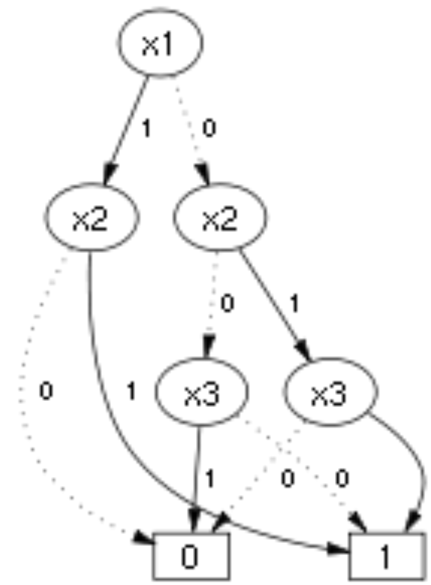
Lecture 11

Pete Manolios
Northeastern

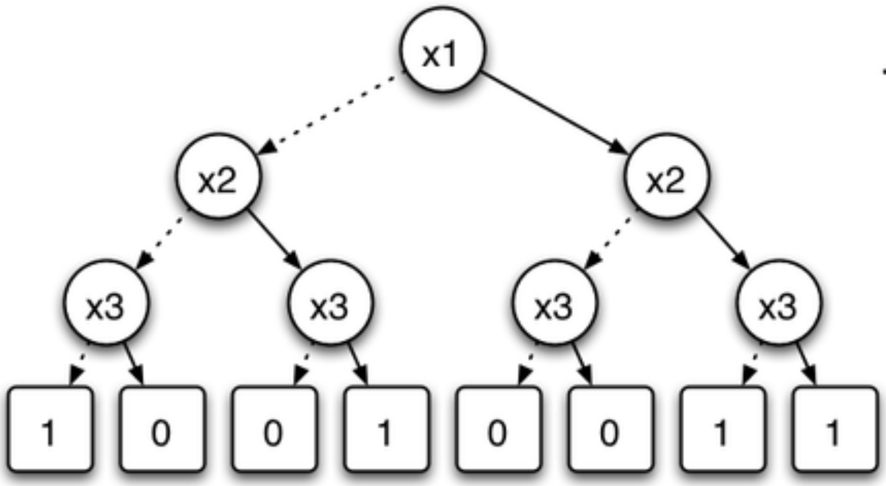
BDDs and Decision Trees

- A BDD on $x_1, \ldots, x_n$ is a DAG G=(V, E) where
  - exactly 1 vertex has indegree 0 (the root)
  - all vertices have outdegree 0 (leaves) or 2 (inner nodes)
  - the inner nodes are labeled from $\{x_1, \ldots, x_n\}$
  - the leaves are labeled from {0, 1}
  - one of the edges from an inner node is labeled by 0; the other by 1
- The BDD G=(V, E) represents a Boolean function, say f
  - for any assignment A in B^n, f(A) is computed recursively from root
  - if we reach a leaf, return the label
  - for inner nodes, say labeled with x_i, take the edge labeled by $A(x_i)$
- A decision tree is a BDD whose graph is a tree
- A BDD is an OBDD if there is a permutation on p={1,2, …, n} s.t. for all edges (u, v) in E, where u, v are labeled by x_i, x_j, we have that $p_i < p_j$
- An OBDD is an ROBDD if it has no isomorphic subgraphs and all children are distinct

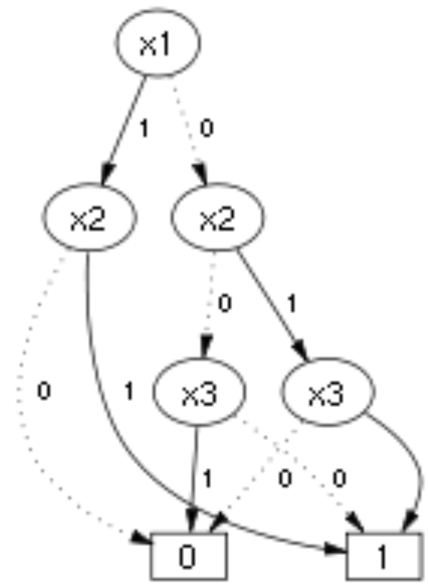


Images from Wikipedia

BDDs and Decision Trees



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Decision Tree for f

ROBDD for f

How do we generate DNF from a decision tree? ROBDD?

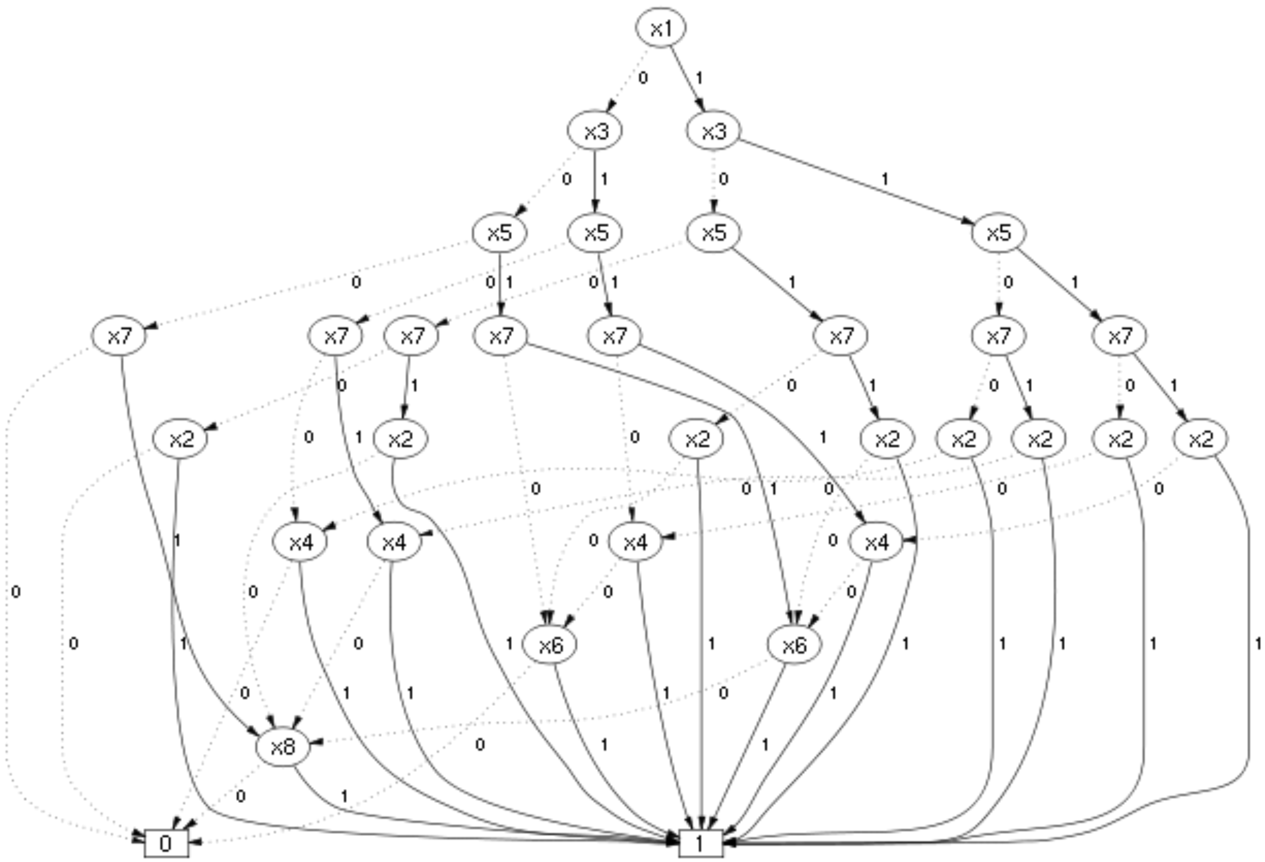
Images from Wikipedia

BDDs

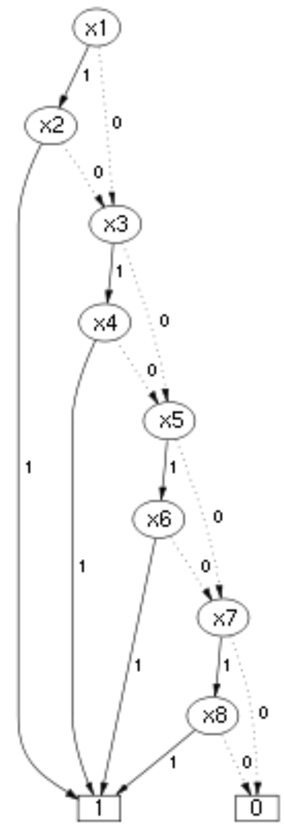
- Decision trees are widely used, e.g., in machine learning (ID3, C4.5, …)
- BDDs are widely used (BDD usually means ROBDD)
  - Popularized by Bryant
  - Very efficient algorithms for constructing, manipulating BDDs
  - Used in verification, synthesis, fault trees, security, AI, model checking, static analysis, …
  - Bryant's paper was the most cited research paper (at some point)
  - Many BDD packages available
- Once a variable ordering is selected, BDDs are canonical!
  - Construct decision tree using Shannon expansion and merge isomorphic nodes, remove nodes who children are equal until you reach a fixpoint
  - To see, this note that BDDs are essentially DFA that recognize strings in $\{0,1\}^n$ and such automata can be minimized (note nodes with equal children remain)
  - So, checking equality is just pointer equality (with appropriate data structures)
  - Can be used for model checking: represent set of reachable states & transition system with BDDs
  - Bryant, Clarke, Emerson & McMillan got 1998 Paris Kanellakis Award for symbolic model checking

Variable Ordering for BDDs

Variable ordering matters: find the best ordering is hard.



Bad Ordering



Good Ordering

Images from Wikipedia

DP SAT Algorithm

- Davis Putnam (1960)
- Input: CNF formula
- Output: SAT/UNSAT
- Idea: apply three rules until
  - Derive the empty clause: UNSAT (identity of v is false)
  - No clauses remain: SAT (identity of $\wedge$ is true)
- Three "rules"
  - Pure literal rule (affirmative-negative rule)
  - Unit resolution rule (unit propagation, BCP, 1-literal rule)
  - Resolution (Called consensus, also used for logic minimization)

Pure Literal Rule

- Given a F, a set of clauses and literal ℓ such
  - ℓ appears in F
  - $\neg\ell$ does not appear in F
  - remove all clauses containing ℓ
- Equisatisfiable because we can make ℓ true
- Notice that this always simplifies F
- Modern SAT solvers tend to not use the rule (efficiency)

Boolean Constraint Propagation

Unit resolution rule:

$$\frac{C, \neg\ell \qquad \ell}{C}$$

- BCP: given a set of clauses including $\{\ell\}$
  - remove all other clauses containing ℓ (subsumption)
  - remove all occurrences of $\neg\ell$ in clauses (unit resolution)
  - repeat until a fixpoint is reached

Resolution

Resolution rule:

$$\frac{C, v \qquad D, \neg v}{C, D} \quad \neg v, v \notin C, D$$

- Soundness of rule: above line implies below line
- If below line is SAT, so is above line (w/ side conditions)

Resolution

Resolution rule:

$$\frac{C, v \qquad D, \neg v}{C, D} \quad \neg v, v \notin C, D$$

Resolution rule:

$$\frac{C_i, p \qquad D_i, \neg p}{C_i, D_i} \quad \neg p \notin C_i \in P, p \notin D_i \in N$$

- Soundness of rule: above line implies below line
- If below line is SAT, so is above line (w/ side conditions)
- Given literal p, set of clauses S, let P be the clauses in S that contain p only positively and let N be the clauses that contain p only negatively. Let E be the rest of the clauses. Then S is sat iff S' = E U the set of all p-resolvents of P and N.
- Proof: If A is an assignment for S, then if A(p)=true, all clauses in N, with ¬p removed are satisfied, so each p-resolvent is satisfied. Similarly if A(p)=false. If A is an assignment for S', then it satisfies all Ci or all Di: suppose it doesn't satisfy Ck, then it must satisfy all Di. If it satisfies all Ci, let A'(p)=false, else A'(p)=true and A'(x)=A(x) otherwise.

DP SAT Algorithm

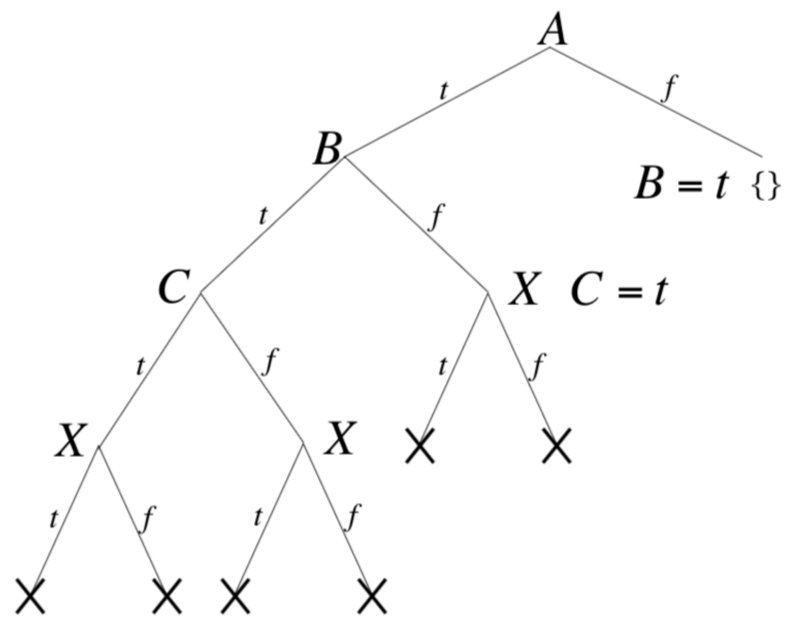
- Input: CNF formula, Output: SAT/UNSAT
- Base case: empty clause: UNSAT
- Base case: no clauses: SAT
  - Apply these two rules until fixpoint
    - Pure literal rule
    - BCP
  - Choose var, say x, perform all possible resolutions, remove trivial clauses and clauses containing x
  - Repeat
- Existentially quantify variables, one at a time
- Problem: space blow-up

DPLL SAT Algorithm

- BCP
- Base case: empty clause: UNSAT
- Remove clauses containing pure literals
- Base case: no clauses: SAT
- Choose some var, say x (has to appear in both phases)
  - Add {x} and recursively call DPLL
  - Add {¬x} and recursively call DPLL
  - If one of the calls returns SAT, return SAT
  - Else return UNSAT
- Correctness follows from Shannon expansion
- In contrast to DP, space is not a problem

DPLL SAT Example

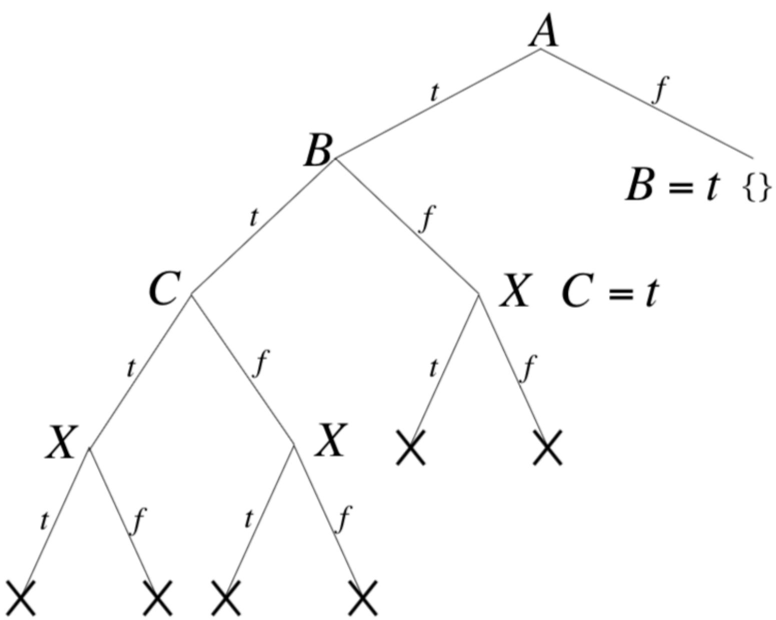
$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$



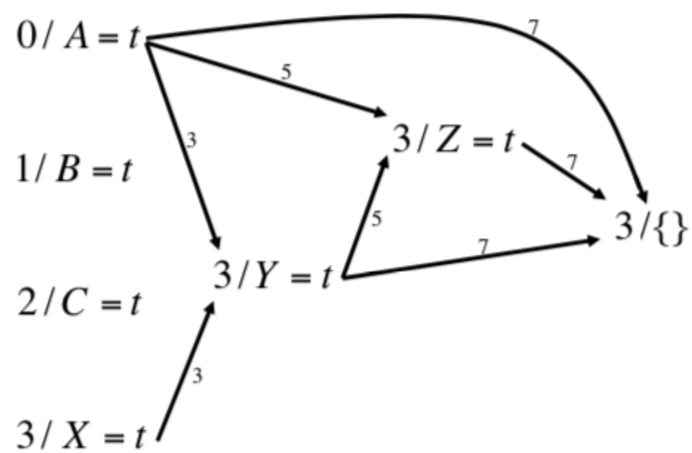
- Note that when DPLL detects contradictions it backtracks chronologically
- When we get a contradiction with X, we try ¬X, then we go back and try ¬C and X, ¬X again, …
- But the real problem was that we set A; can we avoid this exponential search?
- Yes: non-chronological backtracking, a major improvement

Examples/figures from chp. 3 SAT handbook

Implication Graphs

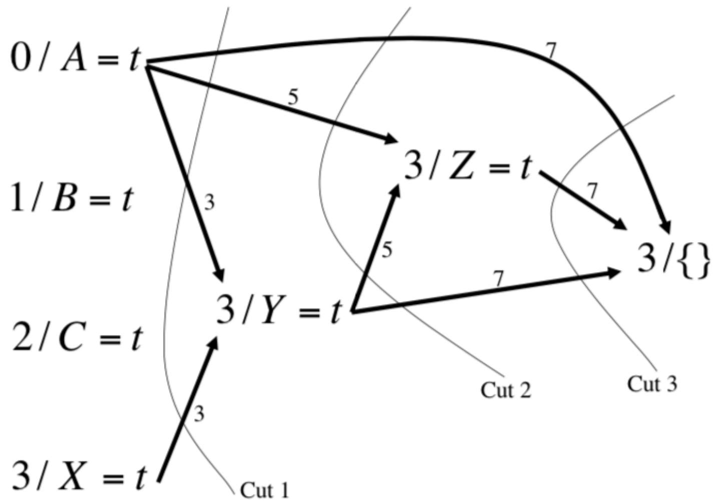


$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$



- Nodes are l/V=v: var V set to v @ level l
- If node implied, justification recorded (clause #, edges from assignments)
- {} denotes contradiction

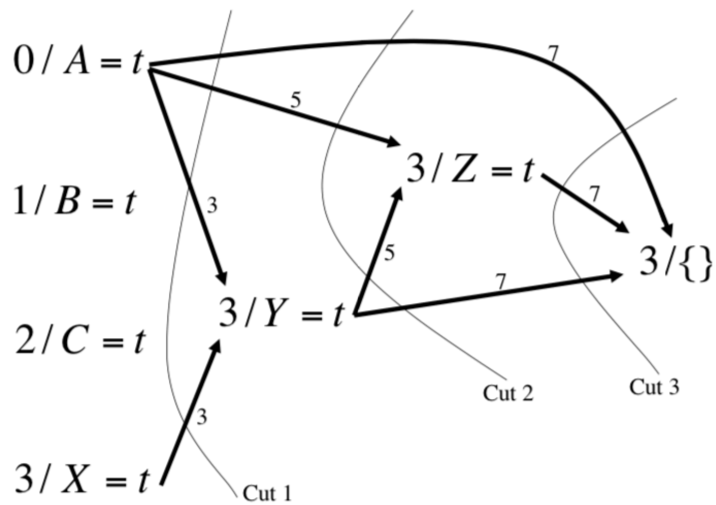
Conflict-Driven Clauses



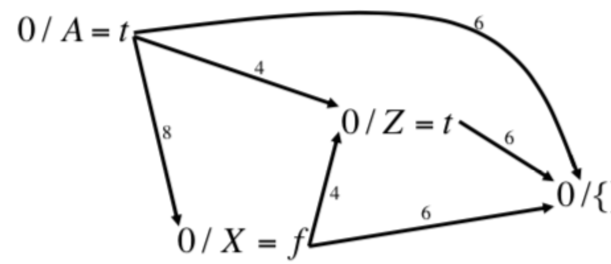
$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$

- Consider any cut of the implication graph that separates decision vars from {}
- The nodes with an edge that crosses the cut are in conflict set
- Negate the assignments in the set to obtain a conflict-driven clause
- Conflict clauses: Cut1: {¬A,¬X}, Cut2: {¬A, ¬Y}, Cut3: {¬A, ¬Z, ¬Y}
- Conflict–driven clauses generated from cuts that contain exactly one variable assigned at the level of conflict are said to be asserting: Cut1 & Cut2 (not Cut 3)

Non-Chronological Backtracking



$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$



- Asserting conflict clauses: Cut1: 8. {¬A,¬X}, Cut2: {¬A, ¬Y}
- Assertion level: 2nd highest level in asserting clause (0 for cuts 1, 2) or -1
- Backtrack to assertion level and add a learned clause (non-chronological!)
- We can now immediately infer (BCP) ¬X (we use Cut1), so we have A, ¬X
- Then by BCP: Z (4), ¬Z (6) so we get a new implication graph
- Asserting clauses: {¬A} at level -1, so we have ¬A, BCP: B and we're done
- Compare to previous search, where the algorithm had to go back a level at a time
- Clause learning can generate exponentially shorter proofs of unsat!

Modern CDCL Solvers

- Based on DPLL, but with conflict-driven clause learning
- Data structures to speed up BCP: 2-watched literal scheme
- Data structures for clause learning
- Decision heuristics: select recently active literals (VSIDS)
- Preprocessing: greedy variable elimination
- Inprocessing: interleave preprocessing & search
- Clause deletion: learned clauses lead to memory & efficiency problems, so delete large, inactive clauses
- Random restarts: keep learned clauses, but restart
  - avoids getting stuck in hard part of search space
  - phase saving: pick last phase of assignment