UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Una implementación de la heurística Colonia de Abejas Artificiales a una instancia del problema de la 3-partición: Tetris

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

P R E S E N T A :

José Ricardo Rodríguez Abreu

DIRECTOR DE TESIS:
Canek Peláez Valdés
2019

Hoja de Datos del Jurado:

1. Datos del alumno
   Rodríguez
   Abreu
   José Ricardo
   5526542430
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   309216139

2. Datos del tutor
   Dr
   Canek
   Peláez
   Valdés

3. Datos del sinodal 1
   Dra
   Adriana
   Ramírez
   Vigueras

4. Datos del sinodal 2
   Dr
   David Guillermo
   Romero
   Vargas

5. Datos del sinodal 3
   Dr
   José de Jesús
   Galaviz
   Casas

6. Datos del sinodal 4
   M en C
   Manuel Cristóbal
   López
   Michelone

7. Datos del trabajo escrito
   Una implementación de la heurística Colonia de Abejas Artificiales a una instancia del problema de la 3-partición: Tetris
   136 p
   2019

Agradecimientos

Mis padres Elia y Edgar que con su inagotable amor y con su ejemplo, siempre me impulsaron y enseñaron a superarme día a día.

A mis hermanos Luis y Mauricio que siempre han sido sinónimo de apoyo y estabilidad. Sin su sacrificio yo no estaría aquí.

A mi hermana Susana quién fue mi cómplice en más de una ocasión y siempre me ha brindado amor incondicional, gracias.

Gracias a mi tutor Canek Peláez Valdés a quien siempre le estaré eternamente agradecido por su guía, asesoría y dedicación a este trabajo.

A Karla Ramírez Pulido, por su paciencia, enseñanza y amistad en cada uno de los años que trabajamos juntos.

También quiero agradecer a José de Jesús Galaviz Casas por haberme recibido en la universidad y haberme mostrado que el crecimiento intelectual y personal siempre debe venir acompañado de humildad y responsabilidad científica.

Quiero agradecer de manera muy particular a Roberto Monroy Argumedo porque gracias a su apoyo incondicional y preocupación casi paternal hacia mi formación, fue que pude construir las bases sobre las que puedo asentar mi futuro. Descanse en paz.

Agradezco a todos mis profesores que a través de las aulas, dentro y fuera de la UNAM, aportaron en mi persona para mi desarrollo académico.

Por último quiero agradecer a la Universidad Nacional Autónoma de México y a la Facultad de Ciencias por brindarme todas las herramientas necesarias para una educación de calidad. De corazón, *¡Goya!*

Índice general

Capítulo 1

Introducción

Desde el inicio del uso de las computadoras y el uso de la *Máquina de Turing* como modelo de cómputo, uno de los principales objetivos ha sido el tratar de resolver problemas de manera automatizada. A partir de la década de los años cuarentas del siglo pasado y en pleno apogeo de la segunda guerra mundial, los programadores afrontaron la gran complicación de usar un enfoque para no orientar a las máquinas a calcular todos los posibles resultados sino disminuir el número de operaciones. Por aquella tormentosa época, matemáticos y criptógrafos no tuvieron opción más que reducir el espacio de búsqueda de sus problemas para optimizar el tiempo de ejecución de sus (muy) limitadas máquinas. Al reducir el espacio de búsqueda contribuyeron a encontrar soluciones de una manera más eficiente [19].

El modelo computacional que actualmente se usa puede resolver problemas cotidianos de manera bastante eficiente, como son mostrar la letra de una canción en uno de los buscadores más usado y más famoso [30]. Esta afirmación parte de la idea de que existen problemas que son fáciles de entender pero difíciles o imposibles de hacer que una computadora dé una respuesta; está demostrado que problemas como *The Halting problem* [1] no pueden ser resueltos por una máquina de Turing. Existen incógnitas que pueden ser fácilmente enunciables como *¿es N un número primo?*[2] o *¿puede una persona recorrer K lugares de forma eficiente sin repetir ninguno y al finalizar regresar al lugar de origen?*[3] pero al ser programadas, pueden tardar años en ser solucionados por una máquina si no se les da herramientas correctas de resolución [1].

Para todos aquellos problemas en los cuales crear un algoritmo y esperar una solución óptima no es posible, ya sea por su complejidad o porque simplemente no se sabe si existe un algoritmo eficientes, se usan otros métodos que aunque posiblemente no sean los óptimos, nos ayudan a dar soluciones que surgen con el objetivo de ser lo más efectivas posibles [48]. Algunas de las compañías de software de uso cotidiano y masivo, empresas tan conocidas como de transporte [61] o comercio electrónico[4], han usado estas alternativas como recurso para mejorar su impacto en el mercado [58], [57].

Dada una problemática bien definida y que se pueda probar que exista dentro de la clase de complejidad *NP*-completo o *NP*-duro y una heurística de optimización combinatoria como las definidas en [48], ¿cómo comparar si la solución propuesta es buena? Una forma es desarrollar la aplicación de la heurística al problema y analizar los resultados obtenidos. Para analizar las soluciones se deberá definir un objetivo a conseguir con los datos de entrada y un comportamiento que encuentre una solución al problema definido. De esta manera, se podría observar que los resultados son eficiente al

[1] La función `HALT` toma de entrada un par $\langle \alpha, x \rangle$ y regresa 1 sí y sólo si la máquina de Turing M_α se detiene dada la entrada x, en un número finito de pasos.

[2] Problema del número compuesto.

[3] Problema del agente viajero.

[4] Conocido como *e-commerce*, incluye empresas como Uber, Amazon, Netflix, Google, entre otras.

menos en algún punto de la ejecución.

El objetivo principal de este trabajo es tomar el problema de calcular los posibles desenlaces de un juego de Tetris y seleccionar una serie de movimientos adecuados, usando una heurística de solución genérica que realice optimización combinatoria como la mencionada en [38].

Se usará la heurística de nombre *Colonia de Abejas Artificiales* como método de cálculo para los posibles desenlaces de cada iteración del problema. La heurística consiste en dividir la búsqueda de posibles soluciones óptimas usando la reducción del espacio con la ayuda de distintas estrategias que se basan en las tácticas de un ente natural: conductas como el de reducir el lugar geográfico de una colonia de abejas para localizar fuentes de alimentos y mejorar el comportamiento de un panal. El propósito de usar esta heurística es adaptar una función de costo que se desarrollará sobre el contexto del juego de Tetris.

Se ha elegido como problema el juego de Tetris por las interesantes conclusiones que se han obtenido del trabajo en [15] y por la familiaridad que presenta para una gran cantidad de personas, por lo que entender sus reglas y lógica no conlleva un reto mayor al del propósito del presente trabajo. Los objetivos a optimizar por la heurística son: maximizar el número de filas eliminadass mientras la computadora juega; maximizar el número de piezas colocadas al finalizar el juego; maximizar el número de veces que se realiza un "tetris" (que es cuando se eliminan simultáneamente cuatro líneas); y minimizar la altura de la fila más alta en cualquier momento durante el juego.

En el capítulo dos se hablará de diferentes tipos de problemas computacionales, terminando con la explicación de un problema *NP*-completo que será el punto de partida para explicar el problema a tratar en esta tesis; Tetris, que será abordado en el capítulo cuatro y la solución a usar será descrita en el capítulo tres. Tanto en el capítulo cinco como en el seis se discutirá el ambiente y la implementación del problema mientras que en el siete los resultados del desempeño de la solución propuesta. Para finalizar este trabajo, las conclusiones se presentan en el capítulo ocho.

Capítulo 2

Problemas *NP*-completos y *NP*-duros

Si bien la palabra *computación* ha existido por cientos de años, fue hasta que su significado cambió que hizo necesario generar una clasificación de problemas. Se mostrará que la clasificación por clases de problemas fue una consecuencia natural temprana y se enuncian las definiciones de dichas clases.

2.1. Historia y definición

A principios del siglo XX, en agosto de 1900 se llevó acabo el segundo congreso internacional de matemáticas en París, Francia. Motivado por el congreso y la llegada de un nuevo siglo, el matemático David Hilbert planteó un conjunto de veintitrés problemas de lo que él llamó "el futuro problema de las matemáticas". Para 1902, Hilbert mantenía un optimismo sobre la resolución de sus problemas, diciendo: *For in mathematics there is no ignorabimus!* [Para las matemáticas no existe el *ignorabimus*[1]] [28]. Veintiocho años más tarde y aunque Hilbert no lo incluyó en su lista de veintitrés problemas originales, enunció otro para la comunidad matemática: crear un algoritmo que tome como entrada un predicado (con un número finito de axiomas y enunciados) y regrese como resultado "Sí" o "No" si es universalmente válido. Este problema es conocido como el *Entscheidungsproblem* o el problema de decisión [33]. En otras palabras, el problema de decisión consiste en averiguar si existe un algoritmo genérico que decida si una fórmula de cálculo de primer orden es un teorema.

En el año 1936, el matemático Alan Turing publicó un artículo que revolucionó e impactó al mundo y a las matemáticas; de nombre *Sobre los números computables, con una aplicación al problema de decisión* [2]; en este trabajo Turing definió *una máquina de cómputo* después denominada como *máquina de Turing* y una *máquina de cómputo universal*. De forma paralela y del otro lado del océano Atlántico, Alonso Church publicó su trabajo llamado en inglés *An Unsolvable Problem of Elementary Number Theory*, donde de manera independiente pero casi simultanea a Turing, trabajó sobre el *Entscheidungsproblem* usando el modelo llamado *cálculo lambda* [11]. Años más tarde, Alan Turing y Alonso Church trabajarían juntos para formular la equivalencia de sus modelos [13].

La máquina de Turing o MT es un modelo matemático de una computadora hipotética la cual usa un conjunto predefinido de reglas para determinar el resultado de un conjunto de variables de entrada[3]. La máquina de cómputo universal o *Máquina de Turing universal* es una máquina que

[1]La palabra *ignorabimus* a la que Hilbert hace referencia, tiene origen en un latinismo que dice *Ignoramus et ignorabimus* y significa "desconocemos y desconoceremos".

[2]El título original es *On computable numbers, with an application to the Entscheidungsproblem.*

[3][29] define formalmente a una máquina de Turing como un 7-tuplo $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$.

puede calcular cualquier secuencia computable dada una descripción de una máquina de Turing M. La máquina universal $U(M)$ puede realizar exactamente los mismos cálculos de M.

En ese mismo artículo Alan Turing, además de plantear el *Entscheidungsproblem* usando su máquina de Turing y la máquina universal para demostrar que el planteamiento de Hilbert era falso, Turing usa el proceso de diagonalización para mostrar que no se puede construir un proceso que tome la descripción general de una MT y nos diga si es libre de ciclos (*circle-free*) o terminará su ejecución; este problema se le conoce como *El problema del paro* o *The Halting Problem*. Con estos resultados Alan Turing crea una primera clasificación de problemas para las MT: los problemas computables y los no computables [60].

Posterior a esta primera clasificación, hubo la necesidad de seguir dividiendo los problemas desde el punto de vista de otra restricción que presentan las MT. Desde el inicio del uso de las primeras computadoras electrónicas, se descubrió que el modelo básico de las máquinas de Turing fallan al considerar la cantidad de tiempo o memoria que necesita una computadora real, un problema crítico en la actualidad pero más aún en aquellos primeros días de la computación moderna. La idea clave para medir el tiempo y el espacio en función de la longitud de la entrada llegó a principio de la década de los sesentas, por Juris Hartmanis y Richard Stearns cuyo artículo *Sobre la complejidad computacional de los algoritmos*[4] sentó las bases de lo que se conoce como complejidad computacional [31].

Al principio de la teoría, las investigaciones giraban en torno a sólo tratar de entender éstas nuevas formas de medición y cómo se relacionaban entre ellas. En los sesentas también se menciona por primera vez el término de *clases de complejidad* y se genera un concepto de eficiencia computacional al medir el tamaño de la entrada con polinomios [18].

Para entender las distintas *clases de complejidad* y antes de enunciar las definiciones formales de *P*, *NP*, *NP*-completo y *NP*-duro, se enunciará un ejemplo que puede ser útil para entender la dificultad de los cálculos que realizan las máquinas para encontrar soluciones de algoritmos no triviales: suponer que existe un gran grupo de estudiantes en los que se necesita que trabajen en equipo para un proyecto, se sabe qué estudiantes se llevan bien entre sí y se desea colocar a los estudiantes en equipos en los que todos se lleven bien. Es deseable que los equipos sean con la menor cantidad de integrantes, dos de ser posible y así todos los estudiantes realicen alguna parte del proyecto. Una forma de encontrar la solución sería calcular todos los posibles equipos y descartar los que tienen personas que no se lleven bien entre sí; pero para una muestra de 40 estudiantes, podrían existir más de 300 mil trillones de posibles equipos (las combinaciones que forman particiones del conjunto potencia de estudiantes). Este programa se le llama *Mínimo apareamiento maximal* [16]. En 1965, Jack Edmonds describió un algoritmo eficiente para resolver el problema de emparejamiento y sugirió una definición formal de *eficiencia computacional* [17]. La clase de problemas con soluciones eficientes (o polinomiales) sería luego renombrada como la clase P[5]. En 1960 Jack Edmonds dijó que un problema es fácil de resolver si el tiempo está limitado por un polinomio en el tamaño de su representación (en la clase P) [55].

Para muchos problemas parecidos y relacionados no se conoce un algoritmo eficiente que pueda resolverlos: regresando al ejemplo de los estudiantes, ¿cuál sería el algoritmo si ahora se hacen grupos de al menos tres? (partición de triángulos) ¿Qué pasa si se quiere sentar a los estudiantes en una mesa redonda sin que sean vecinos de algún alumno incompatible? (Ciclo Hamiltoniano) ¿Y si se ponen a los estudiantes en tres grupos y que cada estudiante se encuentre en un grupo sólo con personas compatibles? (3-coloración). Todos estos problemas tienen en común que dada una posible solución, se puede corroborar que sea correcta en un tiempo eficiente. La colección de problemas que tienen soluciones verificables en tiempo eficiente es conocida como la clase *NP*[6].

[4] *On the Computational Complexity of Algorithms* es el título original.
[5] Viene del inglés *Polynomial Time*.
[6] Viene del inglés *Nondeterministic Polynomial-Time*.

En 1971 Stephen Cook y Leonid Levin hicieron la demostración que lleva sus apellidos, en la que prueban que cualquier problema *NP* puede ser *reducido* en tiempo polinomial por una máquina de Turing determinista a un problema en específico. El concepto (mas no el término) de *NP*-completez fue presentado por primera vez en [12]. En la demostración del trabajo Cook-Levin, los autores muestran el *Problema de satisfacibilidad booleana*, también conocido como `SAT`, como el primer problema *NP*-completo (aunque posteriormente demuestran otros como **3-SAT**). `SAT` consiste en determinar si una fórmula booleana con variables y sin cuantificadores es o no satisfacible; en otras palabras, si existe alguna asignación de valores para sus variables que haga a la expresión verdadera.

Formalmente se dice que si φ es una fórmula booleana con variables $u_1, u_2, \ldots, u_n$ y $z \in \{0,1\}^n$, entonces $\varphi(z)$ denota el valor de φ cuando a las variables de φ le son asignados los valores de z. Una fórmula φ es satisfacible si existe una asignación z tal que $\varphi(z)$ sea verdadera. Si no existe z para $\varphi(z) = \texttt{TRUE}$, se dice que φ es insatisfacible [1]. Sólo un año después de la demostración, en su artículo *Reducibility Among Combinatorial Problems* el computólogo Richard M. Karp, usó las conclusiones del trabajo para definir 21 problemas *NP*-completos adicionales [40].

Para poder enunciar las definiciones formales de las clases *P* y *NP*, se necesita conocer al menos tres definiciones relacionadas previamente: lenguaje decidible, la clase `DTIME` y reductibilidad. Las siguientes siete definiciones formales son tomadas de [1]:

Definición 2.1.1. Se dice que una máquina decide un lenguaje $L \subseteq \{0,1\}^*$ si calcula la función $f_L : \{0,1\}^* \longrightarrow \{0,1\}$, donde $f_L(x) = 1 \Leftrightarrow x \in L$.

Definición 2.1.2. Sea $T : \mathbb{N} \longrightarrow \mathbb{N}$ alguna función. Un lenguaje L es elemento de $\texttt{DTIME}(T(n))$ si y sólo si existe alguna máquina de Turing que corra en tiempo $c \cdot T(n)$ para alguna constante $c > 0$ y que decida L.

Definición 2.1.3. Un lenguaje $L \subseteq \{0,1\}^*$ es polinonialmente **reducible** a un lenguaje $L' \subseteq \{0,1\}^*$ (también llamado *Karp reducible*), denotado por $L \leq_p L'$, si existe una función computable en tiempo polinomial $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ tal que para cada $x \in \{0,1\}^*$, $x \in L$ si y sólo si $f(x) \in L'$.

Dadas las definiciones anteriores y el ejemplo mencionado, se pueden enunciar las definiciones formales de las clases *P* y *NP* y el resto de las clases de complejidad:

Definición 2.1.4. Se define a la clase P como $P = \cup_{c \geq 1} \texttt{DTIME}(n^c)$.

Definición 2.1.5. Un lenguaje $L \subseteq \{0,1\}^*$ está en *NP* si existe un polinomio $p : \mathbb{N} \longrightarrow \mathbb{N}$ y una máquina de Turing M que se ejecute en tiempo polinomial (llamada la verificadora de L) tal que para cada $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)}, \text{ tal que } M(x,u) = 1.$$

Definición 2.1.6. Se dice que L' es *NP*-duro o *NP-hard* si $L \leq_p L'$ para cada $L \in NP$.

Definición 2.1.7. Se dice que L' es *NP*-completo si L' es *NP*-duro y $L' \in NP$.

2.2. 3-Partición

El problema de `3-PARTITION` o 3-Partición es un problema *NP*-completo, el cual enuncia que dado un conjunto finito $\mathcal{A}$ de $3m$ elementos, una cota $b \in \mathbb{Z}^+$ y una "medida"[7] $s(a) \in \mathbb{Z}^+$ con $a \in \mathcal{A}$, tal que satisfaga que $b/4 < s(a) < b/2$ y que $\sum_{a \in \mathcal{A}} s(a) = m \cdot b$, ¿puede $\mathcal{A}$ ser particionado en m conjuntos disjuntos $S_1, S_2, S_3, \ldots, S_m$ tal que para cada $1 \leq i \leq m$, $\sum_{a \in S_i} s(a) = b$? En otras

[7] El autor usa la palabra *"size"* como alternativa a una mejor palabra para la función.

palabras, dado un conjunto de N elementos, ¿puede encontrarse una partición de $N/3$ subconjuntos tal que la suma de los elementos de cada uno debe ser la misma y la cardinalidad de cada subconjunto debe ser tres?

La definición de 3-partición junto con la demostración de su NP-completez apareció en el año de 1979 y es visto como un problema de decisión [21]. Para la demostración de la NP-completez del problema se usa una reducción del problema de la 4-partición, el cual es NP-completo y previamente demostrado a partir del problema general de la partición (Figura 2.1). 3-partición es un problema de los llamados *totalmente o fuertemente NP-completos*, esto quiere decir que sigue siendo NP-completo incluso cuando los enteros (o elementos) de $\mathcal{A}$ están acotados por un polinomio en función de la longitud de la entrada.

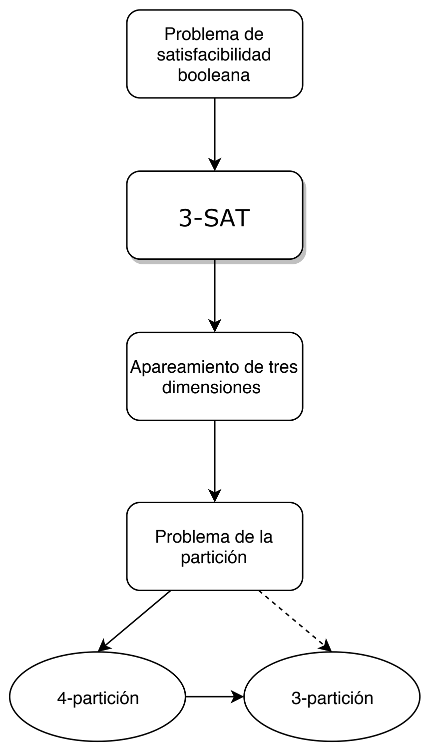

Figura 2.1: Diagrama de la secuencia de las transformaciones usadas para probar que 3-partición es NP-completo.

Este trabajo se encuentra íntimamente relacionado al problema y clasificación de la 3-partición debido al resultado de la *fuerte NP-completez*; esta propiedad será de utilidad cuando se tenga que explicar la simplificación de una representación unaria del problema a implementar, como se discutirá en el capítulo 4. Varias afirmaciones posteriores parten del hecho de que hay una reducción de la 3-partición por lo que no se sabe si existe un método de resolución eficiente al problema presentado en esta tesis.

Capítulo 3

Heurística: Colonia de Abejas Artificiales

Una fuerte característica del ser humano es su forma de proceder para resolver problemas. En las Ciencias de la Computación, la problemática de encontrar una respuesta incluye el camino de describir la forma de resolverlo. En este capítulo se abordan algunas formas de hacer que la computadora ejecute dicho camino y se discute una descripción que se implementa como método de solución.

3.1. Algoritmos

Suponga que una persona va al supermercado y crea una lista con todos los productos de su carrito. Cada producto tiene asociado un precio y se desea averiguar si el total del carrito es menor o igual a la cantidad de dinero disponible para efectuar la compra. ¿Cuál sería un método de solución adecuado para este problema? Aunque pareciera que el problema es muy fácil de resolver, el nivel de detalle que deben tener las indicaciones y pasos asociados a la solución, posiblemente no son triviales de enunciar para una persona que nunca ha descrito estos procesos a una computadora:

- **Input:** $L =$ Una lista de números que representan el precio de cada producto y un número N el cual representa el límite de gastos.
- **Output:** Un valor booleano de `TRUE` si la suma de los valores del carrito es menor o igual al número N, `FALSE` en caso contrario.

Algoritmo 3.1 Algoritmo SumaMenorQue.

procedure SumaMenorQue(L, N)
    $N \leftarrow N -$ SacaPrimero(L)
    while true do
        if $N < 0$ **then**
            return false
        if EsVacio(L) **then**
            return true
        $N \leftarrow N -$ SacaPrimero(L)

Para llegar a la solución de un problema, una persona utiliza muchas veces el instinto que va desarrollando a los largo de los años para analizar, atacar y ejecutar un proceso que da una solución. Si se intenta transmitir dicha técnica a una computadora, la manera más fácil de realizarlo es mediante un proceso llamado *algoritmo*. Se puede decir informalmente que un algoritmo es cualquier procedimiento bien definido que toma algún valor, o conjunto de valores como entrada (o **Input**) y produce algún valor, o conjunto de valores como salida (u **Output**) [14]. El análisis del problema en conjunto a la abstracción de los objetos asociados y el resultado, producen una serie de instrucciones que si se siguen correctamente, bajo una entrada I, siempre regresará una salida O.

La investigación sobre la formalización de la definición de la palabra *algoritmo* sigue siendo motivo de estudio hasta nuestros días [9]. Debido a los distintos tipos de problemas a resolver, existen muy diferentes procesos, entradas y salidas que se pueden producir. Muchas veces existe más de un algoritmo para resolver el mismo problema y algunos de ellos producen una salida de manera más óptima que otros[1] . La llamada “caracterización” de los algoritmos es algo que se ha discutido por más de doscientos años; un ejemplo de esto es el algoritmo de la criba de Eratóstenes que nos permite hallar todos los números primos menores a un número natural m dado.

Para propósitos de esta tesis, la definición informal que se usará será la siguiente: **Un algoritmo es una secuencia de pasos bien definida y finita tal que dada una entrada I, produce siempre la salida** O. Se considerará que los algoritmos deben tener también la característica de finitud ([42], [53] y [14]).

Así como existen problemas en los que escribir el algoritmo para encontrar la solución resulta sencillo y el proceso descrito pueda verse fácilmente eficiente (como el Algoritmo 3.1), existen problemas en los que dar la descripción para obtener la mejor solución no es una opción práctica.

3.1.1. `3-SAT`

Sea el problema `3SAT`, el cual consiste en dado un conjunto de fórmulas $\phi = \{x_1, x_2, x_3, ..., x_n\}$ con cada x_i una fórmula lógica de la forma $x_i = p_i \vee q_i \vee r_i$, con p_i, q_i, r_i, variables o términos lógicos, se deberá encontrar una interpretación $\mathcal{I}$ tal que $\mathcal{I}(\phi) = \mathcal{I}(x_i) = 1 \; \forall i \in \{1, 2, ..., n\}$. Existen algoritmos para resolver este problema y similares[2], sin embargo, su complejidad en tiempo es de la forma $O(X^n)$ con $3|\phi| = n$ y X una constante [46].

Para propósitos demostrativos, se ha creado un algoritmo de búsqueda exhaustiva. El código que se encuentra en el apéndice A sirve para correr ejemplos de fórmulas y encontrar interpretaciones en las que el valor de verdad de la fórmula sea `TRUE` (o 1). En caso de que no exista una interpretación verdadera, el programa implementado regresa el mensaje de que no se pudo realizar una asignación (lo que sería equivalente al valor de `FALSE` en `3-SAT`):

$$\mathcal{I}(s \vee j \vee \neg z) = \begin{cases} \mathcal{I}(s) & = & \texttt{FALSE} \\ \mathcal{I}(j) & = & \texttt{FALSE} \\ \mathcal{I}(z) & = & \texttt{FALSE} \end{cases}$$

Figura 3.1: Salida del programa en el apéndice A.

Se puede ver que $\mathcal{I}(\phi) = 1$ con la asignación de la Figura 3.1 ya que $\mathcal{I}(z) = 0 \to \mathcal{I}(\neg z) = 1$ y por lo tanto, $\mathcal{I}(s \vee j \vee \neg z) = 1$.

Con este mismo programa se puede forzar una fórmula que no tenga solución y así realizar el cálculo de todas las posibles combinaciones de asignaciones. Si se agrega como parámetro

[1]En este caso, el factor para que un algoritmo es más eficiente que otro se basa en la complejidad en espacio o tiempo (*big-O*).

[2]Recordar que en el capítulo 2 se aborda la equivalencia de los problemas NP-completos.

`-no-solucion`, el programa agregará el siguiente conjunto ϕ_{imp} de cláusulas al programa, definido en Cuadro 3.1.

P	Q	R	$\overbrace{(P \vee Q \vee R)}^{\textbf{(a)}}$	$\overbrace{(\neg P \vee Q \vee R)}^{\textbf{(b)}}$	$\overbrace{(P \vee \neg Q \vee R)}^{\textbf{(c)}}$	$\overbrace{(P \vee Q \vee \neg R)}^{\textbf{(d)}}$
T	T	T	T	T	T	T
T	T	F	T	T	T	T
T	F	T	T	T	T	T
T	F	F	T	F	T	T
F	T	T	T	T	T	T
F	T	F	T	T	F	T
F	F	T	T	T	T	F
F	F	F	F	T	T	T

P	Q	R	$\overbrace{(\neg P \vee \neg Q \vee R)}^{\textbf{(e)}}$	$\overbrace{(\neg P \vee Q \vee \neg R)}^{\textbf{(f)}}$	$\overbrace{(P \vee \neg Q \vee \neg R)}^{\textbf{(g)}}$	$\overbrace{(\neg P \vee \neg Q \vee \neg R)}^{\textbf{(h)}}$
T	T	T	T	T	T	F
T	T	F	F	T	T	T
T	F	T	T	F	T	T
T	F	F	T	T	T	T
F	T	T	T	T	F	T
F	T	F	T	T	T	T
F	F	T	T	T	T	T
F	F	F	T	T	T	T

Tabla 3.1: Tabla de verdad con las fórmulas de todas las posibles combinaciones de tres variables P, Q y R en la forma normal conjuntiva *(NFC)*.

Como se puede observar en la tabla Cuadro 3.1, existe siempre para alguna combinación de valores de P, Q y R, el valor de F (o falso) en todas las columnas y filas, por lo tanto, como se ve en la tabla 3.2, se puede suponer que $\nexists\, \mathcal{I} \mid \mathcal{I}(\phi_{imp}) = 1$.

P	Q	R	$(a) \wedge (b) \wedge (c) \wedge (d) \wedge (e) \wedge (f) \wedge (g) \wedge (h)$
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Tabla 3.2: Tabla de verdad con las conjunciones del resultado de la tabla Cuadro 3.1.

```
# python sat.py --no-solucion
No se pudo realizar asignación, se intentó 1 fórmula(s).
Num de asignaciones que se realizaron: 14
```

Listado 3.1: Ejecución de una fórmula sin solución.

Al no existir una posible combinación para este conjunto de fórmulas, el algoritmo en el apéndice A asignará todas las posibles combinaciones de valores a cada una de las variables; para tres variables el número de asignaciones que se realizan es 14, que es el resultado de la suma del total de las llamadas recursivas y es el número total de aristas en el árbol de búsqueda. El hecho de que el árbol sea binario depende sólo de que existan dos posibles valores de verdad para cada variable.

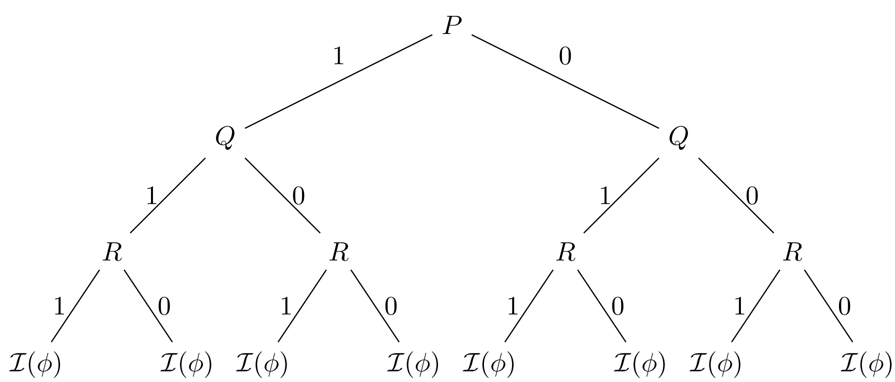

Figura 3.2: Árbol de asignaciones de valores con tres variables.

Para una cláusula con tres variables, el árbol realiza 14 asignaciones, para dos cláusulas con seis variables el árbol tendrá 126 aristas representando las asignaciones, para tres claúsulas $1,022$ y para 21 variables el número se eleva a $4,194,302$. Para 33 variables (u once cláusulas) se corrió el programa durante más de un día debido a que el árbol es demasiado grande para revisar todas las asignaciones en menos de 24 horas[3,4].

El algoritmo dado produce la solución (si es que existe) pero el costo en tiempo es demasiado alto. Este algoritmo es lo que se le denomina de tiempo "exponencial" y esto se debe a que el número de asignaciones crece de la siguiente manera (Figura 3.3):

[3] Se realizó la experimentación y se hizo el total de asignaciones en 26 horas con 25 minutos y 41 segundos.

[4] El número obtenido fue $17,179,869,182$ (diecisiete mil ciento setenta y nueve millones ochocientos sesenta y nueve mil ciento ochenta y dos).

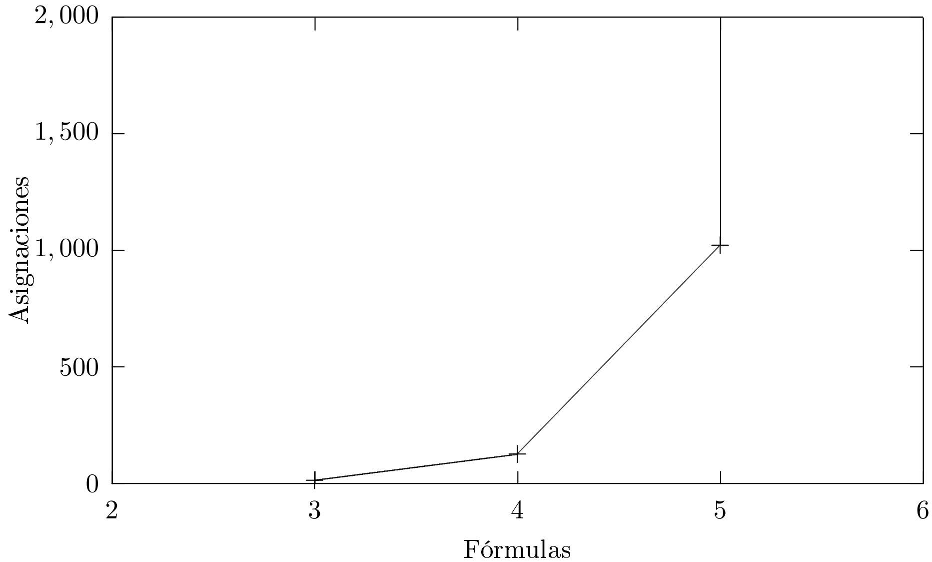

Figura 3.3: Muestra del crecimiento de asignaciones respecto a las fórmulas de ϕ.

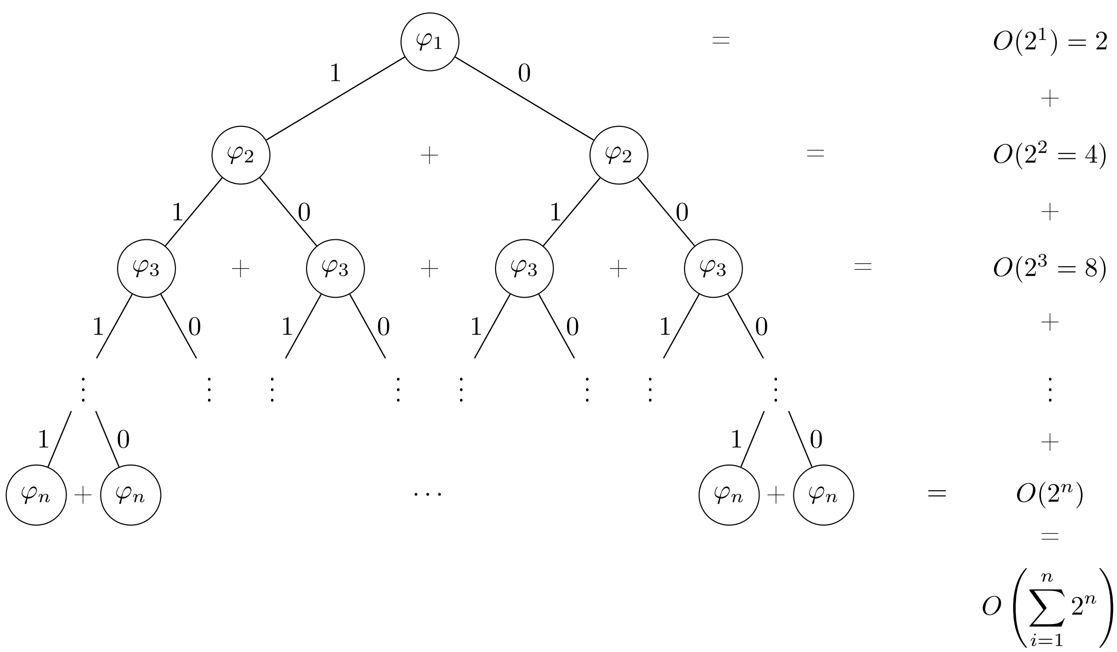

Figura 3.4: Posibles valores de verdad por cada variable en un árbol binario.

En el 2010 se publicó un artículo donde describen un algoritmo que reduce la complejidad a $O(1.439^n)$ y existe una competencia anual que premia a las mejores implementaciones para encontrar soluciones al problema `SAT`,[43] [54], sin embargo las soluciones siguen siendo exponenciales. Habría que mencionar que si se encuentra una solución en tiempo P se estaría demostrando que $P = NP$.

3.2. Heurísticas

Una alternativa a solucionar problemas como `3-SAT` o equivalentes[5] son estrategias y procesos que se utilizan fácilmente debido a la información libremente aplicada para controlar los procesos de resolución de problemas en máquinas. Estos procesos son criterios, métodos o principios para decidir cuál de los varios cursos de acción alternativos promete ser el más eficaz para lograr algún objetivo. Estos procesos reciben el nombre de **heurísticas** [48].

A diferencia de los algoritmos[6] informalmente definidos previamente, las heurísticas también son una secuencia de pasos bien definida pero dada una entrada I, produce una salida $o \in \texttt{OUTPUT}$ con `OUTPUT` un conjunto de posibles valores de solución. Las respuestas dadas por las heurísticas normalmente no suelen ser la mejor solución, pero intentan producir una solución *suficientemente buena* [23]. Generalmente sus salidas son el resultado de ejecuciones de funciones estocásticas y raramente almacenan información del proceso de obtención del resultado previo [32],[48].

Las heurísticas son tan diversas que es difícil clasificarlas exclusivamente en sólo una categoría, sin embargo existen características que comparten ciertas heurísticas: los métodos resolutivos llamados de *búsqueda tabú* son aquellos que determinan una posible respuesta, las marcan y premian la exploración de soluciones alejadas de las posibles respuestas marcadas. Los algoritmos (heurísticos) genéticos están por su parte inspirados en la evolución biológica, modificando poblaciones (de objetos a mejorar) y premiando a las soluciones más cercanas al objetivo. Existen muchos otras heurísticas como las redes neuronales o reocido simulado y si bien todas diferentes, comparten la característica de producir soluciones sin garantizar necesariamente que sea la mejor.

Cuando se habla de heurísticas, es importante mencionar que en algunos textos [48] hablan sobre una cierta *intuición* o criterio cuando se refiere al plantearlas como métodos resolutivos, por los que en algunas ocasiones es difícil entender el *"¿por qué funciona?"*; se propone que la intuición mencionada es la capacidad de hacer predicción sobre un conjunto de datos o la eliminación de información previa innecesaria (o "ruido") [23].

3.2.1. El problema de las 8 reinas

Franz Nauck publicó en 1850 un (ahora) famoso problema para el matemático Gauss: el problema consiste en obtener el método para determinar cómo ocho reinas pueden ser colocadas en un tablero de ajedrez[7] de tal manera que una reina no pueda *tomar* a otra [3]. En otras palabras, dos reinas no pueden estar colocadas en la misma columna, fila o diagonal.

[5] Que sea un problema perteneciente a la clase NP o con una cota $big - O$ que se quisiera reducir.
[6] En algunos textos, como [4] las heurísticas son llamadas *algoritmos heurísticos*.
[7] El problema general es colocar n reinas en un tablero de n^2 casillas.

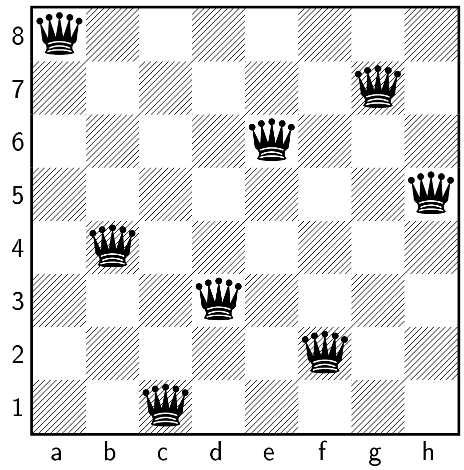

Figura 3.5: Ejemplo de solución al poblema de las 8 reinas.

Poco después del planteamiento (1848), este problema fue resuelto y aparecieron un total de 40 formas de encontras las distintas soluciones entre los años 1849 y 1854 en la revista de ajedrez alemana *Deutsche Schachzeitung* [10]. El problema tiene 92 distintas soluciones de las cuales si se eliminan las no obtenibles por giros o simetrías (es decir, eliminando las soluciones isomorfas) quedan 12 soluciones diferentes. El problema de las 8 reinas es, desde un punto de vista computacional, interesante de usar como ejemplo de un problema que es resuelto por una heurística, debido a que la versión generalizada del problema, de las n-reinas, es un problema NP-completo [22].

Como muestra de la aplicación de una heurística, se implementó un conjunto de funciones que usan una función de costo, una entrada y un factor aleatorio para hayar soluciones al problema de las reinas. El código se puede leer en el Apéndice B. La función de costo y funcionamiento de la heurística es la que se muestra a continuación.

La secuencia de ordenamiento de reinas no es aleatorio sino sistemático, de esta manera se asegura que no se genera la misma combinación de posiciones una y otra vez eficientando la heurística. Al no crear las mismas combinaciones se tendrá más probabilidad de éxito de generar alguna combinación deseada. Una forma de sistematizar el generamiento de posiciones de las reinas es colocándolas una a la vez empezando con un tablero vacío, hasta que todas estén colocadas.

Para poder asignar las reinas se debe tener en consideración que dada una reina puesta con anterioridad, se reduce el número de casillas donde puede ser colocada sin correr el riesgo de ser “tomada” por otra reina. Una casilla es candidata a ser asignada a una reina prioritariamente si al colocarla, ésta deja un número alto de casillas “no atacadas”, esto es, que deja la mayor cantidad de casillas para la posterior asignación del resto de las reinas. Aquí hay un ejemplo:

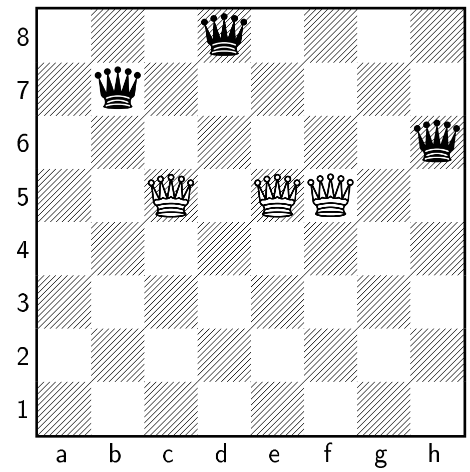

Figura 3.6: Problemática de asignación de una reina en c5, e5 o f5.

Considérese a las reinas blancas como "no asignadas". Para la función de costo, se utilizó como información el número de casillas donde se puedan colocar reinas en la próxima iteración, tratando de maximizarlo para tener más oportunidades de éxito. El numero de casillas que quedan se transforma en el método de asignación:

$$\begin{aligned} f(c5) &= 8 \\ f(e5) &= 9 \\ f(f5) &= 10. \end{aligned}$$

La opción que la heurística tomaría en este caso sería aquella donde la reina es asignada a la casilla $c5$. Algunas iteraciones llegan al caso donde $f(X) = f(Y)$ y aquí es donde se realiza una asignación aleatoria y se escoge cualquiera de las dos. Es posible que se llegue a un punto donde no se pueda seguir asignando más reinas, en este caso la heurística vuelve a partir de cero con un nuevo tablero.

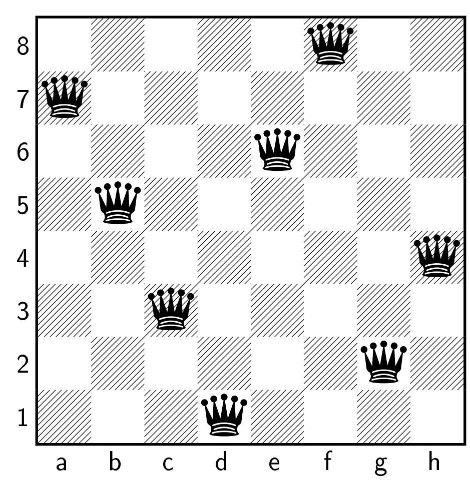

Figura 3.7: Resultado de la ejecución de la heurística de reinas con un tablero de 8 x 8.

3.3. Aproximación numérica como solución a problemas *NP*

La búsqueda de alternativas para resolver problemas no tratables de manera frontal no es un tema nuevo: en 1739 fueron publicadas varias notas de Sir Isaac Newton, las cuales contienen un método para encontrar aproximaciones de raíces a funciones. El método de Newton converge sólo bajo ciertas condiciones; sin estas condiciones el procedimiento fácilmente podría alejarse del resultado esperado [24]. Para obtener soluciones *buenas* dado un método, hay que definir lo que significa *bueno* y esto consiste en un conjunto de condiciones y reglas que servirán para acotar algún resultado previamente definido.

En matemáticas, los métodos de optimización son aquellos que consisten en maximizar o minimizar una función por cada iteración, manipulando los valores dentro de un dominio definido para aproximar a algún objetivo. Los problemas de optimización tienen la forma:

$$\text{mín}\, f_0(x), \text{ sujeto a } f_i(x) \leq b_i, i = 1, ..., m,$$

donde el vector $x = (x_1, ..., x_n)$ es la variable de optimización del problema; la función $f_0 : \mathbb{R}^n \to \mathbb{R}$ es la función de costo u objetivo; las funciones $f_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, ..., m$, son de restricción y las constantes $b_1, ..., b_m$ son los valores límites para las funciones de restricción. Se dice que un vector x^* es óptimo (o una solución del problema) si tiene el menor (o mayor en caso de maximizar) valor objetivo de todos los vectores que satisfacen el límite; para todo z con $f_1(z) \leq b_1, ..., f_m(z) \leq b_m$, se tiene que $f_0(z) \geq f_0(x^*)$ [6].

En Ciencias de la Computación, los algoritmos de aproximación y heurísticas numéricas son métodos de resolución (generalmente a problemas *NP*), que aunque no proveen de *puntos de apoyo* para encontrar la mejor solución, sí ofrecen estos puntos para obtener soluciones que se *aproximan* a la óptima de manera eficiente. Estos métodos de resolución parten de la conjetura de $P \neq NP$ para aseverar su eficiencia [62].

A diferencia de los algoritmos de aproximación, las heurísticas no garantizan un resultado dentro de una constante c de factibilidad cercana al óptimo, ni tienen que cumplir la condición de que termine en tiempo polinomial respecto al tamaño de la entrada. Las heurísticas se pueden comportar de manera muy pobre si se considera el peor caso, sumado a un subconjunto de instancias del problema que harían su desempeño poco destacable. Sin embargo, buenas heurísticas pueden superar el desempeño de muchos algoritmos con muchas instancias [2], reduciendo el espacio de búsqueda de problemas (incluidos los de la clase *NP*-duro).

3.4. Colonia de abejas artificiales

La observación del comportamiento de enjambres ha ganado terreno en el interés de los científicos debido a las formas particulares en las que estas comunidades resuelven sus problemas. Dervis Karaboga enumera dos principales conceptos que son necesarios para que los enjambres obtengan el comportamiento de inteligencia: la organización del enjambre y la división de labores. Karaboga describió en el año 2005 la heurística que se usa en esta tesis y que lleva como nombre *Colonia de abejas artificiales* o *ABC*[8].

El modelo minimalista del comportamiento de una colonia de abejas real que Keraboga describe como base del funcionamiento para su heurística enumera una serie de agentes que tienen como propósito el emular condiciones de un panal de abejas en la intemperie de forma artificial. Las funciones principales están divididas tres componentes esenciales:

[8]Por sus iniciales en inglés *Artificial Bee Colony.*

1. Fuente de alimentación: La fuente es un espacio abierto proveedor de *néctar* que puede o no estar delimitado por ciertas reglas. El valor de la fuente fluctuará respecto a la rentabilidad de la solución.

2. Abejas recolectoras empleadas: Tienen asociada una fuente de alimentación que consumen continuamente. Cada abeja *contiene* la información de su fuente.

3. Abejas recolectoras desempleadas: Buscan continuamente una fuente de alimentación que consumir. Existen a su vez, dos tipos de abejas recolectoras desempleadas:

    a) Exploradoras: Son entre 5-10 % de la colmena. Exploran el espacio de búsqueda para encontrar nuevas fuentes de alimento.

    b) Observadoras: Esperan en la colmena y clasifican las fuentes de alimentación con la información que provee el resto del enjambre.

Para que exista la organización como uno de los conceptos principales de un enjambre, es necesario un proceso de comunicación. Como ocurre con las colmenas de abejas en la naturaleza, Karaboga propone un área de *baile*; este baile que llama en inglés *waggle dance* comunica a las abejas observadoras el valor de la fuente y cada una puede decidir cuál fuente de alimentación tiene la mejor valoración. Las abejas recolectoras empleadas comparten su información en el área de baile junto a una probabilidad proporcional a la rentabilidad de la fuente de comida, reclutando abejas de manera proporcional a la *duración* de su baile.

En un principio, una posible abeja recolectora cualquiera α empezará como una abeja recolectora desempleada. La abeja α no tendrá información de ninguna fuente de comida y tendrá dos opciones:

1. Puede seleccionar convertirse a exploradora y comenzar a buscar alrededor de la colmena *aleatoriamente* en busca de una fuente de alimentos.

2. Puede ser reclutada por otras abejas que estén realizando el *waggle dance*.

Después de localizar una fuente de comida la abeja α la *explora* y posteriormente, la ahora abeja recolectora empleada α realiza una *recolección de néctar* en la fuente y áreas vecinas para regresar a la colmena donde puede continuar con cualquiera de las siguientes tres acciones:

1. Convertirse en una abeja observadora después de abandonar su fuente de alimento.

2. Bailar para reclutar más abejas antes de regresar a su fuente.

3. Continuar consumiendo su fuente de alimento sin reclutar nuevas abejas.

Para la heurística, es importante que no todas las abejas tengan un estado de recolectoras ni observadoras simultáneamente. De acuerdo al artículo original, la experimentación muestra que nuevas abejas obtienen el estado de recolectoras a un ritmo proporcional a la diferencia del número total de abejas y el número de abejas recolectoras actuales.

Para el caso de las abejas y la heurística, las propiedades en las que se basa el comportamiento colectivo y funcionamiento de la colmena, son:

- **Reacción positiva:** Subiendo la cantidad de *néctar* recolectado en una fuente, el número de abejas observadoras que la visitan es mayor.

- **Reacción negativa:** La exploración y explotación de una fuente es abandonada.

- **Fluctuación:** Las abejas exploradoras llevan a cabo búsquedas aleatorias para encontrar nuevas fuentes de alimento.

- **Interacción:** Las abejas se comunican mediante el *waggle dance*.

Para la implementación de la heurística, se propone que la mitad de la colonia esté conformada por abejas recolectoras empleadas artificiales; y la segunda mitad sean clasificadas como observadoras. Para cada fuente de alimento, existe sólo una abeja recolectora empleada; en otras palabras, el número de abejas empleadas es igual al número de fuentes de alimento. Las abejas que agoten su fuente de alimento, se convertirán en abejas exploradoras [38]. Los pasos propuestos por el autor, son los siguientes:

Algoritmo 3.2 Pseudocódigo de ABC.

```
procedure ABC
    repeat
        Inicializa las abejas exploradoras para encontrar fuentes iniciales.
        Mandar a las abejas empleadas para determinar la cantidad de néctar.
        Calcular el valor probable de la fuente que las abejas observadoras visitarán.
        Detener la exploración de las fuentes no seleccionadas.
        Mandar a las abejas exploradoras a buscar nuevas fuentes de forma aleatoria.
        Guardar la mejor fuente de alimento.
    until condiciones sean cumplidas
```

Como muchas otras heurísticas, la búsqueda de las abejas maximiza la proporción dada por la energía (o alimento) obtenida E y el tiempo T de exploración. En problemas de maximización, el objetivo es encontrar el máximo valor de la función $F(\theta)$, $\theta \in R^p$ con R el proceso de recolección. Si θ_i es la posición de la i-ésima fuente de alimentación; $F(\theta_i)$ representa la cantidad de néctar obtenido de la fuente θ_i y es proporcional a la energía $E(\theta_i)$. Sea c el número de ciclo y n el número de fuentes cerca del panal, entonces $P(c) = \{\theta_i(c)|i = 1, 2, ..., n\}$ representa a la muestra de fuentes que son visitadas en el ciclo c. La probabilidad con la que una fuente de alimento localizada en θ_i sea escogida por una abeja observadora puede ser expresada como:

$$P_i = \frac{F(\theta_i)}{\sum_{k=1}^{n} F(\theta_k)}.$$

Después de observar a la abeja hacer el *waggle dance*, las abejas observadoras visitan la fuente θ_i dada la probabilidad P_i y determinan fuentes vecinas para tomar su néctar. La posición de las fuentes vecinas es determinada de la siguiente forma:

$$\theta_i(c+1) = \theta_i(c) \pm \phi_i(c),$$

con $\phi_i(c)$ un factor aleatorio para encontrar una fuente con mayor néctar que θ_i. Si la cantidad de néctar $F(\theta_i(c+1))$ al momento $\theta_i(c+1)$ es mayor que el néctar al momento $\theta_i(c)$, entonces al regresar la abeja al panal comparte la información con otras abejas y la posición de la fuente es actualizada a $\theta_i(c+1)$, de otra manera se mantiene la posición $\theta_i(c)$ de la fuente. Si una fuente i no puede ser actualizada después de un número fijo de intentos, entonces la fuente es abandonada y se explora en busca de una nueva fuente [39].

Capítulo 4

Tetris

La segunda mitad del siglo XX trajo consigo un avance tecnológico significativo como fue la integración gradual de las computadoras, como herramienta práctica para resolver problemas de forma rápida y eficiente. Los videojuegos fueron una consecuencia de un uso lúdico de estos aparatos electrónicos y sus reglas, fuente de curiosidad de investigadores y científicos. Desde su aparición, Tetris fue adoptado rápidamente como un videojuego icónico tanto en el ámbito de investigación científica como lúdico.

4.1. Historia

A principio de la década de los ochenta, Alex Pajitnov trabajaba en un laboratorio de cómputo para la Academia de Ciencias de la entonces Unión Soviética como investigador en el área de inteligencia artificial. En junio de 1984 Pajitnov, impulsado por su gusto a los rompecabezas[1] programó un conjunto de instrucciones que dieron como bases las reglas del juego que llamó Tetris.

Por políticas de su gobierno y temiendo represalias debido a que programar juegos no era parte de su trabajo, Pajitnov decidió no publicar su juego, sin embargo, el código de Tetris se filtró hacia Hungría y poco a poco abrió su paso hasta Estados Unidos, donde fue publicado, comercializado y vendido a millones de jugadores de videojuegos.

Para 1989 al menos seis compañías reclamaban los derechos del juego Tetris para consolas de videojuegos, computadoras personales y equipos portátiles. Con la eventual caída del bloque Soviético los derechos legales sobre Tetris, lentamente regresaron a Pajitnov para comercializar su creación [34].

Actualmente Tetris es uno de los videojuegos más vendidos de la historia con una estimación de alrededor de 170 millones[2] de copias vendidas y muchas variantes alrededor del mundo [37], siendo la consola Game Boy, de la compañía Nintendo, su primer distribuidor mayoritario [36].

Desde el punto de vista matemático y computacional, Tetris ha planteado muchas preguntas y planteamientos, como la posibilidad de jugar de manera infinita sin perder[3] o las combinaciones de movimientos que posee un jugador. Tetris ha sido objeto de estudio por diferentes campos como matemáticas, teoría de la computación, teoría de algoritmos, psicología [7], [8], entre otros.

[1] En particular por el juego de figuras llamadas pentominó.
[2] Copias físicas y digitales.
[3] La pregunta *¿sería posible jugar Tetris por siempre?* fue enunciada en [8] y la imposibilidad bosquejada.

Figura 4.1: Alex Pajitnov, diseñador y creador de Tetris, en la final de la copa mundial 2013 de *Image Microsoft*, en San Petesburgo, Rusia [66].

4.2. Definición del problema

Limitado por los gráficos de una computadora soviética llamada Electronika 60, Pajitnov tomó la decisión de bajar la complejidad del juego de pentominó de 18 piezas, a uno de tetraminó de 7. Adicionalmente agregó lógica al juego como la posibilidad de desaparecer líneas de piezas al ser completadas horizontalmente, creando un juego simple y fácil de entender [34], modificando así el juego original.

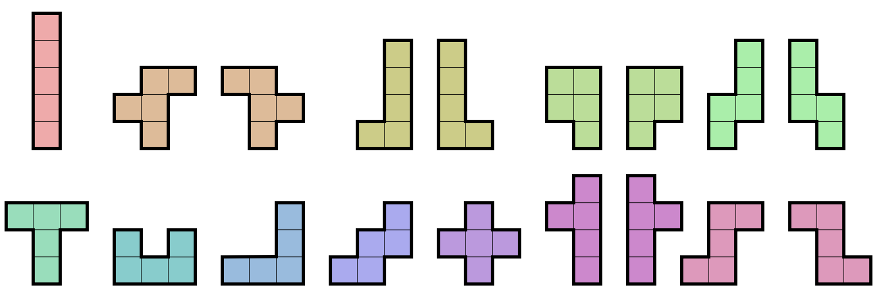

Figura 4.2: 18 piezas del pentominó, juego original en el que se basó Alex Pajitnov que consiste en tratar de acomodar las 18 piezas en un tablero, sin dejar espacios [65].

La versión original del juego programada por Pajitnov hace uso de un tablero de 10 unidades de ancho por 20 unidades de alto. En muchas versiones de Tetris posteriores el tablero puede llegar a tener $n \times m$ unidades. Cuando el juego empieza, el tablero se encuentra vacío; inmediatamente después la primera de las *piezas*, que se les denomina tetraminós, empiezan a aparecer en la parte superior del tablero.

Los tetraminós son un grupos de piezas geométricas conformada por cuatro unidades cuadradas que se definen como *celdas*. Cada celda ocupa exactamente una unidad vacía del tablero y no pueden existir dos celdas en mismo punto $(i \times j)$[4]. Después de aparecer en la pantalla, las piezas bajan fila por fila, de manera pausada hasta llegar al fondo del tablero o a un punto en el que ya no pueda

[4] A este punto también se puede ver como el espacio (i, j) o la columna j, con la fila i.

bajar más debido a que se sobrepondría a alguna otra pieza. Al ya no poder bajar más, el tetraminó actual se mantiene en esa posición mientras un nuevo tetraminó, seleccionado aleatoriamente dentro del grupo de siete posibles figuras (ver figura 4.3) aparece en la parte medio superior del tablero para ser de nuevo colocada en el tablero en alguna posición inferior.

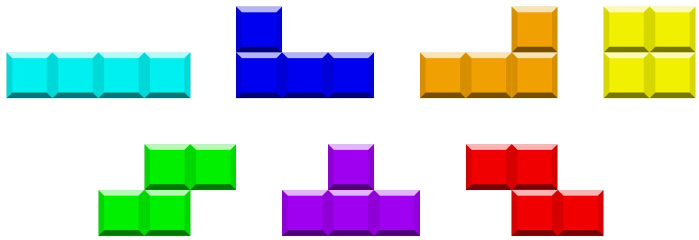

Figura 4.3: Los siete posibles tetrominós [64].

Los jugadores tienen a su disponibilidad la acción de rotar los tetraminós para orientar las piezas mientras éstas caen. Otra acción que puede realizar el jugador es mover las piezas a la columna derecha o izquierda y forzar a la pieza caer un nivel. Los jugadores poseen un tiempo limitado para realizar estas acciones antes de que la pieza sea empujada por la *gravedad* simulada del juego a una fila inferior.

La mayoría de las implementaciones tienen típicamente un recuadro donde aparece la siguiente pieza a jugar; cuando la pieza i está siendo colocada en el tablero, la pieza $(i+1)$ es revelada en el cuadro. A ese cuadro se le llama *cuadro de pista* [15].

Cuando toda una fila se encuentra llena de celdas de los tetrominós, las celdas de los tetrominós en dicha fila son removidas del tablero y las celdas de las piezas en filas superiores son desplazadas hacia niveles inferiores para llenar el espacio dejado por la fila removida. Un *tetris* ocurre cuando cuatro filas son removidas al mismo tiempo. Si los jugadores no remueven celdas lo suficientemente rápido, el tablero del juego se quedará sin espacio para seguir colocando piezas y la partida terminará [8].

Debido a los resultados obtenidos por [8], se sabe que es imposible ganar el juego de Tetris, por lo que el objetivo principal del jugador es maximizar la cantidad de puntos que se van acumulando durante la partida. Estos puntos varían dependiendo de las acciones del jugador y la interacción de las fichas en el tablero (como la desaparición de fichas o realizar un *tetris*).

4.3. Un problema *NP*-completo

En el año 2008, un equipo de computólogos publicó un artículo donde demuestran que programar una computadora para que juegue a Tetris, es un problema *NP*-completo. El objetivo de esta sección será enunciar, explicar los pasos y procedimientos que la publicación [15] usó, para demostrar la *NP*-completez del juego de Tetris. El resultado de esta sección justifica la implementación de la heurística al problema que aborda este trabajo.

Para comenzar con la demostración, el equipo conformado por Erik D. Demaine, Susan Hohenberger y David Liben-Nowell definieron una versión un poco diferente al videojuego. El cambio más importante de la versión que se presenta en el artículo es la clasificación de dos posibles flujos de información que pueden existir: la versión que llaman *offline* es aquella en la cual la entrada del programa contiene la lista de las piezas de forma determinista, finita y ordenada que existirán hasta un tiempo T. La versión *online* es aquella en la que la información de las piezas a jugar en un tiempo T es sólo la pieza P_T y, a lo más, el tetrominó P_{T+1} dentro del cuadro de pista.

Aunque la demostración del problema es sobre la versión del juego *offline*, se supone en el artículo que la versión del juego *online* es al menos (hablando de complejidad computacional) tan difícil como la versión *offline* [15] [5]. Mencionan los autores que es intuitivamente más fácil hacer jugar a la computadora con el conocimiento total de los movimientos posteriores que debe realizar, así que la conclusión de la *NP*-completez de la versión *offline* indica la complejidad de hacer jugar la versión *online*.

Existen cuatro objetivos del juego que son demostrados, tratar de optimizarlos es un problema *NP*-completo:

- Maximizar el número de filas removidas mientras se juega alguna secuencia.
- Maximizar el número de piezas antes de que el juego se termine.
- Maximizar el número de veces que se hace un *tetris*.
- Minimizar la altura de la columna más alta.

Para demostrar que los objetivos (y el juego) es *NP*-completo, primero demuestran que el problema formal del juego, `TETRIS` es elemento de la clase *NP*; luego que la maximización del número de filas removidas es un problema *NP*-duro; los demás puntos los demuestran reduciéndolos entre ellos. La prueba inicial de la pertenencia a *NP*-duro que hacen los autores incluye una reducción a partir del problema visto en el apartado 2.2.

4.3.1. Formalización del juego

Las reglas de Tetris son definidas rigurosamente con el fin de transparentar la reducción y demostraciones hechas con las operaciones y propiedades del juego. El modelo formal propuesto consiste en los siguientes agentes y será respetado en su mayoría durante la implementación de este trabajo:

Tablero. El tablero es una matriz de n filas por m columnas enumeradas de abajo hacia arriba y de izquierda a derecha. La casilla $\langle i, j\rangle$ del tablero puede estar en dos estados: *ocupada* o *libre*. En un tablero válido no existen filas c tal que $\langle c, k\rangle$ con $k \in \{0, 1, ..., m\}$ tenga *ocupadas* a todas sus casillas. Tampoco existen filas completamente vacías que se encuentren por debajo de alguna casilla ocupada.

Piezas. Los ya definidos tetraminós como los mostrados en la figura 4.3. Cada tetraminó tendrá ahora la estructura de la forma $P = \langle t, o, \langle i, j\rangle, f\rangle$ donde cada elemento es respectivamente:

1. Un tipo de pieza. De izquierda a derecha en la figura 4.3, el tipo de pieza sería: `I`, `RS`, `LG`, `T`, `RG`, `LS` y `Sq`.
2. Una orientación dada en grados: $0°, 90°, 180°$ o $270°$.
3. Una posición $\langle m_c, k_c\rangle$ donde se encuentre la casilla centro de la pieza.
4. Un valor de `FIJO` o `MOVIBLE`.

Adicionalmente, cada tipo de pieza posee un centro. Se seleccionará una casilla y esa casilla será considerada el centro de la pieza al ser rotada.

En el estado inicial, las piezas se encuentran en una orientación $0°$, la posición inicial es la $\langle m, \lfloor n/2\rfloor\rangle$ y la pieza tiene el valor de `MOVIBLE`.

Rotación. Un modelo de rotación que es una función computable $R : \langle P, \theta, B\rangle \mapsto P'$, donde P y P' son orientaciones de las piezas, $\theta \in \{-90°, 90°\}$ es el ángulo de rotación y B es el tablero. El artículo define las siguientes condiciones para R:

1. Si $P = \langle t, o, \langle i, j\rangle, f\rangle$ y la rotación es válida, entonces $P' = \langle t, (o+\theta) \mod 360°, \langle i, j\rangle, f\rangle$ para algún $\langle i, j\rangle$. Si la rotación no es válida, entonces $P' = P$.
2. Para determinar la validez de una rotación, R sólo necesita examinar una vecindad de tamaño $O(1)$ de la pieza P.
3. Si todos las casillas de la vecindad de P están vacías, se dice que la rotación es válida o legal.
4. Si la rotación es legal, P' no debe ocupar ninguna casilla ya ocupada por algún otro tetrominó en B.

Reglas del juego. La única regla para las fichas con estado `FIJO` es que no existen movimientos válidos para ésta. Las piezas de la forma $P = \langle t, o, \langle i, j\rangle, \texttt{MOVIBLE}\rangle$ en un tablero B, tienen el siguiente conjunto de *movimientos* [5] disponibles:

1. Rotación en dirección a las manecillas del reloj. $R(P, 90°, B)$.
2. Rotación contraria a las manecillas del reloj. $R(P, -90°, B)$.
3. Desplazamiento a la izquierda. $P' = \langle t, o, \langle i-1, j\rangle, \texttt{MOVIBLE}\rangle$.
4. Desplazamiento a la derecha. $P' = \langle t, o, \langle i+1, j\rangle, \texttt{MOVIBLE}\rangle$.
5. Deja caer. $P' = \langle t, o, \langle i, j-1\rangle, \texttt{MOVIBLE}\rangle$.
6. Asignar estado. Si existe al menos una casilla ocupada debajo de P, entonces $P' = \langle t, o, \langle i, j\rangle, \texttt{FIJO}\rangle$.

Para las reglas del tablero, se define una trayectoria σ de una pieza P. La trayectoria es una secuencia de movimientos válidos del estado inicial de la pieza, hasta la asignación del estado `FIJO` de P. El resultado de una trayectoria sobre el tablero B, es un tablero nuevo B' definido con las siguientes características:

1. El tablero nuevo B' es inicialmente B con la figura P.
2. Si P tiene el estado de `FIJO` y para alguna fila r, cada casilla de r está ocupada en B', las celdas de los tetrominós en r son removidos. Para cada $r' \geq r$, se reemplaza a la fila r' en B' con la fila $r'+1$ de B'. Múltiples filas pueden ser removidas con una sola trayectoria.
3. Si existe alguna casilla ocupada en B' donde debería estar la siguiente pieza en su estado inicial, el jugador pierde.

Para un juego $\langle B_0, P_1, ..., P_p\rangle$, una secuencia de trayectorias Σ es una secuencia $B_0, \sigma_1, B_1, ..., \sigma_p, B_p$ tal que para cada i, la trayectoria de la pieza P_i aplicado al tablero B_{i-1}, genera el tablero B_i.

Problema formal Aunque el artículo contempla varios objetivos previamente mencionados a optimizar, el problema de decisión con el objetivo Φ del juego de Tetris, $\texttt{TETRIS}[\Phi]$, es enunciado formalmente como sigue:

- **Input:** Un juego de Tetris de la forma $\mathcal{G} = \langle B, P_1, P_2, ..., P_p\rangle$.
- **Output:** ¿Existe la secuencia de trayectorias Σ tal que $\Phi(\mathcal{G}, \Sigma)$ no resulte en una partida perdida?

Una vez definido minuciosamente el modelo formal de Tetris, incluyendo las reglas y los objetos que participan en una partida, se procede a discutir su complejidad.

[5] Llámese "movimiento" a un elemento del conjunto de acciones posibles del jugador, en un límite de tiempo.

4.3.2. La clasificación de TETRIS

Se sabe por las definiciones del apartado 2.1, que para que `TETRIS` esté en *NP*-completo, debe cumplir con que `TETRIS` $\in NP$-duro y `TETRIS` $\in NP$.

Para llegar a la conclusión de la *NP*-completez, los autores tuvieron que proponer y demostrar ciertos lemas y teoremas enunciados a continuación:

Teorema 2.1 Para cada objetivo *acíclico*[6] comprobable Φ, se tiene que `TETRIS`$[\Phi] \in NP$.

En el artículo se da un algoritmo *NP* para `TETRIS`$[\Phi]$. Los autores argumentan que dado una Σ aleatoria, se puede verificar que $\Phi(\mathcal{G}, \Sigma)$, en tiempo polinomial[7] ya que también se puede verificar `POLY`$(|\mathcal{G}|, |\Sigma|) =$ `POLY`$(|\mathcal{G}|)$ debido a que cada de las p trayectorias en Σ contiene a lo más $4 \cdot |B| + 1$ estados a verificar, $|B| = n \times m$ con n y m el número de filas y columnas de cada tablero.

Lema 2.2 El objetivo *k-filas-removidas*[8] es `POLY` verificable y acíclico.

La corta demostración de este lema se realiza mediante la argumentación de que `k-filas-removidas` es acíclico porque sólo depende del estado de cada pieza que es colocada al final de cada trayectoria en el tablero. Es `POLY` verificable ya que sólo es necesario recorrer cada tablero que regrese cada trayectoria en a lo más $O(m \cdot n \cdot |\Sigma|)$.

Teorema 3.1 `3-PARTITION` es un problema *NP*-completo.

Este teorema es discutido en la apartado 2.2 y demostrado en [21].

Teorema 3.2 El juego $\mathcal{G}(\mathcal{P})$ es polinomial respecto a $\mathcal{P}$.

Para explicar este teorema, primero hay que explicar la decisión que tomaron Demaine, Hohenberger y Liben-Nowell en su artículo para el problema de reducción:

Los autores decidieron crear la reducción del problema `3-PARTITION` debido a su propiedad de ser fuertemente *NP*-completo; esto incluye valores de entrada a_i y T unitarios. También de enfocan en instancias específicas de `3-PARTITION`:

1. Para cada conjunto $S \subseteq \{a_1, ..., a_{3s}\}$, si $\sum_{a_i \in S} a_i = T$ entonces $|S| = 3$.
2. T es par.
3. Si $\sum_{a_i \in A_j} a_i \neq T$ entonces $\left|T - \sum_{a_i \in A_j} a_i\right| \geq 3s$.

Dada una instancia $\mathcal{P} = \langle a_1, ..., a_{3s}, T\rangle$ que cumpla las tres condiciones de arriba, $\mathcal{G}(\mathcal{P})$ es un juego de Tetris que elimina a todas las piezas si $\mathcal{P}$ tiene como respuesta un valor booleano de `TRUE`.

El tamaño del tablero y el número de piezas dependerá de los valores S y T del problema de la `3-PARTITION`: el tamaño del tablero resultante de $\mathcal{G}(\mathcal{P})$ es de $6T + 22 + 3s + O(1)$ filas por $6s + 3$ columnas, mientras que el número de piezas totales del juego será de:

$$\sum_{i=1}^{3s} [3 + 5a_i + 2] + s + 1 + \left(\frac{3T}{2} + 5\right) = 16s + 5sT + \frac{3T}{2} + 6.$$

[6]Se dice que una función objetivo Φ es acíclica si para todo juego $\mathcal{G}$ en los que exista una trayectoria Σ tal que $\Phi(\mathcal{G}, \Sigma)$ no pierde, entonces existe una trayectoria Σ' tal que $\Phi(\mathcal{G}, \Sigma')$ no pierde y no existen estados repetidos de las piezas en Σ'.

[7]Se nombrará de manera genérica a la función de verificación en tiempo polinomial, función `POLY`.

[8]`k-filas-removidas` $: \mathcal{G} \times \Sigma \to \mathbb{N}$ nos regresa el número de filas removidas durante un juego de Tetris.

Los valores a_i y T están representados como valores unarios (por la naturaleza del sistema de conteo de casillas en el tablero) por lo que se puede suponer siempre una transformación polinomial a la hora de construir el juego conforme a la opinión de los autores de la demostración.

Teorema 4.1 (Completez) Para cada instancia $\mathcal{P}$ que tenga como respuesta un valor booleano de `TRUE` del problema de la `3-PARTITION`, existe una secuencia de trayectorias Σ que *limpie*[9] el tablero $\mathcal{G}(\mathcal{P})$ sin que el juego termine, o el jugador pierda.

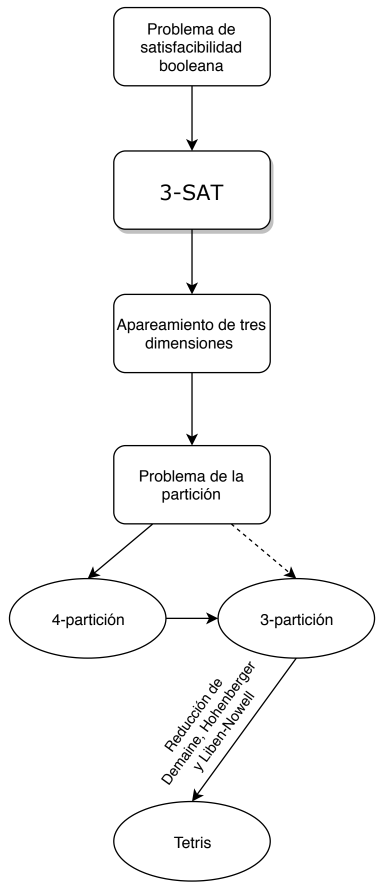

Figura 4.4: Extensión del diagrama de la figura 2.1 de las reducciones polinomiales desde `SAT` hasta Tetris con $\mathcal{G}(\mathcal{P})$.

La forma que proceden en esta demostración es partiendo por la condición de veracidad de `3-PARTITION`; aprovechan la partición de los conjuntos A_i para asociar a los tetrominós con algún número, acomodarlos en algo que los autores llaman *cubetas*, que son divisiones del tablero que se le asigna a cada A_i y así, asegurar no dejar espacios casillas sin llenar. De manera bastante ilustrativa construyen estas secuencia de piezas o configuraciones donde se aseguran no existan huecos en las filas, produciendo así su *limpieza* y remoción del tablero.

[9]En este contexto, limpiar el tablero tiene el significado de mantener el juego sin piezas, es decir, que todas las piezas jugadas sean en algún punto removidas del tablero.

Dada la reducción propuesta `3-PARTITION` $\leq_{\mathcal{G}(\mathcal{P})}$ `TETRIS`, la demostración dentro del artículo y las definiciones de [1] en la apartado 2.1, se puede afirmar que `TETRIS` $\in$ *NP*-duro.

Lema 5.10 El modelo de rotación propuesto es suficiente.

Cubriendo todos los posibles casos con los siete tetraminós, los autores muestran que no existe combinación de rotaciones en el que las piezas puedan (o piezas diferentes a ellas no puedan) llenar ciertos espacios del tablero.

Teorema 5.16 Si existe una estrategia válida para $\mathcal{G}(\mathcal{P})$, entonces $\mathcal{P}$ es una instancia de `3-PARTITION` que tiene respuesta un valor booleano de `TRUE`.

El artículo, va un poco más allá y plantea, discute y demuestra la propiedad de que la robustez o exactitud[10], que significa que cualquier solución encontrada sea correcta.

Inmediatamente después de esta última demostración, el artículo enuncia como consecuencia a todos los teoremas y lemas enunciados arriba, el siguiente teorema:

Teorema 5.17 `TETRIS`[*k-filas-removidas*] es *NP*-completo.

Habría que recordar la definición que se enunció en el apartado 2.1; dice que para que un problema esté dentro de la clase *NP*-completo, debe cumplir que esté tanto en *NP* y en *NP*-duro. Estas propiedades se demuestran en el **Teorema 3.1**. La parte de pertenencia a *NP*-duro se construye por el **Teorema 4.1**, usando la reducción, los teoremas 2.1, 3.1, 3.2 y 5.16 y los lemas 2.2 y 2.10 se concluye que `TETRIS`[*k-filas-removidas*] $\in$ *NP*-completo.

Si bien, el artículo no termina con este teorema y va más allá, demostrando la complejidad de cada uno de los objetivos a optimizar, para propósitos de este trabajo, es suficiente este resultado por lo que en los capítulos posteriores se habla de herramientas técnicas y la implementación del problema.

[10] Viene del inglés *soundness*.

Capítulo 5

Tecnologías utilizadas

Aunque el concepto de complejidad de un algoritmo, clases de complejidad y eficiencia mantienen un estado invariante al entorno de desarrollo de los programas, el análisis de la experimentación está inevitablemente comprometido por el desempeño del hardware y software que se usa. Para darle sentido a los resultados, es muy importante conocer las herramientas usadas, así como el ambiente en el que los experimentos fueron realizados.

5.1. Python 3.5

En la década de los ochenta, en el centro Wiskunde de informática, en Ámsterdam, Guido van Rossum trabajaba como desarrollador de un lenguaje de programación llamado *ABC* cuyo propósito era ser una herramienta de desarrollo fácil de aprender para personas no acostumbradas a programar. Rossum vio la necesidad de crear un lenguaje de *scripting*[1] sobre su proyecto en el lenguaje ABC, por lo que creó una máquina virtual, un programa *parser* y otro de ejecución de comandos; agregó una sintaxis básica, usó indentación para definir bloques y creó un pequeño conjunto de tipos de datos. Así nació Python [51].

Python es un lenguaje de programación de propósito general, Turing completo, dinámicamente tipificado y con influencia de múltiples paradigmas; incluidos pero no limitado a orientación a objetos, funcional e imperativo [50]. Una de las principales características por las que Python es reconocido, es por su manera de definir bloques de código: la indentación de un programa en Python es fundamental para la semántica de un programa.

Python se encuentra desde el 2003 como uno de los lenguajes de programación más populares de acuerdo al índice de clasificación de la comunidad programadora TIOBE [59]. Cientos de compañías como Google, Yahoo, Disney Animation, NASA, IBM, usan Python como lenguaje de desarrollo para resolver problemas en diversas áreas [52].

Actualmente existen muchas versiones de Python; las más usadas son las versiones 2.7 y 3.5 que en conjunto con el administrador de paquetes[2] PIP, proveen de cientos de bibliotecas listas para usar e implementar soluciones a problemáticas de diversa índole [50].

Otra de las ventajas más grande que posee Python sobre otros lenguajes de alto nivel, es su ambiente de desarrollo y ejecución; Python posee como herramienta la creación de sus propios ambientes virtuales con sus propio sistema de archivos aislados e independientes al sistema local. La

[1]En algunos textos referidos como *Guiones del intérprete de comandos*, son programas en las que sus órdenes son ejecutadas de manera secuencial [35].

[2]Un administrador de paquetes es un programa que tiene como propósito la instalación, actualización y configuración de paquetes de software.

ventaja principal de mantener un ambiente de desarrollo independiente, es la instalación y manejos de paquetes en un sistema donde aquellos que poseen otros ambientes virtuales o el mismo ambiente local no interfiera ni genere conflictos por paquetes previamente instalados o por versiones diferentes [63].

En los últimos años Python en su desarrollo se ha orientado a mejorar la eficiencia de tiempo de respuesta de sus llamadas a sistema; la forma en la que Python mejora su velocidad aún siendo un lenguaje que típicamente usa un intérprete, es generando archivos con extensión `.pyc`, que son archivos en lenguaje máquina, los cuales contienen funciones e instrucciones previamente ejecutadas y optimizadas para su posterior uso repetitivo [50]. De cualquier manera, Python sigue teniendo algunos problemas para escalar a sistemas grandes y complejos.

5.1.1. pygame

Una de las razones por las que Python se hizo altamente reconocido, fue por la facilidad de hacer uso de bibliotecas externas al núcleo del lenguaje. Existen una gran cantidad de bibliotecas muy conocidas y usadas mundialmente, ya sea por la facilidad que proveen o por lo conveniente que resultan sus soluciones. Un ejemplo es la biblioteca Numpy que posee un conjunto de funciones de uso científico ampliamente reconocido por la comunidad investigadora, por ser de gran utilidad en numerosos proyectos [47].

La biblioteca pygame es una biblioteca de Python, de código abierto que sirve para realizar aplicaciones multimedia como animaciones o videojuegos; existen alternativas ya programadas que tiene como propósito ejecutar videojuegos publicados originalmente para la consola *Nintendo Entertainment System* o *NES*, sin embargo, las opciones consideradas presentan la inconveniencia de no poder modificar el tablero a voluntad de manera directa y simplificada. [41] [67].

Si bien el propósito de las bibliotecas pygame puede ser la creación de videojuegos donde la optimización de gráficos y ejecución requiere el mayor cuidado, sólo se usará como una herramienta de visualización de resultados debido a la naturaleza de la heurística de colonia de abejas artificiales [49].

5.2. Git

Durante el desarrollo de cualquier proyecto de software existen muchas herramientas útiles para mejorar y hacer más eficiente el flujo de trabajo. Un ejemplo de software útil son los controladores de versiones, también llamados *versionadores*. Los controladores de versiones son sistemas que mantienen un registro (en un lugar llamado repositorio) de todos los cambios realizados a un conjunto de archivos rastreables, produciendo la oportunidad de observar un historial de modificaciones realizadas en un periodo de tiempo [25]. Existen dos categorías principales de controlador de versiones: los versionadores de repositorios centralizados que son aquellos en los que los archivos que están siendo rastreados se encuentran en un repositorio central, único, que todos los desarrolladores modifican; y los versionadores de repositorios distribuidos que mantienen muchas copias del código en distintas computadoras e implementan un sistema de comunicación de cambios mediante un historial. De este último tipo es uno de los versionadores más usados: Git [56].

Git fue desarrollado en el 2005 por el mismo creador de uno de los proyectos más grandes de software libre que existe; el proyecto Linux necesitaba de un controlador de versiones debido a su particularidad de ser un sistema operativo que es modificado por miles de personas alrededor del mundo. Linus Torvalds consideró necesario la creación de Git como herramienta rápida, segura y que soportara desarrollo de código de manera no lineal, para mantener una buena comunicación de desarrollo del proyecto Linux [26] [27].

Si bien el uso básico de Git es sencillo, dominarlo puede llegar a ser un trabajo de años de experiencia. Un programador sin experiencia en la herramienta puede comenzar a realizar seguimiento y registro de cambios en el sistema de archivos con un par de comandos (`git init`, `git add -A`, `git commit` en particular).

Los cambios son guardados dentro de un árbol de registros donde cada rama del árbol pertenecerá a una secuencia de historias. Cada cambio agregado y almacenado genera una huella digital única llamada *hash*. Este *hash* único es el identificador del espacio temporal del historial donde se almacena la información I en el tiempo T [25].

Una de las conveniencias de usar Git como herramienta para proyectos de desarrollo de software es poder regresar a algún punto en una versión del historial. Es conveniente conocer el *hash* con el comando `git log` y familiarizarse con la orden `git revert`.

Es importante aclarar que el uso de Git va más allá de mantener un registro y poder mover el estado actual del proyecto por uno de sus estados almacenados; Git posee características que benefician al trabajo en equipo, depuración de errores, pruebas y mantenimiento, desarrollo remoto y varias características que lo han hecho uno de los programas con mayor número de usuarios y de traducciones: Git existe en doce idiomas y más de seis traducciones parciales.

Durante el desarrollo de este proyecto se usará Git de manera constante para llevar un registro de todos los cambios hechos en el código de la implementación de la heurística.

5.3. Ambiente físico

Invariablemente los resultados obtenidos pueden ser muy diferentes dependiendo el equipo en el que se prueben así como su sistema operativo y su versión correspondiente. El hardware es mejorado con notoriedad cada poco tiempo de acuerdo a las observaciones como las de Mark Kryder o la reconocida ley de Moore, por lo que no es de extrañarse que con el paso de tiempo los resultados parecieran mejorar. El software también sufre de mejoras; algoritmos de funciones de los sistemas operativos son actualizados constantemente para garantizar el mayor rendimiento posible.

El desarrollo, pruebas y análisis fueron realizadas en una computadora ASUS, ZenBook Pro 14 modelo UX450FDX. La velocidad de comunicación de la memoria RAM, 8 GB DDR4 Synchronous es de 2,400 MHz (a 0.4 ns). Los tres niveles de cache (L1, L2 y L3) poseen respectivamente 256 KB, 1 MB y 6 MB. El procesador del ambiente de ejecución es de la marca Intel(R) Core(TM), modelo i5-8265U, x86_64, 8 núcleos de dos hilos cada uno y corre a una velocidad promedio máxima de 3.9 GHz.

El sistema operativo instalado en la computadora es una distribución de GNU/Linux Debian, con una versión actualizada del kernel 4.19.0-0.bpo.4-amd64, que corre sobre la versión de la arquitectura de 64 bits. La versión del lenguaje de programación usada también es relevante debido a que los lenguajes de programación son constantemente modificados y mejorados. El ambiente de desarrollo que se usa es la versión de Python 3.5.3.

Capítulo 6

Análisis y diseño de la implementación

Un buen diseño de software apegado a buenas prácticas de programación contribuyen a la obtención de resultados de calidad, menos errores durante la implementación y un entendimiento más sencillo para terceros. Durante este capítulo, se analizan las decisiones de la implementación y el motivo del diseño. Todas las clases y funciones discutidas a continuación se encuentran en el apéndice C y en la liga `https://github.com/ricardorodab/abc-tetris`.

6.1. Análisis del sistema

Para encontrar soluciones a la colocación de las piezas de Tetris, primero se tiene que partir del poder acceder a toda la información del juego para el análisis de los desenlaces. En el apartado 4.3.1 se realiza la formalización del juego y con ella las reglas a seguir durante la implementación:

1. Se debe diseñar un tablero que almacene la información del conjunto de piezas.
2. Se debe tener piezas que posean un tipo específico, posición y orientación.
3. Se debe programar un modelo de rotación específico para las piezas de acuerdo a su tipo.
4. El tablero, piezas y modelo de rotación deben estar regidos en todo momento por las reglas del juego.

El objeto que contenga los entes principales del juego, deben también tener métodos de obtención de información relevantes a la heurística como datos que sirvan para generar una función de costo y métodos para forzar ciertos comportamientos indicados por un objeto externo (como lo es la misma heurística en este caso) para pasar de un estado del juego a un estado siguiente.

Una regla adicional necesaria para la utilización de cualquier conjunto de reglas a optimizar es que todo objeto que reciba la heurística debe ser "clonable", esto quiere decir que exista un método o función la cual regrese un objeto idéntico al que lo llama y que la heurística tenga la libertad de modificar sin afectar el estado del objeto clonado original.

Del lado de la heurística el elemento más importante a considerar es el comportamiento de la colmena como un conjunto de funciones que realizan las abejas que habitan en ella en un periodo de tiempo. Cada abeja creada dentro de la colmena tiene la tarea de explorar y trabajar o de observar. La necesidad de dividir a las abejas de esta manera se puede entender en el artículo [39] donde el autor de la heurística, Karaboga, realiza pruebas que concluyen en que tener el 50 % de abejas observadoras incrementa el néctar de sus soluciones.

La ejecución del código ocurre de forma modulada, equivalente al modelo de *pipelines* usado por Henry Ford. Durante una iteración de la colmena se deben realizar las siguientes acciones:

1. Si la colmena no tiene abejas, inicializarlas.
2. Mandar a llamar al método que libera a las exploradoras y las transforma en trabajadoras asociadas a una fuente.
3. Mandar a llamar a las abejas trabajadoras para que recolecten néctar de su fuente y generen información para hacer el *waggle-dance*.
4. Hacer que las abejas observadoras escojan una fuente y ejecutar las funciones implementadas para la observación de cada fuente vecina.

El problema específico a resolver no debe afectar el comportamiento de la heurística por lo que se puede aprovechar las propiedades de lenguajes en el paradigma funcional que ofrece Python y asignar funciones como parámetros y atributos para que la ejecución de las abejas sea independiente al problema y sean los atributos de cada abeja los llamados para evaluar las fuentes.

Un último paso para poder operar con la heurística y el juego es el conjunto de métodos que realizan la comunicación entre ambos; los resultados se deben unir de alguna manera ya que el resultado de cada iteración es la entrada de la siguiente. Las funciones que las abejas esperan y necesitan para realizar su trabajo dentro de la heurística también es necesario comunicarlo a la hora de instanciar la colmena.

Una buena práctica es que los parámetros de experimentación no sean modificados directamente en el código sino que se deben encontrar como entrada, en algún archivo de configuración para su fácil modificación y preservar así un buen diseño. Es conveniente implementar métodos asociados al manejo del archivo de entrada así como un objeto que contenga siempre estos parámetros desde el principio de la ejecución del programa.

6.1.1. Abejas observadoras

Las abejas observadoras tienen un nivel de dificultad un poco mayor de diseño e implementación debido a la naturaleza de las acciones que toman. Una abeja observadora primero le es asignada una fuente i de las ya trabajadas con probabilidad

$$P_i = \frac{F(\theta_i)}{\sum_{k=1}^{n} F(\theta_k)}.$$

Se debe considerar el caso donde después de recorrer todas las fuentes, la abeja observadora se queda sin alguna asignación de fuente. Una abeja observadora no se debe quedar sin fuente por lo que de existir un abeja en este caso se debe volver a iterar sobre todas las fuentes hasta que le sea asignada. Una solución para evitar este problema del no determinismo a la hora de asignar parejas, es duplicar la probabilidad de asignación de todas las soluciones por cada iteración para evitar que se recorra demasiada veces la lista de fuentes.

Un problema aún mayor es la localización de fuentes vecinas. En el caso particular de Tetris, se necesita alguna manera de conseguir una fuente vecina a un punto del tablero. La definición de fuente vecina es

$$\theta_i(c+1) = \theta_i(c) \pm \phi_i(c)$$

donde $\theta_i(c)$ es la solución i-ésima. El primer enfoque resolutivo de este problema fue el jugar movimientos aleatorios posteriores a una fuente, sin embargo dichos movimientos serían correspondientes a un punto futuro de la partida por lo que no se considera una fuente vecina sino la fuente i explotada como lo haría una abeja trabajadora. Una forma en la que $\phi_i(c)$ es encontrada es mantener un registro de movimientos que se jugaron para llegar a $\theta_i(c)$ y poder cambiar algunos movimientos previos con ayuda de un *historial*.

El historial no es más que una lista de movimientos L_θ que se han hecho hasta un tiempo T. Es posible eliminar un número aleatorio δ_1 de movimientos para posteriormente agregar en la lista otro número de movimientos aleatorios δ_2 hasta llegar a T con una lista de movimientos L_ϕ. Los tiempos T fueron seleccionados como los puntos en el que las piezas son fijadas, esto debido a que ese comportamiento es una invariante para todas las piezas del juego.

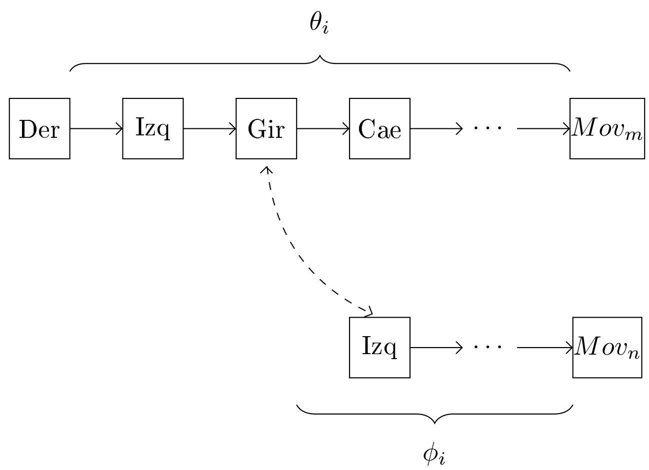

Figura 6.1: Historial de movimientos dentro de un juego de Tetris.

6.1.2. Función de costo

La función de costo o también conocida como función *fitness* es la encargada de calificar una solución propuesta por la colmena para su posterior uso. Si una función de costo es de baja calidad, los resultados serán de igual forma de baja calidad. Una buena función de costo deberá premiar con una mejor calificación a los tableros que resulten con una mejor estrategia de juego y deberá desanimar aquellos tableros que se consideren resultados de malas decisiones. Es necesario que por cada modo de juegos o solicitud a resolver exista una función de costo enfocada a resolver dicha solicitud como en el caso de Tetris y la heurística de abejas artificiales, la cual usa una función de costo llamada "néctar" que es de tipo *higher is better* (entre más grande el resultado, mejor solución).

Adicionalmente a la función para conseguir el néctar, la colmena necesita otras tres funciones para su correcto funcionamiento y ofrece una cuarta función completamente opcional para un comportamiento positerativo:

- **Función para buscar fuente:** Esta función es la que usarán las abejas exploradoras para entregar una fuente que habrá que explotar.

- **Función para explotar fuente:** Esta función es la que usarán las abejas trabajadoras para seguir explotando una fuente y modificar su estado.

- **Función de observación:** Esta función es la que usarán las abejas observadoras y, para la instancia de Tetris específica, es la función que eliminará movimientos en el historial para crear una nueva combinación de movimientos jugados.
- **Función opcional de término:** Esta función realiza una acción adicional sobre las fuentes al finalizar la iteración de la colmena, en el caso de Tetris limpia las filas si es que se deben limpiar.

6.2. Diseño del sistema

Para lograr el objetivo de crear un sistema que resuelva hacer operaciones con el problema de Tetris, operar con una heurística para encontrar soluciones y comunicarse entra ambos con una lógica sencilla pero manteniendo buenas prácticas de programación, se dividió en tres componentes principales para poder lograr apegarse al principio de orientación a objetos: la heurística ABC, el juegos de Tetris y la comunicación Abejas-Tetris.

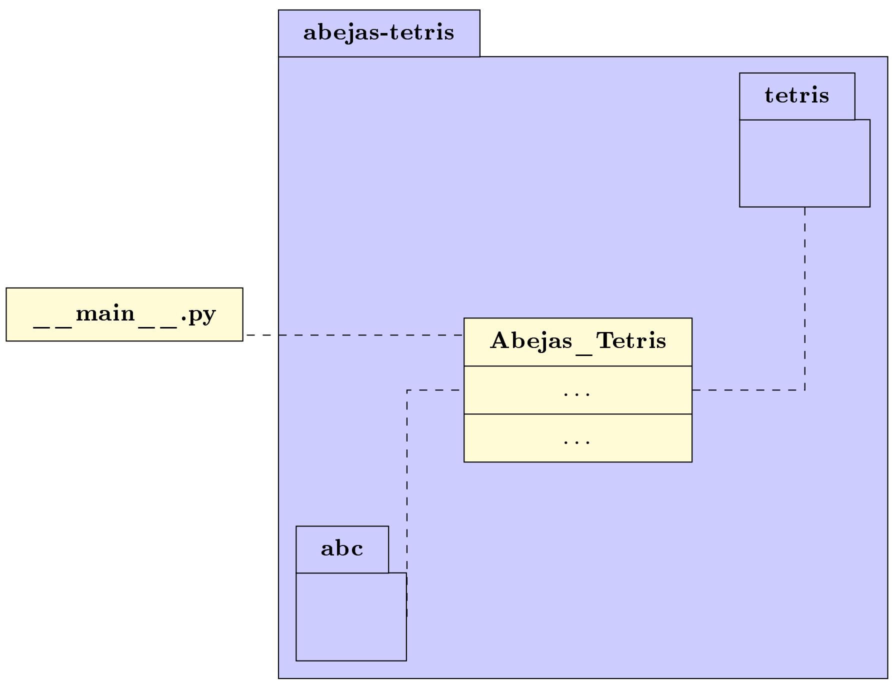

Figura 6.2: Estructura básica del sistema.

6.2.1. Orientación a Objetos

La decisión de realizar un diseño que siga los principios de la Orientación a Objetos (u OO) tiene como sustento la abstracción necesaria de los entes participantes de la heurística, que responderán a estímulos o mensajes recibidos mediante la ejecución de métodos o funciones del juego de Tetris.

Para un programador acostumbrado a la OO, el modelo propuesto por Dervis Karaboga [38] donde enuncia un conjunto de individuos participantes como son las abejas, colmena y fuentes, posee una relación bastante fácil de traducir a entes que tienen estructura, comportamiento e identidad (que son las componentes principales que todo objeto tiene en su descripción [20]) y que comparten un propósito definido en la heurística sin importar su comportamiento unitario.

La experiencia personal como estudiante de la carrera de Ciencias de la Computación en la Facultad de Ciencias, suma un conjunto de prácticas donde la abstracción a la hora de realizar la toma de decisión sobre las clases, atributos y métodos así como el lenguaje de programación, paradigmas y tecnologías son refinada con proyectos, clases y prácticas desde el primer semestre hasta el último de la estancia como alumno.

Por último, el paradigma orientado a objetos tiene como una de sus características principales más atractivas lo intuitivo de la descripción de objetos reales, con atributos que a su vez pueden ser objetos con sus propios atributos compuestos previamente definidos [44]. Por parte del juego de Tetris esta característica es muy atractiva debido a que el tablero está constituido por un cojunto de entes (como son las casillas, piezas, puntos) que habrá que definir a su vez y que en conjunto posee un comportamiento que dependerá del estado unitario así como la comunicación de todos los objetos dentro del juego.

6.2.2. Diseño de Tetris

Una partida de Tetris lleva varios agentes involucrados como lo son el tablero, que puede estar vacío o formado por piezas; y las piezas que están formadas por cuatro casillas y posee un identificador llamado *tipo*, que es alguna constante para indicar cuál de las siete posibles piezas es. Cada casilla que conforma la pieza posee un estado de movimiento que puede ser fijo o movible y un punto de la forma (x, y) que indica su posición en el tablero.

Se definieron cinco clases y dos enumeraciones para construir una lógica robusta y modular. La primera clase y la más sencilla es la clase `Punto`, conformado por dos números enteros, x y y. Cada objeto de la clase `Casilla` posee un objeto punto y un atributo llamado `_fija` que es una bandera para conocer el estado actual de esa casilla; si la bandera se encuentra en `True` significa que la casilla es parte de una pieza que se está jugando actualmente.

La clase `Pieza` por su parte posee un atributo del tipo `Tipo`, que es una enumeración con siete posibles valores que sirve para identificar a los tetrominós, una orientación de la forma $x = (90 \times k) \mod 360$, una posición que es un punto de la casilla principal de la pieza y una lista de cuatro elementos que contiene a las casillas.

El tablero del juego posee referencias a las casillas de las piezas en una matriz bidimensional de casillas, una pieza actual y una pieza anterior. Aún cuando el tablero guarda las casillas, el estado de las piezas dejan de importar cuando una nueva pieza es ingresada al tablero por lo que el objeto pieza es eventualmente eliminado por el recolector de basura. Es en el objeto tablero donde las dimensiones del juego son almacenadas así como la puntuación[1] del juego. Si bien la clase `Tablero` tiene definidos métodos que modifican el estado de una partida de Tetris, la lógica ha sido delegada a la clase `Tetris` que es la encargada de mandar a llamar a los métodos para modificar el juego.

La clase `Tetris` además de poseer un tablero, un historial de movimientos y un estado booleano llamado `_game_over` que indica si el juego ha finalizado, es la clase destinada a ser usada como interfaz para ser ejecutada por la heurística por lo que la lógica y datos del juego son modificados y obtenidos en su totalidad a través de sus métodos.

[1]Se toma como puntuación al número de filas removidas durante una partida de Tetris.

Figura 6.3: Diagramas UML del paquete `tetris`

6.2.3. Diseño de la heurística ABC

Para un buen desempeño de la heurística y que funcione de acuerdo a la descripción de Karaboga, una colonia de abejas artificiales constituyen una colmena válida cuando existen tres tipos de abejas: las abejas exploradoras, las abejas empleadas o trabajadoras y las abejas observadoras. El tipo de cada abeja es independiente y puede ser representado por una constante, es por ello que se decidió realizar una enumeración con los tipos de abejas en la clase `Tipo_Abeja`.

La clase, `Abeja` si bien sencilla y relativamente corta, está construida pensando en que el problema que la heurística resuelva no sea exclusivamente el juego de Tetris sino algo más general. Las abejas tienen un tipo, una fuente que es el punto de partida de cada abeja y es `None` al principio de la creación de cada objeto, una variable `_limite` que indica el número de veces que una abeja puede visitar a una fuente, una variable `_delta` que le sirve a las abejas observadoras para saber cuánto “alejarse” de la fuente original.

Las abejas tienen además cuatro funciones que será explicada su implementación en el siguiente capítulo debido a la particularidad de que su definición tiene un impacto directo sobre los resultados. Cada función será usada dependiendo el rol que cada abeja tenga que cumplir en la colmena:

- `_busca_fuente`: Dado un estado inicial, esta función deberá manejar de alguna forma el cómo encontrar fuentes. La función debe recibir un objeto del tipo de la fuente y debe regresar un objeto de tipo de la fuente.
- `_observadoras`: Dado una fuente y un delta, esta función debe buscar fuentes vecinas partiendo de la fuente que reciba y "alejándose" una delta distancia. La función debe regresar un objeto de tipo de la fuente.
- `_nectar`: Es la función que habrá que definir con mayor cuidado puesto que todas las soluciones y comportamiento de la colmena se verá afectado por esta función. La función recibe una fuente y de alguna forma la evalúa para regresar un número. La forma en la que se trabaja en la colmena es *higher is better*, esto quiere decir que entre más grande el número que esta función regrese, mejor la fuente es.
- `_explotar`: Esta es una función que recibe una fuente en un tiempo T_i y regresa la evaluación de una nueva fuente en algún estado T_{i+k}. La nueva fuente sustituirá a la fuente anterior dentro de la colmena.

La colmena es el objeto que comunica a todas las abejas y fuentes, es el origen principal de datos de la heurística. La colmena tiene un tamaño constituido de la siguiente manera:

$$\frac{|colmena|}{2} = |Empleadas \cup Exploradoras| = |Observadoras|.$$

Otro atributo importante de la colmena es la fuente inicial. Sin una fuente inicial, una abeja no podría empezar a explorar, trabajar ni observar debido a que debe existir un estado inicial para que todas las abejas trabajen en la colmena.

Figura 6.4: Diagramas UML del paquete `abc`.

Durante una iteración de la colmena, las funciones principales a ser llamadas en orden son `_itera_exploradoras()`, `_itera_empleadas()` e `_itera_observadoras()`. Si a la colmena le fue asignada una función positerativa, al terminar una iteración a cada fuente i se mandará a ejecutar la función `_termino_iteracion(`i`)`.

Cada una de las funciones con prefijo `_itera_` se recorre al conjunto de abejas de cierto tipo y con ayuda de métodos y funciones auxiliares, realizan las acciones que cada abeja tiene destinada para que en la suma del conjunto de las acciones de todas las abejas contribuyan a seleccionar las mejores fuentes y construir una mejor solución en cada iteración.

6.3. Comunicación heurística-emulador

El diseño actual delega la responsabilidad del buen funcionamiento casi totalmente al objeto que hace uso de las clases `Colmena` y `Tetris`. Dichos objetos deberán invocar a algunas funciones previas a la interacción de ambos, como la asignación de las funciones a la colmena o desactivar la opción de quitar filas en automático dentro de un juego de Tetris. La clase encargada de realizar estas llamadas y generar un comportamiento armonioso entre ambos objetos es la clase `Abejas_Tetris`.

Abejas_Tetris
+ pierde : bool + online : bool + colmena : Colmena + lista_piezas : list<Tipo> - _x : int - _y : int - _tetris : Tetris
+ __init__(online : bool, size_colmena : int, limite_it : int delta: float, alto : int, ancho : int) : Abejas_Tetris + set_lista_piezas(piezas : list<str>) : void + init_colmena() : void + juega_online(iteraciones : int, limpieza : bool) : Tetris + interactivo() : void + pinta_historia(historia : list<Movimiento>) : void ...

Figura 6.5: Atributos y métodos de la clase `Abejas_Tetris`.

Dentro de la clase `Abejas_Tetris` se almacenan como atributos una partida de Tetris y una colmena con abejas, una bandera que nos dice el modo de juego, la lista de piezas que se jugarán y una segunda bandera que nos dice si el juego ha finalizado. Es necesario definir también las funciones de evaluación que las abejas usarán ya que es en este punto donde la colmena debe recibir cada una de las funciones para la evaluación de los tableros que más adelante se discutirán.

6.4. Visualización de datos

Una vez que se haya creado un historial de ejecuciones, es natural querer ver de manera secuencial las decisiones tomadas por la heurística y seguir paso a paso cada iteración del tablero para analizar el impacto de la función de costo y el comportamiento de la colmena de una manera visual. Dado que se necesita tanto de la solución propuesta por la colmena y las operaciones sobre un tablero (que tiene de estado inicial una partida nueva), el lugar más conveniente para crear las funciones de visualización es el mismo lugar en el que ocurre la lógica de ambos.

El método `pinta_solucion()` dentro de la clase `Abejas_Tetris`, recibe una lista de movimientos e inicializa los parámetros de la biblioteca de visualización *pygame*. Mientras la lista que recibe no esté vacía o el juego no llegue a un estado donde el usuario[2] pierda, el método mandará a la lógica del juego la orden de moverse a un estado siguiente al ejecutar el movimiento que saque de la cabeza de la lista, posteriormente mandará a llamar el método `dibuja()` que tomará el estado del tablero y las fichas para actualizar su posición y dibujar una nueva versión del juego.

Abejas_Tetris
...
... + pinta_solucion(solucion : list<Movimiento>) : void + set_gui() : void + dibuja() : void - __dibuja_fichas() : void - __dibuja_tablero() : void

Figura 6.6: Funciones y métodos de visualización.

Existen tres maneras de visualizar una partida y estas dependerán del modo de juego: cuando se ha terminado de realizar una ejecución de la heurística, ya sea por medio de una búsqueda de semillas o una partida aleatoria, el modo interactivo que espera a que las indicaciones sean introducidas desde la terminal. El último modo es una forma de visualizar resultados anteriores y que sólo necesita una lista de movimientos para dibujar una partida de Tetris.

6.5. Funciones y métodos adicionales

Como parte del análisis del problema se realizaron decisiones de diseño que pueden ser importantes para entender el funcionamiento del sistema. Algunas de estas decisiones fueron el resultado de problemas que se encontraron en diseños previos como parte de la optimización o manejo funciones intrínsecas del lenguaje.

Algunas de las decisiones tomadas parecieran no son congruentes con características del lenguaje o los patrones de diseño más usados, sino que responden a la experiencia de una primera implementación en otro lenguaje[3].

6.5.1. Creación y rotaciones de las piezas

La creación de las casillas de las piezas no es generado dentro de la clase `Pieza`, sino dentro del archivo `./abejas_tetris/tetris/tipo_pieza.py` debido a que con sólo conocer el punto donde se desea colocar el objeto y su orientación, es posible deducir siempre con un número constante de operaciones (a lo más tres) la posición del resto de las casillas.

La única operación que realizan las casillas como parte del objeto pieza, la operación de rotación, tampoco se encuentra dentro de la clase sino que se delega a la función `rota()` dentro del mismo archivo donde se crean. La decisión detrás de separar estas funciones de la clase `Pieza` es que no se

[2]El usuario en este caso particular es la colmena.

[3]Dicho proyecto fue creado en el lenguaje C y se puede ver el funcionamiento y código fuente en la siguiente dirección: `https://github.com/ricardorodab/abejas-tetris`.

necesita información adicional ni operar con los atributos del objeto para generar la acción de crear y girar las casillas, generando una mayor limpieza en el código y aumentando una velocidad mayor al no depender del estado de las casillas previas de una pieza al girar sino de operaciones constantes.

6.5.2. Archivo de parámetros globales

Dentro del archivo `./etc/config.cfg` existen variables globales que se usan para la ejecución del proyecto y dentro de éste se pueden colocar parámetros de la colmena como su tamaño, las deltas de las abejas observadoras, el límite de veces que una abeja trabaja sobre una fuente específica así como parámetros del juego de Tetris como el alto y ancho del tablero.

Todos estos valores son de vital importancia durante la ejecución de la heurística y el menor cambio de éstos, produce una salida de diferente calidad.

Para poder instanciar un objeto de algunas clases del proyecto es necesario haber obtenido los parámetros que se encuentran dentro del archivo con la función `get_config()` que se encuentra en la ruta `./abejas_tetris/parse_config.py`.

La creación de un archivo de parámetros independientes responde a la necesidad de experimentar con los distintos pesos que se les da a los valores de ejecución del programa.

6.5.3. Generador de números aleatorios

Debido a la naturaleza de las heurísticas y el proyecto, en muchas partes partes del código se pueden observar que se utilizan funciones como `get_random()`, `get_randrange()` y `get_randbits()`. Debido a que durante la experimentación es deseable poder mantener un control sobre los resultados, se usan lo que se conocen como *semillas generadoras de números pseudoaletorios* para asegurar que el programa sea replicable. Para forzar sólo un objeto que genere los números aleatorios, se usa el patrón de diseño conocido como *singleton*, para solicitar sólo un generador que funcione a la vez con sólo una semilla.

Para colocar una semilla se usa una función llamada `set_random()` que recibe como parámetro la semilla como un número entero. Junto este método se escribió un algoritmo que tiene como propósito el hacer búsqueda de semillas generadoras con resultados favorables que obtengan buenas partidas de Tetris. Para realizar la búsqueda se debe colocar la constante `-1` en la variable `SEMILLA` dentro del archivo de parámetros globales, colocar el número de semillas que se desean probar en la variable `SEMILLA_ITERA` y correr el *script* `./scripts/llena-semillero.sh` que se encarga de crear el archivo con las semillas aleatorias.

6.5.4. Constantes

Existen para este sistema valores constantes que no se desean cambiar como son el color de los tipos de las distintas piezas, el tamaño de los bloques o el tamaño de los márgenes. Además de ser un buen diseño y agregarle legibilidad al código, se tomó la decisión de crear un archivo autónomo ya que estos valores son completamente independientes de la ejecución de la heurística y del juego. Los valores constantes dentro del proyecto se usan para la visualización de los resultados y son propios de la biblioteca *pygame*.

Si por alguna razón se desearan cambiar los valores constantes, todos se mantienen en la ruta `./abejas_tetris/constantes.py`. La constante `ESCALADO`, es la responsable del tamaño de los objetos de visualización por lo que si se modificara, la interfaz gráfica se vería afectada haciendo que su visualización sea de menor o mayor tamaño.

Capítulo 7

Experimentación y resultados

Para entender el comportamiento de la implementación al modificar los parámetros de entrada, es de gran importancia analizar los resultados y conducir a la heurística hacia un estado de evaluación que se considere positivo. Como consecuencia a la experimentación del sistema, el ajuste de los pesos y las funciones evaluadoras se deben ir refinando de tal manera que al observar el movimiento de las piezas e ignorar el funcionamiento interno del proyecto, un espectador podría llegar a coincidir en las decisiones tomadas por la colmena. Durante este capítulo se explicarán las distintas estrategias de evaluación y los resultados obtenidos de cada una de ellas.

7.1. Funciones de comportamiento

Como bien se mencionó, cada objeto creado desde la clase `Abeja` posee cuatro atributos que tienen la particularidad de ser funciones que serán llamadas por la colmena a la hora de iterar sobre cada uno de los tipos de abeja. Todas las funciones, excepto la de las abejas observadoras, reciben un objeto de tipo `T`, que es el tipo de la fuente, la función de las abejas observadoras recibe un objeto de tipo genérico `T` y un tipo `float`. El tipo `T` en la implementación de este proyecto es la clase `Tetris`.

La función que se almacena dentro del atributo `_busca_fuente` : $T \to T$, es usada por las abejas exploradoras y al igual que todas las funciones que poseen el resto de las abejas de la colmena, tiene la característica de que el parámetro `fuente` que recibe puede ser ignorado en el caso de que la función se desee definir de forma independiente a algún estado anterior como es usual en las heurísticas. En el caso particular del juego de Tetris, la función sí utiliza el parámetro como un punto de partida de las abejas y regresa un juego de Tetris de tipo `Tetris`, donde una pieza es colocada en alguna posición de forma aleatoria.

En el atributo `_explotar` : $T \to float$ se guarda la función que todas las abejas empleadas usarán con la fuente que tienen asociada. Cada empleada de manera aleatoria buscará colocar una pieza dentro de la solución previa. Aunque es muy parecida a la función para buscar una fuente, esta función es diferente a `_busca_fuente()` porque el valor regresado no es la fuente, sino la evaluación del resultado de Tetris. El estado del juego sí es modificado y la partida de Tetris "avanza" en el tiempo.

El atributo `_observadoras` : $T \times float \to T$ posee un comportamiento un poco más complejo: en primer lugar clona el objeto que recibe para evitar modificar el estado de un objeto original asociado a alguna abeja empleada. Una vez teniendo un objeto clon, la abeja observadora realiza movimientos inversos dentro del historial reciente del juego de Tetris. En otras palabras, elimina pasos que la abeja empleada o exploradora hicieron para llegar al estado actual. Después de realizar los movimientos inversos, se le agrega de manera aleatoria nuevos movimientos hasta llegar al punto donde la pieza vuelva a ser fijada en el tablero. Las acciones de retirar movimientos y agregar nuevos puede verse como *alejarse* de la fuente original para conseguir una fuente en una *distancia* δ. De vuelta en la colmena, si la solución a distancia δ es mejor que la función original, la nueva fuente será la localizada por la abeja observadora. Son las abejas observadoras las encargadas de explorar los mínimos locales mientras que las exploradoras buscan los mínimos globales.

La última función es la de néctar o también llamada función de costo o *fitness*. Almacenada en `_nectar` : $T \rightarrow float$, esta función evaluará los juegos de Tetris y les asignará un número n que entre mayor sea, mejor será la cantidad de néctar y por lo tanto mejor la solución. Definir esta función conlleva una complejidad un poco mayor debido al número de variables que pueden influenciar en una partida de Tetris y que deben ser consideradas, ya que la definición tendrá un impacto directo sobre la calidad de resultados obtenidos; una función pobremente definida tendrá de forma obligada resultados pobres[1].

Debido que ninguna de las funciones mencionadas anteriormente están estrictamente definidas para resolver Tetris, se deben realizar pruebas y analizar los resultados para ajustar los parámetros que utilizan.

7.2. Métricas de desempeño

El modo en que un jugador humano mide su desempeño jugando Tetris es generalmente usando tres factores de medición: el tiempo de juego, el puntaje obtenido y el número de filas que son removidas durante una partida. Para la implementación propuesta, el puntaje obtenido será sustituido por el valor de la función de costo y el tiempo de juego que será medido por el número de piezas colocadas en una partida.

El propósito principal a optimizar será el número total de filas que son removidas por la colmena durante una partida completa de Tetris. Al optimizar las filas removidas, también hará posible que se incremente la función de costo y el número de casillas no ocupadas por fichas, aumentando el número de piezas que se puedan colocar en el tablero.

Existe en muchas versiones de Tetris modernos un modo de juego llamado *40 lines* que consiste en remover 40 filas de una partida de Tetris para considerar que el jugador haya ganado [45]. En este trabajo, nos limitaremos a conseguir la mitad para considerar un historial de movimientos buenos dado una lista de fichas.

El número de piezas jugadas también se usa como una métrica para afirmar que un tablero es mejor sobre otro. Si ambos tableros generan la remoción de quince filas pero un tablero juega 20 piezas más que el otro, significa que las piezas presentan un mayor nivel de horizontalidad y por lo tanto existe una probabilidad mayor de que el tablero con mayor número de piezas pueda desaparecer una fila o al menos obtenga más tiempo de juego.

Las fichas a jugar deben ser generadas a partir del un programa en *script* que se encuentra en el archivo `./scripts/lista-fichas.sh`. El programa generará un archivo en formato `.csv` que tendrá mil fichas aleatorias en su representación de forma de cadena. Se debe correr el *script* cada vez que se desee generar una nueva combinación de fichas.

7.3. Funciones de costo

Como se ha explicado a lo largo de este trabajo, la función de costo tiene un gran impacto sobre los resultados que las heurísticas obtienen. Desafortunadamente son muchas las variables que

[1] En el caso de Tetris, un resultado pobre es considerado aquel en el que un número corto de iteraciones, lleva al estado de *game_over*.

influyen en una partida de Tetris que hay que considerar, tanto valores positivos como negativos que generan una calificación en el tablero.

Para ejemplificar cómo una función de costo puede impactar directamente al juego, se puede suponer que existe T un tablero vacío, f_1 y f_2 funciones definidas de la siguiente manera:

$$\begin{aligned} f_1 : T &\rightarrow float \\ f_2 : T &\rightarrow float. \end{aligned}$$

Para el siguiente ejemplo, se puede también suponer que las fichas a jugar por la colmena poseen el siguiente orden en específico: `L = {I, Rg, Lg, Sq, I, T, Ls, Rs, I}`.

Sea $f_1(T)$ tal que regresa un número mayor si cada pieza P_i se encuentra inmediatamente a la derecha de cada pieza P_{i-1}. Si no existe lugar a la derecha debido a que la pieza P_{i-1} está en el límite derecho del tablero, entonces las abejas considerarán una mejor solución el colocar la pieza en el extremo superior izquierdo. En la figura 7.1 se puede observar que la función no indica a las abejas que dejar casillas vacías entre niveles inferiores y superiores afecta negativamente a la partida. Si existen demasiadas casillas vacías, con un número bajo de piezas la probabilidad de perder crece de manera rápida.

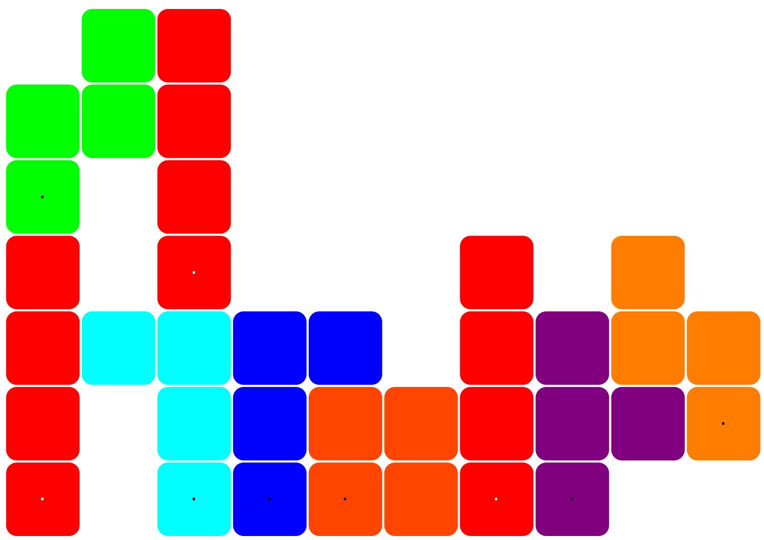

Figura 7.1: Una mala función de costo deja muchas casillas atrapadas y cubiertas.

La función $f_2(T)$ se define de tal forma que si el tablero T se encuentra vacío, se coloca la primera pieza en la posición más cercana al borde izquierdo. Si el tablero no está vacío entonces existe una casilla superior h y el resultado de la función es mayor si la siguiente ficha colocada no supera dicho límite y se intentan llenar todas las casillas por debajo de h, dándole un peso mayor a las que se encuentren más cercanas al borde inferior y más cercanas al borde izquierdo.

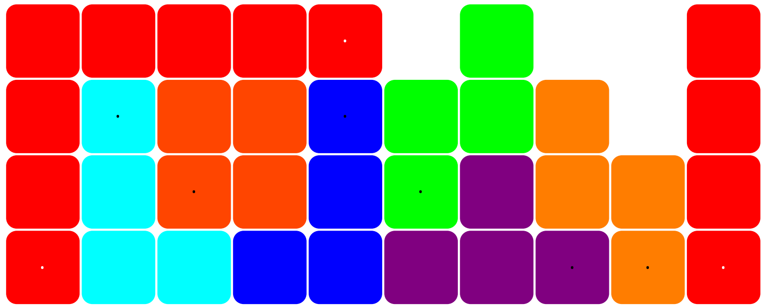

Figura 7.2: Una función que pareciera funcionar mejor al trata de cubrir todos los espacios.

Como se puede observar en la figura 7.2, para la instancia de fichas `L` existen dos filas completamente llenas al finalizar la última iteración por lo que las casillas en dichas filas serían removidas y el propósito principal de la heurística que es jugar por más tiempo y eliminar el mayor número de filas se incrementaría.

Aunque pareciera que la segunda función es mejor que la primera, el valor final de la heurística cambia si las fichas que contiene `L` fueran `L = {I, I, Sq, Sq, Sq, Sq}`. El resultado de la función $f_1(T)$ y $f_2(T)$ se pueden ver respectivamente en la figura 7.3 y figura 7.4. Se observa que incluso con una función generadora de néctar que se considere no genere tan buenos resultados, como lo es f_1, con la entrada correcta puede funcionar mejor que otra función que genere mejores resultados con un conjunto mayor de valores de entrada.

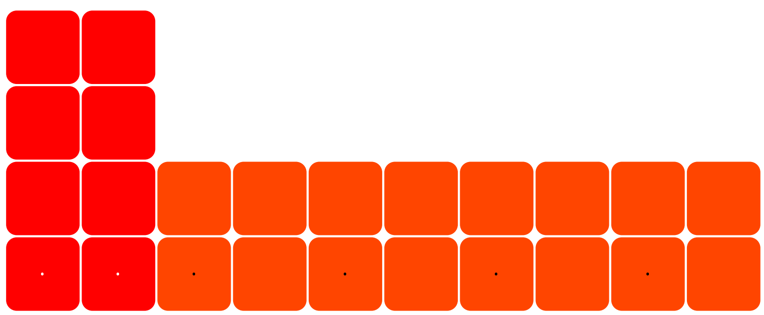

Figura 7.3: Una mala función puede dar buenos resultados si la entrada que recibe coincide con la lógica propuesta.

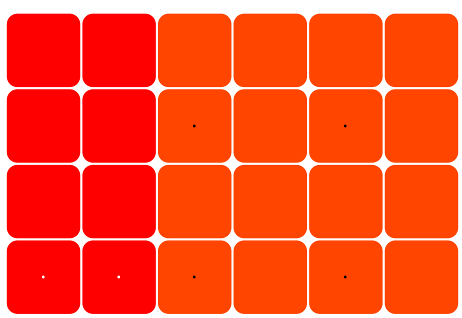

Figura 7.4: La función de costo que originalmente daba buenos resultados con una entrada específica no realiza mejoras considerables en la partida de un tablero de 10×20.

7.3.1. Filas entre pesos negativos

La primera función de costo que pareció funcionar de una manera general fue una división de valores de peso positivos entre negativos. Se hicieron una decena de pruebas utilizando exclusivamente un solo valor a optimizar sin obtener más de tres o cuatro filas removidas por juego pero la primer fórmula que se acerca al objetivo es:

$$\frac{1 + \texttt{filas-removidas}}{\texttt{horizontalidad} + \texttt{atrapados} + \texttt{cubiertos} + \texttt{altura}}.$$

Sea y_1 la altura de la casilla ocupada más arriba en el tablero y y_2 la casilla más abajo, la `horizontalidad` del tablero se define como $|y_1 - y_2|$. Una casilla (x, y) se considera `atrapada` si la casilla en la posición está vacía y sus vecinos inmediatos, las casillas en las posiciones $(x+1, y)$, $(x, y+1)$, $(x-1, y)$ y $(x, y-1)$ no se encuentran vacías. A diferencia de las casillas atrapadas, la casilla (x_n, y_i) se considera `cubierta` si existe una casilla ocupada en una posición (x_m, y_j) tal que $j > i$ y $n = m$.

Esta función de néctar tiene la peculiaridad de tener cuatro variables de peso negativo y sólo un peso positivo, así que lo que las abejas hacen no es buscar un buen tablero, sino el tablero con el menor peso negativo. Si de paso las abejas encuentran una forma de eliminar filas, entonces se quedan con ese tablero. La remoción de filas sube considerablemente el valor de la fuente debido a que la variable `filas-removidas` es en realidad el número de filas que se han quitado durante la ejecución del juego por una constante muy alta.

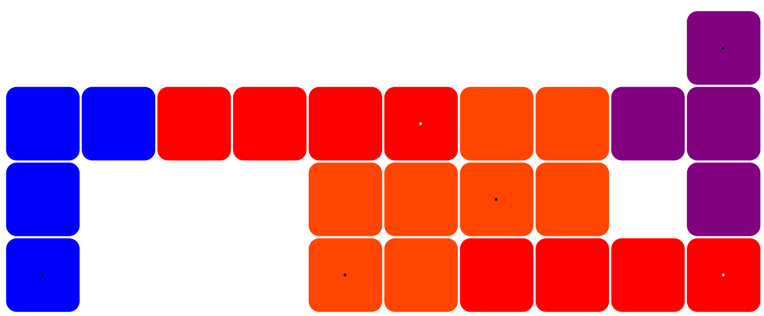

Figura 7.5: De izquierda a derecha las fichas son `Lg, I1, Sq1, Sq2, I2, T`.

Si se analiza la figura 7.5 se puede observar que la suma de los pesos negativos es bastante alta y su néctar no es muy bueno:

1. No hay ninguna fila llena de casillas ocupadas por lo que la variable `filas-removidas` es igual a cero.
2. Existe una casilla atrapada entre las fichas `T, I` y `Sq2`.
3. Existen siete casillas consideradas como cubiertas, seis debajo de las fichas `Lg, I` más la casilla atrapada.
4. La horizontalidad tiene el valor de uno ya que la casilla ocupada más abajo alcanzable desde el tope del tablero tiene altura tres y la más arriba tiene altura cuatro.
5. La altura total de la partida tiene el valor de cuatro.

Suponiendo que todos los pesos negativos son multiplicados por peso uno, nos queda el siguiente valor del néctar de la fuente:

$$\frac{1+0}{1+1+7+4} = \frac{1}{13} \approx 0.077$$

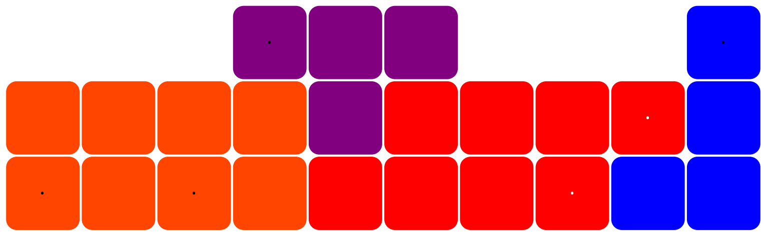

Figura 7.6: Tablero que toma las piezas de la figura 7.5 y la función de filas entre pesos negativos.

Un valor más aceptable por esta función de néctar sería el tablero de la figura 7.6 donde podemos observar que existen dos filas completamente llenas, lo que eleva el valor de la variable `filas-removidas` a $2 \times c$ donde c es un peso constante. No se observan espacios entre las casillas ocupadas por lo que `atrapados` $=$ `cubiertos` $= 0$. La altura así como el valor que mide la horizontalidad del tablero, es la misma antes y después de limpiar el tablero, es decir, remover las filas no cambia los valores de `altura` $=$ `horizontalidad` $= 1$. Suponiendo una $c \geq 2$ y sustituyendo $c = 2$, la fórmula final de la función queda de la siguiente manera:

$$\frac{1+(2\times c)}{1+0+0+1} = \frac{1+4}{2} = \frac{5}{2} = 2.5$$

Si en la colmena existieran ambos tableros tanto de la figura 7.5 como de la figura 7.6, las abejas observadoras escogerían ir a analizar fuentes vecinas de la fuente que tiene el néctar con valor 2.5 o mayor y se mantendrían dichas fuentes con mayor probabilidad que el resto.

La función es un indicador de lo bueno o malo que puede llegar a ser un tablero sobre otro, sin embargo las variables que se utilizan pueden influenciar mucho o muy poco sobre un tablero ya que existen casos en los que una vez que una variable eleve su valor sobre la evaluación, minimice el impacto de las otras y las abejas simplemente las ignoren porque su optimización no tiene gran ventaja.

Las primeras pruebas muestran que la función trabaja de la forma que se creía que lo haría. En la figura 7.7 se ve un crecimiento de filas eliminadas del juego conforme las pruebas de las distintas semillas avanzan.

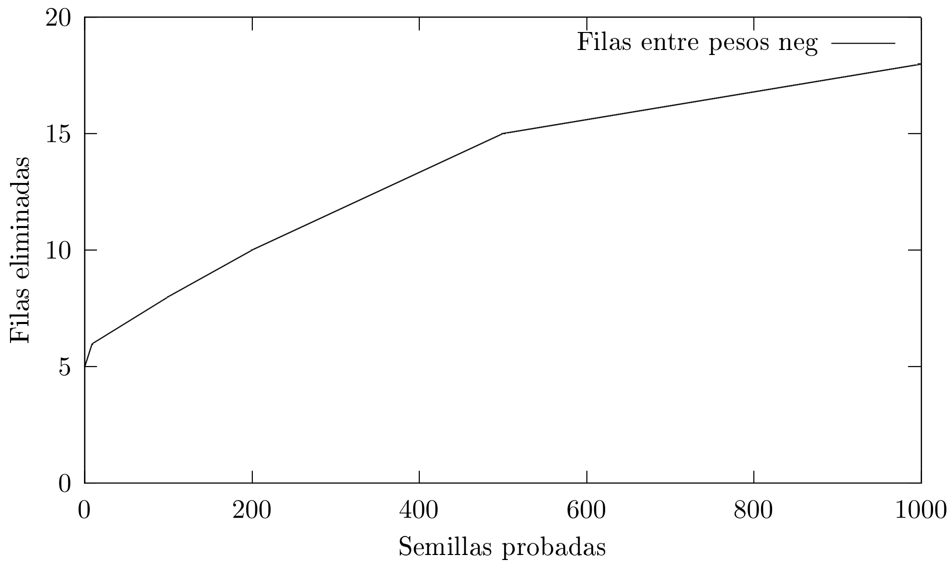

Figura 7.7: Relación de crecimiento de filas removidas por semillas usadas.

7.3.2. *Raining skyline* ponderado

Una estrategia diferente es obtener el número de casillas en las que una pieza puede ser colocada y tratar de optimizar ese número. Al optimizar el número de casillas disponibles se garantiza que el juego le dará un peso mayor de manera automática a aquellos tableros en los que existan menos piezas; en otras palabras, la heurística con esta función de costo entregará una mejor evaluación a aquellos tablero en los que se remueven filas. A continuación se enuncia un ejemplo de cómo obtiene el valor del tablero esta función a la que se le da el nombre de `cuenta_descubiertos()`:

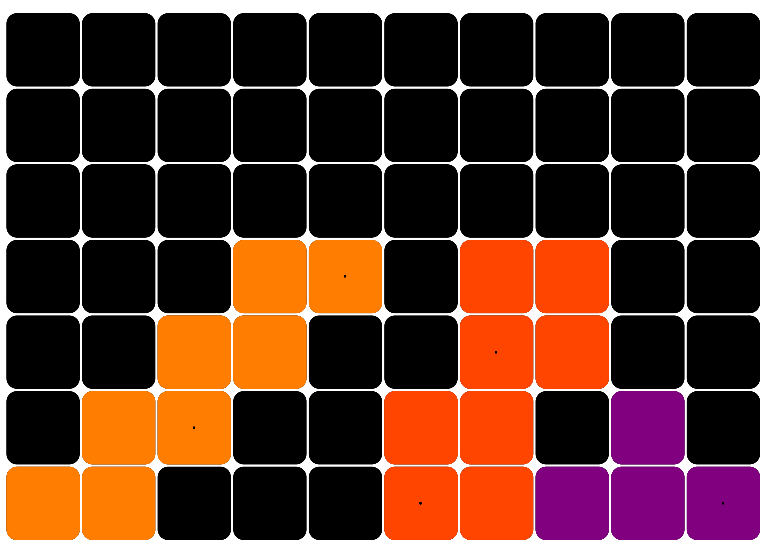

Figura 7.8: Tablero aleatorio con cinco piezas de altura cuatro.

Sea un tablero T en un tiempo t, donde T posee al menos una pieza, la evaluación de T dependerá del número de casillas que posea el *raining skyline*[2] de las piezas colocadas en el tablero. La manera de obtener este *raining skyline* es tomando una línea imaginaria que pasa por todas las casillas que no tienen a ninguna otra casilla ocupada arriba de ellas; en otras palabras, la línea se dibuja en la frontera superior del polinomio que crean todas las columnas con su casilla ocupada más alta. La posición de la casilla $cas \in C_{sup}$ es la altura más alta de cada columna del tablero, en caso de no existir una casilla ocupada en la columna, el valor de y de cas es $y = 0$.

Al terminar de obtener todas las $cas \in C_{sup}$ de cada una de las columnas, la manera de obtener la cantidad rs de casillas del *raining skyline* se realiza de la siguiente forma:

$$rs = \sum_{i=1,j=1}^{ancho,alto} j \ \mid \ (i,j) \in C_{sup}.$$

Una vez obtenido el número rs de casillas dentro del *raining skyline*, se sabe que si existen n casillas totales en el tablero T, el número de casillas fuera del *raining skyline* cf está dado por $cf = n - rs$. Una vez obtenido el valor cf, se puede devolver el valor del néctar del tablero de la siguiente forma:

$$f(T) = \frac{cf}{n}.$$

Para obtener el mayor provecho de esta función, es necesario que la bandera de configuración `LIMPIEZA` que se encuentra en archivo de configuraciones globales esté desactivada (con valor `False`) debido a que si está activada cada tablero eliminará piezas cada vez que se calcule el néctar y de esta forma se producirá una pérdida de información al tener menos casillas en el *raining skyline*.

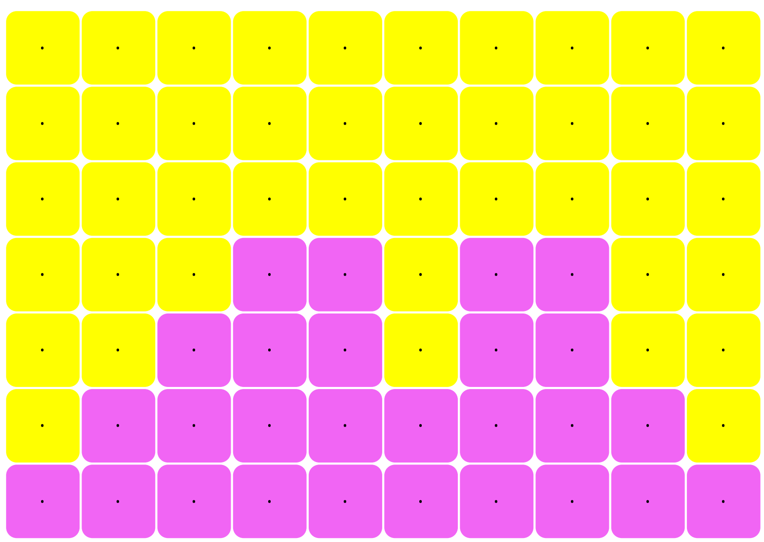

Figura 7.9: Ejemplo de cuál es el raining skyline (en rosado) del tablero en la figura 7.8.

En un comienzo se experimentó considerando que todas las casillas tienen en principio el mismo peso pero los resultados durante la experimentación no hicieron que superara a 5 el número de filas eliminadas durante la ejecución. Se probaron $5,000$ semillas diferentes pero usando sólo la función

[2]"Horizonte de lluvia" en inglés.

como se definió, la experimentación mostró casos en los que incluso un tablero posee mejor néctar con un menor número de filas removidas.

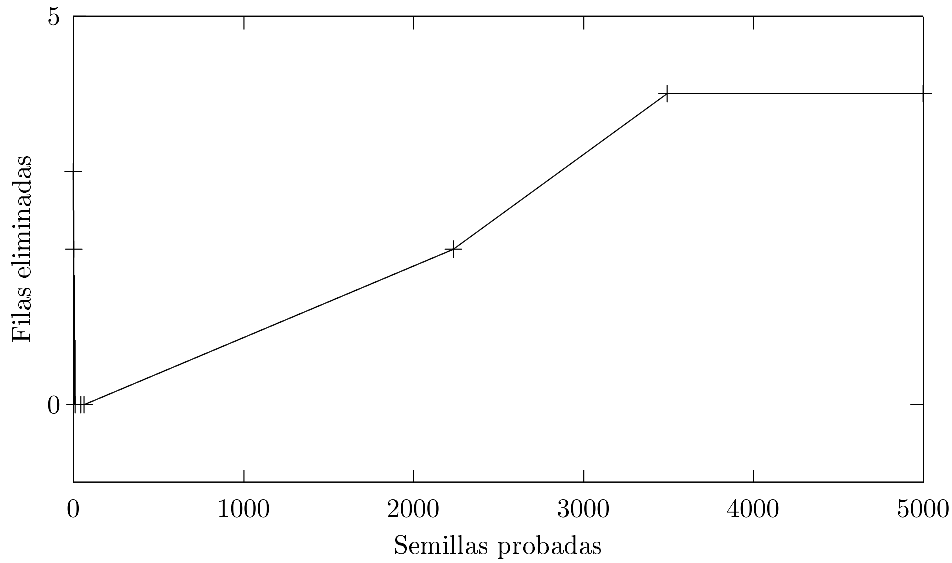

Figura 7.10: Relación de crecimiento de filas eliminadas por semillas usando la función de *raining skyline* sin ponderar.

El siguiente paso fue asignarle a cada casilla, vacía y ocupada un valor que es directamente proporcional a su posición y en el tablero. Este valor que le es asignado tiene el propósito de indicarle a las abejas dos cosas sobre el tablero:

1. Entre más abajo coloquen las abejas la pieza, menor peso restarán al conjunto de piezas disponibles.

2. La altura del tablero es un indicador de lo "saludable" que es la decisión tomada.

La función de costo $f(x)$ entonces es redefinida de la siguiente manera para que pueda considerar no sólo la altura sino también el peso de cada una de las casillas que se encuentran dentro de la altura:

$$rs = \sum_{i=1,j=1}^{ancho,alto} j \mid j > b \;\wedge\; (i,b) \in C_{sup}$$

$$f(T) = \frac{rs}{\sum_{i=1}^{alto}(ancho \times i)}.$$

La función ahora además de darle prioridad a tableros en los que se trata de conseguir la menor área de casillas ocupadas, de estos tableros se considerará mejor solución aquellos en los que las casillas que se tengan que ocupar sean casillas inferiores.

Los primeros resultados de esta función fueron mejores que las de su primer versión. Si bien los movimientos mejoraron el tablero en función del tiempo considerablemente, lo hizo de manera gradual con base a el número de semillas que se probaron como se puede ver en la figura 7.11

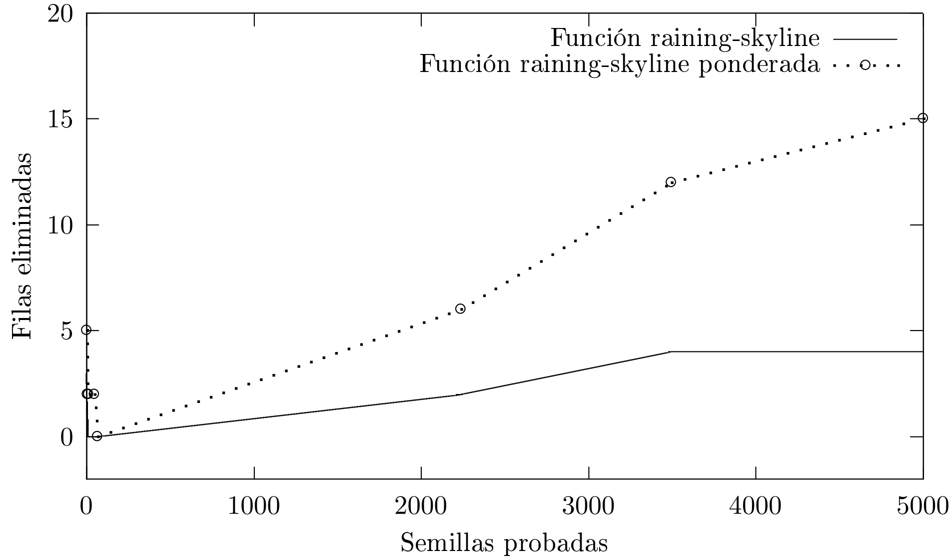

Figura 7.11: Relación de crecimiento de filas eliminadas por semillas usando la función de *raining skyline* con y sin ponderar las casillas.

Aunque el resultado se acerca al objetivo (20 filas eliminadas), las pruebas no se acercan al resultado esperado tan rápidamente. Algunas alternativas son aumentar el tamaño de la colmena o la distancia que las abejas observadoras recorren pero ambas opciones tienen la desventaja que el tiempo por semilla se incrementa.

Se puede observar en la figura 7.11 que el primer resultado obtenido por la función elimina cinco filas pero pronto encuentra una semilla que le genera un mejor resultado y las filas que son eliminadas caen a sólo dos. Aunque la función trabaja correctamente y su valor de evaluación en efecto puede llegar a ser mayor independientemente del número de filas que retira, el quitar filas es el objetivo que se desea optimizar.

Para solucionar el problema de la disminución de filas eliminadas, lo que se hace es modificar de nuevo la función de tal manera que el obtener un número mayor de filas eliminadas, impacte positivamente al resultado de la evaluación. La obtención de los pesos y la fórmula para obtener el *raining skyline* no se desea modificar por lo que sólo agregaremos una nueva variable; sea fe el número de filas eliminadas en un tablero de Tetris, la nueva función queda de la siguiente manera:

$$f(T) = \left(\frac{rs}{\sum_{i=1}^{alto} (ancho \times i)} \right) \times (1 + fe).$$

Donde rs no cambia su definición de la segunda versión de la función. Inmediatamente al poner a prueba la nueva función, con pocas semillas se observa un impacto favorable muy notable en el desempeño de la función de costo. Se puede observar y comparar el resultado de esta función en la figura 7.12.

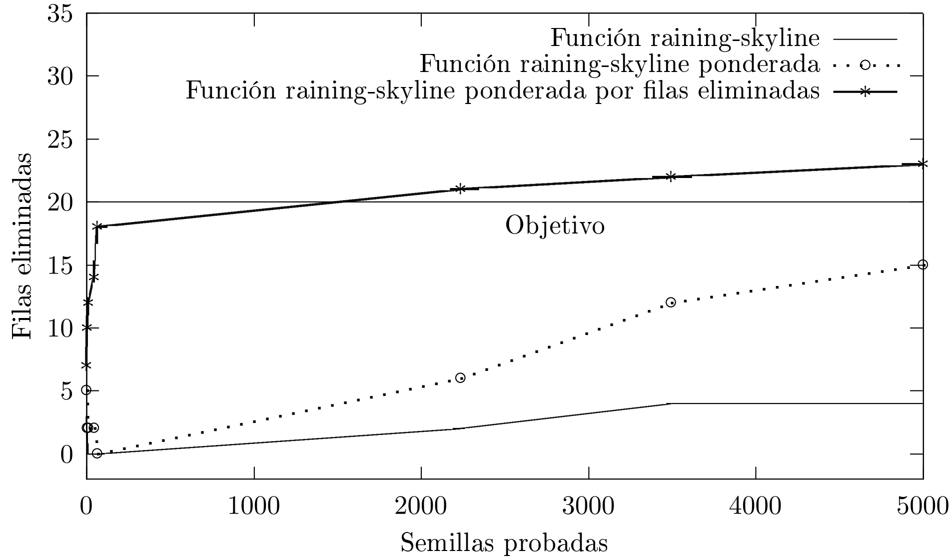

Figura 7.12: Relación de crecimiento de filas eliminadas por semillas usando la función de *raining skyline* con y sin ponderar las casillas por el número de filas eliminadas.

7.3.3. Función híbrida

Se ha creado una tercera función de evaluación que no es más que el producto de ambas funciones previamente definidas. No utiliza todos los pesos negativos sino que divide sobre el número de casillas cubiertas.

$$f(T) = \frac{rs \times (1 + \texttt{filas\_eliminadas})}{(1 + \texttt{cubiertos})}.$$

Las pruebas realizadas con colmenas de cien, quinientos y mil abejas no mostraron un cambio significativo en el comportamiento; por el contrario, los resultados de tres de las 5 pruebas resultaron en resultados pobres a comparación de previos resultados con cada función por lo que se descartó la función.

7.4. Análisis de resultados

Existen además de los resultados de la función de *fitness* otros factores que modifican e influyen tanto positiva como negativamente el resultado de una ejecución del programa. A continuación se discuten algunos como el tamaño de la colmena y la búsqueda de semillas para el generador de números pseudoaleatorio.

7.4.1. Tamaño de la colmena

El tamaño de la colmena tiene una gran influencia en la ejecución y su tamaño se mantuvo constante en los resultados anteriores: 50 abejas observadoras y 50 abejas exploradoras para comenzar la colmena.

Se puede observar con pocas semillas que si se incrementa el tamaño de la colmena, la calidad de las soluciones también lo hace. La mayor desventaja de aumentar el tamaño de la colonia es que el tiempo de ejecución por parte de la colmena también aumenta drásticamente; una prueba realizada en la computadora que se describe en el apartado 5.3, muestra que corriendo la función de evaluación *filas entre pesos negativos* en una colmena de tamaño de 100 abejas, (50 exploradoras y 50 observadoras) tardan 9.96 segundos en terminar de crear un juego con seis filas removidas. Bajo las mismas condiciones pero con una colonia de tamaño mil, (500 exploradoras y 500 observadoras) las abejas tardan 4 minutos con 35.7 segundos, sin embargo el resultado del juego de prueba eliminó 28 filas.

Usando la función *raining skyline* con una colmena de cien abejas, el número de filas que desaparece son cinco con sólo una ejecución que toma 8.76 segundos. Al elevar el número de abejas de nuevo a mil, la solución entregada es 1.3 veces mejor, con un resultado de 91 piezas jugadas y 26 filas removidas. Es de nuevo el tiempo de respuesta el que se eleva a 3 minutos con 34.48 segundos.

De las 22, 000 semillas que se corrieron para probar el comportamiento de las funciones de néctar, 15, 000 se corrieron con una colonia de tamaño cien, 6, 000 con una colonia de tamaño 200 y 1, 000 con tamaño mil. El resultado de las colonias de mil sobre las de cien abejas obtuvo una mejor solución por un aproximado de 490 % filas más pero el tiempo de iteración sobre las semillas se incrementó en 26 veces el tiempo que llevó ejecutar las colmenas de cien abejas. Este resultado de tiempo es poco práctico para un juego de Tetris en tiempo real.

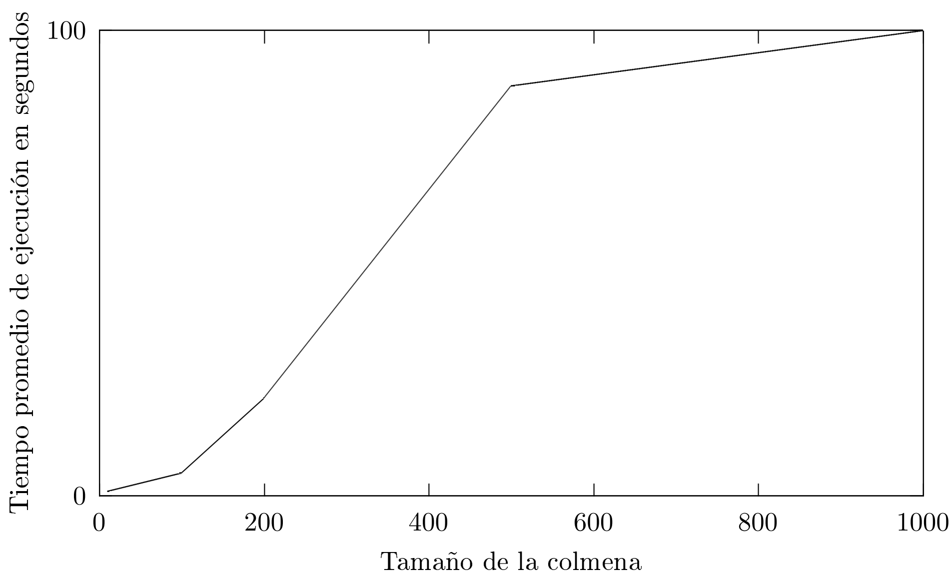

Figura 7.13: Resultado de evaluar promedios de tiempos de ejecución con distintos tamaños de la colmena.

Por último, se debe mencionar que un mayor tiempo de ejecución no es sinónimo de un mal desempeño en una función; las mejores funciones deberán hacer que el tablero juegue más piezas, por lo que calcular la posición de las piezas siguientes llevan mayor tiempo de ejecución. Prueba de ello es la mejor solución encontrada durante la experimentación, se encontró en un tiempo aproximado de 7 minutos.

7.4.2. Experimentación de semillas

Para cada función de evaluación de probaron al menos 10, 000 semillas diferentes y se observó un comportamiento similar a la hora de experimentar con muchos valores; el número de pruebas que las distintas semillas contribuían a la búsqueda de un mejor tablero, se estabilizaba después de las primeras cien semillas.

Independientemente de la función usada, el número de filas removidas es mejorada en el uso de las primeras 15 semillas, posteriormente hay un aumento de la función de costo muy reducido y se estabiliza. En el caso de la experimentación con las 10,000 semillas, ninguna función mejoró pasando la semilla $6,000$ y sólo una mejoró después de la $3,500$.

7.4.3. Resultados de funciones

El éxito de la heurística consiste en delegar toda responsabilidad a la definición de la función *fitness* y debido a la naturaleza de las funciones generadoras de números aleatorios, es imposible conocer con certeza, como ya se ha discutido, si una función es estrictamente mejor que otra para una mayor cantidad de listas de piezas de entrada. Dicho lo anterior, se tomó una muestra para comparar los resultados de las dos funciones definidas en este capítulo. Lo que se busca con esta comparación es mostrar de manera informal una similitud de comportamiento con base a un conjunto de pruebas pequeño pero modificando los mismos parámetros para ambas funciones.

Para la muestra que se tomó se consideró una colmena de 300 abejas, un límite de iteración de cada fuente de 50, una distancia δ de 0.07 y los siguientes pesos:

$$\begin{aligned}
\texttt{PESO_HORIZONTALIDAD} &= 1 \\
\texttt{PESO_ATRAPADOS} &= 3 \\
\texttt{PESO_CUBIERTOS} &= 2 \\
\texttt{PESO_FILA_REMOVIDA} &= 1000 \\
\texttt{PESO_ALTURA} &= 1.
\end{aligned}$$

Los resultados son mostrados en la gráfica de la figura 7.14 y se puede corroborar que el comportamiento de ambas funciones es muy similar.

Para colmenas muy grandes, la función de división de pesos mostró una ligera mejoría tanto como en tiempo de respuesta así como en resultados obtenidos, sin embargo, la *raining skyline* demostró ser más eficiente en cuanto a estabilidad de resultados se trata.

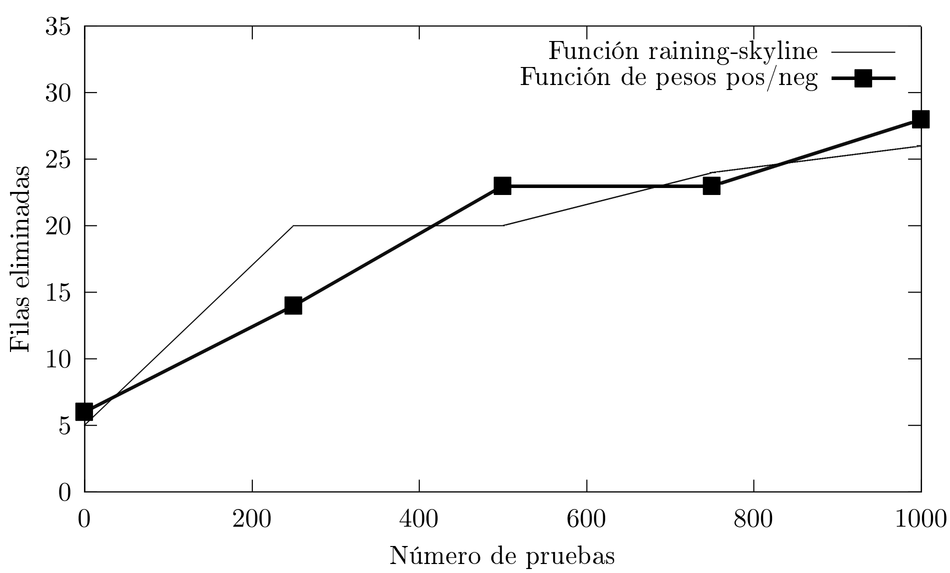

Figura 7.14: Comparación de las funciones propuestas sobre una muestra y valores específicos.

7.5. Conclusión de la evaluación

La meta fue superada rápidamente conforme el refinamiento de las funciones de costo. Se consiguieron muestras mayores a las propuestas por el método de juego *40 lines*.

Los resultados obtenidos superaron las expectativas que se propusieron al inicio del diseño y las pruebas. Dos muestras de la experimentación se pueden encontrar en las direcciones `https://youtu.be/F-Nxjvu8fPA` y `https://youtu.be/rjpVh7kaREY`. La primera muestra toma la función de *raining skyline* con 500 abejas, esta muestra supera la primera meta propuesta como objetivo de la experimentación. La segunda muestra toma la función de pesos y una colmena de tamaño 1000. Esta muestra supera las 70 filas eliminadas y en el video se está documentando el historial de movimientos del resultado con casi nueve minutos de juego.

Capítulo 8

Conclusiones y trabajo futuro

El origen y motivación de realizar este trabajo surgió de una primera implementación realizada en un curso optativo que lleva por nombre *Seminario de Ciencias de la Computación B* y tenía por tema principal *Heurísticas de Optimización Combinatoria*. Si bien la primera implementación (que se puede encontrar en la siguiente dirección: `https://github.com/ricardorodab/abejas-tetris`) conseguía realizar combinaciones de juego con cierto grado de éxito, careció (por una cuestión de tiempo) del detalle de diseño, desarrollo de funciones de costo y pruebas que sí son realizadas en este trabajo.

Esta segunda implementación se originó como un experimento para entender el comportamiento de una heurística documentada a un problema ampliamente conocido. Es el deseo del autor que la compilación de terminología ocupada en conjunto a las definiciones mencionadas, teoremas, ejemplos y experimentos resulten de una mayor facilidad de entendimiento al lector para despertar su curiosidad y si lo desea, continuar las investigaciones que este trabajo introduce.

Se tomaron las bases teóricas para explicar por qué la selección del problema de Tetris. Se explicó el funcionamiento de las heurísticas como lo es la *Colonia de Abejas Artificiales* o *ABC* por sus siglas en inglés. Se desarrolló la justificación de tomar una heurística sobre otros métodos resolutivos y se argumentó el porqué del diseño que se implementó para realizar la experimentación.

Aunque el enfoque dado a la forma de resolver el problema de Tetris fue reducido a dos funciones, es parte del diseño la facilidad de comunicarle a la heurística ABC la implementación de nuevas funciones de desempeño para extender la experimentación a nuevos métodos.

El conjunto de resultados generados por la implementación muestran que:

1. Sin importar que el problema de Tetris sea un problema *NP*-completo, existen métodos prácticos que pueden encontrar soluciones de buena calidad.

2. La heurística ABC es un buen método de resolución de problemas ya que la función de costo de la que depende sus resultados es independiente al comportamiento del colectivo en la colmena.

Un paso siguiente a la solución propuesta en este trabajo es crear evaluaciones *offline* tomando en consideración las piezas siguientes como origen de las fuentes de la heurística. Esta propuesta queda fuera del alcance de este trabajo debido a que las funciones y el diseño usado deberán ser modificados de manera profunda; sin embargo, la heurística debería mantener siempre la misma estructura sin cambios considerables.

Otra optimización necesaria sobre el diseño de la heurística es la paralelización de las tareas dentro de la colmena. Cada tipo de abeja puede ser dividida a un proceso productor-consumidor independiente que espere a ser alimentada de fuentes para su trabajo. El resultado mostrado no cambiará con dicha optimización pero el tiempo de respuesta por parte de la heurística se vería reducido.

Para finalizar este trabajo hay que mencionar que las metas propuestas como tiempo de juego y número de filas eliminadas, no sólo fueron alcanzadas sino que superadas por más del doble y se plantea la pregunta *¿podría la heurística vencer a un jugador experto en tiempo de juego real con una mejor función de costo?*

Apéndice A

Algoritmo 3SAT

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

__author__ = "José Ricardo Rodríguez Abreu"
__license__ = "GPL"
__version__ = "1.0.0"
__email__ = "ricardo_rodab@ciencias.unam.mx"

from random import randrange
import random
import sys

# Lista básica para las variables.
var_base = ["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q",
            "r","s","t","u","v","w","x","y","z"]

# La lista que usaremos. Podemos extenderla tanto como queramos.
variables = []

"""
 Esta función crea la lista de variables que usaremos como disponibles.
 La lista más pequeña es igual a la lista var_base.
 Ejemplo: [a1, b1, c1, a2, b2, c2, ...]
"""
def set_variables(num_var):
    if num_var < len(var_base):
        for var in var_base:
            variables.append(var)
    else:
        it = 0
        for contador in range(num_var):
            if contador > 0 and contador % len(var_base) == 0:
                it = it + 1
            var = var_base[contador % len(var_base)]
            var = var + str(it)
```

```
            variables.append(var)

"""
Esta función crea una cláusula: para propósitos del problema
la cláusula está formada por tres términos y tiene la forma:
(p V q V r). Si una variable sale dos veces, el hash ignora la repetición.
"""
def crea_clausula():
    var1 = variables[randrange(len(variables))]
    var2 = variables[randrange(len(variables))]
    var3 = variables[randrange(len(variables))]
    return {var1 : bool(random.getrandbits(1)),
    var2 : bool(random.getrandbits(1)), var3 : bool(random.getrandbits(1))}

"""
Esta función crea una fórmula: para propósitos del problema
la fórmula es un conjunto de cláusulas en su forma FNC:
A ^ B ^ C ^ D ^ ...
"""
def crea_formula(limit):
    l = []
    for i in range(limit):
        l = l + [crea_clausula()]
    return l

"""
Esta función crea un hashmap con los valores
por default de todas las literales (que es None).
No confundir con la interpretacion.
"""
def asigna_valores(formula):
    valores = {}
    for clausula in formula:
        for literal in clausula.keys():
            if not literal in valores:
                valores.update({literal : None})
    return valores

"""
Esta función interpreta una cláusula regresando su valor de verdad.
"""
def interpreta_clausula(clausula, valores):
    valor = False
    for literal in clausula.keys():
        value = valores[literal]
        if not clausula[literal]:
            value = not value
        valor = valor or value
    return valor
```

```python

"""
Con esta función modificamos el valor de algún término o literal.
"""
def set_value(variable, valores, valor):
    valores[variable] = valor

"""
Método para visualizar de forma "bonita" una fórmula.
"""
def str_formula(formula):
    st = ""
    for clausula in formula:
        st = st + " ("
        for literal in clausula.keys():
            if not clausula[literal]:
                st = st + " ¬"
            else:
                st = st + " "
            st = st + literal + " " + "V"
        st = st[:-1]
        st = st + ") ^"
    st = st[:-1]
    return st

"""
Para que no exista una asignación de valores posibles y forzamos a
la evaluación exaustiva de todos los posibles valores de las variables,
añadimos un conjunto de términos de la siguiente forma:
"FORM_IMP  = (P V Q V R) ^ (P V Q V ¬R) ^ (P V ¬Q V R) ^ (P V ¬Q V ¬R) ^
(¬P V Q V R) ^ (¬P V Q V ¬R) ^ (¬P V ¬Q V R) ^ (¬P V ¬Q V ¬R)" de tal manera
que cuando se tenga que encontrar los valores, no exista nunca alguna
combinación tal que I(FORM_IMP) = 1.
"""
def crea_formula_imposible(formula):
    formula_imp = [{'IMP1' : True, 'IMP2' : True, 'IMP3' : True},
                   {'IMP1' : False, 'IMP2' : True, 'IMP3' : True},
                   {'IMP1' : True, 'IMP2' : False, 'IMP3' : True},
                   {'IMP1' : True, 'IMP2' : True, 'IMP3' : False},
                   {'IMP1' : False, 'IMP2' : False, 'IMP3' : True},
                   {'IMP1' : False, 'IMP2' : True, 'IMP3' : False},
                   {'IMP1' : True, 'IMP2' : False, 'IMP3' : False},
                   {'IMP1' : False, 'IMP2' : False, 'IMP3' : False}]
    return formula + formula_imp

"""
Dada una asignación de valores para cada incógnita, esta función regresa
el valor de la interpretación de la fórmula.
"""
```

```
def interpreta_formula(formula, valores):
    valor = True
    for clausula in formula:
        valor = interpreta_clausula(clausula, valores) and valor
        if not valor:
            return False
    print("La fórmula es: " + str_formula(formula))
    print("La interpretación es: " + str(valores))
    return True

"""
Inicializa los valores y fórmulas para poder
realizar experimentos sobre el problema 3-SAT
"""
def buscar_interp(limit, imposible):
    if limit < 1 and not imposible:
        print("Argumentos inválidos.")
        exit(0)
    set_variables(limit * 3)
    formula = crea_formula(limit)
    if imposible:
        formula = crea_formula_imposible(formula)
    valores = asigna_valores(formula)
    lista = valores.keys()
    __buscar_interp_aux__(formula, valores, lista)
    return False

"""
Función auxiliar "privada" que hace llamadas recursivas para
asignar valores a las variables. Realizar estas asignaciones
tiene una complejidad de O(2^n).
"""
def __buscar_interp_aux__(formula, valores, lista):
    global num_asign
    variable_local = lista.pop()
    num_asign = num_asign + 1
    set_value(variable_local, valores, False)
    if len(lista) == 0:
        if interpreta_formula(formula, valores):
            print("Encontró una solución!")
            exit(0)
    else:
        lista = __buscar_interp_aux__(formula,valores, lista)
    num_asign = num_asign + 1
    set_value(variable_local, valores, True)
    if len(lista) == 0:
        if interpreta_formula(formula, valores):
            print("Encontró una solución!")
            exit(0)
```

```
    if len(lista) > 0:
        lista = __buscar_interp_aux__(formula,valores,lista)
    set_value(variable_local, valores, None)
    lista = lista + [variable_local]
    return lista

def help():
    return "Ejecutar el programa pasando como primer dato el " \
     "número de cláusulas que desea seguido de la opión --loop si desea " \
     "correr el programa hasta encontrar una solución o --no-solucion si " \
     "desea correr todas las posibles asignaciones de valores pero sin " \
     "solución. Si se corren ambas opciones --loop --no-solucion el " \
     "programa nunca acaba."

if __name__ == "__main__":
    if sys.argv.__len__() < 2:
        print(help())
    else:
        args = []
        for i in range(sys.argv.__len__()):
            args.append(sys.argv[i])
        num_asign = 0
        contador = 1
        if str.isdigit(sys.argv[1]):
            num_formulas = int(sys.argv[1])
        else:
            num_formulas = 0
        loop = "--loop" in args
        imposible = "--no-solucion" in args
        if loop:
            while not buscar_interp(num_formulas, imposible):
                print("Fórmulas intentadas: " + str(contador))
                num_asign = 0
                contador = contador + 1
        else:
            buscar_interp(num_formulas, imposible)
        print("No se pudo realizar asignación, se intentó " \
        + str(contador) + " fórmula(s).")
        print("Num de asignaciones que se realizaron: " + str(num_asign))
```

Apéndice B

Heurística N-Reinas

```c
/* ---------------------------------------------------------------
 * reinas.c
 * version 1.0
 * Copyright (C) 2019  José Ricardo Rodríguez Abreu.
 * Facultad de Ciencias,
 * Universidad Nacional Autónoma de México, México.
 *
 * Este programa es software libre; se puede redistribuir
 * y/o modificar en los terminos establecidos por la
 * Licencia Publica General de GNU tal como fue publicada
 * por la Free Software Foundation en la version 2 o
 * superior.
 *
 * Este programa es distribuido con la esperanza de que
 * resulte de utilidad, pero SIN GARANTIA ALGUNA; de hecho
 * sin la garantia implicita de COMERCIALIZACION o
 * ADECUACION PARA PROPOSITOS PARTICULARES. Vease la
 * Licencia Publica General de GNU para mayores detalles.
 *
 * Con este programa se debe haber recibido una copia de la
 * Licencia Publica General de GNU, de no ser asi, visite el
 * siguiente URL:
 * http://www.gnu.org/licenses/gpl.html
 * o escriba a la Free Software Foundation Inc.,
 * 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 * ---------------------------------------------------------------
 */

/**
 * @file reinas.c
 * @author Jose Ricardo Rodriguez Abreu
 * @date 29 Junio 2019
 * @brief Heurística simple para el problema de las N-reinas..
 *
 * Este programa utiliza una heuristica simple iterativa
```

```
 * sobre el famoso problema las N-Reinas. Se busca la manera de
 * acomodar fichas de reinas dado un tablero de ajedrez sin que
 * se coman las unas a las otras.
 *
 * El programa usa el estandar de documentacion que define el uso de
 * doxygen.
 *
 * @see http://www.stack.nl/~dimitri/doxygen/manual/index.html
 * @see https://github.com/ricardorodab/tesis
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>


/**
 * @def MAX
 *
 * Usamos un pequeño macro para crear una función
 * que nos regrese el mayor de dos números.
 *
 */
#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))


/**
 * @brief Inicializa y aloja en la memoria el tablero a usar.
 *
 * Crea un tablero e inicializa todas las casillas en cero.
 * @param size - Es el tamaño del tablero.
 * @return Un tablero de sizeXsize.
 */
int** inicializa_tablero(int size)
{
  int i, j;
  int **tablero = malloc(size * sizeof *tablero);
  for (i = 0; i < size; i++)
    {
      tablero[i] = malloc(size * sizeof *tablero[i]);
    }

  for (i = 0; i < size; i++)
    {
      for (j = 0; j < size; j++)
        {
          tablero[i][j] = 0;
        }
```

```
    }
  return tablero;
}


/**
 * @brief Libera la memoria del tablero.
 *
 * Realiza un free del tablero; de todas sus casillas.
 * @param size - Es el tamaño del tablero.
 * @param tablero - Es el tablero a liberar.
 */
void free_tablero(int size, int** tablero)
{
  int i;
  for (i = 0; i < size; i++)
    {
      free(tablero[i]);
    }
  free(tablero);
}


/**
 * @brief Imprime el tablero en la linea de comandos.
 *
 * Crea de forma más legible un tablero a imprimir en la línea de comandos.
 * @param size - Es el tamaño del tablero.
 * @param tablero - Es el tablero actual.
 */
void imprime_tablero(int size, int** tablero)
{
  int i, j;
  for (i = 0; i < size; i++)
    {
      for(j = 0; j < (size*2)+1; j++)
        {
          printf("_");
        }
      printf("\n");
      printf("|");
      for (j = 0; j < size; j++)
        {
          if (tablero[i][j] == -1)
            {
              printf("Q");
            }
          else
            {
```

```
              printf(" ", tablero[i][j]);
            }
          printf("|");
        }
      printf("\n");
    }
  for(j = 0; j < (size*2)+1; j++)
    {
      printf("_");
    }
  printf("\n");
}


/**
 * @brief Actualiza si así se debe, todas las casillas..
 *
 * Dado una nueva reina, actualiza el tablero para ver reflejado que casillas
 * pueden colorar una reina posteriormente.

 * @param size - Es el tamaño del tablero.
 * @param x - Es la posición X de la nueva reina.
 * @param y - Es la posición Y de la nueva reina.
 * @param tablero - Es el tablero actual.
 */
void actualiza_tablero(int size, int x, int y, int** tablero)
{
  int i, j, h, l;
  for (i = 0; i < size; i++)
    {
      tablero[x][i] = 1;
      tablero[i][y] = 1;
    }
  tablero[x][y] = -1;

  if (x > y)
    {
      i = x - y;
      j = 0;

      h = x;
      l = y;
    }
  else
    {
      i = 0;
      j = y - x;

      h = x;
```

```
      l = y;
    }

  while (i < size && j < size)
    {
      if(i == x && j == y)
        {
          i++;
          j++;
          continue;
        }
      if(tablero[i][j] == -1)
        {
          i++;
          j++;
          continue;
        }
      tablero[i][j] = 1;
      i++;
      j++;
    }
  while (h < size && l >= 0)
    {
      if(h == x && l == y)
        {
          h++;
          l--;
          continue;
        }
      if(tablero[h][l] == -1)
        {
          h++;
          l--;
          continue;
        }
      tablero[h][l] = 1;
      h++;
      l--;
    }
}


/**
 * @brief Regresa un tablero con una nueva reina.
 *
 * Coloca una nueva reina y regresa el nuevo tablero. Si es necesario libera
 * el espacio.
 *
 * @param x - Es la posición X de la nueva reina.
```

```
 * @param y - Es la posición Y de la nueva reina.
 * @param size - Es el tamaño del tablero.
 * @param tablero - Es el tablero antes de colocar a la nueva reina.
 * @param libera- Es una bandera que usamos para liberar espacio al final.
 * @return El numero de semillas que podemos tomar del semillero.
 */
int** asigna_reina_aux(int x, int y, int size, int** tablero, int libera)
{
  int i,j;
  int** new_tablero = inicializa_tablero(size);
  for (i = 0; i < size; i++)
    {
      for (j = 0; j < size; j++)
        {
          if ((i == x && j == y) || tablero[i][j] == -1)
            {
              new_tablero[i][j] = -1;
              actualiza_tablero(size, i, j, new_tablero);
            }
        }
    }
  if (libera)
    {
      free_tablero(size, tablero);
    }
  return new_tablero;
}


/**
 * @brief La función que orienta a la heurística.
 *
 * Toda heurística necesita una función de costo que es el fórmula que indica
 * qué tan buena idea es realizar alguna asignación.
 *
 * @param size - Es el tamaño del tablero.
 * @param x - Es la posición x de la reina a analizar.
 * @param y - Es la posición y de la reina a analizar.
 * @param tablero - Es el tablero del juego.
 * @return Un número con la cantidad de casillas asignables en el caso de
 * asignar a la reina en la posición (x,y).
 */
int funcion_costo(int size, int x, int y, int** tablero)
{
  int **tmp = asigna_reina_aux(x, y, size, tablero, 0);
  int contador = 0;
  int i, j;
  for(i = 0; i < size; i++)
    {
```

```
      for (j = 0; j < size; j++)
        {
          if (tmp[i][j] == 0)
            contador++;
        }
    }

  free_tablero(size, tmp);
  return contador;
}


/**
 * @brief Asigna una reina en alguna fila.
 *
 * Asigna la siguiente reina en el tablero.
 *
 * @param x_number - Es el número de fila.
 * @param size - Es el tamaño del tablero.
 * @param tablero - Es el tablero del juego.
 * @return Un tablero con la reina asignada.
 */
int** asigna_reina(int x_number, int size, int** tablero)
{
  int *x = tablero[x_number];
  int i;
  int disponibles = 0;
  int casillas[size];
  for (int i = 0; i < size; i++)
    {
      if (x[i] == 0)
        {
          casillas[disponibles++] = i;
        }
    }
  if (disponibles == 0)
    {
      return NULL;
    }
  else
    {
      int costo[disponibles];
      for (i = 0; i < disponibles; i++)
        {
          costo[i] = funcion_costo(size, x_number,casillas[i], tablero);
        }
      int costo_max = INT_MIN;;
      int casilla_ganadora = INT_MIN;;
      for (i = 0; i < disponibles; i++)
```

```
        {
          if (costo[i] == 0 && x_number < size - 1)
            {
              continue;
            }
          else if (costo[i] > costo_max)
            {
              costo_max = costo[i];
              casilla_ganadora = casillas[i];
            }
          else if (costo[i] == costo_max && rand() & 1)
            {
              costo_max = costo[i];
              casilla_ganadora = casillas[i];
            }
        }
      if (casilla_ganadora == INT_MIN)
        {
          return NULL;
        }

      tablero = asigna_reina_aux(x_number, casilla_ganadora, size, tablero,1);
    }
  return tablero;
}


/**
 * @brief La función principal de la heurística.
 *
 * Asigna en cada iteración una reina usando una función de costo.
 *
 * @param size - Es el tamaño del tablero.
 * @param tablero - Es el tablero del juego.
 * @return El tablero con todas las asignaciones.
 */
int** heuristica(int size, int** tablero)
{
  int i;
  for (i = 0; i < size; i++)
    {
      tablero = asigna_reina(i, size, tablero);
      if (tablero == NULL)
        {
          return NULL;
        }
    }

  return tablero;
```

```c
}


/**
 * @brief Método main de la clase.
 *
 * @param argc - Es el número de parámetros recibidos.
 * @param argv - Son los parámetros que recibe.
 * @return 0 si la ejecución se lleva sin problemas.
 */
int main(int argc, char** argv)
{
  int size = 8;
  if(argc > 1)
    {
      int tmp = atoi(argv[1]);
      size = MAX(tmp, size);
    }

  printf("El tamaño del tablero es de (%d X %d).\n", size, size);
  srand(time(NULL));
  int** resultado = NULL;
  int intentos = 0;
  while(resultado == NULL)
    {
      intentos++;
      int **tablero = inicializa_tablero(size);
      resultado = heuristica(size, tablero);
    }
  printf("\n\n\n");

  printf("Número de intentos: %d.\n", intentos);
  imprime_tablero(size,resultado);
  return 0;
} // Fin de reinas.c
```

Apéndice C

Código Fuente

C.1. tetris

C.1.1. punto.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
class Punto:
    """Un simple punto para coordenadas."""
    def __init__(self, x, y):
        """
        Parameters
        ----------
        x : int
            La coordenada x del punto.
        y : int
            La coordenada y del punto.
         """
        self._x = x
        self._y = y

    def get_x(self):
        """
        Regresa la variable X del punto.
        """
        return self._x

    def get_y(self):
        """
        Regresa la variable Y del punto.
        """
        return self._y

    def set_x(self, valor):
```

```
        """
        Asigna la variable X del punto.

        Parameters
        ----------
        valor : int
            Es el nuevo valor de la coordenada.
        """
        self._x = valor

    def set_y(self, valor):
        """
        Asigna la variable Y del punto.

        Parameters
        ----------
        valor : int
            Es el nuevo valor de la coordenada.
        """
        self._y = valor

    def same(self, punto):
        """
        Revisa que dos puntos sean el mismo.

        Parameters
        ----------
        punto : Punto
            Es el otro punto.
        """
        return self._x == punto.get_x() and self._y == punto.get_y()

    def clona(self):
        """
        Crea un mismo punto.
        """
        return Punto(self._x, self._y)
```

C.1.2. casilla.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
class Casilla:
    """ Una casilla es un espacio atómico en un tablero"""
    def __init__(self, punto, tipo=None):
        """
        Parameters
        ----------
```

```
        punto : Punto
            Es el punto de la casilla
        tipo : Tipo
            Es el tipo que la que apareció
        """
        self._punto = punto
        self._fija = False
        self._tipo = tipo

    def get_tipo(self):
        """
        Regresa el tipo de la casilla.
        """
        return self._tipo

    def set_tipo(self, tipo):
        """
        Asigna el tipo de la casilla.

        Parameters
        ----------
        tipo : Tipo
            Es el nuevo tipo de la casilla.
        """
        self._tipo = tipo


    def get_punto(self):
        """
        Regresa el punto de la casilla.
        """
        return self._punto

    def clona(self):
        """
        Regresa una casilla clonada.
        """
        c = Casilla(self._punto.clona())
        c.set_tipo(self._tipo)
        if self.get_fija():
            c.set_fija()
        return c

    def set_punto(self, punto):
        """
        Asigna el punto de la casilla.

        Parameters
        ----------
```

```
        punto : Punto
            Es el nuevo punto de la casilla.
        """
        self._punto = punto

    def set_fija(self, fija=True):
        """
        Coloca como final la posición de la casilla.
        """
        self._fija = fija

    def get_fija(self):
        """
        Regresa el valor de la casilla en el tablero.
        """
        return self._fija
```

C.1.3. movimiento.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
from enum import Enum

class Movimiento(Enum):
    """Este enum representa los posibles movimientos de usuario
    en un juego de Tetris. """
    DER = 1
    IZQ = 2
    CAE = 3
    GIR = 4
    FIJ = 5

```

C.1.4. tipo_pieza.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
from .casilla import *
from .punto import *
from enum import Enum

class Tipo(Enum):
    """Este enum representa las posibles piezas del juego de Tetris"""
    I = 1
    RS = 2
    LG = 3
```

```
    T = 4
    RG = 5
    LS = 6
    Sq = 7

def instancia_i(pos):
    """ Regresa una lista con las casillas de la pieza I """
    # La casilla media es 0Ő00
    p2 = Punto(pos.get_x() - 1, pos.get_y())
    p3 = Punto(pos.get_x() + 1, pos.get_y())
    p4 = Punto(pos.get_x() + 2, pos.get_y())
    c1 = Casilla(p2)
    c2 = Casilla(pos)
    c3 = Casilla(p3)
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def instancia_rs(pos):
    """ Regresa una lista con las casillas de la pieza RS """
    # La casilla media es:
    #     Ő0
    #    00
    p1 = Punto(pos.get_x() - 1, pos.get_y() + 1)
    p2 = Punto(pos.get_x(), pos.get_y() + 1)
    p4 = Punto(pos.get_x() + 1, pos.get_y())
    c1 = Casilla(p1)
    c2 = Casilla(p2)
    c3 = Casilla(pos)
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def instancia_lg(pos):
    """ Regresa una lista con las casillas de la pieza LG """
    # La casilla media es:
    #    0
    #    Ő00
    p1 = Punto(pos.get_x(), pos.get_y() - 1)
    p3 = Punto(pos.get_x() + 1, pos.get_y())
    p4 = Punto(pos.get_x() + 2, pos.get_y())
    c1 = Casilla(p1)
    c2 = Casilla(pos)
    c3 = Casilla(p3)
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def instancia_t(pos):
    """ Regresa una lista con las casillas de la pieza T """
    # La casilla media es:
    #     0
```

```
    #    0Ó0
    p1 = Punto(pos.get_x() - 1, pos.get_y())
    p3 = Punto(pos.get_x(), pos.get_y() - 1)
    p4 = Punto(pos.get_x() + 1, pos.get_y())
    c1 = Casilla(p1)
    c2 = Casilla(pos)
    c3 = Casilla(p3)
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def instancia_rg(pos):
    """ Regresa una lista con las casillas de la pieza RG """
    # La casilla media es:
    #      0
    #    00Ó
    p1 = Punto(pos.get_x() - 2, pos.get_y())
    p2 = Punto(pos.get_x() - 1, pos.get_y())
    p4 = Punto(pos.get_x(), pos.get_y() - 1)
    c1 = Casilla(p1)
    c2 = Casilla(p2)
    c3 = Casilla(pos)
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def instancia_ls(pos):
    """ Regresa una lista con las casillas de la pieza LS """
    # La casilla media es:
    #    0Ó
    #     00
    p1 = Punto(pos.get_x() - 1, pos.get_y())
    p3 = Punto(pos.get_x(), pos.get_y() + 1)
    p4 = Punto(pos.get_x() + 1, pos.get_y() + 1)
    c1 = Casilla(p1)
    c2 = Casilla(pos)
    c3 = Casilla(p3)
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def instancia_sq(pos):
    """ Regresa una lista con las casillas de la pieza Sq """
    # La casilla media es:
    #    Ó0
    #    00
    p2 = Punto(pos.get_x() + 1, pos.get_y())
    p3 = Punto(pos.get_x(), pos.get_y() + 1)
    p4 = Punto(pos.get_x() + 1, pos.get_y() + 1)
    c1 = Casilla(pos)
    c2 = Casilla(p2)
    c3 = Casilla(p3)
```

```
    c4 = Casilla(p4)
    return [c1,c2,c3,c4]

def get_casillas(tipo, posicion):
    """ Esta función regresa lista de casillas dependiendo el tipo"""
    if tipo is Tipo.I:
        return instancia_i(posicion)
    elif tipo is Tipo.RS:
        return instancia_rs(posicion)
    elif tipo is Tipo.LG:
        return instancia_lg(posicion)
    elif tipo is Tipo.T:
        return instancia_t(posicion)
    elif tipo is Tipo.RG:
        return instancia_rg(posicion)
    elif tipo is Tipo.LS:
        return instancia_ls(posicion)
    else:
        return instancia_sq(posicion)

def rota_i(casillas):
    """ Regresa una lista con los puntos rotados de la pieza I."""
    # La casilla media es 0□́00
    p1 = casillas[0].get_punto()
    p2 = casillas[1].get_punto()

    if p1.get_x() == p2.get_x():
        cas = instancia_i(p2)
        np1 = cas[0].get_punto()
        np2 = cas[1].get_punto()
        np3 = cas[2].get_punto()
        np4 = cas[3].get_punto()
        return [np1, np2, np3, np4]

    np1 = Punto(p2.get_x(), p2.get_y() - 1)
    np2 = Punto(p2.get_x(), p2.get_y())
    np3 = Punto(p2.get_x(), p2.get_y() + 1)
    np4 = Punto(p2.get_x(), p2.get_y() + 2)
    return [np1, np2, np3, np4]

def rota_rs(casillas):
    """ Regresa una lista con los puntos rotados de la pieza RS."""
    # La casilla media es:
    #      □́0
    #     00
    p3 = casillas[2].get_punto()
    p4 = casillas[3].get_punto()

    if p3.get_x() == p4.get_x():
```

```
        cas = instancia_rs(p3)
        np1 = cas[0].get_punto()
        np2 = cas[1].get_punto()
        np3 = cas[2].get_punto()
        np4 = cas[3].get_punto()
        return [np1, np2, np3, np4]

    np1 = Punto(p3.get_x() - 1, p3.get_y() - 1)
    np2 = Punto(p3.get_x() - 1, p3.get_y())
    np3 = Punto(p3.get_x(), p3.get_y())
    np4 = Punto(p3.get_x(), p3.get_y() + 1)
    return [np1, np2, np3, np4]


def rota_lg(casillas):
    """ Regresa una lista con los puntos rotados de la pieza LG."""
    # La casilla media es:
    #     0
    #     Ó00
    p1 = casillas[0].get_punto()
    p2 = casillas[1].get_punto()
    p3 = casillas[2].get_punto()

    if p1.get_y() == p2.get_y():
        if p1.get_y() > p3.get_y():
            cas = instancia_lg(p2)
            np1 = cas[0].get_punto()
            np2 = cas[1].get_punto()
            np3 = cas[2].get_punto()
            np4 = cas[3].get_punto()
            return [np1, np2, np3, np4]
        np1 = Punto(p2.get_x(), p2.get_y() + 1)
        np2 = Punto(p2.get_x(), p2.get_y())
        np3 = Punto(p2.get_x() - 1, p2.get_y())
        np4 = Punto(p2.get_x() - 2, p2.get_y())
        return [np1, np2, np3, np4]
    elif p1.get_y() > p2.get_y():
        np1 = Punto(p2.get_x() - 1, p2.get_y())
        np2 = Punto(p2.get_x(), p2.get_y())
        np3 = Punto(p2.get_x(), p2.get_y() - 1)
        np4 = Punto(p2.get_x(), p2.get_y() - 2)
        return [np1, np2, np3, np4]
    else:
        np1 = Punto(p2.get_x() + 1, p2.get_y())
        np2 = Punto(p2.get_x(), p2.get_y())
        np3 = Punto(p2.get_x(), p2.get_y() + 1)
        np4 = Punto(p2.get_x(), p2.get_y() + 2)
        return [np1, np2, np3, np4]
```

```
def rota_t(casillas):
    """ Regresa una lista con los puntos rotados de la pieza T."""
    # La casilla media es:
    #     0
    #    0Ó0
    p1 = casillas[0].get_punto()
    p2 = casillas[1].get_punto()
    p3 = casillas[2].get_punto()
    p4 = casillas[3].get_punto()

    if p1.get_x() == p4.get_x():
        if p2.get_x() > p3.get_x():
            cas = instancia_t(p2)
            np1 = cas[0].get_punto()
            np2 = cas[1].get_punto()
            np3 = cas[2].get_punto()
            np4 = cas[3].get_punto()
            return [np1, np2, np3, np4]
        np1 = Punto(p2.get_x() + 1, p2.get_y())
        np2 = Punto(p2.get_x(), p2.get_y())
        np3 = Punto(p2.get_x(), p2.get_y() + 1)
        np4 = Punto(p2.get_x() - 1, p2.get_y())
        return [np1, np2, np3, np4]
    elif p3.get_y() > p2.get_y():
        np1 = Punto(p2.get_x(), p2.get_y() + 1)
        np2 = Punto(p2.get_x(), p2.get_y())
        np3 = Punto(p2.get_x() - 1, p2.get_y())
        np4 = Punto(p2.get_x(), p2.get_y() - 1)
        return [np1, np2, np3, np4]
    else:
        np1 = Punto(p2.get_x(), p2.get_y() - 1)
        np2 = Punto(p2.get_x(), p2.get_y())
        np3 = Punto(p2.get_x() + 1, p2.get_y())
        np4 = Punto(p2.get_x(), p2.get_y() + 1)
        return [np1, np2, np3, np4]

def rota_rg(casillas):
    """ Regresa una lista con los puntos rotados de la pieza RG."""
    # La casilla media es:
    #      0
    #    00Ó
    p1 = casillas[0].get_punto()
    p3 = casillas[2].get_punto()
    p4 = casillas[3].get_punto()

    if p3.get_y() == p4.get_y():
        if p1.get_y() > p3.get_y():
            cas = instancia_rg(p3)
            np1 = cas[0].get_punto()
```

```
            np2 = cas[1].get_punto()
            np3 = cas[2].get_punto()
            np4 = cas[3].get_punto()
            return [np1, np2, np3, np4]
        np1 = Punto(p3.get_x() + 2, p3.get_y())
        np2 = Punto(p3.get_x() + 1, p3.get_y())
        np3 = Punto(p3.get_x(), p3.get_y())
        np4 = Punto(p3.get_x(), p3.get_y() + 1)
        return [np1, np2, np3, np4]
    elif p3.get_y() > p4.get_y():
        np1 = Punto(p3.get_x(), p3.get_y() - 2)
        np2 = Punto(p3.get_x(), p3.get_y() - 1)
        np3 = Punto(p3.get_x(), p3.get_y())
        np4 = Punto(p3.get_x() + 1, p3.get_y())
        return [np1, np2, np3, np4]
    else:
        np1 = Punto(p3.get_x(), p3.get_y() + 2)
        np2 = Punto(p3.get_x(), p3.get_y() + 1)
        np3 = Punto(p3.get_x(), p3.get_y())
        np4 = Punto(p3.get_x() - 1, p3.get_y())
        return [np1, np2, np3, np4]

def rota_ls(casillas):
    """ Regresa una lista con los puntos rotados de la pieza LS."""
    # La casilla media es:
    #    0Ó
    #     00
    p1 = casillas[0].get_punto()
    p2 = casillas[1].get_punto()

    if p2.get_x() == p1.get_x():
        cas = instancia_ls(p2)
        np1 = cas[0].get_punto()
        np2 = cas[1].get_punto()
        np3 = cas[2].get_punto()
        np4 = cas[3].get_punto()
        return [np1, np2, np3, np4]

    np1 = Punto(p2.get_x(), p2.get_y() - 1)
    np2 = Punto(p2.get_x(), p2.get_y())
    np3 = Punto(p2.get_x() - 1, p2.get_y())
    np4 = Punto(p2.get_x() - 1, p2.get_y() + 1)
    return [np1, np2, np3, np4]

def rota_sq(casillas):
    """ Regresa una lista con los puntos rotados de la pieza Sq."""
    # La casilla media es:
    #    Ó0
    #    00
```

```python
    p1 = casillas[0].get_punto()
    p2 = casillas[1].get_punto()
    p3 = casillas[2].get_punto()
    p4 = casillas[3].get_punto()

    np1 = Punto(p1.get_x(), p1.get_y())
    np2 = Punto(p2.get_x(), p2.get_y())
    np3 = Punto(p3.get_x(), p3.get_y())
    np4 = Punto(p4.get_x(), p4.get_y())
    return [np1, np2, np3, np4]

def rota(tipo, casillas):
    """ Regresa una lista con los puntos rotados."""
    if tipo is Tipo.I:
        return rota_i(casillas)
    elif tipo is Tipo.RS:
        return rota_rs(casillas)
    elif tipo is Tipo.LG:
        return rota_lg(casillas)
    elif tipo is Tipo.T:
        return rota_t(casillas)
    elif tipo is Tipo.RG:
        return rota_rg(casillas)
    elif tipo is Tipo.LS:
        return rota_ls(casillas)
    else:
        return rota_sq(casillas)

```

C.1.5. pieza.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
import abejas_tetris.tetris
from .tipo_pieza import *

class Pieza:
    """ Representa una pieza dentro del tablero. """
    def __init__(self, tipo, posicion):
        """
        Parameters
        ----------
        tipo : Tipo
                El tipo de los 7 posibles de pieza.
        posicion : Punto
                Es un punto de tipo (X,Y).
        """
        self._tipo = tipo
```

```
        self._orientacion = 0
        self._posicion = posicion
        self._casillas = self.__get_casillas()
        for i in self._casillas:
            i.set_tipo(tipo)
        self._fijo = False

    def clona(self):
        """
        Regresa un clon del objeto
        """
        pos = Punto(self._posicion.get_x(), self._posicion.get_y())
        clone = Pieza(self._tipo, pos)
        clone.set_puntos(self.get_casillas_self())
        clone.set_orientacion(self._orientacion)
        return clone

    def get_orientacion(self):
        """
        Regresa la orientación de la pieza de forma X = (n*90)mod360
        """
        return self._orientacion

    def set_orientacion(self, orientacion):
        """
        Asigna una orientación a la pieza

        Parameters
        ----------
        orientacion : int
                Es el número modulo 360.
        """
        self._orientacion = orientacion

    def set_puntos(self, casillas):
        """
        Asigna los puntos al objeto clonando las casillas.

        Parameters
        ----------
        casillas : list(Casilla)
                Es una lista de casillas.
        """
        self._casillas[0] = casillas[0].clona()
        self._casillas[1] = casillas[1].clona()
        self._casillas[2] = casillas[2].clona()
        self._casillas[3] = casillas[3].clona()

    def get_casillas_self(self):
```

```
        """
        Regresa las casillas que tiene el objeto.
        """
        return self._casillas

    def get_puntos(self):
        """
        Regresa puntos que representan la posición de la pieza.
        """
        p1 = self._casillas[0].get_punto().clona()
        p2 = self._casillas[1].get_punto().clona()
        p3 = self._casillas[2].get_punto().clona()
        p4 = self._casillas[3].get_punto().clona()
        return [p1, p2, p3, p4]

    def rota(self):
        """
        Realiza la rotación de la pieza incluyendo las casillas.
        """
        self._orientacion = (self._orientacion + 90) % 360
        puntos = rota(self._tipo, self._casillas)
        i = 0
        while i < 4:
            self._casillas[i].set_punto(puntos[i])
            i = i + 1

    def mueve_derecha(self):
        """
        Mueve las casillas de la pieza hacia la derecha.
        """
        for i in self._casillas:
            punto = i.get_punto()
            punto.set_x(punto.get_x() + 1)

    def mueve_izquierda(self):
        """
        Mueve las casillas de la pieza hacia la izquierda.
        """
        for i in self._casillas:
            punto = i.get_punto()
            punto.set_x(punto.get_x() - 1)

    def baja(self):
        """
        Mueve las casillas de la pieza hacia abajo.
        Parameters
        ----------
        """
        for i in self._casillas:
```

```python
            punto = i.get_punto()
            punto.set_y(punto.get_y() + 1)

    # Este método sólo se usa por las abejas observadoras
    # para regresar a un estado previo.
    def sube(self):
        """
        Mueve las casillas de la pieza hacia arriba.
        """
        for i in self._casillas:
            punto = i.get_punto()
            punto.set_y(punto.get_y() - 1)

    def fija(self):
        """
        Cambia el estado de todas las casillas.
        """
        return self._fijo

    def casillas(self):
        """
        Regresa las casillas de la pieza.
        """
        return self._casillas

    def get_tipo(self):
        """
        Regresa el tipo de la pieza.
        """
        return self._tipo

    # Funciones auxiliares:

    def __get_casillas(self):
        return get_casillas(self._tipo, self._posicion)
```

C.1.6. tablero.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
from abejas_tetris.tetris.punto import *
from abejas_tetris.tetris.pieza import *
from abejas_tetris.tetris.movimiento import *
from abejas_tetris.tetris.tipo_pieza import *

class Tablero:
    def __init__(self, x, y):
        """
```

```
        Parameters
        ----------
        x : int
                Es el ancho del tablero.
        y : int
               Es el alto del tablero.
        """
        self._x = x
        self._y = y
        self._pieza = None
        self._pieza_anterior = None
        self._limpia_automatico = True
        self._tablero = []
        self._num_tetris = 0
        for i in range(self._x):
            fila = []
            for j in range(self._y):
                fila.append(None)
            self._tablero.append(fila)

    def get_casilla(self, x, y):
        """
        Regresa lo que haya en la posición dada.

        Parameters
        ----------
        x : int
            Es el ancho del tablero.
        y : int
            Es el alto del tablero.
        """
        return self._tablero[x][y]

    def set_limpieza_automatica(self, limpieza=True):
        """
        Asigna la variable para que el tablero se limpie solo.

        Parameters
        ----------
        limpieza=True : bool
            Es la bandera para que se limpie o no solo.
        """
        self._limpia_automatico = limpieza

    def get_limpieza(self):
        """
        Regresa la bandera para saber si eliminar o no
        los tetrominoes de forma automática.
        """
```

```
        return self._limpia_automatico

    def clona(self):
        """
        Regresa un objeto clon.
        """
        clon = Tablero(self._x, self._y)
        clon.copia_tablero(self._tablero)
        if self._pieza != None:
            clon.asigna_pieza_clonada(self._pieza.clona())
        if self._pieza_anterior != None:
            clon.asigna_pieza_anterior(self._pieza_anterior.clona())
        clon.set_num_tetris(self._num_tetris)
        clon.set_limpieza_automatica(self._limpia_automatico)
        return clon

    def copia_tablero(self, tablero):
        """
        Dado un tablero, copia lo que haya en él sobre el objeto.

        Parameters
        ----------
        tablero : list(list(Casilla))
            Es el tablero a copiar.
        """
        for i in range(self._x):
            for j in range(self._y):
                if tablero[i][j] != None:
                    self._tablero[i][j] = tablero[i][j].clona()
                else:
                    self._tablero[i][j] = None

    def asigna_pieza_anterior(self, pieza):
        """
        Dado una pieza, asigna la pieza anterior.

        Parameters
        ----------
        pieza : Pieza
            Es la pieza anterior jugada.
        """
        self._pieza_anterior = pieza

    def asigna_pieza_clonada(self, pieza):
        """
        Asigna pieza actual.

        Parameters
        ----------
```

```
        pieza : Pieza
            Asigna una pieza previamente clonada.
        """
        self._pieza = pieza
        self.__actualiza_pieza([])

    def punto_inicial(self):
        """
        Regresa el punto inicial de la pieza.
        """
        return Punto(int(self._x / 2), 2)

    def set_pieza(self, tipo):
        """
        Asigna una nueva pieza.

        Parameters
        ----------
        tipo : Tipo
            Es el tipo de la nueva pieza.
        """
        p = Pieza(tipo, self.punto_inicial())
        for i in p.casillas():
            punto = i.get_punto()
            x = punto.get_x()
            y = punto.get_y()
            if self._tablero[x][y] != None:
                return False
        self._pieza = p
        self.__actualiza_pieza([])
        return True

    def requiere_pieza(self):
        """
        Regresa verdadero si el tablero no tiene pieza actual.
        """
        return self._pieza == None

    def altura_maxima(self):
        """
        Regresa la altura de la pieza más alta.
        """
        altura = 0
        x = 0
        while x < self._x:
            y = 0
            while y < self._y:
                if self._tablero[x][y] != None:
                    altura_local = self._y - y
```

```
                    if altura <= altura_local:
                        altura = altura_local
                y = y + 1
            x = x + 1
        return (altura)

    def altura_minima(self):
        """
        Regresa la altura de la pieza más al fondo.
        """
        y = self._y - 1
        while y >= 0:
            x = 0
            while x < self._x:
                if self._tablero[x][y] == None:
                    return self._y - (y + 1)
                x = x +1
            y = y - 1
        return self._y

    def movimiento_valido(self, move):
        """
        Regresa True si el movimiento que recibe es jugable.

        Parameters
        ----------
        move : Movimiento
            Es un movimiento a revisar si es válido.
        """
        if self._pieza == None:
            return False
        if move == Movimiento.CAE:
            return self.__check_cae()
        elif move == Movimiento.DER:
            return self.__check_der()
        elif move == Movimiento.IZQ:
            return self.__check_izq()
        elif move == Movimiento.GIR:
            return self.__check_gir()
        else:
            return False

    def juega_movimiento(self, move):
        """
        Mueve el estado del tablero un movimiento más adelante.

        Parameters
        ----------
        move : Movimiento
```

```python
            Es el movimiento jugado.
        """
        if self._pieza == None:
            return False
        if self.movimiento_valido(move):
            puntos_previos = self._pieza.get_puntos()
            if move == Movimiento.CAE:
                self._pieza.baja()
            elif move == Movimiento.DER:
                self._pieza.mueve_derecha()
            elif move == Movimiento.IZQ:
                self._pieza.mueve_izquierda()
            else:
                self._pieza.rota()
            self.__actualiza_pieza(puntos_previos)
            return True
        else:
            return False

    def restaura_pieza(self):
        """ Si la pieza es vacía vuelve a colocar una anterior"""
        if self._pieza == None and self._pieza_anterior != None:
            self._pieza = self._pieza_anterior
            for i in self._pieza.casillas():
                i.set_fija(False)
            self.__actualiza_pieza([])

    def juega_movimiento_inverso(self, move):
        """
        Mueve el estado del tablero un movimiento hacia atrás.

        Parameters
        ----------
        move : Movimiento
            Es el movimiento que se tiene que retirar.
        """
        self.restaura_pieza()
        if self._pieza == None:
            return False
        puntos_previos = self._pieza.get_puntos()
        if move == Movimiento.CAE and self.__check_cae(True):
            self._pieza.sube()
        elif move == Movimiento.DER and self.__check_izq():
            self._pieza.mueve_izquierda()
        elif move == Movimiento.IZQ and self.__check_der():
            self._pieza.mueve_derecha()
        elif move == Movimiento.FIJ:
            raise Exception("No se puede deshacer fijar pieza")
        elif move == Movimiento.GIR:
```

```
            tipo = self._pieza.get_tipo()
            tres_giro = tipo == Tipo.LG or tipo == Tipo.T or tipo == Tipo.RG
            if tres_giro:
                self._pieza.rota()
                self._pieza.rota()
                if self.__check_gir():
                    self._pieza.rota()
                else:
                    self._pieza.rota()
                    self._pieza.rota()
                    return False
            elif self.__check_gir():
                self._pieza.rota()
            else:
                return False
        else:
            return False
        self.__actualiza_pieza(puntos_previos)
        return True

    def limpia(self):
        """
        Limpia las filas llenas.
        """
        if not self._limpia_automatico:
            self._cuenta_filas_removidas()
            self.__revisa_filas()

    def puede_limpiar(self):
        """
        Regresa las filas que pueden ser eliminadas.
        """
        cuenta_filas = 0
        limite_x = self._x
        limite_y = self._y
        columna = limite_y - 1
        while columna >= 0:
            hay_none = False
            fila = 0
            while fila < limite_x:
                if self._tablero[fila][columna] == None:
                    hay_none = True
                fila = fila + 1
            if not hay_none:
                cuenta_filas = cuenta_filas + 1
            columna = columna - 1
        return cuenta_filas

    def puede_fijar(self):
```

```
        """
        Regresa True si la pieza actual puede fijarse en esta posición.
        """
        if self._pieza == None:
            return False
        for i in self._pieza.casillas():
            punto = i.get_punto()
            x = punto.get_x()
            y = punto.get_y()
            if self._y == y + 1:
                return True
            if self._tablero[x][y + 1] != None:
                if self._tablero[x][y + 1].get_fija():
                    return True
        return False

    def fijar(self):
        """
        Fija la pieza actual en su posición actual.
        """
        if self.puede_fijar():
            if self._pieza == None:
                return False
            self._pieza_anterior = self._pieza
            for i in self._pieza.casillas():
                i.set_fija()
            self._pieza = None
            if self._limpia_automatico:
                self._cuenta_filas_removidas()
                self.__revisa_filas()
            return True
        else:
            return False

    def print(self):
        """
        Imprime en la consola una representación más
        amigable del tablero.
        """
        blanco ="#"
        for i in range(self._y):
            cad = ''
            spa = '_'
            for j in range(self._x):
                cad = cad + '|'
                spa = spa + '__'
                if self._tablero[j][i] != None:
                    cad = cad + blanco
                else:
```

```
                    cad = cad + " "
            print(cad)
            print(spa)

    def cuenta_espacios(self, fila):
        """
        Nos dice cuantas casillas con espacios hay en cierta fila.

        Parameters
        ----------
        fila : int
            Es la fila a revisar.
        """
        fila_real = self._y - (fila + 1)
        espacios = 0
        x = 0
        while x < self._x:
            if self._tablero[x][fila_real] == None:
                espacios = espacios + 1
            x = x + 1
        return espacios

    def cuenta_atrapados(self):
        """
        Nos dice cuantas casillas rodeadas de fichas hay.
        """
        x = 0
        atrapado = 0
        while x < self._x:
            y = 0
            while y < self._y:
                rodeado = True
                if self._tablero[x][y] == None:
                    if x - 1 >= 0:
                        rodeado = rodeado and self._tablero[x-1][y] != None
                    if x + 1 < self._x:
                        rodeado = rodeado and self._tablero[x+1][y] != None
                    if y - 1 >= 0:
                        rodeado = rodeado and self._tablero[x][y-1] != None
                    if y + 1 < self._y:
                        rodeado = rodeado and self._tablero[x][y+1] != None
                    if rodeado:
                        atrapado = atrapado + 1
                y = y + 1
            x = x +1
        return atrapado


    def cuenta_cubiertos(self):
```

```
        """
        No dice cuantas casillas vacías tiene alguna no vacia arriba.
        """
        cubiertos = 0
        for i in range(self._x):
            for j in range(self._y):
                if self._tablero[i][j] == None:
                    cubiertos = cubiertos + self._helper_cubiertos(i, j)
        return cubiertos

    def num_tetris(self):
        """
        Nos dice cuántas filas de han ido hasta ahora.
        """
        return self._num_tetris

    def set_num_tetris(self, num):
        """
        Asigna un número de filas retiradas.

        Parameters
        ----------
        num : int
            Es el nuevo número de filas retiradas del tablero.
        """
        self._num_tetris = num

    # Funciones auxiliares:

    # Regresa 1 si la casilla esta cubierta por otra.
    def _helper_cubiertos(self, x, y):
        if y < 0:
            return 0
        if self._tablero[x][y] != None:
            return 1
        return self._helper_cubiertos(x, y - 1)

    # Cuenta si hay filas que quitar.
    def _cuenta_filas_removidas(self):
        limite_x = self._x
        limite_y = self._y
        columna = limite_y - 1
        while columna >= 0:
            hay_none = False
            fila = 0
            while fila < limite_x:
                if self._tablero[fila][columna] == None:
                    hay_none = True
                fila = fila + 1
```

```
            if not hay_none:
                self._num_tetris = self._num_tetris + 1
            columna = columna - 1

    # Revisa si puede caer la pieza actual.
    def __check_cae(self, inverso=False):
        for i in self._pieza.casillas():
            punto = i.get_punto()
            x = punto.get_x()
            y = punto.get_y()
            if inverso:
                if y == 0:
                    return False
            else:
                if y + 1 >= self._y:
                    return False
            if inverso:
                if self._tablero[x][y - 1] != None:
                    if self._tablero[x][y - 1].get_fija():
                        return False
            else:
                if self._tablero[x][y + 1] != None:
                    if self._tablero[x][y + 1].get_fija():
                        return False
        return True

    # Revisa si puede moverse a la derecha la pieza actual.
    def __check_der(self):
        for i in self._pieza.casillas():
            punto = i.get_punto()
            x = punto.get_x()
            y = punto.get_y()
            if self._x <= x + 1:
                return False
            if self._tablero[x + 1][y] != None:
                if self._tablero[x + 1][y].get_fija():
                    return False
        return True

    # Revisa si puede moverse a la izquierda la pieza actual.
    def __check_izq(self):
        for i in self._pieza.casillas():
            punto = i.get_punto()
            x = punto.get_x()
            y = punto.get_y()
            if x - 1 < 0:
                return False
            if self._tablero[x - 1][y] != None:
                if self._tablero[x - 1][y].get_fija():
```

```
                    return False
        return True

    # Revisa si puede rotar la pieza actual.
    def __check_gir(self):
        puntos = rota(self._pieza.get_tipo(), self._pieza.casillas())
        for i in puntos:
            if i.get_x() >= self._x or i.get_x() < 0:
                return False
            elif i.get_y() >= self._y or i.get_y() < 0:
                return False
            else:
                valor = self._tablero[i.get_x()][i.get_y()]
                if valor != None:
                    if valor.get_fija():
                        return False
        return True

    # Actualiza los puntos de la pieza actual.
    def __actualiza_pieza(self, puntos_viejos):
        for i in puntos_viejos:
            x = i.get_x()
            y = i.get_y()
            self._tablero[x][y] = None
        for i in self._pieza.casillas():
            punto = i.get_punto()
            x = punto.get_x()
            y = punto.get_y()
            self._tablero[x][y] = i

    # Revisa las filas si hay que eliminar.
    def __revisa_filas(self):
        limite_x = self._x
        limite_y = self._y
        fila = limite_y - 1
        while fila >= 0:
            hay_none = False
            columna = 0
            while columna < limite_x:
                if self._tablero[columna][fila] == None:
                    hay_none = True
                columna = columna + 1
            if not hay_none:
                self.__limpia(fila)
                fila = fila + 1
            fila = fila - 1

    # Elimina todos las casillas de una fila.
    def __limpia(self, fil):
```

```
        limite_x = self._x

        if self._pieza_anterior != None:
            eliminar_anterior = False
            for i in self._pieza_anterior.casillas():
                punto = i.get_punto()
                y = punto.get_y()
                eliminar_anterior = eliminar_anterior \
                    or y == fil \
                    or y + 1 == 0
                punto.set_y(y + 1)
            if eliminar_anterior:
                self._pieza_anterior = None

        for i in range(limite_x):
            self._tablero[i][fil] = None
        while fil > 0:
            for i in range(limite_x):
                self._tablero[i][fil] = self._tablero[i][fil - 1]
            fil = fil - 1
        for i in range(limite_x):
            self._tablero[i][0] = None
```

C.1.7. tetris.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"

from abejas_tetris.tetris.tablero import *
from abejas_tetris.tetris.indexer import *


from abejas_tetris.my_random import get_random, get_randrange, get_randbits

class Tetris:
    """ Representa un juego de tetris con todos sus componentes."""
    def __init__(self, x, y, tablero=None):
        """
        Parameters
        ----------
        x : int
                Es el ancho del tablero.
        y : int
               Es el alto del tablero.
        tablero : Tablero
                Es un tablero por si el objeto es clonado.
        """
        if tablero == None:
```

```
            self._tablero = Tablero(x,y)
        else:
            self._tablero = tablero
        self._x = x
        self._y = y
        self._historial = []
        self._piezas_jugadas = 0
        self._game_over = False
        self.id = get_index()

    def desactiva_limpieza_automatica(self):
        """
        Quita la remoción automática de filas llenas.
        """
        self._tablero.set_limpieza_automatica(False)

    def activa_limpieza_automatica(self):
        """
        Activa la remoción automática de filas llenas.
        """
        self._tablero.set_limpieza_automatica()

    def get_altura(self):
        """
        Regresa la altura total del tablero.
        """
        return self._y

    def get_ancho(self):
        """
        Regresa el ancho total del tablero.
        """
        return self._x

    def set_game_over(self, flag):
        self._game_over = flag

    def game_over(self):
        """
        Bandera que nos dice si ya perdimos.
        """
        return self._game_over

    def set_piezas_jugadas(self, number):
        """
        Asigna un número de piezas jugadas.

        Parameters
        ----------
```

```
        number : int
            Es el número total de piezas jugadas.
        """
        self._piezas_jugadas = number

    def altura_maxima(self):
        """
        Regresa la altura máxima actual del tablero.
        """
        return self._tablero.altura_maxima()

    def altura_minima(self):
        """
        Regresa la altura mínima actual del tablero.
        """
        return self._tablero.altura_minima()

    def ultimo_movimiento(self):
        """
        Regresa el último movimiento hecho en el juego.
        """
        return self._historial[len(self._historial) - 1]

    def clona(self):
        """
        Regresa una instancia idéntica del juego de Tetris.
        """
        clon = Tetris(self._x, self._y, self._tablero.clona())
        historial_clone = []
        for i in range(len(self._historial)):
            historial_clone.append(self._historial[i])
        clon.set_historial(historial_clone)
        clon.set_piezas_jugadas(self._piezas_jugadas)
        clon.set_game_over(self._game_over)
        if not self._tablero.get_limpieza():
            clon.desactiva_limpieza_automatica()
        return clon

    def set_pieza(self, tipo=None):
        """
        Asigna una pieza nueva.

        Parameters
        ----------
        tipo : Tipo
            Es el tipo de la pieza a jugar.
        """
        if self._tablero.requiere_pieza() and not self._game_over:
            if tipo == None:
```

```
                piezas = [Tipo.I, Tipo.LG, Tipo.LS, \
                    Tipo.T, Tipo.RS, Tipo.RG, Tipo.Sq]
                tipo = piezas[get_randrange(len(piezas))]
            self._piezas_jugadas = self._piezas_jugadas + 1
            self._game_over = not self._tablero.set_pieza(tipo)
            return self._game_over
        return False

    def puede_fijar(self):
        """
        Nos dice si el tablero puede fijar la pieza actual.
        """
        return self._tablero.puede_fijar()

    '''
    Para la vista
    '''
    def mueve(self, move):
        """
        Mueve una pieza en el tablero.

        Parameters
        ----------
        move : Movimiento
            Es el movimiento del tablero.
        """
        if move == None:
            raise Exception()
        elif move == Movimiento.FIJ:
            return False
        elif self._tablero.juega_movimiento(move):
            self._historial.append(move)
            return True
        else:
            return False

    '''
    Para la vista
    '''
    def fija(self):
        """
        Fija la pieza actual en el tablero.
        """
        if self._tablero.puede_fijar():
            if self._tablero.fijar():
                self._historial.append(Movimiento.FIJ)
                return True
            else:
                return False
```

```python
        else:
            return False

    def mueve_o_fija(self, move=None):
        """
        Realiza una acción en el tablero para avanzar el juego.

        Parameters
        ----------
        move : Movimiento
            Si es que pasan un movimiento, se realiza el movimiento.
        """
        moves = [Movimiento.CAE, Movimiento.DER, Movimiento.IZQ, Movimiento.GIR]
        moves_posibles = []
        for i in moves:
            if self._tablero.movimiento_valido(i):
                moves_posibles.append(i)
        fijar = self._tablero.puede_fijar()
        if fijar and move == Movimiento.FIJ:
            if self._tablero.fijar():
                self._historial.append(Movimiento.FIJ)
                return True
            return False
        if len(moves_posibles) == 0 and fijar:
            if self._tablero.fijar():
                self._historial.append(Movimiento.FIJ)
                return True
            return False
        elif len(moves_posibles) == 0:
            self._game_over = True
            return False
        elif fijar:
            if move == None:
                move = moves_posibles[get_randrange(len(moves_posibles))]
            # El 0.3 funciona bastante bien
            if get_random() < 0.3:
                if self._tablero.fijar():
                    self._historial.append(Movimiento.FIJ)
                    return True
                return False
            else:
                if self._tablero.juega_movimiento(move):
                    self._historial.append(move)
                    return True
                return False
        else:
            if move == None:
                move = moves_posibles[get_randrange(len(moves_posibles))]
            if self._tablero.juega_movimiento(move):
```

```
                self._historial.append(move)
                return True
            return False

    def siguiente_random(self, tipo=None, move=None):
        """
        Juega un movimiento para avanzar en el tiempo de juego.

        Parameters
        ----------
        tipo : Tipo
            Es el tipo de pieza siguiente a jugar si se necesita.
        move : Movimiento
            Es el movimiento a jugar.
        """
        if self._tablero.requiere_pieza():
            if tipo == None:
                piezas = [Tipo.I, Tipo.LG, Tipo.LS, \
                    Tipo.T, Tipo.RS, Tipo.RG, Tipo.Sq]
                tipo = piezas[get_randrange(len(piezas))]
            self._piezas_jugadas = self._piezas_jugadas + 1
            return self._tablero.set_pieza(tipo)
        moves = [Movimiento.CAE, Movimiento.DER, Movimiento.IZQ, Movimiento.GIR]
        moves_posibles = []
        for i in moves:
            if self._tablero.movimiento_valido(i):
                moves_posibles.append(i)
        fijar = self._tablero.puede_fijar()
        if len(moves_posibles) == 0 and fijar:
            self._tablero.fijar()
            self._historial.append(Movimiento.FIJ)
            return True
        elif len(moves_posibles) == 0:
            self._game_over = True
            return False
        elif fijar:
            if move == None:
                move = moves_posibles[get_randrange(len(moves_posibles))]
            if get_random() < 0.3:
                self._tablero.fijar()
                self._historial.append(Movimiento.FIJ)
                return True
            else:
                self._historial.append(move)
                self._tablero.juega_movimiento(move)
                return True
        else:
            if move == None:
                move = moves_posibles[get_randrange(len(moves_posibles))]
```

```
            self._historial.append(move)
            self._tablero.juega_movimiento(move)
            return True

    def limpia(self):
        """
        Limpia el tablero de ser necesario, fila por fila.
        """
        self._tablero.limpia()

    def puede_limpiar(self):
        """
        Regresa la cantidad de filas que se pueden eliminar.
        """
        return self._tablero.puede_limpiar()

    def get_casilla(self, x, y):
        """
        Regresa lo que se encuentre en la casilla (X,Y).

        Parameters
        ----------
        x : int
            Es el ancho a revisar.
        y : int
            Es el alto a revisar.
        """
        return self._tablero.get_casilla(x,y)

    def piezas_jugadas(self):
        """
        Regresa el número total de piezas jugadas.
        """
        return self._piezas_jugadas

    def set_historial(self, historial):
        """
        Asigna un historial a nuestra partida.

        Parameters
        ----------
        historial : list(Movimiento)
            Es una lista de movimientos previos jugados.
        """
        self._historial = historial

    def get_historial(self):
        """
        Regresa una lista de movimientos previos jugados.
```

```
        """
        return self._historial

    def elimina_historial(self, delta=1):
        """
        Elimina un número delta de movimientos del historial.

        Parameters
        ----------
        delta : float
            Es la variable que dice que tanto nos
            alejaremos de la fuente original.
        """
        primer_fija_visto = False
        while get_random() > delta and len(self._historial) > 0:
            mov = self._historial.pop()
            if mov == Movimiento.FIJ and primer_fija_visto:
                self._historial.append(mov)
                return None
            elif mov == Movimiento.FIJ:
                primer_fija_visto = True
                continue
            else:
                valor = self._tablero.juega_movimiento_inverso(mov)
                if not valor:
                    return None

    def num_movimientos(self):
        """
        Nos dice el número de movimientos que se han hecho hasta ahora.
        """
        return len(self._historial)

    def requiere_pieza(self):
        """
        Regresa True si el juego necesita una pieza para continuar.
        """
        return self._tablero.requiere_pieza()

    def imprime_tablero(self):
        """
        Imprime en la consola una representación del tablero.
        """
        self._tablero.print()

    def movimiento_valido(self, move):
        """
        Regresa True si el movimiento recibido es válido para el juego.

```

```
        Parameters
        ----------
        move : Movimiento
            Es el movimiento a revisar.
        """
        return self._tablero.movimiento_valido(move)

    def cuenta_espacios(self, fila):
        """
        Cuenta cuantas casillas en blanco hay en una fila.

        Parameters
        ----------
        fila : int
            Es la fila a revisar.
        """
        return self._tablero.cuenta_espacios(fila)

    def cuenta_atrapados(self):
        """
        Cuenta cuantas casillas None están rodeadas.
        """
        return self._tablero.cuenta_atrapados()

    def cuenta_cubiertos(self):
        """
        Cuenta cuantas casillas None tienen arriba de ellas una no None.
        """
        return self._tablero.cuenta_cubiertos()

    def num_tetris(self):
        """
        Nos dice cuántas filas se han desaparecido hasta este punto.
        """
        return self._tablero.num_tetris()

    def __hash__(self):
        """
        Se sobrescribe el método hash para asegurar la reproducción
        del programa con las semillas.
        """
        return self.id

    def __eq__(self, other):
        """
        Se sobrescribe el método eq para comparar tetris por id.
        """
        if not isinstance(other, Tetris):
            return NotImplemented
```

```
            return self.id == other.id

    def __ne__(self, other):
        """
        Se sobrescribe el método ne para comparar tetris por id.
        """
        if not isinstance(other, Tetris):
            return NotImplemented
        return not self.__eq__(other)
```

C.2. abc

C.2.1. tipo_abeja.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
from enum import Enum

class Tipo_Abeja(Enum):
    """Este enum representa los 3 tipos de Abejas que hay."""
    EMP = 1
    EXP = 2
    OBS = 3
```

C.2.2. abeja.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"

from .tipo_abeja import *

class Abeja():
    """ Una representación de una abeja que trabaja en una colmena."""
    def __init__(self, tipo=None, id=0, delta=0):
        """
        Parameters
        ----------
        tipo : Tipo_Abeja
            Es uno de los tres tipos que puede ser.
        id : int
            Es el identificador único de cada abrja.
        delta : float
            Es el número que las abejas observadoras se alejarán.
        """
        self._tipo = tipo
```

```
        self._id = id
        self._fuente = None
        self._limite = 0
        self._delta = delta
        self._busca_fuente = None
        self._observadoras = None
        self._nectar = None
        self._explotar = None

    def asigna_funciones(self, fuente, observadoras, nectar, explotar):
        """
        Asigna las funciones que necesitan las abejas para trabajar.

        Parameters
        ----------
        fuente : function
            Es la función que busca una fuente.
        observadoras : function
            Es la función que usarán las observadoras para evaluar vecindades.
        nectar : function
            Es la función principal de evaluación de las fuentes.
        explotar : function
            Es la función que trabajará una fuente hasta que se agote.
        """
        self._busca_fuente = fuente
        self._observadoras = observadoras
        self._nectar = nectar
        self._explotar = explotar

    def get_tipo(self):
        """
        Regresa el tipo de la abeja.
        """
        return self._tipo

    def set_tipo(self, tipo):
        """
        Asigna un tipo a la abeja.

        Parameters
        ----------
        tipo : Tipo_Abeja
            Es el nuevo tipo de la abeja.
        """
        self._tipo = tipo
        self._limite = 0

    def get_id(self):
        """
```

```
        Regresa el id único de la abeja.
        """
        return self._id

    def set_fuente(self, fuente):
        """
        Asigna una nueva fuente a la abeja.

        Parameters
        ----------
        fuente : T
            La nueva fuente que la abeja trabajará.
        """
        self._fuente = fuente
        self._limite = 0

    def get_fuente(self):
        """
        Regresa la fuente actual de la abeja.
        """
        return self._fuente

    def observa_solucion(self):
        """
        Si es abeja observadora, corre su función con su fuente.
        """
        if not self._tipo == Tipo_Abeja.OBS:
            raise Exception("Abeja no debe observar")
        return self._observadoras(self._fuente, self._delta)

    def explota_fuente(self):
        """
        Si es abeja empleada, corre su función con su fuente.
        """
        if not self._tipo == Tipo_Abeja.EMP:
            raise Exception("Abeja no debe explotar")
        return self._explotar(self._fuente)

    def busca_fuente(self):
        """
        Si es abeja exploradora, corre su función con su fuente.
        """
        if not self._tipo == Tipo_Abeja.EXP:
            raise Exception("Abeja no debe explorar")
        self._busca_fuente(self._fuente)

    def get_nectar(self):
        """
        Evalúa con la función de néctar la fuente de la abeja.
```

```
        """
        if not self._tipo == Tipo_Abeja.EMP:
            raise Exception("Abeja no debe evaluar")
        return self._nectar(self._fuente)

    def get_limite(self):
        """
        Regresa el límite de la abeja sobre la fuente actual.
        """
        return self._limite

    def incrementa_iteracion(self):
        """
        Incrementa en uno el límite sobre las fuentes trabajadas.
        """
        self._limite = self._limite + 1

    def __hash__(self):
        """
        Se sobrescribe el método hash para asegurar la reproducción
        del programa con las semillas.
        """
        return self._id

    def __eq__(self, other):
        """
        Se sobrescribe el método eq para comparar tetris por id.
        """
        if not isinstance(other, Abeja):
            return NotImplemented
        return self._id == other.get_id()

    def __ne__(self, other):
        """
        Se sobrescribe el método ne para comparar tetris por id.
        """
        if not isinstance(other, Abeja):
            return NotImplemented
        return not self.__eq__(other)
```

C.2.3. colmena.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"

from .abeja import *
from .tipo_abeja import *
from abejas_tetris.my_random import get_random, get_randrange, get_randbits
```

```python


class Colmena():
    """ El conjunto de información que todas las abejas necesitan. """
    def __init__(self, size, fuente_inicial, limite, delta_obs):
        """
        Parameters
        ----------
        size : int
            Es el tamaño de la colmena.
        fuente_inicial : T
            Es de donde partirán todas las abejas.
        limite : int
            Es el número máximo de veces que una abeja visita una fuente.
        delta_obs : float
            Es el número que las abejas observadoras se alejarán de su fuente.
        """
        self._size = size
        self._fuente_ini = fuente_inicial.clona()
        # abejas : int -> Abeja
        self._abejas = {}
        # exploradoras : int -> Abeja
        self._exploradoras = {}
        # observadoras: int -> Abeja
        self._observadoras = {}
        # empleadas: int -> Abeja
        self._empleadas = {}
        # T -> int
        self._fuente_abeja = {}
        self._limite = limite
        # T -> int
        self._fuentes = {}
        self._suma_fuentes = 0
        self._delta_observacion = delta_obs
        self._busca_fuente = None
        self._observadoras_fun = None
        self._nectar_fun = None
        self._termino_iteracion = None
        self._explotar_fuente_fun = None
        self._funcion_comparativa = None

    def set_funcion_nectar(self, calcular_nectar_function):
        """
        Asigna la función para evaluar el néctar.

        Parameters
        ----------
        calcular_nectar_function : function
            Es la función a asignar.
```

```
        """
        self._nectar_fun = calcular_nectar_function

    def set_funcion_buscar_fuente(self, busca_fuente_function):
        """
        Asigna la función para buscar una fuente.

        Parameters
        ----------
        calcular_nectar_function : function
            Es la función a asignar.
        """
        self._busca_fuente = busca_fuente_function

    def set_funcion_observacion(self, determina_observacion_function):
        """
        Asigna la función para que las observadoras busquen fuentes cercanas.

        Parameters
        ----------
        calcular_nectar_function : function
            Es la función a asignar.
        """
        self._observadoras_fun = determina_observacion_function

    def set_funcion_explotar_fuente(self, explorar_fuente):
        """
        Asigna la función para seguir explorando una fuente.

        Parameters
        ----------
        calcular_nectar_function : function
            Es la función a asignar.
        """
        self._explotar_fuente_fun = explorar_fuente

    def set_funcion_termino_iteracion(self, termino_iteracion):
        """
        Asigna la función para al finalizar una iteración, evaluar.

        Parameters
        ----------
        calcular_nectar_function : function
            Es la función a asignar.
        """
        self._termino_iteracion = termino_iteracion

    def set_funcion_comparativa(self, funcion_comparativa):
        """
```

```
        Asigna la función para comparar soluciones entre ellas.

        Parameters
        ----------
        funcion_comparativa : function
            Es la función a asignar.
        """
        self._funcion_comparativa = funcion_comparativa

    def actualiza_fuente_inicial(self, fuente_inicial):
        """
        Actualiza la fuente inicial que es de donde parten las exploradoras.

        Parameters
        ----------
        fuente_inicial : T
            Es la nueva fuente a partir.
        """
        self._fuente_ini = fuente_inicial.clona()

    def inizializa_abejas(self):
        """
        Crea una colmena con abejas que esperen ser llamadas.
        """
        mitad = int(self._size / 2)
        id = 0
        for _ in range(mitad):
            abja = Abeja(Tipo_Abeja.EXP,id, self._delta_observacion)
            self._abejas[id] = abja
            self._fuente_abeja[abja.get_fuente()] = id
            self._exploradoras[abja.get_id()] = abja
            self._asigna_funciones(id)
            id = id + 1
        for _ in range(self._size - mitad):
            abja = Abeja(Tipo_Abeja.OBS,id, self._delta_observacion)
            self._abejas[id] = abja
            self._observadoras[id] = abja
            self._asigna_funciones(id)
            id = id + 1

    def get_nectar_actual(self):
        """
        Regresa el valor de la mejor solución.
        """
        if self._fuente_ini == None:
            return 0
        return self._nectar_fun(self._fuente_ini)

    def itera_colmena(self):
```

```
        """
        Itera sólo una ocasión a todas las abejas para que trabajen.
        """
        self._suma_fuentes = 0
        self._itera_exploradoras()
        self._llamada_post_iteracion()
        self._itera_empleadas()
        self._itera_observadoras()
        self.actualiza_fuente_inicial(self.get_mejor_solucion())
        self._llamada_post_iteracion()


    def get_solucion_final(self):
        """
        Regresa el ultimo resultado con la mayor calificación.
        """
        if self._fuente_ini == None:
            self._fuente_ini = self.get_mejor_solucion()
        return self._fuente_ini

    def get_mejor_solucion(self):
        """
        Regresa la mejor solución hasta ahora.
        """
        solucion = None
        valor_max = 0

        for i in self._fuentes:
            valor = self._nectar_fun(i)
            if valor > valor_max or solucion == None:
                solucion = i
                valor_max = valor

        if self._funcion_comparativa != None and False:
            for i in self._fuentes:
                if self._funcion_comparativa(solucion) < self._funcion_comparativa(i):
                    solucion = i
        return solucion.clona()

    # Funciones auxiliares:

    # Asigna las funciones de la colmena a todas las abejas.
    def _asigna_funciones(self, id):
        abja = self._abejas[id]
        abja.asigna_funciones(self._busca_fuente, self._observadoras_fun, \
             self._nectar_fun, self._explotar_fuente_fun)

    # Esta función transforma a todas las abejas exploradoras a una con fuente.
    def _itera_exploradoras(self):
```

```
        eliminar = []
        for id in self._exploradoras:
            abja = self._abejas[id]
            if abja.get_fuente() != None:
                del self._fuentes[abja.get_fuente()]
                del self._fuente_abeja[abja.get_fuente()]
            abja.set_fuente(self._fuente_ini.clona())
            self._fuente_abeja[abja.get_fuente()] = id
            abja.busca_fuente()
            abja.set_tipo(Tipo_Abeja.EMP)
            eliminar.append(id)
            self._empleadas[id] = abja
            self._fuentes[abja.get_fuente()] = \
                self._nectar_fun(abja.get_fuente())
        for id in eliminar:
            del self._exploradoras[id]

    # Avanzamos en el tiempo las fuentes para que califiquemos su desempeño.
    def _itera_empleadas(self):
        eliminar = []
        for id in self._empleadas:
            abja = self._empleadas[id]
            if abja.get_limite() > self._limite:
                del self._fuente_abeja[abja.get_fuente()]
                del self._fuentes[abja.get_fuente()]
                eliminar.append(id)
                abja.set_fuente(None)
                abja.set_tipo(Tipo_Abeja.EXP)
                self._exploradoras[abja.get_id()] = abja
            else:
                nectar = abja.explota_fuente()
                self._suma_fuentes = self._suma_fuentes + nectar
                self._fuentes[abja.get_fuente()] = nectar
                abja.incrementa_iteracion()
        for id in eliminar:
            del self._empleadas[id]

    # Mandamos a las observadoras a ver el waggle-dance y buscar vecindades.
    def _itera_observadoras(self):
        # waggle es la función que le asigna a las abejas su fuente
        self._waggle_dances()

        # checamos cuales fuentes no fueron asignadas.
        # Asignadas tiene de llave la fuente y de valor el id de la abeja.
        eliminar = list(self._fuentes)

        for id in self._observadoras:
            abja = self._observadoras[id]
            if abja.get_fuente() in eliminar:
```

```
                eliminar.remove(abja.get_fuente())

        # eliminamos las fuentes no asignadas.
        for fuente in eliminar:
            id = self._fuente_abeja[fuente]
            abja = self._abejas[id]
            del self._fuente_abeja[fuente]
            del self._fuentes[fuente]
            del self._empleadas[abja.get_id()]
            abja.set_fuente(None)
            abja.set_tipo(Tipo_Abeja.EXP)
            self._exploradoras[abja.get_id()] = abja

        # observamos las fuentes asignadas para ver si
        # la exploración local mejora el resultado.
        for id in self._observadoras:
            abja = self._observadoras[id]
            if abja.get_fuente() is None:
                continue
            fuente_delta = abja.observa_solucion()
            nectar_delta = self._nectar_fun(fuente_delta)
            nectar_original = self._fuentes[abja.get_fuente()]
            if nectar_delta > nectar_original:
                self._actualiza_fuentes(fuente_delta, abja.get_fuente())

        # Reiniciamos la fuente de las observadoras para la siguiente itercación
        for id in self._observadoras:
            abja = self._observadoras[id]
            abja.set_fuente(None)

    # Una función que regrese True o False dependiendo un factor.
    # Esa función es la probabilidad de escoger una fuente.
    def _rueda_ruleta(self, nectar, factor):
        if self._suma_fuentes == 0 or factor < 0.00000004:
            return bool(get_randbits(1))
        return (get_random() * factor)  <= (nectar/self._suma_fuentes)

    # Esta función genera parejas de observadoras con fuentes.
    def _waggle_dances(self):
        llaves_fuentes = list(self._fuentes)
        if len(llaves_fuentes) <= 0:
            return
        for id in self._observadoras:
            abja = self._observadoras[id]
            it = 0
            factor = 1
            while abja.get_fuente() == None:
                fuente = llaves_fuentes[it]
                nectar = self._fuentes[fuente]
```

```
                if self._rueda_ruleta(nectar, factor):
                    abja.set_fuente(fuente)
                elif (it + 1) == len(llaves_fuentes):
                    it = 0
                    factor = factor / 2
                else:
                    it = it + 1

    # Actualiza las fuentes observadas que sean mejores que las originales.
    def _actualiza_fuentes(self, fuente_delta, fuente_original):
        if self._termino_iteracion != None:
            self._termino_iteracion(fuente_delta)
        id = self._fuente_abeja[fuente_original]
        abja = self._abejas[id]

        del self._fuentes[fuente_original]
        del self._fuente_abeja[fuente_original]

        self._fuentes[fuente_delta] = self._nectar_fun(fuente_delta)
        self._fuente_abeja[fuente_delta] = abja.get_id()

        abja.set_fuente(fuente_delta)

        # Pero si la asignamos a todas después de limpiarla entonces
        # ya no se puede observar, o si?
        for i in self._observadoras:
            abja_obs = self._observadoras[i]
            if not (abja_obs.get_fuente() in self._fuentes):
                abja_obs.set_fuente(None)

    # Actualiza las fuentes si se requiere de alguna operación posterior.
    def _llamada_post_iteracion(self):
        if self._termino_iteracion != None:
            if self._fuente_ini != None:
                self._termino_iteracion(self._fuente_ini)
            for i in self._fuentes:
                self._termino_iteracion(i)
```

C.3. abejas_tetris

C.3.1. abejas_tetris.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"

import abejas_tetris as abeja_tetris
import abejas_tetris.tetris as tetris
```

```python
from abejas_tetris.tetris.tipo_pieza import *
from abejas_tetris.tetris.movimiento import *
from abejas_tetris.tetris.casilla import *
from abejas_tetris.tetris.punto import *
from abejas_tetris.constantes import *
from abejas_tetris.abc.colmena import *
from abejas_tetris.funciones_online import *
import time
from abejas_tetris.my_random import get_random, get_randrange, get_randbits
import pygame
from pygame.locals import *
import math

lista_piezas = []

def get_piezas():
    global lista_piezas
    return lista_piezas

class Abejas_Tetris():
    """ La interfaz de comunicación entre la heurística y el juego. """
    def __init__(self, online=True, size_colmena=150, \
        limite_it=50, delta=0.1, alto=20, ancho=10):
        """
        Parameters
        ----------
        online : bool
            Es una bandera de modo del juego.
        size_colmena : int
            Es el tamaño que tendrá la colmena de abejas.
        limite_it : int
            Es el límite de iteraciones que una abeja hace sobre una fuente.
        delta : float
            Es qué tan lejos llegará la abeja observadora.
        alto : int
            Es el alto del tablero.
        ancho : int
            Es el ancho del tablero.
        """
        self._x = ancho
        self._y = alto
        t = tetris.Tetris(self._x, self._y)
        self._tetris = t
        self.pierde = False
        self.online = online
        self.colmena = Colmena(size_colmena, self._tetris, limite_it, delta)
        self.lista_piezas = []

    def set_lista_piezas(self, piezas):
```

```
        global lista_piezas
        lista_piezas = []
        self.lista_piezas = []
        """
        Asigna una lista de piezas a la lista global de piezas.
        Parameters
        ----------
        piezas : list(str)
            Es la lista de piezas en su representación de string.
        """
        for i in piezas:
            if i == 'I':
                self.lista_piezas.append(Tipo.I)
                lista_piezas.append(Tipo.I)
            elif i == 'LG':
                self.lista_piezas.append(Tipo.LG)
                lista_piezas.append(Tipo.LG)
            elif i == 'LS':
                self.lista_piezas.append(Tipo.LS)
                lista_piezas.append(Tipo.LS)
            elif i == 'RG':
                self.lista_piezas.append(Tipo.RG)
                lista_piezas.append(Tipo.RG)
            elif i == 'RS':
                self.lista_piezas.append(Tipo.RS)
                lista_piezas.append(Tipo.RS)
            elif i == 'T':
                self.lista_piezas.append(Tipo.T)
                lista_piezas.append(Tipo.T)
            else:
                self.lista_piezas.append(Tipo.Sq)
                lista_piezas.append(Tipo.Sq)

    def init_colmena(self):
        """
        Inicializa los valores de las funciones de la colmena.
        """
        if self.online:
            self.colmena.set_funcion_nectar(funcion_nectar_online)
            self.colmena.set_funcion_buscar_fuente(funcion_buscar_fuente_online)
            self.colmena.set_funcion_observacion(funcion_observacion_online)
            self.colmena.set_funcion_termino_iteracion(funcion_limpiadora)
            self.colmena.set_funcion_explotar_fuente(explota_tablero)
            self.colmena.set_funcion_comparativa(num_tetris)
            self.colmena.inizializa_abejas()

    def juega_online(self, iteraciones=None, limpieza=False):
        """
        Juega Tetris con abejas de manera online.
```

```

        Parameters
        ----------
        iteraciones=None : int
            Es el número de iteraciones que correrá la partida.
        """
        if not limpieza:
            self._tetris.desactiva_limpieza_automatica()
            self.colmena.actualiza_fuente_inicial(self._tetris)
        else:
            self._tetris.activa_limpieza_automatica()
        self.init_colmena()
        if iteraciones == None:
            cond = self._tetris.game_over()
            while not cond:
                self.colmena.itera_colmena()
                cond = self.colmena.get_solucion_final().game_over()
        else:
            for i in range(iteraciones):
                self.colmena.itera_colmena()
                if self._tetris.game_over():
                    return self.colmena.get_solucion_final()
        return self.colmena.get_solucion_final()

    def interactivo(self):
        """
        Hace una pequeña interfaz para ver el funcionamiento paso a paso del
        juego.
        """
        tetris_tmp = self._tetris
        self._tetris = tetris.Tetris(self._x, self._y)
        self.set_gui()
        pieza = 0
        while not self._tetris.game_over():
            if self._tetris.requiere_pieza():
                tipo = self.lista_piezas[pieza]
                pieza = pieza + 1
                self._tetris.set_pieza(tipo=tipo)
            moves = [Movimiento.CAE, Movimiento.DER, \
                Movimiento.IZQ, Movimiento.GIR]
            moves_posibles = []
            for i in moves:
                if self._tetris.movimiento_valido(i):
                    moves_posibles.append(i)
            if self._tetris.puede_fijar():
                moves_posibles.append(Movimiento.FIJ)
            print("¿Cuál es el siguiente movimiento?")
            print("Movimientos válidos:")
            j = 0
```

```
            for i in moves_posibles:
                print(str(j) + ":" + str(i))
                j = j + 1
            print("Ingrese número de movimiento:")
            mov = int(input())
            move = moves_posibles[mov]
            if move != Movimiento.FIJ:
                self._tetris.mueve(move=move)
            else:
                self._tetris.fija()
                print('###### NECTAR ######')
                print(funcion_nectar_online(self._tetris))
                print('###### FILAS ELIMINADAS #####')
                print(self._tetris.num_tetris())
            self.dibuja()
            time.sleep(0.05)
        self.quit_gui()
        self._tetris = tetris_tmp

    def pinta_historia(self, historia=[]):
        """
        Pinta una lista de movimientos.

        Parameters
        ----------
        historial : list(Movimiento)
            Es el historial de movimientos hechos que pintar.
        """
        print("Pintando el historial:")
        tetris_tmp = self._tetris
        self._tetris = tetris.Tetris(self._x, self._y)
        self.set_gui()
        pieza = 1
        tipo = self.lista_piezas[0]
        self._tetris.set_pieza(tipo=tipo)
        moves = historia
        while len(moves) > 0:
            move = moves.pop(0)
            if move == Movimiento.FIJ:
                self._tetris.fija()
                tipo = self.lista_piezas[pieza]
                self._tetris.set_pieza(tipo=tipo)
                pieza = pieza + 1
            else:
                self._tetris.mueve(move=move)
            self.dibuja()
            time.sleep(0.05)
        print('###### NECTAR ######')
        print(funcion_nectar_online(self._tetris))
```

```python
        print('###### FILAS ELIMINADAS #####')
        print(self._tetris.num_tetris())
        time.sleep(5)
        self.quit_gui()
        self._tetris = tetris_tmp

    def pinta_solucion(self, solucion):
        """
        Dada una solución de un juego de Tetris, crea una GUI y la muestra.

        Parameters
        ----------
        solucion : Tetris
            Es el juego de Tetris a dibujar.
        """
        tetris_tmp = self._tetris
        historial = solucion.get_historial()
        moves = []
        for i in historial:
            moves.append(i)
        self._tetris = tetris.Tetris(self._x, self._y)
        self.set_gui()
        tipo = self.lista_piezas[self._tetris.piezas_jugadas()]
        self._tetris.set_pieza(tipo=tipo)
        while len(moves) > 0:
            move = moves.pop(0)
            if move == Movimiento.FIJ:
                self._tetris.fija()
                tipo = self.lista_piezas[self._tetris.piezas_jugadas()]
                self._tetris.set_pieza(tipo=tipo)
            else:
                self._tetris.mueve(move=move)
            self.dibuja()
            time.sleep(0.05)
        print('###### NECTAR ######')
        print(funcion_nectar_online(self._tetris))
        print('###### FILAS ELIMINADAS #####')
        print(self._tetris.num_tetris())
        time.sleep(5)
        self.quit_gui()
        self._tetris = tetris_tmp


    def random(self):
        """
        Juega de forma completamente aleatoria.
        """
        while not self.pierde:
            tipo = self.__random_tipo()
```

```
            move = self.__random_move()
            self.pierde = not self._tetris.siguiente_random(tipo=tipo, \
                move=move)
            self.dibuja()
            time.sleep(0.05)

    def quit_gui(self):
        """
        Sale de la interfaz gráfica.
        """
        pygame.font.quit()
        pygame.display.quit()
        pygame.mixer.quit()

    def set_gui(self):
        """
        Inizializa los valores que pygame debe tener al principio.
        """
        self._gui = True
        # Tablero:
        self.resx = self._x*ANCHO_BLOQUE+2*TABLERO_ALTURA+TABLERO_MARGEN
        self.resy = self._y*ALTO_BLOQUE+2*TABLERO_ALTURA+TABLERO_MARGEN
        # Lineas:
        self.frontera_arriba = pygame.Rect(0,0,self.resx,TABLERO_ALTURA)
        self.frontera_abajo = \
            pygame.Rect(0,self.resy-TABLERO_ALTURA,self.resx,TABLERO_ALTURA)
        self.frontera_izq = pygame.Rect(0,0,TABLERO_ALTURA,self.resy)
        self.frontera_der = \
            pygame.Rect(self.resx-TABLERO_ALTURA,0,TABLERO_ALTURA,self.resy)
        self.inicio_x = math.ceil(self.resx/2.0)
        self.inicio_y = TABLERO_MARGEN_SUPERIOR+TABLERO_ALTURA + TABLERO_MARGEN
        # False means no rotate and True allows the rotation.
        self.colores = {
            Tipo.I : ROJO,      # I
            Tipo.RS : VERDE,  # S
            Tipo.LG : AZUL,     # J
            Tipo.Sq : NARANJA, # O
            Tipo.LS : DORADO,    # Z block
            Tipo.T : MORADO,  # T block
            Tipo.RG : AZUL_CIAN    # J block
        }
        pygame.init()
        pygame.font.init()
        pygame.mixer.init()
        pygame.mixer.music.load("./etc/tetris-theme.mp3")
        pygame.mixer.music.play()
        time.sleep(1)
        self.pantalla = pygame.display.set_mode((self.resx,self.resy))
        pygame.display.set_caption("Tetris")
```

```python

    def dibuja(self):
        """
        Dibuja todo lo que haya en el tablero del juego.
        """
        self.pantalla.fill(NEGRO)
        self.__dibuja_tablero()
        self.__dibuja_fichas()
        pygame.display.flip()

    # Funciones auxiliares

    # Dibuja las fichas.
    def __dibuja_fichas(self):
        for x in range(self._x):
            for y in range(self._y):
                if self._tetris.get_casilla(x,y) != None:
                    tipo = self._tetris.get_casilla(x, y).get_tipo()
                    bloque = self._get_block(x, y)
                    pygame.draw.rect(self.pantalla, \
                        self.colores.get(tipo), bloque)
                    pygame.draw.rect(self.pantalla, \
                        NEGRO, bloque, MARGEN_BLOQUE)

    # Dibuja todos los bloques de las fichas.
    def _get_block(self, x, y):
        bx = (x + 0.3)*ALTO_BLOQUE + TABLERO_MARGEN
        by = (y + 0.4)*ANCHO_BLOQUE + 0
        return pygame.Rect(bx,by,ANCHO_BLOQUE,ALTO_BLOQUE)

    # Dibuja el tablero.
    def __dibuja_tablero(self):
        pygame.draw.rect(self.pantalla, BLANCO, self.frontera_arriba)
        pygame.draw.rect(self.pantalla, BLANCO, self.frontera_abajo)
        pygame.draw.rect(self.pantalla, BLANCO, self.frontera_izq)
        pygame.draw.rect(self.pantalla, BLANCO, self.frontera_der)

    # Función que regresa los puntos de la casillas.
    def __get_puntos(self, casillas):
        puntos = []
        for i in casillas:
            puntos.append(i.get_punto())
        return puntos

    # Regresa un Tipo de manera aleatoria.
    def __random_tipo(self):
        piezas = [Tipo.I, Tipo.LG, Tipo.LS, Tipo.T, Tipo.RS, Tipo.RG, Tipo.Sq]
        return piezas[get_randrange(len(piezas))]

```

```python
    # Regresa un Movimiento (menos FIJ) de forma aleatoria.
    def __random_move(self):
        moves = [Movimiento.CAE, Movimiento.DER, Movimiento.IZQ, Movimiento.GIR]
        moves_posibles = []
        for i in moves:
            if self._tetris.movimiento_valido(i):
                moves_posibles.append(i)
        if len(moves_posibles) > 0:
            return moves_posibles[get_randrange(len(moves_posibles))]
        return None
```

C.3.2. funciones_online.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "José Ricardo Rodríguez Abreu"
__email__ = "ricardo_rodab@ciencias.unam.mx"
import abejas_tetris
import math
from abejas_tetris.tetris.movimiento import *
from abejas_tetris.my_random import get_random, get_randrange, get_randbits
import logging
PESO_HORIZONTALIDAD = 1
PESO_ATRAPADOS = 1
PESO_CUBIERTOS = 1
PESO_FILA_REMOVIDA = 1
PESO_ALTURA = 1
PESO_VULNERABLE = 1
FUNCION = None

def set_pesos(data):
    global PESO_HORIZONTALIDAD
    global PESO_ATRAPADOS
    global PESO_CUBIERTOS
    global PESO_FILA_REMOVIDA
    global PESO_ALTURA
    global PESO_VULNERABLE
    global FUNCION
    PESO_HORIZONTALIDAD = data['peso_horizontalidad']
    PESO_ATRAPADOS = data['peso_atrapados']
    PESO_CUBIERTOS = data['peso_cubiertos']
    PESO_FILA_REMOVIDA = data['peso_fila_removida']
    PESO_ALTURA = data['peso_altura']
    PESO_VULNERABLE = data['peso_vulnerable']
    logging.getLogger().setLevel('INFO')
    FUNCION = data['funcion']
    if FUNCION == 'pesos':
        logging.info('Se usará Pesos entre pesos negativo.')
    elif FUNCION == 'skyline':
        logging.info('Se usará raining skyline.')
```

```
    else:
        logging.info('Se usará una combinación de ambas funciones.')

def funcion_nectar_online(fuente):
    if fuente.game_over() and fuente.num_tetris() > 100:
        return math.inf
    elif FUNCION == 'pesos':
        return filas_pesos_negativos(fuente)
    elif FUNCION == 'skyline':
        return skyline(fuente)
    else:
        return hibrido(fuente)

def hibrido(fuente):
    skyline_val = skyline(fuente)

    cubiertos_val = (cuenta_cubiertos(fuente) + 1)
    cubiertos_val = PESO_CUBIERTOS * cubiertos_val

    num_tetris_val = num_tetris(fuente)

    return (skyline_val * (num_tetris_val + 1)) / (cubiertos_val + 1)

def skyline(fuente):
    # Ponderado:
    total = get_total_convexo_ponderado(fuente.get_ancho(), fuente.get_altura())
    disponibles = cuenta_descubiertos(fuente)
    return (disponibles / total) * (1 + num_tetris(fuente))

    # No ponderado:
    #total = get_total_convexo_no_ponderado(fuente.get_ancho(), fuente.get_altura())
    #disponibles = cuenta_descubiertos_no_ponderado(fuente)
    #return (disponibles / total)


# Ganancia de cada fuente
def filas_pesos_negativos(fuente):
    global PESO_HORIZONTALIDAD
    global PESO_ATRAPADOS
    global PESO_CUBIERTOS
    global PESO_FILA_REMOVIDA
    global PESO_ALTURA
    global PESO_VULNERABLE
    global FUNCION
    #Entre menor, mejor.
    atrapados = cuenta_atrapados(fuente)
    # Entre menor, mejor.
    horizontal = horizontalidad(fuente)
    # Entre mayor, mejor.
```

```
    num_tetris_var = num_tetris(fuente)
    # Entre menor, mejor.
    cubiertos = cuenta_cubiertos(fuente)
    # La altura siempre debe ser baja
    altura = fuente.altura_maxima()
    # La funcion, entre mayor, mejor.

    horizontal = PESO_HORIZONTALIDAD * horizontal
    atrapados = PESO_ATRAPADOS * atrapados
    cubiertos = PESO_CUBIERTOS * cubiertos
    num_tetris_var = PESO_FILA_REMOVIDA * num_tetris_var
    altura =  PESO_ALTURA * altura

    if (horizontal + atrapados + cubiertos + altura) == 0:
        return 0

    return (1 + num_tetris_var) / (horizontal + atrapados + cubiertos + altura)

# Explotar las abejas empleadas el tablero
def explota_tablero(fuente):
    funcion_buscar_fuente_online(fuente)
    return funcion_nectar_online(fuente)

# Como buscar una fuente
def funcion_buscar_fuente_online(fuente, coloca_pieza=True):
    lista_piezas = abejas_tetris.get_piezas()
    if fuente.game_over() or fuente.piezas_jugadas() == len(lista_piezas):
        fuente.set_game_over(True)
        return
    if fuente.requiere_pieza() and coloca_pieza:
        pieza = lista_piezas[fuente.piezas_jugadas()]
        fuente.set_pieza(tipo=pieza)
    condicion_termino = fuente.requiere_pieza()
    while not condicion_termino:
        fuente.mueve_o_fija()
        condicion_termino = fuente.requiere_pieza()

# Como observar y buscar fuentes vecindades
def funcion_observacion_online(fuente, delta):
    fuente_clon = fuente.clona()
    if fuente.game_over():
        return fuente_clon
    fuente_clon.elimina_historial(delta)
    funcion_buscar_fuente_online(fuente_clon, True)
    return fuente_clon

# Función de ejecución posterior a cada fuente.
def funcion_limpiadora(fuente):
    fuente.limpia()
```

```python

def num_tetris(fuente):
    return fuente.puede_limpiar() + fuente.num_tetris()

"""
    FUNCIONES AUXILIARES:
"""

def horizontalidad(fuente):
    altura_max = fuente.altura_maxima()
    altura_min = fuente.altura_minima()
    return abs(altura_max - altura_min)

def cuenta_atrapados(fuente):
    return fuente.cuenta_atrapados()

def cuenta_cubiertos(fuente):
    return fuente.cuenta_cubiertos()

# Funciones auxiliares del raining skyline no ponderado

def cuenta_descubiertos_no_ponderado(fuente):
    total = 0
    x = fuente.get_ancho()
    y = fuente.get_altura()
    for i in range(x):
        total = total + get_altura_nop(i, y, fuente)
    return total

def get_altura_nop(x, y, fuente):
    for i in range(y):
        if fuente.get_casilla(x, i) != None:
            return get_altura_no_ponderada(x, i - 1, fuente)
    return get_altura_no_ponderada(x, y - 1, fuente)

def get_altura_no_ponderada(x, y, fuente):
    if y < 0:
        return 0
    if y == 0:
        return 1
    return 1 + get_altura_no_ponderada(x, y - 1, fuente)

def get_total_convexo_no_ponderado(x, y):
    return x * y

# Funciones auxiliares del raining skyline ponderado

def cuenta_descubiertos(fuente):
    total = 0
```

```
    x = fuente.get_ancho()
    y = fuente.get_altura()
    for i in range(x):
        total = total + get_altura(i, y, fuente)
    return total

def get_altura(x, y, fuente):
    for i in range(y):
        if fuente.get_casilla(x, i) != None:
            return get_altura_ponderada(i - 1, fuente)
    return get_altura_ponderada(y - 1, fuente)

def get_altura_ponderada(y, fuente):
    if y < 0:
        return 0
    if y == 0:
        return fuente.get_altura()
    return (fuente.get_altura() - y) + get_altura_ponderada(y - 1, fuente)

def get_total_convexo_ponderado(x, y):
    if y == 0:
        return x
    return ((y + 1) * x) + get_total_convexo_ponderado(x, y - 1)
```

Bibliografía

[1] Arora, Sanjeev y Boaz Barak: *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[2] Ausiello, Giorgio, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi y V. Kann: *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, Berlin, Heidelberg, 1st edición, 1999, ISBN 3540654313.

[3] Ball, W.W.R. y H.S.M. Coxeter: *Mathematical Recreations and Essays*. Dover Recreational Math Series. Dover Publications, 1987, ISBN 9780486253572. `https://books.google.com.mx/books?id=Ze5LDwAAQBAJ`.

[4] Basalo, P.M.R., I.R. Laguna y F.R. Diez: *Algoritmos heurísticos y aplicaciones a métodos formales*. Universidad Complutense de Madrid, 2000, ISBN 9781449261429. `https://books.google.com.mx/books?id=ps7qnQAACAAJ`.

[5] Boumaza, Amine: *How to design good Tetris players*. `https://hal.inria.fr/hal-00926213`, working paper or preprint, 2013.

[6] Boyd, Stephen y Lieven Vandenberghe: *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004, ISBN 0521833787.

[7] Brzustowski, John: *Can you win at TETRIS?* Tesis de Doctorado, University of Waterloo, 1992. `https://open.library.ubc.ca/collections/ubctheses/831/items/1.0079748`.

[8] Burgiel, Heidi: *How to Lose at Tetris*. Mathematical Gazette, 81:194–200, 1997.

[9] Buss, Samuel R., Alexander S. Kechris, Anand Pillay y Richard A. Shore: *The Prospects for Mathematical Logic in the Twenty-First Century*. The Bulletin of Symbolic Logic, 7(2):169–196, 2001, ISSN 10798986. `http://www.jstor.org/stable/2687773`.

[10] Campbell, Paul J.: *Gauss and the eight queens problem: A study in miniature of the propagation of historical error*. Historia Mathematica, 1977, ISSN 1090249X.

[11] Church, A.: *An Unsolvable Problem of Elementary Number Theory*. Princeton University Press, 1935. `https://books.google.com.mx/books?id=ps9znQAACAAJ`.

[12] Cook, Stephen A.: *The Complexity of Theorem-proving Procedures*. En *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, páginas 151–158, New York, NY, USA, 1971. ACM. `http://doi.acm.org/10.1145/800157.805047`.

[13] Copeland, B. Jack: *The Church-Turing Thesis*. En Zalta, Edward N. (editor): *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2019 edición, 2019.

[14] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest y Clifford Stein: *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edición, 2009, ISBN 0262033844, 9780262033848.

[15] Demaine, Erik D., Susan Hohenberger y David Liben-Nowell: *Tetris is Hard, Even to Approximate*. CoRR, cs.CC/0210020, 2002. `http://arxiv.org/abs/cs.CC/0210020`.

[16] Demange, M. y T. Ekim: *Minimum Maximal Matching Is NP-Hard in Regular Bipartite Graphs*. En Agrawal, Manindra, Dingzhu Du, Zhenhua Duan y Angsheng Li (editores): *Theory and Applications of Models of Computation*, páginas 364–374, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg, ISBN 978-3-540-79228-4.

[17] Edmonds, J.: *Paths, trees, and flowers*. Canadian Journal, 1965. `https://www.cambridge.org/core/journals/canadian-journal-of-mathematics/article/paths-trees-and-flowers/08B492B72322C4130AE800C0610E0E21`.

[18] Fortnow, Lance y Steven Homer: *A Short History of Computational Complexity*. Bulletin of the EATCS, 80:95–133, 2002.

[19] Galaviz Casas, José y Arturo Magidin: *Introducción a la Criptología*. Vínculos Matemáticos, N°15.2000:151–155, 2003. `ftp://ftp.iingen.unam.mx/Documentacion/Libros/Seguridad/FCiencias/cripto.pdf`.

[20] Gaona, A.L.: *Introduccion Al Desarrollo de Programas Con Java*. Prensas de ciencias. UNAM, 2007, ISBN 9789703243174. `https://books.google.com.mx/books?id=29zE8HTdJ1QC`.

[21] Garey, Michael R. y David S. Johnson: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990, ISBN 0716710455.

[22] Gent, Ian P., Christopher Jefferson y Peter Nightingale: *Complexity of n-Queens Completion*. Journal of Artificial Intelligence Research, 59:815–848, aug 2017, ISSN 1076-9757. `https://jair.org/index.php/jair/article/view/11079`.

[23] Gigerenzer, Gerd: *Why Heuristics Work*. Informe técnico 1, Max Planck Institute for Human Development, 2008. `https://doi.org/10.1111/j.1745-6916.2008.00058.x`, PMID: 26158666.

[24] Gil, Amparo, Javier Segura y Nico Temme: *Numerical Methods for Special Functions*. SIAM, Enero 2007, ISBN 978-0-89871-634-4.

[25] *Getting Started - About Version Control*. `https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control`. Acceso: 2019-11-17.

[26] *Getting Started - A Short History of Git*. `https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git`. Acceso: 2019-11-17.

[27] *Initial revision of "git", the information manager from hell*. `https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290`. Acceso: 2019-11-17.

[28] Grattan-Guinness, Ivor: *A Sideways Look at Hilbert's Twenty-three Problems of 1900*. Informe técnico 7, AMS, 2000.

[29] Gurovich, E.V. y Universidad Nacional Autónoma de México. Facultad de Ciencias: *Introduccion a la Teoria de la Computacion*. Prensas de ciencias. UNAM, Facultad de Ciencias, 2015, ISBN 9786070268304. `https://books.google.com.mx/books?id=NXQE8NJw9d4C`.

[30] Harikumar, Surajkumar y Manikandan Srinivasan: *Google's PageRank Algorithm : An Analysis, Implementation and Relevance today*. Informe técnico, Google, 1999.

[31] Hartmanis, J y R E Stearns: *On the Computational Complexity of Algorithms*. Informe técnico, American Mathematical Society, 1965.

[32] *Definition of heuristic in English*. `https://www.lexico.com/en/definition/heuristic`. Acceso: 2019-11-17.

[33] Hilbert, D y W Ackermann: *Principles of mathematical logic*. Chelsea scientific books. Chelsea Pub. Co., 1950. `https://books.google.com.mx/books?id=jgFKAAAAMAAJ`.

[34] Hoad, Phil: *Tetris: how we made the addictive computer game*. The Guardian, web, 2014.

[35] *Guíones del intérprete de comandos (shell scripts o archivos por lotes)*. `https://elpuig.xeill.net/Members/vcarceler/c1/didactica/apuntes/ud3/na6`. Acceso: 2019-11-17.

[36] Johnson, Bobbie: *How Tetris conquered the world, block by block*. The Guardian, web, 2009.

[37] Johnson, Maxime: *Tetris atteint les 100 millions de téléchargements payants (et une petite histoire du jeu)*. maximejohnson.com, web, 2010.

[38] Karaboga, Dervis: *An idea based on honey bee swarm for numerical optimization*. Informe técnico, Technical report-tr06, Erciyes university, Engineering Faculty, 2005.

[39] Karaboga, Dervis y Bahriye Basturk: *On the performance of artificial bee colony (ABC) algorithm*. Applied soft computing, 8(1):687–697, 2008.

[40] Karp, Richard M: *Reducibility among Combinatorial Problems*, páginas 85–103. Springer US, Boston, MA, 1972, ISBN 978-1-4684-2001-2. `https://doi.org/10.1007/978-1-4684-2001-2{_}9`.

[41] Kauten, Christian: *Tetris (NES) for OpenAI Gym*. `https://github.com/Kautenja/gym-tetris`, 2019.

[42] Knuth, D. E.: *The art of computer programming. Vol.1: Fundamental algorithms*. 1978.

[43] Kutzkov, Konstantin y Dominik Scheder: *Using CSP To Improve Deterministic 3-SAT*. Informe técnico, IT University of Copenhagen, Denmark, 2010.

[44] Lewis, John, William Loftus y MyCodeMate: *Java Software Solutions: Foundations of Program Design*. Addison-Wesley Publishing Company, USA, 2007, ISBN 1405887990. `https://dl.acm.org/citation.cfm?id=1554921`.

[45] *40 lines*. `https://tetris.fandom.com/wiki/40_lines`. Acceso: 2019-11-17.

[46] Marques-Silva, João: *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*. Informe técnico, Portuguese Conference on Artificial Intelligence, 1999.

[47] *NumPy*. `https://numpy.org/`. Acceso: 2019-11-17.

[48] Pearl, J.: *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley, 1984, ISBN 9780201055948. `https://books.google.com.mx/books?id=0XtQAAAAMAAJ`.

[49] *Pygame*. `https://www.pygame.org`. Acceso: 2019-11-17.

[50] *python.* https://www.python.org/. Acceso: 2019-11-17.

[51] *The Making of Python, A Conversation with Guido van Rossum, Part I.* https://www.artima.com/intv/pythonP.html. Acceso: 2019-11-17.

[52] *OrganizationsUsingPython.* https://wiki.python.org/moin/OrganizationsUsingPython. Acceso: 2019-11-17.

[53] Rogers, Jr., Hartley: *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, MA, USA, 1987, ISBN 0-262-68052-1.

[54] *The international SAT Competitions web page.* http://www.satcompetition.org/. Acceso: 2019-11-17.

[55] Schrijver, Alexander: *Polyhedral Combinatorics and Combinatorial Optimization.* http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.5357, 2004.

[56] Scopatz, A. y K.D. Huff: *Effective Computation in Physics: Field Guide to Research with Python.* O'Reilly Media, 2015, ISBN 9781491901588. https://books.google.de/books?id=6IkNCgAAQBAJ.

[57] Smith, Brent y Greg Linden: *Two Decades of Recommender Systems at Amazon.com.* Informe técnico, IEEE Computer Society, 2017, ISBN 10897801/17.

[58] Smith, Greg Linden Brent y Jeremy York: *Amazon.com Recommendations.* Informe técnico, IEEE Computer Society, 2003. http://computer.org/internet/.

[59] *TIOBE Index.* https://www.tiobe.com/tiobe-index/. Acceso: 2019-11-17.

[60] Turing, A: *On Computable Numbers with an Application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society, 1936, ISSN 1460244X.

[61] *ETA Phone Home: How Uber Engineers an Efficient Route.* https://eng.uber.com/engineering-an-efficient-route/. Acceso: 2019-11-17.

[62] Vazirani, Vijay V.: *Approximation Algorithms.* Springer-Verlag, Berlin, Heidelberg, 2001, ISBN 3-540-65367-8.

[63] *Creation of virtual environments.* https://docs.python.org/3/library/venv.html. Acceso: 2019-11-17.

[64] Wikipedia, the free encyclopedia: *Tetrominoes IJLO STZ Worlds.svg*, 2006. https://en.wikipedia.org/wiki/File:Tetrominoes_IJLO_STZ_Worlds.svg, Acceso: 2019-11-17.

[65] Wikipedia, the free encyclopedia: *All 18 Pentominoes.svg*, 2008. https://en.wikipedia.org/wiki/File:All_18_Pentominoes.svg, Acceso: 2019-11-17.

[66] Wikipedia, the free encyclopedia: *Atomic force microscopy*, 2013. https://upload.wikimedia.org/wikipedia/commons/7/7a/Alexey_Pajitnov%2C_Games_Designer_%26_Creator_of_Tetris.jpg, Acceso: 2019-11-17.

[67] *Nes-py Emulation System.* https://pypi.org/project/nes-py/. Acceso: 2019-11-17.