02246 - Model Checking

$$M \models \Phi ?$$

Lecture 04 - Model Checking Algorithms

Technical University of Denmark

System

refine/implement

Requirements

abstract model

$M \models \Phi$?

some logic here

Finite-state transition System

some smart algorithm here

yes :)

may be :|

no :(

Key points of this lecture

Global model checking algorithm, recursive on state sub-formulas (i.e. bottom-up on the AST).

Satisfaction set characterisations of CTL formulas in existential normal form.

Dedicated algorithms for CTL formulas in existential normal form, which exploit the satisfaction set characterisations.

The complexity of CTL model checking is polynomial in the size of the transition system and the CTL formula.

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Reading material

Section 6.4 of “Principles of Model Checking”

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

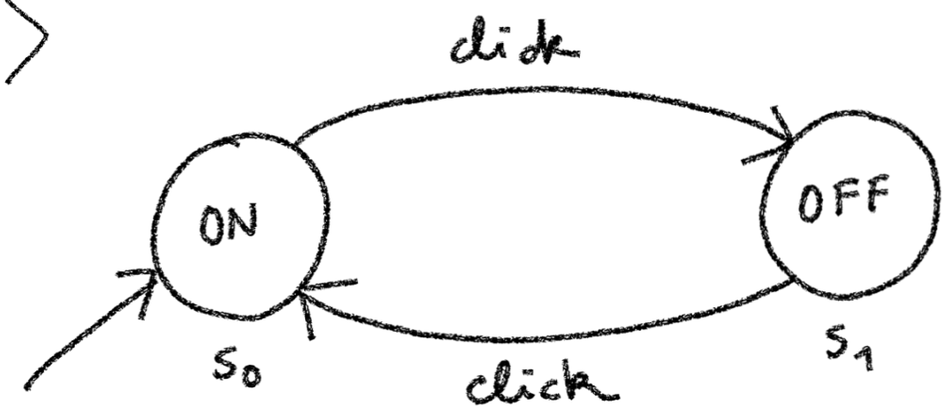
Transition Systems

Def. A transition system is a tuple

$$\langle S, A, \rightarrow, L, AP, I \rangle$$

such that

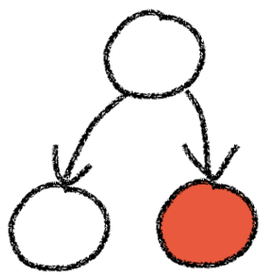
- S is a set of states
- A is a set of "Actions"
- $\rightarrow \subseteq S \times A \times S$ is a set of transitions
- $L : S \longrightarrow 2^{AP}$ is a labelling function
- AP is a finite state of "Atomic Propositions"
- $I \subseteq S$ is the set of initial states



Intuition of “next” operator

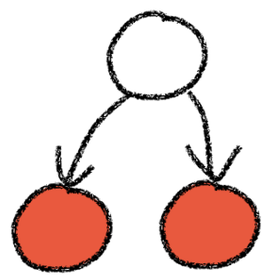
$\exists \bigcirc \phi$

"in the next state ϕ holds"



$\forall \bigcirc \phi$

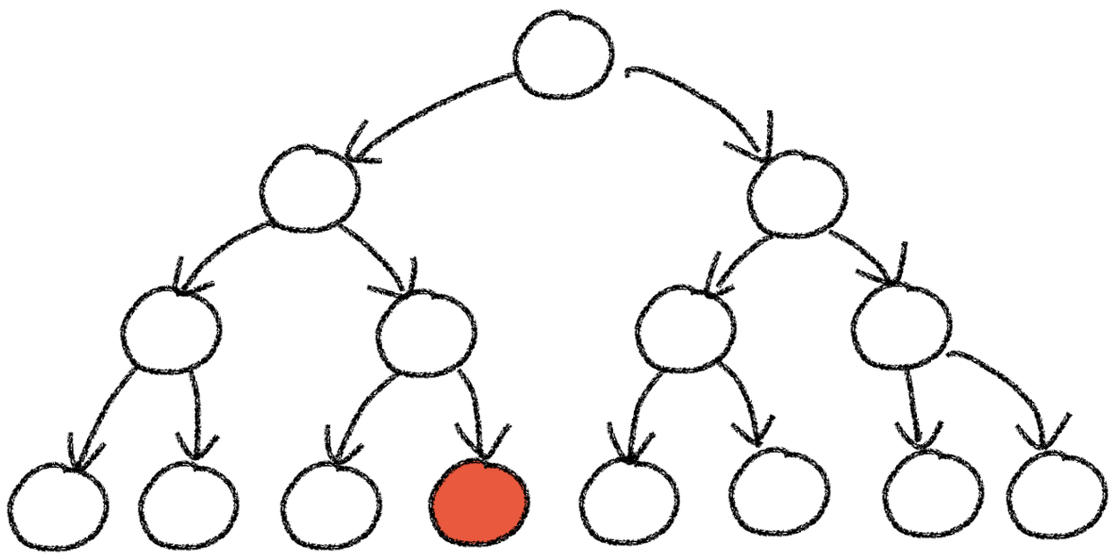
"in all next states ϕ holds"



Intuition of “eventually” and “always” operators

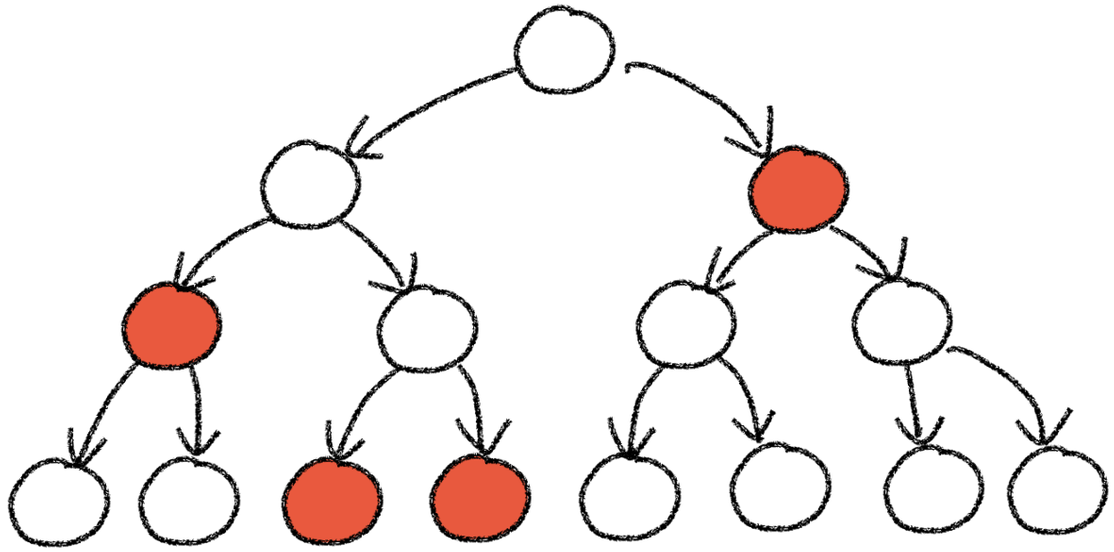
$\exists \Diamond \phi$

"ϕ holds potentially"



$\forall \Diamond \phi$

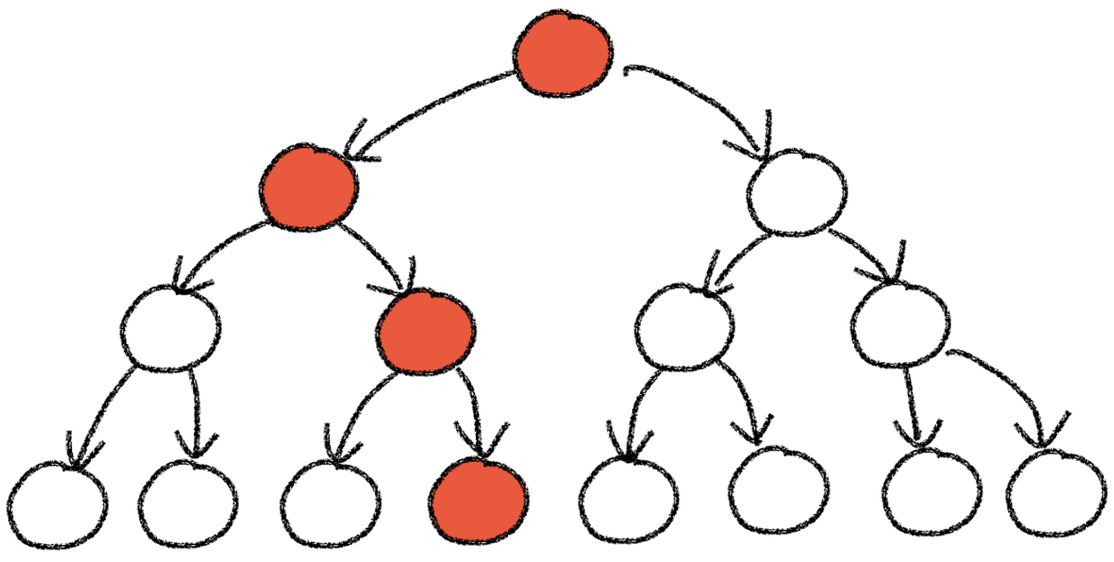
"ϕ is inevitable"



Focus of today

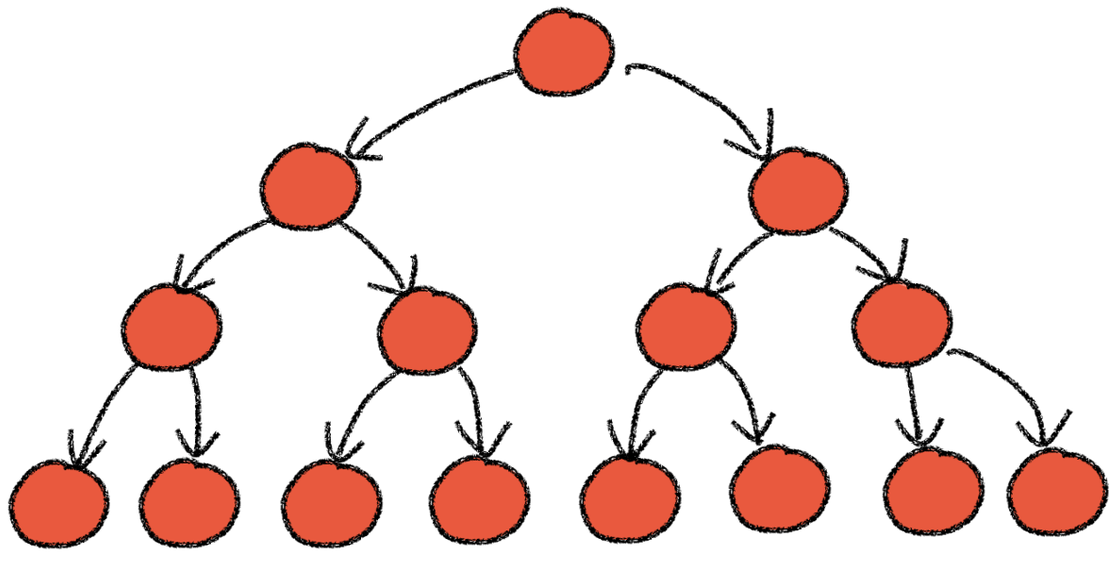
$\exists \Box \phi$

"potentially always ϕ"



$\forall \Box \phi$

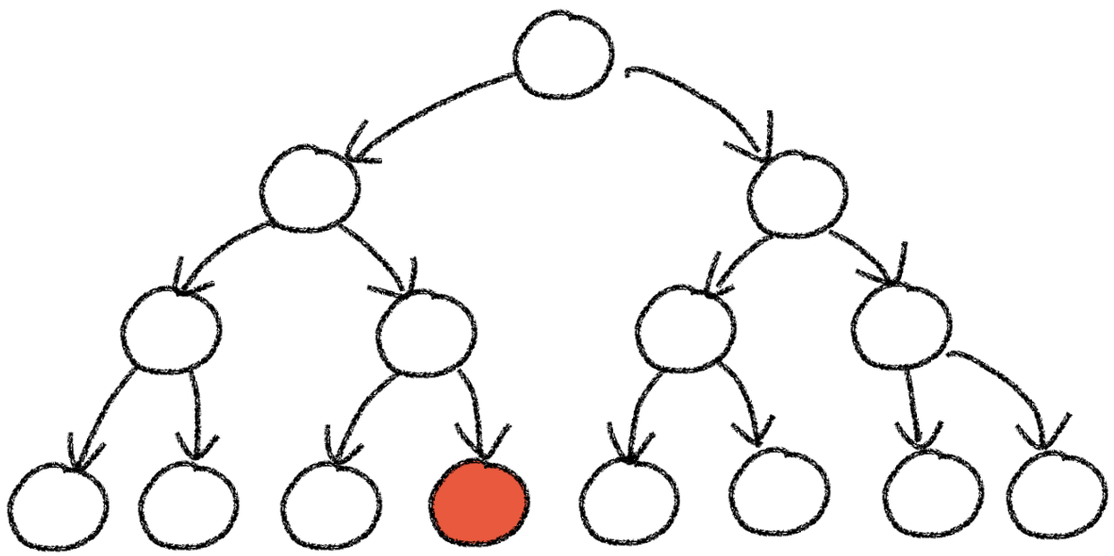
"Globally always ϕ"



Intuition of the “until” operator

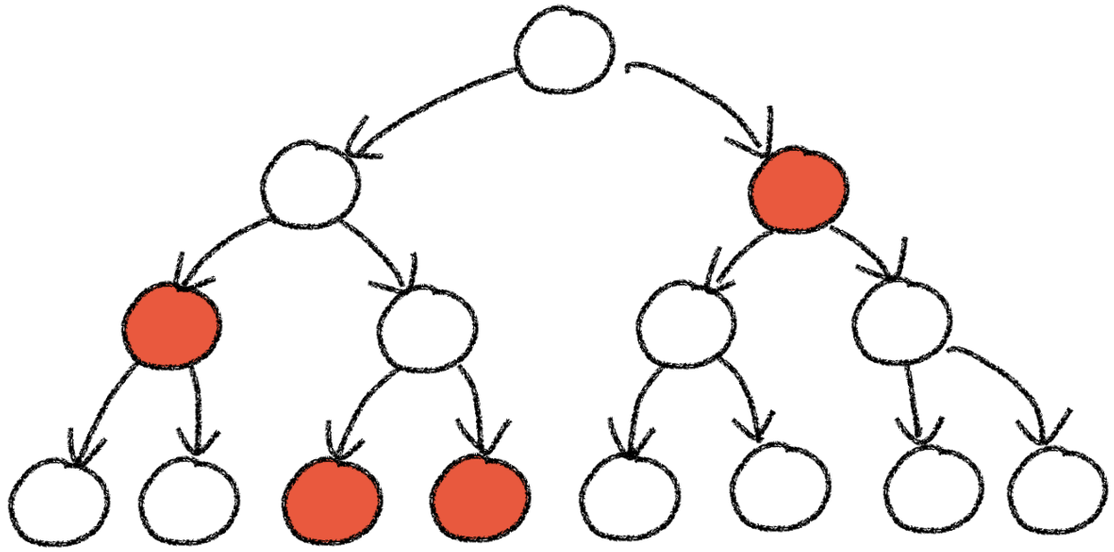
$\exists \Diamond \phi$

"ϕ holds potentially"



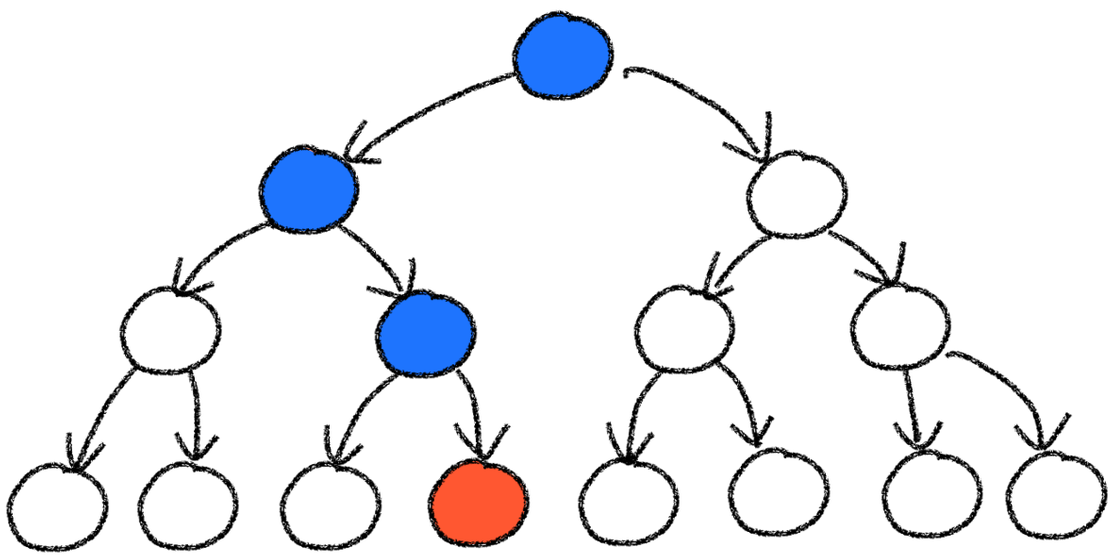
$\forall \Diamond \phi$

"ϕ is inevitable"



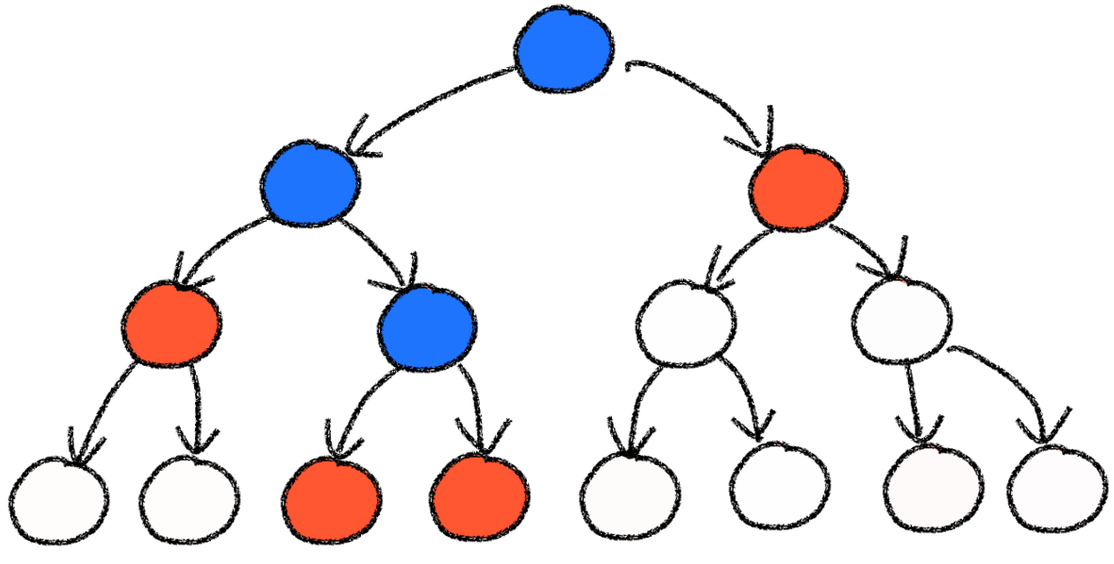
$\exists \phi_1 \cup \phi_2$ ← **Focus of today**

"there is a path s.t. $\phi_1 \cup \phi_2$ holds"



$\forall \phi_1 \cup \phi_2$

"in all paths $\phi_1 \cup \phi_2$ holds"



CTL - formal semantics

We define the formal semantics of CTL as 2 relations.

A relation between states and state formulas

$s \models true$ (holds always)

$s \models p$ iff $p \in L(s)$

$s \models \neg\phi$ iff $s \not\models \phi$

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$

$s \models \exists\psi$ iff $\exists\pi \in Paths(s) \,.\, \pi \models \psi$

$s \models \forall\psi$ iff $\forall\pi \in Paths(s) \,.\, \pi \models \psi$

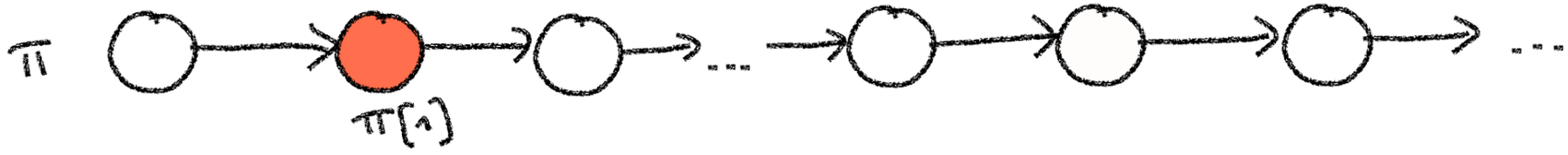
and a relation between paths and path formulas (next slide)

$\pi \models \psi$ iff …

Semantics of path formulas

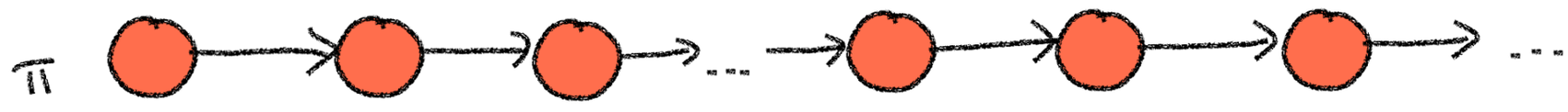
And here is the formal semantics of path formulas

$\pi \vDash \bigcirc \phi$ iff $\pi[1] \vDash \phi$

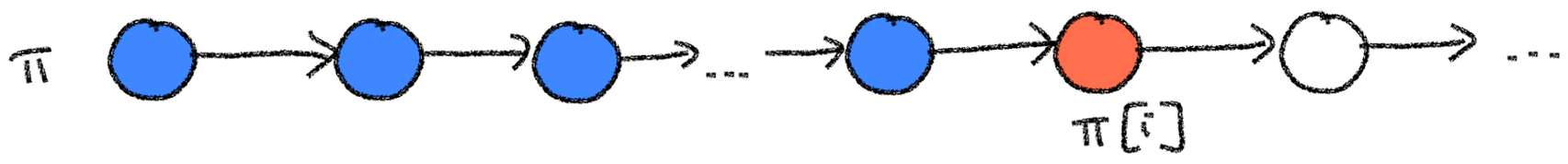


$\pi \vDash \Diamond \phi$ iff $\exists i \in \mathbb{N} \,.\, \pi[i] \vDash \phi$

$\pi \vDash \Box \phi$ iff $\forall i \in \mathbb{N} \,.\, \pi[i] \vDash \phi$



$\pi \vDash \phi_1 \mathsf{U} \phi_2$ iff $\exists i \in \mathbb{N} \,.\, \pi[i] \vDash \phi_2 \wedge \forall 0 \leq j < i \,.\, \pi[j] \vDash \phi_1$



Naïve algorithms for state formulas

$s \models true$

$s \models p$ iff iff $p \in L(s)$

$s \models \neg\phi$ iff $s \not\models \phi$

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$

easy programming task

$s \models \exists\phi$ iff $\exists\pi \in Paths(s) \,.\, \pi \models \psi$

$s \models \forall\phi$ iff $\forall\pi \in Paths(s) \,.\, \pi \models \psi$

Naïve algorithms for state formulas

$s \models true$

$s \models p$ iff iff $p \in L(s)$

$s \models \neg\phi$ iff $s \not\models \phi$

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$

easy programming task

$s \models \exists\phi$ iff $\exists \pi \in Paths(s) \,.\, \pi \models \psi$

$s \models \forall\phi$ iff $\forall \pi \in Paths(s) \,.\, \pi \models \psi$

```
modelCheck(s, ψ) =
    for each π in Paths(s) do
        if modelCheck(π, ψ) return true;
    end
    return false;
```

What's wrong with this?

Naïve algorithms for state formulas

$s \models true$

$s \models p$ iff iff $p \in L(s)$

$s \models \neg\phi$ iff $s \not\models \phi$

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$

easy programming task

$s \models \exists\phi$ iff $\exists \pi \in Paths(s) \,.\, \pi \models \psi$

$s \models \forall\phi$ iff $\forall \pi \in Paths(s) \,.\, \pi \models \psi$

```
modelCheck(s, Ψ) =
    for each π in Paths(s) do
        if modelCheck(π, Ψ) return true;
    end
    return false;
```

1. We need to know how to generate paths
2. There are infinitely many/exponential number of paths

Naïve algorithms for path formulas

$\pi \models \bigcirc \phi$ iff $\pi[1] \models \phi$

↳ modelCheck(π, $\bigcirc\phi$) = return modelCheck($\pi[\ldots]$, ϕ)

$\pi \models \Diamond \phi$ iff $\exists i \in \mathbb{N} . \pi[i] \models \phi$

↳ modelCheck(π, $\Diamond\phi$) = for each state in π ...

$\pi \models \Box \phi$ iff $\forall i \in \mathbb{N} . \pi[i] \models \phi$

↳ modelCheck(π, $\Box\phi$) = for each state in π ...

$\pi \models \phi_1 \mathsf{U} \phi_2$ iff $\exists i \in \mathbb{N} . \pi[i] \models \phi_2 \wedge \forall 0 \leq j < i . \pi[j] \models \phi_1$

↳ modelCheck(π, $\phi_1 \mathsf{U} \phi_2$) = for each state in π ...

What's wrong with this?

Naïve algorithms for path formulas

$\pi \models \bigcirc \phi$ iff $\pi[1] \models \phi$

↳ modelCheck(π, $\bigcirc\phi$) = return modelCheck($\pi[\dots]$, ϕ)

$\pi \models \Diamond \phi$ iff $\exists i \in \mathbb{N} . \pi[i] \models \phi$

↳ modelCheck(π, $\Diamond\phi$) = for each state in π ...

$\pi \models \Box \phi$ iff $\forall i \in \mathbb{N} . \pi[i] \models \phi$

↳ modelCheck(π, $\Box\phi$) = for each state in π ...

$\pi \models \phi_1 \mathsf{U} \phi_2$ iff $\exists i \in \mathbb{N} . \pi[i] \models \phi_2 \wedge \forall 0 \leq j < i . \pi[j] \models \phi_1$

↳ modelCheck(π, $\phi_1 \mathsf{U} \phi_2$) = for each state in π ...

Infinite paths!

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

CTL grammars recall

We can get rid of the "always" and "eventually" operators

$$\phi ::= true \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\psi \mid \forall\psi$$

$$\psi ::= \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi_1 \mathsf{U} \phi_2$$

Why can we do this?

Alternatively, we can get rid of the universal quantifier

$$\phi ::= true \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\psi \mid \forall\psi$$

$$\psi ::= \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi_1 \mathsf{U} \phi_2$$

The above grammar yields CTL formulas in so-called "**existential normal form**"

Existential normal form (ECTL)

The grammar for ECTL as we saw it in Lecture 03

$$\phi ::= true \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\psi$$

$$\psi ::= \bigcirc\phi \mid \Box\phi \mid \phi_1 \mathsf{U} \phi_2$$

Equivalent (“flattened”) grammar with just state formulas

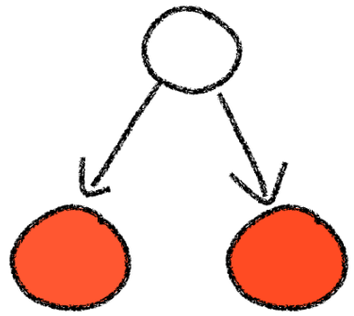
$$\phi ::= true \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\bigcirc\phi \mid \exists\Box\phi \mid \exists\phi_1 \mathsf{U} \phi_2$$

NOTE: we focus on this grammar to reduce the number of algorithms, but dedicated algorithms for all CTL operators can be designed.

From CTL to ECTL

We can go from CTL to ECTL using these equivalences

$$\forall \bigcirc \phi \equiv \neg \exists \bigcirc \neg \phi$$

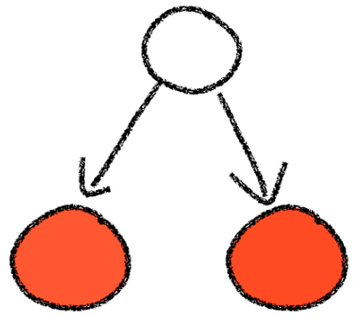


$$\forall(\phi_1 \mathsf{U} \phi_2) \equiv \underbrace{\neg \exists((\neg \phi_2)\mathsf{U}(\neg \phi_1 \wedge \neg \phi_2))}_{\text{"you cannot reach a state that satisfies neither } \phi_1 \text{ nor } \phi_2 \text{ without traversing a state that satisfies } \phi_2\text{"}} \wedge \underbrace{\neg \exists \Box \neg \phi_2}_{\text{"}\phi_2 \text{ is inevitable"}}$$

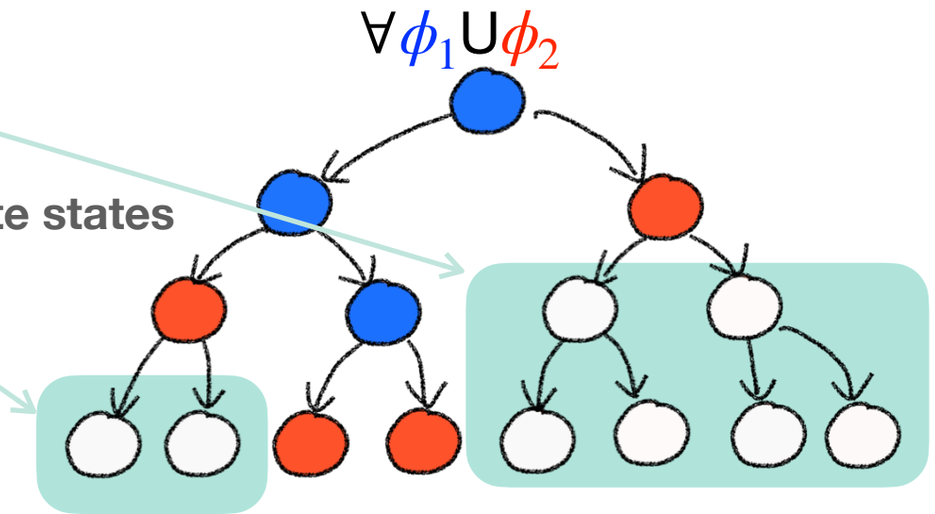
From CTL to ECTL

We can go from CTL to ECTL using these equivalences

$$\forall \bigcirc \phi \equiv \neg \exists \bigcirc \neg \phi$$



$$\forall(\phi_1 \mathsf{U} \phi_2) \equiv \underbrace{\neg \exists((\neg \phi_2)\mathsf{U}(\neg \phi_1 \wedge \neg \phi_2))}_{\text{"you cannot reach a state that satisfies neither } \phi_1 \text{ nor } \phi_2 \text{ without traversing a state that satisfies } \phi_2\text{"}} \wedge \underbrace{\neg \exists \Box \neg \phi_2}_{\text{"}\phi_2 \text{ is inevitable"}}$$



Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Satisfaction Sets - recall

We define the **satisfaction set** of a CTL state formula as the set of states that satisfy the formula

$$sat(\phi) = \{s \mid s \vDash \phi\}$$

Semantics over a TS - recall

We say that a transition system satisfies a formula if all its initial states satisfy the formula:

$$T \models \phi \text{ iff } \forall s \in I \,.\, s \models \phi$$

or, equivalently

$$T \models \phi \text{ iff } I \subseteq sat(\phi)$$

We use the latter in our algorithm

```
modelCheck(TS,phi) = {
   return I ⊆ sat(phi);
}
```

Main algorithm

Now that we have our main algorithm

```
modelCheck(TS,phi) = {
   return I ⊆ sat(phi);
}
```

All we need to do is to implement function **`sat(…)`** for ECTL

```
sat(true) = …
sat(not phi) = …
sat(phi1 or phi2) = …
sat(EX phi) = …
sat(EG phi) = …
sat(E(phi1 U phi2) = …
```

Main algorithm

Now that we have our main algorithm

```
modelCheck(TS,phi) = {
   return I ⊆ sat(phi);
}
```

All we need to do is to implement function `sat(…)` for ECTL

```
sat(true) = …
sat(not phi) = …
sat(phi1 or phi2) =
sat(EX phi) = …
sat(EG phi) = …
sat(E(phi1 U phi2) = …
```

ideas about this one?

ideas about this one?

ideas about this one?

Recursion

We will see that **`sat(…)`** is recursive

```
sat(true) = return S;
sat(not phi) = S \ sat(phi)
sat(phi1 or phi2) = sat(phi1) U sat(phi2)
sat(EX phi) = …
sat(EG phi) = …
sat(E(phi1 U phi2) = …
```

Hence, invoking **`sat(phi)`** will recursively compute the satisfaction sets for all sub-formulas of **`phi`**

You can think of satisfaction sets being computed bottom-up on the parse tree of **`phi`**

Bottom-up computation

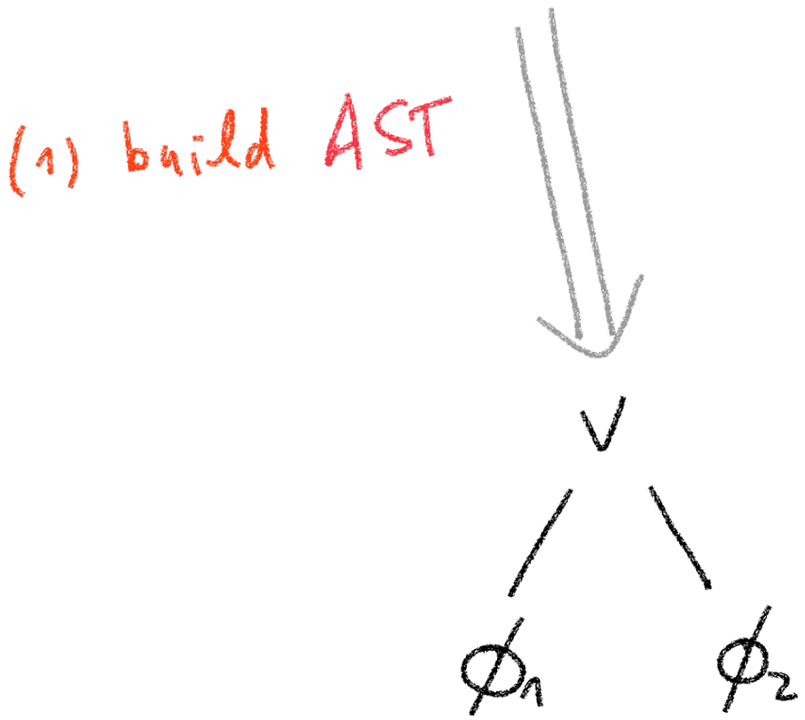
You can think of satisfaction sets being computed bottom-up on the **abstract syntax tree** of the formula

$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$

Bottom-up computation

You can think of satisfaction sets being computed bottom-up on the abstract syntax tree of the formula

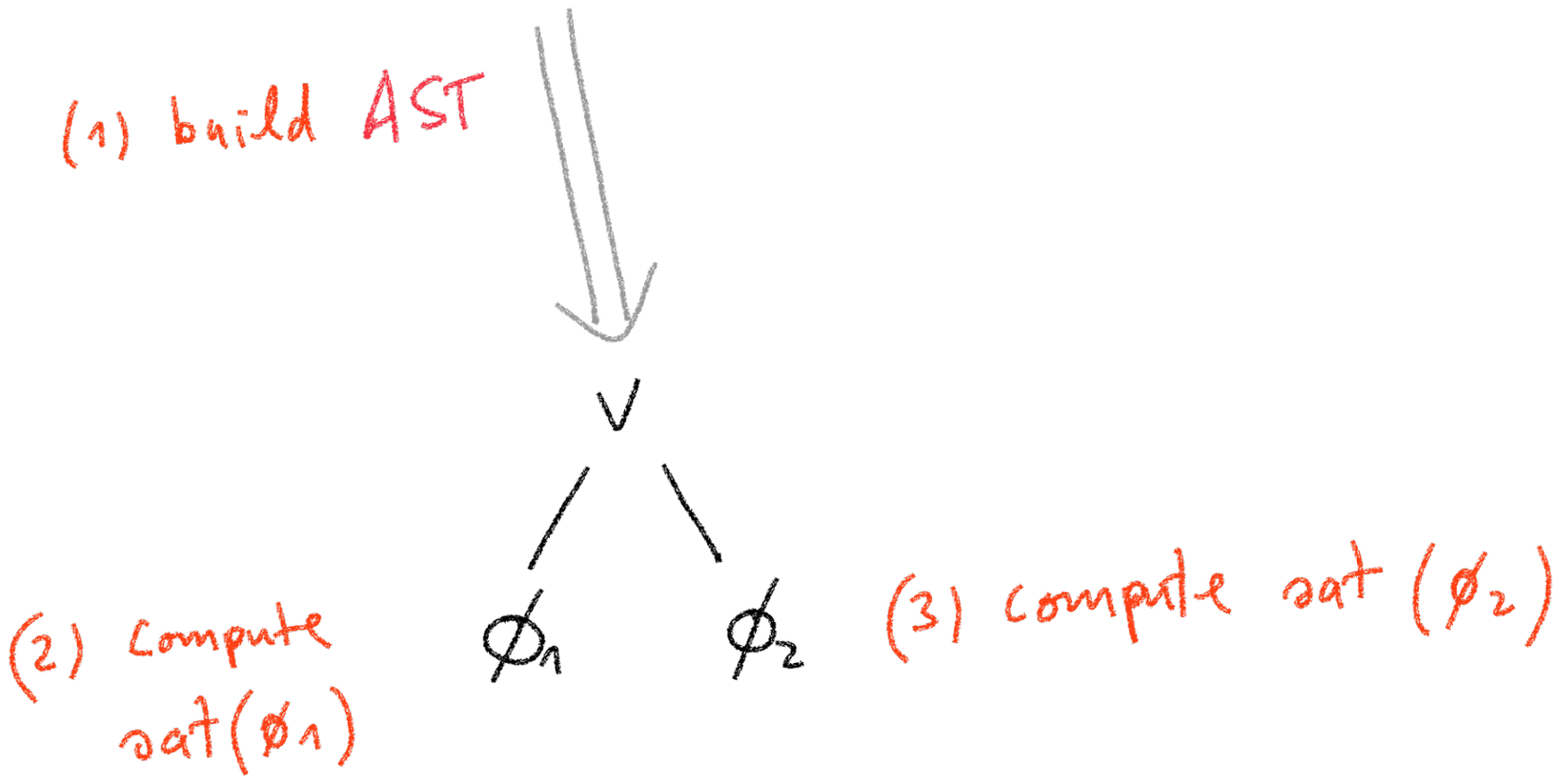
$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$



Bottom-up computation

You can think of satisfaction sets being computed bottom-up on the abstract syntax tree of the formula

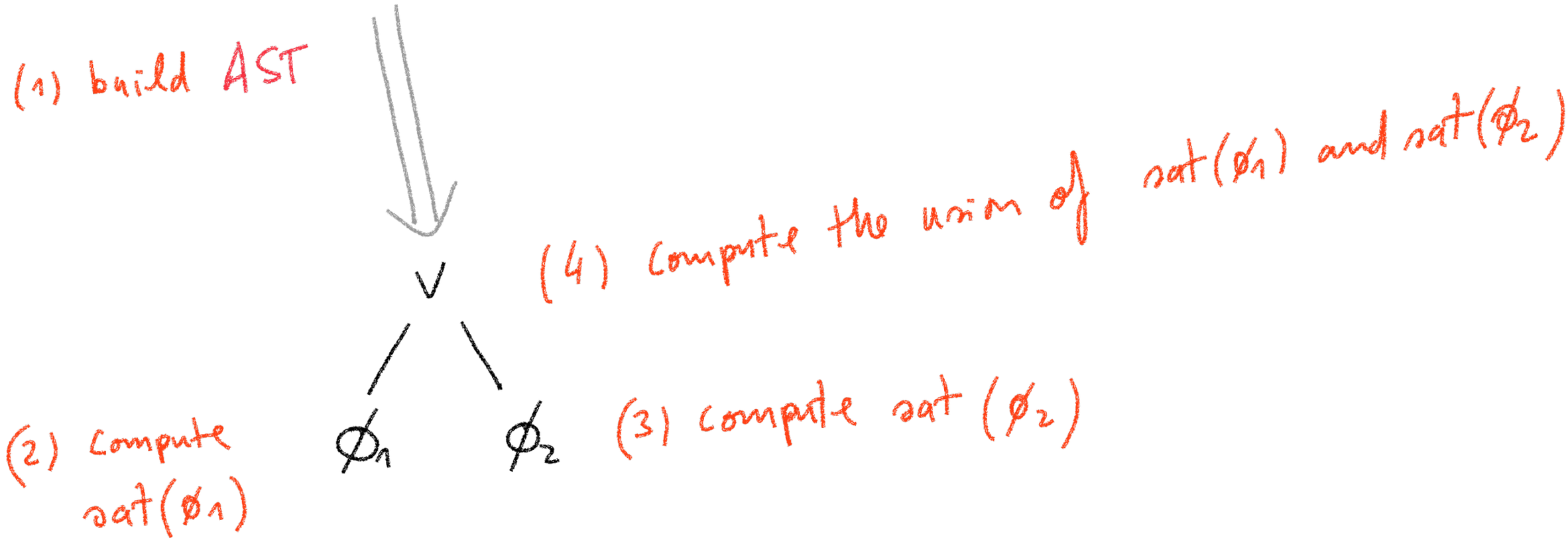
$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$



Bottom-up computation

You can think of satisfaction sets being computed bottom-up on the abstract syntax tree of the formula

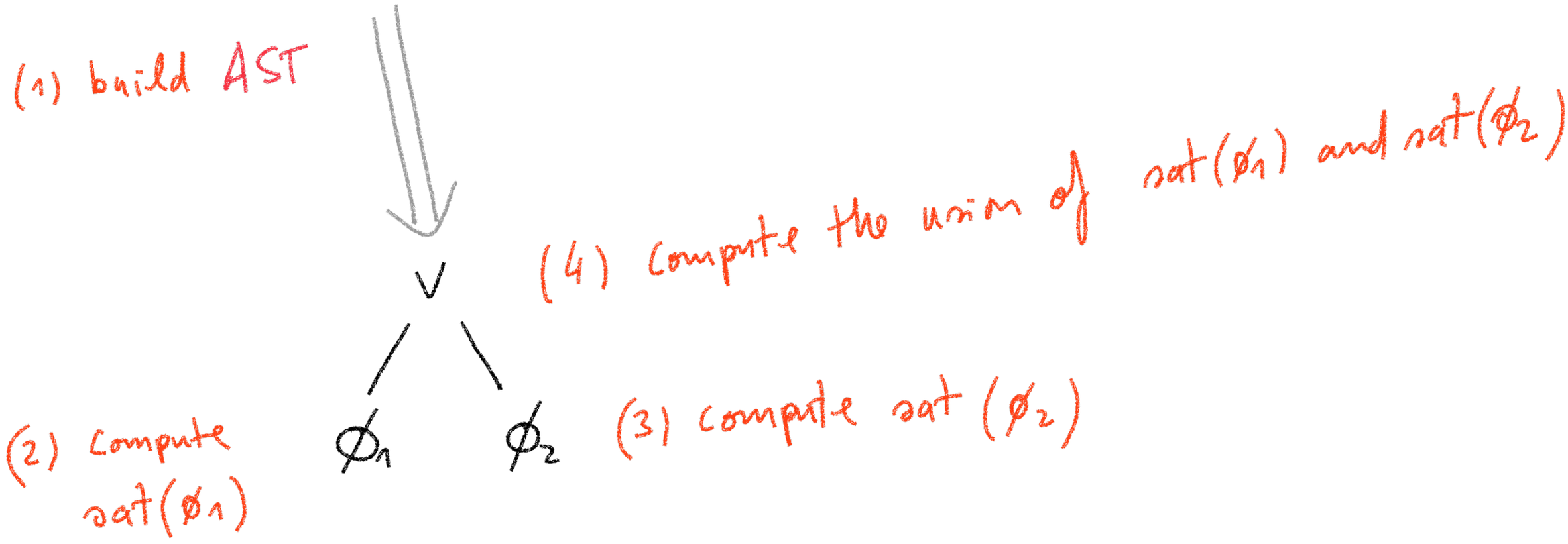
$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$



Bottom-up computation

You can think of satisfaction sets being computed bottom-up on the abstract syntax tree of the formula

$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$



We're applying sat(…) recursively —> thus the bottom-up approach

Memoisation

We can apply **memoisation** to avoid recomputing twice the same satisfaction sets. How?

- Store results of expensive function calls in a table (hashing)
- Use the results whenever the call occurs again

This can happen in general, but especially in the transformation from CTL to ECTL:

$$\forall(\phi_1 \mathsf{U} \phi_2) \equiv \neg\exists((\neg\phi_2)\mathsf{U}(\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\exists\Box\neg\phi_2$$

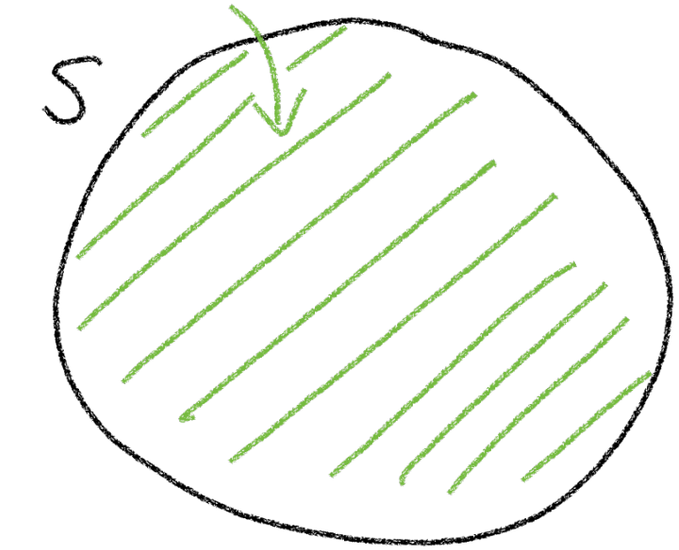
It is also convenient when doing model checking, especially by hand.

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Satisfaction sets characterisation of propositional fragment

$sat(true) = S$



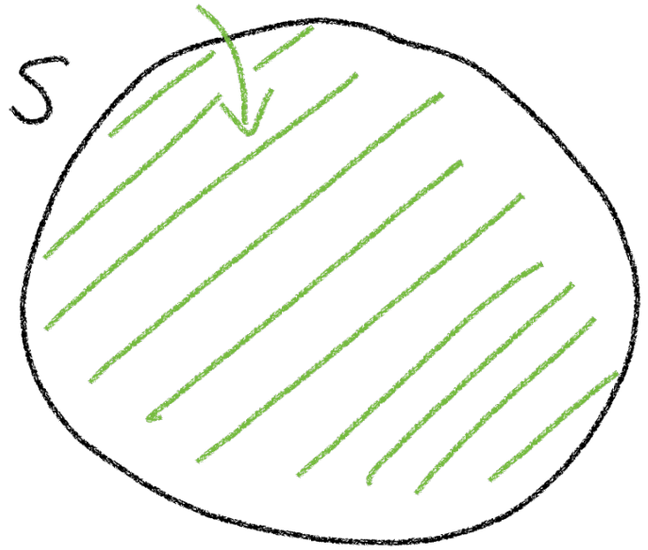
$sat(\neg\phi) = \ldots$

$sat(p) = \ldots$

$sat(\phi_1 \vee \phi_2) = \ldots$

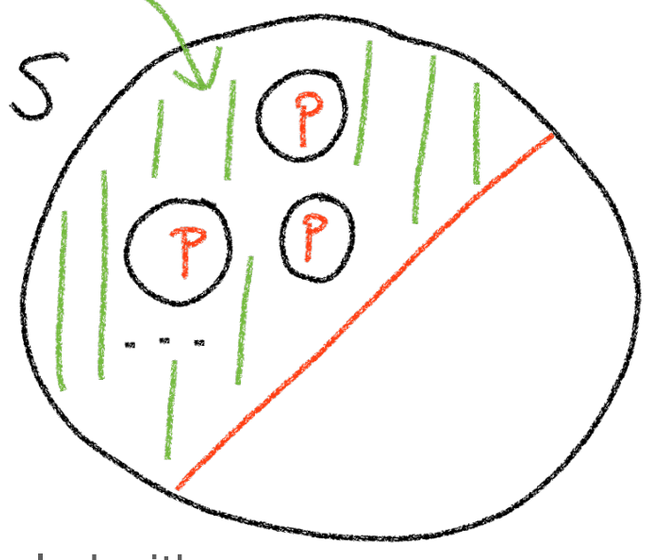
Satisfaction sets characterisation of propositional fragment

$sat(true) = S$



$sat(\neg\phi) = \ldots$

$sat(p) = \{s \in S \mid p \in L(s)\}$

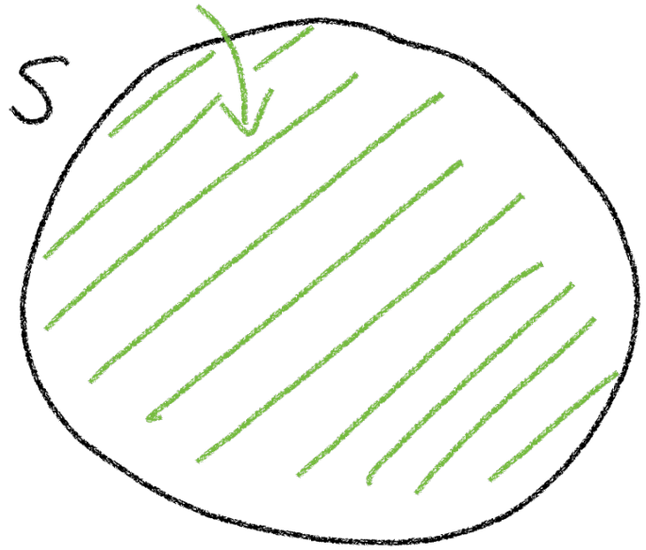


$sat(\phi_1 \vee \phi_2) = \ldots$

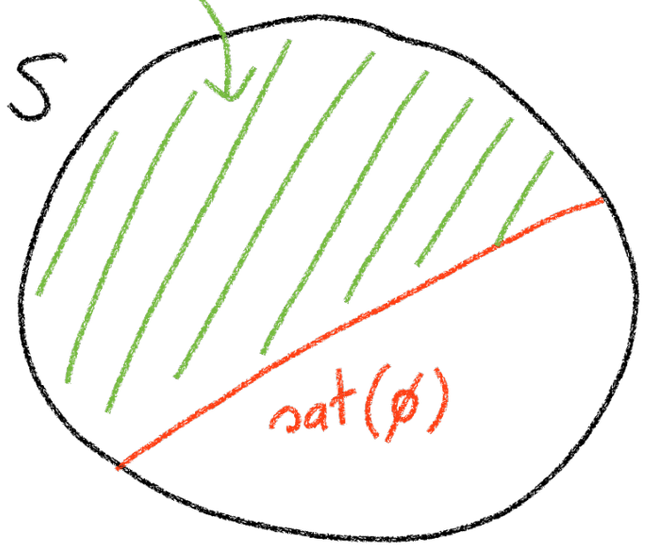
All states labeled with p

Satisfaction sets characterisation of propositional fragment

$sat(true) = S$

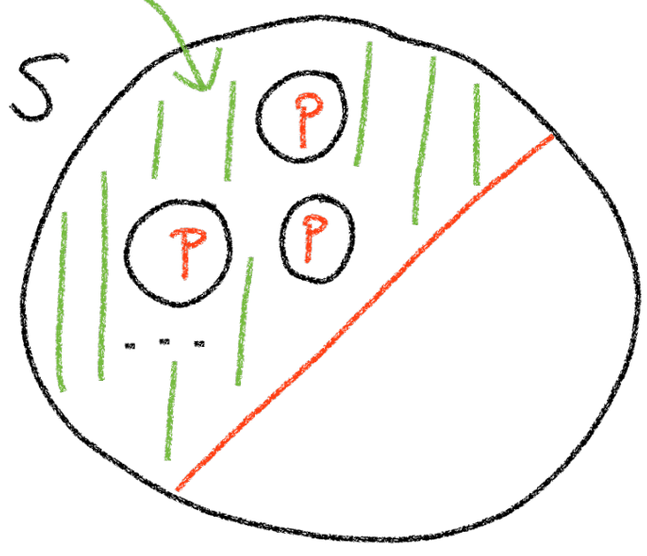


$sat(\neg\phi) = S \setminus sat(\phi)$



Complement of ϕ

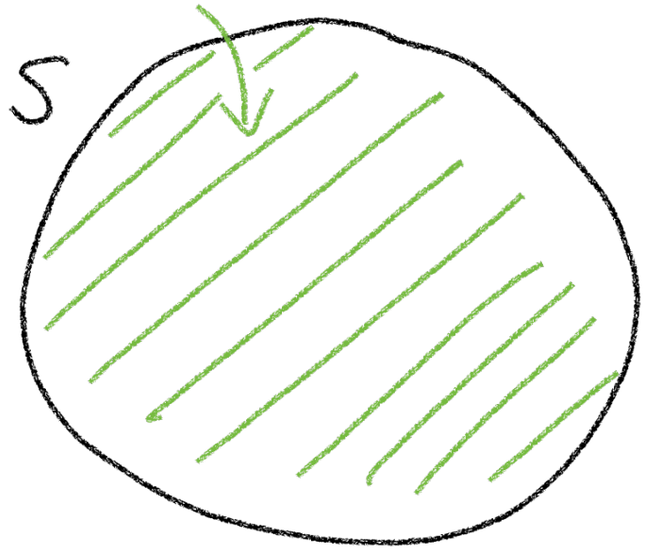
$sat(p) = \{s \in S \mid p \in L(s)\}$



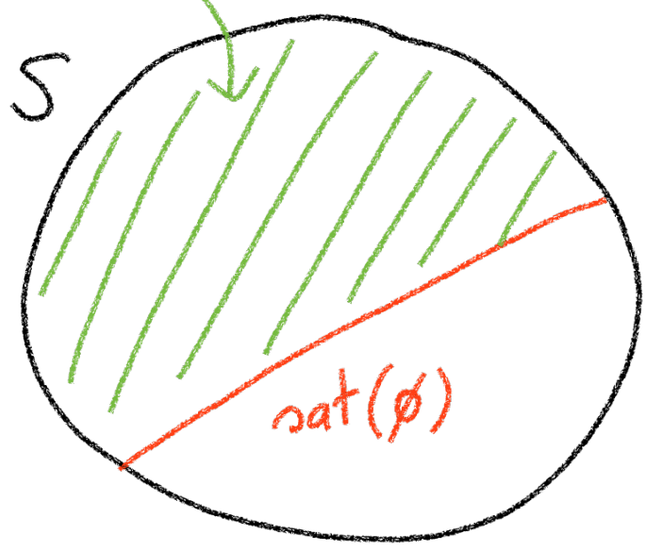
$sat(\phi_1 \vee \phi_2) = \ldots$

Satisfaction sets characterisation of propositional fragment

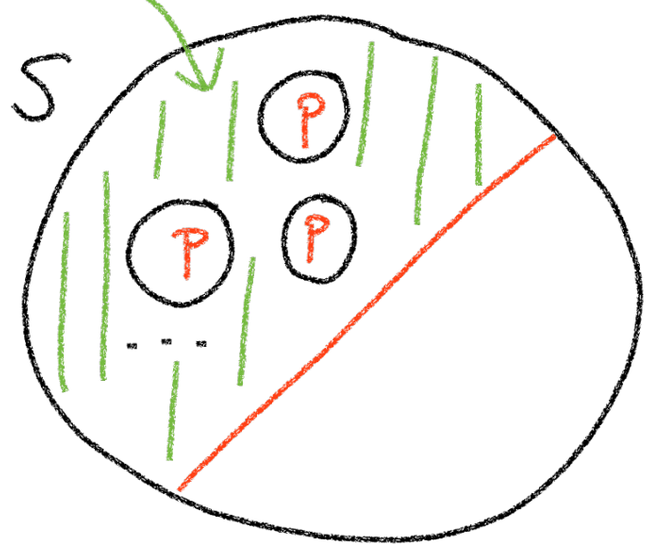
$$sat(true) = S$$



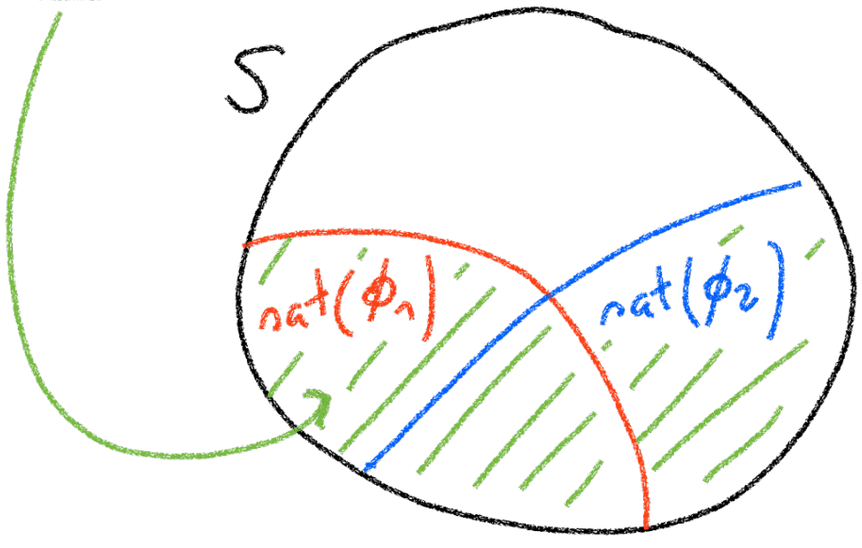
$$sat(\neg\phi) = S \setminus sat(\phi)$$



$$sat(p) = \{s \in S \mid p \in L(s)\}$$



$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$

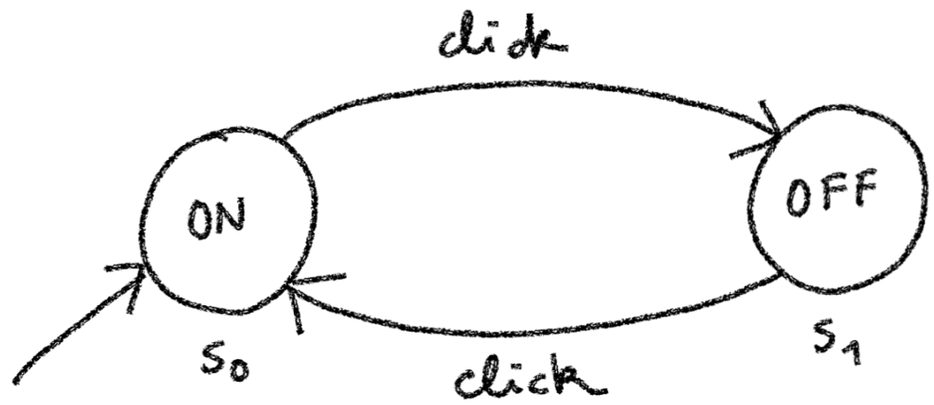


Satisfaction sets characterisation for EX

Assume we have a function $Post : S \to 2^S$

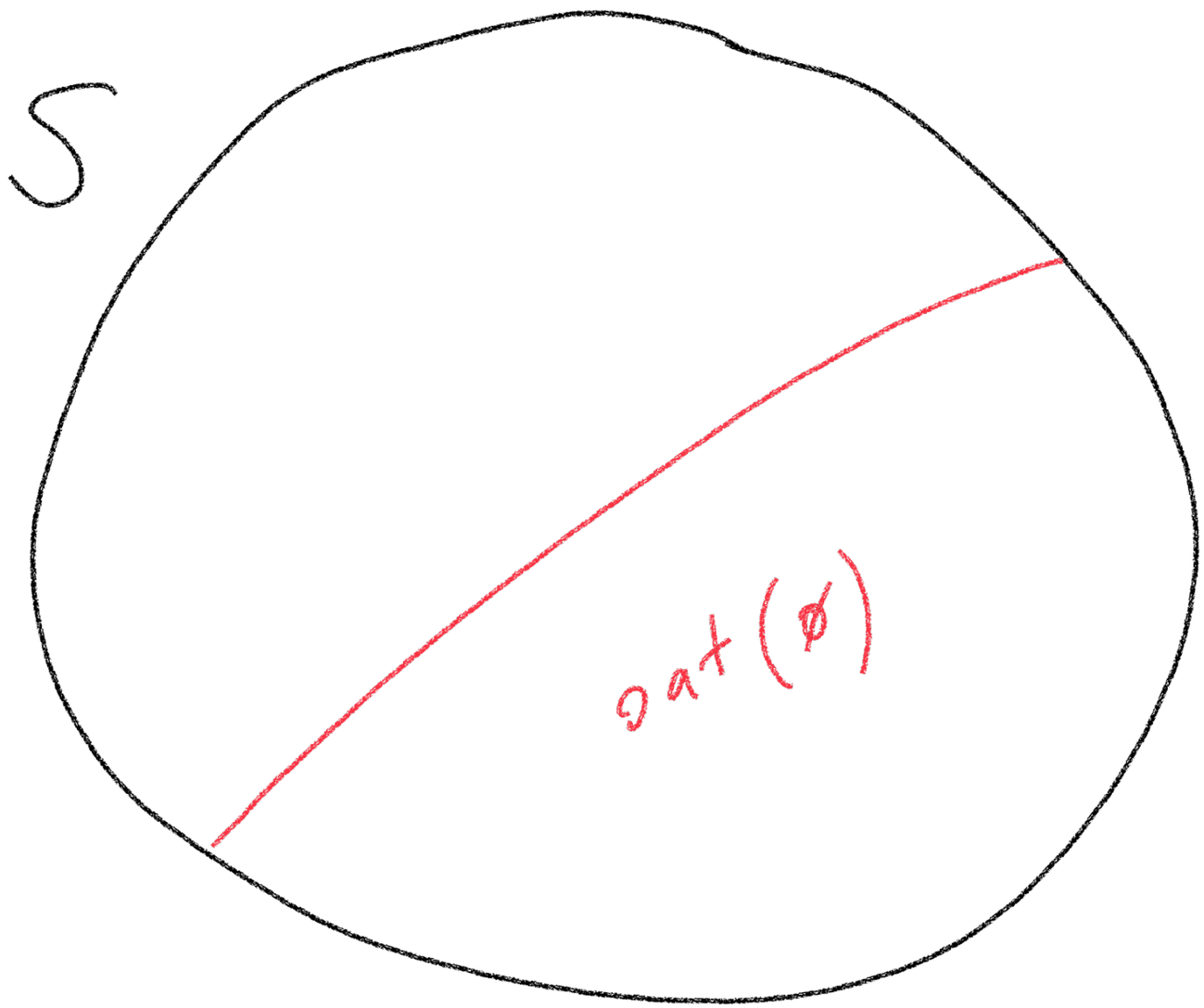
It returns all the successors of a given state ***s***

Example: $Post(s_0) = s_1$



Satisfaction sets characterisation for EX

$$sat(\exists \bigcirc \phi) = \{s \in S \mid \text{can you characterise such states?} \}$$



Satisfaction sets characterisation for EG and EU

Not so trivial, we will see them in detail in a while…

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Algorithms for basic cases

So far we have seen the following basic cases

$$sat(true) = S$$

$$sat(\neg\phi) = S \setminus sat(\phi)$$

$$sat(p) = \{s \in S \mid p \in L(s)\}$$

$$sat(\phi_1 \vee \phi_2) = sat(\phi_1) \cup sat(\phi_2)$$

$$sat(\exists \bigcirc \phi) = \{s \in S \mid Post(s) \cap \mathsf{sat}(\phi) \neq \emptyset\}$$

These cases can be easily implemented with a suitable representation of states and transitions, amenable for set operations (complement, union, etc.)

NOTE: A popular approach is to use BDDs to represent sets of states using boolean formulas (not covered in this course).

Computation Tree Logic (CTL)

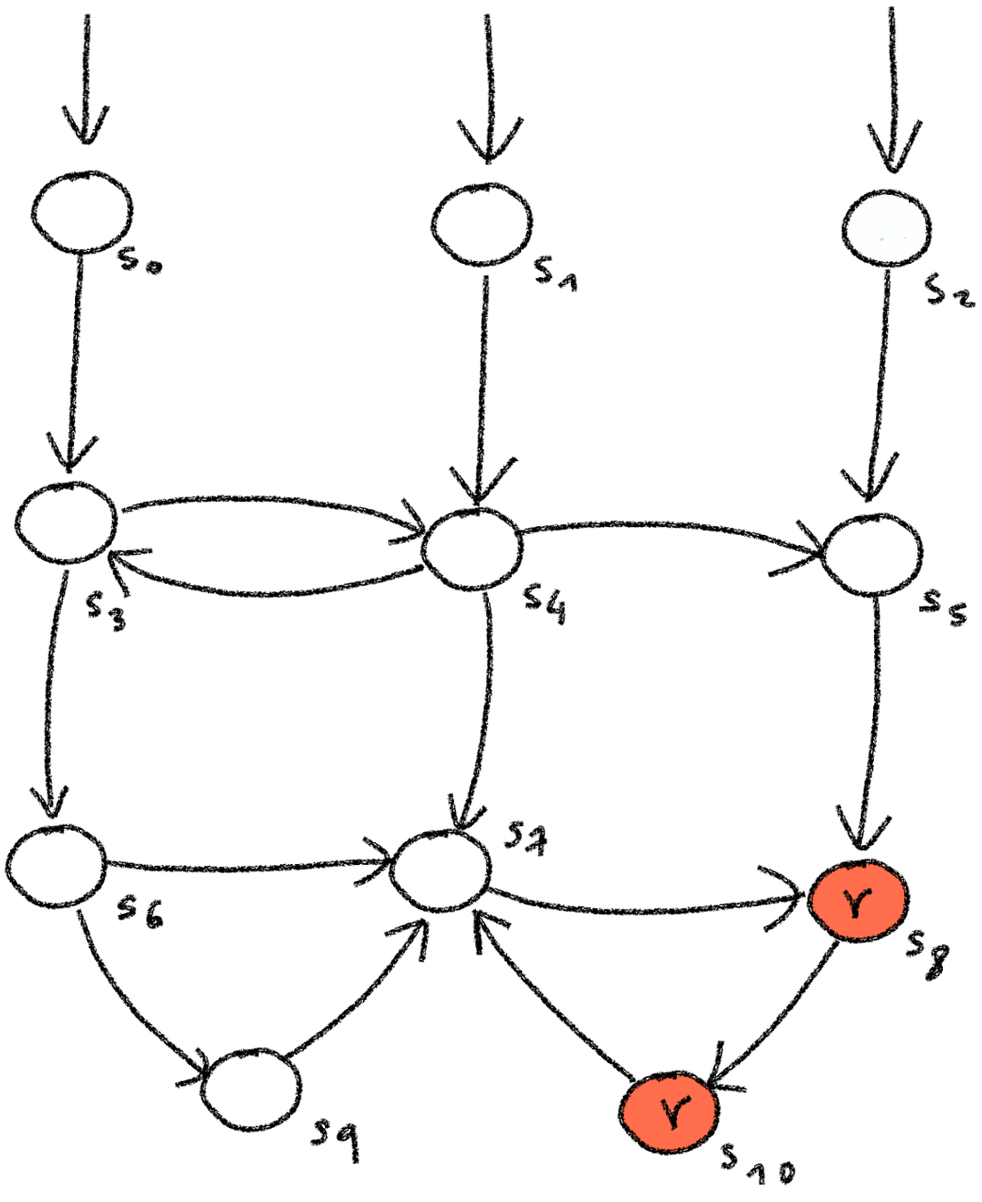
- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Let's start simply with EF

Before we see the algorithm or EU let us first look at the **simplest case** of EF.

Remember: $\exists \Diamond \phi \equiv \exists true \ \mathsf{U} \ \phi$

How would you compute the sat(EFr) in this TS?



The idea

The algorithm is essentially a backward reachability search.

We will now see why such algorithm is correct.

Expansion law for EF

The expansion law for EF

$$\exists \Diamond \phi \equiv \phi \vee \exists \bigcirc \exists \Diamond \phi$$

provides a recursive definition of `sat`

$$sat(\exists \Diamond \phi) = sat(\phi \vee \exists \bigcirc \exists \Diamond \phi)$$

(applying expansion law)

$$= sat(\phi) \cup sat(\exists \bigcirc \exists \Diamond \phi)$$

(def. of sat(...\/...))

$$= sat(\phi) \cup \{s \in S \mid Post(s) \cap sat(\exists \Diamond \phi) \neq \emptyset\}$$

(def. of sat(EX...))

A fix point computation

From our recursive definition

$$sat(\exists \Diamond \phi) \equiv sat(\phi) \cup \{s \in S \mid Post(s) \cap sat(\exists \Diamond \phi) \neq \emptyset\}$$

We know that we look
for a set T such that

A fix point computation

From our recursive definition

$$sat(\exists \Diamond \phi) \equiv sat(\phi) \cup \{s \in S \mid Post(s) \cap sat(\exists \Diamond \phi) \neq \emptyset\}$$

We know that we look for a set T such that

(1) $T \supseteq sat(\phi)$

A fix point computation

From our recursive definition

$$sat(\exists \Diamond \phi) \equiv sat(\phi) \cup \{s \in S \mid Post(s) \cap sat(\exists \Diamond \phi) \neq \emptyset\}$$

We know that we look for a set T such that

(1) $T \supseteq sat(\phi)$

(2) $Post(s) \cap T \neq \emptyset \Rightarrow s \in T$

A fix point computation

From our recursive definition

$$sat(\exists \Diamond \phi) \equiv sat(\phi) \cup \{s \in S \mid Post(s) \cap sat(\exists \Diamond \phi) \neq \emptyset\}$$

We know that we look for a set T such that

(1) $T \supseteq sat(\phi)$

(2) $Post(s) \cap T \neq \emptyset \Rightarrow s \in T$

It can be shown that $sat(\exists \Diamond \phi)$

is the smallest set T satisfying (1) and (2)

How would you prove it?
(Solution in the book)

What would be the largest set T satisfying (1) and (2)?

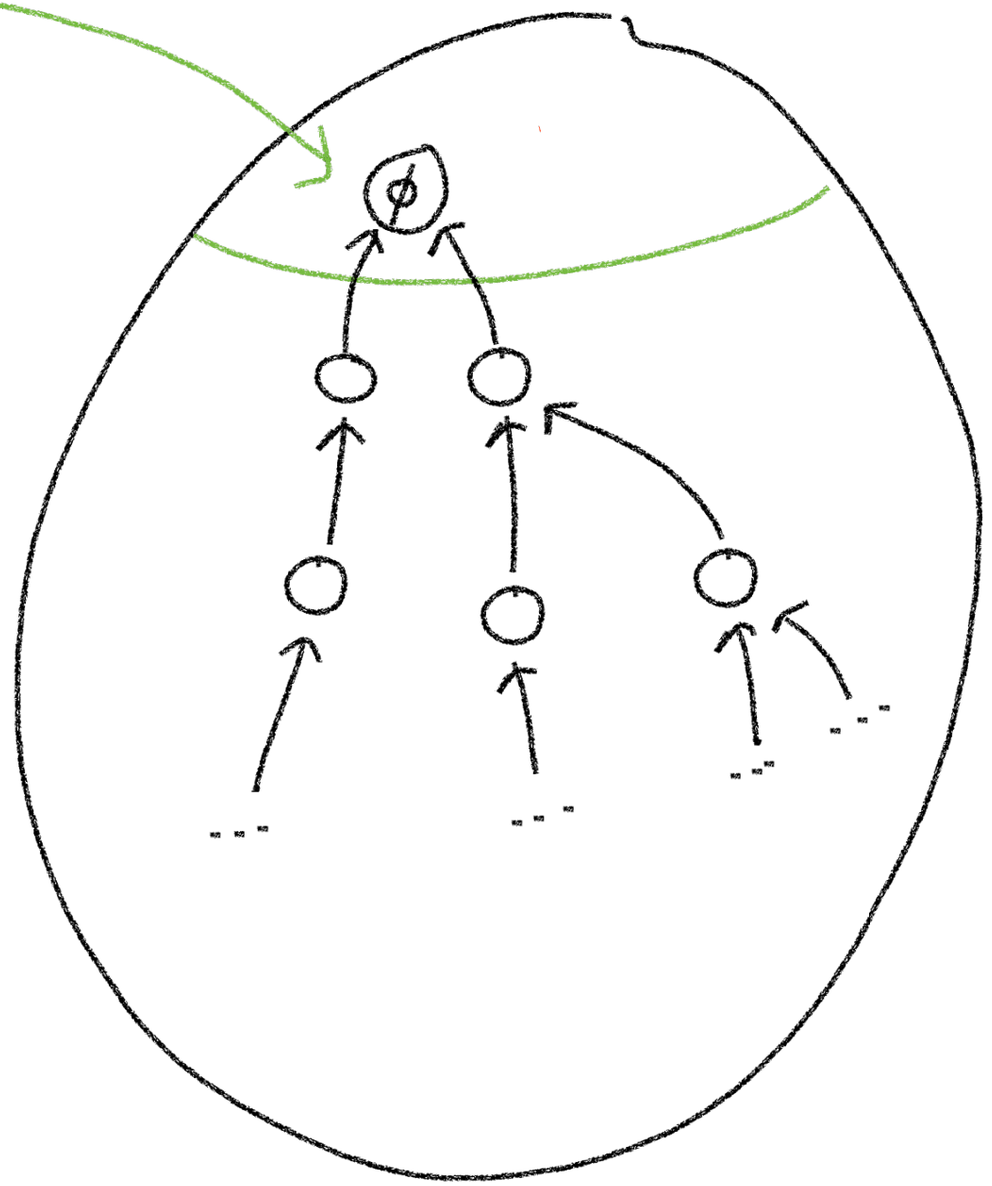
A fix point computation

We can then just perform a standard least fix point calculation

for LFP we're going to add things

$T_0 := sat(\phi)$

this is what we reach in 0 steps



This is essentially a backwards reachability search

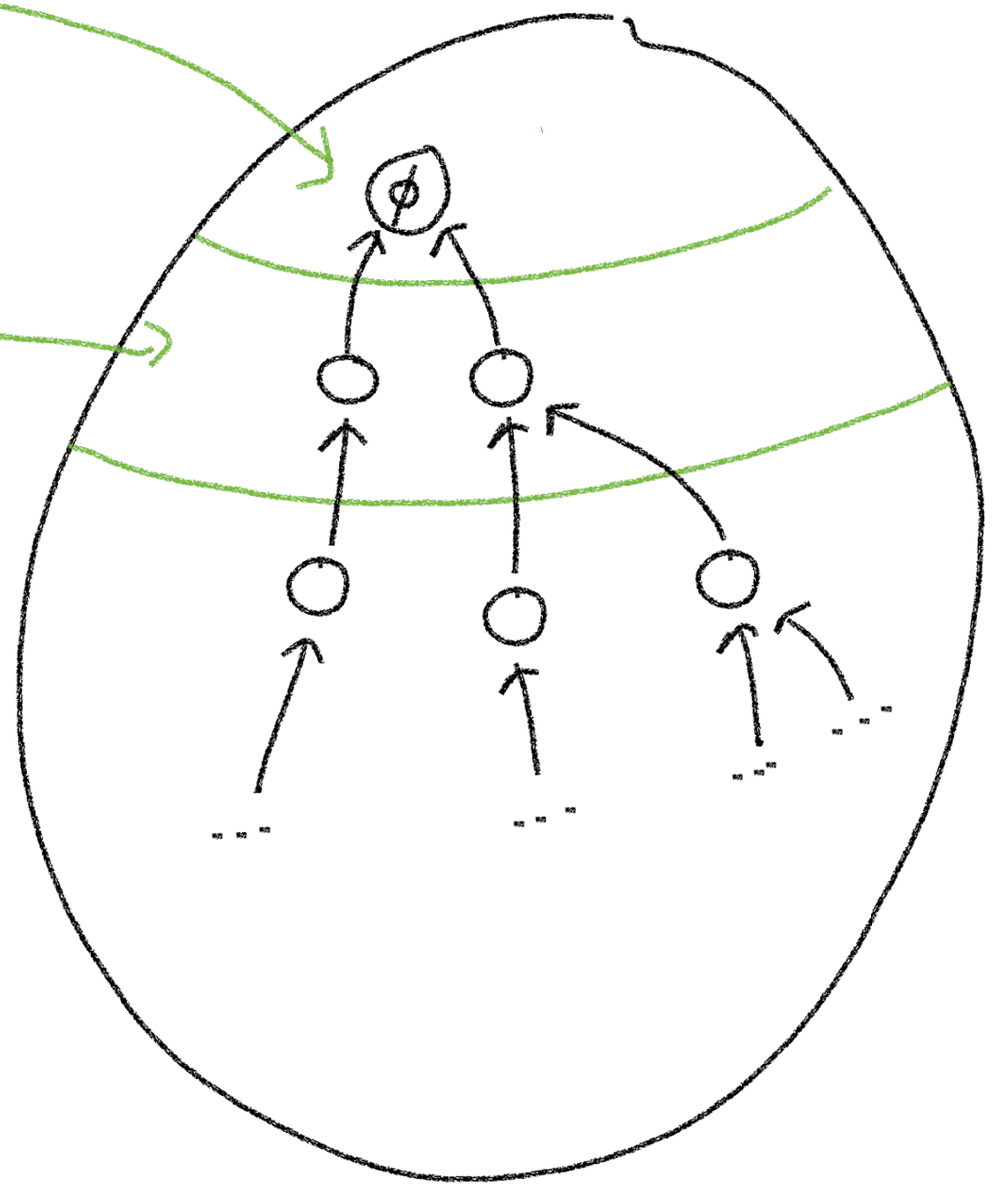
A fix point computation

We can then just perform a standard least fix point calculation

$T_0 := sat(\phi)$

$T_1 := T_0 \cup \{s \in S \backslash T_0 \mid Post(s) \cap T_0 \neq \emptyset\}$

this is what we reach in 1 steps



This is essentially a backwards reachability search

A fix point computation

We can then just perform a standard least fix point calculation

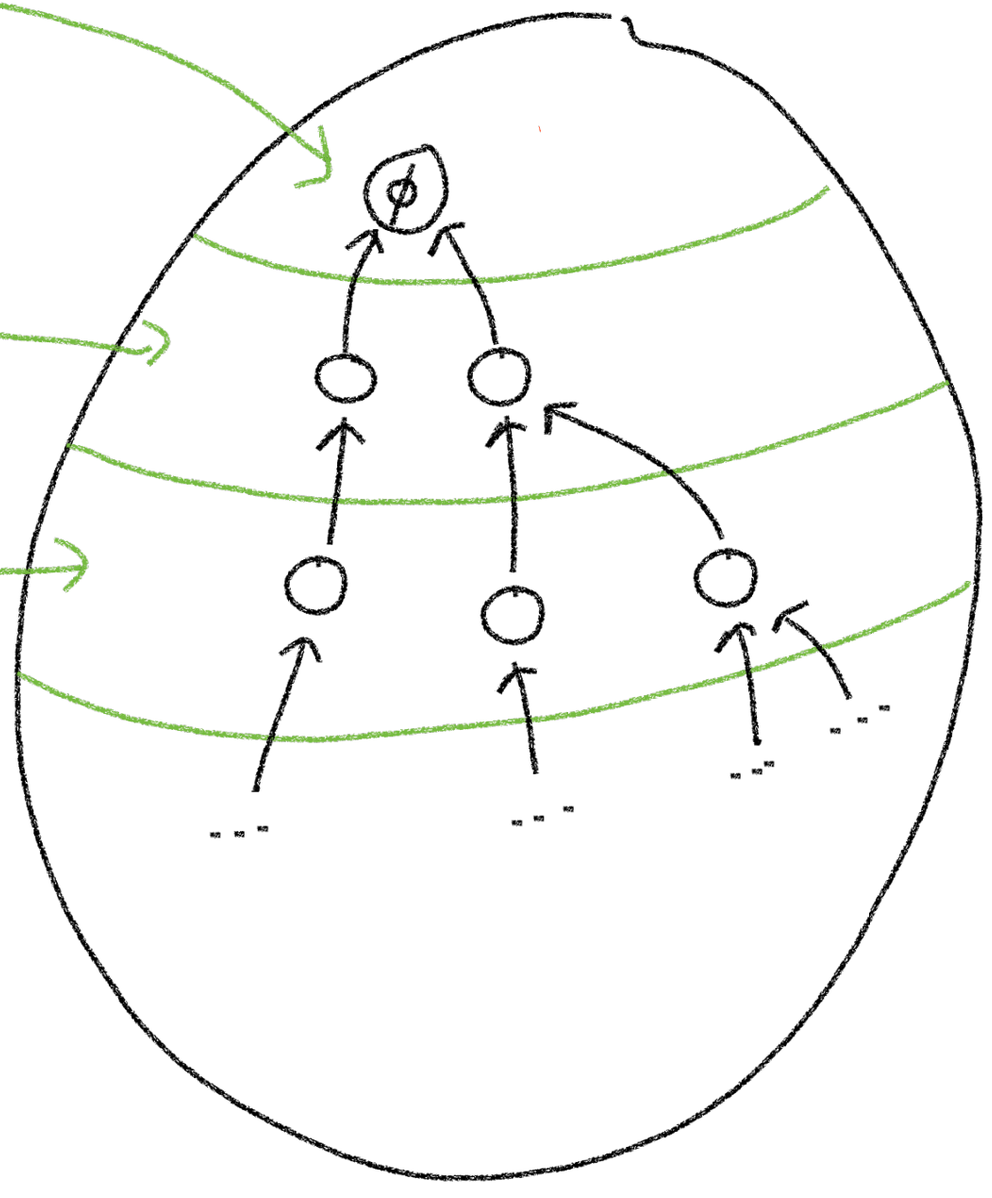
$T_0 := sat(\phi)$

$T_1 := T_0 \cup \{s \in S \backslash T_0 \mid Post(s) \cap T_0 \neq \emptyset\}$

$T_2 := T_1 \cup \{s \in S \backslash T_1 \mid Post(s) \cap T_1 \neq \emptyset\}$

this is what we reach in 2 steps

...



This is essentially a backwards reachability search

Algorithm for EF

Finally, the algorithm…

$T := sat(\phi)$; //the set of states to be returned as a result

$W := sat(\phi)$; // working set (states to be explored)

while $W \neq \emptyset$ do

    remove some s from W;

    foreach $s' \in Pre(s)$ do

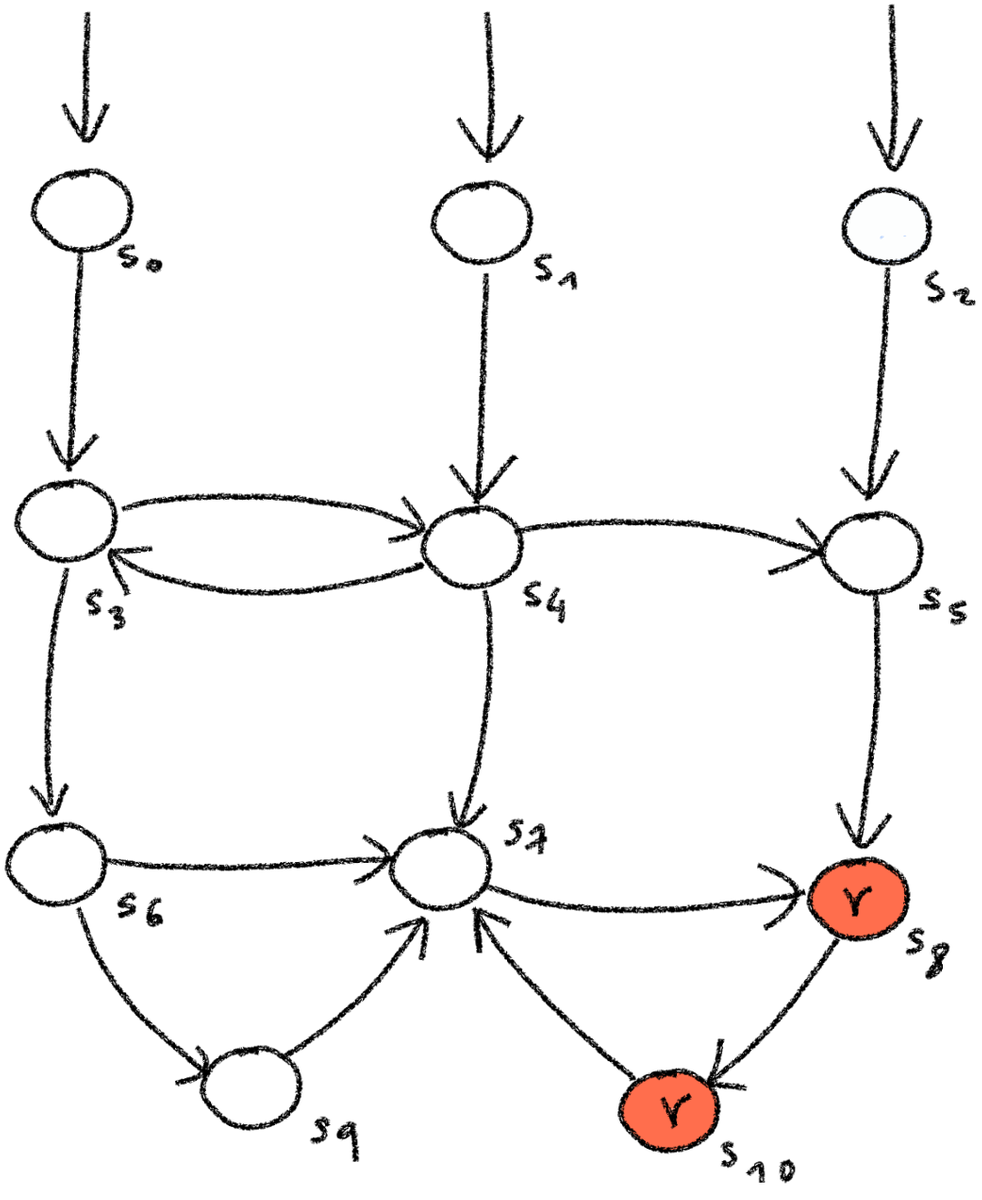
        if $s' \notin T$ then

            $T := T \cup s'$;

            $W := W \cup s'$;

return T;

Example



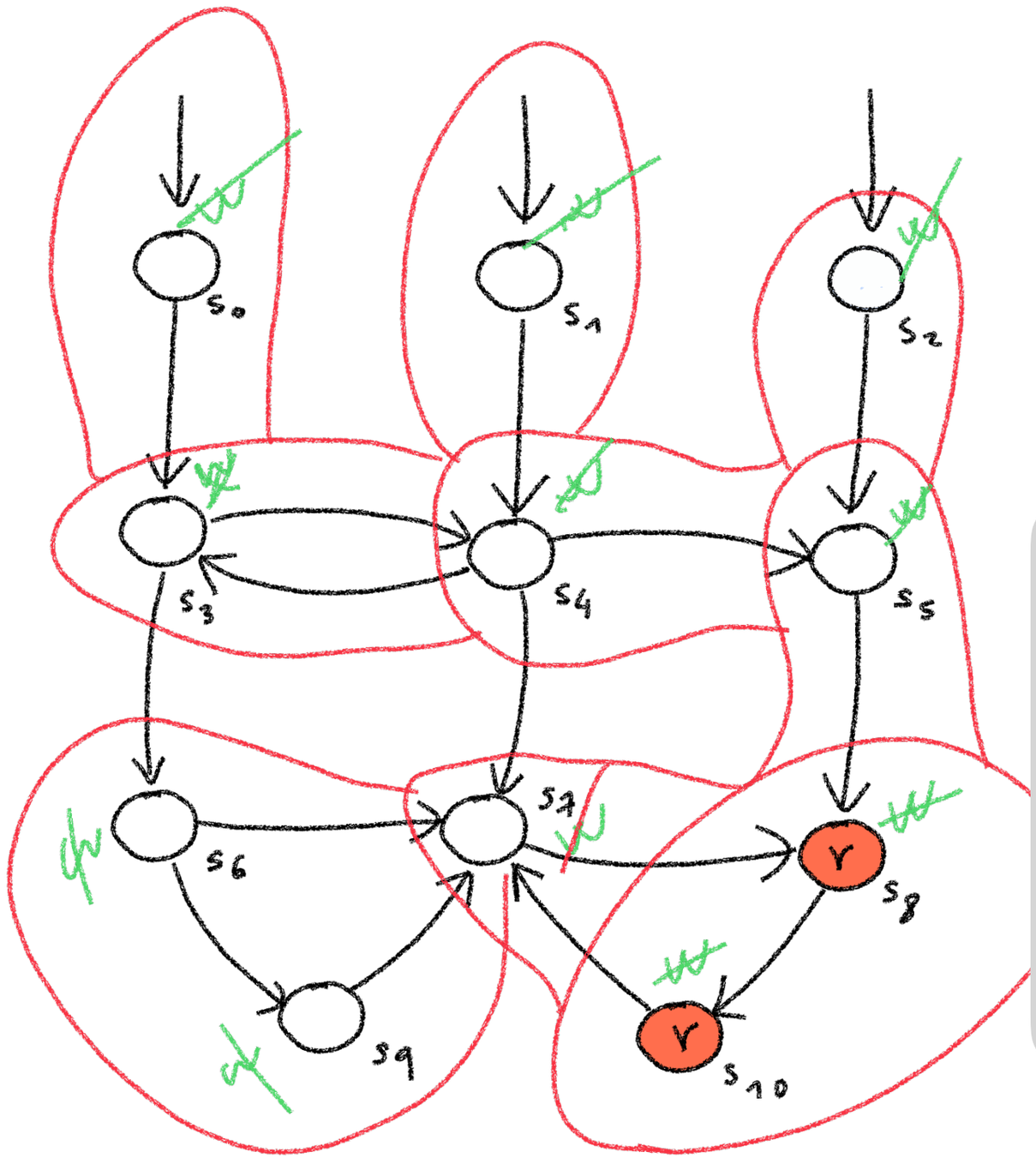
sat (∃ ◇ r)

Sets :- T

- W

$T := sat(\phi)$; //the set of states to be returned as a result
$W := sat(\phi)$; // working set (states to be explored)
while $W \neq \emptyset$ do
    remove some s from W;
    foreach $s' \in Pre(s)$ do
        if $s' \notin T$ then
            $T := T \cup s'$;
            $W := W \cup s'$;
return T;

Example



sat (∃ ◇ r)

Sets :- T

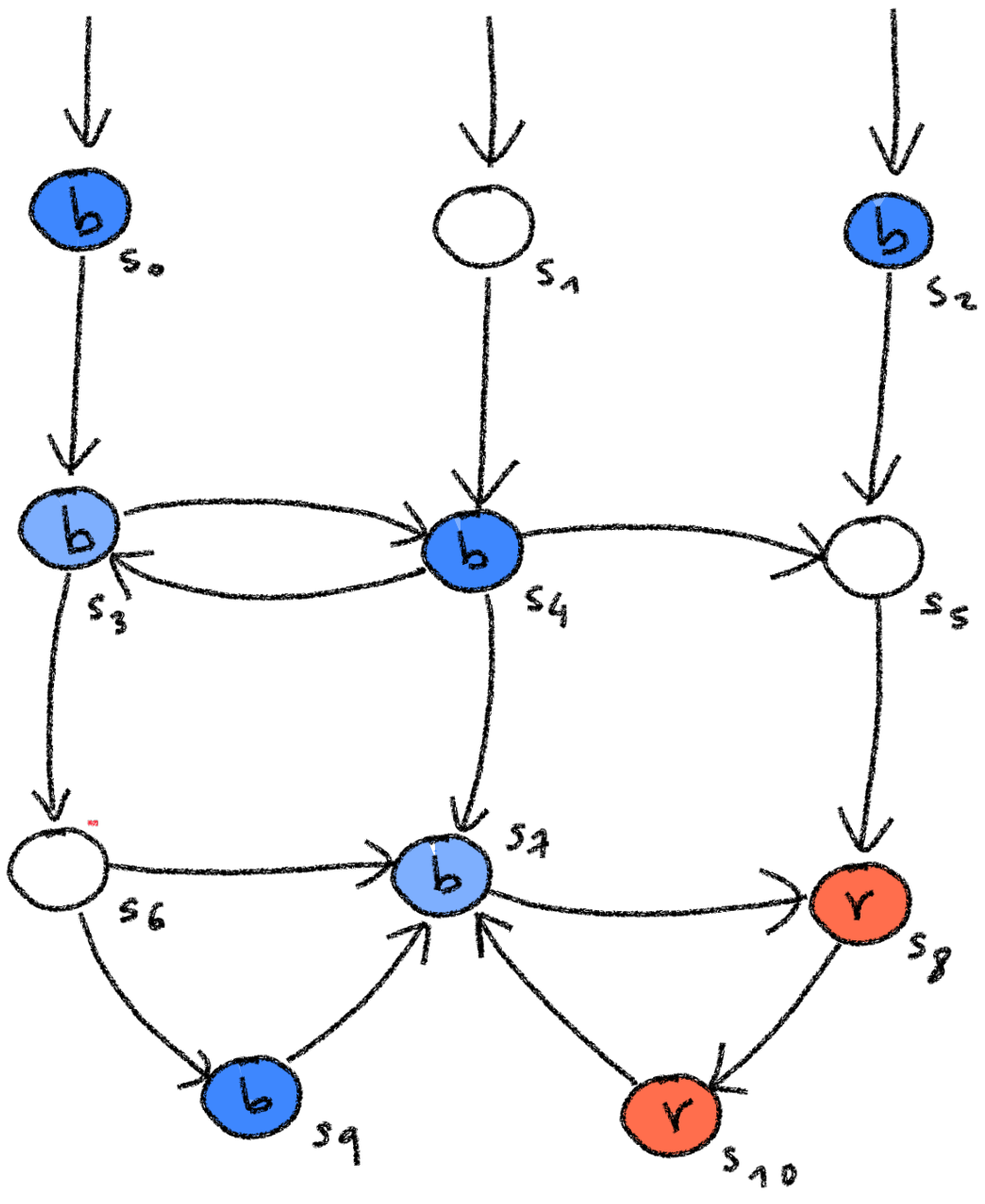
- W

$T := sat(\phi)$; //the set of states to be returned as a result
$W := sat(\phi)$; // working set (states to be explored)
while $W \neq \emptyset$ do
    remove some s from W;
    foreach $s' \in Pre(s)$ do
        if $s' \notin T$ then
            $T := T \cup s'$;
            $W := W \cup s'$;
return T;

Let us now extend what we saw to the general case of the until operator

Let's start with an example

How would you compute the sat(E b U r) in this TS?



Expansion law for EU

The expansion law for EU

$$\exists \phi_1 \mathsf{U} \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists \phi_1 \mathsf{U} \phi_2)$$

Provides a recursive definition of `sat`

$$\begin{aligned}
sat(\exists \phi_1 \mathsf{U} \phi_2) &\equiv sat(\phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists \phi_1 \mathsf{U} \phi_2)) \\
&\equiv sat(\phi_2) \cup sat((\phi_1 \wedge \exists \bigcirc \exists \phi_1 \mathsf{U} \phi_2)) \\
&\equiv sat(\phi_2) \cup (sat(\phi_1) \cap sat(\exists \bigcirc \exists \phi_1 \mathsf{U} \phi_2)) \\
&\equiv sat(\phi_2) \cup (sat(\phi_1) \cap \{s \in S \mid Post(s) \cap sat(\exists \phi_1 \mathsf{U} \phi_2) \neq \emptyset\}) \\
&\equiv sat(\phi_2) \cup \{s \in sat(\phi_1) \mid Post(s) \cap sat(\exists \phi_1 \mathsf{U} \phi_2) \neq \emptyset\}
\end{aligned}$$

A fix point computation

From our recursive definition

$$sat(\exists \phi_1 \mathsf{U} \phi_2) \equiv sat(\phi_2) \cup \{s \in sat(\phi_1) \mid Post(s) \cap sat(\exists \phi_1 \mathsf{U} \phi_2) \neq \emptyset\}$$

We know that we look for a set T such that

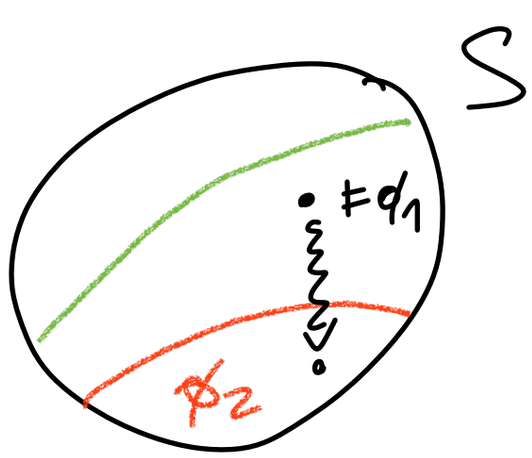
(1) $T \supseteq sat(\phi_2)$

A fix point computation

From our recursive definition

$$sat(\exists \phi_1 \mathsf{U} \phi_2) \equiv sat(\phi_2) \cup \{s \in sat(\phi_1) \mid Post(s) \cap sat(\exists \phi_1 \mathsf{U} \phi_2) \neq \emptyset\}$$

We know that we look for a set T such that



(1) $T \supseteq sat(\phi_2)$

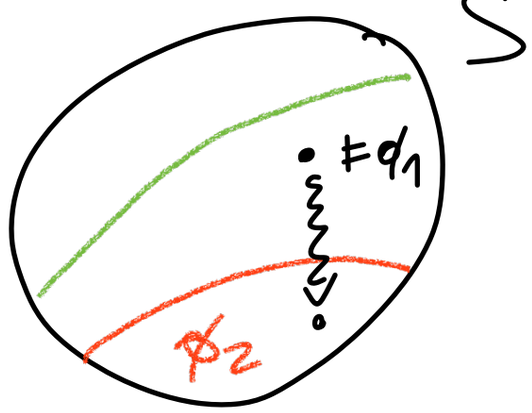
(2) $s \in sat(\phi_1)$ and $Post(s) \cap T \neq \emptyset$ implies $s \in T$

A fix point computation

From our recursive definition

$$sat(\exists \phi_1 \mathsf{U} \phi_2) \equiv sat(\phi_2) \cup \{s \in sat(\phi_1) \mid Post(s) \cap sat(\exists \phi_1 \mathsf{U} \phi_2) \neq \emptyset\}$$

We know that we look for a set T such that



(1) $T \supseteq sat(\phi_2)$

(2) $s \in sat(\phi_1)$ and $Post(s) \cap T \neq \emptyset$ implies $s \in T$

It can be shown that $sat(\exists \phi_1 \mathsf{U} \phi_2)$

is the smallest set T satisfying (1) and (2)

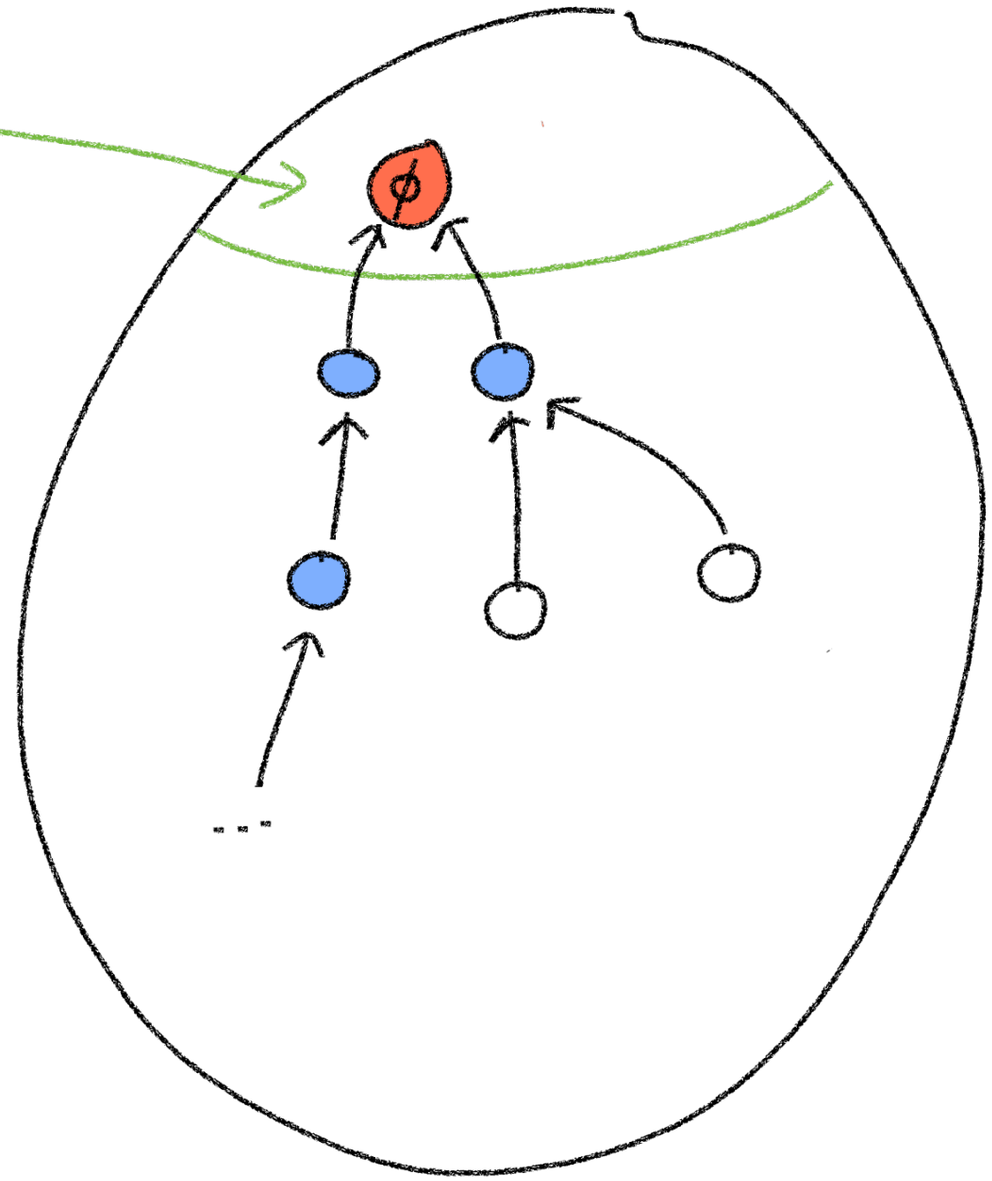
How would you prove it?
(Solution in the book)

What would be the largest set T satisfying (1) and (2)?

A fix point computation

We can then just perform a standard least fix point calculation

$T_0 := sat(\phi)$



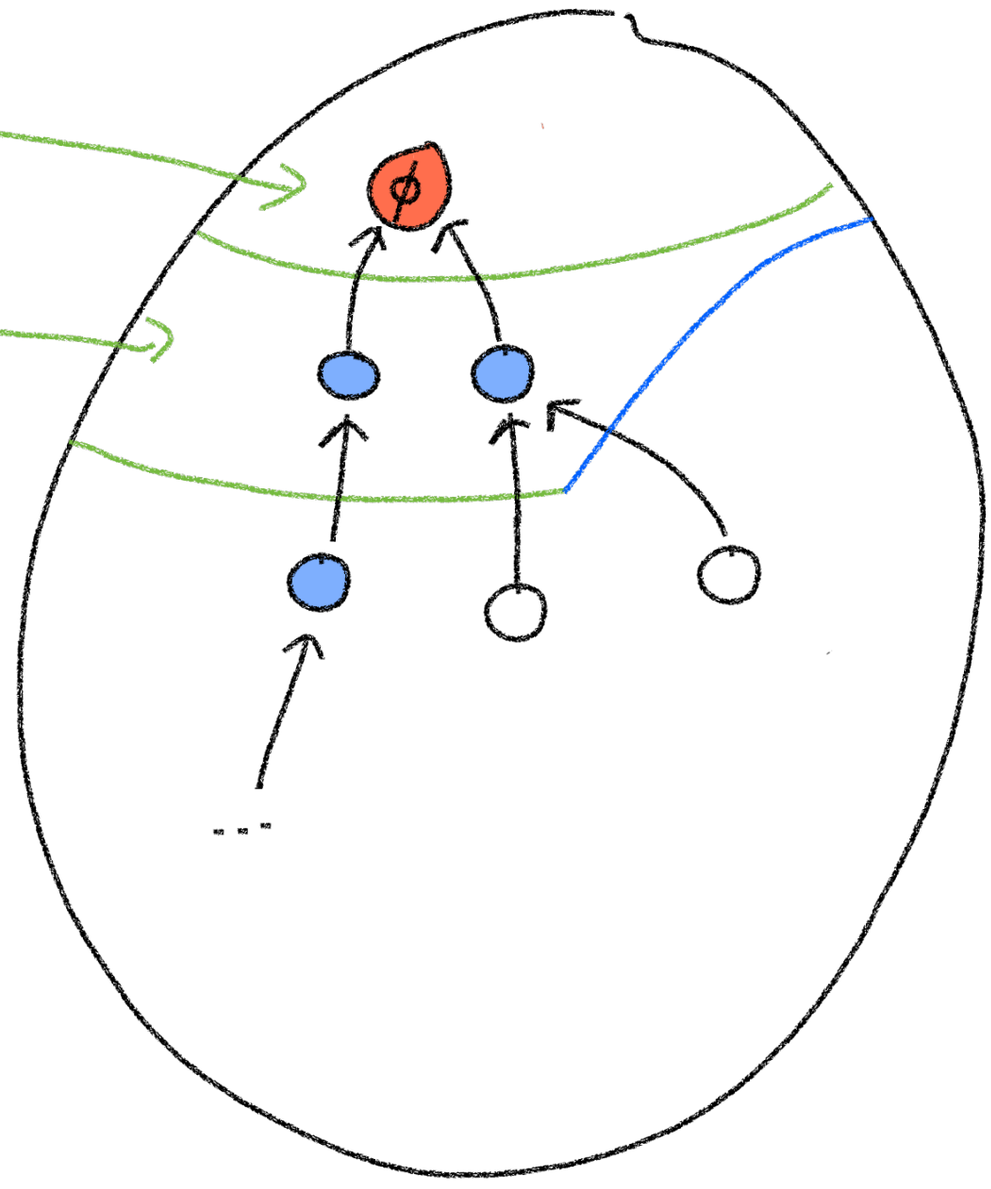
This is essentially a backwards reachability search,
but when going back we are only allow to follow states satisfying phi_1 (blue states)

A fix point computation

We can then just perform a standard least fix point calculation

$T_0 := sat(\phi)$

$T_1 := T_0 \cup \{s \in sat(\phi_1) \setminus T_0 \mid Post(s) \cap T_0 \neq \emptyset\}$



This is essentially a backwards reachability search,
but when going back we are only allow to follow states satisfying phi_1 (blue states)

A fix point computation

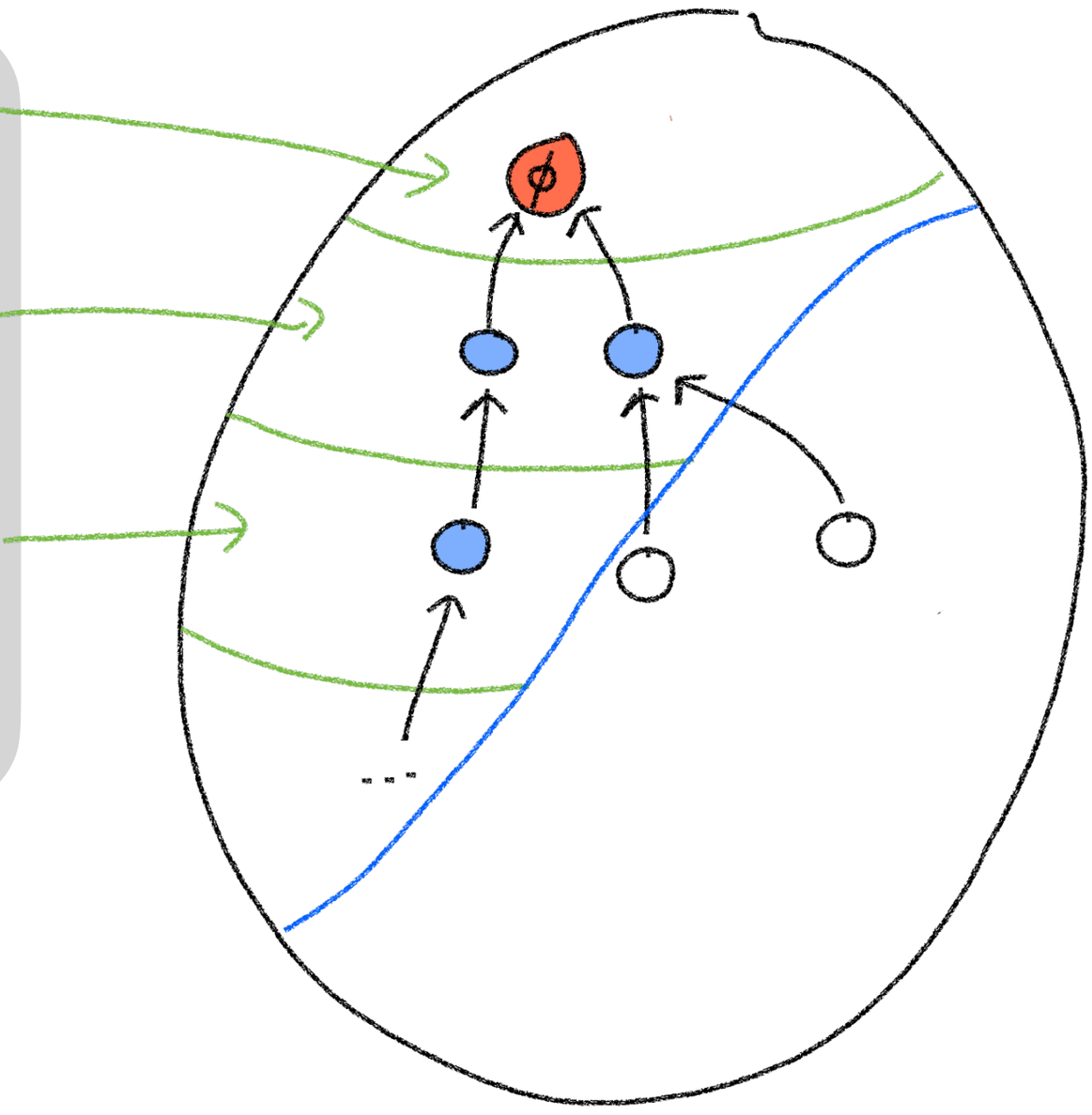
We can then just perform a standard least fix point calculation

$T_0 := sat(\phi)$

$T_1 := T_0 \cup \{s \in sat(\phi_1) \setminus T_0 \mid Post(s) \cap T_0 \neq \emptyset\}$

$T_2 := T_1 \cup \{s \in sat(\phi_1) \setminus T_1 \mid Post(s) \cap T_1 \neq \emptyset\}$

...



This is essentially a backwards reachability search,
but when going back we are only allow to follow states satisfying phi_1 (blue states)

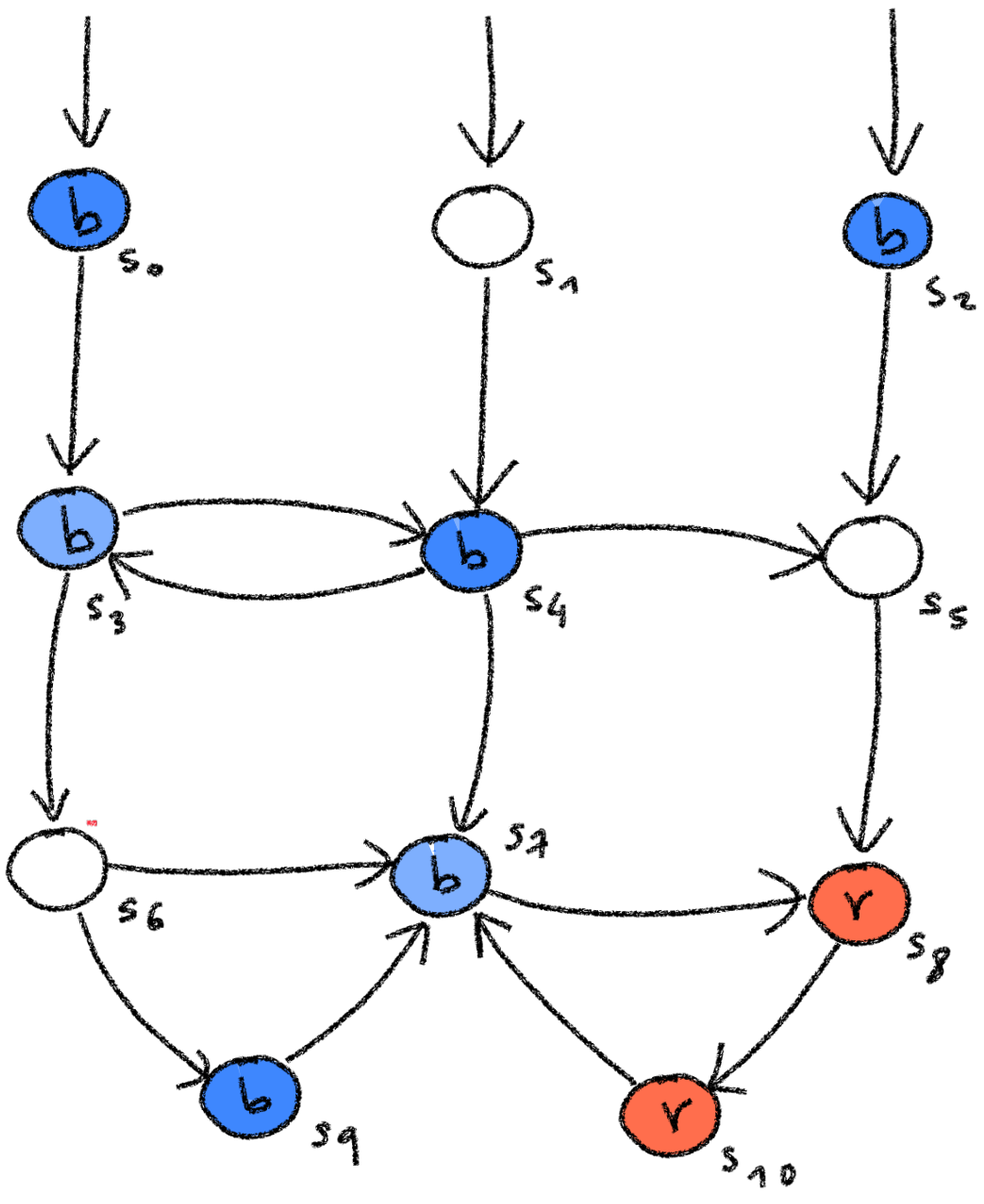
Algorithm for EU

Finally, the algorithm…

$T := sat(\phi_2)$;
$W := sat(\phi_2)$;
while $W \neq \emptyset$ do
    remove some s from W;
    foreach $s' \in Pre(s)$ do
        if $s' \notin T$ and $s' \in$ sat(ϕ_1) then
            $T := T \cup s'$;
            $W := W \cup s'$;
return T;

Can you spot the diff w.r.t to the algorithm for sat(EF…)?

An example

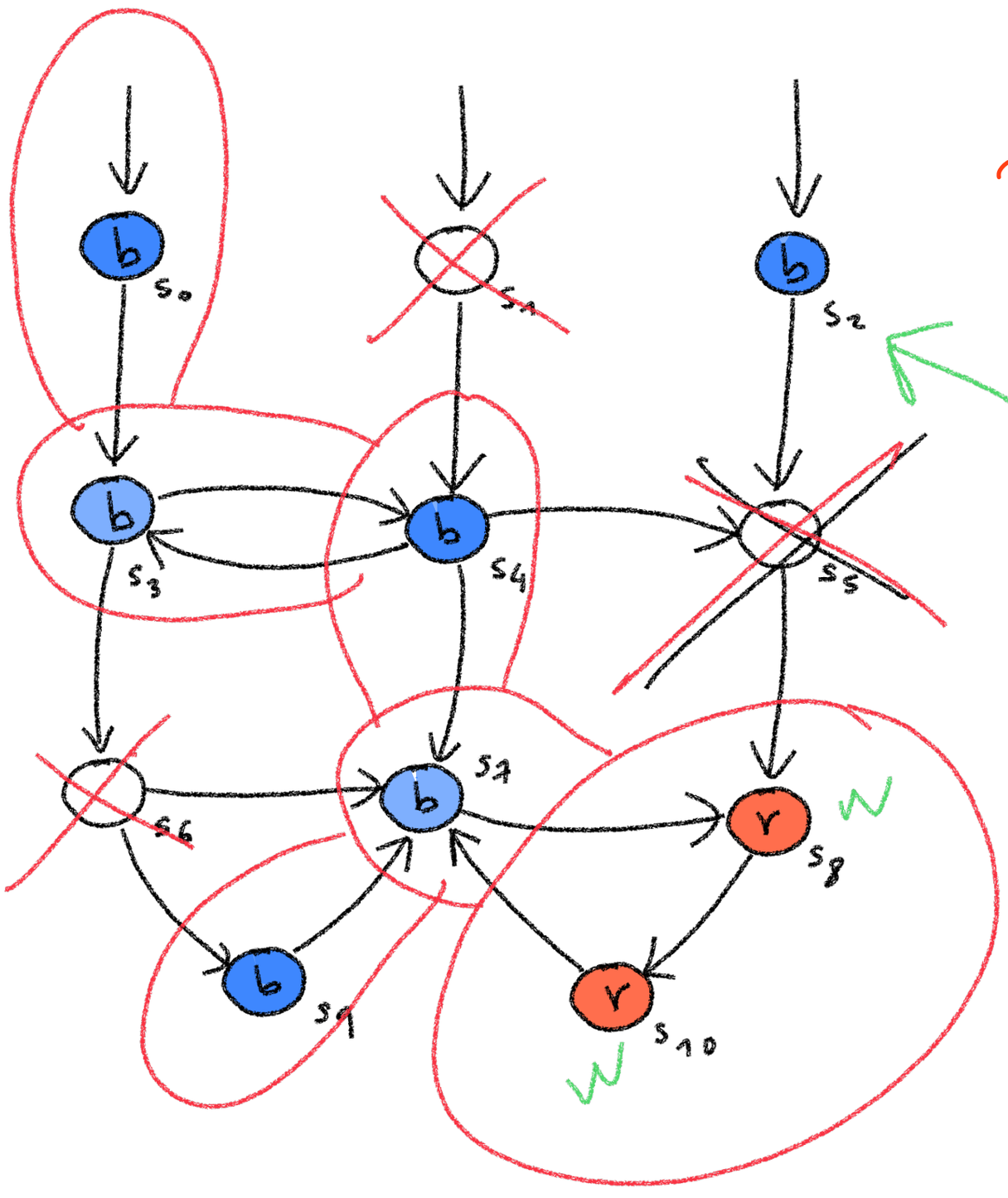


$sat(\exists(b \cup r))$

Sets :- T

- W

An example

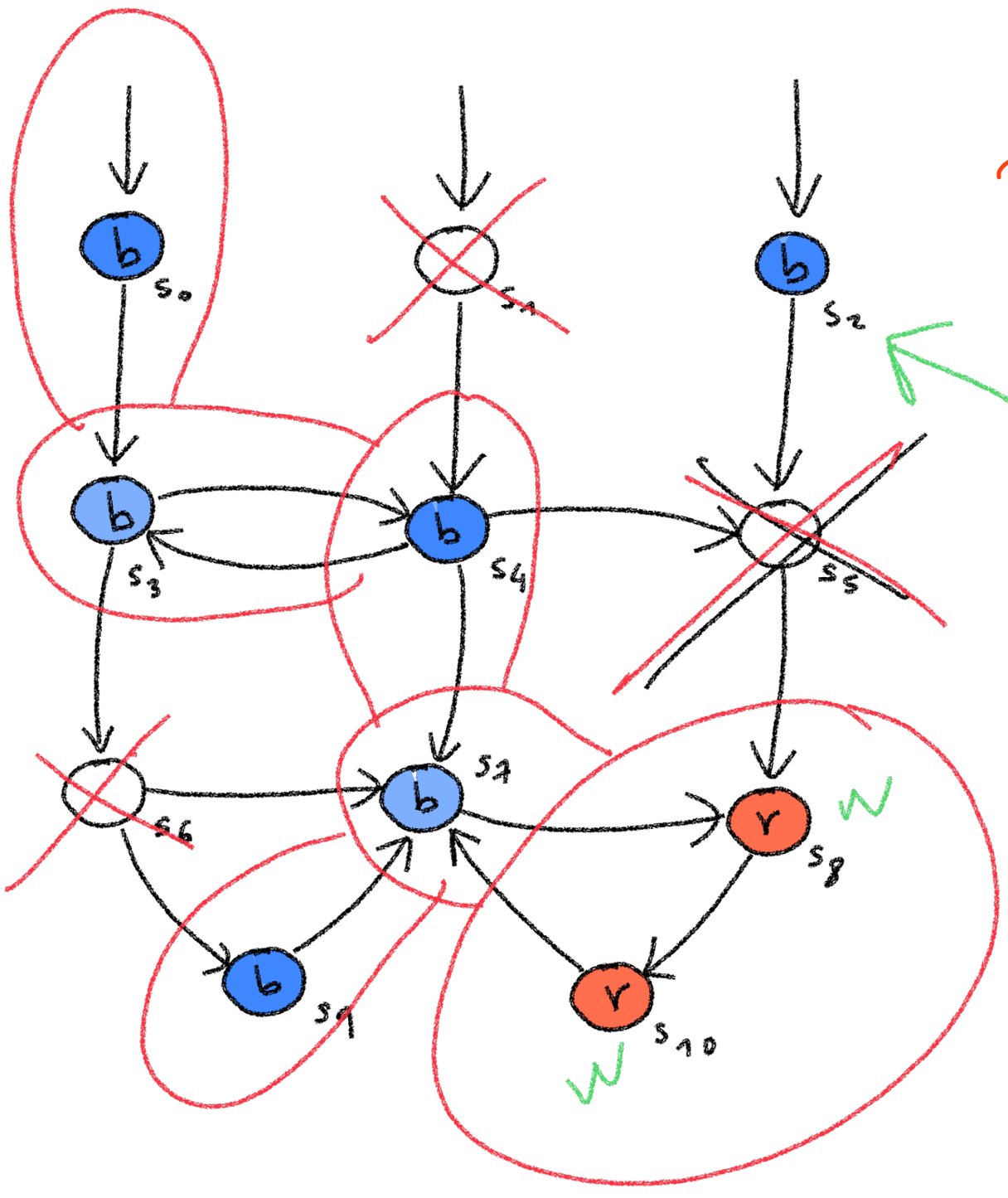


$sat(\exists(b \cup r))$

Sets :- T

- W

An example



$sat(\exists(b \cup r))$

Sets :- T

- W

Recall:

$T \models \phi$ iff $\forall s \in I . s \models \phi$

Thus this formula doesn't hold on this TS

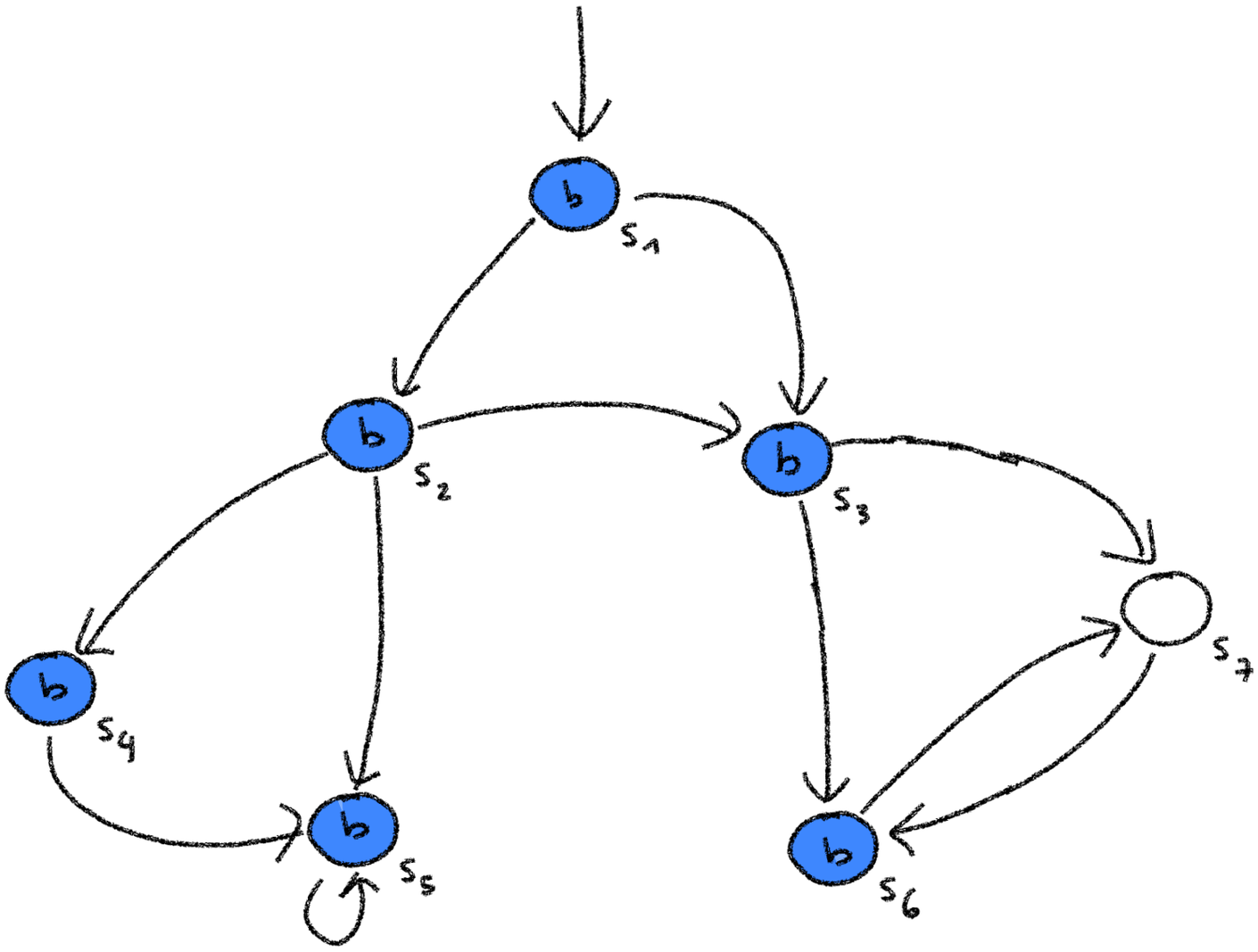
Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity

Perhaps unsurprisingly we can tell a similar story for EG

How would you compute the sat(EGb) in this TS?

$sat(\exists \Box b)$



Expansion law for EG

The expansion law for EG

$$\exists \Box \phi \equiv \phi \wedge \exists \bigcirc \exists \Box \phi$$

Provides a recursive definition of `sat`

$$sat(\exists \Box \phi) \equiv sat(\phi \wedge \exists \bigcirc \exists \Box \phi)$$

(applying expansion law)

$$\equiv sat(\phi) \cap sat(\exists \bigcirc \exists \Box \phi)$$

(def. of sat(.../\...))

$$\equiv sat(\phi) \cap \{s \in S \mid Post(s) \cap sat(\exists \Box phi) \neq \emptyset\}$$

(def. of sat(EX...))

$$\equiv \{s \in sat(\phi) \mid Post(s) \cap sat(\exists \Box phi) \neq \emptyset\}$$

(sat(phi) is a subset of S)

A fix point computation

From our recursive definition

$$sat(\exists \Box \phi) \equiv \{s \in sat(\phi) \mid Post(s) \cap sat(\exists \Box \phi) \neq \emptyset\}$$

We know that we look
for a set T such that

A fix point computation

From our recursive definition

$$sat(\exists \Box \phi) \equiv \{s \in sat(\phi) \mid Post(s) \cap sat(\exists \Box \phi) \neq \emptyset\}$$

We know that we look for a set T such that

(1) $T \subseteq sat(\phi)$

A fix point computation

From our recursive definition

$$sat(\exists \Box \phi) \equiv \{s \in sat(\phi) \mid Post(s) \cap sat(\exists \Box \phi) \neq \emptyset\}$$

We know that we look for a set T such that

(1) $T \subseteq sat(\phi)$

(2) $s \in T$ implies $Post(s) \cap T \neq \emptyset$

A fix point computation

From our recursive definition

$$sat(\exists \Box \phi) \equiv \{s \in sat(\phi) \mid Post(s) \cap sat(\exists \Box \phi) \neq \emptyset\}$$

We know that we look for a set T such that

(1) $T \subseteq sat(\phi)$

(2) $s \in T$ implies $Post(s) \cap T \neq \emptyset$

It can be shown that $sat(\exists \Box \phi)$

is the largest set T satisfying (1) and (2)

What would be the smallest set T satisfying (1) and (2)?

How would you prove it?
(Solution in the book)

A fix point computation

We can then just perform a standard greatest fix point calculation

for GFP we're going to remove things

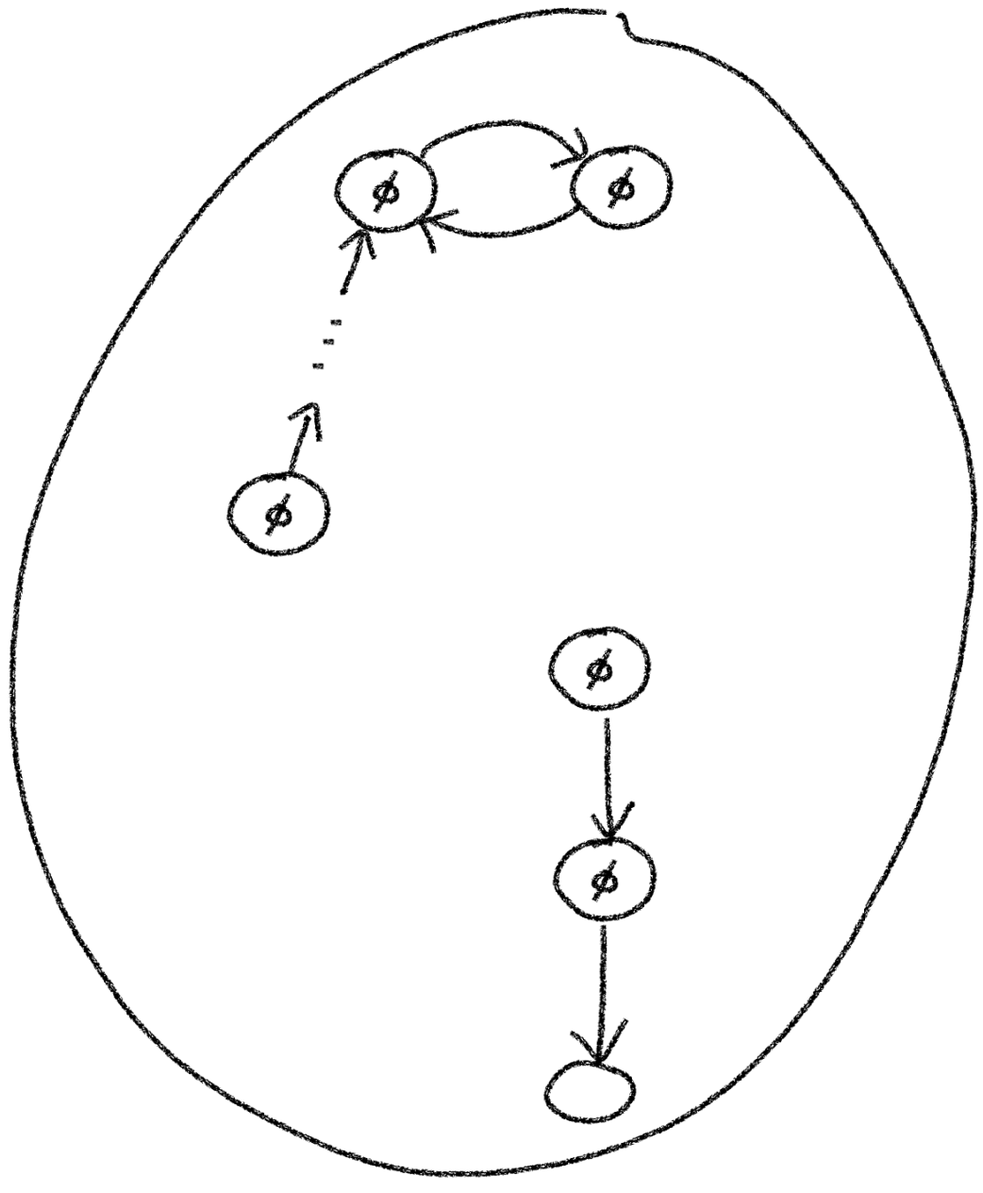
...

$$T_2 := T_1 \backslash \{s \in T_1 \mid Post(s) \cap T_1 = \emptyset\}$$

Remove successors who have a non-empty intersection with the previous state

$$T_1 := T_0 \backslash \{s \in T_0 \mid Post(s) \cap T_0 = \emptyset\}$$

$$T_0 := sat(\phi)$$



A fix point computation

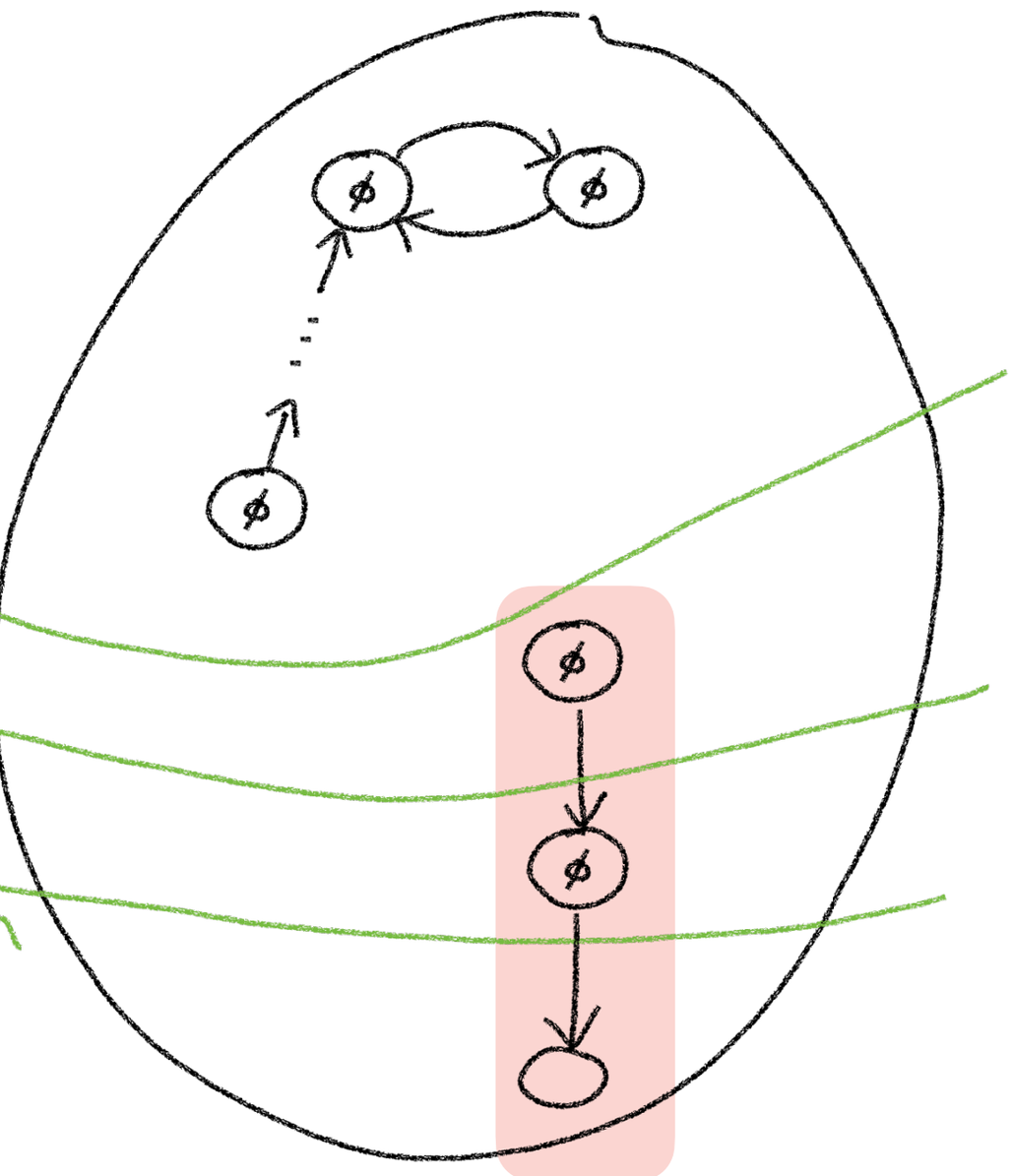
We can then just perform a standard greatest fix point calculation

…

$T_2 := T_1 \backslash \{s \in T_1 \mid Post(s) \cap T_1 = \emptyset\}$

$T_1 := T_0 \backslash \{s \in T_0 \mid Post(s) \cap T_0 = \emptyset\}$

$T_0 := sat(\phi)$



Finally, the algorithm…

$T := sat(\phi)$; //the set of states to be returned as a result

$W := S \backslash sat(\phi)$; //working set (states to be explored)

while $W \neq \emptyset$ do

    remove some s from W;

    foreach $s' \in Pre(s)$ do

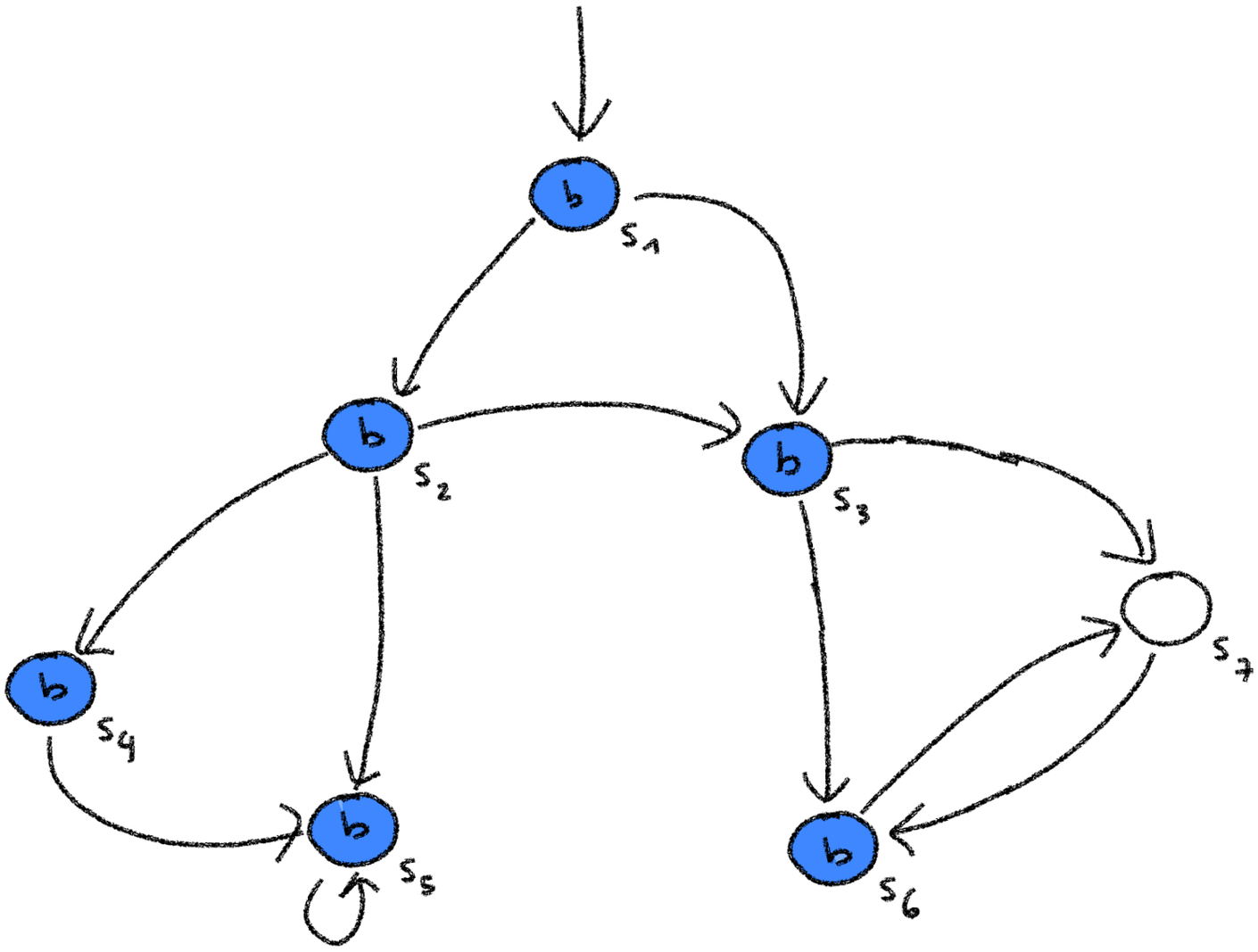
        if $s' \in T$ and $Post(s') \cap T = \emptyset$ then

            $T := T \backslash s'$;

            $W := W \cup s'$;

return T;

An example

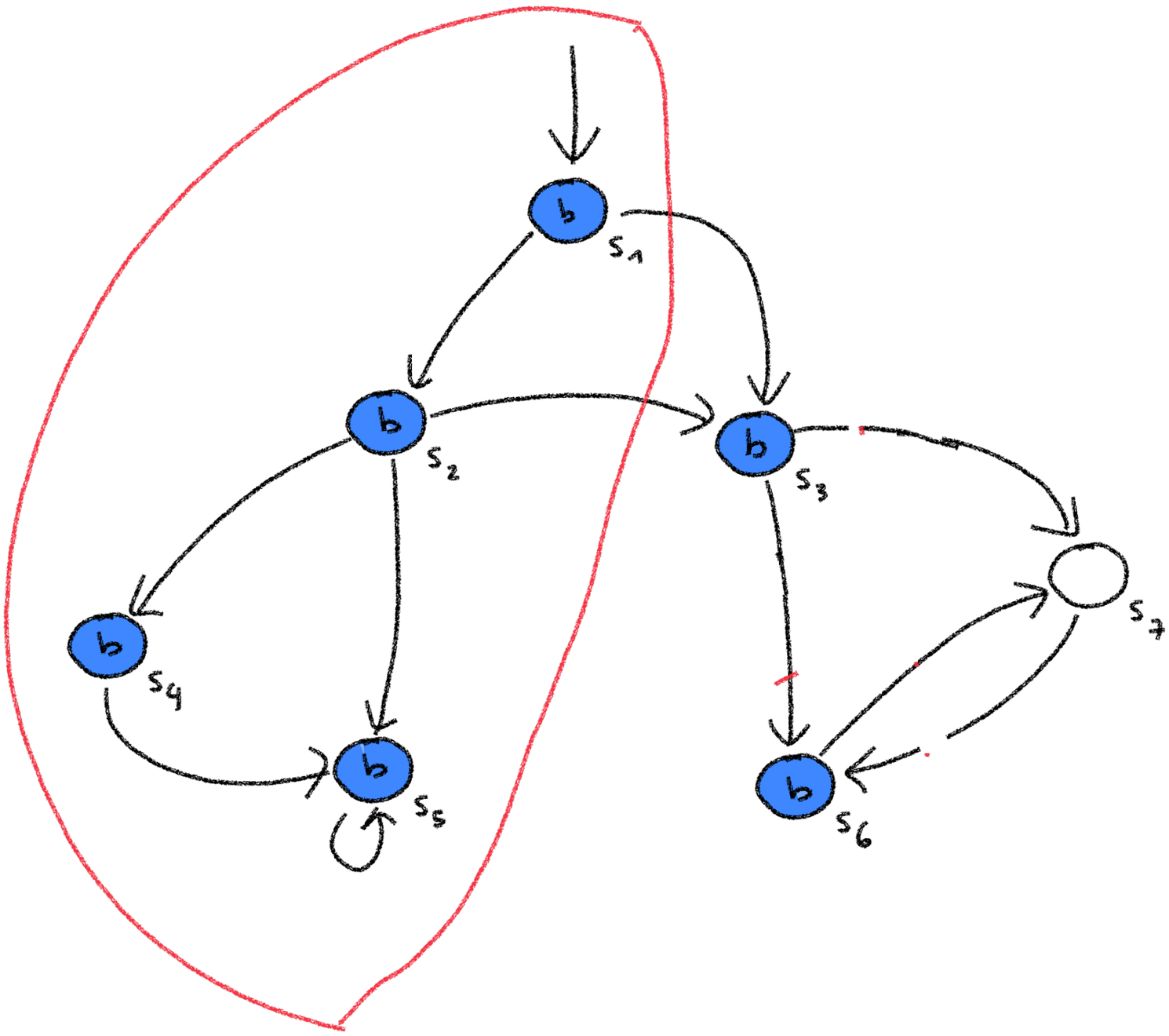


sat $(\exists \Box b)$

$T := sat(\phi)$; //the set of states to be returned as a result
$W := S \setminus sat(\phi)$; // working set (states to be explored)
while $W \neq \emptyset$ do
    remove some s from W;
    foreach $s' \in Pre(s)$ do
        if $s' \in T$ and $Post(s') \cap T = \emptyset$ then
            $T := T \setminus s'$;
            $W := W \cup s'$;
return T;

Sets :
- T
- W

An example



sat (∃ □ b)

$T := sat(\phi)$; //the set of states to be returned as a result
$W := S \setminus sat(\phi)$; // working set (states to be explored)
while $W \neq \emptyset$ do
    remove some s from W;
    foreach $s' \in Pre(s)$ do
        if $s' \in T$ and $Post(s') \cap T = \emptyset$ then
            $T := T \setminus s'$;
            $W := W \cup s'$;
return T;

Sets :
- T
- W

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Overall sat(…) algorithm

We have now completed all cases of our algorithm:

```
sat(true) = …
sat(not phi) = …
sat(phi1 or phi2) = …
sat(EX phi) = …
sat(EG phi) = …
sat(E(phi1 U phi2) = …
```

The overall complexity is linear in the size of the transition system and the formula.

Idea:

- One call to sat(…) per sub-formula -> polynomial time in the size of the formula
- Every sat(…) algorithm uses polynomial time in the size of the transition system:
  - Set operations used in the algorithms can be run in polynomial time.
  - In sat(EG…), sat (E…U…) a state can only be once in the worklist W (once removed, it can't be added)

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

Key points of this lecture

Global model checking algorithm, recursive on state sub-formulas (i.e. bottom-up on the AST).

Satisfaction set characterisations of CTL formulas in existential normal form.

Dedicated algorithms for CTL formulas in existential normal form, which exploit the satisfaction set characterisations.

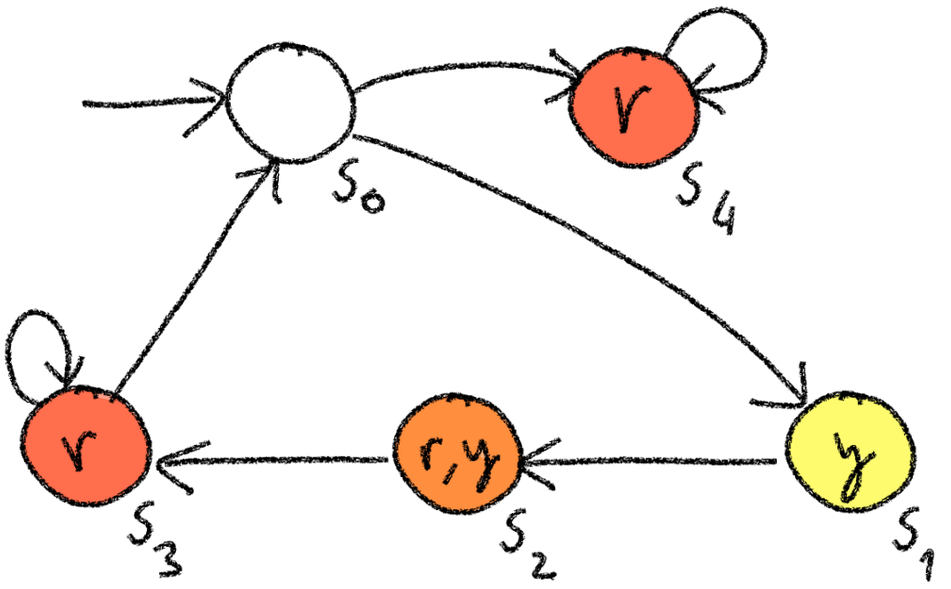
The complexity of CTL model checking is polynomial in the size of the transition system and the CTL formula.

Computation Tree Logic (CTL)

- Reading material
- Formal semantics and naïve algorithms
- Existential normal form CTL (ECTL)
- Global model checking algorithm
- Satisfaction sets characterisation of ECTL
- MC algorithms for basic cases
- MC algorithms for EU
- MC algorithms for EG
- Overall complexity
- Exercises & Homework

APPENDIX: Exercises

Exercise 04.1



For the transition system above, use the model checking algorithms seen in this lecture to decide if it satisfies the following formulas.

(1) $\exists \Box r$

(2) $\exists \bigcirc \forall \Box r$

(3) $\forall (y \mathsf{U} r)$

Check your solutions using PRISM

Exercise 04.2

Design an algorithm to compute the below satisfaction set directly (without transforming it into ECTL).

$$sat(\forall \Diamond \phi))$$

HINT: You can modify one of the algorithms seen in class.

Argue for its correctness.