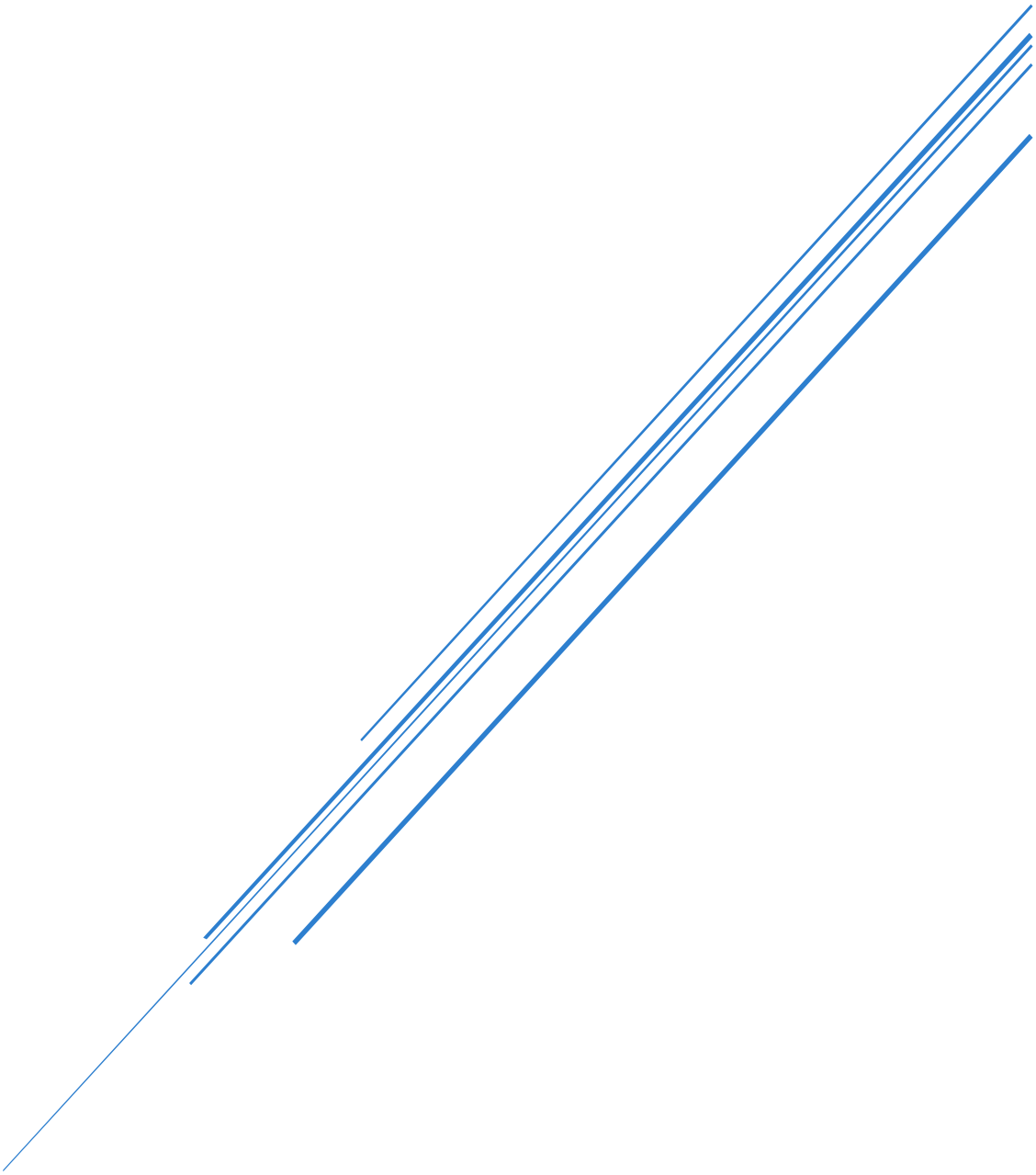
PROYECTO CNB

Implementación y simulación de redes celulares generadoras de expresiones regulares



Santiago Millán Giner
Universidad Politécnica de Valencia - MIARFID

Contenido

Introducción 2
Simulador 2
Operaciones 2
Mutación 2
Cruce 2
Filtros 3
Célula 3
Red celular 4
Compilador 4
Parser 4
Diseñador de la red 4
Instrucciones de uso 5

Introducción

El objetivo de este proyecto es implementar un software que permita diseñar y ver el comportamiento redes de procesadores genéticos que procesan expresiones regulares. Para ello, se ha desarrollado un compilador, que a partir de una gramática regular en la forma normal de Chomsky, generé un objeto de Python simulador de red celular, así el usuario puede crear nuevas redes, sin necesidad de simularlas o utilizar las redes sin necesidad de compilarlas.

La arquitectura de este proyecto se basa en separar estás dos funcionalidades y aplicar una estrategia bottom-up, donde primero se desarrollan las operaciones más básicas del sistema, para posteriormente construir funcionalidades más complejas. El motivo principal por el cual se ha optado por esta filosofía de diseño, es que se quiere primero las partes pequeñas, para asegurar su correcto funcionamiento y conocer sus limitaciones de coste computacional antes de implementar funcionalidades más grandes. Por ello, se ha implementado en cada archivo una o varias funciones de test, que se ejecutan llamando el propio archivo de código fuente de Python como principal.

Simulador

Operaciones

Los procesadores celulares solo pueden tener operaciones de mutación o de cruce. Para implementar dichas operaciones definieron las clases `Mutation` y `Crossover`, que heredan de una clase abstracta `CellOperation`.

Dicha clase, programada en el archivo `cell_operation.py` contiene:

- `poplulation`: Atributo que contiene una lista con todos los individuos a los que se le puede aplicar la operación, es decir, la población de la célula que posee la operación. Cada individuo esta formado por una lista de símbolos, y su vez cada símbolo es una lista strings.
- `apply`: Método sin argumentos que cuando se invoca realiza una operación con uno de los individuos de la población aleatoriamente. Devuelve un string que representa la operación realizada tras la llamada.
- `__repr__`: Método especial para representar el objeto mediante texto

Mutación

Esta clase programada en el archivo `mutation.py`, incorpora un atributo `rules`, que es un diccionario almacena las reglas de producción, teniendo como clave el símbolo de entrada y su valor todos los posibles símbolos de salida asociados.

Cuando se llama al método `apply`, se escoge un individuo aleatorio de la población y se recorre secuencialmente cada símbolo, guardando todas las mutaciones posibles en una lista de tuplas. Posteriormente, se elige un reemplazo de la lista. Gracias a este algoritmo, se asegura que siempre que haya un reemplazo se va a realizar, la posición del reemplazo realizado es equiprobable y el coste es lineal respecto a la longitud del individuo. Como inconveniente, no siempre que se llame a esta función va a escoger una cadena que pueda mutar, en ese caso, no realizará ninguna operación.

Para representar el objeto de la clase `Mutation`, simplemente se muestra el nombre de la operación “MUTACIÓN” junto al conjunto de reglas en un formato más agradable para el usuario

Cruce

La clase ha sido programa en el archivo `crossover.py`. Su método apply elige dos cadenas de la población (las dos cadenas pueden ser el mismo individuo). Posteriormente se extraen todos los prefijos y sufijos de cada cadena (incluyendo el símbolo vacío del inicio). Finalmente crean nuevos individuos, con cada combinación de prefijo de una cadena y sufijo de otra.

Esta operación tiene un coste computacional elevado, pero proporcional al número de hijos generados.

Para representa el objeto de la clase Crossover simplemente se muestra el mensaje “CROSSOVER”, ya que no tiene ningún atributo adicional.

Filtros

Cada célula tiene un filtro de entrada y de salida, donde se especifica el conjunto de cadenas que pueden entrar o salir de la célula. Inicialmente, se pensó en usar varios `set` con los contextos permitidos y no permitidos, aunque era una idea simple que funcionaba bien para filtrar por símbolos individuales, se complicaba a la hora de tener en cuenta contextos que involucran más de un símbolo.

Como consecuencia, al final se optó por programar objetos-función en la clase `filter.py`, que siguen una metodología parecida la clase `CellOperation` mencionada anteriormente. Cada filtro, de entrada o de salida, esta representado por un objeto `Filter` consta de dos atributos, su representación y una función que es llamada cuando se invoca su método especial `__call__` (este se invoca cuando llamas al objeto como una función).

Después se crearon funciones para poder crear estos filtros de forma más simple y cómoda en otras partes del programa o por el usuario, como las siguientes:

- `permited(allowed_set)`: Devuelve un filtro que realiza una búsqueda secuencial con parada de algún símbolo incluido en allowed_set.
- `excluded(forbidden_set)`: Devuelve el mismo filtro que el anterior, pero esta vez si se encuentra devuelve false.
- `logical_and(filter0,filter1)`: Devuelve un filtro producto de hacer la operación and con ambos filtros
- `logical_or(filter0,filter1)`: Lo mismo pero con el operador or
- `empty()`: Permite cualquier cadena, por tanto siempre devuelve true.

Además, para incrementar la facilidad de uso de las operaciones lógicas se ha hecho que los métodos especiales `__and__` y `__or__` devuelvan un `Filter` como el anteriormente comentado.

Célula

Cada procesador está conformado por un objeto de tipo `Cell`, dicha clase está programada en `cell.py`. Cada instancia contiene los siguientes atributos:

- `name:str:` Nombre de la célula por la cual se identifica
- `operation:CellOperation:` Operación de célula Mutación o Crossover
- `inFilter:Filter:` Filtro de entrada, combina permitados y prohidibos, por defecto toma el filtro `empty`, es decir, permite cualquier símbolo
- `outFilter:Filter:` Filtros de salida, lo mismo, los mismo que los anteriores.
- `population:list[list[str]:` Lista de individuos de la población que contiene la célula (variable compartida con self.operation)
- `connections: list[Cell]`: Lista de conexiones de salida a otras células.

Esta clase, tiene un método `simulation_tick`, que se puede llamar una vez cada paso de simulación. Cuando se realiza la llamada actualiza el estado aleatoriamente, realizando la operación que contiene la célula o migrando el individuo a otra célula.

Dicho proceso de migración se realiza llamando al método `exit`, donde se escoge a elige un individuo entre todos los que cumplen con el filtro de salida de la célula origen y deja de ser parte de `self.population`. Posteriormente se elige una de las células vecinas como destino, después se llama al método `enter` de la esta, para que válide si pasa por el filtro de entrada, en caso afirmativo se añade a la población destino, en caso contrario el individuo se pierde y será eliminado por el recolector de basura, ya que no hay ninguna referencia apuntando a dicha variable.

Todas las funciones utilizadas para dar un paso de simulación devuelven una traza que muestra los cambios realizados en el individuo, si este se a movido, si ha sido borrado o se ha cruzado. Incluso también se contempla que no se haya realizado ninguna operación, en caso de que no haya población para realizar la acción escogida.

La célula al igual que las clases descritas anteriormente también tiene una función de representación. En este caso representa todos sus atributos, utilizando las funciones de representación de cada atributo.

Red celular

El simulador en su totalidad se implementa la clase `CellNet` en el fichero `cell_net.py`. Esta simplemente se conforma una lista de células como las descritas anteriormente. Para ejecutar el simulador únicamente hay que llamar al método `run`, que consta de un bucle en el que en cada iteración selecciona una célula aleatoria y ejecuta un paso de simulación.

La condición de parada del bucle puede ser el tiempo en segundos o el número de pasos de simulación, según se especifique en los parámetros de entrada `ticks` o `seconds`. El usuario seleccionar ambas opciones a la vez, entonces el bucle de parará cuando haya se haya cumplido una de las condiciones de parada. En caso de especificar ninguna, se lanza una excepción para evitar bucles infinitos.

Finalmente, tras completar la ejecución del bucle de simulación, se devuelve una traza, donde cada línea representa un tick de simulación. Si el usuario quiere ver el estado del sistema mostrarla o volcarla en un fichero, se ha implementado el método `__repr__` que muestra el estado y las propiedades de cada una de las células del sistema. En caso de que el usuario quiera obtener a una célula a partir de su nombre puede usar el método `getCell`.

Compilador

Parser

El archivo `parser.py` extrae los elementos del código fuente del usuario para crear un objeto de tipo `Grammar`, que contiene un diccionario con todas las reglas de producción, el símbolo generador de la gramática y un diccionario con todos los símbolos. Dicho objeto tras ser analizado por el parser será enviado a el programa principal del compilador para diseñar la red.

El código fuente de la gramática puede ser escrito en un fichero o se le puede pasar al usuario mediante un fichero. La sintaxis del código es muy sencilla, cada carácter es un símbolo, y cada línea debe de contener una regla de producción formado por un símbolo terminal junto a una lista de cadenas de salida. El símbolo no terminal debe de ser separado de las cadenas de producción con el operador flecha `"->"` y el símbolo las cadenas de salida se separan entre sí con el operador barra `"|"`. Se considera símbolo generador la primera entrada de la primera regla. Para facilitar la comodidad de la escritura de reglas, se ignoran tabuladores, líneas vacías y espacios.

En caso de error sintáctico, el parser lanza una excepción con un mensaje que muestra la línea y el motivo del error.

A continuación se muestra un ejemplo del formato de sintaxis aceptada por dicho parser.

```
S->aA|bB
A->aA|a
B->bB|b
```

Diseñador de la red

El diseño de la red esta programado en el fichero `compiler.py`, en su función `compile`. Dicha función, tiene como parámetro de entrada un objeto de tipo `Grammar` generado anteriormente por el parser.

Para diseñar la red, se utiliza el algoritmo descrito utilizado en el libro *"Genetic Programming and Evolvable Machines (2022)" Teorema 1 – página 141*, donde se relata como construir una red genera cualquier gramática regular en forma normal Chomsky con 3 procesadores.

En primer lugar, se describen todos los conjuntos que conforman el vocabulario del sistema:

- N = **no terminales**: se obtienen de la clave del diccionario de reglas de producción
- T = **terminales**: son los símbolos obtenidos por la gramática exceptuando N
- $\mathrm{H} = \widehat{\mathrm{N}}$= **no terminales auxiliares**: se expresan como "$[\hat{}\mathrm{N}]$", por las limitaciones de los caracteres en ASCII
- TN = **concatenación de terminal y no terminal**: se expresa como "$[\mathrm{TN}]$" y es un conjunto formado de símbolos únicos, que representan una concatenación de un símbolo terminal y otro no terminal de la gramática.

- $V = N \cup T \cup H \cup TN =$ **vocabulario del sistema**: Unión de todos los conjuntos descritos anteriormente, incluyendo los auxiliares.

Una vez descritos esos conjuntos se procede a construir las 3 células del sistema usando los constructores de la clase `Cell`, las operaciones de la clase `CellOperation` y los filtros definidos en `filters.py`.

Para construir las operaciones de mutación, fue necesario crear el diccionario con todas las reglas para la célula $N1$ y $N3$. En primer lugar, as reglas de mutación de $N1$ son las mismas reglas de producción que en la gramática de entrada, uniendo las producciones en un solo símbolo. En segundo lugar el conjunto de reglas de N3, se ha unido dos conjuntos de reglas (diccionarios), uno que para cada símbolo conjunto devuelve su terminal y otra que para cada símbolo auxiliar no terminal, devuelve el base, es decir, le quita el símbolo "^".

Los filtros fueron sencillos de implementar, ya que la mayoría de ellos se podían hacer combinando distintos filtros implementados anteriormente. No obstante, para algunos filtros que emplean subcadenas con más de un símbolo fue necesario definir otras funciones especiales. Por ejemplo, para el filtro de salida N2, es necesario crear un filtro para que comprobará que la cadena al menos tiene 2 símbolos de longitud. También fue necesario crear otro filtro para que comprueba si una cadena es permitida en N3, si están en el siguiente conjunto $\{[aA]\hat{A} : a \in T \wedge A \in N\}$, usando tuplas para expresar todo el conjunto, y recorriendo secuencialmente la cadena para comprobar si está o no incluido.

Finalmente, una vez declaradas todas las células se conectan todas entre sí y de devuelve un objeto de tipo `CellNet`, sobre el cuál el usuario puede hacer sus simulaciones.

Instrucciones de uso

El usuario debe de tener instalado una versión de Python igual o más reciente que la 3.9, para hacer uso de tanto de compilador o el simulador. El código esta pensado para ser utilizado desde línea de comandos de Python, un cuaderno Jupyter u otro programa de Python.

Para compilar obtener un simuladore de redes celulares a partir de un gramática `compiler.compileCode` pasándole como argumento el código o `compiler.compileFile` con el nombre de fichero como argumento.

Una vez obtenida la red puedes usar el método `run`, para comenzar la simulación, con los argumentos `ticks` y `seconds`, puedes especificar cuanto dura la simulación. Tras finalizar la simulación se te devuelve la traza del fichero. Para consultar el estado del sistema puedes imprimir la red celular. Si se quiere acceder a los datos de una célula se puede hacer a través del método `getCell(nombre)`