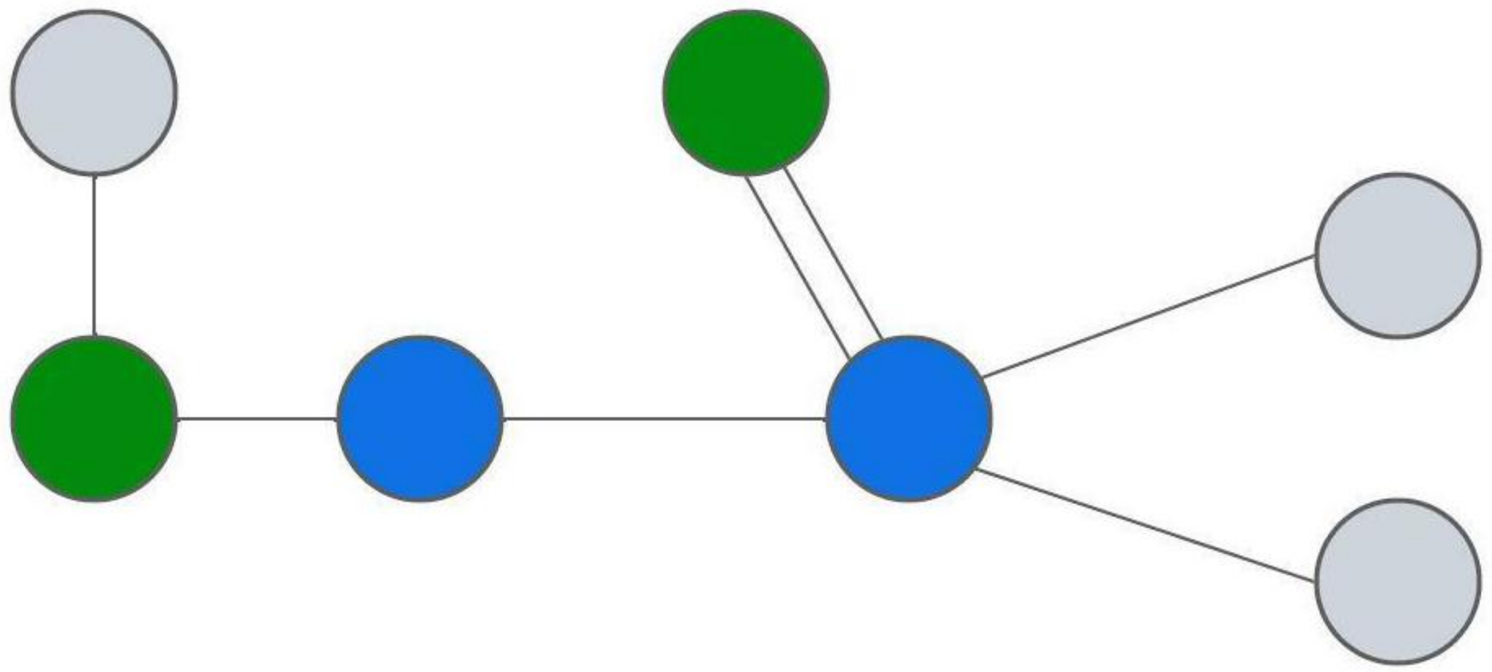
# Networks in DT

a1_hacc



# Networks in DT

a1_hacc

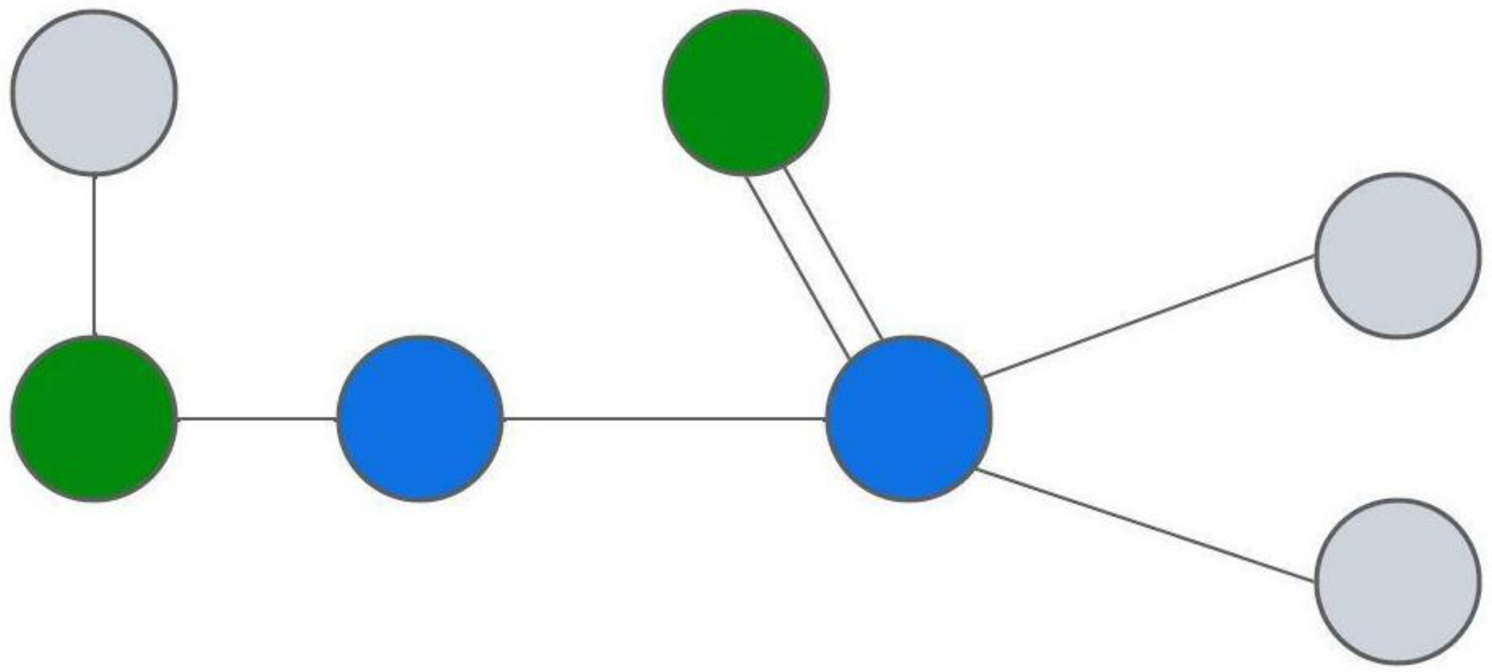


yes

# Networks in DT

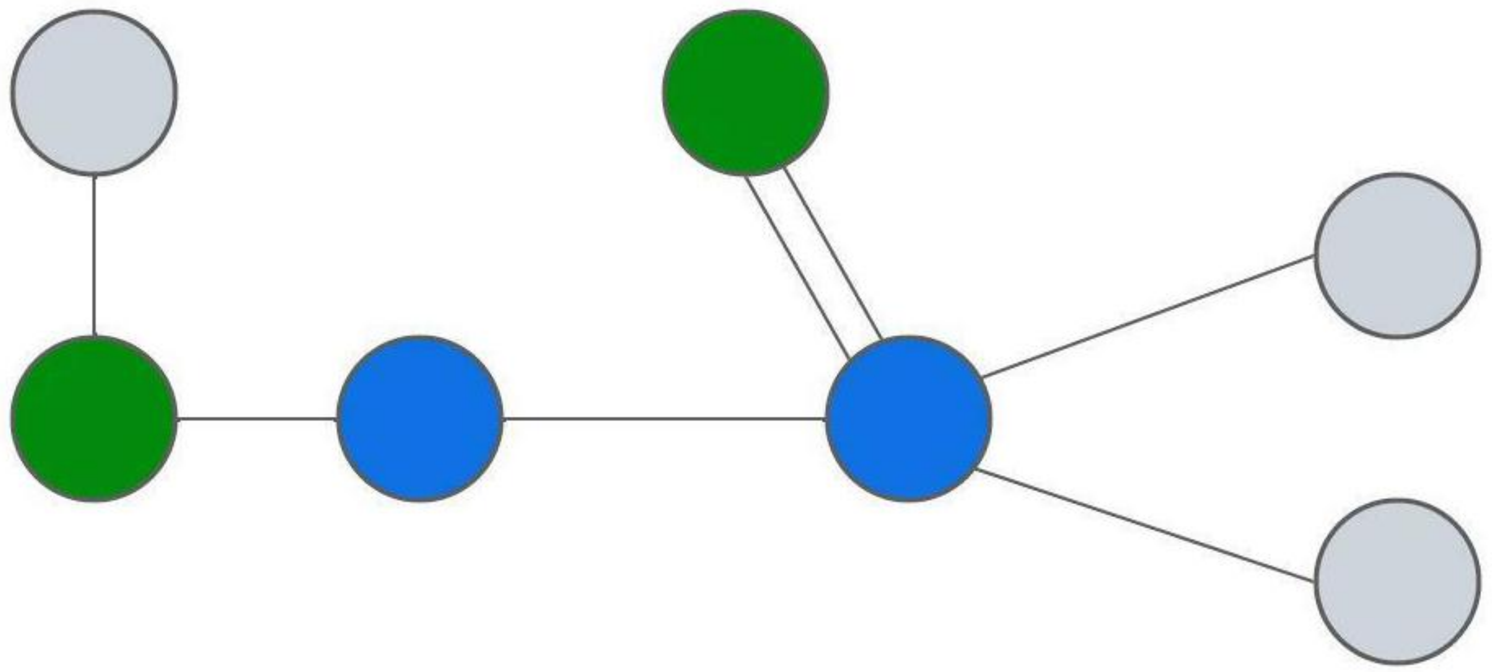
yes

a1_hacc



yes

# Networks in DT



a1_hacc

yes

no

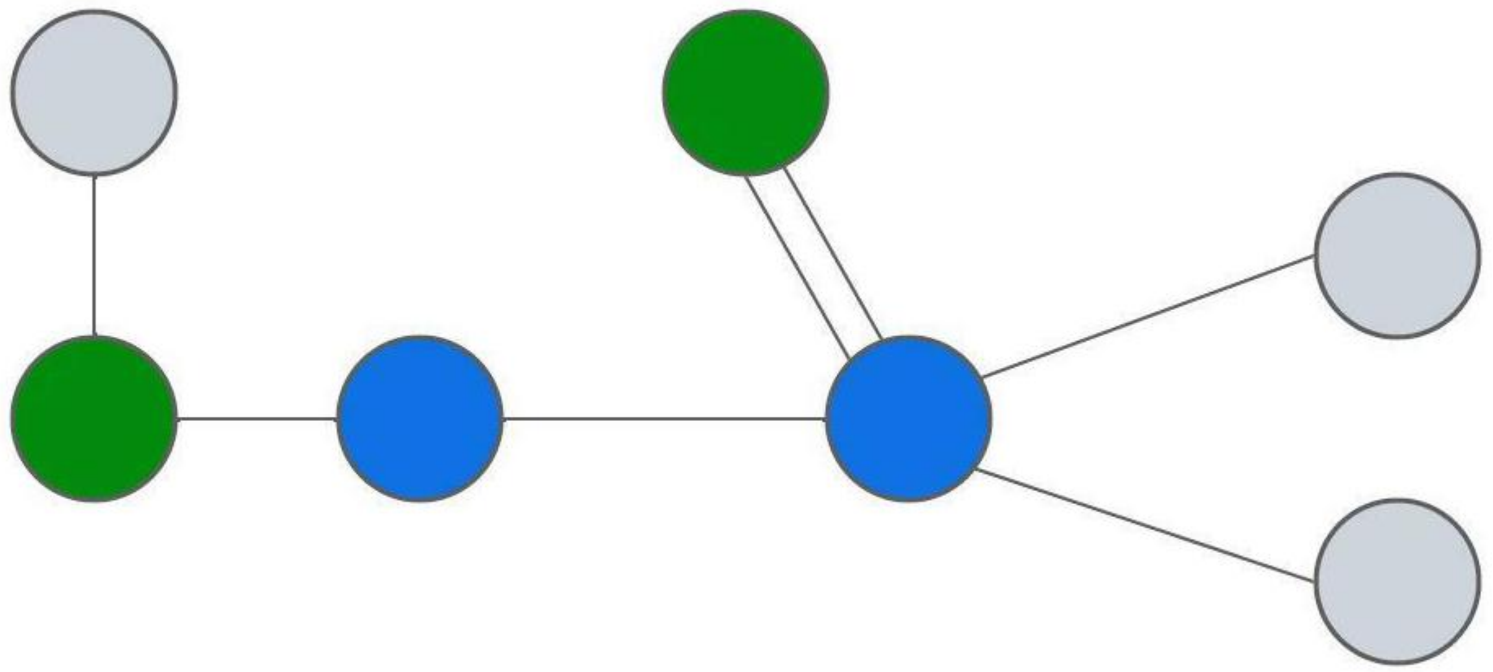
yes

# Networks in DT

a1_hacc

yes

no

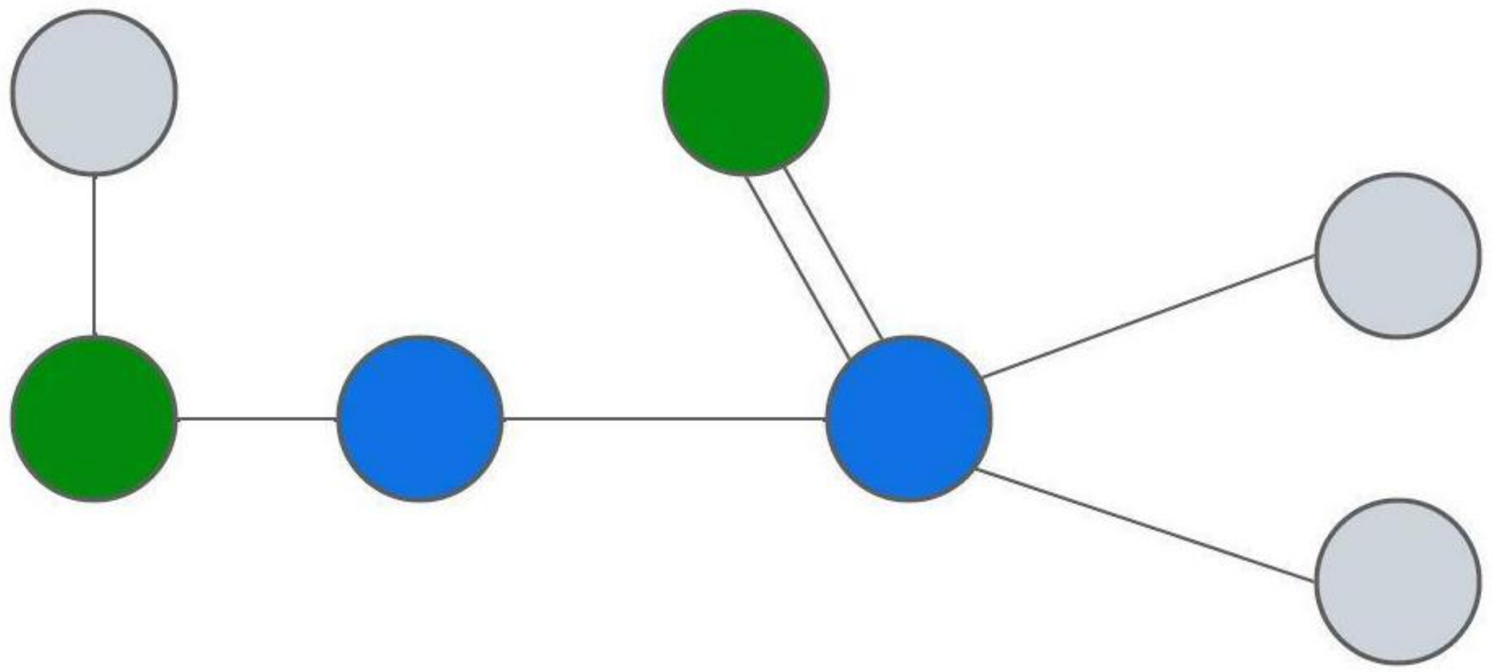


yes

positive

# Networks in DT

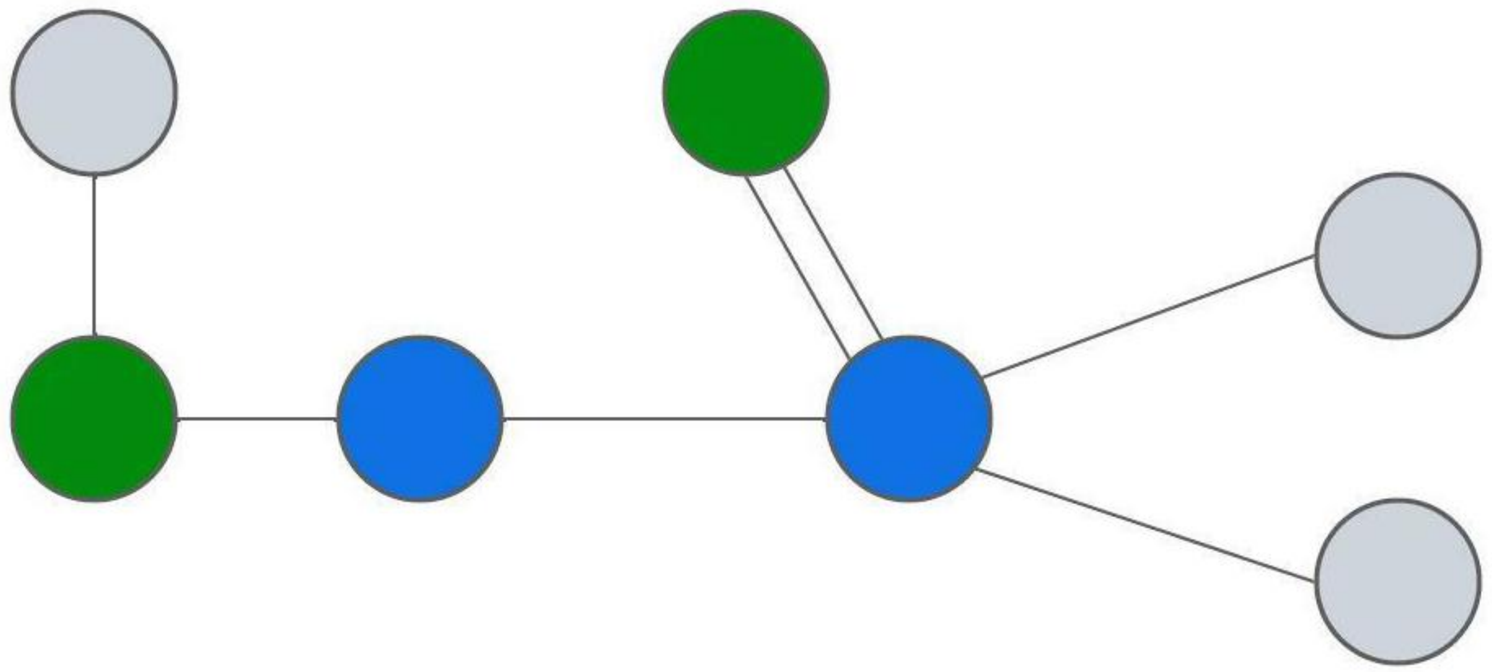
yes a1_hacc no



yes

positive

# Networks in DT



a1_hacc

yes

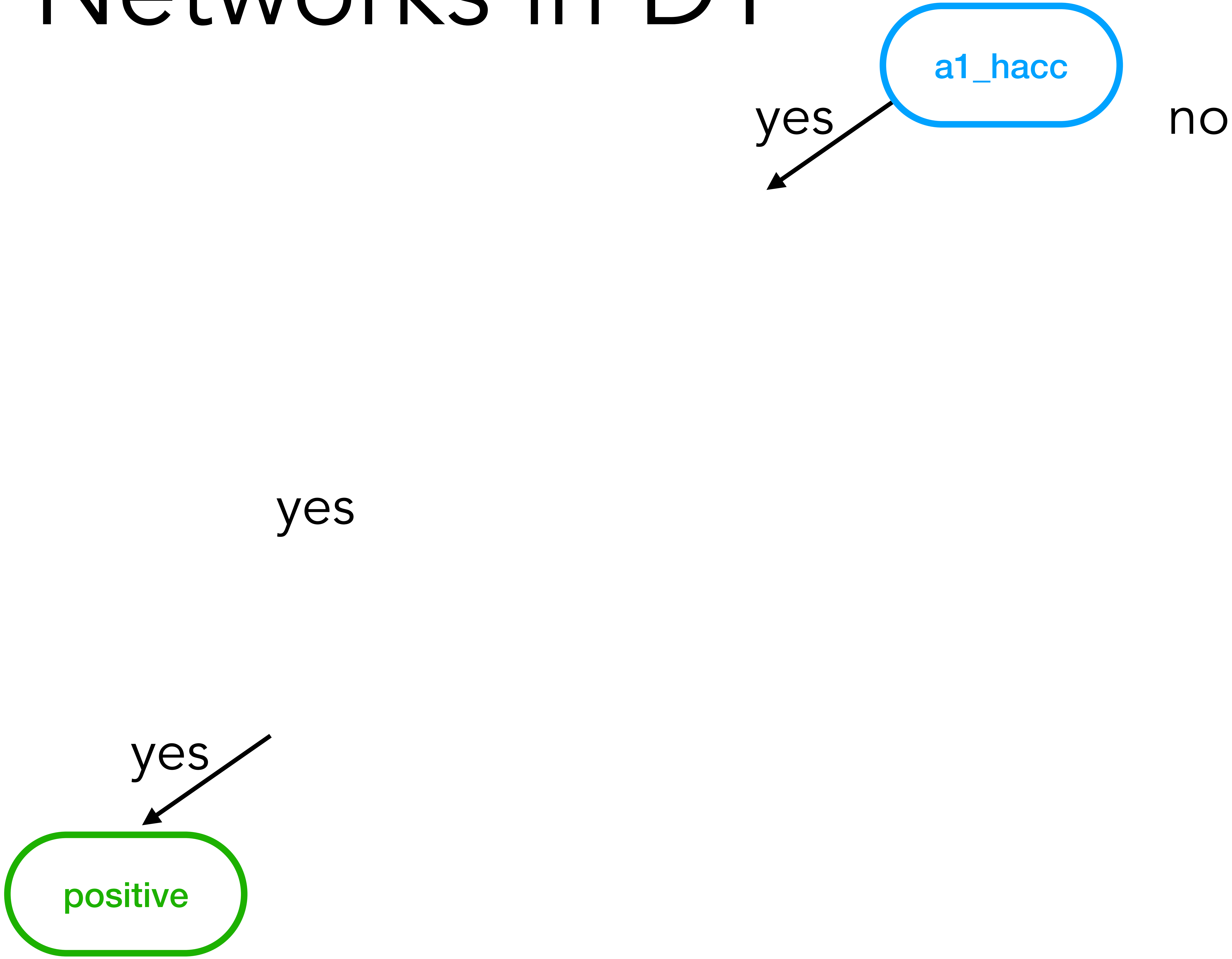
no

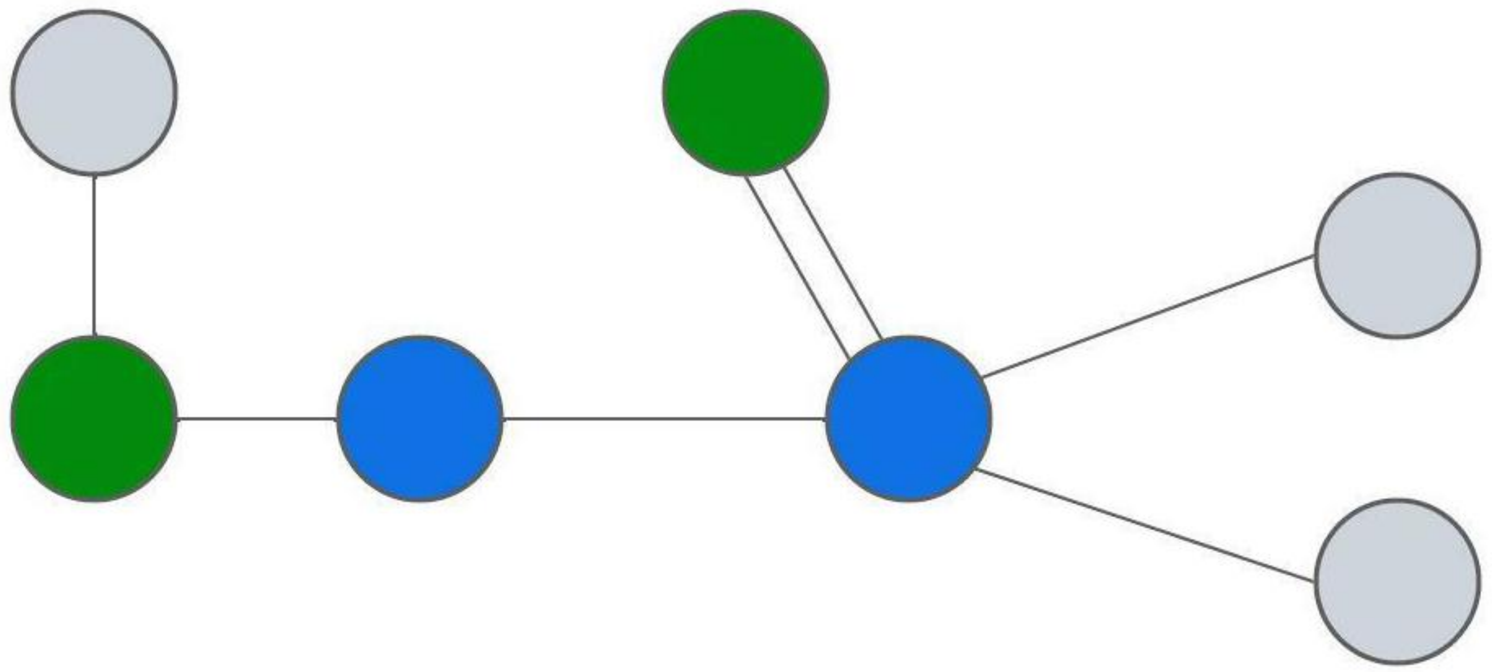
yes

yes

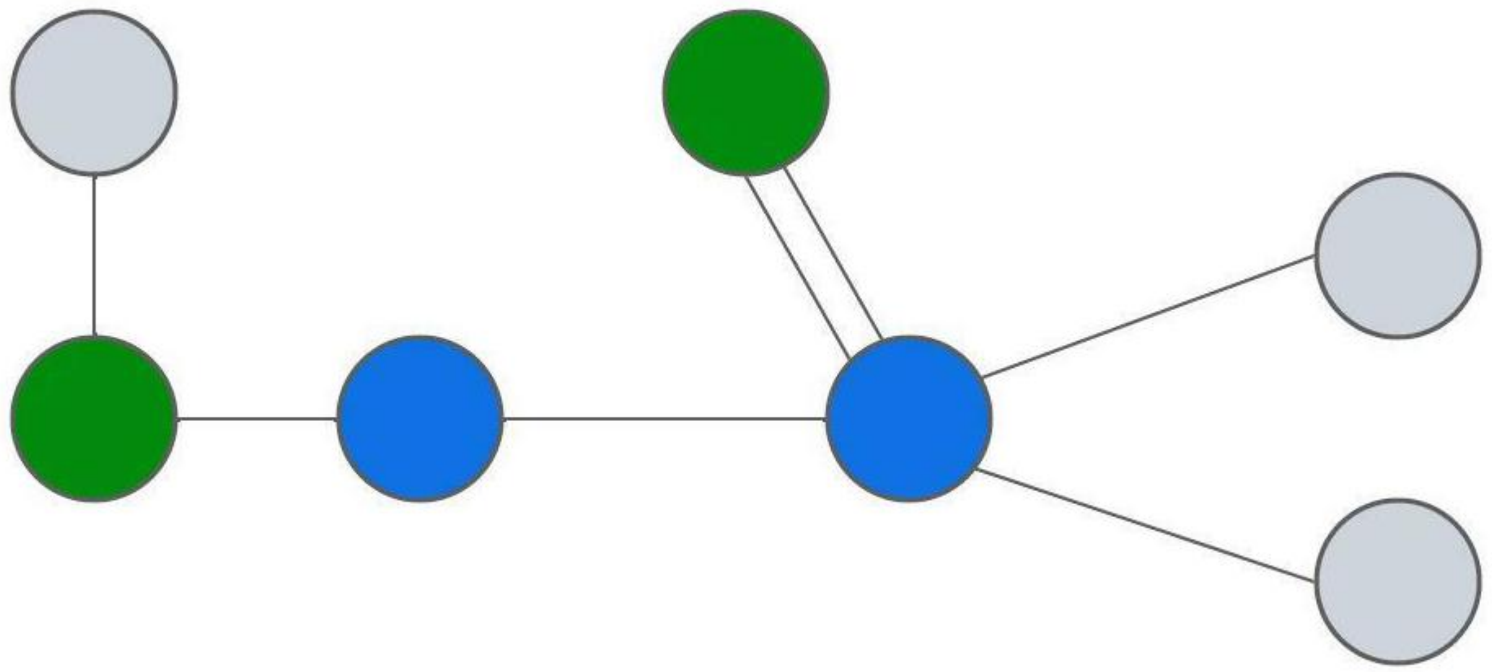
positive

# Networks in DT



# Networks in DT



a1_hacc

yes

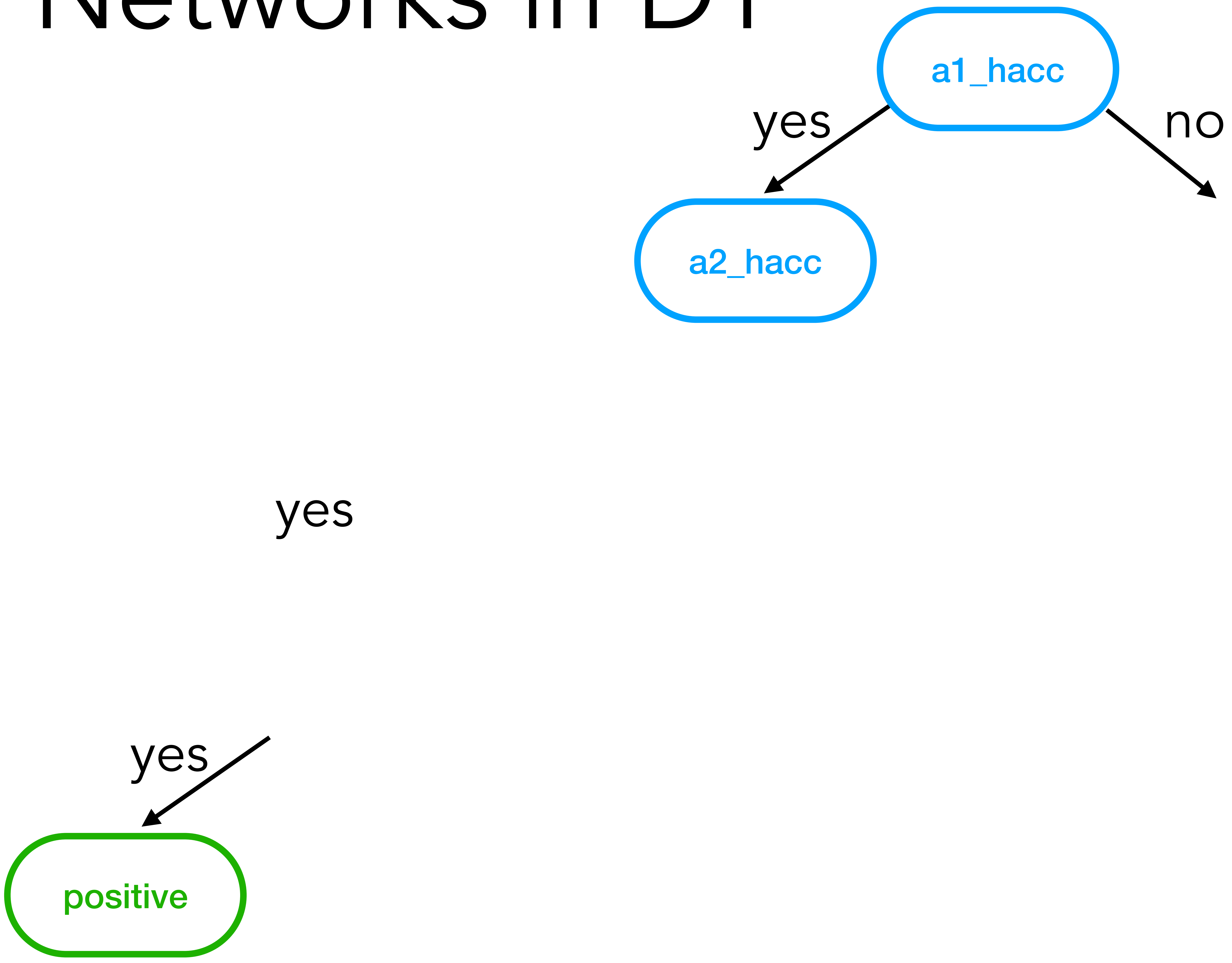
no

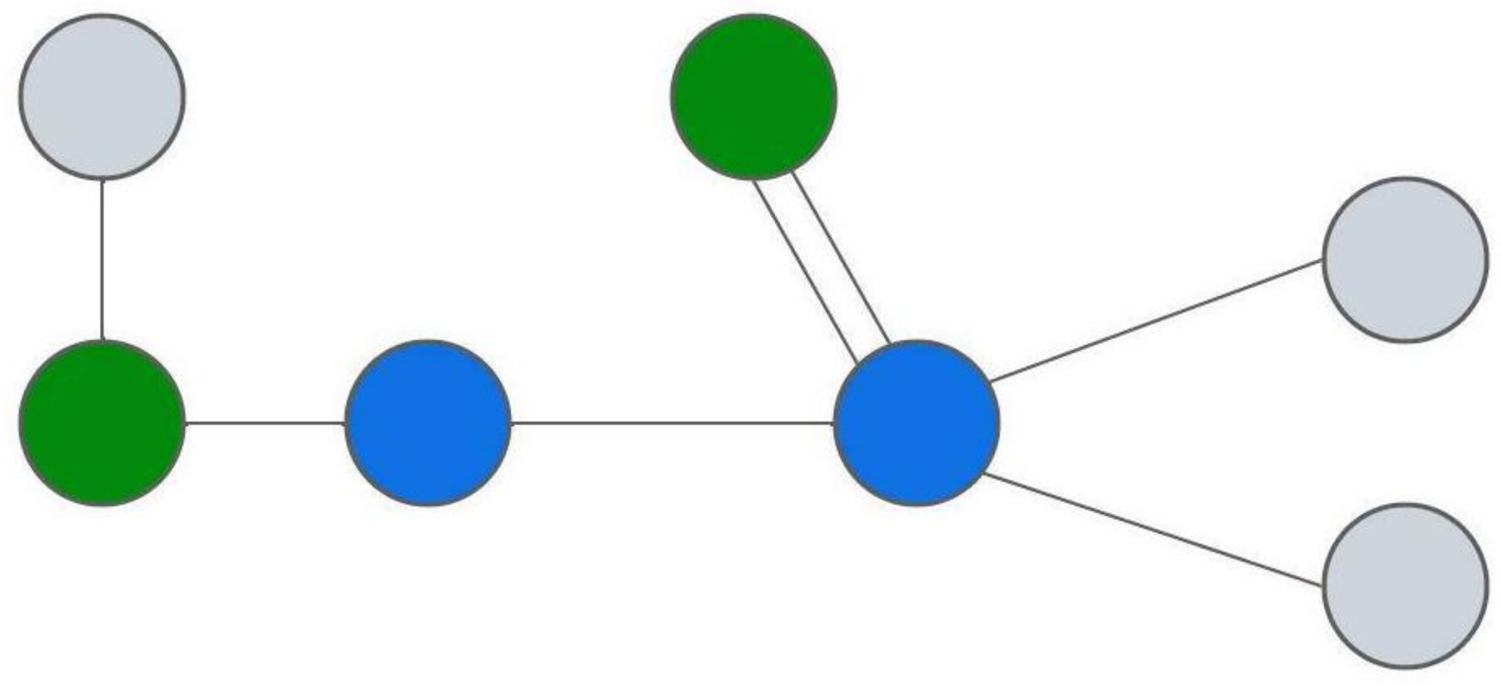
yes

yes

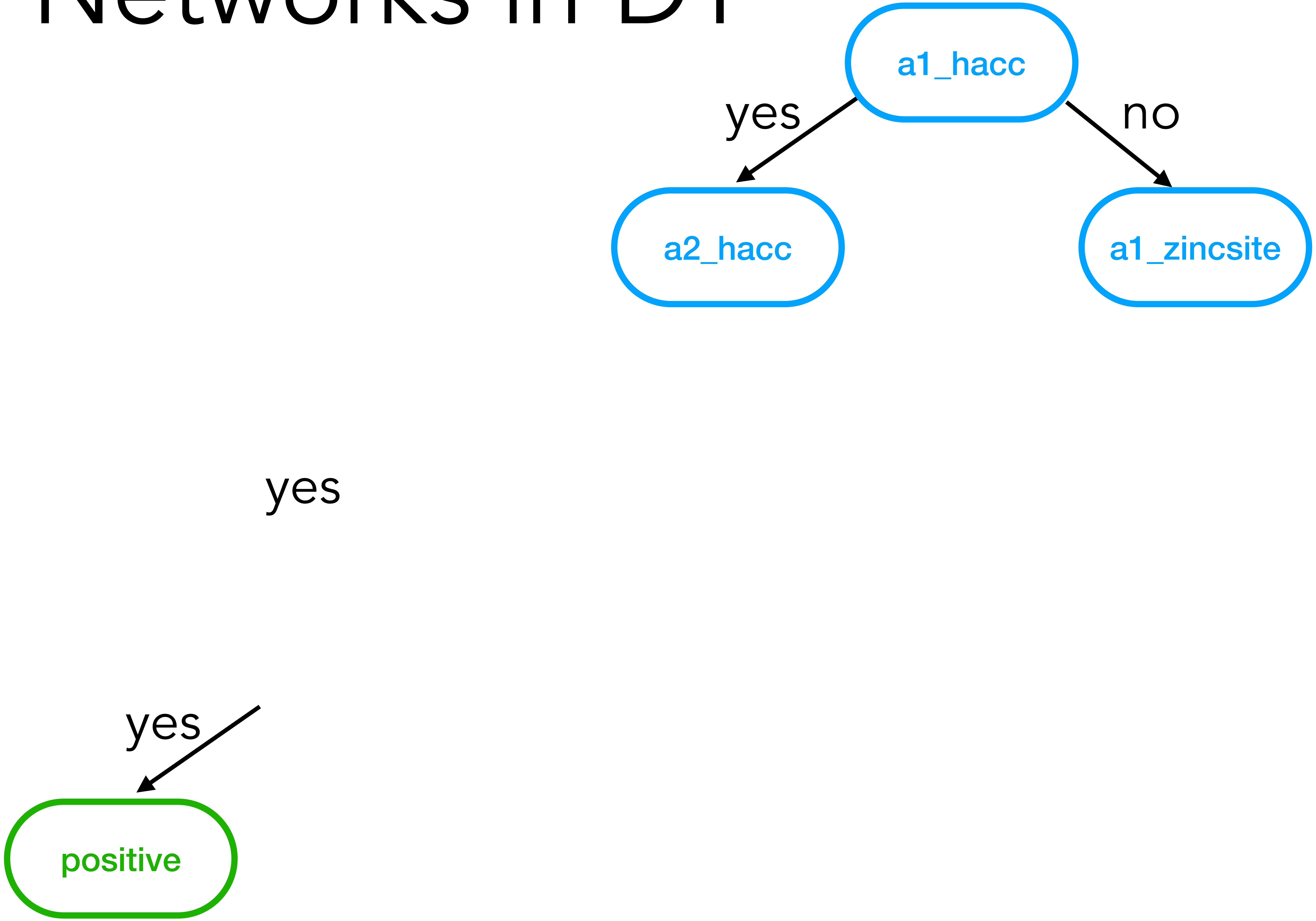
positive

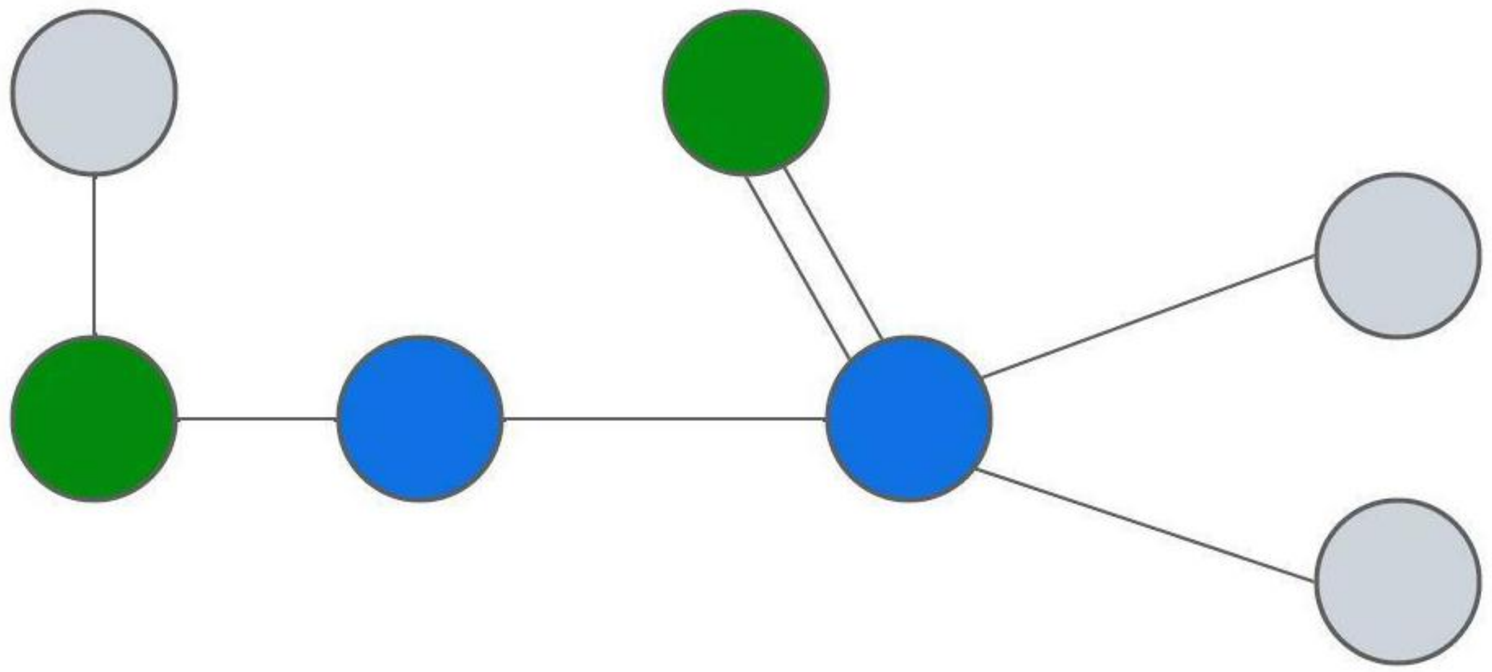
# Networks in DT



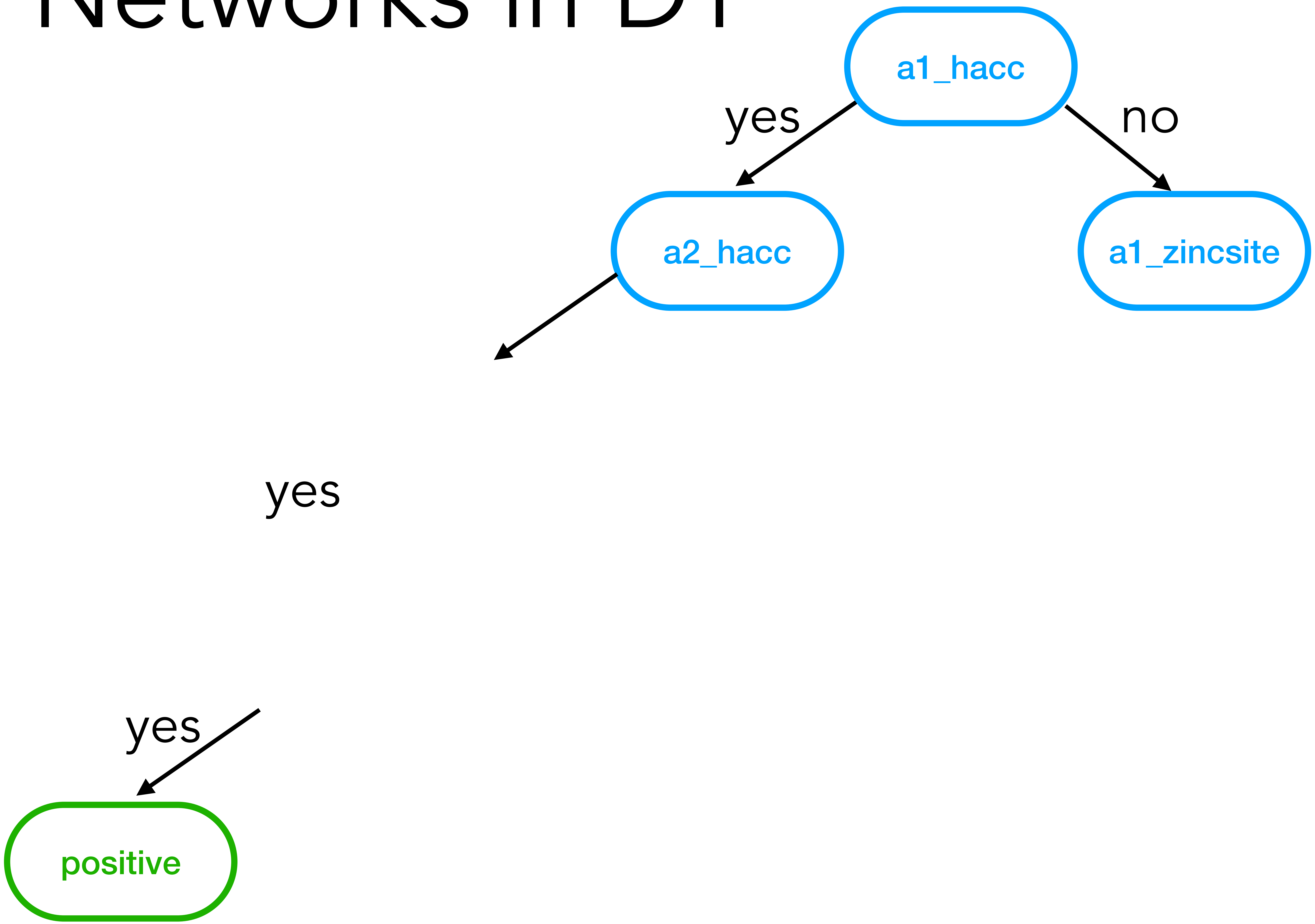



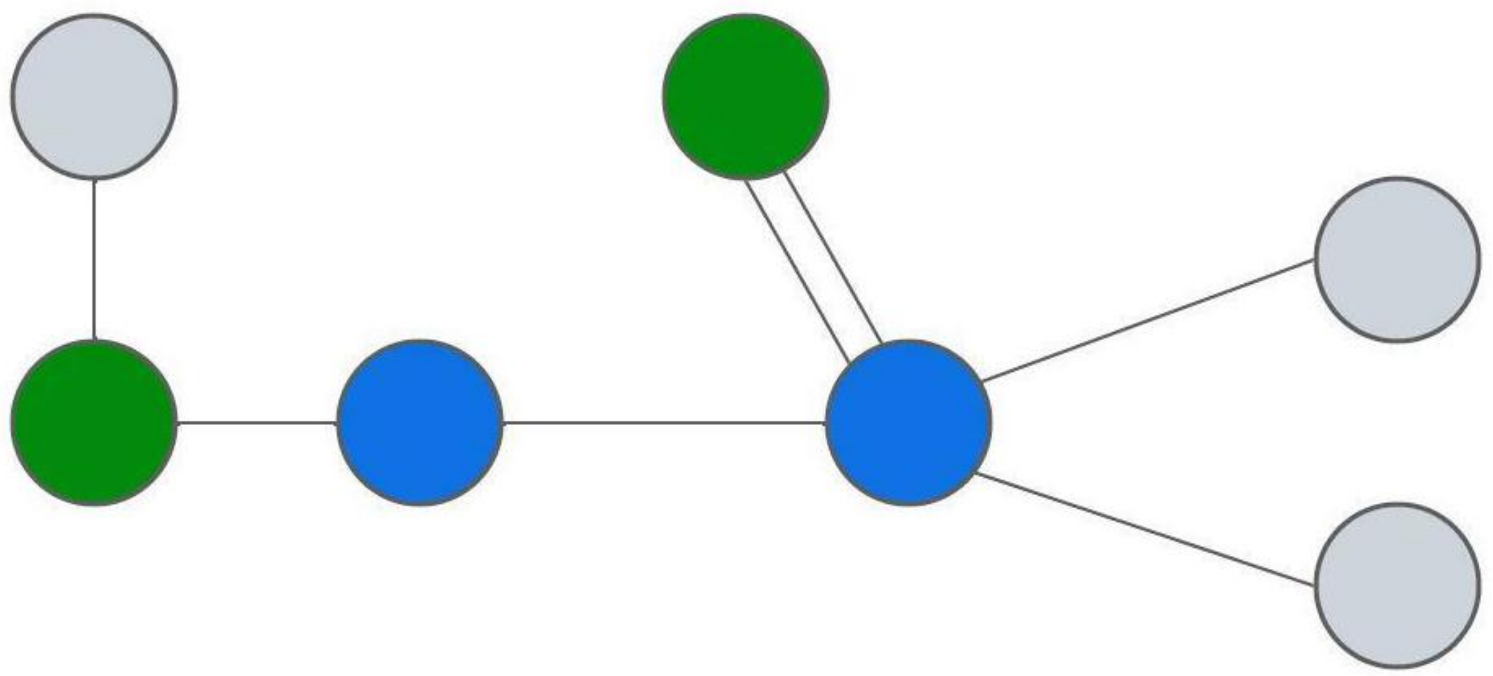
# Networks in DT



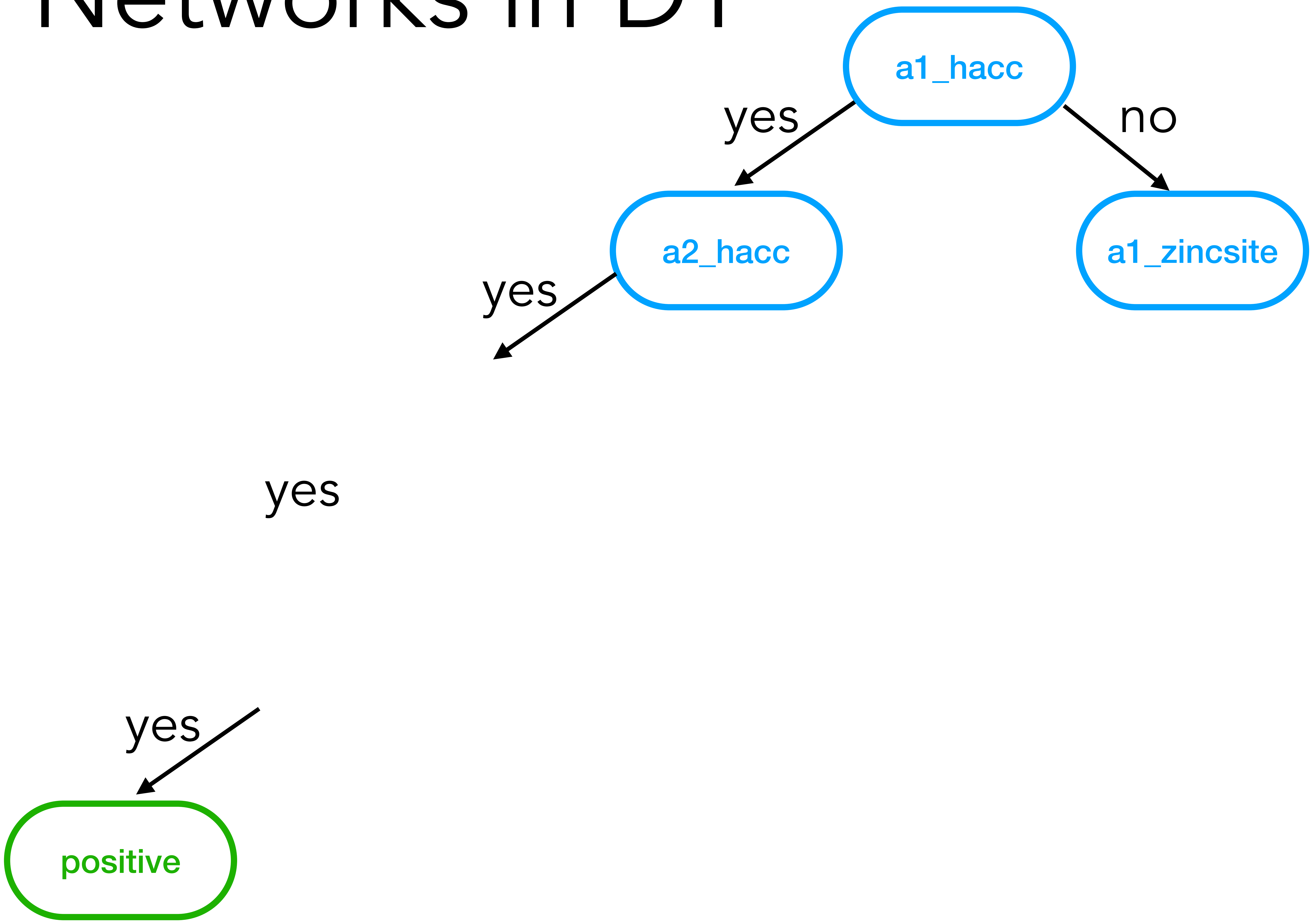



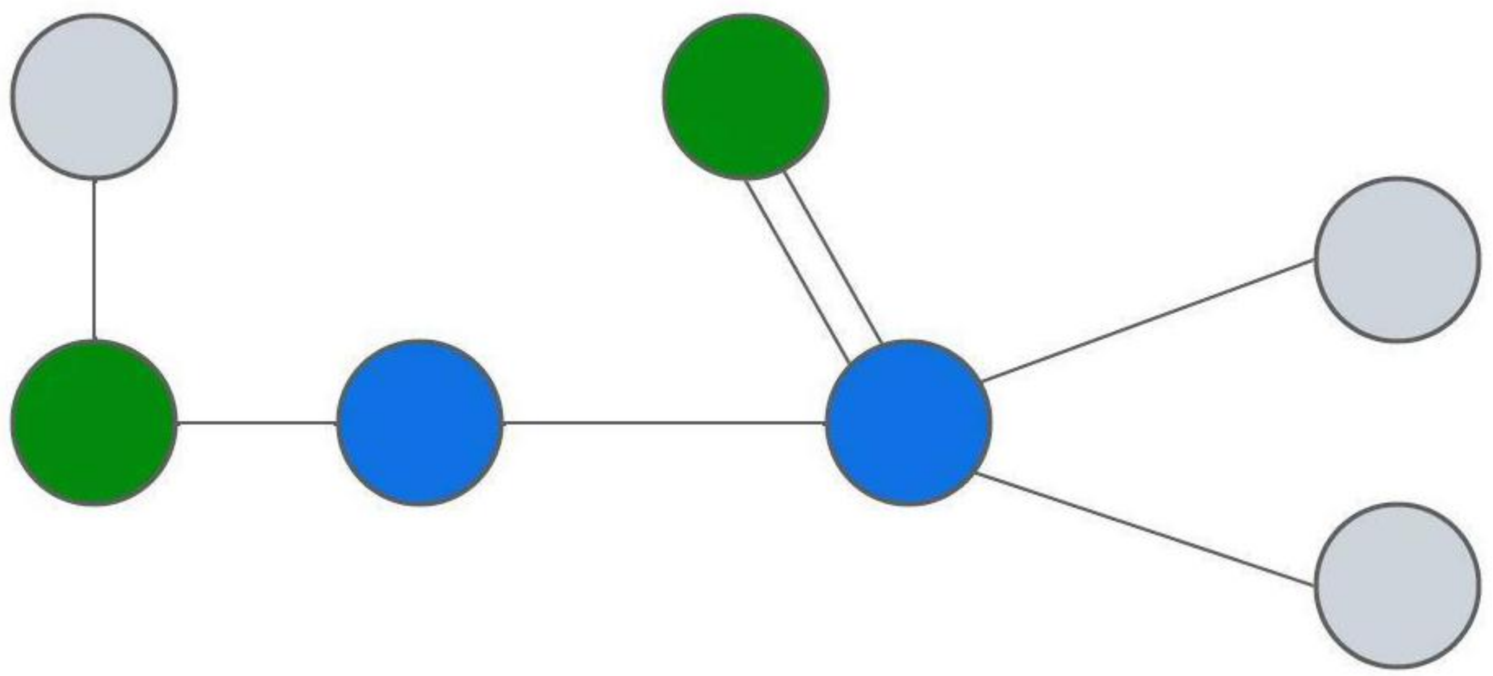
# Networks in DT



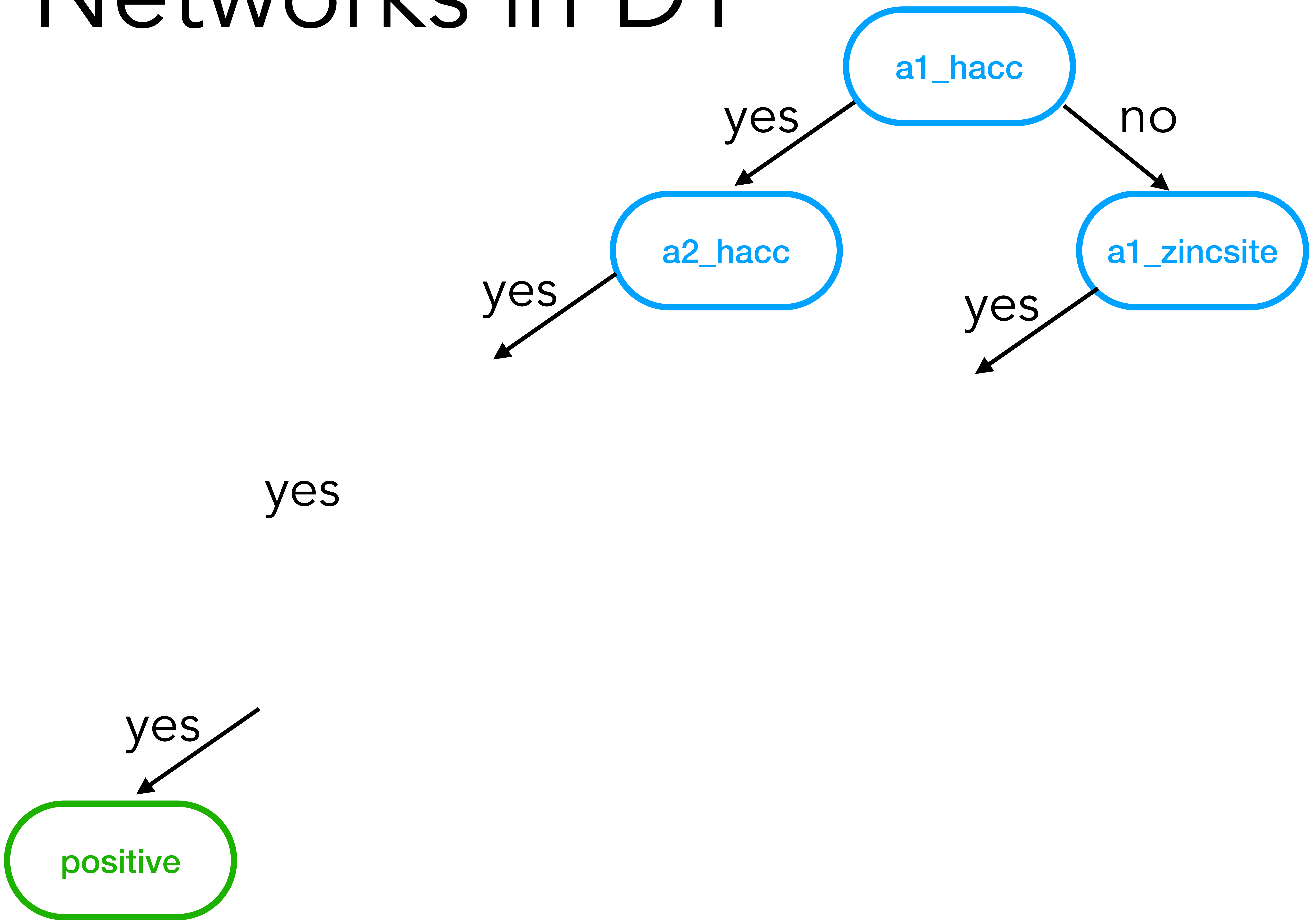
# Networks in DT

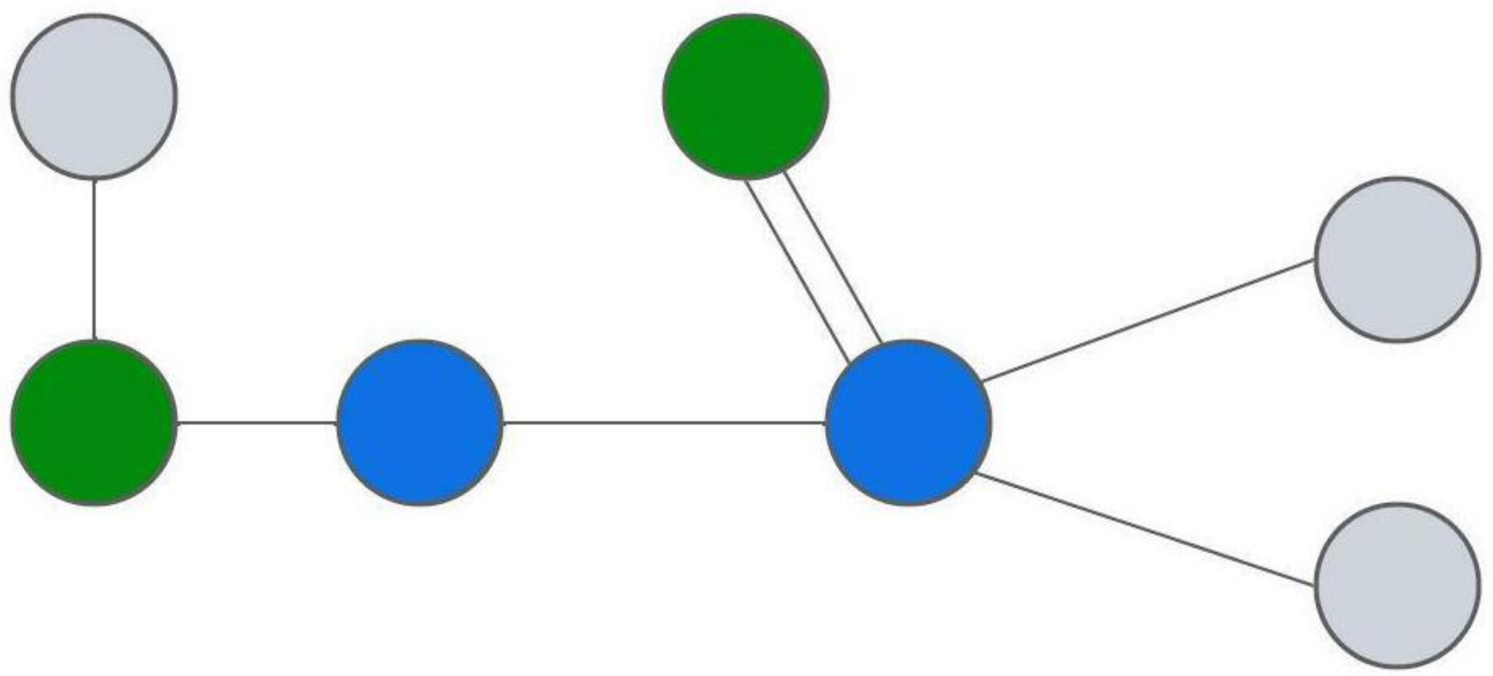


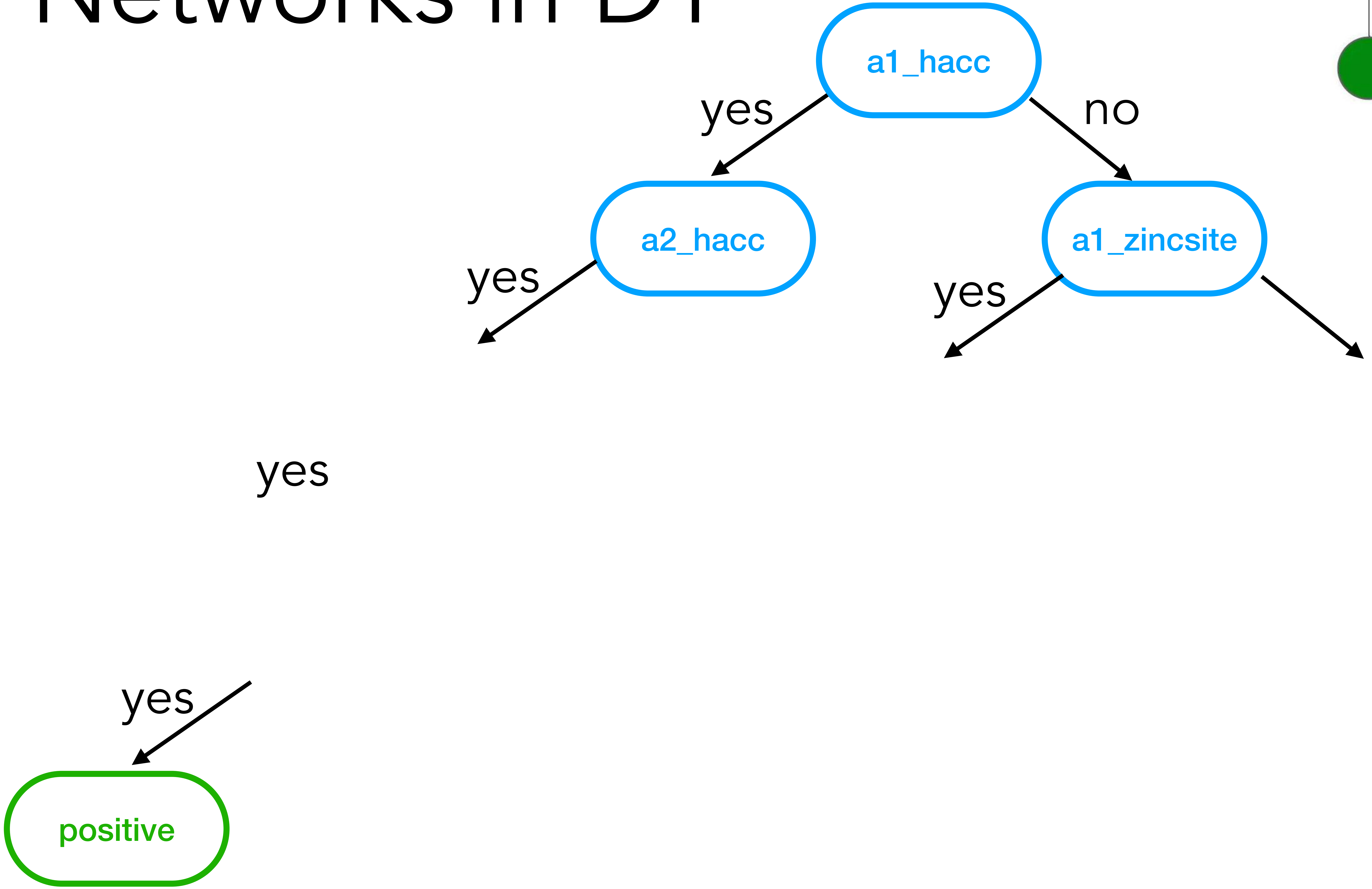
# Networks in DT

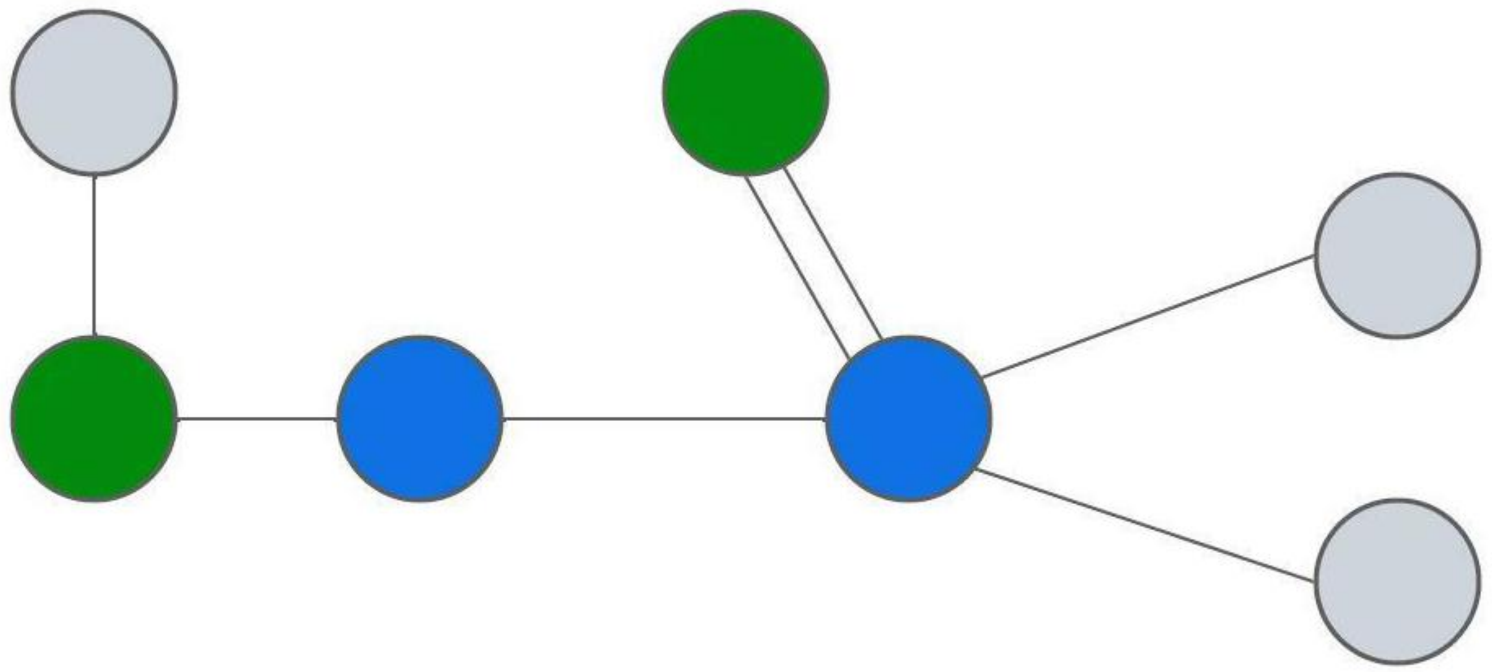
# Networks in DT



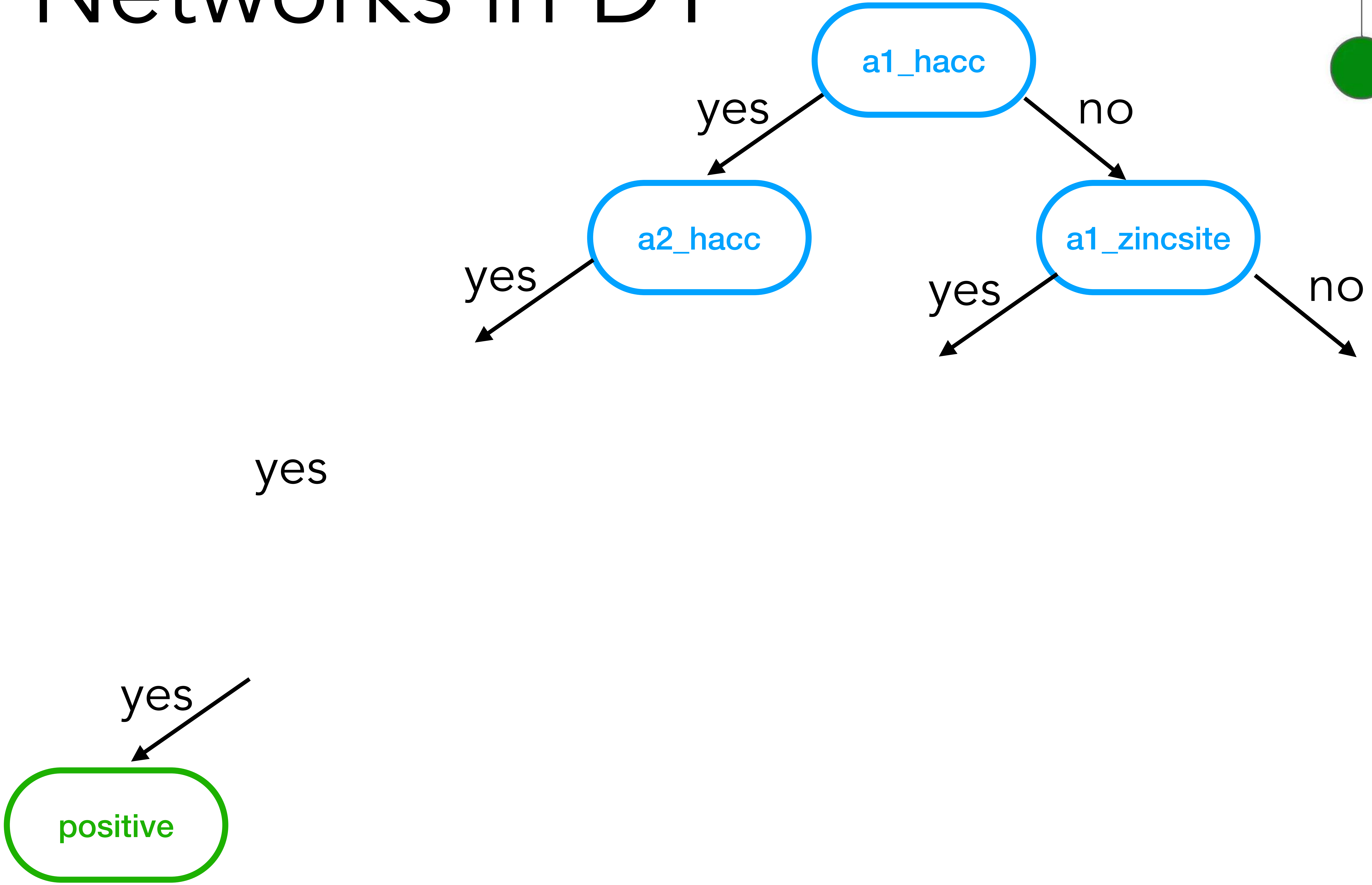



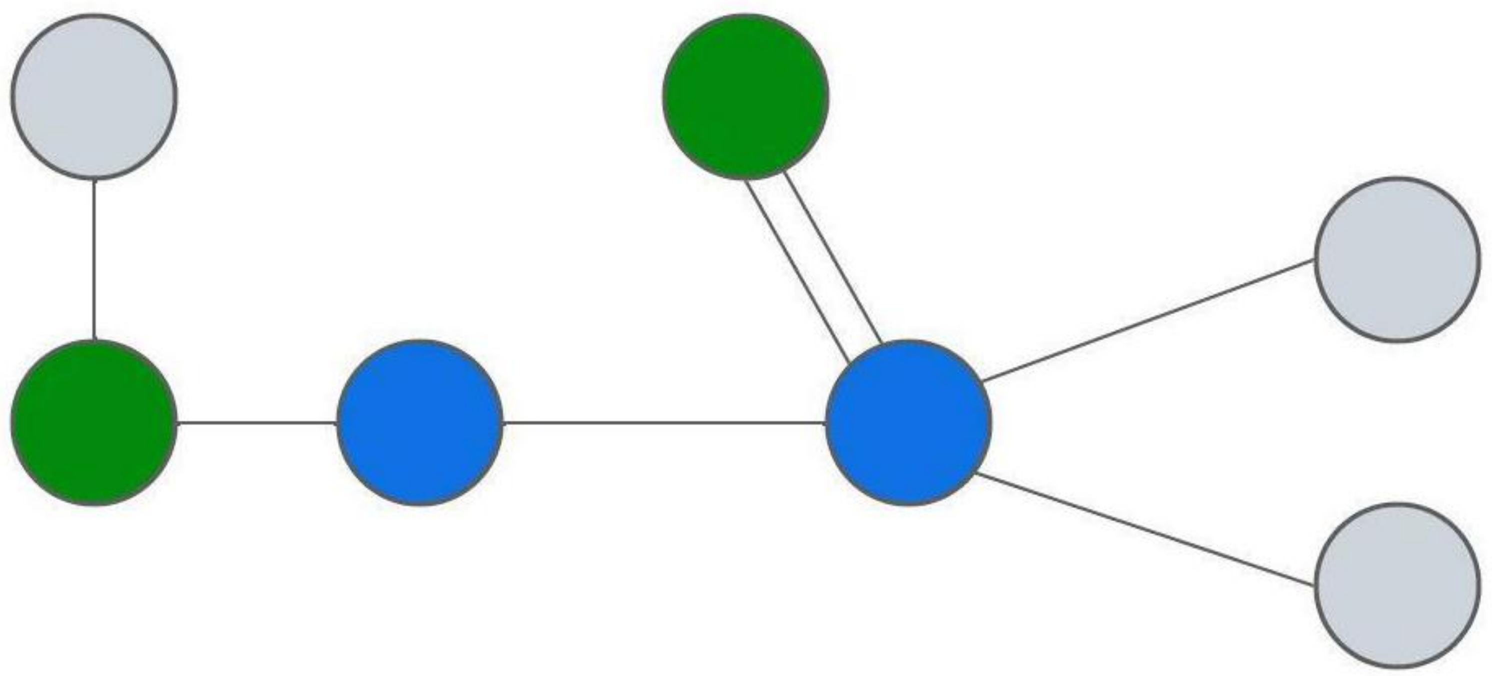
# Networks in DT



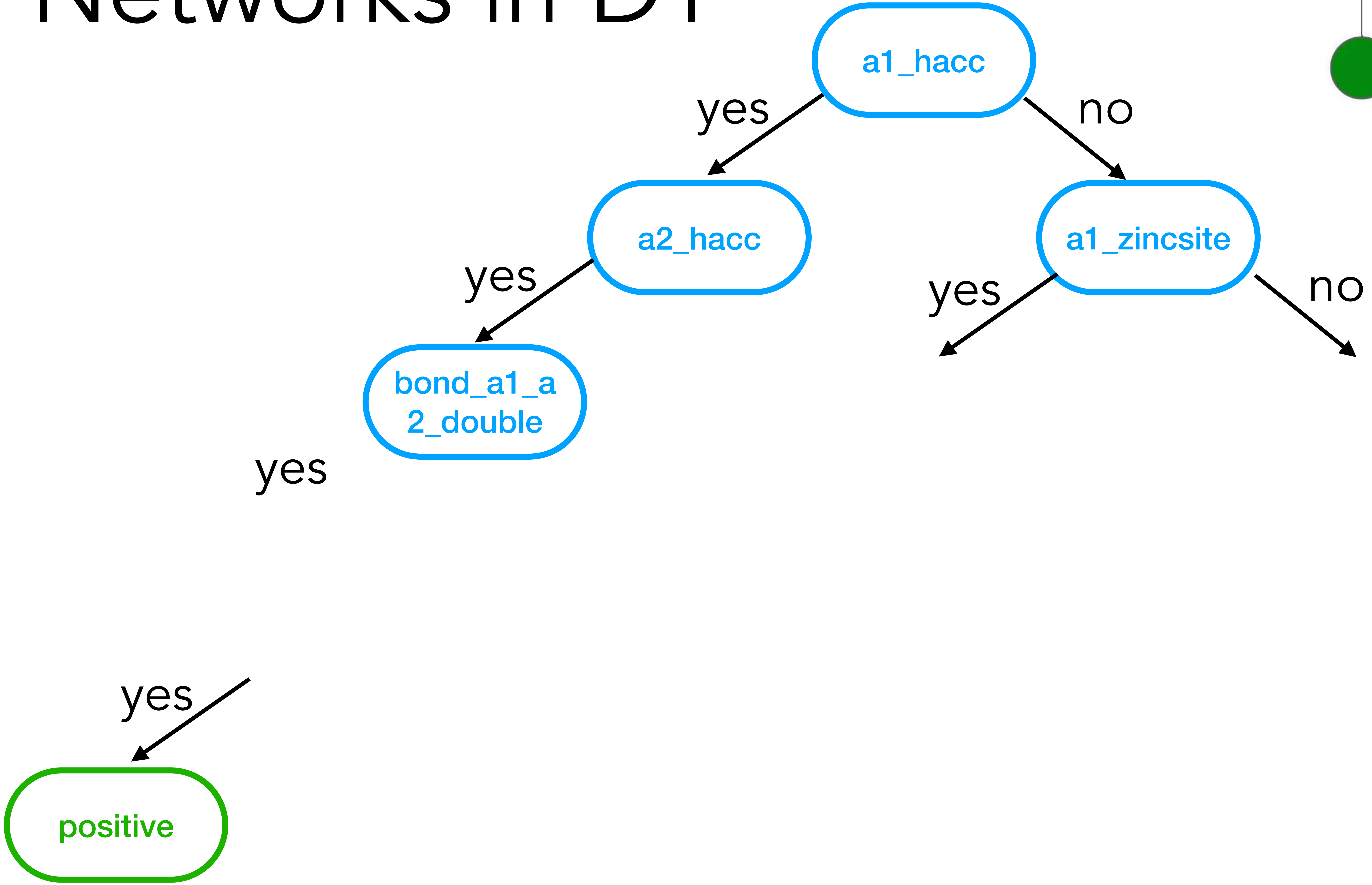



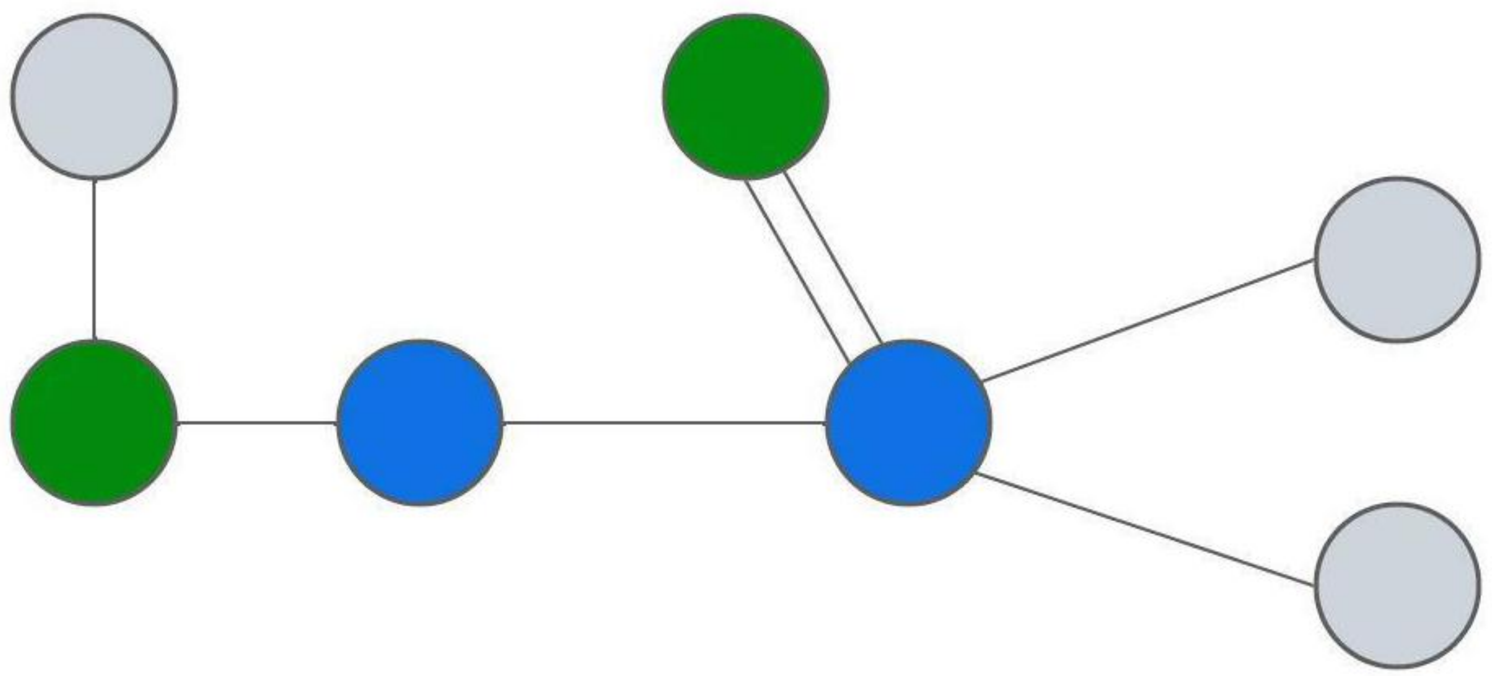
# Networks in DT



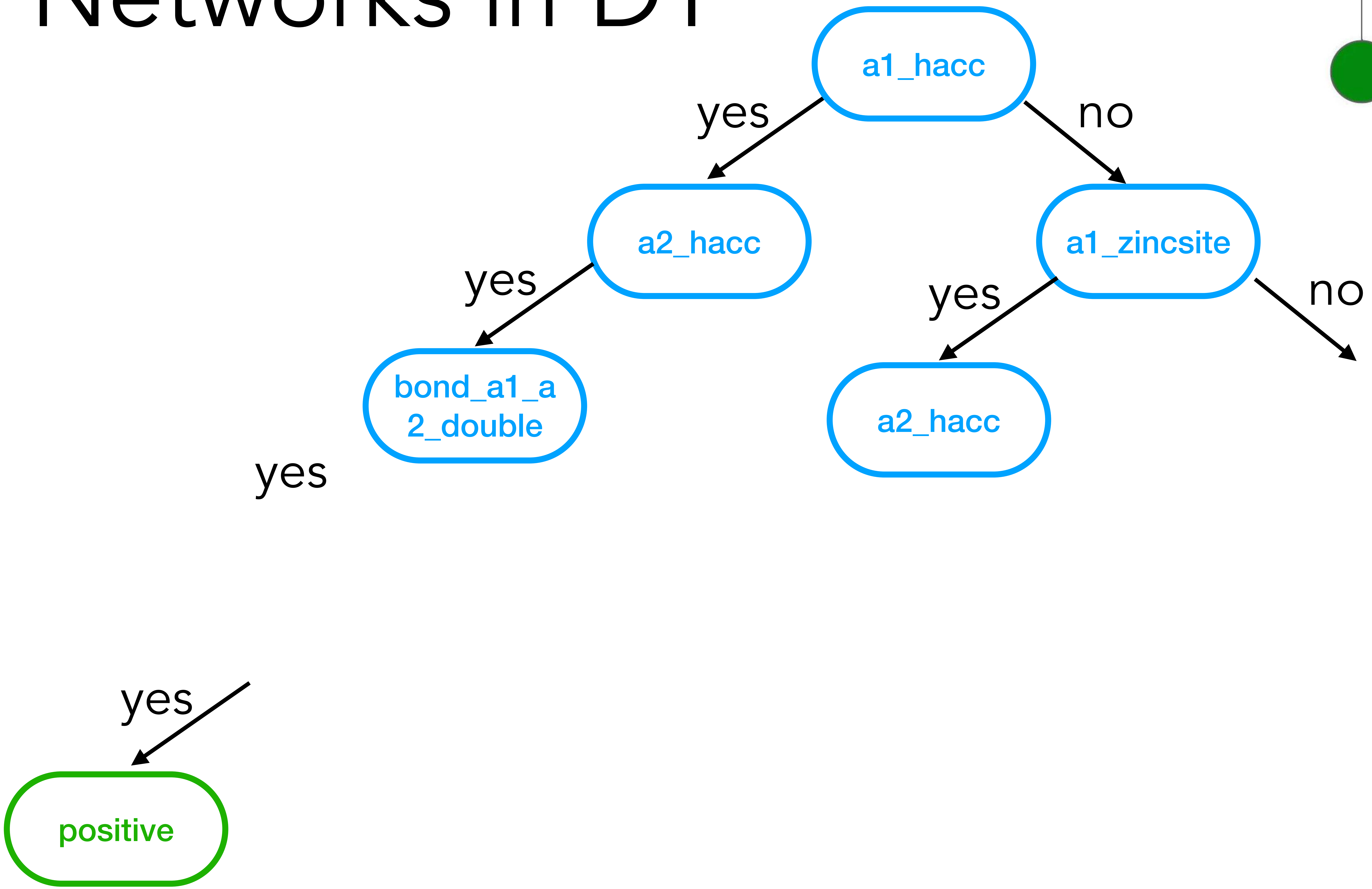



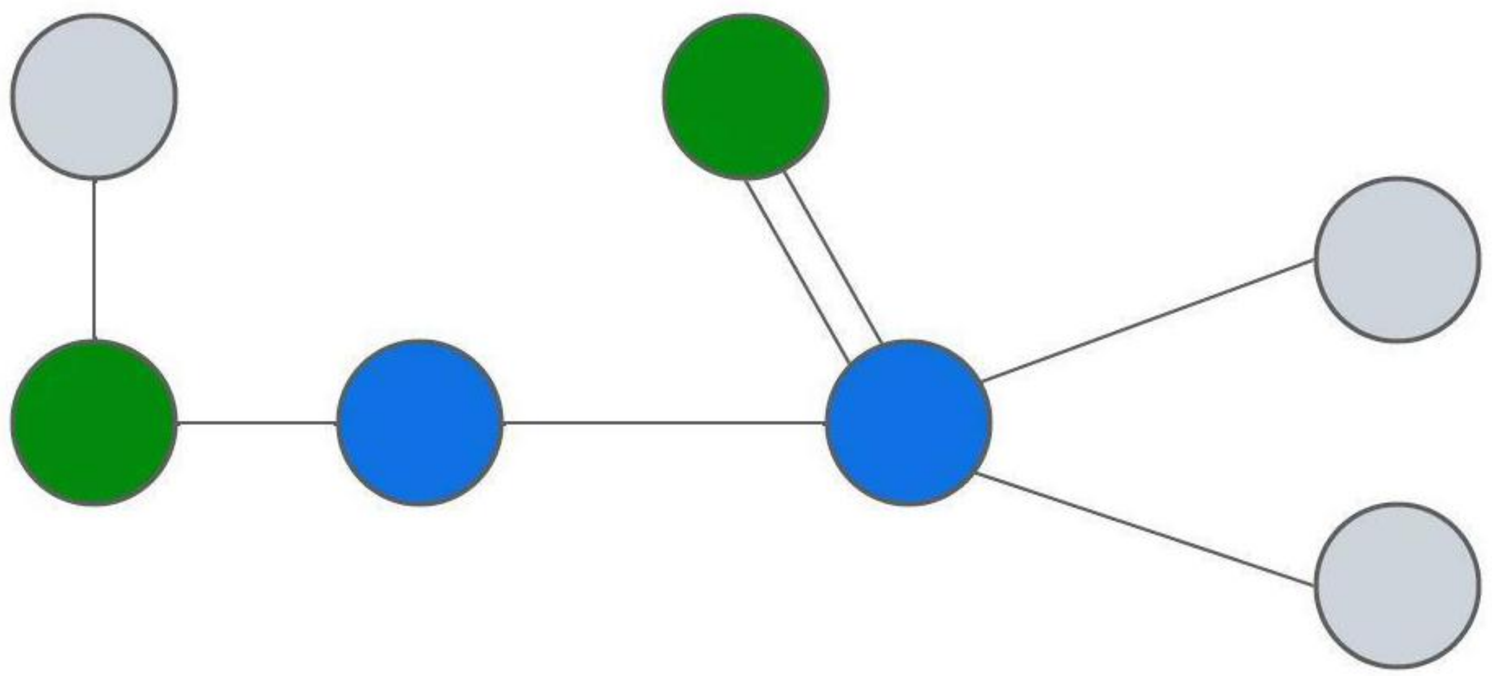
# Networks in DT



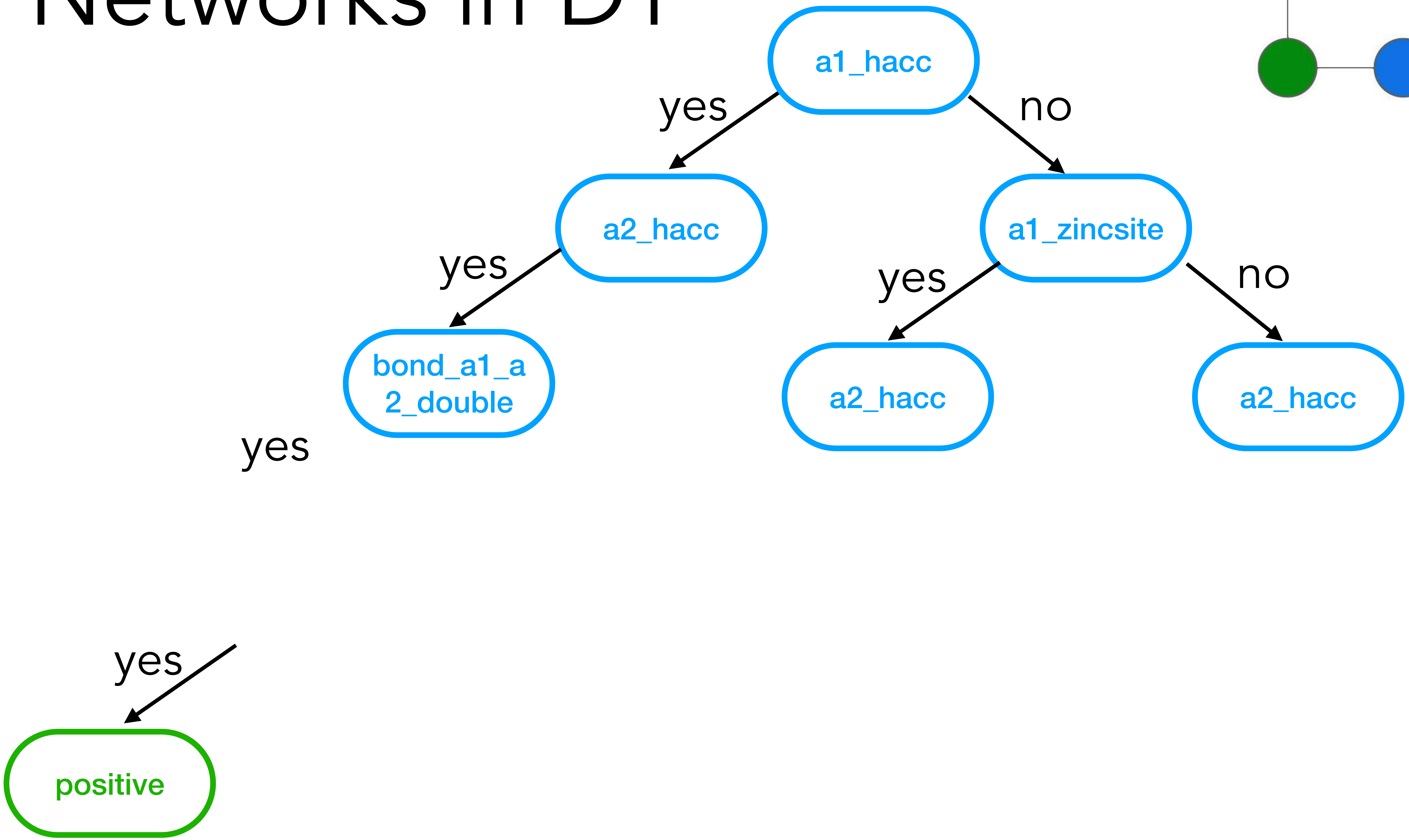



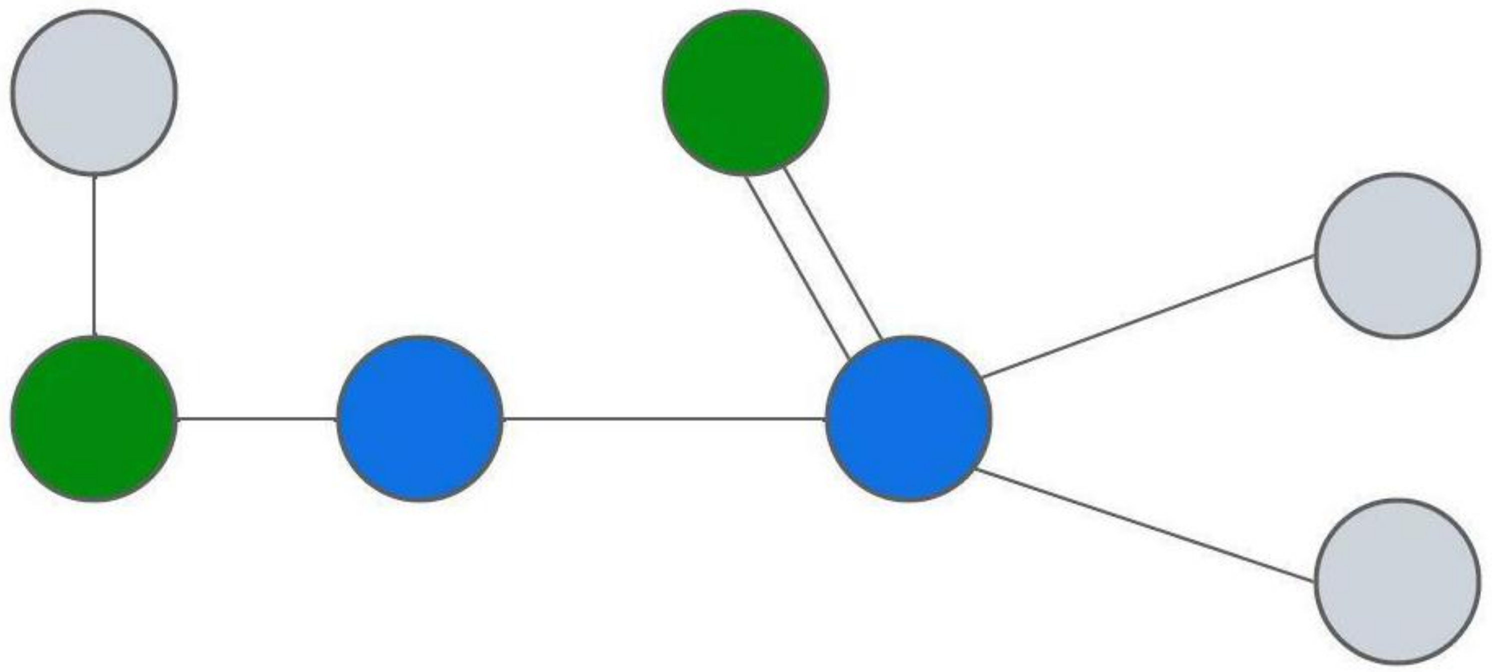
# Networks in DT



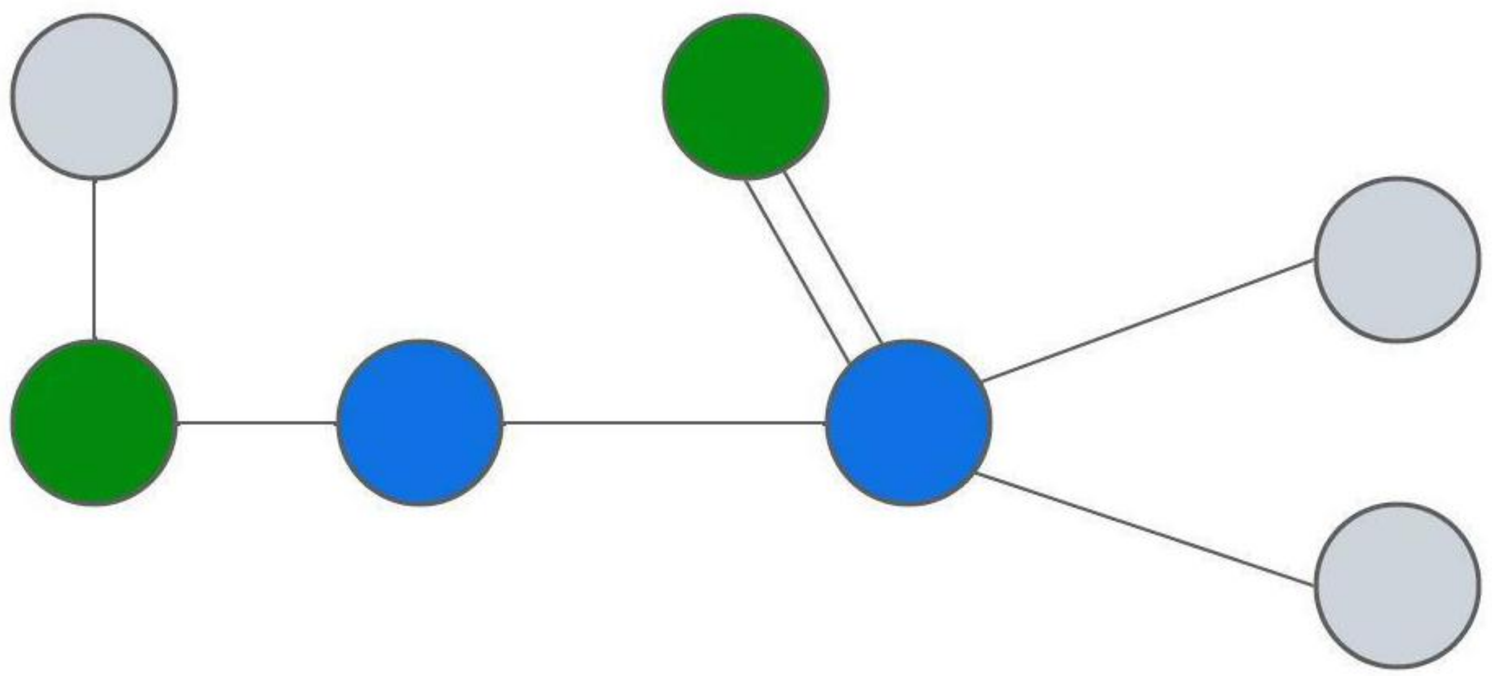



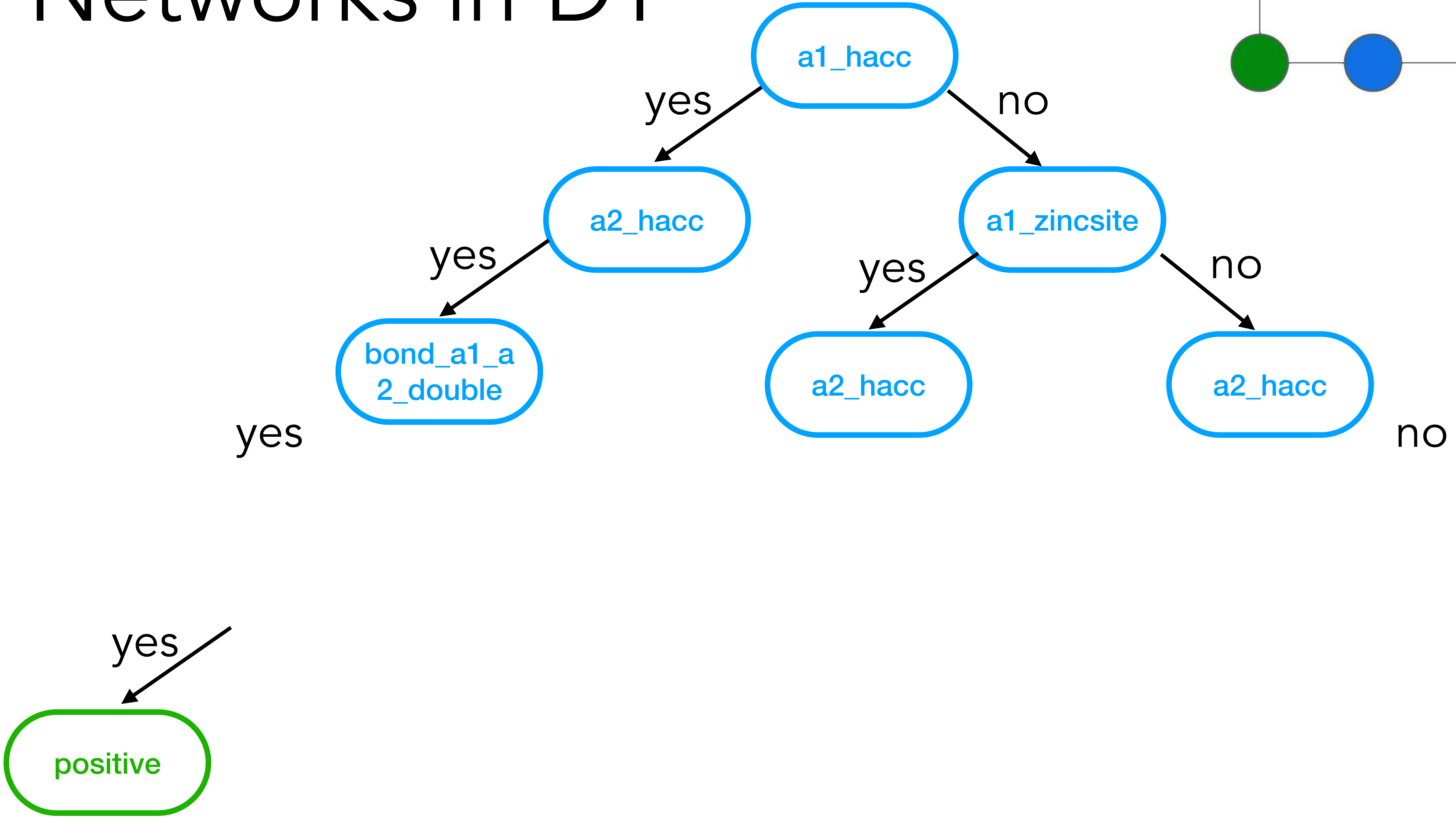
# Networks in DT



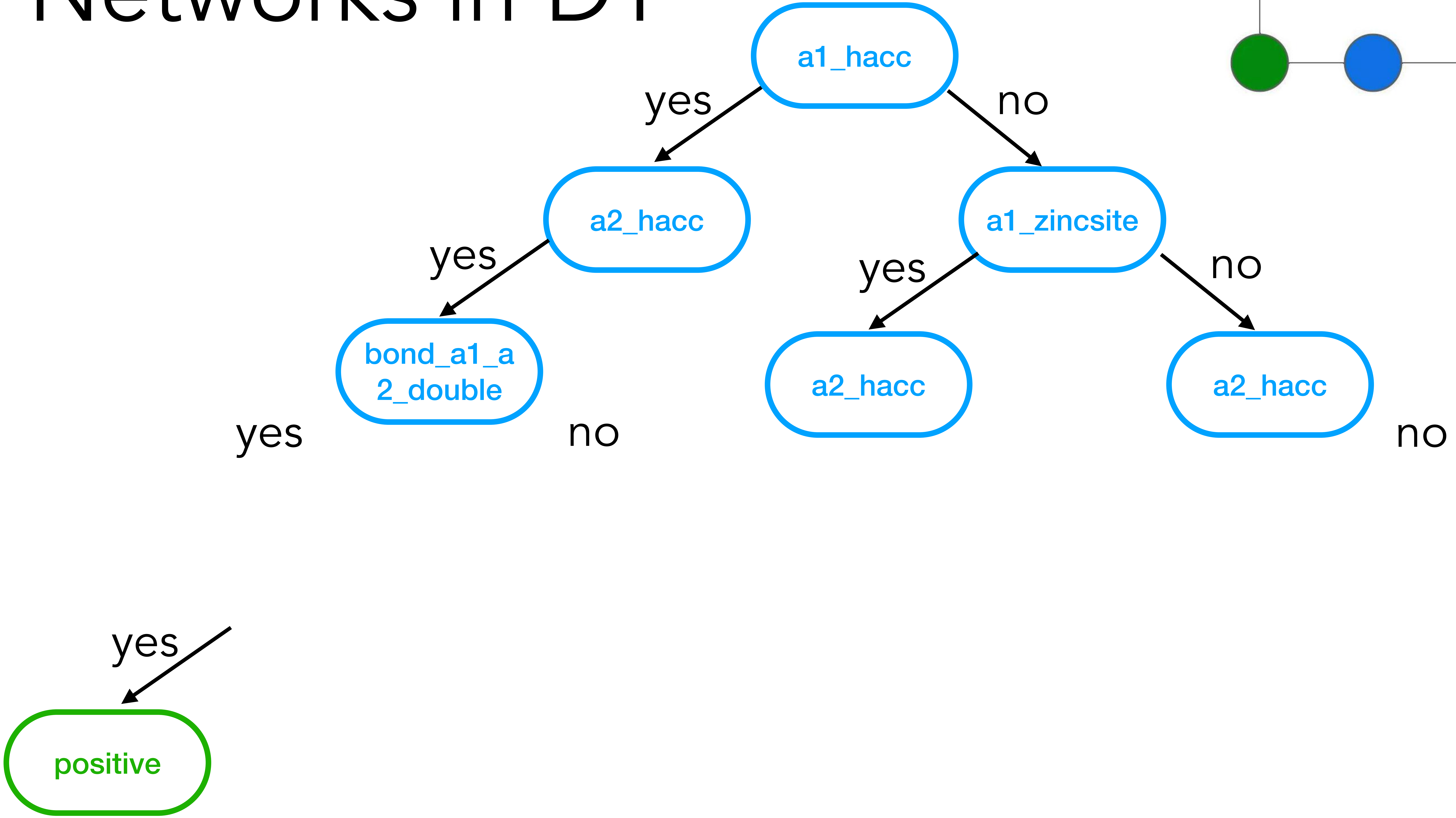



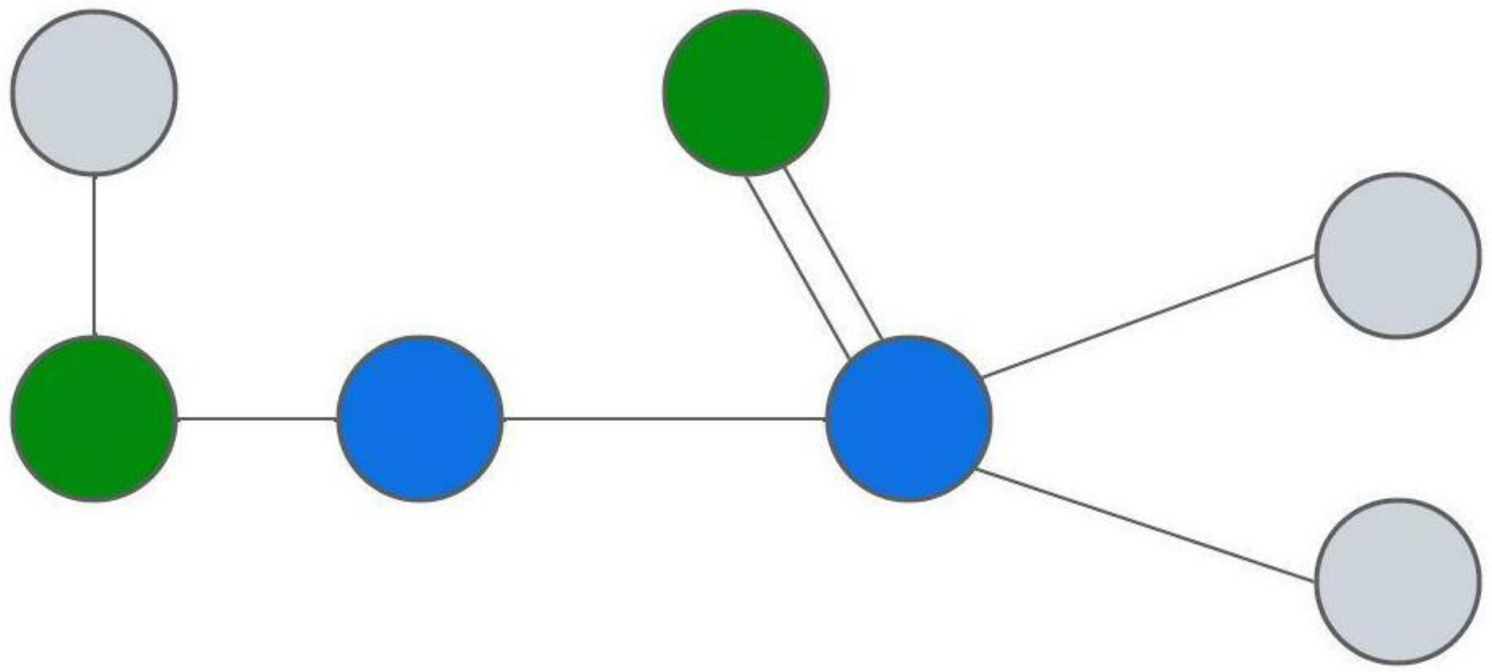
# Networks in DT



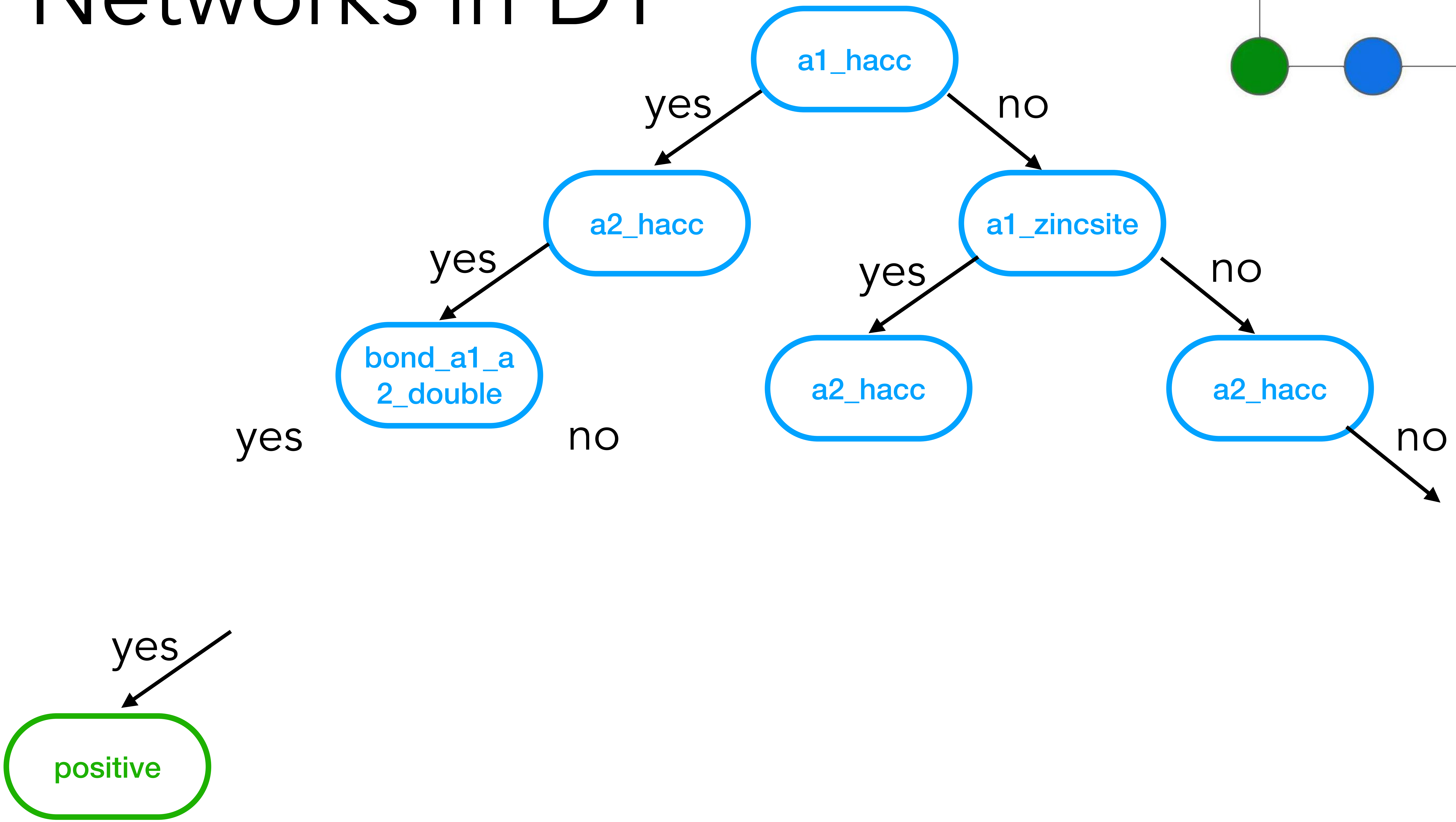


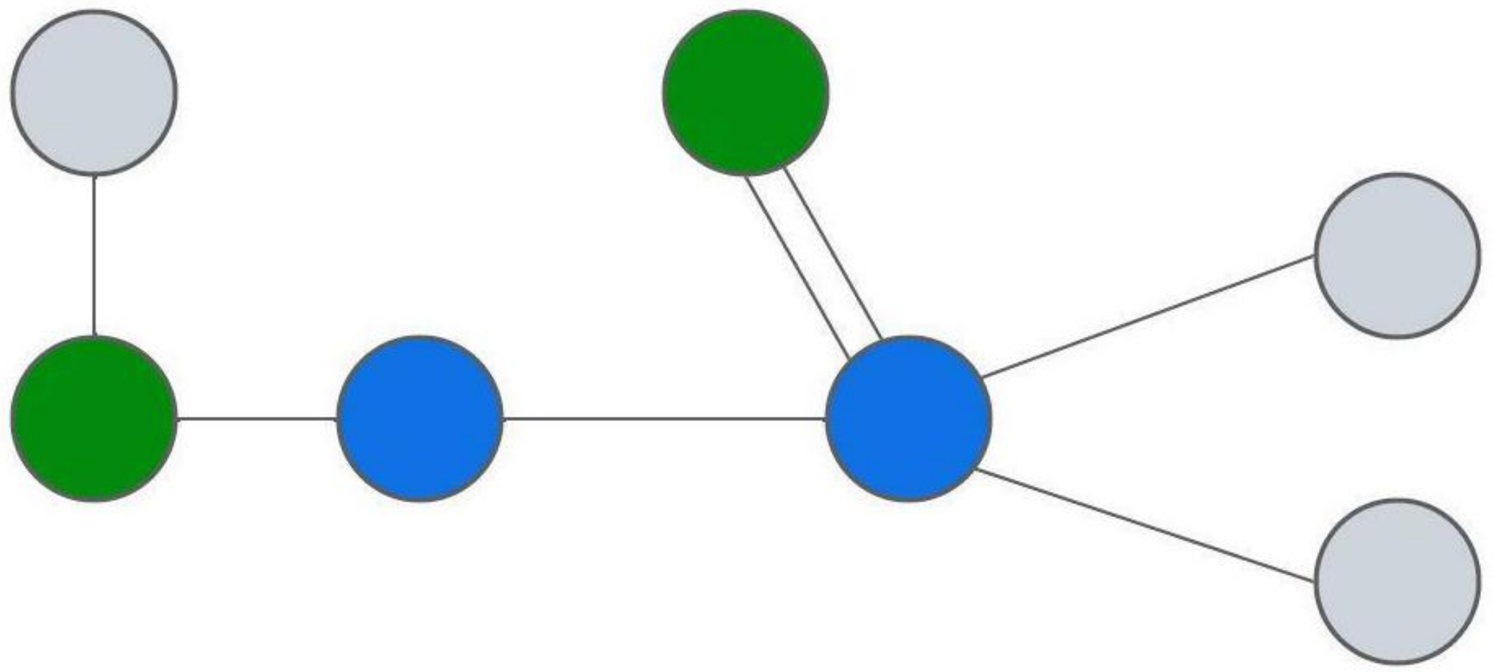
# Networks in DT



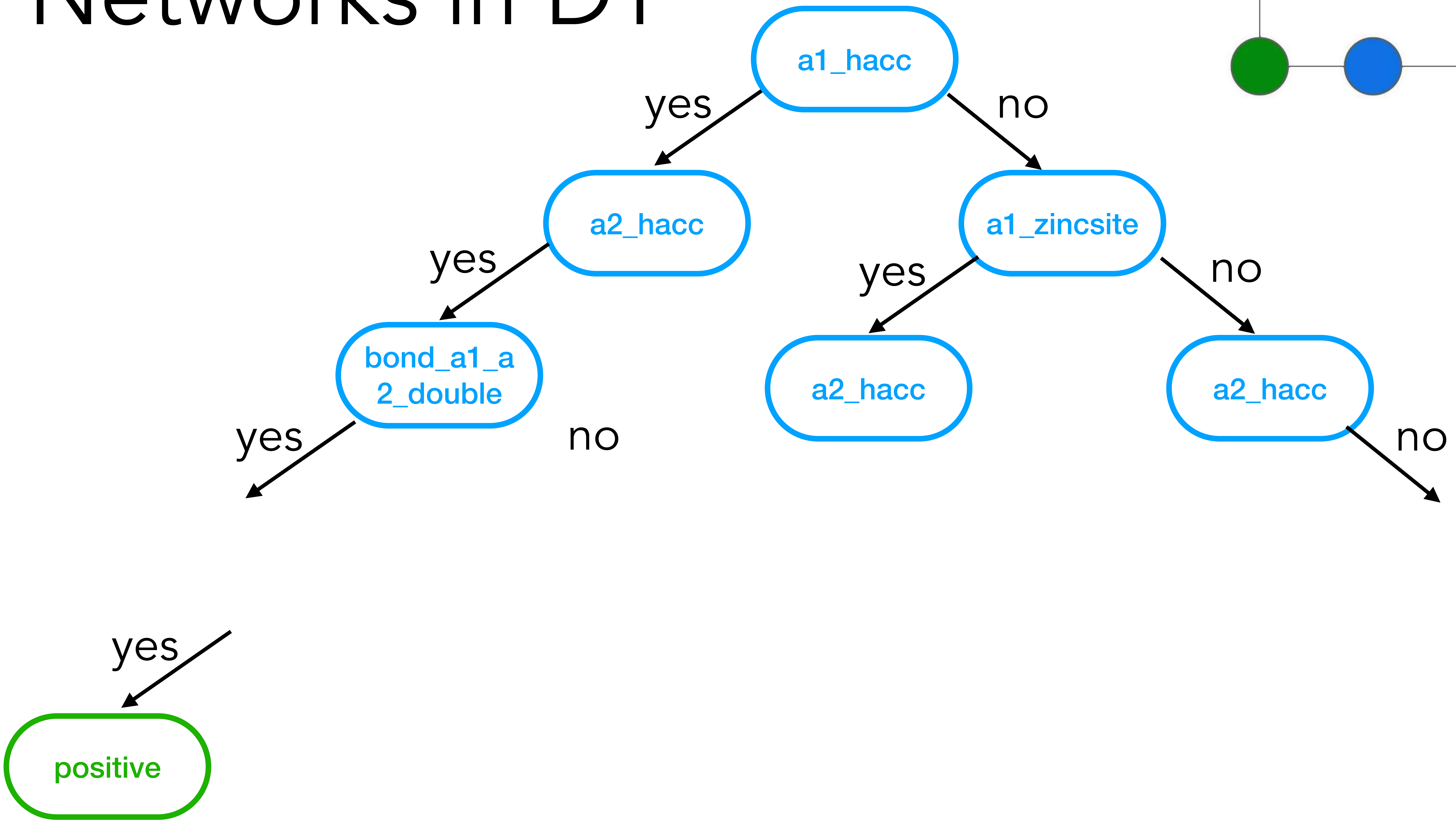



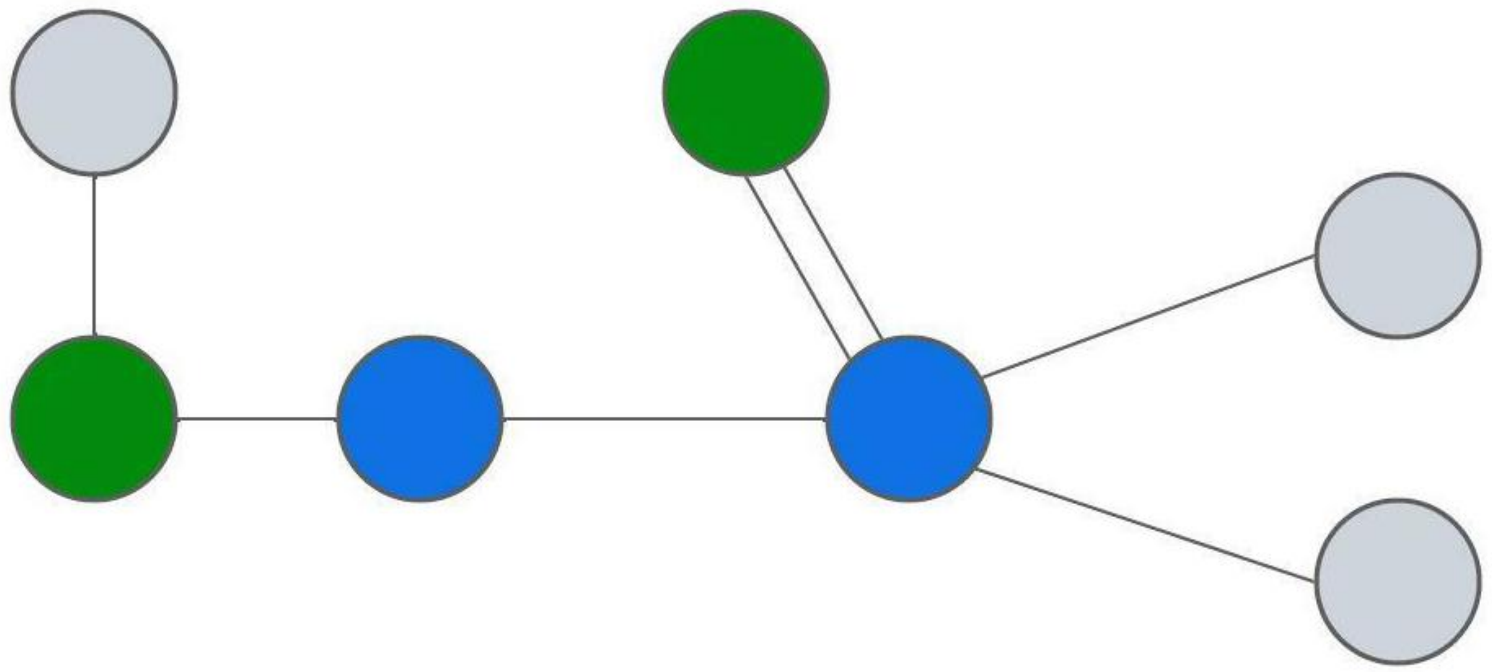
# Networks in DT



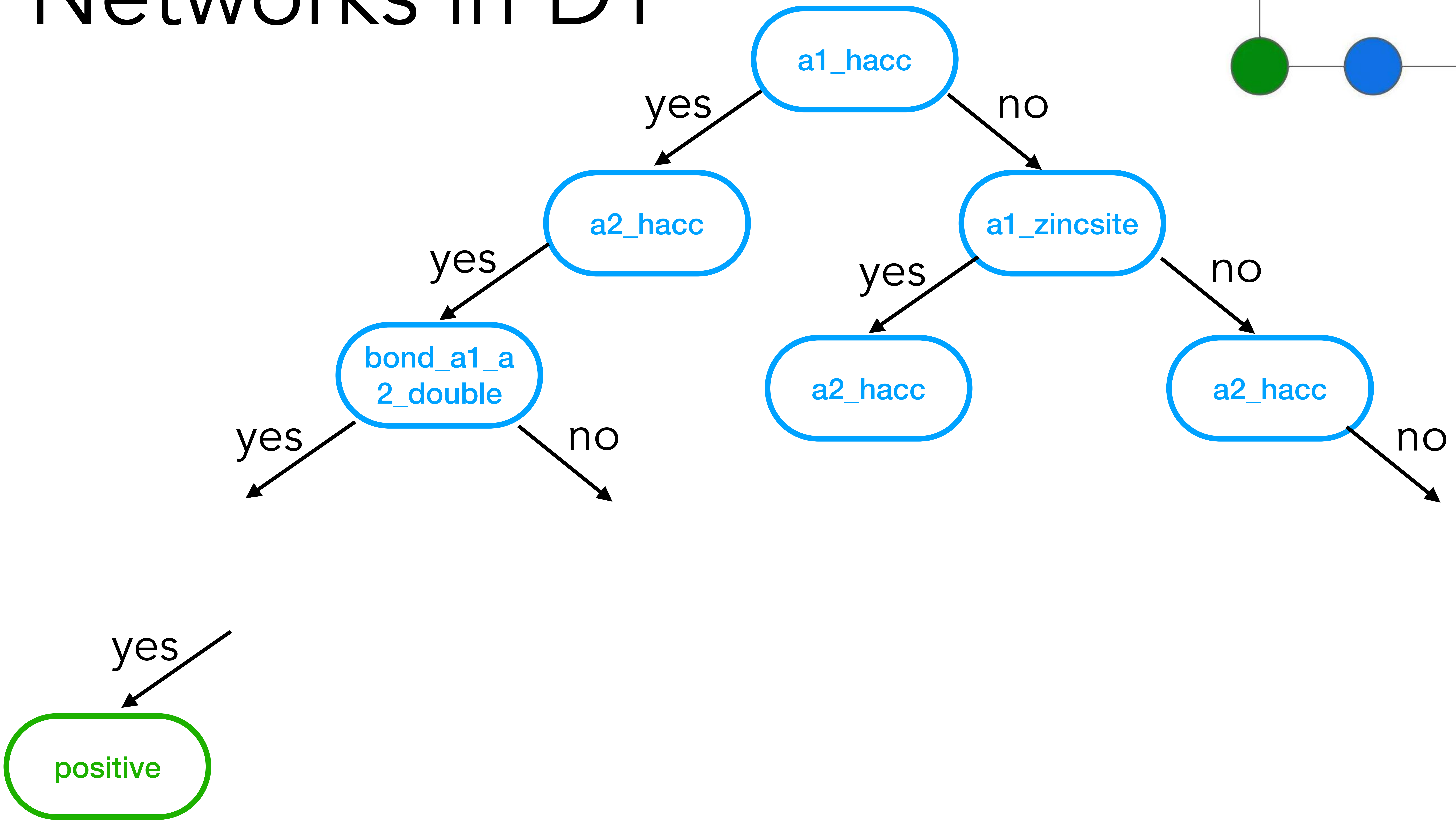



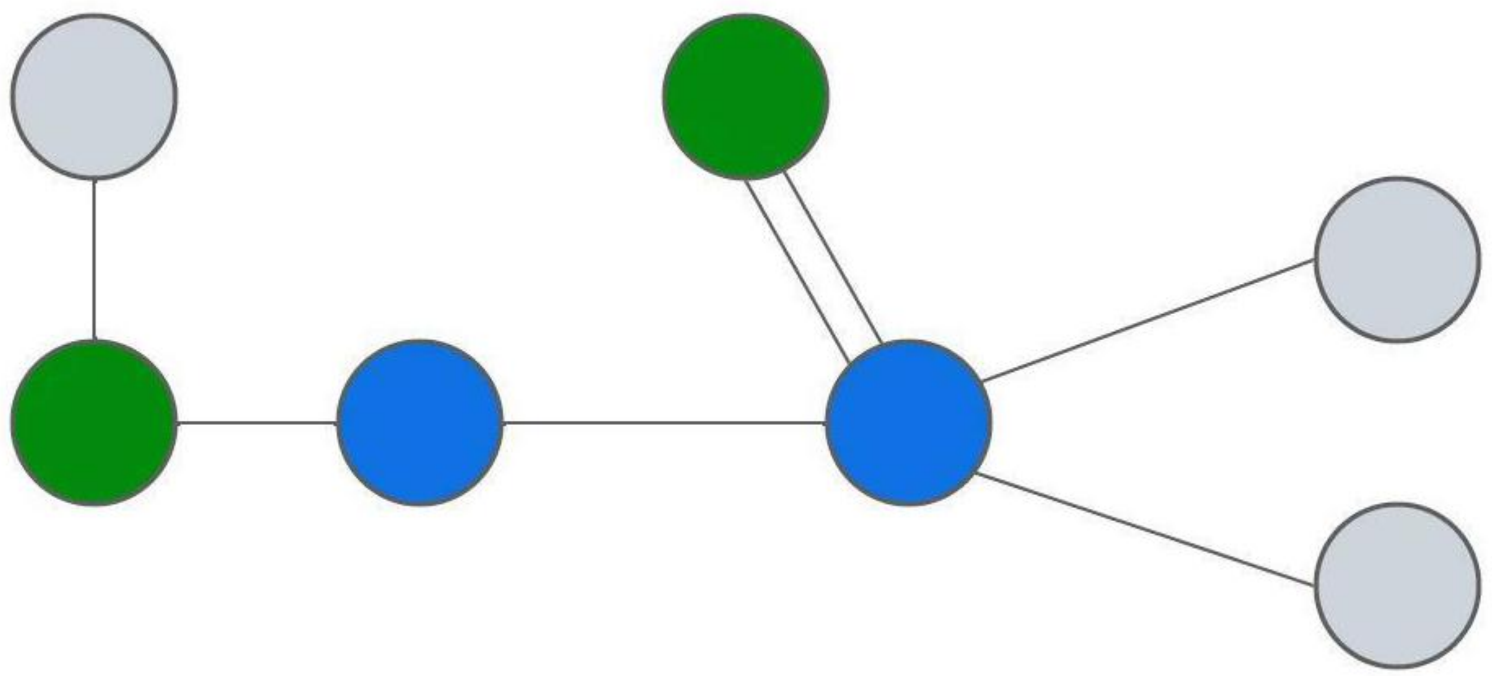
# Networks in DT



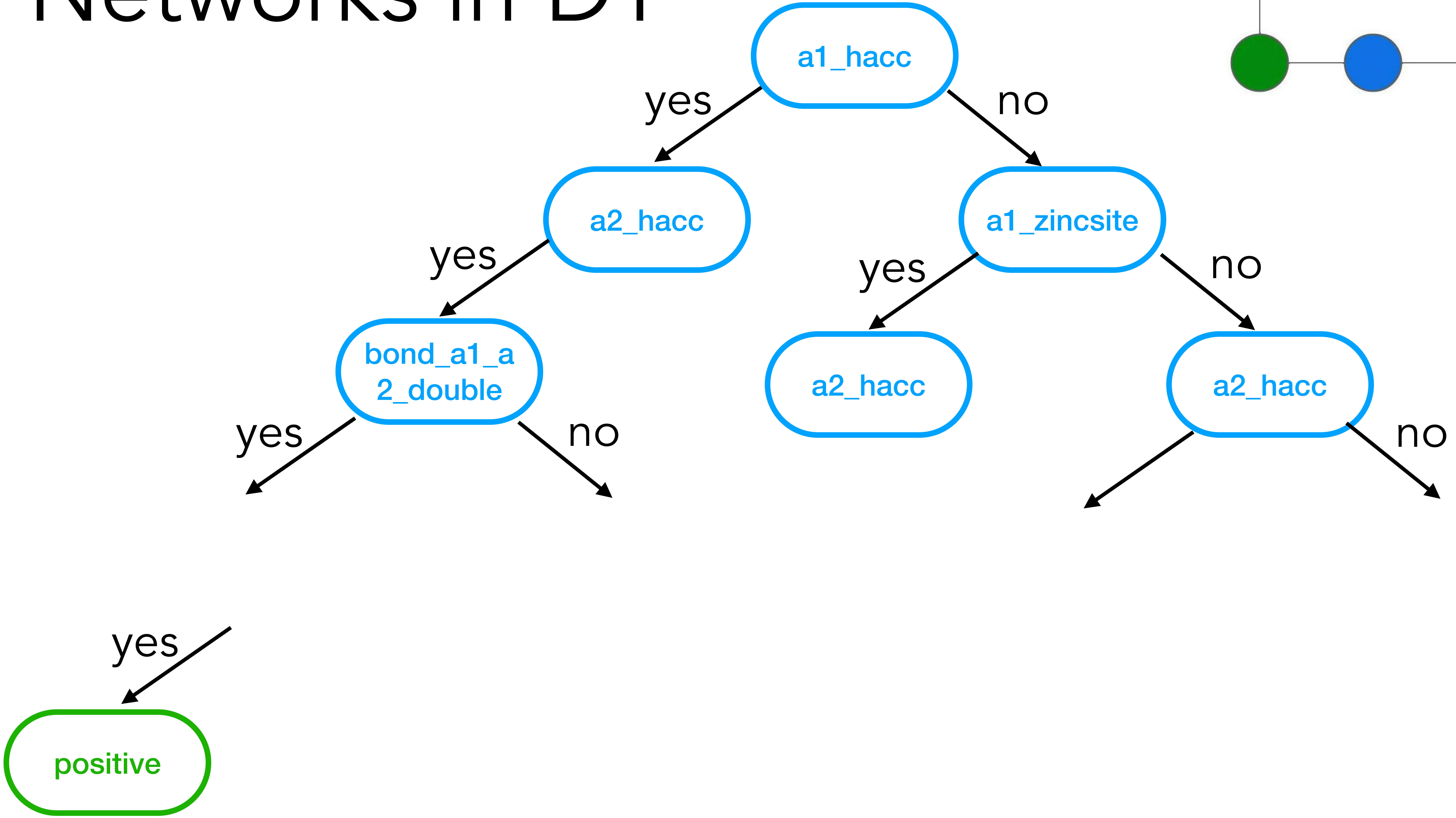



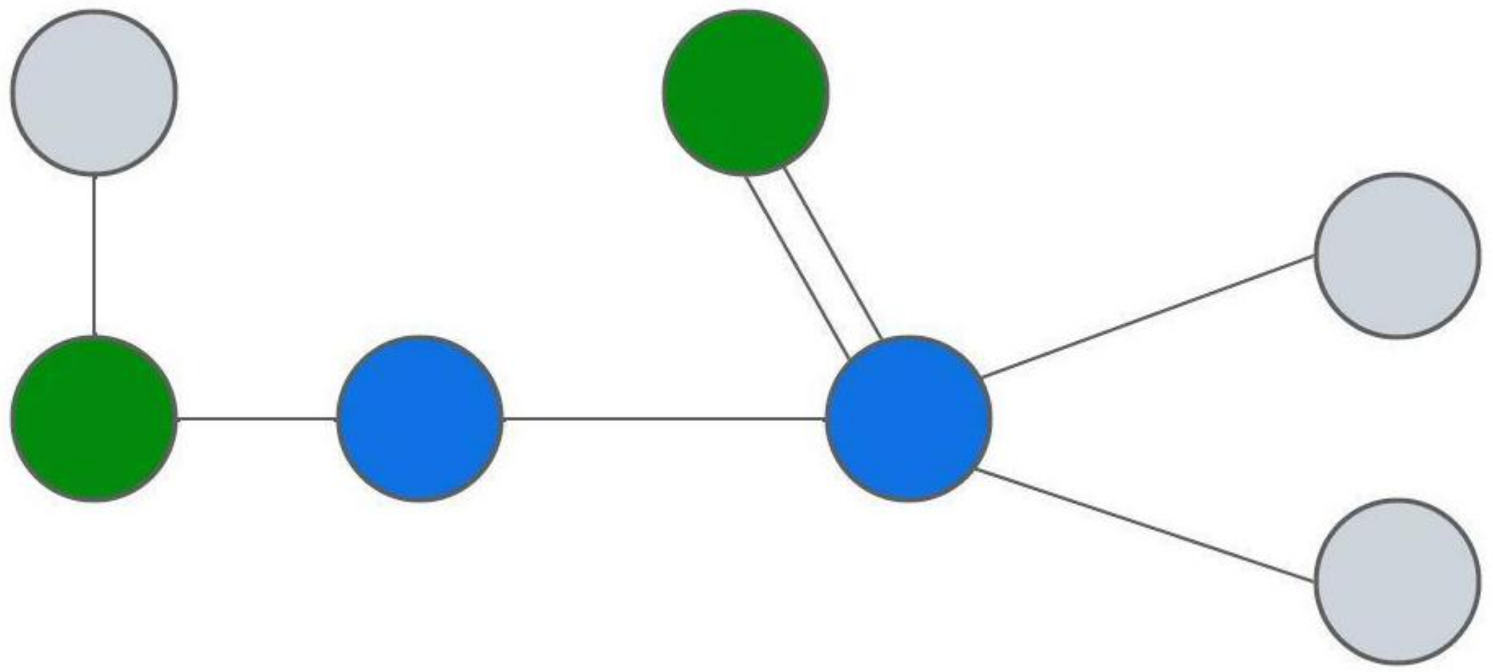
# Networks in DT



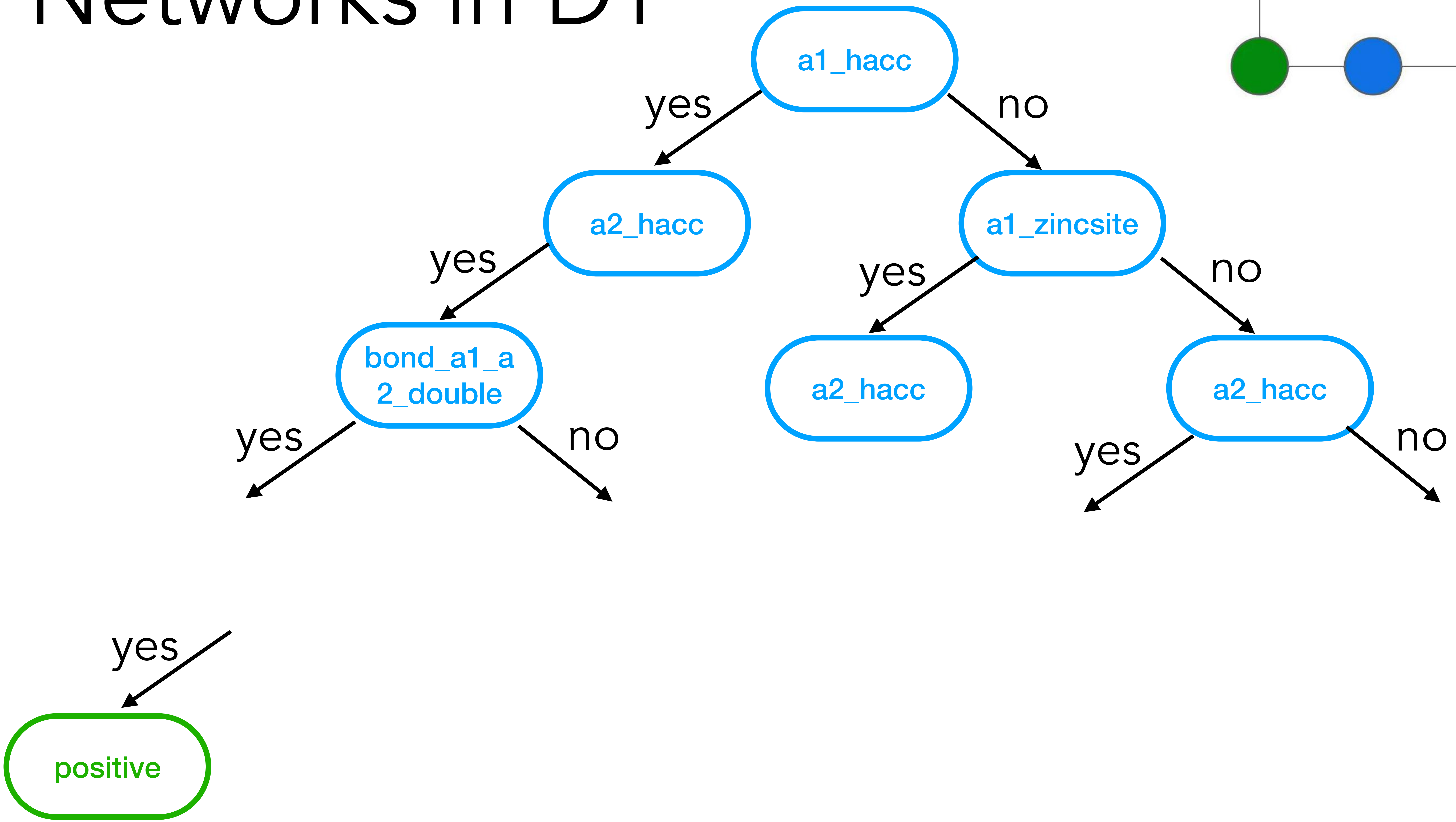



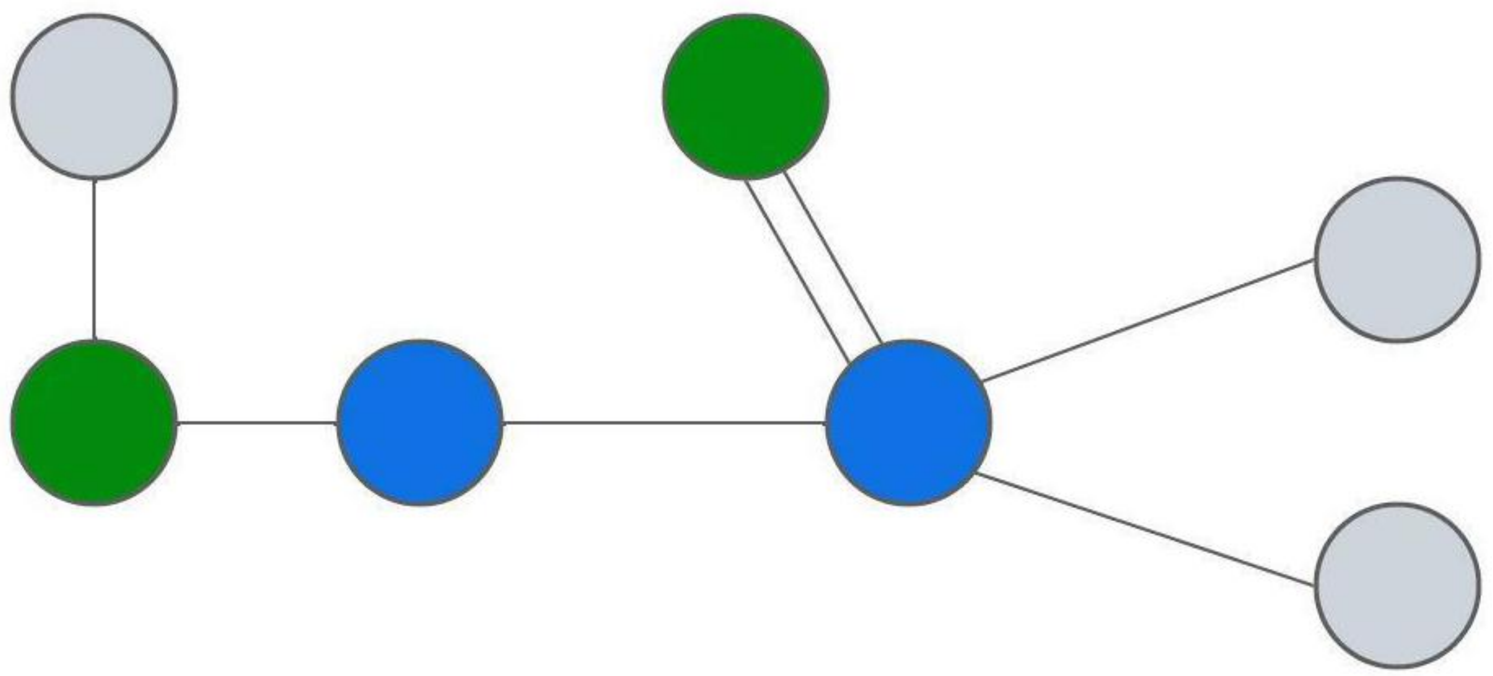
# Networks in DT



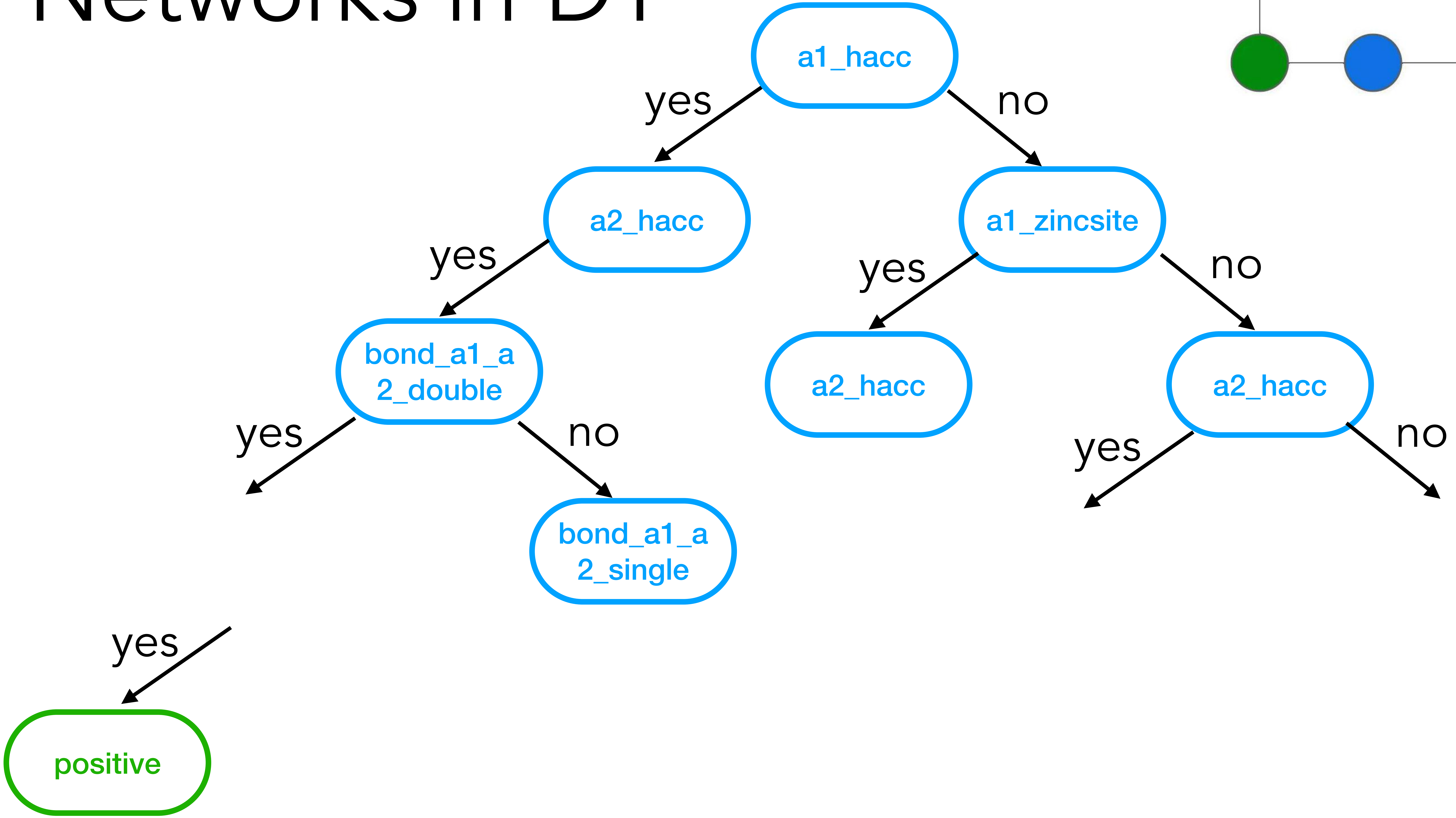



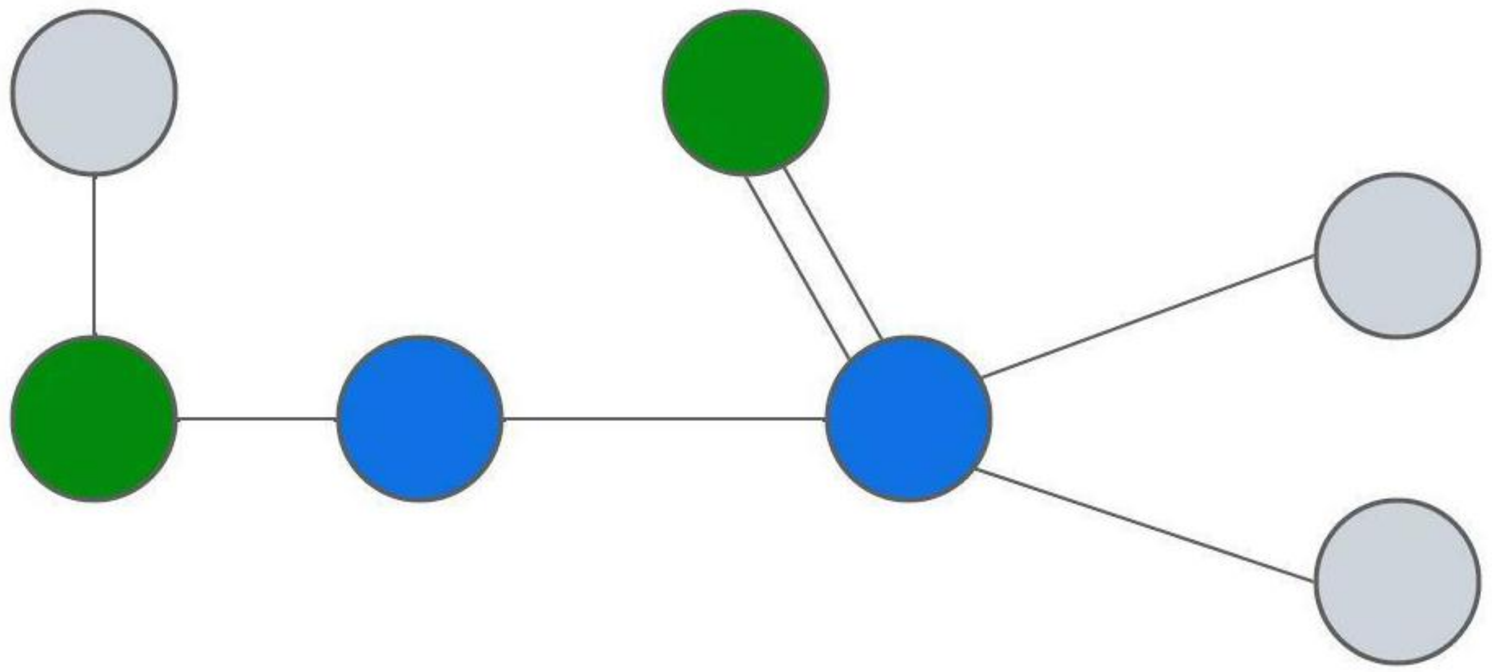
# Networks in DT



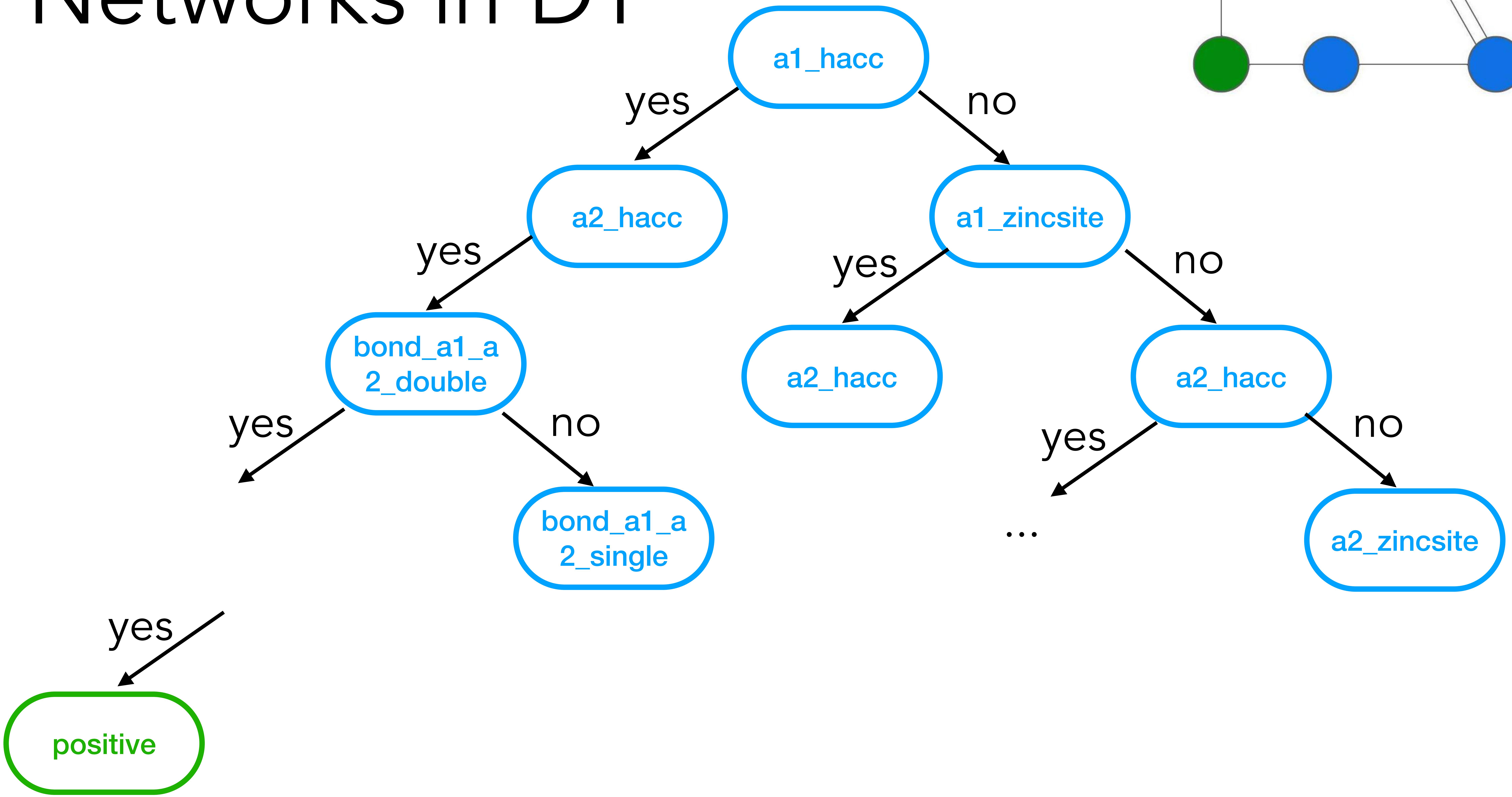
# Networks in DT

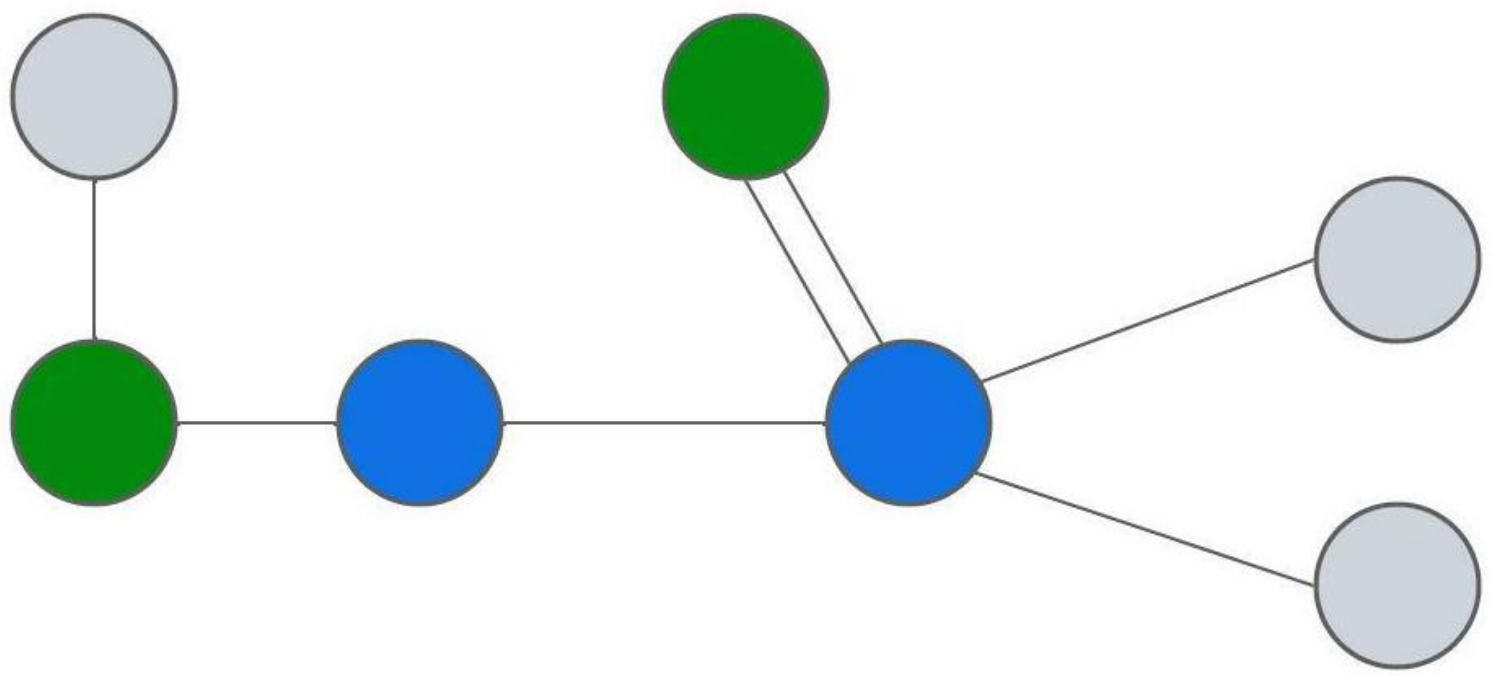


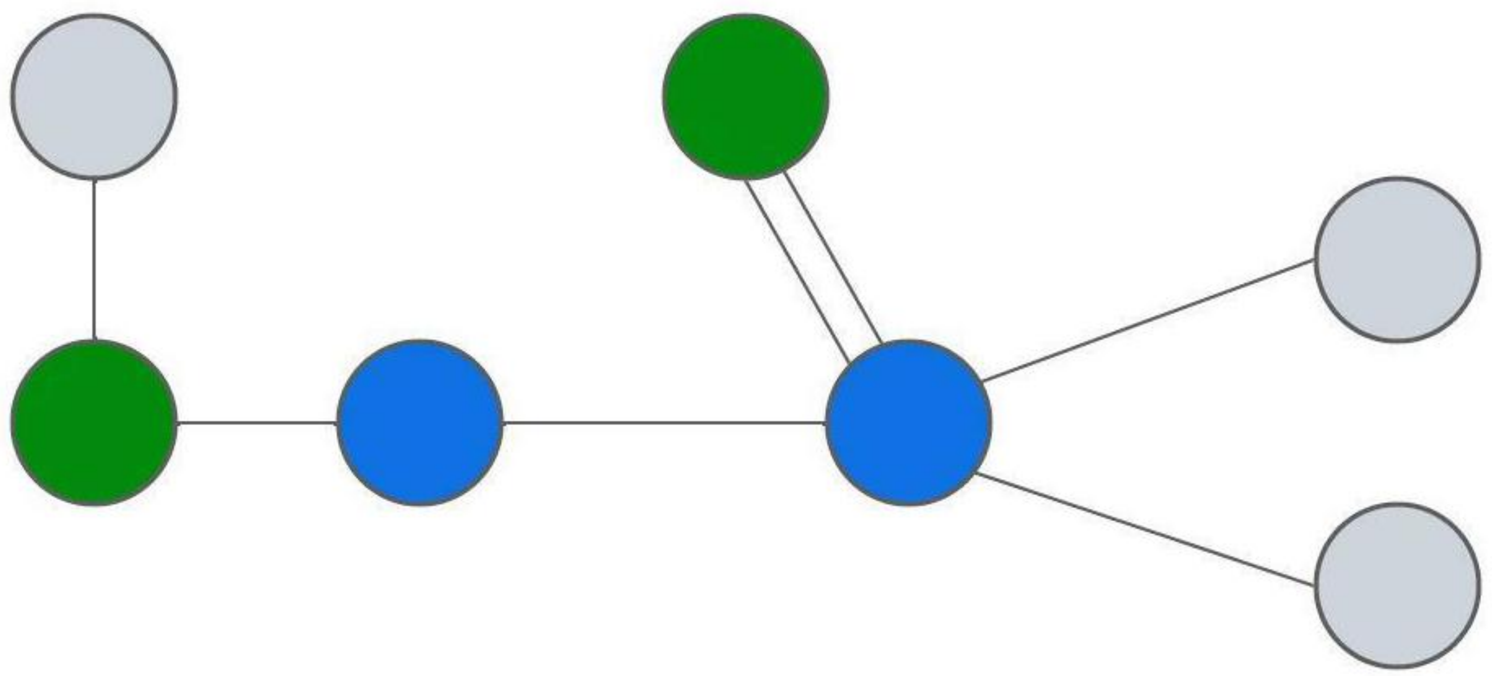
# Networks in DT

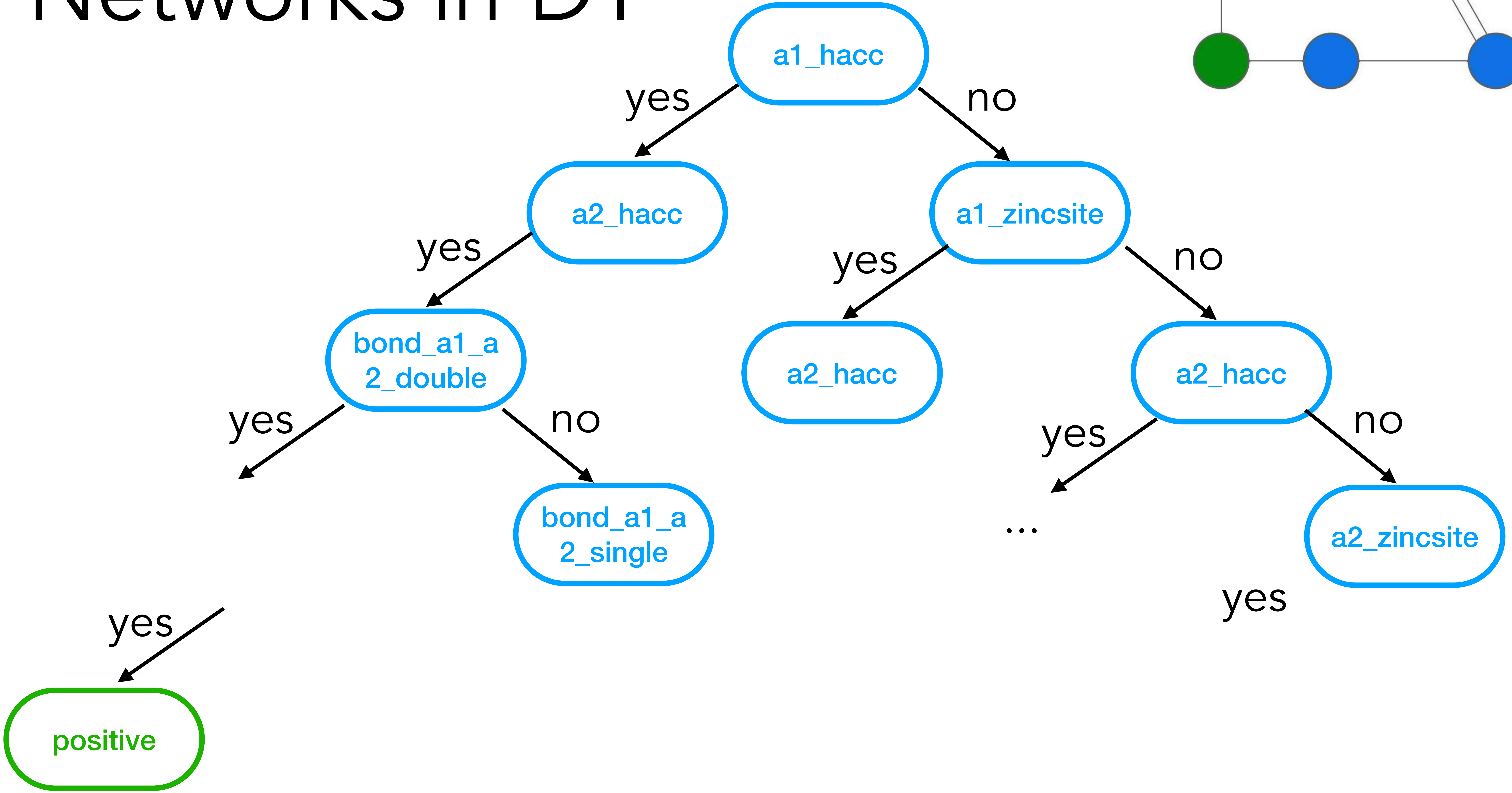
# Networks in DT



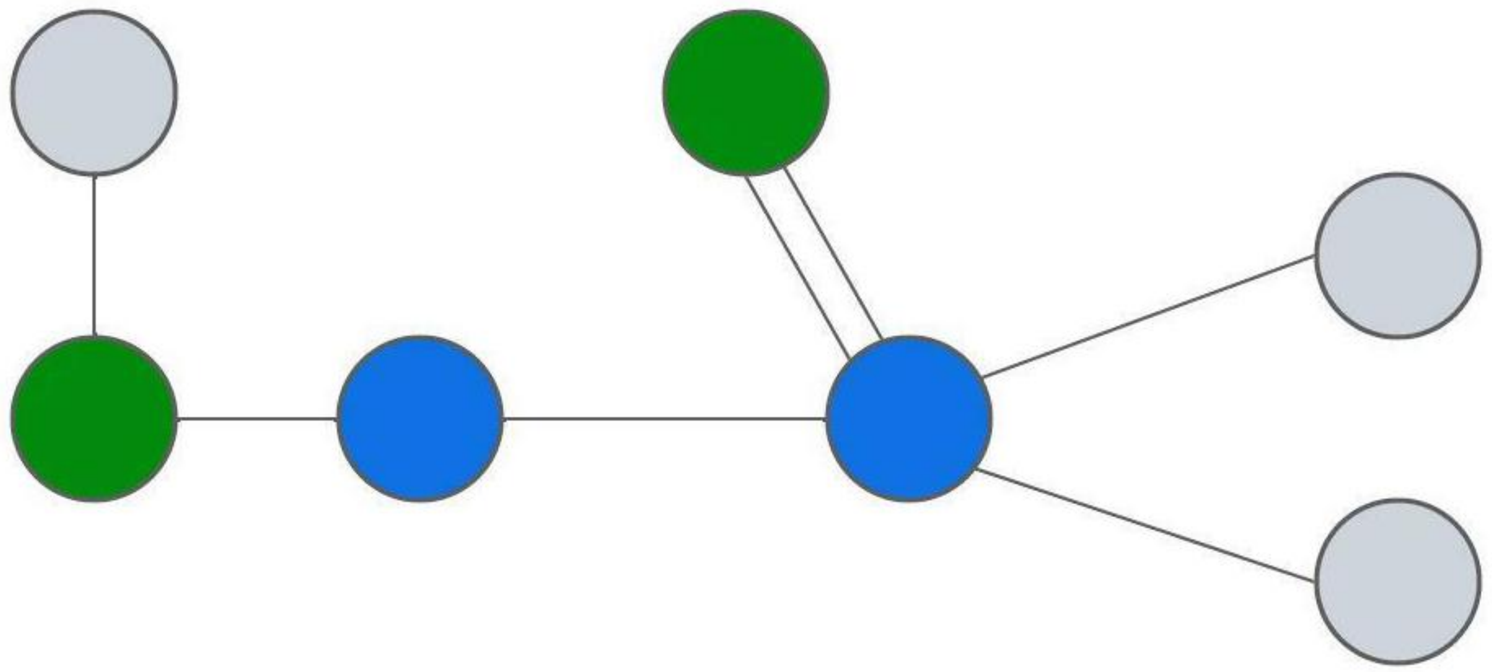
# Networks in DT

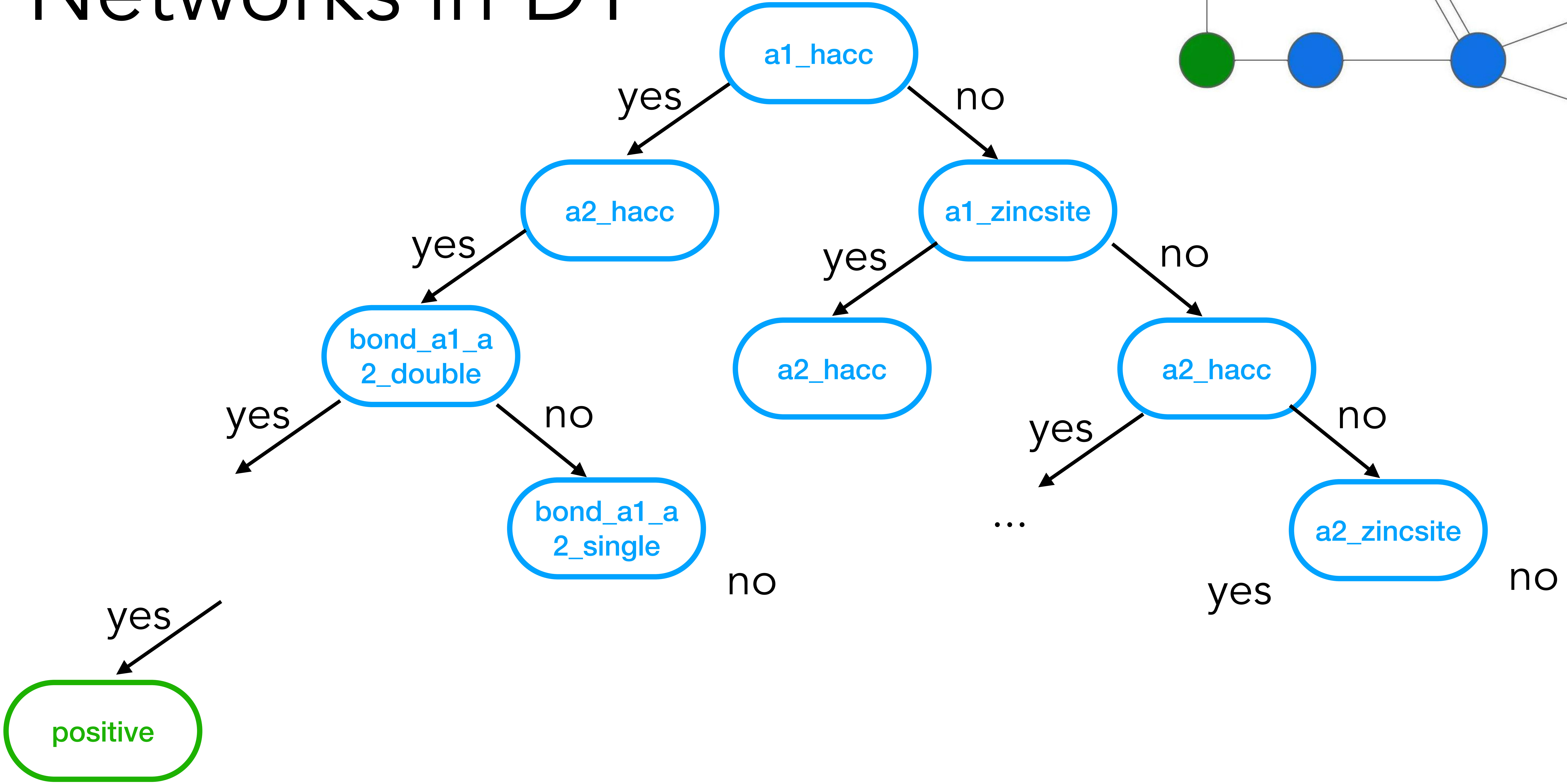




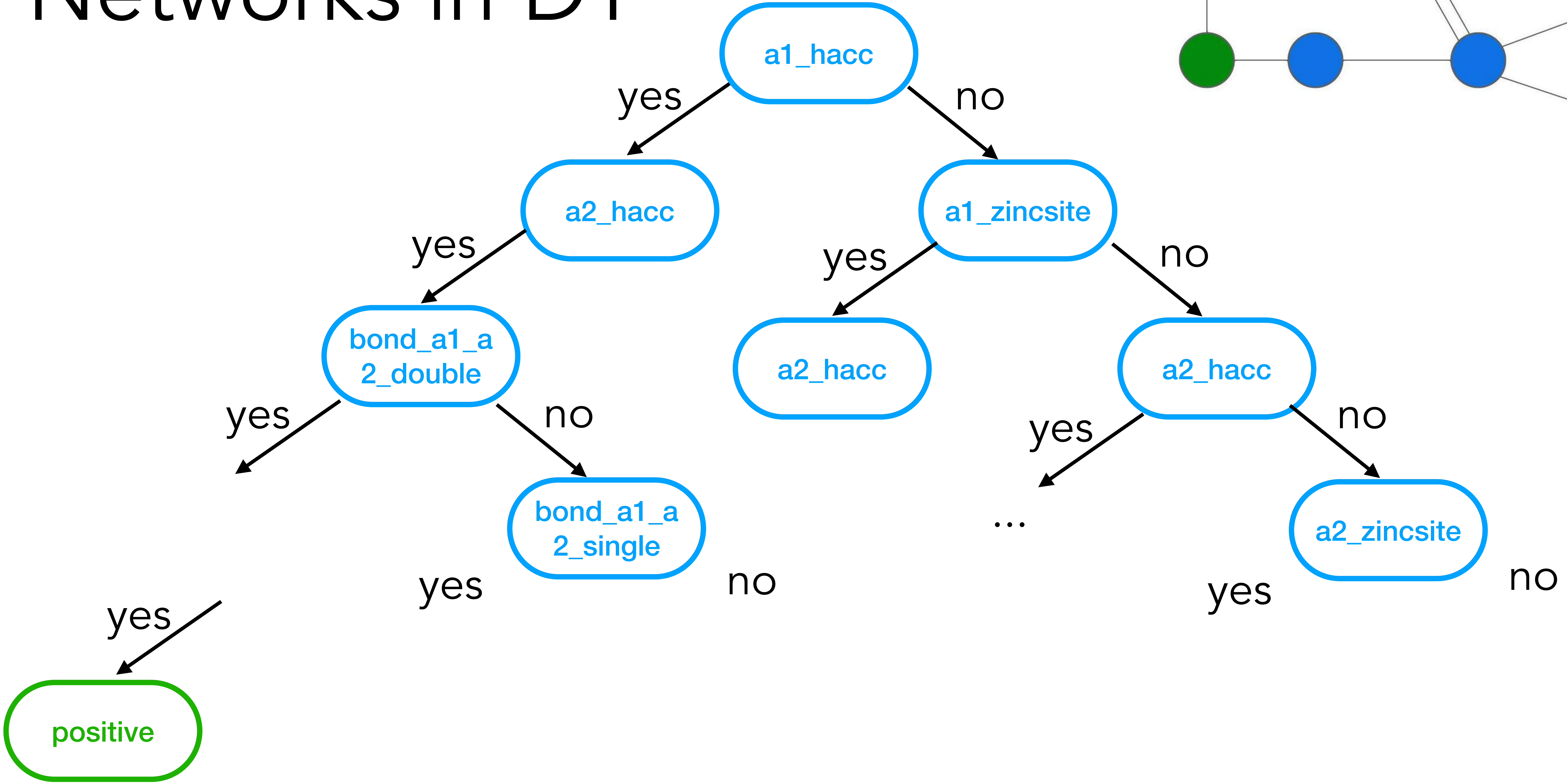
# Networks in DT

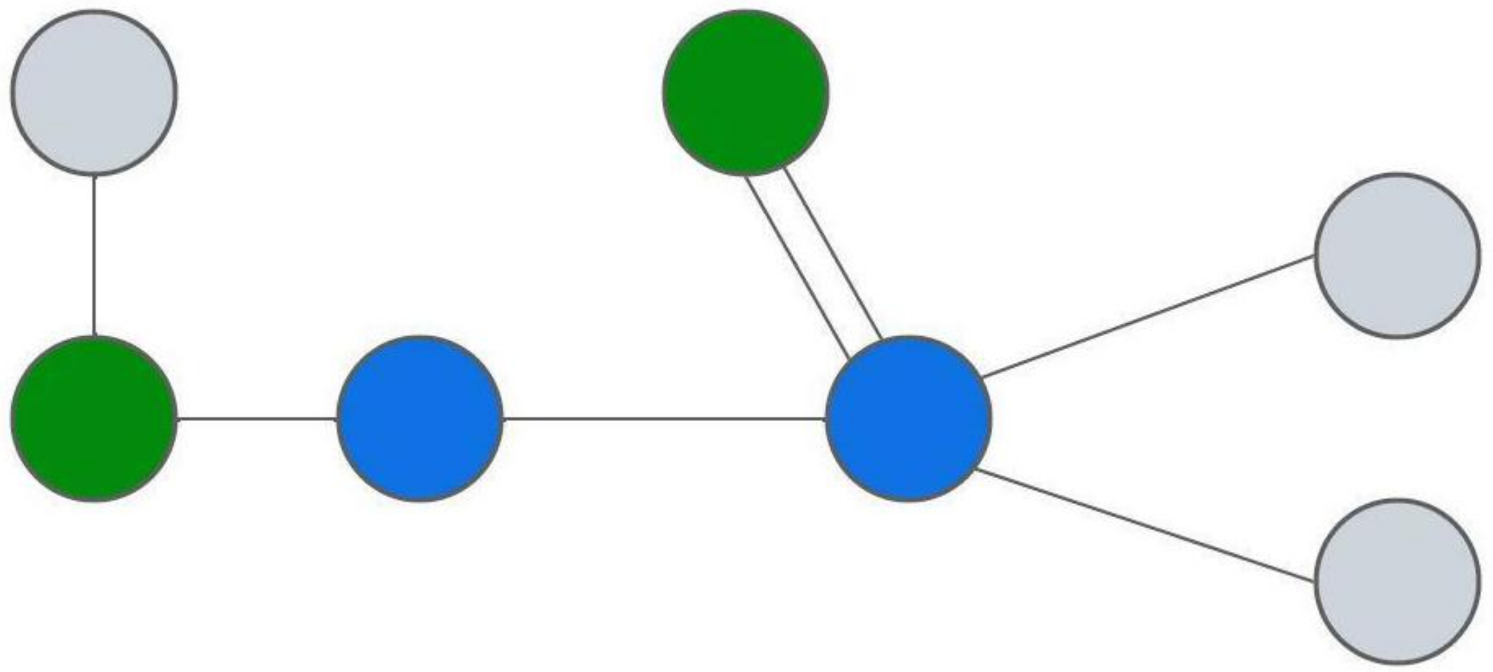
# Networks in DT





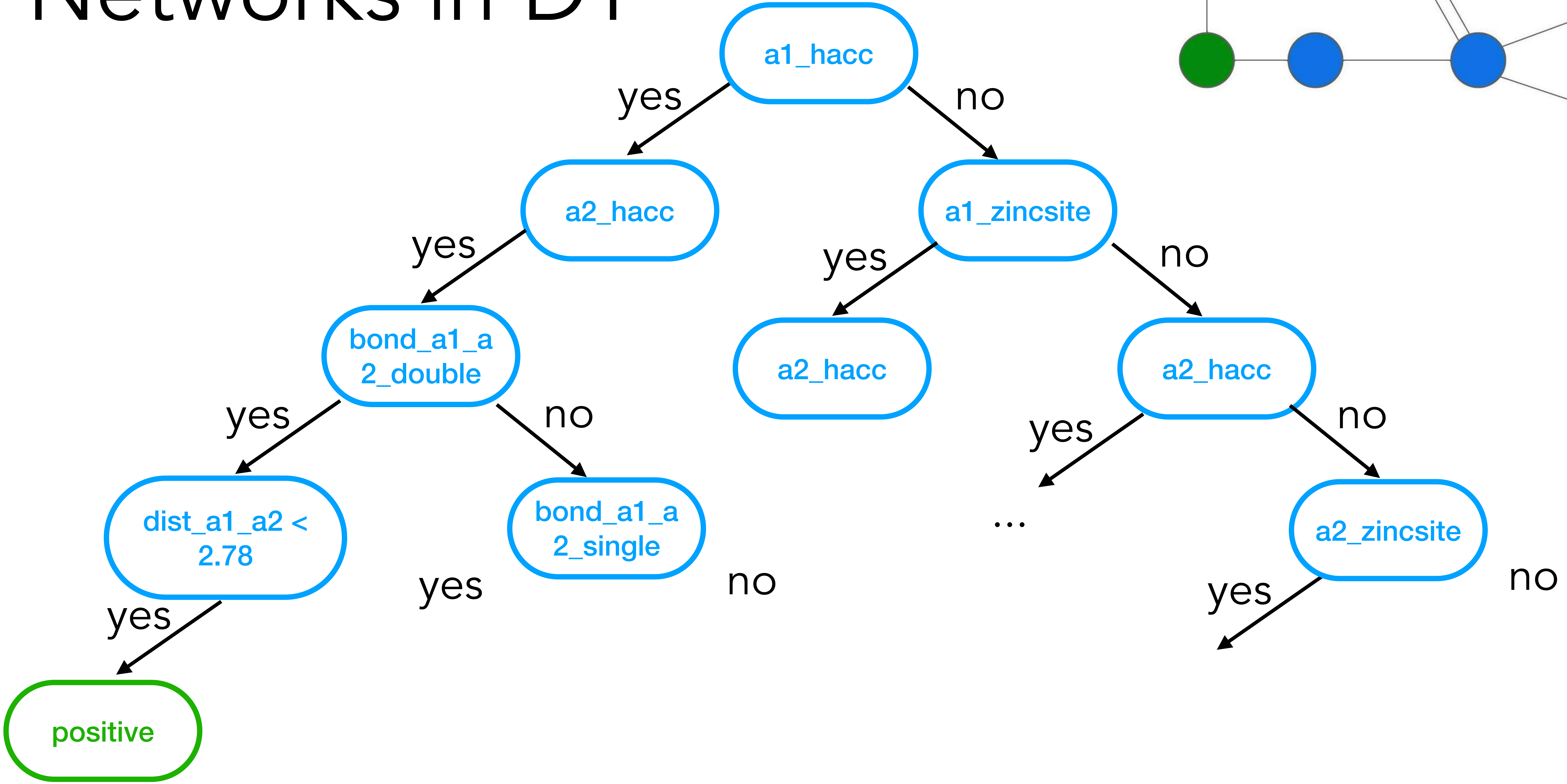
# Networks in DT

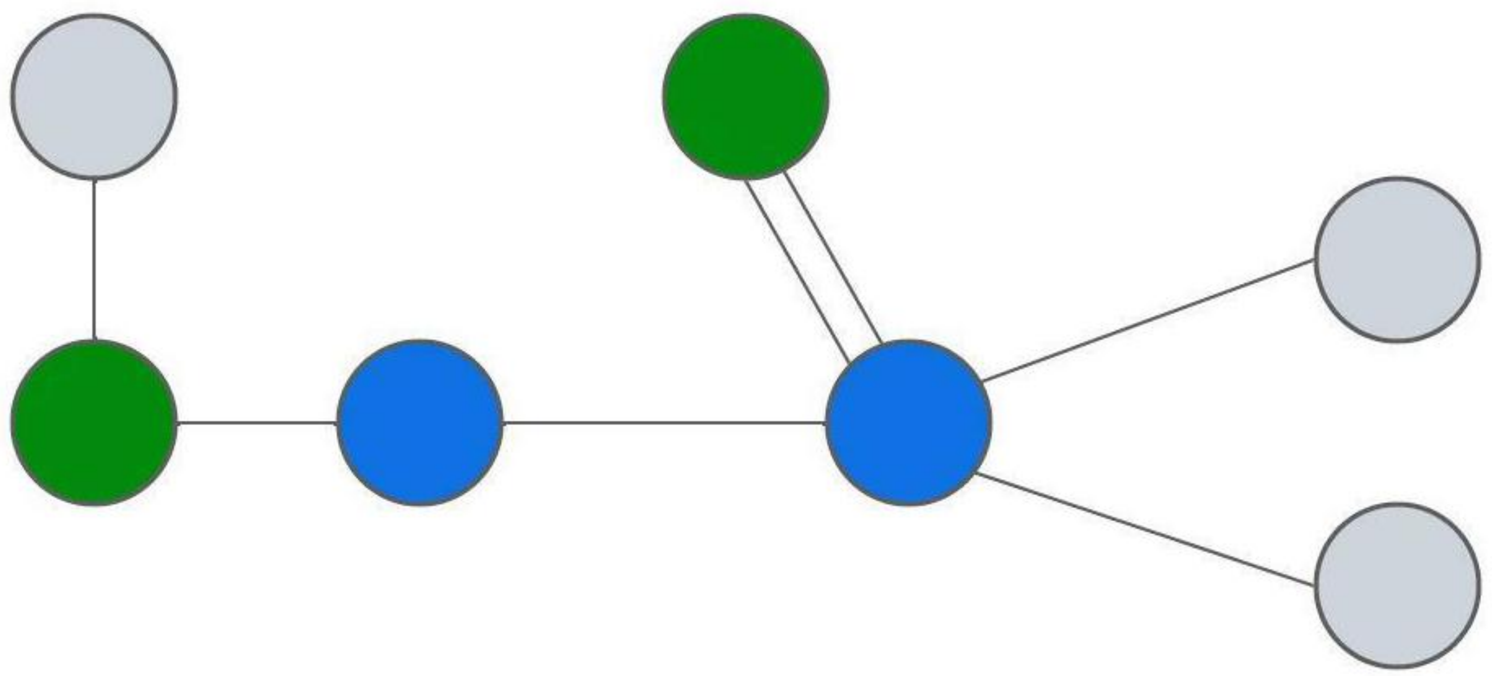


# Networks in DT

# Networks in DT



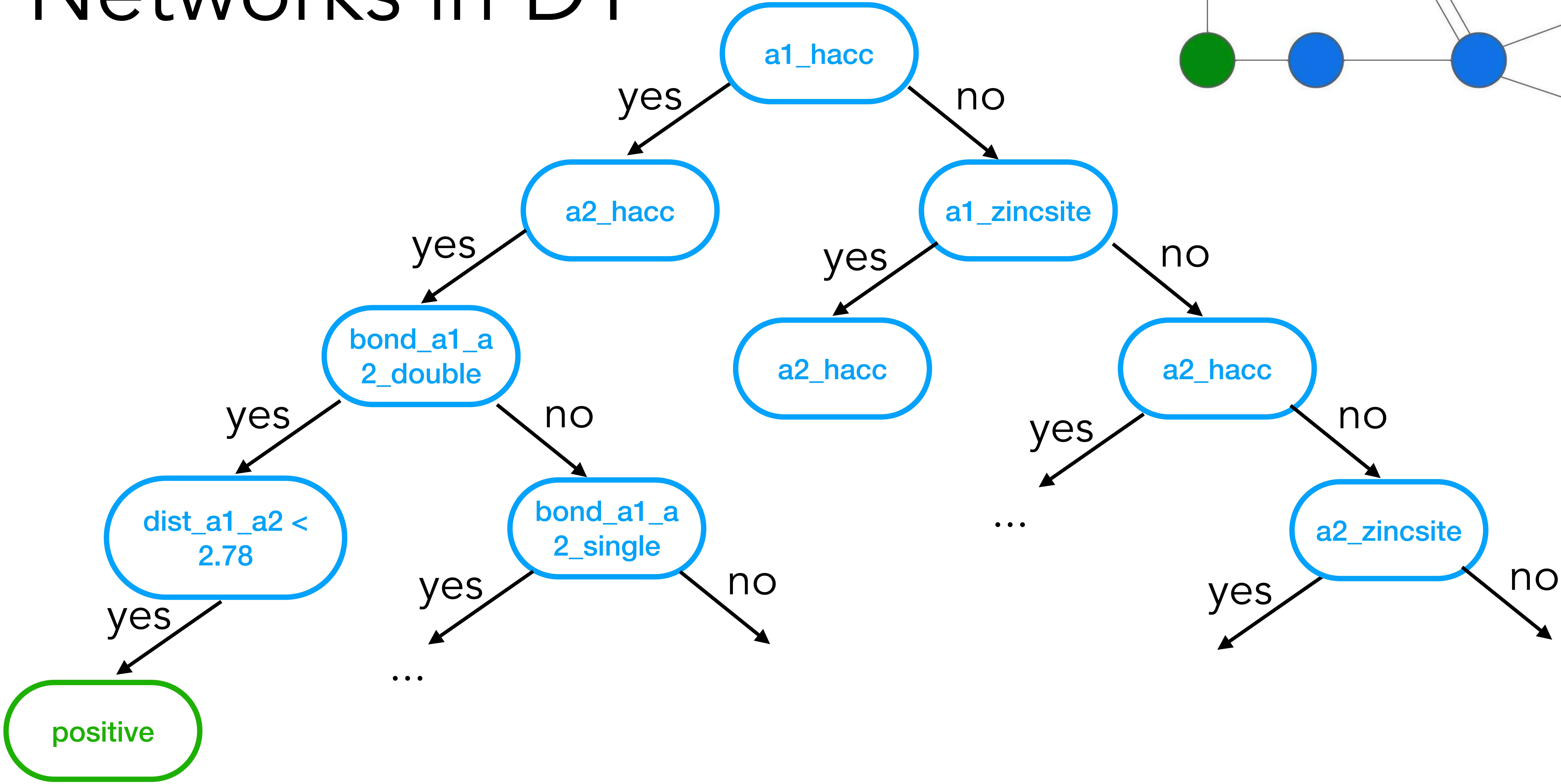



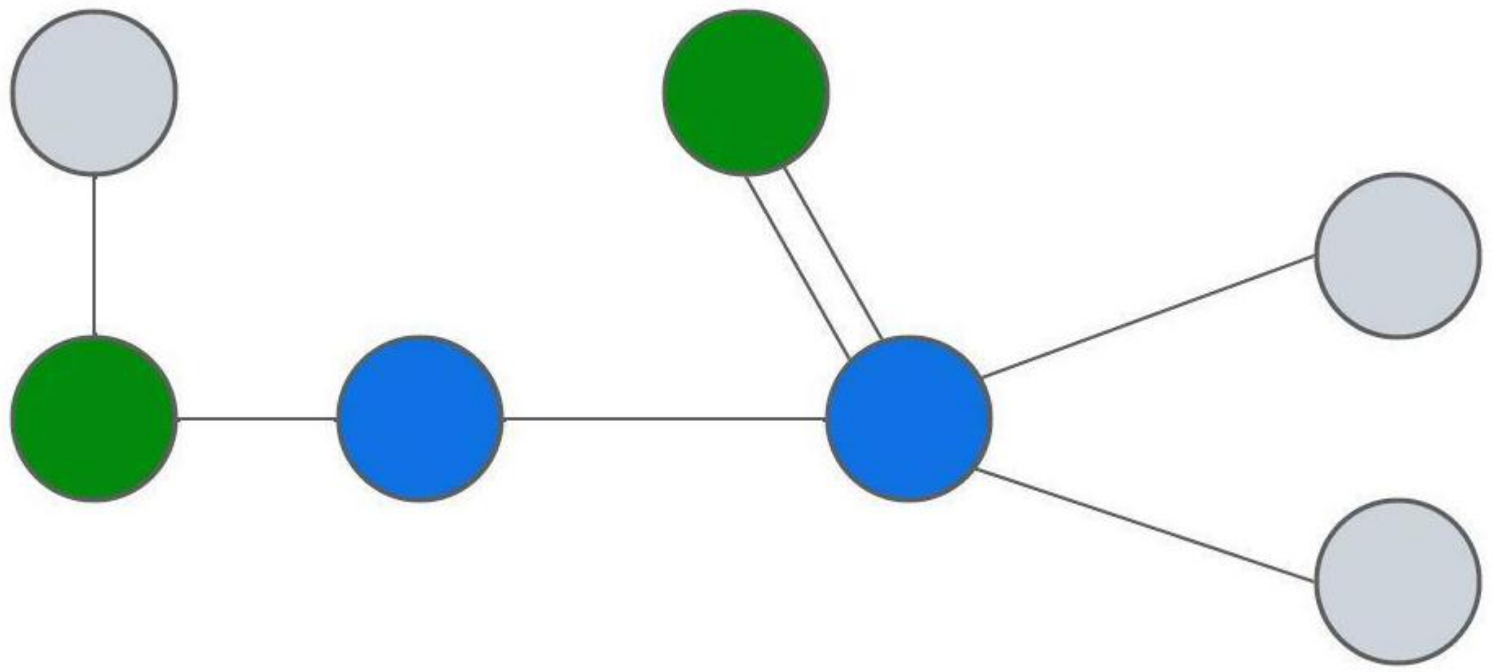
# Networks in DT

# Networks in DT

# Networks in DT

# Networks in DT



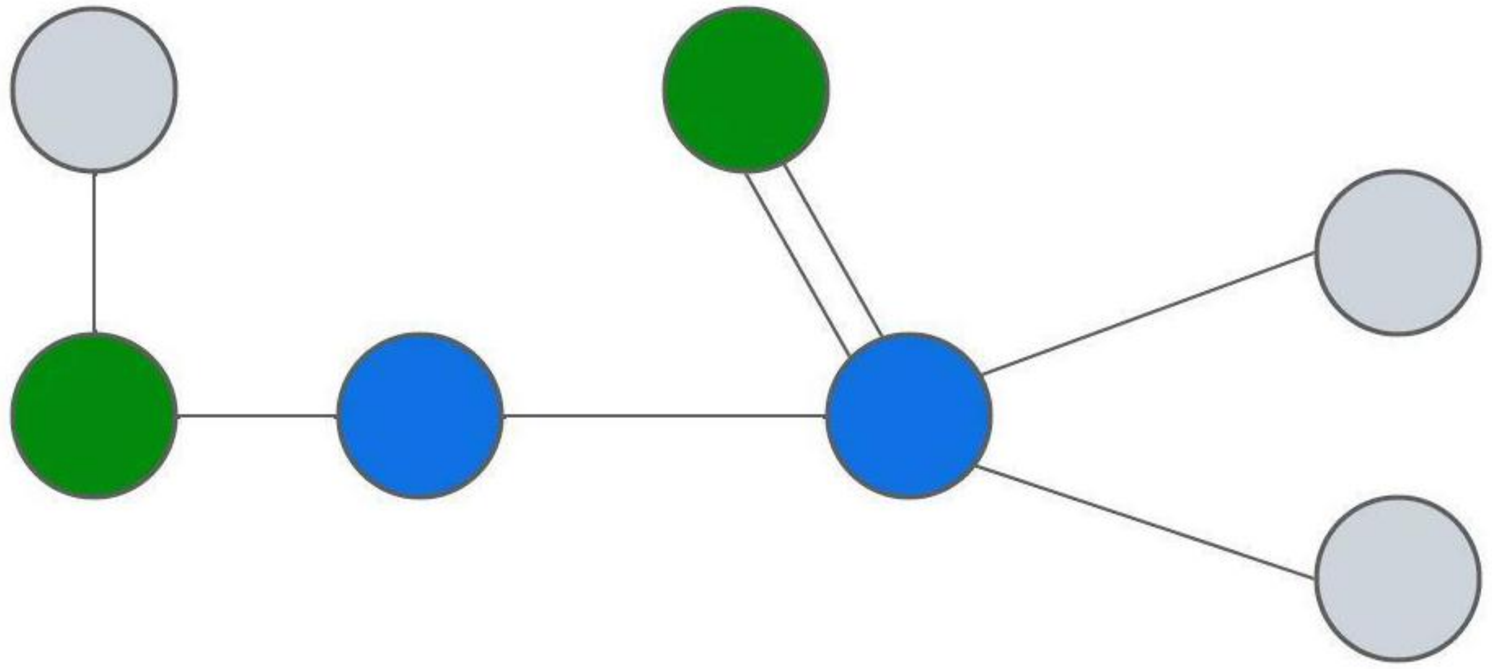



# Networks in DT

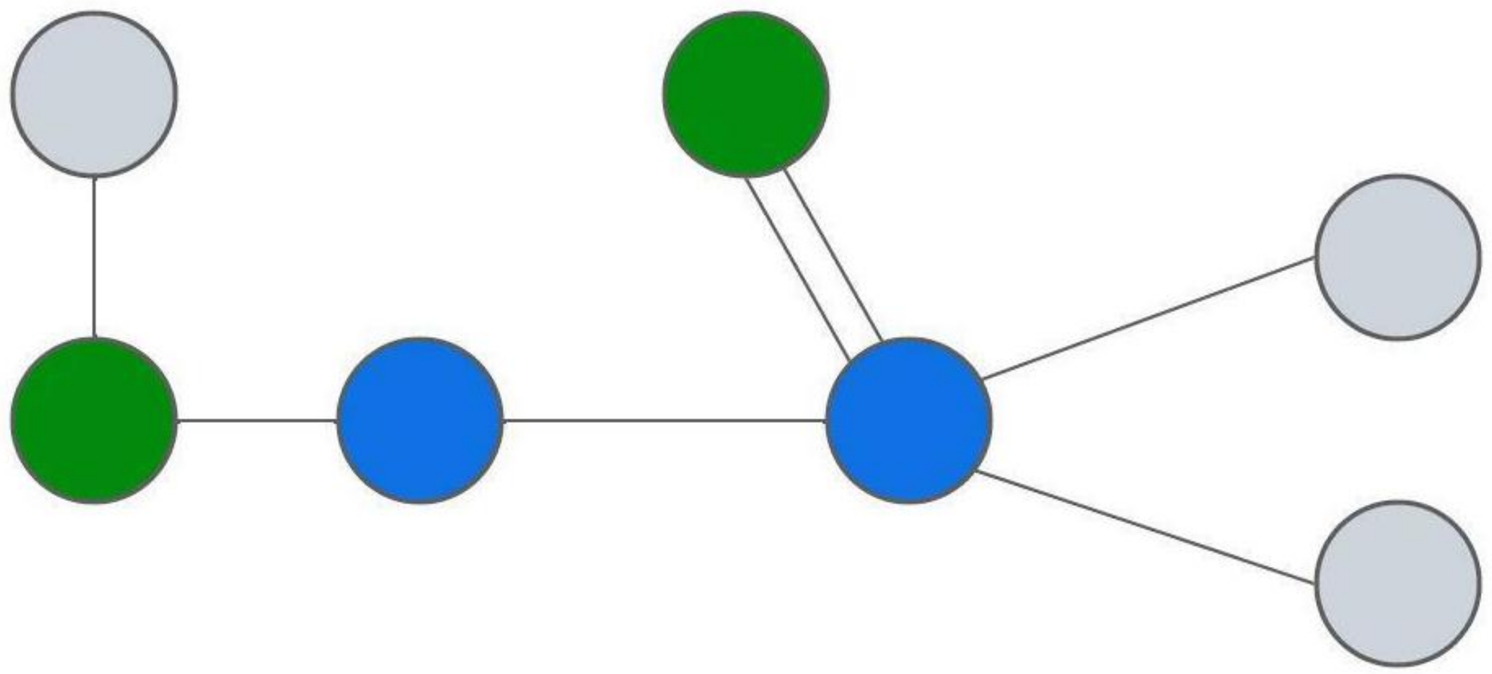
# Networks in DT

# Networks in DT

# Networks in ILP

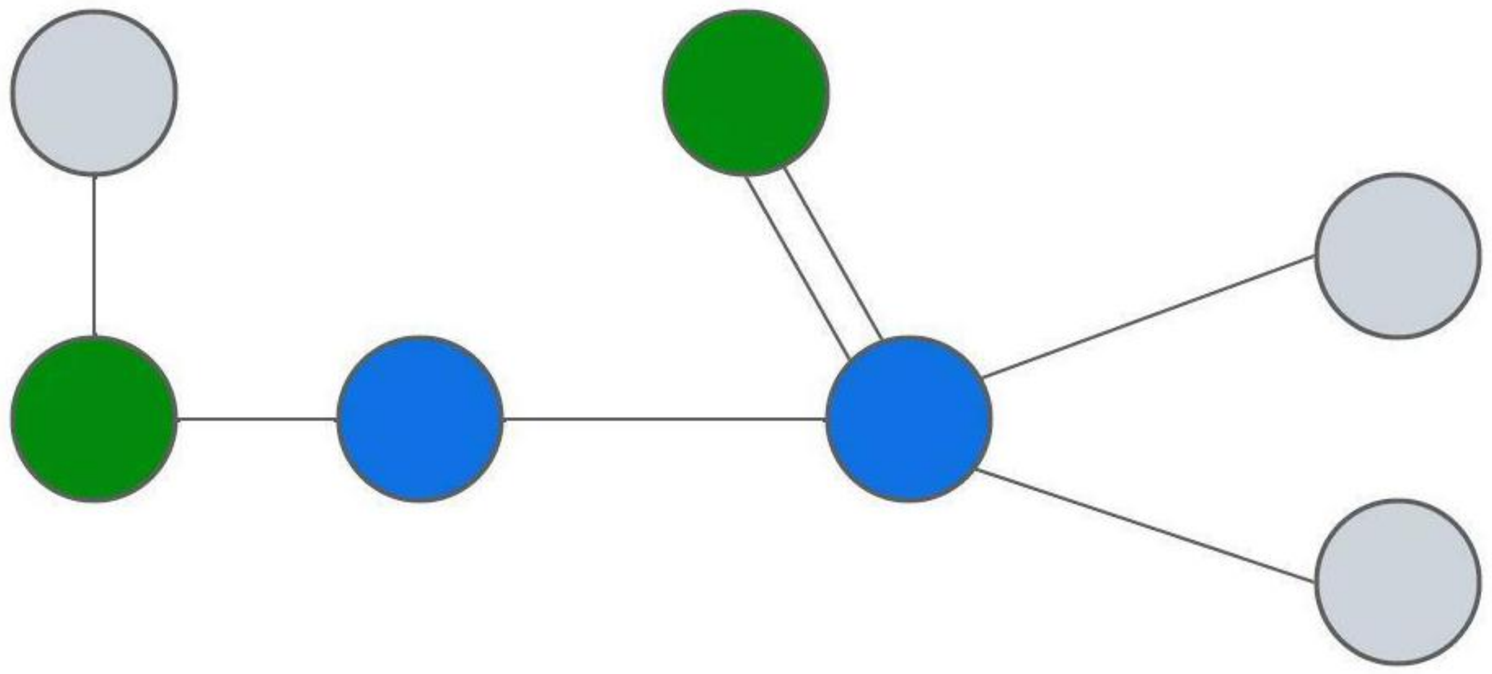


# Networks in ILP

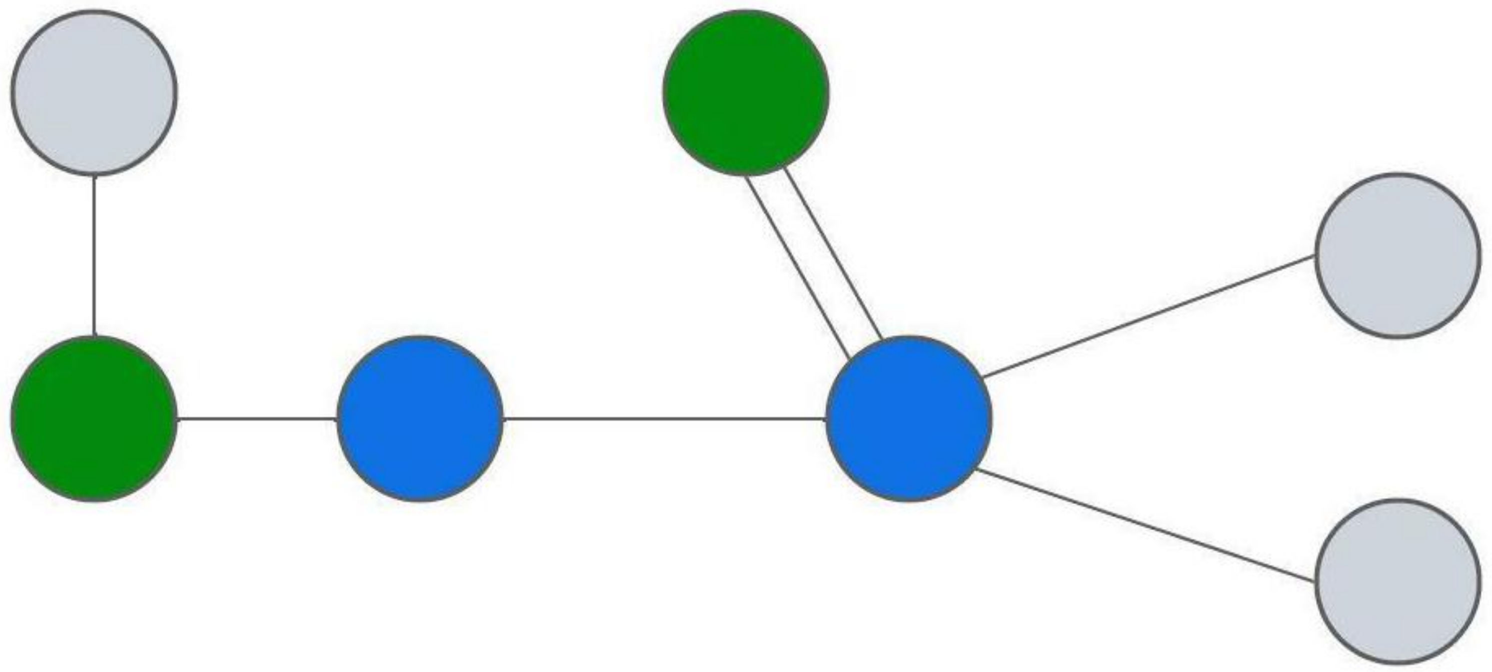


```
% positive example
pharma(molecule1).

% negative example
pharma(molecule2).

% background knowledge
zincsite(a1).
hdonor(a2).
hacc(a3).
bond(a1,a2,single).
bond(a4,a5,double).
distance(a1,a2,1.57).
distance(a2,a3,1.26).
...
```

# Networks in ILP



# Networks in ILP



```
pharma(A):-
   zincsite(A,B),
   hacc(A,C),
   dist(A,B,C,D),
   leq(D,3.58),
   geq(D,1.78),
   hacc(A,E),
   hacc(A,F),
   bond(A,E,F,single).
pharma(A):-
   hacc(A,B),
   hacc(A,C),
   bond(A,B,C,double),
   dist(A,B,C,D),
   leq(D,2.78).
```

# Recap

ILP can:

- Generalise from small amount of data
- Learns hypotheses that are understandable
- Learn from relational data

# Part 2: Building an ILP system

# Part 2: Building an ILP system

How does ILP work?

We have told you that ILP is machine learning with logic.

# Recap: Decision tree learning

Should I play tennis today?

| Day | Weather | Temperature | Humidity | Wind | Play? |
|---|---|---|---|---|---|
| 1 | Sunny | 80 | High | Weak | No |
| 2 | Cloudy | 66 | High | Weak | Yes |
| 3 | Sunny | 43 | Normal | Strong | Yes |
| 4 | Cloudy | 82 | High | Strong | Yes |
| 5 | Rainy | 65 | High | Strong | No |
| 6 | Rainy | 42 | Normal | Strong | No |
| 7 | Rainy | 70 | High | Weak | Yes |
| 8 | Sunny | 81 | High | Strong | No |
| 9 | Cloudy | 69 | Normal | Weak | Yes |
| 10 | Rainy | 67 | High | Strong | No |

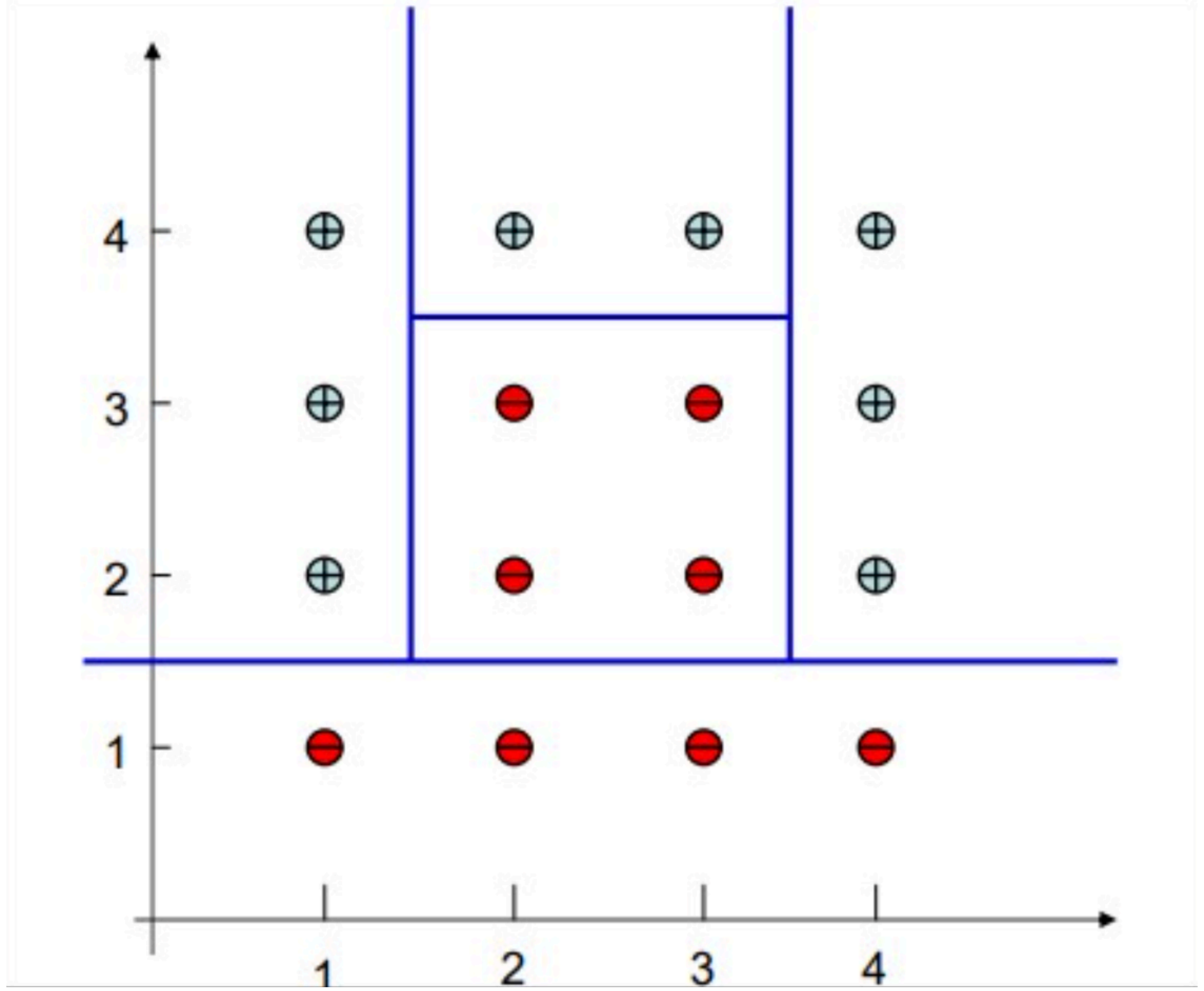
# Recap: Decision tree learning

Step one: what is the goal?

Separate positive examples from negative ones

How do we achieve that?

Reducing information gain



# Recap: Decision tree learning

Step two: how do we represent data?

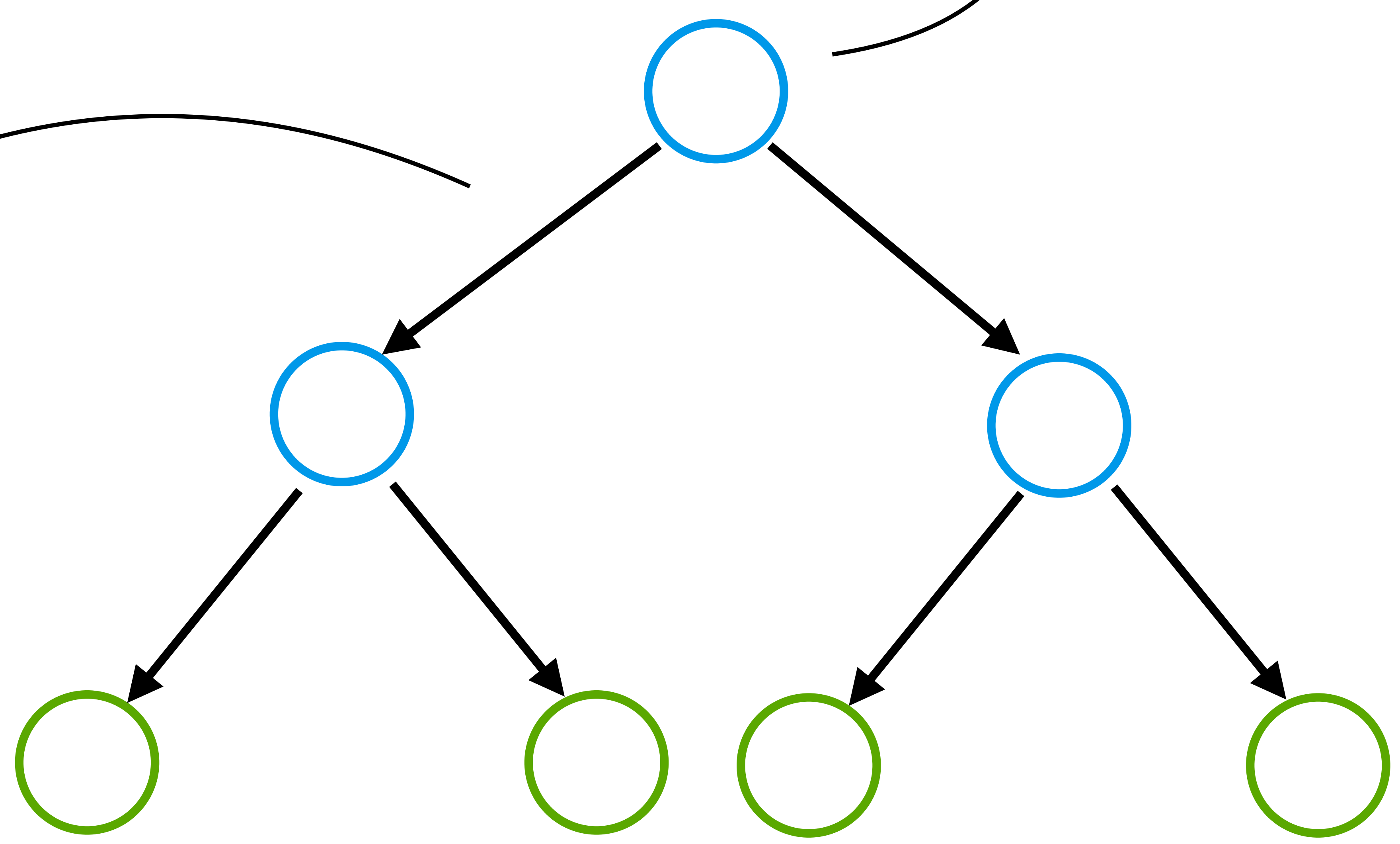
Tabular data

| Day | Weather | Temperature | Humidity | Wind | Play? |
|---|---|---|---|---|---|
| 1 | Sunny | 80 | High | Weak | No |
| 2 | Cloudy | 66 | High | Weak | Yes |
| 3 | Sunny | 43 | Normal | Strong | Yes |
| 4 | Cloudy | 82 | High | Strong | Yes |
| 5 | Rainy | 65 | High | Strong | No |
| 6 | Rainy | 42 | Normal | Strong | No |
| 7 | Rainy | 70 | High | Weak | Yes |
| 8 | Sunny | 81 | High | Strong | No |
| 9 | Cloudy | 69 | Normal | Weak | Yes |
| 10 | Rainy | 67 | High | Strong | No |

# Recap: Decision tree learning

Step three: how do the models look like?

Recursively structured trees



# Recap: Decision tree learning

Step four: What is the hypothesis space?

The set of all tree up to a certain depth

# Recap: Decision tree learning

Step five: How do we search the hypothesis space?

From simpler to more complicated,
step by step

# Recap: Decision tree learning

Step five: How do we search the hypothesis space?

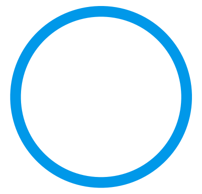
From simpler to more complicated,
step by step

What is the best first feature to split on?

# Recap: Decision tree learning

Step five: How do we search the hypothesis space?

From simpler to more complicated,
step by step



What is the best first feature to split on?

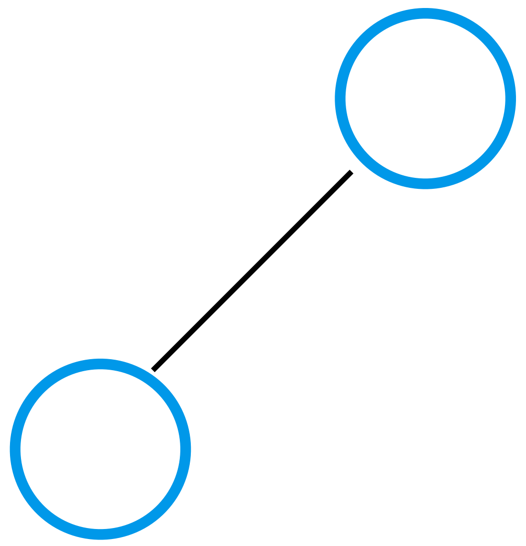
Select and commit!

# Recap: Decision tree learning

Step five: How do we search the hypothesis space?

From simpler to more complicated,
step by step

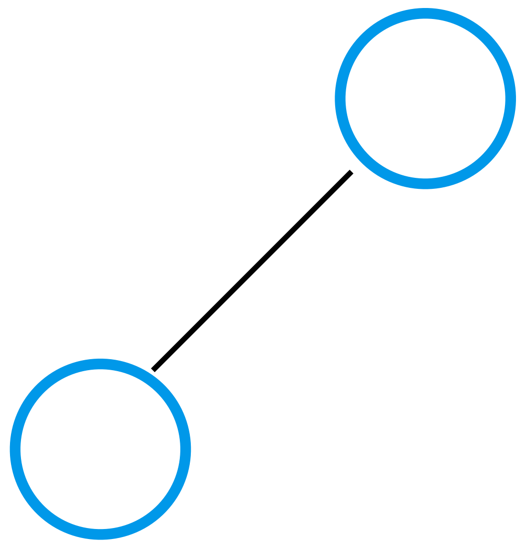
What is the best feature to take next,
for points that satisfy the previous criteria?



# Recap: Decision tree learning

Step five: How do we search the hypothesis space?

From simpler to more complicated,
step by step



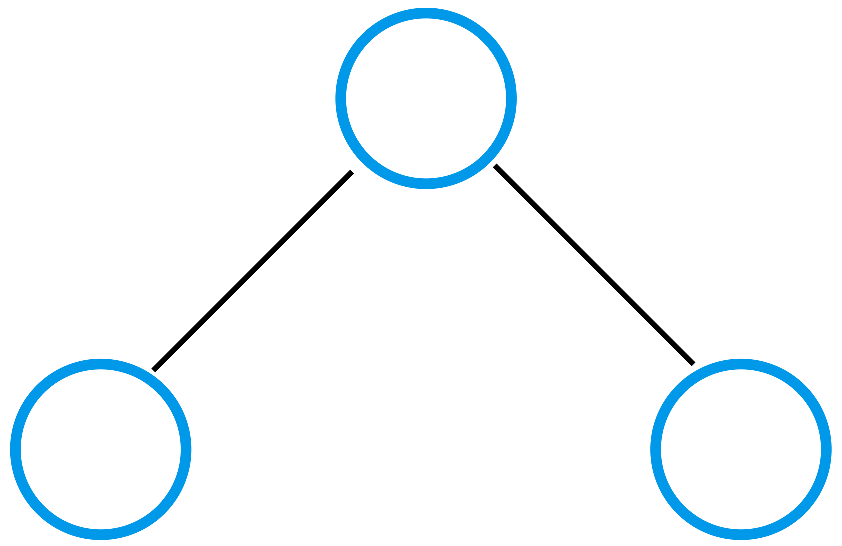
What is the best feature to take next,
for points that satisfy the previous criteria?

Select and commit!

# Recap: Decision tree learning

Step five: How do we search the hypothesis space?

From simpler to more complicated,
step by step



What is the best feature to take next,
for points that **did not** satisfy
the previous criteria?

# Recap: Decision tree learning

Step one: what is the goal?

Step two: how do we represent data?

Step three: how do the models look like?

Step four: What is the hypothesis space?

Step five: How do we search the hypothesis space?

# From decision trees to ILP

Step one: what is the goal?

# From decision trees to ILP

Step one: what is the goal?

Still the same, splitting positive  from negative examples

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

As logic programs (facts)

| Day | Weather | Temperature | Humidity | Wind | Play? |
|---|---|---|---|---|---|
| 1 | Sunny | 80 | High | Weak | No |
| 2 | Cloudy | 66 | High | Weak | Yes |
| 3 | Sunny | 43 | Normal | Strong | Yes |
| 4 | Cloudy | 82 | High | Strong | Yes |

→

```
weather(day1, sunny).
temperature(day1, 80).
humidity(day1, high)
wind(day1, weak).
```

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

Step three: how do the models look like?

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

Step three: how do the models look like?

As logic programs

$$\text{play}(Day, yes) \leftarrow \text{weather}(Day, sunny), \text{wind}(Day, weak)$$

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

Step three: how do the models look like?

Step four: What is the hypothesis space?

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

Step three: how do the models look like?

Step four: What is the hypothesis space?

All valid logic programs

# From decision trees to ILP

Step one: what is the goal?

Step two: how do we represent data?

Step three: how do the models look like?

Step four: What is the hypothesis space?

Step five: How do we search the hypothesis space?

# From decision trees to ILP

Step one: what is the goal?

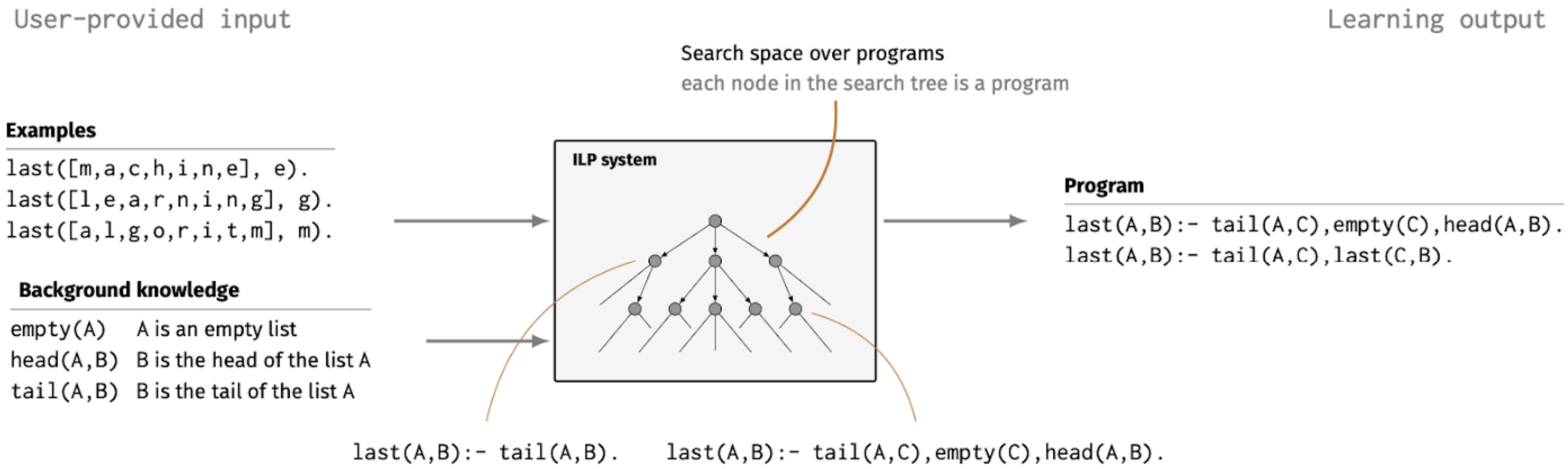
Step two: how do we represent data?

Step three: how do the models look like?

Step four: What is the hypothesis space?

Step five: How do we search the hypothesis space?

See the rest of the tutorial


User-provided input
Learning output
Search space over programs
each node in the search tree is a program
Examples
last([m,a,c,h,i,n,e], e).
last([l,e,a,r,n,i,n,g], g).
last([a,l,g,o,r,i,t,m], m).
Background knowledge
empty(A) A is an empty list
head(A,B) B is the head of the list A
tail(A,B) B is the tail of the list A
ILP system
Program
last(A,B):- tail(A,C),empty(C),head(A,B).
last(A,B):- tail(A,C),last(C,B).
last(A,B):- tail(A,B).
last(A,B):- tail(A,C),empty(C),head(A,B).

Why do we **want** to represent everything in logic?

# Part 2: Building an ILP system

How does ILP work?

Representation language

# Which logic programming language?

# Propositional logic

| | red | green | blue | triangle | rectangle | square | circle | contact_p1 | contact_p2 | contact_p3 | contact_p4 | small | medium | large |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| piece1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| piece2 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| piece3 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| piece4 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

piece1_green.
piece2_blue.
piece2_triangle.
piece1_contact_p2.
piece4_triangle.

# Propositional logic

Limited expressivity (same as DT learners)

Difficult to model problems (not relational)

No recursion

# Full first-order logic

Intractable

$$\forall A.\exists B.\forall C \; right(A,B) \wedge right(B,C) \wedge blue(A) \wedge red(B) \rightarrow contact(A,B) \vee \neg \, square(B).$$

# Horn logic

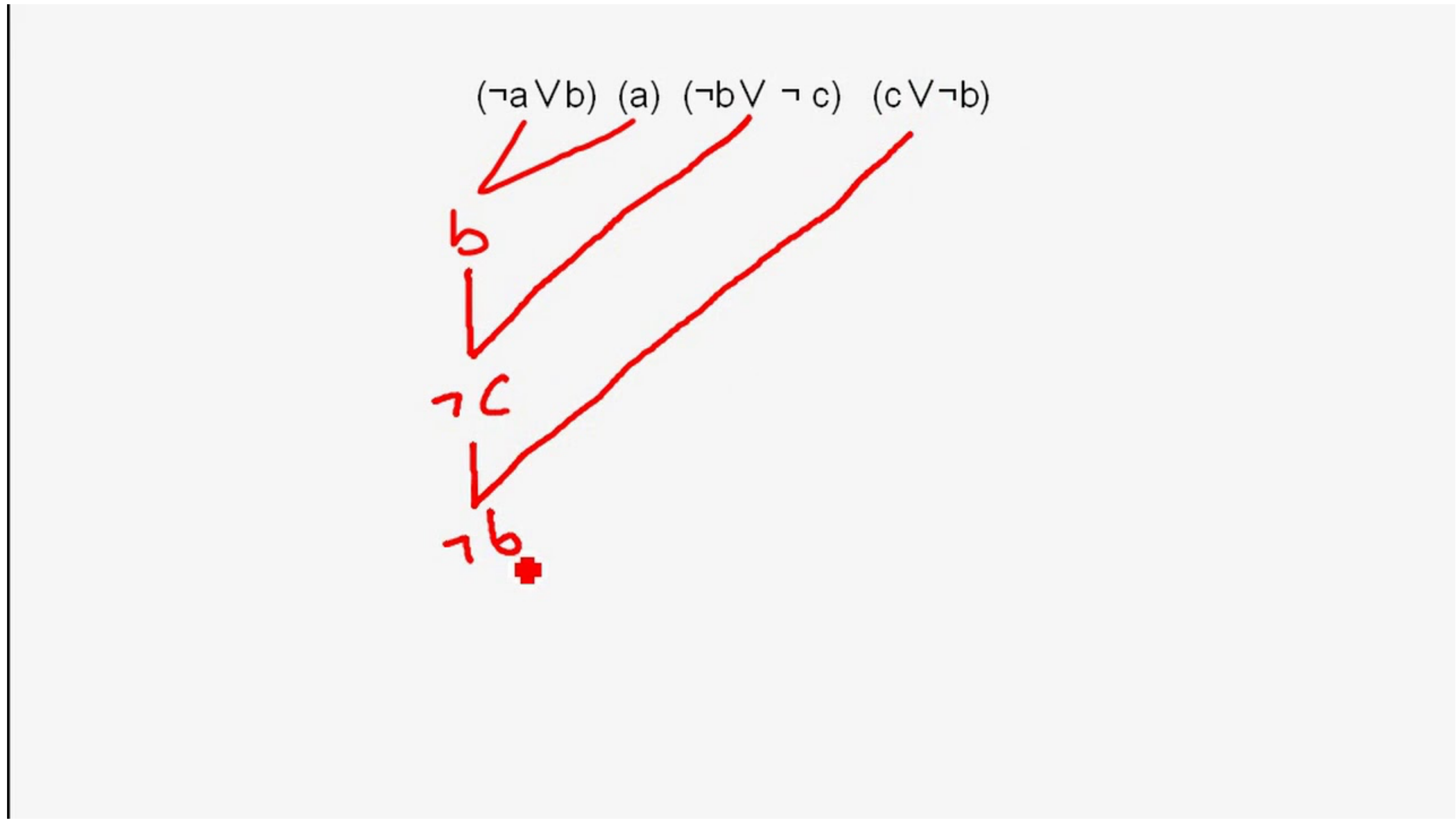
The foundation of most automated reasoning used in SAT etc

zendo(A) ← piece(A,B), blue(B).
blue(p1).

# Horn logic

Important for ***resolution*** because:

- the resolvent of two Horn clauses is itself a Horn clause
- the resolvent of a goal clause and a definite clause is a goal clause



# Prolog

Search uses SLD-resolution (backwards chaining)

# Prolog advantages

Turing complete

Lists and complex data structures

Complex numerical reasoning

# Prolog disadvantages

Not guaranteed to terminate

# Datalog

Definite programs without functional symbols and minor syntactic restrictions

# Datalog advantages

Guaranteed to terminate

Sufficient for most problems in this tutorial

Has nice properties, such as a unique minimal model

# Datalog disadvantages

Not Turing complete (no functional symbols)

# Database vs program

If it uses logical function symbols, it is considered a program.

If it does not, it is considered a database.

# Monotonicity

A logic is **monotonic** when adding knowledge to it does not reduce the logical consequences of that theory.

# Monotonicity

A logic is **non-monotonic** if some conclusions can be removed/ invalidated by adding more knowledge.

# Monotonic logic

```
%% program
sunny.
happy:- sunny.

%% consequences
sunny.
happy.
```

# Monotonic logic

```
%% program
sunny.
happy:- sunny.

%% consequences
sunny.
happy.
```

```
%% program
sunny.
happy:- sunny.
happy:- rich.

%% consequences
sunny.
happy.
```

# Non-monotonic programs

Most use *negation-as-failure* (NAF) (Clark, 1977).

An atom is false if it cannot be proven true.

# Non-monotonic logic

```
%% program
sunny.
happy:- sunny, not weekday.

%% consequences
sunny.
happy.
```

# Non-monotonic logic

```
%% program
sunny.
happy:- sunny, not weekday.

%% consequences
sunny.
happy.
```

```
%% program
sunny.
happy:- sunny, not weekday.
weekday.

%% consequences
sunny.
weekday.
```

# Non-monotonic logic

+ more compact representations

- more difficult to learn, especially recursive programs

# Answer set programming

Language extensions over Datalog, such as choice rules and constraints

# Answer set programming

Language extensions over Datalog, such as choice rules and constraints

A high-level modelling language for SAT/MaxSAT

# Break time

# Part 2: Building an ILP system

How does ILP work?

Search direction

# ILP is search

How do we search the hypothesis space?

# Subsumption

```
C1 = f(A,B):- head(A,B)
C2 = f(X,Y):- head(X,Y),odd(Y).
```

# Subsumption

```
C1 = f(A,B):- head(A,B)
C2 = f(X,Y):- head(X,Y),odd(Y).
```

Then $C_1$ subsumes $C_2$ because

$$\{\mathtt{f(A,B)},\ \neg\mathtt{head(A,B)}\}\theta \subseteq \{\mathtt{f(X,Y)},\ \neg\mathtt{head(X,Y)},\neg\mathtt{odd(Y)}\}$$

with $\theta = \{A/X, Y/B\}$.

# Specialisations

If we add a literal to a rule, it can only become more specific and entail fewer examples

# Specialisations

```
happy(A):-
    lego_builder(A).
```

*subsumes*

```
happy(A):-
    lego_builder(A),
    enjoys_lego(A)
```

# Generalisations

If we add a rule to a program, it can only become more general and entail more examples

only holds for monotonic logic!

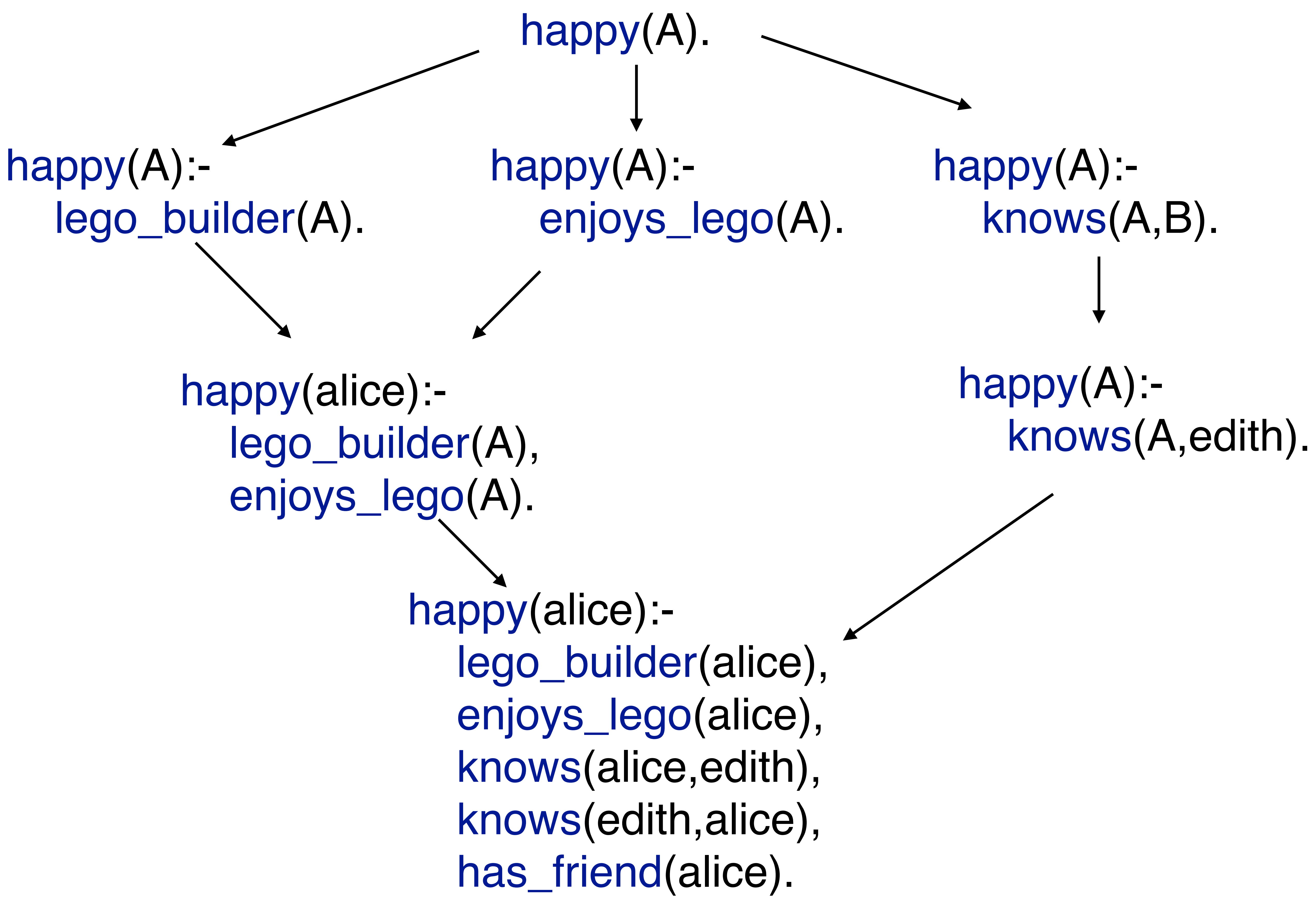
# Generalisations

```
happy(A):- lego_builder(A), enjoys_lego(A).
happy(A):- lego_builder(A), knows(A,B), enjoys_lego(B).
```

*subsumes*

```
happy(A):- lego_builder(A), enjoys_lego(A)
```

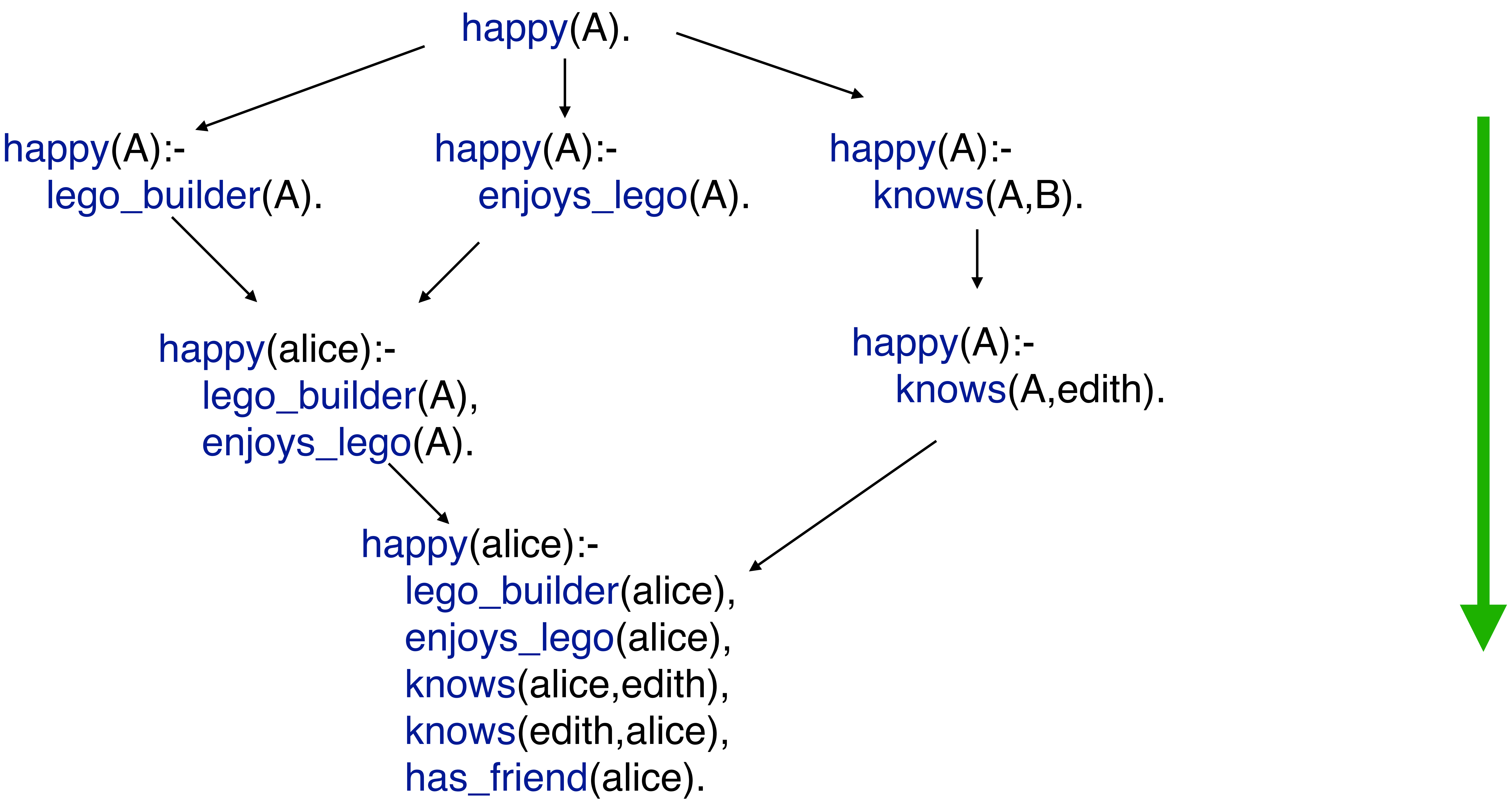
# Subsumption lattice



# Top-down

Start with a general hypothesis and iteratively specialise it

*FOIL, TILDE, HYPER,  QuickFOIL, Progol*, Aleph**


happy(A).
happy(A):- lego_builder(A).
happy(A):- enjoys_lego(A).
happy(A):- knows(A,B).
happy(alice):- lego_builder(A), enjoys_lego(A).
happy(A):- knows(A,edith).
happy(alice):- lego_builder(alice), enjoys_lego(alice), knows(alice,edith), knows(edith,alice), has_friend(alice).

# Top-down

Use example coverage to guide the search, such as through hill climbing and A*

# Top-down

1. Find a good rule that covers some of the positive examples and add it to the program

2. Repeat but focus on `uncovered` examples

# Top-down advantages

Recursion

# Top-down disadvantages

Inefficient

Constants

# Bottom-up

Start with a specific hypothesis and iteratively generalise it

*CIGOL, GOLEM, XHAIL, Progol*, Aleph**

# Bottom-up

```
happy(A).

happy(A):-
   lego_builder(A).

happy(A):-
   enjoys_lego(A).

happy(A):-
   knows(A,B).

happy(alice):-
   lego_builder(A),
   enjoys_lego(A).

happy(A):-
   knows(A,edith).

happy(alice):-
   lego_builder(alice),
   enjoys_lego(alice),
   knows(alice,edith),
   knows(edith,alice),
   has_friend(alice).
```

# Bottom-up

Use example coverage to guide the search, such as through hill climbing and A*

# Bottom-up advantages

Fast

Constants

# Bottom-up disadvantages

Optimality (overfitting)

Recursion

# Top-down and bottom-up

Bottom-up:

1. Find the most specific rule **R** for each example

Top-down

2. Search the generalisations of **R** in a top-down way

*Progol, Aleph*

# Top-down and bottom-up

Search is bound from below by step 1.

Solutions generalise well because of Step 2.

# Top-down and bottom-up advantages

Efficiency

Large rules

Many rules

# Top-down and bottom-up disadvantages

Overfitting

Recursion

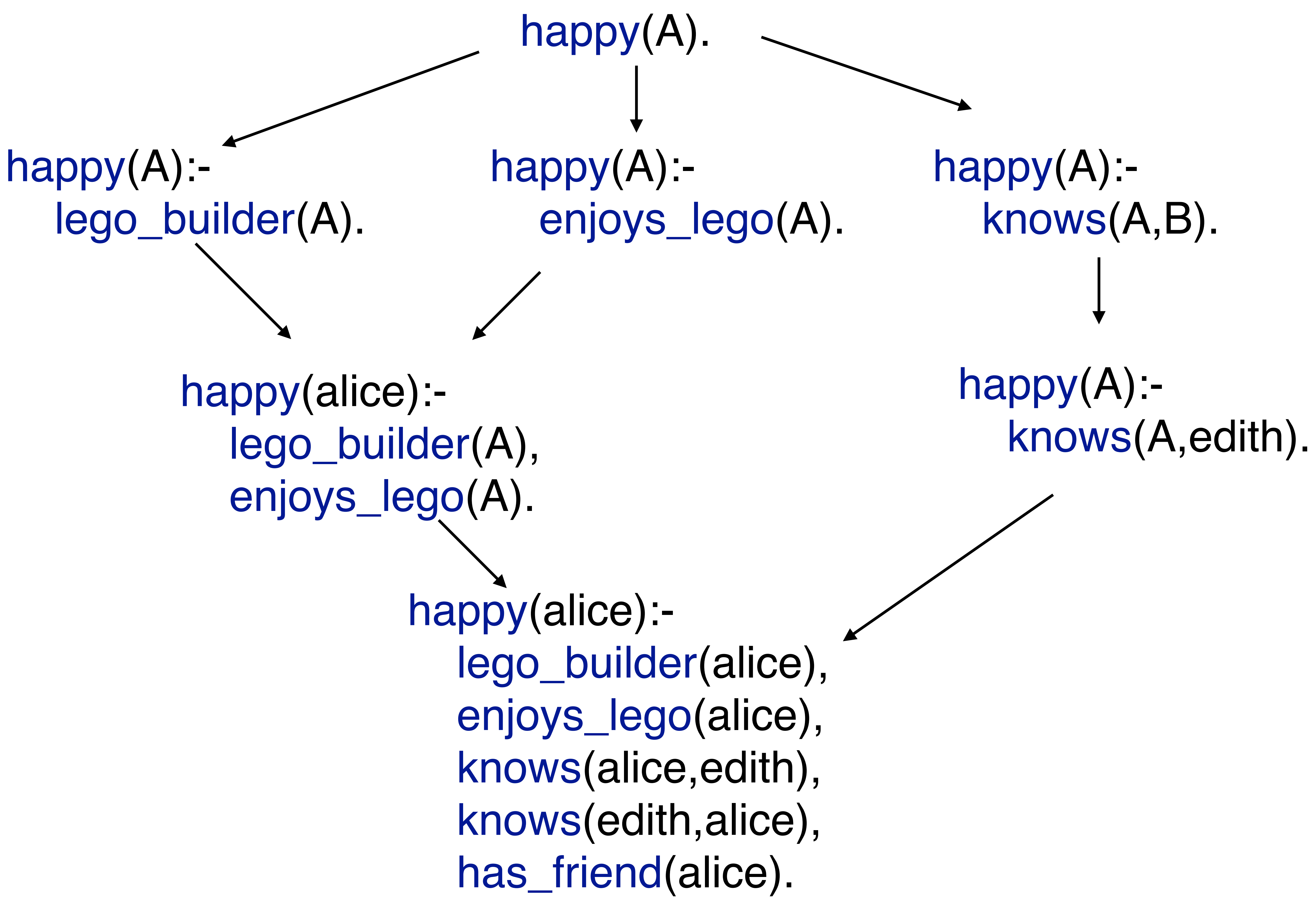
Predicate invention

# Meta-level

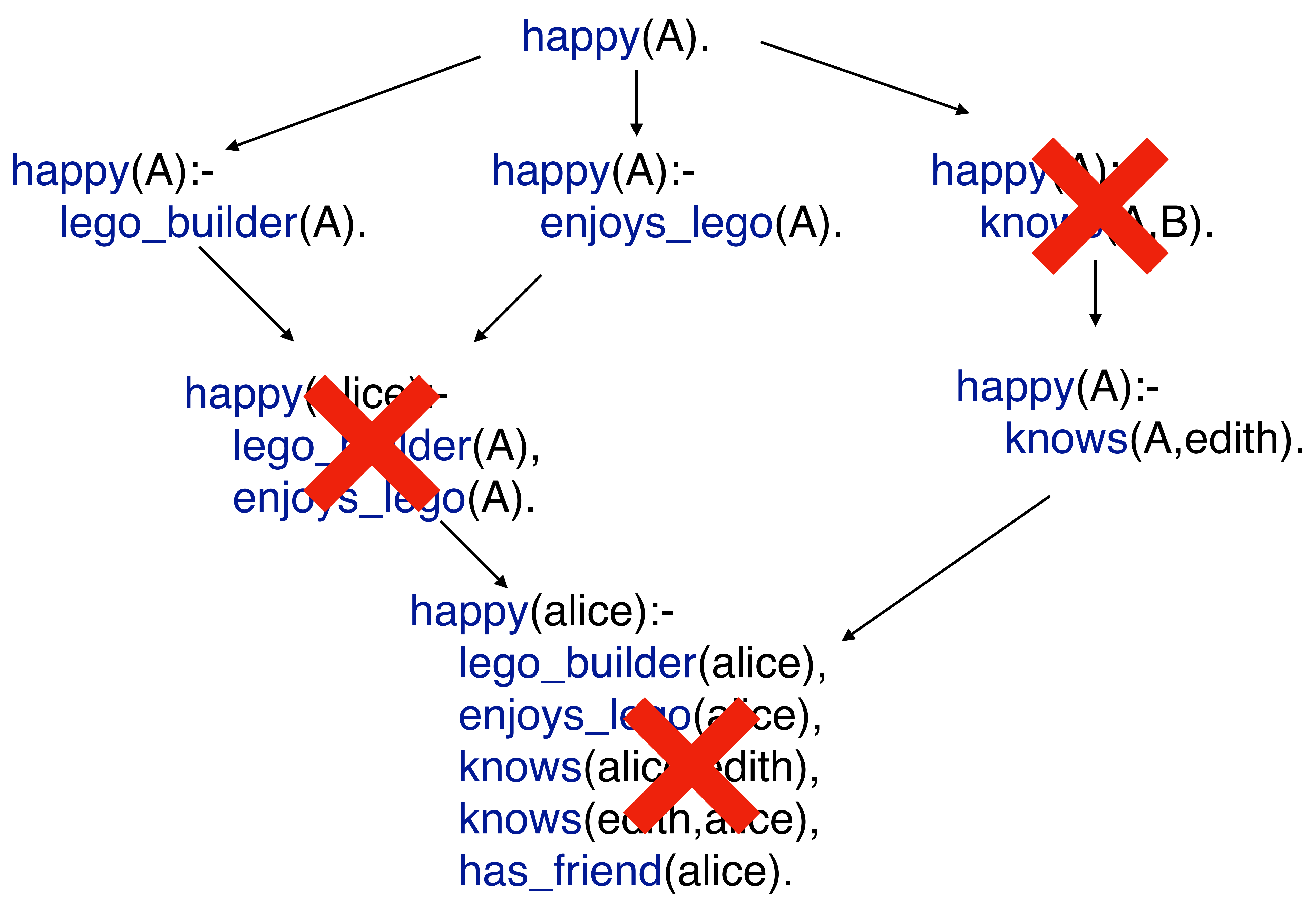
Search all over

*ASPAL, Metagol, ILASP, HEXMIL, DILP, Apperception, Popper*

# Meta-level


happy(A).
happy(A):- lego_builder(A).
happy(A):- enjoys_lego(A).
happy(A):- knows(A,B).
happy(alice):- lego_builder(A), enjoys_lego(A).
happy(A):- knows(A,edith).
happy(alice):- lego_builder(alice), enjoys_lego(alice), knows(alice,edith), knows(edith,alice), has_friend(alice).

# Meta-level



happy(A).
happy(A):- lego_builder(A).
happy(A):- enjoys_lego(A).
happy(A):- knows(A,B).
happy(alice):- lego_builder(A), enjoys_lego(A).
happy(A):- knows(A,edith).
happy(alice):- lego_builder(alice), enjoys_lego(alice), knows(alice,edith), knows(edith,alice), has_friend(alice).

# Meta-level

Use a dedicated solver (SAT/SMT/ASP) to perform to search

# Meta-level advantages

Recursion

Completeness

Optimality

# Meta-level disadvantages

Small domains

Small rules

# Part 2: Building an ILP system

How does ILP work?

Language bias

# How to define the hypothesis space?

The hypothesis is the space of all possible hypotheses that can be built.

An inductive bias is essential to restrict the hypothesis space.

# Mode declarations

Specify which symbols may appear in rules (and their types and directions)

# Mode declarations

Specify which symbols may appear in rules (and their types and directions)

```
modeh(*,target(+list,-char)).
modeb(*,member(+list,-char)).
modeb(*,tail(+list,-list)).
modeb(*,empty(+list)).
```

# Mode declarations

Specify which symbols may appear in rules (and their types and directions)

```
modeh(*,target(+list,-char)).
modeb(*,member(+list,-char)).
modeb(*,tail(+list,-list)).
modeb(*,empty(+list)).
```

```
target(A,B):- member(A,B).
```

✓

# Mode declarations

Specify which symbols may appear in rules (and their types and directions)

```
modeh(*,target(+list,-char)).
modeb(*,member(+list,-char)).
modeb(*,tail(+list,-list)).
modeb(*,empty(+list)).
```

```
target(A,B):- member(A,B).
target(A,B):- tail(A,C), member(C,B).
```

# Mode declarations

Specify which symbols may appear in rules (and their types and directions)

```
modeh(*,target(+list,-char)).
modeb(*,member(+list,-char)).
modeb(*,tail(+list,-list)).
modeb(*,empty(+list)).
```

```
target(A,B):- member(A,B).
target(A,B):- tail(A,C), member(C,B).
target(A,B):- tail(A,C), tail(C,B).
```

# Part 3: features

# Recursion

# Recursion

```
connected(A,B):- edge(A,B).
```

# Recursion

```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),edge(C,B).
```

# Recursion

```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),edge(C,B).
connected(A,B):- edge(A,C),edge(C,D),edge(D,B).
```

# Recursion

```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),edge(C,B).
connected(A,B):- edge(A,C),edge(C,D),edge(D,B).
connected(A,B):- edge(A,C),edge(C,D),edge(D,E),edge(E,B).
```

# Recursion

```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),edge(C,B).
connected(A,B):- edge(A,C),edge(C,D),edge(D,B).
connected(A,B):- edge(A,C),edge(C,D),edge(D,E),edge(E,B).
```

- Cannot generalise to arbitrary depth
- Difficult to learn because of its size

# Recursion

```
connected(A,B):- edge(A,B).
```

# Recursion

```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),connected(C,B).
```

# Recursion

```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),connected(C,B).
```

- Easier to learn because of its size
- Need fewer examples

# Predicate invention

Automatically invent new symbols

# Predicate invention

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
```

# Predicate invention

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
```

# Predicate invention

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),mother(D,B).
```

# Predicate invention

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),father(D,B).
```

# Predicate invention

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),father(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- father(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),mother(C,D),mother(D,B).
```

# Predicate invention

```
greatgrandparent(A,B):- mother(A,C),mother(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- mother(A,C),father(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),father(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),father(C,D),mother(D,B).
greatgrandparent(A,B):- father(A,C),mother(C,D),father(D,B).
greatgrandparent(A,B):- father(A,C),mother(C,D),mother(D,B).
```

- Difficult to learn because of its size
- Need many examples

# Predicate invention

```
greatgrandparent(A,B):- inv(A,C),inv(C,D),inv(D,B).
inv(A,B):- mother(A,B).
inv(A,B):- father(A,B).
```

# Predicate invention

```
greatgrandparent(A,B):- inv(A,C),inv(C,D),inv(D,B).
inv(A,B):- mother(A,B).
inv(A,B):- father(A,B).
```

- Easier to learn because of its size
- Need fewer examples

# Predicate invention + recursion

The combination is **essential** to learn many complex problems

Irene Stahl: The Appropriateness of Predicate Invention as Bias Shift Operation in ILP. Mach. Learn. 20(1-2): 95-117 (1995).

# Predicate invention + recursion

Find the maximum value of a list and add it to every element

# Predicate invention + recursion

```
f(A,B):- inv1(A,Max), ….
inv1(A,B):- head(A,B), empty(B).
inv1(A,B):- head(A,B), inv1(A,C), B>C.
inv1(A,B):- head(A,C), inv1(A,B), B=<D.
```

# Predicate invention + recursion

```
f(A,B):- inv1(A,Max), inv2(A,Max,B).
inv1(A,B):- head(A,B), empty(B).
inv1(A,B):- head(A,B), inv1(A,C), B>C.
inv1(A,B):- head(A,C), inv1(A,B), B=<D.
inv2(A,Max,B):- empty(A), empty(B).
inv2(A,Max,B):- prepend(H1,T1,A), add(Max,H1,H2),
                    inv2(T1,Max,T2), prepend(H2,T2,B).
```

# Negation

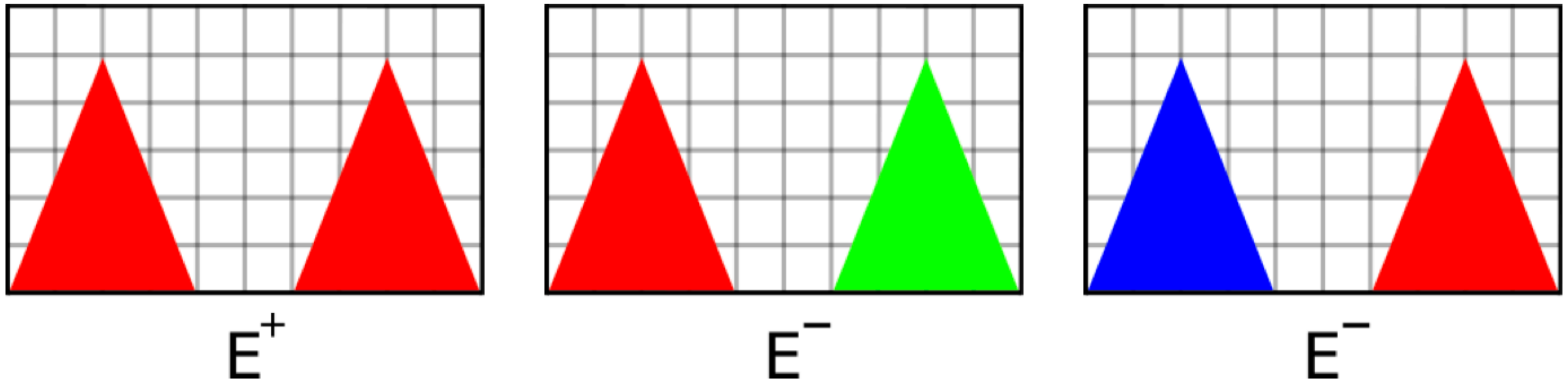
$$B = \left\{ \begin{array}{l} \texttt{bird(A):- penguin(A)} \\ \texttt{bird(alvin)} \\ \texttt{bird(betty)} \\ \texttt{bird(charlie)} \\ \texttt{penguin(doris)} \end{array} \right\} E^+ = \left\{ \begin{array}{l} \texttt{flies(alvin)} \\ \texttt{flies(betty)} \\ \texttt{flies(charlie)} \end{array} \right\} E^- = \left\{ \texttt{flies(doris)} \right\}$$

# Negation

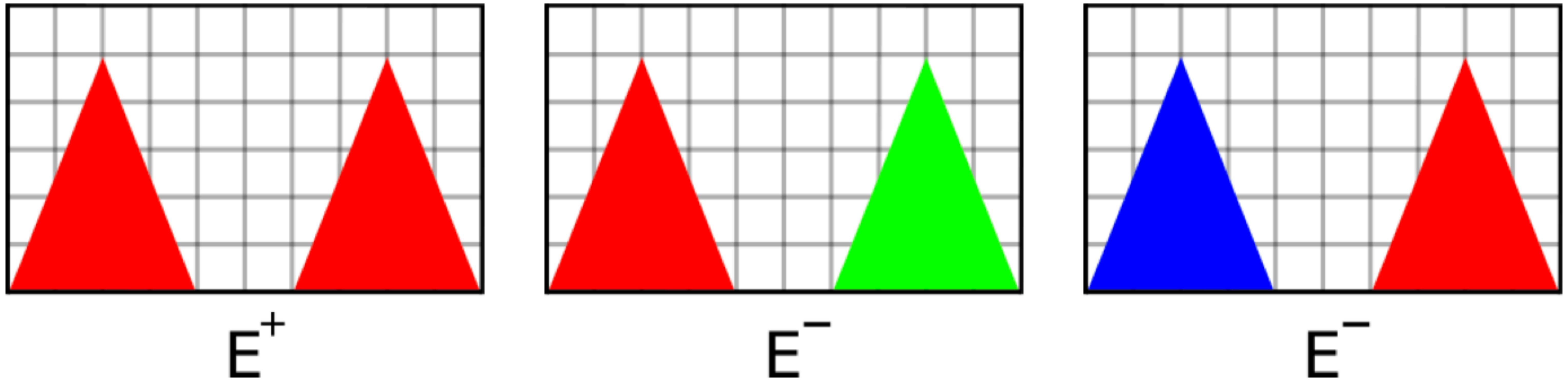
$$B = \left\{ \begin{array}{l} \texttt{bird(A):- penguin(A)} \\ \texttt{bird(alvin)} \\ \texttt{bird(betty)} \\ \texttt{bird(charlie)} \\ \texttt{penguin(doris)} \end{array} \right\} E^{+} = \left\{ \begin{array}{l} \texttt{flies(alvin)} \\ \texttt{flies(betty)} \\ \texttt{flies(charlie)} \end{array} \right\} E^{-} = \left\{ \texttt{flies(doris)} \right\}$$

$$H = \left\{ \texttt{flies(A):- bird(A), not penguin(A)} \right\}$$

# Predicate invention + negation

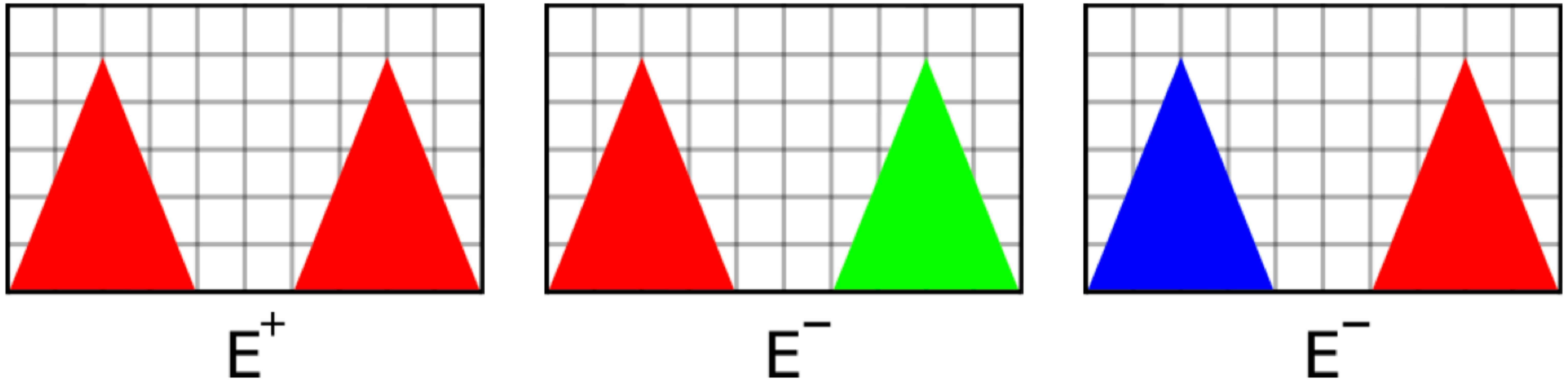


# Predicate invention + negation



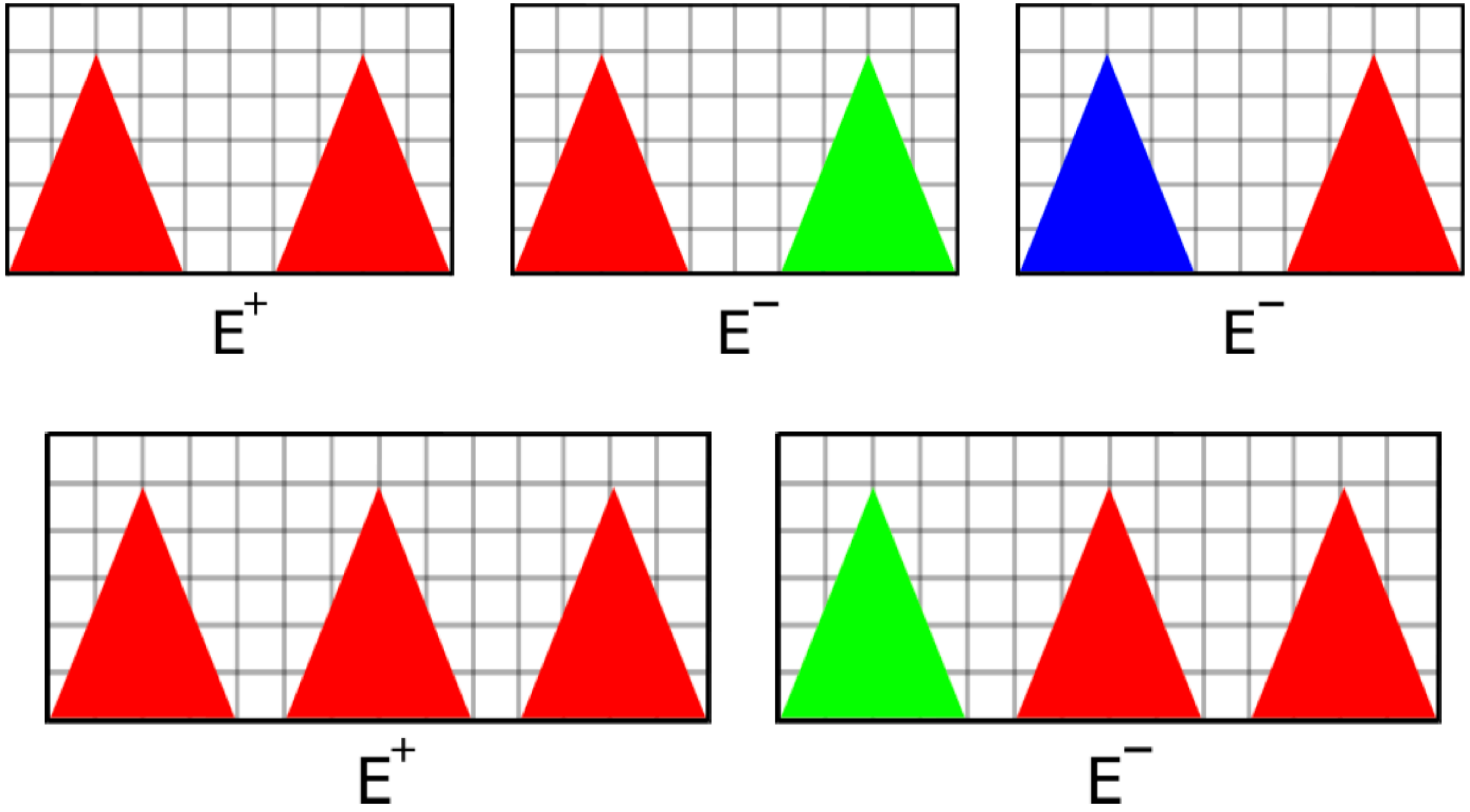

*"there are two red cones"*

# Predicate invention + negation




f(S):- cone(S,A),red(A),cone(S,B),red(B),all_diff(A,B).

# Predicate invention + negation



# Predicate invention + negation

*“there are exactly two cones and both are red”*

*or*

*“there are exactly three cones and all three are red”*

# Predicate invention + negation

very messy program here

# Predicate invention + negation

```
f(S):- not inv1(S).
inv1(S):- cone(S,P), not red(P).
```

# Predicate invention + negation

```
f(S):- not inv1(S).
inv1(S):- cone(S,P), not red(P).
```

*there is a cone that is not red*

# Predicate invention + negation

*it is not true that there is a cone that is not red*

```
f(S):- not inv1(S).
inv1(S):- cone(S,P), not red(P).
```

*there is a cone that is not red*

# Predicate invention + negation

```
f(S):- not inv1(S).
inv1(S):- cone(S,P), not red(P).
```

*all the cones are red*

# Higher-order invention

| Input | Output |
|---|---|
| [alic**e**,bo**b**,charli**e**] | [alic,bo,charli] |
| [inductiv**e**,logi**c**,programmin**g**] | [inductiv,logi,programmin] |
| [ferrar**a**,orlean**s**,londo**n**,kyot**o**] | [ferrar,orlean,londo,kyot] |

Cropper, Morel, and Muggleton, “Learning higher-order logic programs”, MLJ 2020

# Higher-order invention

```
f(A,B):-map(A,B,inv1).
inv1(A,B):-inv2(A,C),tail(C,D),inv2(D,B).
inv2(A,B):-reduceback(A,B,concat).
```

# Higher-order invention

```
f(A,B):-map(A,B,inv1).
inv1(A,B):-inv2(A,C),tail(C,D),inv2(D,B).
inv2(A,B):-reduceback(A,B,concat).
```

*invents reverse*

# Higher-order invention

*invents droplast*

```
f(A,B):-map(A,B,inv1).
inv1(A,B):-inv2(A,C),tail(C,D),inv2(D,B).
inv2(A,B):-reduceback(A,B,concat).
```

*invents reverse*

# Higher-order invention

*reuses inv2*

*invents droplast*

```
f(A,B):-map(A,B,inv1).
inv1(A,B):-inv2(A,C),tail(C,D),inv2(D,B).
inv2(A,B):-reduceback(A,B,concat).
```

*invents reverse*

# Higher-order invention

| Input | Output |
|---|---|
| [alic**e**,bo**b**,**charlie**] | [alic,bo] |
| [inductiv**e**,logi**c**,**programming**] | [inductiv,logi] |
| [ferrar**a**,orlean**s**,londo**n**,**kyoto**] | [ferrar,orlean,londo] |

# Higher-order invention

```
f(A,B):-map(A,C,inv1),inv1(C,B).
inv1(A,B):-inv2(A,C),tail(C,D),inv2(D,B).
inv2(A,B):-reduceback(A,B,concat).
```

# Higher-order invention

*invents droplast*

*reuses droplast*

```
f(A,B):-map(A,C,inv1),inv1(C,B).
inv1(A,B):-inv2(A,C),tail(C,D),inv2(D,B).
inv2(A,B):-reduceback(A,B,concat).
```

# Optimality: textual complexity

```
f(A):- element(A,1).
f(A):- element(A,2).
f(A):- element(A,3).
f(A):- element(A,4).
f(A):- element(A,5).
f(A):- element(A,6).
f(A):- element(A,7).
f(A):- element(A,8).
f(A):- element(A,9).
f(A):- element(A,10).
```

# Optimality: textual complexity

```
f(A):- element(A,101),element(A,102).
```

# Optimality: efficiency

| **input** | **output** |
|---|---|
| sheep | e |
| alaca | a |
| chicken | ? |

Cropper & Muggleton, "Learning efficient logic programs", MLJ 2019

# Optimality: efficiency

| **input** | **output** |
| --- | --- |
| sheep | e |
| alaca | a |
| chicken | c |

# Optimality: efficiency

```
f(A,B):- head(A,B),tail(A,C),element(C,B).
f(A,B):- tail(A,C),f(C,B).
```

# Optimality: efficiency

```
f(A,B):- head(A,B),tail(A,C),element(C,B).
f(A,B):- tail(A,C),f(C,B).
```

O(n^2)

# Optimality: efficiency

```
f(A,B):- mergesort(A,C),inv1(C,B).
inv1(A,B):- head(A,B),tail(A,C),head(C,B).
inv1(A,B):- tail(A,C),inv1(C,B).
```

# Optimality: efficiency

```
f(A,B):- mergesort(A,C),inv1(C,B).
inv1(A,B):- head(A,B),tail(A,C),head(C,B).
inv1(A,B):- tail(A,C),inv1(C,B).
```

O(n log n)

# Optimality: efficiency

```
f(A,B):- mergesort(A,C),inv1(C,B).
inv1(A,B):- head(A,B),tail(A,C),head(C,B).
inv1(A,B):- tail(A,C),inv1(C,B).
```

Predicate invention and recursion!

# Noise

- noisy examples
- noisy BK

# Noisy examples

**Almost all ILP systems handle noisy examples!**

# Noisy examples

**Sequential covering or divide-and-conquer**

- Aleph, Progol, FOIL, TILDE, ATOM, QuickFOIL

# Noisy examples

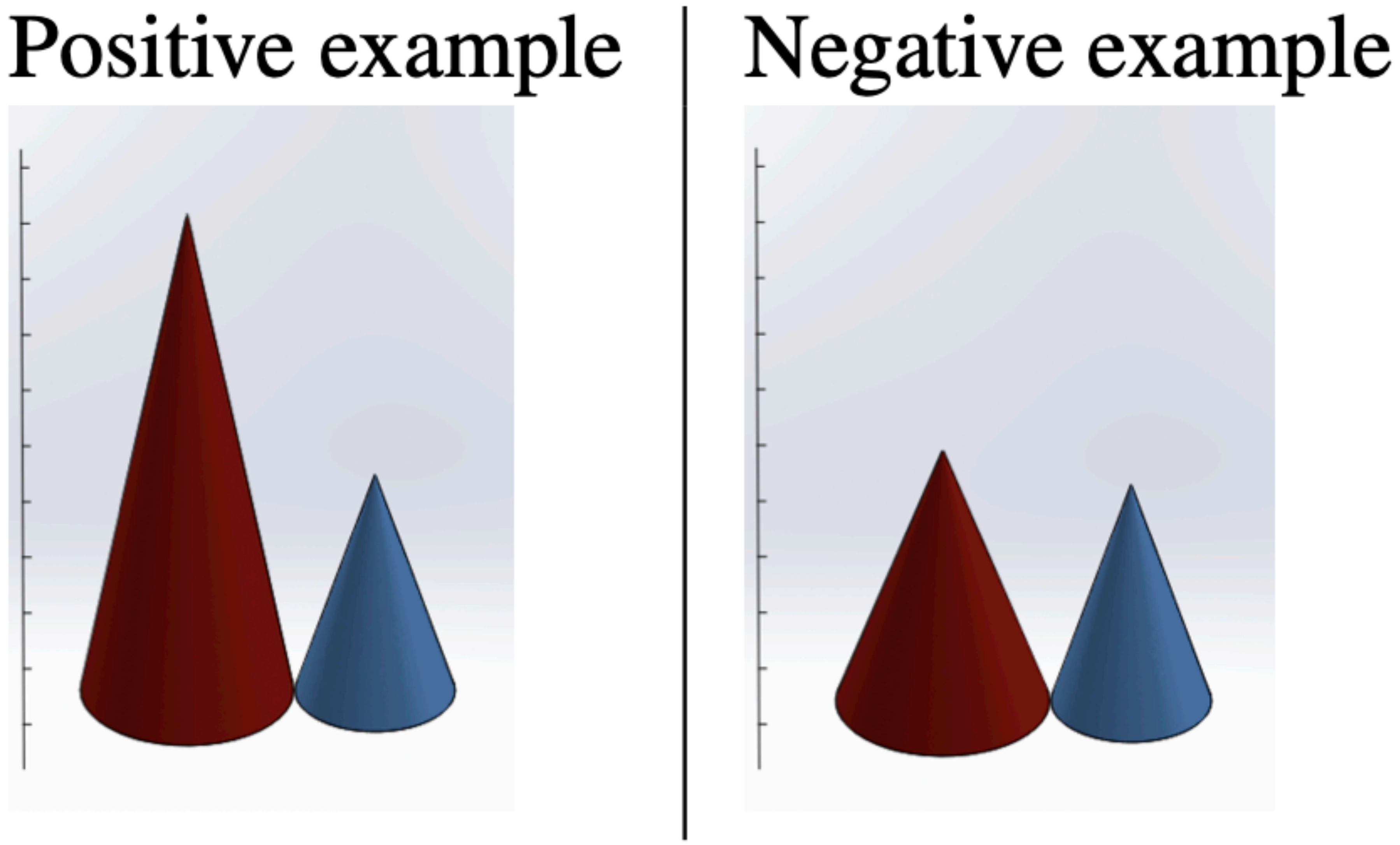
**Solver optimisation**

- ILASP, Popper

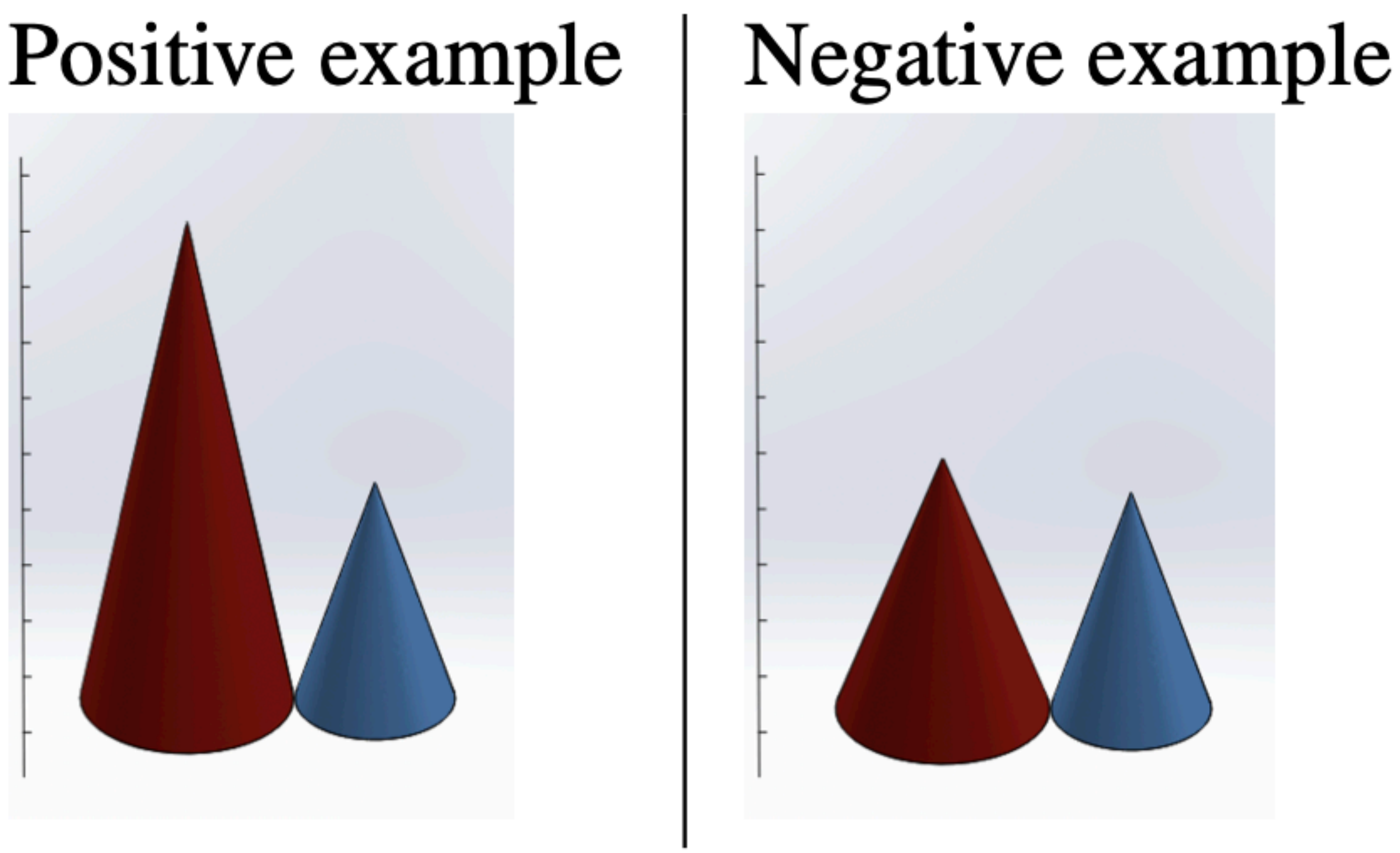
# Noisy BK

Almost no ILP systems handle noisy BK!

# Numerical data



# Numerical data




zendo(A):- piece(A,B),contact(B,C),size(C,D),geq(D,7).

# Numerical data

```
equilibrium(A):- mass(A,B),forces(A,C),sum(C,D),mult(B,9.807,D).
```

# Numerical data

```
pharma(A):- zinc(A,B), hacc(A,C), dist(A,B,C,D), leq(D,4.18), geq(D,2.22).
pharma(A):- hacc(A,C), hacc(A,E), dist(A,B,C,D), geq(D,1.23), leq(D,3.41).
pharma(A):- zinc(A,C), zinc(A,B), bond(B,C,du), dist(A,B,C,D), leq(D,1.23).
```

# Break time

# Part 4: ILP systems

# TILDE

Divide-and-conquer strategy: recursively split the data using a conjunction with the highest information gain

# TILDE

**Given**:
- Classes C
- Mode declarations M
- Positive (E+) and negative (E-) examples as interpretations
- BK in the form of a definite program

# TILDE

**Given**:

- Classes C
- Mode declarations M
- Positive (E+) and negative (E-) examples as interpretations
- BK in the form of a definite program

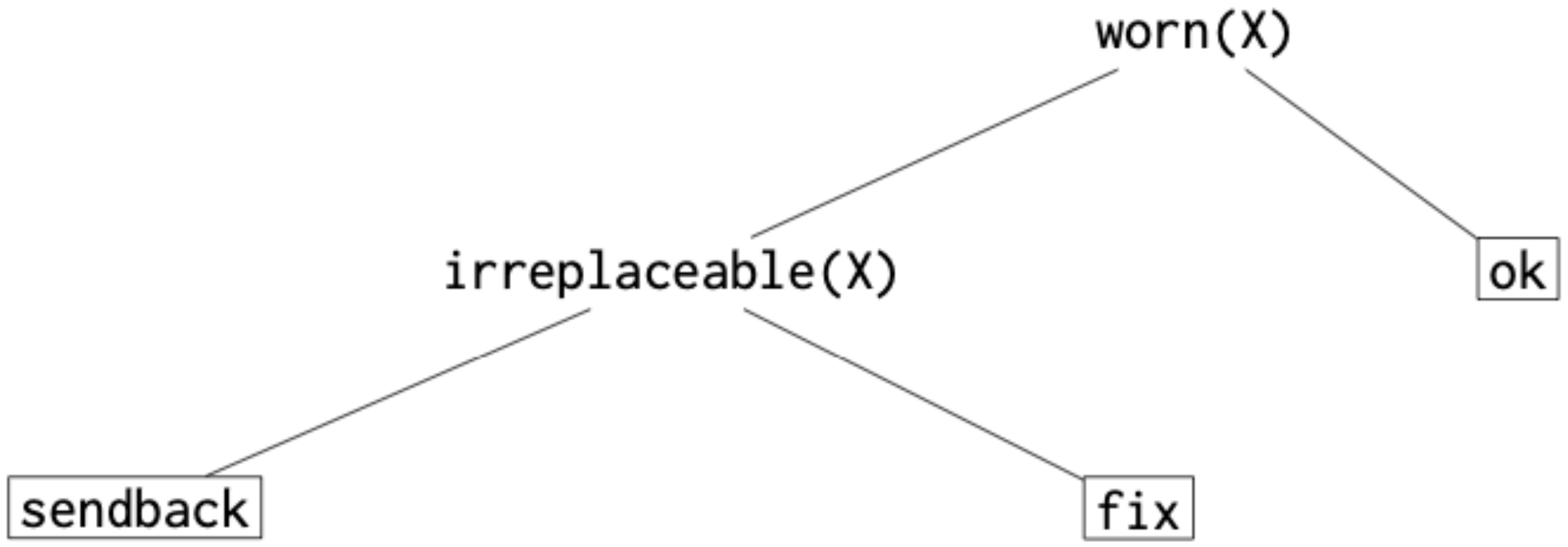
**Return**:

A normal program hypothesis H such that:

- H is consistent with M
- H is complete and consistent

# TILDE

```
class(X,sendback) :- worn(X), irreplaceable(X), !.
class(X,fix) :- worn(X), !.
class(X,ok).
```



# TILDE

**Advantages:**
- Can learn normal logic programs
- Supports both categorical and numerical data

**Disadvantages:**
- Does not support recursion
- Need for lookahead

# ASPAL

1. Generate all possible rules

# ASPAL

1. Generate all possible rules

2. Use an ASP solver to find a subset of the rules that is complete and consistent

# ASPAL

**Given**:

- Mode declarations M
- B in the form of a normal program
- Positive (E+) and negative (E-) examples as a set of facts
- A penalty function γ

# ASPAL

**Given**:

- Mode declarations M
- B in the form of a normal program
- Positive (E+) and negative (E-) examples as a set of facts
- A penalty function γ

**Return**:

A normal program hypothesis H such that:

- H is consistent with M
- H is complete and consistent
- The penalty function γ is minimal

# ASPAL

$$B = \left\{ \begin{array}{l} \texttt{bird(alice).} \\ \texttt{bird(betty).} \\ \texttt{can(alice,fly).} \\ \texttt{can(betty,swim).} \\ \texttt{ability(fly).} \\ \texttt{ability(swim).} \end{array} \right\}$$

$$E^+ = \left\{ \texttt{penguin(betty).} \right\} E^- = \left\{ \texttt{penguin(alice).} \right\}$$

$$M = \left\{ \begin{array}{l} \texttt{modeh(1, penguin(+bird)).} \\ \texttt{modeb(1, bird(+bird)).} \\ \texttt{modeb(*,not can(+bird,\#ability))} \end{array} \right\}$$

# ASPAL

```
penguin(X):- bird(X).
penguin(X):- bird(X), not can(X,fly).
penguin(X):- bird(X), not can(X,swim).
penguin(X):- bird(X), not can(X,swim), not can(X,fly).
```

# ASPAL

Builds rules with extra ‘abducible’ literals.

```
penguin(X):- bird(X), rule(r1).
penguin(X):- bird(X), not can(X,C1), rule(r2,C1).
penguin(X):- bird(X), not can(X,C1), not can(X,C2), rule(r3,C1,C2).
```

A flag which denotes whether this rule has been selected

# ASPAL

```
bird(alice).
bird(betty).
can(alice,fly).
can(betty,swim).
ability(fly).
ability(swim).
penguin(X):- bird(X), rule(r1).
penguin(X):- bird(X), not can(X,C1), rule(r2,C1).
penguin(X):- bird(X), not can(X,C1), not can(X,C2), rule(r3,C1,C2).
0 {rule(r1),rule(r2,fly),rule(r2,swim),rule(r3,fly,swim\}4.
goal : - penguin(betty), not penguin(alice).
: - not goal.
```

# ASPAL

Guess which rules should be included

```
bird(alice).
bird(betty).
can(alice,fly).
can(betty,swim).
ability(fly).
ability(swim).
penguin(X):- bird(X), rule(r1).
penguin(X):- bird(X), not can(X,C1), rule(r2,C1).
penguin(X):- bird(X), not can(X,C1), not can(X,C2), rule(r3,C1,C2).
0 {rule(r1),rule(r2,fly),rule(r2,swim),rule(r3,fly,swim\}4.
goal : - penguin(betty), not penguin(alice).
: - not goal.
```

# ASPAL

The role of the ASP solver is to:

- prove the positive examples
- disprove the negative examples
- guess rules when necessary

# ASPAL

```
rule(r2,c(fly)).
```

```
penguin(A):- not can(A,fly).
```

# ASPAL - why does it work?

It combines the search for a solution with example coverage.

By using ASP solvers, it can jump around the search space.

ASP solvers are really good!

# ASPAL advantages

Simple

Recursion

Optimality

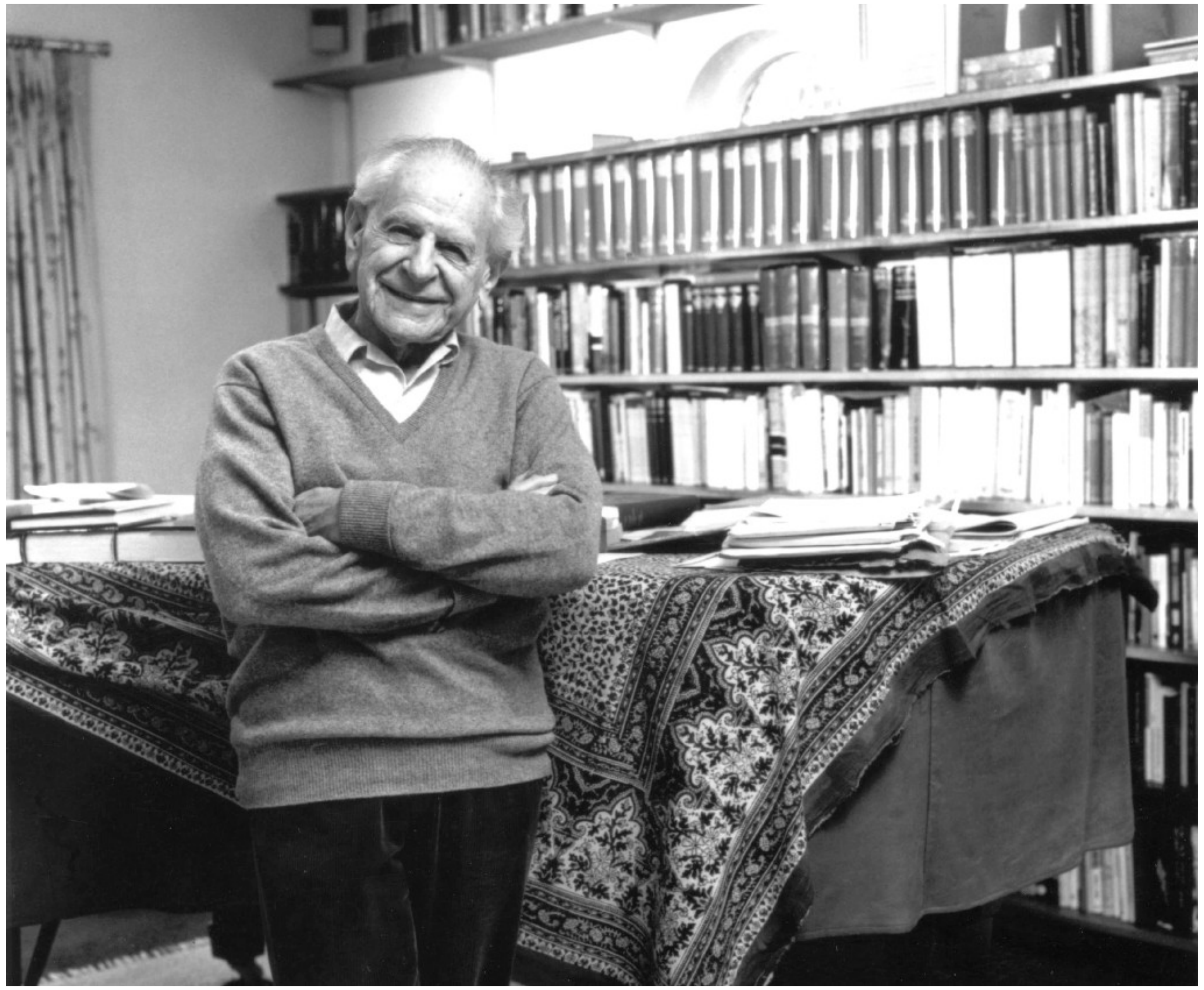
Efficient for small rules

# ASPAL disadvantages

Cannot learn large rules

Cannot handle large BK

# Popper



**Friday, February 10**
JT3: Machine Learning 1
9:30 – 10:45

# Popper

1. Generate *programs* one-at-a-time

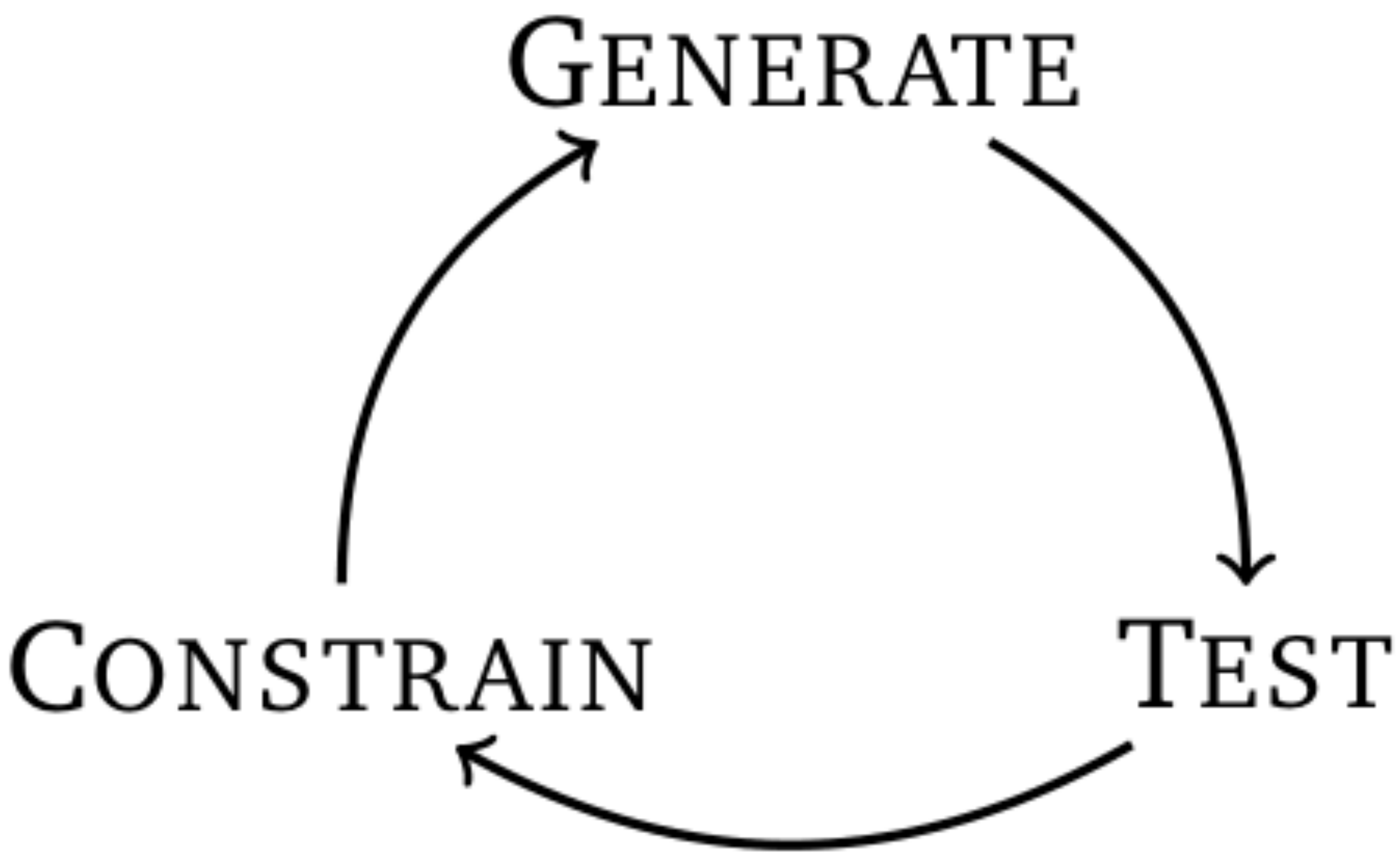
# Popper

1. Generate *programs* one-at-a-time
2. Test programs on the data and use the outcome to build *syntactic* constraints on the hypothesis space

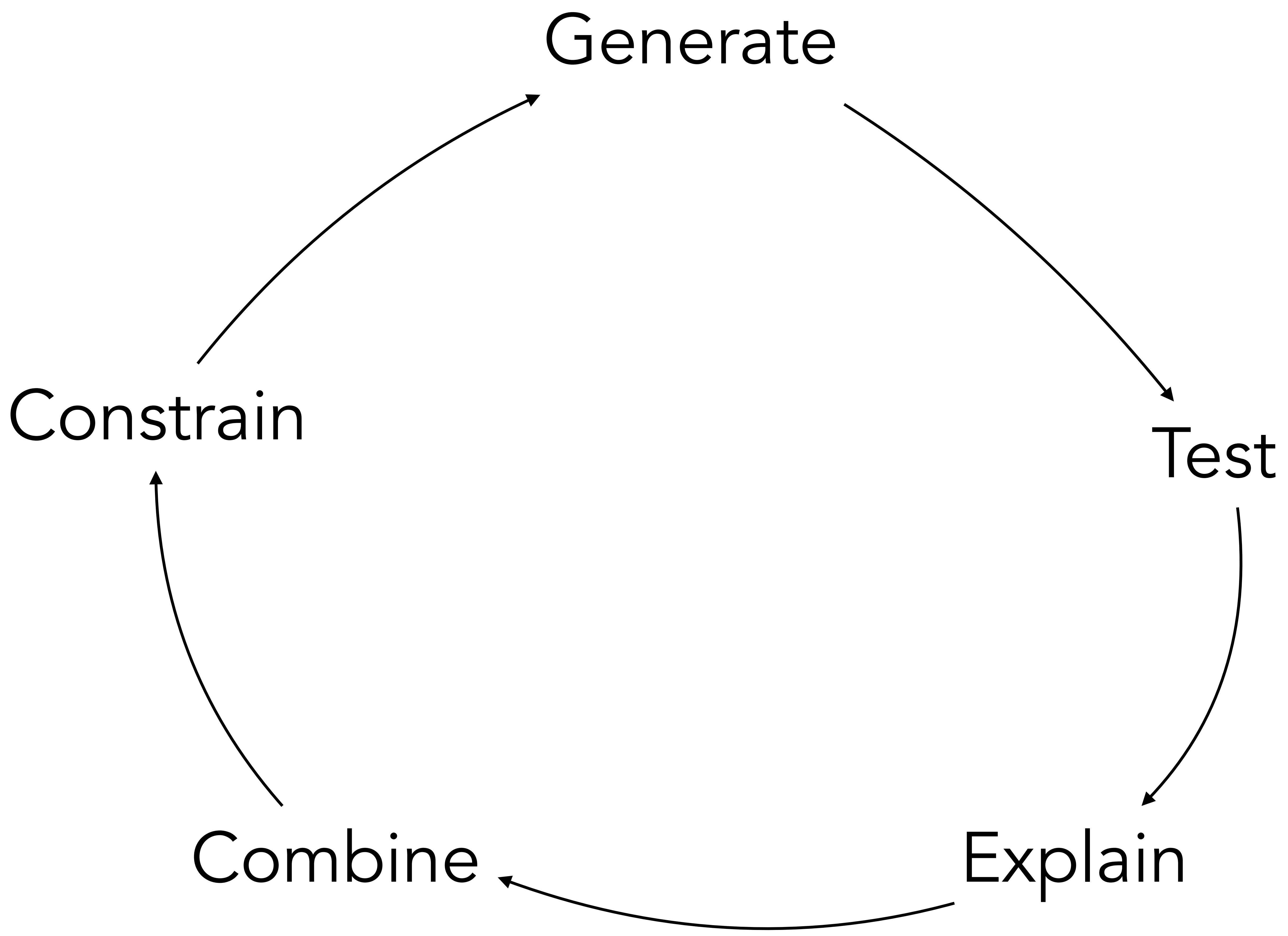
# Popper

1. Generate *programs* one-at-a-time
2. Test programs on the data and use the outcome to build *syntactic* constraints on the hypothesis space
3. Use the constraints to guide the search

# Popper



# Popper



Illustrative example

| input | output |
| --- | --- |
| laura | a |
| penelope | e |
| emma | m |
| james | e |

$$E^+ = \left\{ \begin{array}{l} \texttt{last([l,a,u,r,a],a).} \\ \texttt{last([p,e,n,e,l,o,p,e],e).} \end{array} \right\}$$

$$E^- = \left\{ \begin{array}{l} \texttt{last([e,m,m,a],m).} \\ \texttt{last([j,a,m,e,s],e).} \end{array} \right\}$$

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
h_1 = \{ \texttt{last(A,B):- head(A,B).} \} \\
h_2 = \{ \texttt{last(A,B):- head(A,B),empty(A).} \} \\
h_3 = \{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \} \\
h_4 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \} \\
h_5 = \{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

$$h_1 = \{ \texttt{last(A,B):- head(A,B).} \}$$

$$h_1 = \{\ \texttt{last(A,B):- head(A,B).}\ \}$$

| input | output | entailed |
| --- | --- | --- |
| laura | a | **no** |
| penelope | e | **no** |
| emma | m | no |
| james | e | no |

$$h_1 = \{\ \texttt{last(A,B):- head(A,B).}\ \}$$

| input | output | entailed |
|---|---|---|
| laura | a | **no** |
| penelope | e | **no** |
| emma | m | no |
| james | e | no |

H1 is too specific

# Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\} \\
h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

# Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

# Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\cancel{\mathrm{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{\mathrm{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\mathrm{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
\mathrm{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathrm{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathrm{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathrm{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathrm{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

# Prune specialisations

$$
\mathscr{H}_1 = \left\{
\begin{array}{l}
\cancel{\mathrm{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{\mathrm{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{\mathrm{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\mathrm{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathrm{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathrm{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathrm{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathrm{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

| input | output | entailed |
| --- | --- | --- |
| laura | a | yes |
| penelope | e | yes |
| emma | m | **yes** |
| james | e | no |

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

| input | output | entailed |
| --- | --- | --- |
| laura | a | yes |
| penelope | e | yes |
| emma | m | **yes** |
| james | e | no |

H4 is too general

# Prune generalisations

$$\mathcal{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \{ \texttt{last(A,B):- head(A,B).} \}} \\
\cancel{h_2 = \{ \texttt{last(A,B):- head(A,B),empty(A).} \}} \\
\cancel{h_3 = \{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \}} \\
h_4 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \} \\
h_5 = \{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

# Prune generalisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \{\texttt{last(A,B):- head(A,B).}\}} \\
\cancel{h_2 = \{\texttt{last(A,B):- head(A,B),empty(A).}\}} \\
\cancel{h_3 = \{\texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).}\}} \\
\cancel{h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}} \\
h_5 = \{\texttt{last(A,B):- reverse(A,C),head(C,B).}\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

# Prune generalisations

$$\mathcal{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\cancel{h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\}} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \cancel{\texttt{last(A,B):- tail(A,C),head(C,B).}} \\ \cancel{\texttt{last(A,B):- reverse(A,C),head(C,B).}} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

# Prune generalisations

$$
\mathscr{H}_1 = \left\{
\begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\cancel{h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\}} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \cancel{\texttt{last(A,B):- tail(A,C),head(C,B).}} \\ \cancel{\texttt{last(A,B):- reverse(A,C),head(C,B).}} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \cancel{\texttt{last(A,B):- tail(A,C),head(C,B).}} \\ \cancel{\texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).}} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

$$h_5 = \{\, \texttt{last(A,B):- reverse(A,C),head(C,B).} \,\}$$

$$h_5 = \{\, \texttt{last(A,B):- reverse(A,C),head(C,B).} \,\}$$

| input | output | entailed |
| --- | --- | --- |
| laura | a | yes |
| penelope | e | yes |
| emma | m | no |
| james | e | no |

H5 does not fail, so return it

# Popper

1. Generate (ASP)
2. Test (Prolog)
3. Constrain (ASP)

# Popper

1. Generate (ASP)
2. Test (Prolog)
3. Explain (Prolog)
4. Combine (ASP)
5. Constrain (ASP)

# Popper - why does it work?

Decomposes the learning problem

# Popper - why does it work?

Never repeats itself

# Popper - why does it work?

Reasons about syntax, not semantics

# Popper - why does it work?

Uses the right tool for the job

# Popper advantages

Optimality
Recursion
Infinite BK
Complex numerical reasoning
Predicate invention
Programs with many rules
Programs with *moderately* sized rules

# Popper disadvantages

Noisy data

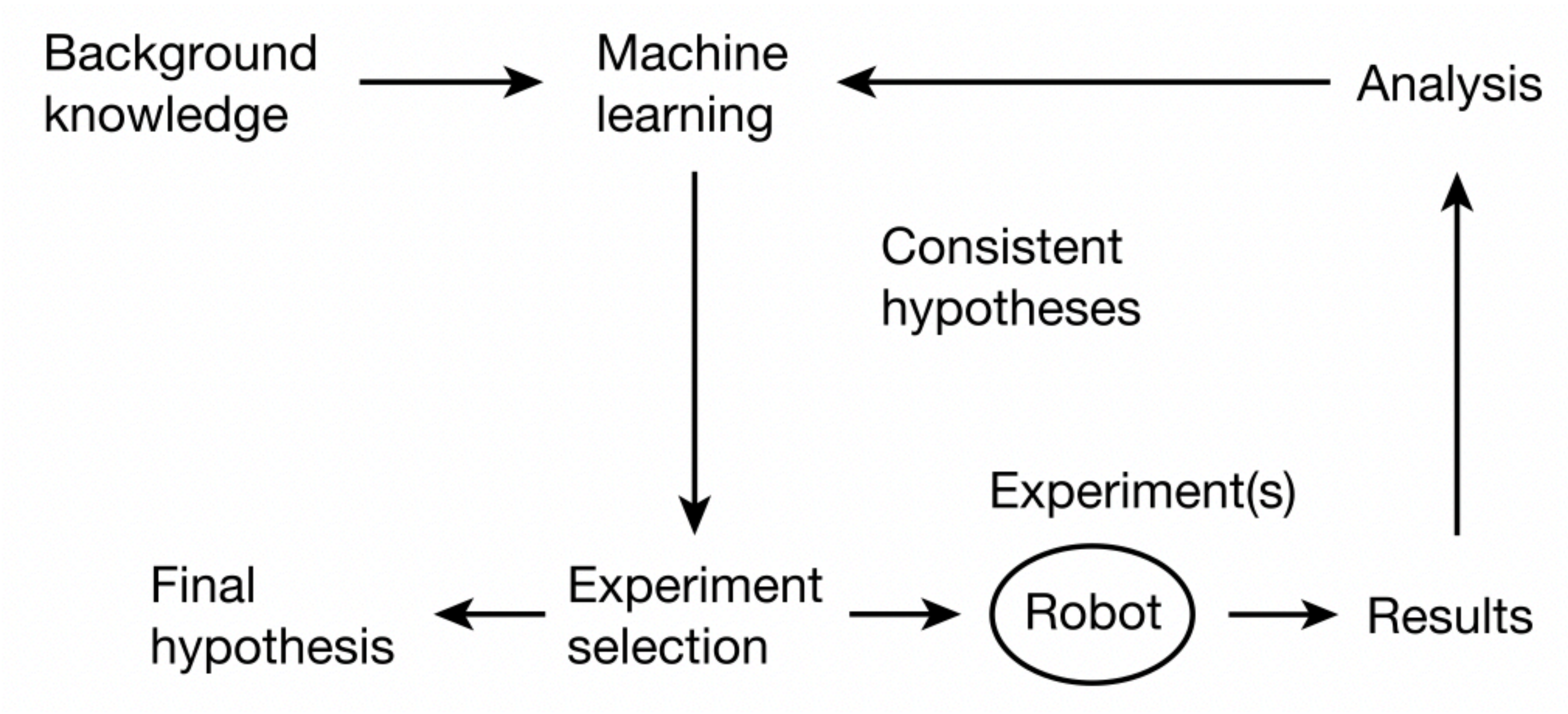
Cannot learn large rules (20+ literals)

# Part 5: Applications

# Robot scientist

*King et al. Nature, 2004*

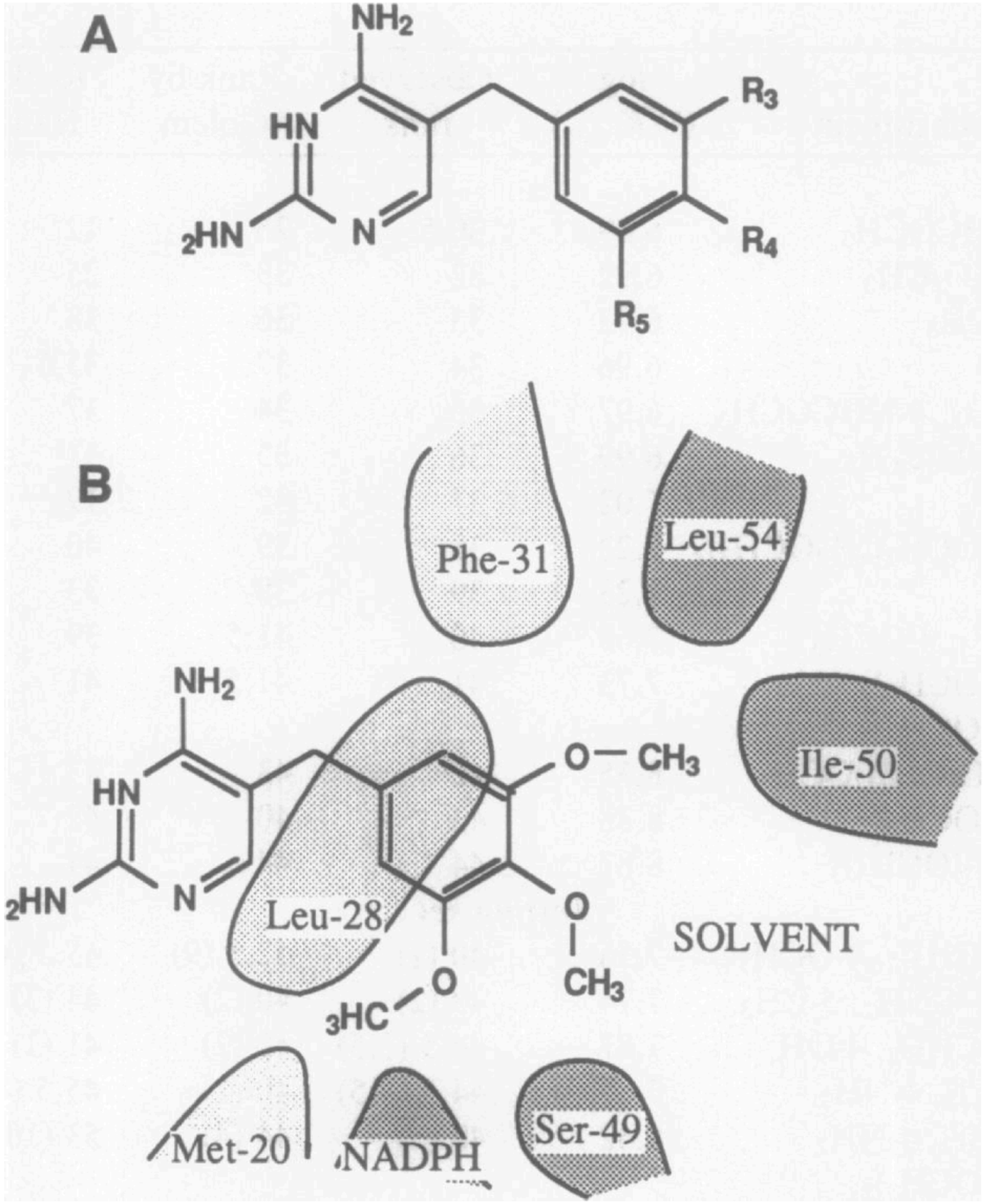
# Robot scientist



# Robot scientist

The first machine to discover new scientific knowledge independently of its human creators

# Drug design




*King et al. Proceedings of the National Academy of Sciences, 1992*

# Drug design

```
great(A,B):-
    struc(A,C,D,E),
    struc(B,F,h,h),
    h_donor(C,hdonO),
    polarisable(C,polaril),
    flex(F,G),
    flex(C,H),
    great_flex(G,H),
    great6_flex(G).
```

# Drug design

*Drug A is better than drug B if:*

*drug B has no substitutions at positions 4 and 5,*

*and drug B at position 3 has flexibility >6,*

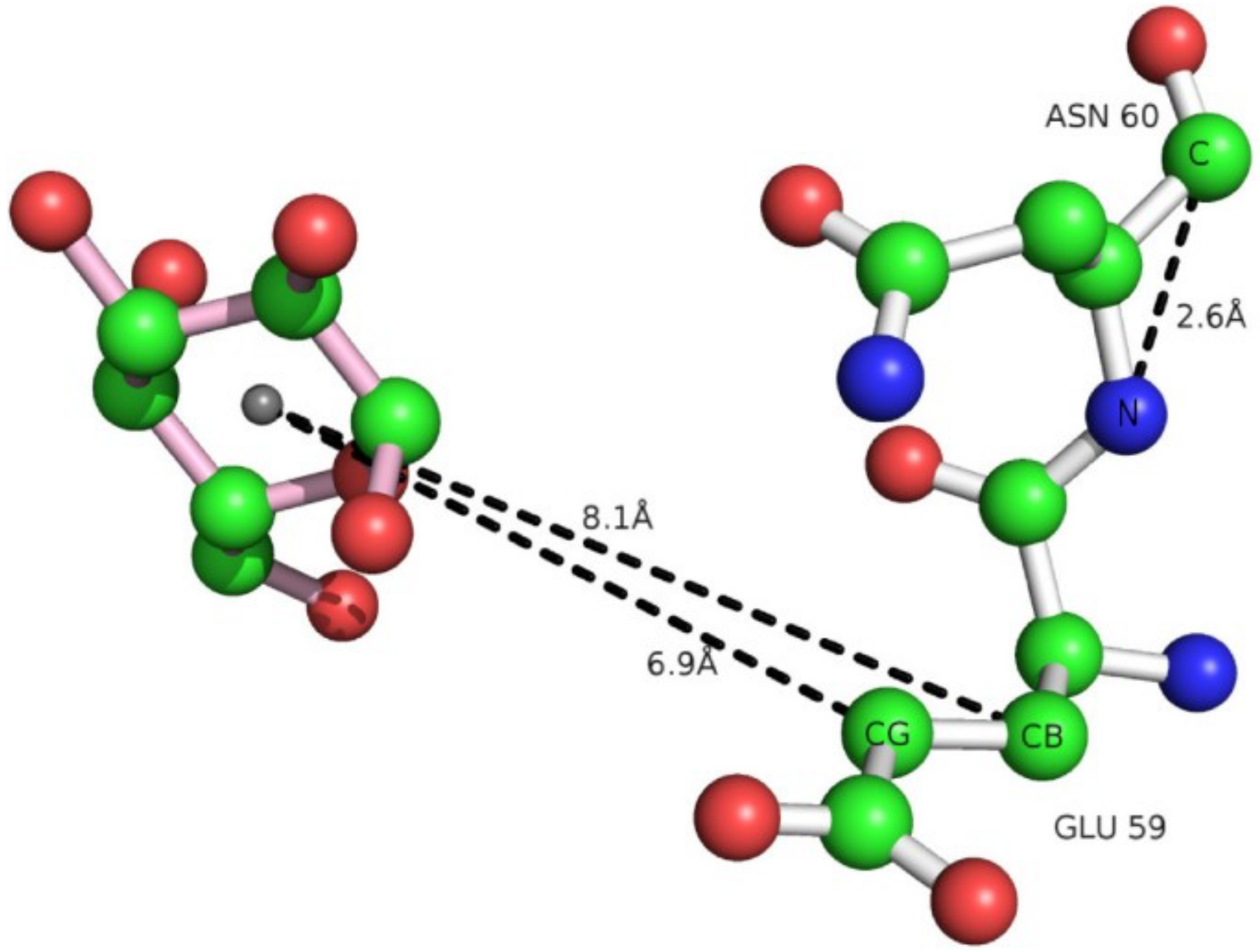
*and drug A at position 3 has polarisability = 1,*

*and drug A at position 3 has hydrogen donor = 0,*

*and drug A at position 3 is less flexible than drug B at position 3.*

**Ando, Howard Y., et al. "Discovering H-bonding rules in crystals with inductive logic programming." Molecular pharmaceutics 3.6 (2006): 665-674.**

# Scientific discovery




```
bind(A):-
  has_aminoacid(A,B,asp),
  atom_to_atom_dist(B,B,'N','OD2',4.6,0.5),
  has_amino_acid(A,C,leu),
  has_amino_acid(A,D,cys),
  atom_to_center_dist(B,'C',7.6,0.5).
```

*Santos et al. BMC Bioinformatics, 2012*

# Data curation

| task | input | output |
| --- | --- | --- |
| f | philip.larkin@sj.ox.ac.uk | Philip Larkin |

# Data curation

| task | input | output |
|---|---|---|
| f | philip.larkin@sj.ox.ac.uk | Philip Larkin |

```
f(A,B):-
  inv1(A,C),skip1(C,D),space(D,E),
  inv1(E,F),skiprest(F,B).
inv1(A,B):-
  uppercase(A,C),copyword(C,B).
```

# Data curation

| task | input | output |
|---|---|---|
| f | philip.larkin@sj.ox.ac.uk | Philip Larkin |

```
f(A,B):-
  inv1(A,C),skip1(C,D),space(D,E),
  inv1(E,F),skiprest(F,B).
inv1(A,B):-
  uppercase(A,C),copyword(C,B).
```

~10 seconds

# Data curation

| task | input | output |
|---|---|---|
| g | tony | Tony |
| | | |

# Data curation

| task | input | output |
|---|---|---|
| g | tony | Tony |
| | | |

g(A,B):-uppercase(A,C),copyword(C,B).

# Data curation

| task | input | output |
|---|---|---|
| g | tony | Tony |
| f | philip.larkin@sj.ox.ac.uk | Philip Larkin |

```
g(A,B):-uppercase(A,C),copyword(C,B).
```

# Data curation

| task | input | output |
|---|---|---|
| g | tony | Tony |
| f | philip.larkin@sj.ox.ac.uk | Philip Larkin |

```
g(A,B):-uppercase(A,C),copyword(C,B).

f(A,B):-g(A,C),skip1(C,D),space(D,E),
    g(E,F),skiprest(F,B).
```

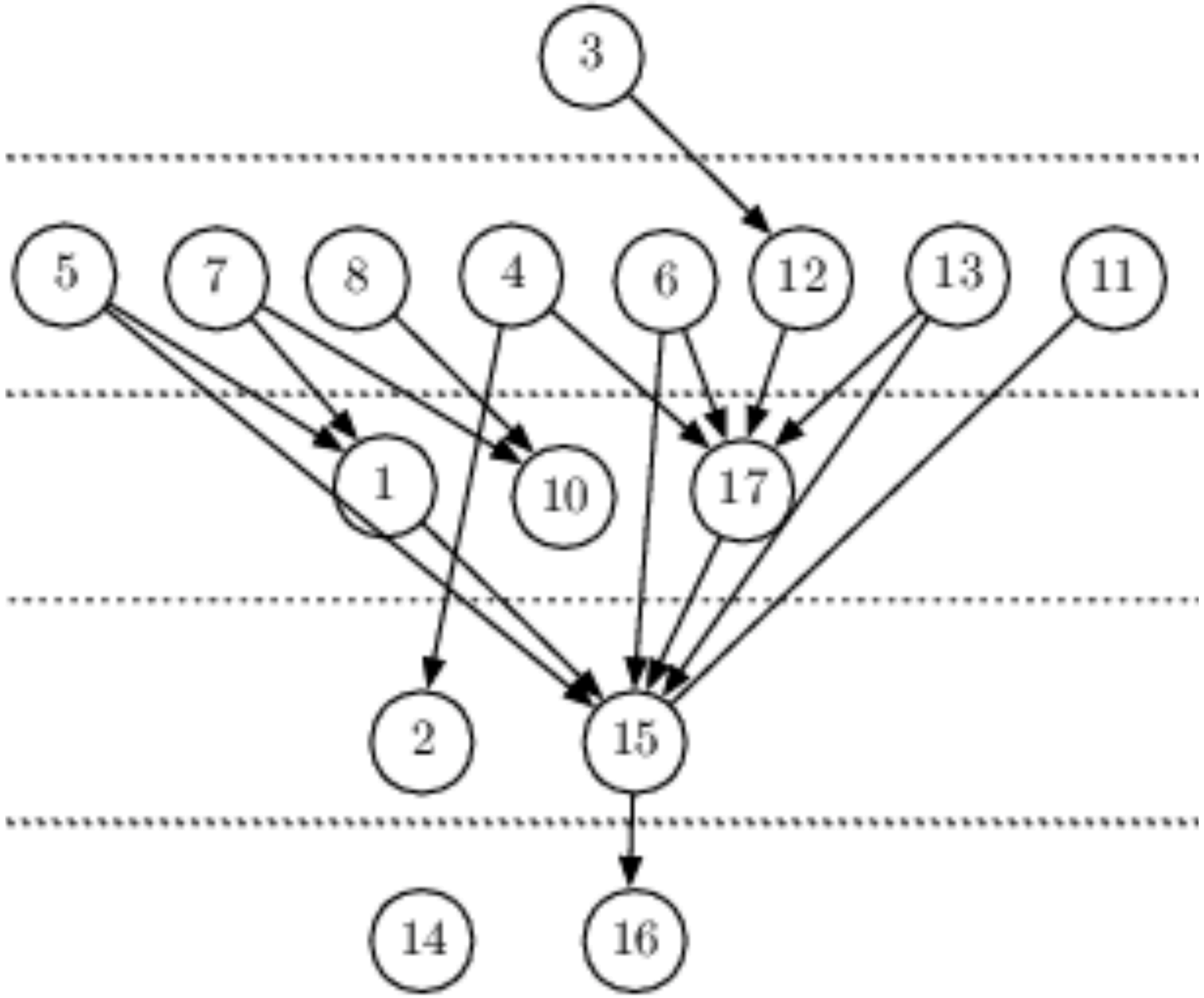
# Data curation

| task | input | output |
|---|---|---|
| g | tony | Tony |
| f | philip.larkin@sj.ox.ac.uk | Philip Larkin |

```
g(A,B):-uppercase(A,C),copyword(C,B).

f(A,B):-g(A,C),skip1(C,D),space(D,E),
    g(E,F),skiprest(F,B).
```

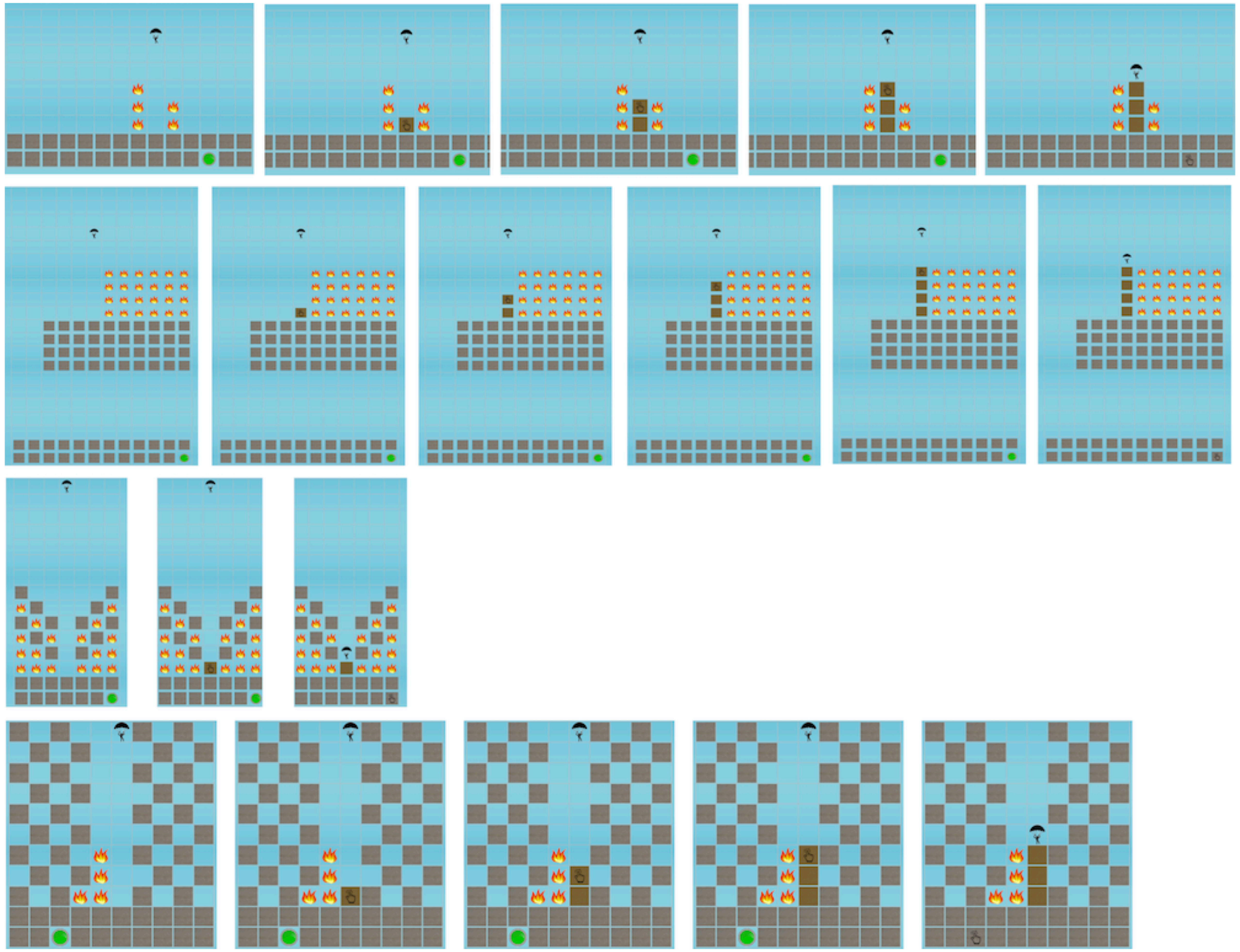
2 seconds*

# Data curation




Lin et al. ECAI2014

# Game playing



Silver et al. AAAI, 2020

# Part 6: Challenges and opportunities

# Part 6: Challenges and opportunities

Challenges

# Usability

*"while over 100 ILP systems have been constructed since 1991, less than a handful can even begin to be used meaningfully by ILP practitioners other than the original developers"*

Mach Learn
DOI 10.1007/s10994-011-5259-2

**ILP turns 20**

**Biography and future challenges**

**Stephen Muggleton · Luc De Raedt · David Poole · Ivan Bratko · Peter Flach · Katsumi Inoue · Ashwin Srinivasan**

Received: 8 May 2011 / Accepted: 29 June 2011

# Usability

Many systems are prototypes and are not maintained

Systems are inconsistent among themselves (w.r.t. language bias)

Only the developers know how to use the systems properly

# Usability

*"You often need a PhD in ILP to use any of the tools"*

# What do we need?

Better engineered tools

# What do we need?

Better maintained tools

# What do we need?

Standardisation

# What do we need?

Standardisation

**Dimacs**

DIMACS format is a standard interface to SAT solvers.

# Language bias

The biggest deterrent from ILP

# Language bias

**weak bias**: too slow to be usable

**strong bias**: fast learning but might exclude the target program

# Language bias what should we do?

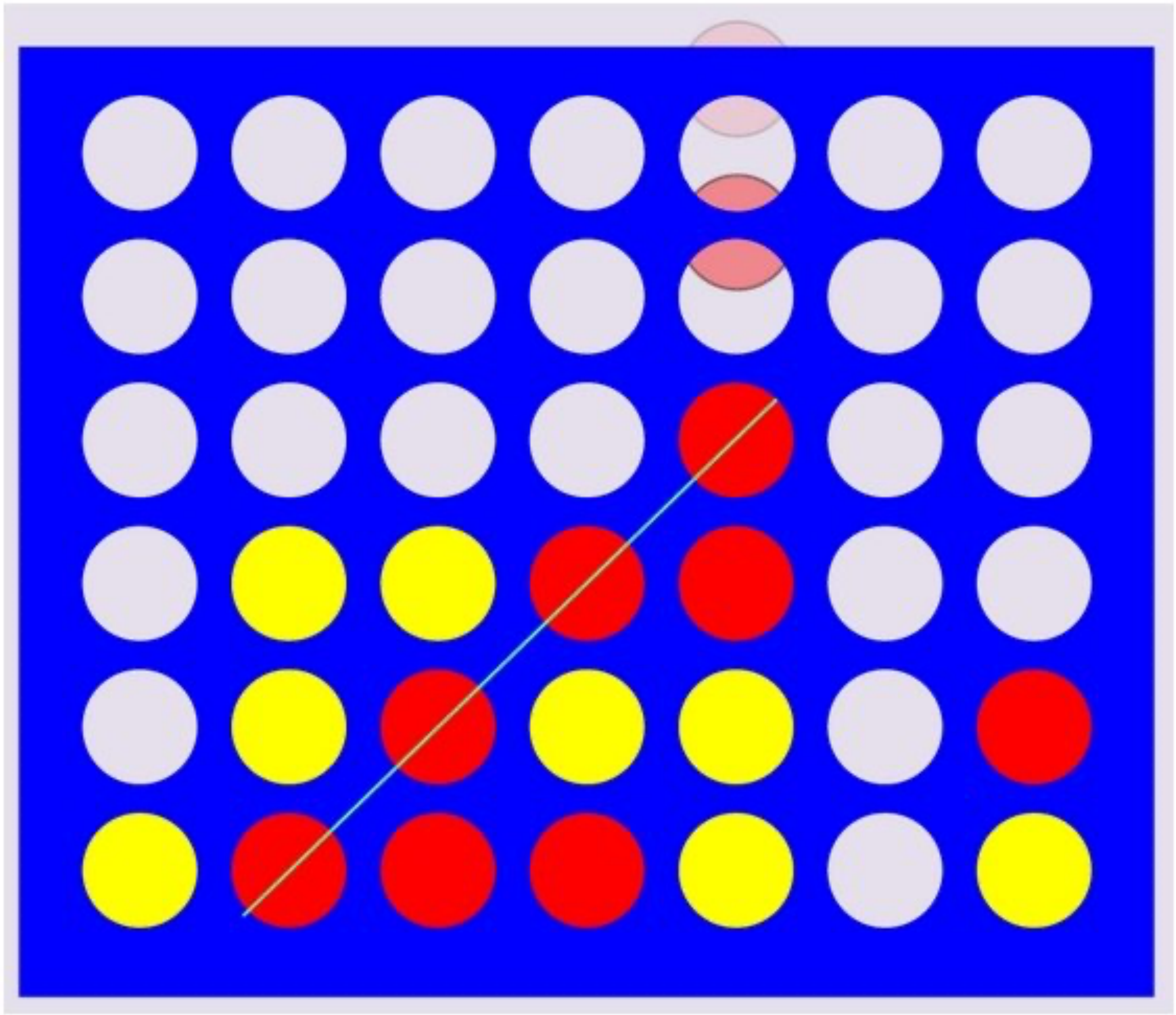
Automatically identify an appropriate language bias

A vastly under-researched area of ILP!

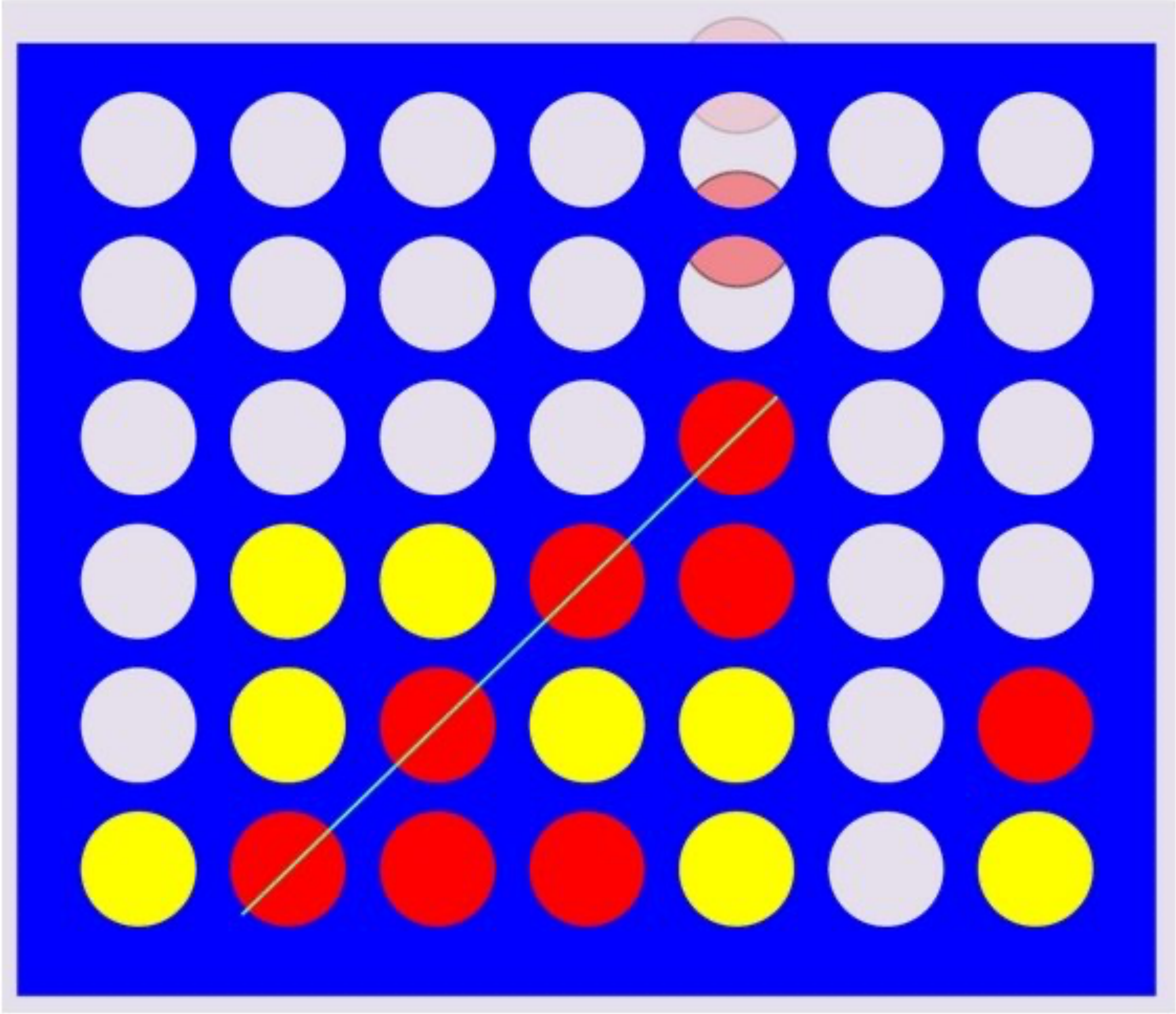
# Predicate invention

Predicate invention is central for complex tasks

# Predicate invention



# Predicate invention



# Predicate invention

**Challenge:** what is a useful predicate to invent?

**Recent progress:** find reoccurring subprograms from available solutions to similar problems

# Predicate invention

Discover useful and reusable abstractions before and during learning

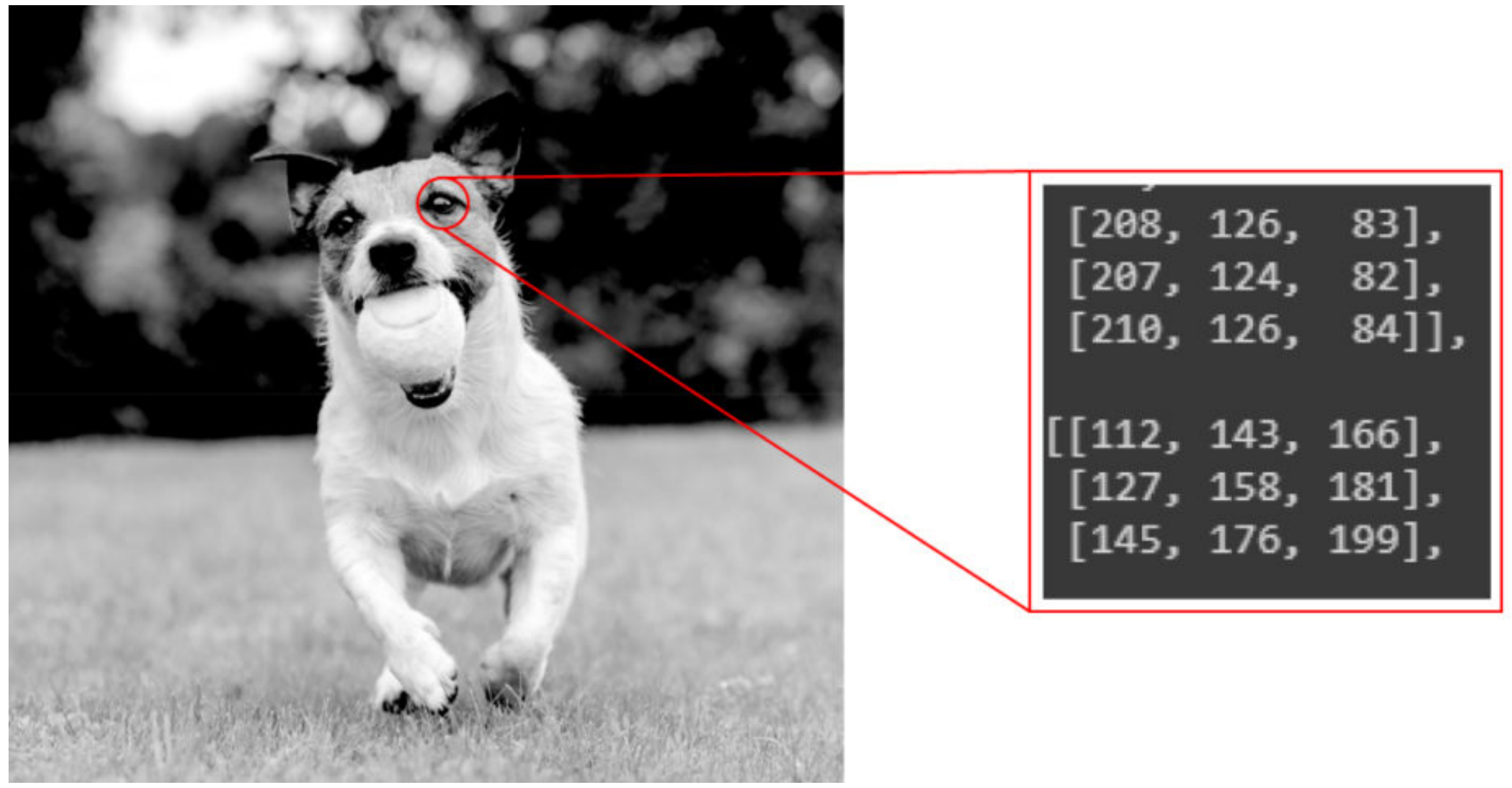
*Dumancic et al, "Knowledge Refactoring for Inductive Program Synthesis", AAAI 2021*

# Learning from raw data

# Learning from raw data

ILP assumes data to be structured, but plenty of data available in raw formats

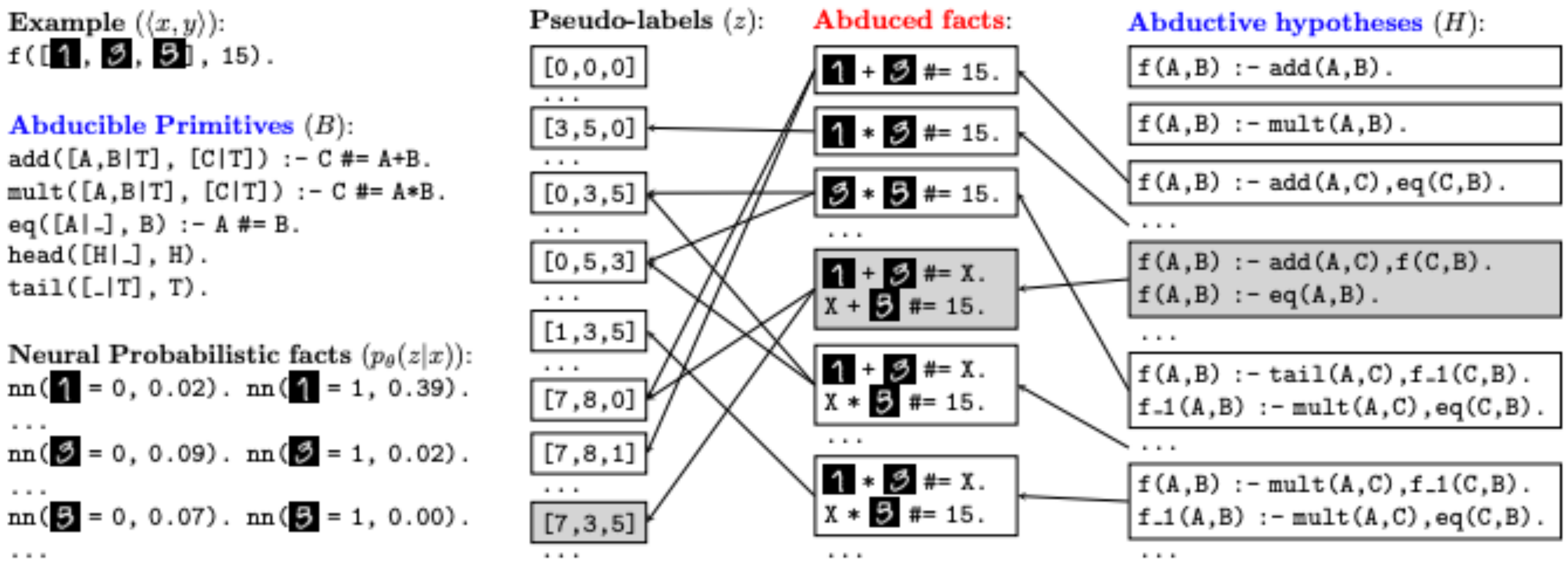
```
connected(A,B):- edge(A,B).
connected(A,B):- edge(A,C),connected(C,B).
```




Not every problem is representable in symbolic form

# Learning from raw data

Some progress in recent years




Evans et al, "Making sense of sensory input", 2021

Dai & Muggleton, "Abductive Knowledge Induction From Raw Data", 2021

# Learning from raw data

What should we aim for?

Techniques that treat learning to perceive and learning a program as integrated components

# Learning with uncertain data

ILP assumes that BK is correct, but real world is often uncertain

Do birds fly?

How about this bird?

# Learning with uncertain data

Various probabilistic logics have been developed since '90s

**Challenge:** probabilistic logic programs are not efficient

```
0.8 :: weather(sunny).
```

Query: what is the probability that a particular statement is true?

*Problog, Markov logic networks, Probabilistic Soft Logic, …*

# Learning with uncertain data

What should we aim for?

Handling uncertainties in BK, especially in lifelong learning

# Relevance of BK

BK is treated as a monolithically construct

Only a tiny percentage of BK is relevant for a task

How do we discover a relevant part of BK?

# Scalability?

*“ILP does not scale to real-world problems”*

# What does scalability mean?

# What does scalability mean?

Many rules?

Large rules?

Large numbers of examples?

Large amounts of BK?

# What does scalability mean?

~~Many rules?~~

Almost all ILP systems can learn programs with 100s of rules

Large rules?

Large numbers of examples?

Large amounts of BK?

# What does scalability mean?

~~Many rules?~~

~~Large rules?~~

Large numbers of examples?

Large amounts of BK?

Aleph can learn programs with rules with 100s of literals

# What does scalability mean?

~~Many rules?~~

~~Large rules?~~

~~Large numbers of examples?~~

Large amounts of BK?

QuickFOIL can learn programs from 2+ million examples

# What does scalability mean?

~~Many rules?~~

~~Large rules?~~

~~Large numbers of examples?~~

~~Large amounts of BK?~~

QuickFOIL can learn programs from 200 million background facts

# What is not scalable?

Learning programs with long chains of reasoning

# Part 6: Challenges and opportunities

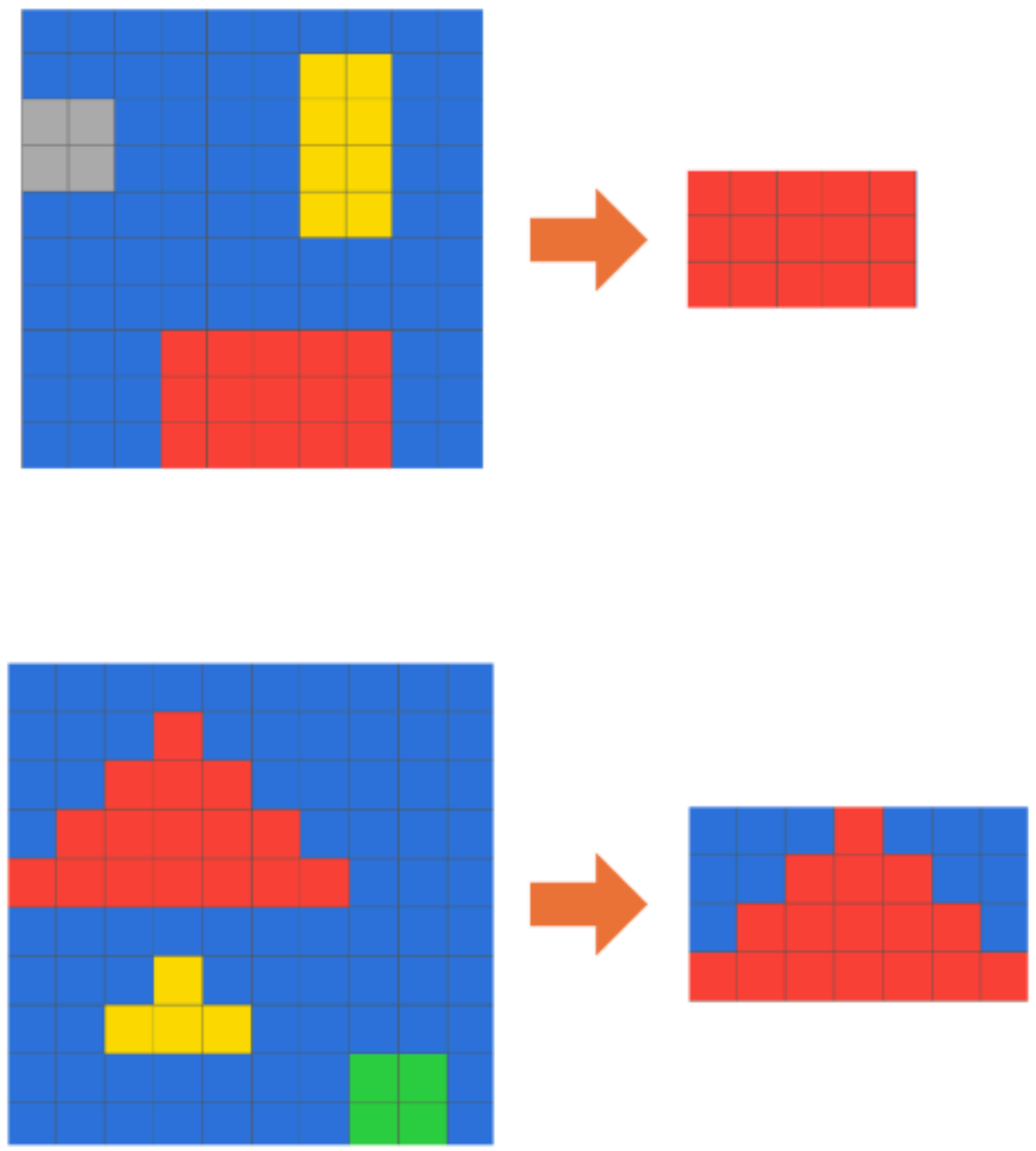
Grand challenges

# Challenging problems

Push ILP beyond what is currently possible

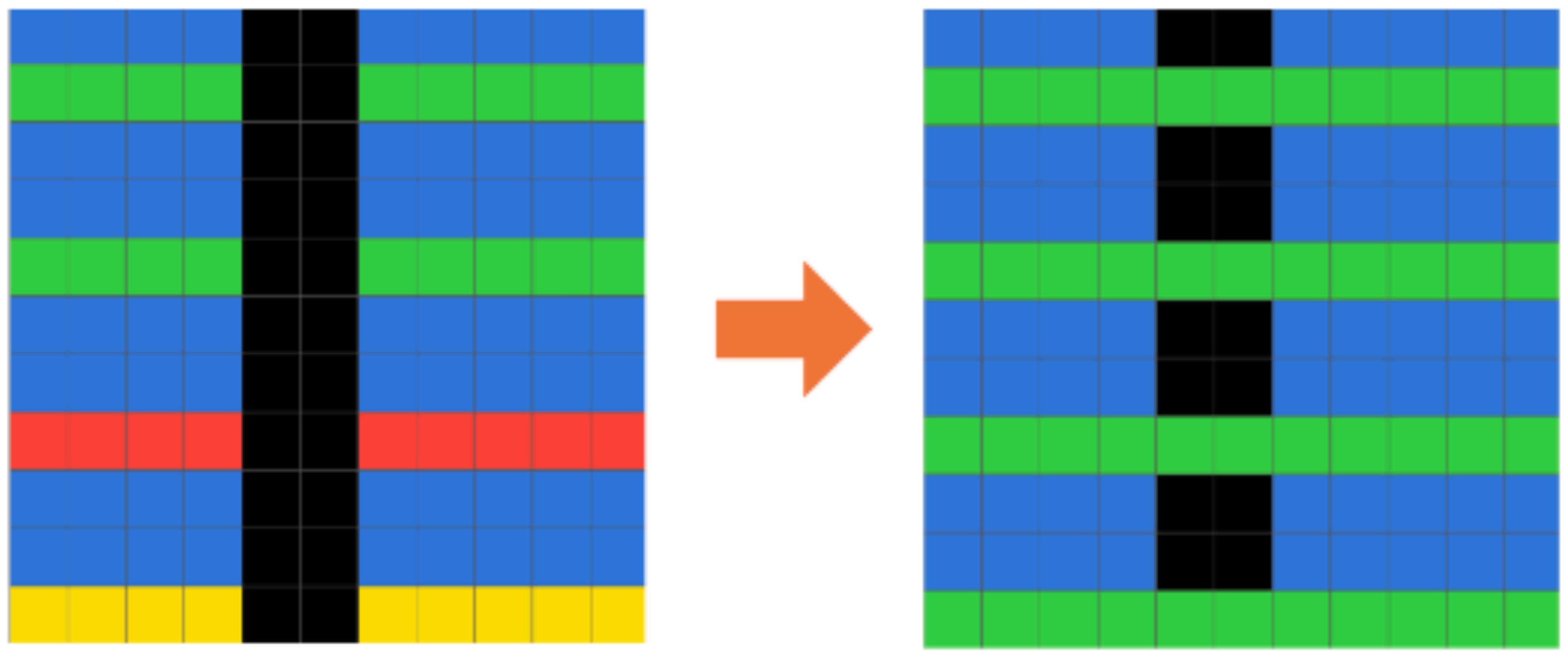
Require some of the outlined challenges to be solved

# Abstraction and Reasoning Corpus



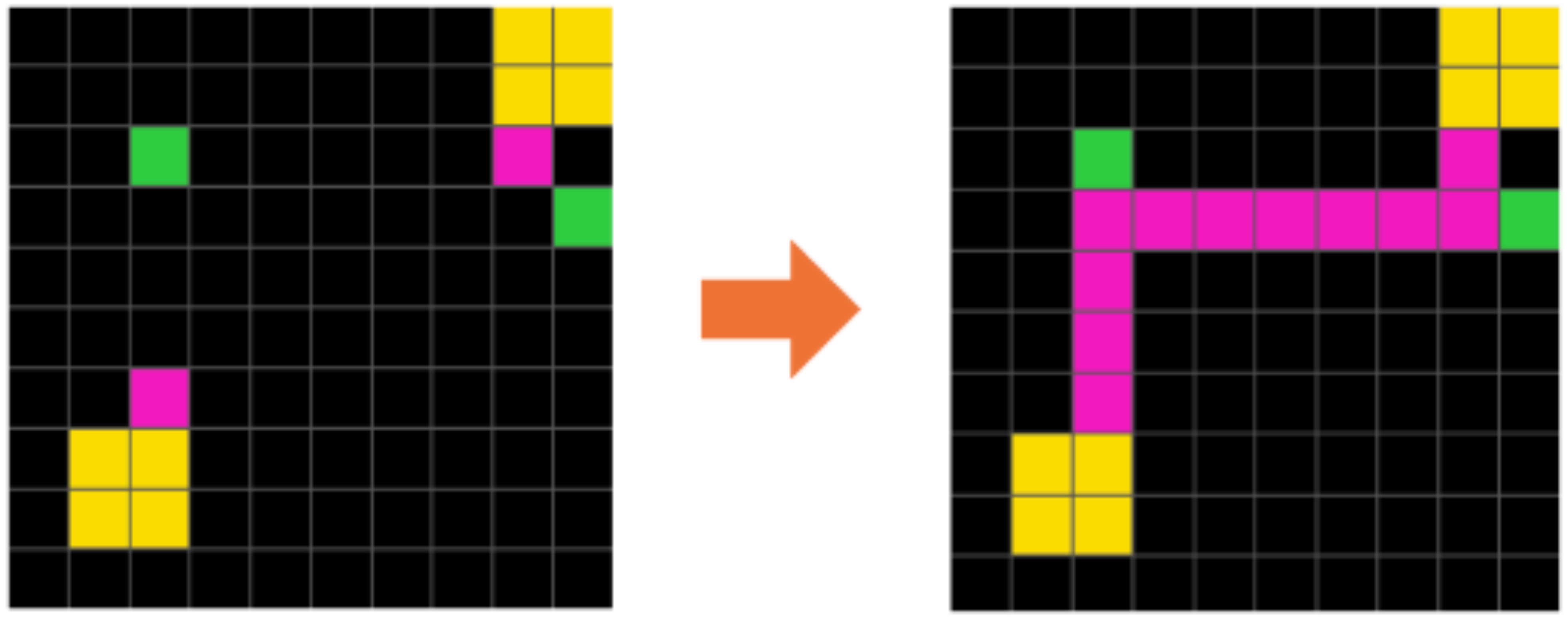
Find the largest object and copy it

# Abstraction and Reasoning Corpus



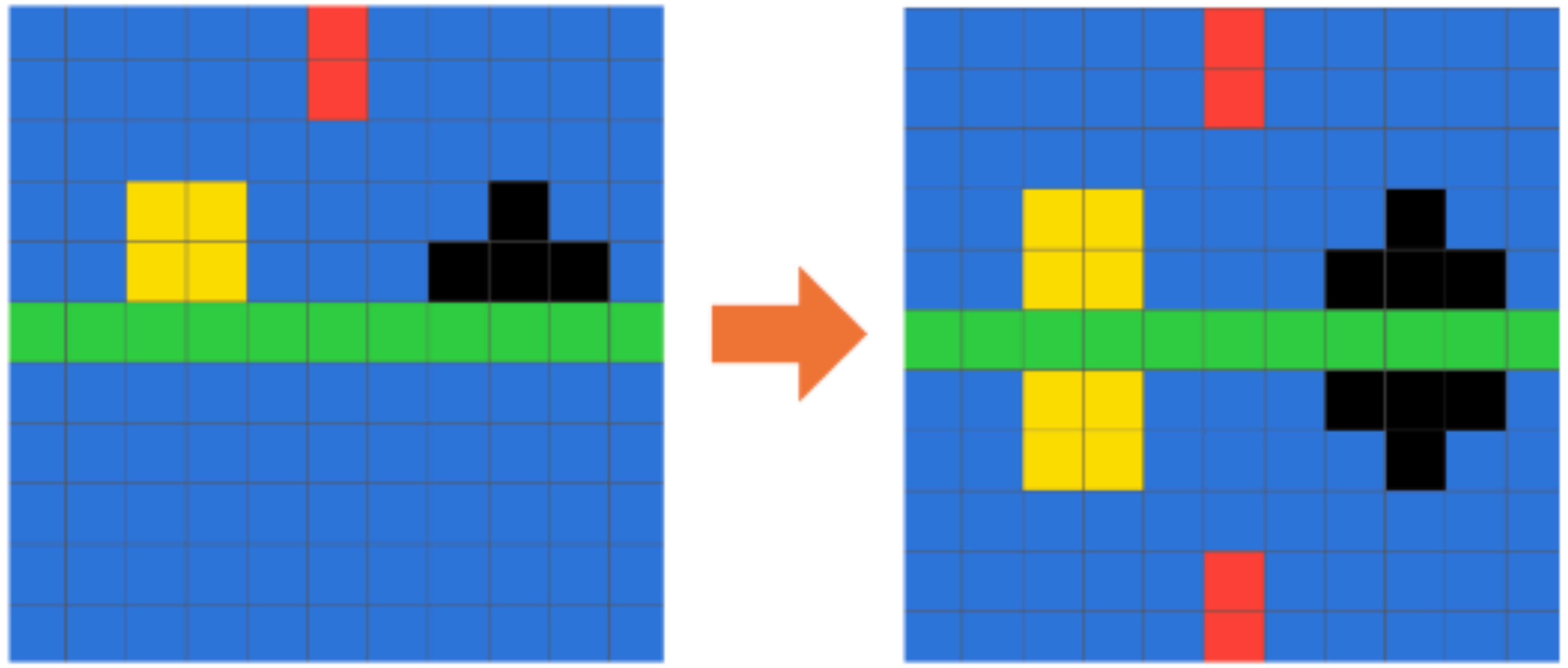
Identify the lines, complete them,
and paint with the most frequent color

# Abstraction and Reasoning Corpus



Connect yellow boxes through purple pixels,
you are allowed to turn only at the green box

# Abstraction and Reasoning Corpus



Mirror over the green line

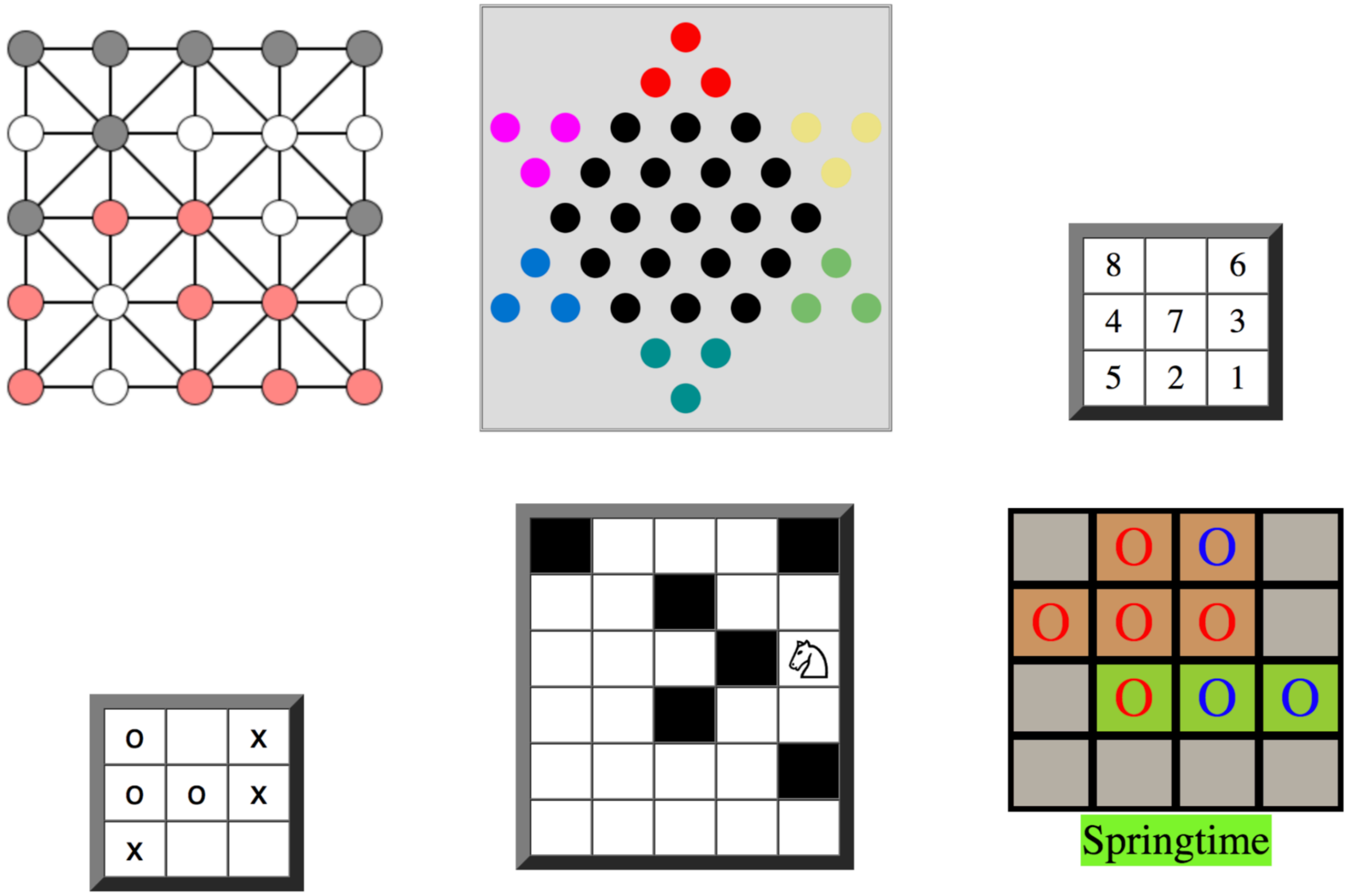
# Abstraction and Reasoning Corpus

"Simple" high-level solutions, but requires to bridge the gap from pixels

Only a few examples of every task

Solutions are programs

# Inductive general game playing




Can we learn the rules (semantics) of games from observations?

# Rock, paper, scissors

```
next_score(P,N):-
    true_score(P,N),
    draws(P).
next_score(P,N):-
    true_score(P,N),
    loses(P).
next_score(P,N2):-
    true_score(P,N1),
    succ(N2,N1),
    wins(P).
```

```
draws(P):-
    does(P,A),
    does(Q,A),
    distinct(P,Q).
loses(P):-
    does(P,A1),
    does(Q,A2),
    distinct(P,Q),
    beats(A2,A1).
wins(P):-
    does(P,A1),
    does(Q,A2),
    distinct(P,Q),
    beats(A1,A2).
```

*draws/1, loses/1, wins/1 are not provided as BK!

# Why is IGGP interesting?

Many diverse games

Not hand-crafted by a system designer

Cannot predefine the perfect language bias

Need to learn perfect rules!

# IGGP is hard

SOTA performance is learning perfect rules for 40% of the games

# What is needed for IGGP?

Negation

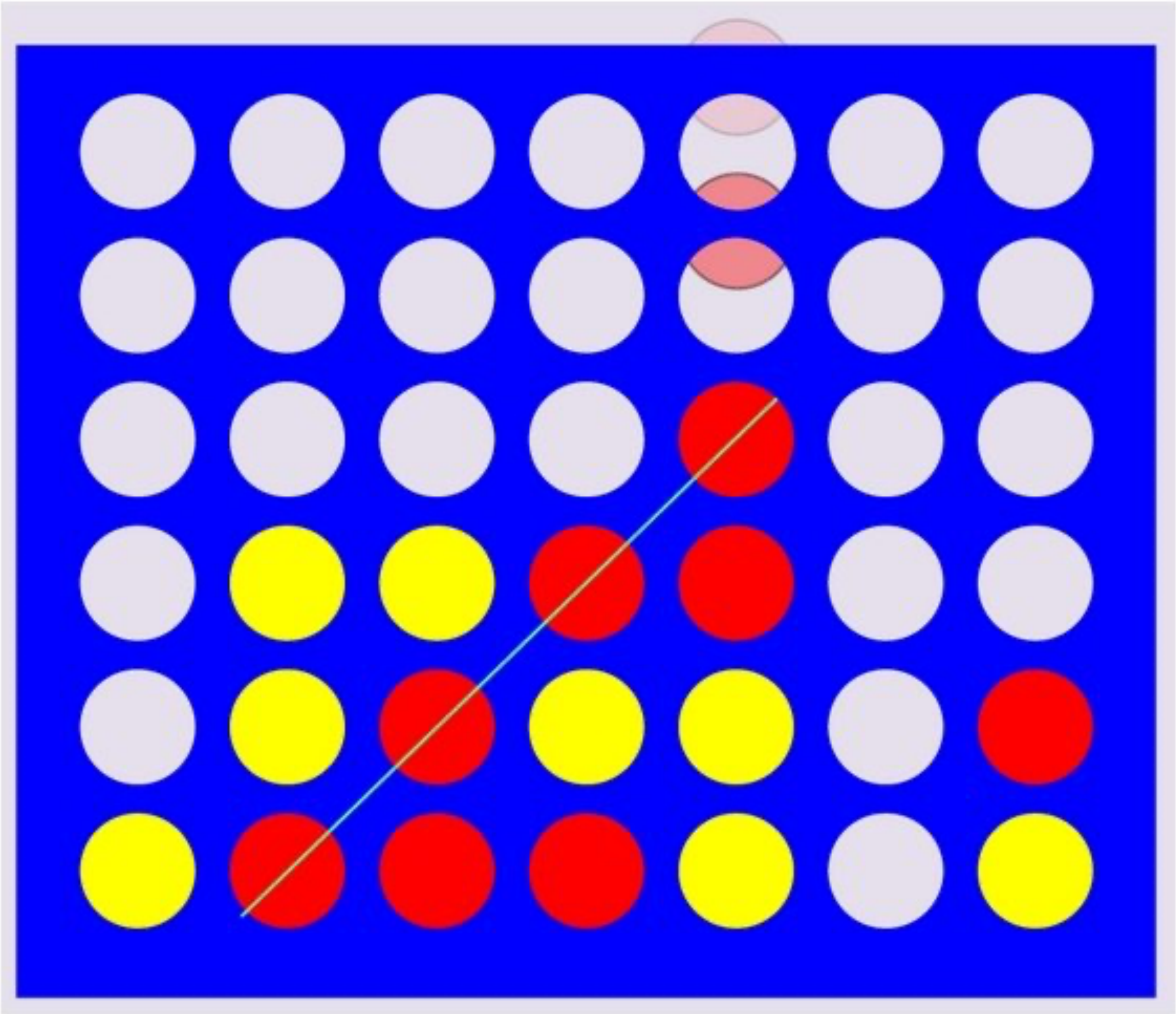
Predicate invention

Very large rules

Not overfitting

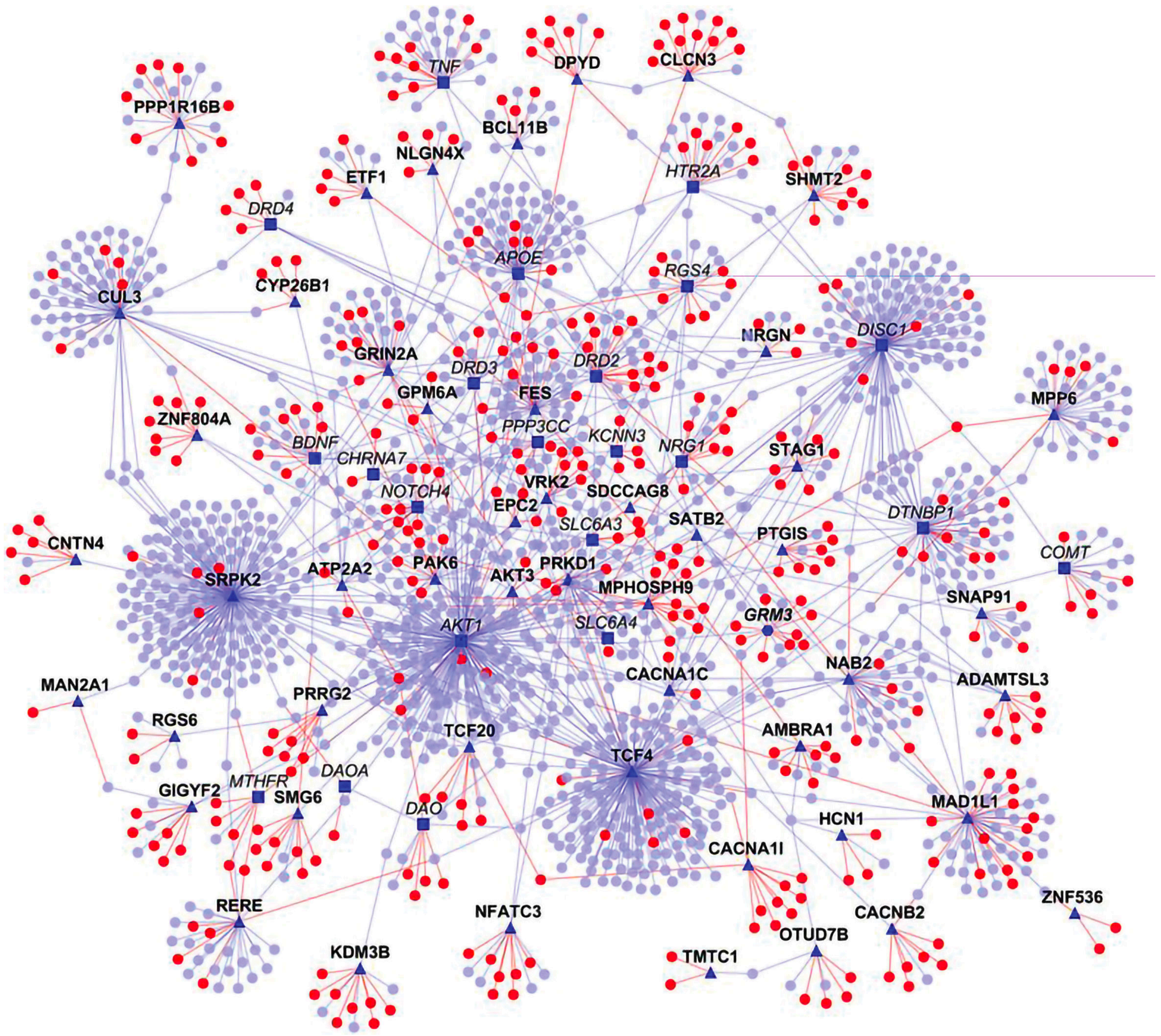
https://github.com/andrewcropper/iggp

# IGGP is hard



Need to invent the concept of a *line* and reason about it

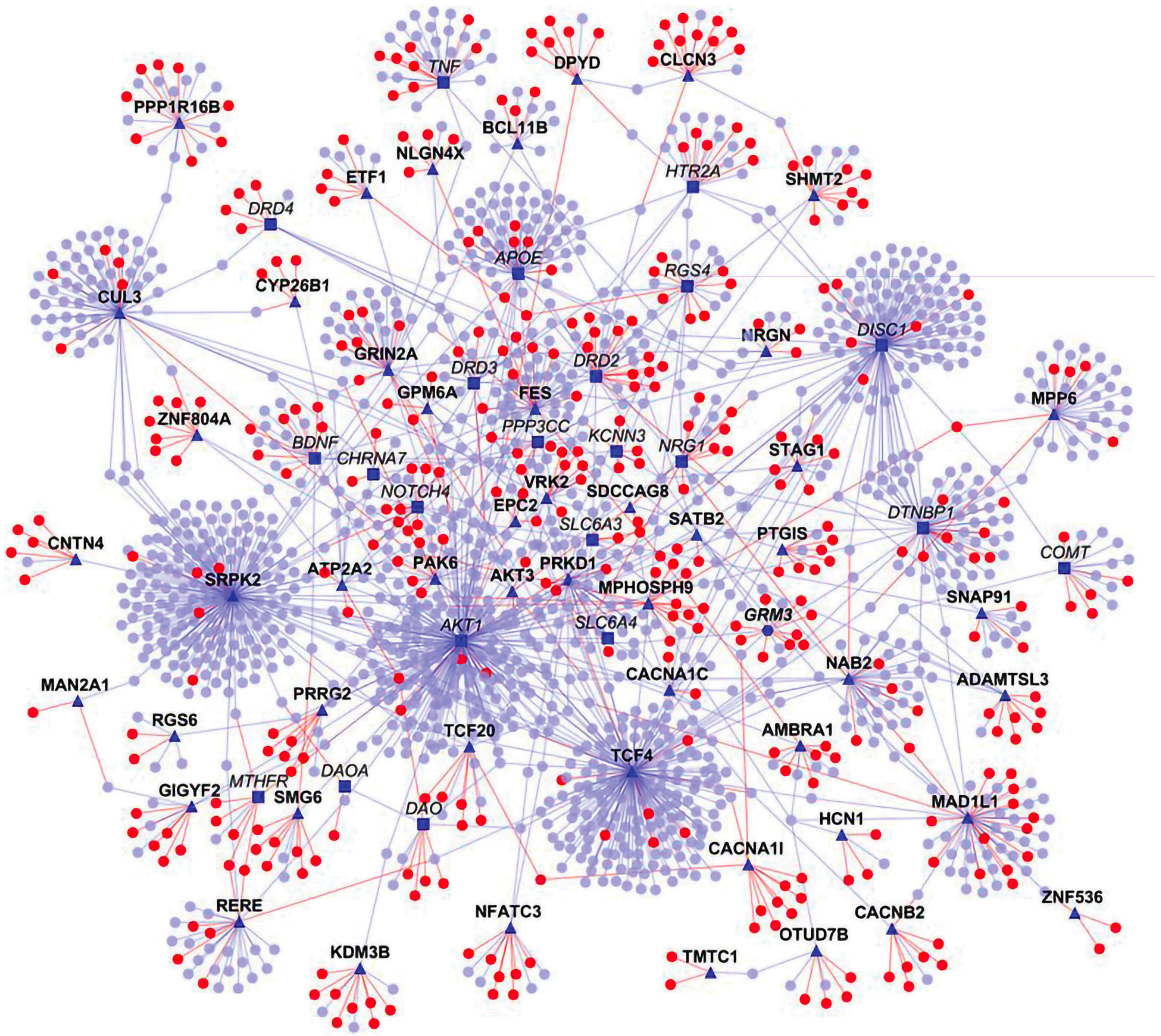
# Large biological knowledge bases




Vast amounts of biological data

Protein interaction networks
Gene expressions
Molecular functional interactions
...

# Large biological knowledge bases




- Many of them are relational
- Require discovering rules about interactions
- Need to be explainable

- Early successes of ILP

# Visual question answering

# Visual question answering

- Need to understand an image
- Turn question into a query
- Integrate common sense knowledge

# Scientific discovery

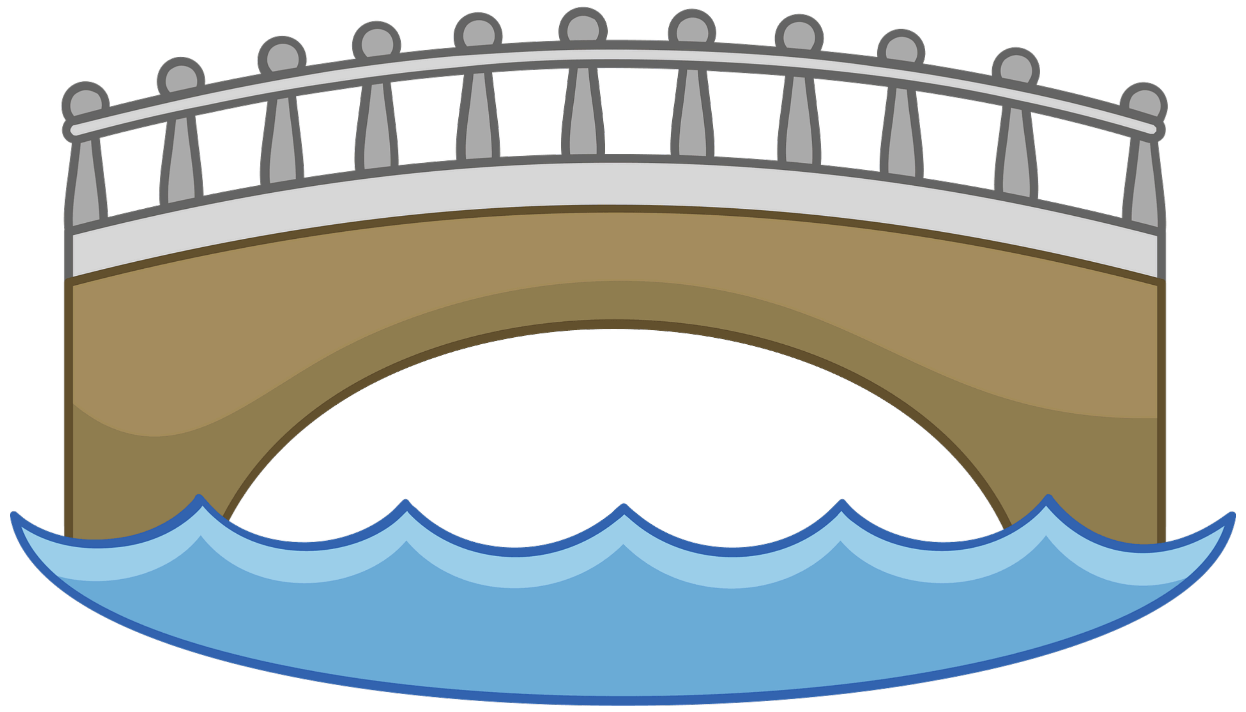

- Learn with prior knowledge
- Hypotheses need to be interpretable
- Test and refine
- Experiments should verifiable

# Part 6: Challenges and opportunities

Opportunities

# Loads of opportunities

Machine learning



ILP

Constraint solving

Knowledge representation

Databases

# Constraint solving community

Recent approaches frame the ILP problem as a constraint problem:

- ASPAL
- ATOM
- ILASP
- Popper
- HEXIL
- Apperception

# Constraint solving community

Recent approaches frame the ILP problem as a constraint problem:

- ASPAL
- ATOM
- ILASP
- Popper
- HEXIL
- Apperception

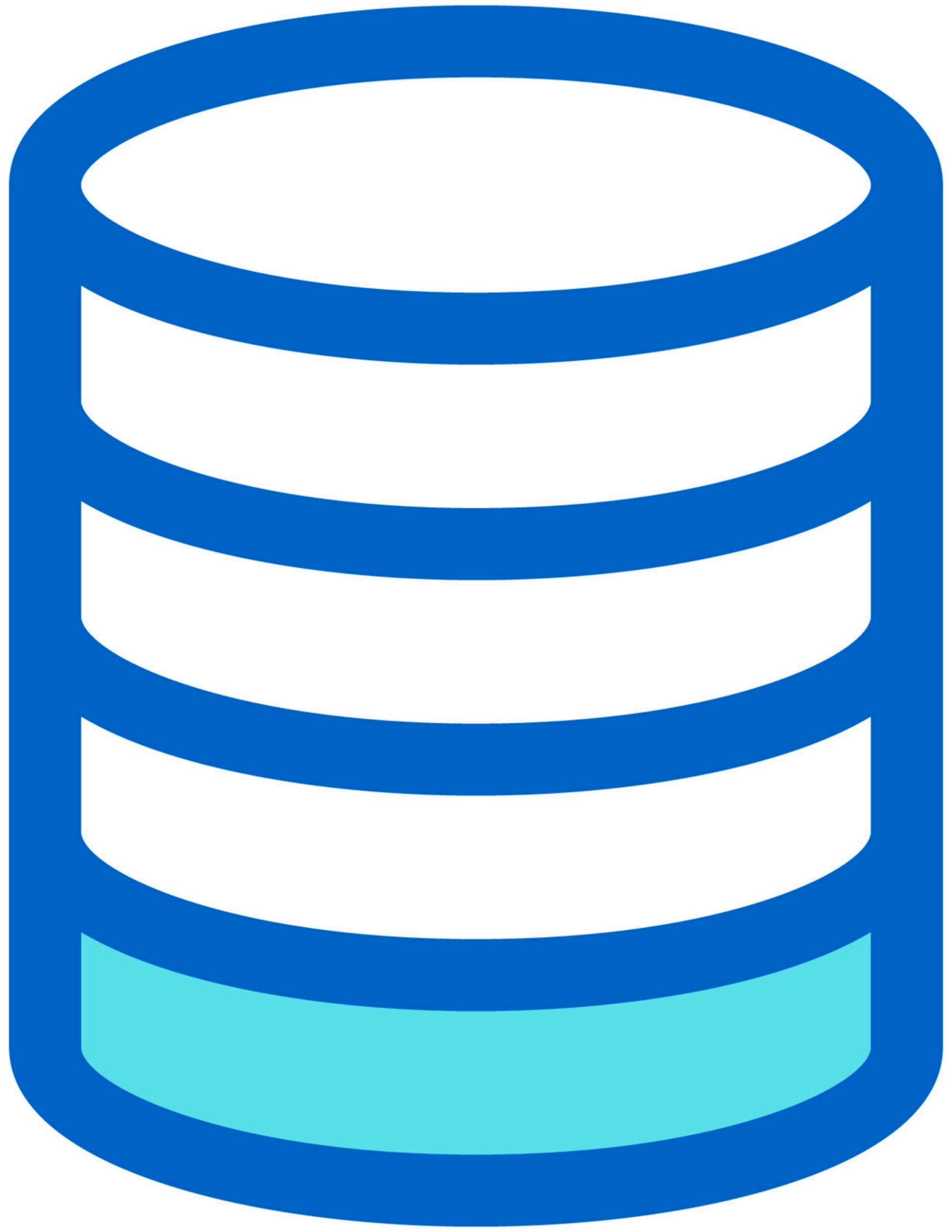
*All (except ATOM) use ASP*

# Constraint solving opportunities

Can we model these problems better?

Are other solving approaches better (SAT,SMT,CP)?

# Database/Datalog community



For many ILP applications, Datalog suffices

Databases can help scale ILP significantly (QuickFOIL)

# Database/Datalog opportunities

Can we use ideas from databases to ILP scale to larger amounts of BK?

Can we use ILP for query synthesis in Datalog/SQL systems?

# Knowledge representation community

ILP solves the knowledge acquisition problem

# Knowledge representation community

How can we assemble large (consistent) knowledge bases with ILP?

Meta-reasoning: can what we know help us to learn new things faster?

# Wrap-up

# Wrap up

Inductive logic programming: ML + logic

Attractive features: Small data, interpretable, relational

Attractive capabilities: Recursion, optimality, predicate invention

Lots of opportunities for interaction with other communities

# References

Inductive Logic Programming. S. Muggleton. New Generation Computing 1991.

Inductive logic programming at 30: a new introduction A. Cropper and S. Dumančić, JAIR 2022.