DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Técnicas de Diseño de Algoritmos

Viernes 11 de Abril de 2014

Algoritmos y Estructuras de Datos III
Entrega de TP

Grupo $\texttt{0111}_2$

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Benegas, Gonzalo	958/12	gsbenegas@gmail.com
Duarte, Miguel	904/11	miguelfeliped@gmail.com
Niikado, Marina	711/07	mariniiik@yahoo.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

1.1. Objetivos

Mediante la realización de este trabajo práctico se pretende realizar una introducción a la implementación y el análisis de las técnicas algorítmicas básicas para resolución de problemas.

Se analizan en particular las técnicas de *algoritmos golosos*, y de *backtracking*.

1.2. Pautas de trabajo

Se brindan tres problemas, escritos en términos coloquiales, en donde para cada uno de ellos se requiere **encontrar un algoritmo** que brinde una **solución particular**, acotado por una determinada **complejidad temporal**. El algoritmo debe, además, ser **implementado** en un lenguaje de programación a elección. Los datos son proporcionados, y deben ser devueltos bajo formatos específicos de *input* y de *output*.

Posteriormente se deben realizar análisis teóricos y empíricos tanto de la de **correctitud** como de la **complejidad temporal** para cada una de las soluciones propuestas.

1.3. Metodología utilizada

Para cada ejercicio, se brinda primeramente una **descripción** del problema planteado, a partir de la cual se realiza una **abstracción** hacia un **modelo formal**, que permite tener un **entendimiento preciso** de las pautas requeridas.

Se expone, cuando las hay, una **enumeración de las características** elementales del problema; estas son aquellas que permiten **encuadrarlo** dentro de una **familia de problemas** típicos.

Se desarrolla posteriormente un análisis del conjunto **universo de posibles soluciones** (o factibles), caracterizando matemáticamente el concepto de **solución correcta** y, en los casos en que se solicita **optimización**[1], se definen las condiciones que dan forma ya sea a todo el subconjunto de **soluciones óptimas** que se encuadran dentro de las pretenciones del problema, o a una **solución particular** dentro del mismo (la cual denominamos *mejor solución*).

Luego de caracterizar para todo conjunto posible de entradas «*cómo se compone el conjunto solución*» correspondiente, se desarrolla un **pseudocódigo** en el que se expone «*cómo llegar a ese conjunto*»[2].

Habiendo planteado la **hipótesis de resolución** se demuestra, de manera informal o mediante inferencias matemáticas según sea necesario, que el **algoritmo propuesto** realmente permite obtener la **solución correcta**[3].

Después de demostrar la **correctitud de la solución**, y su **optimalidad** en caso de existir varias soluciones correctas, se realiza un análisis teórico de la **complejidad temporal** en donde se estima el comportamiento del algoritmo en términos de tiempo. Este análisis en particular se realiza con el objetivo de obtener una *cota superior asintótica*[4].

[1] Es decir, que la solución pertenezca al *subconjunto de soluciones que* ***maximicen*** *o* ***minimicen*** *una determinada función*

[2] Una explicación coloquial, obviando detalles puramente implementativos: arquitectura, lenguaje, etc.

[3] En caso de existir más de una solución correcta, se demuestra que el algoritmo obtiene al menos una de ellas o, dicho de otro modo, para problemas de optimización, se demuestra que ninguna del resto de las soluciones correctas es mejor que la solución propuesta por nuestro algoritmo

[4] Aunque se mencionan, sobre todo en el caso del *Algoritmo de Backtracking*, algunas «familias de entrada» particulares bajo las cuales el algoritmo propuesto presenta un comportamiento mucho mejor al peor caso.

Luego de calcular la **cota de complejidad temporal**, se realiza una **verificación** empírica, junto con una **exposición gráfica** de los resultados obtenidos, mediante la combinación de técnicas básicas de medición y análisis de datos.

2. Instrucciones de uso

2.1. Herramientas utilizadas

Para la realización de este trabajo se utilizaron un conjunto de herramientas, las cuales se enumeran a continuación:

- `C++` como lenguaje de programación
    - `gcc` como compilador de C++
- `python` y `bash` para la realización de scripts
    - `python` para generar casos de prueba
    - `bash` para automatizar las mediciones
    - `python/matplotlib` para plotear los gráficos
- LaTeX para la redacción de este documento
- Se testeó Sistemas Operativos
    - `Debian GNU/Linux`
    - `Ubuntu`
    - `FreeBSD`, compilando a través de `gmake`
    - `Windows`, a través de `cygwin`

3. Desarrollo del TP

3.1. Problema 1: Camiones sospechosos

No se reentregará este problema.

3.2. Problema 2: La joya del Río de la Plata

No se reentregará este problema.

3.3. Problema 3: Rompecolores

3.3.1. Descripción

Este problema consiste en ubicar en un tablero la mayor cantidad de piezas posibles siguiendo ciertas reglas.

Consideraciones

- El tablero contiene $n \times m$ casilleros cuadrados, n filas y m columnas.
- Piezas existentes: $1, ..., n \times m$. (Cantidad total de piezas: $n \times m$).
- Una pieza es cuadrada y puede tener de 1 a 4 colores distintos. A cada lado (sup, izq, der, inf) le corresponde un color.
- Las piezas no se pueden rotar.
- Colores posibles: $1, ..., c$ (c entero positivo).
- 2 piezas pueden ubicarse en casilleros adyacentes sólo si sus lados adyacentes tienen el mismo color. Podría ocurrir que no sea posible llenar completamente el tablero con las piezas existentes.
- **El problema se deberá resolver utilizando la técnica de *Backtracking* eligiendo algunas podas para mejorar los tiempos de ejecución del programa.**

Ejemplos

Ejemplo 3.3.1.1. Para un tablero de 3×3 y uno de 2×2, suponiendo un caso donde ninguna ficha puede colocarse adyacente a otra, una de las posibles soluciones sería la siguiente:

En el tablero de 3×3 se podrían colocar las fichas $1, 2, 3, 4$ y 5, y en el de 2×2, las fichas 1 y 2.

1	0	2
0	3	0
4	0	5

1	0
0	2

Ejemplo 3.3.1.2. Se tiene un tablero de 2×2, los colores $1, 2, 3$ y las piezas **1**,**2**,**3**,**4** :

	1	
3	**1**	2
	2	

	3	
2	**2**	2
	1	

	3	
1	**2**	3
	2	

	1	
1	**4**	2
	2	

En este caso, la cantidad máxima de piezas que se pueden colocar en el tablero es 3. Entonces las posibles soluciones serían:

1	**2**
0	**4**

0	**2**
3	**1**

Algoritmos de backtracking (ideas y análisis general)

El objetivo del problema es diseñar un algoritmo utilizando la técnica de **backtracking**. El algoritmo de **backtracking** se puede concebir como una técnica recursiva de recorrido de grafos[5], estableciendo un paralelismo entre «el universo de soluciones», y los nodos de un **grafo arbol n-ario**[6], en donde cada nodo representa una solución posible, y n representa la cantidad máxima de soluciones distintas que pueden desprenderse a partir de realizar un cambio determinado en la solución anterior (*vecindad*). Además puede, según el caso, ser interpretado como un árbol en donde cada nodo representa **una solución o sub-solución no necesariamente "válida"** (o "bien formada"), y en donde sólamente **las hojas del árbol** serían *soluciones "correctamente formadas"* (aunque no necesariamente factibles).

Podemos separar también el tipo de problema en dos casos: los **problemas de decisión**, para los que es necesario que la solución cumpla ciertas condiciones particulares determinados por la denominada **función de selección**, y los **problemas de optimización**, los cuales son derivados de los anteriores, y en donde se desea obtener la **"mejor"** de las soluciones válidas[7] en base a una **función objetivo**, entendiendo a esta última como «*la función que establece cuándo una solución es mejor que otra*».

Pequeño ejemplo sobre las ideas anteriores

Un ejemplo concreto de los párrafos anteriores sería un problema en el que se nos pidiese encontrar una lista de 10 números del 1 al 20, tal que la lista sea estrictamente creciente. Tendríamos entonces un problema de decisión, en el que la función de selección estaría determinada por las condiciones *"que la lista sea estrictamente creciente"*, *"que el número esté entre 1 y 20"*, y *"que el largo de la lista sea 10"*, es decir, en donde tomaríamos como soluciones factibles a todas aquellas en que «**lista[i]** $<$ **lista[i+1] y además** $1 \leq$ **lista[i]** ≤ 20**, y además** $|$**lista**$| = 20$»[8]. Si se agregara, además, la condición "quiero de todas las listas posibles la que maximice la sumatoria de cada uno de sus componentes", me encontraría entonces con un **problema de optimización**.

En este ejemplo, entonces, una **solución factible** sería cualquiera que cumpla con los criterios de selección, mientras que una **solución correctamente formada**, es decir, aquellas que podrían pertenecer a las hojas de un posible arbol / "universo de soluciones" (tal y como se menciona en el párrafo introductorio), sería simplemente "una lista de 10 números"[9].

De esta forma, para el ejemplo anterior, se podría establecer un algoritmo en que cada nodo interno del árbol de soluciones representase la **subestructura** de una o más soluciones, sin llegar a ser una solución, más concretamente, se podría asumir que el nodo raíz del arbol

[5]Quizás la oración no está expresada de la mejor forma; al decir "se puede concebir como..." se quiere dar a entender que no se está dando la *definición estricta* de backtracking, sino más bien una *interpretación* de la misma. Se dio por sobreentendido el hecho de que en su sentido más básico es "una técnica de resolución de problemas"; así, representando al "universo de soluciones" como un grafo, **y en el contexto del párrafo**, creemos que la idea que se intenta exponer es válida.

[6]O incluso un grafo conexo con ciclos, en el caso de un problema muy mal modelado.

[7]Es decir, las que cumplan con el criterio de selección. Tener en cuenta que esto último no implica necesariamente la existencia de funciones inválidas, sino que puede darse el caso en que el criterio de selección admita que todas las soluciones a un problema determinado son válidas, y por consiguiente simplemente se esté buscando obtener la mejor de ellas.

[8]Se cometen abusos varios de notación para no ahondar en detalles innecesarios.

[9]Esto último es debatible si se considera el tamaño como un criterio de selección; en todo caso habría que formalizar aún más y no viene al caso.

es una lista vacía, y que cada hijo es una lista que contiene a los elementos de su padre, y cuyo tamaño es superior al anterior en 1, de forma tal que es simple ver que: *cada nodo comparte una subestructura con sus nodos-hermanos heredada desde su nodo-padre, y sólamente las hojas del arbol son soluciones.*

Análisis general

Siguiendo la línea de pensamiento anterior al ejemplo, teniendo en cuenta todas las ideas ya expuestas, calcular el gráfo completo del universo de soluciones (factibles y no factibles) para este problema tendría una complejidad de $\mathcal{O}(n^n)$, *o incluso infinita* si no se estableciese una asbtracción idónea al momento de transformar el universo de soluciones en un grafo. Además, incluso luego de establecer una abstracción en la que el grafo no considerase, por ejemplo, soluciones imposibles, el tamaño del grafo sigue siendo muy grande[10]. Por esta razón, para realizar esta tarea, se utiliza el concepto de *grafo implícito*, mediante el cual se establece matemáticamente la composición del mismo, sin la necesidad de expresar cada una de sus componentes, **sino a través de una descripción formal de sus nodos y aristas**, en donde **cada nodo es una solución**, y **cada arista es la transformación de una solución a una solución distinta dentro de su *vecindad*** de manera que, dado un caso de prueba particular, sea posible recrearlo en el momento mismo en que se realiza el recorrido del grafo.

Para asegurar la correctitud según los criterios expuestos en el párrafo anterior, es importante establecer una *vecindad* tal que el universo de soluciones tenga una subestructura adecuada para la realización de "podas", es decir, que dados dos $nodos - hijos$ derivados de un mismo $nodo - padre$, se puedan encontrar en estos características en común derivadas directamente de su padre; más concretamente, el nodo padre debe representar una solución o subsolución para la cual las funciones de selección y objetivo valúen de una forma idónea con relación a sus hijos. Por ejemplo, si en un ejercicio de optimización se lograse establecer un grafo de soluciones de tal forma que dado un nodo padre sus nodos derivados tengan una valuación menor o igual en la función objetivo, y se buscase la solución que la maximizara, sería fácil implementar entonces una poda que, en el momento en que una subsolución alcance un valor menor al deseado, *corte* la rama de soluciones determinada por el susodicho nodo.

Para este algoritmo en particular, el problema planteado es de **optimización**, ya que se desea obtener una solución válida (debe cumplir ciertos requisitos, como que cada pieza coincida en sus colores con sus aledañas), pero que además maximice la **función objetivo**, definiendo a esta última como, dado un tablero determinado, la sumatoria de los casilleros que contienen ficha.

Definición 3.3.1.1. Sean T el conjunto de todos los posibles tableros, definimos la función auxiliar $t.lleno : \mathbb{N}_{\geq 1} \rightarrow [0,1]$, tal que

$$\begin{cases} t.lleno(i) = 1 & \text{si T[i] contiene una pieza} \\ t.lleno(i) = 0 & \text{de lo contario} \end{cases}$$

La **función objetivo** es, dado un tablero $t \in T$, $f : T \rightarrow \mathbb{N}$, de la forma

$$f(T) := \sum_{i=1}^{\#casilleros} t.lleno\Big(T\,(i)\Big)$$

[10] De orden factorial, como se expone luego.

3.3.2. Planteamiento de resolución

Idea general

La idea es aplicar el algoritmo de backtracking según las consideraciones expuestas anteriormente. Para ello, recorreremos el tablero en un orden determinado (de izquierda a derecha, de arriba a abajo), evaluando en cada paso la ficha correspondiente a la posición donde se encuentra el recorrido: de tal forma que en cada paso se evalúe la posición siguiente a la que se evaluó en el paso anterior.

En cada paso del algoritmo se colocará una ficha, y luego se llamará recursivamente a la función, probando así las fichas posteriores. Es importante tener en cuenta que en cada llamada recursiva el algoritmo probará con M fichas, y se llamará recursivamente para cada una de ellas.

Es fácil ver que, *en su versión más básica*, este comportamiento lleva a tener una complejidad del orden de $\mathcal{O}\left(M^N\right)$, siendo $N = n * m$ la cantidad de posiciones del tablero, y M la cantidad de piezas, ya que se realizan N iteraciones (una por cada posición del tablero), y en cada iteración se prueban M piezas, lo cual resulta en $\overbrace{M * M * M * ... * M}^{N\ veces} = M^N$. Hay que tener en cuenta que esta forma de recorrer el universo de soluciones está analizando también las soluciones no factibles, es decir las que resultan imposibles, tal como "colocar la misma pieza en todos los casilleros".

Poda: Función de Selección

La primer poda posible resulta totalmente trivial, ya que se trata de probar en cada iteración sólamente las piezas que no se hayan colocado ya en el tablero. Aunque esta poda es simple, y no es más que una aplicación incompleta de las restricciones impuestas por la función de selección, se la menciona, ya que de esta forma se reduce la complejidad a $\mathcal{O}\left(\overbrace{(M) * (M-1) * ... * (M-(N-1)) * (M-(N))}^{\text{N multiplicaciones, restando una pieza en cada una}}\right)$, es decir $\mathcal{O}\left(\frac{M!}{(N-1)!}\right) \subseteq \mathcal{O}(M!)$. Simplificando, diremos que esta cota reduce la complejidad del orden exponencial $\mathcal{O}\left(M^N\right)$ al orden factorial $\mathcal{O}(M!)$.

Sobre la misma poda anterior, se puede implementar una mejora, que es probar en cada iteración sólamente las piezas cuyos colores sean coherentes con los colores de las piezas que ya se encuentren colocadas en el tablero. Así, si una posición tiene una pieza a la izquierda cuyo color derecho es verde, yo debería poder colocar en esa posición sólamente las piezas cuyo color izquierdo es verde. Esta última poda, tal y como está planteada, en realidad se trata básicamente de **reducir el recorrido del universo de soluciones a "las soluciones factibles"**, es decir, las que generan un tablero final válido. Conceptualmente no es distinta a la poda mencionada en el párrafo anterior, sino que es más bien una extensión de la misma, ya que las dos se encargan de analizar en cada iteración la correctitud del tablero en términos de la **función de selección**. De hecho, esta poda se encarga de **seleccionar sólamente las fichas que cumplan *estrictamente* con la función de selección**, mientras que la anterior sólo aplica la restricción de forma parcial.

Es imposible, sin embargo, establecer *a priori* una caracterización precisa del orden de complejidad del algoritmo bajo las condiciones de esta nueva poda, ya que la misma depende no solo del tamaño de entrada, sino del contenido de la entrada. Puede darse, por ejemplo, el caso de una entrada en la que todas las piezas tengan el mismo color en todos sus lados, lo cual significaría que habría que probar todas las piezas en todas las posiciones, lo cual conllevaría una complejidad de $\mathcal{O}(M!)$. También puede darse el caso en que la disposición de la entrada sea de forma tal que luego de colocar la primer pieza, sólo exista en cada paso una pieza posible, con lo cual la complejidad sería del orden de $\mathcal{O}\left(M * 1^{N-1}\right)$, es decir $\mathcal{O}(M)$, lineal sobre la cantidad de piezas. Por la razón expuesta en este párrafo diremos que, a pesar de que según la disposición de la entrada la misma puede ser mucho menor, la complejidad en el peor caso está igualmente acotada por $\mathcal{O}(M!)$.

Por todo lo antes expuesto, además, podemos asegurar que al final del recorrido (es decir, luego de evaluar todas las hojas del árbol) se habrán evaluado *estrictamente* todas las **soluciones factibles**, ya que el algoritmo habrá recorrido en cada paso cada posición del tablero, **evaluando las piezas que cumplan estrictamente con el criterio de selección**.

Poda: Función Objetivo

Como se expuso en párrafos anteriores, la solución brindada por el algoritmo debe no solo cumplir con las condiciones de una determinada función de selección, sino también maximizar cierta función objetivo (*ref: 3.3.1.1*), la cual depende de la cantidad de piezas dispuestas en el tablero o, inversamente, de la cantidad de "agujeros", es decir, posiciones en que no exista ninguna pieza.

Dado que las soluciones comparten una subestructura, en donde tomando un determinado nodo d[11] del arbol de soluciones podemos asegurar que «para todas las hojas para las que exista un camino simple hacia ese nodo que contenga a los hijos de ese nodo»[12] la solución representada por esas hojas contiene a todas las piezas que ya se encuentran determinadas en el nodo d, es válido afirmar que si una deteriminada subsolución contiene cierta cantidad de "agujeros", entonces todas sus soluciones derivadas también van a contener esos mismos "agujeros". Así, podemos decir que *la cantidad de agujeros de una hoja es siempre mayor o igual a la cantidad de agujeros de su padre*.

Nota. Antes de continuar, vale hacer una aclaración ligada a la implementación, y es que a efectos de abstraer este concepto, y mejorar la claridad del algoritmo se consideró al "agujero" o "casillero libre" como otra pieza, representada por la constante `PIEZA_VACÍA` (es decir, independientemente del *tipo* con el que se elija representar a la pieza, nos referiremos a esta pieza como `PIEZA_VACÍA`).

A la luz de las consideraciones anteriores surge esta poda: el objetivo es **cortar una *rama***[13] en el momento en que se encuentra una subsolución que, por su cantidad de piezas vacías, **no es candidata a maximizar la función objetivo** (y ya que no lo es la subsolución, por las consideraciones ya expuestas, se puede asumir que tampoco lo serán las soluciones derivadas de ella).

[11] Notación: llamémoslo «subsolución»
[12] O coloquialmente: que exista una "relación de parentezco" con ese nodo
[13] Es decir, una subsolución junto con todas sus soluciones derivadas

Nota. A efectos de simplificar la explicación, procedemos a mencionar primero una versión alternativa de esta poda, la cual quizás resulte más intuitiva. Dada la recursión una determinada posición n, la cual define una subsolución T_n, es fácil ver que si T_n contiene p `piezas no-vacías`, siendo N la cantidad de posiciones, se cumple que:

- Las soluciones derivadas de T_n contienen al menos p piezas.
- Las soluciones derivadas de T_n contienen a lo sumo $p+(N-n)$ piezas.

De esta forma, si uno va guardando la cantidad máxima de piezas encontradas, uno puede cortar una rama cuando sabe que cualquier solución derivada de la misma contendrá una cantidad menor.

Definición 3.3.2.1. Dada una determinada subsolución T_n, en donde n representa a la posición sobre la cual se encuentra la recursión, decimos que la misma contiene una «*cantidad máxima posible de piezas vacías*», y esta es la cantidad de piezas vacías que el tablero tendría en el peor caso. En otras palabras, **la cantidad máxima de piezas vacías de un tablero T_n es la cantidad de piezas vacías que ya contiene la subsolución T_n sumada a todas las piezas que aún le faltan por poner en las sucesivas $T_{n+...}$**. Se asume para ello que el peor caso posible es un tablero en donde todas las piezas $\in \{n+1, ..., N\}$, siguientes a la posición actual resulten incompatibles y, por lo tanto, vacías. La cantidad de piezas de ese intervalo es, entonces, $N - n$.

$$cantidadMaximaAgujeros(T_n) = cantidadAgujeros(T_n) + (N - n)$$

Implementación de la poda

En un principio, podemos asumir que toda combinación posible de fichas es **candidata a ser mejor solución**.

En cada paso recursivo, podemos analizar cuánto es el valor de `cantidadMaximaAgujeros(`T_n`)` para el tablero T_n actual de ese paso, y contrastarlo contra la **menor cantidad máxima de agujeros encontrada hasta el momento**, la cual desde el punto de vista implementativo es una variable ajena a la recursión (por ejemplo, una variable global, o un parámetro de la propia recursión). Siempre que se encuentre una «cantidad máxima de agujeros» menor a la global, se actualizará el valor de la segunda. Para ello, se inicializará esta última variable con el valor del peor caso posible ($N \times M$), es decir el de un tablero en donde todas las piezas sean vacías.

Como dijimos anteriormente, **dada una subsolución** S, podemos asegurar que todas sus soluciones derivadas tendrán al menos la misma cantidad de piezas vacías. Utilizando esta última consideración, dada una subsolución con k_S piezas vacías, y sea X_{min} la `menor cantidad máxima de agujeros encontrada hasta el momento`, de tal forma que $X_{min} <= k_S$, podemos afirmar que la cantidad de `piezas no vacías` de cualquier solución derivada de esa **subsolución** k_S no es en particular **mayor** a la cantidad de `piezas no vacías` del resto de las soluciones, ya que por lo antes expuesto sabemos que dada la existencia de X_{min} existe al menos una subsolución T tal que su cantidad máxima de piezas vacías es exactamente $k_T = X_{min} <= k_S$, lo cual es equivalente a afirmar que la ***cantidad mínima de piezas no vacías de cualquier solución derivada de T*** es $N - k_T = N - X_{min}$, y dado que $k_T <= k_S$ se deduce que $(-K_T) >= (-kS)$, de lo que finalmente se desprende que $T - K_T >= T - k_S$ es decir la *cantidad mínima de piezas no vacías de cualquier solución derivada* de T es mayor o igual a la *cantidad mínima de piezas no vacías de cualquier solución derivada* de S.

De todo lo anterior surge que mediante esta poda se logra descartar ciertas soluciones, cuando se está seguro de que las mismas no maximizan la función objetivo o, en caso de hacerlo, de que ya se encontró[14] al menos una solución que la maximice de igual o mejor forma.

Corolario 3.3.2.1. *Dado un tablero de T **posiciones**, y T **piezas cualesquiera**, sin tomar a consideración sus colores, podemos asegurar que la **cantidad mínima de piezas** que podemos colocar en el mismo es de $\lceil T/2 \rceil$, de forma tal que las piezas se distribuyan tal y como los casilleros de un mismo color en un `Tablero de Ajedrez`. Esto es así debido a que esta distribución nos asegura que en las posiciones indicadas ninguna pieza tiene contacto inmediato con cualquier otra pieza, por lo que no existe riesgo de incompatibilidad. De forma recíproca, podemos también asegurar que la **menor cantidad máxima de agujeros** que el tablero puede llegar a contener es de a lo sumo $\lfloor T/2 \rfloor$. Estos valores pueden ser utilizados, entonces, como inicialización de las variables globales mencionadas en la explicación de la poda.*

Esta misma observación nos permite, también, afirmar que dado un tablero cualquiera existe al menos una solución factible.

[14] o al menos, implícitamente, se tiene certeza de su existencia

3.3.3. Detalle del algoritmo

```
Tablero backtrack( Tablero t, IndiceDePiezas ip, int posicion )
{
  Tablero mejorTablero = NULL;
  if ( t.yaEncontreUnTableroMejor( posicion ) )
  {
    return mejorTablero;
  }
  IteradorIndiceDePiezas it = ip.dameIterador( posicion );
  // Me fijo si estoy antes de la ultima posicion
  if ( posicion < t.cantidadDePosiciones - 1 )
  {
    while ( it.hayPiezasPosibles() )
    {
      // Si es asi, entonces llamo a backtrack para cada pieza posible
      t.ponerPiezaEnPosicion( *it, posicion );
      ip.marcarPiezaUtilizada( it );
      // Hago recursion en el backtracking
      Tablero otroTablero = NULL;
      otroTablero = backtrack( t, ip, posicion + 1 );
      if ( !mejorTablero )
      {
        mejorTablero = otroTablero;
      }
      else
      {
        if ( !otroTablero )
        {
          ip.marcarPiezaDisponible ( it );
          // Si la recursion me devolvio un puntero a nulo termino el BT
          break;
        }
        else
        {
          if ( mejorTablero < otroTablero )
          {
            mejorTablero = otroTablero;
          }

        }
      }
      ip.marcarPiezaDisponible ( it );
      it++;
    }
  }
  else
  {
    // (si estoy en la ultima posicion del tablero)
    // Intento colocar la ultima pieza
```

```
    if ( it.hayPiezasPosibles() )
    {
      t.ponerPiezaEnPosicion( *it, posicion );
    }
    // Creo una copia del tablero final
    mejorTablero = new Tablero( t );
    // Pongo una pieza transparente en el lugar donde puse (o no) una pieza
    t.ponerPiezaEnPosicion( PIEZA_VACIA, posicion );
  }
  return mejorTablero;
}
```

3.3.4. Justificación formal de correctitud

Se demostró la correctitud de este algoritmo y sus podas en la propia descripión del mismo: por un lado afirmamos que la **correctitud de un algoritmo genérico de Backtracking**, en su sentido más básico, **resulta trivial**, amparandonos en la propia definición de **Backtracking**, siendo que el algoritmo "*analiza el universo de soluciones*". Luego, durante la descripción de las dos podas implementadas, se demostró que, en el caso de la poda relacionada con la «Función de Selección» se estaban eliminando las soluciones no factibles, las cuales iban a ser de todas formas descartadas por el algoritmo durante el recorrido de las soluciones, y que en el caso de la poda relacionada con la «Función Objetivo» se estaban eliminando todas aquellas soluciones para las que dadas las condiciones de la subestructura implementada se podía preveer que no iban a cumplir con el objetivo de maximizar la función requerida.

3.3.5. Cota de complejidad temporal

La complejidad de este algoritmo pertenece a la familia de $\mathcal{O}(M!)$, en donde M es la cantidad de piezas (contando la pieza vacía). Esto está demostrado en la sección `3.3.2:Planteamiento de resolución` (página 11), en el apartado de la poda relacionada con la función de selección.

3.3.6. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente.

Input			
n	m	c	
sup_1	izq_1	der_1	inf_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
sup_{nxm}	izq_{nxm}	der_{nxm}	inf_{nxm}

Output		
x_1	$\dots$	x_m
$\vdots$		$\vdots$
x_n	$\dots$	x_{nxm}

- n: #filas.
- m: #columnas.
- c: #colores.
- sup_i, izq_i, der_i, inf_i: colores (entre 1 y c) de los lados de la pieza i.
- x_i: número de la pieza en la casilla i del tablero. ("0" si no hay ninguna pieza).

Separamos en casos y mostramos ejemplos para cada uno:

- Todas las piezas son iguales:

- El tablero se completa.

Input	Output
2 2 1	1 2
1 1 1 1	3 4
1 1 1 1	
1 1 1 1	
1 1 1 1	

Input	Output
2 2 2	1 2
1 2 2 1	3 4
1 2 2 1	
1 2 2 1	
1 2 2 1	

En estos ejemplos el tablero se completaría colocando las 4 piezas de cualquier forma. No hay restricciones.

- El tablero tiene la mínima cantidad de piezas ($(n \times m)/2$).

Input	Output	Output
2 2 2	1 0	0 1
1 2 1 2	0 2	2 0
1 2 1 2		
1 2 1 2		
1 2 1 2		

En este caso sólo se pueden colocar las piezas intercaladas. Se podrían tomar 2 piezas cualesquiera para la solución (Ej: piezas 3 y 4).

- Todas las piezas son diferentes:

  - El tablero se completa.

Input	Output
3 3 8	9 2 3
8 2 6 2	4 5 6
1 1 2 4	7 8 1
1 2 7 5	
3 4 4 6	
4 4 3 7	
5 3 6 8	
6 4 1 2	
7 1 2 7	
8 8 1 3	

  - El tablero tiene la mínima cantidad de piezas ($(n \times m)/2$).

Input	Output	Output
2 2 3	1 0	0 1
2 3 2 2	0 2	2 0
1 1 2 2		
1 3 2 2		
2 1 2 2		

En este caso sólo se pueden colocar las piezas intercaladas. Se podrían tomar 2 piezas cualesquiera para la solución (Ej: piezas 3 y 4).

- El tablero no se completa pero se llena más que el mínimo posible.

Input	Output
2 2 3	1 2
1 2 1 1	0 3
1 1 1 2	
2 2 3 2	
3 2 3 3	

- Existe alguna pieza diferente al resto:
  - El tablero se completa.

Input	Output
2 2 2	1 3
2 2 1 2	2 4
2 2 1 2	
2 1 2 2	
2 1 2 2	

  - El tablero tiene la mínima cantidad de piezas ($(n \times m)/2$).

Input	Output	Output
2 2 3	1 0	0 1
2 2 1 2	0 4	2 0
3 3 3 1		
3 2 1 1		
2 2 1 2		

En este caso sólo se pueden colocar las piezas intercaladas. Se podrían tomar 2 piezas cualesquiera para la solución (Ej: piezas 3 y 4).

  - El tablero no se completa pero se llena más que el mínimo posible.

Input	Output
2 2 3	1 3
2 2 1 2	0 4
3 2 3 1	
3 1 3 2	
2 2 1 2	

Ejecutamos el programa con los distintos ejemplos y se llegó a la solución esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

3.3.7. Medición empírica de la performance

Para realizar las pruebas de tiempo se nos presentaron varias complicaciones. Dado que la complejidad del algoritmo implementado es exponencial, el tiempo de ejecución se vuelve inmenso ante tamaños de entrada relativamente pequeños. Por ejemplo, dado un tamaño de entrada de $n = 4$, $m = 4$, $c = 10$, tenemos una cantidad de fichas $M = (n * m) + 1 = 17$, y dado que la complejidad del algoritmo es $\mathcal{O}(M!)$, esto es equivalente a afirmar que existe una constante k para la cual a partir de un n_0 nuestro algoritmo está acotado superiormente por $k * (M!)$. Puesto que decimos que este $M!$ en realidad surge de la cantidad de soluciones analizadas, si imaginásemos que el procesador demorase un ciclo de clock en calcular una solución[15], averiguar todo el universo de soluciones bajo las condiciones expuestas en este ejemplo tardaría a lo sumo 17! ciclos de clock o, suponiendo un procesador de 3GHz[16], unos 110419 segundos (aproximadamente media hora).

Debido a lo expuesto anteriormente, nos resultó casi imposible medir el comportamiento del algoritmo para tamaños de tablero muy grandes. Se realizó una primer medición comparando el rendimiento del algoritmo con todas sus podas, contra el algoritmo sin la "poda objetivo" y el algoritmo sin ambas podas.

Se realizaron distintas mediciones, para las cuales se crearon casos de prueba particulares. Se adjunta al informe, dentro de la carpeta del ejercicio, archivos

- **input_output_testSinPodas.tar.bz**
- **input_output_testExponencial.tar.bz**
- **input_output_testConPodas.tar.bz**
- **input_output_testNormal.tar.bz**

los cuales contienen los input y output utilizados respectivamente para generar los sucesivos gráficos.

Para realizar las pruebas, en algunos casos (si el tiempo de ejecución no era alto) se midió varias veces sobre un mismo caso de prueba, y para eliminar el ruido o las mediciones erróneas se los filtró tomando luego en cada x los valores cuartiles superior e inferior correspondientes al tiempo, promediándolos posteriormente. En el caso de tomar una sola muestra por cada valor de x, se los volcó luego en el gráfico sin realizarles ningún tratamiento.

Como se puede apreciar en el **`gráfico 1`**, el cual muestra el tamaño de entrada (n*m) en relación con el tiempo en segundos (en escala logarítmica) el algoritmo presenta comportamientos parecidos ante instancias de entrada pequeña, y las mismas se empiezan a diferenciar de forma notoria a partir del tamaño de entrada 6. Todas las entradas utilizadas fueron generadas de forma pseudoaleatoria. En el caso del caso "sin podas", se testeó el Backtracking puro, es decir, sin ningún tipo de poda aplicada. Es notable ver cómo ante un tamaño de entrada 9 (tablero de 3x3) el algoritmo sin podas tardó aproximadamente un minuto. Se intentó testear para tamaños de entrada más grandes, pero los tiempos de ejecución fueron muy grandes, por lo que no resultó práctico realizar mediciones exploratorias para este tipo de tamaños. En el caso de la medición "sin poda objetivo", se eliminó sólamente la poda sobre la función de selección, la encargada de descartar las ramas que derivaran en tableros imposibles. Así se puede apreciar una mejora notable en el rendimiento del algoritmo frente a la versión sin podas. Tener en cuenta sobre todo la escala logarítmica del `eje Y`, la cual implica

[15] Es el mínimo valor que le podemos asignar sin involucrarnos demasiado con la arquitectura/microarquitectura del procesador

[16] Es decir, $3 * 2^30 ciclos por segundo$.

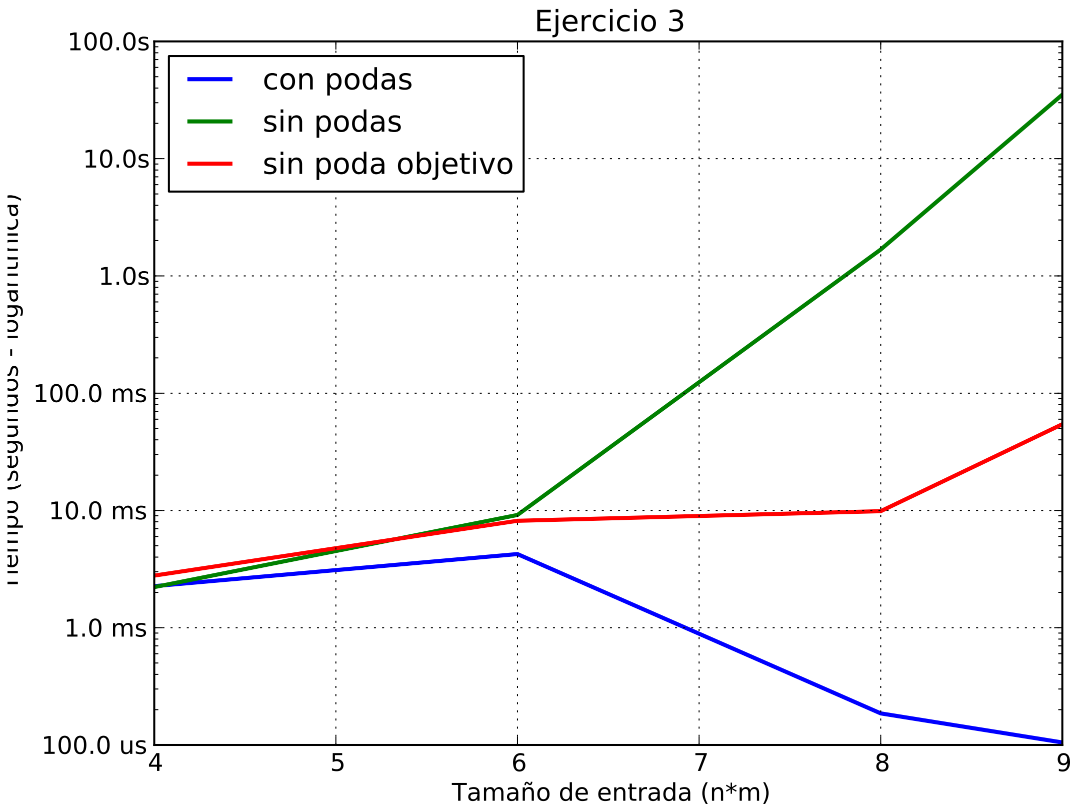

Figura 1: Comparación de las podas

que el algoritmo tardó unas 1000 veces menos en relación al tiempo de ejecución sin podas[17]. En el caso del algoritmo "con podas", se puede apreciar un descenso en el tiempo a partir del tamaño de entrada 6. Esto podría parecer extraño, pero su explicación es que dada la forma de la entrada, los tableros más pequeños resultaban imprácticos para la aplicación de la poda, por lo cual el rendimiento del algoritmo era casi el mismo con podas que sin podas. Luego, al crecer el tamaño del tablero, crece la utilización de la poda, con lo cual parecería que el algoritmo tarda "cada vez menos"[18]. Si se pudiese extender el análisis a tamaños mayores a 9, seguramente se podría ver cómo el algoritmo crece luego de esa "bajada".

En **`gráfico 2`** se hizo una comparación de la complejidad según el tipo de entrada proporcionada. Se estandarizaron para ello tres tipos de casos:

- aleatorio: Dados n, m y la cantidad de colores, se generan `n*m` fichas en donde para cada una todos sus colores son un número pseudoaleatorio entre 1 y cantidad de colores.
- trivial: Dados n, m y la cantidad de colores, se generan `n*m` fichas, en donde todas tienen el mismo color.
- piezas incompatibles: Dados n, m y la cantidad de colores, se generan `n*m` fichas en donde dados los colores 1,2,3 y 4, todas tienen 1 arriba, 2 abajo, 3 a la izquierda y 4 a la derecha. Es fácil ver que todas las fichas son incompatibles en este escenario, por lo

[17] Dado que el logaritmo es en base 10, una marca de línea punteada sobre el eje representa un aumento de 10 veces sobre la marca anterior. Así, es fácil calcular cuánto se diferencian los algoritmos

[18] Notar que su tope es en 100us, y no en 0.

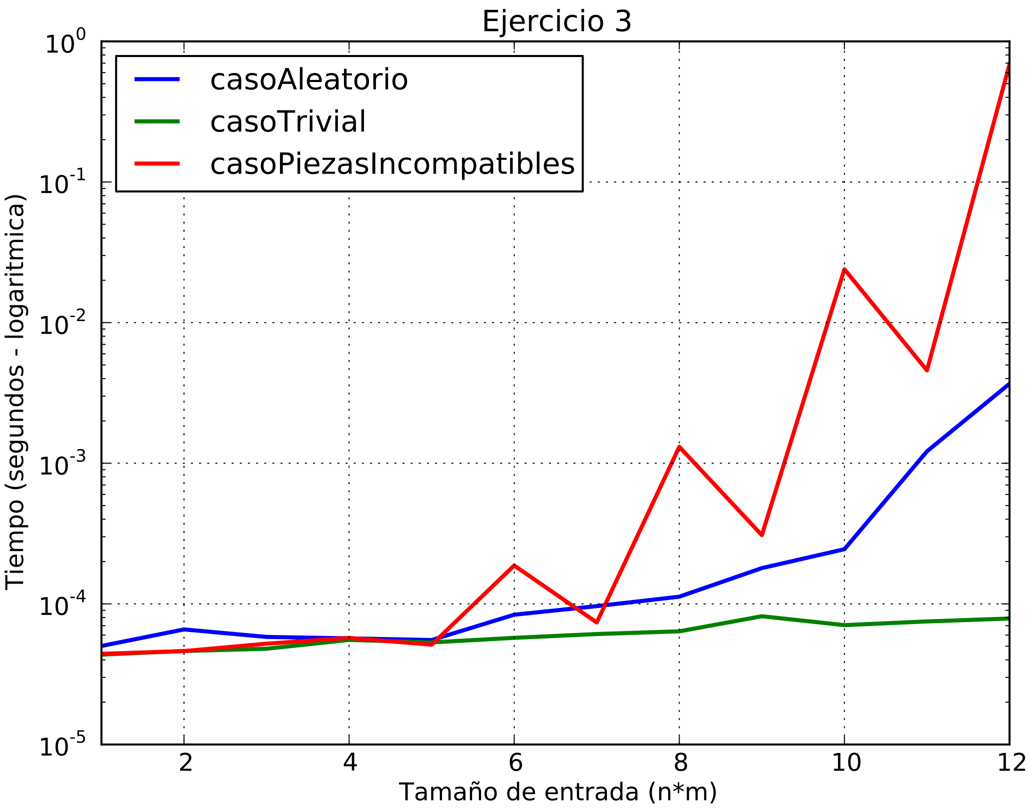

Figura 2: Comparación de la complejidad según el tipo de entrada

que el único tablero posible es el que las fichas se intercalan para no tocarse unas con otras.

En el gráfico se puede apreciar que el comportamiento para el caso de piezas incompatibles es muy malo, ya que la poda objetivo no puede ser aplicada, por lo que la recursión debe cubrir todos los casos factibles. La complejidad en este caso sería de $\mathcal{O}(M!)$, como ya se expuso debidamente. Para el caso trivial, el comportamiento es casi constante (por lo que se puede apreciar en el gráfico), y lo que se puede afirmar del caso aleatorio es que está acotado por los otros dos casos; aunque empíricamente, luego de haber realizado sucesivos testeos, parecería válido afirmar que su comportamiento es mucho mejor que "el peor caso".

Por último, en **`gráfico 3`** y **`gráfico 4`** se quiso reflejar el comportamiento de nuestro algoritmo para el mejor caso (3), y para los casos en que no se puede asumir nada (4). A diferencia de los otros gráficos en los que el contraste se realizó contra versiones de menor eficiencia, aquí nos permitimos extender el rango de entrada, con lo cual se puede apreciar la naturaleza exponecial del algoritmo en el caso de los casos aleatorios, y el hecho de que la poda realiza un muy buen trabajos para tamaños de entrada gigantes, como 4000, en el caso en que la entrada sea favorable. En el **`gráfico 3`** se puede ver concretamente el `comportamiento de la poda`, y no representa un caso real, ya que las entradas fueron elegidas particularmente para la muestra. Se aprecia "cuán buena" puede llegar a ser la poda. Recordar que el eje se encuentra en escala logarítmica.

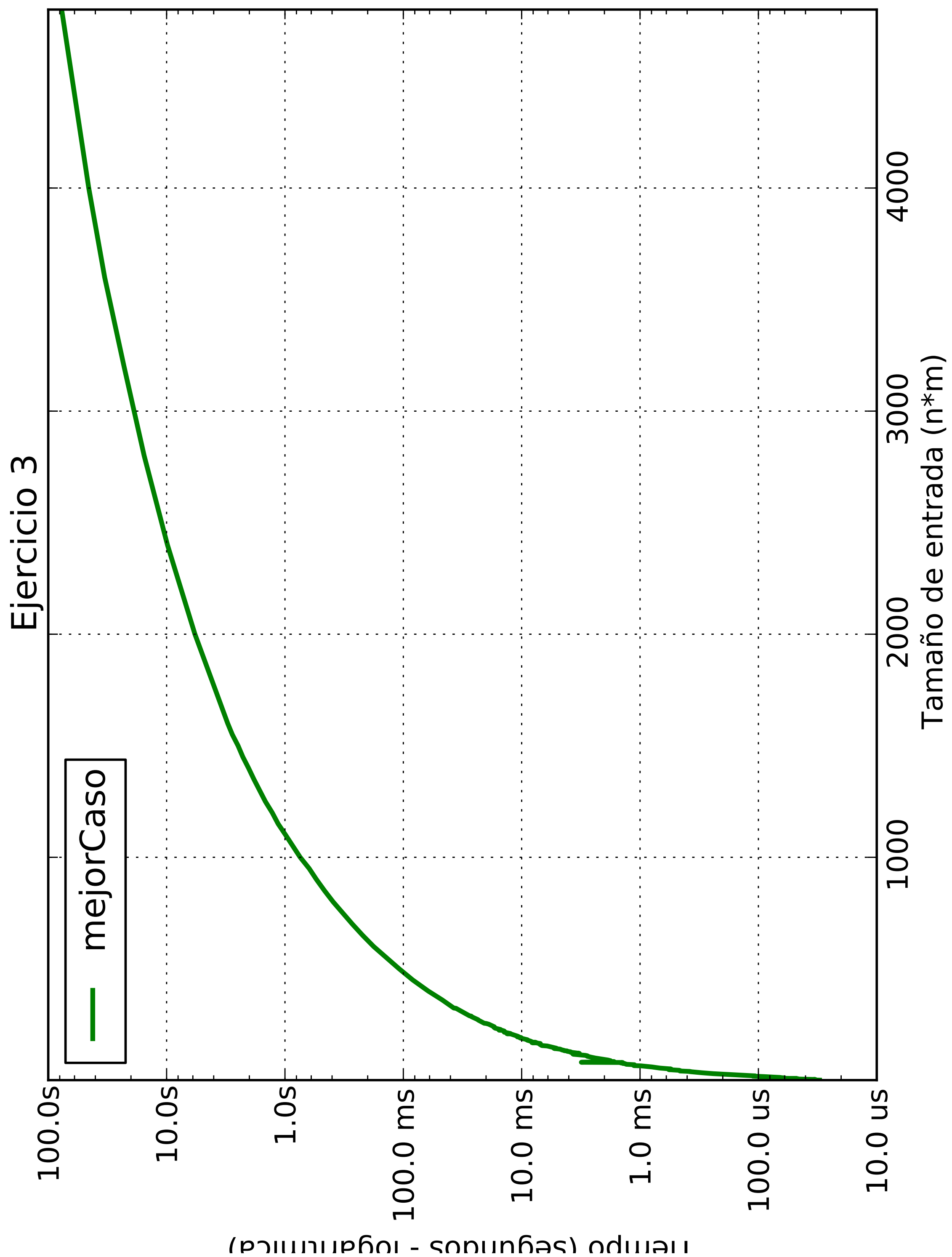

Figura 3: Comportamiento en el mejor caso

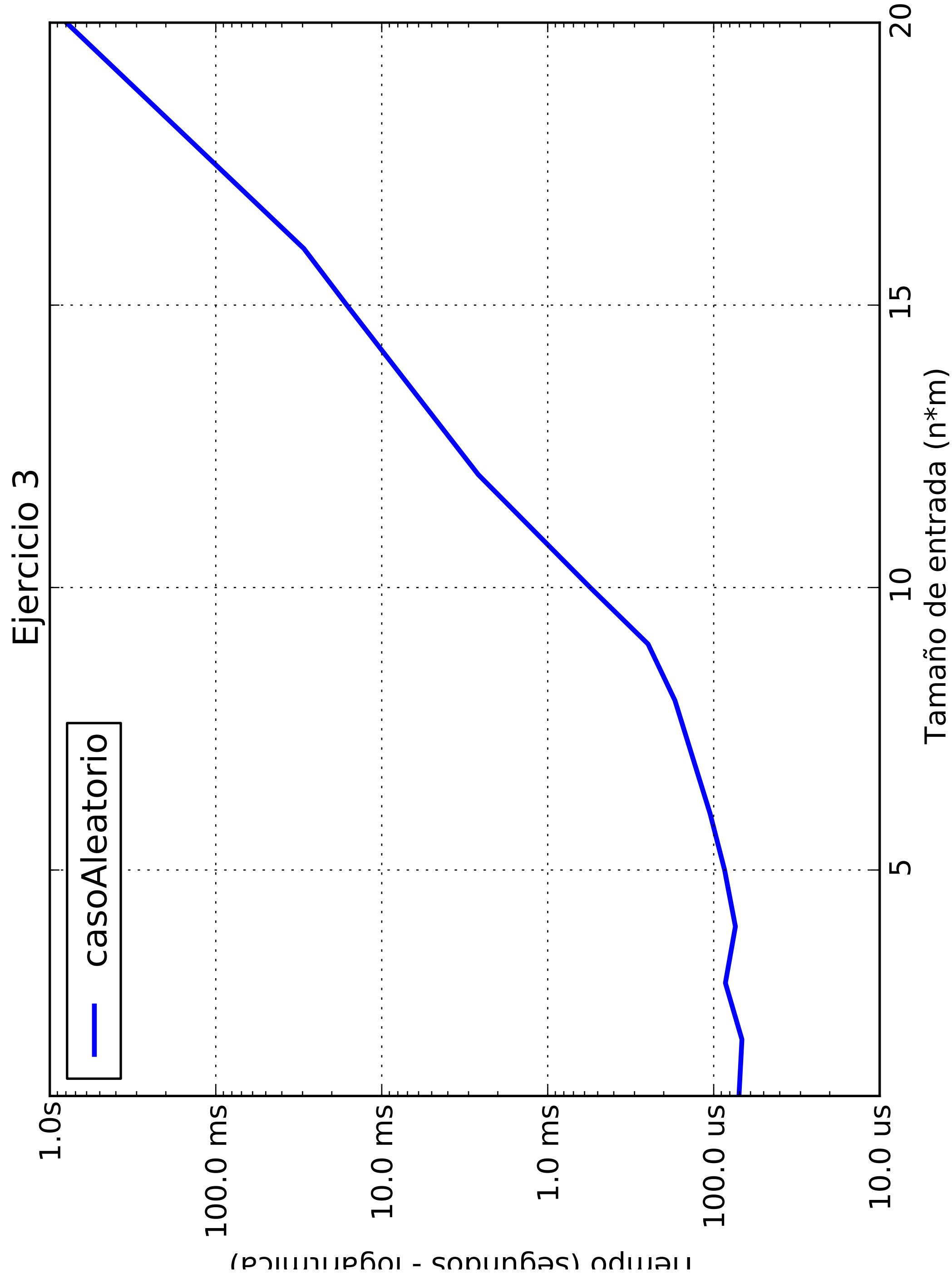

Figura 4: Comportamiento en casos aleatorios

4. Apéndices

4.1. Código Fuente (resumen)

4.1.1. Problema 3: Rompecolores

ej3

ej3.h

```
#ifndef EJ3_H
#define EJ3_H

#include "../common/headers.h"
#include "../common/ParserDeParametros.h"
#include "../common/Timer.h"
#include "TestCaseEj3.h"

#endif
```

ej3.cpp

```
#include "ej3.h"
#include "Tablero.h"
#include "IndiceDePiezas.h"

Tablero& backtrack( Tablero&, IndiceDePiezas&, uint32_t );
void imprimeTablero( Tablero& );

int main( int argc, char** argv )
{
  // Parseo los parametros con que fue llamado el ejecutable
  ParserDeParametros parser( argc, argv );
  // Esta clase representa un caso de prueba, y lo toma desde el input que
      le provee el parser
  TestCaseEj3 testcase ( parser.dameInput() );
  Timer timer ( parser.dameTime() );

  // Itero sobre los distintos casos de prueba hasta obtener un testcase
      nulo
  while ( testcase.tomarDatos() != false )
  {
    // Mido el tiempo inicial.
    timer.setInitialTime( "todoElCiclo" );
    // Obtengo los parametros del testcase.
    uint32_t cantidadDeFilas = testcase.dameCantidadDeFilas();
    uint32_t cantidadDeColumnas = testcase.dameCantidadDeColumnas();
    uint32_t cantidadDeColores = testcase.dameCantidadDeColores();
    vector<TestCaseEj3::Pieza>& listaDePiezas = testcase.dameListaDePiezas
        ();
    // Inicializo el tablero (estructura auxiliar)
    Tablero tablero( cantidadDeFilas, cantidadDeColumnas, listaDePiezas );
    // Inicializo el indice de piezas (estructura auxiliar)
    IndiceDePiezas indiceDePiezas( cantidadDeColores, listaDePiezas,
        tablero );
    //Obtengo el mejor tablero a traves de backtracking
    Tablero& mejorTablero = backtrack( tablero, indiceDePiezas, 0 );
    // Mido el tiempo final
    timer.setFinalTime( "todoElCiclo" );
    timer.saveAllTimes();
```

```
    // Devuelvo el resultado con el formato solicitado
#ifndef TIME

    for ( uint32_t columna = 0; columna < mejorTablero.cantidadDeColumnas;
       columna++ )
    {
      for ( uint32_t fila = 0; fila < mejorTablero.cantidadDeFilas; fila++
         )
      {
        uint32_t posicion = ( columna * mejorTablero.cantidadDeFilas ) +
           fila;
        uint32_t pieza = mejorTablero[ posicion ];
        parser.dameOutput() << pieza << " ";
      }

      parser.dameOutput() << endl;
    }

#endif
    delete &mejorTablero;
  }

  return 0;
}

Tablero& backtrack( Tablero& t, IndiceDePiezas& ip, uint32_t posicion )
{
  DEBUG_ENTER; _C( "Entrando a recursion en posicion: " << posicion + 1 );
  Tablero* mejorTablero = NULL;
#ifndef SINPODAOBJETIVO

  if ( t.yaEncontreUnTableroMejor( posicion ) )
  {
    _C( "PODANDO EN POS " << posicion << " PORQUE EXISTE UN TABLERO mejor
       QUE CUALQUIERA DE ESTA RAMA" );
    return *mejorTablero;
  }

#endif
  IteradorIndiceDePiezas& it = ip.dameIterador( posicion );
  _C( "Obtenido iterador al indice de piezas" );

  // Me fijo si estoy antes de la ultima posicion
  if ( posicion < t.cantidadDePosiciones - 1 )
  {
#ifndef SINOPTIMIZACION

    while ( it.hayPiezasPosibles() )
    {
      _C( "Pieza disponible: " << *it );
      // Si es asi, entonces llamo a backtrack para cada pieza posible
      t.ponerPiezaEnPosicion( *it, posicion );
      ip.marcarPiezaUtilizada( it );
      // Hago recursion en el backtracking
      Tablero* otroTablero = NULL;
      otroTablero = &( backtrack( t, ip, posicion + 1 ) );
      _C( "VOLVIENDO A POSICION " << posicion );

      if ( !mejorTablero )
      {
        mejorTablero = otroTablero;
      }
      else
```

```
      {
        if ( !otroTablero )
        {
          ip.marcarPiezaDisponible ( it );
          // Si la recursion me devolvio un puntero a nulo termino el BT
          break;
        }
        else
        {
          if ( *mejorTablero < *otroTablero )
          {
            delete mejorTablero;
            mejorTablero = otroTablero;
          }
          else
          {
            delete otroTablero;
          }
        }
      }

      ip.marcarPiezaDisponible ( it );
      it++;
    }

#else

    for ( uint32_t it = 1; it < t.cantidadDePiezas(); it++ )
    {
      t.ponerPiezaEnPosicion( it, posicion );
      // Hago recursion en el backtracking
      Tablero* otroTablero = NULL;
      otroTablero = &( backtrack( t, ip, posicion + 1 ) );

      if ( !mejorTablero )
      {
        mejorTablero = otroTablero;
      }
      else
      {
        if ( !otroTablero )
        {
          // Si la recursion me devolvio un puntero a nulo termino el BT
        }
        else
        {
          if ( *mejorTablero < *otroTablero )
          {
            delete mejorTablero;
            mejorTablero = otroTablero;
          }
          else
          {
            delete otroTablero;
          }
        }
      }
    }

    t.ponerPiezaEnPosicion( TestCaseEj3::PIEZA_VACIA, posicion );
    Tablero* otroTablero = NULL;
    otroTablero = &( backtrack( t, ip, posicion + 1 ) );

```

```
    if ( !mejorTablero )
    {
      mejorTablero = otroTablero;
    }
    else
    {
      if ( !otroTablero )
      {
        // Si la recursion me devolvio un puntero a nulo termino el BT
      }
      else
      {
        if ( *mejorTablero < *otroTablero )
        {
          delete mejorTablero;
          mejorTablero = otroTablero;
        }
        else
        {
          delete otroTablero;
        }
      }
    }

#endif
  }
  else
  {
    // (si estoy en la ultima posicion del tablero)
    // Intento colocar la ultima pieza
    if ( it.hayPiezasPosibles() )
    {
      t.ponerPiezaEnPosicion( *it, posicion );
    }

    // Creo una copia del tablero final
    mejorTablero = new Tablero( t );
    // Pongo una pieza transparente en el lugar donde puse (o no) una pieza
    t.ponerPiezaEnPosicion( TestCaseEj3::PIEZA_VACIA, posicion );
  }

  delete ( &it );
  _C( "Saliendo de recursion en posicion: " << posicion + 1 ); DEBUG_ENTER;
  return *mejorTablero;
}

/**
 * Dado un tablero, lo imprime a consola (stderr).
 */
void imprimeTablero( Tablero& t )
{
  for ( uint32_t y = 0; y < t.cantidadDeFilas; y++ )
  {
    for ( uint32_t x = 0; x < t.cantidadDeColumnas; x++ )
    {
      uint32_t posicion = y * t.cantidadDeColumnas + x;
      cerr << t[posicion] << " ";
    }

    cerr << endl;
  }

  cerr << endl;
```

```
}
```

Tablero

Tablero.h

```
#ifndef __TABLERO_H__
#define __TABLERO_H__
#include "ej3.h"

class Tablero
{
private:
  vector<TestCaseEj3::Pieza>& _listaDePiezas;
  vector<uint32_t> _piezasEnElTablero;
  uint32_t _cantidadDePosicionesVacias;
  uint32_t _mejorCantidadDePosicionesVacias;

  // Poda
  uint32_t _mejorCantidadDePosicionesVaciasPosible;
  void _calcularMejorCantidadDePiezasPosible();
  uint32_t _ultimaPosicionAgregada;
public:
  Tablero( uint32_t p_filas, uint32_t p_columnas, vector<TestCaseEj3::Pieza>& );
  uint32_t cantidadDePiezas( void );
  /**
   * Operador binario, indica si un tablero es menor a otro
   */
  inline bool operator< ( Tablero& t )
  {
    return this->_cantidadDePosicionesVacias > t._cantidadDePosicionesVacias;
  }
  /**
   * Permite acceder a la pieza ubicada en una posicion determinada
   */
  const inline uint32_t& operator[] ( uint32_t posicion ) const
  {
    return this->_piezasEnElTablero[posicion];
  }
  /**
   * Dada una posicion en el tablero,
   * devuelve la pieza que se encuentra a la izquierda
   */
  const uint32_t dameLaPiezaDeIzquierdaDePosicion( uint32_t );
  /**
   * Dada una posicion en el tablero,
   * devuelve la pieza que se encuentra arriba
   */
  const uint32_t dameLaPiezaDeArribaDePosicion( uint32_t );
  /**
   * Dada una posicion en el tablero,
   * devuelve la pieza que se encuentra en esa posicion
   */
  //const uint32_t dameLaPiezaEnPosicion( uint32_t );
  /**
   * Coloca una pieza en la posicion indicada
   */
  void ponerPiezaEnPosicion( uint32_t, uint32_t );
  const uint32_t cantidadDeFilas;
  const uint32_t cantidadDeColumnas;
  const uint32_t cantidadDePosiciones;
  inline const uint32_t cantidadDePosicionesLlenas( void )
```

```
  {
    return this->cantidadDePosiciones - this->_cantidadDePosicionesVacias;
  };
  //uint32_t mejorTableroHastaElMomento( void );
  bool yaEncontreElMejorTableroPosible( void );
  bool yaEncontreUnTableroMejor( uint32_t );
  void imprimeTablero();
private:

};

#endif
```

Tablero.cpp

```
#include "Tablero.h"

Tablero::Tablero( uint32_t p_filas, uint32_t p_columnas, vector<TestCaseEj3
    ::Pieza>& listaDePiezas )
  :
  _listaDePiezas( listaDePiezas ),
  _cantidadDePosicionesVacias( p_filas* p_columnas ),
  _mejorCantidadDePosicionesVacias( _cantidadDePosicionesVacias / 2 + 1 ),
  cantidadDeFilas( p_filas ),
  cantidadDeColumnas( p_columnas ),
  cantidadDePosiciones( p_filas* p_columnas )
{
  this->_piezasEnElTablero.assign( p_filas * p_columnas, 0 );
  // Poda
  this->_calcularMejorCantidadDePiezasPosible();
}
const uint32_t Tablero::dameLaPiezaDeArribaDePosicion( uint32_t posicion )
{
  bool noEsPrimeraFila = ( posicion >= this->cantidadDeColumnas );

  if ( noEsPrimeraFila )
  {
    _C( "La pieza arriba de la posicion " << posicion + 1 << " es: " <<
        this->_piezasEnElTablero[posicion - this->cantidadDeColumnas] );
    return this->_piezasEnElTablero[posicion - this->cantidadDeColumnas];
  }
  else
  {
    _C( "La posicion " << posicion + 1 << " contiene una pieza vacia a
       arriba" );
    return TestCaseEj3::PIEZA_VACIA;
  }
}
const uint32_t Tablero::dameLaPiezaDeIzquierdaDePosicion( uint32_t posicion
    )
{
  bool noEsPrimeraColumna = ( posicion % this->cantidadDeFilas != 0 );

  if ( noEsPrimeraColumna )
  {
    _C( "La pieza izquierda de la posicion " << posicion + 1 << " es: " <<
       this->_piezasEnElTablero[posicion - 1] );
    return this->_piezasEnElTablero[posicion - 1];
  }
  else
  {
```

```
    _C( "La posicion " << posicion + 1 << " contiene una pieza vacia a su izquierda" );
    return TestCaseEj3::PIEZA_VACIA;
  }
}
uint32_t Tablero::cantidadDePiezas( void )
{
  return this->_listaDePiezas.size();
}
/*
uint32_t Tablero::cantidadDePosiciones( void )
{
  return this->_cantidadDePosiciones;
}
*/
void Tablero::ponerPiezaEnPosicion( uint32_t pieza, uint32_t posicion )
{
  _C( "Poniendo pieza: " << pieza << " en posicion: " << posicion + 1 );

  if ( this->_piezasEnElTablero[posicion] == TestCaseEj3::PIEZA_VACIA )
  {
    if ( pieza != TestCaseEj3::PIEZA_VACIA )
    {
      this->_cantidadDePosicionesVacias--;
      this->_piezasEnElTablero[posicion] = pieza;

      if ( this->_cantidadDePosicionesVacias < this->_mejorCantidadDePosicionesVacias )
      {
        _C( "Se encontro un nuevo mejor tablero, con " << this->_cantidadDePosicionesVacias << " posiciones vacias, la mejor era " <<
          this->_mejorCantidadDePosicionesVacias << "." );
        this->_mejorCantidadDePosicionesVacias = this->_cantidadDePosicionesVacias;
      }
    }
  }
  else
  {
    if ( pieza == TestCaseEj3::PIEZA_VACIA )
    {
      this->_cantidadDePosicionesVacias++;
    }

    this->_piezasEnElTablero[posicion] = pieza;
  }

#ifdef DEBUG
  this->imprimeTablero();
#else
  /*
  cerr << " ";

  for ( uint32_t x = 0; x < ( this->cantidadDeColumnas * 2 ) + 1; x++ )
  {
    cerr << "-";
  }

  cerr << endl;

  for ( uint32_t y = 0; y < this->cantidadDeFilas; y++ )
  {
```

```
    cerr << "| ";

    for ( uint32_t x = 0; x < this->cantidadDeColumnas; x++ )
    {
      uint32_t posicion = y * this->cantidadDeColumnas + x;
      cerr << Tablero::operator[]( posicion ) << " ";
    }

    cerr << "|";
    cerr << endl;
  }

  cerr << " ";

  for ( uint32_t x = 0; x < ( this->cantidadDeColumnas * 2 ) + 1; x++ )
  {
    cerr << "-";
  }

  cerr << endl;
  */
#endif
  this->_ultimaPosicionAgregada = posicion;
}
/*uint32_t Tablero::mejorTableroHastaElMomento( void )
{
  return this->cantidadDePosiciones - _mejorCantidadDePosicionesVacias;
}*/
bool Tablero::yaEncontreElMejorTableroPosible( void )
{
  return _mejorCantidadDePosicionesVacias <= this->
      _mejorCantidadDePosicionesVaciasPosible;
}
bool Tablero::yaEncontreUnTableroMejor ( uint32_t posicion )
{
  DEBUG_INT( this->_mejorCantidadDePosicionesVacias );
  DEBUG_INT( this->cantidadDePosiciones );
  DEBUG_INT( posicion );
  DEBUG_INT( this->_mejorCantidadDePosicionesVacias + this->
      cantidadDePosiciones - posicion );
  DEBUG_INT( _cantidadDePosicionesVacias );
  return this->_mejorCantidadDePosicionesVacias + this->
      cantidadDePosiciones - posicion
          <= _cantidadDePosicionesVacias;
}
void Tablero::imprimeTablero()
{
  for ( uint32_t y = 0; y < this->cantidadDeFilas; y++ )
  {
    for ( uint32_t x = 0; x < this->cantidadDeColumnas; x++ )
    {
      uint32_t posicion = y * this->cantidadDeColumnas + x;
      cerr << Tablero::operator[]( posicion ) << " ";
    }

    cerr << endl;
  }
}
void Tablero::_calcularMejorCantidadDePiezasPosible()
{
  _mejorCantidadDePosicionesVaciasPosible = 0;
  return;
  multiset<uint32_t> coloresIzquierda;
```

```
  multiset<uint32_t> coloresDerecha;
  multiset<uint32_t> coloresArriba;
  multiset<uint32_t> coloresAbajo;

  for ( uint32_t i = 1; i < this->_listaDePiezas.size(); i++ )
  {
    coloresIzquierda.insert( this->_listaDePiezas[i].colorIzquierda );
    coloresDerecha.insert( this->_listaDePiezas[i].colorDerecha );
    coloresArriba.insert( this->_listaDePiezas[i].colorArriba );
    coloresAbajo.insert( this->_listaDePiezas[i].colorAbajo );
  }

  multiset<uint32_t>::iterator it, it2;

  for ( it = coloresIzquierda.begin(); it != coloresIzquierda.end(); it++ )
  {
    it2 = coloresDerecha.find( *it );

    if ( it2 != coloresDerecha.end() )
    {
      coloresDerecha.erase( it2 );
    }
  }

  for ( it = coloresDerecha.begin(); it != coloresDerecha.end(); it++ )
  {
    it2 = coloresIzquierda.find( *it );

    if ( it2 != coloresIzquierda.end() )
    {
      coloresIzquierda.erase( it2 );
    }
  }

  for ( it = coloresArriba.begin(); it != coloresArriba.end(); it++ )
  {
    it2 = coloresAbajo.find( *it );

    if ( it2 != coloresAbajo.end() )
    {
      coloresAbajo.erase( it2 );
    }
  }

  for ( it = coloresAbajo.begin(); it != coloresAbajo.end(); it++ )
  {
    it2 = coloresArriba.find( *it );

    if ( it2 != coloresArriba.end() )
    {
      coloresArriba.erase( it2 );
    }
  }

  uint32_t piezasInfumables = max( coloresDerecha.size(), max(
      coloresIzquierda.size(), max( coloresArriba.size(), coloresAbajo.size
      () ) ) );
  _C( "Atencion: Hay como minimo " << piezasInfumables << " piezas
      incompatibles en este tablero" );
  this->_mejorCantidadDePosicionesVaciasPosible = piezasInfumables / 2;
  this->_mejorCantidadDePosicionesVaciasPosible = 0;
  _C( "Atencion: Se estima un minimo de " << this->
      _mejorCantidadDePosicionesVaciasPosible << " posiciones en blanco
```

```
        para este tablero." );
    //this->_mejorCantidadDePosicionesVaciasPosible = 0;
}
```

IndiceDePiezas

IndiceDePiezas.h

```
#ifndef __INDICEDEPIEZAS_H__
#define __INDICEDEPIEZAS_H__
#include "ej3.h"
#include "Tablero.h"

class IteradorIndiceDePiezas;
class IndiceDePiezas
{
  friend class IteradorIndiceDePiezas;
private:
  Tablero& _t;
  vector<TestCaseEj3::Pieza>& _listaDePiezas;
  vector<IteradorIndiceDePiezas*> _iteradores;
  typedef vector<uint32_t> listaDePiezas;
  vector< vector< stack< listaDePiezas > > > _indiceDeDosColores;
  vector<bool> _indicePiezasDisponibles;
  stack< listaDePiezas > _indiceSecuencial;
  void _imprimirIndiceDeDosColores();
  //listaDePiezas _indiceSecuencial;
  //IteradorIndiceDePiezas* _it;
  //
public:
  IndiceDePiezas( uint32_t, vector<TestCaseEj3::Pieza>&, Tablero& );
  ~IndiceDePiezas();
  IteradorIndiceDePiezas& dameIterador( uint32_t );
  bool puedeColorarPiezaEnPosicion( uint32_t, uint32_t );
  void marcarPiezaUtilizada( IteradorIndiceDePiezas& );
  void marcarPiezaDisponible( IteradorIndiceDePiezas& );

  //: listaDePiezas(p_listaDePiezas), v(p_cantidadDeColores, vector<
      listaDisponibles>(3, listaDisponibles()) ){

};

#include "IteradorIndiceDePiezas.h"
//#include "IteradorMedio.h"

#endif
```

IndiceDePiezas.cpp

```
#include "IndiceDePiezas.h"

IndiceDePiezas::IndiceDePiezas( uint32_t p_cantidadDeColores, vector<
    TestCaseEj3::Pieza>& p_listaDePiezas, Tablero& t ):
  _t( t ),
  _listaDePiezas( p_listaDePiezas ),
  _indiceDeDosColores(
    p_cantidadDeColores + 1,
    vector< stack< listaDePiezas > >(
      p_cantidadDeColores + 1,
      stack< listaDePiezas >(
        deque<listaDePiezas>( 1, listaDePiezas( 0 ) )
      )
    )
  ),
  _indicePiezasDisponibles(
```

```
    p_listaDePiezas.size(), true
  ),
  _indiceSecuencial(
    deque<listaDePiezas>( 1, listaDePiezas( 0 ) )
  )
{
  for ( uint32_t i = 1; i < this->_listaDePiezas.size(); i++ )
  {
    TestCaseEj3::Pieza pieza = this->_listaDePiezas[i];
    uint32_t colorIzquierda = pieza.colorIzquierda;
    uint32_t colorArriba = pieza.colorArriba;
    _indiceDeDosColores[colorIzquierda][colorArriba].top().push_back( i );
    _indiceSecuencial.top().push_back( i );
  }

  _imprimirIndiceDeDosColores();
}

void IndiceDePiezas::_imprimirIndiceDeDosColores()
{
  for ( uint32_t i = 1; i < _indiceDeDosColores.size(); i++ )
  {
    for ( uint32_t j = 1; j < _indiceDeDosColores.size(); j++ )
    {
      for ( vector<uint32_t>::iterator it = _indiceDeDosColores[i][j].top()
          .begin(); it != _indiceDeDosColores[i][j].top().end(); it++ )
      {
        _C( "IC[" << i << "][" << j << "] = " << *it );
      }
    }
  }
}

IndiceDePiezas::~IndiceDePiezas( )
{
}

IteradorIndiceDePiezas& IndiceDePiezas::dameIterador( uint32_t posicion )
{
  IteradorIndiceDePiezas* it = NULL;
  // Obtengo las piezas de la izquierda y de arriba
  uint32_t piezaIzquierda = _t.dameLaPiezaDeIzquierdaDePosicion( posicion )
      ;
  uint32_t piezaArriba = _t.dameLaPiezaDeArribaDePosicion( posicion );
#ifndef SINPODASELECCION
  bool arribaVacio = ( piezaArriba == TestCaseEj3::PIEZA_VACIA );
  bool izquierdaVacio = ( piezaIzquierda == TestCaseEj3::PIEZA_VACIA );

  if ( arribaVacio || izquierdaVacio )
  {
    _C( "Creando iterador secuencial" );
    it = new IteradorSecuencial( *this, posicion );
  }
  else
  {
    _C( "Creando iterador por colores" );
    it = new IteradorColores( *this, posicion );
  }

#else
  _C( "Creando iterador secuencial" );
  it = new IteradorSecuencial( *this, posicion );
#endif
```

```
  //this->_iteradores.push_back( it );
  return *it;
}
void IndiceDePiezas::marcarPiezaUtilizada( IteradorIndiceDePiezas& it )
{
  _C( "Marcando como utilizada la pieza: " << *it );

  if ( *it == TestCaseEj3::PIEZA_VACIA )
  {
    _C( "La pieza es vacia" );
    it.utilizarPiezaTransparente();
  }
  else
  {
    _imprimirIndiceDeDosColores();
    // Marco la pieza como no disponible
    this->_indicePiezasDisponibles[*it] = false;
    // Obtengo la pieza del listado de piezas
    TestCaseEj3::Pieza pieza = this->_listaDePiezas[*it];
    _C( "Pusheo una copia del indice para esos dos colores" );
    this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].push(
      this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].top()
    );
    _C( "Borro el elemento de la nueva lista" );
    this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].top().erase(
      lower_bound( this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].top().begin(),
                   this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].top().end(), *it )
    );
    _C( "Pusheo una copia del indice secuencial" );
    this->_indiceSecuencial.push(
      this->_indiceSecuencial.top()
    );
    _C( "Borro el elemento del indice secuencial" );
    this->_indiceSecuencial.top().erase(
      lower_bound( this->_indiceSecuencial.top().begin(),
                   this->_indiceSecuencial.top().end(), *it )
    );
    DEBUG_INT( this->_indiceSecuencial.top().size() );
  }
}
void IndiceDePiezas::marcarPiezaDisponible( IteradorIndiceDePiezas& it )
{
  if ( *it != TestCaseEj3::PIEZA_VACIA )
  {
    _C( "Marcando como disponible la pieza: " << *it );
    // Marco la pieza como disponible
    this->_indicePiezasDisponibles[*it] = true;
    // Obtengo la pieza del listado de piezas
    TestCaseEj3::Pieza pieza = this->_listaDePiezas[*it];
    this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].pop();
    this->_indiceSecuencial.pop();
  }
}

bool IndiceDePiezas::puedeColorarPiezaEnPosicion( uint32_t i_pieza, uint32_t posicion )
{
```

```
  TestCaseEj3::Pieza pieza = this->_listaDePiezas[i_pieza];
  uint32_t piezaArriba = this->_t.dameLaPiezaDeArribaDePosicion( posicion );
  bool arriba = piezaArriba == TestCaseEj3::PIEZA_VACIA || pieza.colorArriba == this->_listaDePiezas[piezaArriba].colorAbajo;
  uint32_t piezaIzquierda = this->_t.dameLaPiezaDeIzquierdaDePosicion( posicion );
  bool izquierda = piezaIzquierda == TestCaseEj3::PIEZA_VACIA || pieza.colorIzquierda == this->_listaDePiezas[piezaIzquierda].colorDerecha;
  return arriba && izquierda;
}
```

IteradorIndiceDePiezas

IteradorIndiceDePiezas.h

```
#ifndef __ITERADORINDICEDEPIEZAS_H__
#define __ITERADORINDICEDEPIEZAS_H__
#include "ej3.h"

//class IndiceDePiezas;
#include "IndiceDePiezas.h"
class IteradorIndiceDePiezas
{
private:
  bool _piezaTransparenteUtilizada;
  bool _utilizarPiezaTransparente;
protected:
  vector<bool>& _indicePiezasDisponibles;
  vector<uint32_t>& _indiceSecuencial;
  vector<uint32_t>* _indiceColores;
  IndiceDePiezas::listaDePiezas* _v;
  IndiceDePiezas::listaDePiezas::iterator _v_it;
  IndiceDePiezas& _ip;
  uint32_t _posicion;
public:
  IteradorIndiceDePiezas( IndiceDePiezas&, uint32_t );
  virtual ~IteradorIndiceDePiezas() = 0;
  virtual bool hayPiezasPosibles() = 0;
  virtual IteradorIndiceDePiezas& operator++( int );
  virtual uint32_t operator*();
  void utilizarPiezaTransparente();
};

#include "IteradorSecuencial.h"
#include "IteradorColores.h"
#endif
```

IteradorIndiceDePiezas.cpp

```
#include "IteradorIndiceDePiezas.h"

IteradorIndiceDePiezas::IteradorIndiceDePiezas( IndiceDePiezas& ip,
    uint32_t posicion )
  : _indicePiezasDisponibles( ip._indicePiezasDisponibles ),
    _indiceSecuencial ( ip._indiceSecuencial.top() ),

    _ip( ip ),
    _posicion( posicion )
{
  uint32_t piezaIzquierda = ip._t.dameLaPiezaDeIzquierdaDePosicion(
      posicion );
  uint32_t piezaArriba = ip._t.dameLaPiezaDeArribaDePosicion( posicion );

  if ( piezaIzquierda != TestCaseEj3::PIEZA_VACIA && piezaArriba !=
      TestCaseEj3::PIEZA_VACIA )
  {
    this->_indiceColores = &( ip._indiceDeDosColores
                              [ip._listaDePiezas[piezaIzquierda].
                                  colorDerecha]
                              [ip._listaDePiezas[piezaArriba].colorAbajo]
                              .top() );
  }
```

```

  this->_piezaTransparenteUtilizada = false;
  this->_utilizarPiezaTransparente = false;
  _C( "Inicializado IteradorIndiceDePiezas" );
}

IteradorIndiceDePiezas::~IteradorIndiceDePiezas()
{
}

bool IteradorIndiceDePiezas::hayPiezasPosibles()
{
  return !_piezaTransparenteUtilizada;
}
IteradorIndiceDePiezas& IteradorIndiceDePiezas::operator++( int )
{
  _C( "IteradorIndiceDePiezas::operator++" );
  this->_utilizarPiezaTransparente = true;
  return *this;
}
uint32_t IteradorIndiceDePiezas::operator*()
{
  if ( this->_utilizarPiezaTransparente )
  {
    _C( "Utilizando pieza transparente" );
    return TestCaseEj3::PIEZA_VACIA;
  }
  else
  {
    return *( this->_v_it );
  }
}
void IteradorIndiceDePiezas::utilizarPiezaTransparente()
{
  this->_piezaTransparenteUtilizada = true;
}
```

IteradorSecuencial

IteradorSecuencial.h

```
#ifndef __ITERADORSECUENCIAL_H__
#define __ITERADORSECUENCIAL_H__
#include "IteradorIndiceDePiezas.h"

class IteradorSecuencial : public IteradorIndiceDePiezas
{
public:
  IteradorSecuencial( IndiceDePiezas&, uint32_t );
  ~IteradorSecuencial();
  IteradorSecuencial& operator++( int );
  bool hayPiezasPosibles();
private:
};

#endif
```

IteradorSecuencial.cpp

```
#include "IteradorSecuencial.h"

IteradorSecuencial::IteradorSecuencial( IndiceDePiezas& ip, uint32_t
    posicion )
  : IteradorIndiceDePiezas( ip, posicion )
{
  _v = &( this->_indiceSecuencial );
  _v_it = _v->begin();

  if ( !this->_indicePiezasDisponibles[*( this->_v_it )] ||
       !this->_ip.puedeColorarPiezaEnPosicion( *( this->_v_it ), this->
           _posicion  )
     )
  {
    this->operator++( 0 );
  }
}
IteradorSecuencial::~IteradorSecuencial()
{
}
IteradorSecuencial& IteradorSecuencial::operator++( int )
{
  // Primero, avanzo al menos una pieza en el vector secuencial
  if ( this->_v_it != this->_v->end() )
  {
    _C( "IteradorSecuencial::++" );
    ( this->_v_it )++;
  }

  /*
    Luego avanzo, mientras queden piezas en el vector secuencial
    siempre que las piezas que me encuentre ya esten utilizadas,
    o que esten desocupadas pero no me sirvan...
  */
  while ( this->_v_it != this->_v->end() && (
            !this->_indicePiezasDisponibles[*( this->_v_it )] ||
            !this->_ip.puedeColorarPiezaEnPosicion( *( this->_v_it ), this
                ->_posicion )
          ) )
```

```
  {
    _C( "IteradorSecuencial::++" );
    ( this->_v_it )++;
  }

  // Si ya no hay piezas en el vector secuencial, llamo a mi papa
  if ( this->_v_it == this->_v->end() )
  {
    IteradorIndiceDePiezas::operator++( 0 );
    return *this;
  }

  return *this;
}
bool IteradorSecuencial::hayPiezasPosibles( )
{
  bool hayPiezasPosibles = this->_v_it != this->_v->end();
  return hayPiezasPosibles || IteradorIndiceDePiezas::hayPiezasPosibles();
}
```

IteradorColores

IteradorColores.h

```
#ifndef __ITERADORCOLORES_H__
#define __ITERADORCOLORES_H__
#include "IteradorIndiceDePiezas.h"

class IteradorColores : public IteradorIndiceDePiezas
{
public:
  IteradorColores( IndiceDePiezas&, uint32_t );
  ~IteradorColores();
  IteradorColores& operator++( int );
  bool hayPiezasPosibles();
private:
};

#endif
```

IteradorColores.cpp

```
#include "IteradorColores.h"

IteradorColores::IteradorColores( IndiceDePiezas& ip, uint32_t posicion )
  : IteradorIndiceDePiezas( ip, posicion )
{
  _C( "Inicializando IteradorColores." );
  _v = this->_indiceColores;
  _v_it = _v->begin();
  _C( "El tamanio del iterador de colores es: " << _v->size() );

  if ( _v->begin() == _v->end() )
  {
    this->operator++( 0 );
  }
}
IteradorColores::~IteradorColores()
{
}
IteradorColores& IteradorColores::operator++( int )
{
  // Primero, avanzo al menos una pieza en el vector
  if ( this->_v_it != this->_v->end() )
  {
    _C( "IteradorColores::++" );
    ( this->_v_it )++;
  }

  if ( this->_v_it == this->_v->end() )
  {
    IteradorIndiceDePiezas::operator++( 0 );
  }

  return *this;
}
bool IteradorColores::hayPiezasPosibles( )
{
  bool hayPiezasPosibles = this->_v_it != this->_v->end();
  return hayPiezasPosibles || IteradorIndiceDePiezas::hayPiezasPosibles();
}
```

4.2. Informe de Modificaciones

Se realizaron las siguientes modificaciones:

- Se cambiaron los nombres de las secciones "Hipótesis de Resolución" por "Planteamiento de Resolución"; el contenido sigue siendo el mismo.

`Sección 1:`**`Introducción`**` (página 3):`

- Correcciones y aclaraciones mínimas en el texto.
- Se creó `Sección 2:`**`Instrucciones de uso`**` (página 5)` en donde se aclaran detalles relativos al código y/o scripts provistos en el paquete de entrega, y se movió el apartado de "Herramientas Utilizadas" a `Subsección 2.1:`**`Herramientas utilizadas`**` (página 5)`.

`Subsección 3.3:`**`Problema 3: Rompecolores`**` (página 22):`

- Se realizaron las correcciones marcadas por el profesor en **`3.3.1:Descripción`** (página 8).
- Para la primer parte[19] del contenido que figuraba en la sección "Hipótesis de Resolución", se agregaron aclaraciones a algunas de las correciones hechas, se eliminó/reformuló por completo la abstracción formal matemática[20]. Además, se trasladó todo el contenido de estos párrafos a **`3.3.1:Descripción`** (página 8), ya que consideramos que el análisis realizado en esos párrafos es más propio del modelado/descripción del problema; es decir que es independiente de la resolución que hayamos ideado.
- El texto que figuraba en la segunda parte del contenido de la sección "Hipótesis de Resolución" se eliminó/reformuló por completo, haciendo un análisis y descripción más detallados de los objetivos del algoritmo y las podas implementadas. No se hizo demasiado hincapié en los detalles estrictamente implementativos, sino en las nociones o ideas generales que inspiraron el código.
- El código del problema 3 fue reescrito por completo, utilizando una versión con clases, permitiendo así realizar diversas abstracciones que permitieron (a nuestro gusto) una mejor organización y comprensión del algoritmo. Por ejemplo, se implementó la clase IndiceDePiezas, la cual contiene todo el comportamiento relativo al índice con el que se eligen las piezas que serán utilizadas, y la clase abstracta IteradorIndiceDePiezas, cuyas clases heredadas (IteradorSecuencial, IteradorColores) son instanciadas por el Índice de Piezas, y contienen un comportamiento común a ambas, y un comportamiento propio que depende de la posición del tablero en que se encuentre el iterador.

[19] Consta del análisis del tipo de problema brindado, del universo de soluciones posible, y la formalización de la función objetivo.

[20] A causa de la notación utilizada, que era incorrecta, y de que llegamos a la conclusión de que no era necesario ni deseable ese nivel de "formalidad" en la notación, cuando las mismas ideas pueden bien ser explicadas, sin perder su formalidad, en lenguaje coloquial.

4.3. Ejercicios Adicionales (reentrega)

Problema 3:

1. ¿Cómo debería modificarse el algoritmo implementado para que el mismo funcione en el RompeColores 2.0?

No se debería modificar de manera sustancial. Ahora hay que chequear que al poner una ficha en la última columna, su color derecho concuerde con el color izquierdo de la primera ficha de la fila. Además, al poner una ficha en la última fila, su color inferior debe concordar con el color superior de la primera ficha de la columna. Nuestro algoritmo determinaba las fichas posibles para éstas posiciones solamente de acuerdo a su color superior e izquierdo. Ahora, antes de colocar la ficha, hay que satisfacer unas condiciones extra. Podemos iterar sobre la lista de piezas que ibamos a colocar ahi en Rompecolores1.0, y preguntar si satisfacen o no la condición.

2. ¿Cómo afecta el nuevo esquema a las podas implementadas en la primera versión?

Con respecto a nuestra optimización de utilizar PiezasPorColores, se podrían mantener otras estructuras que de acuerdo a 3 colores "izquierda, arriba, derecha", o "izquierda, arriba, abajo", nos devuelva todas las piezas que tienen esos colores en tales bordes. De esta forma podemos saber rápidamente que fichas se pueden poner cuando estamos en una posición de la última columna o última fila. Implicaría un mayor costo temporal y espacial, pero asintóticamente seriá el mismo costo que la versión anterior. La poda de cortar la rama cuando, aún llenando todas las posiciones que me quedan por recorrer, no llegamos a superar al mejor cubrimiento de tablero que encontramos, seguría exactamente igual. Lo mismo ocurre con la poda de terminar la ejecución cuando ya encontramos una solución que cubre todo el tablero.

3. ¿Pueden proponer nuevas podas para este nuevo esquema que no podrían implementarse en el esquema anterior?

Hay ciertas podas que resultan mucho más intuitivas con el nuevo esquema. Intentemos acotar la cantidad de casillas que cubre la solución óptima, y de esta forma parar la ejecución cuando encontramos una solución que llega a la cota. Por ejemplo, para cada ficha que tiene el color azul arriba, si queremos colocarla en el tablero debería existir una ficha con el color azul abajo. La idea es contar, para cada color x de 1 a c, la diferencia entre la cantidad de fichas que tienen arriba x y la cantidad de fichas que tienen abajo x, lo mismo con la diferencia de fichas que lo tienen a la izquierda y a la derecha. Si tomamos "p" la mayor de estas diferencias entre todos los colores, sabemos que hay por lo menos p fichas que no vamos a poder colocar. Luego si llegamos a una solución que coloque n - p fichas, ya podemos parar la ejecución.