NGUYEN_MEGA_RABARY_TP_SAT

January 10, 2019

1 TP Scheduling with SAT

1.0.1 Trinôme : Adrian MEGA - Duc Hau NGUYEN - Anais RABARY

```
In [1]: from satire import *
        #import numpy as np
```

- Paramètre

```
In [155]: debug = True
          # Output filename
          cnf_encodeDomain = 'encodeDomain.cnf'
          cnf_encodeLess = 'encodeLess.cnf'
          cnf_encodeGeneralLess = 'encodeGeneralLess.cnf'
          cnf_encodePrecedence = 'encodePrecedence.cnf'
          cnf_encodeImpliesPrecedence = 'encodeImpliesPrecedence.cnf'
```

```
In [124]: def write_cnf(nb_var, nb_clause, clauses, cnf_filename):
              '''
              write_cnf write down result
              '''
              with open(cnf_filename,"w")  as _out:

                  # Comment
                  _out.write('c ' + cnf_filename + '\n')
                  _out.write('c \n')

                  # Heading
                  _out.write('p cnf ' + str(nb_var) + ' ' + str(nb_clause) + '\n')

                  # Clauses
                  for c in clauses:
                      _out.write(c + ' 0\n')

                  if debug:
                      print "Written to " + cnf_filename
```

```
In [5]: def run_solver(solver):
            outcome = solver.restartSearch()
            print 'Satisfiable' if outcome == TRUE else 'Unsatisfiable' if outcome == FALSE els
            print solver.getStatistics()
```

1.1 Part 2 - SAT Encoding

- Implement a function in python encodeDomain (a, b) that encodes a domain $[a, b]$ using the order encoding.

```
In [95]: def encodeDomain(a, b):
             assert a <= b, 'Expected a <= b, got ' + a +' <= '+ b
             n = b - a + 1  # nb variable
             nb_clause = (n*(n-1))/2 + 1 # Clauses for direct Encoding

             clauses = ['']

             for i in range(1, n+1):

                 clauses[0] += str(i) + ' '

                 for j in range(i+1, n+1):
                     clauses.append(str(-i) + ' ' + str(-j))

             if debug:
                 print('p cnf ' + str(n) + ' ' + str(nb_clause))
                 for c in clauses:
                     print(c)

             write_cnf(n, nb_clause, clauses, cnf_encodeDomain)
```

```
In [96]: encodeDomain(2,5)
         solver = Solver()
         solver.readDimacs(cnf_encodeDomain)
         run_solver(solver)
```

```
p cnf 4 7
1 2 3 4
-1 -2
-1 -3
-1 -4
-2 -3
-2 -4
-3 -4
Written to encodeDomain.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 0

  infer -1 (reason=None) [1]
  -1
   infer -4 (reason=None) [2]
   -1 -4
```

```
    infer -3 (reason=None) [3]
    infer 2 (reason=(3 2 4 1)) [3]
    -1 2 -3 -4
     infer -6 (reason=None) [4]
     -1 2 -3 -4 -6
      infer -7 (reason=None) [5]
      -1 2 -3 -4 -6 -7
       infer -5 (reason=None) [6]
       -1 2 -3 -4 -5 -6 -7
        infer -8 (reason=None) [7]
        -1 2 -3 -4 -5 -6 -7 -8
1
Satisfiable
number of choices = 7
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 8
cpu time = 6
```

- Let X and Y be two integer variables with a domain equal to $[a, b]$ where $a < b \in \mathbb{N}^*$. Implement a function **encodeLess(a, b)** that encodes the constraint $X \leq Y$.

```
In [112]: def encodeLess(a, b):
              assert a <= b, 'Expected a <= b, got ' + str(a) +' <= '+ str(b)
              n = (b - a + 1)
              nb_variable = n*2
              nb_clauses = 2 * (n - 1) + \
                           n + \
                           2
              # 2(n-1) Clause for ordering x and y + n clauses for X <= Y

              X = []
              Y = []
              clauses = [''] * nb_clauses

              # Regroup variables
              for i in range(n):
                  X.append(i + 1)
                  Y.append(i + 1 + n)

              # No all 0 solution
              for i in range(n):
                  clauses[0] += str(X[i]) + ' '
                  clauses[1] += str(Y[i]) + ' '

              # Clauses for ordering domain
              for i in range(n-1):
```

```
            # X has n - 1 clauses, Y start from n
            clauses[2 + i] =  str(-X[i]) + ' ' + str(X[(i+1)])
            clauses[2 + i + n - 1] = str(-Y[i]) + ' ' + str(Y[(i+1)])

        # Clauses for x[i] = 0 -> y[j] = 0
        for i in range(n):
            clauses[2 + 2*n - 2 + i] = str(X[i]) + ' ' + str(-Y[i])

        if debug :
            for c in clauses:
                print(c)

        write_cnf(n, nb_clauses, clauses, cnf_encodeLess)
```

```
In [113]: encodeLess(1,3)
          solver = Solver()
          solver.readDimacs(cnf_encodeLess)
          run_solver(solver)
```

```
1 2 3
4 5 6
-1 2
-2 3
-4 5
-5 6
1 -4
2 -5
3 -6
Written to encodeLess.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 1

  infer -2 (reason=None) [1]
  infer -5 (reason=(2 -5)) [1]
  infer -1 (reason=(-1 2)) [1]
  infer -4 (reason=(-4 5)) [1]
  infer 3 (reason=(1 3 2)) [1]
  infer 6 (reason=(4 6 5)) [1]
  -1 -2 3 -4 -5 6
   infer -8 (reason=None) [2]
   -1 -2 3 -4 -5 6 -8
    infer -7 (reason=None) [3]
    -1 -2 3 -4 -5 6 -7 -8
1
Satisfiable
number of choices = 3
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 8
```

```
cpu time = 6
```

- Let X and Y be two integer variables such that $\mathbb{D}_x = [a, b]$ and $\mathbb{D}_y = [c, d]$ where $a < b \in \mathbb{N}$ and $c < d \in \mathbb{N}$. Implement a function **encodeGeneralLess(a, b, c, d)** that encodes the constraint $X \leq Y$.

```
In [120]: def encodeGeneralLess (a, b, c, d):
              assert a <= b, 'Expected a <= b, got ' + str(a) +' <= '+ str(b)
              assert c <= d, 'Expected c <= d, got ' + str(c) +' <= '+ str(d)

              nb_var_X = b - a + 1
              nb_var_Y = d - c + 1

              nb_variable = nb_var_X + nb_var_Y
              nb_clauses = (nb_var_X - 1) + \
                           (nb_var_Y - 1) + \
                           max(a - c, 0) + \
                           (min(b,d) - max(a, c) + 1) +\
                           2

              # Clauses for ordering X + Y + clause for x <= y
              X = []
              Y = []
              clauses = []

              # Regroup variables
              for i in range(nb_var_X):
                  X.append(i + 1)
              for i in range(nb_var_Y):
                  Y.append(nb_var_X + i + 1)

              # No all 0 solution
              valid_x = ''
              for i in range(nb_var_X):
                  valid_x += str(X[i]) + ' '
              clauses.append(valid_x)

              valid_y = ''
              for i in range(nb_var_Y):
                  valid_y += str(Y[i]) + ' '
              clauses.append(valid_y)

              # Clauses for ordering domain
              for i in range(nb_var_X - 1):
                  clauses.append( str(-X[i]) + ' ' + str(X[(i+1)]) )
              for i in range(nb_var_Y - 1):
                  clauses.append( str(-Y[i]) + ' ' + str(Y[(i+1)]) )
              # Clauses for X <= Y
              for i in range(a - c):
                  # if a > c then Y_i = 0 for i in c..a-1
                  clauses.append(str( -Y[i] ))

              if a <= d :
                  decal_a_c = a - c
                  for i in range(max(c - a, 0), min(nb_var_Y, nb_var_X)):

                      # For those in [max(a,c) ... min(b, d)], X_i=0 --> Y_i = 0
                      print 'decal_a_c + i =', decal_a_c + i, ' len Y = ', len(Y)
                      clauses.append(str(X[i]) + ' ' + str(-Y[decal_a_c + i]))

              if debug:
                  for c in clauses:
                      print(c)
                  print 'Nb clause = ', len(clauses),' || Expected = ', nb_clauses

              write_cnf(nb_variable, nb_clauses, clauses, cnf_encodeGeneralLess)
```

```
In [122]: print("=== Test X and Y in [1..3] ===")
          encodeGeneralLess(1, 3, 1, 3)
          solver = Solver()
          solver.readDimacs(cnf_encodeGeneralLess)
          run_solver(solver)

          print("\n=== Test X in [1..3] and Y in [2..4] ===")
          encodeGeneralLess(1, 3, 2, 4)
          solver = Solver()
          solver.readDimacs(cnf_encodeGeneralLess)
          run_solver(solver)

          print("\n=== Test X in [2..4] and Y in [1..5] ===")
          encodeGeneralLess(2, 4, 1, 5)
          solver = Solver()
          solver.readDimacs(cnf_encodeGeneralLess)
          run_solver(solver)

          print("\n=== Test X in [3..4] and Y in [1..2], expected Unsatisfiable ===")
          encodeGeneralLess(3, 4, 1, 2)
          solver = Solver()
          solver.readDimacs(cnf_encodeGeneralLess)
          run_solver(solver)
```

```
=== Test X and Y in [1..3] ===
decal_a_c + i = 0  len Y =  3
decal_a_c + i = 1  len Y =  3
decal_a_c + i = 2  len Y =  3
1 2 3
4 5 6
-1 2
-2 3
-4 5
-5 6
1 -4
2 -5
3 -6
Nb clause =  9  || Expected =  9
Written to encodeGeneralLess.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 0

  infer -2 (reason=None) [1]
  infer -5 (reason=(2 -5)) [1]
  infer -1 (reason=(-1 2)) [1]
  infer -4 (reason=(-4 5)) [1]
  infer 3 (reason=(1 3 2)) [1]
  infer 6 (reason=(4 6 5)) [1]
  -1 -2 3 -4 -5 6
   infer -8 (reason=None) [2]
   -1 -2 3 -4 -5 6 -8
    infer -7 (reason=None) [3]
    -1 -2 3 -4 -5 6 -7 -8
1
Satisfiable
number of choices = 3
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 8
cpu time = 5

=== Test X in [1..3] and Y in [2..4] ===
decal_a_c + i = 0  len Y =  3
decal_a_c + i = 1  len Y =  3
1 2 3
4 5 6
-1 2
-2 3
-4 5
-5 6
2 -4
3 -5
Nb clause =  8  || Expected =  8
Written to encodeGeneralLess.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 0
```

```
  infer -2 (reason=None) [1]
  infer -4 (reason=(2 -4)) [1]
  infer -1 (reason=(-1 2)) [1]
  infer 3 (reason=(1 3 2)) [1]
  -1 -2 3 -4
   infer -5 (reason=None) [2]
   infer 6 (reason=(6 5 4)) [2]
   -1 -2 3 -4 -5 6
    infer -8 (reason=None) [3]
    -1 -2 3 -4 -5 6 -8
     infer -7 (reason=None) [4]
     -1 -2 3 -4 -5 6 -7 -8
1
Satisfiable
number of choices = 4
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 8
cpu time = 2

=== Test X in [2..4] and Y in [1..5] ===
decal_a_c + i = 1  len Y =  5
decal_a_c + i = 2  len Y =  5
decal_a_c + i = 3  len Y =  5
1 2 3
4 5 6 7 8
-1 2
-2 3
-4 5
-5 6
-6 7
-7 8
-4
1 -5
2 -6
3 -7
Nb clause =  12  || Expected =  12
Written to encodeGeneralLess.cnf
 infer -4 (reason=None) [0]
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 1
 -4
  infer -7 (reason=None) [1]
  infer -6 (reason=(-6 7)) [1]
  infer -5 (reason=(-5 6)) [1]
  infer 8 (reason=(8 5 6 7 4)) [1]
  -4 -5 -6 -7 8
   infer -2 (reason=None) [2]
   infer -1 (reason=(-1 2)) [2]
   infer 3 (reason=(1 3 2)) [2]
   -1 -2 3 -4 -5 -6 -7 8
1
Satisfiable
number of choices = 2
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 8
cpu time = 4

=== Test X in [3..4] and Y in [1..2], expected Unsatisfiable ===
1 2
3 4
-1 2
-3 4
-3
-4
Nb clause =  6  || Expected =  6
Written to encodeGeneralLess.cnf
 infer -3 (reason=None) [0]
 infer -4 (reason=None) [0]
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 1
 -3 -4
0
Unsatisfiable
number of choices = 0
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 1
cpu time = 1
```

Let X and Y be two integer variables such that $\mathbb{D}_x = [a, b]$ and $\mathbb{D}_y = [c, d]$ where $a < b \in \mathbb{N}^*$ and $c < d \in \mathbb{N}$. Let $k \in \mathbb{N}$. Implement a function **encodePrecedence(a, b, c, d, k)** that encodes the constraint $X + k \leq Y$.

```
In [143]: def encodePrecedence (a, b, c, d, k):
              assert a <= b, 'Expected a <= b, got ' + str(a) +' <= '+ str(b)
              assert c <= d, 'Expected c <= d, got ' + str(c) +' <= '+ str(d)
              assert k >= 0, 'Expected k >= 0, got ' + str(k)

              born_sup = max(b + k, d)

              X = []
              Y = []
              clauses = []

              # Regroup variables
```

```
for i in range(born_sup):
    X.append(i + 1)
for i in range(born_sup, 2*born_sup):
    Y.append(i + 1)

# No all 0 solution
valid_x = ''
for i in range(a, b + 1):
    valid_x += str(X[i - 1]) + ' '
clauses.append(valid_x)

valid_y = ''
for i in range(c, d + 1):
    valid_y += str(Y[i - 1]) + ' '
clauses.append(valid_y)

# Clauses for ordering domain
for i in range(born_sup-1):
    clauses.append(
        str(-X[i]) + ' ' + str(X[(i+1)])
    )

for i in range(born_sup-1):
    clauses.append(
        str(-Y[i]) + ' ' + str(Y[(i+1)])
    )

# Clauses for X + k <= Y
for i in range(c, a + k):
    # if a + k > c then Y[c..a+k] = 0
    clauses.append(
        str(-Y[i-1])
    )

for i in range(max(c, a + k), min(d, b+k)+1):
    clauses.append(
        str(X[i-1]) + ' ' + str(-Y[(i-1)])
    )

if debug:
    for c in clauses:
        print(c)
    #print 'Nb clause = ', len(clauses),' || Expected = ', nb_clauses

nb_variable = 2 * born_sup
nb_clauses = len(clauses)
write_cnf(nb_variable, nb_clauses, clauses, cnf_encodePrecedence)
```

```
In [164]: print("=== Test X and Y in [1..3], k = 0 ===")
          encodePrecedence(1, 3, 1, 3, 0)
          solver = Solver()
          solver.readDimacs(cnf_encodePrecedence)
          run_solver(solver)

          print("=== Test X in [1..3], Y in [2..4], k = 0 ===")
          encodePrecedence(1, 3, 2, 4, 0)
          solver = Solver()
          solver.readDimacs(cnf_encodePrecedence)
          run_solver(solver)

          print("=== Test X in [1..3], Y in [2..4], k = 1 ===")
          encodePrecedence(1, 3, 2, 4, 1)
          solver = Solver()
          solver.readDimacs(cnf_encodePrecedence)
          run_solver(solver)

          print("=== Test X in [1..2], Y in [3..4], k = 3 ===")
          encodePrecedence(4, 5, 1, 2, 1)
          solver = Solver()
          solver.readDimacs(cnf_encodePrecedence)
          run_solver(solver)
          print("Expected : Unsatisfiable")
```

```
=== Test X and Y in [1..3], k = 0 ===
1 2 3
4 5 6
-1 2
-2 3
-4 5
-5 6
1 -4
2 -5
3 -6
Written to encodePrecedence.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 0

  infer -2 (reason=None) [1]
  infer -5 (reason=(2 -5)) [1]
  infer -1 (reason=(-1 2)) [1]
  infer -4 (reason=(-4 5)) [1]
  infer 3 (reason=(1 3 2)) [1]
  infer 6 (reason=(4 6 5)) [1]
  -1 -2 3 -4 -5 6
   infer -8 (reason=None) [2]
   -1 -2 3 -4 -5 6 -8
    infer -7 (reason=None) [3]
    -1 -2 3 -4 -5 6 -7 -8
1
Satisfiable
number of choices = 3
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 8
cpu time = 2
=== Test X in [1..3], Y in [2..4], k = 0 ===
1 2 3
6 7 8
-1 2
-2 3
-3 4
-5 6
-6 7
-7 8
2 -6
3 -7
Written to encodePrecedence.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 1

  infer -3 (reason=None) [1]
  infer -7 (reason=(3 -7)) [1]
  infer -2 (reason=(-2 3)) [1]
  infer -6 (reason=(-6 7)) [1]
  infer -1 (reason=(-1 2)) [1]
  -1 -2 -3 -6 -7
 infer 3 (reason=(3)) [0]
 infer 4 (reason=(-3 4)) [0]
 3 4
  infer -2 (reason=None) [1]
  infer -6 (reason=(2 -6)) [1]
  infer -1 (reason=(-1 2)) [1]
  infer -5 (reason=(-5 6)) [1]
  -1 -2 3 4 -5 -6
   infer -7 (reason=None) [2]
   infer 8 (reason=(7 8 6)) [2]
   -1 -2 3 4 -5 -6 -7 8
1
Satisfiable
number of choices = 3
number of learnt clauses = 0
number of conflicts = 1
number of propagations = 11
cpu time = 7
=== Test X in [1..3], Y in [2..4], k = 1 ===
1 2 3
```

```
6 7 8
-1 2
-2 3
-3 4
-5 6
-6 7
-7 8
2 -6
3 -7
4 -8
Written to encodePrecedence.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 0

  infer -3 (reason=None) [1]
  infer -7 (reason=(3 -7)) [1]
  infer -2 (reason=(-2 3)) [1]
  infer -6 (reason=(-6 7)) [1]
  infer -1 (reason=(-1 2)) [1]
  -1 -2 -3 -6 -7
 infer 3 (reason=(3)) [0]
 infer 4 (reason=(-3 4)) [0]
 3 4
  infer -2 (reason=None) [1]
  infer -6 (reason=(2 -6)) [1]
  infer -1 (reason=(-1 2)) [1]
  infer -5 (reason=(-5 6)) [1]
  -1 -2 3 4 -5 -6
   infer -7 (reason=None) [2]
   infer 8 (reason=(7 8 6)) [2]
   -1 -2 3 4 -5 -6 -7 8
1
Satisfiable
number of choices = 3
number of learnt clauses = 0
number of conflicts = 1
number of propagations = 11
cpu time = 19
=== Test X in [1..2], Y in [3..4], k = 3 ===
4 5
7 8
-1 2
-2 3
-3 4
-4 5
-5 6
-7 8
-8 9
-9 10
```

```
-10 11
-11 12
-7
-8
-9
-10
Written to encodePrecedence.cnf
 infer -7 (reason=None) [0]
 infer -8 (reason=None) [0]
 infer -9 (reason=None) [0]
 infer -10 (reason=None) [0]
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 1
 -7 -8 -9 -10
0
Unsatisfiable
number of choices = 0
number of learnt clauses = 0
number of conflicts = 0
number of propagations = 1
cpu time = 2
Expected : Unsatisfiable
```

Let X and Y be two integer variables such that $\mathbb{D}_x = [a, b]$ and $\mathbb{D}_y = [c, d]$ where $a < b \in \mathbb{N}$ and $c < d \in \mathbb{N}$. Let $z \in \mathbb{N}$ represents the positive literal (in DIMACS format) of a Boolean variable Z. Let $k \in \mathbb{N}$. Implement a function **encodeImpliesPrecedence(z, a, b, c, d, k)** that encodes the constraint $Z \rightarrow (X + k \leq Y)$

```
In [167]: def encodeImpliesPrecedence(z, a, b, c, d, k):
              assert z >= 0, 'Expected z = [0,1], got ' + str(z)
              assert a <= b, 'Expected a <= b, got ' + str(a) +' <= '+ str(b)
              assert c <= d, 'Expected c <= d, got ' + str(c) +' <= '+ str(d)
              assert k >= 0, 'Expected k >= 0, got ' + str(k)

              born_sup = max(b + k, d)

              X = []
              Y = []
              Z = -1 if z > 0 else 1
              clauses = []

              # Regroup variables
              for i in range(born_sup):
                  X.append(i + 2)
              for i in range(born_sup, 2*born_sup):
                  Y.append(i + 2)

              # No all 0 solution
              valid_x = ''
              for i in range(a, b + 1):
                  valid_x += str(X[i - 1]) + ' '
              clauses.append(valid_x)

              valid_y = ''
              for i in range(c, d + 1):
                  valid_y += str(Y[i - 1]) + ' '
              clauses.append(valid_y)

              # Clauses for ordering domain
              for i in range(born_sup-1):
                  clauses.append(
                      str(-X[i]) + ' ' + str(X[(i+1)])
                  )

              for i in range(born_sup-1):
                  clauses.append(
                      str(-Y[i]) + ' ' + str(Y[(i+1)])
                  )

              # Clauses for Z --> (X + k <= Y) <=> not(Z) or (X + k <= Y)
              # => add Z to every clause below

              # Clauses for X + k <= Y
              for i in range(c, a + k):
                  # if a + k > c then Y[c..a+k] = 0
                  clauses.append(
                      str(Z) + ' ' + str(-Y[i-1])
                  )

              for i in range(max(c, a + k), min(d, b+k)+1):
                  clauses.append(
                      str(Z) + ' ' + str(X[i-1]) + ' ' + str(-Y[(i-1)])
                  )

              if debug:
                  for c in clauses:
                      print(c)
                  #print 'Nb clause = ', len(clauses),' || Expected = ', nb_clauses

              nb_variable = 2 * born_sup + 1
              nb_clauses = len(clauses)
              write_cnf(nb_variable, nb_clauses, clauses, cnf_encodeImpliesPrecedence)

In [169]: print("=== Test X in [4..5], Y in [1..2], Z = 15, k = 1 ===")
          encodeImpliesPrecedence(15, 4, 5, 1, 2, 1)
          solver = Solver()
```

```
        solver.readDimacs(cnf_encodeImpliesPrecedence)
        run_solver(solver)
        print("Expected : Satisfiable with solution has -1 (Z activate)")

        print("\n=== Test X in [4..5], Y in [1..2], Z = 0, k = 1 ===")
        encodeImpliesPrecedence(0, 4, 5, 1, 2, 1)
        solver = Solver()
        solver.readDimacs(cnf_encodeImpliesPrecedence)
        run_solver(solver)
        print("Expected : Satisfiable")
```

```
=== Test X in [4..5], Y in [1..2], Z = 15, k = 1 ===
5 6
8 9
-2 3
-3 4
-4 5
-5 6
-6 7
-8 9
-9 10
-10 11
-11 12
-12 13
-1 -8
-1 -9
-1 -10
-1 -11
Written to encodeImpliesPrecedence.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 1

  infer -1 (reason=None) [1]
  -1
   infer -9 (reason=None) [2]
   infer -8 (reason=(-8 9)) [2]
   -1 -8 -9
 infer 9 (reason=(9)) [0]
 infer -1 (reason=(-1 -9)) [0]
 infer 10 (reason=(-9 10)) [0]
 infer 11 (reason=(-10 11)) [0]
 infer 12 (reason=(-11 12)) [0]
 infer 13 (reason=(-12 13)) [0]
 -1 9 10 11 12 13
  infer -8 (reason=None) [1]
  -1 -8 9 10 11 12 13
   infer -6 (reason=None) [2]
   infer -5 (reason=(-5 6)) [2]
   -1 -5 -6 -8 9 10 11 12 13
 infer 6 (reason=(6)) [0]
 infer 7 (reason=(-6 7)) [0]
 -1 6 7 9 10 11 12 13
  infer -5 (reason=None) [1]
  infer -4 (reason=(-4 5)) [1]
  infer -3 (reason=(-3 4)) [1]
  infer -2 (reason=(-2 3)) [1]
  -1 -2 -3 -4 -5 6 7 9 10 11 12 13
   infer -8 (reason=None) [2]
   -1 -2 -3 -4 -5 6 7 -8 9 10 11 12 13
1
Satisfiable
number of choices = 6
number of learnt clauses = 0
number of conflicts = 2
number of propagations = 17
cpu time = 11
Expected : Satisfiable with solution has -1 (Z activate)

=== Test X in [4..5], Y in [1..2], Z = 0, k = 1 ===
5 6
8 9
-2 3
-3 4
-4 5
-5 6
-6 7
-8 9
-9 10
-10 11
-11 12
-12 13
1 -8
1 -9
1 -10
1 -11
Written to encodeImpliesPrecedence.cnf
restart with limit = 100 -- number of conflicts = 0 -- cpu time = 2

  infer -1 (reason=None) [1]
  infer -11 (reason=(1 -11)) [1]
  infer -10 (reason=(1 -10)) [1]
  infer -9 (reason=(1 -9)) [1]
  infer -8 (reason=(1 -8)) [1]
  -1 -8 -9 -10 -11
 infer 1 (reason=(1)) [0]
 1
  infer -9 (reason=None) [1]
  infer -8 (reason=(-8 9)) [1]
  1 -8 -9
 infer 9 (reason=(9)) [0]
 infer 10 (reason=(-9 10)) [0]
 infer 11 (reason=(-10 11)) [0]
 infer 12 (reason=(-11 12)) [0]
 infer 13 (reason=(-12 13)) [0]
 1 9 10 11 12 13
  infer -8 (reason=None) [1]
  1 -8 9 10 11 12 13
   infer -6 (reason=None) [2]
   infer -5 (reason=(-5 6)) [2]
   1 -5 -6 -8 9 10 11 12 13
 infer 6 (reason=(6)) [0]
 infer 7 (reason=(-6 7)) [0]
 1 6 7 9 10 11 12 13
  infer -8 (reason=None) [1]
  1 6 7 -8 9 10 11 12 13
   infer -5 (reason=None) [2]
   infer -4 (reason=(-4 5)) [2]
   infer -3 (reason=(-3 4)) [2]
   infer -2 (reason=(-2 3)) [2]
   1 -2 -3 -4 -5 6 7 -8 9 10 11 12 13
1
Satisfiable
number of choices = 6
number of learnt clauses = 0
number of conflicts = 3
number of propagations = 20
cpu time = 14
Expected : Satisfiable
```