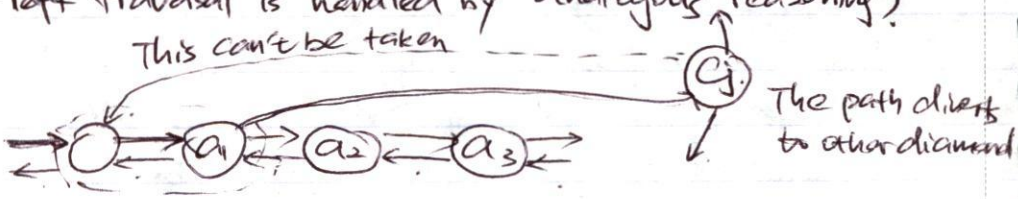
That is, the visit to c_j must be one of the detours defined for G. Suppose, this is not the case. Then one of the following two cases must hold.

Case 1

Taking the left-to-right traversal of a horizontal row only (the right-to-left traversal is handled by analogous reasoning)



c_j.

This path cannot visit a_2 from a_1 because a_1 is already visited.

a_0 — a_3 — — — It cannot enter a_1 or a_3 already visited.

- So, this path cannot visit a_2, contradiction to the assumption that this path is a Ham Path

Case 2

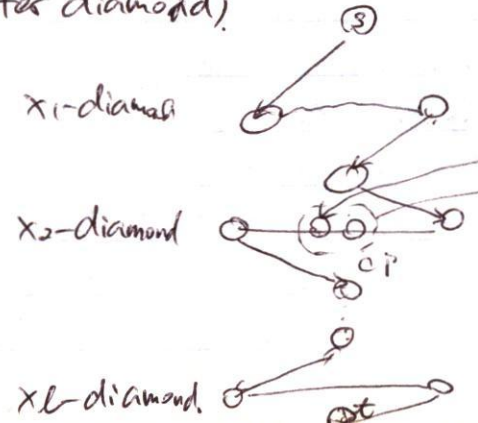
This path cannot visit a_2 from a_1, already visited.

a_2 — a_3 ∴ It cannot enter a_1 or as e is not taken as the path diverts to other diamond.

- So, a_2 cannot be visited, a contradiction to the assumption that this path is a HAM path.

11/10

(⇐) Suppose G has a hampath P from s to t. By the claim from the previous class, P must traverse the var diamonds in order of $x_1, \dots, x_\ell$ and the detour to each clause node c_j must be one of the defined detours, no detour can drive to a previous or later diamond.

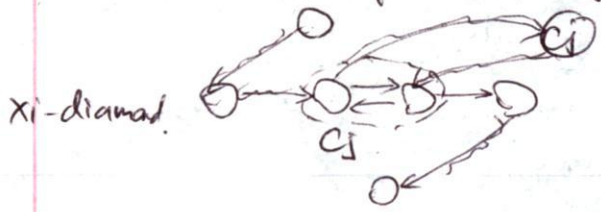


Define an assignment A for φ by
A(x_i)=1 if P zig-zag the x_i-diamond
A(x_i)=1 zag-zig — —

We show A satisfies φ
A satisfies each clause c_j
$1 \le j \le k$.

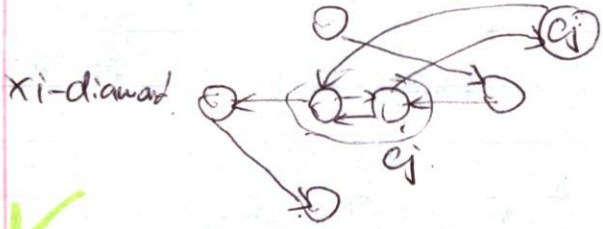
The clause node c_j must have a defined detour to/from it on P

If this detour is from left to right. from the C_j-pair of x_i-diamond



Then P must traverse the x_i-diamond zig-zag.
Then $A(x_i) = 1$, and x_i appears in C_j
So, A satisfies C_j.

If the detour is from right to left. from ·····



Then P must traverse the x_i ····
zag-zig.
Then $A(\bar{x}_i) = 1$. and $\bar{x}_i$ appears in C_j.
So. A satisfies C_j.

UHAMPATH described in Theorem 7.55 on page 319

Self-study HAMPATH $\leq_p$ UHAMPATH

Summerizey of NP-Complete problems

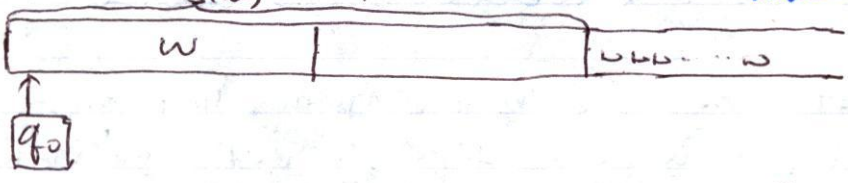
- All NPC. problems are polynomially equivalent. For all $A, B \in$ NPC., $A \leq_p B$ and $B \leq_p A$.
- If any one NPC problem has a polynomial-time algorithm, all have (widly believed to be unlikely)
- If any one ····· does not have a poly time algorithm. none has (widly believed to be likely)
- Many NPC problem are practically useful.
(上一次上课发的资料).

Chapter 8: Space Complexity

Let M be a DTM that halts on all input strings.

$D_n = \{ w \in \Sigma^* \mid |w| = n \}$ = the set of all input strings of size n.

$s(w)$ = the total # of distinct cells M reads on input string w.



$s(w) < |w|$ is possible since M may not read all of w.

$$W_M^{\text{space}}(n) = \max \{ s(w) \mid w \in D_n \}$$

Let M be an NTM where all branches of computations trees halt on all input strings.

$s(w)$ = the max # of distinct tape cells M reads on any branches on input w.

$W_M^{space}(n) = \max\{s(w) \mid w \in D_n\}$

We will omit superscript "space" when there is no confusion.

SPACE$(f(n)) = \{L \mid L = L(M)$ for a 1-tape decider DTM, M, with $W_M^{space}(n) = O(f(n))\}$

NSPACE$(f(n)) = \{L \mid L = L(M)$ for a 1-tape decider NTM, M, $\ldots\}$

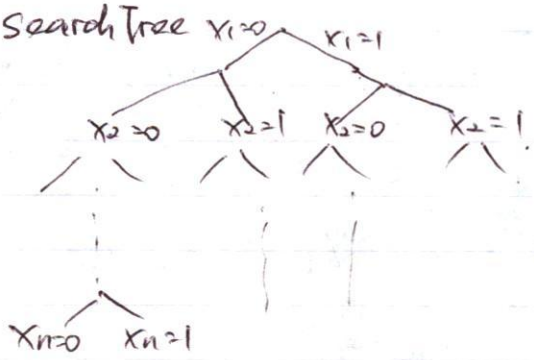
$PSPACE = \bigcup_{k \geq 0} SPACE(n^k)$ PSPACE = NPSPACE.

$NPSPACE = \bigcup_{k \geq 0} NSPACE(n^k)$ NP $\subseteq$ PSPACE.

$\Rightarrow$ SAT $\in$ SPACE(n). Use a brute-force algorithm trying all 0,1 values for all variable in φ $x_1 \ldots x_n$.

On tape, store boolean formula. φ $O(n)$

0/1 Search Tree



On tape, store an assignment to the vars $O(n)$

work space to evaluate φ on an assignment. $O(n)$

Total = $O(n)$

Space can be re-used, but time cannot be

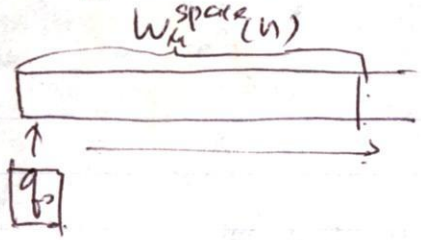
Basic Theorem Let M be any DTM or NTM that halts on all input strings, and let T and Q be its tape alphabet and state set. Then

(i) $W_M^{space}(n) - 1 \leq W_M^{time}(n) \leq d^{W_M^{space}(n)+1}$.

(ii) $\log_d W_M^{time}(n) - 1 \leq W_M^{space}(n) \leq W_M^{time}(n) + 1$

$d = |T| + |Q|$

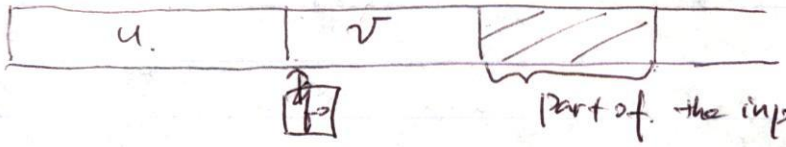
Proof: In the start configuration, the head is reading the 1st cell. To read $W_M^{space}(n)$ cells, the head must move at least $W_M^{space}(n) - 1$ times.



i.e., at least $W_M^{space}(n) - 1$ transitions.

So, $W_M^{space}(n) - 1 \leq W_M^{time}(n)$

We represent the configuration by uqv where uv are actually read by M.



Let S be any transition sequence or a branch of the computer tree of M. Earlier in the course, we showed S only contains distinct configurations. For any configuration uqv in S, $|uv| \leq W_M^{space}(n)$, hence $|uqv| \leq W_M^{space}(n)+1$

Hence the length of S is at most $(|\Gamma| + |Q|)^{W_M^{space}(n)+1}$

Hence $W_M^{time}(n) \leq (|\Gamma| + |Q|)^{W_M^{space}(n)+1}$

This proves (i). (ii) follows from (i) by solving for $W_M^{space}(n)$.

It is one of most important.

Theorem 1 (Savitch) For any $f: N \to R^+$ s.t. $f(n) \geq n$ for all n, $NSPACE(f(n)) \subseteq SPACE(f^2(n))$

If L is decided by an NTM, N, with $W_N^{space}(n) = O(f(n))$, then L is decided by a DTM, M, with $W_M^{space}(n) = O(f^2(n))$

proof Let $L \in NSPACE(f(n))$. Then $L = L(N)$ for a decider NTM, N, with $W_N(n) = O(f(n))$. N is modified slightly as follows:

Whenever N accepts an input w, it clears all non-blank tape symbols by $\sqcup$ and enters a new, unique accepting configuration $q_{accept}\sqcup\sqcup\cdots$ $O(f(n))$

Call this configuration C_{accept}. (This does not change the space complexity)

Let $W_N(n) \leq d \cdot f(n)$, where d is a constant. The simulating DTM relies on the following configuration graph of N for input size n.

Nodes are all configurations using at most $d \cdot f(n)+1$ symbols in the original sense, including all symbols on tape.

Edges: $\exists$ directed edge $C_1 \to C_2$ iff $C_1 \vdash_N C_2$

procedure $PATH_N(C_1, C_2, t, n) = \begin{cases} \text{accept if } \exists \text{ path from } C_1 \text{ to } C_2 \text{ of length} \leq t \text{ in the graph} \\ \text{reject, otw.} \end{cases}$

C_1 C_2 are configurations in the graph.

t is an integer, n is an integer representing the input size

(Note: Path is called CANYIELD in the book)

Body code Divide-and-conquer recursive procedure.

if $t=0$: if $c_1 = c_2$ return accept else reject.

if $t=1$: if $c_1 = c_2$ or $c_1 \vdash_N c_2$ return accept else reject.

if $t>1$: foreach configuration c_m s.t $(|c_m| \le df(n)+1)$

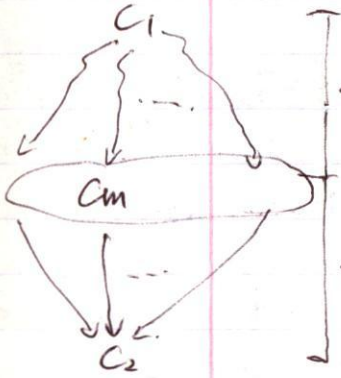
{ $x_1 = \text{Path}_N(c_1, c_m, \lfloor \frac{t}{2} \rfloor, n)$;

$x_2 = \text{Path}_N(c_m, c_2, \lceil \frac{t}{2} \rceil, n)$;

if both x_1 and x_2 are accept, return accept;

}

reject. if. no such c_m exists



From basic Theorem, the length of any branch of Ns computation tree $\le d^{df(n)+1}$, $d = |\Gamma| + |Q|$ (alphabet, state/control)

Hence the length of any path from C_{start} to C_{accept} is $\le d^{df(n)+1}$ in the configuration graph. Now, define the DTM, M, as follows:

Given input w.

1. Compute $n = |w|$ and $f(n)$, assume $f(n)$ is computable.
2. Call. $\text{Path}_N(C_{start}, C_{accept}, d^{df(n)+1}, n)$.

($d^{df(n)+1}$: just upper bound. suffices as upper bound. since)

M decides $L = L(N)$

Space analysis of M.

Computing $n = |w|$ can be done in $O(n)$ space.

Since $n \le f(n)$ for all n., it can be done in $O(f(n))$ space and hence in $O(f^2(n))$ space.

(Note: Say $f(n) < 1$, e.g. $f(n) = c < 1$ Then $f^2(n) < f(n)$)

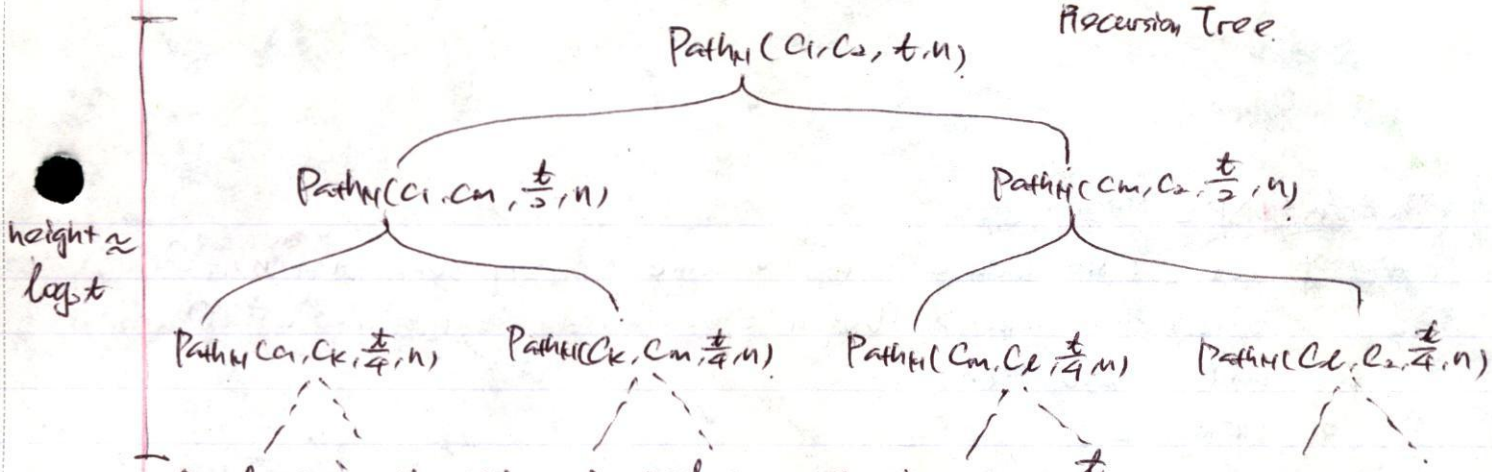
For now, we assume $f(n)$ can be computed in $O(f^2(n))$ space.

More about this later.

Analysis of the space requirement of $\text{Path}_N(c_1, c_2, t, n)$.

Its recursion tree is an almost balanced binary tree.

Recursion Tree



At depth i, the value of 3rd parameter becomes $\approx \frac{t}{2^i}$

Eventually $\frac{t}{2^i} \approx 1$, $t \approx 2^i \Leftrightarrow i \approx \log_2 t$

So the height of the recursion tree $\approx$ stack peak size $\approx \log_2 t$

Each call to pathN needs to store 3 configurations $C_1 C_2 C_m$ requiring space $O(3 \cdot (df(n)+1)) = O(f(n))$

Hence the total space requirement =:

(stack peak size) $\times O(f(n)) =$

$\log_2 t \times O(f(n))$

2 is constant

Hence the space requirement of $Path_N(C_{start}, C_{accept}, d^{df(n)+1}, n)$ is

$\log_2 d^{df(n)+1} \times O(f(n)) =$

$(df(n)+1) \log_2 d \times O(f(n)) =$

$O(f^2(n))$ This concludes Theorem 1 except the issue of:

Can $f(n)$ be computed in $O(f^2(n))$ space?

DTM

1. If this is not the case,

2. Call $Path_N(C_{start}, C_{accept}, d^{df(n)+1}, n)$. Try calling $Path_N(C_{start}, C_{accept}, d^i, n)$, $i = 1, 2, 3, \ldots$ until PathN accepts or reject.

It can be shown that $W_M^{time}(n) = d^{O(f(n))}$

assuming $f(n)$ can be computed in $d^{O(f(n))}$ time

$PSPACE = \bigcup_{k \geq 0} USPACE(n^k)$

$NPSPACE = \bigcup_{k \geq 0} NSPACE(n^k)$

✓

Theorem 2 PSPACE = NPSPACE

proof PSPACE $\subseteq$ NPSPACE is obvious as every DTM is an NTM.

By Theorem 1, $NSPACE(n^k) \subseteq SPACE((n^k)^2) = SPACE(n^{2k}) \in PSPACE$ except $k=0$. Theorem 1 has the condition $f(n) \geq n$ for all n.

$NSPACE(n^0) = NSPACE(1)$. This is decided in $O(1)$ space, which can be decided in $O(n)$ space. $n^0 = 1$

So, $NPSPACE(1) \subseteq NSPACE(n) = SPACE(n^2) \subseteq PSPACE$.

Theorem 3 NP ⊆ NPSPACE.

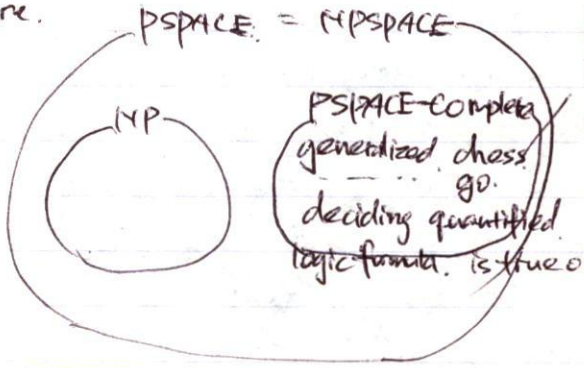
proof Let $L \in NP$. Then $L = L(N)$ for some NTM, N, with $W_N^{time}(n) = O(n^k)$.
By Basis theorem (1), $W_N^{space}(n) \leq W_N^{time}(n) + 1$. But $W_N^{time}(n) = O(n^k)$,
and so is $W_N^{space}(n)$.
Hence, $L \in$ NPSPACE.

$\forall x \exists y \forall z \exists w f(x, y, z, w)$

Theorem 4 NP ⊆ PSPACE.

conjectured picture.

NP ⊂ PSPACE



Widely believed that nondeterminism does not help.

A language B is **PSPACE-complete** if

1. $B \in$ PSPACE, and
2. For all $A \in$ PSPACE, $A \leq_p B$ (PSPACE-hardness)
   ↑ polynomial time reduction

Example of PSPACE-Complete language
TQBF: True Qudified Boolean Formulas

A quantified Boolean formula has the form,
$Q_1 x_1 Q_2 x_2 \cdots Q_k x_k \varphi(x_1, x_2, \ldots x_k)$ where each Q_i is $\forall$ or $\exists$ and
$\varphi(x_1, x_2, \ldots x_k)$ is a quantifier-free formula of k variables.
like, $\forall x_1 \exists x_2 [(x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2})]$
For all values (0, 1) of x_1, there exists one value for x_2, s.t
$(x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2})$ is true.
The problem: Decide if a given QBF is true or false
Let's show TQBF ∈ PSPACE.

0/1 evaluation algorithm

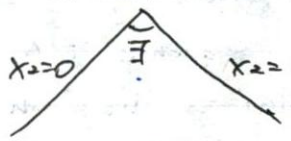
For k vars. the tree height = k.

$1 \wedge 1 = 1$ and

$= \forall x_1 \exists x_2 [(x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2})]$

$0 \vee 1 = 1$ or.

$\exists x_2 [(0 \vee x_2) \wedge (1 \vee \bar{x_2})]$ $\quad$ $\exists x_2 [(1 \vee x_2) \wedge (0 \vee \bar{x_2})]$ $\quad$ $1 \vee 0 = 1$



$[(0 \vee 0) \wedge (1 \vee 1)]$ $[(0 \vee 1) \wedge (1 \vee 0)]$ $[(1 \vee 0) \wedge (0 \vee 1)]$ $[(1 \vee 1) \wedge (0 \vee 0)]$

$=0$ $=1$ $=1$ $=0$

Eval(F)

if F has no quantifiers (all vars have 0/1 substituted)

evaluate F and return the value

if $F = \exists x_i \varphi(x_i, \cdots x_k)$ // $\varphi(x_i, \cdots x_k)$ has quantifiers

// $Q_{i+1}, \cdots Q_k$ over $x_{i+1}, \cdots x_k$

{ e_1 = Eval($\varphi(0, x_{i+1}, \cdots x_k)$);

e_2 = Eval($\varphi(1, x_{i+1}, \cdots x_k)$);

return $e_1 \vee e_2$;

}

if $F = \forall x_1 \varphi(x_i, \cdots, x_k)$

{ e_1 = Eval($\varphi(0, x_{i+1}, \cdots x_k)$);

e_2 = Eval($\varphi(1, x_{i+1}, \cdots x_k)$);

return $e_1 \wedge e_2$;

}

space requirement of Eval

n = size of F with k vars $x_1, \cdots x_k$ in some encoding

store F $O(n)$

each stack frame stores a reduced F $O(n)$

work space for evaluation. $O(n)$

stack peak size = depth of the evaluation tree = k

In total, $O(n) \times k = O(n) \times O(n) = O(n^2)$

$k = O(n)$

TQBF $\in$ PSPACE

For all $A \in PSPACE$, $A \leq_p TQBF$

Proof outline. Let $A \in PSPACE$.
Then A is decided by a DTM, M, with $W^{space}_M(n) = O(n^k)$
It is possible to give a poly-time reduction of s.t
$\forall w \in \Sigma^*$, $w \in A \Leftrightarrow f(w)$ is a QBF evaluating to 1.
The formula $f(w)$ simulates the transition sequence of M on w,
using a recursive technique ~~simulate~~ similar to Theorem 1c Savitch)

TQBF is known to be in NP
(widely believed to be not in NP)
Nondeterminism does not seem to help for evaluation of $\forall$
There seems to be no polynomial-size certificate.

Formula Game Problem (2-player game interpretation of TQBF)
Let φ be any QBF, $Q_1x_1\, Q_2x_2 \cdots Q_kx_k\, \varphi(x_1, x_2, \cdots x_k)$
Player E takes turns on $\exists x_i$ They select 0 or 1 for x_i
" A " $\forall x_i$ in their turns.
The game is played in order of the quantifiers $Q_1, \cdots Q_k$.
E wins if φ evaluates to 1 on the selected 0/1 values
A wins " " 0 " " " " .
So a game sequence corresponds to a path from the root to a leaf in the evaluation tree.
A player has a winning strategy if it can select values of x_i's in its turns to win no matter what values are selected by the opponent
For this problem: for any given φ, one of the players always has a winning strategy

So this is in PSPACE → Problem Decide if E has a winning strategy.

Theorem: φ is true iff E has a winning strategy (φ is false iff A has a winning strategy)

Proof
- "$\forall x_i$" is interpreted as "for all values of x_i selected by A"
- "$\exists x_i$" is " " "there exists a value of x_i to make φ true."
- "$\forall x_i$": For any value of x_1 selected by A
- "$\exists x_i$": For There exists no value of x_i to make φ true

$\exists x_1 \exists x_2 \exists x_3 \cdots \exists x_k \varphi(x_1, \cdots x_k) = 1$ $\quad \forall x_1 \forall x_2 \forall x_3 \cdots \forall x_k \varphi(x_1, \cdots x_k) = 0$

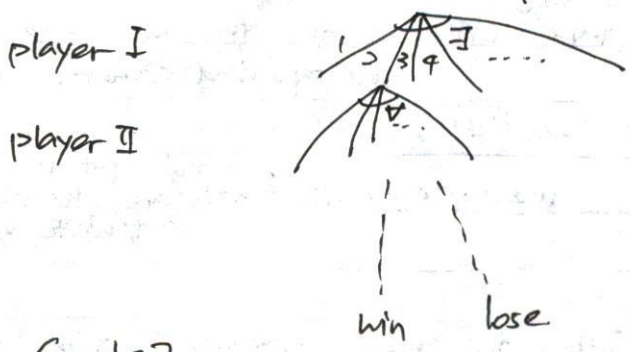
$= 0$ $\quad = 1$

Generalized chess/go

Input: a configuration on $n \times n$ board

Decide if the 1st player ~~or the 2nd~~ has a winning strategy.

The variables represent the possible moves



This can be represented by a QBF

In principle, we can evaluate this QBF by evaluation tree.

Running this requires too much time and memory.

Google]

AI planning problems can be formulated as QBF problem.

11/22. 开机 widly PIC

Theorem 5 If $A \leq_p B$ and $B \in NP$, then $A \in NP$

proof Idea Construct a verifier (NTM) for A from the verifier (NTM) for B and the poly-time reduction from A to B.

Theorem 6 If $\exists A \in NP \cap PSPACE\text{-Complete}$, then $NP = PSPACE$ $(NP \cap PSPACE \neq \phi)$

proof $NP \subseteq PSPACE$ is provide in Theorem 4 We will show $PSPACE \subseteq NP$

Suppose $A \in NP \cap PSPACE$-complete. By def of PSPACE-Complete. for all $L \in PSPACE$, $L \leq_p A$

But. $A \in NP$, so by Theorem 5 $L \in NP$. Hence $PSPACE \subseteq NP$

Theorem 7 If $\exists A \in P \cap PSPACE$-Complete, then $P = NP = PSPACE$.

Proof Suppose $A \in P \cap PSPACE$-Complete. Since $P \subseteq NP$, $NP = PSPACE$

Since $A \in PSPACE$-Complete, $\forall L \in PSPACE$, $L \leq_p A$. Since $A \in P$, $L \in P$

Hence. $PSPACE \subseteq P$ Hence we have $P \subseteq NP = PSPACE \subseteq P$

Hence $P = NP = PSPACE$

Theorem 8 If $\exists A \in NPC \cap PSPACE$-Complete, then $NPC = PSPACE$-Complete

手抓图2. **proof.** similar to Thm 6 proof.

Now we turn to inside P

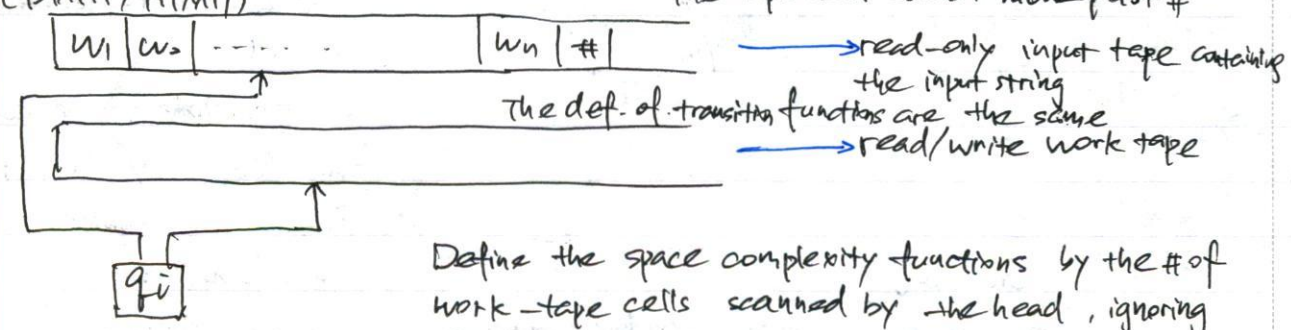
Class L and NL

Motivation: Other than the input itself, how much / little space do we need to decide a language?

Especially the input size is large.

$\log n$ is a typical example of sub-linear function

A DTM/NTM is a 2-tape TM with the following components: (DTMB / NTMB) with the read-only tape



The tape head cannot move past #

→ read-only input tape containing the input string

The def. of transition functions are the same

→ read/write work tape

Define the space complexity functions by the # of work-tape cells scanned by the head, ignoring the input.

Define $SPACE(f(n)) = \{L(M) \mid M$ is a DTMB with $W^{space}_M(n) = O(f(n))\}$

$NSPACE(f(n)) = \{L(M) \mid$ -- -- NTMB -- -- -- -- -- -- $\}$

$L = SPACE(\log n)$

$NL = NSPACE(\log n)$

Theorem 1. $L \subseteq NL$

Example of L: $\{0^k 1^k \mid k \geq 0\}$ (does not need work tape)

1. Scan the input and reject if any 0 follows immediately after 1 ↙
2. Count the # of 0's and 1's on the work tape using binary numbers and compare. $n = |w|$ $w = 0^k 1^k$.

This can be done in $O(\log_2 n)$

Example of NL. PATH Problem: Input (G, s, t) stored in the input

Decide if ∃ path from start in a directed graph G

We encode nodes by binary numbers from 1 to m. Each binary number needs $O(\log_2 m)$ space.

PATH $\in \mathcal{L}$? a big question

We'll show PATH $\in NL$-Complete

$n = s$; $i = 1$; n, i, m s.t. are all in binary numbers

While ($i \le m$) // m = # of nodes

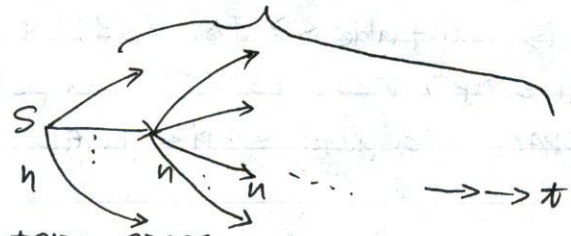
{ among the nodes adjacent to n , nondeterministically select the ~~number~~ node to visit, and update n to it;

if ($n = t$) accept;

$i = i + 1$;

}

reject;



This needs $O(\log n)$ work-tape space.

Theorem 2

(i) If $A \le_{\mathcal{L}} B$ and $B \in \mathcal{L}$ then $A \in \mathcal{L}$

(ii) $A \le_{\mathcal{L}} A$ (reflexivity)

(iii) $A \le_{\mathcal{L}} B$ and $B \le_{\mathcal{L}} C \Rightarrow A \le_{\mathcal{L}} C$ (transitivity)

(iv) If $A \in NL$-Complete, $A \le_{\mathcal{L}} B$, and $B \in NL$, then $B \in NL$-Complete

(v) If $\exists A \in \mathcal{L} \cap NL$-Complete, then $\mathcal{L} = NL$ (span problem)

If $\Sigma^* \to \Sigma^*$ is computed by an $\mathcal{L}$-ST if for any $w \in \Sigma^*$, placed on the input tape, the $\mathcal{L}$-ST writes $f(w)$ on tape and halts (the output)

Such f is said to be log-space computable

A is log-space reducible to B, $A \le_{\mathcal{L}} B$, if there exists a log-space computable function f s.t. for $\forall w \in \Sigma^*$, $w \in A \Leftrightarrow f(w) \in B$

~~And~~ A language B is NL-complete if

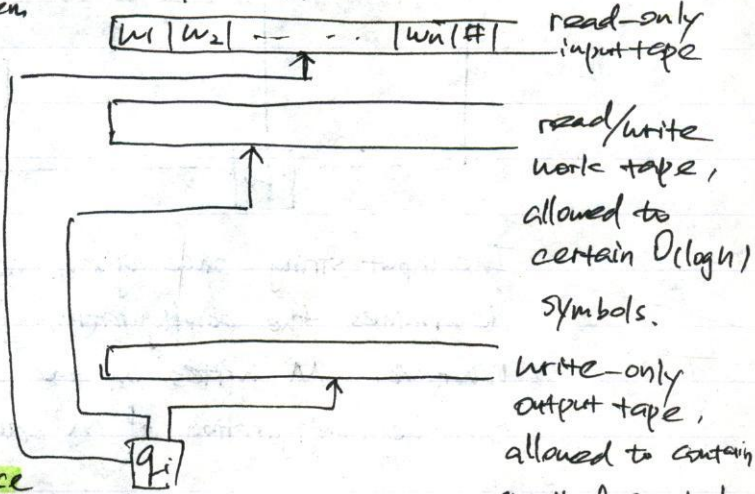
(i) $B \in NL$, and

(ii) For all $L \in NL$, $L \le_{\mathcal{L}} B$

Log-space reduction

Many-to-one reduction that needs $O(\log n)$ space.

A log-space transducer is a 3-tape DTM with the following components:



11/29

$\forall L_1 \in NP$, $L_1 \le_p SAT \le_p TQBF \le_p L_2$ for $\forall L_2 \in PSPACE - Complete$

poly-time reduction

$\varphi(x_1, \dots x_n) \mapsto \exists x_1 \cdots \exists x_n \varphi(x_1, \dots x_n)$

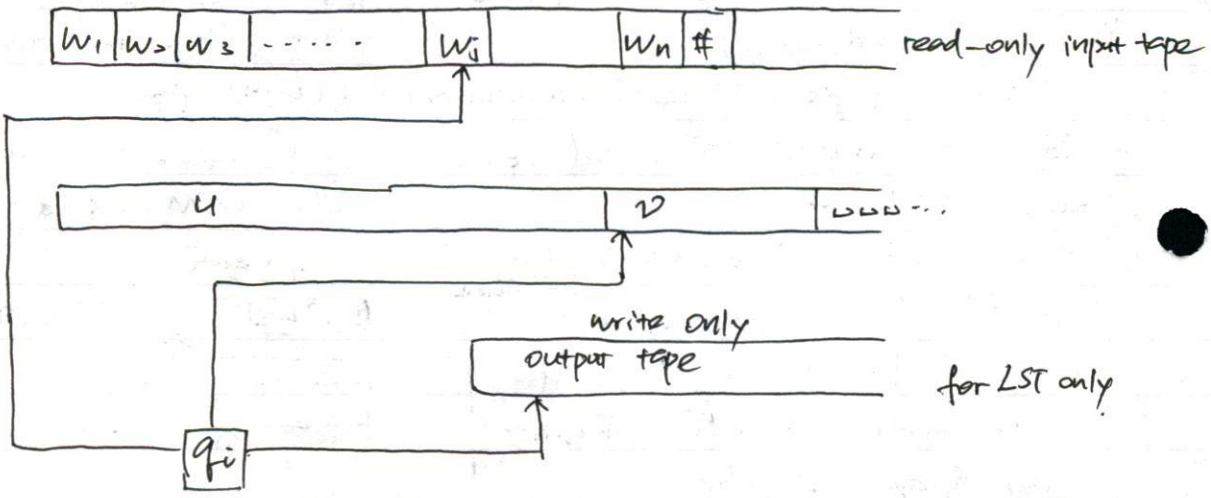
φ is satisfiable $\Leftrightarrow \exists x_1 \cdots \exists x_n \varphi(x_1, \dots x_n)$ is true

Theorem $\forall L_1 \in NP$, $\forall L_2 \in PSPACE - Complete$, $L_1 \le_p L_2$

Theorem $PSPACE - Complete \subseteq NP - HARD = \{A \mid \text{for } \forall L \in NP,\ L \le_p A\}$

Theorem 3 Let M be any DTM, NTM, or LST with $W_M^{space}(n) \le f(n)$
Then $W_M^{time}(n) \le n \cdot \alpha^{f(n)+1} = n \cdot 2^{O(f(n))}$, where $\alpha = |\Gamma| + |Q|$

Proof Consider configuration of DTM, NTM or LST



The input string, once given, never changes. So the position number j determines the configuration for the input tape

Whatever M writes on the output tape has no effect on its transition

So, the configuration of M can be represented by $juqiv$

(u, v for work tape; j input position number)

By a proof analogous to Basic Theorem, $W_M^{time}(n) \le n \cdot \alpha^{f(n)+1}$

Theorem 4 Let M be any DTM, NTM, or LST with $W_M^{space}(n) \ge O(\log n)$
Then $W_M^{time}(n) = O(n^k)$

Proof By theorem 3, $W_M^{time}(n) \le n \cdot 2^{O(\log n)}$ Let $\log n = \log_a n = \frac{\log_2 n}{\log_2 a}$

$2^{O(\log n)} \le 2^{c \log n} = 2^{c \frac{\log_2 n}{\log_2 a}} = (2^{\log_2 n})^{\frac{c}{\log_2 a}} = n^{\frac{c}{\log_2 a}}$

$\frac{c}{\log_2 a}$ is constant

Amount of this can be $O(\log n)$

M_1 only writes a portion of output needed by M_2's next transition. This is possible

Theorem 5 if $A \leq_L B$, $A \leq_P B$

Proof Suppose $A \leq_L B$ Then on LST, M_1 computes the reduction from A to B with $W_{M_1}^{space}(n) = O(\log n)$. By Theorem 4. $W_{M_1}^{time}(n) = O(n^k)$. M_1 is a 3-tape DTM.

By Theorem 7.8 in the book, M_1 can be simulated by an equivalent 1-tape DTM, M_1', with $W_{M_1'}^{time}(n) = O(W_{M_1}^{time}(n))^2) = O((n^k)^2) = O(n^{2k})$

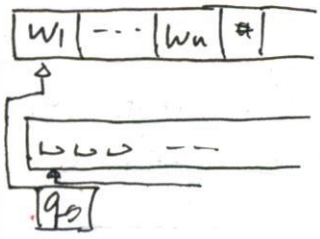
Theorem 7.8 requires the condition $W_{M_1}^{time}(n) \geq n$. If $W_{M_1}^{time}(n) < n$, let M_1 redundently read all n input symbols, to make $W_{M_1}^{time}(n) \geq n$.

Theorem 6 PATH $\in$ NL-Complete | (G, s, t)
Decide if $\exists$ path from s to t.

proof We saw PATH $\in$ NL by a nondeterministic algorithm. We prove $\forall A \in NL$, $A \leq_L$ PATH. Suppose $A \in NL$. Then A is decided by an NTM(1), M_1 with $W_{M_1}^{space}(n) = O(\log n)$

proof idea We encode computation of M_1 by a graph. Specially we use the configuration graph used in Savitch's Theorem.

Let $W_{M_1}^{space}(n) \leq d \cdot \log n$. We slightly modify M_1 so that whenever it enters an accepting configuration, it clears all non-blank tape symbols and enters a unique accepting configuration C_{accept}. This does not change space requirement. Define a log-space reduction f as follows. For any w, $f(w) = (G, s, t)$ where G is the configuration graph of M_1 for input string w as defined in Savitch's Theorem. s is the start configuration of M_1 on w, $C_{start} = 1 q_0 w$, $t = C_{accept}$.



the nodes are all the configurations of M_1 of the form $j u q_i v$ where $1 \leq j \leq n$ is a binary number and $|uqiv| \leq d \cdot \log n + 1$

the edges are a directed edge representing $C_1 \vdash_{M_1} C_2$

Space requirement for j is $O(\log n)$ // Space requirement for $uqiv$ is $O(\log n)$

So space requirement for any configuration is $O(\log n)$.

The equivalent condition holds: $w \in A \Leftrightarrow M_1$ accepts $w \Leftrightarrow \exists$ path from $C_{start} = s$ to $C_{accept} = t$ in G. (G, s, t) can be generated by an LST.

Generating nodes: One by one, generate all possible configuration $j u q_i v$ on work tape and write it on output tape

This can be done in $O(\log n)$ space of work tape (fun in controls state and the transition function of the LST)

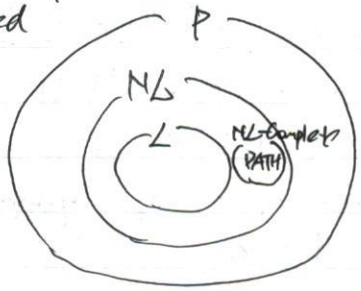
needs $O(\log n)$ space. Generating edges: One by one, generate all possible pairs of configurations $\rightarrow (C_1, C_2)$ on work tape. check if $C_1 \vdash_{M_1} C_2$ (can be checked in additional constant space by encoding M_1's transition) if so, write (C_1, C_2) on output tape

s.t. requires $O(\log n)$ space.

Theorem 7 $NL \subseteq P$

proof Let $A \in NL$. By theorem 6, $A \leq_L PATH$, by theorem 5, $A \leq_p PATH$
There is a straightforward deterministic poly-time algo to decide PATH
(depth-first or breadth-first search)
So, $PATH \in P$. So $A \in P$

~~Conjuction~~ picture
Conjectured



Approximation Algorithms

No poly-time algo is known to any of these

Many NP-Complete problems have corresponding optimization problems (minimization/maximization)

3 eg. of minimum
- Minimum Vertex Cover
- Travelling Salesperson: Find a Ham cycle of minimum total cost
- Minimum Graph coloring: Find a minimum k value that can color the graph

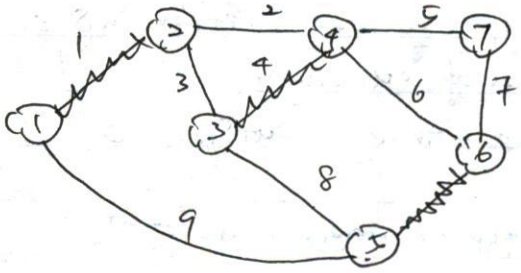
- Max clique - Find a largest clique
- Max cut in the graph
- knapsack problem - capacity C of knapsack, size/profits of n objects
  - Find a subset of objects that fit in C and maximizes the total profits

Let $C(I)$ = the value of a solution found by the approx algo for input I
$C^*(I)$ = the optimum value for input I
The approx algo is said to have a ratio bound of $\rho(n)$ if
- For max problems
$$1 \leq \frac{C^*(I)}{C(I)} \leq \rho(\text{size}(I)) \text{ for all input } I$$
- For min problems
$$1 \leq \frac{C(I)}{C^*(I)} \leq \rho(\text{size}(I)) \text{ for all input } I$$

Generally, as the ratio gets closer to 1, the approx algo's efficiency (slows down) decreases.

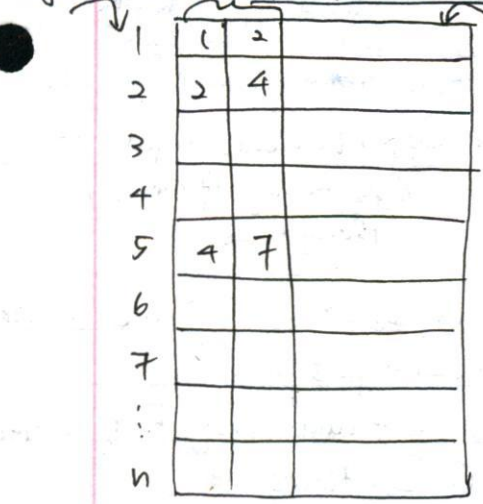
MIN-VERTEX-COVER Input: undirected $G=(V,E)$. Initially, all edges are unmarked.

for each edge $e \in E$ do
  if (e is untouched by any marked edges)
    mark e; ↳ does not share end points



output the end vertices of the marked edges.

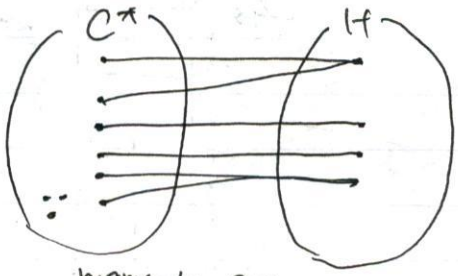
output {1, 2, 3, 4, 5, 6}



$O(|E| \times |E|)$
$= O(|E|^2)$

Hence $|H| \le |C^*|$
$|C| = 2|H| \le 2|C^*|$

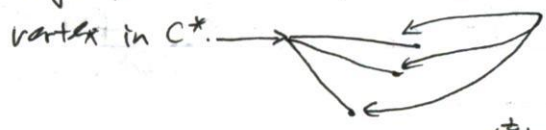
Hence $|H|$



Theorem Let C be the vertex cover computed by the algorithm, and let C^* be a min vertex cover for the same input graph. Then $|C| \le 2|C^*|$

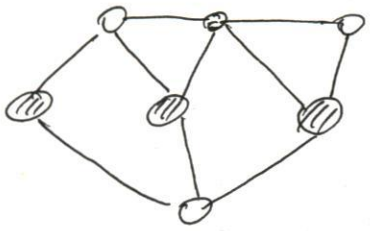
proof Let H be the set of marked edges. All edges in H are disjoint (they do not share any end vertices) since only untouched edges are marked. Since the output is the end vertices of H, $|C| = 2|H|$.

Now C^* is a vertex cover so it covers all edges in H. Since H is disjoint, each vertex in C^* covers at most one edge in H.



Hence the correspondence between C^* and H is many to one

得证.

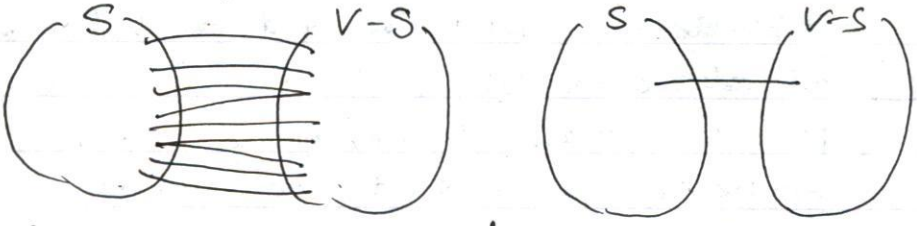


min vertex cover has 4 vertices

$\frac{c}{c^*} = \frac{6}{4} = 1.5$.

MAX-CUT problem Input: connected undirected $G=(V,E)$

A cut is a division of V into S and $V-S$



A cut edge is one with one end vertex in S and the other in $V-S$

Find a cut with max # of cut edges.

$S=\phi$; $T=V$; increased = true; cutSize = 0

while (increased)

{ for (each $v \in V$) flip the boolean value of v

{ move v to the other side and check if the cut size increases;

if (increases) { update cutSize } scan the edge array and decide if each edge is cut or not and tally the # of cut edges

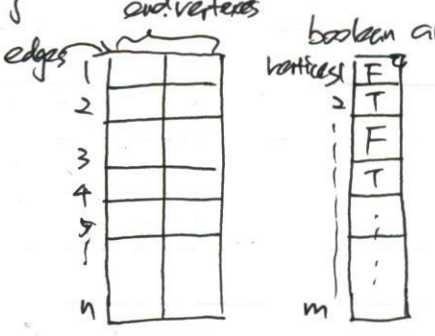
else {undo the move of v}

}

if (cut size has not increased in the for-loop) {increased = false;}

// no moving of any vertex increased the cut size

}



F: in S

T: in $V-S=T$

"increased" can increase at most $|E|$ times.

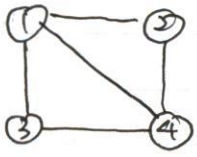
So, # of while-loop body is iterated $O(|E|)$ times.

The for-loop iterates $|V|$ times.

As single execution of The body of the for-loop takes $O(|E|)$ times

In total, $O(|E| \times |V| \times |E|)$

eg:



$S=\phi$ $T=\{1,2,3,4\}$ cutSize = 0

$v=1$ $\{1\}$ $\{2,3,4\}$ cutSize = 3

$v=2$ ~~$\{1,2\}$ $\{3,4\}$ cutSize = 3~~ undo

$v=3$ ~~$\{1,3\}$ $\{2,4\}$ cutSize = 3~~ undo

$v=4$ $\{1,4\}$ $\{2,3\}$ cutSize = 4 → Solution

v=1 ~~{4} {1,2,3} cutsize=3~~ undo

v=2 ~~{1,4,2} {3} cutsize=2~~ undo

v=3 ~~{1,4,3} {2} cutsize=2~~ undo

v=4 ~~{1} {2,3,4} cutsize=3~~ undo

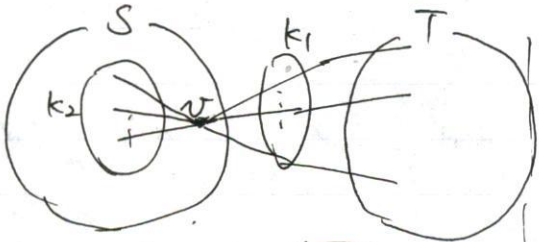
terminate

12/6. **Theorem.** Let C = total # of cut edges computed by the algorithm,
C^* = the max # of cut edges

Then $\frac{C^*}{C} \leq 2$.

Proof Let (S,T) be the cut produced by the algorithm. Consider any $v \in V$
Without loss of generality, assume $v \in S$. Let

k_1 = # of cut edges from v, k_2 = # of uncut edges from v



Hence, $\sum_{v \in V}$ (# of uncut edges from v) $\leq$
$\sum_{v \in V}$ (# of cut edges from v)

2 x (the # of uncut edges) $\leq$
2 x (the # of cut edges)

the # of uncut edges $\leq$ the # of cut edges
$\|$ $\|$
v C

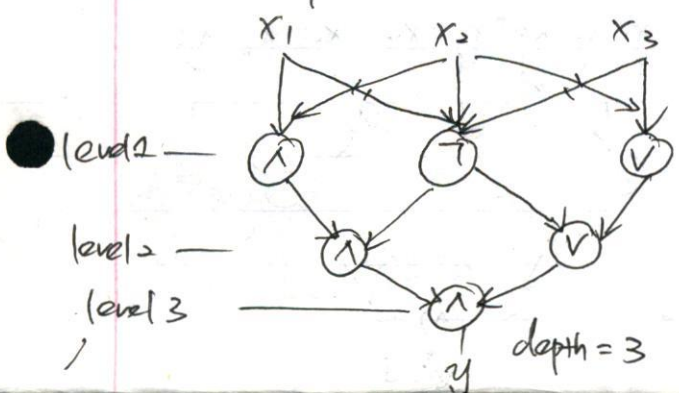
Let |E| = # of edges in G
$|C^*| \leq |E| = v + C \leq 2C$

claim $k_1 \geq k_2$

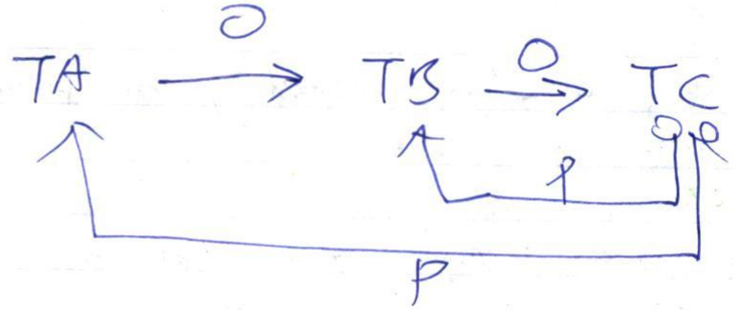
Proof Suppose $k_1 < k_2$. Move v to T.
Then the k_1 edges are uncut edges and the k_2 edges are cut edges. So the updated # of cut edges is
(previous # of cut edges) $+ k_2 - k_1$
But $k_2 - k_1 > 0$
So, the # of cut edges have increased, and the algorithm should have moved v. but it did not, a contradiction.

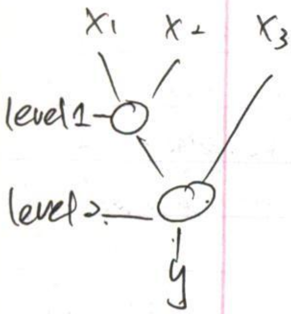
Circuit Complexity and its relation to P-completeness

A Boolean circuit consists of AND ($\wedge$) gates, OR ($\vee$) gates, NOT ($\neg$) gates and n input variable $x_1, \cdots x_n$. The wire edges are acyclic



Every circuit C defines a Boolean function
$C(x_1, x_2, \cdots x_n)$
A circuit family $\dot{C}$ is an infinite list of circuits $(C_0, C_1, C_2 \cdots C_n, \cdots)$ where each circuit C_n has (n) input variables. C_0 has no input vars and always returns 0 or 1.





4 pipe

x 2

th Out =8

TA ext T.

Th.

TA, TB, TC

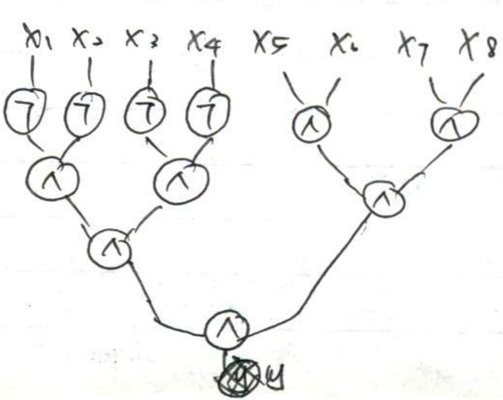
Tc

We say C defines a language $A \subseteq \{0,1\}^*$ if each string w with $|w|=n$

$w \in A \Leftrightarrow C_n(w) = 1$

$\{0^k 1^k \mid k \geq 0\}$ can be defined by a circuit family

0 0 0 0 1 1 1 1



The size of a circuit is the # of gates in it.

The depth of ... is the # of gates in a longest path from the inputs to the output.

The size complexity of a circuit family $C = (C_0, \dots C_n, \dots)$ is
(also known called the processor complexity)
$f(n)$ = the size of C_n.

The depth complexity of ... is
(also called parallel complexity)
$g(n)$ = the depth of C_n.

The gates at the same level can be computed in parallel.

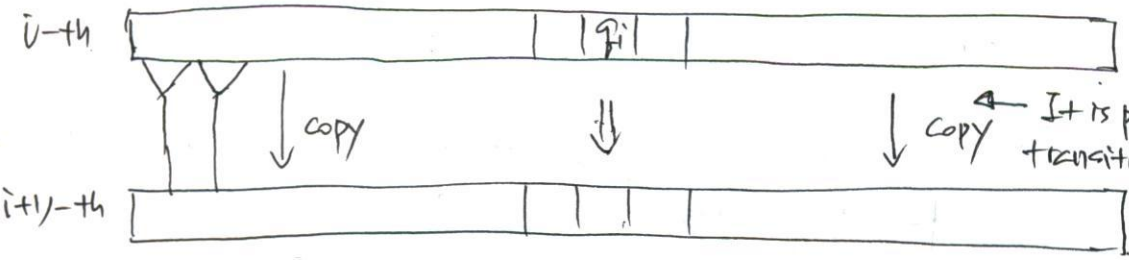
C defines $A \subseteq \{0,1\}^*$. we also say C decides A.

Fact 1 If $A \subseteq \{0,1\}^*$ is decided by a DTM, M, with $W_M^{time}(n) = O(f(n))$, then A is decided by a circuit family with the size complexity $O(f(n))$

Proof Idea Let $W_M^{time}(n) \le a f(n)$ for some constant a.

Consider an $af(n) \times af(n)$ tableau for M like the one we used for the proof of Cook-Levin Theorem. The i-th row represents the ith configuration.

Consider the transition from the i-th config to the (i+1)-th config



It is possible transition by a row of gates may require a constant # of multiple levels. to implement this

We can encode the control states by 0/1 sequence.

Since each row of gates has a constant # of gates,
the total # of circuits is $O((af(n))^2)$

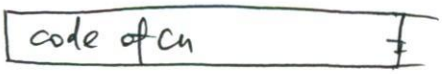
Corollary, If $A \in P$, A has a polynomial circuit size complexity.

Corollary, If A does not have a polynomial circuit size complexity, then $A \notin P$.

The circuit family as defined is unsatisfactory as a computing device since it is an infinite object.

So we require that there is a TM that computes (produces) C_n from n.

A circuit family is called uniform if there exists a log-space transducer that outputs C_n from the input 1^n (unary code of n)



For $i \geq 1$, define:

$NC^i = \{A \subseteq \{0,1\}^* \mid A$ is decided by a uniform circuit family with $O(n^k)$ size (processor) complexity, for some constant k, and $O(\log^i n)$ depth (parallel) complexity$\}$

Nick Pippenger

$\log^i n = (\log n)^i = O(n)$ for any i

less than linear parallel time.

$NC = \bigcup_{i \geq 1} NC^i$

"Nick's class"

$(\log n)^{100} \leq n$ does not hold for $n \leq 10^{299}$

does hold for $n \geq 10^{300}$ 10^{80}

12/8.

Fact 2 (i) $NC^1 \subseteq L \subseteq NL \subseteq NC^2$

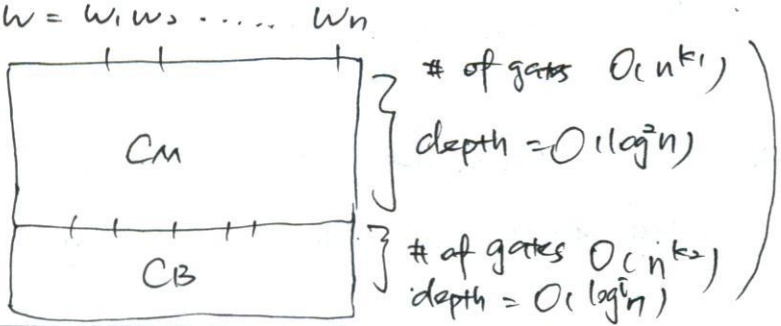
(iii) Any log-space reduction by LST can be implemented by a generalized NC^2 where circuits may have multiple outputs

Fact 3 If $A \leq_L B$ and $B \in NC$, then $A \in NC$

Proof Idea Suppose $A \leq_L B$ and $B \in NC$. Then there is a log-space reduction from A to B computed by an LST, M,

Also, there is a uniform circuit family C_B in NC^i, for some i, deciding B. By Fact 2, M can be implemented by a uniform circuit family C_M, in generalized NC^2.

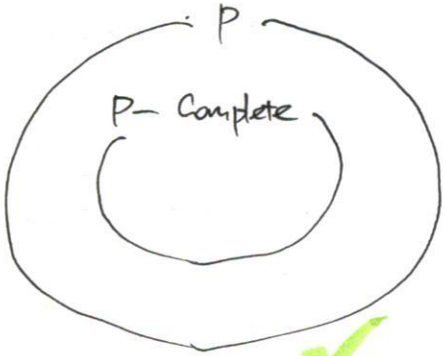
A uniform circuit family to decide A can be constructed by sequentially combining C_M and C_B.



A language B is P-complete if

(1) $B \in P$ and

(2) For each $A \in P$, $A \leq_L B$



Fact For any $A, B \in P$ s.t. $B \neq \phi$ and $B \subset \Sigma^*$, $A \leq_p B$

Proof Let $x_1 \in B$ and $x_2 \notin B$

Define for any w,

$$f(w) = \begin{cases} x_1 & \text{if } w \in A \\ x_2 & \text{if } w \notin A \end{cases}$$

$A \in P$, so there can be decided in poly-time

Then f is a poly-time reduction from A to B

✓ CIRCUIT-VALUE problem $\in$ P-Complete

Input: a circuit C and an assignment x for C's inputs

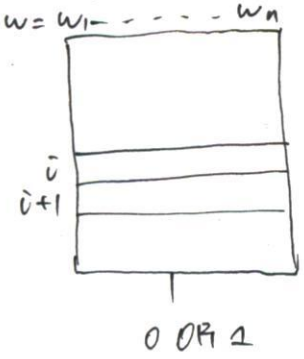
Output: decide if C evaluates to 1 on x

Proof Idea clearly, this can be decided ~~by~~ in poly time.

Let $A \in P$. Assume $A \subseteq \{0,1\}^*$, We can show $A \leq_L$ CIRCUIT-VALUE.

Let M be a poly-time DTM deciding A. Consider again $a f(n) \times a f(n)$ tableau of configuration sequence on input w. where $w_M^{time}(n) \leq a f(n)$, for some polynomial $f(n)$

This can be simulated by rows of ~~circuits~~ gates to produce a circuit C_A to decide if $w \in A$.



Hence, $w \in A \Leftrightarrow C_A(w) = 1$

C_A has highly repetitive structure so that it can be constructed by an LST;

build one gate at a time on work tape and write it out on output tape. $O(\log n)$

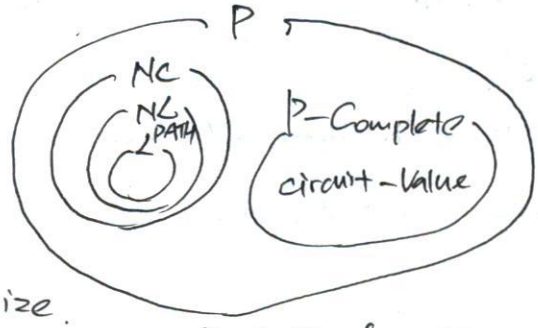
$NC^1 \subseteq L \subseteq NL \subseteq NC^2$

$NC^1 C NC^2 C \ldots C NC^i C$

Fact $NC \subseteq P \subseteq NC = P$ unknown).

proof Idea Let $A \in NC$. Then A is decided by a uniform circuit-family C_A with $O(n^k)$ size.

We construct a poly-time DTM, M, to simulate C_A



widly believed picture

Given any w:

(1) Compute $n = |w|$ in unary notation 1^n — can be done in poly time by Theorem 4 in L/NL-topic

(2) Simulate the LST to produce C_n of $CA = (C_0, C_1, C_2 \cdots C_n) \cdots)$ from 1^n

(3) Simulate C_n on w to decide if $C_n(w) = 1$ This is exactly CIRCUIT-VALUE problem. This is in P.

C_n has $O(n^k)$ size

Hence, (1), (2), (3) run in poly time.

Fact 6 If $\exists B \in P\text{-Complete} \cap NC$, then $NC = P$. (Evidence suggests this is unlikely)

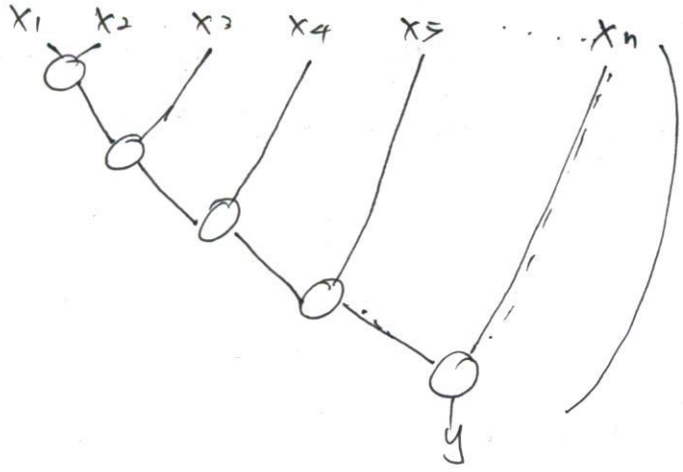
Proof $NC \subseteq P$ by Fact 5. Suppose $B \in P\text{-Complete} \cap NC$

Let $A \in P$ Since $B \in P\text{-Complete}$, $A \leq_L B$. But $B \in NC$

By Fact 3 $A \in NC$. So $P \subseteq NC$

NC represents a class of highly parallelizable decision problems by $O(n^k)$ processors and $O(\log^i n)$ parallel time. $\log^i n = o(n)$

No one has found $B \in P\text{-Complete} \cap NC$. P-complete is believed to be a class of inherently sequential decision problems.



depth $= n-1$ & $O(\log^i n)$

A suprising result

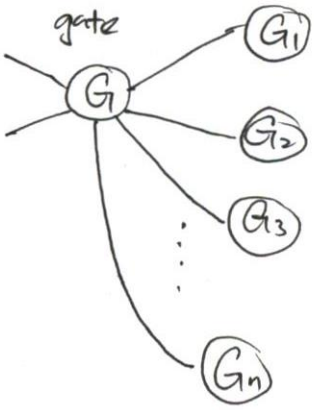
Boolean formula evaluation problem:

Decide if the formula evaluates to 1 on the given assignment.

This is in NC^1.

The Boolean formula is translated into Postfix-longer-Operand-First formula.

Circuits can share values of gates, but Boolean expressions cannot.



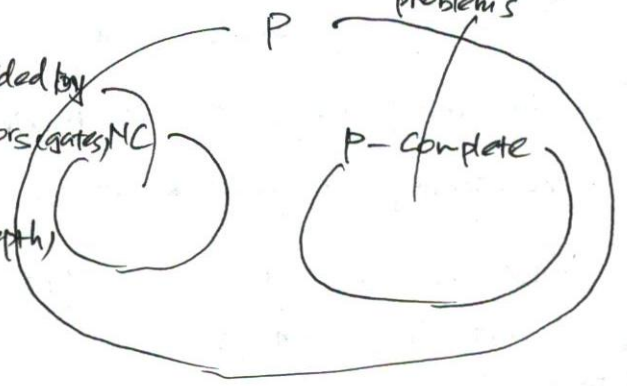
BG_i is a Boolean Formula representing gate G_i.

BG is a Boolean Formula representing gate G

$BG_1(BG, \_)$ $BG_2(\_, BG)$ $BG_3(\_, BG)$ ... $BG_n(BG, \_)$

cannot be shared must be copies of BG.

believed to be inherently sequential problems



12/13

$NC = \bigcup_{i \geq 1} NC^i$

$NC^1 \subseteq NC^2 \subseteq NC^3 \subseteq \cdots \subseteq NC^i \subseteq NC^{i+1} \subseteq \cdots$

Whether any of these is $=$ is unknown

The following has been proved. If $NC^{i+1} = NC^i$ for some i, then $NC = NC^i$

Therefore there are two possibilities:

(i) $NC^1 \subset NC^2 \subset \cdots \subset NC^i = NC^{i+1} = NC^{i+2} = \cdots$ for some i

(ii) $NC^1 \subset NC^2 \subset \cdots \subset NC^i \subset NC^{i+1} \subset NC^{i+2} \subset \cdots$ for all i

Example of P-complete problems.

- DTM acceptence in $\leq n$ steps

  Input: a DTM, M, an input string w, an integer $n \geq 0$ given as unary

  Decide if M accepts w in $\leq n$ steps

11/7

Limits to parallel Computation:

P-Completeness Theory

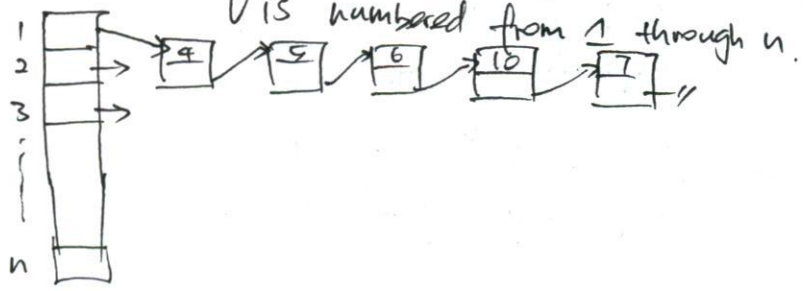
Greenlaw, Hoover, Ruzzo

Realistically, highly parallelizable should be $O(n^2)$ processors and $O(\log^2 n)$ parallel time.

- Depth-first Search

  Input: An undirected graph $G = (V, E)$ given by an ordered adjacency list and two vertices. $u, v \in V$.

Decide if u is visited before v in the depth-first search starting from vertex 1 using the given adjacency list



- Maximal independent set

An independent set of one undirected graph is a set of vertices s.t no two of them are adjacent.

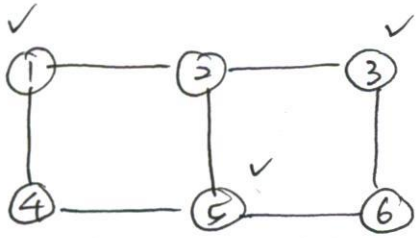
An independent set is maximal if no other vertex can be added while maintaining an independent set.

Find the lexicographically first maximal independent set w.r.t the given numbering $1,\dots,n$ of vertices.

→ standard lexigraphic ordering of strings (tuples) of $1,\dots,n$
$\langle 3,5,6,9,12\rangle$

with respect to.

```
I = ∅;
for j=1 to n do
    if (j is not adjacent to any vertex in I)
        I = I ∪ {j};
```



$I = \{1, 3, 5, \cancel{6}\}$
$\langle 1,3,5\rangle$

Finding maximum independent set is (generalized) NP-complete problems.

There is an MC circuit to find some maximal independent set if the numbering of vertices is immaterial?

Decision version

~~Decide if~~ Input: $G=(V,E)$, a specific numbering of V by $1,\dots,n$ and $v \in V$

Decide if v is in the lexigraphic first independent w.r.t the given numbering of the vertices.

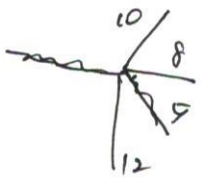
- Lexicographic maximal clique.

- Nearest TSP Heuristic

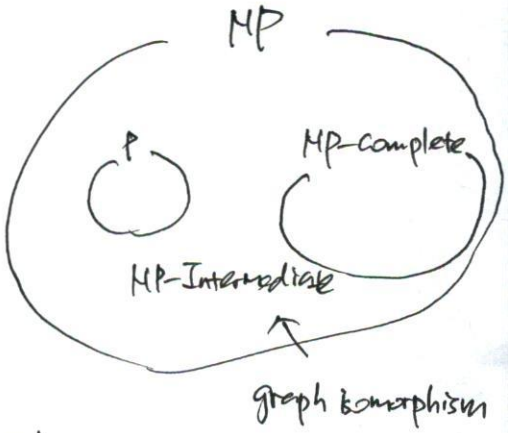
Input: A complete, undirected graph with positive integer edge weights, and two vertices s and t.

Decide if the nearest neighbor tour (does not find a real solution) starting at s visits t as the last node in the tour.

heuristic visit one of the nearest adjacent vertex from any node
└ not visited



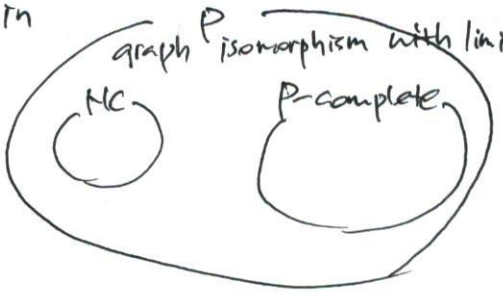
- Unification of two expressions
  used in theorem proving and PROLOG language.



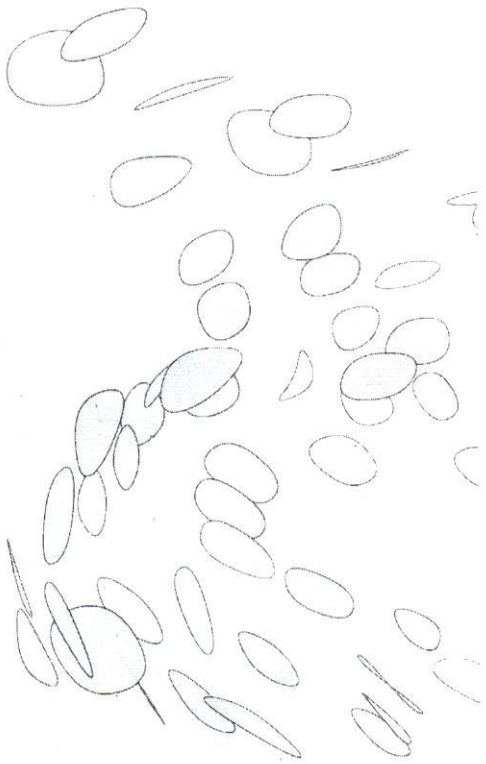
- Given a context-free grammar G and a string x,
  decide if $x \in L(G)$
  ---- $L(G) = \emptyset$

- Generalized Geograph Game restricted to acyclic graph

- An example of unknown suspected to be in
  P- (~~=~~NC ∪ P-complete)
  Graph Isomorphism limited to graphs s.t
  all vertex degrees $\leq k$ for a constant k



degree(v) = 5



DOI:10.1145/1536616.1536637

Satisfiability solvers can now be effectively deployed in practical applications.

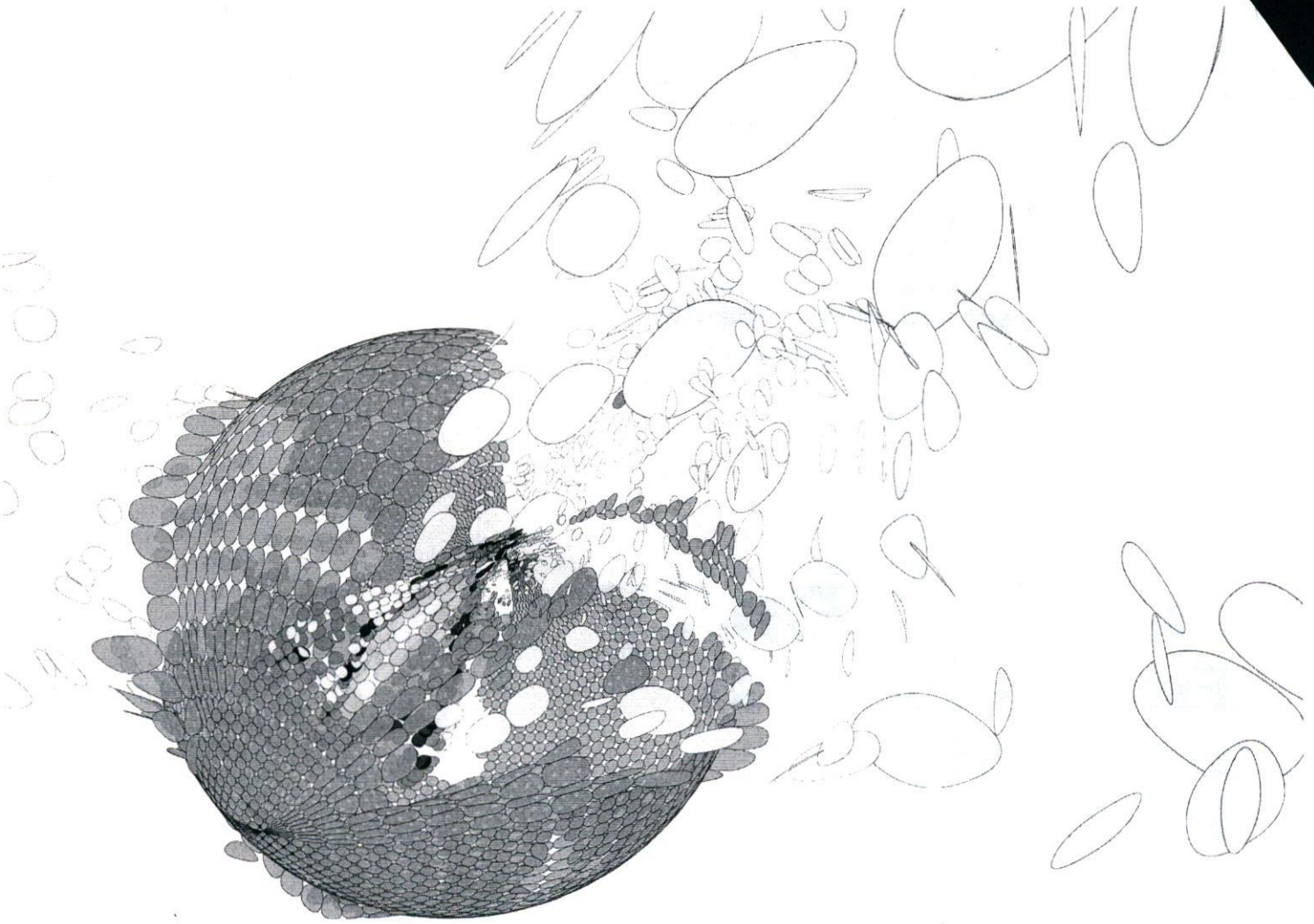
BY SHARAD MALIK AND LINTAO ZHANG

Boolean Satisfiability
From Theoretical Hardness to Practical Success

THERE ARE MANY practical situations where we need to satisfy several potentially conflicting constraints. Simple examples of this abound in daily life, for example, determining a schedule for a series of games that resolves the availability of players and venues, or finding a seating assignment at dinner consistent with various rules the host would like to impose. This also applies to applications in computing, for example, ensuring that a hardware/software system functions correctly with its overall behavior constrained by the behavior of its components and their composition, or finding a plan for a robot to reach a goal that is consistent with the moves it can make at any step. While the applications may seem varied, at the core they all have variables whose values we need to determine (for example, the person sitting at a given seat at dinner) and constraints that these variables must satisfy (for example, the host's seating rules).

In its simplest form, the variables are *Boolean* valued (true/false, often represented using 1/0) and *propositional logic formulas* can be used to express the constraints on the variables.[15] In propositional logic the *operators* AND, OR, and NOT (represented by the symbols $\wedge$, $\vee$, and $\neg$ respectively) are used to construct formulas with variables. If x is a Boolean variable and f, f_1 and f_2 are propositional logic formulas (subsequently referred to simply as formulas), then the following recursive definition describes how complex formulas are constructed and evaluated using the constants 0 and 1, the variables, and these operators.

- x is a formula that evaluates to 1 when x is 1, and evaluates to 0 when x is 0
- $\neg f$ is a formula that evaluates to 1 when f evaluates to 0, and 0 when f evaluates to 1
- $f_1 \wedge f_2$ is a formula that evaluates to 1 when f_1 and f_2 both evaluate to 1, and



evaluates to 0 if either f_1 or f_2 evaluate to 0

- $f_1 \vee f_2$ is a formula that evaluates to 0 when f_1 and f_2 both evaluate to 0, and evaluates to 1 if either f_1 or f_2 evaluate to 1

$(x_1 \vee \neg x_2) \wedge x_3$ is an example formula constructed using these rules. Given a valuation of the variables, these rules can be used to determine the valuation of the formula. For example: when ($x_1 = 0$, $x_2 = 0$, $x_3 = 1$), this formula evaluates to 1 and when ($x_3 = 0$), this formula evaluates to 0, regardless of the values of x_1 and x_2. This example also illustrates how the operators in the formula provide constraints on the variables. In this example, for this formula to be true (evaluate to 1), x_3 must be 1.

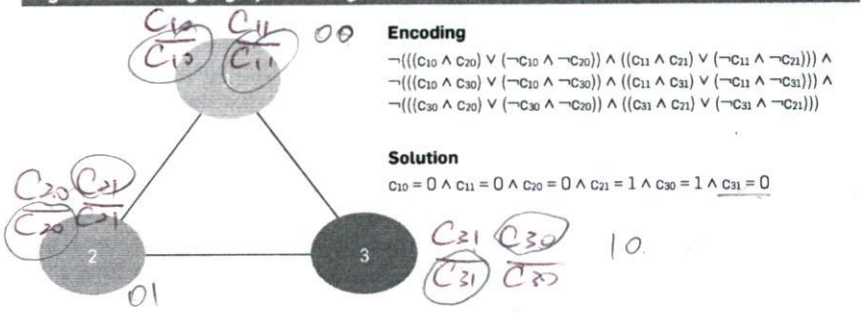
Boolean Satisfiability

A satisfying assignment for a formula is an assignment of the variables such that the formula evaluates to 1. It simultaneously satisfies the constraints imposed by all the operators in the formula. Such an assignment may not always exist. For example the formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ cannot be satisfied by any of the four possible assignments 0/0, 0/1, 1/0, 1/1 to x_1 and x_2. In this case the problem is overconstrained. This leads us to a definition of the Boolean Satisfiability problem (also referred to as Propositional Satisfiability or just Satisfiability, and abbreviated as SAT): *Given a formula, find a satisfying assignment or prove that none exists.* This is the constructive version of the problem, and one used in practice. A simpler decision version, often used on the theoretical side, just needs to determine if there exists a satisfying assignment for the formula (a yes/no answer). It is easy to see that a solver for the decision version of the problem can easily be used to construct a solution to the constructive version, by solving a series of n decision problems where n is the number of variables in a formula.

Many constraint satisfaction problems dealing with non-Boolean variables can be relatively easily translated to SAT. For example, consider an instance of the classic graph coloring problem where an n-vertex graph needs to be checked for 4-colorability, that is, determining whether each vertex can be colored using one of four possible colors such that no two adjacent vertices have the same color. In this case, the variables are the colors $\{c_0, c_1, c_2, \ldots, c_{n-1}\}$ for the n vertices, and the constraints are that adjacent vertices must have different colors. For this problem the variables are not Boolean and the constraints are not directly expressed with the operators $\{\wedge, \vee, \neg\}$. However, the variables and constraints can be encoded into a propositional formula as follows. Two Boolean variables, c_{i0}, c_{i1}, are used in a two-bit encoding of the four possible values of the color for vertex i. Let i and j be adjacent vertices.

ILLUSTRATION BY GWEN WAYNE

Figure 1. Encoding of graph coloring.



The constraint $c_i \neq c_j$ is then expressed as $\neg((c_{i0} == c_{j0}) \wedge (c_{i1} == c_{j1}))$, here $==$ represents equality and thus this condition checks that both bits in the encoding do not have the same value for i and j. Further, $(c_{i0} == c_{j0})$ can be expressed as $(c_{i0} \wedge c_{j0}) \vee (\neg c_{i0} \wedge \neg c_{j0})$, that is, they are both 1 or both 0. Similarly for $(c_{i1} == c_{j1})$. If we take the conjunction of the constraints on each edge, then the resulting formula is satisfiable if and only if the original graph coloring problem has a solution. Figure 1 illustrates an instance of the encoding of the graph coloring problem into a Boolean formula and its satisfying solution.

Encodings have been useful in translating problems from a wide range of domains to SAT, for example, scheduling basketball games,[40] planning in artificial intelligence,[20] validating software models,[17] routing field programmable gate arrays,[28] and synthesizing consistent network configurations.[29] This makes SAT solvers powerful engines for solving constraint satisfaction problems. However, SAT solvers are not always the best engines—there are many cases where specialized techniques work better for various constraint problems, including graph coloring (for example, Johnson et al.[19]). Nonetheless, it is often much easier and more efficient to use off-the-shelf SAT solvers than developing specialized tools from scratch.

One of the more prominent practical applications of SAT has been in the design and verification of digital circuits. Here, the translation to a formula is very straightforward. The functionality of digital circuits can be expressed as compositions of basic logic gates. A logic gate has Boolean input signals and produces Boolean output signals. The output of a gate can be used as an input to another gate. The functions of the basic logic gates are in direct correspondence to the operators $\{\wedge, \vee, \neg\}$. Thus various properties regarding the functionality of logic circuits can be easily translated to formulas. For example, checking that the values of two signals s_1 and s_2 in the logic circuit are always the same is equivalent to checking that their corresponding formulas f_1 and f_2 never differ, that is, $(f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f_2)$ is not satisfiable.

This technique can be extended to handle more complex properties involving values on sequences of signals, for example, a request is eventually acknowledged. For such problems, techniques that deal with temporal properties of the system, such as model checking, are used.[6] Modern SAT solvers have also been successfully applied for such tasks.[3,26] One of the main difficulties of applying SAT in checking such properties is to find a way to express the concept of "*eventually*." In theory, there is no tractable way to express this using propositional logic. However, in practice it is often good enough to just set a *bound* on the number of steps. For example, instead of asking whether a response to a request will eventually occur, we ask whether there will be a response within k clock cycles, where k is a small fixed number. Similar techniques have also been used in AI planning,[20] for example, instead of determining if a goal is reachable, we ask whether we can reach the goal in k steps. This unrolling technique has been widely adopted in practice, since we often only care about the behavior of the system within a small bounded number of steps.

Theoretical Hardness: SAT and NP-Completeness

The decision version of SAT, that is, determining if a given formula has a satisfying solution, belongs to the class of problems known as *NP-complete*.[8,12] An instance of any one of these problems can be relatively easily transformed into an instance of another. For example, both graph coloring and SAT are NP-complete, and earlier we described how to transform a graph coloring instance to a SAT instance.

All currently known solutions for NP-Complete problems, in the worst case, require runtime that grows exponentially with the size of the instance. Whether there exist subexponential solutions to NP-Complete problems is arguably the most famous open question in computer science.[a] Although there is no definitive conclusion, the answer is widely believed to be in the negative. This exponential growth in time complexity indicates the difficulty of scaling solutions to larger instances.

However, an important part of this characterization is "worst case." This holds out some hope for the "typical case," and more importantly the typical case that might arise in specific problem domains. *In fact, it is exactly the non-adversarial nature of practical instances that is exploited by SAT solvers.*

Solving SAT

Most SAT solvers work with a restricted representation of formulas in conjunctive normal form (CNF), defined as follows. A literal l is either a positive or a negative occurrence of a variable (for example, x or $\neg x$). A clause, c, is the OR of a set of literals, such as $(l_1 \vee l_2 \vee l_3 \ldots \vee l_n)$. A CNF formula is the AND of a set of clauses, such as $(c_1 \wedge c_2 \wedge c_3 \wedge c_m)$. An example CNF formula is:

$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_4)$$

The restriction to CNF is an active choice made by SAT solvers as it enables their underlying algorithms. Further, this is not a limitation in terms of the formulas that can be handled. Indeed, with the addition of new auxiliary variables; it is easy to translate any formula into CNF with only a linear increase in size.[36] However, this representation is not used exclusively and there has been recent success with

a http://www.claymath.org/millennium/.

HW: Question related to this polynomial Alg.

solvers for non-clausal representations (for example, NFLSAT[18]).

Most practically successful SAT solvers are based on an approach called systematic search. Figure 2 depicts the search space of a formula. The search space is a tree with each vertex representing a variable and the out edges representing the two decision choices for this variable. For a formula with n variables, there are 2^n leaves in the tree. Each path from the root to a leaf corresponds to a possible assignment to the n variables. The formula may evaluate to 1 or 0 at a leaf (colored green and red respectively). Systematic search, as the name implies, systematically searches the tree and tries to find a green leaf or prove that none exists.

The NP-completeness of the problem indicates that we will likely need to visit an exponential number of vertices in the worst case. The only hope for a practical solver is that by being smart in the search, almost all of the tree can be pruned away and only a minuscule fraction is actually visited in most cases. For an instance with a million variables, which is considered within the reach of modern solvers, the tree has 2^{10^6} leaves, and in reasonable computation time (about a day), we may be able to visit a billion (about 2^{30}) vertices as part of the search—a numerically insignificant fraction of the tree size!

Most search-based SAT solvers are based on the so called *DPLL approach* proposed by Davis, Logemann, and Loveland in a seminal *Communications* paper published in 1962.[9] (This research builds on the work by Davis and Putnam[10] and thus Putnam is often given shared credit for it.). Given a CNF formula, the DPLL algorithm first heuristically chooses an unassigned variable and assigns it a value: either 1 or 0. This is called branching or the *decision* step. The solver then tries to deduce the consequences of the variable assignment using deduction rules. The most widely used deduction rule is the *unit-clause* rule, which states that if a clause in the formula has all but one of its literals assigned 0 and the remaining one is unassigned, then the only way for the clause to evaluate to true, and thus the formula to evaluate to true, is for this last unassigned literal to be assigned to 1. Such clauses are called unit clauses and the forced assignments are called implications. This rule is applied iteratively until no unit clause exists. Note that this deduction is enabled by the CNF representation and is the main reason for SAT solvers preferring this form.

If at some point there is a clause in the formula with all of its literals evaluating to 0, then the formula cannot be true under the current assignment. This is called a *conflict* and this clause is referred to as a conflicting clause. A conflict indicates that some of the earlier decision choices cannot lead to a satisfying solution and the solver has to backtrack and try a different branch value. It accomplishes this by finding the most recent decision variable for which both branches have not been taken, flip its value, undo all variable assignments after that decision, and run the deduction process again. Otherwise, if no such conflicting clause exists, the solver continues by branching on another unassigned variable. The search stops either when all variables are assigned a value, in which case we have hit a green leaf and the formula is satisfiable, or when a conflicting clause exists when all branches have been explored, in which case the formula is unsatisfiable.

Consider the application of the algorithm to the formula shown in Figure 2. At the beginning the solver branches on variable x_1 with value 1. After branching, the first clause becomes unit and the remaining free literal $\neg x_2$ is implied to 1, which means x_2 must be 0. Now the second clause becomes unit and $\neg x_3$ is implied to 1. Then $\neg x_4$ is implied to 1 due to the third clause. At this point the formula is satisfied, and the satisfying assignment corresponds to the 8th leaf node from the left in the search tree. (This path is marked in bold in the figure.) As we can see, by applying the unit-clause rule, a single branching leads directly to the satisfying solution.

Many significant improvements in the basic DPLL algorithm have been

Figure 2. Search space of a formula.

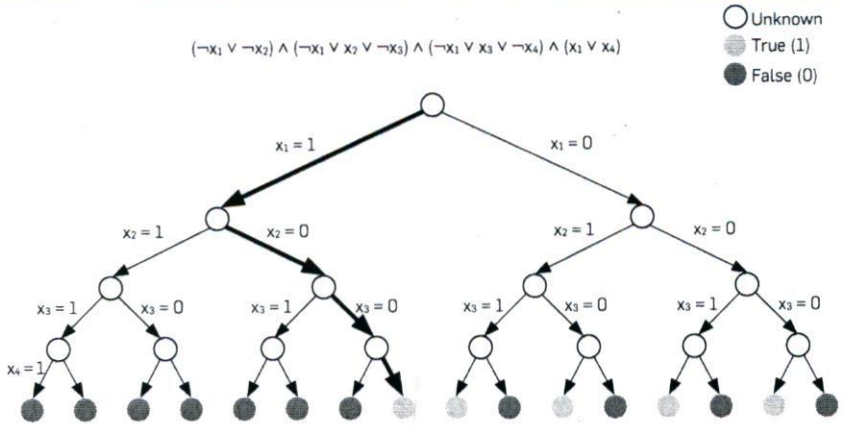
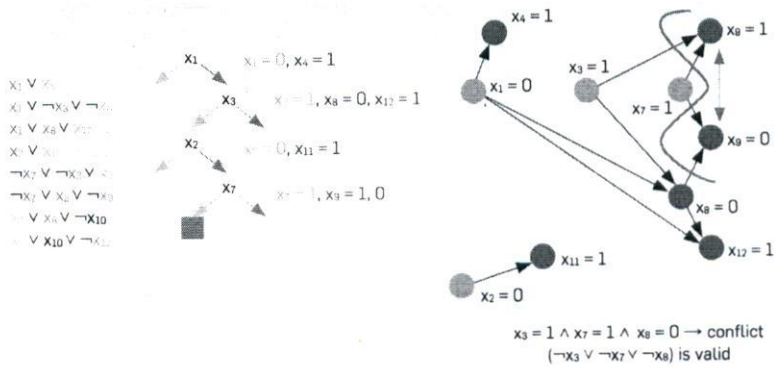


Figure 3. Conflict-driven learning and non-chronological backtracking.



proposed over the years. In particular, a technique called conflict-driven learning and non-chronological backtracking[2, 24] has greatly enhanced the power of DPLL SAT solvers on problem instances arising from real applications, and has become a key element of modern SAT solvers. The technique is illustrated in Figure 3. The column on the left lists the clauses in the example formula. The colors of the literals show the current assignments during the search (red representing 0, green 1, and black representing unassigned). The middle graph shows the branching and implications at the current point in the search. At each vertex the branching assignment is shown in blue and the implications in gray. The first branching is on x_1, and it implies x_4=1 (because of the first clause), the second branching is on x_3, and it implies x_8=0 and x_{12}=1 and so on. The right graph shows the implication relationships between variables. For example, x_4=1 is implied because of x_1=0, so there is a directed edge from node x_1=0 to node x_4=1. x_8=0 is implied because of both x_1=0 and x_3=1 (the red literals in the second clause), therefore, these nodes have edges leading to x_8=0.

After branching on x_7 and implying x_9=1 because of the 5th clause, we find that the 6th clause becomes a conflicting clause and the solver has to backtrack. Instead of flipping the last decision variable x_7 and trying x_7=0, we can learn some information from the conflict. From the implication graph, we see that there is a conflict because x_9 is implied to be both 1 and 0. If we consider a cut (shown as the orange line) separating the conflicting implications from the branching decisions, we know that once the assignments corresponding to the cut edges are made, we will end up with a conflict, since no further decisions are made. Thus, the edges that cross the cut are, in some sense, responsible for the conflict. In the example, x_3, x_7, and x_8 have edges cross the cut, thus the combination of x_3=1, x_7=1, and x_8=0 results in the conflict. We can learn from this and ensure that this assignment combination is not tried in the future. This is accomplished by recording the condition ($\neg x_3 \vee \neg x_7 \vee x_8$). This clause, referred to as a learned clause, can be added to the formula. While it is redundant in the sense that it is implied by the formula, it is nonetheless useful as it prevents search from ever making the assignment (x_3=1, x_7=1, x_8=0) again.

Further, because of this learned clause, x_7 = 1 is now implied after the second decision, and we can backtrack to this earlier decision level as the choice of x_2 = 0 is irrelevant to the current conflict. Since such backtracking skips branches, it is called non-chronological backtracking and helps prune away unsatisfiable parts of the search space.

Recent Results

Recent work has exposed several significant areas of improvement now integral to modern SAT solvers.[22] The first deals with efficient implementation of the unit-clause rule using a technique called two-literal watching. The second area relates to improvements in the branching step by focusing on exhausting local sub-spaces before moving to new spaces. This is accomplished by placing increased emphasis on variables present in recently added conflict clauses. Another commonly used technique is random restart,[13] which periodically restarts the search while retaining the learned clauses from the current search to avoid being stuck in a search sub-space for too long. Other recent directions include formula preprocessing for clause and variable elimination,[11] considering algorithm portfolios that use empirical hardness models to choose among their constituent solvers on a per-instance basis[39] and using learning techniques to adjust parameters of heuristics.[16] With the advent of multicore processing, there is emerging interest in efficient multi-core implementations of parallel SAT solvers.[14]

The original Davis Putnam algorithm[10] based on resolution is often regarded as the first algorithm for SAT and has great theoretical and historical significance. However, this algorithm suffers from a space growth problem that makes it impractical. Reduced Ordered Binary Decision Diagrams (ROBDDs)[5] are a canonical representation of logic functions, that is, each function has a unique representation for a fixed variable ordering. Thus, ROBDDS can be used directly for SAT. However, ROBDDs also face space limitations with increasing instance size. Stålmarck's algorithm[35] uses breadth-first search instead of depth-first search as in DPLL, and has been shown to be practically useful. Its performance relative to DPLL based solvers is unclear as public versions of efficient implementations of Stålmarck's algorithm are not available due to its proprietary nature.

When represented in CNF, SAT can be regarded as a discrete optimization problem with the objective to maximize the number of satisfied clauses. If this max value is equal to the total number of clauses, then the instance is satisfiable. Many discrete optimization techniques have been explored in the SAT context, including simulated annealing,[33] tabu search,[25] neural networks,[34] and genetic algorithms.[23]

Figure 4. Speedup of SAT solvers in recent years.