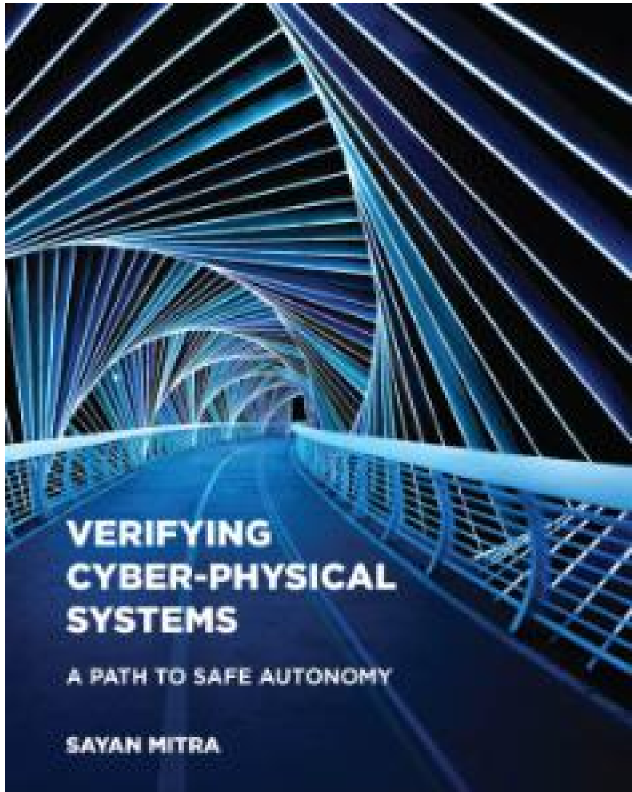
Lecture 3: Satisfiability

Huan Zhang

huan@huan-zhang.com



Slides adapted from Prof. Sayan Mitra's slides in Fall 2021
Some of the slides for this lecture are adapted from slides by Clark Barrett

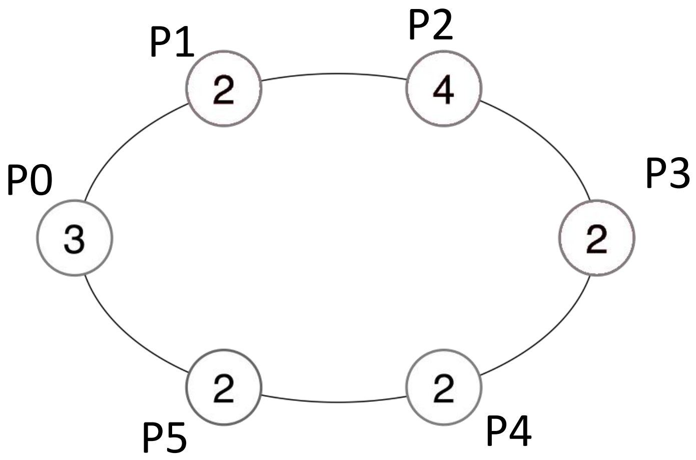
Readings

- Chapter 7
- Appendix C

Outline

- Review on the proofs of inductive invariance properties
- Propositional Satisfiability problem
- Normal forms
- DPLL algorithm (next lecture)

Dijkstra's mutual exclusion Algorithm ['74]



Who has the token?

N processes: 0, 1, …, N-1

state of each process j is a single integer variable x[j] ∈ {0, 1, 2, K-1}, where K > N

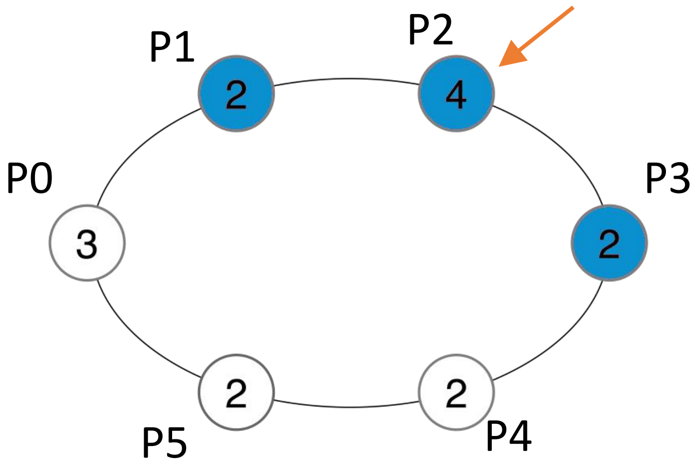
The "update" action is defined differently for P0 vs. others

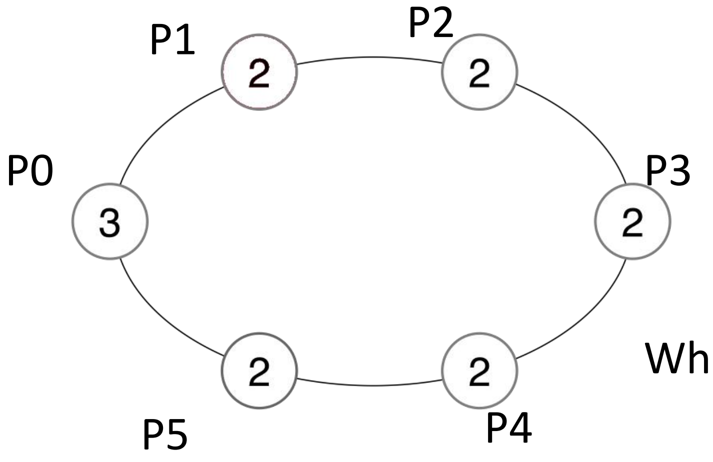
P_0 if x[0] = x[N-1] then x[0] := x[0] + 1 mod K

P_j, j > 0 if x[j] ≠ x[j -1] then x[j] := x[j-1]

p_i has TOKEN if and only if the blue conditional is true

Dijkstra's mutual exclusion Algorithm ['74]





Who has the token?

P1

N processes: 0, 1, …, N-1

state of each process j is a single integer variable x[j] ∈ {0, 1, 2, K-1}, where K > N

The "update" action is defined differently for P0 vs. others

P_0 if $x[0] = x[N-1]$ then $x[0] := x[0] + 1 \bmod K$

P_j, $j > 0$ if $x[j] \neq x[j-1]$ then $x[j] := x[j-1]$

p_i has TOKEN if and only if the blue conditional is true

A language for specifying automata (IOA)

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
     update(i:ID)
  variables
     x:[ID -> Val]  initially forall i:ID x[i] = 0
  transitions
     update(i:ID)
       pre i = 0 /\ x[i] = x[N-1]
       eff x[i] := (x[i] + 1) %  K

     update(i:ID)
       pre i >0  /\ x[i] ~= x[i-1]
       eff x[i] := x[i-1]
```

Automaton $\mathcal{A} = \langle X, \Theta,\ A,\ \mathcal{D} \rangle$

Reachable states and invariants

A state $\boldsymbol{u}$ is ***reachable*** if there exists an execution α such that $\alpha.lstate = \boldsymbol{u}$

$Reach_{\mathcal{A}}(\Theta)$: set of states reachable from Θ by automaton $\mathcal{A}$

An ***invariant*** is a set of states I such that $Reach_{\mathcal{A}} \subseteq I$

Proving invariants by induction (Chapter 7)

Theorem 7.1. Given a automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $\boldsymbol{x} \in \Theta$ implies $\boldsymbol{x} \in I$, and
- (Transition closure) for any $\boldsymbol{x} \rightarrow_a \boldsymbol{x}'$ and $\boldsymbol{x} \in I$ implies $\boldsymbol{x}' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

Proving invariants by induction for Dijkstra

Theorem 7.1. Given a automaton $\mathcal{A} = \langle X, \Theta,\ A,\ \mathcal{D}\rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $x \in \Theta$ implies $x \in I$, and
- (Transition closure) for any $x \rightarrow_a x'$ and $x \in I$ implies $x' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

- I_1: "Exactly one process has the token".

(Start condition): Fix a $\boldsymbol{x} \in \Theta$. $\boldsymbol{x} \vDash \forall i\ \boldsymbol{x}[x[i]] = 0$ therefore $\boldsymbol{x} \vDash I_1$

(Transition closure): Fix a $\boldsymbol{x} \rightarrow_a \boldsymbol{x}'$ such that $\boldsymbol{x} \in I$.

Two cases to consider.

1. If $a = update(0)$ then
   a) since $\boldsymbol{x} \vDash Pre(update(0))$ it follows that $\boldsymbol{x}[x[0] = \boldsymbol{x}[x[N-1]$
   b) since $\boldsymbol{x} \vDash I_1$ it follows that $\forall i > 0\ \boldsymbol{x}[x[i]] = \boldsymbol{x}[x[i-1]$
   c) $\boldsymbol{x}'[x[0] \neq \boldsymbol{x}'[x[N-1]$ by applying (a) and $Eff(update(0))$ to $\boldsymbol{x}$
   d) $\boldsymbol{x}'[x[1] \neq \boldsymbol{x}'[x[0]$ by applying (b) and $Eff(update(0))$ to $\boldsymbol{x}$
   e) $\forall i > 1\ \boldsymbol{x}'[x[i]] = \boldsymbol{x}'[x[i-1]$ by applying (b) *and* $Eff(update(0))$ to $\boldsymbol{x}$

   Therefore $\boldsymbol{x}' \vDash I$.
2. If $a = update(i)$, $\mathrm{i} > 0$ then fix arbitrary $i > 0$ … (do it as an exercise)

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
     update(i:ID)
  variables
     x:[ID -> Val] initially forall i:ID x[i] = 0
  transitions
     update(i:ID)
       pre i = 0 /\ x[i] = x[(N-1)]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)
       pre i >0  /\ x[i] ~= x[i-1]
       eff x[i] := x[i-1]
```

From above **Theorem** it follows that I_1 is an invariant of DijkstraTR

Proving invariants by induction for Dijkstra

Theorem 7.1. Given a automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $x \in \Theta$ implies $x \in I$, and
- (Transition closure) for any $x \rightarrow_a x'$ and $x \in I$ implies $x' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

- I_1: "Exactly one process has the token".

(Start condition): Fix a $\boldsymbol{x} \in \Theta$. $\boldsymbol{x} \vDash \forall i\ \boldsymbol{x}[x[i]] = 0$ therefore $\boldsymbol{x} \vDash I_1$

(Transition closure): Fix a $\boldsymbol{x} \rightarrow_a \boldsymbol{x}'$ such that $\boldsymbol{x} \in I$.

Two cases to consider.

1. If $a = update(0)$ then
   a) since $\boldsymbol{x} \vDash Pre(update(0))$ it follows that $\boldsymbol{x}[x[0] = \boldsymbol{x}[x[N-1]$
   b) since $\boldsymbol{x} \vDash I_1$ it follows that $\forall i > 0\ \boldsymbol{x}[x[i] = \boldsymbol{x}[x[i-1]$
   c) $\boldsymbol{x}'[x[0] \neq \boldsymbol{x}'[x[N-1]$ by applying (a) and $Eff(update(0))$ to $\boldsymbol{x}$
   d) $\boldsymbol{x}'[x[1] \neq \boldsymbol{x}'[x[0]$ by applying (b) and $Eff(update(0))$ to $\boldsymbol{x}$
   e) $\forall i > 1\ \boldsymbol{x}'[x[i] = \boldsymbol{x}'[x[i-1]$ by applying (b) and $Eff(update(0))$ to $\boldsymbol{x}$

   Therefore $\boldsymbol{x}' \vDash I$.
2. If $a = update(i)$, $\mathrm{i} > 0$ then fix arbitrary $i > 0$ … (do it as an exercise)

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
     update(i:ID)
  variables
     x:[ID -> Val] initially forall i:ID x[i] = 0
   transitions
     update(i:ID)
       pre i = 0 /\ x[i] = x[(N-1)]
       eff x[i] := (x[i] + 1) %  K

     update(i:ID)
      pre i >0  /\ x[i] ~= x[i-1]
      eff x[i] := x[i-1]
```

Can we prove this part automatically?
Yes! Use a *satisifiability* solver! (HW1)

From above **Theorem** it follows that I_1 is an invariant of DijkstraTR

Boolean *satisfiability* problem

Given a *well-formed formula* in propositional logic, determine whether there exists a satisfying solution

Example: $\alpha(x_1, x_2, \dots, x_n) \equiv (x_1 \wedge x_2 \vee x_3) \wedge (x_1 \wedge \neg x_3 \vee x_2)$

Set of variables: $X = \{x_1, x_2, \dots, x_n\}$,

Each variable is Boolean: $type(x_i) = \{0,1\}$

Formula α is *well-formed* if it uses propositional operators, and ∧ , or ∨, not ¬, iff ↔ etc., properly

Recall, a valuation **x** of X maps each x_i to a value 0 or 1

A valuation **x** of X *satisfies* α is each each x_i in α replaced by the corresponding value in **x** evaluates to *true*. We write this as $\boldsymbol{x} \vDash \alpha$

Otherwise, we write $\boldsymbol{x} \nvDash \alpha$

Example: with $\boldsymbol{x} \equiv \langle x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0 \rangle$; $\boldsymbol{x} \vDash \alpha$

Boolean *satisfiability* problem (SAT)

Given a well-formed formula in propositional logic, determine whether there exists a satisfying solution

Restatement: $\exists \boldsymbol{x} \in val(X): \boldsymbol{x} \vDash \alpha$?

If the answer is ”No” then α is said to be *unsatisfiable*

Aside. If $\forall \boldsymbol{x} \in val(X): \boldsymbol{x} \vDash \alpha$ then α is said to be *valid* or *a tautology*

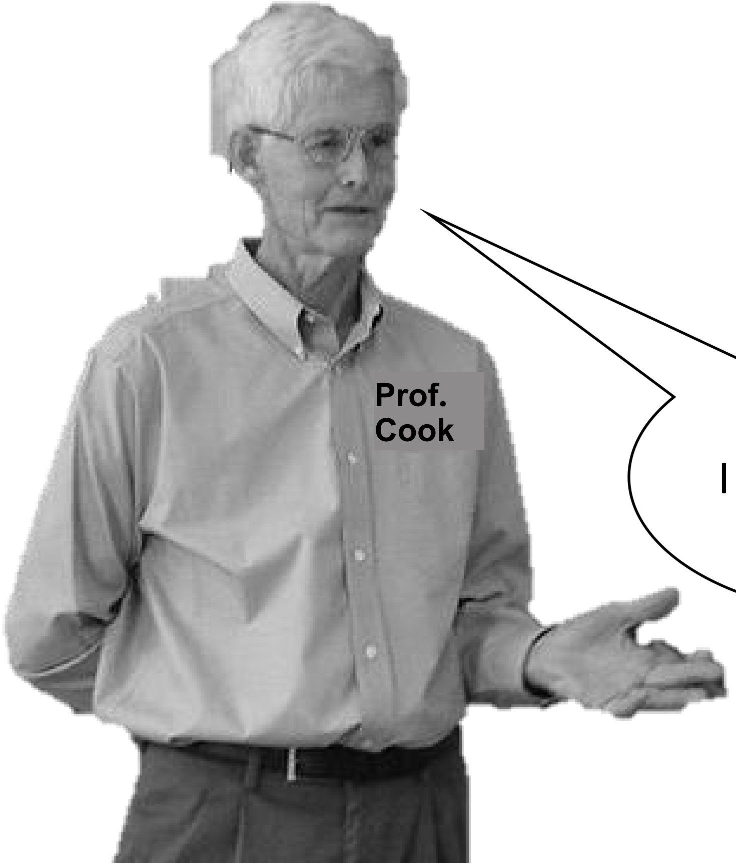
If α is valid then $\neg\alpha$ is unsatisfiable

α and α' are *tautologically equivalent* if they have the same truth tables

$$\forall \boldsymbol{x} \in val(X): \boldsymbol{x} \vDash \alpha \leftrightarrow \boldsymbol{x} \vDash \alpha'$$

What is a naïve method for solving SAT?

What is the complexity of this approach? How many evaluations of $\alpha(x_1, x_2, \ldots, x_n)$?



I Don’t Get No ***Satisfaction***, but I try, try, try,…

I can prove why.

Stephen A. Cook:
The Complexity of Theorem-Proving Procedures. [STOC 1971](): 151-158

Slide by Sayan Mitra using pictures from Wikipedia and cartoonstock.com

SAT is NP-complete

SAT was the first problem shown to be NP-complete [Cook 71]

2-SAT can be solved in polynomial time (Exercise)

(Read definition of NP: Nondeterministic Polytime in Appendix C)

This has real implications

1. Essentially we don't know better than the naïve algorithm
2. A solver for SAT can be used to solve any other problem in the NP class with only polytime slowdown. i.e., makes a lot of sense to build SAT solvers
3. SAT/SMT solving is the cornerstone of *many* verification procedures

Stephen Cook, The complexity of theorem-proving procedures. In Proceedings of the third annual ACM symposium on theory of computing. STOC '71.

Online SAT solvers [edit]

- BoolSAT – Solves formulas in the DIMACS-CNF format or in a more
- Logictools – Provides different solvers in javascript for learning, co
- minisat-in-your-browser – Solves formulas in the DIMACS-CNF for
- SATRennesPA – Solves formulas written in a user-friendly way. Ru
- somerby.net/mack/logic – Solves formulas written in symbolic logic

Offline SAT solvers [edit]

- MiniSAT – DIMACS-CNF format and OPB format for it's companio
- Lingeling – won a gold medal in a 2011 SAT competition.
  - PicoSAT – an earlier solver from the Lingeling group.
- Sat4j – DIMACS-CNF format. Java source code available.
- Glucose – DIMACS-CNF format.
- RSat – won a gold medal in a 2007 SAT competition.
- UBCSAT. Supports unweighted and weighted clauses, both in the
- CryptoMiniSat – won a gold medal in a 2011 SAT competition. C++ MiniSat 2.0 core, PrecoSat ver 236, and Glucose into one package,
- Spear – Supports bit-vector arithmetic. Can use the DIMACS-CNF
  - HyperSAT – Written to experiment with B-cubing search space solver from the developers of Spear.
- BASolver
- ArgoSAT
- Fast SAT Solver – based on genetic algorithms.
- zChaff – not supported anymore.

thousands variables
millions of clauses
are solvable

The international SAT Competitions web page

Current Competition

	SAT 2019 Race
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda

Past Competitions

	SAT 2018 Competition
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda

	SAT 2017 Competition
Organizers	Marijn Heule, Matti Järvisalo, Tomáš Balyo
Slides	Slides used at SAT 2017
Proceedings	Descriptions of the solvers and benchmarks
Benchmarks	Available here
Solvers	Available here

	Gold	Silver	Bronze	Gold	Silver
	Agile Track			Main Track	
SAT+UNSAT	CaDiCaL Agile, CaDiCaL NoProof	Glu_VC	Glucose 4.1	Maple LCM Dist, Maple LCM, MapleLRB LCMOccRestart, MapleLRB LCM	MapleCOMSPS LRB VSIDS 2, MapleCOMSPS LRB VSIDS
	Parallel Track			No-Limit Track	
SAT+UNSAT	Syrup24, Syrup48	Plingeling	Painless MapleCOMSPS	COMiniSATPS Pulsar	MapleCOMPSPS LRB VSIDS 2, MapleCOMPSPS LRB VSIDS

	SAT 2016 Competition
Organizers	Marijn Heule, Matti Järvisalo Tomáš Balyo
Proceedings	Descriptions of the solvers and benchmarks
Benchmarks	Available here
Solvers	Available here

	Gold	Silver	Bronze	Gold	Silver	Bronze
	Agile Track			Main Track		

Details

We will assume α to be in *conjunctive normal form (CNF)*

literals: variable or its negation, e.g., x_3, $\neg x_3$

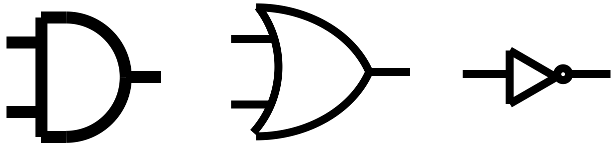
clause: disjunction (or) of literals, e.g., $(x_1 \lor x_2 \lor \neg x_3)$

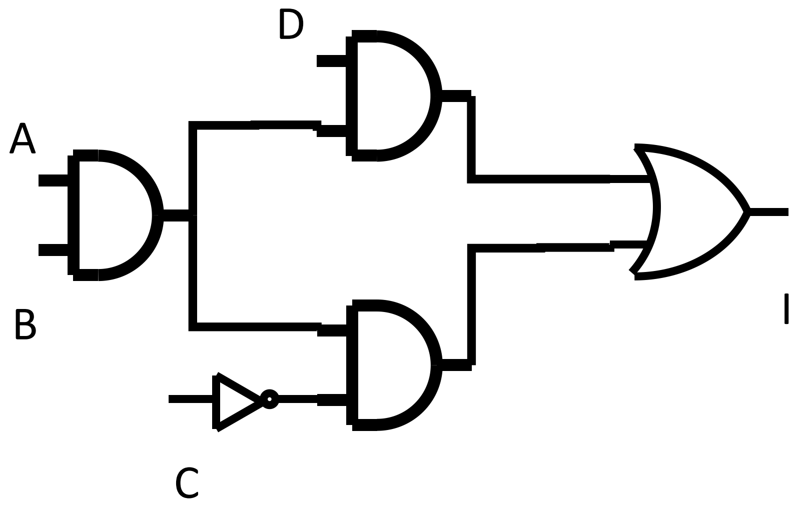
CNF formula: conjunction (and) of clauses,

e.g., $(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_2 \lor x_1)$

A variable may appear *positively* or *negatively* in a clause

Logic and circuits





$$I \equiv \big(D \wedge (A \wedge B)\big) \vee \big(\neg C \wedge (A \wedge B)\big)$$

Repeated subexpression is inefficient

Solution: rename $(A \wedge B) \leftrightarrow E$

$I' \equiv (D \wedge E) \vee (\neg C \wedge E) \wedge \big((A \wedge B) \leftrightarrow E\big)$

I and I' are **not** *tautologically equivalent*

$C = 0,\ A = B = 1,\ E = 0$ satisfies I

But they are *equisatisfiable,* i.e., I is satisfiable iff I' is also satisfiable

Recall that:

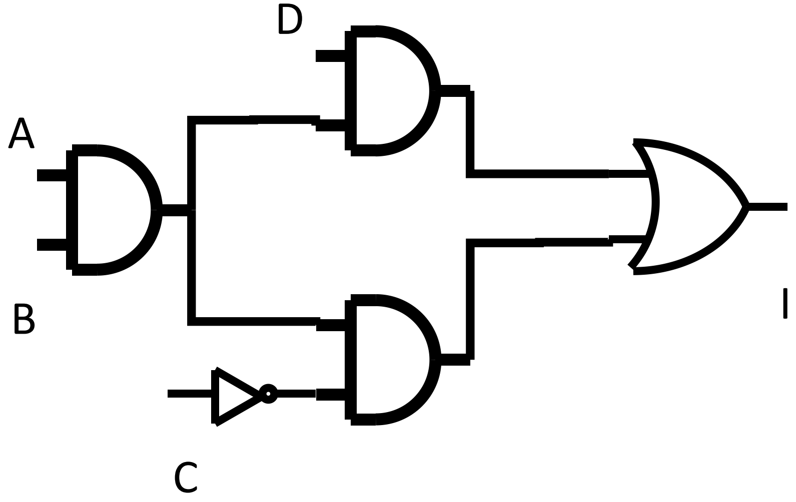
$A \leftrightarrow B$

$(A \rightarrow B) \wedge (B \rightarrow A)$

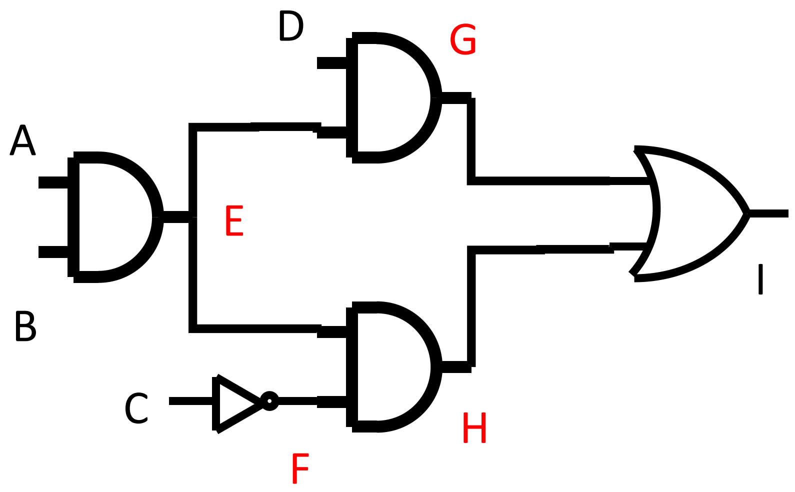
$(\neg A \vee B) \wedge (\neg B \vee A)$

Converting to CNF

- View the formula as a graph
- Give new names (variables) to non-leafs
- Relate the inputs and the outputs of the nonleafs and add this as a new clause
- Take conjunction of all of this



Converting to CNF



- $F \leftrightarrow \neg C$
  - $F \rightarrow \neg C \wedge \neg C \rightarrow F$
  - $(\neg F \vee \neg C) \wedge (C \vee F)$
- $(A \wedge B) \leftrightarrow E$
  - $\big((A \wedge B) \rightarrow E\big) \wedge \big(E \rightarrow (A \wedge B)\big)$
  - $(\neg(A \wedge B) \vee E) \wedge \big(\neg E \vee (A \wedge B)\big)$
  - $(\neg A \vee \neg B \vee E) \wedge \big(\neg E \vee A) \wedge (\neg E \vee B)\big)$
- $(G \vee H) \leftrightarrow I$
  - $\big((G \vee H) \rightarrow I\big) \wedge \big(I \rightarrow (G \vee H)\big)$
  - $(\neg G \wedge \neg H \vee I) \wedge (\neg I \vee G \vee H)$
  - $(\neg G \vee I) \wedge (\neg H \vee I) \wedge (\neg I \vee G \vee H)$
- $(D \wedge E) \leftrightarrow G$
  - $(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E)$
- $(F \wedge E) \leftrightarrow H$
  - $(\neg F \vee \neg E \vee H) \wedge \big(\neg H \vee F) \wedge (\neg H \vee E)\big)$

Standard representations of CNF

- $(\neg A \vee \neg B \vee E) \wedge \left(\neg\, E \vee A\right) \wedge (\neg E \vee B)\big)$
- $(A' + B' + E)(E' + A)(E' + B)$
- $(-1\ \ -2\ \ 5)(-5\ 1)(-5\quad 2)$ DIMACS

- SMTLib: computer readable, standard format

  https://smtlib.cs.uiowa.edu/language.shtml

Davis Putnam Logemann Loveland Algorithm (DPLL) 1962

Transform the given formula α by applying a **sequence of satisfiability preserving rules**

If final result has an empty clause then *unsatisfiable*

if final result has no clauses then the formula is *satisfiable*

Davis Putnam Algorithm (DP) 1960

Rule 1. **Unit propagation**

Rule 2. **Pure literal**

Rule 3. **Resolution**

DP 1960

Rule 1. **Unit propagation**

A clause has a single literal

$$\alpha \equiv \ldots \wedge \ldots \wedge p \wedge \ldots \wedge \ldots$$

What choice do we really have?

$$\alpha \equiv \ldots \wedge (x_1 \vee \neg p \vee x_2) \wedge p \wedge \ldots \wedge (\neg x_3 \vee \neg p \vee x_1) \ldots$$

DP 1960

Rule 1. **Unit propagation**

A clause has a single literal

$$\alpha \equiv \ldots \wedge \ldots \wedge p \wedge \ldots \wedge \ldots$$

What choice do we really have?

$$\alpha' \equiv \ldots \wedge (x_1 \vee x_2) \wedge \ldots \wedge (\neg x_3 \vee x_1) \ldots$$

α and α' are equisatisfiable

Davis Putnam Logemann Loveland Algorithm (DPLL) 1962

Rule 1. **Unit propagation**

Rule 2. **Pure literal**

A literal appears only positively (or negatively) in α

$$\alpha \equiv \dots \wedge (x_1 \vee \neg p \vee x_2) \wedge (x_4 \vee \neg p) \wedge \dots \wedge (\neg x_3 \vee \neg p \vee x_1) \dots$$

p does not appear anywhere

Makes sense to set $p = 0$ and remove all occurrences of $\neg p$

Davis Putnam Logemann Loveland Algorithm (DPLL) 1962

Rule 1. **Unit propagation**

Rule 2. **Pure literal**

A literal appears only positively (or negatively) in α

$$\alpha \equiv \ldots \wedge (x_1 \vee \neg p \vee x_2) \wedge (x_4 \vee \neg p) \wedge \ldots \wedge (\neg x_3 \vee x_1) \ldots$$

p does not appear anywhere

Makes sense to set $p = 0$ and remove all clauses in which $\neg p$ occurs

α and α' are equisatisfiable

$$\alpha' \equiv \ldots \wedge \ldots \wedge \ldots \wedge (\neg x_3 \vee x_1) \ldots [p = 0]$$

Davis Putnam Algorithm (DP) 1960

Rule 1. **Unit propagation**

Rule 2. **Pure literal**

Rule 3. **Resolution**

Choose a literal p that appears with both polarity in α. Suppose $(\ell_1 \vee \ell_2 \vee p)$ be a clause in which p appears positively, and $(k_1 \vee k_2 \vee \neg p)$ be a clause in which p appears negatively

Then the resolved clause is $(\ell_1 \vee \ell_2 \vee k_1 \vee k_2)$

Pairwise, resolve each clause in which p appears positively with a clause in which p appears negatively, and take the conjunction of all the results

Why is the result equisatisfiable?

What is the size of the resulting formula?

DPLL modifies resolution in DP with recursive DFS rule

Rule 1. **Unit propagation**

Rule 2. **Pure literal**

Rule 3’. Let Δ be the current set of clauses. Choose a literal p in Δ**.**

Check satisfiability of $\Delta \cup \{ p \}$ (guessing $p = 1$)

If satisfiable then return True else

return result of checking satisfiability of $\Delta \cup \{ \neg p \}$

This is essentially a depth first search

Assignments

- HW1 (due Feb 11th)
  - Install Z3
- Keep thinking about class projects! Form teams (max 2 people).
- More on DPLL next lecture