

# SAT Solver: NP? Rather, Lots of Problems

Pedro Rodriguez - 15-11264 and Daniel Pinto - 15-11139

## 1 Specs del equipo usado

Todas las pruebas se corrieron en el mismo equipo, el cual cuenta con los siguientes specs:

- **Procesador:** Ryzen 7 3800x, 3.9GHz.
- **RAM:** 16GB, 3200MHz.
- **Java Heap Size:** 4GB
- SSD.

## 2 Modelado de variables Inicial

Por motivos de simplicidad, el dominio de las variables utilizadas sera los números enteros, incluso cuando nos referimos a datos no numéricos. Por ejemplo, aunque el nombre de los equipos son de tipo string, como este numero es finito, existe una inyección de la cantidad efectiva de equipos a los números enteros. Esta decisión es para facilitar las cuentas haciendo el mapeo implícito.

Decidimos que cada variable representa una configuración para un encuentro para un equipo. Es decir, que incorporara: el equipo, el día en el que juega, el encuentro en el que juega (primer partido del día, segundo,...) y si juega como local o visitante. En otras palabras, el tipo es modelado como:

```
case class P
  ( team      : N
  , day       : M
  , match_no  : R
  , isLocal   : 2
  )
```

Que una variable  $P_{t,d,m,0}$  sea verdadera, significa que el equipo  $t$  jugo en el día  $t$  en el encuentro  $m$  como local (0). De esto es directamente deducible que:

$$|TeamVar| = N \cdot M \cdot R \cdot 2$$

## 3 Restricción 1: Todos contra todos

La primera restricción nos dice que: *Todos los participantes deben jugar dos veces con cada uno de los otros participantes, una como "visitantes" y la otra como "locales". Esto significa que, si hay 10 equipos, cada equipo jugará 18 veces.*

Bajo nuestro modelo de variables, esta restricción se puede plantear en lógica proposicional como:

---

Pedro Rodriguez - 15-11264: [15-11264@usb.ve](mailto:15-11264@usb.ve)

Daniel Pinto - 15-11139: [15-11139@usb.ve](mailto:15-11139@usb.ve)

$$\begin{aligned}
& \bigwedge teamA, teamB \\
& \quad | teamB > teamA \\
& : \left( \bigvee dayA, matchA, dayB, matchB \right. \\
& \quad , matchAL = P_{teamA, dayA, matchA, 0}, matchAV = P_{teamA, dayA, matchA, 1} \\
& \quad , matchBL = P_{teamB, dayB, matchB, 0}, matchBV = P_{teamB, dayB, matchB, 1} \\
& \quad | dayB > dayA \wedge teamB > teamA \\
& \quad \left. : (matchAL \wedge matchBL) \vee (matchAV \wedge matchBV) \right)
\end{aligned}$$

Lo cual es intimidante, veamos solo el cuantificador mas interno:

$$\begin{aligned}
& \bigvee dayA, matchA, dayB, matchB \\
& \quad , matchAL = P_{teamA, dayA, matchA, 0}, matchAV = P_{teamA, dayA, matchA, 1} \\
& \quad , matchBL = P_{teamB, dayB, matchB, 0}, matchBV = P_{teamB, dayB, matchB, 1} \\
& \quad | dayB > dayA \\
& \quad : (matchAL \wedge matchBL) \vee (matchAV \wedge matchBV)
\end{aligned}$$

Esta condición nos dice que para un par de equipos **distintos y fijos**  $teamA, teamB$ , existe (recordando que la  $\vee$ -oria tiene la misma semántica que existe en conjuntos finitos) un  $dayA, matchA$  (fecha inicial) y un  $dayB, matchB$  (una fecha posterior) tal que: El equipo  $teamA$  se enfrenta al equipo  $teamB$  en ambas fechas, una como local y una como visitante.

Notemos que esta condición es justamente lo que necesitamos, si los términos fijos fuesen variables, es decir, que si abstraemos con un para todo obtenemos la restricción deseada. Lo cual es exactamente lo que hace la  $\wedge$ -oria (que se comporta como un para todo para conjuntos finitos).

Algo de notar es la forma que posee el termino después de realizar la distributividad  $(matchAL \wedge matchBL) \vee (matchAV \wedge matchBV)$  (paso necesario para transformar en CNF), la formula se reescribe como algo de la forma:

$$X_1 \wedge X_2 \wedge X_3 \wedge X_4 := (p \vee r) \wedge (q \vee r) \wedge (p \vee s) \wedge (q \vee s)$$

Y al estar dentro de una  $\vee$ -oria:

$$\bigvee X_1 \wedge X_2 \wedge X_3 \wedge X_4$$

Se **generaría una cantidad exponencial de clausulas con respecto al numero de clausuras iniciales**. Lo cual nos lleva a la pregunta: cuantas clausulas tenemos?

Pues como la única restricción que tenemos dentro del cuantificador  $\vee$  es que  $dayB > dayA$ , vamos a tener un total de clausulas base igual a:

$$\frac{M \cdot (M + 1)}{2} \cdot R^2$$

Y por lo tanto, un total de clausulas en CNF equivalente a:

$$\frac{N \cdot (N + 1)}{2} \cdot 2^{\frac{M \cdot (M + 1)}{2} \cdot R^2}$$

Es decir, exponencial en el numero de partidos y días. Como referencia, para un tournament de tan solo 4 equipos, 3 días y 2 partidos por día generaría un total de:  $\sim 10^8$  clausulas.

## 4 Restricción 2: Contrario a la creencia popular, si hay $m$ palomas y $n$ agujeros...

*Dos juegos no pueden ocurrir al mismo tiempo.*

Teniendo en cuenta que pueden haber  $R$  partidos por día y  $N$  equipos, una forma de abordar esto es mediante filtro: Generar todos los posibles  $R$  subconjuntos de los  $N^2 - N$  (descontando encuentros identidades) posibles encuentros y filtrar aquellos que ocurran al mismo tiempo.

Por lo tanto definiremos:

$$Encounters = \{(teamA, teamB) \in N \times N \mid teamA \neq teamB\}$$

Como el conjunto de posibles encuentros.

$$UDE = \binom{Encounters}{R}$$

Como el conjunto de todos los encuentros posibles en un día sin restricciones (Unrestricted Day Encounters). En donde realizamos el siguiente abuso de notación: La función  $\binom{n}{k} : A \times \mathbb{N} \rightarrow \mathcal{P}(A)$  toma un conjunto  $A$ , un número  $N$ , y devuelve todos los subconjuntos de tamaño  $N$  de  $A$ .

Luego definiremos la función  $i_{day} : \mathcal{P}(N \times N) \rightarrow \mathcal{P}(\mathbb{B}(P))$  la cual inyecta los elementos al conjunto de expresiones booleanas que poseen como variables el conjunto  $P$  previamente definido. Es decir, esta función se comporta como un parser que traduce encuentros en forma de tuplas, a encuentros en forma de lógica proposicional.

$$\begin{aligned} i_{day} XS = \{ & BS \mid X \in XS \wedge (teamA, teamB) \in X \wedge matchNo \in R \\ & \wedge P_{teamA, day, matchNo, 0}, P \wedge P_{teamA, day, matchNo, 1} \in P \\ & \wedge (P_{x, day, matchNo, z}, P'_{x', day, matchNo, z} \in P \implies x = x') \\ & \wedge |P| = R \wedge \wedge P \in PS \\ & \wedge \bigwedge P = B \wedge B \in BS \\ & \} \end{aligned}$$

Así, podemos definir:

$$X_f := \bigwedge_{ude \in UDE, dayA} f(ude) : \left( \bigvee i_{day}(ude) \right)$$

Como la familia de conjuntos indexados por la función  $f$  que retorna la expresión booleana que representa todas las posibles configuraciones de juegos. Si dejamos que  $f \_ = True$  la función constante que retorna verdadero entonces  $X_f$  retorna el conjunto que satisface la restricción 2.

Al observar la forma que posee  $X_f$ :

$$X_f = \bigwedge \left( \bigvee \left( \bigwedge P \right) \right)$$

Vemos que esta no está en CNF, de hecho, tenemos que distribuir los  $\vee/\wedge$  mas internos, dándonos otro comportamiento exponencial cuyo exponente depende de la cantidad de posibles configuraciones para un día:

$$2^{\binom{N^2 - N}{R}}$$

Sin embargo, a diferencia de las demás formulas, este número es demasiado pesimista. Esto es debido a que el esquema lógico propuesto en este informe impone un algoritmo declarativo sumamente ineficiente, y que además la función  $f$  combinara varias restricciones para filtrar elementos de este conjunto.

## 5 Restricción 3: Aunque se coma callado NO se come dos veces

*Un participante puede jugar a lo sumo una vez por día.*

Esta restricción se logra instanciando el  $X_f$  con un  $f$  tal que rechace todos aquellos *udes* que un equipo participe mas de una vez en cualquiera de las componentes.

## 6 Restricción 4: Two wrongs DON'T make a right

*Un participante no puede jugar de "visitante" en dos días consecutivos, ni de "local" dos días seguidos.*

Esta restricción se traduce a una simple implicación:

$$\bigwedge_{day, team, matchupToday, matchupTomorrow} | : P_{team, day, matchupToday, x} \implies \sim P_{team, day+1, matchupTomorrow, x}$$

Lo cual, después de desugar se convierte en:

$$\bigwedge_{day, team, matchupToday, matchupTomorrow} | : \sim P_{team, day, matchupToday, x} \vee \sim P_{team, day+1, matchupTomorrow, x}$$

Que ya esta en CNF, y por lo tanto, no existe la necesidad de pasarle la transformación.

## 7 Restricción 5,6,7...: O'clock? OH NOOOOOOOO<sub>(problem)</sub>

*Todos los juegos deben empezar en horas "en punto" (por ejemplo, las 13:00:00 es una hora válida pero las 13:30:00 no).*

Esta restricción viene de gratis con la representación del tipo  $P$ . No hay necesidad de codificarla. Al igual que esta restricción, las demás son dadas por el tipo.

## 8 La realidad

Debido al comportamiento exponencial de las restricciones 1 y 2, el programa realmente no puede generar las clausulas en CNF, esto no es necesariamente debido al tamaño final en disco, sino al tamaño en memoria. Debido a que estamos trabajando con scala, este no solo impone limitaciones de memoria en la VM (la cual se le asigno un heap size de 4GB), sino que también la representación en memoria de los tipos es muy superior a la de un lenguaje de mas bajo nivel.

Tomando como ejemplo la restricción 1 con 4 equipos, 3 días y dos partidos. Como vimos, esto genera  $10^8$  clausulas, si consideramos que cada entero en scala toma 4 bytes (y hay 4 enteros en cada representación), obtenemos un total de 1GB en RAM para la representación final de esa única restricción. Lo cual es una cota inferior demasiado laxa tomando en cuenta los contextos recursivos, punteros, y demás estructuras adicionales que se utilizan para construir el termino.

Por lo tanto, se decidió debilitar el problema con el propósito de mostrar que la maquinaria adicional (los parsers, y el runner de glucose) están hechos y corriendo. Esto no significa que las restricciones no hayan sido codificadas. Estas se pueden ver en el archivo Encodings. En particular:

Para la restricción 2:

```

// Representa La restriccion 2.
def generate_matches_for_day(entryFormat: EntryFormat) : Algebra[TeamEncoding] = {
  /* We encode participants not by their names, but rather
  by their position in the participants vector */
  val matchups = for {
    a <- 0 until entryFormat.participants.length
    b <- 0 until entryFormat.participants.length if a != b
  } yield(a,b)
  val days
    = 0
    until
    entryFormat.start_date.until(entryFormat.end_date,ChronoUnit.DAYS).toInt + 1
  val total_matches
    = get_number_of_matches_per_day(entryFormat.start_time,entryFormat.end_time)
  val daily_matchups = matchups.combinations(total_matches) // UDE
  // Builds the (teamLocal,teamVisitor,match) for the P_team_day_match_local
  .map(_.zipWithIndex.map(it => (it._1._1,it._1._2,it._2) ))
  // the f function
  .filter(is_naively_legal_matchup)
  .toSeq
  // We had an iterable, we transfor it to a boolean expression of the form
  // v (^ xs) as instructed
  val daily_matchups_ast = embed_possible_matchups(daily_matchups)
  // We build the actuals P_team_day_match_local by mapping what we had
  val rs = days.map((d : Int) => daily_matchups_ast.map(make_team_encoding(d)))
  // We had v(^ xs), after this we have ~(v(^xs))
  rs.drop(1).foldLeft(rs.head)((acc,b) => acc & b)
}

```

Para la restricción 1:

```

def generate_match_everybody_restriction(entryFormat: EntryFormat) : Algebra[TeamVar] = {
  val day_upper_limit
    = entryFormat.start_date.until(entryFormat.end_date, ChronoUnit.DAYS).toInt
  val days = 0 until day_upper_limit + 1
  val total_matches
    = get_number_of_matches_per_day(entryFormat.start_time,entryFormat.end_time)

  // Direct translation of Restriction 1
  var clauses_seq : Seq[Algebra[TeamVar]] = Seq.empty
  for (a<- 0 until entryFormat.participants.length;
    b <- a+1 until entryFormat.participants.length)
  {
    var acc: Seq[Algebra[TeamVar]] = Seq.empty
    for (dayA <- days; matchA <- 0 until total_matches) {
      val matchAL
        = VariableTerm(TeamVar(a,dayA,matchA,true))
        & VariableTerm(TeamVar(b,dayA,matchA,false))
      val matchAV
        = VariableTerm(TeamVar(a,dayA,matchA,false))
        & VariableTerm(TeamVar(b,dayA,matchA,true))
      for (dayB <- dayA + 1 until day_upper_limit; matchB <- 0 until total_matches) {
        val matchBL
          = VariableTerm(TeamVar(a,dayB,matchB,false))
          & VariableTerm(TeamVar(b,dayB,matchB,true))

```

```

val matchBV
  = VariableTerm(TeamVar(a,dayB,matchB,true))
  & VariableTerm(TeamVar(b,dayB,matchB,false))
// this generates something like:
// (P_a_dayA_matchA_local ^ P_b_dayB_matchB_visitor)
// | (P_a_dayA_matchA_vistor ^ P_b_dayB_matchB_local)
// that is, this for generates all possible ways
// that two contestants can battle each other: one as local
// and one as visitor for each possible day/matchup configuration.
acc = ((matchAL & matchBL) | (matchAV & matchBV) ) += acc
}
}
// we need to combine the configurations using `Or`, since it suffices that
// one is true for two contestants
// to fight each other as local and then visitor.
clauses_seq = acc.drop(1).foldLeft(acc.head)((acc,clause) => acc | clause) += clauses_seq
}

// we link with `And` since we need
clauses_seq.drop(1).foldLeft(clauses_seq.head)((acc,t) => acc & t)
}

```

## 9 Parafernalia y Miscelánea

Esta sección tiene como objetivo describir la estructura del proyecto. En particular como correrlo.

El lenguaje seleccionado fue Scala 3.0 con el manejador de paquetes SBT. Por lo tanto, es altamente recomendable utilizar un IDE como intelliJ para compilar y correr el proyecto.

El proyecto utiliza el archivo de configuración `options.yaml` para configurar todas las variables que necesita el proyecto, en particular, posee el siguiente esquema:

```

entryPoint      : path/a/donde/esta/el/json/a/leer
glucoseExecutable : /path/al/ejecutable/de/glucose
glucoseInput     : /path/a/donde/se/guarda/el/input/de/glucose
glucoseOutput    : /path/a/donde/se/guarda/el/output/de/glucose
output          : /path/donde/se/guardan/los/resultados

```

## 10 Conclusiones

Todo problema solucionado con SAT presenta dos cuellos de botella: el primero es la generación de la formula en CNF, mientras que el segundo es el de demostrar satisfacibilidad. Aunque el 2do problema ya esta mayormente resuelto, mediante SAT Solvers eficientes como lo es glucose. El primero aun queda de parte del desarrollador.

En el reporte actual, se decidió afrontar un problema de scheduling mediante técnicas intuitivas: modelado directo, y el uso del algoritmo clásico de conversión (empuje de negaciones y luego empuje de distributividades). Lo cual genera representaciones en memoria tan grandes, que solo tendrían sentido si se almacenan en disco.

Una recomendación para evitar este tipo de problemas es tanto mudar el algoritmo clásico de conversión a uno iterativo, evitando así parcialmente el problema de llenado del heap size. Adicionalmente, una vez que se modele de manera intuitiva el problema, es recomendado realizar alguna transformación, como lo es la **transformación de Tseytin** para mantener lineal el tamaño de

la transformación.

Finalmente, aunque scala provee grandes facilidades para el desarrollo de este tipo de problemas (ADT, derivación automática de esquemas recursivos, funtores, sobrecarga de operadores, etc). Este impone restricciones demasiado fuertes sobre el uso de memoria (lo cual puede ser en parte culpa de tener que correr en la JVM). Por lo tanto, para futuros proyectos es recomendable al menos alejarse de esta infraestructura.