CS3383 Unit 5.0: Backtracking and SAT

David Bremner

March 31, 2024

Outline

Combinatorial Search

- Backtracking
- SAT
- Tractable kinds of SAT

N-queens

Problem Description

Given an $n \times n$ chess board, can you place n queens so that no two are in the same row, column, or diagonal.

	Q		
			Q
Q			
		Q	

$\checkmark$

	Q		
			Q
	Q		
		Q	

no

- One per row/col is easy to enforce

Representing Chessboards

- We only care about cases where there is 1 queen per column
- Represent a $n \times n$ board as an array of n integers, meaning which row.
- `None` for not chosen yet.

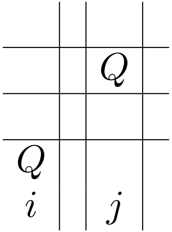
	Q		
			Q
Q			
		Q	

$(1, 3, 0, 2)$

		Q	
Q			

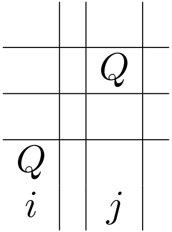
$(0, \texttt{None}, 2, \texttt{None})$

Detecting collisions

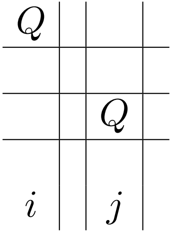


$$Q[j] - Q[i] = j - i$$

Detecting collisions

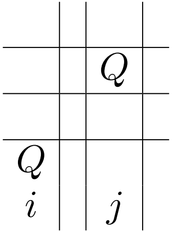


$$Q[j] - Q[i] = j - i$$

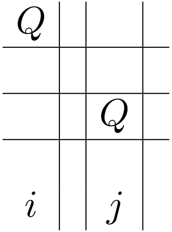


$$Q[j] - Q[i] = i - j$$

Detecting collisions



$$Q[j] - Q[i] = j - i$$



$$Q[j] - Q[i] = i - j$$

- And one more (easy) case

Backtracking Requirements

1. A representation for partial solutions

Backtracking Requirements

1. A representation for partial solutions
2. A procedure to expand a problem into smaller subproblems

Backtracking Requirements

1. A representation for partial solutions
2. A procedure to expand a problem into smaller subproblems
3. A test for partial solutions that returns

Backtracking Requirements

1. A representation for partial solutions
2. A procedure to expand a problem into smaller subproblems
3. A test for partial solutions that returns

   `True` if the solution is complete (Success)

Backtracking Requirements

1. A representation for partial solutions
2. A procedure to expand a problem into smaller subproblems
3. A test for partial solutions that returns

   `True` if the solution is complete (Success)

   `False` if there is no way to complete (Failure)

Backtracking Requirements

1. A representation for partial solutions
2. A procedure to expand a problem into smaller subproblems
3. A test for partial solutions that returns

`True`	if the solution is complete (Success)
`False`	if there is no way to complete (Failure)
`None`	if neither can be quickly determined. (Uncertainty)

Generic Backtracking

```
def backtrack(P0):
    S = [P0]
    while len(S) > 0:
        P = S.pop()
        result = test(P)
        if result == True:
            return P
        elif result == None:
            for R in expand(P):
                S.append(R)
    return False
```

Backtracking for N-Queens: Framework

representation $Q[1...n]$ where $Q[i]$ is row chosen, or `None`.

Backtracking for N-Queens: Framework

representation $Q[1...n]$ where $Q[i]$ is row chosen, or `None`.

expand For some $Q[i] =$ `None`, try $Q[i] = 0...n-1$

Backtracking for N-Queens: Framework

representation $Q[1...n]$ where $Q[i]$ is row chosen, or `None`.

expand For some $Q[i] =$ `None`, try $Q[i] = 0...n-1$

```
def test(Q):
  default = True
  for i in range(len(Q)):
    if Q[i]==None:
      default = None
    else:
      for j in range(i):
        if Q[i] - Q[j] in [0,i-j,j-i]:
          return False
  return default
```

Backtracking for N-Queens: Expand

```
def expand(Q):
  i=0; S=[]
  while Q[i] != None:
    i+=1
  for j in range(len(Q)):
    R=Q[:] # copy
    R[i] = j
    S.append(R)
  return S
```

Backtracking for subset sum

Subset Sum

Given $X \subset \mathbb{Z}^+$, T

Decide Is there a subset of X that sums to T

Backtracking for subset sum

Subset Sum

Given $X \subset \mathbb{Z}^+$, T

Decide Is there a subset of X that sums to T

- If (X, T) has feasible solution Z, for all $y \in X$, either the solution includes y or not.

Backtracking for SubsetSum

```
def SubsetSum(X,T):
    if T == 0:
        return true
    elif T<0 or len(X) == 0:
        return False

    (y,rest) = (X[0],X[1:])
    return SubsetSum(rest, T-y) \
        or SubsetSum(rest,T)
```

The SAT Problem

Conjunctive Normal Form (CNF)

Variables $\{ x_1 ... x_n \}$

Literals $L = \{ x_i, \bar{x}_i \mid \text{variable } x_i \}$

Clause $\{ z_1, ..., z_k \} \subset L$

The SAT Problem

Conjunctive Normal Form (CNF)

Variables $\{ x_1 \ldots x_n \}$

Literals $L = \{ x_i, \bar{x}_i \mid \text{variable } x_i \}$

Clause $\{ z_1, \ldots, z_k \} \subset L$

Propositional Satisfiability (SAT)

Instance Set of clauses S

Question $\exists$ setting of variables to 0, 1 such that each clause has at least one true literal?

SAT Example

$$\{\,\{\,1,2,3\,\},\{\,-1,-2,-3\,\}\,\} = \{\,\{\,x_1,x_2,x_3\,\},\{\,\bar{x}_1,\bar{x}_2,\bar{x}_3\,\}\,\}$$

(A) $=$

Truth Table

x_1	x_2	x_3	A
0	0	0	0
0	0	1	1
0	1	0	
	$\vdots$		

SAT Example

$$\{\,\{\,1, 2, 3\,\}, \{\,-1, -2, -3\,\}\,\} = \{\,\{\,x_1, x_2, x_3\,\}, \{\,\bar{x}_1, \bar{x}_2, \bar{x}_3\,\}\,\}$$

$$\text{(A)} \qquad = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Truth Table

x_1	x_2	x_3	A
0	0	0	0
0	0	1	1
0	1	0	
	$\vdots$		

Backtracking for SAT

representation	(reduced) clauses
test	if empty clause, return `False`. If no clauses, return `True`. Otherwise return `None` (UNKNOWN)
expand	$P_0 = \text{reduce}(P, x_j)$, $P_1 = \text{reduce}(P, \bar{x}_j)$ for some j.

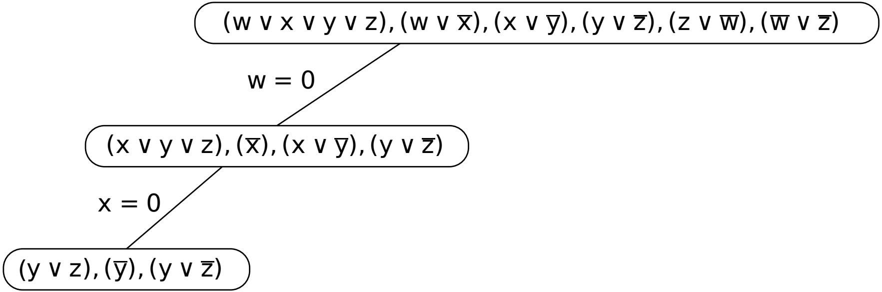
Backtracking SAT Example II

$$(w \vee x \vee y \vee z), (w \vee \overline{x}), (x \vee \overline{y}), (y \vee \overline{z}), (z \vee \overline{w}), (\overline{w} \vee \overline{z})$$

$$w = 0$$

$$(x \vee y \vee z), (\overline{x}), (x \vee \overline{y}), (y \vee \overline{z})$$

Backtracking SAT Example II



Backtracking SAT Example II

$(w \vee x \vee y \vee z), (w \vee \overline{x}), (x \vee \overline{y}), (y \vee \overline{z}), (z \vee \overline{w}), (\overline{w} \vee \overline{z})$

$w = 0$

$(x \vee y \vee z), (\overline{x}), (x \vee \overline{y}), (y \vee \overline{z})$

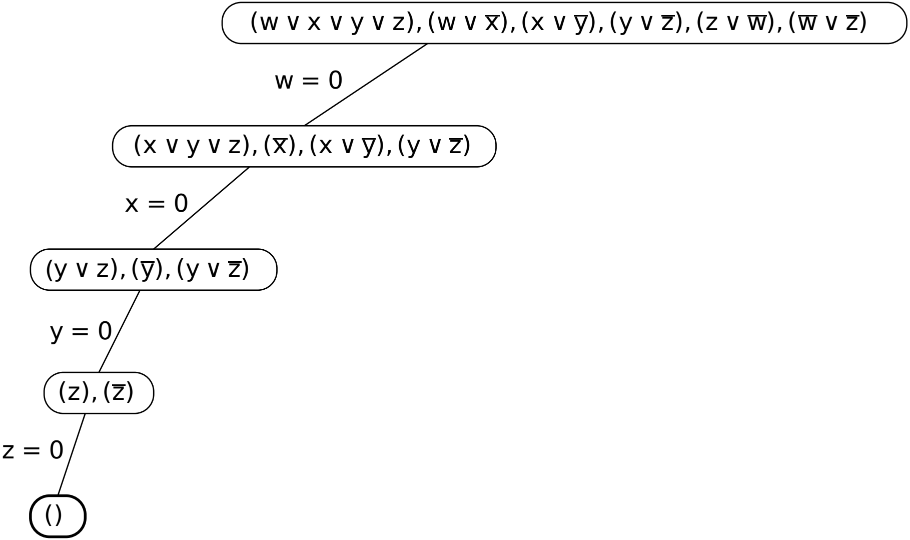
$x = 0$

$(y \vee z), (\overline{y}), (y \vee \overline{z})$

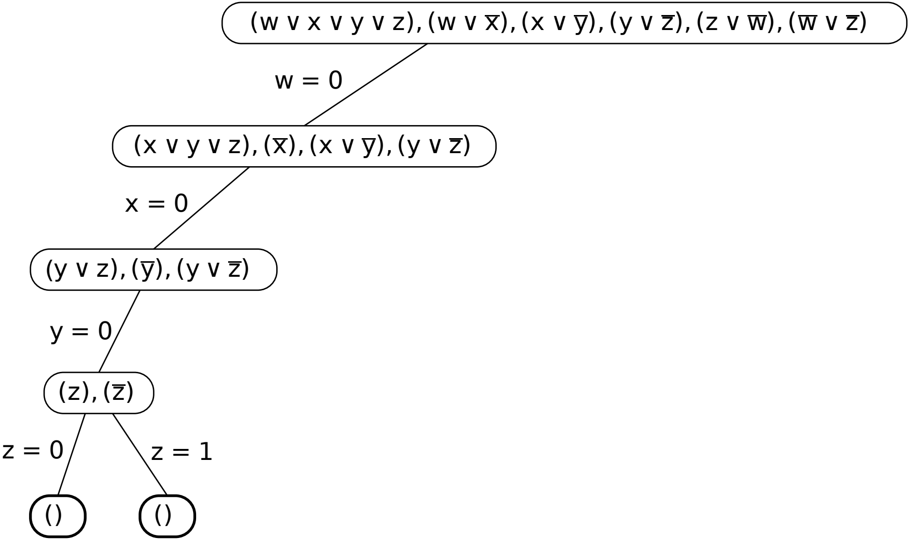
$y = 0$

$(z), (\overline{z})$

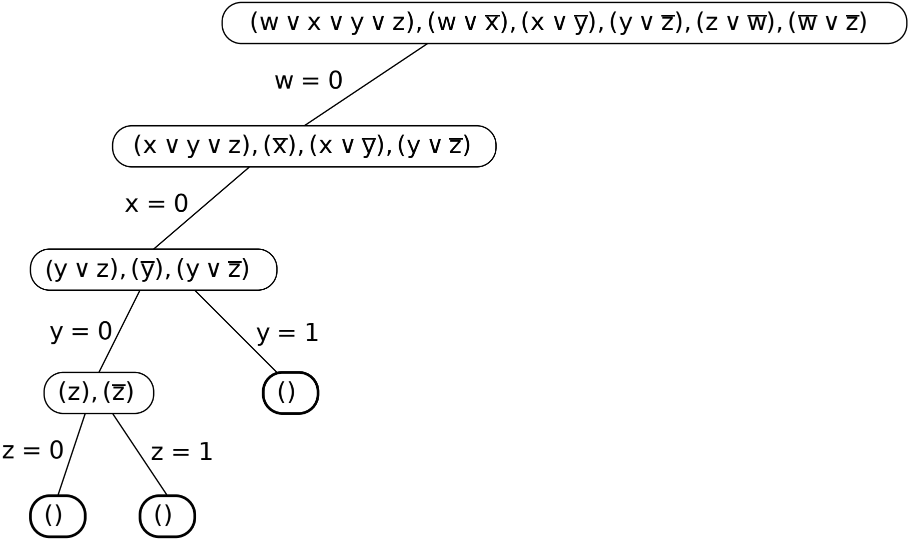
Backtracking SAT Example II



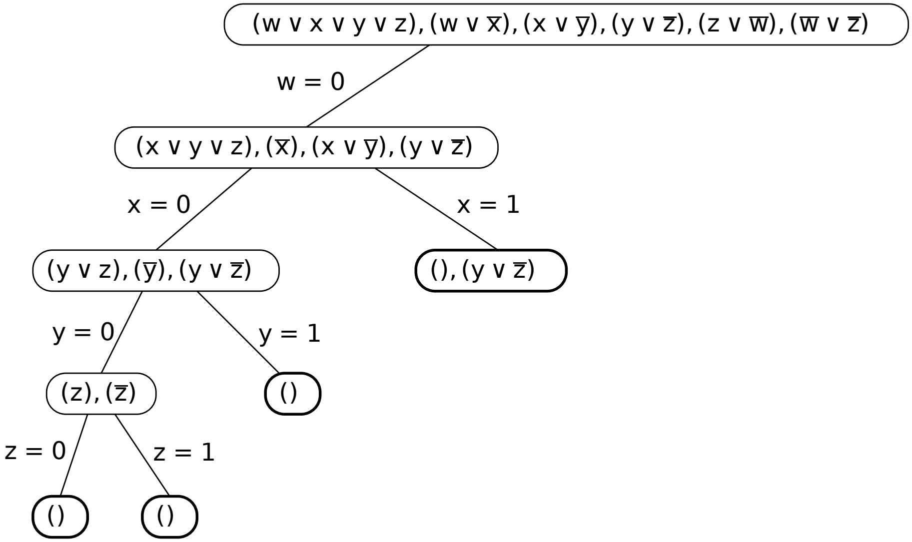
Backtracking SAT Example II



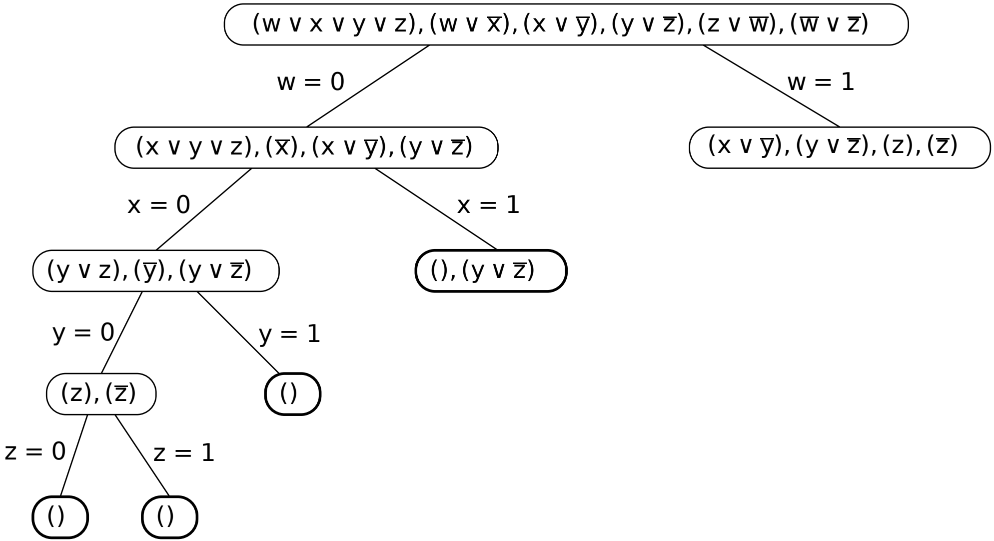
Backtracking SAT Example II



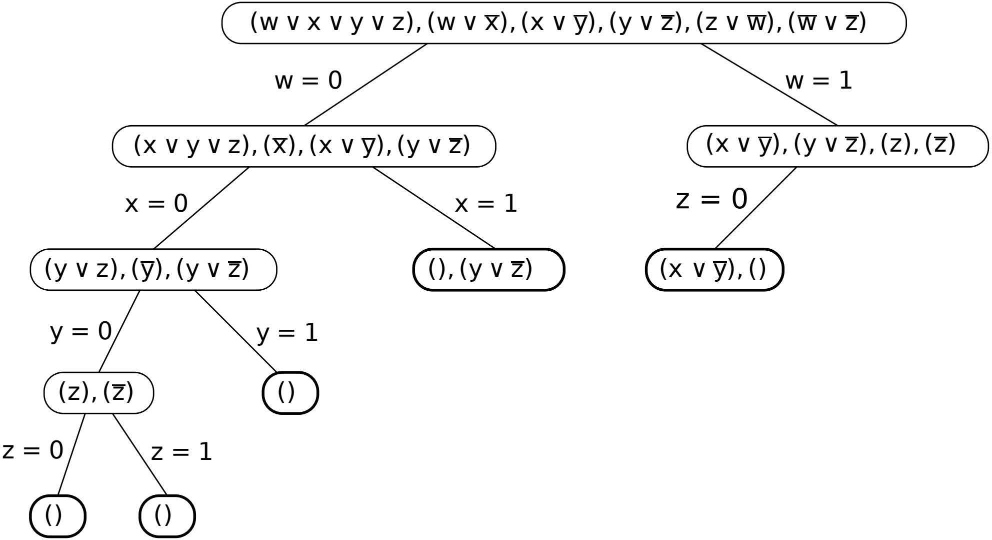
Backtracking SAT Example II



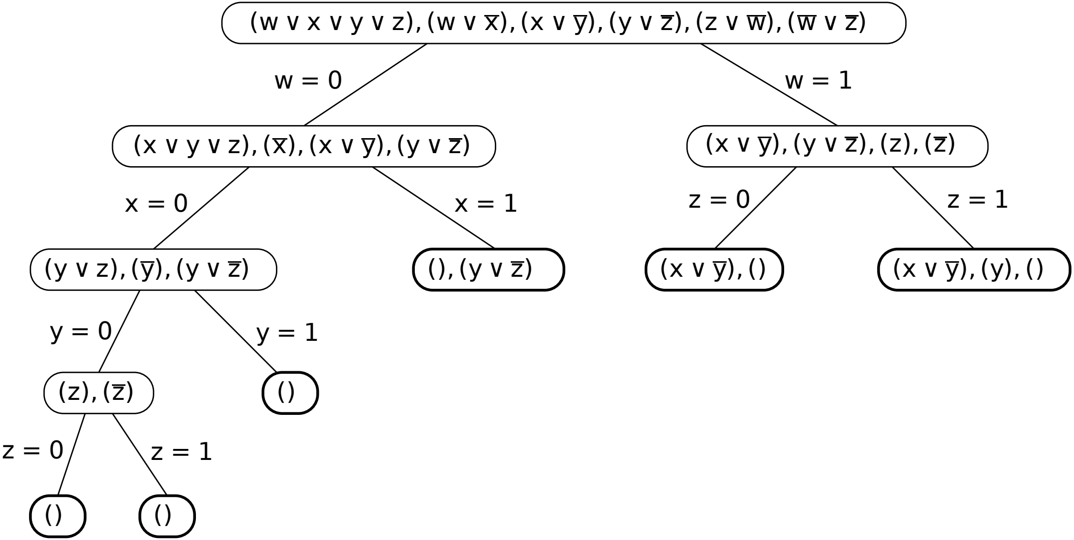
Backtracking SAT Example II



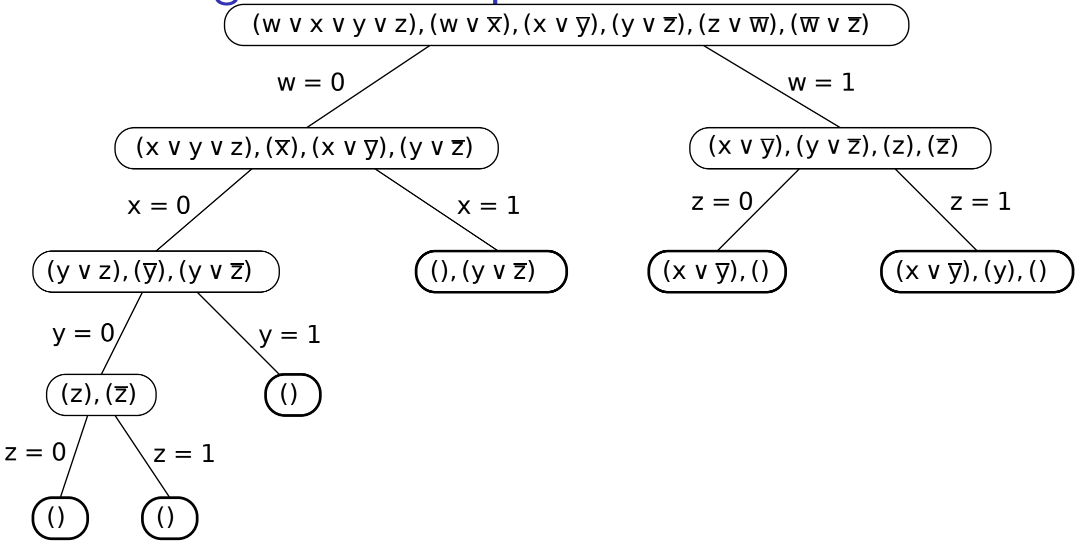
Backtracking SAT Example II



Backtracking SAT Example II



Backtracking SAT Example II



We tried 11 possibilities. Maximum?

Backtracking Sat test

```python
def test(clauses):
    if (len(clauses)) == 0:
        return True
    for clause in clauses:
        if len(clause)==0:
            return False
    return None
```

Backtracking Sat expand

```
def reduce(clauses,literal):
  out=[]
  for C in clauses:
    if not literal in C:
      new=[z for z in C if z != -1*literal]
      out.append(new)
  return out

def expand(clauses):
  j = clauses[0][0]
  return [reduce(clauses,j),
          reduce(clauses,-j)]
```

2SAT

- In 2SAT problem every clause has at most 2 elements

2SAT

- In 2SAT problem every clause has at most 2 elements
- 2SAT is solvable in polynomial time, but not quite trivially.

2SAT

- In 2SAT problem every clause has at most 2 elements
- 2SAT is solvable in polynomial time, but not quite trivially.
- Greedy fails on

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_1 \vee x_4) \wedge (\bar{x}_4 \vee x_5) \wedge (\bar{x}_4 \vee x_6)$$

2SAT

- In 2SAT problem every clause has at most 2 elements
- 2SAT is solvable in polynomial time, but not quite trivially.
- Greedy fails on

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_1 \vee x_4) \wedge (\bar{x}_4 \vee x_5) \wedge (\bar{x}_4 \vee x_6)$$

- to maximize number of clauses satisfied,
  choose $x_1 \leftarrow 1$, $x_4 \leftarrow 0$

2SAT

- In 2SAT problem every clause has at most 2 elements
- 2SAT is solvable in polynomial time, but not quite trivially.
- Greedy fails on

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_1 \vee x_4) \wedge (\bar{x}_4 \vee x_5) \wedge (\bar{x}_4 \vee x_6)$$

- to maximize number of clauses satisfied, choose $x_1 \leftarrow 1$, $x_4 \leftarrow 0$
- solvable with *unit propagation*

Horn SAT

Horn formulas

implication $(z \wedge w \wedge q) \Rightarrow u$. LHS is all positive, RHS one positive literal

negative clauses $(\bar{x} \vee \bar{w} \vee \bar{y})$. All literals negated.

Horn formulas as CNF

- negative clauses are already CNF

Horn formulas as CNF

- negative clauses are already CNF
- implications use the following transformations

$$\left(\bigwedge_{i=1}^{k} x_i\right) \Rightarrow y$$

$$\neg\left(\bigwedge_{i=1}^{k} x_i\right) \vee y$$

$$\left(\bigvee_{i=1}^{k} \bar{x}_i\right) \vee y$$

Horn formulas as CNF

- negative clauses are already CNF
- implications use the following transformations

$$(\bigwedge_{i=1}^{k} x_i) \Rightarrow y$$
$$\neg(\bigwedge_{i=1}^{k} x_i) \vee y$$
$$(\bigvee_{i=1}^{k} \bar{x}_i) \vee y$$

- special CNF with at most one positive literal.

Unit propagation

```
def UnitProp(S):
  Q = [ c for c in S if len(c)==1 ]
  while len(Q)>0:
    z = Q.pop()[0]; T = []
    for C in S:
      C = [j for j in C if j!=-z]
      if len(C)==0: return False
      if len(C)==1: Q.append(C)
      if not z in C: T.append(C)
    if len(T)==0: return True
    S = T
  return S
```

Unit propagation (with assignment)

```
def UnitProp(S):
  Q = [ c for c in S if len(c)==1 ]; V = []
  while len(Q)>0:
    z = Q.pop()[0]; T = []
    V.append(z)
    for C in S:
      C = [j for j in C if j!=-z]
      if len(C)==0: return (False,V)
      if len(C)==1: Q.append(C)
      if not z in C: T.append(C)
    if len(T)==0: return (True,V)
    S = T
  return S
```

Solving Horn SAT with Unit Propagation

1. Apply unit propagation
2. If no contradiction is detected, set the remaining variables to false.

Claim 1

If the procedure detects a contradiction, the instance is unsatisfiable

Solving Horn SAT with Unit Propagation

1. Apply unit propagation
2. If no contradiction is detected, set the remaining variables to false.

Claim 1

If the procedure detects a contradiction, the instance is unsatisfiable

Claim 2

If no contradiction is detected, the resulting assignment is valid

Solving Horn SAT with Unit Propagation

1. Apply unit propagation
2. If no contradiction is detected, set the remaining variables to false.

Claim 1

If the procedure detects a contradiction, the instance is unsatisfiable

Claim 2

If no contradiction is detected, the resulting assignment is valid

Proof

any remaining clause has at least one negative literal

Solving 2SAT with Unit Propagation

Claim

Applying `reduce`, followed by unit propagation, always yields either a contradiction (empty clause), or a subset of the original clauses.

Solving 2SAT with Unit Propagation

```
def two_sat(clauses):
  if len(clauses) == 0:
      return True
  j = clauses[0][0]
  R0 = UnitProp(reduce(clauses,-j))
  R1 = UnitProp(reduce(clauses,j))
  if True in [ R0, R1 ]: return True
  if R0 == False and R1 == False: return False
  if R0 == False:
      return two_sat(R1)
  else:
      return two_sat(R0)
```

2SAT with Unit Propagation

Correctness

- if `two_sat` returns `False`, the formula is unsatisfiable.
- if `two_sat` returns `True`, the formula is satisfiable.

By induction on number of clauses; base case: no clauses. Suppose the function is correct for $j < k$ clauses.

return `False`

directly both choices for x_j led to a contradiction.

indirectly we found a contradiction in some subset of the original clauses.

2SAT with Unit Propagation

Correctness

- if `two_sat` returns `False`, the formula is unsatisfiable.
- if `two_sat` returns `True`, the formula is satisfiable.

By induction on number of clauses; base case: no clauses. Suppose the function is correct for $j < k$ clauses.

return `True`

directly
: one of our choices for x_j, along with unit prop., satisfied all clauses.

indirectly
: all clauses containing x_j removed. remaining clauses satisfiable by induction.

Backtracking Sat with Unit Propagation

```python
def backtrack(P0):
    S = [P0]
    while len(S) > 0:
        P = S.pop()
        P1 = UnitProp(P)
        if P1 != False:
            if P1 == True:
                return P1
            else:
                for R in expand(P1):
                    S.append(R)
    return False
```