Logarithmic Space

Lee Zaniewski[1]

[1]Lane Department of Computer Science and Electrical Engineering
West Virginia University

Outline

Outline

Outline

Outline

Definitions

L

The set of problems that can be solved in $O(\log n)$ space using a Deterministic Turing Machine.

NL

The set of problems that can be solved in $O(\log n)$ space using a Non-deterministic Turing Machine.

Definitions

L

The set of problems that can be solved in $O(\log n)$ space using a Deterministic Turing Machine.

NL

The set of problems that can be solved in $O(\log n)$ space using a Non-deterministic Turing Machine.

Definitions

L

The set of problems that can be solved in $O(\log n)$ space using a Deterministic Turing Machine.

NL

The set of problems that can be solved in $O(\log n)$ space using a Non-deterministic Turing Machine.

Outline

Relation between **NL** and $\mathbf{NC}_2$

Theorem

$\mathbf{NL} \subseteq \mathbf{NC}_2$

Proof

This follows from the reachability method. In order to determine whether an input, x, is accepted by a nondeterministic logarithmic-space Turing machine **N**, we simply produce the configuration graph of **N** on input x, and determine whether an accepting node is reachable from the initial node. Since Reachability is in $\mathbf{NC}_2$, the theorem follows.

Relation between **NL** and $\mathbf{NC}_2$

Theorem

$\mathbf{NL} \subseteq \mathbf{NC}_2$

Proof

This follows from the reachability method. In order to determine whether an input, x, is accepted by a nondeterministic logarithmic-space Turing machine **N**, we simply produce the configuration graph of **N** on input x, and determine whether an accepting node is reachable from the initial node. Since Reachability is in $\mathbf{NC}_2$, the theorem follows.

Relation between **NL** and $\mathbf{NC}_2$

Theorem

$\mathbf{NL} \subseteq \mathbf{NC}_2$

Proof

This follows from the reachability method. In order to determine whether an input, x, is accepted by a nondeterministic logarithmic-space Turing machine **N**, we simply produce the configuration graph of **N** on input x, and determine whether an accepting node is reachable from the initial node. Since Reachability is in $\mathbf{NC}_2$, the theorem follows.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

Relation between $\mathbf{NC}_1$ and $\mathbf{L}$

Theorem

$\mathbf{NC}_1 \subseteq \mathbf{L}$

Proof

We will prove this by giving an algorithm that will evaluate in logarithmic space any uniform family of circuits with logarithmic depth. This algorithm will be composed of three logarithmic-space algorithms.

Algorithm No. 1

This algorithm simply generates the family of circuits to be evaluated. These circuits may have five different gate, True, False, NOT, OR, and AND gates. A NOT gate can have one predecessor, and OR and AND gates have exactly two predecessors.

cont

Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

cont

Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

cont

Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

cont

Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

cont

Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

cont

Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

cont

Algorithm No. 2

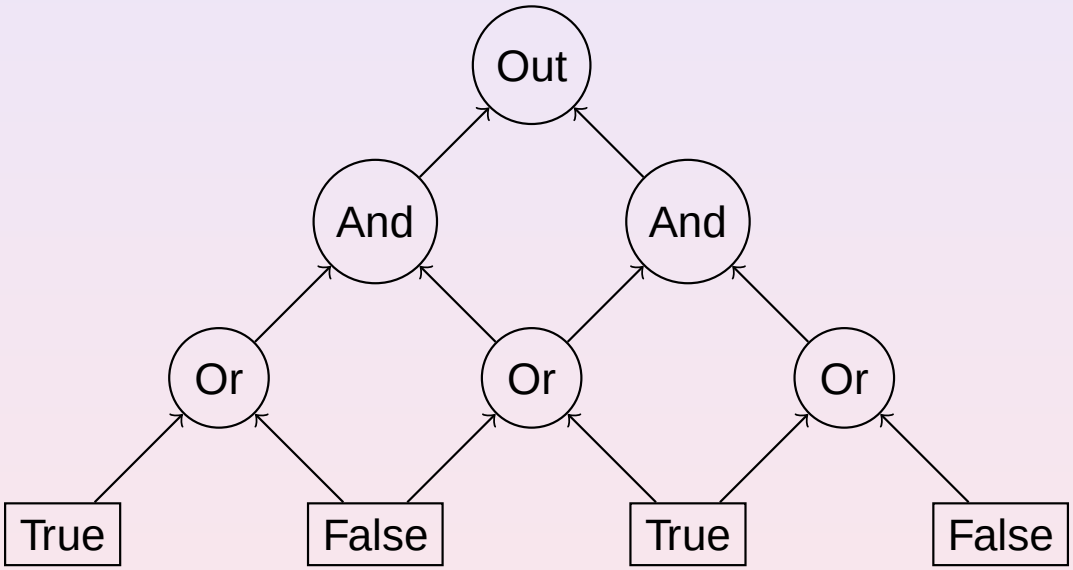
A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

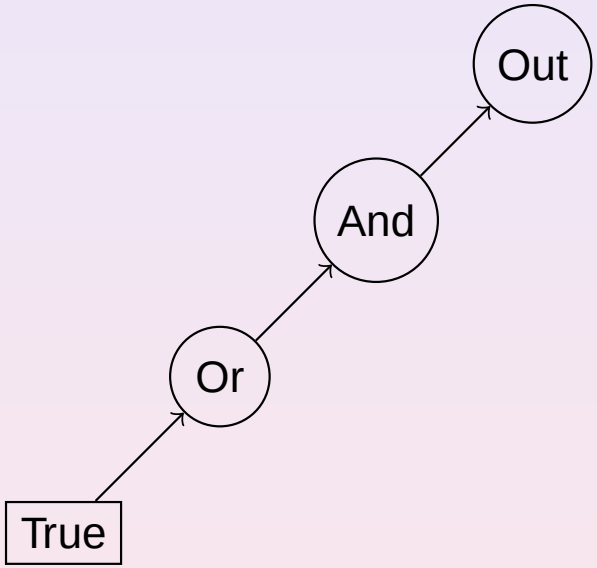
cont

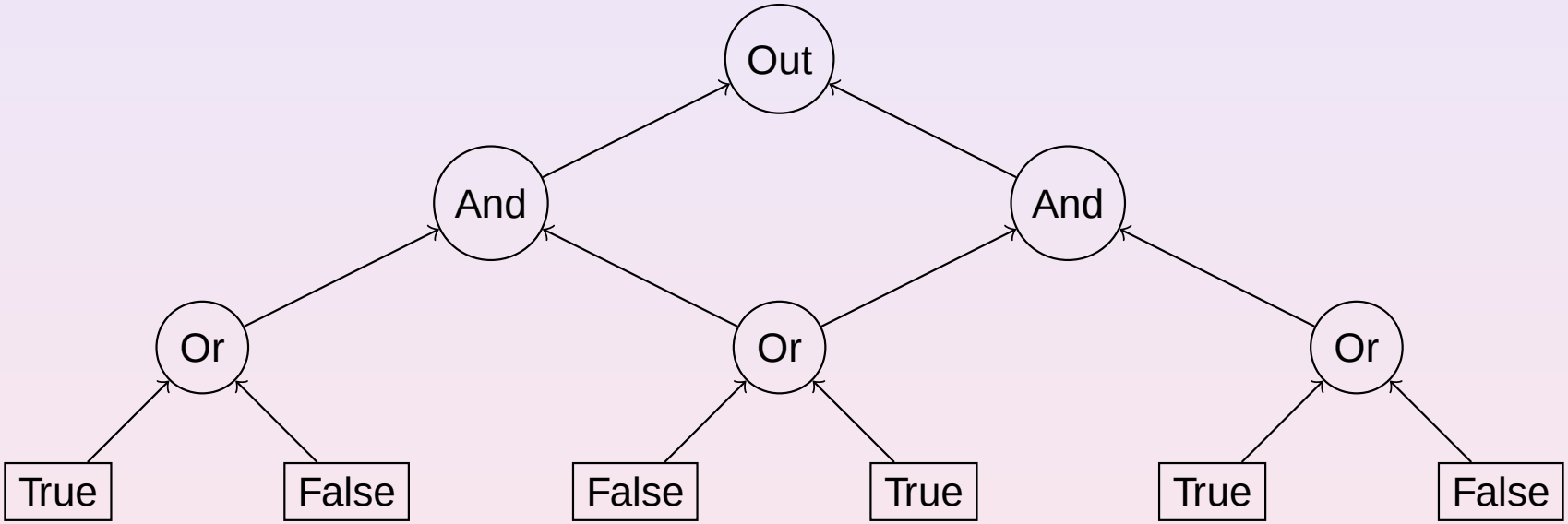
Algorithm No. 2

A gate may be the predecessor to more than one gate. This algorithm takes this circuit and transforms it into one where each gate is predecessor to exactly one gate. We consider all possible paths in the original circuit, starting from the output and going towards the inputs. How can we represent these paths? We represent the path by a bit string of length equal to that of the path, where each bit indicates whether the next gate visited in the path is the first or second predecessor of the previous gate. Since the given circuit has logarithmic depth, this takes up logarithmic space. The equivalent tree-like circuit that we construct will have these paths as gates. Note that gates reachable by multiple paths with be represented once for each of these paths. These paths can be constructed one by one, reusing space so that the algorithm fits in logarithmic space.

Example




Out
And
Or
True


Out
And
And
Or
Or
Or
True
False
False
True
True
False

cont

Algorithm No. 3

This algorithm recursively evaluates the circuit outputted by Algorithm No. 2. For an AND gate, we evaluate the first predecessor. If the outcome is False, the gate's value is false. If it is true we need to evaluate the second predecessor the gate's value will be the same as it's second predecessor. For OR gates, the roles of true and false are reversed. For NOT gates, we simply reverse the single predecessor gate's value. True and False gates are already evaluated. When the output is evaluated, then the algorithm is done.

Corollary

From the previous two theorems, we can see that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

cont

Algorithm No. 3

This algorithm recursively evaluates the circuit outputted by Algorithm No. 2. For an AND gate, we evaluate the first predecessor. If the outcome is False, the gate's value is false. If it is true we need to evaluate the second predecessor the gate's value will be the same as it's second predecessor. For OR gates, the roles of true and false are reversed. For NOT gates, we simply reverse the single predecessor gate's value. True and False gates are already evaluated. When the output is evaluated, then the algorithm is done.

Corollary

From the previous two theorems, we can see that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

cont

Algorithm No. 3

This algorithm recursively evaluates the circuit outputted by Algorithm No. 2. For an AND gate, we evaluate the first predecessor. If the outcome is False, the gate's value is false. If it is true we need to evaluate the second predecessor the gate's value will be the same as it's second predecessor. For OR gates, the roles of true and false are reversed. For NOT gates, we simply reverse the single predecessor gate's value. True and False gates are already evaluated. When the output is evaluated, then the algorithm is done.

Corollary

From the previous two theorems, we can see that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

cont

Algorithm No. 3

This algorithm recursively evaluates the circuit outputted by Algorithm No. 2. For an AND gate, we evaluate the first predecessor. If the outcome is False, the gate's value is false. If it is true we need to evaluate the second predecessor the gate's value will be the same as it's second predecessor. For OR gates, the roles of true and false are reversed. For NOT gates, we simply reverse the single predecessor gate's value. True and False gates are already evaluated. When the output is evaluated, then the algorithm is done.

Corollary

From the previous two theorems, we can see that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

cont

Algorithm No. 3

This algorithm recursively evaluates the circuit outputted by Algorithm No. 2. For an AND gate, we evaluate the first predecessor. If the outcome is False, the gate's value is false. If it is true we need to evaluate the second predecessor the gate's value will be the same as it's second predecessor. For OR gates, the roles of true and false are reversed. For NOT gates, we simply reverse the single predecessor gate's value. True and False gates are already evaluated. When the output is evaluated, then the algorithm is done.

Corollary

From the previous two theorems, we can see that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

cont

Algorithm No. 3

This algorithm recursively evaluates the circuit outputted by Algorithm No. 2. For an AND gate, we evaluate the first predecessor. If the outcome is False, the gate's value is false. If it is true we need to evaluate the second predecessor the gate's value will be the same as it's second predecessor. For OR gates, the roles of true and false are reversed. For NOT gates, we simply reverse the single predecessor gate's value. True and False gates are already evaluated. When the output is evaluated, then the algorithm is done.

Corollary

From the previous two theorems, we can see that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$.

Outline

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in L if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Reachability

Theorem

Reachability is **NL-complete**.

Proof

Immerman's Theorem shows that reachability is in **NL**.
We shall now show that any language in **NL** is reducible to Reachability. Suppose the language **L** is decided by the $\log n$ space-bounded Turing machine N. Given input x, we can construct in logarithmic space the configuration graph of N on input x, denoted $G(N, x)$, same as we did in the Reachability method. We can assume that $G(N, x)$ has a single accepting node, called n. It certainly has a single initial node, called 1. x is in **L** if and only if the produced instance of Reachability has a "yes" answer.

Outline

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.

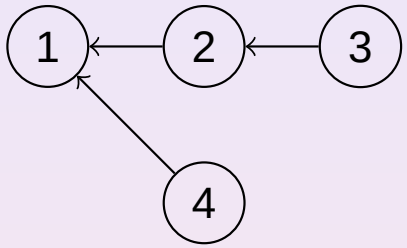
2SAT

Theorem

2SAT is NL-complete

Proof

2SAT is in NL, see Theorem 9.1 in the book. To prove completeness, we shall reduce UNREACHABILITY to 2SAT. What's wrong with this? **NL** = **coNL** according to Immerman's Theorem. We'll start with a graph that is acyclic. What's wrong with this? Acyclic Rechability is still **NL-complete**. We reduce the unreachability problem for such a graph to 2SAT by simulating each edge (x, y) of the graph by a clause $(\neg x \vee y)$, where we have a Boolean variable for each node in the graph. If we now add the clauses (s) and $(\neg t)$ for the start and target nodes s and t, it is clear the resulting instance of 2SAT is satisfiable if and only if there is no path from s to t in the given graph.



Equation

$s = 1$
$t = 3$
$(x_1)(\neg x_1, x_2)(\neg x_2, x_3)(\neg x_1, x_4)(\neg x_3)$

Outline

L-completeness

L-completeness

All languages in **L** are **L-complete**. Why? This is because a reduction is meaningful only within a class that is computationally stronger than the reduction.

L-completeness

L-completeness

All languages in **L** are **L-complete**. Why? This is because a reduction is meaningful only within a class that is computationally stronger than the reduction.

Outline

Informal Definition

Nondeterminism

An alternative definition of a Nondeterministic Turing machine based on configurations is that a configuration of an NDTM leads to acceptance if and only if it is either a final accepting configuration or at least one of its successors leads to acceptance. Each configuration can be thought of as an OR of its successor configurations.
A machine deciding complement of the same language could be thought of as having all ANDs.

Alternation

Suppose that we allow both modes in a nondeterministic machine. Some configurations would accept only if all of its successors did, and is therefore an AND configuration, while others accept if any of their successors accept, and is therefore an OR configuration.

Informal Definition

Nondeterminism

An alternative definition of a Nondeterministic Turing machine based on configurations is that a configuration of an NDTM leads to acceptance if and only if it is either a final accepting configuration or at least one of its successors leads to acceptance. Each configuration can be thought of as an OR of its successor configurations.
A machine deciding complement of the same language could be thought of as having all ANDs.

Alternation

Suppose that we allow both modes in a nondeterministic machine. Some configurations would accept only if all of its successors did, and is therefore an AND configuration, while others accept if any of their successors accept, and is therefore an OR configuration.

Informal Definition

Nondeterminism

An alternative definition of a Nondeterministic Turing machine based on configurations is that a configuration of an NDTM leads to acceptance if and only if it is either a final accepting configuration or at least one of its successors leads to acceptance. Each configuration can be thought of as an OR of its successor configurations.
A machine deciding complement of the same language could be thought of as having all ANDs.

Alternation

Suppose that we allow both modes in a nondeterministic machine. Some configurations would accept only if all of its successors did, and is therefore an AND configuration, while others accept if any of their successors accept, and is therefore an OR configuration.

Informal Definition

Nondeterminism

An alternative definition of a Nondeterministic Turing machine based on configurations is that a configuration of an NDTM leads to acceptance if and only if it is either a final accepting configuration or at least one of its successors leads to acceptance. Each configuration can be thought of as an OR of its successor configurations.
A machine deciding complement of the same language could be thought of as having all ANDs.

Alternation

Suppose that we allow both modes in a nondeterministic machine. Some configurations would accept only if all of its successors did, and is therefore an AND configuration, while others accept if any of their successors accept, and is therefore an OR configuration.

Informal Definition

Nondeterminism

An alternative definition of a Nondeterministic Turing machine based on configurations is that a configuration of an NDTM leads to acceptance if and only if it is either a final accepting configuration or at least one of its successors leads to acceptance. Each configuration can be thought of as an OR of its successor configurations.
A machine deciding complement of the same language could be thought of as having all ANDs.

Alternation

Suppose that we allow both modes in a nondeterministic machine. Some configurations would accept only if all of its successors did, and is therefore an AND configuration, while others accept if any of their successors accept, and is therefore an OR configuration.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Formal Definition

Alternating Turing Machine

An alternating Turing machine is a nondeterministic Turing machine $N = (K, \Sigma, \Delta, s)$ in which the set of states, K, is partitioned into to sets, K_{AND} and K_{OR}. We'll now define the term eventually accepting. Any leaf with state "yes" is eventually accepting. Any configuration with state in K_{AND} is eventually accepting if and only if all of its successor configurations are eventually accepting. Any configuration with state in K_{OR} is eventually accepting if at least one of its successor configurations is eventually accepting. N accepts input x if the initial configuration is eventually accepting.

ATIME and ASPACE

ATIME and **ASPACE** are defined in a similar manner for alternating Turing machines that **NTIME** and **NSPACE** are defined for Nondeterministic Turing machines.

Outline

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

MONOTONE CIRCUIT PROBLEM

Theorem

The MONOTONE CIRCUIT VALUE problem is **AL-complete**.

Proof

We will begin by proving that the problem is in **AL**. Any guesses? The input of our alternating Turing machine is a circuit. The machine examines the output gate of the circuit. If it is an AND gate, then it enters an AND state, if it is an OR gate, then it enters an OR state. The machine nondeterministically chooses one predecessor and does the same thing. If an input gate is encountered, the machine accepts on True, and rejects on False. We can use induction to prove that this machine accepts if and only if the circuit evaluates to True. Additionally, the machine needs only remember the gate currently under consideration, and therefore only logarithmic space is used.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is "yes" then the gate is a true gate, and false if it is a "no" state.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is "yes" then the gate is a true gate, and false if it is a "no" state.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is "yes" then the gate is a true gate, and false if it is a "no" state.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is "yes" then the gate is a true gate, and false if it is a "no" state.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is "yes" then the gate is a true gate, and false if it is a "no" state.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is ”yes” then the gate is a true gate, and false if it is a ”no” state.

Continued

We will now show that any language in **AL** is reducible to the problem. Consider such a language, L, the corresponding Turing Machine, M, and an input, x. We shall construct a circuit such that it evaluates to True if and only if M accepts x.
The gates of the circuit are all pairs of the form (C, i), where C is a configuration of N on input x, and i stands for the step number, and integer 0 and $|x|^k$, the time bound of the machine. The purpose of the step number is to make the circuit acyclic. There is an arc from gate (C_1, i) to (C_2, j) if and only if C_2 yields in one step C_1 and $j = i + 1$ f C is in K_{OR}, the gate is an OR gate; if it is in K_{AND} it is an AND gate; if it is ”yes” then the gate is a true gate, and false if it is a ”no” state.

Outline

AL=P

Theorem

AL = P

Proof

Both classes are closed under reductions, and they have the same complete problem.
MONOTONE CIRCUIT VALUE is of the same difficulty as CIRCUIT VALUE.

AL=P

Theorem

AL = P

Proof

Both classes are closed under reductions, and they have the same complete problem. MONOTONE CIRCUIT VALUE is of the same difficulty as CIRCUIT VALUE.

Outline

Definitions

Undirected Reachability

Given the undirected graph G and two nodes, s and t, are s and t in the same connected component?

RL

A problem is in class RL if a probabilistic log-space Turing Machine exists such that on any input x, the probability that M accepts x is greater than 1/2 if x is in the language, and zero otherwise. This is similar to the class RP, but in logarithmic space. We will prove that Undirected Reachability is in RL.

Definitions

Undirected Reachability

Given the undirected graph G and two nodes, s and t, are s and t in the same connected component?

RL

A problem is in class RL if a probabilistic log-space Turing Machine exists such that on any input x, the probability that M accepts x is greater than 1/2 if x is in the language, and zero otherwise.This is similar to the class RP, but in logarithmic space. We will prove that Undirected Reachability is in RL.

Definitions

Undirected Reachability

Given the undirected graph G and two nodes, s and t, are s and t in the same connected component?

RL

A problem is in class RL if a probabilistic log-space Turing Machine exists such that on any input x, the probability that M accepts x is greater than 1/2 if x is in the language, and zero otherwise.This is similar to the class RP, but in logarithmic space. We will prove that Undirected Reachability is in RL.

Outline

Algorithm

Algorithm

Start at node s. Choose an edge, (s, x) at random and follow the edge to the new node, x. Do the same with this edge. Repeat this $2n|E|$ times. If t is reached, then accept, otherwise reject.

Algorithm

Algorithm

Start at node s. Choose an edge, (s, x) at random and follow the edge to the new node, x. Do the same with this edge. Repeat this $2n|E|$ times. If t is reached, then accept, otherwise reject.

Algorithm

Algorithm

Start at node s. Choose an edge, (s, x) at random and follow the edge to the new node, x. Do the same with this edge. Repeat this $2n|E|$ times. If t is reached, then accept, otherwise reject.

Algorithm

Algorithm

Start at node s. Choose an edge, (s, x) at random and follow the edge to the new node, x. Do the same with this edge. Repeat this $2n|E|$ times. If t is reached, then accept, otherwise reject.

Outline

Analysis

Single Step

Let v_t be the node that is visited at an individual step. Let $v_t = i$ and (i, j) be in E. Whats the probability that $v_{t+1} = j$? prob$[v_{t+1} = j] = \frac{1}{d_i}$ where d_i is the degree of node i. Let $p_t[i]$ be the probability that at any given step, t, $v_t = i$. For the rest of the analysis, we will also assume that each node has a self loop.

Analysis

Single Step

Let v_t be the node that is visited at an individual step. Let $v_t = i$ and (i, j) be in E. Whats the probability that $v_{t+1} = j$? $\text{prob}[v_{t+1} = j] = \frac{1}{d_i}$ where d_i is the degree of node i. Let $p_t[i]$ be the probability that at any given step, t, $v_t = i$. For the rest of the analysis, we will also assume that each node has a self loop.

Analysis

Single Step

Let v_t be the node that is visited at an individual step. Let $v_t = i$ and (i, j) be in E. Whats the probability that $v_{t+1} = j$? prob$[v_{t+1} = j] = \frac{1}{d_i}$ where d_i is the degree of node i. Let $p_t[i]$ be the probability that at any given step, t, $v_t = i$. For the rest of the analysis, we will also assume that each node has a self loop.

Analysis

Single Step

Let v_t be the node that is visited at an individual step. Let $v_t = i$ and (i, j) be in E. Whats the probability that $v_{t+1} = j$? prob$[v_{t+1} = j] = \frac{1}{d_i}$ where d_i is the degree of node i. Let $p_t[i]$ be the probability that at any given step, t, $v_t = i$. For the rest of the analysis, we will also assume that each node has a self loop.

Analysis

Single Step

Let v_t be the node that is visited at an individual step. Let $v_t = i$ and (i, j) be in E. Whats the probability that $v_{t+1} = j$? prob$[v_{t+1} = j] = \frac{1}{d_i}$ where d_i is the degree of node i. Let $p_t[i]$ be the probability that at any given step, t, $v_t = i$. For the rest of the analysis, we will also assume that each node has a self loop.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.

Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.

Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.

Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.

Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.

Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.

Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.

Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.

Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.

Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.

Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.

Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.

Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.

Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.

Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.

Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.
Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.
Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.
Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.
Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.
Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.
Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Analysis Continued

Lemma

If G is a connected graph, then $\lim_{t\to\infty} p_t[i] = \frac{d_i}{2|E|}$ for all i.

Proof

Since the random walk is equally likely to visit each neighbor of the current node, we can think that p_{t+1}'s are formed from the p_t's as follows: Each node i splits its $p_t[i]$ into d_i equal parts, and passes one such portion to each one of its neighbors (including itself). Each node adds up the portions received from its neighbors, resulting in $p_{t+1}[i]$.
Let $\delta[i] = p_t[i] - \frac{d_i}{2|E|}$ be the deviation at node i from the asymptotic value. . Since $p_t[i] = \frac{d_i}{2|E|} + d_t[i]$, the splitting and passing can be thought of keeping the $\frac{d_i}{2|E|}$ part and passing the $\delta_t[i]$'s.
Let $\Delta_t = \sum_{i\in V} |\delta_t[i]|$, be the sum of the absolute values of the deviations, also called the total absolute deviation.
Because the δ_t's are simply being passed from one node to its neighbor, the sum of the absolute values of the graphs cannot increase. However, it can decrease if two $\delta_t[i]$'s of opposite sign ever meet at a node.

Continued

Continued

Since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t \geq \frac{\Delta_i}{2|V|}$, and a node $^-$ with $\delta_t \leq -\frac{\Delta_t}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^-. The positive and negative deviation can travel along this path. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. This is the same for the negative deviation. After $m \leq n$ steps, a positive deviation of at east $\frac{\Delta}{2|V|^n}$ will cancel an equal amount negative deviation. Thus in n steps, the maximum total deviation is now $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. So the limit of $p_t[i]$ as the sum of the absolute values of the deviation approaches zero is $\frac{d_i}{2|E|}$.

Continued

Continued

Since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t \geq \frac{\Delta_i}{2|V|}$, and a node $^-$ with $\delta_t \leq -\frac{\Delta_t}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^-. The positive and negative deviation can travel along this path. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. This is the same for the negative deviation. After $m \leq n$ steps, a positive deviation of at east $\frac{\Delta}{2|V|^n}$ will cancel an equal amount negative deviation. Thus in n steps, the maximum total deviation is now $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. So the limit of $p_t[i]$ as the sum of the absolute values of the deviation approaches zero is $\frac{d_i}{2|E|}$.

Continued

Continued

Since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t \geq \frac{\Delta_i}{2|V|}$, and a node $^-$ with $\delta_t \leq -\frac{\Delta_t}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^-. The positive and negative deviation can travel along this path. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. This is the same for the negative deviation. After $m \leq n$ steps, a positive deviation of at east $\frac{\Delta}{2|V|^n}$ will cancel an equal amount negative deviation. Thus in n steps, the maximum total deviation is now $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. So the limit of $p_t[i]$ as the sum of the absolute values of the deviation approaches zero is $\frac{d_i}{2|E|}$.

Continued

Continued

Since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t \geq \frac{\Delta_i}{2|V|}$, and a node $^-$ with $\delta_t \leq -\frac{\Delta_t}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^-. The positive and negative deviation can travel along this path. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. This is the same for the negative deviation. After $m \leq n$ steps, a positive deviation of at east $\frac{\Delta}{2|V|^n}$ will cancel an equal amount negative deviation. Thus in n steps, the maximum total deviation is now $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. So the limit of $p_t[i]$ as the sum of the absolute values of the deviation approaches zero is $\frac{d_i}{2|E|}$.

Continued

Continued

Since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t \geq \frac{\Delta_i}{2|V|}$, and a node $^-$ with $\delta_t \leq -\frac{\Delta_t}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^-. The positive and negative deviation can travel along this path. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. This is the same for the negative deviation. After $m \leq n$ steps, a positive deviation of at east $\frac{\Delta}{2|V|^n}$ will cancel an equal amount negative deviation. Thus in n steps, the maximum total deviation is now $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. So the limit of $p_t[i]$ as the sum of the absolute values of the deviation approaches zero is $\frac{d_i}{2|E|}$.

Continued

Continued

Since at time t the total absolute deviation is Δ_t, there is a node i^+ with $\delta_t \geq \frac{\Delta_i}{2|V|}$, and a node $^-$ with $\delta_t \leq -\frac{\Delta_t}{2|V|}$. There is a path $[i_0 = i^+, i_1, \ldots, i_m, \ldots, i_{2m} = i^-]$ with an even number of edges between i^+ and i^-. The positive and negative deviation can travel along this path. At least a positive deviation equal to $\frac{1}{|V|^m}$ of the original amount will arrive at the middle node i_m. This is the same for the negative deviation. After $m \leq n$ steps, a positive deviation of at east $\frac{\Delta}{2|V|^n}$ will cancel an equal amount negative deviation. Thus in n steps, the maximum total deviation is now $\Delta_t \cdot (1 - \frac{1}{|V|^n})$. So the limit of $p_t[i]$ as the sum of the absolute values of the deviation approaches zero is $\frac{d_i}{2|E|}$.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

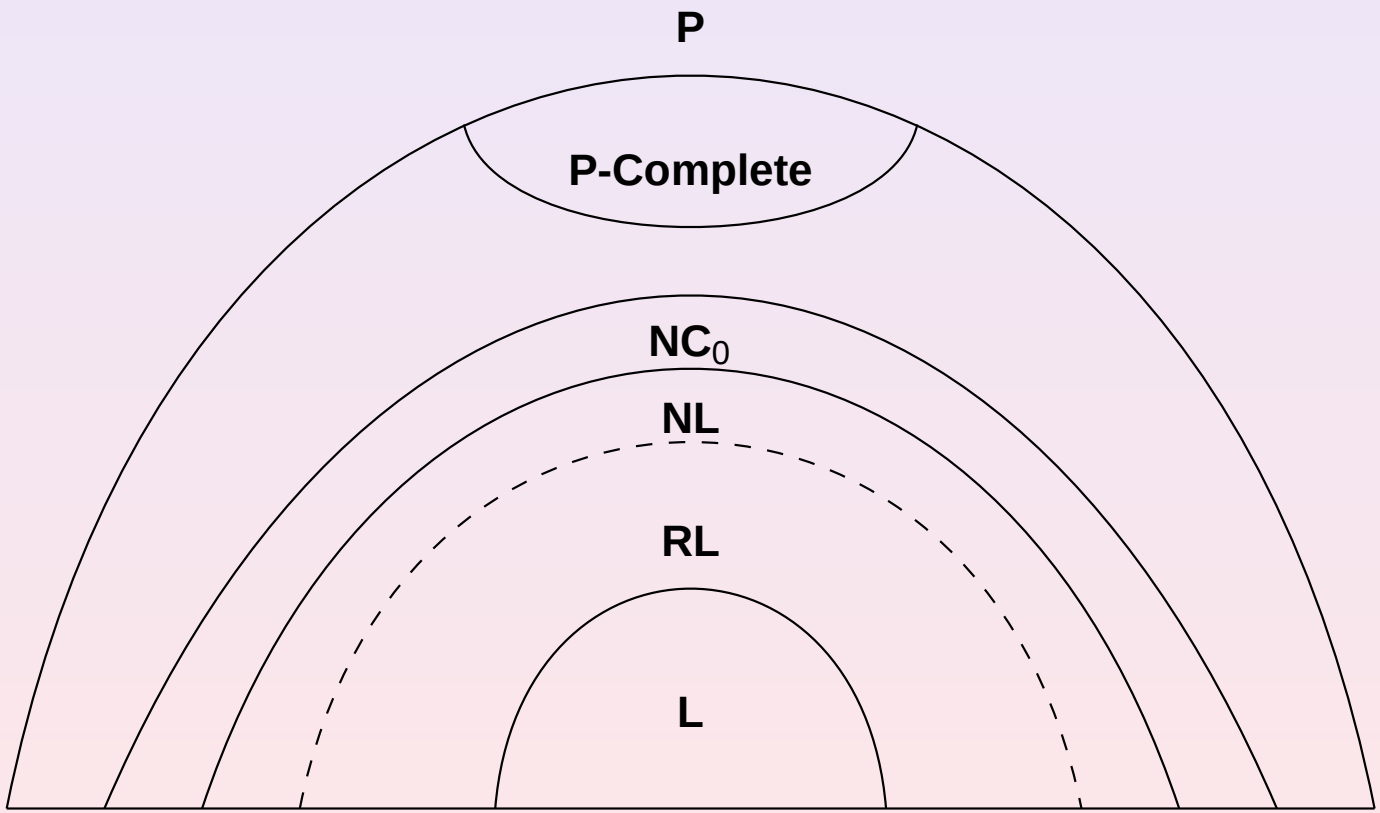
The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.

Asymptotic?

The problem is that this is an asymptotic result with exponentially slow convergence. However, stated another way, the lemma says that asymptotically and on the average, the walk returns to i every $\frac{2|E|}{d_i}$ steps. The expected return time does not change at various stages of the walk, so this lemma holds true no matter where you are in the graph. This means that from the very beginning, the expected time between two successive visits of the walk to node i is precisely $\frac{2|E|}{d_i}$.

Suppose that there is a path from s to t. We know that every $\frac{2|E|}{d_s}$ steps we will be returning to s. So, after an expected number $\frac{d_i}{2}$ (which is $|E|$ steps total) of such returns the walk will head in the right direction. We can do the same analysis with the next node on the path to t, and find that we expect to arrive at t after n such loops. Doing twice as many loops allows us to claim a 1/2 probability of return true if t is reachable.


P
P-Complete
NC0
NL
RL
L