

# Osnovni principi logičke sinteze (drugi deo)

---

Stefan Nikolić

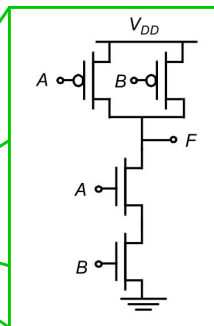
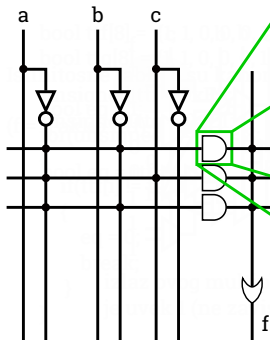
Departman za matematiku i informatiku  
Prirodno-matematički fakultet, Novi Sad

30.10.2024.

# Sa prošlog časa: minimizacija funkcija u dvostepenoj logici

$$f = a'b'c' + a'b'c + ab'c + abc' + abc = ab + ac + a'b'$$

abc	f
000	1
001	1
010	0
011	0
100	0
101	1
110	1
111	1



$$t_d = RC \ln 2$$

$$R \propto \#ins$$

U praksi retko imamo kapije sa više od 4 ulaza  $\implies$  i dvostepena logika je zapravo višestepena

Bolje rezultate možemo postići direktnom manipulacijom višestepenim mrežama

I-NE grafovi (eng. AND-INVERTER GRAPHS (AIGs))

---

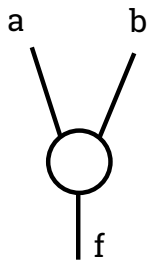
# ALG: definicija

ALG je usmereni aciklični graf (eng. **DIRECTED ACYCLIC GRAPH** (DAG)) sa tri tipa čvorova:

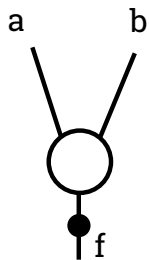
- primarni ulaz (eng. **PRIMARY INPUT** (PI)): ima ulazni stepen nula i predstavlja ulaz u mrežu ili konstante 0 i 1
- unutrašnji čvor (eng. ALG node): ima ulazni stepen dva i vrši logičko I nad svoja dva ulaza
- primarni izlaz (eng. **PRIMARY OUTPUT** (PO)): ima ulazni stepen jedan i izlazni stepen nula i predstavlja izlaz iz mreže

Grane grafa po potrebi mogu biti invertovane

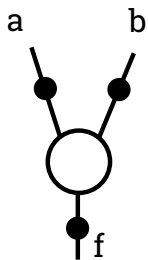
# Primeri jednostavnih AIG-a



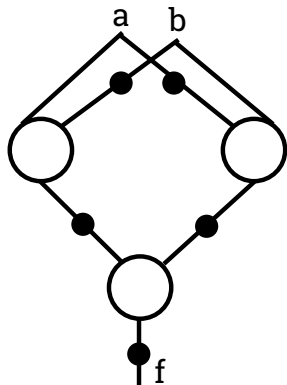
$$f = ab$$



$$f = (ab)'$$

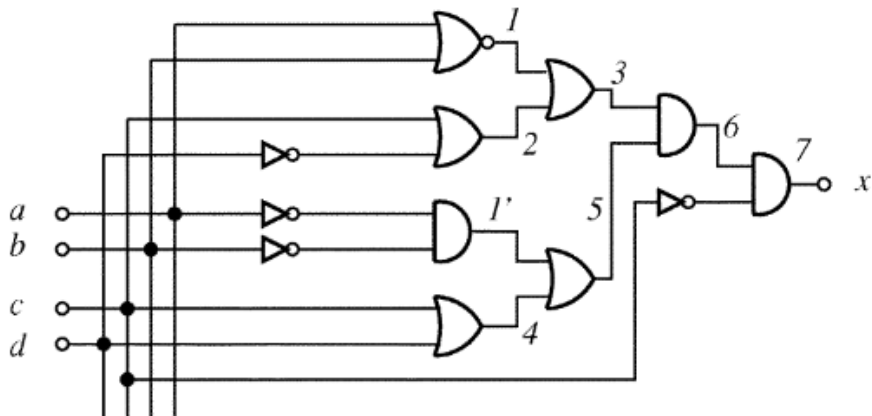


$$f = (a'b')' = a + b$$

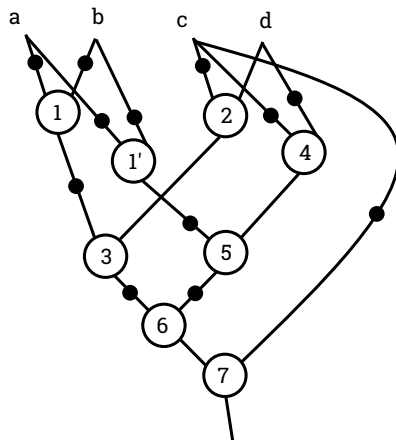
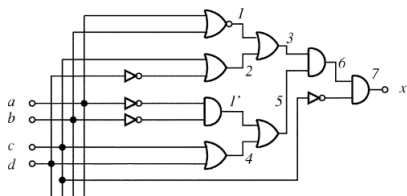


$$f = ab' + a'b = a \oplus b$$

Kako bi AIG izgledao u ovom slučaju?



## Topološki obilazimo ulaznu mrežu i generišemo odgovarajuće I kapije





Kombinaciona provera ekvivalencije (eng.

COMBINATIONAL EQUIVALENCE CHECKING (CEC))

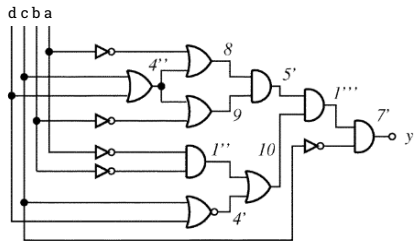
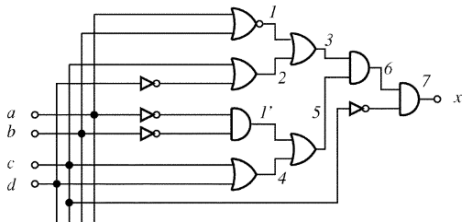
---

Pretpostavimo da smo izvršili neku transformaciju funkcije  $x$  i dobili funkciju  $y$

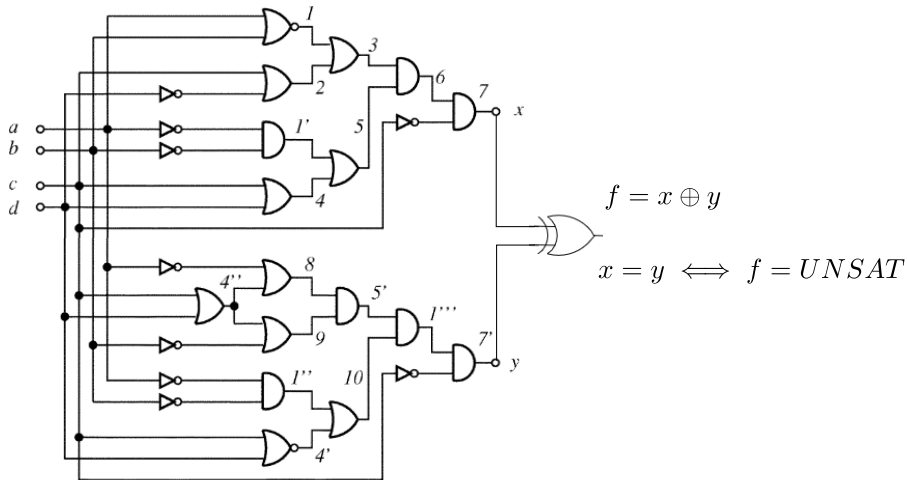
Kako da budemo sigurni da je transformacija bila ispravna i da je  $y = x$ ?

Na raspolaganju nam je SAT rešavač koji za datu funkciju  $f$  predstavljenu u CNF formi može da nam kaže da li je zadovoljiva ili ne (da li postoji dodela vrednosti promenljivama za koju je  $f = 1$ )

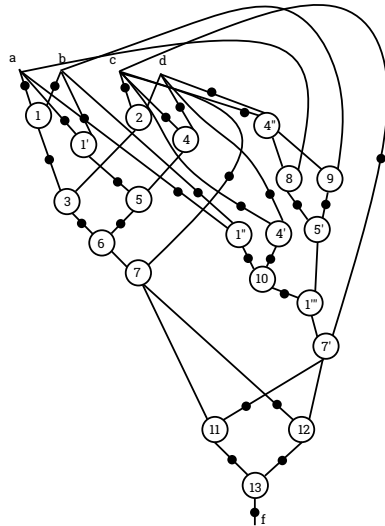
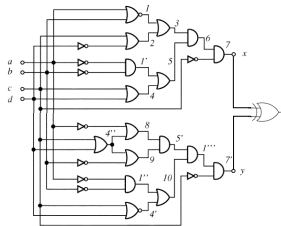
# Primer: da li je $x = y$ ?



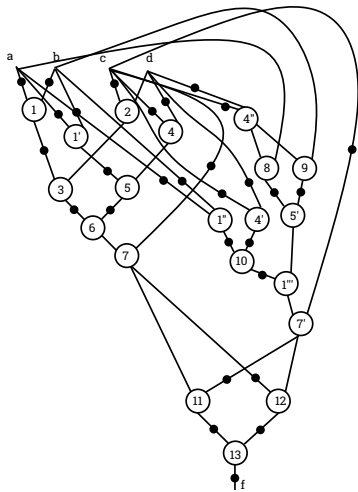
## Majter (eng. MITER): ulazi kratko spojeni, EKSILI na izlazima



# AIG Miter



# Pretvaranje u CNF



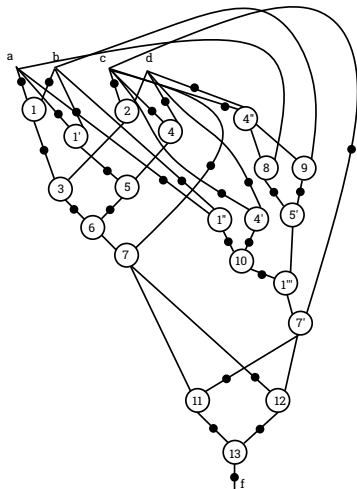
$$7 \iff 6 \wedge c'$$

$$(7 \implies (6 \wedge c'))((6 \wedge c') \implies 7)$$

$$(7' \vee (6 \wedge c'))((6 \wedge c')' \vee 7)$$

$$(7' \vee 6)(7' \vee c')(6' \vee c \vee 7)$$

# Pretvaranje u CNF



$$7 \iff 6 \wedge c'$$

$$(7 \implies (6 \wedge c'))((6 \wedge c') \implies 7)$$

$$(7' \vee (6 \wedge c'))((6 \wedge c')' \vee 7)$$

$$(7' \vee 6)(7' \vee c')(6' \vee c \vee 7)$$

U opštem slučaju:

$$y = ab: (a' + b' + y)(a + y')(b + y')$$

$$y = a': (a' + y')(a + y)$$

## Cejtinova (ru. Цейтин, eng. Tseytin) transformacija

Formiramo proizvod formula nezavisno formulisanih za svaki čvor (izlaz svakog čvora direktno zavisi samo od njegovih ulaza)

Skup zadovoljavajućih dodela vrednosti promenljivama tako dobijene formule je identičan skupu ulaznih vektora za koje mreža na uzlazu daje 1

Broj promenljivih odgovara broju čvorova,  $n$ , a broj činilaca (eng. **CLAUSE**) je otprilike  $3 \times n$  (za **AI**G; transformacija važi za proizvoljnu mrežu)



## Prednosti AIG-a

Ovde vidimo prvu prednost AIG-a: sve čvorove, osim eventualno nekih PO gde nam može biti potreban invertor, transformišemo prostom zamenom odgovarajućih ulaza u Cejtinovu transformaciju dvoulazne I-kapije

Uniformnost omogućuje značajnu optimizaciju implementacije struktur podataka (na primer, alokacija memorije)

Još neke prednosti ćemo videti kasnije

# SAT rešavači u tri rečenice

SAT rešavači koriste princip sličan onom koji smo koristili za enumeraciju kombinacija implikanti u algoritmu Kvajn-Meklaskog (detalji slede kasnije)

U najgorem slučaju moraju da posete svih  $2^n$  mogućih dodela

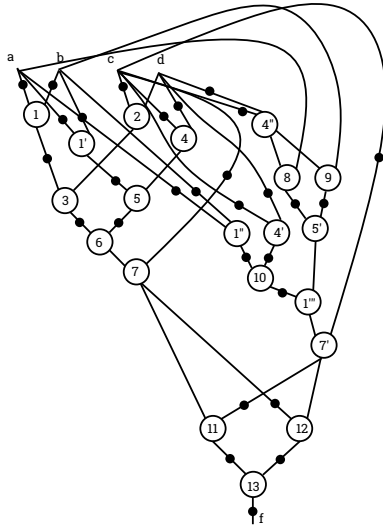
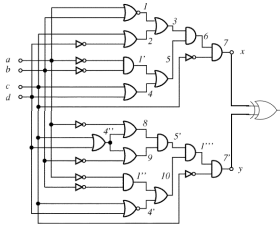
Prema tome, veoma je važno da majter bude što manji

# Sinergija sinteze i verifikacije

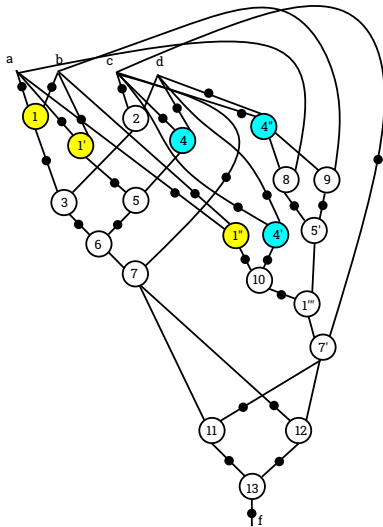
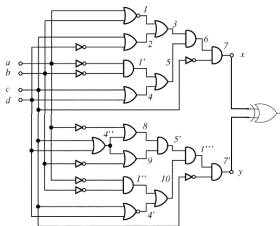
Glavni zadatak logičke sinteze je minimizacija kola, a upravo to je bitno i za uspešnu verifikaciju

⇒ mnoge ideje iz verifikacije su našle primenu i u sintezi i obrnuto

# Kako možemo da minimizujemo ovaj majter?



# Možemo da izbacimo duplikate



Možda deluje besmisleno, ali...



## Kada su dva čvora duplikati?

$$f_1 = l'_1 \wedge l_1''$$

$$f_2 = l'_2 \wedge l_2''$$

$l'_1, l_1'', l'_2, l_2''$  su literali (PI ili izlazi drugih unutrašnjih čvorova u oba polariteta)

## Kada su dva čvora duplikati?

Dva čvora su duplikati akko su im neuređeni skupovi ulaznih literala identični (logičko I je komutativno)

Kako poredimo neuređene skupove?

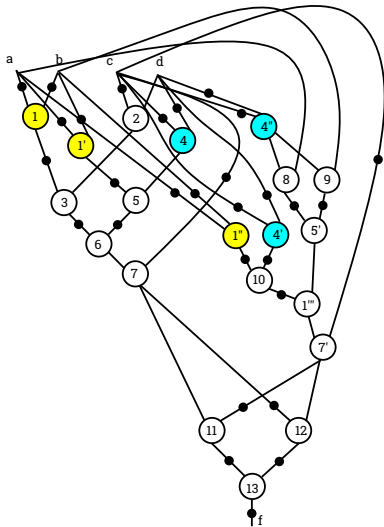
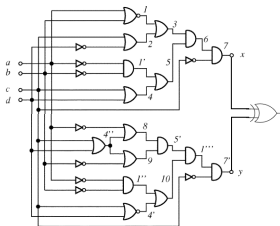


## Kada su dva čvora duplikati?

Dva čvora su duplikati akko su im neuređeni skupovi ulaznih literala identični (logičko I je komutativno)

Uredimo skup svih literala, zatim uredimo oba skupa koja poredimo u istom poretku i poredimo ih kao uređene skupove

# Kako da izbacimo duplikate?



# Naivno izbacivanje duplikata

```
for u in V
  for v in V \ {u}
    if v == dup(u)
      izbaci v; preveži sve sledbenike v na u
```

Koliko ovde ima poređenja?

# Naivno izbacivanje duplikata

```
for u in V
  for v in V \ {u}
    if v == dup(u)
      izbacij v; preveži sve sledbenike v na u
```

Broj poređenja kvadratno zavisi od veličine redundantnog grafa (lako može imati milione čvorova)

## Konstante su nam bitne; kako da ih smanjimo?

```
for u in V
  for v in V \ {u}
    if v == dup(u)
      izbaci v; preveži sve sledbenike v na u
```

Setimo se enumeracije kombinacija implikanti sa prošlog časa

# Graf svakako gradimo inkrementalno

Umesto da na kraju izbacujemo duplikate, pri svakom dodavanju proveravamo da li je čvor već dodat i ako jeste, ne dodajemo ga

# Strukturno hešovanje (eng. STRUCTURAL HASHING, strashing)

```
Algorithm create_and2( $p_1, p_2$ ) {  
  /* constant folding */  
  if ( $p_1 == \text{CONST\_0}$ ) return CONST_0;  
  if ( $p_2 == \text{CONST\_0}$ ) return CONST_0;  
  if ( $p_1 == \text{CONST\_1}$ ) return  $p_2$ ;  
  if ( $p_2 == \text{CONST\_1}$ ) return  $p_1$ ;  
  if ( $p_1 == p_2$ ) return  $p_1$ ;  
  if ( $p_1 == \neg p_2$ ) return CONST_0;  
  
  /* rank order inputs to catch commutativity */  
  if ( $\text{rank}(p_1) > \text{rank}(p_2)$ ) swap( $p_1, p_2$ );  
  
  /* check for isomorphic entry in hash table */  
  if ( $(p = \text{hash\_lookup}(p_1, p_2)) == \text{NULL}$ )  
     $p = \text{new\_and\_vertex}(p_1, p_2)$ ;  
  return  $p$ ;  
}
```

Fig. 1. Algorithm **create\_and2** for the AND constructor.

Sličan pristup kao i kod algoritma Kvajna i Meklaskog: čim imamo priliku da nešto uprostimo, to i uradimo



IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 21, NO. 12, DECEMBER 2002

1377

## Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification

Andreas Kuehlmann, *Senior Member, IEEE*, Viresh Paruthi, Florian Krohm, and Malay K. Ganai, *Member, IEEE*

# Ali je danas standard i u logičkoj sintezi

## FRAIGs: A Unifying Representation for Logic Synthesis and Verification

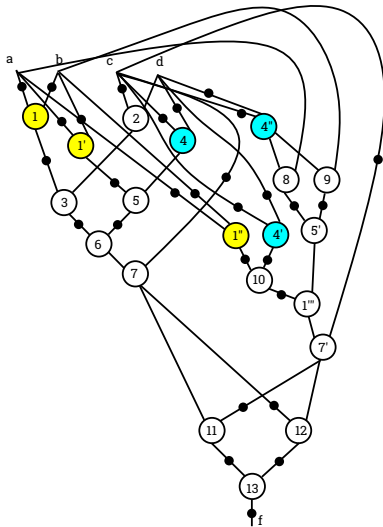
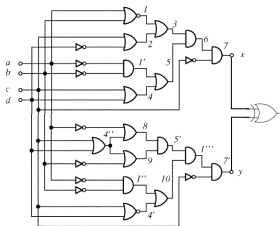
Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, Robert Brayton

Department of EECS, University of California, Berkeley  
{alanmi, satrajit, jiejiang, brayton}@eecs.berkeley.edu

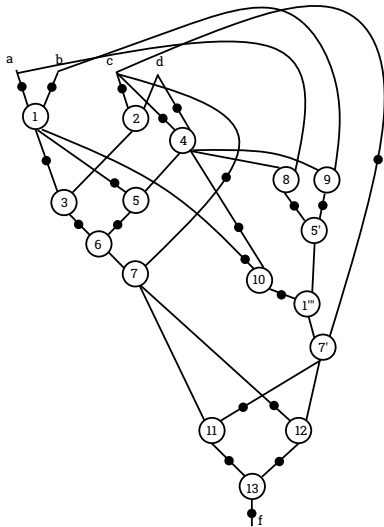
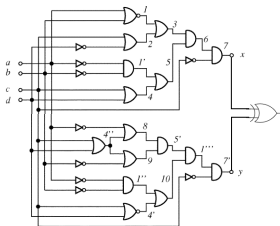
Table 1. Node count after strashing and FRAIGing.

Name	ins	outs	ff-lits	<i>fraig -n</i>	<i>fraig -r</i>	<i>fraig</i>
des	256	245	6084	5530	3895	3876
c1355	41	32	992	550	537	537
c6288	32	32	4675	2354	2339	2336
i10	257	224	4355	2869	2436	2274
s15850	611	684	7303	4344	4020	3884
pj1	1769	1063	34533	18744	16834	16471
b14	32	54	17388	9419	6300	5855
b17	37	97	57311	32422	28916	27907
Ratio				100.0	85.6	82.8

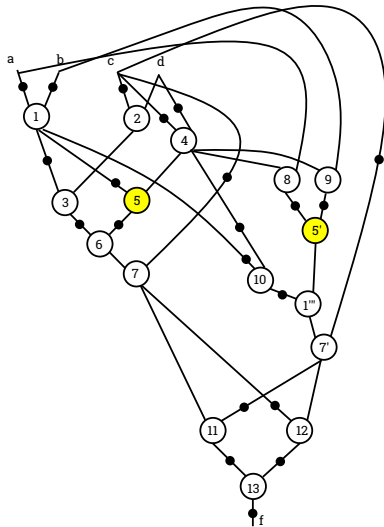
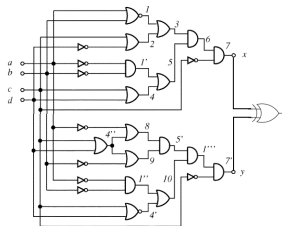
# Naš majter nakon strešovanja



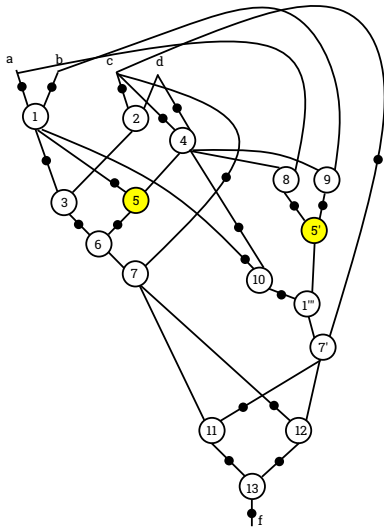
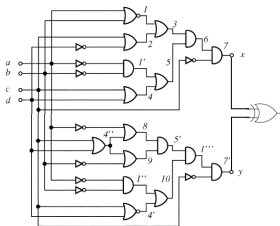
# Naš majter nakon strešovanja



Još uvek postoje čvorovi sa istom funkcijom ( $5 = 5' = a'b' + c + d$ )



# Kako da osiguramo da ih nema?



## Koristeći prethodne zaključke

Prilikom dodavanja svakog čvora, koristimo SAT rešavač da vidimo da li je čvor sa istom funkcijom već prisutan i ako jeste, ne dodajemo novi

Šta da radimo ako je poziv SAT rešavača suviše skup? (a jeste)



Šta bismo mogli da uradimo ako bismo imali na raspolaganju istinitosne  
tablice funkcija svih čvorova?

# Istinitosne tablice možemo konstruisati za do 16 promenljivih

Ključno zapažanje: ako znamo  $m$  unosa na istim pozicijama istinitosnih tablica funkcija  $f$  i  $g$ , ako se bar jedan od tih unosa razlikuje, funkcije  $f$  i  $g$  ne mogu biti iste  $\implies$  nema potrebe da pozivamo SAT rešavač

## Kako tražimo dati unos u istinitosne tablice?

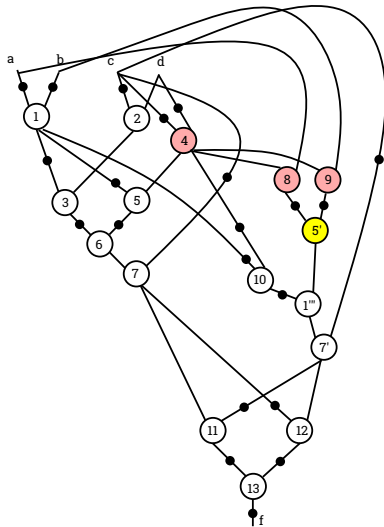
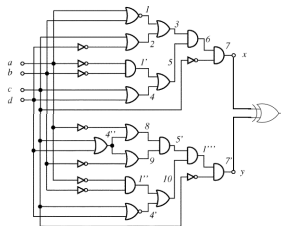
Dve definicije:

fenin čvora  $u$  (eng. **FANIN**( $u$ ),  $FI(u)$ ) = skup direktnih prethodnika čvora  $u$  (8 i 9 za 5')

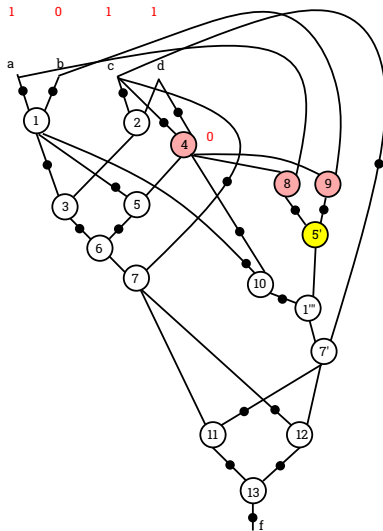
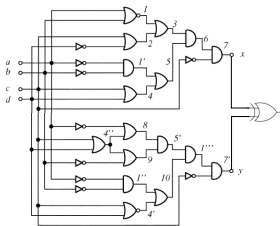
tranzitivni fenin čvora  $u$  (eng. **TRANSITIVE FANIN**,  $TFI(u)$ ) = skup svih čvorova iz kojih je moguće doći do čvora  $u$

Slično definišemo i (tranzitivni) fenaut za sledbenike

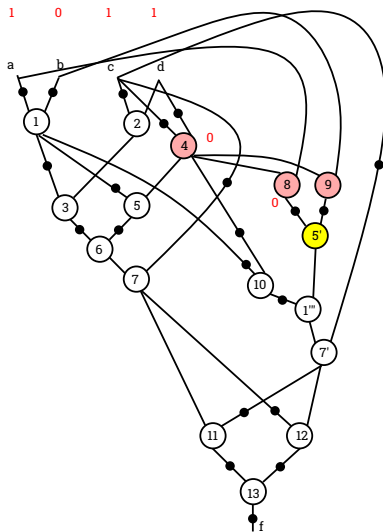
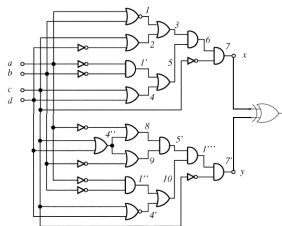
# TFI čvora 5'



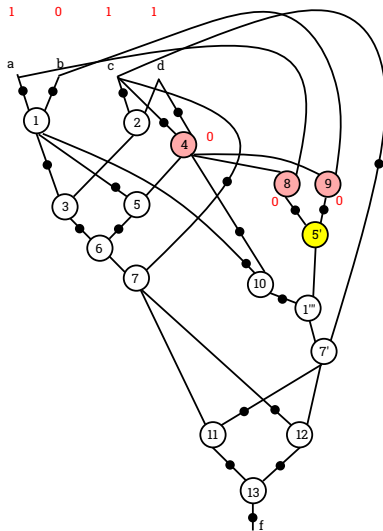
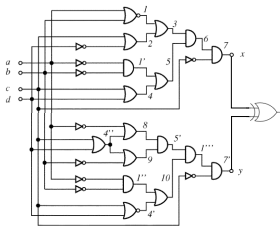
# Obilazimo TFI u topološkom poretku i računamo izlaz svakog čvora



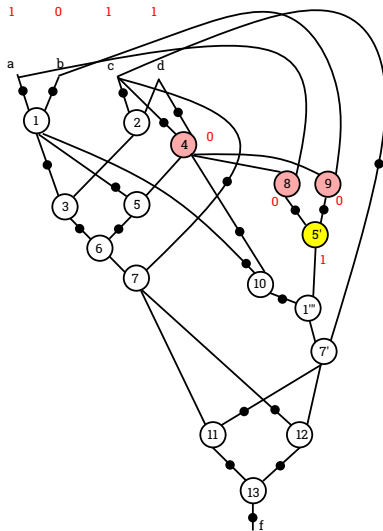
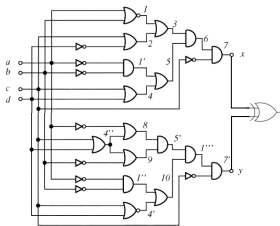
# Obilazimo TFI u topološkom poretku i računamo izlaz svakog čvora



# Obilazimo TFI u topološkom poretku i računamo izlaz svakog čvora

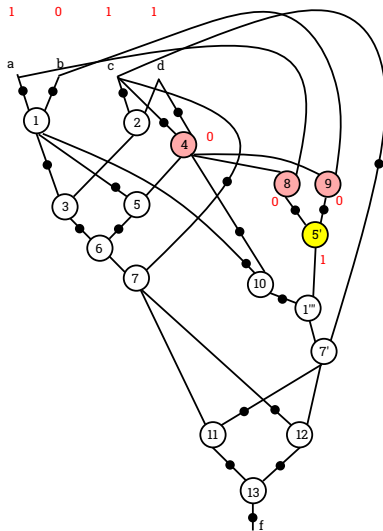
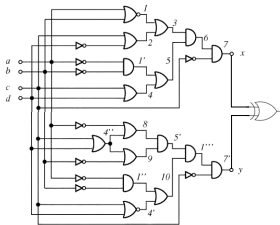


# Obilazimo TFI u topološkom poretku i računamo izlaz svakog čvora





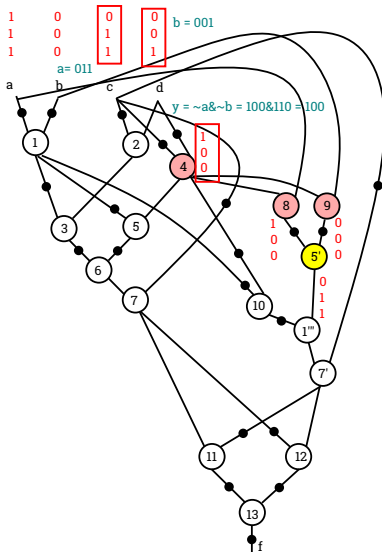
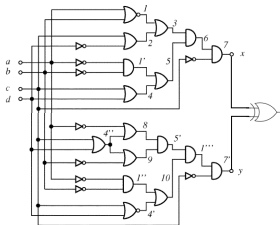
# Ako tražimo više unosa, kako da ubrzamo račun?



# Ponovo operacije nad bitovima

`y = a & b` //svaka promenljiva je 64 bita  $\implies$  možemo paralelno simulirati  
64 vektora

## Bit-paralelna simulacija



Ako unapred odaberemo vektore koje ćemo koristiti, možemo čuvati rezultate simulacije za svaki čvor

Pošto čvorove dodajemo u topološkom poretku, svi ulazi dodavanog čvora  $u$  će već biti dostupni  $\implies$  simulaciju dobijamo pomoću jedne instrukcije za svaka 64 vektora

Rezultate simulacije možemo koristiti i kao ključ za novu heš tabelu

Ako rezultat simulacije čvora  $u$  ne postoji u heš tabeli, onda je on klasa za sebe i ne moramo pozivati SAT rešavač

Ako postoji, iz heš tabele izvlačimo skup svih čvorova sa datim rezultatom simulacije i nad njima pozivamo SAT rešavač

# FRAIG (functionally-reduced AIG) algoritam Miščenka i ostalih

```
Aig_Node * OperationAnd( Aig_Man * p, Aig_Node * n1, Aig_Node * n2 )
{
    Aig_Node * res, * cand, * temp; Aig_NodeArray * class;
    /** trivial cases ***/
    if ( n1 == n2 ) return n1;
    if ( n1 == NOT(n2) ) return 0;
    if ( n1 == const ) return 0 or n2;
    if ( n2 == const ) return 0 or n1;
    if ( n1 < n2 ) { /** swap the arguments ***/
        temp = n1; n1 = n2; n2 = temp;
    }
    /** one level structural hashing ***/
    res = HashTableLookup( p->pTableStructure, n1, n2 );
    if ( res ) return res;
    res = CreateNode( p, n1, n2 );
    HashTableAdd( p->pTableStructure, res );
    if ( p->FlagUseOneLevelHashing ) return res;
    /** functional reduction ***/
    class = HashTableLookup( p->pTableSimulation, n1, n2 );
    if ( class == NULL ) {
        class = CreateNewSimulationClass( res );
        HashTableAdd( p->pTableSimulation, class ); return res;
    }
    for each node cand in class
        if ( CheckFunctionalEquivalence( cand, res ) ) {
            AddNodeToEquivalenceClass( class, res ); return cand;
        }
    AddNodeToSimulationClass( class, res ); return res;
}
```

## FRAIGs: A Unifying Representation for Logic Synthesis and Verification

Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, Robert Brayton

Department of EECS, University of California, Berkeley  
{alanmi, satrajit, jiejiang, brayton}@eecs.berkeley.edu

**Table 1. Node count after strashing and FRAIGing.**

Name	ins	outs	ff-lits	<i>fraig -n</i>	<i>fraig -r</i>	<i>fraig</i>
des	256	245	6084	5530	3895	3876
c1355	41	32	992	550	537	537
c6288	32	32	4675	2354	2339	2336
i10	257	224	4355	2869	2436	2274
s15850	611	684	7303	4344	4020	3884
pj1	1769	1063	34533	18744	16834	16471
b14	32	54	17388	9419	6300	5855
b17	37	97	57311	32422	28916	27907
Ratio				100.0	85.6	82.8

# Neki zaključci

Naizgled trivijalne optimizacije često donose najviše koristi (najgore je ništa ne optimizovati)

Svaki sledeći procenat uštede je sve teže dobiti

Pažljiva implementacija čestih operacija i dobar izbor strukture podataka su ključ uspeha

Minimizacija skupih operacija je moguća uz adekvatno filtriranje

Adekvatno keširanje prethodnih rezultata je takođe jako bitno



# Glavni princip moderne optimizacije logičkih kola

---

## **DAG-Aware AIG Rewriting** **A Fresh Look at Combinational Logic Synthesis**

**Alan Mishchenko**

Department of EECS  
University of California, Berkeley  
Berkeley, CA 94720

alanmi@eecs.berkeley.edu

**Satrajit Chatterjee**

Department of EECS  
University of California, Berkeley  
Berkeley, CA 94720

satrajit@eecs.berkeley.edu

**Robert Brayton**

Department of EECS  
University of California, Berkeley  
Berkeley, CA 94720

brayton@eecs.berkeley.edu

# Osnovna ideja

1. Podelimo AIG na podgrafove
2. Obilazimo jedan po jedan podgraf
3. Zamenimo ga najboljim podgrafom koji imamo na raspolaganju, tako da funkcionalnost bude zadržana
4. Iteriramo dok imamo vremena/dok postoji napredak

Kako bismo mogli da podelimo AIG?

# Enumeracija preseka

Presek (eng. **CUT**) DAG-a sa korenom  $u$  čvoru  $u$  je skup čvorova  $L$  takvih da svaka putanja od nekog PI do  $u$  prolazi kroz neki element skupa  $L$  i kroz svaki element skupa  $L$  prolazi barem jedna putanja od nekog PI do  $u$  koja ne prolazi ni kroz jedan drugi element skupa  $L$

Elemente skupa  $L$  nazivamo listovima (eng. **LEAVES**)

Pokrivanje (eng. **COVER**) preseka je skup svih čvorova između  $L$  i  $u$ , bez  $L$

Zapremina preseka je kardinalnost njegovog pokrivanja

Trivijalni presek je sam čvor  $u$

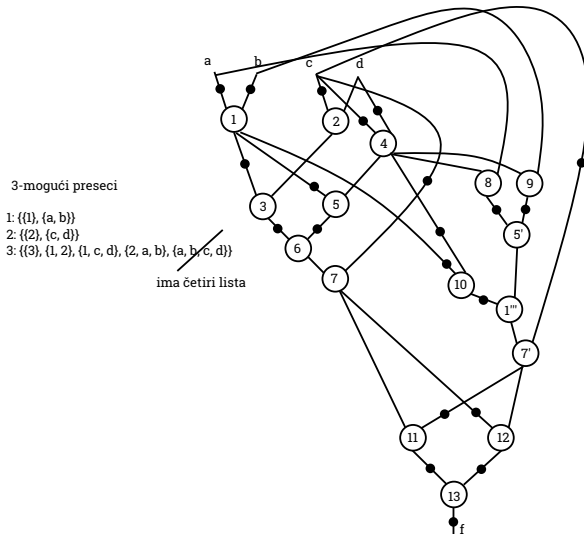
$k$ -moguć presek (eng.  $k$ -**FEASIBLE CUT**) je presek koji ima najviše  $k$  listova

# Enumeracija preseka

Zapažanje: Neka su  $C_1$  i  $C_2$  skupovi preseka direktnih predaka čvora  $u$ . Ako izvršimo Dekartov proizvod  $C_1$  i  $C_2$ , a zatim svaki par u proizvodu zamenimo unijom njihovih elemenata, dobićemo skup svih preseka čvora  $u$  osim trivijalnog

$\implies$  preseke generišemo u topološkom poretku

# Primer



## Kako bismo mogli da podelimo AIG?

Graf obilazimo u topološkom poretku i za svaki čvor izlistavamo  $k$ -moguće preseke, gde je  $k$  odabrano tako da možemo lako da vršimo poređenja i zamene (više o tome uskoro)



# Osnovna ideja

1. Podelimo AIG na podgrafove
2. Obilazimo jedan po jedan podgraf
3. Zamenimo ga najboljim podgrafom koji imamo na raspolaganju, tako da funkcionalnost bude zadržana
4. Iteriramo dok imamo vremena/dok postoji napredak

# Odakle nam zamenski grafovi?

1. Podelimo AIG na podgrafove
2. Obilazimo jedan po jedan podgraf
3. Zamenimo ga najboljim podgrafom koji imamo na raspolaganju, tako da funkcionalnost bude zadržana
4. Iteriramo dok imamo vremena/dok postoji napredak

# Prekompajlirane biblioteke

1. U nekom skupu relevantnih kola, izbrojimo najčešće funkcije
  2. Za svaku od njih generišemo biblioteku implementacija
  3. Ukoliko tokom zamene naiđemo na neku novu funkciju, izgenerišemo biblioteku **U LETU** (eng. **ON THE FLY**) i kešujemo je za dalje zamene
- (u suštini transfer learning)

# NPN klasifikacija

Primetimo da funkcije  $f$  i  $g$  koje možemo prevesti jednu u drugu permutacijom i/ili negacijom pojedinih ulaza i/ili negacijom izlaza imaju isti skup AIG struktura, izuzev komplementacije nekih grana na periferiji  
 $\implies$  dovoljno je da čuvamo biblioteku za predstavnika svake klase, a ne za sve njene elemente

To obezbeđuje značajnu uštedu (na primer, postoji 65536 funkcija 4 promenljive, a samo 222 NPN klase)

# Kako vršimo zamenu?

Setimo se prošlog časa (pohlepni odabir implikanti)

## Varijanta 1

1. Uzmi implikantu koja pokriva najviše temena
2. Uprosti matricu pokrivanja
3. Ako je prazna, završi, ako ne, vrati se na 1)

Možemo direktno odabrati zamenski graf koji minimizuje neku metriku (najčešće broj čvorova) posmatran u izolaciji



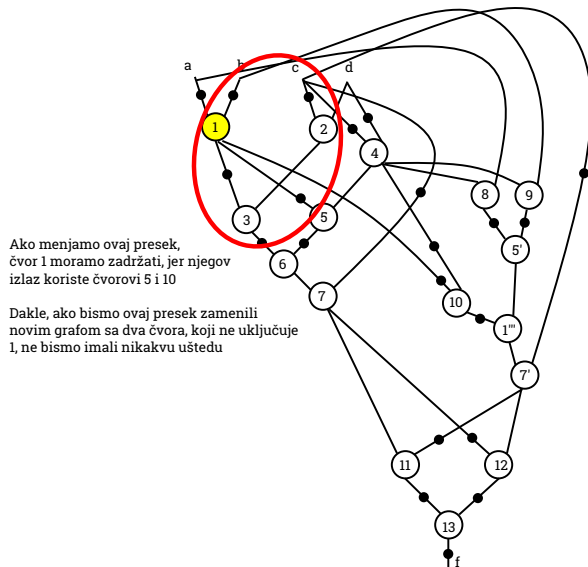
No, to ne uzima u obzir nemogućnost uklanjanja čvorova iz postojećeg preseka koji predstavljaju ulaz nekim čvorovima van preseka

## Varijanta 2 sa prošlog časa

1. Počni od praznog parcijalnog rešenja
2. Za sve preostale implikante, kopiraj parcijalno rešenje i proširi ga datom implikantom
3. Zameni parcijalno rešenje proširenjem koje je ostavilo najmanje nepokrivenih temena (i najmanje implikanti) nakon uprošćenja matrice pokrivanja

Sada smo od linearnog broja generisanih proširenja u najgorem slučaju došli do kvadratnog, ali je potencijalno kvalitet dobijenog rešenja veći

## Razmatranje efekta uklanjanja



# Ceo algoritam Miščenka i ostalih

```
Rewriting( network AIG, hash table PrecomputedStructures, bool UseZeroCost )
{
    for each node N in the AIG in the topological order {
        for each 4-input cut C of node N computed using cut enumeration {
            F = Boolean function of N in terms of the leaves of C
            PossibleStructures = HashTableLookup( PrecomputedStructures, F );
            // find the best logic structure for rewriting
            BestS = NULL; BestGain = -1;
            for each structure S in PossibleStructures {
                NodesSaved = DereferenceNode( AIG, N );
                NodesAdded = ReferenceNode( AIG, S );
                Gain = NodesSaved - NodesAdded;
                Dereference( AIG, S ); Reference( AIG, N );
                if ( Gain > 0 || (Gain == 0 && UseZeroCost) )
                    if ( BestS == NULL || BestGain < Gain )
                        BestS = S; BestGain = Gain;
            }
            if ( BestS == NULL ) continue;
            // use the best logic structure to update the netlist
            NodesSaved = DereferenceNode( AIG, N );
            NodesAdded = ReferenceNode( AIG, S );
            assert( BestGain == NodesSaved - NodesAdded );
        }
    }
}
```

**Figure 1.** 4-input rewriting algorithm.

Zašto samo 4 ulaza?

## Zašto samo 4 ulaza?

1. Istinitosne tablice i simulacija su nam dovoljne za sve
2. Možemo da enumerišemo sve neredundantne AIG strukture za svaku od relevantnih NPN klasa

Mane?

Mali preseki pružaju ograničenu mogućnost za optimizaciju

Kasnija unapređenja su omogućila razmatranje i većih funkcija



# Neka otvorena pitanja

Kako da generišemo kola koja su lakše poveziva?

Jedna mogućnost: umesto merenja uštede čvorova pozivamo ML model koji predviđa promenu povezivosti

Ovaj veoma jednostavan pristup je omogućio postizanje jednako kvalitetne optimizacije kao i kombinacija velikog mnoštva heuristika koje su ranije dominantno bile u uoptrebi, ali u deliću procesorskog vremena

Ključ je opet u efikasnim strukturama podataka i dobrom balansiranju pregnerisanja i dopune u letu

# Stare metode ipak nisu prevaziđene

Kada se previše heuristika nagomila tako da više niko ne zna šta je korisno a šta ne, pojavi se neko ko sve značajno uprosti, ali onda ponovo nastupa period gomilanja, jer su dodatne optimizacije neophodne sa povećanje efikasnosti jednostavnih metoda

# Stare metode ipak nisu prevaziđene

 <https://cseweb.ucsd.edu/classes/fa23/cse248-a/papers/logic/ls-handout.pdf>

— | + Automatic Zoom ▾

## Logic Synthesis in a Nutshell

**Jie-Hong Roland Jiang**

*National Taiwan University, Taipei, Taiwan*

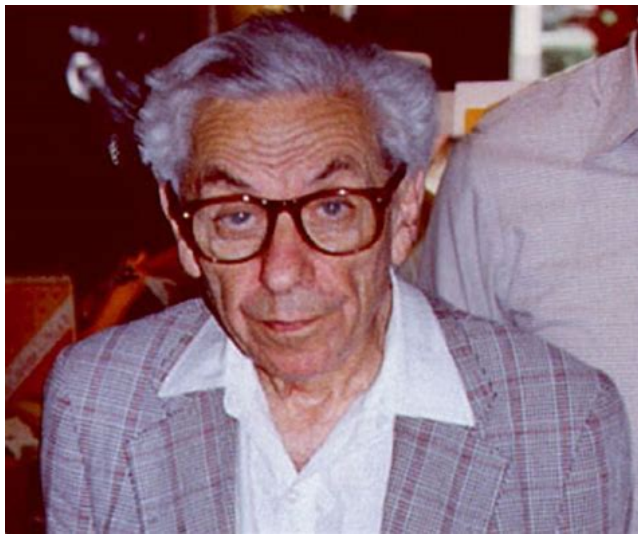
**Srinivas Devadas**

*Massachusetts Institute of Technology, Cambridge, Massachusetts*

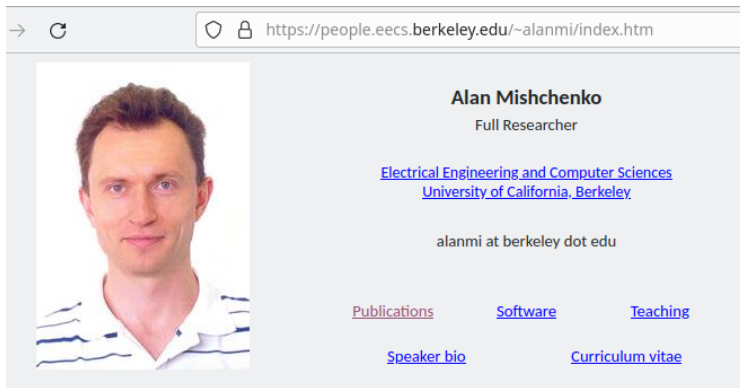
July 13, 2010

Odličan pregled klasičnih metoda možete naći ovde

Da li znate ko je ovo?



# U svetu logičke sinteze, to je



Implementacija je jako bitna i dobro je s vremena na vreme pisati novu

# Primer iz ABC-a

```
#define ABC_ALLOC(type, num) ((type *) malloc(sizeof(type) * (size_t)(num)))
#define ABC_CALLOC(type, num) ((type *) calloc((size_t)(num), sizeof(type)))
#define ABC_FALLOC(type, num) ((type *) memset(malloc(sizeof(type) * (size_t)(num)), 0xff, sizeof(type) * (size_t)(num)))
#define ABC_FREE(obj) ((obj) ? (free((char *) (obj)), (obj) = 0) : 0)
#define ABC_REALLOC(type, obj, num) \
    ((obj) ? ((type *) realloc((char *) (obj), sizeof(type) * (size_t)(num))) : \
    ((type *) malloc(sizeof(type) * (size_t)(num))))

static inline int  Abs_Int( int a ) { return a < 0 ? -a : a; }
static inline int  Abs_MaxInt( int a, int b ) { return a > b ? a : b; }
static inline int  Abs_MinInt( int a, int b ) { return a < b ? a : b; }
static inline word Abs_MaxWord( word a, word b ) { return a > b ? a : b; }
static inline word Abs_MinWord( word a, word b ) { return a < b ? a : b; }
static inline float Abs_Float( float a ) { return a < 0 ? -a : a; }
static inline float Abs_MaxFloat( float a, float b ) { return a > b ? a : b; }
static inline float Abs_MinFloat( float a, float b ) { return a < b ? a : b; }
static inline double Abs_AbsDouble( double a ) { return a < 0 ? -a : a; }
static inline double Abs_MaxDouble( double a, double b ) { return a > b ? a : b; }
static inline double Abs_MinDouble( double a, double b ) { return a < b ? a : b; }

static inline int  Abs_Float2Int( float Val ) { union { int x; float y; } v; v.y = Val; return v.x; }
static inline float Abs_Int2Float( int Num ) { union { int x; float y; } v; v.x = Num; return v.y; }
static inline word Abs_Dbl2Word( double Dbl ) { union { word x; double y; } v; v.y = Dbl; return v.x; }
static inline double Abs_Word2Dbl( word Num ) { union { word x; double y; } v; v.x = Num; return v.y; }
static inline int  Abs_Base2Log( unsigned n ) { int r; if ( n < 2 ) return (int)n; for ( r = 0, n--; n >= 1, r++ ) {} return r; }
static inline int  Abs_Base64Log( unsigned n ) { int r; if ( n < 2 ) return (int)n; for ( r = 0, n--; n >= 10, r++ ) {} return r; }
static inline char * Abs_UtilStrsav( char * s ) { return s ? strcpy(ABC_ALLOC(char, strlen(s)+1), s) : NULL; }
static inline char * Abs_UtilStrsavTwo( char * s, char * a ) { char * r; if (!a) return Abs_UtilStrsav(s); r = ABC_ALLOC(char, strlen(s)+strlen(a)+1); sprintf(r, "%s%s", s, a ); return r; }
```

poseban alokator memorije

```
// create the table
p->pTable = ABC_ALLOC( Rwr_Node_t *, p->nFuncs );
memset( p->pTable, 0, sizeof(Rwr_Node_t *) * p->nFuncs );
// create the elementary nodes
p->pNode = Extra_MemFixedStart( sizeof(Rwr_Node_t) );
p->vForest = Vec_PtrAlloc( 100 );
Rwr_ManAddVar( p, 0x0000, fPrecompute ); // constant 0
Rwr_ManAddVar( p, 0xAAAA, fPrecompute ); // var A
Rwr_ManAddVar( p, 0xCCCC, fPrecompute ); // var B
Rwr_ManAddVar( p, 0xF0F0, fPrecompute ); // var C
Rwr_ManAddVar( p, 0xFF00, fPrecompute ); // var D
p->nClasses = 5;
// other stuff
p->nTravIds = 1;
p->pPerms4 = Extra_Permutations( 4 );
p->vLevNums = Vec_IntAlloc( 50 );
p->vFanIns = Vec_PtrAlloc( 50 );
p->vFanInsCur = Vec_PtrAlloc( 50 );
p->vNodesTemp = Vec_PtrAlloc( 50 );
if ( ! fPrecompute )
{ // precompute subgraphs
  Rwr_ManPrecompute( p );
  Rwr_ManPrint( p );
}
else
{ // load saved subgraphs
  Rwr_ManLoadFromArray( p, 0 );
  Rwr_ManPrint( p );
  Rwr_ManPreprocess( p );
}
```

efikasnost ispred lakoće održavanja



# Još saveta direktno od njega

← → ↺ <https://www.youtube.com/watch?v=bXdAIqeu9ck> ☆

YouTube RS Search 🔍 🎤

IEEE CASS RS Talks 2022  
UFRGS, Porto Alegre, Brazil

October 14, 2022, 1:30 PM (Brasilia Time, GMT-3)

Alan Mishchenko, University of Berkeley, USA

Towards Next Generation Logic Synthesis and Verification

YouTube Live @ IEEE CASS Rio Grande do Sul Chapter  
<https://www.youtube.com/cassriograndedosul>

Participants: CASS Ricardo Reis, Andre Reis, Alan Mishchenko

Logos: UFRGS, INF, UFRGS, IEEE, CAS

Play (k) 0:11 / 55:35

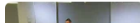
Live chat replay is not available for this video.

All

From your search

From IEEE CASS Rio Gr >

CASS Talks 2022 - Alan Mishchenko, University of Berkeley, USA -




Circuit Minimization with QBF

# Malo istorije

← → ↺ <https://www.youtube.com/watch?v=r1mWxjVirwE> 🔒 ⚙️


☰ YouTube<sup>RS</sup> Search 🔍 🎤

📺 🔔 9+ S



▶ ⏮ 🔊 1:37:39 / 1:50:35 ⏸ ⏪ ⏩ ⚙️ 📺 🖥️ 🔍


**Oral History of Robert "Bob" Brayton**

 **Computer History M...**  
151K subscribers

**Subscribe**

👍 25 🗨️ ➦ Share ...

**All** From Computer History Muse... History >

 **Starfall** - Sovietwave Mix ⋮

# Malo istorije

← → ↺ [https://www.youtube.com/watch?v=q\\_2eBd9JUWg](https://www.youtube.com/watch?v=q_2eBd9JUWg) 🔒 ⏏ ⚙

☰ YouTube RS

alberto sangiovanni vincentelli cass 🔍 🎤

📺 9+ S

YouTube Live @ IEEE CASS Rio Grande do Sul Chapter  
<https://www.youtube.com/cassriograndedosul>

IEEE CASS RS Talks 2020  
Porto Alegre, Brazil  
Instituto de Informática, UFRGS

1:30 PM (Brasilia Time, GMT-3), September 18, 2020

Prof. Alberto Sangiovanni-Vincentelli, UC Berkeley, USA

From Transistors to Systems: the Role of EDA in an Evolving World

**Abstract:**  
Information technology moves rapidly to an increasingly decentralized and collaborative environment (the Cloud) with rich interfaces to the physical world (the Internet of Things and CyberPhysical Systems). In particular, it has been predicted that by 2025 several billions (thousands per person) of electronic devices will be available. These devices will allow making the computing infrastructures available to humans and supporting societal scale applications that are unthinkable today. However, even today we are facing a number of severe challenges in applications such as autonomous vehicles, that should be monitored carefully with respect to safety, security and privacy concerns, not to mention the consequences of a devastating pandemic. The advances in system technology are clearly enabled by semiconductor technology. The capability of designing chips with billions of transistors is tested in the evolution of EDA. EDA is about rigorous and automated approaches to assemble complex systems out of components so that they are optimized with respect to a number of goals including power consumption, speed, reliability, safety and security. I will review the evolution of EDA, from its inception to today and the new challenges posed by autonomy and security with potential solutions. A glimpse at the use of Machine Learning and its premises will be provided with caveats about intrinsic limitations.

**Short CV:**  
Alberto Sangiovanni-Vincentelli is the Edgar L. and Harold H. Buttner Chair of EECS, University of California, Berkeley. He has been with Berkeley since 1975. Special advisor to the Dean of Engineering for Entrepreneurship and Chair of the Faculty Advisors to the Berkeley Accelerator SkyDeck Awards (among others). Kaufman for pioneering contributions to EDA. IEEE/SEI Maxwell Medal "for groundbreaking contributions that have had an exceptional impact on the development of electronics and electrical engineering or related fields." Inaugural co-founded Cadence and Synopsys, the two leading EDA companies with combined evaluation in MAGADQ of close to 600M USD. Member of the Board of Directors, Cadence, KPT, GEO, ExpertSystem, UltraSoC (chamars). Co-chair, consulted for companies worldwide including Intel, IBM, ST, Mercedes, BMW, UCL, GPC, Chameleon, International Advisory Council, Philips Innovation Director, Intel EXPO, member of NAE, IEEE and ACM Fellow. Honorary Doctorate from Aalborg University and from KTH (Sweden). Published 1,120 papers and 19 books with 115 co-index and graduated over 100 Doctorate Students.

UFRGS PAMICRO CAS

Play (k) 0:05 / 2:07:47

🔍 ⚙ 📺 📱 🖥

## CASS Talks 2020 - Alberto Sangiovanni-Vincentelli, UC Berkeley, USA - September 18, 2020

All

From your search

From IEEE CASS Rio Gr



IEEE CASS ...



Subscribed



14



Share



CASS Talks 2020



CASS Talks 2020 - David Pan,



# Kako radi SAT rešavač?

---

# Setimo se pretrage sa prošlog časa

Osnovni princip:

1. Minimizujemo matricu pokrivanja koliko god možemo
2. Ako je matrica prazna, našli smo pokrivanje
3. U suprotnom, granamo se dalje (dodajemo novu implikantu)
4. Ako je (procenjena ukupna) cena pokrivanja veća od najmanje cene poznatog pokrivanja, vraćamo se nazad (backtracking)

## Traženje zadovoljavajuće dodele u CNF SAT-u je u osnovi vrlo slično

1. Minimizujemo matricu pokrivanja koliko god možemo
2. Ako je matrica prazna, našli smo pokrivanje
3. U suprotnom, granamo se dalje (dodajemo novu implikantu)
4. Ako je (procenjena ukupna) cena pokrivanja veća od najmanje cene poznatog pokrivanja, vraćamo se nazad (backtracking)

(u osnovnoj formulaciji problema) Potrebna nam je jedna dodela, pa čim je nađemo obustavljamo pretragu

## Jedan primer

Neka je formula čiju zadovoljivost proveravamo  $f = \omega_1 \omega_2 \omega_3 \omega_4 \omega_5 \omega_6 \dots$  i neka su proizvodi od značaja za ovaj primer redom  $\omega_1 = (x_1, x_{31}, x'_2)$ ,  $\omega_2 = (x_1, x'_3)$ ,  $\omega_3 = (x_2, x_3, x_4)$ ,  $\omega_4 = (x'_4, x'_5)$ ,  $\omega_5 = (x_{21}, x'_4, x'_6)$ ,  $\omega_6 = (x_5, x_6)$

Neka smo trenutno na drugom nivou stabla pretrage (jedna promenljiva je prethodno bila fiksirana) i neka  $x_{21}$  postaje 0

Vršimo zamenu vrednosti promenljive  $x_{21}$  u svim zbirovima u kojima učestvuje

$$\omega_1 = (x_1, x_{31}, x'_2), \omega_2 = (x_1, x'_3), \omega_3 = (x_2, x_3, x_4), \omega_4 = (x'_4, x'_5), \omega_5 = (0, x'_4, x'_6), \\ \omega_6 = (x_5, x_6)$$



Vršimo zamenu vrednosti promenljive  $x_{21}$  u svim zbiorima u kojima učestvuje

$$\omega_1 = (x_1, x_{31}, x'_2), \omega_2 = (x_1, x'_3), \omega_3 = (x_2, x_3, x_4), \omega_4 = (x'_4, x'_5), \omega_5 = (0, x'_4, x'_6), \\ \omega_6 = (x_5, x_6)$$

Dalje uprošćavanje nije moguće, pa moramo ponovo da se granamo. Neka sada  $x_{31}$  postaje 0.

Vršimo zamenu vrednosti promenljive  $x_{31}$  u svim zbировima u kojima učestvuje

$$\omega_1 = (x_1, 0, x'_2), \omega_2 = (x_1, x'_3), \omega_3 = (x_2, x_3, x_4), \omega_4 = (x'_4, x'_5), \omega_5 = (0, x'_4, x'_6), \\ \omega_6 = (x_5, x_6)$$

Ponovo nije moguće dalje uprošćenje. Neka je sledeće grananje (na nivou 4) na promenljivoj koju ovde ne vidimo.

## Neka je grananje na nivou 5 $x_1 = 0$

$$\omega_1 = (0, 0, x'_2), \omega_2 = (0, x'_3), \omega_3 = (x_2, x_3, x_4), \omega_4 = (x'_4, x'_5), \omega_5 = (0, x'_4, x'_6), \\ \omega_6 = (x_5, x_6)$$

Da li je moguće još neko uprošćenje?

Ako neki nezadovoljen zbir ima samo jednu promenljivu koja još nije odlučena, onda on implicira vrednost te promenljive

Taj zbir nazivamo jediničnim (eng. **UNIT CLAUSE**)

Neka je grananje na nivou 5  $x_1 = 0$

$$\omega_1 = (0, 0, x'_2), \omega_2 = (0, x'_3), \omega_3 = (x_2, x_3, x_4), \omega_4 = (x'_4, x'_5), \omega_5 = (0, x'_4, x'_6), \\ \omega_6 = (x_5, x_6)$$

$$\omega_1 \implies x_2 = 0, \omega_2 \implies x_3 = 0$$

## Nastavljamo dalje

$$\omega_3 = (0, 0, x_4), \omega_4 = (x'_4, x'_5), \omega_5 = (0, x'_4, x'_6), \omega_6 = (x_5, x_6)$$

$$\omega_3 \implies x_4 = 1$$

## Nastavljamo dalje

$$\omega_4 = (0, x'_5), \omega_5 = (0, 0, x'_6), \omega_6 = (x_5, x_6)$$

$$\omega_4 \implies x_5 = 0, \omega_5 \implies x_6 = 0$$

## Nastavljamo dalje

$$\omega_5 = (0, 0, x'_6), \omega_6 = (0, x_6)$$

$$\omega_5 \implies x_6 = 0, \omega_6 \implies x_6 = 1 \implies \text{KONFLIKT}$$

Moramo se vratiti na prethodni nivo odlučivanja



Postupak uprošćavanja koji smo upravo videli nazivamo propagacijom Bulovih veza (eng. **BOOLEAN CONSTRAINT PROPAGATION** (BCP))

Slično kao i kod uprošćavanje matrice pokrivanja u algoritmu Kvajn-Meklaskog, oduzima većinu vremena (i do 90%; nalazi se u unutrašnjoj petlji)

## Naivni BCP (ovaj koji smo malopre primenili)

Kad literal postane 0, prođemo kroz sve zbirove koji ga sadrže i proverimo da li su jedinični

## Princip dva praćena literala (eng. TWO WATCHED LITERALS)

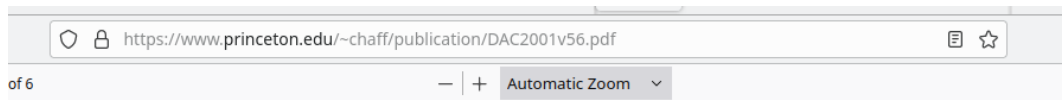
Zapažanje: Ako zbir ima barem dva neodlućena literala, ne moramo ga posetiti

## Princip dva praćena literala (eng. TWO WATCHED LITERALS)

Postupak:

1. Za svaki zbir nasumično odaberemo dva literala koja pratimo (na primer, prva dva)
2. Ako jedan od ta dva literala postane nula, proveravamo da li je zbir postao jedinični
3. Ako nije (postoji barem još jedan neodlučen literal), biramo novi literal za praćenje, iz skupa neodlučenih

Ovakva BCP implementacija je jedan od ključnih doprinosa razvoju CDCL SAT rešavača



## Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz  
Department of EECS  
UC Berkeley

[moskewcz@alumni.princeton.edu](mailto:moskewcz@alumni.princeton.edu)

Conor F. Madigan  
Department of EECS  
MIT

[cmadigan@mit.edu](mailto:cmadigan@mit.edu)

Ying Zhao, Lintao Zhang, Sharad Malik  
Department of Electrical Engineering  
Princeton University

[{yingzhao, lintaoz, sharad}@ee.princeton.edu](mailto:{yingzhao, lintaoz, sharad}@ee.princeton.edu)

# Analiza konflikata

---

## Da li možemo nešto da naučimo iz konflikata?

Određena parcijalna dodela nas je dovela do konflikta

Ako shvatimo koja je to dodela, možemo je izbeći u ostatku pretrage

I možemo da odredimo koliko visoko u stablu pretrage je potrebno da skočimo da bismo se vratili na pravi put

Čvorovi označavaju fiksirane promenljive

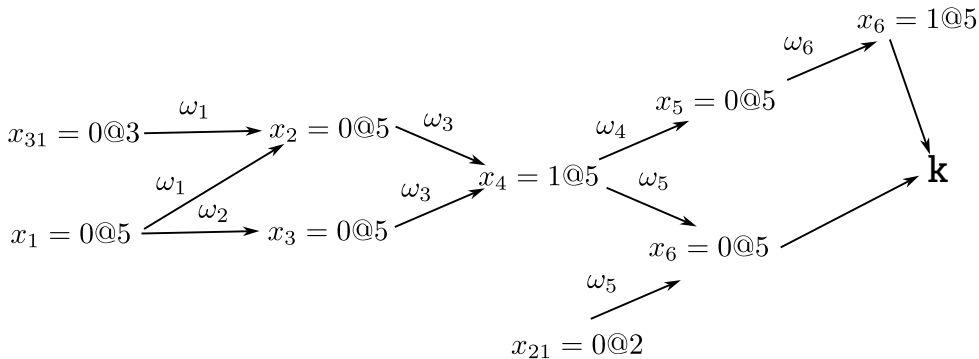
Pored svakog upisujemo nivo odluke  $u$  kom je data promenljiva fiksirana, kao i vrednost

Između čvorova  $u$  i  $v$  postoji grana akko postoji zbir  $\omega$  koji je implicirao vrednost promenljive  $v$  a sadrži i promenljivu  $u$

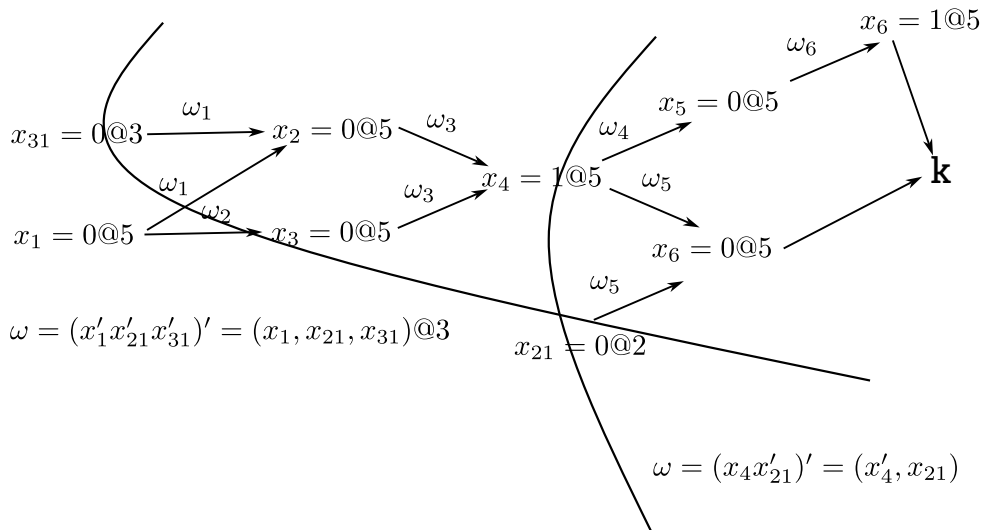
Grane označavamo odgovarajućim  $\omega$



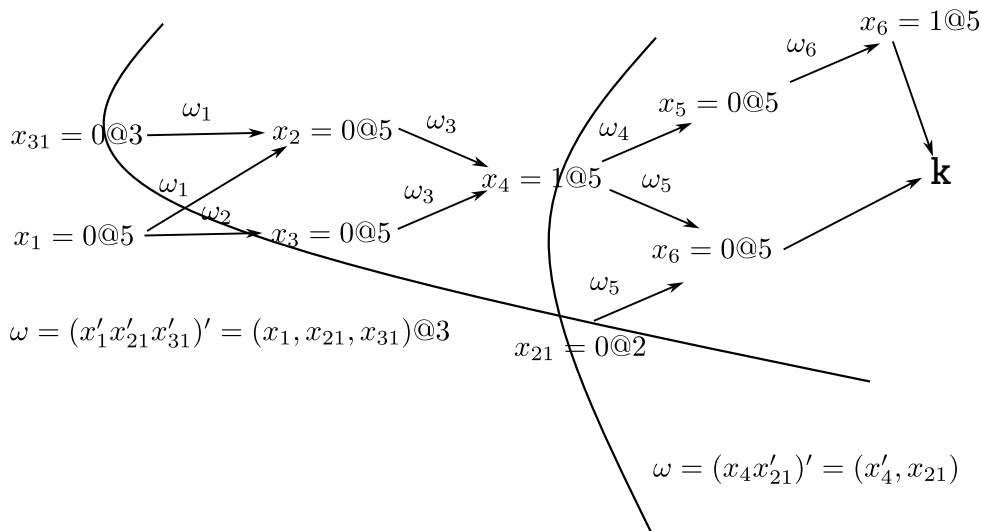
## U našem primeru



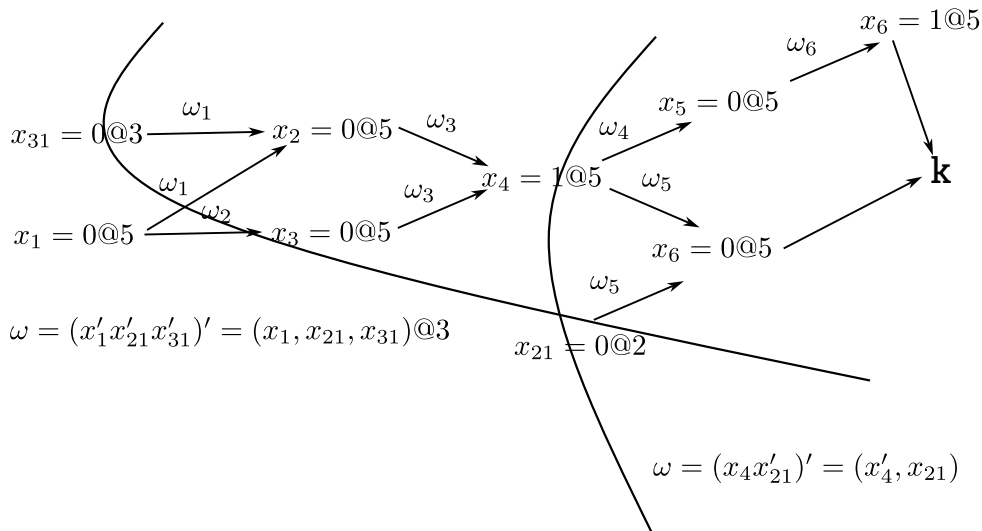
# Svaki presek generiše neku dodelu koju je potrebno izbeći

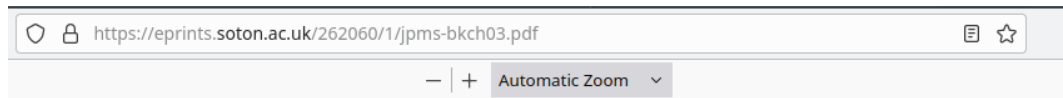


Tim zbiorovima možemo proširiti formulu kako bismo izbegli iste greške



Posle flipovanja poslednje odluka, najniži nivo u konfliktu nam je cilj povratnog skoka





# GRASP-A NEW SEARCH ALGORITHM FOR SATISFIABILITY

João P. Marques Silva  
*Cadence European Laboratories,  
IST/INESC-ID,  
1000 Lisboa, Portugal*

Karem A. Sakallah  
*Department of EECS,  
University of Michigan,  
Ann Arbor, Michigan 48109-2122*

# Inkrementalno rešavanje

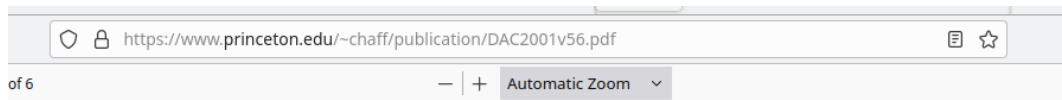
Ako rešavamo mnogo srodnih problema, neki naučeni zbirovi će ostati validni između njih i možemo ih upotrebiti

ABC to koristi u FRAIG-ovanju, na primer

## Neka dodatna pitanja

- Kako odlučiti koju sledeću promenljivu fiksirati i na koju vrednost?
- Koje naučene zbirove zaboraviti i kada?
- Restartovanje rešavača

Klasične odgovore je dao Chaff i oni su još uvek dominantno u upotrebi



## Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz  
Department of EECS  
UC Berkeley

[moskewcz@alumni.princeton.edu](mailto:moskewcz@alumni.princeton.edu)

Conor F. Madigan  
Department of EECS  
MIT




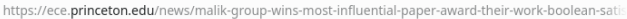


[cmadigan@mit.edu](mailto:cmadigan@mit.edu)

Ying Zhao, Lintao Zhang, Sharad Malik  
Department of Electrical Engineering  
Princeton University


[{yingzhao, lintaoz, sharad}@ee.princeton.edu](mailto:{yingzhao, lintaoz, sharad}@ee.princeton.edu)



# Još malo motivacije



PRINCETON ENGINEERING



PRINCETON  
ELECTRICAL AND COMPUTER  
ENGINEERING

About ▾People ▾Academics ▾Research ▾News ▾

Malik, the George van Ness Lothrop Professor in Engineering, and his group introduced in a 2001 [paper](#) an algorithm and a software tool capable of solving complex logic puzzles, known as Boolean Satisfiability problems, enabling more efficient computer chip design and impacting a wide range of other fields. An example of one of these problems, referred to as SATs, is the conundrum of seating 10 people at a dinner party where guest A wants to sit with guest B or C, but B is friends with D, who does not like guest C... When the number of guests increases, the problem becomes exponentially more difficult to solve. Problems in many fields and businesses can be represented as these kinds of puzzles. One of the most prominent influences of Malik's program is in the chip industry, where SAT problems are used to determine whether or not the computer chip will work correctly in billions of situations, before the chip is built. The project was pioneered as part of the senior thesis project of two undergraduates working in Malik's group, Conor Madigan and Matthew Moskewitz, both class of 2000. They named the tool Chaff and teamed up with Malik and graduate students Ying Zhao and Lintao Yang to further develop the program and publish the paper.

The publication was selected as the most influential paper of the decade spanning 2000-2010 by the Design Automation Conference (DAC), which is sponsored by the Association for Computing and Machinery and the Institute for Electrical and Electronics Engineers. The award is known as a "test of time award" because it recognizes work that has made a substantial, lasting impact on industry and academia. It is the oldest conference in the field of electronic design automation, and has given an award to one paper for each decade since the 1960s.

Ponovo je ključ u minimizaciji posla i što većoj “reciklaži” izračunatih rezultata

U osnovi CDCL algoritama je DPLL (Davis–Putnam–Logemann–Loveland) algoritam iz 1961. Već su oni znali za BCP, ali je bilo potrebno da dva studenta osnovnih studija 40 godina kasnije uoče kako da ga efikasnije implementiraju (praćenje dva literala je takođe već postojalo ali u znatno složenijoj i manje efikasnoj formi)

# Daleko smo od toga da razumemo šta rešavači zapravo rade

The image shows a YouTube video player interface. The browser address bar at the top displays the URL `https://www.youtube.com/watch?v=ZNghg_yz76Y`. The YouTube header includes the logo, a search bar, and navigation icons. The video content area shows a presentation slide with the title "Are we making progresses?" and the question "Do we know more now than 10 years ago?". The slide lists a strategy: "⇒ Progress is essentially based on using a portfolio of methods" and "Round Robin of many different strategies", followed by a bulleted list: "• Scoring Variables", "• Choosing Phases", "• Restarting", "• Handling special formulas (XOR, ...)", and "• Handling the clause database". A small video inset in the top right corner shows a speaker named "Laurent Simon". The video player controls at the bottom show the video is at 2:18 / 41:28. Below the player, the video title "Towards an (Experimental) Understanding of SAT Solvers" is displayed, along with the channel name "Simons Institute" (63.4K subscribers) and a "Subscribe" button. Engagement icons for likes (33), shares, and a menu are also visible.

Are we making progresses?

Do we know more now than 10 years ago?

⇒ Progress is essentially based on using a portfolio of methods

Round Robin of many different strategies

- Scoring Variables
- Choosing Phases
- Restarting
- Handling special formulas (XOR, ...)
- Handling the clause database

Laurent Simon

Simons Institute, Berkeley, Bordeaux

Feb, 10th 2021

Powered by Zoom

CC

Towards an (Experimental) Understanding of SAT Solvers

Simons Institute  
63.4K subscribers

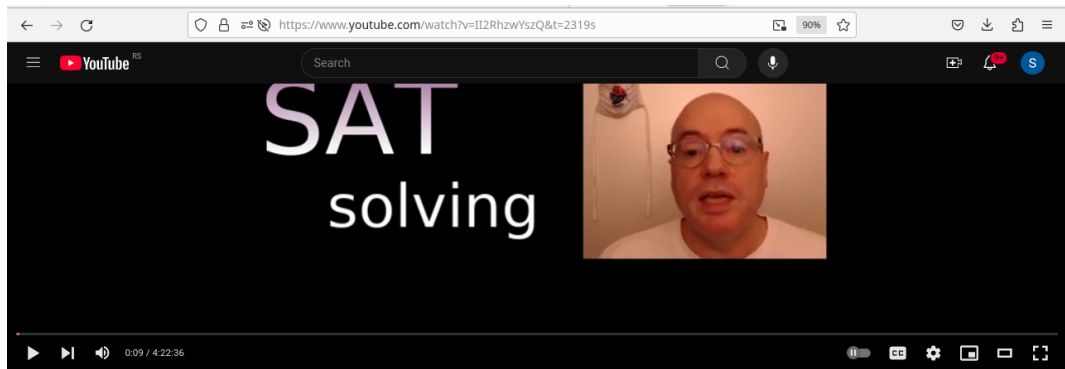
Subscribe

33

Share

All From your search From Simons Institute >

# Ipak, za one koji žele da saznaju više



## SAT-Solving



Simons Institute  
63.4K subscribers

Subscribe

136



Share

Clip



8.5K views 3 years ago

Armin Biere (Johannes Kepler University)

<https://simons.berkeley.edu/talks/sat...>

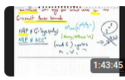
Satisfiability: Theory, Practice, and Beyond Boot Camp ...more

All

From your search

From the series

From



SAT-Centered Complexity  
Theory

Simons Institute

922 views • Streamed 3 years ago



A Peek Inside SAT Solvers - Jon

“Yes, there’s an end to scaling. But there’s no end to creativity.”

**Robert Dennard**

The inventor of DRAM