



UNIVERSIDAD DE LA HABANA
FACULTAD DE MATEMÁTICA Y COMPUTACIÓN

Battle-sim: Simulador de enfrentamientos bélicos

Proyecto de Inteligencia Artificial, Compilación y Simulación

Equipo de desarrollo:
Rocio Ortiz Gancedo - C311
Carlos Toledo Silva - C311
Ariel A. Triana Pérez - C311

Índice general

1. Introducción	1
2. La Simulación	2
2.1. El mapa	2
2.2. Los objetos	4
3. Battle Script, el lenguaje de dominio específico para ejecutar el proyecto	7
3.1. Gramática	7
3.2. Compilador	9
3.3. Tokenizador	9
4. Reflejo de las asignaturas en el proyecto	17
4.1. Simulación	17
4.2. Compilación	17
4.3. Inteligencia Artificial	17
5. Aplicaciones	18

1. Introducción

A lo largo de la historia, los conflictos bélicos han estado fuertemente ligados al desarrollo de la humanidad. Existen pruebas que desde la prehistoria, los hombres luchaban entre ellos por tierras y recursos naturales. Con el pasar del tiempo, los hombres fueron evolucionando, y así también lo hicieron los objetivos de los conflictos bélicos, los armamentos y estrategias utilizados en estos conflictos.

El objetivo de este proyecto es el desarrollo de un programa que permita la simulación de diferentes batallas que se hayan producido en un pasado distante, en épocas más recientes e incluso simular batallas futuristas o con elementos de fantasía. Además se podrían simular batallas entre diferentes épocas, por ejemplo podríamos enfrentar 300 soldados armados con las más modernas armas contra 1000 soldados armados con espadas y escudos.

2. La Simulación

Como planteamos en el capítulo anterior, se quiere desarrollar un programa que permita la simulación de enfrentamientos bélicos entre dos o más bandos.

Para esto se tienen pensado los siguientes aspectos que van a ser fijos en cada una de las simulaciones:

1. La existencia de un mapa o terreno donde se producirá el enfrentamiento. Este tendrá propiedades que se serán modificables como las dimensiones, el relieve, la hidrografía, etc. La idea es que este se represente por una matriz bidimensional.
2. Las acciones serán por turnos. Como tenemos dos bandos les llamaremos: bando A y bando B. En el turno del bando A cada una de las unidades de A realizará una y solo una acción (ya sea moverse hacia otra posición, atacar o mantener la posición). Luego de esto se pasará al turno del bando B, que al igual que A, podrá hacer una y solo una acción con cada una de sus unidades. Esta forma de implementación permite un comportamiento de acción-reacción entre los dos bandos, asemejándose a lo que ocurre en la vida real.

2.1. El mapa

El mapa consiste en una matriz bidimensional de m filas y n columnas. Cada objeto de la matriz es un objeto `Cell`. Un objeto `Cell` representa cada una de las casillas que conforman el mapa y tiene los siguientes atributos:

- **passable**: Valor entre 0 y 10 que indica cuan accesible es una celda. Las unidades tienden a buscar las celdas que tengan este parámetro lo más alto posible, pues mientras mayor sea este valor, pueden hacer ataques más poderosos.
- **row**: Este parámetro es un número entero que indica la fila en la que se encuentra la celda.
- **col**: Este parámetro es un número entero que indica la columna en que se encuentra la celda.
- **height**: Este parámetro es un número entre 0 y 1 que indica la altura de la casilla. En dependencia de este parámetro la casilla será terrestre o marina.
- **bs_object**: Este parámetro hace referencia al objeto que se encuentra en la casilla. Si en la casilla no se encuentra ningún objeto entonces este parámetro es `None`.

El mapa para una simulación se crea instanciando una clase `LandMap` con los siguientes argumentos:

- Número de filas m
- Número de columnas n
- Un array bidimensional de m filas por n columnas tal que cada posición i, j del array es un número entre 0 y 10 que indica cuan accesible es la celda i, j
- Un array bidimensional de m filas por n columnas tal que cada posición i, j del array es un número entre 0 y 1 que indica la altura de celda i, j
- Un número entre 0 y 1 que indica el nivel del mar. Todas las celdas cuya altura sea menor a este número serán consideradas como celdas marinas y todas las celdas superior a este número serán consideradas terrestres.

Generación aleatoria de mapas La representación abstracta de un mapa es una matriz de alturas, o sea, una matriz de valores en el intervalo $[0, 1]$, donde la noción de nivel del mar es 0.45, o sea, todo valor $x > 0,45$ es una elevación y todo valor $x < 0,45$ es una depresión cubierta por agua. Es importante poder generar un mapa de alturas con un porcentaje de relieve determinado por el usuario que sea lo más realista posible pero esto de forma aleatoria. Por tal razón queda totalmente descartado el enfoque que va por generar una matriz M donde $M[i, j]$ es un número totalmente al azar, pues se obtendrían matrices de ruido similares a la estática en las señales de televisión, además se violaría la restricción del por ciento de elevaciones.

Luego de una consulta bibliográfica, se encontraron algunos procedimientos para la generación de mapas de alturas muy interesantes:

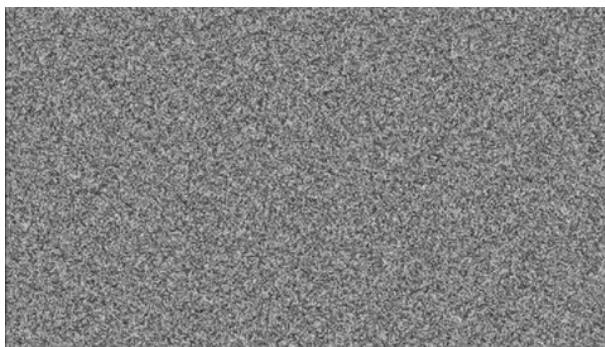


Figura 1. Estática en la señal de TV

- **Algoritmo de Perlin Noise:** debe su nombre a su creador, Ken Perlin, que durante el rodaje de la película Tron creó el algoritmo con el fin de crear texturas procedimentales para los efectos generados por el ordenador. La idea tras el algoritmo es generar una nueva matriz de dimensión menor a la requerida y superponerla a la requerida escalando de forma que encajen perfectamente. A cada esquina de cada celda de la nueva matriz se le asigna un vector gradiente pseudoaleatorio. Además se determinan los valores offset que representan la posición de cada punto de la matriz original dentro de la celda de la nueva matriz. Luego, para cada esquina de la celda que encierra un punto se determina el vector que va desde ella hasta el punto y se hace el producto escalar de ese vector con el vector gradiente. Se realiza una interpolación lineal entre todos los resultados de los productos escalares y este es el valor del punto. El comportamiento de este algoritmo no puede ser modificado pero aún así fue de gran significación en la época y hoy día.
- **Algoritmo Voronoi:** consiste en dividir el espacio en diagramas de Voronoi, que son, simplemente, particiones de un plano en regiones basadas en la distancia a ciertos puntos específicos del plano. Ese conjunto específico de puntos se denomina “semillas, sitios o generadores”. Se suelen especificar de antemano y para cada uno de estos sitios se genera una región del plano que consiste en todos los puntos que están más cerca de este sitio que de ningún otro.
- **Algoritmo de cortes:** tiene un comportamiento poco intuitivo pero interesante, se selecciona un corte en la matriz, se define como corte una línea recta que divida el plano o terreno en dos mitades, luego se decide elevar una de las mitades y bajar o no modificar las otras. Este mismo proceso se repite miles de veces y se obtienen resultados aceptables. El algoritmo es ineficiente, por la cantidad de iteraciones necesarias para obtener buenos resultados.
- **Algoritmos de erosión térmica e hídrica:** algoritmos que simulan los procesos naturales de erosión térmica e hídrica. En el caso de la térmica se fija un valor T que representa el valor a partir del cual se va a erosionar, y se recorre la matriz buscando los puntos que cumplen que la diferencia de altura con sus vecinos es superior a T . En dichos puntos el algoritmo resta valor de altura a sus vecinos cuya distancia sea mayor que T de forma que todas las distancias queden menor que T . En el caso de la erosión hídrica se define una matriz de agua y una matriz de sedimentos la cual representa un porcentaje de material que puede ser movido por la erosión. Y se realiza un procedimiento parecido a la erosión térmica.

Ahora estos algoritmos generan mapas de alturas con apariencia realista, pero no son suficientes para cumplir las restricciones de relieve. Por tanto, nuestra propuesta es la implementación de un algoritmo evolutivo que genera un mapa de alturas realista y satisfaga la restricción anteriormente mencionada.

Los algoritmos evolutivos son estrategias de optimización y búsqueda de soluciones que toman como inspiración la evolución en distintos sistemas biológicos. La idea fundamental de estos algoritmos es mantener un conjunto de individuos que representan una posible solución del problema. Estos individuos se mezclan y compiten entre sí, siguiendo el principio de selección natural por el cual sólo

los mejor adaptados sobreviven al paso del tiempo. Esto redundará en una evolución hacia soluciones cada vez más aptas.

El algoritmo evolutivo en cuestión inicia con una población de individuos generados utilizando el algoritmo de Perlin Noise implementado en la biblioteca de Python bajo el nombre `perlin-noise`. Luego se realizan un cantidad de iteraciones seleccionando los individuos más adaptados, mezclándolos y mutándolos. Se considera individuo mejor adaptado aquel cuyo valor `fit` sea mayor. Este valor se calcula como sigue:

```
def fit_func(self, heightmap: HeightMap):
    count = sum(sum(heightmap.__map__ > self.__sea__))
    per = count / prod(self.__shape__)
    return per if per < self.percentage else 0
```

2.2. Los objetos

Un objeto es todo lo que se puede poner en el mapa y cada objeto ocupa una y solo una casilla del mapa. Estos tienen propiedades como el id que es único para cada objeto, los puntos de vida que determinan el estado de un objeto y la defensa un parámetro que indica cuan resistente es un objeto a los daños que puede sufrir durante la simulación. Todos estos parámetros son números de 1 a 10. Cuando la vida de un objeto llegue a 0, este se destruye desapareciendo del mapa. Los objetos se clasifican en dos tipos: unidades y objetos estáticos. Los objetos además tienen definidas dos funciones `put_in_cell` cuya función es colocar al objeto en alguna posición de un mapa. Si se intenta poner un objeto en una posición diferente a su tipo este automáticamente se destruye. La otra función `take_damage`, a partir de un ataque sufrido indica como se reducen los puntos de vida del objeto.

Objetos estáticos Los objetos estáticos los podemos definir como objetos propios del ambiente. Estos no pertenecen a ningún bando y no pueden realizar acciones pero sí pueden ser afectados por las acciones que realicen las unidades. Estos solo pueden ser puestos en celdas terrestres. Ejemplos de estos objetos pueden ser árboles, rocas, muros, etc. Todos estos objetos son terrestres.

Unidades Las unidades son los agentes de la simulación. El objetivo de cada una de las unidades es destruir a las unidades enemigas (las que no pertenecen a su mismo bando), y para ello podrá analizar parte del ambiente en el que se encuentra y tomar la decisión que sea más conveniente según sean las circunstancias.

Dado que el ambiente estará cambiando constantemente, las unidades serán agentes casi puramente reactivos. La arquitectura empleada para definir el comportamiento de las unidades es la Arquitectura de Brooks (de categorización o inclusión) que recordemos tiene las siguientes características:

- La toma de decisión se realiza a través de un conjunto de comportamientos para lograr objetivos (reglas de la forma situación → acción).
- Las reglas pueden dispararse de manera simultánea por lo que debe un mecanismo para escoger entre ellas.

Dado esto se definió un sistema experto que actuará como la función del agente. Este será descrito posteriormente.

Propiedades de las unidades

Las unidades además de las descritas anteriormente que tienen todos los objetos cuentan con las siguientes propiedades:

- **side**: Una instancia de la clase `Side` que indica el bando al que pertenece la unidad
- **attack**: Valor entre 1 y 10 que marca la capacidad de causar daños a sus oponentes.

- **moral**: Valor entre 1 y 10 que marca la moral con la unidad encara la batalla. Cuanto mayor es ese valor más efectivos son sus ataques y sus defensas.
- **offensive**: Valor entre 1 y 10 que indica cuan ofensiva es una unidad. Un valor alto la hace más ofensiva y un valor bajo la hace más defensiva.
- **min_range**: Valor entero entre 0 y 10 que indica el rango mínimo al que se debe encontrar un enemigo para que la unidad pueda atacarlo.
- **max_range**: Valor entero entre 0 y 10 que indica el rango máximo al que se debe encontrar un enemigo para que la unidad pueda atacarlo.
- **radio**: Valor entero entre 1 y 9 que indica el número de casillas que son afectadas por un ataque de la unidad. Si es 1 se afecta solo a la casilla seleccionada para el ataque. Si es 9 se afectan la casilla seleccionada y las 8 adyacentes a esta. Si es $1 < \text{radio} < 9$, entonces se toman como casillas afectadas la seleccionada y $\text{radio} - 1$ casillas adyacentes a esta.
- **vision**: Valor entre 1 y 10 que indica la cantidad de celdas en una determinada dirección, que la unidad puede “ver” (saber que objetos están en dicha celda).
- **intelligence**: Valor entre 0 y 10 que indica la inteligencia de la unidad. Mientras más inteligente sea una unidad con mayor precisión puede calcular los atributos de sus enemigos.
- **recharge_turns**: Turnos que demora la unidad en recargar después de hacer un ataque. Mientras esté recargando la unidad no podrá atacar pero si puede moverse.
- **solidarity**: Valor booleano que indica si la unidad es solidaria o no.
- **movil**: Valor booleano que indica si la unidad puede desplazarse por el mapa.

Sistema experto A continuación se explicará el sistema experto implementado que actúa como función del agente la cual describe el comportamiento del agente durante un turno.

Lo primero que la función hace es buscar si se existe un enemigo al que se pueda atacar. Para eso primero se comprueba si la unidad no está recargando, si lo está se reduce en uno los turnos que debe esperar para atacar, si no lo está se busca el mejor enemigo para atacar.

El mejor enemigo para atacar se determina de la siguiente forma:

Por cada casilla atacable (casilla que se encuentra a una distancia de la unidad entre su rango mínimo y su rango máximo), se chequea si en la casilla hay un enemigo. Si el radio de ataque es mayor que 1 y hay una unidad amiga cerca del enemigo que pudiera verse afectada por el ataque, se ignora este enemigo. Esto se definió así para evitar el daño ocasionado por fuego amigo.

Entonces si el enemigo no es ignorado debido a lo anterior, se calcula el costo de atacar al enemigo. Para dicho cálculo la unidad estima la vida y la defensa del enemigo y con estas estimaciones calcula cuantos turnos podría tomarle destruir a dicho enemigo. Mientras mayor sea la inteligencia de la unidad más precisas serán las estimaciones.

Luego de haber analizado todos los enemigos, el seleccionado por la unidad para atacarlo es aquel cuyo costo es menor. Si no se detecta ningún posible enemigo a atacar entonces la unidad no realiza un ataque en el turno.

Si la unidad no realiza un ataque, entonces, en caso de que se móvil, chequea si se puede mover a alguna casilla adyacente a la que se encuentra. Esto se hace de la siguiente forma:

Se fija un costo en infinito. Luego por cada una de las celdas que la rodean en todos los puntos cardinales posibles (NW, N, NE, W, E, SW, S y SE) se calcula el costo de moverse a dicha celda, y nos quedamos con la celda cuyo costo es el menor. Luego si el menor costo detectado es menor a infinito la unidad se mueve a dicha celda en caso contrario la unidad mantiene su posición.

Ahora veamos como calcular el costo de que una unidad se mueva de una celda a otra.

Si la celda nueva es intransitable (tiene el parámetro **passable** en 0), el tipo de la celda es diferente al tipo de la unidad o si en la celda hay algún objeto se devuelve infinito. Estos son todos los casos en los que la unidad no puede moverse a dicha celda. Para las unidades terrestres si la diferencia de alturas entre dos celdas es muy grande (mayor a 0.3), tampoco pueden avanzar a dicha celda, retornándose infinito.

Entonces en un primer momento se fija el costo en $10 - \text{passable}/2$, de esta manera se premia ir a celdas más transitables. A continuación se comprueba si esa celda está entre las que la unidad recuerda

como ya visitadas. Si así es, el coste se incrementará en `passable/3`. Así logramos incentivar que las unidades visiten celdas no visitadas con anterioridad

Ahora se comprueba si la celda se encuentra en zona “amiga”, es decir, si esa celda es adyacente a alguna celda en la que se encuentre algún compañero de su bando. Si así es y nuestra unidad protagonista es solidaria, el coste se reducirá a la mitad. Si es una zona amiga pero la unidad no es solidaria, el coste solo se reducirá dividiéndose por la raíz cuadrada de 2. De esta manera se incentiva que las unidades tiendan a permanecer en grupo y más cuanto más solidarias son.

Si la celda que se está estudiando es una celda en la que nuestra unidad tendrá al alcance un enemigo, su coste se reducirá en el valor del parámetro `offensive` multiplicado por la inversa de la raíz cuadrada de la distancia mínima hasta el enemigo. Además la unidad observa las celdas cercanas a la celda que se está estudiando que estén dentro de su rango de visión. Si detecta que al moverse a dicha celda se encontrará en rango de algún enemigo el costo se aumenta en 1.1 por cada enemigo que pudiera atacar a la unidad. De esta manera se está diciendo que, cuanto más ofensiva sea nuestra unidad, más se aproximará al enemigo, aunque con cierta precaución.

A continuación se hace una breve explicación de las funciones implementadas a las unidades. Además de las ya mencionadas, implementadas en “BSObject”, un agente “BSUnit” tiene implementadas las siguientes funciones:

- `calculate_distance(self, cell1, cell2) -> int` con la que la unidad calcula la distancia entre dos celdas. La distancia se calcula como la cantidad mínima de celdas que tiene recorrer la unidad para ir de una celda a otra.
- `nearby_friend(self, cell) -> bool` con la que el agente determina si moviéndose a la celda de entrada estará cerca de un compañero.
- `enemy_in_range(self, cell)` con la que la unidad determina si moviéndose a la celda de entrada estará cerca de un compañero.
- `in_range_of_enemy(self, cell) -> int` con la que la unidad detecta a los enemigos de que pueden atacarla si se mueve a dicha celda.
- `move_cost_calculate(self, cell, type) -> float` con la que la unidad calcula el costo de moverse a la celda.
- `enemy_cost_calculate(self, enemy) -> float` con la que la unidad calcula el costo de atacar a un determinado enemigo.
- `enemy_to_attack(self)` con la que la unidad selecciona, de ser posible, un enemigo para atacarlo.
- `take_damage(self, damage)` que es una redefinición del método de “BSObject” para tener en cuenta la moral de la unidad.
- `attack_enemy(self, enemy)` la cual es la acción de atacar a un enemigo.
- `move_to_cell(self, cell)` la cual es la acción de moverse a una celda determinada.
- `def turn(self, type)` con la que la unidad determina que hacer durante un turno, el parámetro `type` hace referencia al tipo de celda que la unidad puede visitar.

Los métodos `put_in_cell` y `turn` se redefinen en las clases “LandUnit” y “NavalUnit” haciendo un llamado a las funciones respectivas de “BSUnit” pasando “earth” y “water” al parámetro `type` respectivamente.

3. Battle Script, el lenguaje de dominio específico para ejecutar el proyecto

El usuario final del proyecto necesita un medio para especificar las condiciones del enfrentamiento a simular, entiéndase cómo son las unidades, los soldados, el mapa, el movimiento de las unidades, etcétera. Para ello se implementó un lenguaje de dominio específico (DSL, por sus siglas en inglés *Domain Specific Language*) con el nombre de Battle Script, que permite la creación de nuevas unidades, la creación de obstáculos, la creación de mapas, la inserción de unidades y obstáculos en el mapa, la creación de bandos, y la ejecución de la simulación.

3.1. Gramática

Se diseñó la siguiente gramática para el lenguaje:

```

bs_file ->  classes statements EOF
          |  EOF

classes -> class_def  classes
          |  class_def

statements ->  statement statements
              |  statement

statement ->  func_def
              |  if_def
              |  while_def
              |  decl ';'
              |  assign ';'
              |  return_stat ';'
              |  'break' ';'
              |  'continue' ';'
              |  expressions ;

func_def ->  'function' return_type NAME '(' params ')'' ->' block
          |  'function' return_type NAME '(' ')'' ->' block

if_def ->  'if' expression '->' block elif_def
          |  'if' expression '->' block else_def
          |  'if' expression '->' block

elif_def ->  'elif' expression '->' block elif_def
            |  'elif' expression '->' block else_def
            |  'elif' expression '->' block

else_def ->  'else' '->' block

class_def ->  'class' NAME 'is' NAME '->' '{' constructor functions '}'
            |  'class' NAME 'is' NAME '->' '{' constructor '}'

functions -> func_def functions

```

```

    | func_def

constructor -> 'constructor' '(' params ')' '->' '{' attributes '}'
    | 'constructor' '(' ')' '->' '{' attributes '}'
    | 'constructor' '(' ')' '->' '{' '}'

attributes -> attr_def attributes
    | attr_def

attr_def -> type 'this' '.' NAME '=' expression ';'

while_def -> 'while' expression '->' block

return_type -> 'void'
    | type

type -> 'number'
    | 'bool'
    | NAME

assign -> NAME '=' expression

decl -> type NAME '=' expression

return_stmt -> 'return' expression
    | 'return'

block -> '{' statements '}'

params -> type NAME ',' params
    | type NAME

expressions -> expression ',' expressions
    | expression

expression -> disjunction 'if' disjunction 'else' expression
    | disjunction

disjunction -> conjunction 'or' disjunction
    | conjunction

conjunction -> inversion 'and' conjunction
    | inversion

inversion -> 'not' inversion
    | comparision

comparision -> sum compare_par

```

```

        |    sum

compare_par ->  'eq' sum
               |  'neq' sum
               |  'lte' sum
               |  'lt' sum
               |  'gte' sum
               |  'gt' sum

sum ->  sum '+' term
       |  sum '-' term
       |  term

term -> term '*' factor
       |  term '/' factor
       |  term '%' factor
       |  factor

factor ->  '+' factor
          |  '-' factor
          |  pow

pow ->  primary '^' factor
       |  primary

primary -> primary '.' NAME
          |  primary '(' args ') '
          |  primary '(' ' ') '
          |  atom

args -> expression ',' args
       | expression

atom -> NAME
       |  'True'
       |  'False'
       |  'None'
       |  NUMBER
       |  list

list -> '[' expressions ']'
       |  '[' ']'

```

3.2. Compilador

AQUI VA UNA MUELA BISCA DE C6MO ES EL COMPILADOR A GRANDES RASGOS

3.3. Tokenizador

Un Tokenizador, comúnmente llamado Lexer, es un ente encargado de dividir la cadena de texto de entrada del compilador en tokens del alfabeto del lenguaje Battle Script, identificando el tipo del token y enviándolo a la siguiente etapa del proceso de compilación.

Para la implementación del tokenizador fue necesario un sistema de expresiones regulares, una clase para representar un token, así como su tipo y la definición de los tokens del lenguaje, además de la clase propia del tokenizador.

Sistema de expresiones regulares Una expresión regular es una definición recursiva de un lenguaje donde a es la expresión regular para $L(a) = \{a\}$ y ϵ es la expresión regular para $L(\epsilon) = \{\epsilon\}$. Si s y r son expresiones regulares entonces:

- $(s)|(r)$ es la expresión regular para la unión de lenguajes $L(s) \cup L(r)$
- $(s)(r)$ es la expresión regular para la concatenación de lenguajes $L(s)L(r)$
- $(s)^*$ es la expresión regular para la clausura del lenguaje $L(s)^* = \bigcup_{k=0}^{\infty} L(s)^k$

Se implementó un sistema de expresiones regulares que admite los siguientes operadores:

- `|` que hace la unión de dos expresiones regulares.
- La concatenación de expresiones regulares de la siguiente forma, si a y b son expresiones regulares entonces ab es la expresión de la concatenación.
- `*` que hace la clausura del lenguaje que representa la expresión regular.
- `?` que busca la coincidencia de la expresión regular una vez o ninguna.
- `+` que busca la coincidencia de la expresión regular una o más veces.
- `.` que busca la coincidencia de ninguno o cualquier caracter.
- `\` que permite la inclusión de los operadores anteriores en un expresión regular como un caracter.

La gramática para el lenguaje de las expresiones regulares que se implementó es la siguiente:

```

regex = exp

exp      = term '|' exp
          | term

term     = factor term
          | factor

factor   = primary '*'
          | primary '+'
          | primary '?'
          | primary

primary  = '(' exp ')'
          | '\\' CHAR
          | CHAR
          | '.'

```

Una expresión regular en el sistema se implementó utilizando la clase `Regex` de Python. Una expresión regular se construye con la cadena de texto que representa el patrón de la expresión regular. Este patrón se compila y se devuelve un Autómata Finito No Determinista (NFA) que se utiliza para los procesos de saber si una cadena pertenece al lenguaje representado por la expresión, o encontrar todas las coincidencias de la expresión en una cadena de texto.

```

class Regex:
    def __init__(self, pattern: str):
        self.pattern : str = pattern

```

```

self.nfa : NFA = self.compile()

def compile(self) -> NFA:
    lex: Lexer = Lexer(self.pattern)
    parser : Parser = Parser(lex)
    tokens : List[Token] = parser()
    handler: Handler = Handler()

    nfa_stack : List[NFA]= []

    for t in tokens:
        handler.handlers[t.name](t, nfa_stack)

    if len(nfa_stack) == 1:
        return nfa_stack.pop()
    raise Exception("Bad regex!")

def match(self, string: str) -> bool:
    return self.nfa.match(string)

def find_all(self, string: str) -> List[Match]:
    return self.nfa.find_all(string)

```

El proceso de compilación de una expresión regular es bastante sencillo, se tiene un tokenizador que recorre la cadena y cada vez que encuentra un operador o un carácter devuelve el token correspondiente, si encuentra el operador de escape (\) devuelve un token de tipo **CHAR** con el carácter siguiente. El proceso de Parsing consiste en recibir el conjunto de tokens provenientes del tokenizador y llamar al método `exp` que este parsea el no terminal del mismo nombre de la gramática y va parseando el resto de no terminales según corresponda, o sea, se implementó un Parsing Recursivo Descendente.

Luego del proceso de parsing y verificar que la secuencia de tokens es correcta, se recorre la secuencia de tokens y haciendo uso de la clase **Handler** se construye el autómata de la expresión regular.

La clase **Handler** es la encargada de dado un conjunto de tokens construir el autómata correspondiente, para ello implementa un método para cada operador de la gramática, estos métodos saben construir el autómata que resulta de aplicar la operación a uno, dos o más autómatas según el operador. La clase implementa el algoritmo Thompson's construction para la construcción del

- **Reconcimiento de un carácter:** el método de la clase **Handler** encargado de construir dicho autómata construye el estado inicial `s0` y el final `s1` y añade la transición con el carácter en cuestión, como se muestra en la figura siguiente:



Figura 2. Autómata finito no determinista para el reconocimiento de un carácter

- **Operación de Unión:** el método encargado de construir el autómata solo debe los estados iniciales y finales, además añadir ϵ -transiciones desde el estado inicial hacia los autómatas y desde los autómatas hacia el estado final.

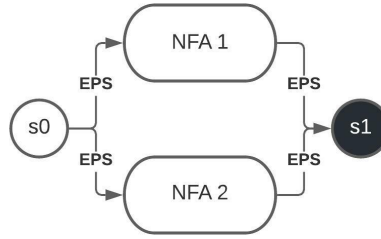


Figura 3. Autómata finito no determinista para el operador unión

- **Operación de Concatenación:** el método encargado de construir el autómata solo debe añadir una ϵ -transición desde el estado final del primer autómata hacia el estado inicial del segundo autómata.



Figura 4. Autómata finito no determinista para el operador concatenación

- **Operación *:** el método encargado de construir el autómata solo debe los estados iniciales y finales, además añadir ϵ -transiciones desde el estado inicial hacia el autómata y desde el autómata hacia el estado final, así como una desde s_0 hacia s_1 para reconocer la no aparición del patrón, y una transición ϵ desde el estado final del autómata hacia el inicial para reconocer las repeticiones.

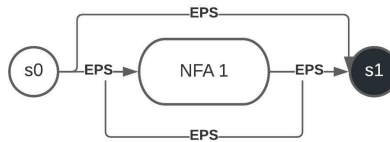


Figura 5. Autómata finito no determinista para el operador *

- **Operación +:** el método encargado de construir el autómata solo debe los estados iniciales y finales, además añadir ϵ -transiciones desde el estado inicial hacia el autómata y desde el autómata hacia el estado final, y una transición ϵ desde el estado final del autómata hacia el inicial para reconocer las repeticiones.
- **Operación .:** el método encargado de construir dicho autómata construye los estados iniciales y finales ya añade un transición para cada caracter y añade un ϵ -transición para reconocer la ocurrencia de ningún caracter. El autómata quedaría como el siguiente:
- **Operación ?:** el método encargado de construir dicho autómata debe añadir una ϵ -transición desde el estado inicial del autómata hacia el estado final del autómata para reconocer la no aparición del patrón.



Figura 6. Autómata finito no determinista para el operador +

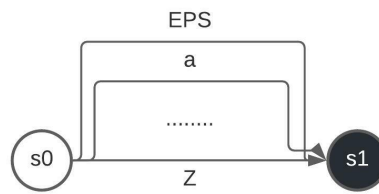


Figura 7. Autómata finito no determinista para el operador .



Figura 8. Autómata finito no determinista para el operador ?

Luego que se construye el autómata para una expresión regular el proceso de compilación ha terminado.

Entonces, ¿cómo saber si una cadena de texto pertenece al alfabeto representado por una expresión regular? Para ello existen varios enfoques entre ellos están:

- **Enfoque inocente o tonto:** utilizando backtracking y revisando los operadores y la cadena. Este enfoque es bien costo y no aprovecha las potencialidades de los autómatas anteriormente contruidos.
- **Enfoque DFA:** se construye un Autómata Finito Determinista (DFA) a partir del no determinista anteriormente construido y se hace una pasada sobre él. Este enfoque es correcto, pero implica un procesamiento extra para construir el DFA.
- **Enfoque NFA:** este es el enfoque implementado en el proyecto en el método `match` de una `Regex`, que consiste en hacer una pasada sobre el autómata no determinista y revisar si coincide la cadena, pero el no determinismo trae un problema consigo, es que en un estado `x` no se sabe con total seguridad que transición aplicar. Para ello, la propuesta es ejecutar las transiciones a la vez y ver si por alguna se llega al estado final. Esto hace que el tiempo de reconocimiento sea lineal respecto al tamaño de la cadena de entrada.

Entonces, ¿cómo encontrar todas las coincidencias de un patrón dentro de una cadena de texto?. Para ello se modificó el método anterior y cada vez que no se podía continuar porque venía un caracter desconocido para la expresión regular se volvía al estado inicial. Además en cada posición se revisa si se ha llegado al estado final, se devuelve la coincidencia y se vuelve al estado inicial. Este procedimiento se implementó en método `find_all` de las `Regex`. Este procedimiento es lineal con respecto al tamaño de la cadena.

Luego de implementado el sistema de expresiones regulares se implementó la clase `Token` y el enum `TokenType`.

Token y TokenType Un token en el proyecto se representa con la clase `Token` que tiene la siguiente implementación. La propiedad `regex` almacena la expresión regular que coincide con el token, la propiedad `name` representa el nombre del tipo de token, la propiedad `lexeme` almacena la cadena de texto que se extrajo de la cadena de entrada como token, es decir, si se tiene un token `t` entonces `t.regex.match(t.lexeme)` es verdadero. Además, un token almacena donde comienza y termina `lexeme` en la cadena de entrada.

```
class Token:
    @propertyclass Token:
    @property
    def regex(self) -> Regex:
        return self.type.value[0]

    @property
    def name(self) -> str:
        return self.type.value[1]

    def __init__(self, token_type: TokenType, lexeme: str, start: int, end: int):
        self.type: TokenType = token_type
        self.lexeme: str = lexeme
        self.start: int = start
        self.end: int = end

    @property
```



```

def name(self) -> str:
    return self.type.value[1]

def __init__(self, token_type: TokenType, lexeme: str, start: int, end: int):
    self.type: TokenType = token_type
    self.lexeme: str = lexeme
    self.start: int = start
    self.end: int = end

```

El tipo de los tokens se definió utilizando el enum `TokenType` que representa la expresión regular que lo representa y el nombre del token, por ejemplo: para el token que representa la palabra reservada `eq` el `TokenType` correspondiente es `Eq = (Regex("eq"), "eq")`

Algoritmo del tokenizador El algoritmo del tokenizador sigue la siguiente idea general, se le asocia a cada token de la gramática un nivel de precedencia que representa cuán relevante es un token sobre otro. Por ejemplo si se tienen los siguientes: `Token((Regex("eq"), "eq"), "eq", 1, 2)` y `Token((Regex("(a|b|...|Z)+"), "NAME"), 'eq', 1, 2)` donde el primer token tiene precedencia 1 y el segundo tiene 2, entonces el token más relevante es el primero. Entonces, se tiene una lista `tokens` y se recorre la lista de tokens de la gramática y a cada uno se le piden todas coincidencias en la cadena de entrada. Luego se recorren las posiciones `i` de la cadena de entrada, y se añade a `tokens` aquel token que comience en `i` y tenga menor precedencia.

```

class Tokenizer:
    def __call__(self, bs_content_file: str) -> Iterable[Token]:
        tokens: List[Token] = []

        matches = {}
        for token_def in TOKENS:
            matches[token_def] = token_def.type.value[0].find_all(bs_content_file)

        i = 0
        while i < len(bs_content_file):
            token_in_i: List[Tuple[TokenDefinition, Match]] = []

            # Get all matches
            for k, v in matches.items():
                for t in v:
                    if t.start == i:
                        token_in_i.append((k, t))

            # Get match with highest precedence
            if len(token_in_i):
                token_in_i = sorted(token_in_i, key=lambda tup: tup[0].precedence)
                token = Token(token_in_i[0][0], \
                               token_in_i[0][1].value, \
                               token_in_i[0][1].start, \
                               token_in_i[0][1].end)
                tokens.append(token)
                i = token.end

            i += 1

```

```
return deque(tokens)
```

Este algoritmo tiene complejidad temporal $O(n * m)$ donde m es la cantidad de tokens, y n el tamaño de la cadena de entrada.

4. Reflejo de las asignaturas en el proyecto

4.1. Simulación

Tendríamos como sistema el enfrentamiento bélico. Las entidades serían las unidades, estructuras y elementos del terreno. Como relaciones tendríamos por ejemplo la distancia entre estas entidades, el daño que le causa una unidad a otra, etc. Como proceso tendríamos el movimiento de las unidades, el ataque de una unidad a otra, etc.

Este sistema es observable, permitiéndonos, al ejecutar simulaciones del mismo, obtener resultados y sacar conclusiones a partir de estos. Es controlable pues las unidades realizan acciones según estrategias y la simulación ocurre según reglas definidas. Es modificable pues podemos agregar y eliminar unidades, además de cambiar reglas y estrategias, lo que nos permite obtener diferentes resultados.

4.2. Compilación

Se definirá un lenguaje en el cual se puedan definir diferentes unidades, estructuras y sus respectivas estadísticas, modificar las características del terreno, crear estrategias y definir reglas para la simulación.

4.3. Inteligencia Artificial

Como las unidades se tendrán que mover por el mapa, para lograr un movimiento eficiente de las mismas utilizaremos el algoritmo A*.

Además como tenemos dos bandos enfrentándose nos auxiliaremos de un algoritmo Minimax para realizar los movimientos que harán los bandos en sus respectivos turnos, utilizando una heurística basada en la situación actual del mapa y la estrategia definida por el usuario.

Es muy posible que con el avance de la asignatura y el desarrollo del proyecto utilicemos más herramientas.

5. Aplicaciones

Este proyecto nos permite recrear batallas y estudiar diferentes finales alternativos según se hubieran comportado diferentes parámetros. Además podemos predecir el desenlace de futuros enfrentamientos y cuales podrían ser las mejores estrategias para que uno u otro bando saliese victorioso, además del costo que podría suponer dicho conflicto para ambos bandos.