Unsolvable Problems

Part One

What problems can we solve with a computer?

What does it mean to solve a problem?

Self-Reference :Danger:

- Last time, we saw examples of self-reference that were fine, and some that created paradoxes:

True or False?

```
"This string is 34 characters long."
"This sentence is false."
"This sentence is true."
```

Proofs by Contradiction in Number Theory

- One way to think about proofs by contradiction is that they lead to a kind of “impossible” situation that is similar to the paradoxes. Here is a simple example:
  - **Thm.** There is no greatest integer.
  - **Proof, by contradiction.** Assume for the sake of contradiction that there is a greatest integer, call it g.
  - *[Now we will use g to write a mathematical expression that is a syntactically valid mathematical expression that should be fine to write,* ***if* g were real***.]*
  - Let $x = g + 1$.
  - We see that $x > g$.
  - But this is a contradiction, because g is the greatest integer.
  - So the assumption is false and the theorem is true. ■

Proofs by Contradiction in Number Theory

- One way to think abo
they lead to a kind of
similar to the parado
  - **Thm.** There is no gre
  - **Proof, by contradict**
contradiction that the
  - *[Now we will use g to*
is a syntactically valid mathematical expression that should be fine to write, ***if* g were real***.]*
  - Let $x = g + 1$.
  - We see that $x > g$.
  - But this is a contradiction, because g is the greatest integer.
  - So the assumption is false and the theorem is true. ■

Observation: there is other math we could have done on g that would **not** have led to an impossible situation. (For example, "Let $x = g - 1$.") "Fixing the bug" in the math doesn't actually fix anything, because it wasn't the math that was buggy. It was g itself. The math did what it needed to do to *expose* the problem with g.

A Decider for A_{TM}?

- ***Recall:*** A_{TM} is the language of the universal Turing machine.
- We know that $\langle M, w \rangle \in A_{TM}$ if and only if M accepts w.
- The universal Turing machine U_{TM} is a *recognizer* for A_{TM}. Could we build a *decider* for A_{TM}?

What does this program do?

```
bool willAccept(string program, string input)
{
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, input)) {
      reject();
   } else {
      accept();
   }
}
```

What does this program do?

```
bool willAccept(string program, string input)
{
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, inp
      reject();
   } else {
      accept();
   }
}
```

Try running this program on any input.
What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

We wrote code that breaks basic logic/reality!

- If A_{TM} is decidable, we can construct a TM that determines what it's going to do in the future (whether it will accept its input), then actively chooses to do the opposite.
- This leads to an impossible situation with only one resolution: ***A_{TM} must not be decidable!***

New: writing this up as a proof

Theorem: $A_{TM} \notin \mathbf{R}$.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

In both cases we reach a contradiction, so our assumption must have been wrong.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbf{R}$.

Theorem: $A_{TM} \notin \mathbf{R}$.

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

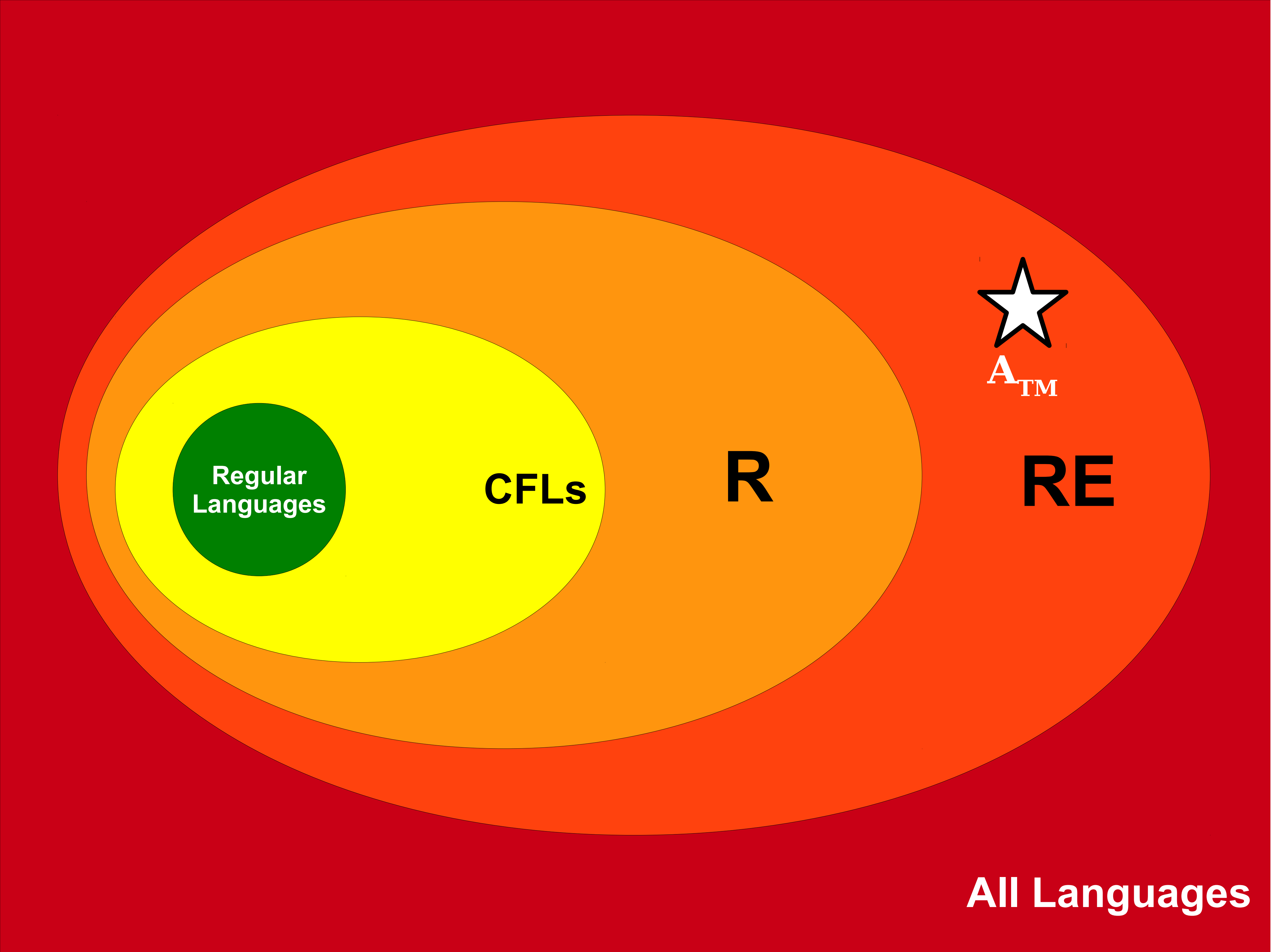
Given this, we could then construct this program P:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) reject();
    else accept();
}
```

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbf{R}$. ■


Regular Languages
CFLs
R
RE
A_{TM}
All Languages

What Does This Mean?

- In one fell swoop, we've proven that
  - A_{TM} is ***undecidable***; there is no general algorithm that can determine whether a TM will accept a string.
  - $\mathbf{R} \neq \mathbf{RE}$, because $A_{TM} \notin \mathbf{R}$ but $A_{TM} \in \mathbf{RE}$.
- What do these two statements really mean? As in, why should you care?

$A_{TM} \notin \mathbf{R}$

- The proof we've done says that

  There is no possible way to design an algorithm that will determine whether a program will accept an input.

- Notice that our proof just assumed there was some decider for A_{TM} and didn't assume anything about how that decider worked. In other words, no matter how you try to implement a decider for A_{TM}, you can never succeed!

$A_{TM} \notin \mathbf{R}$

- At a more fundamental level, the existence of undecidable problems tells us the following:

  There is a difference between what is true and what we can discover is true.

- Given an TM and any string w, either the TM accepts the string or it doesn't – *but there is no algorithm we can follow that will always tell us which it is!*

R ≠ RE

- Because **R ≠ RE**, there are some problems where "yes" answers can be checked, but there is no algorithm for deciding what the answer is.
- ***In some sense, it is fundamentally harder to solve a problem than it is to check an answer.***

More Impossibility Results

The Halting Problem

- The most famous undecidable problem is the ***halting problem***, which asks:

Given a TM *M* and a string *w*,
will *M* halt when run on *w*?

- As a formal language, this problem would be expressed as

***HALT* = { ⟨*M*, *w*⟩ | *M* is a TM that halts on *w* }**

- How hard is this problem to solve?
- How do we know?

HALT ∈ **RE**

- ***Claim:*** *HALT* ∈ **RE**.
- ***Idea:*** If you were certain that a TM *M* halted on a string *w*, could you convince me of that?
- Yes – just run *M* on *w* and see what happens!

```
int main() {
   TM M = getInputTM();
   string w = getInputString();

   feed w into M;
   while (true) {
      if (M is in an accepting state) accept();
      else if (M is in a rejecting state) accept();
      else simulate one more step of M running on w;
   }
}
```

$HALT \notin \mathbf{R}$

- ***Claim:*** $HALT \notin \mathbf{R}$.
- If *HALT* is decidable, we could write some function

```
bool willHalt(string program,
              string input)
```

  that accepts as input a program and a string input, then reports whether the program will halt when run on the given input.
- Then, we could do this…

What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

It loops on the input!

… this program loops on this input?

What does this program do?

```
bool willHalt(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willHalt(me, input)) {
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

Imagine running this program on some input. What happens if...

... this program halts on that input?
It loops on the input!

... this program loops on this input?
It halts on the input!

Theorem: *HALT* $\notin$ **R**.

Proof: By contradiction; assume that *HALT* $\in$ **R**. Then there's a decider *D* for *HALT*, which we can represent in software as a method `willHalt` that takes as input the source code of a program and an input, then returns true if the program halts on the input and false otherwise.

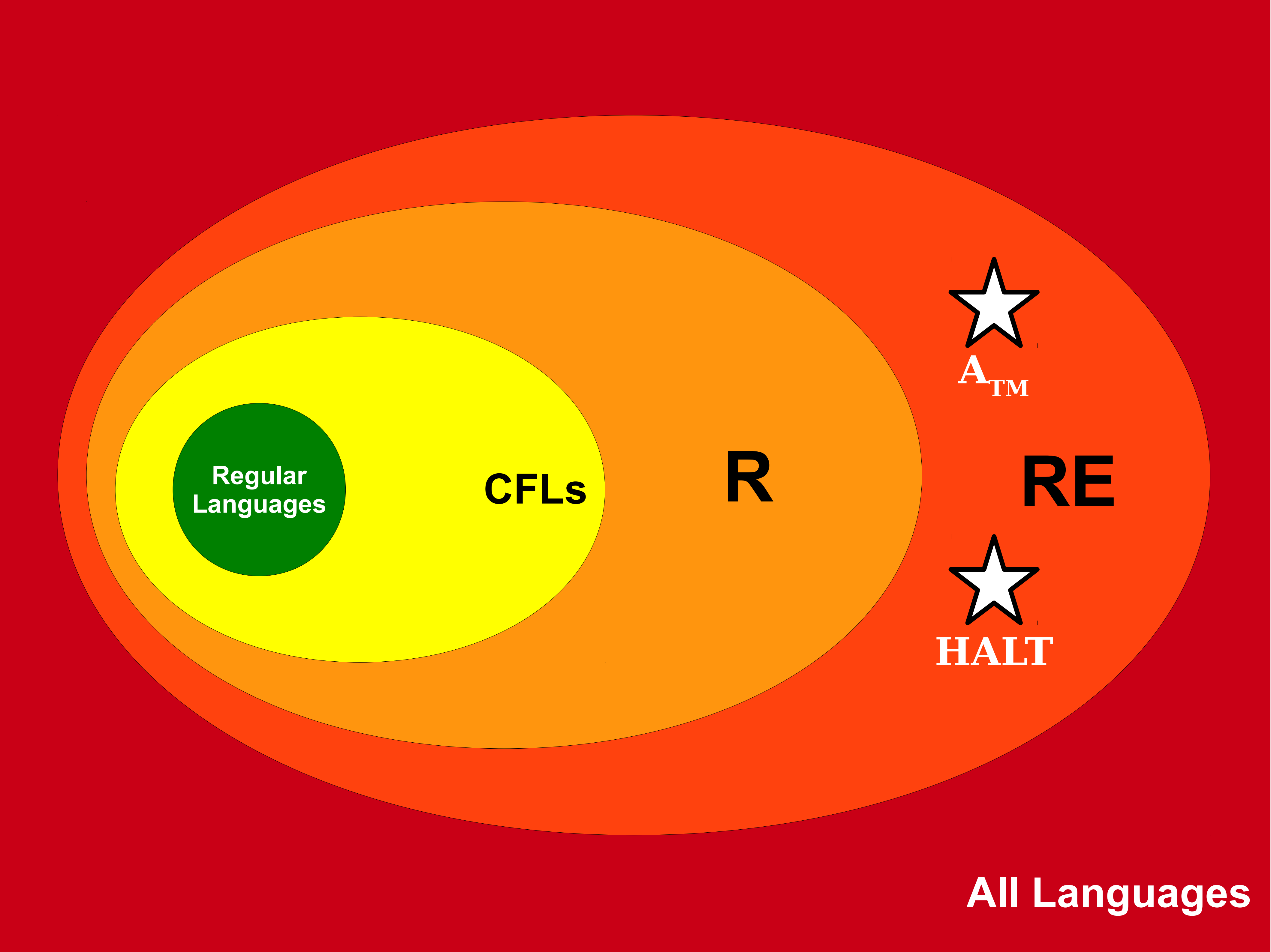
Given this, we could then construct this program *P*:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willHalt(me, input)) while (true) { /* loop! */ }
    else accept();
}
```

Choose any string *w* and trace through the execution of program *P* on input *w*, focusing on the answer given back by the `willHalt` method. If `willHalt(me, input)` returns true, then *P* must halt on its input *w*. However, in this case *P* proceeds to loop infinitely on *w*. Otherwise, if `willHalt(me, input)` returns false, then *P* must not halt its input *w*. However, in this case *P* proceeds to accept its input *w*.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, *HALT* $\notin$ **R**. ■


Regular Languages
CFLs
R
RE
A_{TM}
HALT
All Languages

So What?

- These problems might not seem all that exciting, so who cares if we can't solve them?
- Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.

Beyond **R** and **RE**

Beyond **R** and **RE**

- We've now seen how to use self-reference as a tool for showing undecidability (finding languages not in **R**).
- We still have not broken out of **RE** yet, though.
- To do so, we will need to build up a better intuition for the class **RE**.

What exactly is the class **RE**?

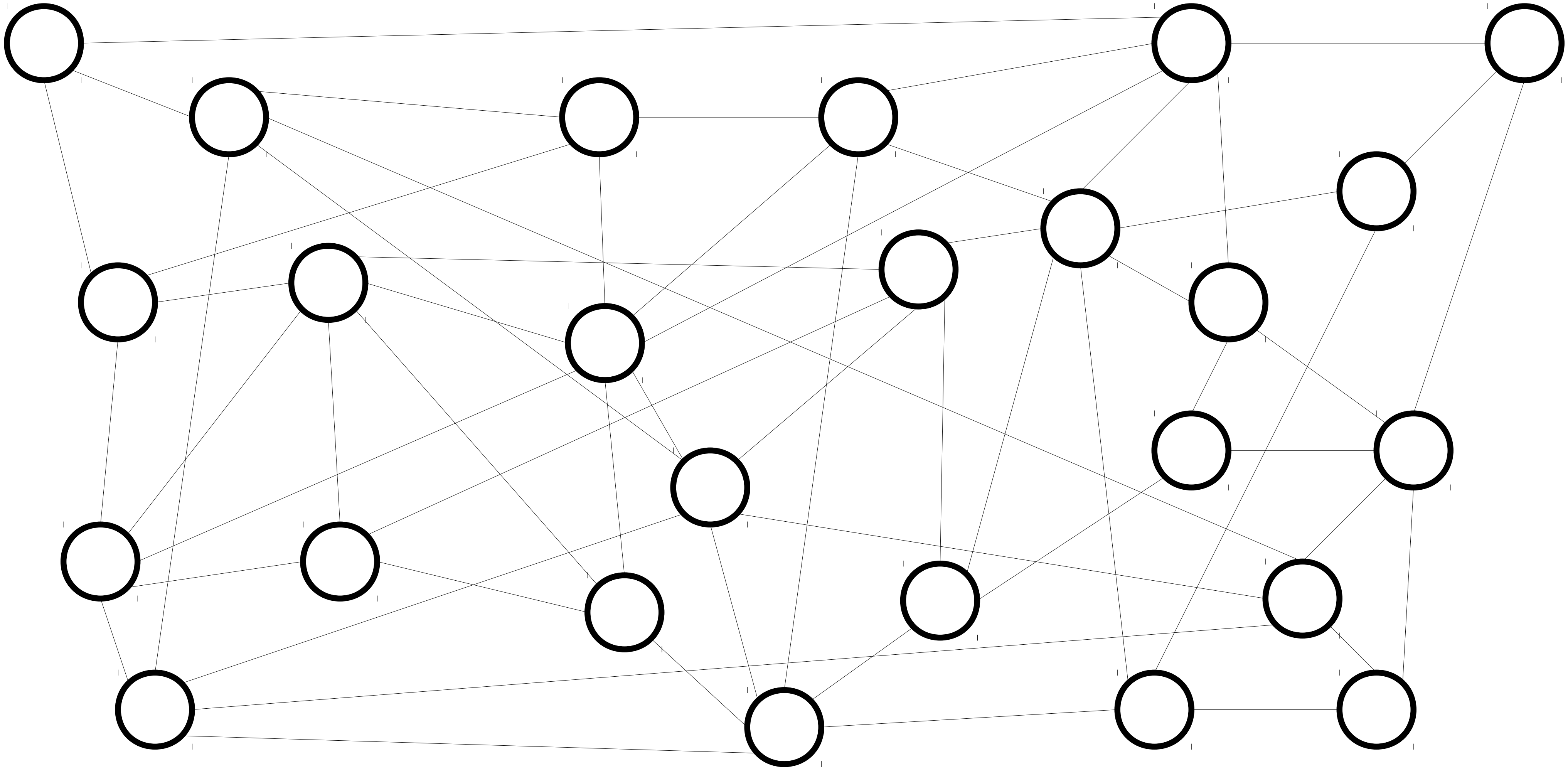
RE, Formally

- Recall that the class **RE** is the class of all recognizable languages:

$$\mathbf{RE} = \{ L \mid \text{there is a TM } M \text{ where } \mathscr{L}(M) = L \}$$

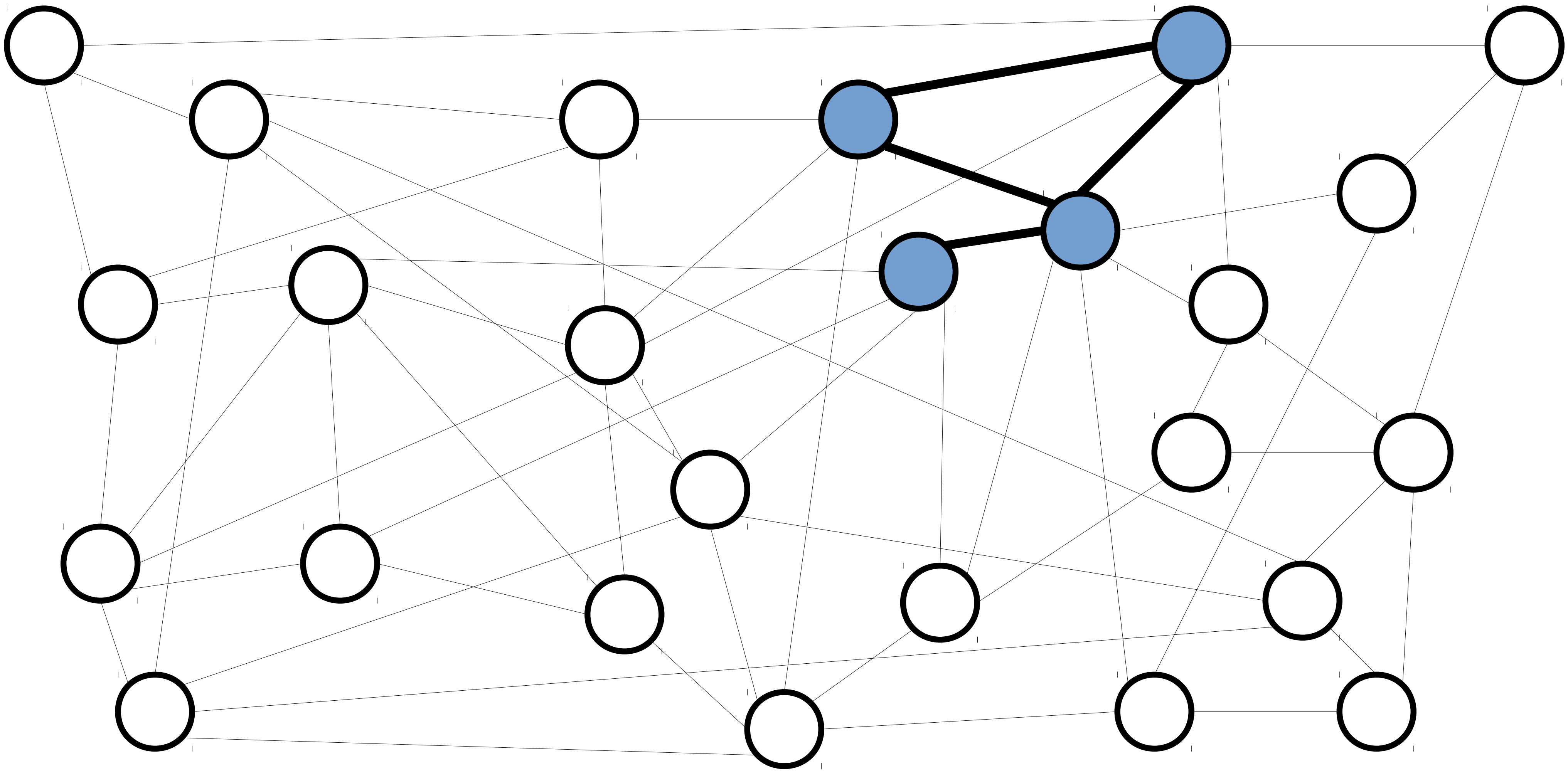
- Since $\mathbf{R} \neq \mathbf{RE}$, there is no general way to "solve" problems in the class **RE**, if by "solve" you mean "make a computer program that can always tell you the correct answer."
- So what exactly *are* the sorts of languages in **RE**?

Get ready to answer
some questions in rapid-fire style!
(about 10 seconds per question)



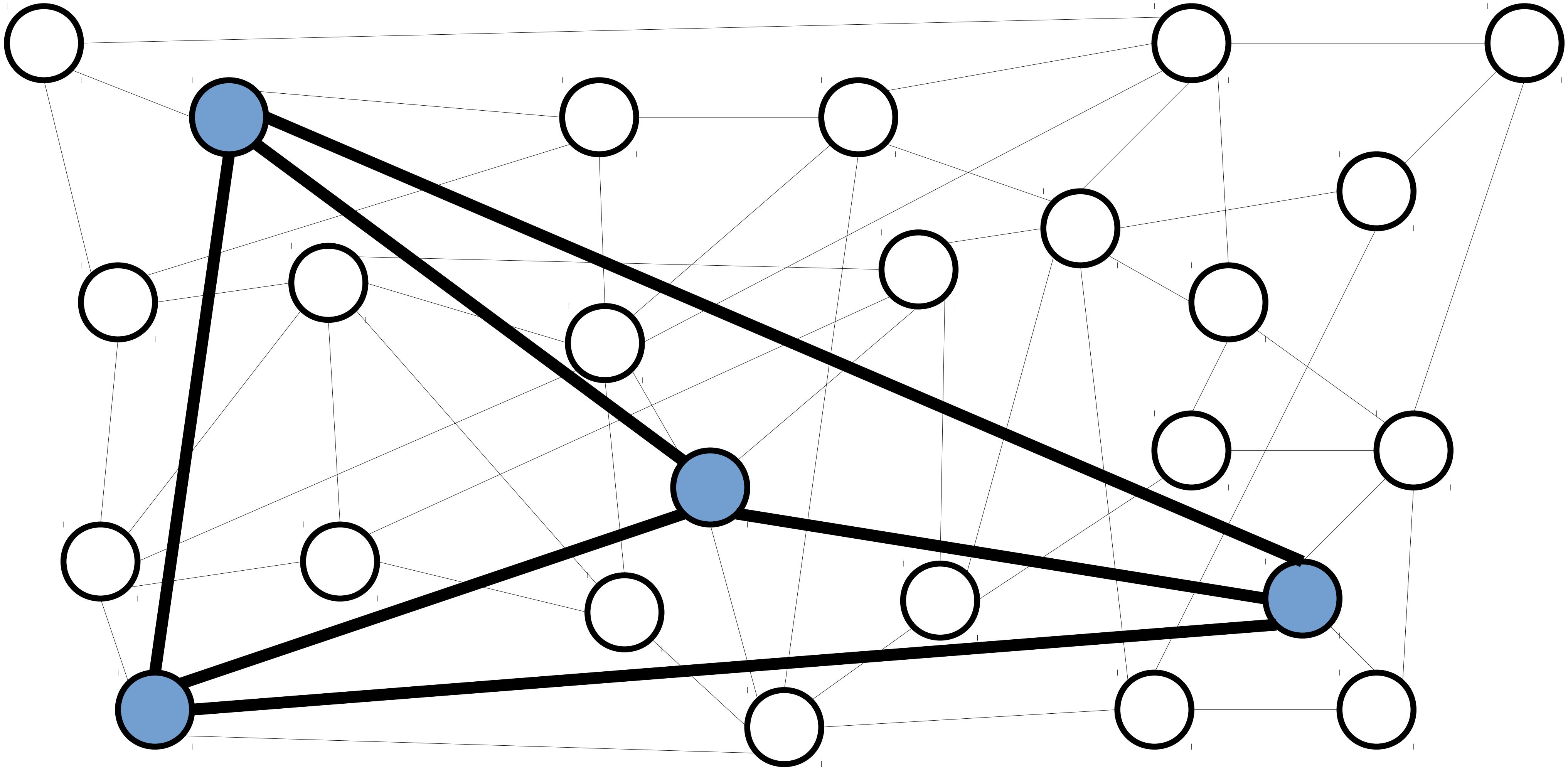
QUICK REACTION: Does this graph contain a 4-clique?

Answer at **PollEv.com/cs103** or text **CS103** to **22333**
once to join, then **Y, N,** or **? (for “I don’t know”)**.



WITH A HINT: Does this graph contain a 4-clique?

Answer at **PollEv.com/cs103** or text **CS103** to **22333**
once to join, then **Y, N,** or **? (for "I don't know")**.



WITH A *NEW* HINT: Does this graph contain a 4-clique?

Answer at **PollEv.com/cs103** or text **CS103** to **22333** once to join, then **Y, N,** or **? (for "I don't know")**.

Key Intuition:

A language L is in **RE** if, for any string w, if you are *convinced* that $w \in L$, there is some piece of evidence you could provide to convince someone else.

Discussion Question:

A language L is in **RE** if, for any string w, if you are *convinced* that $w \in L$, there is some piece of evidence you could provide to convince someone else.

What about for a $w \notin L$? What would a piece of evidence for that look like?

More rapid-fire questions!

(don’t need to vote this time)

Verification

		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

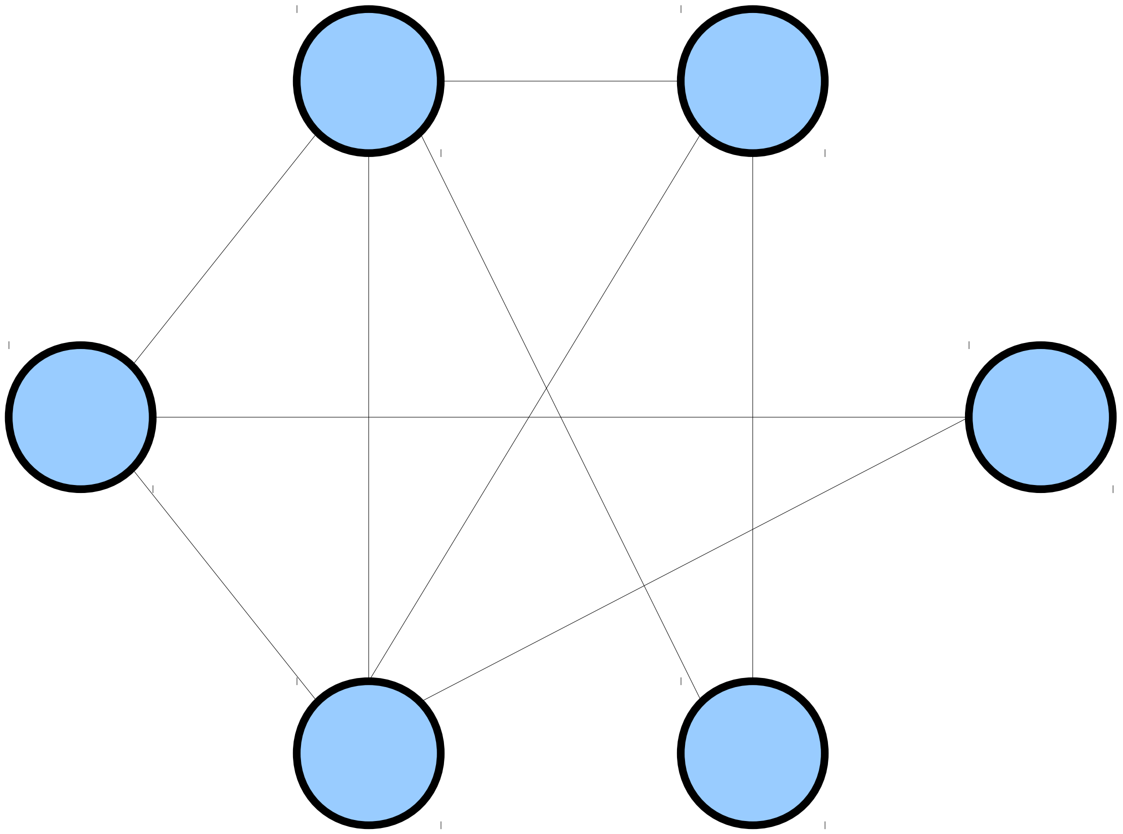
Does this Sudoku puzzle
have a solution?

Verification

2	5	7	9	6	4	1	8	3
4	9	1	8	7	3	6	5	2
3	8	6	1	2	5	9	4	7
6	4	5	7	3	2	8	1	9
7	1	9	5	4	8	3	2	6
8	3	2	6	1	9	5	7	4
1	6	3	2	5	7	4	9	8
5	7	8	4	9	6	2	3	1
9	2	4	3	8	1	7	6	5

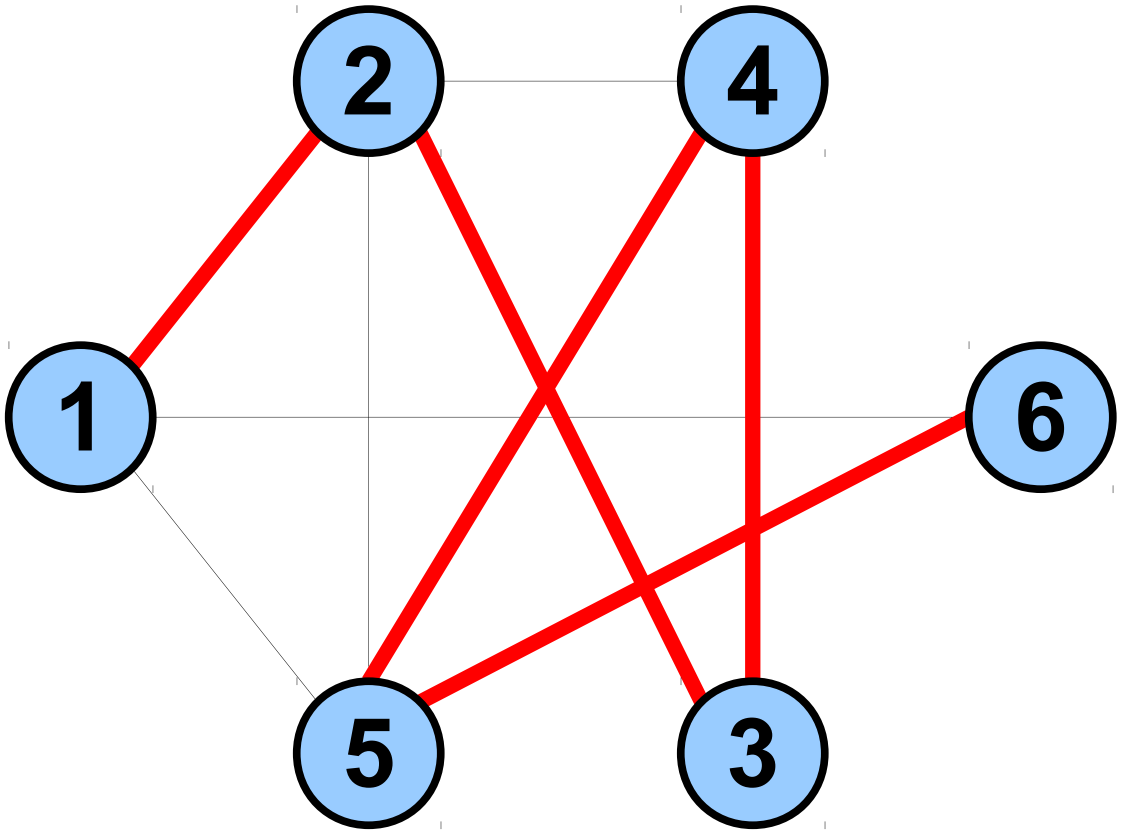
Does this Sudoku puzzle
have a solution?

Verification



Does this graph have a ***Hamiltonian path*** (a simple path that passes through every node exactly once?)

Verification



Does this graph have a ***Hamiltonian path*** (a simple path that passes through every node exactly once?)

Verification

11

Does the hailstone sequence terminate for this number?

Verification

11

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

34

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

17

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

52

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

26

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

13

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

40

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

20

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

10

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

5

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

16

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

8

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

4

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

2

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

1

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

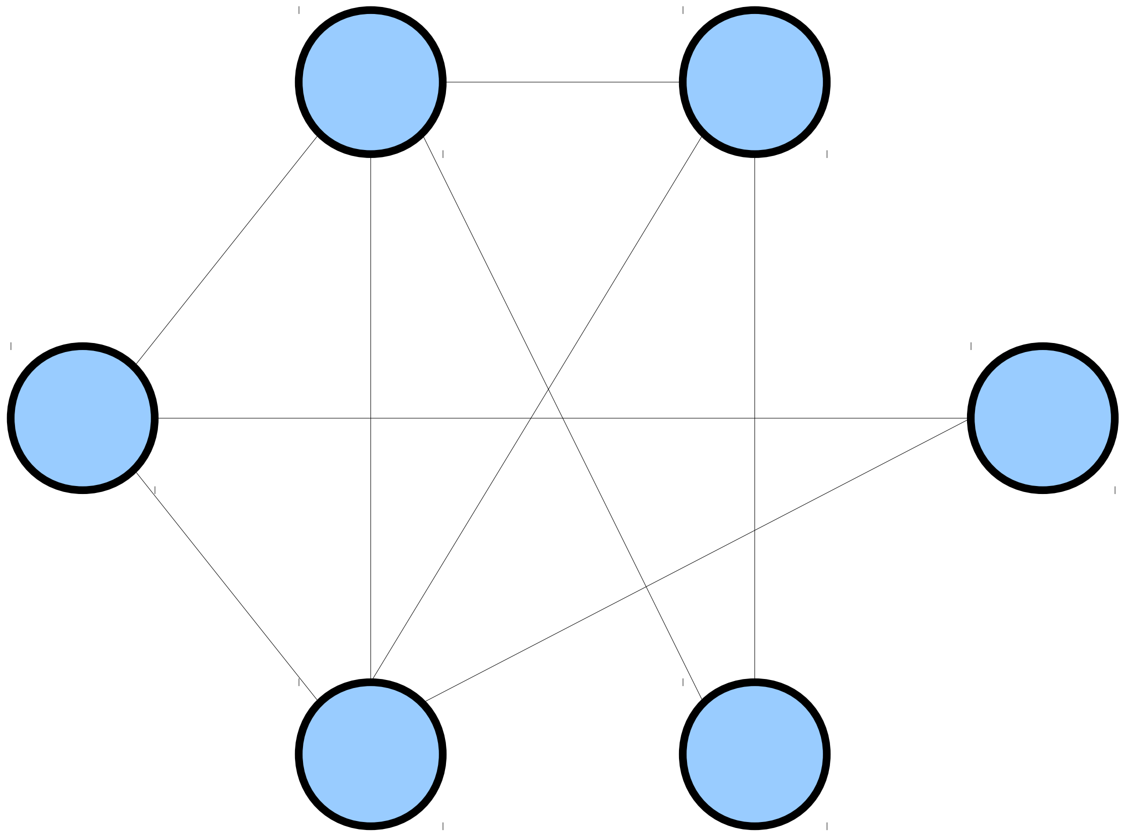
Does this Sudoku puzzle
have a solution?

Verification

1	1	7	1	6	1	1	1	1
1	1	1	1	1	3	1	5	2
3	1	1	1	1	5	9	1	7
6	1	5	1	3	1	8	1	9
1	1	1	1	4	1	1	2	1
8	1	2	1	1	1	5	1	4
1	1	3	2	1	7	1	1	8
5	7	1	4	1	1	1	1	1
1	1	4	1	8	1	7	1	1

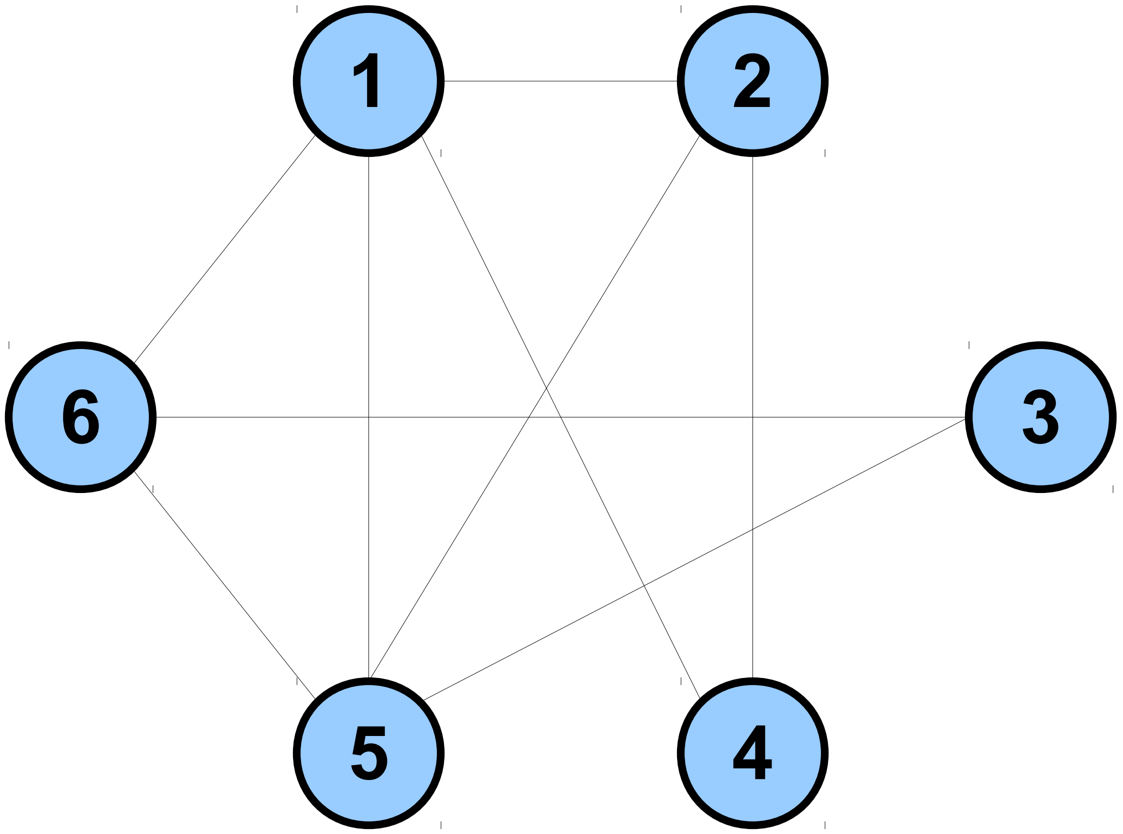
Does this Sudoku puzzle
have a solution?

Verification



Does this graph have a ***Hamiltonian path*** (a simple path that passes through every node exactly once?)

Verification



Does this graph have a ***Hamiltonian path*** (a simple path that passes through every node exactly once?)

Verification

11

Does the hailstone sequence
terminate for this number?

Verification

11

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

34

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

17

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

52

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

26

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

13

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

- In each of the preceding cases, we were given some problem and some evidence supporting the claim that the answer is “yes.”
- Given the correct evidence, we can be certain that the answer is indeed “yes.”
- Given incorrect evidence, we aren't sure whether the answer is “yes.”
    - Maybe there's *no* evidence saying that the answer is “yes,” or maybe there is some evidence, but just not the evidence we were given.
- Let's formalize this idea.

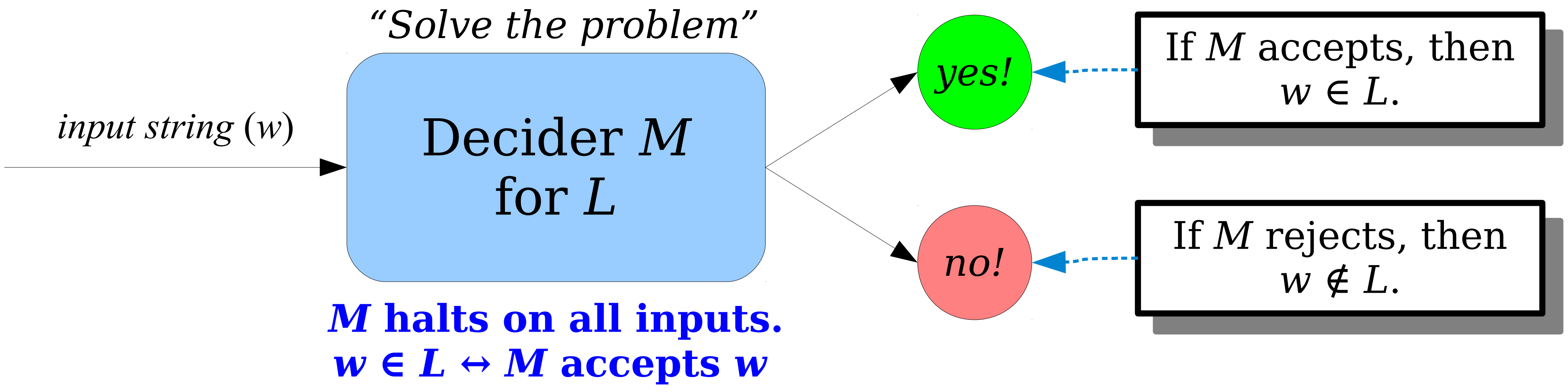
Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

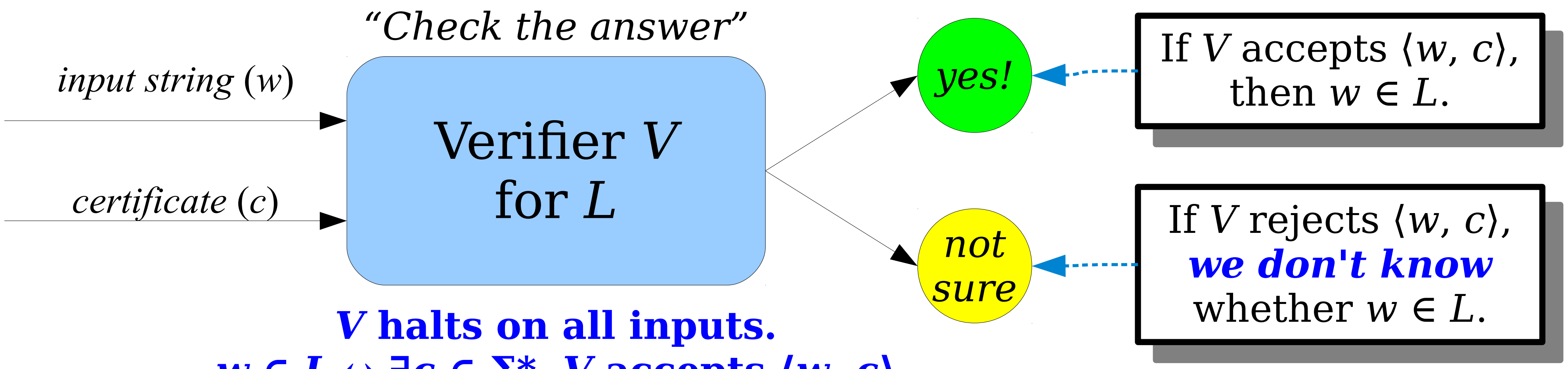
$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- A string c where V accepts $\langle w, c \rangle$ is called a ***certificate*** for w.
- Intuitively, what does this mean?

Deciders and Verifiers



M halts on all inputs.
$w \in L \leftrightarrow M$ accepts w



"Check the answer"
input string (w)
certificate (c)
Verifier V for L
yes!
not sure
If V accepts $\langle w, c\rangle$, then $w \in L$.
If V rejects $\langle w, c\rangle$, **we don't know** whether $w \in L$.

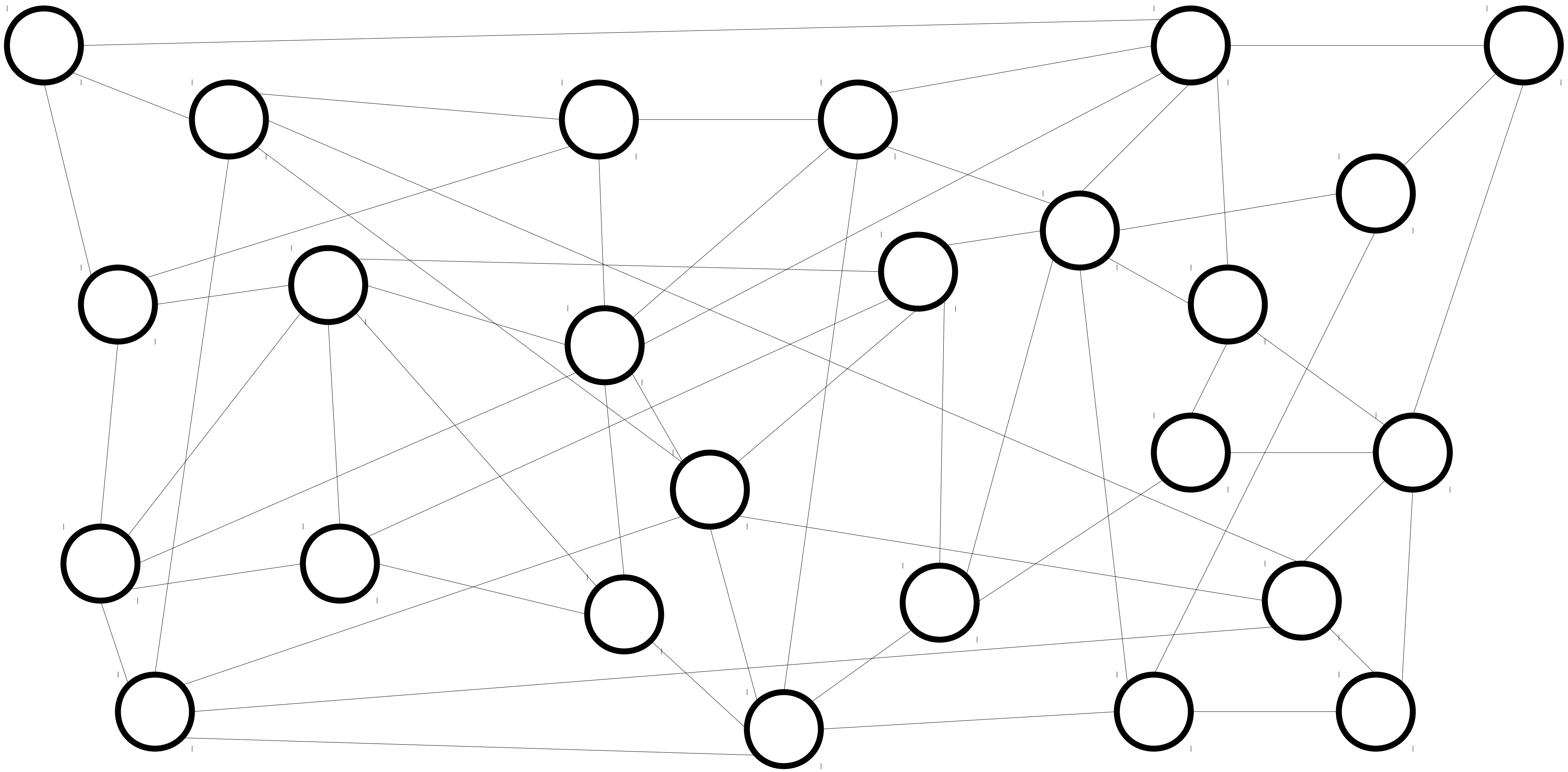
V halts on all inputs.
$w \in L \leftrightarrow \exists c \in \Sigma^*.\ V$ accepts $\langle w, c\rangle$

Verifiers

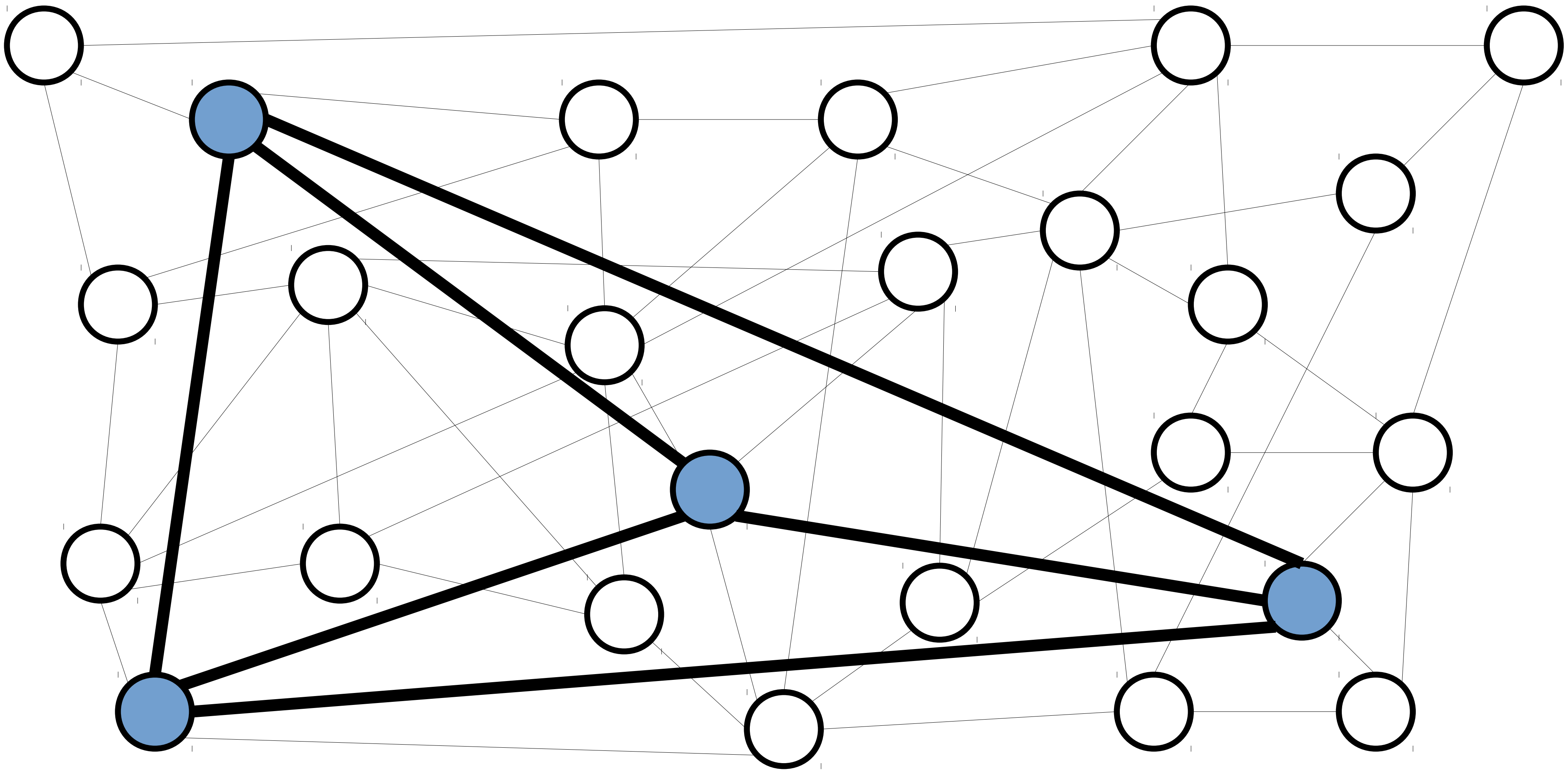
- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

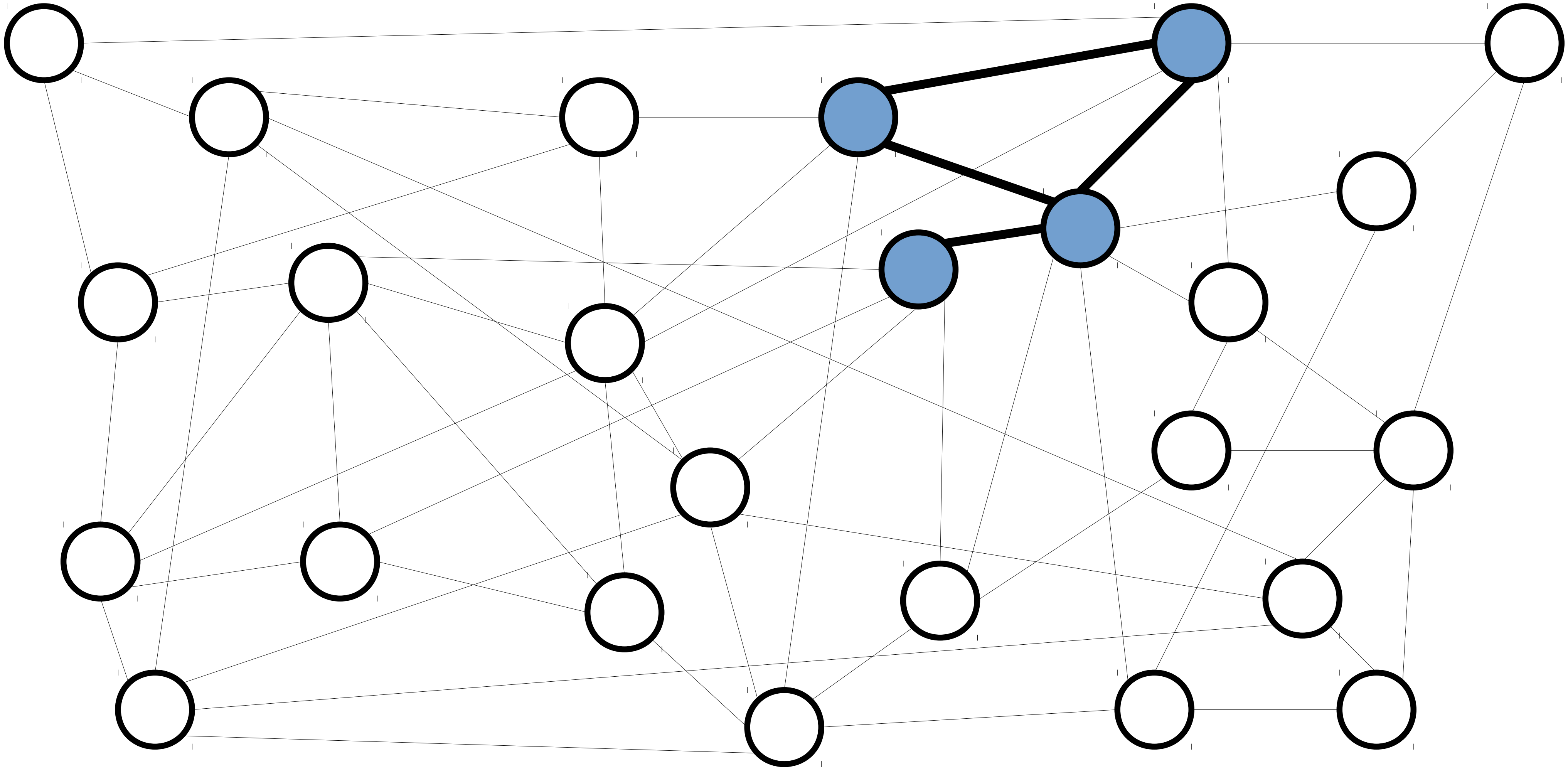
- Some notes about V:
  - If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
  - If V does not accept $\langle w, c \rangle$, then either
    - $w \in L$, but you gave the wrong c, or
    - $w \notin L$, so no possible c will work.



Decider for L = $\{\langle G\rangle \mid G$ is a graph with a 4-clique$\}$ would look for a 4-clique and accept/reject this graph.



- If a **verifier** V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.



- If a **verifier** V does not accept $\langle w, c\rangle$, then either
  - $w \in L$, but you gave the wrong c, or
  - $w \notin L$, so no possible c will work.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - Notice that c is existentially quantified. Any string $w \in L$ must have at least one c that causes V to accept, and possibly more.
  - V is required to halt, so given any potential certificate c for w, you can check whether the certificate is correct.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - Notice that $\mathscr{L}(V) \neq L$. *(Good question: what <u>is</u> $\mathscr{L}(V)$?)*
  - The job of V is just to check certificates, not to decide membership in L.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c\rangle}$$

- Some notes about V:
  - Although this formal definition works with a string c, remember that c can be an encoding of some other object.
  - In practice, c will likely just be "some other auxiliary data that helps you out."

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

- Let's see how to build a verifier for L.

Verification

11

Does the hailstone sequence
terminate for this number?

Verification

11

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

34

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

17

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

52

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

26

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

13

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

40

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

20

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

10

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

5

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

16

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

8

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

4

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Verification

2

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

Verification

1

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

```
bool checkHailstone(int n, int c) {
    for (int i = 0; i < c; i++) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    return n == 1;
}
```

- Do you see why $\langle n \rangle \in L$ iff there is some c such that `checkHailstone(n, c)` returns true?
- Do you see why `checkHailstone` always halts?

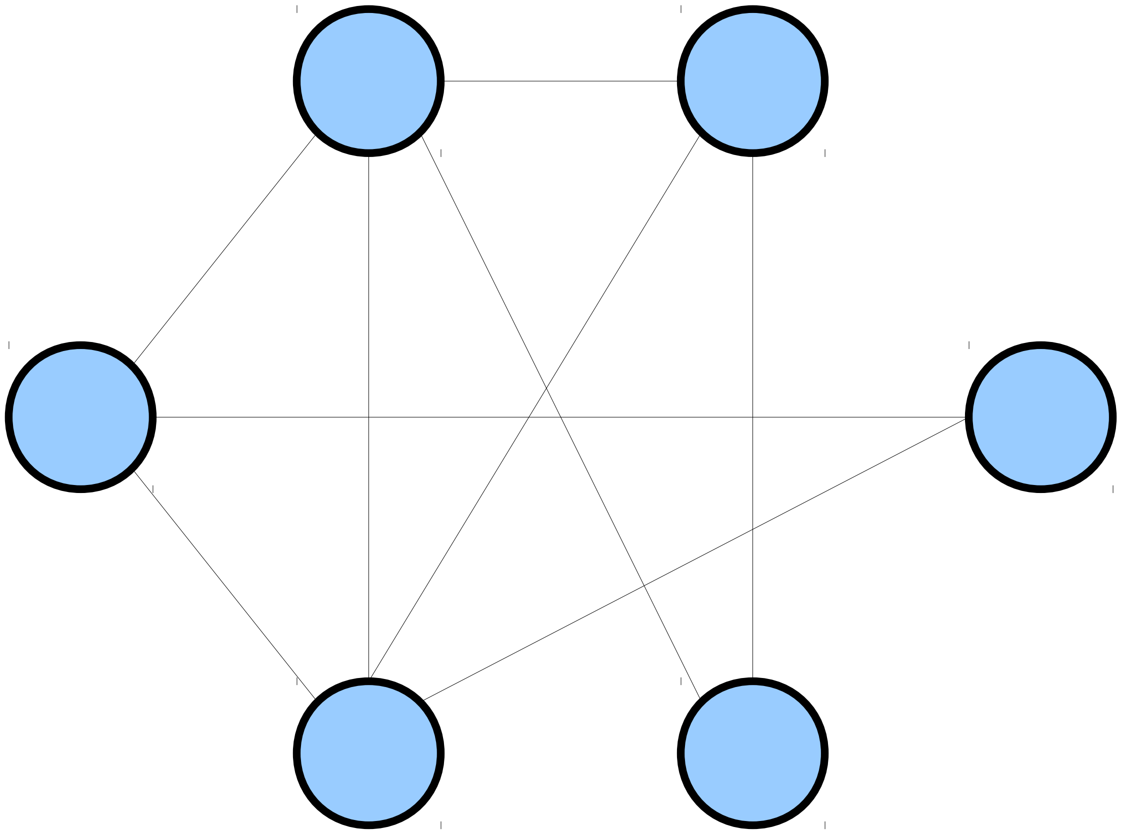
Some Verifiers

- Let L be the following language:

  $L = \{ \langle G \rangle \mid G$ is a graph and G has a Hamiltonian path $\}$

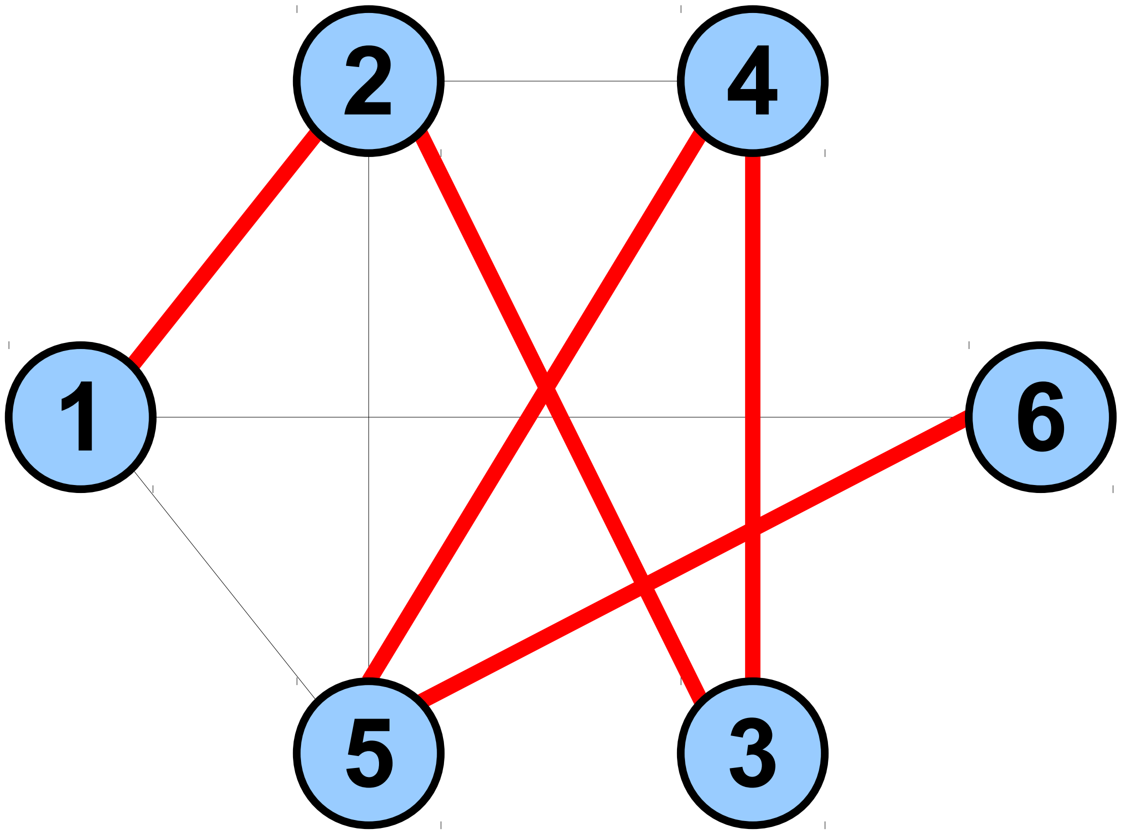
- (A Hamiltonian path is a simple path that visits every node in the graph.)
- Let's see how to build a verifier for L.

Verification



Is there a simple path that goes through every node exactly once?

Verification



Is there a simple path that goes through every node exactly once?

Some Verifiers

- Let L be the following language:

$$L = \{ \langle G \rangle \mid G \text{ is a graph with a Hamiltonian path } \}$$

```
bool checkHamiltonian(Graph G, vector<Node> c) {
   if (c.size() != G.numNodes()) return false;
   if (containsDuplicate(c)) return false;

   for (size_t i = 0; i < c.size() - 1; i++) {
       if (!G.hasEdge(c[i], c[i+1])) return false;
   }
   return true;
}
```

- Do you see why $\langle G \rangle \in L$ iff there is a c where `checkHamiltonian(G, c)` returns true?
- Do you see why `checkHamiltonian` always halts?

Some Verifiers

- Consider A_{TM}:

  $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$

- This is a ***canonical*** example of an undecidable language. There's no way, in general, to tell whether a TM M will accept a string w.
- Although this language is undecidable, it's an **RE** language, and it's possible to build a verifier for it!