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

2021

Índice general

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.

2.2. Los objetos

Un objeto es todo lo que se puede poner el mapa y cada objeto ocupa una y solo una casilla del mapa. Estos tienen propiedades como el id que s 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 si 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.
- `ofensive`: 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 $1 <$ `radio` < 9, entonces se toman como casillas afectadas la seleccionada y `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:

```
1 def turn(self, type_unit):
2
```

```
3       enemy=None
4       if self.turns_recharging == 0:
5           enemy = self.enemy_to_attack()
6       else:
7           self.turns_recharging -= 1
8
9       if enemy != None:
10          self.attack_enemy(enemy)
11          self.turns_recharging = self.recharge_turns
12      else:
13          cost = 10000
14          cell=self.cell
15          for i in range(self.cell.row-1, self.cell.row+2):
16              if i >= self.map.no_rows:
17                  break
18              if i < 0:
19                  continue
20              for j in range(cell.col-1, cell.col+2):
21                  if j >= self.map.no_columns:
22                      break
23                  if j < 0 or (i == self.cell.row and j == self.cell.col):
24                      continue
25                  new_cost = self.move_cost_calculate(self.map[i][j], type_unit)
26                  if new_cost < cost:
27                      cost = new_cost
28                      cell = self.map[i][j]
29          if cost < 10000:
30              self.move_to_cell(cell)
```

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 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 celtas es muy grande (mayor a 0.3), tampoco pueden avanzar a dicha celda, retornándose infinito.

-

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

El usuario final del proyecto necesita un medio para específicar 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 ejecucción 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 CóMO 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 envíandolo 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 regulares 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 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 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 caracter devuelve el token correspondiente, si encuentra el operador de escape (\) devuelve un token de tipo `CHAR` con el caracter 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 dos autómatas.

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.