EE 677 VLSI CAD Project: DPLL Algorithm Implementation

Rohan Pathak
Electrical Department
IIT Bombay
15D070006
15D070006@iitb.ac.in

Pawan Khanna
Electrical Department
IIT Bombay
15D100015
15D100015@iitb.ac.in

***Abstract*—In computer science, the DavisPutnamLogemannLoveland (DPLL) algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.**

***Index Terms*—VLSI, CAD, DPLL, SAT, satisfiability problem**

I. Introduction

The SAT problem is important both from theoretical and practical points of view. In complexity theory it was the first problem proved to be NP-complete, and can appear in a broad variety of applications such as model checking, automated planning and scheduling, and diagnosis in artificial intelligence. Another application that often involves DPLL is automated theorem proving or satisfiability modulo theories (SMT), which is a SAT problem in which propositional variables are replaced with formulas of another mathematical theory.

II. The Algorithm

So our aim is to find a possible solution for the given problem statement in the form of Product of Sums (PoS). We use the fact that to get the output as 1, each individual clause needs to be 1 as well.
The DPLL algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value. The simplification step essentially removes all clauses that become true under the assignment from the formula, and all literals that become false from the remaining clauses.
Following are the 2 heurestic steps we follow at each recursion step.

A. Unit Propagation

If a clause is a unit clause, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary value to make this literal true. Thus, no choice is necessary. In practice, this often leads to deterministic cascades of units, thus avoiding a large part of the naive search space.

B. Pure literal elimination

If a propositional variable occurs with only one polarity in the formula, it is called pure. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted.

Unsatisfiability of a given partial assignment is detected if one clause becomes empty, i.e. if all its variables have been assigned in a way that makes the corresponding literals false. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. Unsatisfiability of the complete formula can only be detected after exhaustive search.

III. Input and Output Format

A. Input Format

Input should have each clause on a new line. Each new variable should be represented by a single alphabet. To denote the inverse of the signal, the sign '~' should be used and the sign '+' should be used to denote OR between the signals.

B. Output Format

If a solution is possible then, we output 'True, one of the possible solution is :' followed by the variables and their values which satisfy the given equation. The values for the variables can have 3 values. The values '0' and '1' do not need any explanations. The third value is'-1', it denotes the 'Don't Care' condition. That specific variable can have any value '0' or '1' and it won't affect the outcome if the other variables have the given values. Now if the equation can never be satisfied, we print 'False, no solution possible'.

IV. Implementation

We maintain a global variable namely 'values' and 'variables'. The variable 'variables' is a list of list containing all the variables at the 1st dimension with its value at the 2nd dimension. Following are the functions we created

A. *inputpos*

Short for 'Input Product of Sums'. This function reads the input from the file 'inputfile.txt' and return the equation called 'phi' which is a list of list. We read the variable 'a' as [a,1] and read the variable '~a' as [a,0].

B. *clause*$_v alue(phi)$

Takes the equation 'phi' as the input and initialises all the clause values to '-1' i.e their value as not decided.

C. *assign(variable,value)*

The input 'value' is a list where 'value[0]' is the variable to be assigned the value 'value[1]'. And then update it for all the variables in the list 'variable'.

D. *list of var(phi)*

Gives the list of all different variables in the equation 'phi'.

E. *unit propagation(phi,values)*

Finds a unit clause and then returns the variable in it. It also cross checks that the value of that clause is '-1' i.e it is not already taken before.

F. *pure clause(phi,values)*

Extracts all the variables appearing in pure form and returns a lis of lis contianing the variables and the form in which they are present.

G. *update val(var upd,val upd,phi upd)*

Updates the value of the variable from 'val upd' in the equation 'phi upd' as well as 'var upd' list.

H. *dpll(phi,value)*

Implements the DPLL algorithm using all the above functions.

I. *Main Function*

Reads the input using the 1st function above and then initialises the variables 'values' and 'variables' and then calls the DPLL algorithm. If a solution exists, it then prints the soltion otherwise prints not possible.

References

[1] $https : //en.wikipedia.org/wiki/DPLL_algorithm$
[2] Lecture notes on DPLL

APPENDIX WITH CODE

```python
import numpy as np
import copy

variables = []  #Store the variables
#################### READ INPUT ####################
def inputpos():
  #Input is read from the file
  file1 = open("input_file.txt","r")
  L = file1.readlines()
  file1.close()

  count_clauses = len(L)  #Number of clauses
  phi = []        #Stores the whole equation in a 3d format
  for i in range(0,count_clauses):
    temp_clause = []
    tilde = False   #Initialising tilde as not present
    for j in range(0,len(L[i])):
      temp_literal = []
      if(L[i][j] == '~'):
        tilde = True  #Tilde is present
      elif (L[i][j] != '+' and L[i][j] != '\n'):  #Variable is present
        if tilde == True:
          temp_literal.append(L[i][j])
          temp_literal.append(0)
          temp_clause.append(temp_literal)
          tilde = False
        else:
          temp_literal.append(L[i][j])
          temp_literal.append(1)
          temp_clause.append(temp_literal)
    phi.append(temp_clause)
  return phi

#################### INITIALISES CLAUSE VALUES TO -1####################
def clause_value(phi):
  return [-1]*len(phi)

#################### ASSIGNS VALUE TO VARIABLE ####################
def assign(variable,value):
  for i in range(0,len(variable)):
    if(variable[i][0]==value[0]):
      variable[i][1]=value[1]
      break

#################### UPDATES THE VARIABLES LIST ####################
def list_of_var(phi):
  list_of_var = []
  variables = []
  for i in range(0,len(phi)):#For each clause in phi
    for j in range(0,len(phi[i])):#For each variable in clause
      temp_var = phi[i][j][0]   #Temporary variable
      if not(temp_var in list_of_var):  #if the temp_var is already not added in the pure variables list
        list_of_var.append(temp_var)
        variables.append([temp_var,-1])
  return variables

#################### UNIT PROPAGATION ####################
def unit_propagation(phi,values):
  length_phi = len(phi)
  var = []
  i=0
  while i<length_phi:   #Carry on till we find the 1st unit clause
    length_clause = len(phi[i])
    if length_clause ==1  and values[i]==-1:#If unit clause value is not decided
      var = phi[i]
      break
    i = i+1

  return var

#################### EXTRACTS THE PURE VARIABLES ####################
def pure_clause(phi,values):
```

```
    pure_var = []
    for i in range(0,len(phi)):
        for j in range(0,len(phi[i])):
            if(values[i]==-1):
                temp_var = phi[i][j][0]
                temp_val = phi[i][j][1]
                count = 0 #Number of times the temperory variable is used in opposite value
                for i1 in range(0,len(phi)):
                    for j1 in range(0,len(phi[i1])):
                        if(values[i1]==-1):
                            if temp_var == phi[i1][j1][0]:
                                if temp_val^phi[i1][j1][1]==1:
                                    count = count+1
                #If count==0, then it means it is a pure clause
                if count==0:
                    if not([temp_var,temp_val] in pure_var):  #if the temp_var is already not added in the pure
                        variables list
                        temp = [temp_var,temp_val]
                        pure_var.append(temp)

    return pure_var

#################### UPDATES THE VARIABLES IN THE CLAUSES ####################
def update_val(var_upd,val_upd,phi_upd):
    del_list = []
    for i in range(0,len(phi_upd)):
        if val_upd[i]==-1:
            for j in range(0,len(phi_upd[i])):
                if var_upd[0] == phi_upd[i][j][0] and phi_upd[i][j][1] == 1 and var_upd[1]==1:
                    val_upd[i] = 1
                elif var_upd[0] == phi_upd[i][j][0] and phi_upd[i][j][1] == 0 and var_upd[1]==0:
                    val_upd[i] = 1
                elif var_upd[0] == phi_upd[i][j][0] and phi_upd[i][j][1] == 1 and var_upd[1]==0:
                    del_list.append(j)
                elif var_upd[0] == phi_upd[i][j][0] and phi_upd[i][j][1] == 0 and var_upd[1]==1:
                    del_list.append(j)
                #else:
                #   print(i,j,"no")
            while(del_list):
                del phi_upd[i][del_list.pop()]
            if(not phi_upd[i]):
                val_upd[i] = 0

#################### IMPLEMENT THE DPLL ALGORITHM ####################
def dpll(phi,value):
    global variables

    #UNIT PROPAGATION
    unit_var = unit_propagation(phi,value)
    if (unit_var):
        update_val(unit_var,value,phi)
        assign(variables,unit_var)

    #PURE CLAUSE
    pure_var = pure_clause(phi,value)

    #UPDATE
    for i in range(0,len(pure_var)):
        assign(variables,pure_var[i])
        update_val(pure_var[i],value,phi)

    #Copying phi and values so that we can call recursion since
    #python implements calling by reference
    new_phi1= copy.deepcopy(phi)
    new_value1 = copy.deepcopy(value)
    new_phi2= copy.deepcopy(phi)
    new_value2 = copy.deepcopy(value)

    sat = 1
    for i in range(0,len(value)):
        if value[i] == 0:
            sat = 0
            break
        elif value[i]== -1:
```

```
      sat = -1
      break

  if(sat==1):
    return True    #satisfiable
  elif(sat==0):
    return False

  unvar = '0'   #Unassigned variable
  for i in range(0,len(variables)):
    if(variables[i][1]==-1):
      unvar = variables[i][0]     #var to be assigned
      break

  if(unvar != 0): #If we have assigned a variable to unvar
    update_val([unvar,0],new_value1,new_phi1)
    #assign unvar = 0
    assign(variables,[unvar,0])
    sat = 1
    for i in range(0,len(new_value1)):
      if new_value1[i] == 0:
        sat = 0
        break
      elif new_value1[i]== -1:
        sat = -1
        break
    if(sat==1):
      return True    #satisfiable
    elif(sat==-1):
      if(dpll(new_phi1,new_value1) == True):
        return True

    update_val([unvar,1],new_value2,new_phi2) #assign unvar = 1
    assign(variables,[unvar,1])
    sat = 1
    for i in range(0,len(new_value2)):
      if new_value2[i] == 0:
        sat = 0
        break
      elif new_value2[i]== -1:
        sat = -1
        break
    if(sat==1):
      return True    #satisfiable
    elif(sat==0):
      return False
    elif(sat==-1):
      if(dpll(new_phi2,new_value2) == True):
        return True
      else:
        return False

#################### MAIN FUNCTION ####################
if __name__== "__main__":
  phi = inputpos()        #Read the input
  values = clause_value(phi)    #Initialise the values to -1
  variables = list_of_var(phi)  #Gives the list of variables

  if dpll(phi,values):      #If the solution exists
    print("True, one of the possible soultion is:")
    print(variables)
  else:              #If no solution exists
    print("False, no soultion possible")
```