TRABAJO FIN DE GRADO

Doble Grado en Ingeniería Informática y Matemáticas

Entendiendo la teoría de nudos mediante la simulación y la informática gráfica.

Autora
Cristina Zuheros Montes.

Tutores
Alejandro J. León Salas
Antonio Martínez López

Granada, Diciembre de 2016

Entendiendo la teoría de nudos mediante la simulación y la informática gráfica.

Cristina Zuheros Montes.

Palabras clave: nudo, trenza, handle...PONER LAS PALABRAS CLAVE.

Resumen

//HACER RESUMEN AQUI!

PONER TITULO EN INGLES!!!

Cristina Zuheros Montes.

Keywords: knot, braid, handle....PONER AQUI KEYWORDS!!!!!

Abstract

//PONER RESUMEN DE MILLON DE PALABRAS INGLES.!!!!!!

Índice general

Índice de figuras

Capítulo 1

Introducción

HACER INTRO AQUIII!!!

1.1 Objetivos

En la propuesta inicial se propusieron los siguientes objetivos:

1. Estudiar...//PONER LOS OBJETIVOS INICIALES

   //PONER DONDE SE CUBREN LOS OBJETIVOS

Capítulo 2

Teoría de nudos.

Capítulo 3

Teoría de trenzas.

Capítulo 4

Desarrollo informático.

4.1 Pseudoalgoritmos.

En esta sección vamos a ver algunos pseudoalgoritmos referentes a los algoritmos que hemos implementado. Nos vamos a centrar en aquellos que tienen mayor complejidad e interés, sin entrar en gran detalle. Es importante destacar que estos algoritmos no están establecidos como tales, sino que los hemos implementando intentando reflejar los conceptos matemáticos que hemos ido viendo.

En concreto, vamos a ver los pseudoalgoritmos para el algoritmo de Dehornoy (que descomponemos en varios algoritmos auxiliares), el algoritmo que nos permite comprobar si dos trenzas dadas (o sus cierres) son o no equivalentes entre sí y el algoritmo que nos permite comprobar si una trenza dada (o su cierre) es equivalente a la trenza trivial.

Algoritmo Dehornoy para trenzas.

Vamos a ir viendo los algoritmos auxiliares y concluiremos con el propio algoritmo de Dehornoy. Recordemos que este algoritmo se explica con detalle en la sección **??**.

Creamos el método Simplifica mediante el cuál eliminamos ocurrencias de tipo $\sigma_i^e \sigma_i^{-e}, e \in$ {-1,1} de una palabra que representa a cierta trenza.

Algoritmo 4.1.1. ***Algoritmo Simplifica****(indices_braid)*
ENTRADA: indices_braid (cadena de enteros que representa los cruces de una trenza)
SALIDA: braid_aux (cadena auxiliar para mejor representacion visual)
nueva_braid (cadena de enteros que representa los cruces de la trenza tras eliminar dos cruces consecuetivos opuestos)
encontrado (bool para indicar si se produce simplificación)

```
1    Recorro los cruces de indices_braid
2    Si hay dos cruces seguidos con signos opuestos, creo una copia de
         indices_braid y elimino dichos cruces.
```

Haciendo uso del siguiente algoritmo podremos encontrar las posiciones que delimitan un σ_{minimo}-handle.

Algoritmo 4.1.2. ***Algoritmo encuentra_handle****(indices_braid, minimo)*
ENTRADA: indices_braid (cadena de enteros que representa los cruces de una trenza)
minimo (generador de handle a encontrar)
SALIDA: pos1 (posición inicial del handle encontrado)
pos2 (posición final del handle encontrado)

```
1   Recorro los cruces de indices_braid
2   Si hay un cruce generado por el elemento minimo, me quedo con esa
      posicion como pos1 y su signo. Sino finalizo.
3   Si encuentro un cruce generado por el elemento minimo con signo
      opuesto, me quedo con esa posicion como pos2.
```

Haciendo uso del algoritmo reduccion_base aplicamos una reducción local a la trenza representada por indices_braid sobre el σ_{minimo}-handle que está situado entre las posiciones pos1 y pos2. Es importante destacar el hecho de que este σ_{minimo}-handle no contiene $\sigma_{minimo+1}$-handle. Este detalle lo controlamos en el algoritmo Dehornoy que veremos posteriormente.

Algoritmo 4.1.3. ***Algoritmo reduccion_base****(indices_braid, minimo, pos1, pos2)*
ENTRADA: indices_braid (cadena de enteros que representa los cruces de una trenza)
minimo (generador de handle)
pos1 (posición inicial del handle)
pos2 (posición final del handle)
SALIDA: braid_aux2 (cadena auxiliar para mejor representación visual)
nuevo (cadena de enteros que representa los cruces de la trenza tras aplicar la reducción local al σ_{minimo}-handle entre pos1 y pos2)
simplificado2 (bool auxiliar para mejor representación visual)

```
1   Creo vector_auxiliar
2   Recorro los cruces de indices_braid desde pos1 hasta pos2
3   Si hay un cruce generado por el elemento (minimo+1) aniado al
      vector_auxiliar los 3 cruces correspondientes. Sino, aniado el
      mismo cruce.
4   Creo vector_nuevo con los elementos desde inicio de indices_braid
      hasta pos1, vector_auxiliar, y los elementos desde pos2 hasta
      final de indices_braid.
5   Si indices_braid y vector_auxiliar tienen distinto tamanio, asigno
      a braid_aux2 una cadena con ciertos ceros para mejor
      visualizacion.
```

Estos algoritmos auxiliares no se proporcionan para uso directo al usuario ya que el realmente interesante es el algoritmo de Dehornoy, pero sí podrían ser usados. Veamos entonces el algoritmo de Dehornoy.

Algoritmo 4.1.4. ***Algoritmo dehornoy****(br, N_cortes, Radio, representar)*
ENTRADA: br (trenza)
N_cortes (numero de cortes de las cadenas de la trenza)
Radio (radio de las cadenas de la trenza)
representar (bool para representar las equivalencias de la trenza)
SALIDA: es_trivial (bool que nos indica si la trenza dada es o no trivial)
trenza_final (cadena de enteros que representa a la trenza reducida equivalente a br)

```
1   Si numero_argumentos=1 -> N_cortes=20, Radio=0.5, representar=1.
2   indices_braid = cadena de enteros que representa a la trenza
3   Si br tiene cadenas a la derecha triviales, las eliminamos
       visualmente.
4   Mientras queden handles en la trenza dada...
5         Obtenemos la palabra libremente reducida de indices_braid.
6         Si no se produce reduccion...
7           minimo = generador principal de indices_braid.
8           [pos1,pos2]=encuentra_handle(indices_braid,minimo)
9           Si pos1 y pos2 son posiciones validas....
10                  Busco primer subhandle a realizar en el handle.
11                  Actualizo pos1 y pos2
12                  Aplico reduccion_base a dicho subhandle.
13        Creo matriz con secuencia de palabras generadas en el proceso
14  Si representar, muestro las trenzas usando dicha matriz.
```

Finalmente cabe comentar que el algoritmo de Dehornoy se podrá aplicar sobre trenzas cerradas. El proceso que se realizará internamente será exactamente el mismo que para trenzas, ya que el cerrar la trenza no aporta información nueva para el algoritmo de Dehornoy. Por este mismo motivo, a la hora de realizar la visualización del algoritmo de Dehornoy sobre una trenza cerrada, veremos las transformaciones sobre la trenza y no sobre la trenza cerrada.

Algoritmo de equivalencia para trenzas.

Para ver si dos trenzas dadas son equivalentes o no entre sí, vamos a implementar algunos de los invariantes que vimos para trenzas. En concreto, vamos a estudiar los invariantes exponente y permutación. Si con estos invariantes no conseguimos analizar la equivalencia de las trenzas, vamos a aplicar el algoritmo de Dehornoy que hemos visto anteriormente.

Algoritmo 4.1.5. ***Algoritmo equivalentes****(br1,br2,explicacion)*
ENTRADA: br1 (trenza1)
br2 (trenza2)
explicacion (bool para mostrar mensajes explicativos)
SALIDA: equi (bool para indicar si br1 y br2 son o no equivalentes)

```
1    Si numero_argumentos=2 -> explicacion=0.
2    Obtengo exponente de ambas trenzas.
3    Si son distintos -> No son equivalentes. Finalizo
4    Obtengo permutacion de ambas trenzas.
5    Si son distintas -> No son equivalentes. Finalizo
6    Genero trenza_auxiliar = br1inver(br2).
7    Aplico Dehornoy a trenza_auxiliar y obtengo final_braid.
8    Si final_braid no es vacia -> No son equivalentes.
9    Sino -> Si explicacion=1 -> represento secuencia de trenzas de
        dehornoy.
```

Algoritmo de equivalencia para trenzas cerradas.

Para analizar la equivalencia de dos trenzas cerradas, vamos a apoyarnos en el algoritmo de equivalencia de las trenzas sin cerrar que hemos visto anteriormente. Además haremos uso de varios algoritmos auxiliares. En concreto, hemos implementado los movimientos de Markov que vimos en la sección **??**.

Sabemos que el movimiento 1 de Markov puede eliminar o añadir cruces. Siempre intentaremos eliminar cruces, mientras que para añadir cruces se tiene que solicitar explícitamente. Lo hacemos así para que las trenzas no aumenten en número de cruces salvo que no quede más opción.

Algoritmo 4.1.6. ***Algoritmo MV1****(br_c, completo)*
ENTRADA: br_c (trenza cerrada)
completo (bool para indicar si se desean aniadir cruces)
SALIDA: a2 (trenza cerrada tras aplicar Movimiento1 eliminando de Markov.)

```
1    Si numero_argumentos=1 -> completo=0.
2    Si el primer cruce de br_c es opuesto al ultimo cruce de br_c -> a2
        = br_c sin dichos cruces.
3    Sino -> Si completo...
4      Obtengo cruce de mayor indice (indice m).
5      Aniado cruces opuestos de indice m a principio y final de br_c.
```

Algoritmo 4.1.7. ***Algoritmo MV2****(br_c)*
ENTRADA: br_c (trenza cerrada)
SALIDA: a3 (trenza cerrada tras aplicar Movimiento2 de Markov.)

```
1    Mientras br_c cambie de cruces...
2       Si br_c tiene un solo cruce -> a3=trenza cerrada trivial.
           Finalizo.
3       Obtengo cruce de mayor indice (indice m).
4       Si se encuentra solo una vez en br_c
5          Si a izquerda o derecha del cruce no tenemos cruces con
              indice m-1 -> a3=br_c sin dicho cruce.
```

Para implementar este algoritmo de equivalencia entre trenzas cerradas hemos analizado en primer lugar los invariantes básicos de ambas trenzas. Si no obtenemos una respuesta de equivalencia o no equivalencia, pasamos a ver el polinomio de Alexander de ambas trenzas cerradas. Una vez analizados los invariantes que hemos estudiado, vamos a ir realizando transformaciones de equivalencia sobre la trenza cerrada generada por el producto de la trenza inicial y la inversa de la segunda trenza.

Ya sabemos que ir haciendo transformaciones de equivalencia sobre una trenza cerrada para ver si es o no equivalente a la trenza cerrada trivial puede conllevar a una serie indefinida de movimientos.

La idea que hemos llevado a cabo consiste en aplicar el Movimiento 1 de Markov, el algoritmo de Dehornoy a la trenza cerrada que obtenemos, el Movimiento 2 de Markov y de nuevo el algoritmo de Dehornoy. Si en alguna de estas transformaciones conseguimos saber si la trenza cerrada es o no equivalente a la trivial, finalizamos el proceso. En caso contrario, realizaremos el mismo proceso un número delimitado de veces.

Algoritmo 4.1.8. ***Algoritmo equivalentes****(br1,br2,explicacion)*
ENTRADA: br1 (trenza cerrada 1)
br2 (trenza cerrada 2)
explicacion (bool para mostrar mensajes explicativos)
SALIDA: equi (bool para indicar si br1 y br2 son o no equivalentes)

```
1   Si numero_argumentos=2 -> explicacion=0.
2   Si el numero de enlaces de br1 y de br2 son distintos -> No son
       equivalentes. Finalizo
3   Si equivalentes@trenza(br1,br2) son equivalentes -> sus cierres son
       equivalentes.
4   Sino ->
5        Obengo polinomio de Alexander de ambas trenzas.
6        Si son iguales salvo signo -> No sabemos si son o no
            equivalentes. equi <- 2.
7        Sino -> No son equivalentes.
8   Si equi=2
9    Creo trenza cerrada br = br1inver(br2)
10   Mientras numero_iteraciones < limite(establecido a 3)
11            a1 = Aplico MV1 a br.
12            [es_trivial2,a2] = Aplico dehornoy a trenza cerrada a1.
13            Si es_trivial2 -> Finalizo.
14            a3 = Aplico MV2 a trenza cerrada a2.
15            [es_trivial4,a4] = Aplico dehornoy a trenza cerrada a3.
16            Si es_trivial4 -> Finalizo.
17            Si no ha cambiado br, a4 = Aplico MV1 completo a br.
```

Algoritmo para ver trivialidad de trenza.

Tanto para ver la trivialidad de una trenza como la trivialidad de una trenza cerrada vamos a usar la misma idea que será hacer uso de los algoritmos de equivalencia entre la trenza dada y la trenza trivial. Veámoslos:

Algoritmo 4.1.9. ***Algoritmo es_trivial****(br,explicacion)*
ENTRADA: br (trenza)
explicacion (bool para mostrar mensajes explicativos)
SALIDA: equi (bool para indicar si br es o no equivalente a la trenza trivial)

```
1    Si numero_argumentos=1 -> explicacion=0.
2    Creo br2 = trenza trivial.
3    Aplico algoritmo equivalentes para br y br2.
```

Algoritmo para ver trivialidad de trenza cerrada.

Algoritmo 4.1.10. ***Algoritmo es_trivial****(br_c,explicacion)*
ENTRADA: br_c (trenza cerrada)
explicacion (bool para mostrar mensajes explicativos)
SALIDA: equi (bool para indicar si br_c es o no equivalente a la trenza cerrada trivial)

```
1    Si numero_argumentos=1 -> explicacion=0.
2    Creo br2_c = trenza cerrada trivial.
3    Aplico algoritmo equivalentes para br_c y br2_c.
```