Computability and Complexity
COSC 4200

$$\mathrm{NL} = \mathrm{coNL}$$

Definition

A *log-space transducer* is a TM with

- a read-only input tape,
- a write-only output tape, and
- a read/write work tape.

The head of the output tape cannot move leftward, so it cannot read what has been written. The work tape may contain $O(\log n)$ symbols.

Definition

A *log-space transducer* is a TM with

- a read-only input tape,
- a write-only output tape, and
- a read/write work tape.

The head of the output tape cannot move leftward, so it cannot read what has been written. The work tape may contain $O(\log n)$ symbols.

Definition

A function that is computed by a log-space transducer is called *log-space computable*.

Definition

We say that A is *log-space mapping reducible* to B, and write $A \leq_{\mathrm{L}} B$, if there is a log-space computable function $f : \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*$,

$$w \in A \Leftrightarrow f(w) \in B.$$

Definition

We say that A is *log-space mapping reducible* to B, and write $A \leq_{\mathrm{L}} B$, if there is a log-space computable function $f : \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*$,

$$w \in A \Leftrightarrow f(w) \in B.$$

Definition

A language B is NL-complete if

1. B is in NL, and
2. every A in NL is log-space reducible to B.

Theorem

If $A \leq_{\mathrm{L}} B$ *and* $B \in \mathrm{L}$*, then* $A \in \mathrm{L}$*.*

Theorem

If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

Proof. Let f be the $\leq_{\mathrm{L}}$-reduction and let M be the log-space algorithm for B.

- The basic idea is to compute $f(w)$ and then run M on $f(w)$ like before.

Theorem

If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

Proof. Let f be the $\leq_{\mathrm{L}}$-reduction and let M be the log-space algorithm for B.

- The basic idea is to compute $f(w)$ and then run M on $f(w)$ like before.
- However, the runtime of f can be polynomial and so the output may have polynomial size. Thus we don't have the space to compute $f(w)$.

Theorem

If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

Proof. Let f be the $\leq_{\mathrm{L}}$-reduction and let M be the log-space algorithm for B.

- The basic idea is to compute $f(w)$ and then run M on $f(w)$ like before.
- However, the runtime of f can be polynomial and so the output may have polynomial size. Thus we don't have the space to compute $f(w)$.
- Instead, each time M needs a bit of $f(w)$, we recompute f until that bit is output. This way we only need to store one output bit of f at a time.

Theorem

If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

Proof. Let f be the $\leq_{\mathrm{L}}$-reduction and let M be the log-space algorithm for B.

- The basic idea is to compute $f(w)$ and then run M on $f(w)$ like before.
- However, the runtime of f can be polynomial and so the output may have polynomial size. Thus we don't have the space to compute $f(w)$.
- Instead, each time M needs a bit of $f(w)$, we recompute f until that bit is output. This way we only need to store one output bit of f at a time.
- We use $O(\log n)$ space to keep track of where M's tape head is. $\square$

Theorem

If $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$.

Corollary

If any NL-complete problem is in L, then $\mathrm{L} = \mathrm{NL}$.

Recall

$\mathrm{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\}$.

We know that $\mathrm{PATH} \in \mathrm{P}$ and $\mathrm{PATH} \in \mathrm{DSPACE}(\log^2 n)$.

Theorem

PATH *is* NL*-complete.*

Theorem

PATH *is* NL*-complete.*

Theorem

PATH *is* NL*-complete.*

Proof. First, we show that $\mathrm{PATH} \in \mathrm{NL}$. On input $\langle G, s, t\rangle$ where G has n vertices, our NTM N nondeterministically follows a path of length up n.

Theorem

PATH *is* NL*-complete.*

Proof. First, we show that $\mathrm{PATH} \in \mathrm{NL}$. On input $\langle G, s, t\rangle$ where G has n vertices, our NTM N nondeterministically follows a path of length up n.

```
N:  On input ⟨G, s, t⟩:
        let n be the number of vertices in G
        let v = s
        for i = n down to 1
            if (v = t) ACCEPT
            if (v has no neighbors) REJECT
            nondeterministically choose a neighbor u of v
            let v = u
        REJECT
```

Theorem

PATH *is* NL*-complete.*

Proof. First, we show that $\mathrm{PATH} \in \mathrm{NL}$. On input $\langle G, s, t\rangle$ where G has n vertices, our NTM N nondeterministically follows a path of length up n.

```
N:  On input ⟨G, s, t⟩:
        let n be the number of vertices in G
        let v = s
        for i = n down to 1
            if (v = t) ACCEPT
            if (v has no neighbors) REJECT
            nondeterministically choose a neighbor u of v
            let v = u
        REJECT
```

- If there is a path from s to t, N will find t on some computation path and accept.

Theorem

PATH *is* NL*-complete.*

Proof. First, we show that $\mathrm{PATH} \in \mathrm{NL}$. On input $\langle G, s, t\rangle$ where G has n vertices, our NTM N nondeterministically follows a path of length up n.

```
N:  On input ⟨G, s, t⟩:
        let n be the number of vertices in G
        let v = s
        for i = n down to 1
            if (v = t) ACCEPT
            if (v has no neighbors) REJECT
            nondeterministically choose a neighbor u of v
            let v = u
        REJECT
```

- If there is a path from s to t, N will find t on some computation path and accept.
- If there is no path from s to t, N always rejects.

Theorem

PATH *is* NL*-complete.*

Proof. First, we show that $\mathrm{PATH} \in \mathrm{NL}$. On input $\langle G, s, t\rangle$ where G has n vertices, our NTM N nondeterministically follows a path of length up n.

```
N:  On input ⟨G, s, t⟩:
        let n be the number of vertices in G
        let v = s
        for i = n down to 1
            if (v = t) ACCEPT
            if (v has no neighbors) REJECT
            nondeterministically choose a neighbor u of v
            let v = u
        REJECT
```

- If there is a path from s to t, N will find t on some computation path and accept.
- If there is no path from s to t, N always rejects.
- N only needs to store v and i, which takes $O(\log n)$ space.

To see that PATH is NL-complete, let $A \in \mathrm{NL}$ and let N be an NTM that decides A in $O(\log n)$ space. If necessary, we modify N so that it has a unique accepting configuration. We will show $A \leq_{\mathrm{L}} \mathrm{PATH}$.

To see that PATH is NL-complete, let $A \in \mathrm{NL}$ and let N be an NTM that decides A in $O(\log n)$ space. If necessary, we modify N so that it has a unique accepting configuration. We will show $A \leq_{\mathrm{L}} \mathrm{PATH}$.

Given an input w, we will construct $\langle G, s, t \rangle$ such that $w \in A \iff \langle G, s, t \rangle \in \mathrm{PATH}$.

To see that PATH is NL-complete, let $A \in \mathrm{NL}$ and let N be an NTM that decides A in $O(\log n)$ space. If necessary, we modify N so that it has a unique accepting configuration. We will show $A \leq_{\mathrm{L}} \mathrm{PATH}$.

Given an input w, we will construct $\langle G, s, t \rangle$ such that $w \in A \iff \langle G, s, t \rangle \in \mathrm{PATH}$.

- The vertices of G are the configurations of N on w.

To see that PATH is NL-complete, let $A \in \mathrm{NL}$ and let N be an NTM that decides A in $O(\log n)$ space. If necessary, we modify N so that it has a unique accepting configuration. We will show $A \leq_{\mathrm{L}} \mathrm{PATH}$.

Given an input w, we will construct $\langle G, s, t\rangle$ such that $w \in A \iff \langle G, s, t\rangle \in \mathrm{PATH}$.

- The vertices of G are the configurations of N on w.
- There is an edge between two configurations if the second one follows from the first via one move of N.

To see that PATH is NL-complete, let $A \in \mathrm{NL}$ and let N be an NTM that decides A in $O(\log n)$ space. If necessary, we modify N so that it has a unique accepting configuration. We will show $A \leq_{\mathrm{L}} \mathrm{PATH}$.

Given an input w, we will construct $\langle G, s, t \rangle$ such that $w \in A \iff \langle G, s, t \rangle \in \mathrm{PATH}$.

- The vertices of G are the configurations of N on w.
- There is an edge between two configurations if the second one follows from the first via one move of N.
- s is the start configuration of N on w.

To see that PATH is NL-complete, let $A \in \mathrm{NL}$ and let N be an NTM that decides A in $O(\log n)$ space. If necessary, we modify N so that it has a unique accepting configuration. We will show $A \leq_{\mathrm{L}} \mathrm{PATH}$.

Given an input w, we will construct $\langle G, s, t \rangle$ such that $w \in A \iff \langle G, s, t \rangle \in \mathrm{PATH}$.

- The vertices of G are the configurations of N on w.
- There is an edge between two configurations if the second one follows from the first via one move of N.
- s is the start configuration of N on w.
- t is the accepting configuration of N.

Here is how a log-space transducer computes the adjacency list representation of G:

- Since N is $O(\log n)$-space bounded, each configuration may be represented by a $c \log n$-bit string for some constant c.
- We loop through all strings of size $c \log n$.
- If a string encodes a valid configuration C, we list all configurations that follow from C via one move of N's transition function. $\square$

Analogously to coNP, we have

$$\mathrm{coNL} = \{A^c \mid a \in \mathrm{NL}\}.$$

While NP versus coNP is open, the log-space analogues of these classes are equal.

Theorem (Immerman (1988) and Szelepcsényi (1988))

$\mathrm{NL} = \mathrm{coNL}$.

Theorem (Immerman (1988) and Szelepcsényi (1988))

$\mathrm{NL} = \mathrm{coNL}$.

Proof. Since PATH is NL-complete, PATH^c is coNL-complete (every problem in coNL $\leq_{\mathrm{L}}$-reduces to PATH^c).

Theorem (Immerman (1988) and Szelepcsényi (1988))

$\mathrm{NL} = \mathrm{coNL}$.

Proof. Since PATH is NL-complete, PATH^c is coNL-complete (every problem in coNL $\leq_{\mathrm{L}}$-reduces to PATH^c).

Thus it suffices to show $\mathrm{PATH}^c \in \mathrm{NL}$. This uses a technique called *inductive counting*.

Theorem (Immerman (1988) and Szelepcsényi (1988))

$\mathrm{NL} = \mathrm{coNL}$.

Proof. Since PATH is NL-complete, PATH^c is coNL-complete (every problem in coNL $\leq_{\mathrm{L}}$-reduces to PATH^c).

Thus it suffices to show $\mathrm{PATH}^c \in \mathrm{NL}$. This uses a technique called *inductive counting*.

Let $\langle G, s, t \rangle$ be an instance of PATH^c. We will design an NL algorithm that accepts $\langle G, s, t \rangle$ if and only if there is *not* a path from s to t.

Let c be the number of vertices reachable from s in G. For now, assume we know c. We will use c to help nondeterministically verify there is no path from s to t.

Let c be the number of vertices reachable from s in G. For now, assume we know c. We will use c to help nondeterministically verify there is no path from s to t.

- M goes through all m vertices in G and nondeterministically guesses which ones are reachable from s.

Let c be the number of vertices reachable from s in G. For now, assume we know c. We will use c to help nondeterministically verify there is no path from s to t.

- M goes through all m vertices in G and nondeterministically guesses which ones are reachable from s.
- For each node u that is guessed to be reachable from s, M tries to nondeterministically guess a path from s to u of length at most m.

Let c be the number of vertices reachable from s in G. For now, assume we know c. We will use c to help nondeterministically verify there is no path from s to t.

- M goes through all m vertices in G and nondeterministically guesses which ones are reachable from s.
- For each node u that is guessed to be reachable from s, M tries to nondeterministically guess a path from s to u of length at most m.
  - If a path from s to u is successfully guessed, M increments a counter.
  - If a path is not successfully guessed, M rejects.

Let c be the number of vertices reachable from s in G. For now, assume we know c. We will use c to help nondeterministically verify there is no path from s to t.

- M goes through all m vertices in G and nondeterministically guesses which ones are reachable from s.
- For each node u that is guessed to be reachable from s, M tries to nondeterministically guess a path from s to u of length at most m.
  - If a path from s to u is successfully guessed, M increments a counter.
  - If a path is not successfully guessed, M rejects.
- If M's counter equals c, then M has guessed all c vertices that are reachable from s:
  - M accepts if t is not one of the guessed vertices.

Now we show how to calculate c.

- For each i, $0 \leq i \leq m$, let A_i be the vertices that are at distance at most i from s.

Now we show how to calculate c.

- For each i, $0 \leq i \leq m$, let A_i be the vertices that are at distance at most i from s.
- Let $c_i = |A_i|$.

Now we show how to calculate c.

- For each i, $0 \leq i \leq m$, let A_i be the vertices that are at distance at most i from s.
- Let $c_i = |A_i|$.
- We have $A_0 = \{s\}$, $A_i \subseteq A_{i+1}$ for all i, and A_m is all vertices that are reachable from s.

Now we show how to calculate c.

- For each i, $0 \leq i \leq m$, let A_i be the vertices that are at distance at most i from s.
- Let $c_i = |A_i|$.
- We have $A_0 = \{s\}$, $A_i \subseteq A_{i+1}$ for all i, and A_m is all vertices that are reachable from s.
- We will show how to calculate c_{i+1} from c_i.

Now we show how to calculate c.

- For each i, $0 \leq i \leq m$, let A_i be the vertices that are at distance at most i from s.
- Let $c_i = |A_i|$.
- We have $A_0 = \{s\}$, $A_i \subseteq A_{i+1}$ for all i, and A_m is all vertices that are reachable from s.
- We will show how to calculate c_{i+1} from c_i.
- At the end, we'll have $c_m = c$.

Assume we have already computed c_i. Here is how to compute c_{i+1} from c_i:

Assume we have already computed c_i. Here is how to compute c_{i+1} from c_i:

- Let $c_{i+1} = 1$. We loop through all vertices of G, guessing which ones are in A_{i+1}:

Assume we have already computed c_i. Here is how to compute c_{i+1} from c_i:

- Let $c_{i+1} = 1$. We loop through all vertices of G, guessing which ones are in A_{i+1}:
  - In an inner loop, we go through all vertices of G, guessing which ones are in A_i. A path of length $\leq i$ is nondeterministically guessed to verify that each guessed vertex is in A_i. (Similar to $\mathrm{PATH} \in \mathrm{NL}$.)

Assume we have already computed c_i. Here is how to compute c_{i+1} from c_i:

- Let $c_{i+1} = 1$. We loop through all vertices of G, guessing which ones are in A_{i+1}:
  - In an inner loop, we go through all vertices of G, guessing which ones are in A_i. A path of length $\leq i$ is nondeterministically guessed to verify that each guessed vertex is in A_i. (Similar to $\mathrm{PATH} \in \mathrm{NL}$.)
  - We use a counter to keep track of how many vertices are verified to be in A_i.

Assume we have already computed c_i. Here is how to compute c_{i+1} from c_i:

- Let $c_{i+1} = 1$. We loop through all vertices of G, guessing which ones are in A_{i+1}:
  - In an inner loop, we go through all vertices of G, guessing which ones are in A_i. A path of length $\leq i$ is nondeterministically guessed to verify that each guessed vertex is in A_i. (Similar to $\mathrm{PATH} \in \mathrm{NL}$.)
  - We use a counter to keep track of how many vertices are verified to be in A_i.
  - For each vertex verified to be in A_i, M tests whether (u, v) is an edge. If it is an edge, then $v \in A_{i+1}$ and we increment c_{i+1}.

Once we have computed c_m:

- We loop through all vertices of G, guessing which ones are in A_m and guessing a path for each starting from s.
- When all c_m vertices and paths have been successfully guessed, the algorithm accepts if t is not one of these vertices.

M: On input $\langle G, s, t\rangle$:

let $c_0 = 1$ // *$A_0 = \{s\}$ has one vertex*

M: On input $\langle G, s, t \rangle$:
let $c_0 = 1$ // *$A_0 = \{s\}$ has one vertex*
for $i = 0$ to $m - 1$ // *compute c_{i+1} from c_i*
let $c_{i+1} = 1$

M: On input $\langle G, s, t\rangle$:

let $c_0 = 1$ // *$A_0 = \{s\}$ has one vertex*

for $i = 0$ to $m - 1$ // *compute c_{i+1} from c_i*

let $c_{i+1} = 1$

for each vertex $v \neq s$ in G:

let $d = 0$ // *c_i is now known; d is used to recount A_i*

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                              // A_0 = {s} has one vertex
    for i = 0 to m − 1                       // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                    // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                  // A_0 = {s} has one vertex
    for i = 0 to m − 1                           // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                        // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                  // A_0 = {s} has one vertex
    for i = 0 to m − 1                           // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                        // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                // verified that u ∈ A_i
```

M: On input $\langle G, s, t\rangle$:

```
let c_0 = 1                              // A_0 = {s} has one vertex
for i = 0 to m − 1                       // compute c_{i+1} from c_i
    let c_{i+1} = 1
        for each vertex v ≠ s in G:
            let d = 0                    // c_i is now known; d is used to recount A_i
            for each vertex u in G:
                nondeterministically either perform or skip these steps:
                    nondeterministically follow a path of length at most i from s
                        if the path does not end at u, REJECT
                        increment d                 // verified that u ∈ A_i
                        if (u, v) is an edge in G
                            increment c_{i+1}       // verified that v ∈ A_{i+1}
```

M: On input $\langle G, s, t\rangle$:

```
let c_0 = 1                                   // A_0 = {s} has one vertex
for i = 0 to m − 1                            // compute c_{i+1} from c_i
    let c_{i+1} = 1
        for each vertex v ≠ s in G:
            let d = 0                         // c_i is now known; d is used to recount A_i
            for each vertex u in G:
                nondeterministically either perform or skip these steps:
                    nondeterministically follow a path of length at most i from s
                        if the path does not end at u, REJECT
                        increment d                   // verified that u ∈ A_i
                        if (u, v) is an edge in G
                            increment c_{i+1}         // verified that v ∈ A_{i+1}
            if d ≠ c_i, REJECT                        // check whether found all of A_i
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                   // A_0 = {s} has one vertex
    for i = 0 to m − 1                            // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                         // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                   // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}         // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                        // check whether found all of A_i

    let d = 0                                     // c_m is now known; d is used to recount A_m
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                  // A_0 = {s} has one vertex
    for i = 0 to m − 1                           // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                        // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                  // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}        // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                       // check whether found all of A_i

    let d = 0                                    // c_m is now known; d is used to recount A_m
    for each vertex u in G:
        nondeterministically either perform or skip these steps:
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                            // A_0 = {s} has one vertex
    for i = 0 to m − 1                     // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                  // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                 // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}       // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                      // check whether found all of A_i

    let d = 0                              // c_m is now known; d is used to recount A_m
    for each vertex u in G:
        nondeterministically either perform or skip these steps:
            nondeterministically follow a path of length at most m from s
                if the path does not end at u, REJECT
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                  // A_0 = {s} has one vertex
    for i = 0 to m − 1                           // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                        // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                   // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}         // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                        // check whether found all of A_i

    let d = 0                                    // c_m is now known; d is used to recount A_m
    for each vertex u in G:
        nondeterministically either perform or skip these steps:
            nondeterministically follow a path of length at most m from s
                if the path does not end at u, REJECT
            if u = t, REJECT                     // found a path from s to t
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                  // A_0 = {s} has one vertex
    for i = 0 to m − 1                           // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                        // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d             // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}   // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                  // check whether found all of A_i

    let d = 0                                    // c_m is now known; d is used to recount A_m
    for each vertex u in G:
        nondeterministically either perform or skip these steps:
            nondeterministically follow a path of length at most m from s
                if the path does not end at u, REJECT
            if u = t, REJECT                     // found a path from s to t
            increment d                          // verified u ∈ A_m
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                   // A_0 = {s} has one vertex
    for i = 0 to m − 1                            // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                         // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                  // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}        // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                       // check whether found all of A_i

    let d = 0                                     // c_m is now known; d is used to recount A_m
    for each vertex u in G:
        nondeterministically either perform or skip these steps:
            nondeterministically follow a path of length at most m from s
                if the path does not end at u, REJECT
            if u = t, REJECT                      // found a path from s to t
            increment d                           // verified u ∈ A_m
    if d ≠ c_m, REJECT           // check whether found all of A_m
```

```
M: On input ⟨G, s, t⟩:
    let c_0 = 1                                   // A_0 = {s} has one vertex
    for i = 0 to m − 1                            // compute c_{i+1} from c_i
        let c_{i+1} = 1
            for each vertex v ≠ s in G:
                let d = 0                         // c_i is now known; d is used to recount A_i
                for each vertex u in G:
                    nondeterministically either perform or skip these steps:
                        nondeterministically follow a path of length at most i from s
                            if the path does not end at u, REJECT
                            increment d                  // verified that u ∈ A_i
                            if (u, v) is an edge in G
                                increment c_{i+1}        // verified that v ∈ A_{i+1}
                if d ≠ c_i, REJECT                       // check whether found all of A_i

    let d = 0                                     // c_m is now known; d is used to recount A_m
    for each vertex u in G:
        nondeterministically either perform or skip these steps:
            nondeterministically follow a path of length at most m from s
                if the path does not end at u, REJECT
            if u = t, REJECT                      // found a path from s to t
            increment d                           // verified u ∈ A_m
    if d ≠ c_m, REJECT          // check whether found all of A_m
    otherwise, ACCEPT           // we have verified that t ∉ A_m
```

Correctness:

- Inductively, there is a computation path where M successively computes $c_0, c_1, \ldots, c_m$.
    - In each pass, M correctly guesses which vertices are in A_i and guesses a path of length $\leq i$ to each guessed vertex.
    - Many computation paths fail and REJECT.

Correctness:

- Inductively, there is a computation path where M successively computes $c_0, c_1, \ldots, c_m$.
    - In each pass, M correctly guesses which vertices are in A_i and guesses a path of length $\leq i$ to each guessed vertex.
    - Many computation paths fail and REJECT.
- On this computation path:
    - M either finds a path from s to t and REJECTS, or
    - determines that $t \notin A_m$ and ACCEPTS.

Efficiency: the algorithm only needs to store

- m (number of vertices),
- u (loop vertex),
- v (loop vertex),
- c_i (count of A_i),
- c_{i+1} (count of A_{i+1}),
- d (recount variable),
- a counter for how many vertices guessed on a path, and
- a pointer to the head of a guessed path.

These all take $O(\log n)$ space.

Efficiency: the algorithm only needs to store

- m (number of vertices),
- u (loop vertex),
- v (loop vertex),
- c_i (count of A_i),
- c_{i+1} (count of A_{i+1}),
- d (recount variable),
- a counter for how many vertices guessed on a path, and
- a pointer to the head of a guessed path.

These all take $O(\log n)$ space.

Therefore this is an NL algorithm for PATH^c. $\square$

Theorem (Immerman (1988) and Szelepcsényi (1988))

$\mathrm{NL} = \mathrm{coNL}$.

The proof extends for other space bounds.

Corollary

For any space-constructible bound $s(n) \geq \log n$,

$$\mathrm{NSPACE}(s(n)) = \mathrm{coNSPACE}(s(n)).$$

Open Problem

Open Problem:

- Does $\mathrm{L} = \mathrm{NL}$?
- Equivalently, is $\mathrm{PATH} \in \mathrm{L}$?

Open Problem

Open Problem:

- Does $\mathrm{L} = \mathrm{NL}$?
- Equivalently, is $\mathrm{PATH} \in \mathrm{L}$?

Notable results:

- Savitch's algorithm (1970) tells us $\mathrm{PATH} \in \mathrm{DSPACE}(\log^2 n)$.

Open Problem

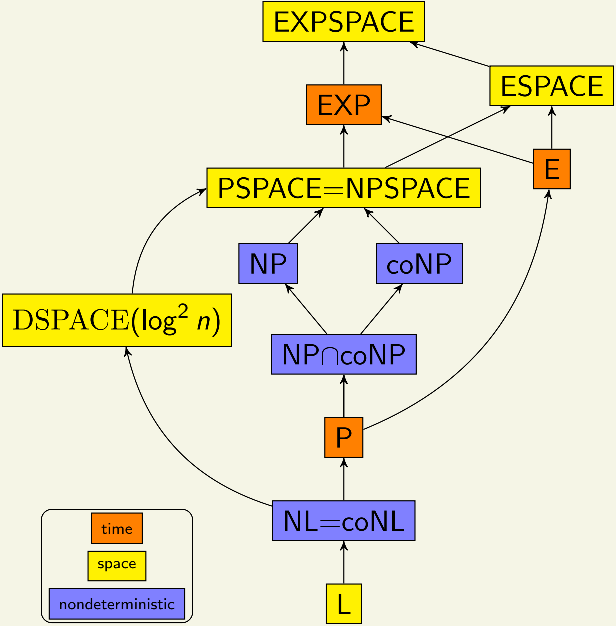
Open Problem:

- Does $\mathrm{L} = \mathrm{NL}$?
- Equivalently, is $\mathrm{PATH} \in \mathrm{L}$?

Notable results:

- Savitch's algorithm (1970) tells us $\mathrm{PATH} \in \mathrm{DSPACE}(\log^2 n)$.
- Reingold (2004) proved that the undirected graph path problem UPATH is in L.

Summary



Open problems:

- P = NP?
- P = PSPACE?
- NP = PSPACE?
- PSPACE = EXP?
- NP = EXP?
- NP $\subseteq$ E? E $\subseteq$ NP?
- PSPACE $\subseteq$ E?
  E $\subseteq$ PSPACE?
- L = NL?
- NL = P?
- NL = NP?
- L = NP?
- NP = coNP?
- P = NP $\cap$ coNP?

Known:

- P $\neq$ E $\neq$ EXP
- L $\neq$ DSPACE($\log^2$ n)
  $\neq$ PSPACE
  $\neq$ ESPACE
  $\neq$ EXPSPACE