EE219C HW1: SAT and BDDs

Vighnesh Iyer

1 Horn-SAT and Renamable Horn-SAT

(a) Recall from class that a HornSAT formula is a CNF formula in which each clause contains at most one positive literal. Give an algorithm to decide the satisifiability of HornSAT formulas in linear time (in the number of variables n).

We can write a HornSAT clause as an implication:

$$\begin{aligned}
\text{In general: } & A \to B \iff \neg A \vee B \\
\text{HornSAT Clause: } & x_p \vee \neg x_{n,1} \vee \neg x_{n,2} \vee \cdots \vee \neg x_{n,l} \\
\text{Group terms: } & (\neg x_{n,1} \vee \neg x_{n,2} \vee \cdots \vee \neg x_{n,l}) \vee x_p \\
\text{Let A} = & (\neg x_{n,1} \vee \neg x_{n,2} \vee \cdots \vee \neg x_{n,l}) \\
\text{Let B} = & x_p \\
\neg A = & (x_{n,1} \wedge x_{n,2} \wedge \cdots \wedge x_{n,l}) \\
\text{Conclude: } & x_p \vee \neg x_{n,1} \vee \neg x_{n,2} \vee \cdots \vee \neg x_{n,l} \iff (x_{n,1} \wedge x_{n,2} \wedge \dots) \to x_p
\end{aligned}$$

We can also handle special-case HornSAT clauses by converting them to implications:

(a) Unit positive literal clause

$$x_p \iff (\mathbf{T} \to x_p)$$

i.e. for the CNF formula to be SAT, x_p must be set to $\mathbf{T}$.

(b) No positive literals in the clause

$$(\neg x_{n,1} \vee \cdots \vee \neg x_{n,l}) \iff ((x_{n,1} \wedge \cdots \wedge x_{n,l}) \to \mathbf{F})$$

Note that if no unit positive literal clauses are present, the formula is immediately satisfiable with the assignment of all variables to $\mathbf{F}$.

HornSAT can only be unsat if there is at least one unit positive literal clause. In this case, we can selectively flip variables to true based on the implications and find the formula is unsat if flipping a variable would contradict a clause with only negative literals. This basically amounts to positive literal unit propagation.

Algorithm 1 Naive HornSAT Solver

```
1: procedure HornSAT(φ)          ▷ φ is a formula in CNF form with m clauses c_i, i = 1, ..., m
2:     A ← [F, ..., F]
3:     while c ← !Satisfied(φ, A) do          ▷ Where c is some unsatisfied clause
4:         if c is of form (T → x_p) then
5:             A[x_p] ← T
6:         else if c is of form ((x_{n,1} ∧ ··· ∧ x_{n,l}) → x_p) then
7:             A[x_p] ← T
8:         else if c is of form ((x_{n,1} ∧ ··· ∧ x_{n,l}) → F) then
9:             return UNSAT
10:    return A
```

However this algorithm runs in polynomial time $\mathcal{O}(n^2)$. Unit propagation can be done in linear time by tracking which clauses can be influenced by a variable flip.

Algorithm 2 Linear HornSAT Solver

```
1: procedure HornSAT(φ)          ▷ φ is a formula in CNF form with m clauses c_i, i = 1, ..., m
2:     LMap ← Preprocess(φ)  ▷ Map from variable v to clauses where v is on the LHS of the
   implication
3:     Stack ← Preprocess(φ)          ▷ Stack of unit positive literal clauses
4:     A ← [F, ..., F]
5:     while c ← Pop(Stack) do          ▷ While the stack has clauses left
6:         if c is of form ((x_n) → F) then
7:             return UNSAT
8:         else if c is of form (_ → x_p && x_p = T) then
9:             continue
10:        else if c is of form (T → x_p) or of form ((x_{n,1} ∧ ··· ∧ x_{n,l}) → x_p) then
11:            A[x_p] ← T
12:            for c' ∈ LMap[x_p] do
13:                c' ← c''          ▷ Where c'' is c' with x_p deleted
14:                if c'' is of form (T → x'_p) then Push(Stack, x'_p)
15:    return A
```

Both of the preprocessing steps can be done while the CNF formula is being parsed. This version runs in $\mathcal{O}(n)$.

(b) Give a polynomial-time algorithm to check whether a formula on n variables comprising m CNF clauses is renamable Horn. Try to express this problem itself as a SAT problem.

Given a CNF formula ϕ with clauses $c_1, \ldots, c_k$, and where each clause c_i has literals $l_{i,1}, \ldots, l_{i,l}$ construct a new CNF formula ϕ_R as such:

$$\phi_R = \bigwedge_{i=1}^{k} \bigwedge_{1 \le j < k \le l} (l_{i,j} \vee l_{i,k})$$

If ϕ_R is satisfiable, then the original formula ϕ is renamable Horn. The solution to ϕ_R indicates what variables should be flipped to make ϕ Horn SAT.

We can show this property by examining a particular clause c_i and clause pair $l_{i,j}, l_{i,k}$ of ϕ, while assuming that ϕ is Horn renamable.

(a) $l_{i,j}$ and $l_{i,k}$ are both **positive literals** $(x_{p,1} \vee x_{p,2})$
To make ϕ_R SAT, at least one of the variables must be set to **T**. This will flip one of the literals in ϕ to make it Horn.

(b) $l_{i,j}$ and $l_{i,k}$ are mixed **positive and negative literals** $(x_p \vee x'_n)$
To make ϕ_R SAT, either $x_p = \mathbf{T}, x_n = \mathbf{F}$, or $x_p = \mathbf{T}, x_n = \mathbf{T}$, or $x_p = \mathbf{F}, x_n = \mathbf{F}$.
In all cases they represent renamings that would make the original clause Horn, or preserve its Horn-ness.

(c) $l_{i,j}$ and $l_{i,k}$ are both **negative literals** $(x'_{n,1} \vee x'_{n,2})$
To make ϕ_R SAT, at most one of the variables can be set to **T**. This will preserve the Horn-ness of the original clause.

So if ϕ_R is SAT, then the positive assignments to the variables indicate that the literals containing that variable in ϕ should be complemented to make the formula HornSAT. Since it takes time $\mathcal{O}(m)$ to construct ϕ_R and it takes polynomial time to solve 2-SAT, this algorithm runs in polynomial time.

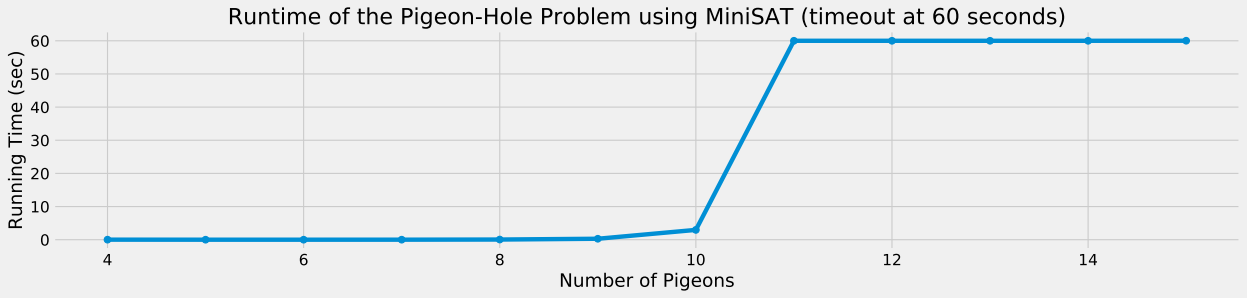
2 The Pigeon-Hole Problem

(a) Encode the pigeon-hole problem as DIMACS CNF for $n = 4, 5, 6, \ldots, 15$. Run it with MiniSAT and plot how the runtimes vary with n. Describe your observations.

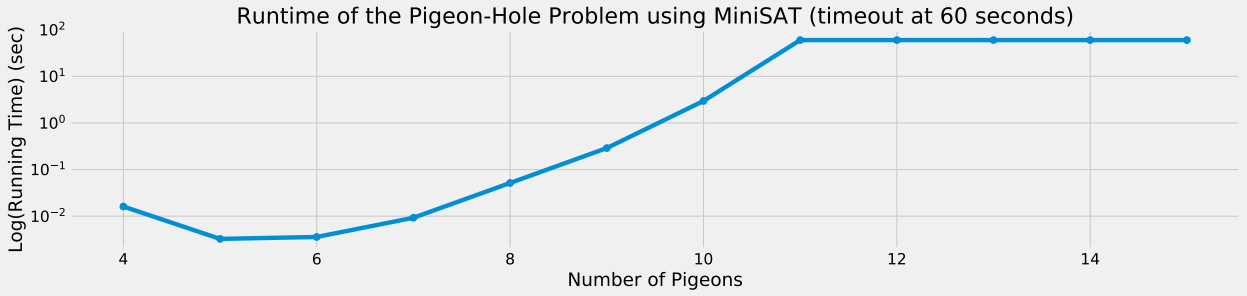
I wrote a package in Scala to handle CNF formulas. Here's the relevant exerpt, which returns the pigeon-hole problem as CNF for n pigeons:

```
def problem(n: Int): CNFFormula = {
val pigeons = 1 to n
val holes = 1 until n
// stride on pigeons with stride size of holes
def pigeonHoleToVariable(p: Int, h: Int): Int = {
  (p-1)*holes.size + h
}
val everyPigeonInOneHole: CNFFormula = pigeons.map {
  p => holes.foldLeft(Set.empty[Int]) {
    case (s, h) => s.union(Set(pigeonHoleToVariable(p, h)))
  }
}.toSet
val noTwoPigeonsPerHole = holes.foldLeft(Set.empty[Set[Int]]) {
  case (ss, h) =>
  ↪  ss.union(pigeons.combinations(2).foldLeft(Seq.empty[Set[Int]]) {
    case (s, p) => s :+ Set(-pigeonHoleToVariable(p(0), h),
    ↪  -pigeonHoleToVariable(p(1), h))
  }.toSet)
}
everyPigeonInOneHole.union(noTwoPigeonsPerHole)
}
```

I wrote a script to run MiniSAT on the produced CNF file and plotted the runtimes (with a 60 second timeout):



There is a noticable increase in runtime for 10 pigeons, and 11+ pigeons require more than 1 minute of runtime. The runtime growth is exponential. This is clear if plotted on a log-scale:



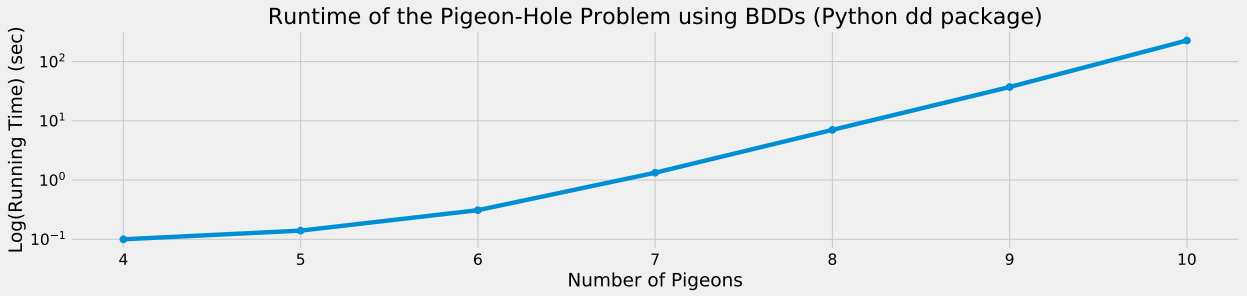
(b) Construct the BDD for the pigeon-hole problem and verify it simplifies to false. What happens when you increase n? Does the variable order matter? Use the Python `dd` package.

Implementation of the pigeon-hole problem BDD using the `dd` package:

```
def pigeonhole(pdfname, n):
    print ('  [Pigeonhole Problem for n=%d]' % n)

    bdd = _bdd.BDD()
    for p in range(n):
        for h in range(n-1):
            bdd.declare('x_%d_%d' % (p, h))

    # All pigeons are in at least 1 hole
    all_in_a_hole = map(lambda p: reduce(lambda x, y: x | y,
    ↪  [bdd.var('x_%d_%d' % (p, h)) for h in range(n-1)]), range(n))
    all_in_a_hole = reduce(lambda x, y: x & y, all_in_a_hole)

    # No 2 pigeons are in the same hole
    no_two_in_same_hole = [~(bdd.var('x_%d_%d' % (c[0], h))) |
    ↪  ~(bdd.var('x_%d_%d' % (c[1], h))) for c in
    ↪  itertools.combinations(range(n), 2) for h in range(n-1)]
    no_two_in_same_hole = reduce(lambda x, y: x & y, no_two_in_same_hole)
```

```
f = all_in_a_hole & no_two_in_same_hole
if (f == bdd.true):
    print('SAT')
else:
    print('UNSAT')
```

This function always prints `UNSAT`.



The runtime of the pigeon-hole problem is exponential in n.

When I enabled dynamic reordering with `bdd.configure(reordering=True)`, the runtime actually got worse by about 10x. This is probably attributed to a lot of time being spent on reordering which is useless in this pathological problem. The variable order doesn't matter since every variable has to be propagated to figure out if the problem is unsat.