SI Práctica 1 Memoria

Explicación de la clase Variable

```
class Word:
    def __init__(self, value="-", name=0,
                 initial_pos=(0, 0), final_pos=(0, 0),
                 length=0, orientation="horizontal",
                 feasibles=None, pounds=None, restrictions=None):
        self.value = value
```

Representa a cada palabra en el crucigrama

-. Value representa el valor.

-. Name representa el nombre. La primera variable tiene nombre 1.

-. Hay tres tipos de orientaciones: “horizontal”, “vertical” y “isolated”. La última se corresponde con variables encerradas en una sola casilla de forma que no tiene ninguna casilla no llena dyacente en ninguna de las direcciones norte, este, sur, oeste.

-. Feasibles es una lista

-. Pounds es una tabla hash de pares <key:value> donde key representa el nombre de la variable que ha realizado la poda y value es la lista de los valores que se han podado de la variable afectada por la poda

-. Restrictions es una tabla has de pares <key:value>, donde key representa el nombre de la variable que restringe y value se corresponde con las clases Restricciones asociadas

Explicación de la clase Restricción

```
class Restriction:
    def __init__ (self, word_restrainer = None, word_restricted = None,
                  x_coordinate = 0, y_coordinate = 0,
                  letter_of_restriction = "-",
                  AC3_check = False):
```

Word_restrainer -> palabra que restringe

Word_restricted -> palabra restringida

X_coordinate -> coordenada x de la casilla común

Y_coordinate -> coordenada y de la casilla común

Letter_of_restrction – valor de la celda cómun

AC3_check -> es una flag que indica si el arco (Word_restrainer, Word_restricted) debe de ser revisado o no durante la ejecución del algoritmo AC3

Explicación de cómo se han tratado las casillas con letras fijas.

Estas casillas han sido tratadas como **restricciones reflexivas,** es decir, restricciones de una variable consigo misma. El haber usado esta lógica me ha permitido poder podar de una forma directa los valores de los dominios que no satisfacen que estén puestas estas letras iniciales. Al poder acceder a dichas celdas que contienen dichas letras iniciales en tiempo O(1), he podado aquellos valores de los dominios incompatibles con estas restricciones reflexivas incluso antes de que comenzase el propio **forward_checking** a ponerse en acción

Explicación detallada de los algoritmos implementados

Voy a aprovechar este apartado para explicar de forma resumida cal ha sido mi metodología para afrontar esta práctica y también como ha sido y en qué ha consistido la implementación de la misma:

Metodología

Lo más difícil de esta práctica para mí sin duda ha sido la **depuración.** Los conceptos de **forward_checking, AC3…** los entendía a la perfección desde hace mucho tiempo. Pero aún así, siempre me acababa topando con algún error **invisible** que no era capaz de soluciones (en especial, aquellos que tenían que ver con el uso referencias compartidas y que requerían de usar **deepcopy**)

Para paliar este gran problema, he atacado al problema del crucigrama usando 3 herramientas muy útiles:

-. En primer lugar, **Tests Unitarios**. Estos me han servido para poder probar de forma sólida cada pequeña funcionalidad que iba añadiendo, evitando así el problema de las **regresiones** (errores que se encuentran en las primeras funcionalidades de un proyecto y que no aparecen hasta el final del mismo). Asimismo, me ha sido de gran utilidad ya que, cuando necesitaba cambiar algún detalle de alguna función que ya estaba implementada, simplemente ejecutaba los tests y veía de forma directa si este cambio había **roto** algo o no. En total realicé 51 test unitarios (+ 1 test que sirve para comprobar de forma automática si los dominios podados tras aplicar AC3 al curcigrama de Moodle son los esperados o no).

Para ejecutar los tests, basta con ejecutar:

python -m unittest tests/test_main.py

-. En Segundo lugar, el **uso de github.** Me ha resultado muy útil puesto que, para cada pequeña funcionalidad que creaba, hacía push a mi repositorio remoto de github. De esta forma, si **destrozaba** cualquiera de los tests y no era capaz de ver qué cambio era el que daba el problema, simplemente ejecutaba el comando **git reset –hard <log_del_commit>** y restauraba la práctica a una versión que funcionase a la perfección.

Enllace al github de esta práctica(acceder a la rama “cris”):

https://github.com/cacs2-ua/SI-Recu-Prac1-cacs2.git

-. En tercer lugar, el **uso de un launch.json para utilizar breakpoints.** Ellos me han suministrado una herramienta muy poderosa para poder comprobar de forma precisa el por qué una **nueva** funcionalidad no funciona como debería. El poder parar la ejecución del programa y poder ir línea por línea revisando el valor de todas y cada una de las variables dentro de un instante de la ejecución te ahorra una enorme cantidad de tiempo

Detalles de la implementación

IMPORTANTE

(En mi implementación, la **principal característica** es la utilización de tablas hash. Supe que era buena idea usarlas desde un principio gracias a su propiedad de poder acceder al valor de una key en tiempo constante. De esta forma, a la hora de llamar a los métodos **forward** y **restaura** de **forward_checking**, los accesos se hacían en O(1), y no en O(N), que es la complejidad de buscar si un elemento está en una lista o no. Esta misma lógica también se aplica en la función **revise** y **AC3** y, en general, en cualquier parte de mi código que haga uso de tablas hash)

Mi implementación consiste en lo siguiente:

-. En primer lugar, inicializo las posiciones, longitud y orientación de todas las variables del crucigrama.

-. A continuación, inicializo una tabla hash con los dominios (usando la clase Dominio) iniciales a partir de un fichero de texto. Cada índice de esta tabla hash contiene la longitud de las palabras de dicho dominio.

-. Una vez creada la tabla hash con los dominios, inicializo los dominios iniciales de cada una de las variables

-. Después, inicializo las restricciones iniciales (reflexivas) de cada una de las palabras.

-. Una vez hecho todo esto, se podan de los dominios aquellos valores que no satisfacen las restricciones reflexivas.

-. Con todo esto, tengo creada mi tabla hash de variables, la cual tiene tres **keys**: “horizontal”, “vertical” e “isolated” (en ese order. Cabe mencionar que las variables **isolated** se asignan al mismo tiempo que las variables verticales). La ventaja de tener organizadas las variables así es que, al verificar restricciones, cuando se está ejecutando el método **forward,** no hace falta mirar todas las variables, sino solo aquellas cuya orientación es **opuesta** a la variable que se está ejecutando dentro de **forward** (el caso de las variables **isolated** ya había sido tratadoal podar restricciones reflexivas))

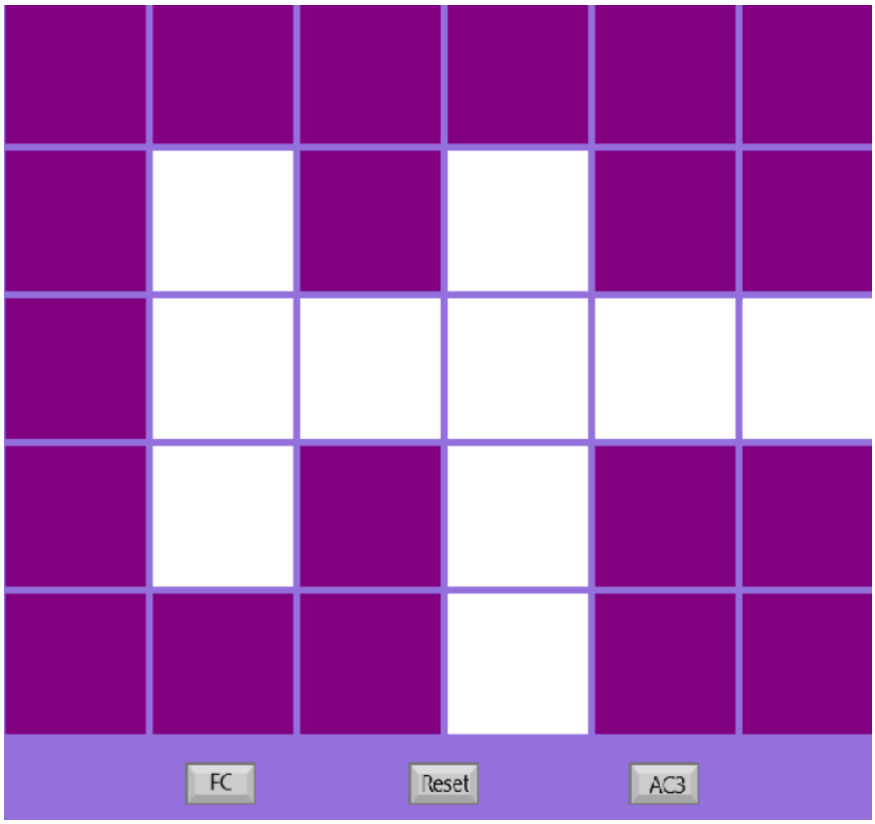
-. Si al haber podado las restricciones reflexivas ha quedado algún dominio vacío, entonces se devuelve Flase y el forward checking no llega a ejecutarse. En caso contrario, se ejecuta el forward checking propiamente dicho:

Para cada variable, se le asigna un valor de sus factibles y se poda los valores factibles de las variables futuras. Si queda algún dominio vacío, se deshacen las eliminaciones causadas por la variable en cuestión y se prueba con el siguiente valor de sus factibles. Si no ha quedado ningún dominio vacío, se pasa a la siguiente variable

Y con esto, quedaría resuelto, si tiene solución, el crucigrama en cuestión.

En cuanto al AC3, si se aplica, deja el problema a resolver en un estado de **arco-consistencia,** de forma que para cualquier variable no existe ningún valor dentro de sus factibles que no sea compatible con ningún valor dentro de los factibles de cualquier otra variable. Si aplica AC3 y después Forward Checking, este último empezará su ejecución directamente con el espacio de dominios en estado de **arco-consistencia**

Especificación formal de un problema pequeño (5 variables máximo) y diccionario de palabras pequeño. La especificación formal debe detallar la tupla <V, E, c, l, a>

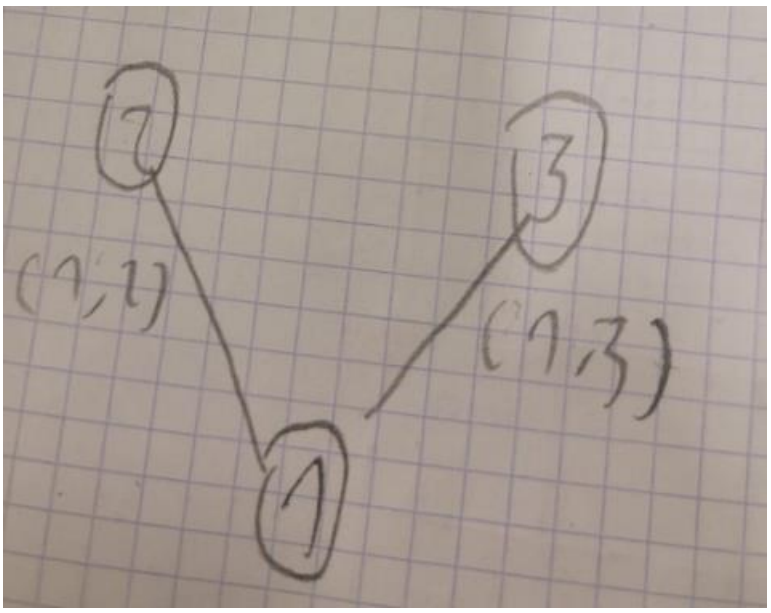


Abordaremos un problema de tamaño reducido. Para ello definiremos 3 variables: una horizontal y dos verticales. La horizontal tendrá un tamaño de 5 y la denominaremos variable 1, mientras que las verticales tendrán tamaños de 3 y 4, y las llamaremos variables 2 y 3, respectivamente.

Para solucionar el problema, utilizaremos un pequeño diccionario con 7 palabras: miel, pomo, gol, rama, menta, sal y araña. El programa asignará las palabras según su tamaño al dominio de cada variable, teniendo la variable 1 las palabras “menta” y “araña” como posibles, la variable 2 con "gol" y "sal", y la variable 3 con “miel”, “pomo” y “rama”. Con esto, tenemos todo adecuadamente preparado para realizar el análisis de este problema de tamaño reducido.

Dibujar el grafo de restricciones del problema especificado.

Podemos visualizar las restricciones de nuestro problema como un grafo donde las restricciones mismas actúan como arcos y los nodos representan nuestras variables. De esta manera, el grafo de restricciones para este problema sería el siguiente:



En nuestro caso, contamos con 2 restricciones debido a los 2 cruces entre palabras presentes en el crucigrama: entre las variables 1 y 2, y las variables 1 y 3.

V = {1, 2, 3}

E = {(1,2), (1,3)}

C = E (los nombres de las aristas coinciden con la nomenclatura de cada arista)

A = {1, 2, 3) (la implementación del problema está hecha de tal forma que el nombre de cada variable coincide con el orden de la asignación correspondiente que le da forward_checking)

Traza del problema pequeño de FC.

Ahora procederemos con una traza manual del funcionamiento del algoritmo Forward Checking para resolver el problema actual.

- Asigna el valor “menta” a la variable 1 (desde su dominio).
- Como no es la última variable, actualiza los dominios del resto de variables según las restricciones.
- Para la variable 2, ni “gol” ni “sal” cumplen la restricción con “menta” (la primera posición de la variable 1 y la segunda de la variable 2 deben coincidir), por lo que su dominio queda vacío.
- Al fallar en la actualización de los dominios de las variables, restaura todas las palabras podadas por la variable 1.
- Asigna entonces el siguiente valor del dominio de la variable 1, que es “araña”.
- Como no es la última variable, actualiza los dominios del resto de variables según las restricciones.
- Para la variable 2, “gol” no cumple la restricción, pero “sal” sí (la A en la primera posición de “araña” y en la segunda de “sal”). Para la variable 3, “miel” y “pomo” serían podadas, pero “rama” no (su segunda letra coincide con la tercera de “araña”). La operación es exitosa ya que no queda ningún dominio vacío.
- Al ser exitosa la actualización de los dominios, pasa a realizar el paso uno desde la variable 2.

- Asigna el valor “sal” desde el dominio (es el único que queda sin podar).
- Como no es la última variable, actualiza los dominios del resto de variables según las restricciones.
- No hay ninguna restricción con la variable 2 para ninguna variable no explorada, por lo que la operación termina con éxito al no quedar ningún dominio vacío.
- Al ser exitosa la actualización de los dominios, pasa a realizar el paso uno desde la variable 3.
- Asigna el valor “rama” desde el dominio (es el único que queda sin podar).
- Como es la última variable, finaliza con éxito la ejecución del algoritmo.
- Se realiza el retorno recursivo de los éxitos de las llamadas al algoritmo y finaliza con éxito la ejecución obteniendo una solución.

Traza del problema pequeño de AC3

- Se crea la lista Q con los arcos de pares de variables que tienen restricciones entre sí (en nuestro caso hay 4 arcos: (1,2), (1,3), (2,1) y (3,1)).
- Como Q no está vacía, se elige el primer arco, que sería (1,2).
- Para el dominio de la variable 1, se comprueba que cada palabra cumpla la restricción con al menos una del dominio de la variable 2.

- Dado que “menta” no cumple restricciones con ninguna palabra de la variable 2, se elimina, pero “araña” sí cumple con “sal”, por lo que se mantiene.
- Aunque el dominio no está vacío, se ha modificado, por lo que se añaden a Q los arcos de restricciones que apuntan hacia él y que no sean con la variable 2, es decir, el arco (3,1).
- Como Q no está vacía, se elige el siguiente arco: (1,3).
- Para el dominio de la variable 1, se verifica que cada palabra cumpla la restricción con al menos una del dominio de la variable 3.
- “Araña” cumple con “rama”, por lo que no se elimina ninguna palabra.
- Al no haber cambios en el dominio y no estar vacío, se elige el siguiente arco de Q: (2,1).
- “Gol” no cumple la restricción con “araña”, por lo que se elimina.
- Debido a este cambio, se añadirían a Q todos los arcos que apuntan hacia la variable 2 y que no provengan de la variable 1, pero como no hay ninguno, no se añade ninguno.
- Se pasa al siguiente arco de Q, que sería (3,1).
- Del dominio de la variable 3, “miel” y “pomo” no cumplen la restricción con “araña”, por lo que se eliminan.
- Como el dominio de la variable 3 no está vacío pero ha cambiado, se añade a la cola Q el arco (1,2).
- Finalmente, habría un par de iteraciones con los arcos (2,1) y (1,2) que no afectarían al resultado, concluyendo exitosamente el algoritmo.

Sección de experimentación y estudio de tiempos:

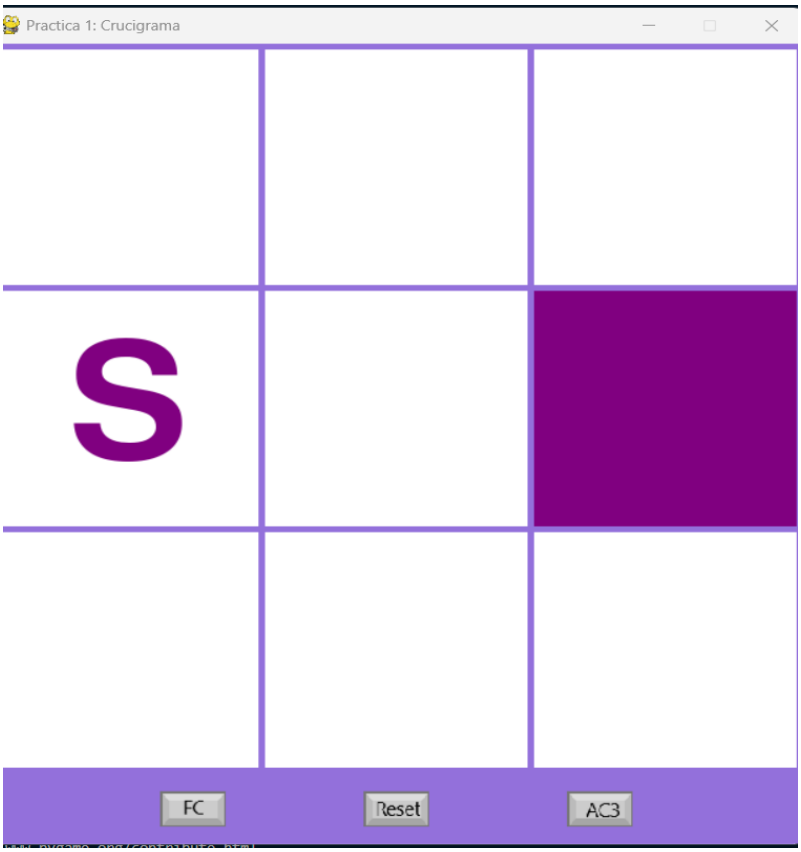
Prueba 1: debug1.txt (3x3)

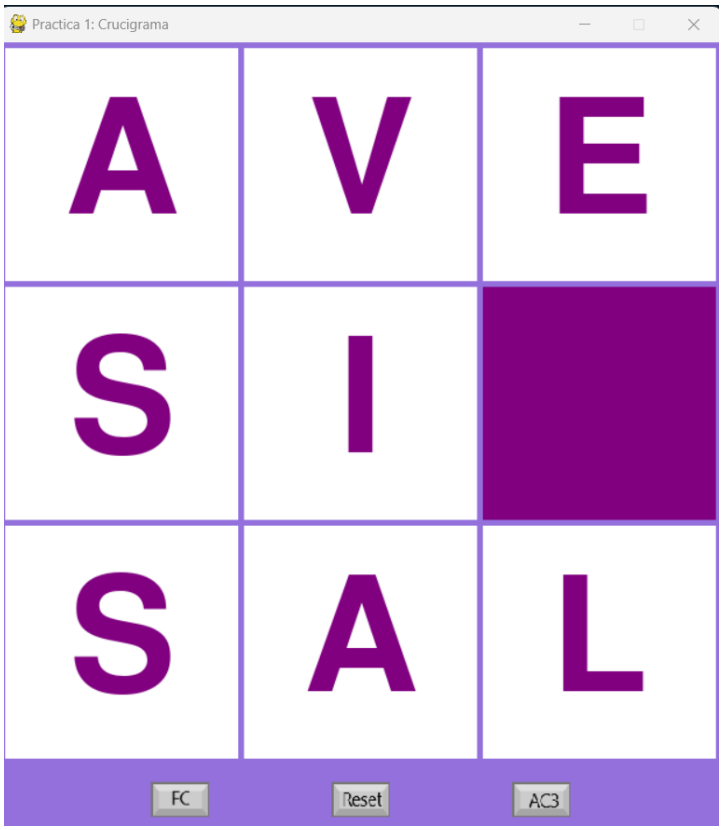
Forward Checking Execution time: 8.761 ms

AC3 Execution time: 35.564 ms

Forward Checking After AC3 Execution time: 3.530 ms

El objetivo de esta prueba es el de comprobar que FC y AC3 funcional con un crucigrama siendo lo más básico posible. Por eso el .txt se llama debug. Fue el crucigrama que ussé para depurar errores que me impedían resolver un crucigrama 5x6. Cuando mi FC logró resolvió este 3x3, pasó de no poder resolver a nada a poder resolver un 20x20 en 20 segundos





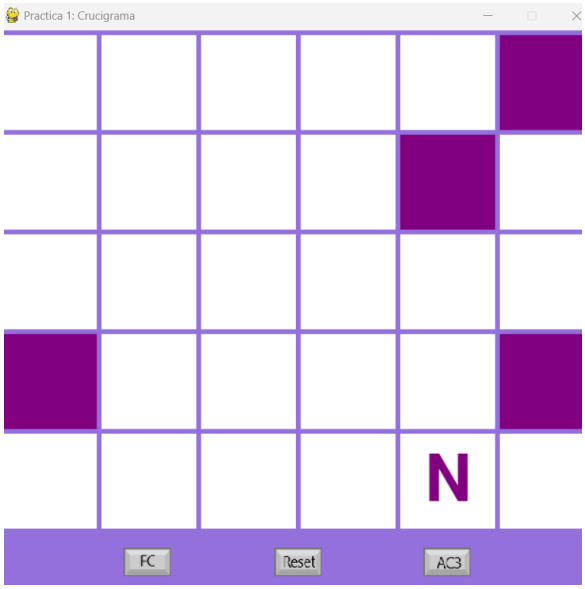
Prueba2: moodle_example.txt (5x6)

Forward Checking Execution time: 46.053 ms

AC3 Execution time: 146.976 ms

Forward Checking After AC3 Execution time: 13.831 ms

El objetivo de esta prueba es el de ver que todo funciona con un diagrama que todavía es sencillo pero el cual ya guarda un mínimo de complejidad:



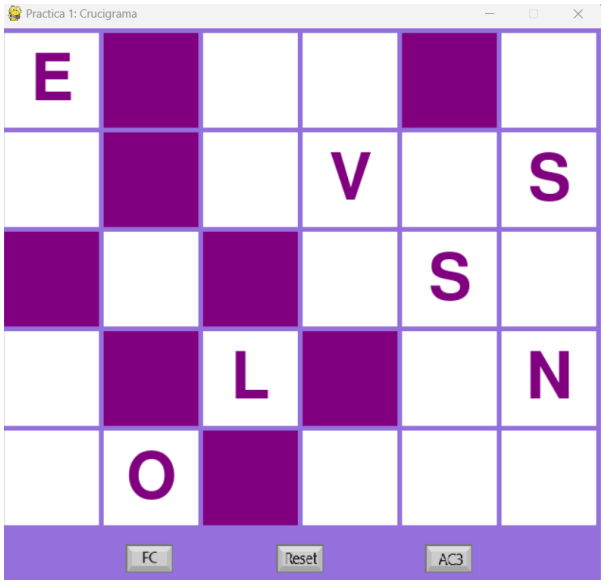
Prueba 3: Mine1.txt (5x6 with 150+ words in domain)

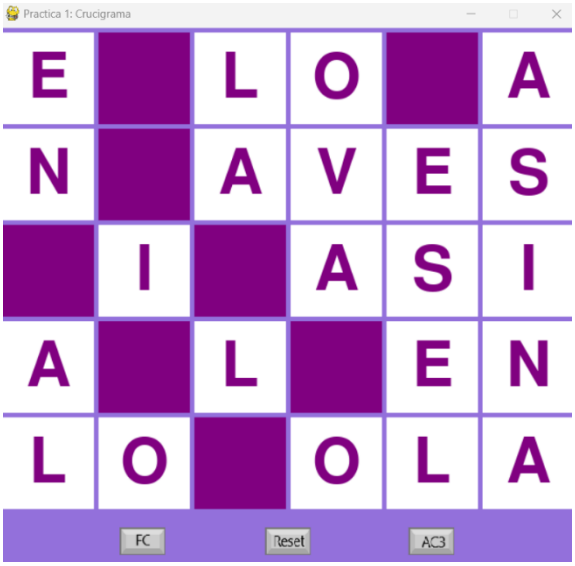
Forward Checking Execution time: 63.273 ms

AC3 Execution time: 81.669 ms

Forward Checking After AC3 Execution time: 15.540 ms

El objetivo de esta prueba es el de comprobar que todo va bien ya con una complejidad más alta, y también, **muy importante**, ver si todo se hace bien cuando se incluye variables aisladas (en este caso hay 2 variables **isolated**) Asimismo, se sigue usando un dominio pequeño





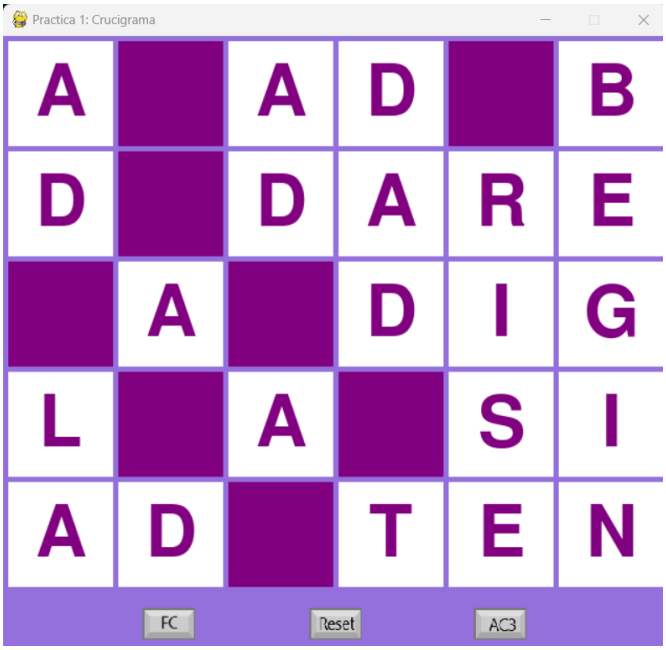
Prueba 4: Mine1.txt (5x6) with domains of more than 3000 words

Forward Checking Execution time: 489.729 ms

AC3 Execution time: 350.156 ms

Forward Checking After AC3 Execution time: 24.833 ms

Se trata del mismo crucigrama que en la prueba anterior pero sin letras iniciales. Además, ahora se emplea un dominio de palabras grande. El objetivo es ver que todo funciona y en un tiempo pequeño cuando el dominio es grande para un crucigrama pequeño:



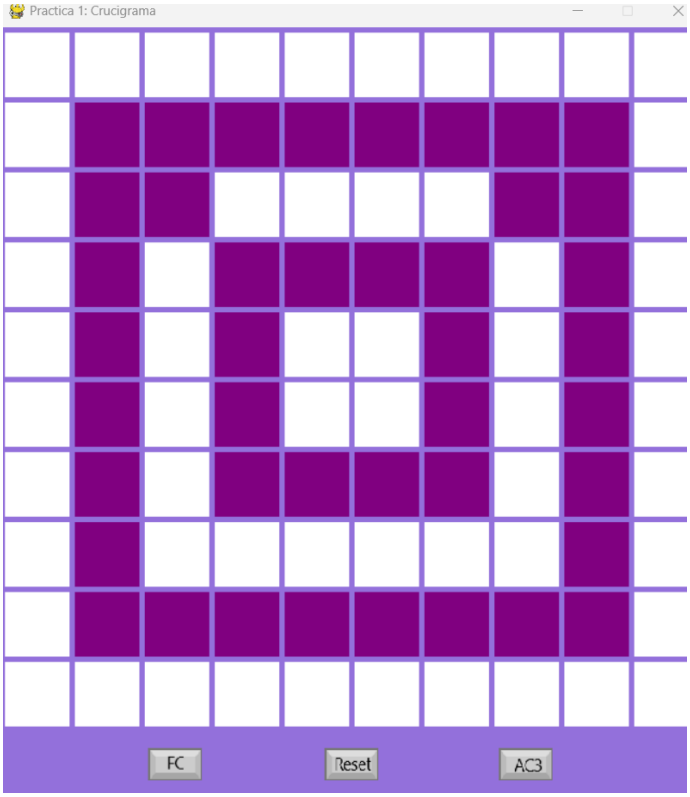
Prueba 15: simple.txt (10x10 and 3000 words domain)

Forward Checking Execution time: 1391.199 ms

AC3 Execution time: 26144.145 ms

Forward Checking After AC3 Execution time: 921.286 ms

A partir de aquí, las pruebas van dirigidas a comprobar el perfecto funcionamiento del programa en crucigramas que simulen un caso de la vida real. Es decir, las tres úllimas pruebas van dirigidas a testear la practicidad real del programa implementado. En esta prueba en concreto se prueba con un crucigrama “grande”, pero que no tiene tantas restricciones:



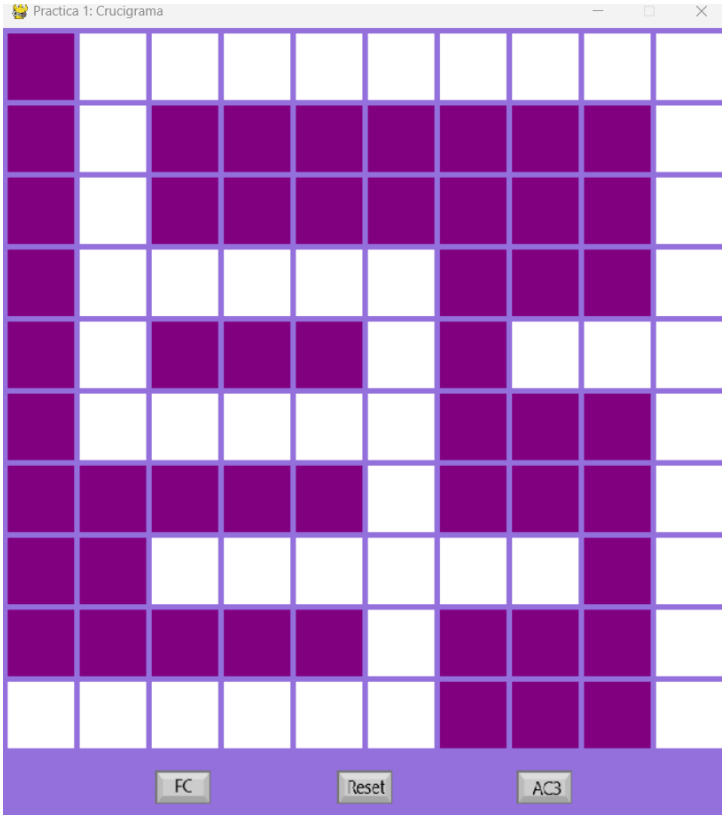
medium.txt (10x10 and 3000 words domain)

Forward Checking Execution time: 1186.752 ms

AC3 Execution time: 72019.004 ms

Forward Checking After AC3 Execution time: 968.730 ms

En esta prueba se testea un crucigrama del mismo tamaño pero con bastantes más restricciones



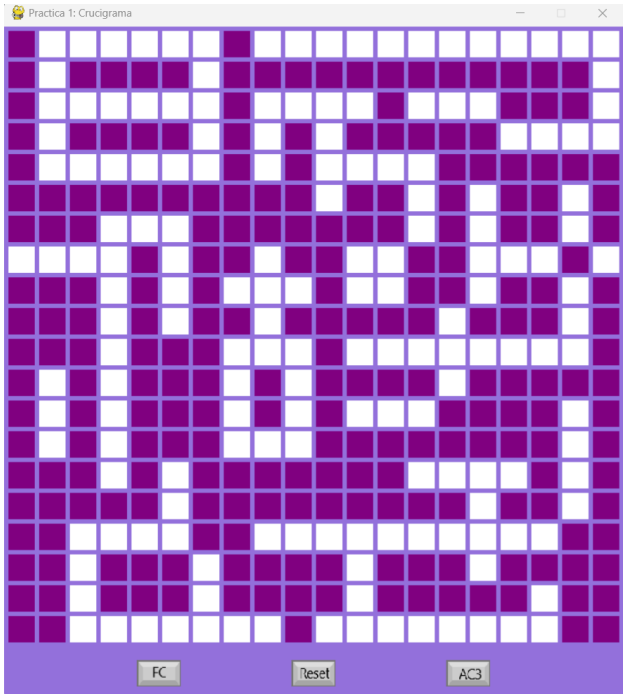
Complex.txt (20x20 and 3000 words domain)

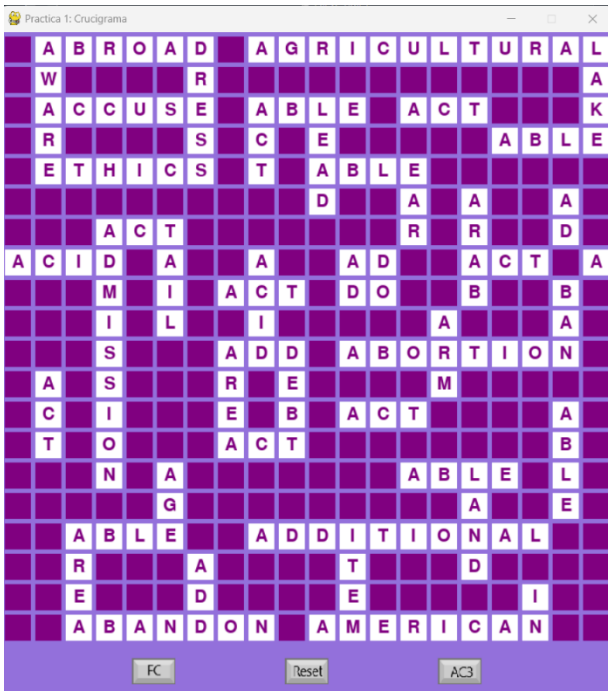
Forward Checking Execution time: 19722.470 ms

AC3 Execution time: 275762.292 ms

Forward Checking After AC3 Execution time: 16940.354 ms

Finalmente, en esta última prueba quería probar con un ejemplo potente de verdad para así comprobar si mi creación había merecido la pena o no. Como se observa en los tiempos, tarda 19 segundos solo con FC y 16s tras AC3. Como ya he mencionado antes, lo que me ha permitdio que mi programa resuelva problemas de verdad ha sido principalmente gracias al uso de tablas hash. Por curiosidad, lo que provoca que el AC3 vaya tan lento es el hecho de que, al tener las Restricciones como clase, cuando se hacen deepcopy sobre ellas, esto ralentiza mucho el programa. Si quisiera acelerar más aún el programa, implementaría las restricciones directamente mediante otra tabla hash y trataría de encontrar una solución alternativa a la deepcopy para solventar el problema de las referencias compartidas de memoria





BIBLIOGRAFÍA

Los tres últimos crucigramas, junto con sus dominios, los he sacado del siguiente github:

https://github.com/MahmoudHussienMohamed/AC3-CPP-Crossword-Solver