Proyecto Integrador
Inteligencia Artificial, Compilación, Simulación

Bryan Machín García, José Alejandro Solís Fernández y Adrianna Alvarez Lorenzo

Universidad de La Habana,
San Lázaro y L. Plaza de la Revolución, La Habana, Cuba
{bryan.machin,jose.solis,adrianna.alvarez}@estudiantes.matcom.uh.cu
http://www.uh.cu

Resumen En el presente informe se discute una propuesta de diseño e implementación de un sistema para un estudiante en particular, que permita simular diferentes ámbitos de aprendizaje a partir de un entorno inicial, con el propósito de obtener una estrategia de aprendizaje según las materias que este desea aprender.

Palabras Claves: aprendizaje, entorno, estrategia, estudiante, simulación

Abstract. This report discusses a proposal for the design and implementation of a system for a particular student, that allows the simulation of differents learning environments from an initial one, with the purpose of obtain a learning strategy according to the subjects he wants to learn.

Keywords: environment, learning, simulation, strategy, student

1. Introducción

El aprendizaje de un curso supone a un estudiante frente a un conjunto de contenidos de los cuales, a priori, puede desconocer absolutamente de su naturaleza. Por esta razón, la optimización de un proceso de aprendizaje es un trabajo que le presentaría altas dificultades. Este trabajo propone una alternativa de solución a dicha problemática.

2. Dominio del Problema

2.1. Elemento

Definición 1 *Es una materia o disciplina que se centra en un área de conocimiento diferenciada.*

Un elemento puede tener un conjunto de dependencias a otros elementos. Es decir, no se puede aprender el elemento e sin aprender el elemento e', si este último es dependencia de e.

2.2. Actividad

Definición 1 *Es una acción que posibilita el desarrollo del proceso de aprendizaje de ciertos elementos.*

Una actividad se constituye de la siguiente manera:

- conjunto de elementos que intervienen en la actividad
- cantidad de puntos de conocimiento que brinda por cada uno de los elementos
- tiempo estimado de duración

2.3. Entorno de Aprendizaje

Un entorno se compone de recursos que determinan cierto ámbito de aprendizaje:

- un estudiante
- un conjunto de elementos
- un conjunto de actividades
- un conjunto de reglas

El ente principal que caracteriza a un entorno de aprendizaje es un *estudiante*, el cual tiene como propósito aprender un subconjunto de elementos del conjunto inicial(*objetivos*), según las condiciones definidas para él en dicho entorno. Estas condiciones están dadas por el estado en que se encuentra el estudiante en cada uno de los elementos que conforma el entorno, denominadas *categorías*.

2.4. Categoría

Definición 1 *Especifica el nivel de aprendizaje en el que se encuentra un estudiante en cierto elemento.*

Las categorías existentes son:

- aprendido
- aprendible
- no aprendido
- olvidado

La transición de una categoría a otra se determina por el cumplimiento de reglas:
Ejemplo: Para que un elemento pase de la categoría *no aprendido* a la categoría *aprendible* es necesario que todas las dependencias de ese elemento estén en *aprendido*.

Además, la probabilidad de dicho tránsito puede estar condicionada por varios factores que se definan.
Ejemplo Un elemento puede pasar de la categoría *olvidado* a la categoría *aprendido* de manera totalmente aleatoria o pudiera ser que la probabilidad dependa de cuántas dependencias de ese elemento se encuentren en la categoría *olvidado* o *aprendido*.

3. Problema

Se desea diseñar e implementar un sistema que permita simular diferentes entornos de aprendizaje para un mismo estudiante a partir de un entorno inicial, basándose en el siguiente criterio:

Un entorno x difiere de un entorno x' en el nivel de aprendizaje de dicho estudiante en los elementos que lo constituyen.

Con lo antes mencionado, se propone obtener una estrategia de aprendizaje de elementos, en la que el estudiante en cuestión logre aprender la cantidad máxima de objetivos posibles, según las condiciones predefinidas de su entorno.

4. Modelación del Problema

Sea E un entorno de aprendizaje y un estudiante S. Se tiene que:

El conjunto de elementos de E se puede representar como un grafo dirigido $G = (V, A)$, donde:

- V es el conjunto de elementos de E
- $A = \{< e_1, e_2 > \in V \times V : e_2 \in D(e_1)\}$, siendo D el conjunto de dependencias de e_1.

G es un grafo acíclico dirigido(DAG). En efecto:
Como un arco $< e_1, e_2 >$ indica que e_1 depende de e_2, entonces sin pérdida de generalidad, la existencia de un ciclo $c = \{e_1, e_2, e_3, e_1\}$ implicaría que e_1 es dependencia de sí mismo, lo cual no tiene sentido en este modelo.

Luego, para determinar el nivel de aprendizaje del estudiante S en cada uno de los vértices de G se verifica el cumplimiento de las reglas definidas. Como resultado se obtiene la ubicación de estos en cada una de las categorías predeterminadas del sistema, al adaptar S a las condiciones impuestas por E.

Dado que G es un DAG, el conjunto de posibles estrategias de aprendizaje para satisfacer los objetivos de S son ordenaciones topológicas de G, pues la construcción de estas se basan en un orden de elementos dado por prioridades, de manera tal que el elemento e sea aprendible cuando se validen las restricciones de esa regla según sus dependencias.

Se considera como solución del problema una secuencia ordenada de los objetivos que deben ser aprendidos. Algunas soluciones pueden ser mejores que otras y el valor por el cuál se comparan es el porciento de aprendizaje obtenido al realizarse.

Para cada objetivo de la solución se buscan varios caminos en los que se aprenden

contenidos que a través del sistema de dependencias permitan aprenderlo. Estos caminos se encuentran siguiendo distintas métricas: buscando dependencias cuyos puntos disponibles a obtener sean mayores o que la cantidad de elementos por aprender para que su categoría mejore sea menor, dado que estos tienen un mayor grado de probabilidad de éxito. También se busca un camino aleatorio para incrementar la exploración del grafo. Además, estos caminos son simulados varias veces, devolviendo su porcentaje de aprendizaje promedio y tiempo que toman en completarse.

```
def search_strat(goal, env, strat_name):
    strat_env = env.clone_environment()
    strat = []
    if strat_name == "avp":
        avps_visit(goal, strat, strat_env, 5)
    elif strat_name == "lm":
        lm_visit(goal, strat, strat_env, 5)
    else:
        rnd_visit(goal, strat, strat_env, 5)
    return strat


def avps_visit(v, stack, env, behind_count):
    if env.student.categories[v] == "Learnable":
        return stack.insert(0, v)
    deps_needed = math.ceil(len(v.dependencies) * env.rules_params[0]) - learned_deps(v, env)
    maxs = []
    for dep in v.dependencies:
        if len(maxs) < deps_needed:
            maxs.append(dep)
            continue
        maxs.sort(key=lambda x: x.available_points)
        if dep.available_points > maxs[0].available_points:
            maxs.pop(0)
            maxs.append(dep)
    for dep in maxs:
        avps_visit(dep, stack, env, len(stack))
    return stack.insert(len(stack) - behind_count, v)

```

Una solución inicial factible se puede encontrar creando un orden topológico con los objetivos. Esta solución se somete a un proceso en el cual se buscan soluciones cercanas a ellas, que pertenezcan a una misma vecindad variable. Se realiza un

solo intercambio en el orden de un par de objetivos y se evalúa el resultado de esta nueva solución. El par de soluciones es, entonces, comparado para obtener la mejor y repetir el proceso. A pesar que de esta manera se busca una mejora de solución, es probable que se caiga en un óptimo local. Para contrarrestar esto, se utiliza la idea de una población de soluciones, aumentando la exploración y abarcando mayor terreno en el espacio de soluciones.

```
25  def build_strategies(main_env):
26      strategies = []
27      add_attributes(main_env.student.goals, "dep_goals", [])
28      goals = topological_sort(main_env.elements, main_env.student.goals)
29      strategies.append(goals)
30      for i in range(5):
31          strategies.append(other_topological_sort(goals))
32      return strategies
33
34
35  def vns(strategy, main_env):
36      a = randint(0, len(strategy) - 1)
37      b = randint(0, len(strategy) - 1)
38      while b == a:
39          b = randint(0, len(strategy) - 1)
40      temp = strategy[a]
41      temp2 = strategy[b]
42      new_strategy = []
43      for i in strategy:
44          if i == temp:
45              new_strategy.append(temp2)
46              continue
47          if i == temp2:
48              new_strategy.append(temp)
49              continue
50          new_strategy.append(i)
51      return [new_strategy, objective_function(new_strategy, main_env)]
52
```

La imagen anterior muestra como se construye un conjunto de soluciones de tamaño variable y como, para una solución particular, se buscan soluciones vecinas.

Para que las simulaciones se puedan lleavar a cabo satisfactoriamente y el resultado de evaluar las soluciones sea certero, es imprescindible un correcto y

apropiado manejo de las categorías en las cuales se encuentran los elementos. Esta tarea es llevada a cabo por un agente que interactúa en y con el ambiente. Cuando este agente toma acción, decide si un elemento cambia de categoría o se mantiene en la misma.

```python
class Categorizer:
    def check_rules(self, env, element):
        env.student.categories[element] = env.rules[env.student.categories[element]](env, element)

    def recheck_categories(self, elements, env):
        add_attributes(elements, "visited", None)
        for element in elements:
            if element.visited is None:
                self.recheck_cat_visit(element, env)
        delete_attributes(elements, "visited")

    def recheck_cat_visit(self, v, env):
        v.visited = 1
        for u in v.dependencies:
            if u.visited is None:
                self.recheck_cat_visit(u, env)
        self.check_rules(env, v)

```

Categorizer

También es importante no mantenerse en una simulación que persigue un objetivo que no puede ser alcanzado. Para evitar esta situación se toma auxilio de otro agente, que observa la repetición de un mismo contenido en la simulación y decide cuando es momento de dejar de intentar aprenderlo para el estado actual del ambiente.

```python
class Adviser:

    def stop(self, count):
        if count > 7:
            r = random()
            if r < 0.5:
                return True
        if count > 5:
            r = random()
            if r < 0.35:
                return True
        return False
```

Adviser

Para aprender un elemento es necesario realizar un conjunto de actividades, por lo que es importante elegir una sucesión de actividades apropiada a realizar. Distintas de estas sucesiones son revisadas, para explorar los escenarios posibles y más probables en los que el estudiante se puede ver envuelto: escogiendo las actividades qué más puntos de aprendizaje le pueda aportar, la que menos tiempo tome o mediante una elección aleatoria.

```
            if next_content in activity.elements:
                activities.append(activity)
        if not activities:
            break
        front = make_front(activities, next_content, env.student)
        r = randint(0, len(front) - 1)
        time += env.perform_activity(front[r])
    p = 0
    for goal in env.student.goals:
        if env.student.categories[goal] == "Learned":
            p += 1
    t = len(main_env.student.goals)
    return [100 * p / t, time, env]


def make_front(activities, element, student):
    front = []
    comparable = False
    for activity in activities:
        for front_activity in front:
            comp = compare_activities(activity, front_activity, element, student)
            if comp == "better":
                comparable = True
                front.remove(front_activity)
                front.append(activity)
                break
            elif not comp == "not_comparable":
                comparable = True
                break
        if not comparable:
            front.append(activity)
    return front
```

En esta imagen se observa como para el próximo elemento que debe ser aprendido se buscan las actividades que puedan intervenir en su aprendizaje. Estas actividades pueden diferir en diversas características y para no elegir arbitrariamente o dar más prioridad a una de esas características sobre otras, se realiza un frente de actividades que no son comparables entre sí: por ejemplo si se tienen las actividades a y b, en donde los puntos que puede otorgar a son mayores a los puntos que puede otorgar b, pero el tiempo de ejecución de a es menor.

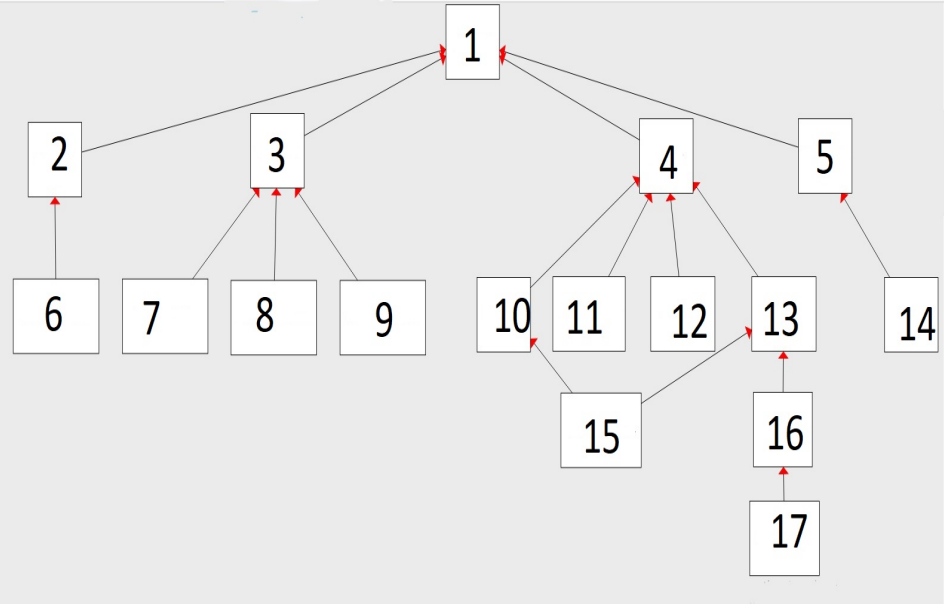
4.1. Ejemplo de modelado

A continuación se presenta un ejemplo de modelado para un proceso de aprendizaje de un curso de Python básico. El curso está compuesto de diversos contenidos, algunos de los cuales presentan dependencias entre si. Los contenidos son:

- 1: Temas básicos
- 2: Operadores
- 3: Colecciones
- 4: Funciones
- 5: Condicionales
- 6: Métodos de cadena
- 7: Métodos de lista
- 8: Métodos de diccionario
- 9: Comprensión de lista
- 10: Decoradores
- 11: Funciones lambda
- 12: Generadores
- 13: Objetos
- 14: Ciclos
- 15: Clases decoradoras
- 16: Herencia
- 17: Herencia múltiple

Para este curso se eligen como objetivos los contenidos de: ciclos, clases decoradoras, herencia múltiple y comprensión de lista. Se definen también las actividades de *video tutorial*, *documento conferencia* y *ejercicio práctico* para cada uno de los distintos temas o contenidos.

Por defecto, solo aquellos temas que no tengan dependencias comenzarán con categoría de aprendible, mientras que los restantes tendrán en un inicio categoría de no aprendible. Los parámetros para la evaluación de reglas de cambio de categoría son fijados por el ususario. En este ejemplo basta con que un elemento tenga el 50 % de sus dependencias aprendidas para convertirse en aprendible, y alcanzar un mínimo de 6 puntos en un contenido para aprenderlo.



Dependencias

Del resultado de la simulación se puede obtener mucha información, como por ejemplo: saber qué elementos fue necesario aprender, en qué orden, cuánto tiempo de estudio se requiere, qué porciento de aprendizaje de objetivos se obtiene, etc.

```
Orden de aprendizaje de contenidos:
1 -- tipos_basicos
2 -- condicionales
3 -- ciclos
4 -- funciones
5 -- objetos
6 -- clases_decoradoras
7 -- herencia
8 -- herencia_multiple
9 -- colecciones
10 -- comprension_de_lista
Porcentaje de objetivos aprendidos: 100.0 %
Tiempo transcurrido: 71.0 h
```

Se observa como para este ejemplo las actividades disponibles son suficientes para lograr totalmente el objetivo trazado.

5. Compilación

Para el manejo del sistema se emplea el lenguaje de dominio específico *learnPro*. Este lenguaje se modela a partir de un conjunto de símbolos, cuya implementación se encuentra en la clase `Symbol`. Esto permitirá posteriormente la definición símbolos terminales y símbolos no terminales.

```
class Symbol(object):
    def __init__(self, name, grammar):
        self.name = name
        self.grammar = grammar

    def __repr__(self):
        return repr(self.name)

    def __str__(self):
        return self.name

    def __add__(self, other):
        if isinstance(other, Symbol):
            return Sentence(self, other)
        raise TypeError(other)

    def __or__(self, other):
        if isinstance(other, Sentence):
            return SentenceList(Sentence(self), other)
        raise TypeError(other)

    def __len__(self):
        return 1

    @property
    def is_epsilon(self):
        return False
```

Mediante el uso de símbolos se facilita la formación de oraciones al agruparse con el operador +, permitiendo el reconocimiento de la cadena especial **epsilon** a través de la propiedad `is_epsilon`. Además, posibilita el acceso a la gramática en la que se definió mediante el campo `Grammar` que contiene cada instancia, así como la consulta de su notación a través del campo `Name`.

En el caso de los símbolos no terminales, su modelación se encuentra en la clase `NonTerminal`, la cual extiende a la clase `Symbol` para permitir reconocer las producciones que tienen a este símbolo como cabecera, mediante el campo `productions` de cada instancia; añadir producciones para ese no terminal a través del operador $\%=$ e incluir las propiedades `is_non_terminal` e `is_terminal` que devolverán `True` o `False` respectivamente. Esto último se

añade de igual manera para los símbolos terminales, cuya modelación se encuentra en la clase `Terminal`.

```python
class NonTerminal(Symbol):
    def __init__(self, name, grammar):
        super().__init__(name, grammar)
        self.Productions = []

    def __str__(self):
        return self.name

    def __mod__(self, other):
        if isinstance(other, Sentence):
            p = Production(self, other)
            self.grammar.add_production(p)
            return self
        if isinstance(other, tuple):
            if len(other) == 2:
                other += (None,) * len(other[0])
            # Debe definirse una regla por cada símbolo de la producción
            if isinstance(other[0], Symbol) or isinstance(other[0], Sentence):
                p = AttributeProduction(self, other[0], other[1:])
            else:
                raise Exception("")
            self.grammar.add_production(p)
            return self
        if isinstance(other, Symbol):
            p = Production(self, Sentence(other))
            self.grammar.add_production(p)
            return self
        if isinstance(other, SentenceList):
            for s in other:
                p = Production(self, s)
                self.grammar.add_production(p)
            return self
        raise TypeError(other)
```

La clase `EOF` modela el símbolo de fin de cadena, cuyo comportamiento se hereda al extender la clase `Terminal`.

```
class EOF(Terminal):
    def __init__(self, grammar):
        super().__init__('eof', grammar)

    def __str__(self):
        return 'eof'
```

Las oraciones y formas oracionales de este lenguaje se modelarán con la clase `Sentence`, siendo una colección de terminales y no terminales.

```
class Sentence(object):
    def __init__(self, *args):
        self._symbols = tuple(x for x in args if not x.is_epsilon)
        self.hash = hash(self._symbols)

    def __len__(self):
        return len(self._symbols)

    def __add__(self, other):
        if isinstance(other, Symbol):
            return Sentence(*(self._symbols + (other,)))
        if isinstance(other, Sentence):
            return Sentence(*(self._symbols + other._symbols))

    def __or__(self, other):
        if isinstance(other, Sentence):
            return SentenceList(self, other)
        if isinstance(other, Symbol):
            return SentenceList(self, Sentence(other))

    def __str__(self):
        return ("%s " * len(self._symbols) % tuple(self._symbols)).strip()

    def __iter__(self):
        return iter(self._symbols)

    def __getitem__(self, index):
        return self._symbols[index]

    def __eq__(self, other):
        return self._symbols == other._symbols

    @property
    def is_epsilon(self):
        return False
```

Con esta se conoce a priori la longitud de la oración, además de que se accede a los símbolos que componen la oración a través del campo `symbols` de cada

instancia, y se puede conocer si dicha oración está completamente vacía a través de la propiedad `is_epsilon`. Mediante el operador + se puede obtener la concatenación con un símbolo u otra oración.

Para logar definir las producciones que tengan la misma cabecera en una única sentencia, se emplea el agrupamiento de oraciones usando el operador |, lo cual se maneja con la clase `SentenceList`.

```
74  class SentenceList(object):
75      def __init__(self, *args):
76          self._sentences = list(args)
77
78      def add(self, symbol):
79          if not symbol and (symbol is None or not symbol.is_epsilon):
80              raise ValueError(symbol)
81          self._sentences.append(symbol)
82
83      def __or__(self, other):
84          if isinstance(other, Sentence):
85              self.add(other)
86              return self
87
88          if isinstance(other, Symbol):
89              return self | Sentence(other)
90
91      def __iter__(self):
92          return iter(self._sentences)
```

En la clase `Epsilon` se modelará tanto la cadena vacía como el símbolo que la representa: ϵ. Dicha clase extiende las clases `Terminal` y `Sentence` por lo que adopta el comportamiento de ambas, sobreescribiendo la implementación del método `is_epsilon` para indicar que toda instancia de la clase reprensenta epsilon.

```python
class Epsilon(Terminal, Sentence):
    def __init__(self, grammar):
        super().__init__('epsilon', grammar)

    def __hash__(self):
        return hash("")

    def __len__(self):
        return 0

    def __str__(self):
        return "e"

    def __repr__(self):
        return 'epsilon'

    def __iter__(self):
        yield from ()

    def __add__(self, other):
        return other

    def __eq__(self, other):
        return isinstance(other, (Epsilon,))

    @property
    def is_epsilon(self):
        return True
```

La clase `Production` modela las producciones, con la cual se puede acceder a la cabecera y cuerpo de cada producción mediante los campos `left` y `right` respectivamente, así como consultar si la producción es de la forma $X \to \epsilon$ haciendo uso de la propiedad `is_epsilon` y bifurcar la producción en cabecera y cuerpo haciendo uso de las asignaciones: `left, right = production`.

- Para definir una producción de la forma $E \to E + T$:

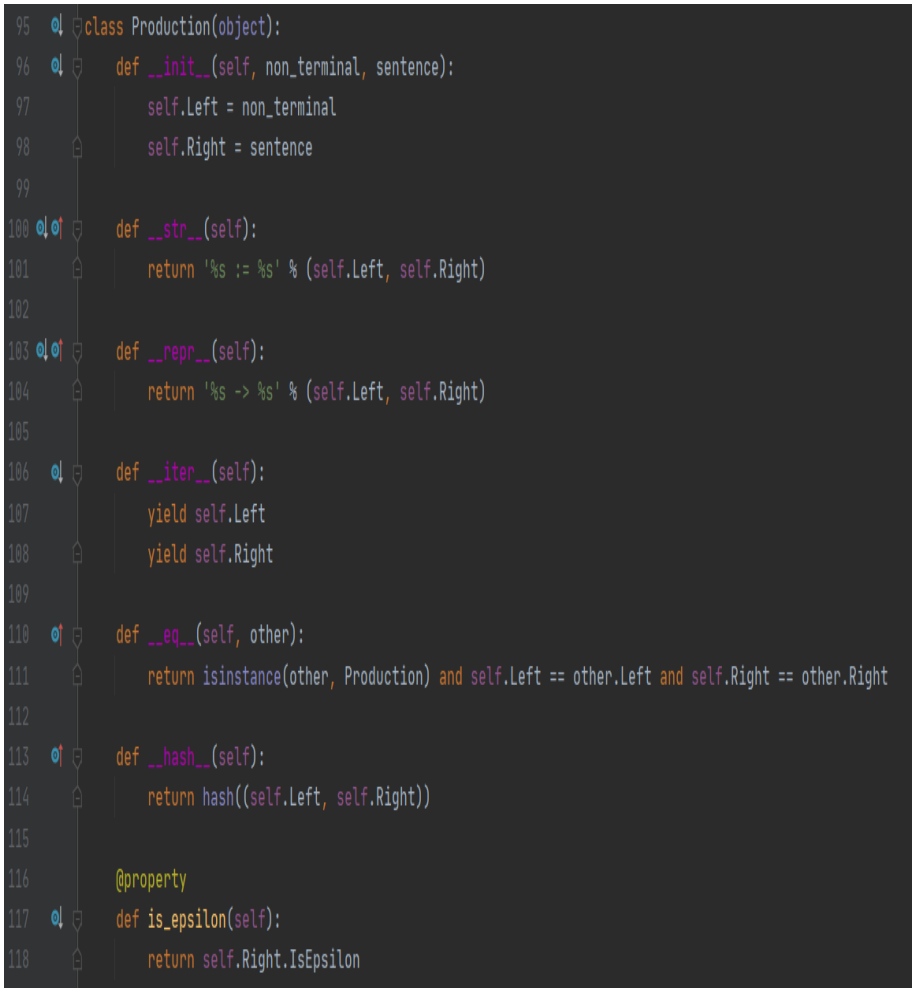
$$E\,\%= E + \textbf{plus} + T$$

- Para definir múltiples producciones de la misma cabecera en una única sentencia ($E \to E + T|E - T|T$):

$$E\,\%= E\ \textbf{plus} + T|E + \textbf{minus} + T|T$$

- Para usar ϵ en una producción, por ejemplo, $S \longrightarrow aS|\epsilon$ se procederá de la siguiente manera:

$$S\,\%= S + a|\textbf{G.Epsilon}$$



```python
class Production(object):
    def __init__(self, non_terminal, sentence):
        self.Left = non_terminal
        self.Right = sentence

    def __str__(self):
        return '%s := %s' % (self.Left, self.Right)

    def __repr__(self):
        return '%s -> %s' % (self.Left, self.Right)

    def __iter__(self):
        yield self.Left
        yield self.Right

    def __eq__(self, other):
        return isinstance(other, Production) and self.Left == other.Left and self.Right == other.Right

    def __hash__(self):
        return hash((self.Left, self.Right))

    @property
    def is_epsilon(self):
        return self.Right.IsEpsilon
```

La modelación de las gramáticas se realiza en la clase `Grammar`. Sus funcionalidades básicas son definir de la gramática los símbolos terminales, a través de los métodos `terminal` y `terminals` y los no terminales mediante `non_terminal` y `non_terminals` y denotar las producciones de la gramática a partir de la aplicación del operador % = entre no terminales y oraciones. A su vez, se puede acceder a todas las producciones a través del campo `Productions` de cada instancia, a los terminales y no terminales mediante los campos `Terminals` y `NonTerminals` respectivamente, y al símbolo inicial, _epsilon y fin de cadena(EOF) a través de los campos StartSymbol, Epsilon y EOF respectivamente.

```
260     def non_terminal(self, name, start_symbol=False):
261         if not name:
262             raise Exception("Empty")
263         term = NonTerminal(name, self)
264         if start_symbol:
265             if self.Start_symbol is None:
266                 self.Start_symbol = term
267             else:
268                 raise Exception('Cannot define more than one start symbol')
269         self.Non_terminals.append(term)
270         self.SymbolDict[name] = term
271         return term
272 
273     def non_terminals(self, names):
274         aux = tuple(self.non_terminal(i) for i in names.strip().split())
275         return aux
276 
277     def terminal(self, name):
278         if not name:
279             raise Exception('Empty')
280         term = Terminal(name, self)
281         self.Terminals.append(term)
282         self.SymbolDict[name] = term
283         return term
284 
285     def terminals(self, names):
286         aux = tuple(self.terminal(i) for i in names.strip().split())
287         return aux
```

Para el manejo de la pertenencia o no de *epsilon* a un conjunto se emplea la clase `ContainerSet`, la cual funciona como un conjunto de símbolos, posibilitando consultar la pertenencia de epsilon al conjunto. Las operaciones que modifican el conjunto devuelven si hubo cambio o no. Dicho conjunto puede ser actualizado con la adición de elementos individuales, con el método `add`, o a partir de otro conjunto, mediante `update` y `hard_update`.

```python
    def add(self, value):
        n = len(self.set)
        self.set.add(value)
        return n != len(self.set)

    def extend(self, values):
        change = False
        for value in values:
            change |= self.add(value)
        return change

    def set_epsilon(self, value=True):
        last = self.contains_epsilon
        self.contains_epsilon = value
        return last != self.contains_epsilon

    def update(self, other):
        n = len(self.set)
        self.set.update(other.set)
        return n != len(self.set)

    def epsilon_update(self, other):
        return self.set_epsilon(self.contains_epsilon | other.contains_epsilon)

    def hard_update(self, other):
        return self.update(other) | self.epsilon_update(other)
```

Por otra parte, el conjunto First de una forma oracional se define como:

- **First**$(w) = \{x \in V_t | w \Rightarrow^* x\alpha, \alpha \in (V_t \bigcup V_n)^*\}$
- $\bigcup\{\epsilon\}$, si $w \rightarrow^* \epsilon$
- $\bigcup\{\}$, en otro caso.

Este es posible computarlo para los símbolos terminales, no terminales y producciones haciendo uso de un método de punto fijo. Para ello los *firsts* se inicializan vacíos y mediante las siguientes reglas se van se van actualizando con la aplicación de forma incremental:

- Si $X \longrightarrow W1|W2|...|Wn$ entonces por definición: First(X) $= \bigcup_i$ First(W_i)
- Si $X \longrightarrow \epsilon$ entonces $\epsilon \in$ First(X)

- Si $W = xZ$ donde x es un símbolo terminal, entonces First(W) = {x}
- Si $W = YZ$ donde Y es un símbolo no terminal y Z una forma oracional, entonces First(Y) $\subseteq$ First(W)
- Si $W = YZ$ y $\epsilon \in$ First(Y) entonces First(Z) $\subseteq$ First(W)

El cálculo de los `firsts` se da por terminado cuando finalice una iteración sin que se produzcan cambios.

Para la implementación de dicho algoritmo se tiene el método `compute_local_first`, que calcula el `First(alpha)`, siendo `alpha` una forma oracional.

```
42  def compute_local_first(firsts, alpha):
43      first_alpha = ContainerSet()
44      try:
45          alpha_is_epsilon = alpha.is_epsilon
46      except:
47          alpha_is_epsilon = False
48      if alpha_is_epsilon:
49          first_alpha.set_epsilon()
50      else:
51          for item in alpha:
52              first_symbol = firsts[item]
53              first_alpha.update(first_symbol)
54              if not first_symbol.contains_epsilon:
55                  break
56          else:
57              first_alpha.set_epsilon()
58      return first_alpha
```

Con el método `compute_firsts` se calculan todos los conjuntos firsts actualizando a los conjuntos iniciales según los resultados de aplicar `compute_local_first` en cada producción.

```python
def compute_firsts(g):
    firsts = {}
    change = True
    # init First(Vt)
    for terminal in g.Terminals:
        firsts[terminal] = ContainerSet(terminal)
    # init First(Vn)
    for non_terminal in g.Non_terminals:
        firsts[non_terminal] = ContainerSet()
    while change:
        change = False
        # P: X -> alpha
        for production in g.Productions:
            x = production.Left
            alpha = production.Right
            first_x = firsts[x]
            # init First(alpha)
            try:
                first_alpha = firsts[alpha]
            except:
                first_alpha = firsts[alpha] = ContainerSet()
            local_first = compute_local_first(firsts, alpha)
            # update First(X) and First(alpha) from CurrentFirst(alpha)
            change |= first_alpha.hard_update(local_first)
            change |= first_x.hard_update(local_first)

    # First(Vt) + First(Vt) + First(RightSides)
    return firsts
```

Una *gramática atributada* es una tupla $< G, A, R >$ donde:

- G $= < S, P, N, T >$ es una gramática libre del contexto
- A es un conjunto de atributos de la forma X $\cdot\alpha$ donde $X \in N \bigcup T$ y α es un identificador único entre todos los atributos del mismo símbolo y,
- R es un conjunto de reglas de la forma $< p_i, r_i >$ donde $p_i \in P$ es una producción $X \longrightarrow Y_1, ..., Y_n$ y r_i es una regla de la forma:
  1. X $\cdot\alpha = f(Y_1 \cdot \alpha_1, ..., Y_n \cdot \alpha_n)$, o
  2. $Y_i \cdot \alpha = f(X \cdot \alpha_0, Y_1 \cdot \alpha_1, ..., Y_n \cdot \alpha_n)$

Los atributos se dividen en dos conjuntos disjuntos: *atributos heredados* y *atributos sintetizados*, como es el caso de α en (1) y en (2) respectivamente.
Las condiciones suficientes para que una gramática sea evaluable son:

- Una gramática atributada es *s-atributada* si y solo si, para toda regla r_i asociada a una producción $X \to Y_1, \ldots, Y_n$, se cumple que r_i es de la forma $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$.
- Una gramática atributada es *l-atributada* si y solo si toda regla r_i asociada a una producción $X \to Y_1, \ldots, Y_n$ es de una de las siguientes formas:
  1. $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$, ó
  2. $Y_i \cdot a_i = f(X \cdot a, Y_1 \cdot a_1, \ldots, Y_{i-1} \cdot a_{i-1})$

A la *API* de gramáticas se añade una nueva clase: `AttributeProduction`.

```
121  class AttributeProduction(Production):
122      def __init__(self, non_terminal, sentence, attributes):
123          if not isinstance(sentence, Sentence) and isinstance(sentence, Symbol):
124              sentence = Sentence(sentence)
125          super(AttributeProduction, self).__init__(non_terminal, sentence)
126          self.attributes = attributes
127
128      def __str__(self):
129          return '%s := %s' % (self.Left, self.Right)
130
131      def __repr__(self):
132          return '%s -> %s' % (self.Left, self.Right)
133
134      def __iter__(self):
135          yield self.Left
136          yield self.Right
137
138      @property
139      def is_epsilon(self):
140          return self.Right.IsEpsilon
```

Con esta clase se modela las producciones de las gramáticas atributadas. Cada una de estas producciones se compone por un símbolo no terminal como cabecera, accesible a través del campo `Left`, una oración como cuerpo, a través del campo `Right` y un conjunto de reglas para evaluar los atributos, accesible a través del campo `atributes`.

Se implementó la clase `Item` para modelar los items del parser LR(1), cuyos

`lookaheads` se almacenarán haciendo uso del parámetro `lookaheads`.

Cada item tiene definido una función `Preview`, la cual devuelve todas las posibles cadenas que resultan de concatenar lo que queda por leer del item tras saltarse x símbolos con los posibles `lookaheads`, que resultan de calcular el `first` de estas cadenas.

```
371     @property
372     def is_reduce_item(self):
373         return len(self.Production.Right) == self.Pos
374
375     @property
376     def next_symbol(self):
377         if self.Pos < len(self.Production.Right):
378             return self.Production.Right[self.Pos]
379         else:
380             return None
381
382     def next_item(self):
383         if self.Pos < len(self.Production.Right):
384             return Item(self.Production, self.Pos + 1, self.Lookaheads)
385         else:
386             return None
387
388     def preview(self, skip=1):
389         return [_self.Production.Right[self.Pos + skip:] + (lookahead,) for lookahead in self.Lookaheads]
390
391     def center(self):
392         return Item(self.Production, self.Pos)
```

Para calcular la clausura se implementó la función `expand`, que recibe un item LR(1) y retorna un conjunto de items que sugiere incorporar debido a la presencia de un $\cdot$ delante de un no terminal.

$$expand(\text{"}Y \to \alpha.X\delta, c\text{"}) = \text{"}X \to .\beta, b\text{"} | b \in First(\delta c)$$

```python
    @staticmethod
    def expand(item, firsts):
        next_symbol = item.next_symbol
        if next_symbol is None or not next_symbol.is_non_terminal:
            return []
        lookaheads = ContainerSet()
        #  (Compute lookahead for child items)
        # calcular el first a todos los preview posibles
        for p in item.preview():
            for first in compute_local_first(firsts, p):
                lookaheads.add(first)
        _list = []
        for production in next_symbol.Productions:
            _list.append(Item(production, 0, lookaheads))
        return _list
```

Luego se implementó la función `compress`, que recibe un conjunto de items LR(1) y devuelve dicho conjunto pero combinando los `lookaheads` de los items con mismo centro.

```python
@staticmethod
def compress(items):
    centers = {}
    for item in items:
        center = item.center()
        try:
            lookaheads = centers[center]
        except KeyError:
            centers[center] = lookaheads = set()
        lookaheads.update(item.Lookaheads)
    return {Item(x.Production, x.Pos, set(lookahead)) for x, lookahead in centers.items()}
```

Teniendo en cuenta ambas funciones, se implementó la función de clausura utilizando la técnica de punto fijo, basándose en lo siguiente:

$CL(I) = I \bigcup \{X \longrightarrow \cdot\beta, b\}$ tales que:

- $Y \longrightarrow \alpha \cdot X\delta, c \in CL(I)$
- $b \in \mathbf{First}(\delta c)$

```python
def closure_lr1(self, items, firsts):
    closure = ContainerSet(*items)
    changed = True
    while changed:
        new_items = ContainerSet()
        # por cada item hacer expand y añadirlo a new_items
        for item in closure:
            e = self.expand(item, firsts)
            new_items.extend(e)
        changed = closure.update(new_items)
    return self.compress(closure)
```

Por otro lado se tiene la implementación de la función `goto(Ii, s)`, que cumple que:

$$\textbf{Goto}(I, X) = CL(\{Y \to \alpha X \cdot \beta, c | Y \to \alpha \cdot X\beta, c \in I\})$$

```python
def goto_lr1(self, items, symbol, firsts=None, just_kernel=False):
    assert just_kernel or firsts is not None, '`firsts` must be provided if `just_kernel=False`'
    items = frozenset(item.next_item() for item in items if item.next_symbol == symbol)
    return items if just_kernel else self.closure_lr1(items, firsts)
```

Esta función recibe como parámetro un conjunto de items y un símbolo, y retorna el conjunto `goto(items, symbol)`. Este método permite darle valor al parámentro `just_kernel=True` para calcular el conjunto de items kernels. En caso contrario, se requiere el conjunto con los `firsts` de la gramática para entonces calcular la clausura.

A continuación se muestra la implementación del algoritmo para construir el automáta LR(1):

```
def build_automata(self, g):
    assert len(g.Start_symbol.Productions) == 1, 'Grammar must be augmented'
    firsts = compute_firsts(g)
    firsts[g.Eof] = ContainerSet(g.Eof)
    start_production = g.Start_symbol.Productions[0]
    start_item = Item(start_production, 0, lookaheads=(g.Eof,))
    start = frozenset([start_item])
    closure = self.closure_lr1(start, firsts)
    automata = State(frozenset(closure), True)
    pending = [start]
    visited = {start: automata}
    while pending:
        current = pending.pop()
        current_state = visited[current]
        for symbol in g.Terminals + g.Non_terminals:
            # (Get/Build `next_state`)
            a = self.goto_lr1(current_state.state, symbol, firsts, True)
            if not a:
                continue
            try:
                next_state = visited[a]
            except KeyError:
                next_state = State(frozenset(self.goto_lr1(current_state.state, symbol, firsts)), True)
                visited[a] = next_state
                pending.append(a)
            current_state.add_transition(symbol.name, next_state)
    automata.set_formatter(multiline_formatter)
    return automata
```

Este parser LR(1) llena la tabla Acción-Goto de la siguiente manera:

- Sea "$X \to \alpha \cdot c\omega, s$" un item del estado I_i y `Goto(`I_i`, c) =` I_j, entonces $ACTION[I_i, c] = 'S'_j$ item Sea "$X \to \alpha., s$ün item del estado I_i, entonces $ACTION[I_i, s] = 'R'_k$ (producción `k` es $X \to \alpha$).
- Sea I_i el estado que contiene el item "$S' \to S., \$$"(S' símbolo inicial), entonces $ACTION[I_i, \$] =$ 'OK'

- Sea "$X \to \alpha.Y\omega$, sïtem del estado I_i Y $Goto(I_i, Y) = I_j$, entonces $Goto[I_i, Y] = j$

```
def build_parsing_table(self):
    g = self.g.augmented_grammar(True)
    automata = self.build_automata(g)
    for i, node in enumerate(automata):
        node.idx = i
    for node in automata:
        idx = node.idx
        for item in node.state:
            p = item.Production
            if item.is_reduce_item:
                if p.Left == g.Start_symbol:
                    self._register(self.action, (idx, self.g.Eof.name), (ShiftReduce.OK, None))
                else:
                    for c in item.Lookaheads:
                        self._register(self.action, (idx, c.name), (ShiftReduce.REDUCE, p))
            else:
                if item.next_symbol.is_terminal:
                    self._register(self.action, (idx, item.next_symbol.name),
                                   (ShiftReduce.SHIFT, node[item.next_symbol.name][0].idx))
                else:
                    self._register(self.goto, (idx, item.next_symbol.name), node[item.next_symbol.name][0].idx)
            pass
```

6. Recomendaciones

El uso de la simulación para la mejora y optimización del aprovechamiento de un proceso de aprendizaje es muy aprovechable y expansible. Sería interesante incluir nuevas categorías y factores que afecten al estudiante y, por consiguiente, su capacidad de aprendizaje frente a una actividad dada.

También explorar el uso de otras metaheurísticas como algoritmos genéticos que crucen dos estrategias y formen una nueva con maneras distintas de alcanzar los objetivos, y el uso de la modelación de planificación.

Otra idea pudiese ser investigar con otro tipo de soluciones, una en la que no se

vean reflejados los contenidos que se aprenden y su orden, sino las actividades que se van realizando.

7. Conclusiones

En este proyecto se ha puesto en práctica mucho del contenido recibido en las diversas asignaturas que lo integran, no solo de manera explícita sino como parte del proceso de analizar, investigar y dar solución al mismo.