The Complexity of Verifying Boolean Programs as Differentially Private

Ludmila Glinskih

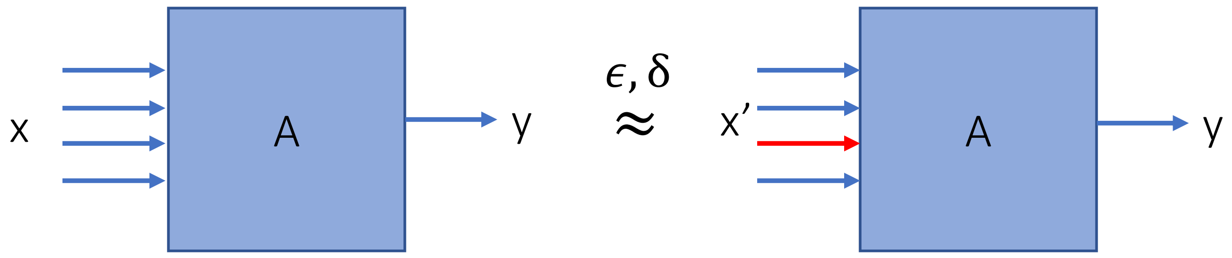
Joint work with Mark Bun and Marco Gaboardi
Boston University

CSF 2022
Haifa, Israel
August 10, 2022

Plan of the talk

1. Prior work
    - How hard is it to verify whether a program is DP for
        - Turing-complete languages
        - Boolean languages with bounded memory without loops
2. Our results and proof ideas
    - BPWhile: Boolean language with loops and finite memory
    - PSPACE-completeness of verifying DP for BPWhile
    - PSPACE-hardness: reduction from TQBF
    - PSPACE algorithm based on computing hitting probabilities in a Markov chain

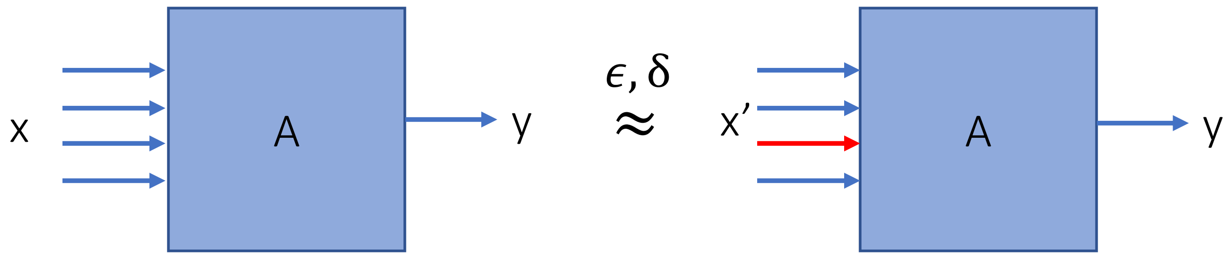
Differential Privacy



C is (ϵ, δ)-differentially private if for every set of possible outputs O, and for every neighboring x, x':

$$\mathrm{P}[C(x) \in O] \leq e^{\epsilon} \cdot \mathrm{P}[C(x') \in O] + \delta$$

Differential Privacy

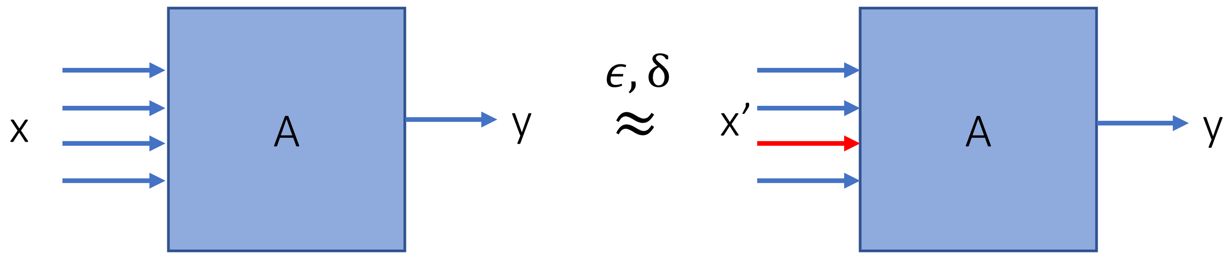


Non-termination is one of the possible outputs

C is (ϵ, δ)-differentially private if for every set of possible outputs *O*, and for every neighboring *x, x'*:

$$\mathrm{P}[C(x) \in O] \leq e^{\epsilon} \cdot \mathrm{P}[C(x') \in O] + \delta$$

Differential Privacy



Non-termination is one of the possible outputs

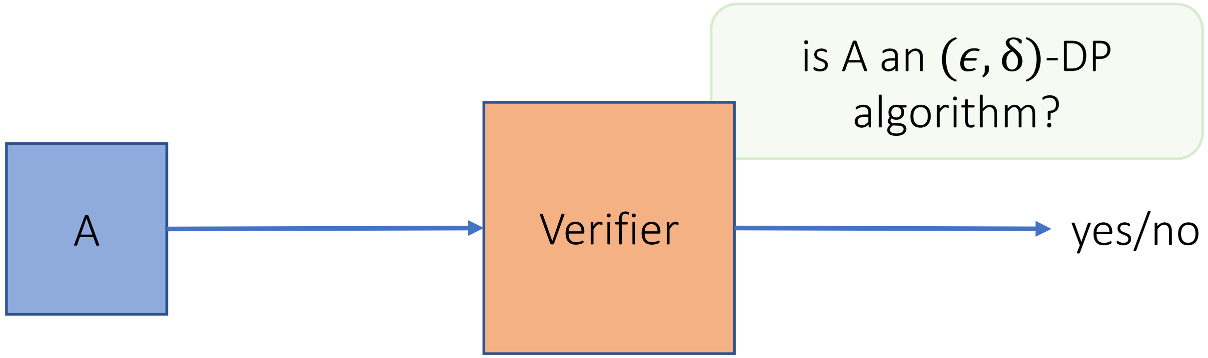
C is (ϵ, δ)-differentially private if for every set of possible outputs *O*, and for every neighboring *x, x'*:

$$\mathrm{P}[C(x) \in O] \leq e^{\epsilon} \cdot \mathrm{P}[C(x') \in O] + \delta$$

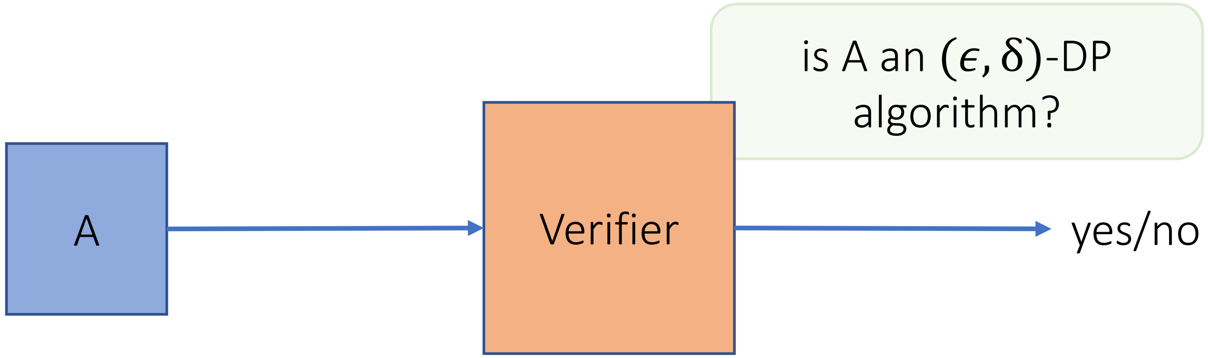
Neighboring relation we consider:

- Inputs are bit strings which differ in one bit
- Can be extended to any bounded polyspace computable relation

Verification of Differential Privacy



Verification of Differential Privacy



Complexity depends on the expressivity of the language and the type of access to the code:

- **Black box:** no information about the algorithm, query access
  - Impossible to verify $(\epsilon, 0)$-DP

[Gilbert, McMillan'19]

Verification of DP: white box

- Linear-time algorithm in the size of automaton
  - For pure-DP

[Chadha, Sistla, Viswanathan'21]

Verification of DP: white box

- Linear-time algorithm in the size of automaton
  - For pure-DP
- #P-hardness for approximating parameters in labelled Markov chains
  - For approximate-DP
  - Undecidable to compute exactly

[Chadha, Sistla, Viswanathan’21]

[Chistikov, Murawski, Purser’19]

Verification of DP: white box

- Linear-time algorithm in the size of automaton
  - For pure-DP

[Chadha, Sistla, Viswanathan’21]

- #P-hardness for approximating parameters in labelled Markov chains
  - For approximate-DP
  - Undecidable to compute exactly

[Chistikov, Murawski, Purser’19]

- Undecidable for languages working with infinite data
  - Undecidable for Turing-complete languages

[Barthe, Chadha, Jagannath, Sistla, Viswanathan’20]

Verification of DP: white box

- Linear-time algorithm in the size of automaton
  - For pure-DP

[Chadha, Sistla, Viswanathan'21]

- #P-hardness for approximating parameters in labelled Markov chains
  - For approximate-DP
  - Undecidable to compute exactly

[Chistikov, Murawski, Purser'19]

- Undecidable for languages working with infinite data
  - Undecidable for Turing-complete languages

[Barthe, Chadha, Jagannath, Sistla, Viswanathan'20]

- For a simple Boolean language with **bounded memory,** if statements and **random assignments**, but **without loops**
  - $coNP^{\#P}$-completeness for $(\epsilon, 0)$-DP
    - Reduction from All-Min-SAT
  - $coNP^{\#P}$-hard and in $coNP^{\#P^{\#P}}$ for (ϵ, δ)-DP

[Gaboardi, Nissim, Purser'20]

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

x ::= $[a - z]^+$ Variable identifiers

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

x ::= $[a - z]^{+}$ Variable identifiers

b ::= *true* | *false* | random | x | b ∧ b | b ∨ b | !b

Boolean expressions

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

x ::= $[a - z]^+$ Variable identifiers

b ::= *true* | *false* | random | x | b ∧ b | b ∨ b | !b Boolean expressions

c ::= skip | x := b | c; c | if b then c else c | **while b then c** Commands

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

x ::= $[a - z]^+$ Variable identifiers

b ::= *true* | *false* | random | x | b ∧ b | b ∨ b | !b Boolean expressions

c ::= skip | x := b | c; c | if b then c else c | **while b then c** Commands

t ::= x | t, x List of Boolean variables

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

x ::= $[a - z]^+$ Variable identifiers

b ::= *true* | *false* | random | x | b ∧ b | b ∨ b | !b Boolean expressions

c ::= skip | x := b | c; c | if b then c else c | **while b then c** Commands

t ::= x | t, x List of Boolean variables

p::= input(t); c; return(t) Programs

BPWhile: Boolean language with While loops

We design a language to meet the following goals:

- Captures classical computations on real computers
- Simple to analyze

x ::= $[a - z]^+$ Variable identifiers

b ::= *true* | *false* | random | x | b ∧ b | b ∨ b | !b Boolean expressions

c ::= skip | x := b | c; c | if b then c else c | **while b then c** Commands

t ::= x | t, x List of Boolean variables

p::= input(t); c; return(t) Programs

Numbers of variables and input bits are fixed in the definition of the program

⇒ Length of the program is an upper bound on the size of the memory that the program uses

Example of a BPWhile program:

$$
\begin{array}{l}
\texttt{input}(\vec{c}, \epsilon); \\
\vec{k} := \lceil \log(2/\epsilon) \rceil; \\
\vec{d} := (2^{\vec{k}+1} + 1)(2^{\vec{k}} + 1)^{n-1}; \\
\vec{u} := \texttt{uniform}(0, \vec{d}]; \\
\vec{z} := 0; \\
\vec{r} := n; \\
\texttt{while}\ \vec{z} < \vec{n} \wedge \vec{r} = n\ \texttt{then} \\
\quad \texttt{if}\ \vec{z} < \vec{c}\ \texttt{then} \\
\quad\quad \texttt{if}\ \vec{u} \leq 2^{\vec{k}(\vec{c}-\vec{z})}(2^{\vec{k}} + 1)^{n-(\vec{c}-\vec{z})} \\
\quad\quad \texttt{then}\ \vec{r} := \vec{z} \\
\quad\quad \texttt{else skip} \\
\quad \texttt{else} \\
\quad\quad \texttt{if}\ \vec{u} \leq d - 2^{\vec{k}(\vec{z}-\vec{c}+1)}(2^{\vec{k}} + 1)^{n-1-(\vec{z}-\vec{c})} \\
\quad\quad \texttt{then}\ \vec{r} := \vec{z} \\
\quad\quad \texttt{else skip} \\
\quad \vec{z} = \vec{z} + 1; \\
\texttt{return}(\vec{z});
\end{array}
$$

Implementation of the Bounded Geometric Mechanism in finite precision arithmetic

[Ghosh, Roughgarden, Sundararajan'09]

[Balcer, Vadhan'17]

Our results

Main result: if A is a BPWhile program, then the problem of verifying whether A is differentially private is PSPACE-complete.

Our results

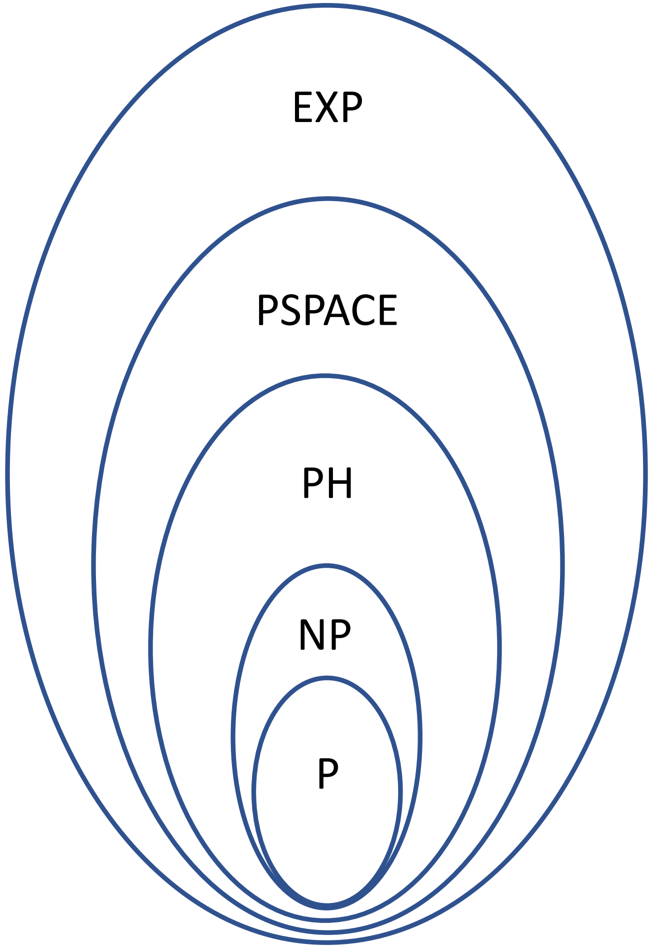
Main result: if A is a BPWhile program, then the problem of verifying whether A is differentially private is PSPACE-complete.

It holds for the following notions of differential privacy:

- $(\epsilon, 0)$-DP
- (ϵ, δ)-DP
- (ϵ, δ)-DP parameters approximation
- Renyi-DP
- Zero-Concentrated-DP
- Truncated Concentrated-DP

PSPACE-completeness

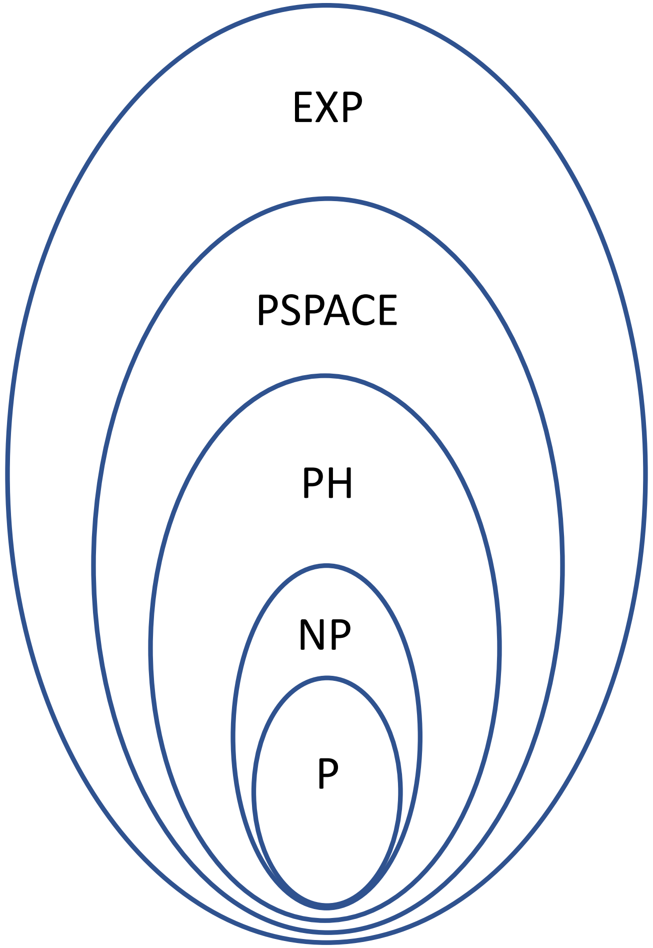
PSPACE-completeness of a problem A implies:



PSPACE-completeness

PSPACE-completeness of a problem A implies:

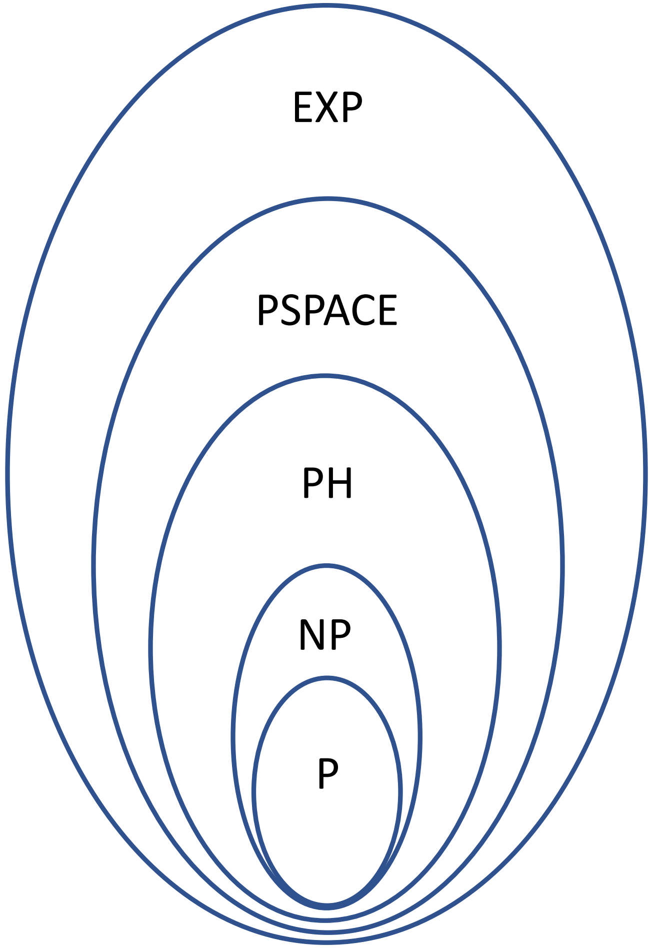
- A is solvable by a TM that uses polynomial space



PSPACE-completeness

PSPACE-completeness of a problem A implies:

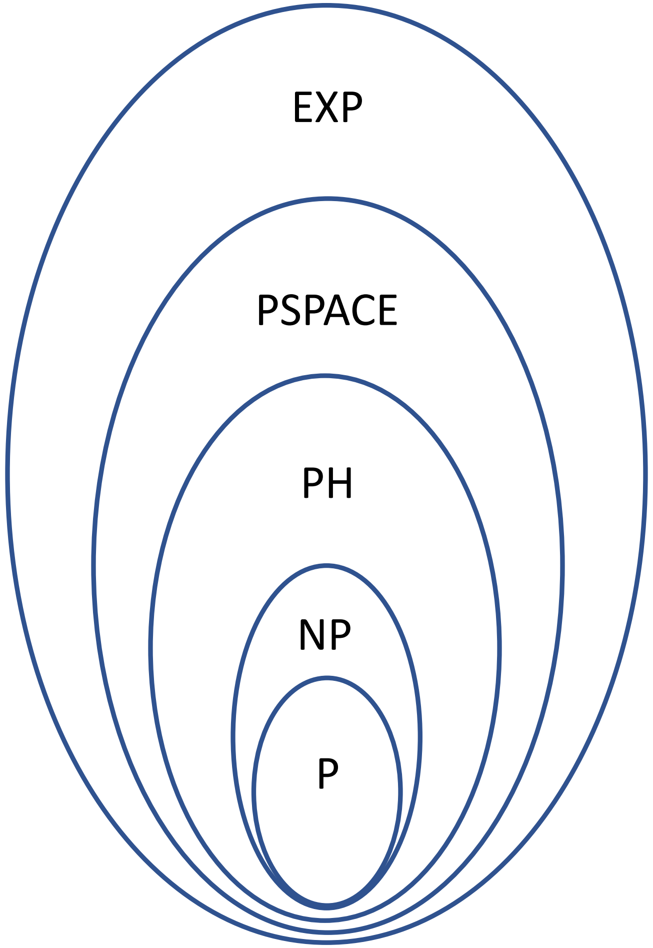
- A is solvable by a TM that uses polynomial space
- A is solvable in exponential time



PSPACE-completeness

PSPACE-completeness of a problem A implies:

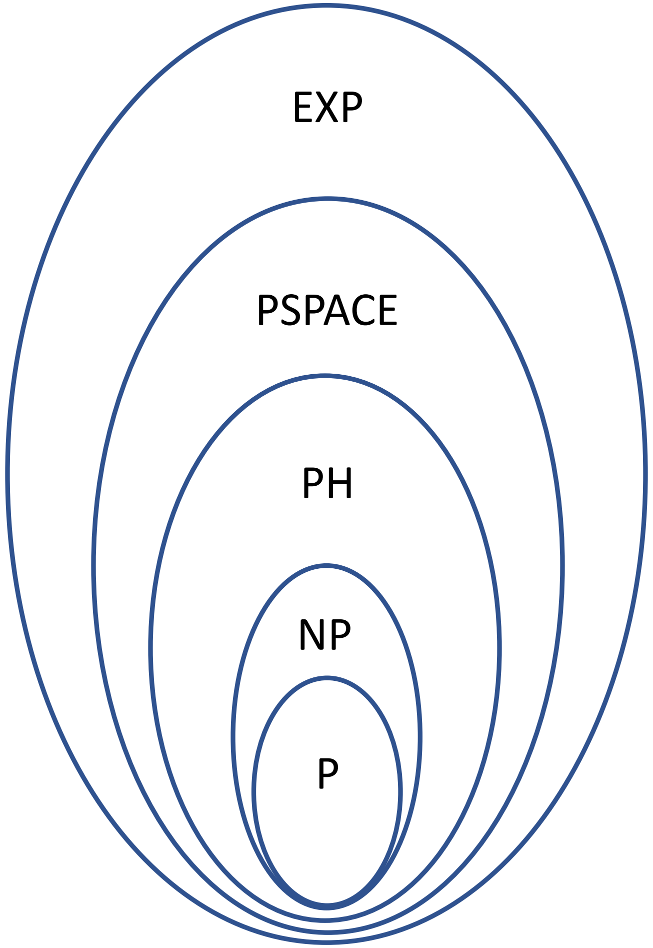
- A is solvable by a TM that uses polynomial space
- A is solvable in exponential time
- A is at least as hard as any problem solvable in polyspace



PSPACE-completeness

PSPACE-completeness of a problem A implies:

- A is solvable by a TM that uses polynomial space
- A is solvable in exponential time
- A is at least as hard as any problem solvable in polyspace
- No polytime algorithm for A, unless P = PSPACE
  - That is widely believed not to be true

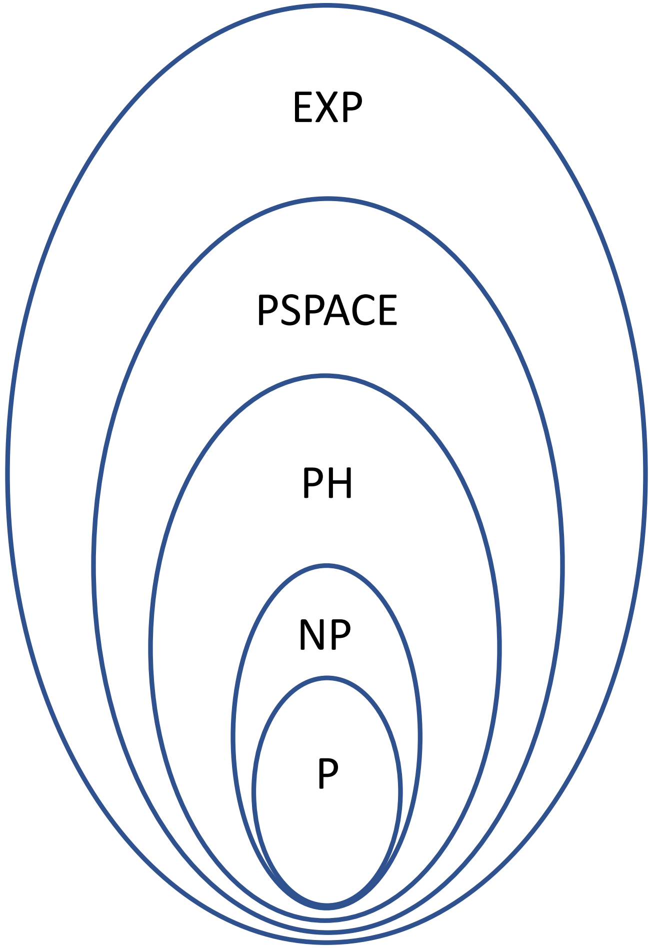


PSPACE-completeness

PSPACE-completeness of a problem A implies:

- A is solvable by a TM that uses polynomial space
- A is solvable in exponential time
- A is at least as hard as any problem solvable in polyspace
- No polytime algorithm for A, unless P = PSPACE

To show PSPACE completeness we need:



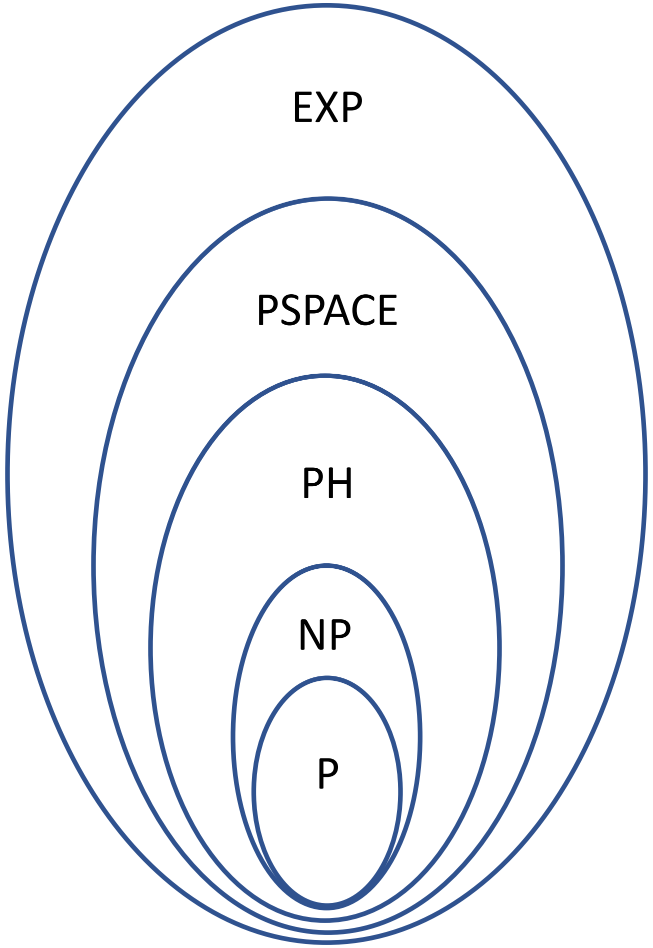
PSPACE-completeness

PSPACE-completeness of a problem A implies:

- A is solvable by a TM that uses polynomial space
- A is solvable in exponential time
- A is at least as hard as any problem solvable in polyspace
- No polytime algorithm for A, unless P = PSPACE

To show PSPACE completeness we need:

1. Show hardness: construct sequence of reductions from TQBF



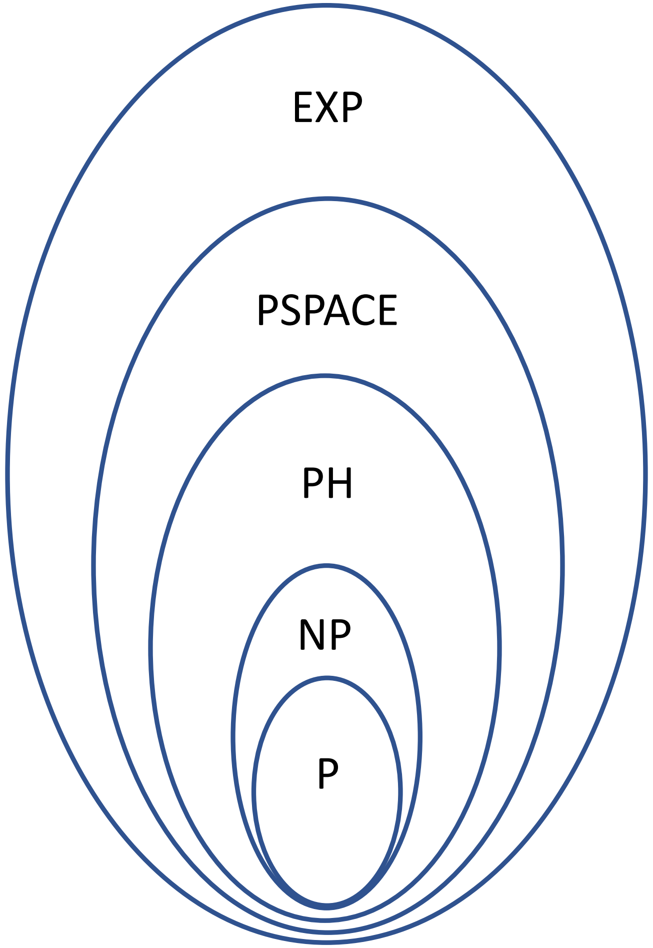
PSPACE-completeness

PSPACE-completeness of a problem A implies:

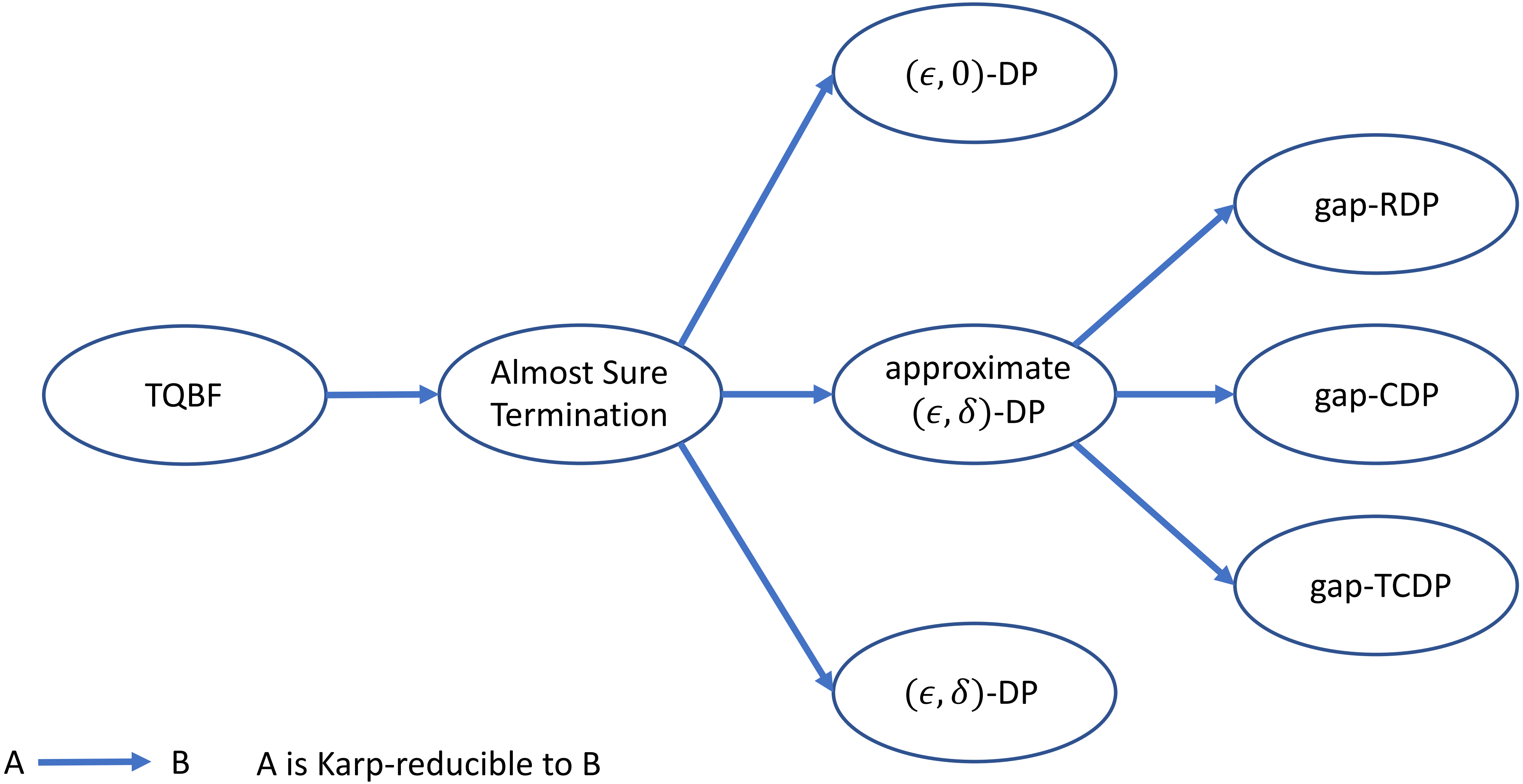
- A is solvable by a TM that uses polynomial space
- A is solvable in exponential time
- A is at least as hard as any problem solvable in polyspace
- No polytime algorithm for A, unless P = PSPACE

To show PSPACE completeness we need:

1. Show hardness: construct sequence of reductions from TQBF
2. Construct polynomial-space algorithm: analyze Markov chain based on the state graph of the program

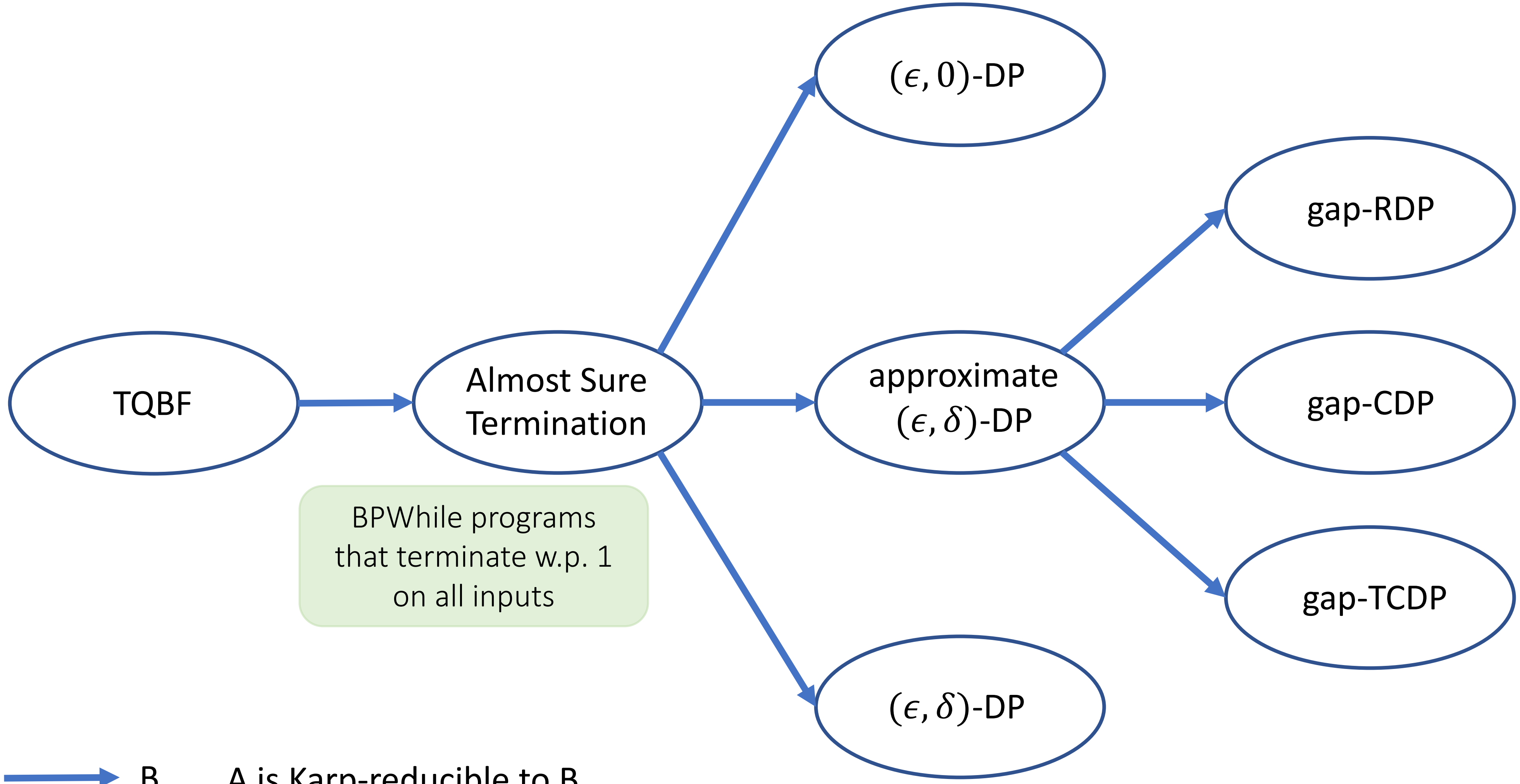


PSPACE-hardness



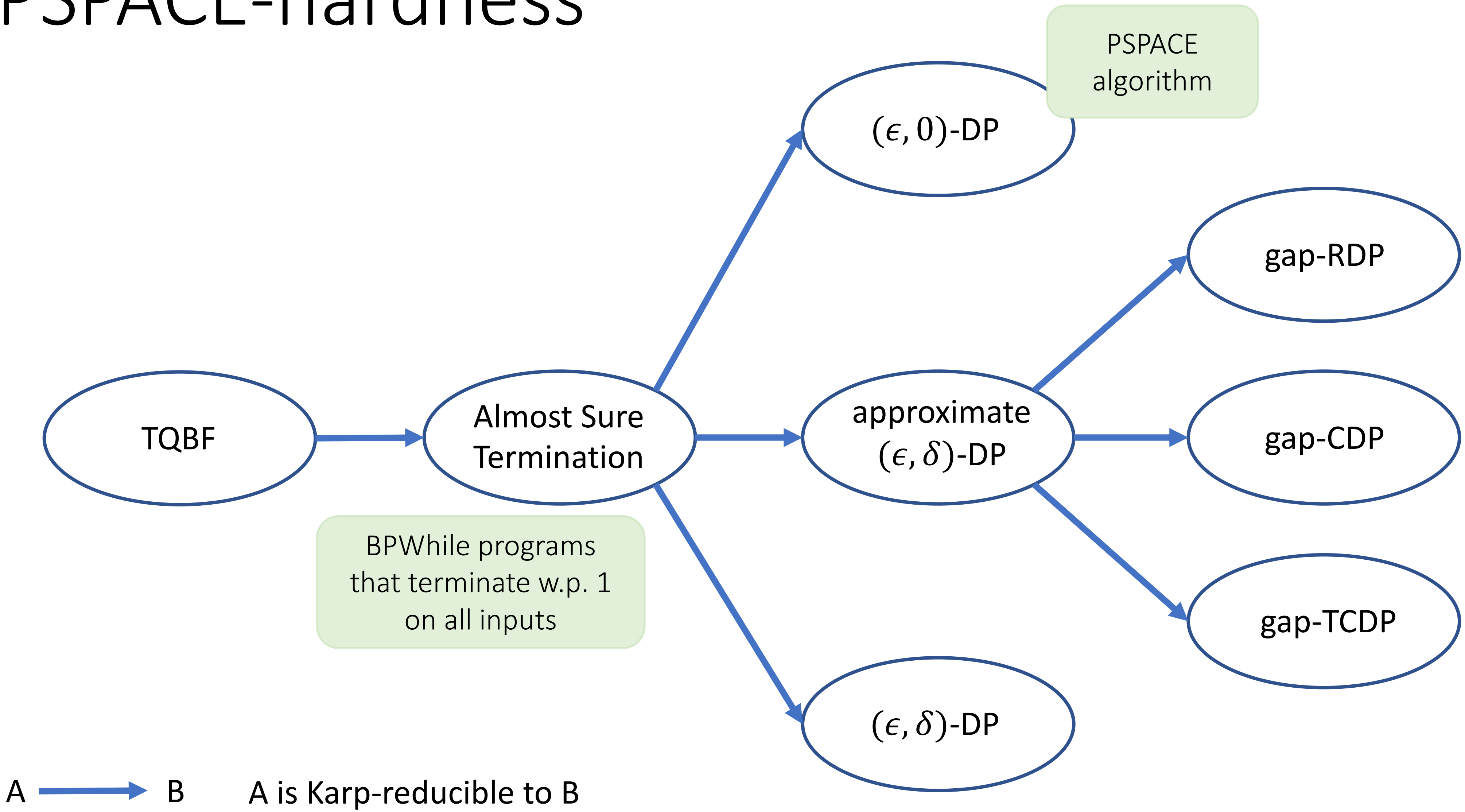
A is Karp-reducible to B

PSPACE-hardness



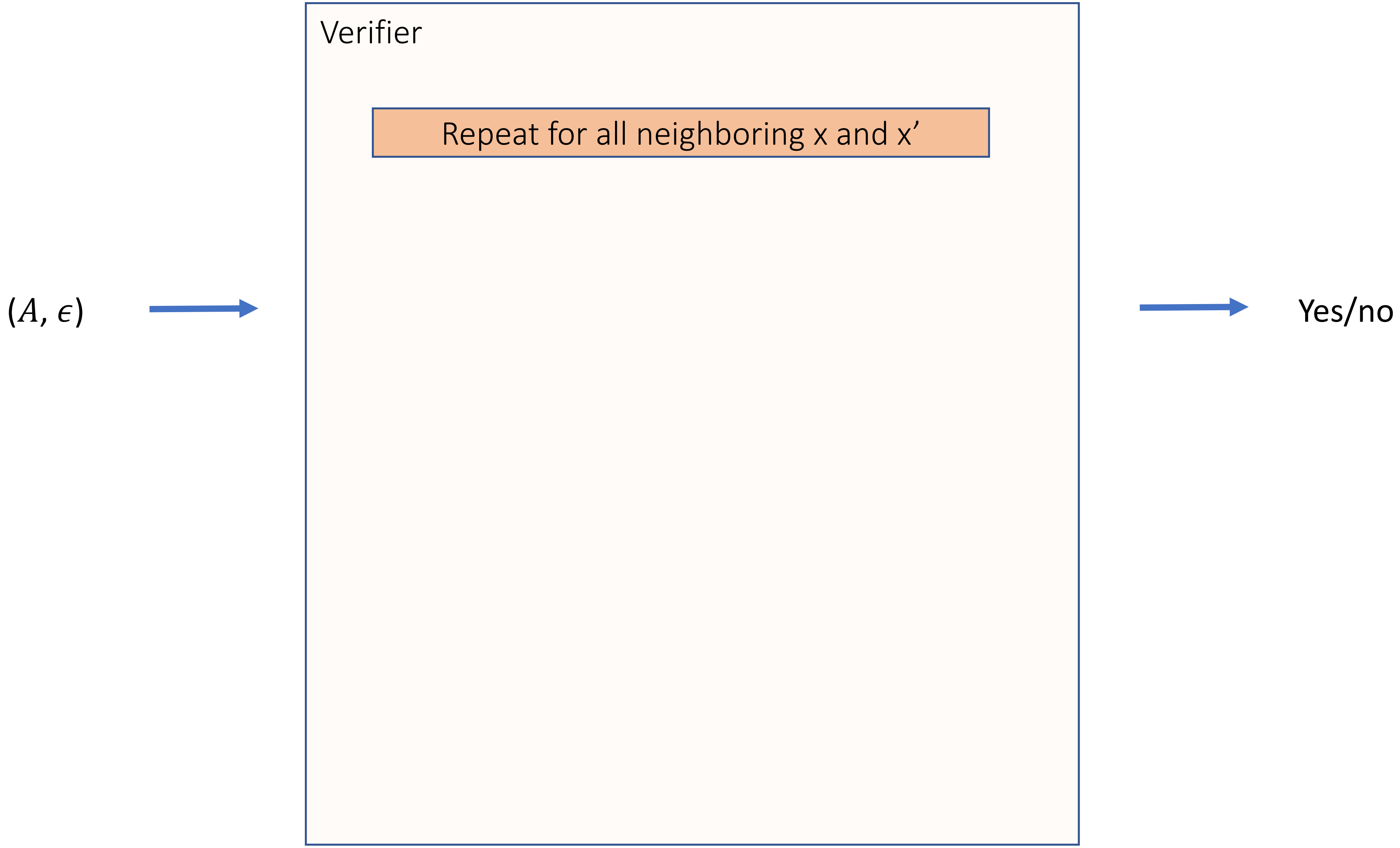
A → B A is Karp-reducible to B

PSPACE-hardness



A is Karp-reducible to B

Polyspace membership: algorithm for (ϵ,0)-DP



Polyspace membership: algorithm for (ϵ,0)-DP

(A, ϵ) →

Verifier

Repeat for all neighboring x and x’

Analyze how distributions on outputs of
A(x) and A(x’) differ
based on the code of A

→ Yes/no

PSPACE membership: state graph

State graph depends on the input values

D(b=1):

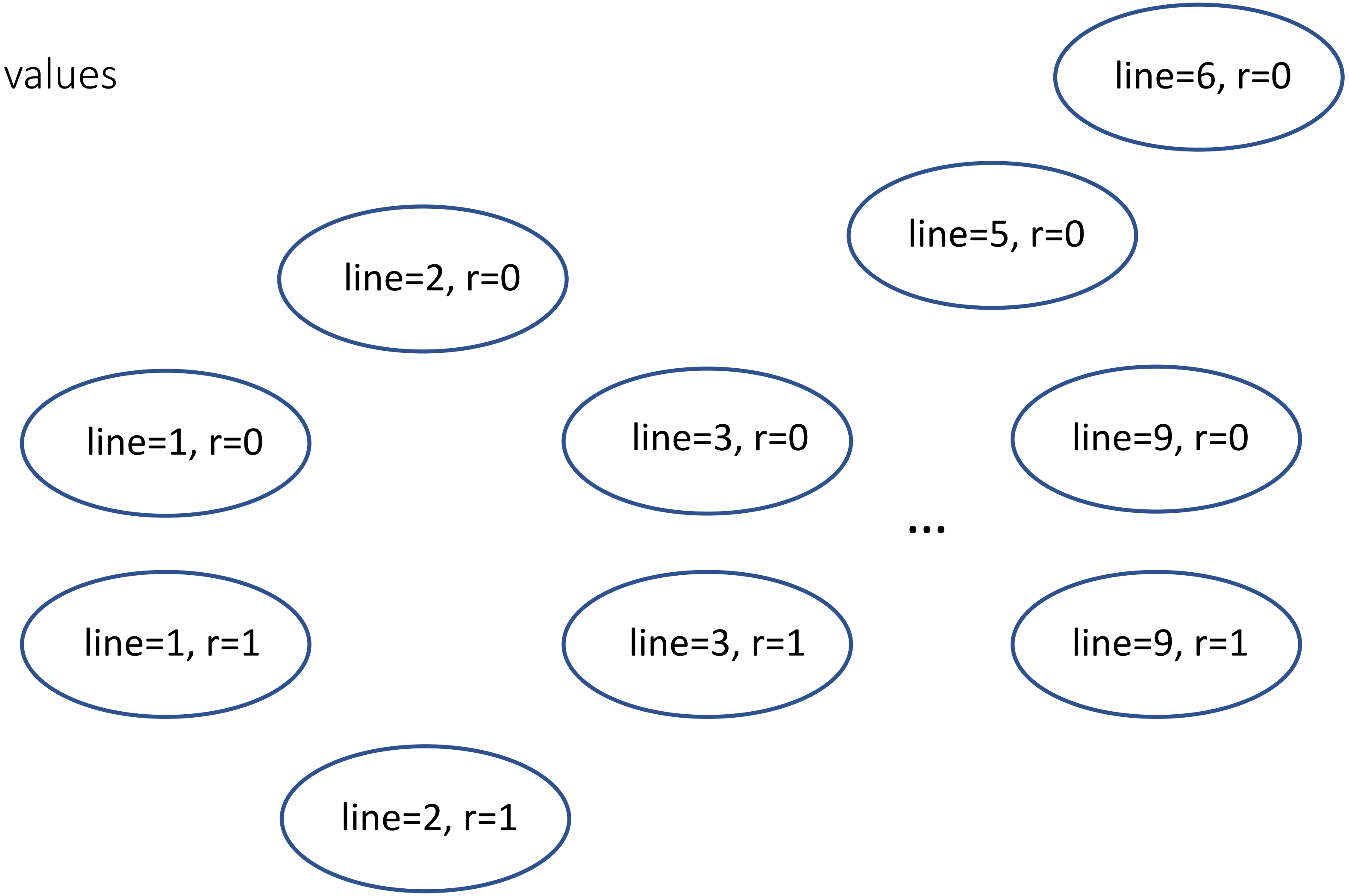
```
input(b);
if b == 1 then
    r = rand();
    if r == 0 then
        while true
            skip;
    else skip;
else skip;
return(1)
```

PSPACE membership: state graph

State graph depends on the input values

D(b=1):

```
1.  input(b);
2.  if b == 1 then
3.      r = rand();
4.      if r == 0 then
5.          while true
6.              skip;
7.      else skip;
8.  else skip;
9.  return(1)
```

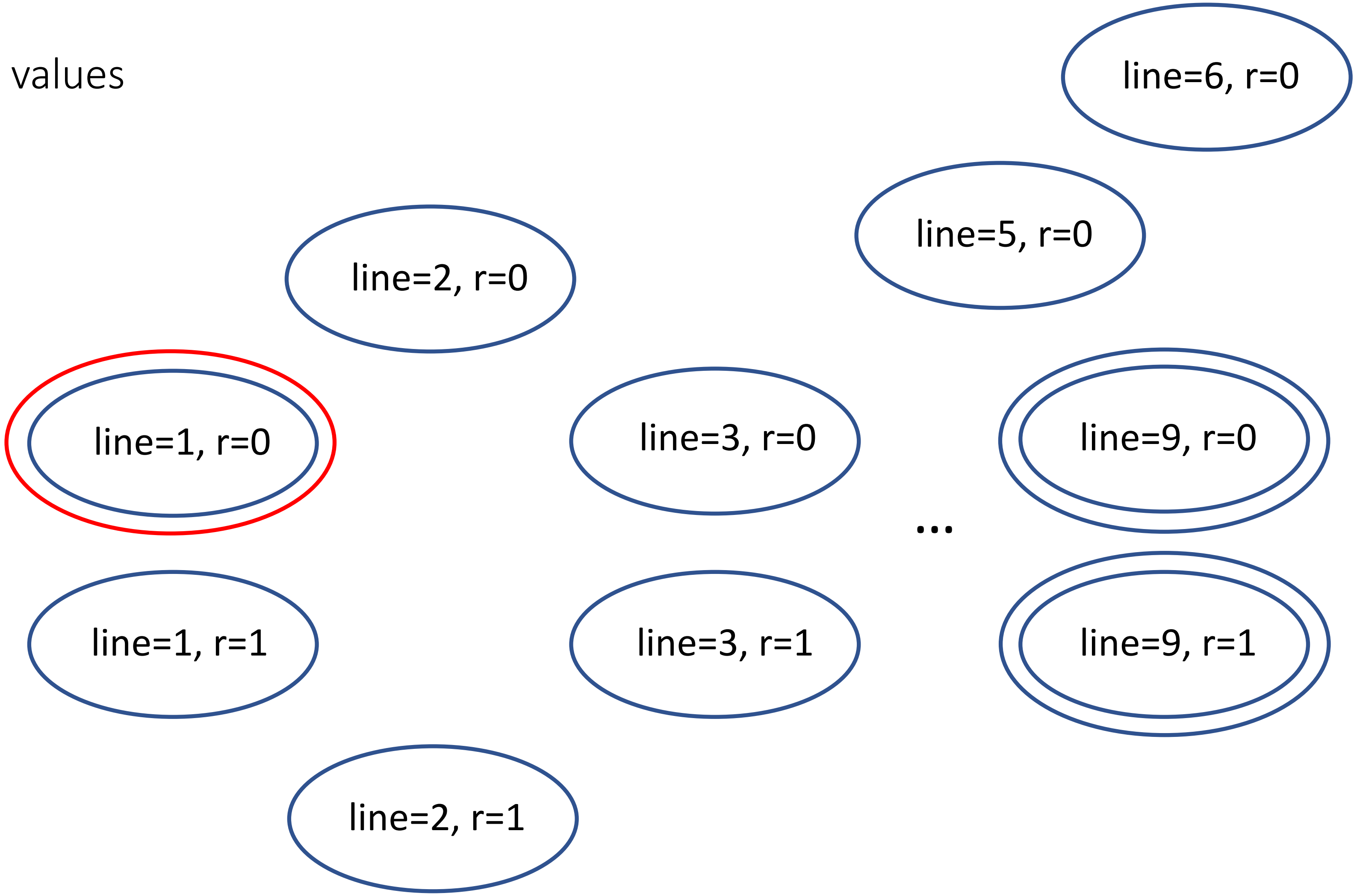


PSPACE membership: state graph

State graph depends on the input values

D(b=1):

```
1.  input(b);
2.  if b == 1 then
3.      r = rand();
4.      if r == 0 then
5.          while true
6.              skip;
7.      else skip;
8.  else skip;
9.  return(1)
```

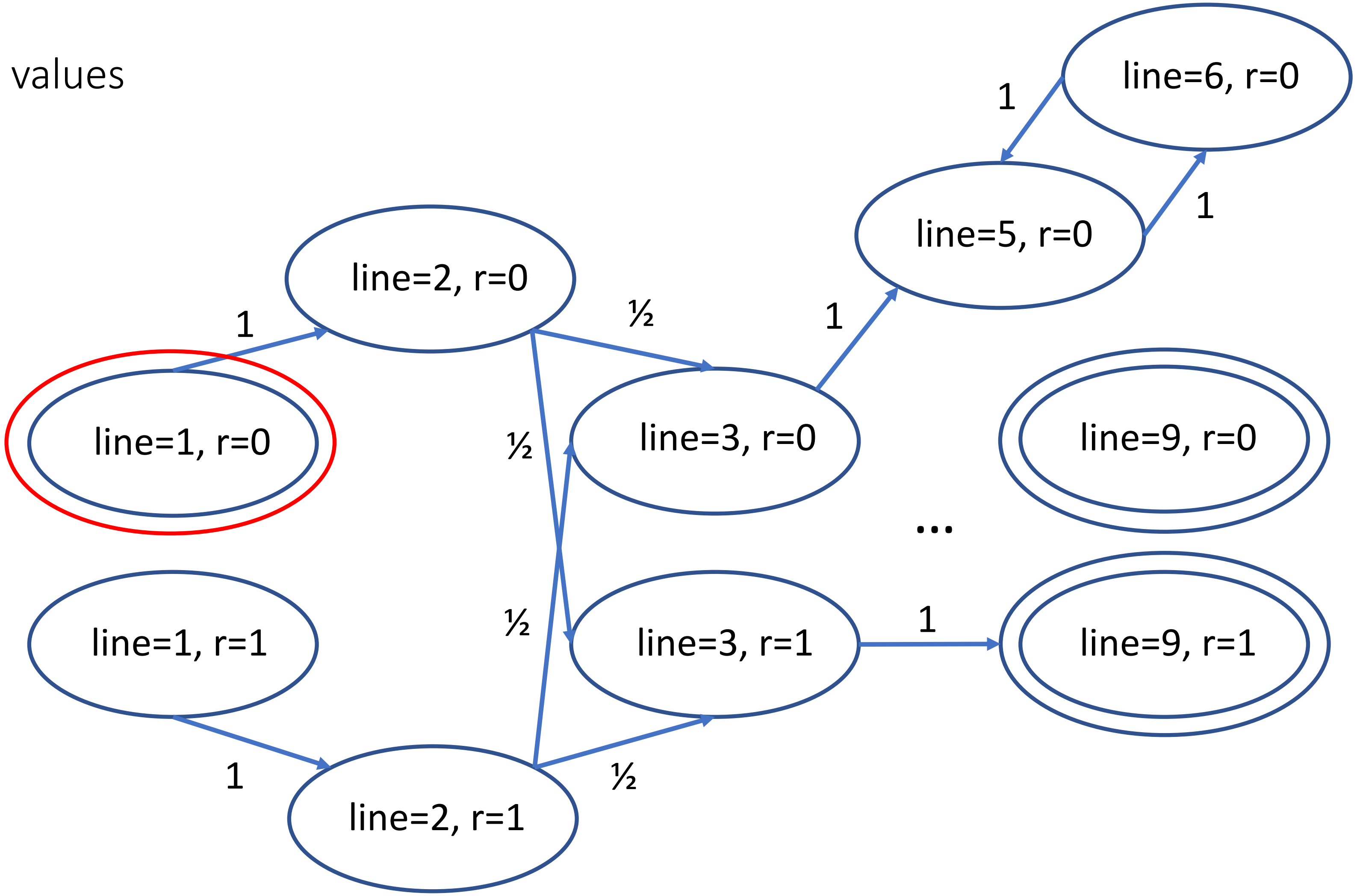


PSPACE membership: state graph

State graph depends on the input values

D(b=1):

```
1.  input(b);
2.  if b == 1 then
3.      r = rand();
4.      if r == 0 then
5.          while true
6.              skip;
7.      else skip;
8.  else skip;
9.  return(1)
```

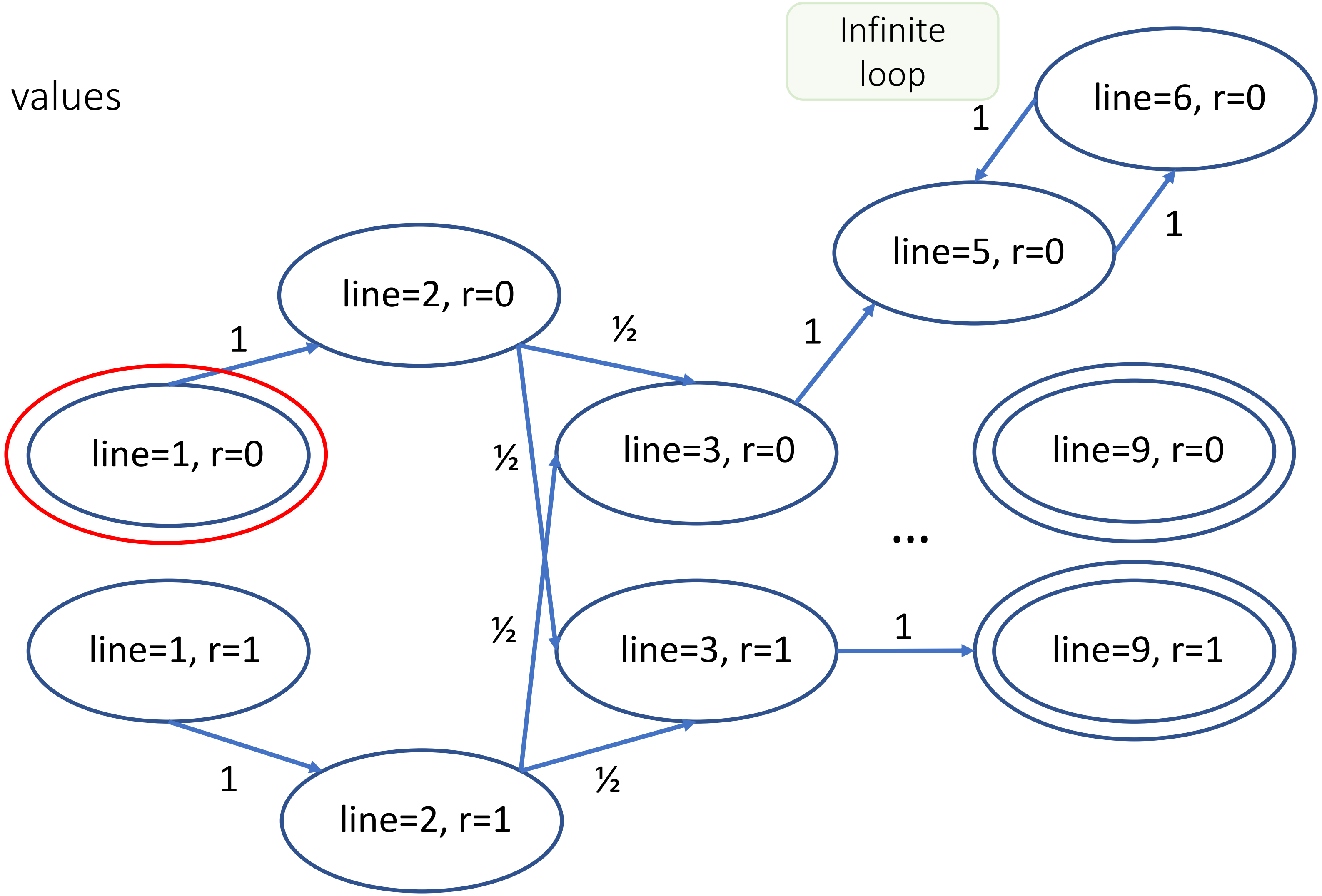


PSPACE membership: state graph

State graph depends on the input values

D(b=1):

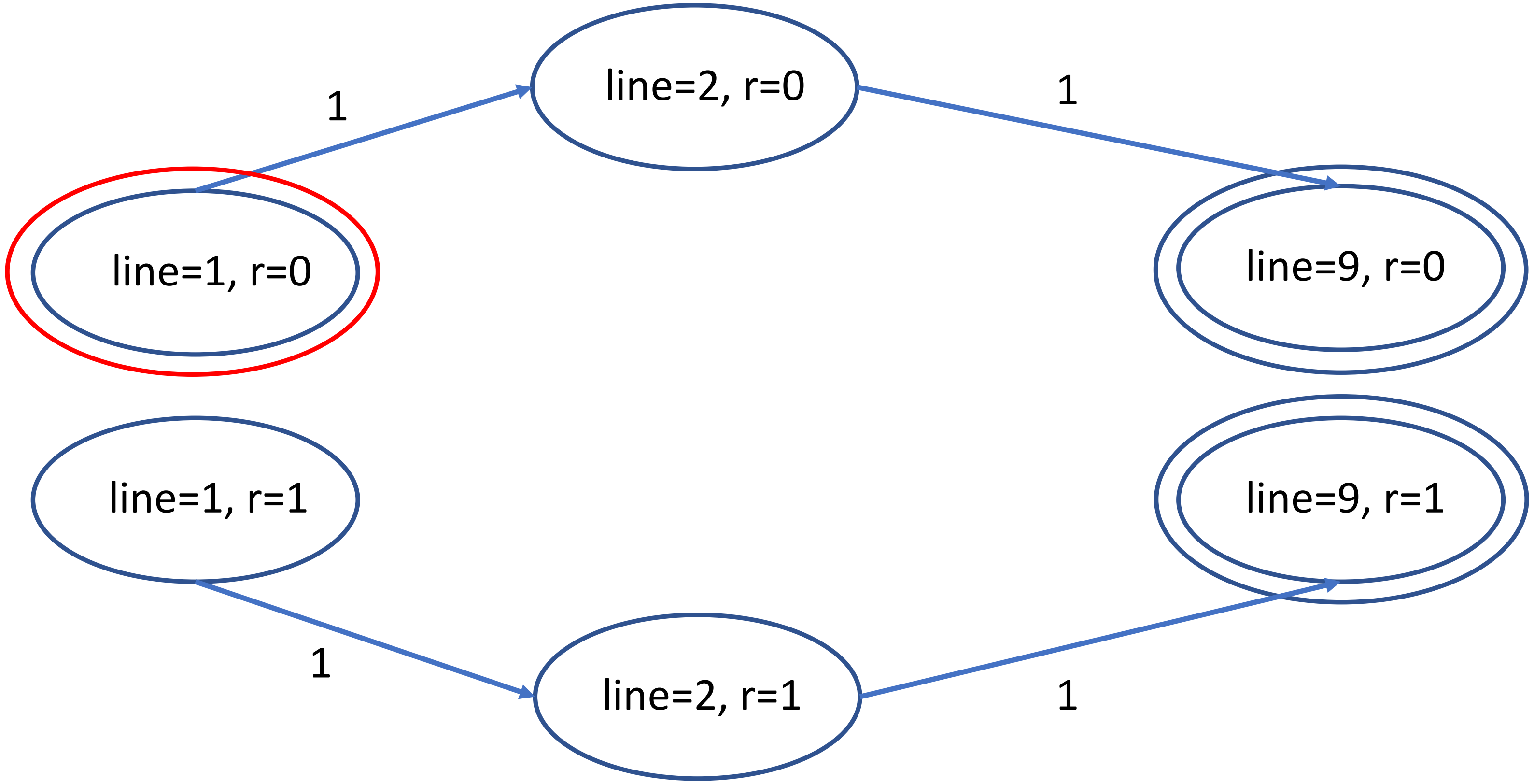
```
1. input(b);
2. if b == 1 then
3.     r = rand();
4.     if r == 0 then
5.         while true
6.             skip;
7.     else skip;
8. else skip;
9. return(1)
```



PSPACE membership: state graph

D(b=0):

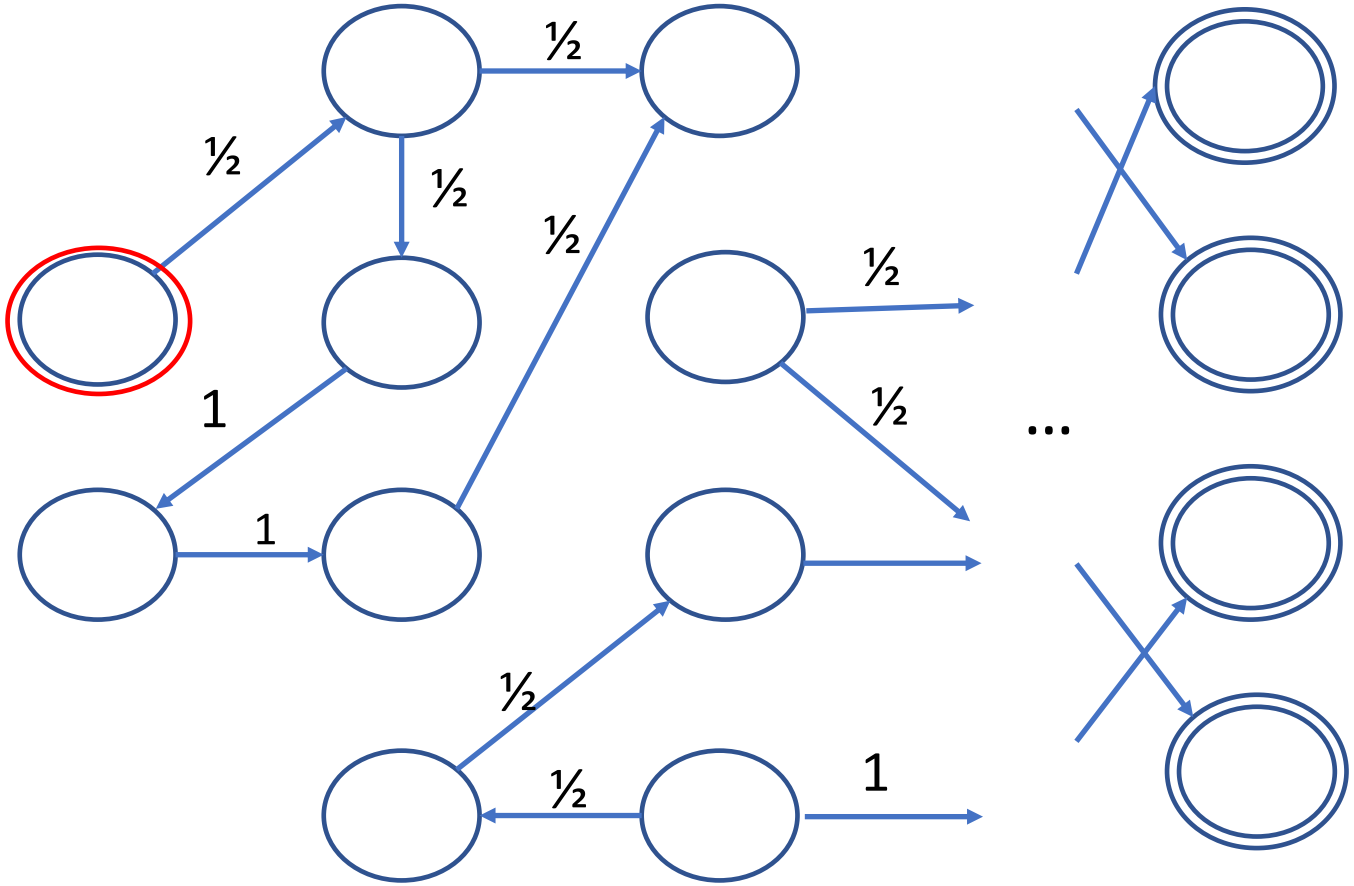
```
1.  input(b);
2.  if b == 1 then
3.      r = rand();
4.      if r == 0 then
5.          while true
6.              skip;
7.      else skip;
8.  else skip;
9.  return(1)
```



PSPACE membership: algorithm for (ϵ,0)-DP

For a program D and all neighboring inputs x, x':

- Construct the Markov chain for $D(x)$ and $D(x')$
- Compute and compare hitting probabilities

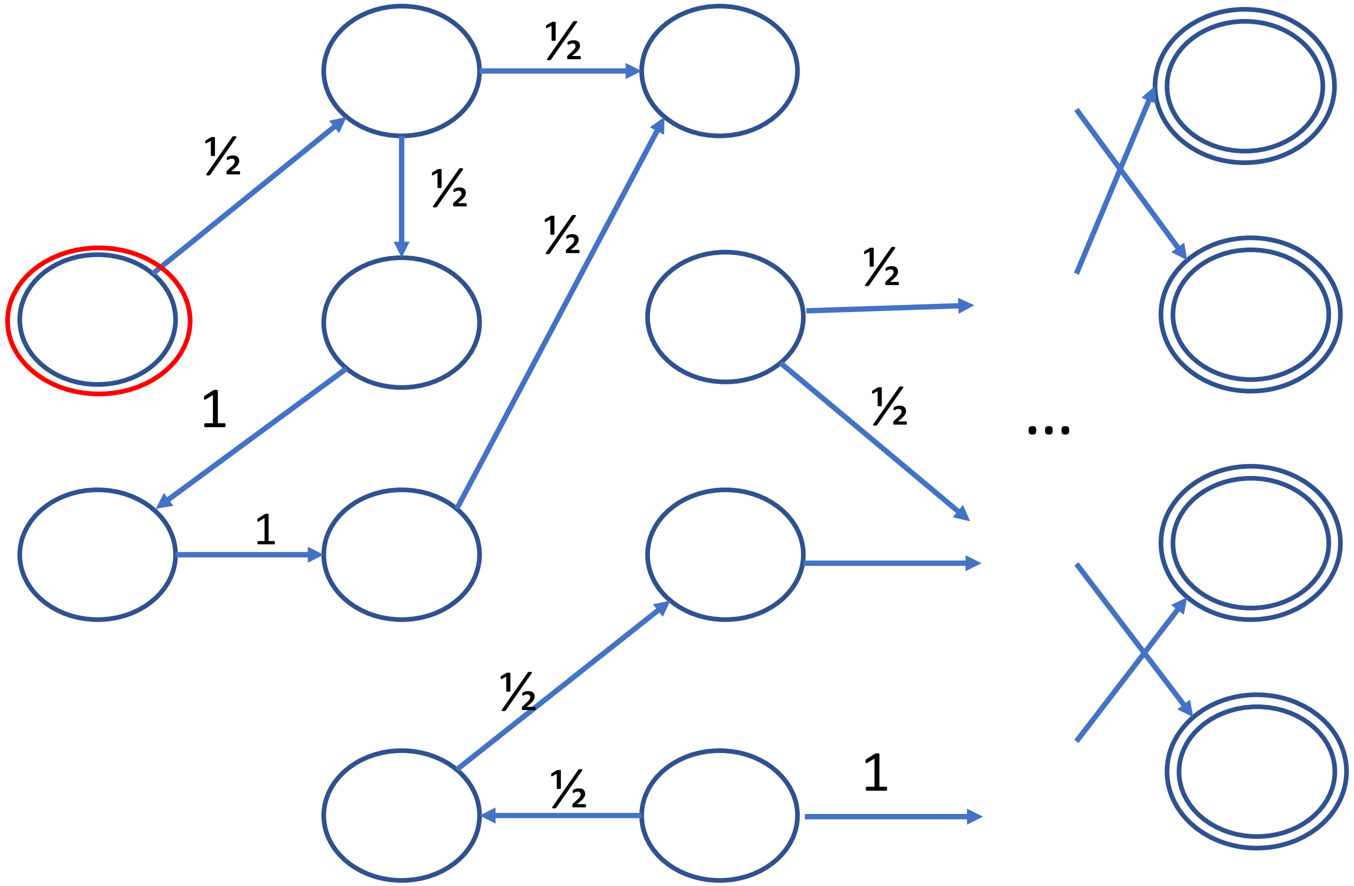


PSPACE membership: algorithm for (ϵ,0)-DP

For a program D and all neighboring inputs x, x':

- Construct the Markov chain for $D(x)$ and $D(x')$
- Compute and compare hitting probabilities

Problem: Markov chain has exp-many states ⇒ cannot store it explicitly



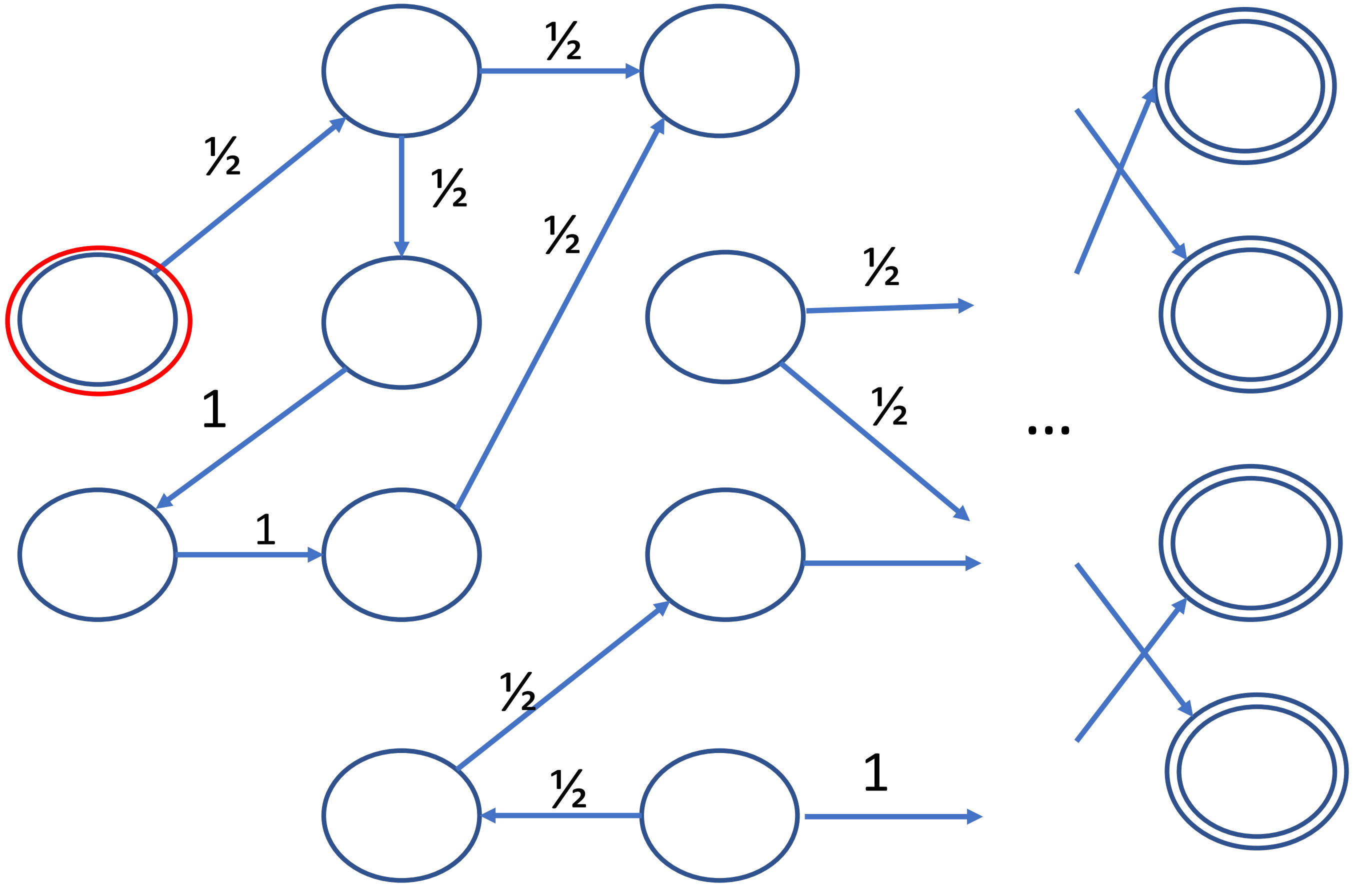
PSPACE membership: algorithm for (ϵ,0)-DP

For a program D and all neighboring inputs x, x':

- Construct the Markov chain for $D(x)$ and $D(x')$
- Compute and compare hitting probabilities

Problem: Markov chain has exp-many states ⇒ cannot store it explicitly

Need **space-efficient** algorithm for computing hitting probabilities with **implicit access** to the Markov chain



Polyspace algorithm for computing hitting probabilities in a Markov chain

Lemma [Simon'81]: If M is a Markov chain with at most 2^L states

- the initial distribution places all mass on one state,
- there is a set F of final states each with only one self-transition,
- every non-final state has outgoing transition probability 0 or ½.

Then there is an $O(L^6)$-space deterministic algorithm that computes the hitting probability of every state in F.

Polyspace algorithm for computing hitting probabilities in a Markov chain

Lemma [Simon'81]: If M is a Markov chain with at most 2^L states

- the initial distribution places all mass on one state,
- there is a set F of final states each with only one self-transition,
- every non-final state has outgoing transition probability 0 or ½.

Then there is an $O(L^6)$-space deterministic algorithm that computes the hitting probability of every state in F.

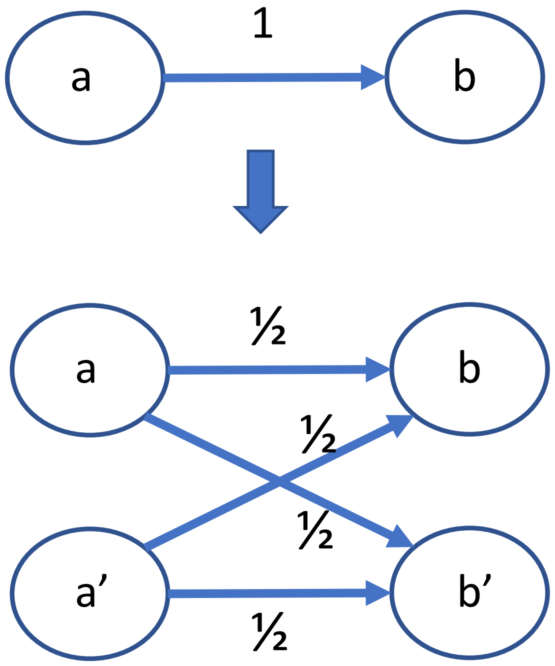
Note: to use the algorithm, we need to replace all transitions labelled by 1 in the state graph of the BPWhile program:

Polyspace algorithm for computing hitting probabilities in a Markov chain

Lemma [Simon'81]: If M is a Markov chain with at most 2^L states

- the initial distribution places all mass on one state,
- there is a set F of final states each with only one self-transition,
- every non-final state has outgoing transition probability 0 or ½.

Then there is an $O(L^6)$-space deterministic algorithm that computes the hitting probability of every state in F.



Note: to use the algorithm, we need to replace all transitions labelled by 1 in the state graph of the BPWhile program:

- Clone all states
- For each state *a* with outgoing edge w.p. 1 replace it by two edges:
  - Edge (a,b) with weight ½ to original state
  - Edge (a,b') with weight ½ to the clone-state b' of b

PSPACE membership: exponentially long numbers

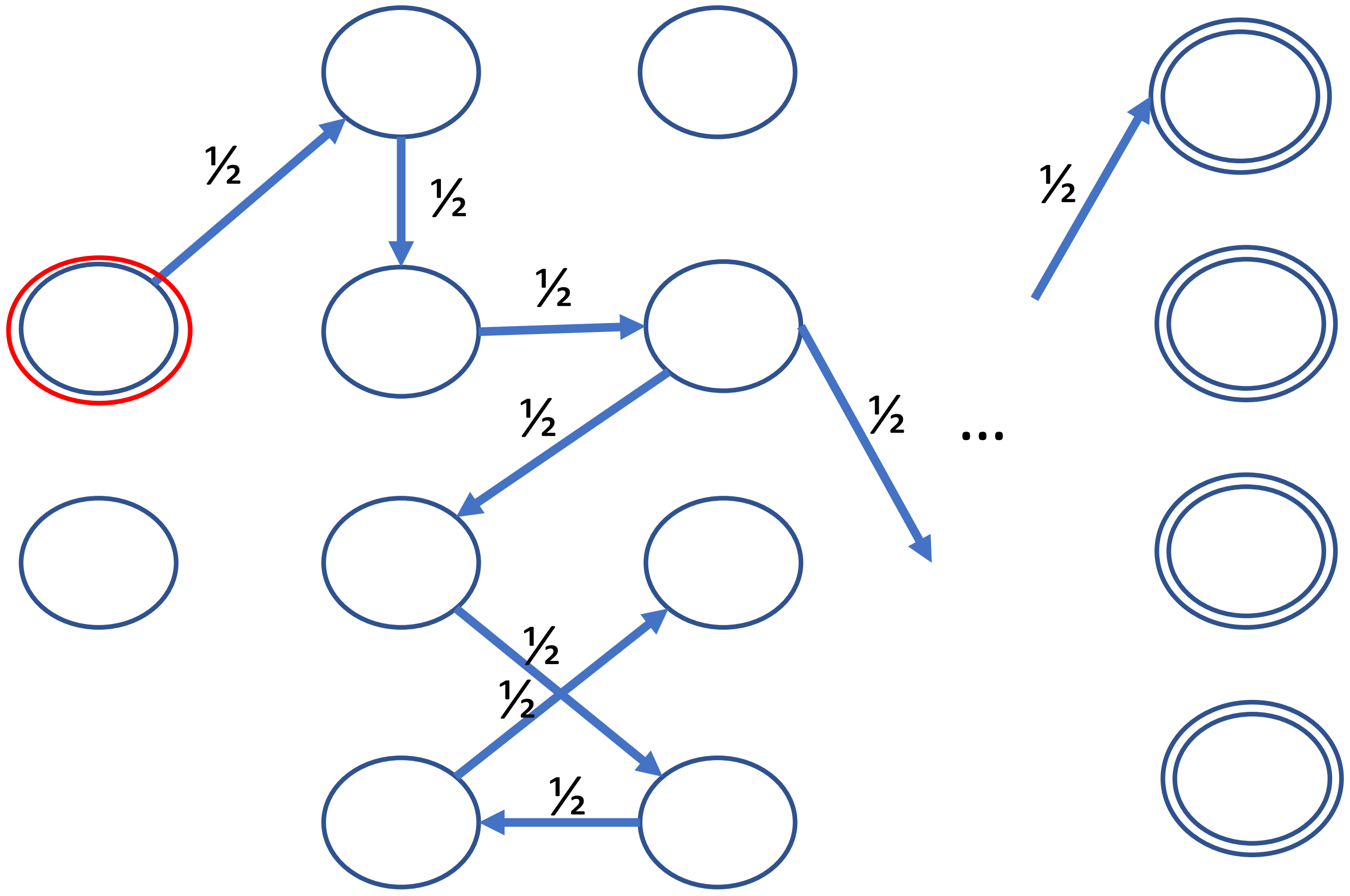
For a program D

for all neighboring inputs x, x':

- Construct the Markov chain for $D(x)$ and $D(x')$
- Compare hitting probabilities of the final states with the same values

Problem:

Markov chain has exp-many states



PSPACE membership: exponentially long numbers

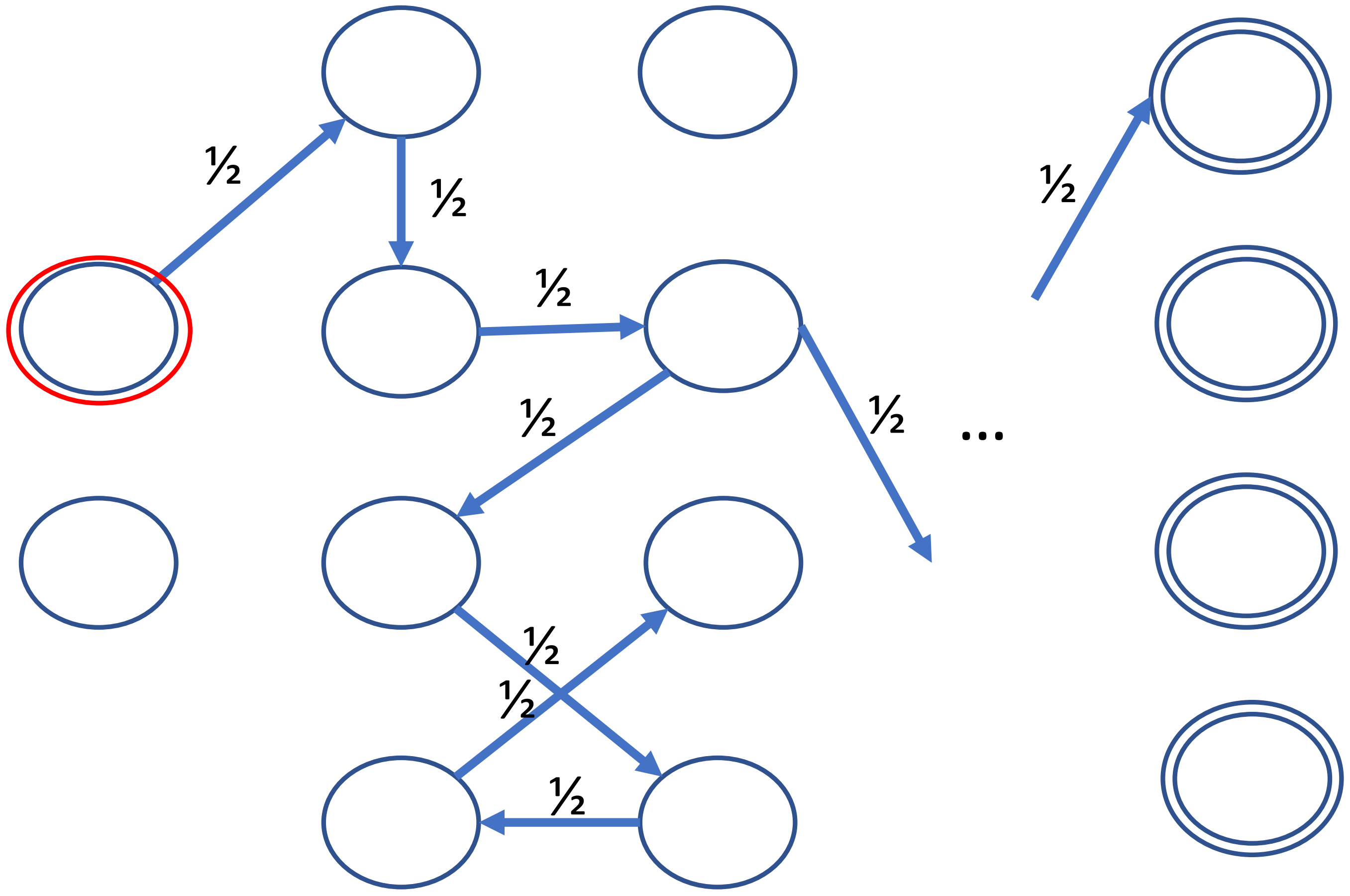
For a program D

for all neighboring inputs x, x':

- Construct the Markov chain for $D(x)$ and $D(x')$
- Compare hitting probabilities of the final states with the same values

Problem:

Markov chain has exp-many states $\Rightarrow$ hitting probabilities can be as small as $\frac{1}{2^{exp}}$



PSPACE membership: exponentially long numbers

For a program D

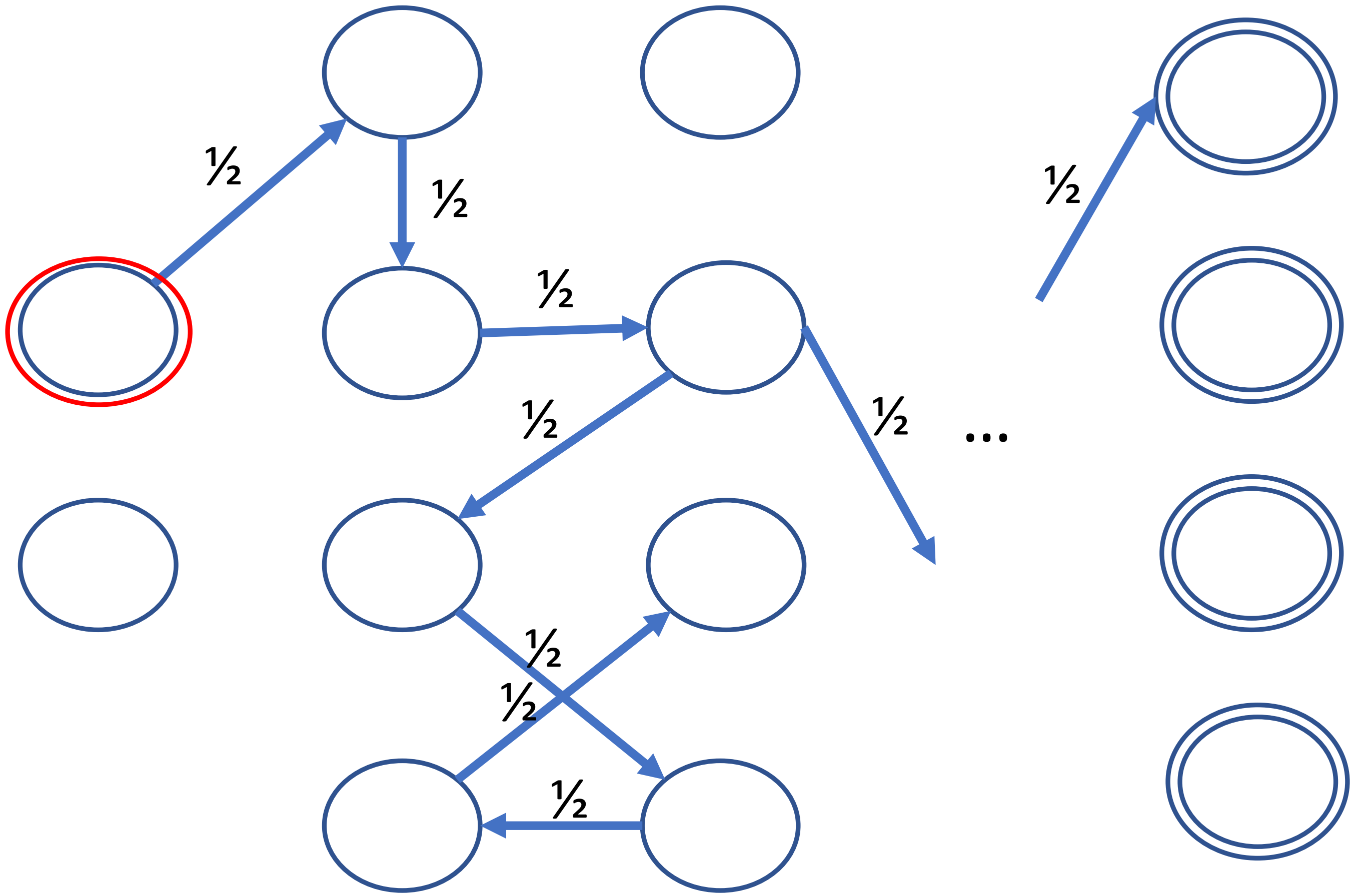
for all neighboring inputs x, x':

- Construct the Markov chain for $D(x)$ and $D(x')$
- Compare hitting probabilities of the final states with the same values

Problem:

Markov chain has exp-many states
⇒ hitting probabilities can be as small as $\frac{1}{2^{exp}}$

⇒ numbers are exponentially long



Operations with exponentially long numbers

1	**0**	**1**	**1**	**1**	**…**	**1**	**0**	**0**	**1**

+

0	**0**	**1**	**0**	**1**	**…**	**1**	**1**	**1**	**1**

1	**1**	**1**	**0**	**1**	**…**	**1**	**0**	**0**	**0**

Operations with exponentially long numbers

	1	**0**	**1**	**1**	**1**	**…**	**1**	**0**	**0**	**1**
+										
	0	**0**	**1**	**0**	**1**	**…**	**1**	**1**	**1**	**1**
	1	**1**	**1**	**0**	**1**	**…**	**1**	**0**	**0**	**0**

Uniform family of log-depth circuits:

- One logspace algorithm provides implicit access to the circuits
- Each circuit has log-depth and poly size

Operations with exponentially long numbers

1	**0**	**1**	**1**	**1**	**…**	**1**	**0**	**0**	**1**

+

0	**0**	**1**	**0**	**1**	**…**	**1**	**1**	**1**	**1**

1	**1**	**1**	**0**	**1**	**…**	**1**	**0**	**0**	**0**

Uniform family of log-depth circuits:

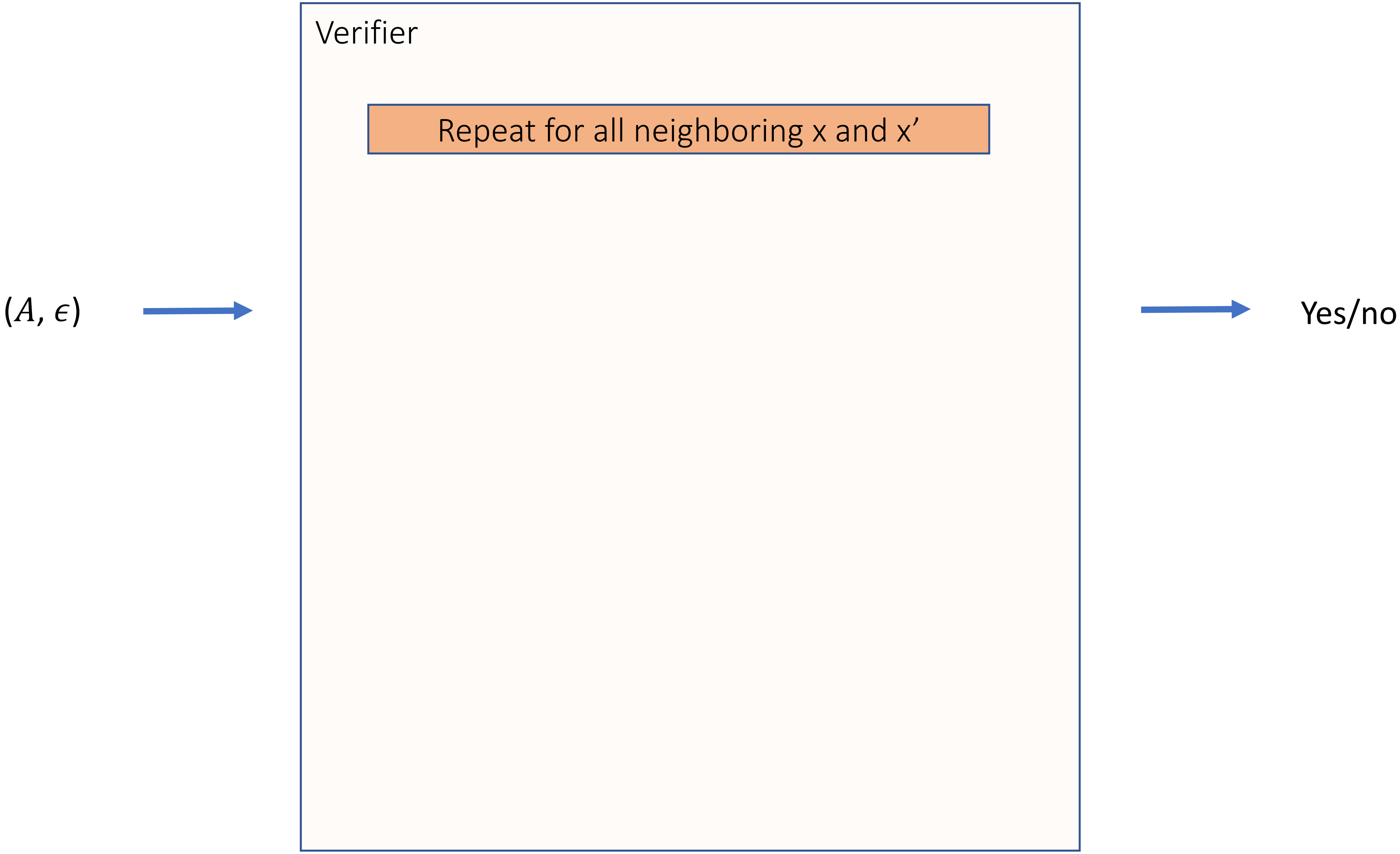
- One logspace algorithm provides implicit access to the circuits
- Each circuit has log-depth and poly size

Lemma:

Uniform families of log-depth circuits exist for:

- Comparison
- Addition
- Multiplication by a fixed rational number
- Multiplication [Ofman’62]
- Square roots [Reif’86]

Polyspace membership algorithm for (ϵ,0)-DP



Polyspace membership algorithm for (ϵ,0)-DP

(A, ϵ) →

Verifier

Repeat for all neighboring x and x'

Provide access to Markov chain corresponding to A(x) and A(x')

→ Yes/no

Polyspace membership algorithm for (ϵ,0)-DP

(A, ϵ) →

Verifier

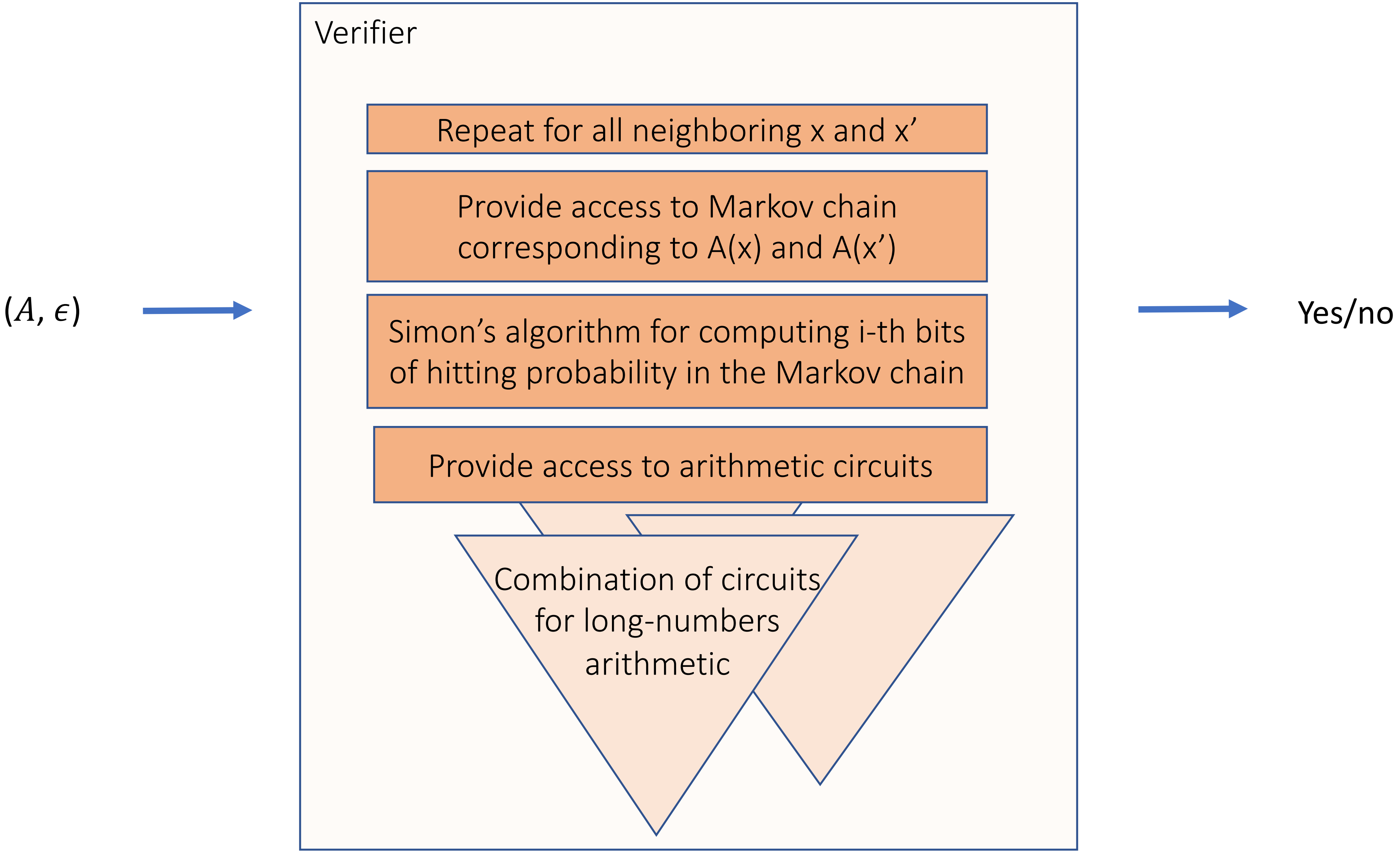
Repeat for all neighboring x and x'

Provide access to Markov chain corresponding to A(x) and A(x')

Simon's algorithm for computing i-th bits of hitting probability in the Markov chain

→ Yes/no

Polyspace membership algorithm for (ϵ,0)-DP



Extending the result to (ϵ, δ)-DP, RDP, zCDP, TCDP

For verifying (ϵ, δ)-DP we use a point-wise definition

- Additionally use a summation of an exponentially long numbers

$$\sum_{o \in \{0,1\}^l \cup \{\perp\}} \left(\delta_{x,x'}(o)\right) \leq \delta$$

$$\delta_{x,x'}(o) = \max(\Pr[C(x) = o] - e^{\epsilon} \Pr[C(x') = o])$$

Extending the result to (ϵ, δ)-DP, RDP, zCDP, TCDP

For verifying (ϵ, δ)-DP we use a point-wise definition

- Additionally use a summation of an exponentially long numbers

$$\sum_{o \in \{0,1\}^l \cup \{\perp\}} \left(\delta_{x,x'}(o)\right) \leq \delta$$

$$\delta_{x,x'}(o) = \max(\Pr[C(x) = o] - e^{\epsilon} \Pr[C(x') = o])$$

For verifying RDP, zCDP, TCDP

- Need to compute Rényi divergence
    - Logarithms
    - Exponentiations to rational degrees

$$D_\alpha(P|Q) = \frac{1}{\alpha - 1} \log \sum \frac{p_i^\alpha}{q_i^{\alpha-1}}$$

Extending the result to (ϵ, δ)-DP, RDP, zCDP, TCDP

For verifying (ϵ, δ)-DP we use a point-wise definition

- Additionally use a summation of an exponentially long numbers

$$\sum_{o \in \{0,1\}^l \cup \{\perp\}} \left(\delta_{x,x'}(o)\right) \leq \delta$$

$$\delta_{x,x'}(o) = \max(\Pr[C(x) = o] - e^{\epsilon} \Pr[C(x') = o])$$

For verifying RDP, zCDP, TCDP

- Need to compute Rényi divergence
  - Logarithms
  - Exponentiations to rational degrees
- Get infinite fractions
  - Computations with a fixed precision η
  - Consider gap-versions of the problem

$$D_\alpha(P|Q) = \frac{1}{\alpha - 1} \log \sum \frac{p_i^\alpha}{q_i^{\alpha-1}}$$

Results and future work

- We showed PSPACE-completeness for the problems of checking:
    - Pure-DP
    - Approximate-DP
    - Gap-RDP
    - Gap-zCDP
    - Gap-TCDP

Results and future work

- We showed PSPACE-completeness for the problems of checking:
    - Pure-DP
    - Approximate-DP
    - Gap-RDP
    - Gap-zCDP
    - Gap-TCDP

- Possibly can extend the result to show PSPACE-completeness of verifying accuracy

Results and future work

- We showed PSPACE-completeness for the problems of checking:
  - Pure-DP
  - Approximate-DP
  - Gap-RDP
  - Gap-zCDP
  - Gap-TCDP

- Possibly can extend the result to show PSPACE-completeness of verifying accuracy
- Improve the exact polynomial in the space complexity of the algorithm
  - Improved analysis and more efficient algorithms for Markov chains analysis and long-numbers arithmetic operations are needed for tighter results

Polyspace algorithm for RDP, zCDP, TCDP

An algorithm is **RDP/zCDP/TCDP** if for a privacy parameter ρ and a fixed/any/bounded $\alpha > 1$ Rényi divergence for any neighboring inputs x, x' is at most $\rho\alpha$

[Mironov'17],[Dwork-Rothblum'16,Bun-Steinke'16], [Bun,Dwork,Rothblum,Steinke'18]

$$D_\alpha(P|Q) = \frac{1}{\alpha - 1} \log \sum \frac{p_i^\alpha}{q_i^{\alpha-1}}$$

$$D_\alpha(C(x)|C(x')) \leq \rho\alpha$$

New problems:

- To compute Rényi divergence we compute
  - Logarithms
  - Exponentiations to rational degrees
- Hence, get infinite fractions

Solution:

- Computations with a fixed precision η
- Consider gap-versions of the problem

We define Gap-RDP on (C, ρ, α, η) as follows:

- Yes-instance, if for all neighboring x, x'

$$D_\alpha(C(x)|C(x')) \leq \rho\alpha$$

- No-instance, if for at least one neighboring x, x'

$$D_\alpha(C(x)|C(x')) \geq \rho\alpha + \frac{1}{2^\eta}$$

Polyspace algorithm for zCDP

Another problem: an algorithm is **zCDP** if **for all values of α** Rényi divergence for any neighboring inputs x, x' is bounded by $\rho\alpha$

Solution: showed that it is sufficient to check values of α from **the bounded range**:

Lemma:

C is ρ-zCDP, then for all neighboring x, x': $D_\alpha(C(x) | C(x')) \leq \rho\alpha$

C is not $(\rho + 2^{-\eta})$-zCDP, then exist neighboring x, x', exists $\alpha \in (1, 1 + 2^{poly(n)}/\rho)$:

- α is a multiple of $2^{-\eta}$
- $D_\alpha(C(x) | C(x')) \geq \rho\alpha + 2^{-\eta-1}$

Polyspace membership algorithm for Gap-zCDP