T-Tree: Un simulador de tratamientos médicos

Ana P. Argüelles Terrón, Alejandro Beltrán Varela, Javier Lopetegui González y Abel Molina Sánchez

Universidad de La Habana, Cuba

Abstract. This paper will describe the process of creating **Treatment Tree(T-Tree)**, a medical treatment simulator. The structure of the project will be discussed, ranging from the creation of a domain-specific language and the entire compiler structure that would translate that language into Python code. The simulation mechanism will be presented with the strategies used for it, as well as the results obtained.

Resumen. En el presente trabajo se describirá el proceso de realización de **Treatment Tree(T-Tree)** un simulador de tratamientos médicos. Se debatirá la estructura del mismo, que va desde la creación de un lenguaje de dominio específico y toda la estructura del compilador que tanspilará dicho lenguaje a código Python. Se presentará el mecanismo de simulación con las estrategias utilizadas para el mismo, así como los resultados obtenidos.

Keywords: simulación de tratamientos · compilador · UCT · agentes · árbol de tratamientos

1 Introducción

El sistema surge a partir de la orientación de un proyecto integrador de las asignaturas de Simulación, Inteligencia Artificial y Compilación impartidas en 3er año de la licenciatura en Ciencias de La Computación de la facultad de Matemática y Computación(MATCOM) de la Universidad de La Habana(UH). Para el desarrollo del mismo se eligió por parte del equipo de trabajo un tema sobre el cual realizar la simulación. La simulación de tratamientos médicos se escogió en base al interés de tener un acercamiento desde la computación a esta área de la ciencia. Se plantearon diferentes estrategias para el enfoque de la simulación, las cuales se plantearán más adelante. En conjunción con el módulo de simulación se desarrolló un compilador con generador de parser LR(1) para la transpilación del DSL a código Python.

2 Simulación de Tratamientos

Durante el debate de las estrategias a seguir en el proyecto surgieron distintos enfoques de hacia donde debería ir enfocado el resultado del mismo. Por un lado existió la idea de que el resultado de la simulación fuera la elección del mejor tratamiento en base a un conjunto de tratamientos(secuencia de intervenciones médicas en el tiempo) predeterminadas por el usuario, simulando la evolución del paciente con la elección de cada una de ellas y eligiendo aquella con mejor resultado. El otro enfoque planteaba que el resultado debía ser un árbol de decisiones de posibles tratamientos a aplicar de acuerdo a los distintos desarrollos de la enfermedad. Este último fue finalmente el elegido para desarrollar el modelo.

La simulación está basada en conjuntos de agentes reactivos: la enfermedad y el tratamiento, que interactúan en el ambiente(cuerpo), compuesto de diferentes parámetros cuyos valores definen sus distintos estados.

La enfermedad se define como un conjunto de *síntomas*, cada uno de los cuales es un agente.

El tratamiento se define con un conjunto de *intervenciones* posibles a aplicar, podemos verlo como el stock de posibles medicamentos disponibles para tatar a un paciente. Cada una de las intervenciones es un agente.

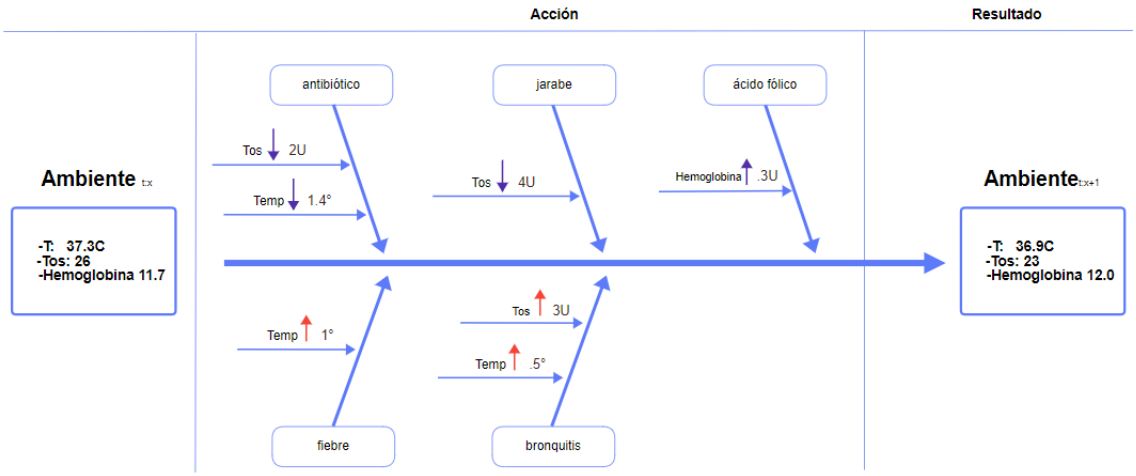
Los *agentes* pueden estar activos o inactivos, para lo cual, cada uno tiene sus condiciones de activación, que dependen del estado del ambiente y/o del tiempo de la simulación. Debido a las características del dominio, cada agente tiene un tiempo de efecto(tiempo durante el cual estará activo) y un tiempo de repetición de su acción. Por tanto, un agente estará activo durante su tiempo de efecto si se cumplen sus condiciones de activación y tendrá acción sobre el ambiente cada tiempo de repetición. De esta forma buscamos representar el comportamiento que pueden tener las enfermedades y los medicamentos, ya que estos tienden a tener efectos graduales en el tiempo más que resultados bruscos en un momento dado. Los agentes de las intervenciones también poseen una cantidad finita de activaciones, ya que sus aplicaciones también estarán condicionadas por la cantidad disponible de la misma.

Véase un ejemplo de definición de estos agentes:

Tenemos un *síntoma* que es la fiebre, que se activa si las plaquetas están bajas. La fiebre estará inicialmente activa durante 72h, teniendo efecto cada 4h. Luego cada 4h, va a aumentar(disminuir) algún(algunos) parámetro del ambiente.

A su vez, tenemos como posible intervención el jarabe, que se puede activar si la tos tiene un valor considerado alto, y tiene un tiempo de acción de 48h, tomándose una dosis cada 8h. Además se cuenta con solo 5 unidades del mismo. Igualmente en cada intervalo actuará modificando parámetros del ambiente.

Se muestra un ejemplo de un posible instante de una simulación:



Simulación en un instante de tiempo

Para la creación de las intervenciones se tiene una clase `Intervention` y para los síntomas una llamada `Symptom`; ambas heredan de la clase `Agent`.

El ambiente(`Environment`) se define como una lista de parámetros(`Parameter`), cada uno con un valor inicial y 2 conjuntos de límites, el primero define el rango en el que se puede considerar que dicho parámetro está estable, el segundo otro rango que contiene al primero y que establece los límites extremos de este. Encontrarse contenido en el segundo pero no en el primero sirve como indicador de afección y estar fuera de este inidicaría estado de muerte del paciente.

2.1 Algoritmo de Simulación

Para la ejecución de la simulación se parte de los siguientes parámetros:

- environment : el ambiente(parámetros sobre los que se va a realizar la simulación)
- tick : representa la cantidad de horas de cada paso de tiempo en la simulación.
- end_time : el tiempo total que se quiere que duré la simulación
- simulations: la cantidad de veces que se va a realizar la simulación
- un conjunto inicial de *síntomas*
- un conjunto inicial de *intervenciones*

Cada *simulación* es una instancia de la clase `Simulation` que se inicializa con los parámetros definidos anteriormente. Cada tick de tiempo se simula que ocurre en el ambiente.

Cada agente activo aumenta su tiempo de actividad y si se encuentra en tiempo de repetición, actúa sobre el ambiente en correspondecia. Si completó su tiempo de estar activo, pasa a formar parte de los agentes inactivos, en otro caso continúa interactuando con el ambiente. En el caso de las intervenciones también se comprueba si quedan suministros de la misma luego de completado el efecto, en caso de no quedar, se elimina. Una vez actualizado el ambiente se pasa a comprobar si se cumple alguna regla de activación de alguno de los síntomas inactivos. Pasando a activarse los que las cumplan.

En este momento se comprueba el conjunto de intervenciones inactivas, y se determinan aquellas candidatas a aplicar. De las posibles intervenciones candidatas se selecciona una y se agrega como agente activo. En el proceso de selección de las intervenciones aplicables en cada paso posible(habrá momentos donde ninguna intervención sea aplicable) se va construyendo y actualizando el *árbol de tratamientos*.

La construcción del árbol de tratamientos(`treatment_tree`) es el objetivo perseguido en la simulación y en su proceso está presente la inteligencia artificial.

Este árbol es común a todas las simulaciones, y cada uno de sus nodos(excepto la raíz) representa la aplicación de una intervención en un instante de tiempo. Cada rama del árbol terminará representando un tratamiento ante la enfermedad. Para su construcción se sigue una estrategia basada en *UCT*(upper confidence bounds applied to trees), política utilizada para la selección en el algoritmo de *Monte Carlo Tree Search*(Russell and Norvig, 2020).

Cada nodo del árbol va a tener asociado un valor de utilidad y un contador de visitas. Una vez que se alcanza un nodo final(último tratamiento aplicado cuando termina la simulación) se calcula el valor de utilidad basado en los parámetros del ambiente, y este se propaga por todos sus ancestros(back propagation).

El valor de utilidad lo calculamos en base a la siguiente métrica:

sea i un parámetro del ambiente:

$p =$ valor del parámetro
$ub =$ límite extremo superior del parámetro
$lb =$ límite extremo inferior del parámetro
$ug =$ límite bueno superior del parámetro
$lg =$ límite bueno inferior del parámetro
$c = (ug - lg)/2$

$n(x) = \frac{x-lb}{ub-lb}$ función para estandarizar los valores absolutos del rango entre $[0, 1]$

$d = |n(p) - n(c)|$ si $p \in [lb, ub]$ y $p \notin [ug, lg]$

$d = 0$ si $p \in [ug, lg]$

$$uv = \sum_i \frac{1}{1+d_i}$$

esta fórmula se aplica cuando todo los valores de los parámetros se encuentran dentro del rango posible, y da mayor valor a la utilidad mientras más cerca del centro de la región estable se encuentre. Si se encuentra dentro de la región estable se suma el valor máximo posible(1).

Cuando al menos un valor de los parámetros se encuentre fuera de los límites aceptables, entonces se llegó a un estado final negativo y para calcular la utilidad, se iguala los valores(p) del resto de parámetros al límite extremos más distante del centro. De esta forma se garantiza que siempre las soluciones que no lleven al estado final al paciente tengan mejor ponderación que aquellas que por al menos un parámetro causaron un estado final.

A la hora de seleccionar qué intervención aplicar en un instante de tiempo, se compruba si ese nodo ya forma parte del árbol, en caso negativo, se agrega al mismo. Luego, para todos los nodos candidatos se calcula su valor de *UCT*(upper confidence bounds applied to trees) y se selecciona el de mayor *UCT*. La política *UCT* permite hacer compensación entre exploración y explotación de soluciones, dándole relevancia no solo al valor de utilidad, sino también a la cantidad de visitas de cada nodo. Para un nodo n, se calcula el límite de confianza superior *UCB* según esta política:

$$UCB = \frac{us(n)}{vc(n)} + \sqrt{\frac{\log vc(p(n))}{vc(n)}}$$

donde:
$vc(n)$ = cantidad de visitas de n
$us(n)$= valor de utilidad de n
$p(n)$ = padre de n.

El primer término es el de explotación, dando el valor promedio de utilidad, mientras que el segundo es el de exploración, donde el sumando va a ser mayor mientras menor sea la cantidad de visitas del nodo.

Si el nodo no ha sido explorado, o sea, su valor $vs(n) = 0$, se hace
$UCB(n) = \infty$, garantizando la exploración.

Cuando hay más de un nodo con el valor máximo, se selecciona aleatoriamente uno de ellos para continuar.

Una vez completadas todas las simulaciones, se tienen valores de probabilidad para cada arista, definidos como el total de visitas del nodo hijo

entre el total de visitas del nodo padre. Cada rama del árbol va a describir un tratamiento frente a la enfermedad, siendo el tratamiento de mayor probabilidad de resultar efectivo frente al desarrollo promedio de la enfermedad aquel que se construye eligiendo la arista de mayor probabilidad en cada paso.

2.2 Módulos de solución

La definició de los parámetros que constituyen el ambiente y a su vez la forma en que son modificados tienen dos vías distintas.

2.2.1 Representación numérica

Un parámetro del ambiente está constituido por:

- nombre
- valor
- límites extremos superior e inferior : un valor del parámetro por encima o por debajo respectivamente de los mismos, implicaría la pérdida del paciente.
- límites estables superior e inferior : un valor del parámetro entre dichos límites repesenta que el mismo está en un valor deseado y bueno para el paciente.

Las acciones de los agentes(síntomas e intervenciones) sobre el ambiente ocurrirán de forma numérica, y como tal serán definidas por el usuario.

Ejemplo de definición de la acción de una intervención:

Se tiene la *dipirona*, una intervención que actuará sobre el parámetro *temperatura(T)* del ambiente. Se define que en cada repetición de su efecto: con una probabilidad del 60%, disminuirá la T en 0.4°C , mientras que con una probabilidad del 20% lo hará en 0.8°C. Quedando un 20% restante donde puede quedarse sin efecto.

2.2.2 Motor Fuzzy

En esta vía de solución se utiliza la lógica difusa y sus reglas para lograr definiciones y reprensentar el conocimiento experto.

Los parámetros del ambiente, además de la composición vista anteriormente, incluyen funciones de membresía para sus valores lingüísticos. Por ejemplo, es posible expresar que la temperatura se puede considerar alta, media o baja, y para cada una dar una función de memebresía. En la implementación se cuenta con dos de las funciones de membresía más utilizadas como son la triangular y la trapezoidal.

Para las acciones de los agentes, estos, en vez de modificaciones numéricas sobre los parámetros, devolverán reglas que serán evaluadas. Las reglas en la lógica utilizada pueden tener una de estas dos variables objetivos: *increase(aumentar)* o *decrease(disminuir)*. Supongamos que para cada una de ellas se difinen tres posibles valores lingúísticos: poco(little), medio(medium), mucho(high); la estructura de una posible regla sería la siguiente:

antecedente: if parámetro X_0 == α and parámetro X_1 == β
consecuente: increse = high
objetivo: parámetro X_2

que se traduce en: si el X_0 es igual a α y X_1 es igual a β, entonces, aumenta mucho el parámetro X_2.

Se utiliza la inferencia de Mandami por mínimos para evaluar los antecedentes con conjunción(*and*). La defuzzificación se realiza utilizando el método del *centroide*. El valor del parámetro objetivo se incrementa o decrementa respectivamente en el valor obtenido de dicho proceso.

3 Compilador

3.1 Lexer

Una vez que el usuario ejecuta el código el texto correspondiente a él debe serr separado en unidades llamadas tokens. Para ello se definió un lenguaje de expresiones regulares que cuenta con tres operaciones: unión , concatenación y clausura. Para este lenguaje se definieron cuatro tipos de tokens: *word*, *or("|")*, *concat(U)* y *star(*)*.

La gramática de este lenguaje cuenta con 7 no terminales(E, X, T, Y, F, Z, A) y 7 terminales(_i, _or, _concat, _star, left_br, right_br, epsilon). Las producciones definidas son:

- $E \to TX$
- $X \to | \; TX$
- $X \to \epsilon$
- $T \to FY$
- $Y \to U \; FY$
- $Y \to \epsilon$
- $F \to AZ$
- $Z \to *Z$
- $Z \to \epsilon$
- $A \to (E)$
- $A \to i$

Se implementó un generador de paser *LL* teiendo en cuenta que este es un lenguaje sencillo y que está sujeto a pocas midicaciones futuras. Este generador de parser construye la tabla *LL* que es utilizada para obtener el *AST* dada la cadena conformada por todas las expresiones de los tokens del *DSL* separadas por “or”. A partir de este *AST* se construye el autómata que se empleará en reconocer las cadenas correspondientes a los tokens del *DSL*.

El método `tokenize_regex_automaton(c, t)` devuelve los tokens de *t* que se forman con la cadena *c*.

3.2 Parser

Para parsear las cadenas de tokens obtenidas se emplea un parser LR(1) porque se trata de un parser robusto y eficiente que, primeramente, permite parsear una cadena en tiempo lineal respecto al tamaño de la misma y resuelve eficientemente ambigüedades que puedan surgir al momento de las reducciones respecto a otros parsers "shift-reduce" como el LR(0) o el SLR(1).

Para implementar este parser se crea la clase `LR1Parser` que hereda de la clase `ShiftReduceParser` que tiene los métodos `build_parser_table` y `parse`. Además se utiliza la clase `Item` para representar los items LR(1). La misma tiene las propiedades `production`, `pos`, `lockahead`, esta útltima implementada utilizando la estructura *frozenset* de *Python* que ofrece la ventaja de ser hasheable.

Una instancia del parser tendrá las propiedades `action` y `goto` que representan las tablas correspondientes del mismo. Para la generación de las tablas se utiliza el algoritmo estudiado que emplea las funciones `closure` y `goto` para obtener el autómata del parser. Para construir este autómata se emplea el método `build_LR1_automaton`.

Luego, una vez obtenido el autómata, a partir de la función de transición del mismo, se generan los posibles estados de las tablas, que, dado un índice de un estado y un símbolo posible para la transición, dirá, de forma determinista, la operación a realiar. En el caso de la tabla `action`, sus posibles estados se representan con los valores `ShifReduceParser.SHIT`, `ShiftreduceParser.REDUCE` y `ShiftReduceParser.OK`, que representan las operación *shift* al leer un teminal de la cadena, la operación *reduce*, cuando se tiene en la pila un estado corrrespondiente a un *ReduceItem* y se lee un teminal que pertenece al *lockahead* del *Item*, y por último el *OK*, que reprersenta el estado en que se ha terminado de parsear una cadena válida del lenguaje.

En el caso de la tabla *goto*, se tiene, dado un estado, que corresponderá al estado en que queda el autómata luego de aplicar una producción, y el símbolo correspondiente al no terminal en la parte izquierda de la producción aplicada, tendrá el valor del siguiente estado del autómata.

Dado una cadena, se intentará parsear utilizando el método `parser` mencionado anteriormente. En este método, se va llevando el estado de la pila y el próximo símbolo de la cadena y si no se encuentra una transición en la tabla se asume que la cadena no pertenece al lenguaje. Este método devuelve las opeaciones *Shift* o *Reduce* realizadas para parsear la cadena, así como las producciones correspondientes a cada *Reduce*.

3.3 DSL

Para la ejecución de la simulación y la representación del dominio del problema se desarrolló un lenguaje de dominio específico.

Para representar los símbolos de la gramática definida para el lenguaje se utilizan las clases `Terminal` y `NonTermainal` mencionadas anteriormente. Por cada no terminal de la gramática se definen las producciones correspondientes al mismo, para ello se utiliza la clase `Production`. La sintaxis empleada para representar las producciones es la siguiente:

```
production = Production(NT,[S,K])
```

donde NT∈N y $S, K \in \{TxN\}^+$.

Además se crea un diccionario, cuyas llaves serán las producciones de la gramática y el valor las reglas asociadas a la producción. La forma en la que se definen las reglas es usando funciones `lambda`, que reciben dos parámetros que ser´an los correspondientes a los atributos heredados y sintetizados y se devuelve el consecuente de la regla que define como modificar el *AST* a partir de esta producción.

A continuación se mostrará un ejemplo de una producción asociada a un no terminal, con una funcionalidad propia del lenguaje propuesto para resolver el problema que permitirá ver claramente la forma en que se define la sintaxis.

```
p_76 = Production(
    def_agent,
    [
        type_,
        idx,
        equal,
        ocur,
        activation_condition,
        colon,
        instance,
        semi,
        effect_time,
        colon,
```

```
            num,
            semi,
            repetition,
            colon,
            num,
            semi,
            action,
            colon,
            idx,
            semi,
            ccur,
        ],
    )
    rules[p_76] = lambda _, s: AgentDefNode(s[1], s[2], s[7], s[11], s[15], s[19])
```

Luego para representar la gramática se utiliza la clase `Grammar` que recibe la lista de no terminales y terminales asociada a la misma, así como un diccionario `productions` que por cada no terminal de la gramática tendrá la lista de producciones asociadas a él. También recibe el diccionario correspondiente a las reglas asociadas a las producciones, y el símbolo distinguido de la gramática que en este caso es *program*.

3.4 Chequeo semántico y estático

Para el chequeo semánticco y estático se realizaron tres recorridos sobre el *AST*:

- *Type Collector*: para recolectar todos los tipos definidos en el código.
- *Type Builder*: para identificar todas las funciones y atributos definidos para cada tipo; verificar que los tipos declarados están definidos y verificar que no existe herencia cíclica.
- *Semantic checker*: para verificar el cumplimiento de las reglas semánticas definidas para el lenguaje.

Se definió la clase `Context` que reúne todos los tipos definidos por el usuario y los *built-in*, así como métodos para chequear la existencia de tipos y sus dependencias.

La clase `Scope` se define para englobar todos aquellos *statements* que conformen el cuerpo definido entre llaves de una clase, una función, un *for*, un *if*. Como un *scope* puede contener a otro se define el método `create_child` que permite anidar *scopes* y definir variables que solo son visibles en un *scope* y sus hijos. Implementa una función para definir y buscar funciones. Además se almacenan las variables locales, es decir, que han sido definidas en el mismo ámbito.

Las clases `Attribute, Variable` y `Method` representan a los atributos, variables y funciones respectivamente, a partir de ellas se puede almacenar información relativa a estas entidades.

`Type` es el tipo de aquellos objetos que representan tipos dentro del lenguaje. Alguno de los tipos declarados son: `MainType`, `IntType`, `BoolType`, `StringType`. Se definieron los métodos `get_attribute`, `set_attribute`, `get_function`, `set_function`, `set_parent`, `all_attributes`, `all_methods` y `conforms_to` (se dice que un tipo *A conforms_to B* si *A* puede ocupar el lugar de un tipo *B*).

Las reglas semánticas de nuestro lenguaje son:

- Dos variables no pueden tener el mismo nombre.
- Dos funciones dentro de un mismo *scope* no pueden tener el mismo nombre.
- Un método se puede sobrescribir si y solo si mantiene exactamente la misma definición para los tipos de retorno y de los argumentos.
- Las operaciones $+, -, *, /$ están definidas entre valores numéricos y devuelven números.
- Las operaciones *or* y *and* se definen entre expresiones *booleanas*.
- La asignación se hace a variables definidas con anterioridad. Dicha variable debe tener mismo tipo que el de la expresión asociada.
- La clase `SemanticError` hereda de `Exception` y permite manejar errores en los *contextos*. Posee un campo *text* que permite especificarr el error ocurrido.
- La clase `VoidType` se utiliza para manejar el tipo de retorno *void* de los métodos.
- El tipo de retorno de una función tiene que coincidir con los tipos de aquellas expresiones que se encuentren dentro de un *return statement*. Cuando el tipo de retorno es *void* no puede existir ningún *return statement* en el cuerpo de la función.

3.5 Transpilador

Para la ejecución del lenguaje definido se implementó un transpilador a lenguaje *Python*. Este realiza un recorrido por el *AST* y va construyendo los *statements* en lenguaje *Python* en dependencia del tipo de nodo de que se trate.

Por ejemplo, la expresión correspondiente a un *for*, representada por un `ForNode`, que en nuestro lenguaje se declara como: *"for <idx> in <expr>{<body_statements>}*, en lenguaje *Python* quedaría de la siguiente forma: *for i in expr*.

4 Arquitectura de la aplicación

Treatment Tree parte de una interfaz de usuario sencilla con un editor de texto que permite al usuario, utilizando el *DSL* definido, programar el en-

torno de su simulación. Una vez se tiene el el set de instrucciones definiadas por el usuario se realiza la recolección de tipos, el construcción de tipos, chequeo semántico y se transpila a código *Python*. Luego este se ejecuta llevándose a cabo la simulación, obteniéndose el árbol de tratamientos como resultado.

References

1. Russell, Stuart J., Norvig, Peter.: Artificial intelligence: a modern approach. 4th edn. (2020)