Computational Complexity Theory

Lecture 7: Relativization (contd.); Space complexity

Indian Institute of Science

Recap: Limits of diagonalization

- Like in the proof of $P \neq EXP$, can we use diagonalization to show $P \neq NP$?
- The answer is No, if one insists on <u>using only the two features of diagonalization</u>.

Recap: Oracle Turing Machines

- Like in the proof of $P \neq EXP$, can we use diagonalization to show $P \neq NP$?
- The answer is No, if one insists on using only the two features of diagonalization.

- Definition: Let $L \subseteq \{0,1\}^*$ be a language. An *oracle TM* M^L is a TM with a special query tape and three special states q_{query}, q_{yes} and q_{no} such that whenever the machine enters the q_{query} state, it immediately transits to q_{yes} or q_{no} depending on whether the string in the query tape belongs to L.
  (M^L has *oracle access* to L)

Recap: Oracle Turing Machines

- Like in the proof of $P \neq EXP$, can we use diagonalization to show $P \neq NP$?
- The answer is No, if one insists on using only the two features of diagonalization.

- Important note: Oracle TMs (deterministic/nondeterministic) have the same two features used in diagonalization: For any fixed $L \subseteq \{0,1\}^*$,

1. There's an efficient universal TM with oracle access to L,
2. Every M^L has infinitely many representations.

Recap: Relativizing results

- Like in the proof of $P \neq EXP$, can we use diagonalization to show $P \neq NP$?
- The answer is No, if one insists on <u>using only the two features of diagonalization</u>.

- Observation: Let $L \subseteq \{0,1\}^*$ be an arbitrarily fixed language. Owing to the 'Important note', the proof of $P \neq EXP$ can be easily adapted to prove $P^L \neq EXP^L$ by working with TMs with oracle access to L.
- We say that the $P \neq EXP$ result <u>*relativizes*</u>.

Recap: Relativizing results

- Like in the proof of $P \neq EXP$, can we use diagonalization to show $P \neq NP$?
- The answer is No, if one insists on <u>using only the two features of diagonalization</u>.

- Is is true that

- either $P^L = NP^L$ for every $L \subseteq \{0,1\}^*$,

- or $P^L \neq NP^L$ for every $L \subseteq \{0,1\}^*$?

Theorem (Baker-Gill-Solovay): The answer is No. Any proof of $P = NP$ or $P \neq NP$ must <u>not</u> relativize.

Recap: Baker-Gill-Solovay theorem

- Theorem: There exist languages A and B such that $P^A = NP^A$ but $P^B \neq NP^B$.
- Proof: Let $A = \{(M, x, 1^m)$: M accepts x in 2^m steps$\}$.
- A is an EXP-complete language under poly-time Karp reduction.
- Then, $P^A = EXP$.
- Also, $NP^A = EXP$. Hence $P^A = NP^A$.

Recap: Baker-Gill-Solovay theorem

- Theorem: There exist languages A and B such that $P^A = NP^A$ but $P^B \neq NP^B$.
- Proof: For any language B let

$L_B = \{1^n$: there's a string of length in $B\}$.

- Observe, $L_B \in NP^B$ for any B. (Guess the string, check if it has length n, and ask oracle B to verify membership.)

Recap: Baker-Gill-Solovay theorem

- Theorem: There exist languages A and B such that $P^A = NP^A$ but $P^B \neq NP^B$.
- Proof: For any language B let

  $L_B = \{1^n :$ there's a string of length in B$\}$.

- Observe, $L_B \in NP^B$ for any B.
- We'll construct B (using diagonalization) in such a way that $L_B \notin P^B$, implying $P^B \neq NP^B$.

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps. Moreover, n will grow monotonically with stages.

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps. Moreover, n will grow monotonically with stages.

whether or not a string belongs to B

The machine with oracle access to B that is represented by i

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps. Moreover, n will grow monotonically with stages.

- Clearly, a B satisfying the above implies $L_B \notin P^B$. Why?

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps. Moreover, n will <u>grow monotonically</u> with stages.

- Stage i: Choose n larger than the length of any string whose status has already been decided. Simulate M_i^B on input 1^n for $2^n/10$ steps.

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps.

- Stage i: If M_i^B queries oracle B with a string whose status has already been decided, answer consistently.

  If M_i^B queries oracle B with a string whose status has not been decided yet, answer 'No'.

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps.

- Stage i: If M_i^B outputs 1 within $2^n/10$ steps then don't put any string of length n in B.

  If M_i^B outputs 0 or doesn't halt, put a string of length n in B.

(This is possible as the status of at most $2^n/10$ many length n strings have been decided during the simulation)

Constructing B

- We'll construct B in stages, starting from Stage 1.
- Each stage determines the status of finitely many strings.
- In Stage i, we'll ensure that the oracle TM M_i^B doesn't decide 1^n correctly (for some n) within $2^n/10$ steps.

- Homework: In fact, we can assume that $B \in \mathrm{EXP}$.

Space Complexity

Space bounded computation

- Here, we are interested to find out how much of *work space* is required to solve a problem.
- For convenience, think of TMs with a separate input tape and one or more work tapes. Work space is the number of cells in the work tapes of a TM M visited by M's heads during a computation.

Space bounded computation

- Here, we are interested to find out how much of *work space* is required to solve a problem.
- For convenience, think of TMs with a separate input tape and one or more work tapes. Work space is the number of cells in the work tapes of a TM M visited by M's heads during a computation.

- Definition. Let $S: \mathbb{N} \rightarrow \mathbb{N}$ be a function. A language L is in DSPACE(S(n)) if there's a TM M that decides L using O(S(n)) work space on inputs of length n.

Space bounded computation

- Here, we are interested to find out how much of *work space* is required to solve a problem.
- For convenience, think of TMs with a separate input tape and one or more work tapes. Work space is the number of cells in the work tapes of a TM M visited by M's heads during a computation.

- Definition. Let S: N → N be a function. A language L is in NSPACE(S(n)) if there's a NTM M that decides L using O(S(n)) work space on inputs of length n, regardless of M's nondeterministic choices.

Space bounded computation

- Here, we are interested to find out how much of *work space* is required to solve a problem.
- For convenience, think of TMs with a separate input tape and one or more work tapes. Work space is the number of cells in the work tapes of a TM M visited by M's heads during a computation.
- We'll simply refer to 'work space' as 'space'. For convenience, assume there's a single work tape.

Space bounded computation

- Here, we are interested to find out how much of <u>*work space*</u> is required to solve a problem.
- For convenience, think of TMs with a separate input tape and one or more work tapes. Work space is the number of cells in the work tapes of a TM M visited by M's heads during a computation.

- Definition. Let $S: \mathbb{N} \longrightarrow \mathbb{N}$ be a function. S is *space constructible* if there's a TM that computes $S(|x|)$ from x using $O(S(|x|))$ space.

Relation between time and space

- Obs. DTIME(S(n)) $\subseteq$ DSPACE(S(n)) $\subseteq$ NSPACE(S(n)).

- Theorem. NSPACE(S(n)) $\subseteq$ DTIME($2^{O(S(n))}$), if S is space constructible.
- Proof. Uses the notion of *configuration graph* of a TM. We'll see this shortly.

Relation between time and space

- Obs. DTIME(S(n)) ⊆ DSPACE(S(n)) ⊆ NSPACE(S(n)).
- Theorem. NSPACE(S(n)) ⊆ DTIME($2^{O(S(n))}$), if S is space constructible.
- Definition.

$$L = \mathrm{DSPACE}(\log n)$$

$$NL = \mathrm{NSPACE}(\log n)$$

$$\mathrm{PSPACE} = \bigcup_{c>0} \mathrm{DSPACE}(n^c)$$

Relation between time and space

- Obs. DTIME(S(n)) ⊆ DSPACE(S(n)) ⊆ NSPACE(S(n)).
- Theorem. NSPACE(S(n)) ⊆ DTIME($2^{O(S(n))}$), if S is space constructible.
- Definition.

$$L = \mathrm{DSPACE}(\log n)$$

$$NL = \mathrm{NSPACE}(\log n)$$

$$\mathrm{PSPACE} = \bigcup_{c>0} \mathrm{DSPACE}(n^c)$$

Giving space at least log n gives a TM at least the power to remember the index of a cell.

Relation between time and space

- Obs. $DTIME(S(n)) \subseteq DSPACE(S(n)) \subseteq NSPACE(S(n))$.

- Theorem. $NSPACE(S(n)) \subseteq DTIME(2^{O(S(n))})$, if S is space constructible.

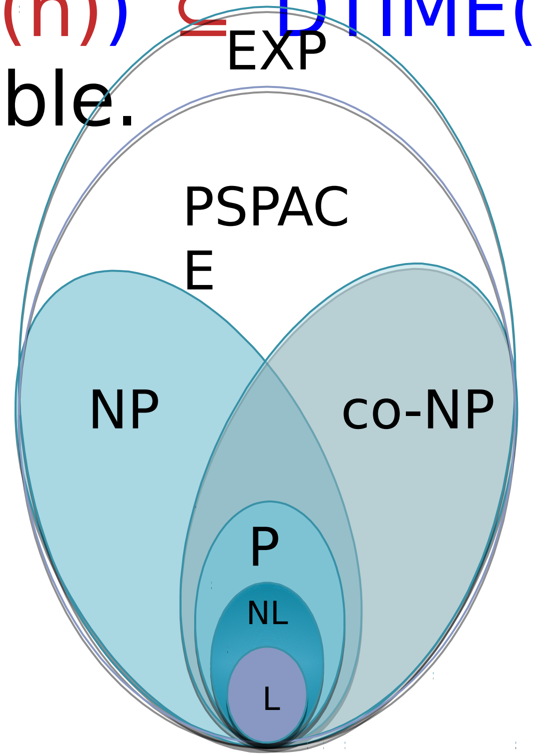
- Theorem. $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$

Run through all certificate choices of the verifier and **reuse** space.

Relation between time and space

- Obs. DTIME(S(n)) ⊆ DSPACE(S(n)) ⊆ NSPACE(S(n)).

- Theorem. NSPACE(S(n)) ⊆ DTIME($2^{O(S(n))}$), if S is space constructible.



Configuration graph

- Definition. A *configuration* of a TM M on input x, at any particular step of its execution, consists of

(a) the nonblank symbols of its work tapes,

(b) the current state,

(c) the current head positions.

It captures a ‘snapshot’ of M at any particular moment of execution.

Configuration graph

- Definition. A *configuration* of a TM M on input x, at any particular step of its execution, consists of

(a) the nonblank symbols of its work tapes,

(b) the current state,

(c) the current head positions.

It captures a 'snapshot' of M at any particular moment of execution.

State info	Input head index	Work tape head index	b_1	…	$b_{S(n)}$

Content of work tape

A Configuration C

Configuration graph

- Definition. A *configuration* of a TM M on input x, at any particular step of its execution, consists of

  (a) the nonblank symbols of its work tapes,

  (b) the current state,

  (c) the current head positions.

It captures a 'snapshot' of M at any particular moment of execution.

State info	Input head index	Work tape head index	b_1	…	$b_{S(n)}$

Note: A configuration C can be represented using $O(S(n))$ bits if M uses $S(n) \geq \log n$ space on n-bit inputs.

Configuration graph

- Definition. A *configuration graph* of a TM M on input x, denoted $G_{M,x}$, is a directed graph whose nodes are all the possible configurations of M on input x. There's an edge from one configuration C_1 to another C_2, if C_2 can be reached from C_1 by an application of M's transition function(s).

Configuration graph

- Definition. A *configuration graph* of a TM M on input x, denoted $G_{M,x}$, is a directed graph whose nodes are all the possible configurations of M on input x. There's an edge from one configuration C_1 to another C_2, if C_2 can be reached from C_1 by an application of M's transition function(s).

- Number of nodes in $G_{M,x} = 2^{O(S(n))}$, if M uses $S(n)$ space on n-bit inputs

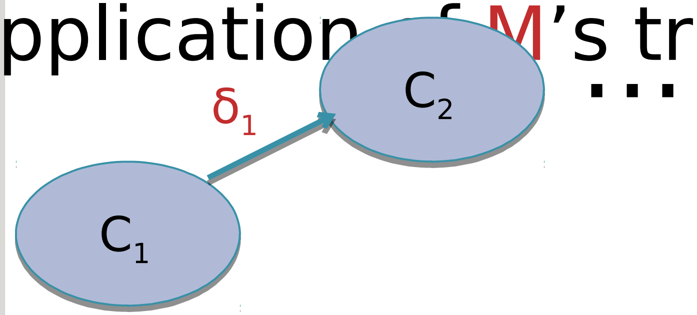
Configuration graph

- Definition. A *configuration graph* of a TM M on input x, denoted $G_{M,x}$, is a directed graph whose nodes are all the possible configurations of M on input x. There's an edge from one configuration C_1 to another C_2, if C_2 can be reached from C_1 by an application of M's transition function(s).

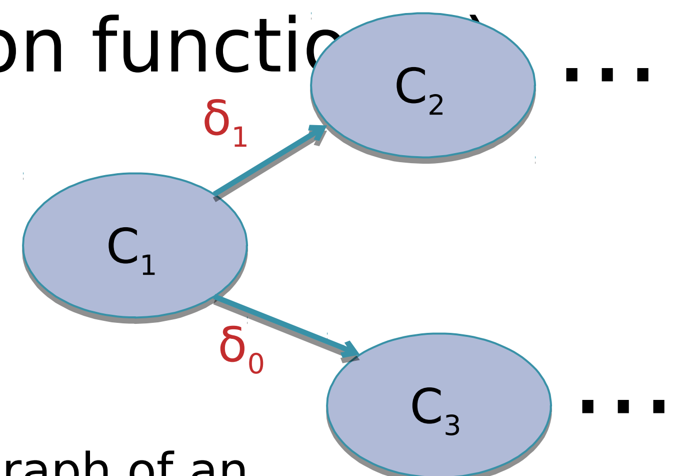
- If M is a DTM then every node C in $G_{M,x}$ has at most one outgoing edge. If M is an NTM then every node C in $G_{M,x}$ has at most <u>two</u> outgoing edges.

Configuration graph

- Definition. A *configuration graph* of a TM M on input x, denoted $G_{M,x}$, is a directed graph whose nodes are all the possible configurations of M on input x. There's an edge from one configuration C_1 to another C_2, if C_2 can be reached from C_1 by an application of M's transition function.



Conf. graph of a DTM



Conf. graph of an NTM

Configuration graph

- Definition. A *configuration graph* of a TM M on input x, denoted $G_{M,x}$, is a directed graph whose nodes are all the possible configurations of M on input x. There's an edge from one configuration C_1 to another C_2, if C_2 can be reached from C_1 by an application of M's transition function(s).

- By erasing the contents of the work tape at the end, bringing the head at the beginning, and having a q_{accept} state, we can assume that there's a unique C_{accept} configuration. Configuration C_{start} is well defined.

Configuration graph

- Definition. A *configuration graph* of a TM M on input x, denoted $G_{M,x}$, is a directed graph whose nodes are all the possible configurations of M on input x. There's an edge from one configuration C_1 to another C_2, if C_2 can be reached from C_1 by an application of M's transition function(s).

- M accepts x if and only if there's a path from C_{start} to C_{accept} in $G_{M,x}$.

Relation between time and space

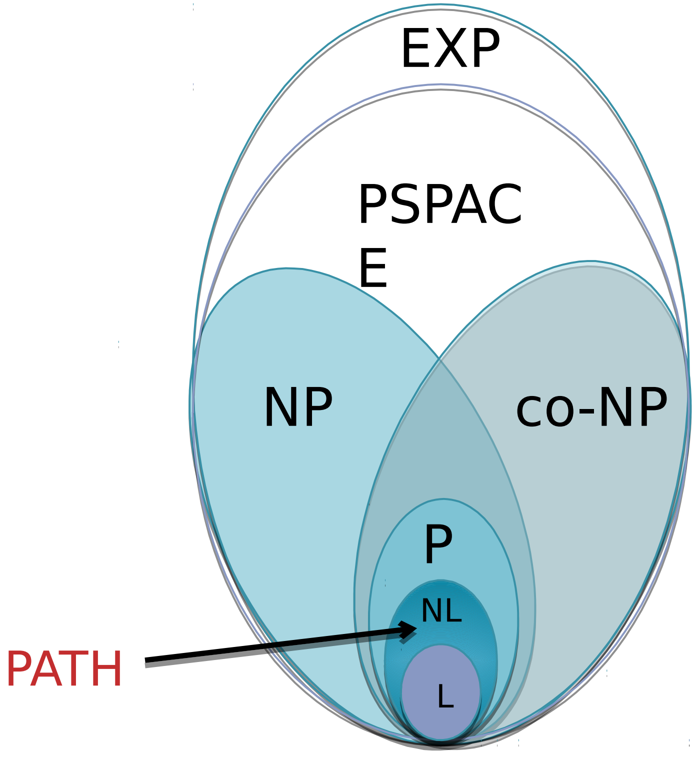
- Obs. $\mathrm{DTIME}(S(n)) \subseteq \mathrm{DSPACE}(S(n)) \subseteq \mathrm{NSPACE}(S(n))$.

- Theorem. $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DTIME}(2^{O(S(n))})$, if S is space constructible.

- Proof. Let $L \in \mathrm{NSPACE}(S(n))$ and M be an NTM deciding L using $S(n)$ space on length n inputs.
- On input x, compute the configuration graph $G_{M,x}$ of M and check if there's a **path** from C_{start} to C_{accept} . Running time is $2^{O(S(n))}$.

PATH: A canonical problem in NL

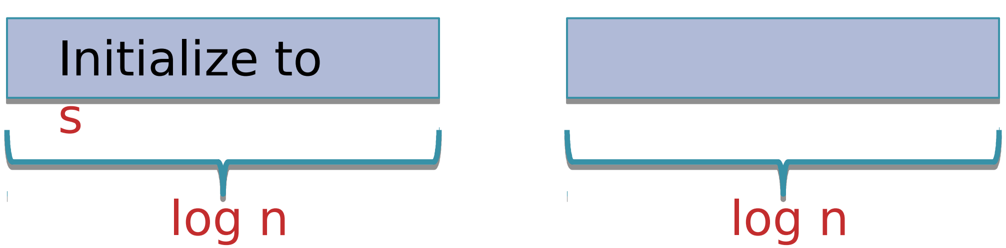
- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.



PATH: A canonical problem in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.

- Proof. Count the no. of vertices in G, let it be n. Set aside two memory locations of log n bits each. Initialize a counter, say Count = n.



Count = n

PATH: A canonical problem in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.

- Proof. Count the no. of vertices in G, let it be n. Set aside two memory locations of log n bits each. Initialize a counter, say Count = n.

Initialize to s

Guess a vertex v_1

Count = n

If there's a edge from s to v_1,
decrease count by 1.
Else o/p 0 and stop

PATH: A canonical problem in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.

- Proof. Count the no. of vertices in G, let it be n. Set aside two memory locations of log n bits each. Initialize a counter, say Count = n.

Set to v_1

Guess a vertex v_2

Count = n-1

If there's a edge from v_1 to v_2, decrease count by 1. Else o/p 0 and stop

PATH: A canonical problem in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.

- Proof. Count the no. of vertices in G, let it be n. Set aside two memory locations of log n bits each. Initialize a counter, say Count = n.

Set to v_2

Guess a vertex v_3

If there's a edge from v_2 to v_3, decrease count by 1. Else o/p 0 and stop

Count = n-2

…and so on.

PATH: A canonical problem in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.
- Proof. Count the no. of vertices in G, let it be n. Set aside two memory locations of log n bits each. Initialize a counter, say Count = n.

Set to v_{n-1}

Set to t

Count = 1

If there's a edge from v_{n-1} to t, o/p 1 and stop. Else o/p 0 and stop

PATH: A canonical problem in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.

- Proof. Count the no. of vertices in G, let it be n. Set aside two memory locations of log n bits each. Initialize a counter, say Count = n.

Set to v_{n-1}

Set to t

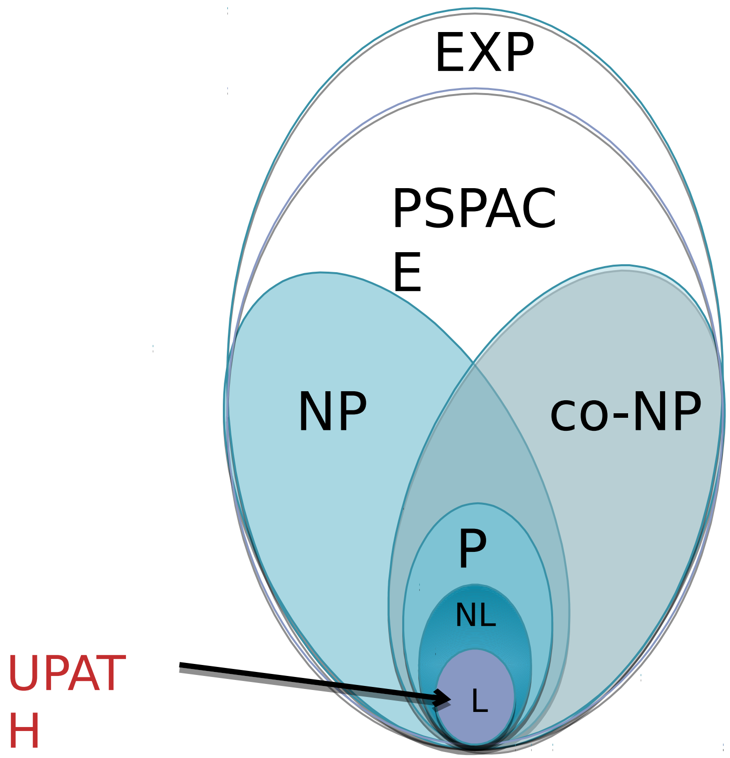
Count = 1

If there's a edge from v_{n-1} to t, o/p 1 and stop. Else o/p 0 and stop.

Space complexity = O(log n)

UPATH: A problem in L

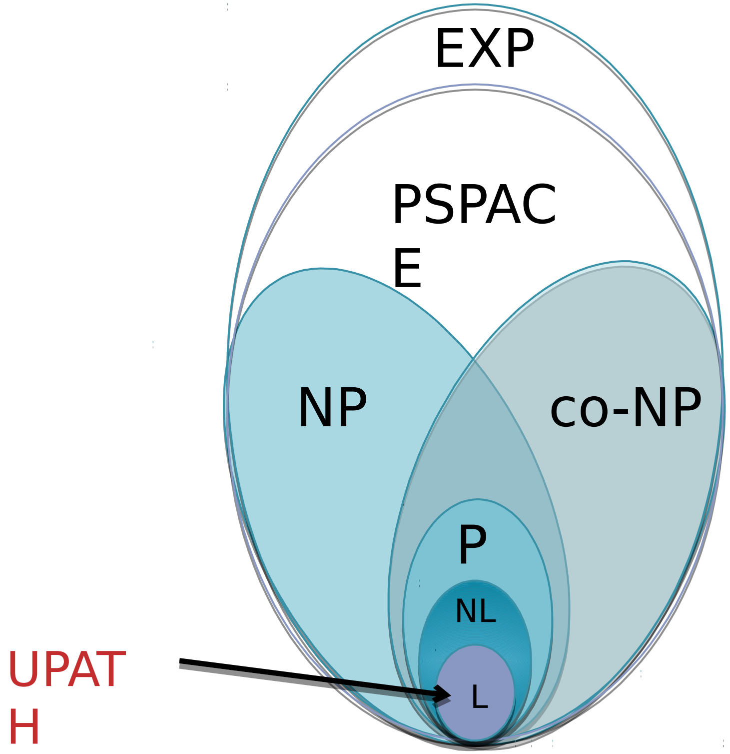
- UPATH = {(G,s,t) : G is an undirected graph having a path from s to t}.
- Theorem *(Reingold)*. UPATH is in L.



UPATH: A problem in L

- UPATH = {(G,s,t) : G is an undirected graph having a path from s to t}.
- Theorem *(Reingold)*. UPATH is in L.

Is PATH in L ?
…more on this later.



PSPACE = NPSPACE

Savitch's theorem

- Theorem. $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DSPACE}(S(n)^2)$, where $S(n)$ is space constructible. (So, PSPACE = NPSPACE)

- Proof. Let $L \in \mathrm{NSPACE}(S(n))$, and M be an NTM requiring $S(n)$ space to decide L. We'll now show that there's a TM N requiring $O(S(n)^2)$ space to decide L.

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof. Let L $\in$ NSPACE(S(n)), and M be an NTM requiring S(n) space to decide L. We'll now show that there's a TM N requiring O(S(n)^2) space to decide L.

- On input x, N checks if there's a path from C_{start} to C_{accept} in $G_{M,x}$ as follows: Let $|x| = n$.

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof. …N computes 2.S(n) + c, the no. of bits required to represent a configuration of M. It also finds out C_{start} and C_{accept}. Then N checks if there's a path from C_{start} to C_{accept} of length at most $2^{2.S(n)+c}$ in $G_{M,x}$ recursively using the following procedure.

- REACH(C_1, C_2, i) : returns 1 if there's a path from C_1 to C_2 of length at most 2^i in $G_{M,x}$; 0 otherwise.

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

Space constructibility of S(n) used here

- Proof. …N computes 2.S(n) + c, the no. of bits required to represent a configuration of M. It also finds out C_{start} and C_{accept}. Then N checks if there's a path from C_{start} to C_{accept} of length at most $2^{2.S(n)+c}$ in $G_{M,x}$ recursively using the following procedure.

- REACH(C_1, C_2, i) : returns 1 if there's a path from C_1 to C_2 of length at most 2^i in $G_{M,x}$; 0 otherwise.

Savitch's theorem

- Theorem. $NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$, where $S(n)$ is space constructible. (So, PSPACE = NPSPACE)

- Proof.
- REACH(C_1, C_2, i) {

  If $i = 0$ check if C_1 and C_2 are adjacent.

  Else, <u>for every</u> configurations C,

  a_1 = REACH(C_1, C, i-1)

  a_2 = REACH(C, C_2, i-1)

  if $a_1=1$ & $a_2=1$, return 1. Else return 0.

  }

Savitch's theorem

- Theorem. NSPACE(S(n)) ⊆ DSPACE(S(n)²), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.
- REACH(C_1, C_2, i) {

  If i = 0 check if C_1 and C_2 are adjacent.

  Else, for every configurations C,

  a_1 = REACH(C_1, C, i-1)

  a_2 = REACH(C, C_2, i-1)

  if a_1=1 & a_2=1, return 1. Else return 0.

  }

Require O(S(n)) space

Savitch's theorem

- Theorem. $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DSPACE}(S(n)^2)$, where $S(n)$ is space constructible. (So, PSPACE = NPSPACE)

- Proof.
- $\mathrm{REACH}(C_1, C_2, i)$ {

    If $i = 0$ check if C_1 and C_2 are adjacent.

    Else, <u>for every</u> configurations C,

    $a_1 = \mathrm{REACH}(C_1, C, i\text{-}1)$

    $a_2 = \mathrm{REACH}(C, C_2, i\text{-}1)$ Reuse space

    if $a_1=1$ & $a_2=1$, return 1. Else return 0.

  }

Savitch's theorem

- Theorem. NSPACE(S(n)) ⊆ DSPACE($S(n)^2$), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.

$$Space(i) = Space(i-1) + O(S(n))$$

- Space complexity: $O(S(n)^2)$

Savitch's theorem

- Theorem. $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DSPACE}(S(n)^2)$, where $S(n)$ is space constructible. (So, PSPACE = NPSPACE)

- Proof.

$$\mathrm{Space}(i) = \mathrm{Space}(i-1) + O(S(n))$$

- Space complexity: $O(S(n)^2)$

$$\mathrm{Time}(i) = 2^{2.S(n)+c}.2.\mathrm{Time}(i-1) + O(S(n))$$

- Time complexity: $2^{O(S(n)^2)}$

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.

$$\text{Space}(i) = \text{Space}(i\text{-}1) + O(S(n))$$

- Space complexity: $O(S(n)^2)$

$$\text{Time}(i) = 2^{2.S(n)+c}.2.\text{Time}(i\text{-}1) + O(S(n))$$

- Time complexity: $2^{O(S(n)\ )}$

Recall, from the previous theorem, there's an algorithm with time complexity $2^{O(S(n))}$, but higher space requirement.