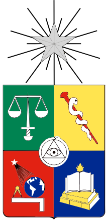

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA MATEMÁTICA

REDES DE AUTÓMATAS Y COMPLEJIDAD COMPUTACIONAL

TESIS PARA OPTAR AL TÍTULO DE INGENIERO CIVIL MATEMÁTICO

PEDRO TOMÁS MONTEALEGRE BARBA

PROFESOR GUÍA:
ERIC GOLES CHACC

MIEMBROS DE LA COMISIÓN:
IVÁN RAPAPORT ZIMERMANN
MARTÍN MATAMALA VÁSQUEZ

SANTIAGO DE CHILE
NOVIEMBRE 2012

Resumen

El presente trabajo consiste en el estudio de la complejidad computacional en algunas redes de autómatas. En particular en el problema de decisión, que llamamos **PER**, el cual consiste en predecir cambios de estado en un nodo determinado cuando la red se actualiza según una regla determinada.

Dentro de los primeros en introducir complejidad computacional a los autómatas celulares (CA) y sistemas relacionados podemos destacar a C. Moore [19] quien estudió la regla de la mayoría estricta en lattices d-dimensionales, y a T. Neary con D.Woods [23], que prueban la **P**-Complitud de la regla 110 en autómatas unidimensionales. Estos estudios son interesantes porque, por un lado, usualmente es muy difícil obtener caracterizaciones del comportamiento general de la dinámica de un autómata celular en el tiempo; y por otro lado, están relacionados con el procesamiento paralelo de la información y algoritmos, ya que algunos CA son capaces de emular una máquina de Turing universal. Luego, la idea es construir un puente entre estos dos aspectos del problema: la naturaleza de su dinámica y sus capacidades algorítmicas.

Por lo general estos trabajos se enfocan en acotar el problema "por arriba", determinando **P**-Complitud. Este trabajo es novedoso en el sentido que a lo anterior agregamos un acotamiento "por abajo", buscando demostrar que cuando el problema no es **P**-Completo, entonces debe ser eficientemente paralelizable, es decir, pertenecer a la clase **NC**. Esto lo haremos variando primero la topología de la red considerando siempre un modo de iterar paralelo, y posteriormente determinando la influencia de distintos modos de iterar en la complejidad.

En términos más concretos, estudiaremos la complejidad computacional de Bootstrap Percolation (Capítulo 2). El resultado principal es que, para esta regla, el problema de decisión **PER** está en **NC** si restringimos al grafo que define a la red a pertenecer a la familia que tiene grado máximo pequeño, y en caso contrario el problema es **P**-Completo. Luego, en el Capítulo 3, cambiaremos a la regla de la mayoría estricta, donde el principal resultado del capítulo será que para esta regla **PER** es **P**-Completo en la familia de grafos planares. Finalmente, en el cuarto capítulo estudiaremos cómo varían los resultados anteriores cuando consideramos distintos modos de actualizar los estados de la red. El resultado principal tendrá relación con los distintos modos de iterar considerando en cada nodo una función booleana AND u OR. Por último, daremos algunas conclusiones y problemas abiertos.

El trabajo aquí expuesto ha permitido una publicación en *Theoretical Computer Science* llamado *The complexity of Bootstrapping Percolation* [10], una charla invitada (Winter FRAC 2012, París), una conferencia invitada (CA2012 Córcega), un artículo actualmente enviado a *Advances in Applied Mathematics* y otro en desarrollo.

Dedicada a la memoria
de Juan Manuel Barba López

Agradecimientos

Agradezco a mi profesor guía Eric Goles, primero por su entusiasmo y compromiso con mi trabajo. También por su paciencia y preocupación no solo por que mi memoria salga a tiempo, si no que también porque de ella salgan publicaciones. Además le quiero agradecer porque su compromiso no termina con la memoria, ayudándome con las postulaciones al doctorado.

Agradezco a la Universidad de Chile, a la Facultad de Ciencias Físicas y Matemáticas, por su formación de excelencia y rigor. A mis profesores, en especial a los profesores Iván Rapaport y Martín Matamala, por sus comentarios e interés en mi memoria. Así mismo quiero agradecer también a la Universidad Adolfo Ibáñez, que me ha recibido muy amablemente todo este período. En especial a los profesores Sergio Rica, Gonzalo Ruz y Marco Montalva. Quiero agradecer también al profesor Ioan Todinca y al Laboratoire d'Informatique Fondamentale d'Orléans (LIFO), de la Université d'Orléans, donde desarrollé gran parte de mi memoria.

Quiero agradecer a mis amigos y compañeros de generación, en especial a Osmar, Sandra, Felipe, Sebastián, Gonzalo, Emilio, Valentina, César, Rodrigo y Johan. Siempre agradezco el cariño de mi familia, de mis padres, de Pablo, Teresa y Sofía.

Por último le agradezco a Alejandra, que ya sabe que la vida es inherentemente secuencial.

Esta tesis ha sido parcialmente financiada gracias a los proyectos Fondecyt 1100003 (E6), Anillo ACT-88, y el Laboratorio de Informática Fundamental de la Universidad Orléans, Francia.

Índice general

Índice de figuras **x**

Introducción **1**

1. Marco Conceptual **3**

1.1. Elementos de teoría de grafos 3
1.1.1. Conexidad y ciclos 4
1.1.2. Árboles, bipartitos y planos 5
1.2. Elementos de Complejidad computacional 5
1.2.1. Modelos de Computación 5
1.2.2. Clases de complejidad 8
1.2.3. Ejemplo de problema en NC (FNC): Sumas de prefijos 12
1.3. Redes de Autómatas 13
1.3.1. Autómatas Celulares 14
1.3.2. Redes Neuronales 15
1.4. Problemas de decisión y redes de autómatas 15

2. Bootstrap Percolation **17**

2.1. La complejidad de Bootstrap Percolation 18
2.1.1. Caso **P**-Completo 18
2.1.2. Caso $\Delta(G) \leq 4$ 21
2.2. Problemas relacionados 25
2.2.1. Bootstrap Percolation no estricto 26
2.2.2. Umbral θ 27

3. Mayoría estricta: caso plano **29**

3.1. Mayoría estricta está en P 30
3.2. La mayoría estricta es P-Completa 32
3.3. Caso plano 34

4. Complejidad y modos de actualizar **40**

4.1. Problema de decisión 40
4.2. Bootstrap Percolation 41
4.3. Regla de la mayoría 42
4.4. Redes AND-OR 43

Bibliografía **55**

A. Análisis de la Complejidad del Algoritmo 3 **56**

B. Análisis de la Complejidad del Algoritmo 4 **59**

C. Análisis de la Complejidad del Algoritmo 5 **63**

D. Artículos **67**
D.1. Artículo Publicado en *Theoretical Computer Science* 67
D.2. Artículo enviado a *Advances in Applied Mathematics* 87

Índice de figuras

1.1. Componentes biconexas de un grafo. Arcos con la misma etiqueta están en la misma componente. Notar que un mismo vértice puede pertenecer a varias componentes biconexas. 5
1.2. Arriba: un AC unidimensional con $N(0) = \{-1, 0, 1\}$, Abajo: El caso con $d = 2$ con vecindades del tipo Moore (izquierda) y von Neumann (derecha) 14

2.1. a) La puerta AND. b) La puerta OR. 19
2.2. a) El diodo. b) Símbolo para representar un diodo. 19
2.3. Puertas AND y OR con diodos. 20
2.4. Puerta OR usada “al revés” para multiplicar la información de un output. . 20
2.5. Ejemplo de una entrada del algoritmo para decidir **PER**. Decidimos sobre el vértice en gris v 22
2.6. Los vértices en gris representan a la componente conexa que contiene al vértice sobre el que decidimos, para el ejemplo de la Figura 2.5 23
2.7. Los vértices que cumplen la propiedad (1) o (2) del Lema 2.1.2 en el ejemplo. Conectamos dichos vértices a un nuevo vértice que llamamos ∞. 24
2.8. En el grafo obtenido al agregar el ∞, nuevamente buscamos componentes biconexas y decidimos si hay alguna que contenga a ∞ y v. 24
2.9. De izquierda a derecha: El diodo en el caso no estricto, las puertas AND y OR para la misma regla. 27
2.10. Representamos el grafo completo K, en este caso K_5, y las $|K|$ conexiones con un vértice rectangular con el número $|K|$ adentro y un solo arco. 28
2.11. De izquierda a derecha: Diodo para la regla Umbral θ, las puertas AND y OR para la misma regla. 28

3.1. a) Un diodo. b) Representación simplificada de un diodo. 32
3.2. a) La puerta AND. b) La puerta OR. 33
3.3. Puerta OR “al revés” para multiplicar la información. 33
3.4. Un cable de largo 3. 33
3.5. Un ejemplo de un circuito y la posición de las puertas en la incrustación plana. Notemos que la coordenada y es aún desconocida para las puertas que no están en la capa de entrada, pero tiene el mismo valor para todas las puertas de una misma capa. Obviamente si aquí simplemente dibujamos los arcos, no obtendremos necesariamente una incrustación plana. 35

3.6. Izquierda: Los dos primeros layers del circuito en el ejemplo de la Figura 3.5. Derecha: Esquema de las posiciones de los vértices que simulan el layer input y los dos outputs correspondientes. 35
3.7. Gadget para el 'Cruce de cables' usando 'semáforos' 36
3.8. Los pasos necesarios para que el gadget de cruce de información transmita desde **a** hacia **e** sin contaminar **d**. Recordar que suponemos que antes de los inputs y después de los outputs irán diodos. 37
3.9. Esquema de la etapa 2: el nodo cuadrado representa que ahí va un gadget de cruce de información. Los arcos con extremos con vértices con coordenada y (1) y (2), o (2) y (3), representan cables de largo 10. 38
3.10. Esquema de la incrustación plata que simula las dos primeras capas del circuito en la Figura 3.6. El nodo cuadrado representa el gadget para cruzar cables. Los arcos con extremos con coordenada y diferente y entre (1) y (8) representan cables de largo 10. 39

4.1. Un diodo 44
4.2. Puerta AND 45
4.3. Puerta OR 45

Introducción

El presente trabajo consiste en el estudio de la complejidad computacional en algunas redes de autómatas. En particular en el problema de decisión, que llamamos **PER**, el cual consiste en predecir cambios de estado en un nodo determinado cuando la red se actualiza según una regla determinada.

Dentro de los primeros en introducir complejidad computacional a los autómatas celulares (CA) y sistemas relacionados podemos destacar a C. Moore [19, 22, 21, 20] quien estudió la regla de la mayoría estricta en lattices d-dimensionales, y a T. Neary con D.Woods [23], que prueban la **P**-Complitud de la regla 110 en autómatas unidimensionales. Estos estudios son interesantes porque, por un lado, usualmente es muy difícil obtener caracterizaciones del comportamiento general de la dinámica de un autómata celular en el tiempo; y por otro lado, están relacionados con el procesamiento paralelo de la información y algoritmos, ya que algunos CA son capaces de emular una máquina de Turing universal. Luego, la idea es construir un puente entre estos dos aspectos del problema: la naturaleza de su dinámica y sus capacidades algorítmicas.

Por lo general estos trabajos se enfocan en acotar el problema "por arriba", determinado **P**-Complitud. Este trabajo es novedoso en el sentido que a lo anterior agregamos un acotamiento "por abajo", buscando demostrar que cuando el problema no es **P**-Completo, entonces debe ser eficientemente paralelizable, es decir, pertenecer a la clase **NC**. Esto lo haremos variando primero la topología de la red considerando siempre un modo de iterar paralelo, y posteriormente determinando la influencia de distintos modos de iterar en la complejidad.

Una de las reglas que estudiaremos será Bootstrap Percolation, la cual consiste en vértices con estados en $\{0, 1\}$ (pasivos y activos, respectivamente), y donde los vértices activos se mantienen siempre activos, y un vértice pasivo pasa a activo si la mayoría estricta de sus vecinos es activa. Los estudios teóricos clásicos relacionados a Bootstrap Percolation trabajan con la pregunta de cual es la mínima cantidad de sitios activos (infectados) de modo que se infecte, digamos con alta probabilidad, toda la estructura. Los resultados en este sentido son complicados y usualmente restringidos a familias específicas de grafos (lattices, grafos cúbicos, etc) [24, 3]. Nuestro trabajo es distinto y complementario al anterior, ya que nos preguntamos por la posibilidad de que un vértice en específico vaya a ser infectado dada una configuración inicial, y si ello se puede producir rápida o eficientemente.

A continuación estudiaremos la regla de la mayoría estricta, que quiere decir que vértices activos y pasivos toman el estado de la mayoría estricta de sus vecinos, pero en el caso en que el grafo que define la red es planar. Finalmente vemos como afectan distintos modos de iterar para estas reglas y otras.

Este trabajo aborda una conjetura que plantea C. Moore en [19], la cual dice que en el lattice 2-dimensional la regla de la mayoría estricta está en **NC**. Demostramos por un lado que si variamos la regla a Bootstrap Percolation la conjetura es cierta, pero si mantenemos la regla y exigimos que el grafo sea solo planar tenemos que el problema es **P**-Completo.

La estructura de los capítulos es la siguiente: En el primer capítulo daremos un marco conceptual necesario para comprender de mejor manera los capítulos siguientes. Al final del primer capítulo formularemos el problema de decisión **PER** con el que mediremos la complejidad computacional de redes de autómatas. En el segundo capítulo estudiaremos la complejidad computacional de Bootstrap Percolation, donde el resultado principal del capítulo será que para esta regla el problema **PER** está en **NC** si restringimos al grafo que define a la red a pertenecer a la familia que tiene grado máximo pequeño, y en caso contrario el problema es **P**-Completo. En el tercer capítulo cambiaremos a la regla de la mayoría estricta, donde el principal resultado del capítulo será que para esta regla **PER** es **P**-Completo en la familia de grafos planares. Finalmente, en el cuarto capítulo estudiaremos como varían los resultados anteriores cuando consideramos distintos modos de actualizar la red, donde el resultado principal del capítulo tendrá relación con los distintos modos de iterar considerando en cada nodo una función booleana AND u OR. Por último daremos algunas conclusiones y problemas abiertos o futuros trabajos.

El trabajo aquí expuesto ha permitido una publicación en *Theoretical Computer Science* llamado *The complexity of Bootstrapping Percolation* [10], una charla invitada (Winter FRAC 2012, París), una conferencia invitada (CA2012 Córcega), un artículo actualmente enviado a *Advances in Applied Mathematics* y otro en desarrollo.

Capítulo 1

Marco Conceptual

En este capítulo introduciremos los conceptos, ejemplos y definiciones necesarios para comprender de mejor manera los resultados y las demostraciones que vendrán en los capítulos siguientes, a saber: Elementos de teoría de grafos, elementos de complejidad computacional destacando las clases que estudiaremos en este trabajo, y la definición formal de una red de autómatas. Finalmente discutiremos de qué forma entenderemos la complejidad computacional en el marco de redes de autómatas, planteando un problema de decisión adecuado.

1.1. Elementos de teoría de grafos

Un *grafo* es un par $G = (V, E)$ de conjuntos tal que $E \subset V \times V$. Los elementos de V diremos que son los *vértices* o *nodos* del grafo G, y los elementos de E son los *arcos* o *aristas*. Un grafo con conjunto de vértices V se dirá que es un grafo *en* V. El conjunto de vértices de un grafo G se referencia como $V(G)$, y el conjunto de arcos como $E(G)$. No distinguiremos por lo general entre un grafo y su conjunto de vértices o arcos. Por ejemplo, usualmente diremos $v \in G$ para referirnos a $v \in V(G)$.

El número de vértices de un grafo G es su *orden* y se escribe $|G|$. Los grafos pueden ser finitos o infinitos, dependiendo de éste valor. Si el conjunto E satisface que (u, v) de E ssi $(v, u) \in E$ diremos que el grafo es *no dirigido*, y de lo contrario diremos que el grafo es *dirigido* o que es un *digrafo*. Cuando el grafo sea no dirigido escribiremos un arco (u, v) como el conjunto $\{u, v\}$ o simplemente uv. Un vértice v es *incidente* a un arco e si $v \in e$, y entonces e es un arco de v. Exceptuando el rigor de las definiciones de este capítulo, en este trabajo solo consideraremos grafos finitos y no dirigidos. Además, a menos que se diga lo contrario, $V = \{1, \ldots, n\}$.

Sean $G = (V, E)$ y $G' = (V', E')$ dos grafos. Si $V' \subset V$ y $E' \subset E$, entonces G' es un *subgrafo* de G (y G es un *supergrafo* de G') y lo denotamos $G' \subset G$. Si G' contiene a todos los arcos $uv \in E$ con $u, v \in V'$ decimos que G' es el *subgrafo inducido* por V' y se escribe $G' = G[V']$.

Dos vértices u, v de G son *adyacentes* o *vecinos*, si $uv \in E$. Si todos los vértices de un grafo G son adyacentes entre sí, entonces decimos que el grafo G es *completo*. Un grafo completo de n vértices se denota K^n.

Al conjunto de *vecinos* de un vértice v de G lo denotamos $N(v) = \{u : uv \in E\}$. En caso que el grafo sea dirigido, tendremos al conjunto de *entradas* $\delta^-(v) = \{u : (u, v) \in E\}$ y al de *salidas* $\delta^+(v) = \{u : (v, u) \in E\}$. Al cardinal de estos conjuntos se les llama *grado de entrada* y *grado de salida*, respectivamente. A un vértice con grado de entrada igual a cero se le dice *fuente* y a un vértice con grado de salida nulo se le dice *pozo*.

El grado $d_G(v)$ de un vértice $v \in G$ es el número sus de vecinos en G, es decir, $d_G(v) = |N(v)|$. Por otra parte, $\Delta(G)$ representa el valor del grado máximo del grafo, es decir, $\Delta(G) = \max_{v \in V}\{d_G(v)\}$. Usualmente el subíndice que indica el grafo se omitirá cuando no haya confusión posible.

1.1.1. Conexidad y ciclos

Un *camino* es un grafo $P = (V, E)$ de la forma

$$V = \{x_0, x_1, \ldots, x_k\} \qquad E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$$

donde los x_i son todos distintos. Decimos entonces que x_0 y x_k están *conectados* por P y son llamados sus *extremos*; los demás vértices $x_1, \ldots, x_{k-1}$ son los vértices *internos* de P. Al número de arcos en un camino se le llama la *longitud* o *largo* del camino. Usualmente nos referimos a un camino como a la secuencia de sus vértices, digamos, $P = x_0x_1 \ldots, x_k$, y llamamos P un camino entre x_0 y x_k.

Si $P = x_0 \ldots x_{k-1}$ es un camino y $k \geq 3$, entonces el grafo $C = (V(P), E(P) \cup \{x_{k-1}x_0\})$ es llamado un *ciclo*. Al igual que para los caminos, usualmente denotamos un ciclo como sus secuencia de vértices, digamos $C = x_0x_1 \ldots, x_kx_0$. El largo de un ciclo es el número de sus arcos (o vértices).

Tenemos entonces que dos vértices u, v están conectados si $u = v$ o si existe un camino $P = x_1, x_2, \ldots, x_k$ tal que $u = x_1$, $v = x_k$ y $x_ix_{i+1} \in E$ para todo $1 \leq i \leq k-1$. Esta relación es una relación de equivalencia en V, y por lo tanto, particiona a V en clases de equivalencia $\{V_j\}_{j=1}^{l}$. Los subgrafos inducidos por estas clases de equivalencia se llaman *componentes conexas* de G. Si G posee una única componente conexa, decimos que G es *conexo*.

Sea $G = (V, E)$ un grafo conexo. Definimos la relación de equivalencia R en E como sigue. Dados dos arcos e y g, tendremos que eRg si y sólo si $e = g$ o bien e y g pertenecen a un mismo ciclo. Tenemos que R también es una relación de equivalencia y, por lo tanto, particiona al conjunto E en clases de equivalencia $\{E_i\}_{i=1}^{s}$. A los subgafos inducidos por las clases de equivalencia se les llama **componentes biconexas** de G (ver Figura 1.2) La distancia $d_G(u, v)$ en G de dos vértices u, v es el largo del u-v camino más corto. Si u y v están en componentes conexas de G distintas $d_G(u, v) = \infty$. Para v un vértice de G definimos la familia de conjuntos $N^k(v)$ para $k \in \mathbb{N}$ como el conjunto de vértices a distancia k de v:

$$N^0(v) = \{v\} \quad , N^1(v) = N(v), \quad N^{k+1}(v) = \{u \in V \ : \ \exists w \in N(v)^k \text{ y } uw \in E\} \backslash N^{k-1}(v)$$

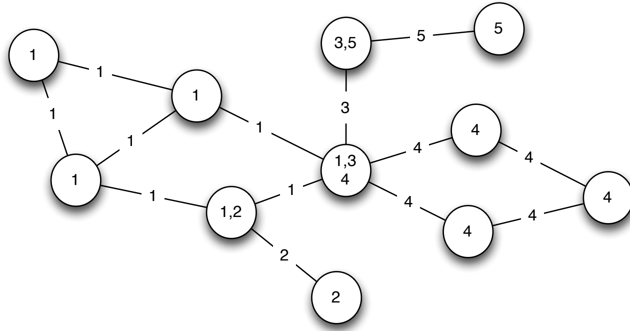

Figura 1.1: Componentes biconexas de un grafo. Arcos con la misma etiqueta están en la misma componente. Notar que un mismo vértice puede pertenecer a varias componentes biconexas.

1.1.2. Árboles, bipartitos y planos

Un grafo que no contiene ciclos se dice *acíclico* o bien un *bosque*. Un grafo conexo y acíclico se conoce como un *árbol*. Los vértices de grado 1 de un árbol son sus *hojas*. Usualmente en un árbol se destaca un vértice especial que llamamos *raíz*. La distancia de cualquier vértice v de un árbol a la raíz se conoce como el *nivel* de v. Sea $G = (V, E)$ grafo conexo, un *árbol recubridor* de G es un árbol cuyos vértices son los vértices de G.

Un grafo $G = (V, E)$ se dice *bipartito* si V admite una partición en dos clases tales que no existan arcos que tengan sus extremos en clases diferentes, o dicho de otro modo, tales que vértices de distintas clases no pueden ser adyacentes. Una caracterización elemental de los grafos bipartitos es aquella que dice que un grafo es bipartito si y solo si, el grafo no contiene ningún ciclo de largo impar.

Una *incrustación* en el plano de un grafo G es una asignación de coordenadas en $\mathbb{R}^2$ para cada vértice de G, donde los arcos corresponden al segmento de recta que une sus extremos. Un *grafo planar* es un grafo que puede ser incrustado en el plano, de modo que los conjuntos que definen sus aristas sólo se intersectan en los vértices. A tal incrustación se le llama *incrustación plana* y al grafo que define se le dice *grafo plano*.

1.2. Elementos de Complejidad computacional

1.2.1. Modelos de Computación

Máquina de Turing

Una *máquina de Turing* M de k-cintas se define como una tupla (Γ, Q, δ) donde

- Γ es un conjunto finito de símbolos. Asumimos que Γ contiene un elemento particu-

lar *blanco*, denotado $\square$; un símbolo designado para el *inicio* $\triangleright$; y los números 0 y 1. Llamamos a Γ el *alfabeto* de M.

- Q es un conjunto finito de los posibles estados internos de M, con un estado designado de *inicio* $q_{iniciar}$ y un estado final o de detención q_{parar}.
- δ es una función $\delta : Q \ \times \Gamma^{k-1} \to Q \times \Gamma^{k-1} \times L, S, R^k$, donde $k \geq 2$. Esta función es llamada la *función de transición* de M.

Podemos entender la definición anterior como una colección de k cintas. Una *cinta* es una línea infinita de celdas, dónde en cada celda se almacena un elemento de Γ. Cada cinta de M está equipada con una *cabeza*, que puede potencialmente escribir o leer símbolos de la cinta una celda a la vez. La computación de la máquina M está dividida en tiempos discretos, y la cabeza se puede mover a la izquierda o a la derecha en cada paso.

La primera cinta de la máquina es designada como la cinta de *entrada*. La cabeza de esta cinta sólo puede leer información de esta cinta, no escribir. La última cinta se conoce como la cinta de *salida*, donde la cabeza lectora sólo puede escribir. Las demás cintas se llaman cintas de *trabajo*.

La máquina tiene también un *registro*, donde puede almacenar un elemento de Q; éste será el *estado* de la máquina en ese instante. Este estado determina su acción en el próximo paso computacional, que queda definido por la función δ: se leen los símbolos de las $k-1$ cintas (todas menos la de salida); se escribe un nuevo símbolo en las $k-1$ cintas (todas menos la de entrada) según lo obtenido de δ; se cambia el registro para obtener un nuevo elemento de Q y luego se mueve la cabeza de cada cinta a la izquierda, a la derecha o se queda en su sitio.

Sean $f : \{0,1\}^* \to \{0,1\}^*$, $T : \mathbb{N} \to \mathbb{N}$ y $E : \mathbb{N} \to \mathbb{N}$ tres funciones y sea M una máquina de Turing. Decimos que M *computa* f si para cada $x \in \ \{0,1\}^*$ escrito en la cinta de entrada de M, cuando M para (entra en estado q_{parar}), con $f(x)$ escrito en la cinta de salida de M. Decimos que M computa f en **tiempo** $T(n)$ si su computación en cualquier entrada x requiere a lo más pasos $T(|x|)$. Por otra parte, que M computa f en **espacio** $E(n)$ si su computación en cualquier entrada x usa a lo más $E(|x|)$ celdas en las cintas de trabajo.

De la definición de una Máquina de Turing $\tau = (Q, \Sigma, \delta)$ es claro que ésta puede ser codificada como un vector finito $\omega_\tau = (q, s, \delta(q,s) = (q', s', l) : q \in Q, s \in \Sigma)$. Más aún, es posible construir una máquina de Turing $\tau_u = (Q_u, \Sigma_u, l_u)$ que es llamada *Universal* porque satisface que para cualquier máquina de Turing τ es posible tomar una codificación ω_τ de ésta en la cinta de τ_u tal que para cualquier condición inicial (q_0, s_0) de τ la máquina τ_u realiza el computo de τ.

Máquina de acceso aleatorio (RAM)

Una máquina *máquina de acceso aleatorio* o RAM (por su nombre en inglés *random access machine*) es una máquina de Turing que tiene acceso a una *memoria de acceso aleatorio*. Formalmente, la máquina tiene un arreglo infinito A inicialmente todo en blanco. Una de las cintas de trabajo de la máquina la llamamos la *cinta de direcciones*. Además la máquina tiene dos símbolos adicionales en su alfabeto, que llamamos L y E, y un estado adicional $q_{acceder}$.

Cuando la máquina entra en el estado $q_{acceder}$, si su cinta contiene $[k]L$ (donde $[k]$ denota la representación binaria de k), entonces se escribe el valor de $A(k)$ en la celda a continuación de L. Si la cinta contiene $[k]E\sigma$ (donde σ es algún símbolo del alfabeto), entonces a $A(k)$ se le asigna σ.

Un elemento importante a destacar para el modelo RAM es que el trabajo con números muy grandes puede ser tramposo, ya que al exigir poder acceder instantáneamente al vector A podríamos construir rápidamente números de muchas cifras, lo cual parece no ser posible de hacer en la realidad. Es por esto que suponemos que para $t \geq \log(n)$ ningún número puede exceder $\mathcal{O}(t)$ bits en t pasos. Finalmente, se tiene que una función booleana f es computable en tiempo $T(n)$ por una máquina RAM, entonces está en **TIEMPO**$(T(n)^2)$.

Máquina paralela de acceso aleatorio (PRAM)

Una generalización natural del modelo RAM a la computación paralela es la *máquina paralela de acceso aleatorio* o PRAM (por su nombre en inglés *parallel random access machine*) introducida independientemente por Wyllie y Goldschlager [6, 7]. El modelo PRAM consiste en una colección de RAMS, que llamamos *procesadores*, que corren en paralelo y se comunican a través de una memoria común.

El modelo básico de una PRAM consiste en una colección no acotada de procesadores RAM $P_0, P_1, P_2, \ldots$ y una cantidad no acotada de celdas de memoria compartida $C_0, C_1, C_2, \ldots$. Cada procesador P_i tiene su propia memoria local, conoce su índice i, y tiene las instrucciones necesarias para directamente e indirectamente leer y/o escribir en la memoria compartida.

Hay dos problemas técnicos importantes que hay que resolver para este modelo. El primero es el modo en que un conjunto finito de los infinitos procesadores son activados. Un modo de resolver esto, que usualmente no se menciona en la literatura, es designar un registro especial a P_0 que especifique el máximo índice de un vértice activo. Cualquier procesador no detenido con un índice menor a este registro puede ejecutar su programa. Inicialmente solo P_0 está activo y todos los demás procesadores están suspendidos esperando a ejecutar su primera instrucción. P_0 entonces calcula el número de procesadores requeridos para el problema en cuestión y carga este valor en su registro especial. La computación procede hasta que P_0 para, en cuyo instante todos los procesadores activos también paran.

El segundo problema técnico concierne a la forma en que se arbitra el acceso a la memoria compartida. En todos los modelos se asume que el ciclo de instrucciones separa las lecturas de las escrituras en la memoria compartida. Cada instrucción PRAM es ejecutada en un ciclo con tres fases: Primero se realizan las lecturas de la memoria compartida, luego se realizan los cómputos asociados y finalmente la escritura. Esto resuelve los conflictos de lectura y escritura en la memoria compartida, pero no elimina los conflictos en el acceso. Esto se resuelve de varias formas, incluyendo:

- CRCW-PRAM (de *concurrent read, concurrent write PRAM*): todos pueden leer y escribir simultáneamente en una misma celda de memoria compartida, pero se requiere algún método de arbitraje para evitar escrituras en la misma celda. Un ejemplo de

arbitraje es asignarle una prioridad a cada procesador según su enumeración.

- CREW-PRAM (de *concurrent read, exclusive write PRAM*): permite lecturas simultáneas de una misma celda de memoria compartida, pero solo un procesador puede escribir en la celda.
- CROW-PRAM (de *concurrent read, owner write PRAM*): igual que antes pero esta vez cada procesador es *dueño* de ciertas celdas, donde sólo él puede escribir.
- EREW-PRAM (de *exclusive read, exclusive write PRAM*): exige que dos procesadores no puedan tener acceso a una misma celda, cualquiera sea, ni para leer ni para escribir.

Nuestro modelo PRAM estándar va a ser el CREW-PRAM. Notemos que como las RAM están imposibilitadas de generar grandes números, esto es, si $t \geq \log(n)$ ningún número puede exceder $\mathcal{O}(t)$ bits en t pasos, una computación de tiempo t con p procesadores no puede almacenar más de $\mathcal{O}(pt^2)$ bits de información. Por lo tanto, para estudiar la complejidad p y t en conjunto serán suficientes para caracterizar adecuadamente los requerimientos de un algoritmo que se ejecute en una PRAM.

Sea M una PRAM. Convendremos lo siguiente para las *entradas y salidas*. Una entrada $x \in \{0,1\}^n$ se presenta en M poniendo el entero n en la celda C_0, y los bits $x_1, \dots x_n$ de x en las celdas compartidas $C_1, \dots, C_n$. M escribe su salida $y \in \{0,1\}^m$ similarmente: el entero m en la memoria compartida C_0 y los bits $y_1, \dots, y_m$ en las celdas de memoria compartida $C_1, \dots, C_m$.

Sea $f : \{0,1\}^* \to \{0,1\}^*$ La función f es computable en **tiempo paralelo** $t(n)$ **usando** $p(n)$ **procesadores** si existe una PRAM M que para cada input $x \in \{0,1\}^n$, la máquina M para en a lo más $t(n)$ pasos, activa a lo más $p(n)$ procesadores y escribe como salida $f(x)$.

1.2.2. Clases de complejidad

Una *clase de complejidad* es un conjunto de funciones que pueden ser computadas bajo ciertas restricciones de recursos. Por lo general nos concentraremos en las funciones booleanas, es decir, las funciones que tienen un bit como salida. Identificamos este tipo de funciones f con el conjunto $L_f = \{x : f(x) = 1\}$ y llamamos estos conjuntos *lenguajes* o *problemas de decisión*.

Identificaremos al problema computacional de computar f (es decir, dado x computar $f(x)$) con el problema de decidir L_f (es decir, dado x, decidir si $x \in L_f$). Decimos que una máquina de Turing *decide* un lenguaje $L \subset \{0,1\}^*$ si computa la función $f_L : \{0,1\}^* \to \{0,1\}$, donde $f_L(x) = 1 \iff x \in L$.

Sea $T : \mathbb{N} \to \mathbb{N}$ y $E : \mathbb{N} \to \mathbb{N}$ dos funciones. Un lenguaje L está en **TIEMPO**$(T(n))$ si existe una máquina de Turing que decide L y que corre en tiempo $\mathcal{O}(T(n))$. Decimos también que L está en **ESPACIO**$(E(n))$ si existe una máquina de Turing que decide L y usa espacio $\mathcal{O}(E(n))$.

Clases P, L, NC, FNC

La clase **P** es la clase de los problemas decidibles en tiempo polinomial en el tamaño de la entrada, es decir, $\mathbf{P} = \bigcup_{c\geq 1} \mathbf{TIEMPO}(n^c)$. La clase **L** es la clase de todos los problemas decidibles por una máquina de Turing en espacio *logarítmico*, es decir, $L \in \mathbf{ESPACIO}(\log(\mathbf{n}))$.

Sea $k \in \mathbb{N}$. La clase $\mathbf{NC}^k$ es la clase de los problemas decidibles en tiempo paralelo $\mathcal{O}(\log^k(n))$ usando $n^{\mathcal{O}(1)}$ procesadores. La clase **NC** es la unión de todas las clases $\mathbf{NC}^k$, es decir $\mathbf{NC} = \bigcup_{k\geq 1} \mathbf{NC}^k$.

Se tiene que si un lenguaje NC para algún modelo de PRAM (CRCW, CREW, EREW, etc), entonces pertenecerá a NC para todos los demás.

La clase $\mathbf{FNC}^k$ es el conjunto de todas las funciones de $\{0,1\}^*$ en $\{0,1\}^*$ que son computables en tiempo paralelo $\mathcal{O}(\log^k(n))$ usando $n^{\mathcal{O}(1)}$ procesadores. La clase **FNC**, como la clase **NC**, es la unión de las clases $\mathbf{FNC}^k$, es decir $\mathbf{FNC} = \bigcup_{k\geq 1} \mathbf{FNC}^k$.

Notemos que podemos simular una PRAM con una RAM (y entonces con una máquina de Turing) simplemente realizando los cálculos que hace cada procesador secuencialmente, uno tras otro, todo en un solo procesador. Es por esto que L es decidible en tiempo $n^{\mathcal{O}(1)}$ si y solo si L es decidible en tiempo paralelo $n^{\mathcal{O}(1)}$ con $n^{\mathcal{O}(1)}$ procesadores. Por lo tanto $\mathbf{NC} \subset \mathbf{P}$. Una pregunta abierta, similar al caso **P** v/s **NP**, es determinar si esta inclusión es estricta.

Por convención decimos que los problemas que pertenecen a **P** son los problemas *factibles*, en el sentido que es posible resolverlos en la práctica en un tiempo razonable. Un problema que pertenece a **NC** sería entonces un problema *factible eficientemente en paralelo*, en el sentido informal que al 'paralelizar' se 'gana' bastante. Si $\mathbf{NC} \neq \mathbf{P}$, significa que existen problemas *inherentemente secuenciales*, en el sentido que son factibles, pero al paralizar no se 'gana mucho'.

Los problemas que parecieran ser los más probables de ser *inherentemente secuenciales* son los problemas **P**-Completos.

Problemas P-Completos

Un lenguaje L es reducible *mucho a uno* a un lenguaje L', escrito $L \leq_m L'$, si existe una función f tal que $x \in L$ si y sólo si $f(x) \in L'$. Decimos que L es reducible en espacio logarítmico a un lenguaje L' si L es reducible mucho a uno a L' por una función f y además f es computable en espacio $\mathcal{O}(\log(n))$.

Un lenguaje L' es **P**-Duro si para todo lenguaje L en **P**, L es reducible en espacio logarítmico a L'. Un lenguaje en **P** que es además **P**-Duro se dice **P**-Completo.

Como dijimos anteriormente, los problemas que parecieran ser los más probables de ser *inherentemente secuenciales* son los problemas **P**-Completos. Esto se debe a que si un lenguaje L es **P**-Completo, pero es *factible eficientemente en paralelo*, es decir está también en **NC**, entonces $\mathbf{P} = \mathbf{NC}$.

Un ejemplo de un problema **P**-Completo es el *problema de simulación de máquinas genéricas* (GMSP) que consiste en, dada una codificación ω_τ de una máquina de Turing τ, y un entero t en **unario**, determinar si la máquina τ acepta x en t pasos. Intuitivamente éste problema está en **P** ya que como t está en unario, una máquina universal puede simular t pasos de τ es tiempo polinomial en el tamaño de la entrada. La **P**-Complitud viene dada del hecho que se puede reducir cualquier lenguaje $L \in \mathbf{P}$ construyendo una codificación ω_{τ_L} de la máquina τ_L que decide L y tomando t como una cota superior del tiempo que toma τ_L en decidir, todo lo cual se puede hacer en espacio logarítmico.

Como una analogía con la teoría de la **NP**-Complitud, considere el *problema de simulación de máquinas no deterministas genéricas* versus **SAT**. Ambos son **NP**-Completos, pero la simplicidad del segundo problema lo hace un punto de inicio razonable al momento de buscar demostrar que cierto problema es **NP**-Completo. De éste modo un problema **P**-Completo análogo a **SAT** sería el *Problema del valor de circuito* **CVP**.

CVP: Problema del valor de circuito.

Un *circuito booleano* de n-entradas y m-salidas es un grafo orientado y acíclico con n fuentes y m pozos, cuyos vértices, que llamamos *puertas* están etiquetados con OR, AND y NOT, donde cada etiqueta representa que el vértice ejecutará la operación lógica correspondiente para sus entradas. Llamamos a las n fuentes *entradas* y a los m pozos *salidas*. Al grado de entrada de una puerta se le llama *fan-in* y al grado de salida se le llama *fan-out*. Tendremos entonces que salvo las entradas, las puertas de tipo OR o AND tendrán fan-in igual a 2, mientras que las puertas de tipo NOT tendrán fan-in igual a uno. El *tamaño* de un circuito ϕ es igual al número de puertas en ϕ. La *profundidad* de ϕ, es la longitud del camino más largo de una entrada a una salida.

Si ϕ es un circuito booleano y $x \in \{V, F\}^n$ es una asignación de verdad de sus entradas, entonces *el valor del circuito* en x, denotado $\phi(x)$ se define del modo natural. Formalmente, para toda puerta v de ϕ, le asignamos el valor $val(v)$ como sigue: Si v es la i-ésima entrada del circuito, entonces $val(v) = x_i$, de lo contrario $val(v)$ se define recursivamente aplicando la operación lógica en los vértices en $\delta^-(v)$. El valor del circuito entonces corresponde al vector $y = (y_1 \ldots, y_m)$, donde y_i es el valor de la salida i-ésima.

La *codificación estándar* $\overline{\phi}$ de un circuito ϕ es una cadena en $\{0,1\}^*$ agrupada en una secuencia de $4-tuplas$ (v, g, l, r), una tupla por cada puerta de ϕ, seguida de dos secuencias de codificaciones de números de vértices $x_1, \ldots, x_n$ y $y_1, \ldots, y_n$. En la codificación, las puertas de ϕ están enumeradas arbitrariamente entre 1 y el tamaño de ϕ. La tupla (v, g, l, r) describe una puerta v y sus conexiones como sigue: la puerta v es del tipo g, con $g \in$ {OR, ANDS, NOT}. Los valores de r y l corresponden a las entradas (si corresponde) de la puerta v. El número de la i-ésima entrada está dado x_i y el de la salida j-ésima está dado por y_j.

El problema del valor de circuito **CVP** propuesto por Ladner [15] corresponde a determinar, dada una codificación $\overline{\phi}$ de un circuito booleano ϕ y una asignación de verdad para sus entradas x, el valor de una de sus salidas y. En [12] hay una demostración de que el **CVP** es **P**-Completo.

Como dijimos anteriormente, el **CVP** juega el mismo rol en la teoría de la **P**-Complitud que el que juega **SAT** en la teoría de la **NP**-Complitud. Como **SAT**, el **CVP** es el problema **P**-Completo *fundamental*, en el sentido en que es el que más frecuentemente se usa para demostrar que otros problemas son **P**-Completos. Análogamente a **SAT**, hay variantes del **CVP** que siguen siendo **P**-Completas, las cuales usualmente simplifican las reducciones, como veremos en los capítulos siguientes. Las simplificaciones que usaremos en este trabajo son:

- **Top-CVP**: **CVP Ordenado topológicamente.**
  Un *orden topológico* de un grafo G que es dirigido y acíclico, es una enumeración de los vértices de tal forma que si $(u, v) \in E$, entonces la enumeración de u es menor que la de v. Una de las propiedades del **CVP** es que para cualquier circuito dado, hay un algoritmo secuencial que en cualquier entrada recorre sus puertas en un orden fijo, de modo que evalúa cada puerta una sola vez y obtiene el valor de la salida designada en tiempo polinomial. La virtud de un circuito topológicamente ordenado es que éste orden está directamente especificado en la entrada.
  Podemos suponer que cualquiera de las variantes del **CVP** que expondremos a continuación tiene la propiedad de ser topológicamente ordenado.
- **MCVP**: **Problema del valor de circuitos monótonos.**
  Se trata de la versión restringida del **CVP** donde el circuito contiene sólo puertas *monótonas*, es decir, de tipo OR o AND. Este problema es útil en la situación (veremos que bastante común) en que las negaciones sean difíciles de simular directamente.
- **AMCVP**: **Problema del valor de circuitos monótonos alternantes.**
  Es un caso especial del problema anterior. Un circuito se dice *alternante* si en cualquier camino desde una entrada a una salida, las puertas alternan entre OR y AND. Además, exigimos que las entradas conecten solo a entradas tipo OR y que las salidas sean todas del tipo OR. La propiedad alternante usualmente reduce el número de interacciones entre dispositivos (o *gadgets*) usados al hacer las reducciones, lo cual por general las simplifica.
- **AM2CVP**: **AMCVP con fan-in 2 y fan-out 2.**
  Este, nuevamente, es una restricción del caso anterior. En este caso, exigimos que todas las puertas del circuito tengan fanin y fanout (grado de entrada y grado de salida, respectivamente) igual a dos, con las excepciones obvias de las entradas, que tienen fanin cero; y las salidas, que tienen fanout cero.
- **SAM2CVP**: **AMCVP con fan-in 2, fan-out 2 y síncrono.**
  Definimos la *capa* o *layer* de una puerta g, denotado $layer(g)$, como cero para las entradas del circuito y para el resto, como el camino más largo hasta una entrada. Un circuito es *síncrono* si todas las entradas de una puerta g vienen de puertas en $layer(g) - 1$. Más aún, exigimos que todas las salidas estén al mismo nivel, digamos, el máximo. Así, podemos particionar las puertas en capas, donde los arcos van entre dos capas contiguas y todas las salidas están en la capa más alta. **SAM2CVP** es la restricción de **AM2CVP** a circuitos síncronos. Notemos que en un circuito que es a la vez alternate y síncrono, se cumple que todos los vértices de una misma capa son del mismo tipo. Además que el fanin y el fanout sean iguales a dos implica que cada capa contiene exactamente la misma cantidad de vértices.

Como dijimos anteriormente, todas estas variantes siguen siendo **P**-Completas. Sin embargo, hay que ser precavido, porque hay variantes importantes que están en **NC**, como es el ejemplo del **PMCVP**: **Problema del valor de circuitos monótonos planos**, para el cual existe un algoritmo que corre en tiempo paralelo $O(\log^4 n)$ [5].

1.2.3. Ejemplo de problema en NC (FNC): Sumas de prefijos

A continuación daremos un ejemplo de un problema en **FNC**, pero que a partir de ellos se pueden fácilmente definir problemas de decisión que están en **NC**. Éste nos servirá por un lado para ilustrar el tipo de técnicas a usar y por otro lado, lo usaremos como herramienta en los capítulos posteriores como subrutina de nuevos algoritmos.

Dado un grupo $H = (X, +)$. La suma de prefijos de una secuencia $x_0, x_1, \ldots, x_n \in H$ es una segunda secuencia $y_0, y_1, \ldots, y_n \in H$ tal que $y_k = \sum_{i=0}^{k} x_i$. En [13] encontramos el siguiente algoritmo paralelo para calcular sumas de prefijos de secuencias de largo n, usando tiempo $\mathcal{O}(\log(n))$ y $\mathcal{O}(n)$ procesadores.

Dado una entrada $x_0, x_1, \ldots, x_n \in H$, calcularemos las sumas de prefijos en paralelo como sigue: (1) Primero computamos las sumas de pares consecutivos, donde en el primer elemento de cada par tendrá un índice impar:

$$z_0 = x_0 + x_1, \quad z_1 = x_2 + x_3, \quad \ldots \quad , z_k = x_{2k} + x_{2k+1}.$$

(2) A continuación recursivamente calcularemos las sumas de prefijos $w_1, \ldots w_{n/2}$ de $z_0, \ldots z_{n/2}$. (3) Finalmente expandimos cada término de la secuencia $w_1, \ldots w_{n/2}$ obteniendo los términos de la suma original

$$y_0 = x_0, \quad y_1 = w_0, \quad y_2 = w_0 + x_2, \quad y_3 = w_1, \quad \text{etc.}$$

Tenemos por otro lado que (1) y (3) requieren tiempo paralelo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores, ya que en ambos casos cada procesador tiene que hacer una suma de dos valores, los cuales tienen una dirección fija. Para el paso (2), notamos que como en cada recursión el tamaño de la entrada se reduce a la mitad, se requiere a lo más tiempo paralelo $\mathcal{O}(\log(n))$. Obtenemos entonces las sumas de prefijos en tiempo $\mathcal{O}(\log(n))$ usando a lo más $\mathcal{O}(n)$ procesadores (en estricto rigor se requiere sólo $\mathcal{O}(n/\log(n))$).

Otros problemas que están en **FNC** y que usaremos en los capítulos posteriores son: encontrar las componentes conexas de un grafo, encontrar las componentes biconexas de un grafo conexo, construir un árbol generador, reconocer un grafo bipartito y reconstruir sus partes [13].

1.3. Redes de Autómatas

Sea $G = (V, E)$ un grafo no necesariamente finito, que podría ser o no dirigido, donde V es el conjunto de vértices y E el conjunto de arcos. Una *Red de Autómatas* definida en G es un trío $\mathcal{A} = (G, Q, (f_i : i \in V))$, donde:

- G es localmente finito, lo que significa que cada vecindad $N(v)$ es finita para todo v, es decir, $|N(v)| < \infty$.
- Q es un conjunto que llamaremos estados y que por lo general se considera finito.
- $f_v : Q^{|N(v)|} \to Q$ es una función, que llamaremos *de transición*, asociada al vértice v.

Cuando el grafo es finito no hay perdida de generalidad en asumir que cada función de transición f_v está definida de $Q^{|V|}$ en Q. Al conjunto $Q^{|V|}$ lo llamaremos conjunto de *configuraciones* de G.

Sobre el conjunto de configuraciones definiremos la función $F : Q^{|V|} \to Q^{|V|}$, que llamaremos la función de transición global, y que corresponde a la función que está construida a partir de las funciones de transición de cada vértice $(f_i : i \in V)$, con algún tipo de modo de iterar, por ejemplo paralelo o secuencial.

El modo de actualizar o iterar un autómata $\mathcal{A}$ puede tomar diferentes formas. La iteración *síncrona* o *en paralelo* de una red $\mathcal{A} = (G, Q, (f_i : i \in V))$ se obtiene cuando todos los vértices de la red se actualizan al mismo tiempo, por lo que la *dinámica* o *trayectoria* de una cierta configuración x viene dada por el conjunto $\{x(t) : t \geq 0\}$, donde $x(0) = x$ y $x(t+1) = F(x(t))$. Para analizar otros modos de actualizar un autómata, consideremos una *palabra* $w = (\omega_1, \omega_2, \ldots, \omega_m) \in \mathcal{P}(V)^m$ es una m-tupla de subconjuntos de V, tales que forman una partición de V. Un modo de iterar *asíncrono* de una red $\mathcal{A} = (G, Q, (f_i : i \in V))$ es aquél que es secuencial en el orden de w pero síncrono en cada ω_i. Más precisamente, si F es la función de transición global de $\mathcal{A}$, y x es una configuración de G, entonces

$$F(x, w) = F(F(x, (\omega_1)), w') \text{ con } w' = (\omega_2, \ldots, \omega_m) \text{ donde } (F(x, (\omega_k)))_i = \begin{cases} f_i(x) & \text{si } i \in \omega_k \\ x_i & \text{si } i \notin \omega_k \end{cases}$$

luego, en este caso la trayectoria de una configuración x es el conjunto $(\{x(t) : t \geq 0\}, w)$ donde $x(0) = x$ y $x(t+1) = F(x, w)$.

Para una palabra $w = (\omega_1, \omega_2, \ldots, \omega_m)$ y un vértice $v \in V$, diremos que $w(v) = k$ si $v \in \omega_k$. Notemos entonces que la iteración *síncrona* es el caso particular en que $w = (V)$ o bien $w(v) = 1$ para todo $v \in V$.

Cuando no haya ambigüedad posible no habrá diferencia entre referirse al autómata $\mathcal{A} = (G, Q, (f_i : i \in V))$, como el trío (G, Q, F) donde F es la regla de transición global asociada a las funciones de transición $(f_i : i \in V)$. Más aún, a menudo nos referirnos al autómata $\mathcal{A}$ como al grafo G con la regla F.

Sea x una configuración de G con la regla F. Diremos que la dinámica converge o entra a un punto fijo si es que existe $T \geq 0$ tal que $x(t) = x(T)$ $\forall t \geq T$, es decir, si existe una configuración y en la trayectoria de x tal que $F(y) = y$. Por otra parte, diremos que la

dinámica entra en un ciclo de largo k si existe $t \geq 0$ tal que $x(t) \neq x(t+1) \neq \cdots \neq x(t+(k-1))$ pero $x(t+k) = x(t)$.

En este trabajo nos enfocaremos en el caso particular en que las redes de autómatas están definidas sobre un grafo finito, conexo y no dirigido, donde además el conjunto de estados es $Q = \{0, 1\}$. En este contexto diremos que un nodo en estado 0 es *pasivo* y un nodo en estado 1 es *activo*.

1.3.1. Autómatas Celulares

Sea $V = \mathbb{Z}^d$. Si un conjunto de arcos $E \subset \mathbb{Z}^d \times \mathbb{Z}^d$ es invariante ante traslaciones, es decir, $(i, j) \in E$ ssi $(j + k, i + k) \in E$, el grafo $G = (V, E)$ se llama *espacio celular*. En términos de vecindades, la condición de ser invariante ante traslaciones se expresa por las igualdades: $N(i) = (i - j) + N(j)$, para cualquier $i, j \in \mathbb{Z}^d$. A partir de $N(i) = i + N(0)$, deducimos que toda la información de la estructura de vecindades está contenida en $N(0)$, por lo que escribimos $G = (\mathbb{Z}^d, N(0))$.

Si $d = 1$, entonces el vecindario $N(0)$ normalmente se considera como $\{-p, \ldots, 0, \ldots, p\}$, y entonces $N(i) = \{i - p, \ldots, i, \ldots, i + p\}$. Cuando $d = 2$, las elecciones más comunes son $N = \{(1, 0), (-1, 0), (0, 0), (0, 1), (0, -1)\}$ conocida como la vecindad de tipo *von Neumann*; y $M = N \cup \{(1, 1), (1, -1), (-1, 1), (-1, -1)\}$, conocida como al vecindad de *Moore*.

Un *Autómata Celular* (AC) es una red de autómatas definida sobre un espacio celular, donde además las funciones de transición son también invariantes ante traslaciones, es decir $f_i = f$ para todo $i \in \mathbb{Z}^d$.

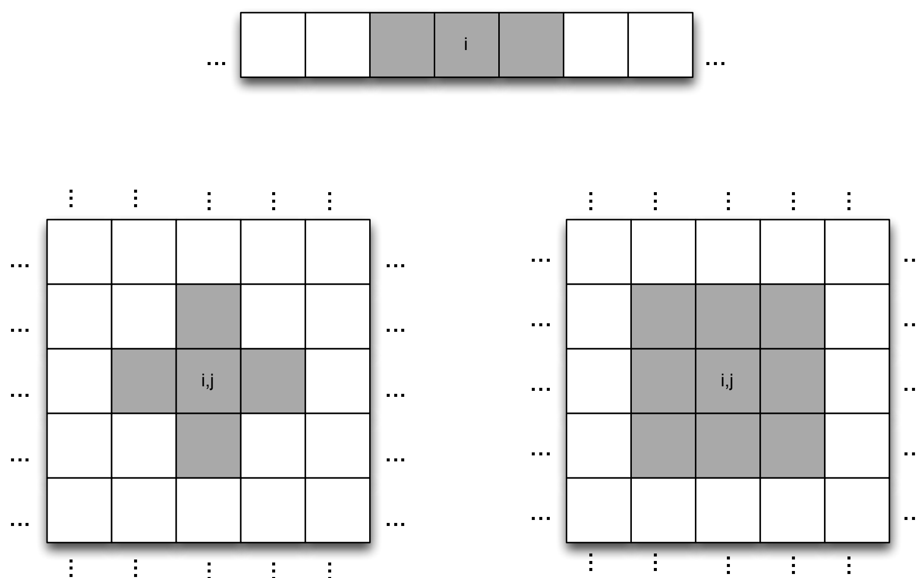

Figura 1.2: Arriba: un AC unidimensional con $N(0) = \{-1, 0, 1\}$, Abajo: El caso con $d = 2$ con vecindades del tipo Moore (izquierda) y von Neumann (derecha)

Una Red de Autómatas se dice *Universal* si puede simular una Máquina de Turing Universal. En [1] tenemos como resultado que cualquier máquina de Turing $\tau = (Q, \Sigma, \delta)$ puede ser simulada por un autómata celular unidimensional $\mathcal{A} = (\mathbb{Z}, \tilde{Q}, f)$ con vecindad $\{-1, 0, 1\}$. Por lo tanto tenemos que existe una Red de Autómatas Universal. Más aún, en [23] se demostró que el AC 110 es universal.

1.3.2. Redes Neuronales

Introducidas por McCulloch y Pitts [17] para modelar ciertas propiedades del sistema nervioso, las Redes Neuronales son capaces de simular cualquier máquina de Turing si se provee un número infinito de células. Más aún, éste tipo de autómatas ha tenido un enorme desarrollo para aplicaciones en varios dominios: física, inteligencia artificial, y biología [9].

Una *Red Neuronal* $\mathcal{N}$ es una clase particular de redes de autómatas, donde se le asocia a cada arista del grafo un *peso*, es decir, si E es el conjunto de aristas, para cada $(i, j) \in E$ asociamos un número real $a_{ij} \in \mathbb{R}$. Si $(i, j) \notin E$, entonces $a_{ij} = 0$. El conjunto de estados será $Q = \{0, 1\}$ y diremos que un vértice en estado 1 está *activo*, mientras que el estado 0 será para los vértices *pasivos*. Desde ahora en adelante $Q = \{0, 1\}$. Las funciones locales son las siguientes:

$$f_i : \{0,1\}^{|N(i)|} \to \{0,1\} \quad , \quad f(x_j \ : \ j \in N(i)) = \eta\left(\sum_{j \in N(i)} a_{ij}x_j - b_i\right)$$

donde η es la función umbral: $\eta(x) = \begin{cases} 1 & si \quad x \geq 0 \\ 0 & si \quad x < 0 \end{cases}$

La matriz $A = (a_{ij} : i, j \in V)$ se llama la matriz de conexiones y $b = (b_i \ : \ i \in V)$ se llama el vector de umbrales. Una *Red Neuronal* $\mathcal{N}$ es la tupla $\mathcal{N} = (V, E, A, b)$ y corresponderá a la red de autómatas definida sobre el grafo $G = (V, E)$, estados $\{0, 1\}$ y con funciones de transición definidas a partir de A y b. Sea $\overline{\eta}(x_i \ : \ i \in V) = (\eta(x_i) \ : \ i \in V)$ la función umbral multidimensional. Podemos entonces escribir la función de transición global de $\mathcal{N}$ como $F(x) = \overline{\eta}(Ax - b)$.

1.4. Problemas de decisión y redes de autómatas

Usualmente los estudios teóricos clásicos asociados a redes de autómatas se encuentran con el hecho que es muy difícil, o poco común, obtener caracterizaciones del comportamiento general de la dinámica de un autómata celular. Por otro lado, las redes de autómatas tienen relación en el proceso paralelo de la información y algoritmos, ya que como dijimos hay ciertos CA que son capaces de simular una máquina de Turing Universal.

En ese sentido, la idea de introducir la complejidad computacional en redes de autómatas permite formar un puente entre esos dos aspectos del problema: su naturaleza dinámica y sus capacidades algorítmicas. Uno de los primeros en hacer esto ha sido Chrisopher Moore [19, 22, 21, 20]. En estos trabajos estudia varios autómatas celulares y sistemas relacionados.

En esta perspectiva, debemos definir qué entenderemos por la complejidad computacional de una red de autómatas. El enfoque que adoptaremos, entre varios posibles, será el de preguntarnos, dada una configuración de un autómata, el posible cambio de estado de un nodo cuando la red se actualiza en paralelo. Llamaremos a este problema **PER**, ya que en algún sentido estudiaremos como la información (nodos activos) *percola* en el grafo.

PER: Dada una red de autómatas con estados 0 y 1 definida sobre un grafo no dirigido G, conexo, y una función de transición global F. Sea x una configuración de G y v un vértice en $V(G)$ inicialmente pasivo, i.e. $x_v = 0$. Nos preguntamos entonces si en la trayectoria definida por x, el vértice v pasará en algún momento a ser activo. En otras palabras ¿Existe $T \geq 0$ tal que $x_v(T) = 1$?.

Estudiaremos entonces este problema para las reglas *Bootstrap Percolation*, *Mayoría estricta*, y otras. Para cada regla variamos la topología de la red obteniendo distintas complejidades, demostrando en unos casos **P**-Complitud, en otros pertenencia a **NC**.

Capítulo 2

Bootstrap Percolation

Sea $G = (V, E)$ un grafo finito, simple, conexo no dirigido, donde V es el conjunto de vértices y E el conjunto de arcos. Consideremos la siguiente función de transición:

$$f_i(x) = \begin{cases} 1 & \text{si} \quad x_i = 1 \\ 1 & \text{si} \quad \sum_{j \in N(i)} x_j > \frac{|N(i)|}{2} \quad \text{y} \quad x_i = 0 \\ 0 & \text{si} \quad \sum_{j \in N(i)} x_j \leq \frac{|N(i)|}{2} \quad \text{y} \quad x_i = 0 \end{cases}$$

En otras palabras, un vértice pasivo se volverá activo si la mayoría estricta de sus vecinos está activa, y luego nunca cambia su estado. Llamamos *Bootstrap Percolation* a la función de transición global dada por $(f_i : i \in V)$.

Éste es un caso especial de Bootstrap Percolation, que corresponde a un modelo introducido a finales de la década de los 70's para estudiar propiedades de algunos materiales magnéticos [4]. Más recientemente ha sido usada para modelar esparcimiento de enfermedades [2], difusión de alertas en redes distribuidas [25], formación de pilas de arena [16], y otras [18].

Los estudios teóricos clásicos relacionados a Bootstrap Percolation trabajan con la pregunta de cual es la mínima cantidad de sitios activos (infectados) de modo que se infecte, digamos con alta probabilidad, toda la estructura. Los resultados en este sentido son complicados y usualmente restringidos a familias específicas de grafos (lattices, grafos cúbicos, etc) [24, 3]. Nuestro enfoque es distinto y complementario al anterior, ya que nos preguntamos por la posibilidad de que un vértice en específico vaya a ser infectado dada una configuración inicial, y si ello se puede producir rápida o eficientemente.

En la próxima sección enunciaremos el resultado principal de este capítulo, el cual dice que cuando el grafo tiene grado pequeño, el problema **PER** con la regla Bootstrap Percolation está en **NC**, y en caso contrario es **P**-Completo. Luego, aplicaremos dicho resultados a reglas similares.

Cabe destacar que todos lo incluido en este capítulo conforma un artículo publicado en *Theoretical Computer Science* [10], en conjunto a los profesores Eric Goles y Ioan Todinca.

2.1. La complejidad de Bootstrap Percolation

Para Bootstrap Percolation tendremos el siguiente teorema, el cual tiene la cualidad de contar con un cierto acotamiento del problema "por arriba" (**P**-Complitud) y "por abajo" (pertenencia a **NC**), con límites muy nítidos:

Teorema 2.1 *Para Bootstrap Percolation:*

1. *En la familia de grafos G tales que $\Delta(G) \geq 5$, el problema* **PER** *es* ***P****-Completo.*
2. *En la familia de grafos G tales que $\Delta(G) \leq 4$, el problema* **PER** *está en* **NC**.

La demostración de este teorema la dividiremos en dos partes, una para cada caso. En la próxima subsección demostraremos el primer caso, la cual consistirá básicamente de explicar como podremos simular un circuito monótono usando *gadgets* (dispositivos) que simularán los distintos elementos del circuito.

En cambio la segunda parte de la demostración requerirá introducir algunos elementos para construir un algoritmo que sea eficiente en paralelo. La clave será utilizar la restricción en el grado para probar que se pueden encontrar estructuras sencillas (ciclos, caminos) que aseguran que un nodo inicialmente pasivo está a salvo de ser afectado por los activos.

2.1.1. Caso P-Completo

De acuerdo con lo expuesto en el primer capítulo, demostraremos la **P**-Complitud reduciendo en espacio logarítmico a uno de los casos restringidos del **CVP** a **PER** en este caso particular. En este caso elegiremos **AM2CVP**, es decir, el circuito es alternante, monótono, fanin 2, fanout 2 y síncrono (en realidad solo será estrictamente necesario que el circuito tenga fanin 2, fanout 2 y sea monótono).

Demostración. (del Teorema 2.1 1.)

Notemos que como los vértices activos se mantienen activos, en a lo más $|V|$ pasos, la dinámica entra a un punto fijo. Como a su vez cada paso se puede simular en tiempo polinomial, **PER** se puede decidir en tiempo polinomial, y luego está en **P**.

Para demostrar que **PER** es **P**-Completo, vamos a reducir el caso restringido del **CVP** a **PER**. Como **AM2CVP** es **P**-Completo, si la reducción usa sólo espacio logarítmico, entonces **PER** será **P**-Completo.

Para reducir **AM2CVP** a **PER**, vamos a construir, dado un circuito monótono, un grafo que simule sus puertas, donde los vértices activos van a representar valores *true*, y los vértices pasivos *false*. Como el circuito es monótono, solo tendremos que simular puertas AND y OR.

La puerta AND (Figura 2.1 (a)) es simulado por un vértice inicialmente pasivo con grado 3, donde dos de sus vecinos serán sus inputs y el otro será el output. Por la regla Bootstrap Percolation, este vértice se volverá activo sólo si dos o más vecinos son activos en un cierto step, por lo que actuará como una puerta AND si esos dos vecinos corresponden a los considerados como inputs. De un modo similar, la puerta OR (Figura 2.1(b)) será simulada por un vértice inicialmente pasivo con grado 5, donde dos vecinos están inicialmente activos (que llamaremos auxiliares) y los otros tres corresponden a los inputs y al output, como en la puerta AND. En este caso, el vértice inicialmente pasivo pasará a activo siempre y cuando al menos uno de sus vecinos, que no sean uno de los auxiliares, sean activos en un cierto step. Por lo tanto se comporta como una puerta OR.

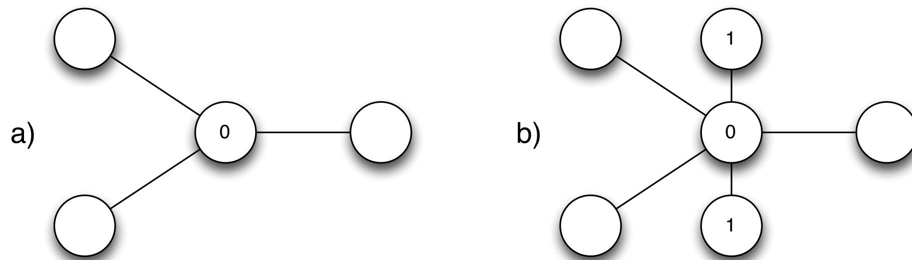

Figura 2.1: a) La puerta AND. b) La puerta OR.

Para evitar problemas con el flujo de información, es decir, para simular un grafo dirigido (el circuito) mediante un grafo no dirigido, la Figura 2.2 (a) muestra la construcción de un *diodo* el cual permite el paso de la información en un solo sentido: si el vértice de la izquierda se activa en un cierto step y el de la derecha está pasivo, entonces el vértice de la derecha pasará activo en 4 steps. En cambio, si el vértice de la izquierda es pasivo derecha es activo, el vértice de la izquierda se mantendrá en su estado. Simplificaremos la notación de un *diodo* usando una flecha (Figura 2.2 (b))

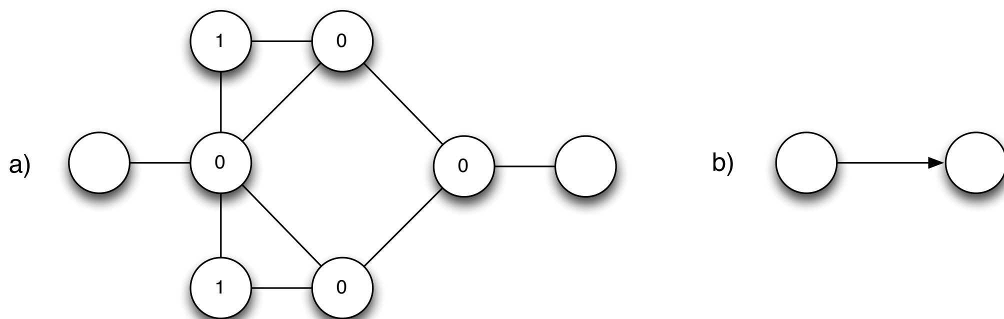

Figura 2.2: a) El diodo. b) Símbolo para representar un diodo.

Modificamos las puertas originales, de modo de tener definiciones más precisas de inputs y outputs. La Figura 2.3 muestra construcciones de las puertas AND y OR con diodos.

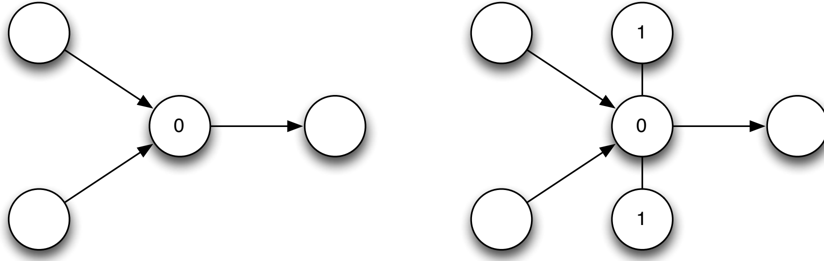

Figura 2.3: Puertas AND y OR con diodos.

Recordemos que en **AM2CVP**, cada puerta del circuito original debe tener fan-out exactamente 2, por lo que usamos las puertas OR "al revés" para multiplicar la información del output (Figura 2.4).

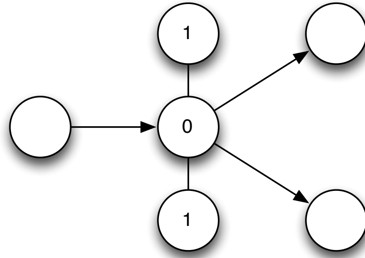

Figura 2.4: Puerta OR usada "al revés" para multiplicar la información de un output.

Luego, dado ϕ un circuito booleano que cumple todas las restricciones de **AM2CVP**, I un input del circuito y s una puerta output, podemos construir un grafo no dirigido G simulando las puertas lógicas con los gadgets recién explicados. Definiendo la configuración inicial x como la que tiene activos a los vértices necesarios para los gadgets, activos a los vértices que simulan puertas lógicas con input Verdadero y pasivos al resto, y tomando como v al vértice que simula la puerta lógica s, tenemos que (G, x, v) pertenece a **PER** si y sólo si (ϕ, I, s) pertenece a **AM2CVP**.

Esta reducción puede ser hecha por una máquina de Turing en espacio logarítmico con respecto al tamaño de la entrada. En efecto, leyendo la entrada (un circuito monótono) de tamaño n, la maquina sólo debe determinar que gadget corresponde a cada puerta (una cantidad constante de caracteres $\mathcal{O}(1)$), y luego determinar la conectividad de cada puerta (dos punteros para representar los inputs y dos para representar los outputs $\mathcal{O}(\log(n))$). Luego, toda la reducción requiere espacio $\mathcal{O}(\log(n))$. □

Observación Notemos que en la construcciones de la puerta OR y del diodo, necesitamos vértices con grado 5. Vamos a probar que hacer estos gadgets con vértices con grado menor que 5 no es posible, a menos que $\mathbf{P} = \mathbf{NC}$.

2.1.2. Caso $\Delta(G) \leq 4$

Probaremos ahora la segunda parte del Teorema 2.1, pero antes necesitamos algunas definiciones. Sea x una configuración de G, entonces $G_x[0] = (V[0], E[0])$ será el subgrafo de G inducido por los vértices pasivos de G dados por x:

$$\begin{aligned} V[0] &= \{u \in V \mid x_u(0) = 0\} \\ E[0] &= \{uw \in E \mid u, w \in V[0]\} \end{aligned}$$

Definición 2.2 *Una alianza de $G = (V, E)$ es un subconjunto de vértices $A \subset V$ donde cada elemento de A tiene al menos tantos vecinos en A como en $V \backslash A$, i.e. para cada $v \in A$, $|N(v) \cap A| \geq |N(v) \cap (V \backslash A)|$.*

Definición 2.3 *Dada una configuración x, decimos que un vértice v es estable para x si $y_v = 0$ para todo $y \in \{x(t) : t \geq 0\}$.*

Observación Un vértice es estable para x si y sólo si pertenece a una alianza A en $G_x[0]$.

Observación En el caso que $\Delta(G) \leq 4$, cualquier ciclo es una alianza, por lo tanto cualquier ciclo en $G_x[0]$ será estable para x.

Notemos que aún cuando G sea conexo, $G_x[0]$ puede que no lo sea. Dado $v \in V[0]$, sea $G[0, v]$ la componente conexa de v en $G_x[0]$,

Lema 2.4 *Sea G con $\Delta(G) \leq 4$ y x una configuración de G. Un vértice v es estable para x si y sólo si existe un camino P en $G_x[0, v]$ que contiene a v y, si u es uno de los extremos de P, entonces:*

(1) u pertenece a un ciclo en $G_x[0]$, o bien
(2) $d_G(u) \leq 2$.

Demostración. Supongamos primero que v es estable. Sabemos que a lo más luego de $T = |V|$ steps, la trayectoria dada por x entra en un punto fijo, esto es $x(T + t) = x(t)$, $\forall t \geq 0$. Como v es estable, tenemos que $x_v(T) = 0$. Consideremos $y = x(T)$, y notemos que $G_y[0, v]$ es un subgrafo de $G_x[0, v]$ que corresponde a todos los vértices de G estables para x. Sea P el camino más largo en $G_y[0, v]$ que contiene a v y sea u uno de sus extremos. Si u tiene a lo más un vecino en $G_y[0, v]$, entonces $d_G(u) \leq 2$, ya que de lo contrario no sería estable. Si por otra parte u tiene al menos dos vecinos en $G_y[0, v]$, como P es el camino más largo que contiene a v y u es uno de sus extremos, entonces ambos vecinos de u deben pertenecer a P. Entonces u pertenece a un ciclo en $G_y[0]$ y luego en $G_x[0]$.

Probemos la recíproca. Sea P un camino en $G_x[0, v]$, con $v \in P$ y sus extremos que satisfacen (1) o (2). Llamemos A al conjunto de vértices de P unido con los ciclos que contienen a los extremos de P que satisfagan (1). Como $\Delta(G) \leq 4$, tenemos que A es una alianza. En efecto, cada vértice interior de P tiene al menos dos vecinos en P, y por lo tanto en A; por otro lado, un extremo de P que satisface (1) tiene en A, por definición, al ciclo que lo contiene, y por lo tanto también tiene al menos dos extremos en A; finalmente si un

extremo de P que satisface (2), entonces tiene al menos a la mitad de sus vecinos en A, ya que tiene al menos al vecino que pertenece a P y tiene a lo más dos. La demostración entonces sigue de la Observación anterior. □

En la introducción mencionamos que encontrar componentes conexas y biconexas de un grafo dado se puede hacer en **NC** . En efecto, encontrarlas requiere a lo más tiempo $\mathcal{O}(\log^2 n)$ usando un total de $\mathcal{O}(n^2)$ procesadores. La demostración de estas proposiciones, que consisten en la descripción de estos algoritmos, se puede encontrar en [13]. Para poder hacer uso de estos resultados, necesitaremos conocer los inputs y los outputs de estos algoritmos, que están detallados en los Algoritmos 1 y 2.

Algoritmo 1 Componentes conexas

Entrada: La matriz de adyacencia A de $n \times n$, representando un grafo no dirigido.

Salida: Un arreglo D de tamaño n, tal que $D(i)$ es igual al vértice más pequeño en la componente conexa que contiene a i.

Algoritmo 2 Componentes biconexas

Entrada: La matriz de adyacencia A de $n \times n$, representando un grafo no dirigido conexo.

Salida: Un arreglo B tal que $B(e) = B(g)$ si y sólo si e y g están en la misma componente biconexa.

Con esto, estamos listos para demostrar la segunda parte del Teorema 2.1.

Demostración. (del Teorema 2.1 2.)

Sea x una configuración de G, y sea v un vértice de G pasivo para x. A partir del Lema 2.1.2, sabemos que para decidir **PER** es suficiente determinar si v pertenece a un camino en $G_x[0, v]$ cuyos extremos satisfacen las propiedades (1) o (2) del lema . Esto se debe hacer eficientemente en paralelo, por lo que no podemos usar métodos de búsqueda tradicionales, como DFS.

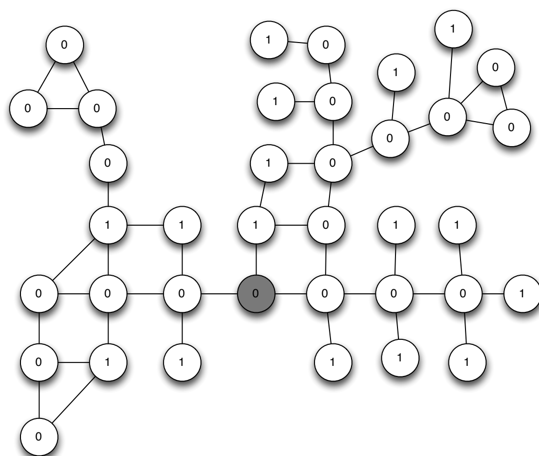

Figura 2.5: Ejemplo de una entrada del algoritmo para decidir **PER**. Decidimos sobre el vértice en gris v

Sea A la matriz de adyacencia de G. Podemos obtener $G_x[0,v]$ usando el Algoritmo 1 y el algoritmo de sumas de prefijos. Primero, para construir $G_x[0]$ calculamos su matriz de adyacencia $A[0]$ a partir de A y x: tomando $\overline{x} = 1 - x$, y luego aplicándole el algoritmo de sumas de prefijos, obtenemos un vector s tal que s_n será el número de vértices pasivos para x, y para todo i, j tal que $x_i = x_j = 0$, si $A_{ij} = 1$, entonces $A[0]_{s_i, s_j} = 1$.

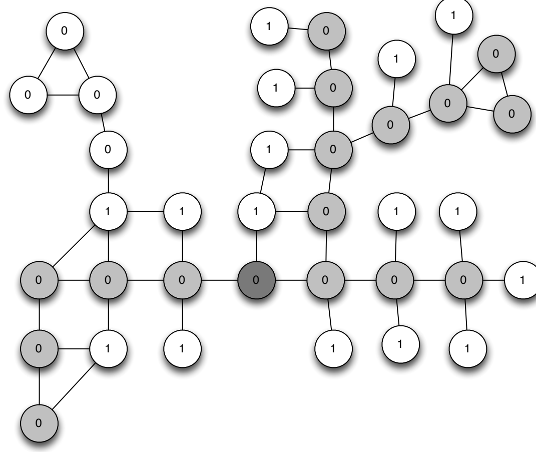

Figura 2.6: Los vértices en gris representan a la componente conexa que contiene al vértice sobre el que decidimos, para el ejemplo de la Figura 2.5

Luego, usando el Algoritmo 1, calculamos las componentes conexas de $G_x[0]$ en tiempo $\mathcal{O}(\log^2(n))$ usando $\mathcal{O}(n^2)$ procesadores. Con el arreglo que retorna el Algoritmo 1, construimos $A[0,v]$, la matriz de adyacencia de $G_x[0,v]$. Notemos que si v está aislado en $G[0,v]$, entonces automáticamente no es estable, ya que G es conexo. Supongamos entonces que v no está aislado en $G_x[0,v]$.

Una componente biconexa que contiene más de dos vértices tiene la propiedad que cualquier par de vértices que pertenezcan a ella están en el mismo ciclo, y recíprocamente cualquier par de vértices que están en un ciclo pertenecen a una componente biconexa. Sigue que a partir del arreglo retornado por el Algoritmo 2 podemos obtener los vértices de $G_x[0,v]$ que satisfacen la condición (1) del Lema 2.1.2.

Para obtener los vértices de $G[0,v]$ que satisfacen (2), podemos obtener a partir de A un vector D, de dimensión n, donde $D_u = d_G(u)$. Esto se puede hacer en tiempo $\mathcal{O}(\log(n))$ usando $\mathcal{O}(n)$ procesadores con el algoritmo de sumas de prefijos. Luego, cualquier vértice u de $G_x[0,v]$ tal que $D_u \leq 2$ satisface (2).

Para determinar si v pertenece a un camino cuyos extremos satisfacen las condiciones anteriores, construiremos a partir de $G_x[0,v]$ un nuevo grafo $\overline{G} = (\overline{V}, \overline{E})$ que le añade a G un nuevo vértice que llamamos ∞, donde:

$$\begin{aligned} \overline{V} &= V[0,v] \cup \{\infty\}, \\ \overline{E} &= E[0,v] \cup \{u\infty \mid u \in V[0,v] \text{ que satisface (1) o (2)}\}. \end{aligned}$$

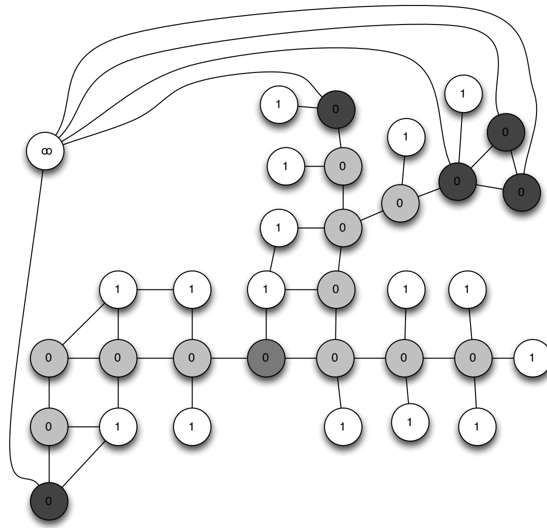

Figura 2.7: Los vértices que cumplen la propiedad (1) o (2) del Lema 2.1.2 en el ejemplo. Conectamos dichos vértices a un nuevo vértice que llamamos ∞.

Notemos que v es estable si y solo si existen dos caminos diferentes de v a ∞. Entonces, v es estable si existe un ciclo en $\overline{G}$ que contenga simultáneamente a v y ∞. En conclusión, los últimos pasos del algoritmo consisten en calcular las componentes biconexas de $\overline{G}$, y decidir si v y ∞ están en la misma componente.

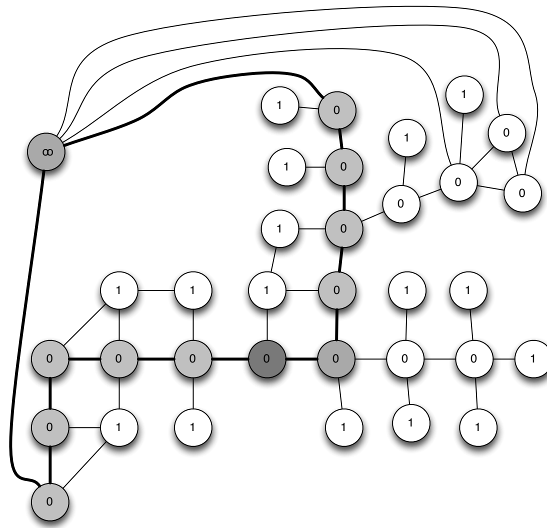

Figura 2.8: En el grafo obtenido al agregar el ∞, nuevamente buscamos componentes biconexas y decidimos si hay alguna que contenga a ∞ y v.

El resultado es el Algoritmo 3.

Algoritmo 3 PER

Entrada: La matriz A de $n \times n$ que corresponde a la matriz de adyacencia de un grafo no dirigido $G = (V, E)$, una configuración x de G y un vértice $v \in V$.

Salida: Aceptar si $\{G, x, v\}$ pertenece a **PER** con la regla Bootstrap Percolation, y rechazar de lo contrario.

1: Calcular $A[0]$, la matriz de adyacencia de $G[0]$ usando x y el algoritmo de sumas de prefijos.
2: Calcular $A[0, v]$, la matriz de adyacencia de $G[0, v]$, usando $A[0]$ en el Algoritmo 1.
3: **si** v está aislado en $A[0, v]$ **entonces**
4: Aceptar y salir.
5: **fin si**
6: Calcular B, la salida del Algoritmo 2 con input $A[0, v]$.
7: Calcular los ciclos en $A[0, v]$ a partir de B y almacenarlos en un arreglo C donde $C_u = 1$ si y sólo si u pertenece a un ciclo en $A[0, v]$.
8: Calcular D, el vector de grados de G.
9: Definir $\overline{A}$, la matriz de adyacencia de $\overline{G}$, usando $A[0, v]$, D y C.
10: Determinar si ∞ está aislado en $\overline{G}$ usando el Algoritmo 1 en $\overline{A}$
11: Calcular $\overline{B}$, la salida del Algoritmo 2 en $\overline{A}$.
12: Usando B determine si v y ∞ están en un mismo ciclo. Rechazar si lo están y aceptar en caso contrario.

La correctitud del algoritmo se obtiene como sigue: Si v es estable, entonces por el Lema 2.1.2, v pertenece a un camino P en $G_x[0, v]$ cuyos extremos satisfacen (1) o (2). Por definición de $\overline{G}$, los extremos de P están conectados a ∞, entonces, existe un ciclo C que contiene a v y ∞. Recíprocamente, si hay un ciclo C en $\overline{G}$ que contiene a v y ∞, entonces, por definición de $\overline{G}$ tenemos que $C - \{\infty\}$ es un camino que contiene a v y cuyos extremos satisfacen las condiciones (1) o (2) del Lema 2.1.2, y luego v es estable.

En el Apéndice A haremos un análisis profundo de la complejidad de éste algoritmo, con lo que concluiremos que se puede hacer en una máquina PRAM en tiempo $\mathcal{O}(\log^2(n))$ con $\mathcal{O}(n^4)$ procesadores, y por lo tanto con la regla Bootstrap Percolation **PER** está en **NC** cuando $\Delta(G) \leq 4$. $\square$

2.2. Problemas relacionados

Aplicaremos ahora los resultados anteriores a otras reglas de transición global, similares a Bootstrap Percolation.

2.2.1. Bootstrap Percolation no estricto

Considere la regla de transición global dada por la siguiente función de transición para cada vértice i:

$$f_i(x) = \begin{cases} 1 & \text{si} \quad x_i = 1, \\ 1 & \text{si} \quad \sum_{j \in N(i)} x_j \geq \frac{|N(i)|}{2} \quad \text{y} \quad x_i = 0, \\ 0 & \text{si} \quad \sum_{j \in N(i)} x_j < \frac{|N(i)|}{2} \quad \text{y} \quad x_i = 0. \end{cases}$$

Es decir, en este caso los vértices pasivos pasan a ser activos si al menos la mitad de sus vecinos es activo, y de ahí en adelante permanece activo. A esta función de transición global la llamamos Bootstrap Percolation *no estricto*.

Para esta regla tenemos un resultado similar al Teorema 2.1:

Teorema 2.5 *Para Bootstrap Percolation no estricto:*

1. *En la familia de grafos G tales que $\Delta(G) \geq 4$, el problema* **PER** *es* ***P****-Completo*
2. *En la familia de grafos G tales que $\Delta(G) \leq 3$, el problema* **PER** *está en* ***NC*** .

Para la demostración de este Teorema, vamos a necesitar una versión del Lema 2.1.2 pero ajustada a la regla no estricta.

Lema 2.6 *Sea G con $\Delta(G) \leq 3$ y una configuración x de G. Un vértice v es estable para x con la regla Bootstrap Percolation no estricta si y sólo si existe un camino P en $G_x[0]$ que contiene a v y, si u es uno de los extremos de P, entonces*

1. *u pertenece a un ciclo en $G[0]$, o bien,*
2. *$d_G(u) = 1$.*

La demostración de este Lema es análoga a la demostración del Lema 2.1.2 pero tomando en cuenta en este caso que si u es un extremo de un camino maximal en $G_x[0]$ que no pertenece a un ciclo, entonces u tiene a lo más un vecino inicialmente pasivo. Como la mayoría de vecinos para pasar de pasivo a activo no es estricta, para que u sea estable se necesitará que $d_G(u) = 1$.

Demostración. (del Teorema 2.5)

1. La Figura 2.9 muestra los gadgets usados para simular un circuito monótono de un modo similar que en la demostración del Teorema 2.1. La demostración de esta parte entonces se sigue directamente de la del Teorema 2.1 (1). Notar que en este caso para construir los gadgets solo requerimos grado 4.
2. Para la segunda parte, considerando el Lema 2.6, modificamos el Algoritmo 3 solo en la definición de $\overline{A}$, donde conectamos con ∞ a los vértices en $G_x[0, v]$ que tengan grado 1 o que pertenezcan a un ciclo en $G[0, v]$. El resto de la demostración se sigue de la del Teorema 2.1 (2).

□

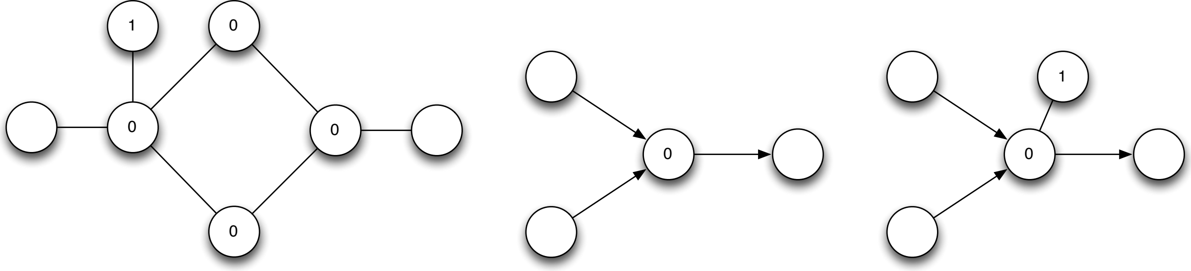

Figura 2.9: De izquierda a derecha: El diodo en el caso no estricto, las puertas AND y OR para la misma regla.

2.2.2. Umbral θ

Sea $\theta > 0$ y $G = (V, E)$ tal que el grado mínimo $\delta(G)$ es mayor o igual a θ, es decir $\forall v \in V d_G(v) \geq \theta$. Consideremos la función de transición global dada por la siguiente función de transición para cada vértice i:

$$f_i(x) = \begin{cases} 1 & \text{si} \quad x_i = 1, \\ 1 & \text{si} \quad \sum_{j \in N(i)} x_j > \theta \quad \text{y} \quad x_i = 0, \\ 0 & \text{si} \quad \sum_{j \in N(i)} x_j \leq \theta \quad \text{y} \quad x_i = 0. \end{cases}$$

Es decir, los vértices pasivos pasan a activos si tienen más de θ vecinos activos y de ahí en adelante se mantienen activos. Llamamos a esta función la *Umbral* θ.

Al igual que en el caso anterior, tenemos un resultado similar al Teorema 2.1:

Teorema 2.7 *Para la regla Umbral θ:*

1. *En la familia de grafos G tales que $\Delta(G) \geq \theta + 3$, el problema* **PER** *es **P**-Completo*
2. *En la familia de grafos G tales que $\Delta(G) \leq \theta + 2$, el problema* **PER** *está en **NC** .*

Lema 2.8 *Sea G con $\Delta(G) \leq \theta + 2$ y una configuración x de G. Un vértice v es estable para x con la regla Bootstrap Percolation no estricta si y sólo si existe un camino P en $G_x[0]$ que contiene a v y, si u es uno de los extremos de P, entonces*

1. *u pertenece a un ciclo en $G[0]$, o bien,*
2. *$d_G(u) \leq \theta + 1$.*

Demostración. Notemos que un vértice pasivo pasará a activo si tiene al menos $\theta + 1$ vecinos activos. Entonces, si $\Delta(G) \leq \theta + 2$ tendremos que un ciclo en $G[0]$ será estable, ya que cada vértice del ciclo tendrá a lo más θ vecinos activos. El resto se sigue análogamente de la demostración del Lema 2.1.2. □

Demostración. (del Teorema 2.7)

1. Como en los otros casos, vamos a simular un circuito monótono. Como $\delta(G) \geq \theta$ vamos a modificar los gadgets del Teorema 2.1 (1), de modo de obtener vértices con grado al menos θ. El truco es tomar los gadgets del Teorema 2.1 (1) y conectar cada vértice con otros p vértices activos conectados entre sí, donde p es algún número conveniente (obviamente dependiente de θ). En otras palabras, conectamos cada vértice de los gadgets con cada vértice de un grafo completo de vértices inicialmente activos.

   La Figura 2.10 muestra como simplificar la notación. Representamos un grafo completo K y las $|K|$ conexiones por un vértice rectangular con el número $|K|$ adentro. Usando

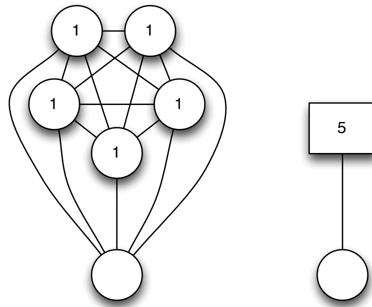

Figura 2.10: Representamos el grafo completo K, en este caso K_5, y las $|K|$ conexiones con un vértice rectangular con el número $|K|$ adentro y un solo arco.

   esto, es fácil construir nuevos gadgets, ver Figura 2.11. Notar que en la construcción de la puerta OR y el diodo, el grado requerido es $\theta + 3$.

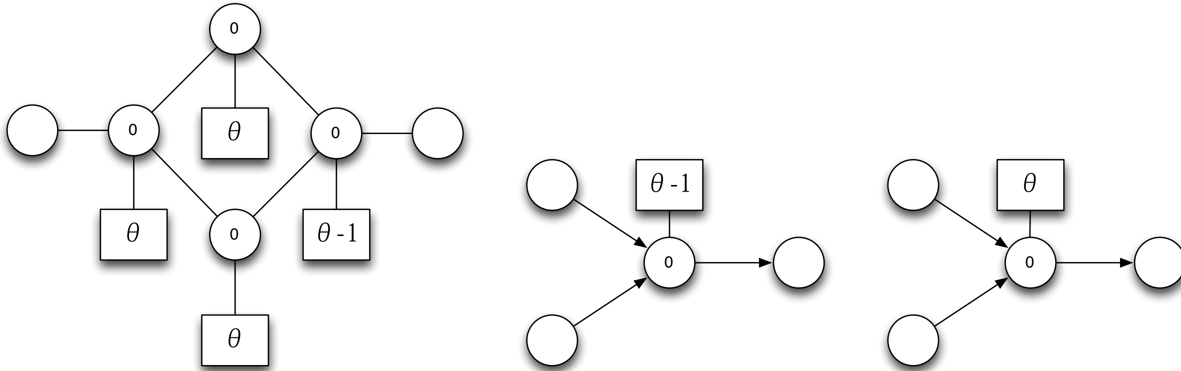

Figura 2.11: De izquierda a derecha: Diodo para la regla Umbral θ, las puertas AND y OR para la misma regla.

2. Para la segunda parte, considerando el Lema 2.8, modificamos el Algoritmo 3 solo en la definición de $\overline{A}$, donde conectamos con ∞ a los vértices en $G_x[0, v]$ que tengan grado menor que $\theta + 1$ o que pertenezcan a un ciclo en $G[0, v]$. El resto de la demostración se sigue de la del Teorema 2.1 (2).

□

Capítulo 3

Mayoría estricta: caso plano

Sea $G = (V, E)$ un grafo finito, simple, conexo no dirigido, donde V es el conjunto de vértices y E el conjunto de arcos. Consideremos la siguiente función de transición:

$$f_i(x) = \begin{cases} 0 & \text{si} \quad \sum\limits_{j \in N(i)} x_j < \dfrac{|N(v)|}{2} \\ x_i & \text{si} \quad \sum\limits_{j \in N(i)} x_j = \dfrac{|N(v)|}{2} \\ 1 & \text{si} \quad \sum\limits_{j \in N(i)} x_j > \dfrac{|N(v)|}{2} \end{cases}$$

En otras palabras, un vértice toma el estado de la mayoría de sus vecinos, y en caso de empate mantiene su estado. A la función de transición global asociada a $(f_i \; : \; i \in V)$ la llamamos la *regla de la mayoría estricta.*

La complejidad computacional de esta regla fue estudiada por C. Moore en [19], donde se demuestra que en el lattice d-dimensional (es decir, en caso que la red de autómatas es un autómata celular) el problema **PER** es **P**-Completo si $d \geq 3$. Para $d = 2$ el problema queda abierto, conjeturando que en ese caso estaría en **NC**.

En nuestro caso estudiaremos la regla de la mayoría estricta cuando restringimos el grafo a ser planar. Recordemos que un grafo es planar si se puede incrustar en $\mathbb{R}^2$ sin que los arcos se crucen. Para esta familia de grafos, demostraremos que el problema es **P**-Completo, lo que de alguna forma nos dice que la planaridad no es suficiente para demostrar la conjetura de Moore, y por lo tanto habría que tomar restricciones estrictas, como limitar el grado máximo, por ejemplo. Tenemos entonces que el resultado principal de este capítulo es el siguiente teorema.

Teorema 3.1 *En la familia de grafos planares y para la regla de la mayoría estricta, el problema* **PER** *es* **P***-Completo.*

Recordemos que un problema es **P**-Completo si y sólo si está en **P** y es **P**-Duro para reducciones a espacio logarítmico. Dividiremos entonces esta demostración en tres partes. Primero probaremos que **PER** está en **P** para la regla de la mayoría estricta. Esto lo haremos definiendo una Red Neuronal equivalente al autómata con la mayoría estricta, y usando los resultados de Goles [9] para asociar a la dinámica una función de Lyapunov decreciente, lo cual implicará que la dinámica entra en un ciclo.

A continuación probaremos la **P**-Dureza para **PER** con la regla de la mayoría estricta en el caso general (cualquier grafo), procediendo de manera similar a la demostración de **P**-Complitud del capítulo anterior. Finalmente probaremos la **P**-Dureza en el caso que restrinjamos a grafos planos. La clave aquí será usar que ahora los nodos activos pueden pasar a pasivos para construir un *gadget* que nos permita atravesar información sin cruzar cables.

Cabe destacar que la **P**-Dureza en el caso de grafos planos implica la **P**-Dureza en el caso general, es más, el resultado de Christopher Moore en [19] que dice que la regla de la mayoría estricta es **P**-Completa también implica la **P**-Dureza en el caso general. Aún así, haremos nuestra propia demostración en el caso general, debido a que nos permitirá explicar mejor la demostración del caso plano.

La demostración entonces se sigue directamente de tres lemas, expuestos en las siguientes secciones.

3.1. Mayoría estricta está en P

Lema 3.2 *Para la regla de la mayoría estricta,* **PER** *está en* **P**.

Demostración. Sean $G = (V, E)$, $v \in V$ y una configuración x con $x_v = 0$.

Para probar que **PER** está en **P**, vamos a probar que a lo más en una cantidad polinomial de pasos la trayectoria obtenida a partir de una configuración x llega a un punto fijo o entra a un ciclo, en otras palabras existen $t_1, t_2 = \mathcal{O}(|V|^c)$, con $c > 0$ tales que $x(t_1) = x(t_2)$.

En caso que esto se cumpla, la trayectoria de cualquier configuración será finita (de cardinal polinomial en el tamaño de la entrada) y por lo tanto para cualquier configuración inicial tendremos un número polinomial de posibles configuraciones en su trayectoria, y entonces simplemente construimos la trayectoria simulando la regla hasta que pasemos el primer ciclo o punto fijo.

Para demostrar esto usaremos la notación y resultados de Goles en [11, 8, 9]. Recordemos a $\eta : \mathbb{R} \to \{0, 1\}$, la función umbral:

$$\eta(x) = \begin{cases} 1 & \text{si} \quad x \geq 0 \\ 0 & \text{si} \quad x < 0 \end{cases}$$

podemos caracterizar la regla de la mayoría usando esta función:

$$x_v(t+1) = \begin{cases} \eta\left(\sum_{u\in N(v)} x_u(t) + x_v(t) - \frac{|N(v)|+1}{2}\right) & \text{si} \quad |N(v)| \text{ es par} \\ \eta\left(\sum_{u\in N(v)} x_u(t) - \frac{|N(v)|}{2}\right) & \text{si} \quad |N(v)| \text{ es impar} \end{cases}$$

Si $\overline{\eta} : \mathbb{R}^{|V|} \to \{0,1\}^{|V|}$ es la función umbral multidimensional, i.e. $\overline{\eta}((x_i)_{i\in V}) = (\eta(x_i))_{i\in V}$, entonces $x(t+1) = \overline{\eta}(Ax(t) - b)$ donde $A = (a_{uv})$ es una matriz cuadrada de orden $|V|$ y b es un vector de tamaño$|V|$ con

$$a_{uv} = \begin{cases} 1 & \text{si} \quad uv \in E \\ 1 & \text{si} \quad i = j \ \wedge \ |N(v)| \text{ es par} \\ 0 & \text{en otro caso} \end{cases}$$

$$b_v = \begin{cases} \frac{|N(v)|+1}{2} & \text{si} \quad |N(i)| \text{ es par} \\ \frac{|N(v)|}{2} & \text{si} \quad |N(i)| \text{ es impar} \end{cases}$$

Usando esta caracterización, llamemos $T(x)$ la trayectoria obtenida a partir de una configuración x y la regla de la mayoría estricta, y definamos el siguiente funcional para $t \geq 1$:

$$E(x(t)) = \sum_{v\in V}(2b_v - \sum_{u\in V} a_{vu})(2x_v(t) + 2x_v(t-1) - 2) - (2x_v(t) - 1)\sum_{u\in V} a_{vu}(2x_u(t-1) - 1),$$

el cual satisface que

$$\Delta_t E = E(x(t)) - E(x(t-1)) = -4\sum_{v\in V}(x_v(t) - x_v(t-2))(\sum_{u\in V} a_{vu}x_u(t-1) - b_v) \leq 0.$$

Por consiguiente, si $\{x(t) : t \in [t_1, t_2]\}$ es un ciclo, entonces $E(x(t))$ es necesariamente constante para todo $t \in [t_1, t_2]$. De hecho, como para todo $x \in \{0,1\}^{|V|}$ y $v \in V$ tenemos que $\sum_{u\in V} a_{vu}x_u \neq b_v$, entonces los ciclos son solo puntos fijos y/o ciclos de largo dos, i.e $t_2 = t_1$ o $t_2 = t_1$.

Entonces tenemos que para toda condición inicial, la trayectoria calculada a partir de x necesariamente entra a un ciclo de largo a lo más dos (considerando un punto fijo como un ciclo de largo 1). Debemos probar ahora que esto ocurre luego de a lo más una cantidad de pasos polinomial en el tamaño del grafo (número de vértices). Definimos el *largo transiente* de una configuración x como el paso en que por primera vez se entra en un ciclo:

$$\tau(x) = \min\{t \geq 0 \mid |T(x(t))| \leq 2 \ , \ x(t) \in T(x)\}$$

y el largo transiente de la red es el más grande de estos valores:

$$\tau(A, b) = \max\{\tau(x(0)) : x(0) \in \{0,1\}^{|V|}\}.$$

También, por un resultado de [11, 8, 9], tenemos que $\tau(A, b) \leq |V|^3$ para la regla de la mayoría estricta. Esto nos dice que en a lo más n^3 pasos la trayectoria entra en un ciclo de largo 2, Como cualquier iteración de esta regla se puede simular en tiempo polinomial, tenemos que **PER** está en **P**. □

3.2. La mayoría estricta es P-Completa

Lema 3.3 *Para la regla de la mayoría estricta,* **PER** *es* **P***-Duro.*

Demostración. Para probar que para la regla de la mayoría estricta **PER** es **P**-Duro, vamos a reducir **AM2CVP**, la versión restringida del **CVP**, a **PER**. Como **AM2CVP** es **P**-Completo, haremos una reducción log-space para así obtener que **PER** es **P**-Duro.

Para reducir **AM2CVP** a **PER** vamos a razonar como en el capítulo anterior, es decir vamos a construir, dado un circuito, un grafo que por una parte simule sus puertas, donde los vértices activos representaran el valor *Verdadero* y los vértices pasivos representaran el valor *Falso*; y por otro lado, sea capaz de mantener un cierto "flujo de información", simulando un grafo dirigido.

Para poder simular un grafo dirigido a partir de un grafo no dirigido, construiremos un *gadget* que nos permitirá pasar la información en un solo sentido, que llamaremos *diodo* (Figura 3.1).

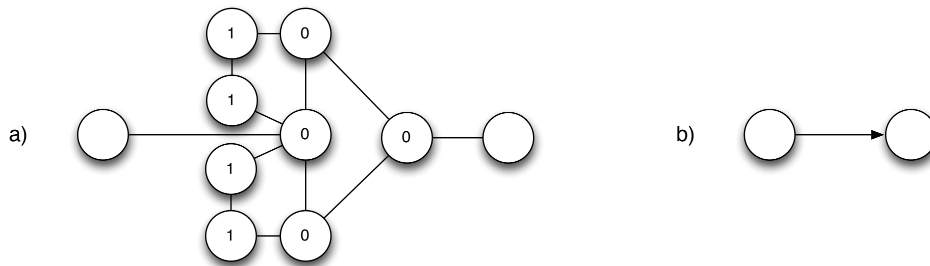

Figura 3.1: a) Un diodo. b) Representación simplificada de un diodo.

Si el vértice a la izquierda está activo y el de la derecha pasivo, entonces luego de 4 pasos el vértice de la derecha pasará a estar activo; si por otro lado el vértice de la izquierda está pasivo y el de la derecha activo, entonces la información no logrará atravesar hasta el vértice de la izquierda, dado que el vértice que está conectado inmediatamente después del vértice de más a la derecha tiene grado tres, con dos nodos pasivos, por lo que se mantendrá pasivo.

Por lo tanto, mediante el uso de diodos podemos asegurar que la información va a recorrer nuestro grafo en un sólo sentido. Por otro lado, como el circuito es monótono sólo necesitamos simular las puertas AND y OR, las cuales quedan determinadas por los *gadgets* de la Figura 3.2. Notar que las flechas dirigidas representan diodos.

La puerta AND es simulada por un vértice de grado tres, donde los diodos entrantes y salientes corresponderán a los inputs y outputs de la puerta lógica. Al tener grado tres, éste pasará de pasivo a activo sólo si sus dos inputs son activos, lo que implica que este gadget puede simular una puerta AND. Similarmente, la puerta OR es simulada por un vértice de grado cinco, donde al igual que en la puerta AND los diodos entrantes y salientes corresponden a los inputs y outputs de la puerta lógica, pero además lo acompañan dos nodos activos y que por su grado permanecerán activos siempre. De este modo, éste vértice pasará de activo

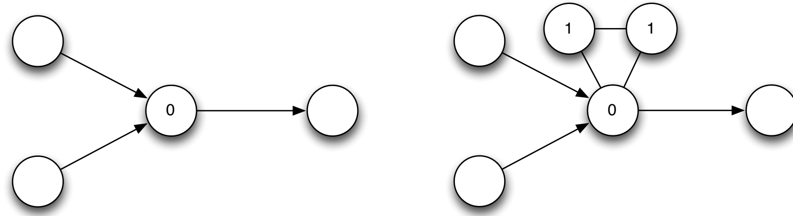

Figura 3.2: a) La puerta AND.   b) La puerta OR.

a pasivo si alguno de los dos inputs es activo, por lo que este gadget puede simular una puerta OR.

Recordemos que en nuestra versión restringida del **CVP**, cada puerta lógica tiene exactamente dos inputs y dos outputs. Completamos entonces nuestros gadgets conectando su output a un gadget OR "al revés", logrando así multiplicar los outputs (Figura 3.3).

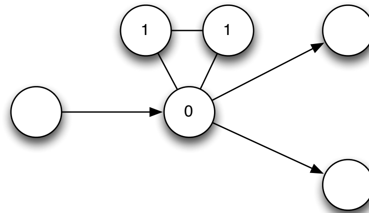

Figura 3.3: Puerta OR "al revés" para multiplicar la información.

Finalmente, conectamos las puertas con cables (Figura 3.4) que corresponden a vértices pasivos de grado 3 y un vecino activo, por lo que permanecen pasivos a menos que alguno de sus vecinos se vuelva activo.

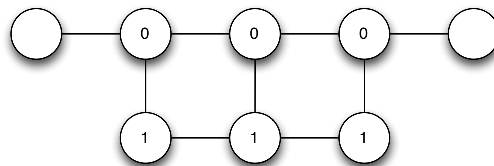

Figura 3.4: Un cable de largo 3.

Luego, dado ϕ un circuito booleano que cumple todas las restricciones de **AM2CVP**, I un input del circuito y s una puerta output, podemos construir un grafo no dirigido G simulando las puertas lógicas con los gadgets recién explicados. Definiendo la configuración inicial x como la que tiene activos a los vértices necesarios para los gadgets, activos a los vértices que simulan puertas lógicas con input Verdadero y pasivos al resto, y tomando como v al vértice que simula la puerta lógica s, tenemos que (G, x, v) pertenece a **PER** si y sólo si (ϕ, I, s) pertenece a **AM2CVP**.

Esta reducción puede ser hecha por una máquina de Turing en espacio logarítmico con respecto al tamaño de la entrada. En efecto, leyendo la entrada (un circuito monótono) de tamaño n, la maquina sólo debe determinar que gadget corresponde a cada puerta (una cantidad constante de caracteres $\mathcal{O}(1)$), y luego determinar la conectividad de cada puerta (dos punteros para representar los inputs y dos para representar los outputs $\mathcal{O}(\log(n))$). Luego, toda la reducción requiere espacio $\mathcal{O}(\log(n))$. □

Aún cuando las construcciones del Lema 3.3 son todas planas, el circuito que intentamos simular puede no serlo, y por lo tanto al hacer la reducción anterior el grafo obtenido puede no resultar plano. Por otra parte como dijimos en el primer capítulo, en [5] tenemos un algoritmo **NC** que resuelve la restricción del **MCVP** a grafos planos. Lo que haremos entonces es construir un grafo plano que simule un circuito monótono arbitrario, para lo cual necesitaremos introducir nuevos *gadgets*.

3.3. Caso plano

Lema 3.4 *Para la familia de grafos planos y la regla de la mayoría estricta,* **PER** *es* **P***-Duro.*

Demostración. En este caso la **P**-Dureza también será obtenida reduciendo un circuito monótono, pero en este caso debemos garantizar que el grafo resultante es plano. Para asegurarlo, nuestro algoritmo no solo construirá un grafo que simule las puertas lógicas del circuito, sino que además irá asignando posiciones a cada vértice en el plano $X - Y$, de modo de obtener un embedding (incrustación) planar.

Sea (ϕ, I, s) una instancia de nuestro caso especial del **CVP** , que llamamos **AM2CVP**, esto es, ϕ es un circuito monótono, fan-in 2, fan-out 2 y síncrono, I es una asignación de valores de la entrada del circuito y s una salida, que queremos verificar si en algún momento cambia su valor de verdad. Además, para este caso será crucial recordar que el circuito está ordenado lexicográficamente, es decir, las puertas están enumeradas de 1 hasta n, donde n es el numero total de puertas del circuito, y la enumeración es de tal forma que si n_1 y n_2 son enumeraciones de dos puertas en los *layers* l_1 y l_2 respectivamente, entonces $l_1 < l_2 \Rightarrow n_1 < n_2$. Tendremos entonces que las puertas que corresponden a las entradas del circuito (layer 0) están enumeradas desde 1 hasta n_0, las puertas del layer 1 desde $n_0 + 1$ hasta n_1 y así sucesivamente. Llamamos entonces $N_l = n_l - n_{l-1}$ al número de puertas en el layer l.

Asignaremos coordenadas en el plano $X - Y$ primero a los vértices que simularán las puertas del circuito, que por la demostración del Lema 3.3 sabemos que son: un vértice pasivo unido a dos vértices activos, si la puerta es del tipo OR; o simplemente un vértice pasivo si se trata de una puerta del tipo AND (Figura 3.2). En ambos casos le asignamos las coordenadas $(2u - 1, p_l)$, donde u es la enumeración del vértice a simular, relativa a su layer ($u \in \{1, \ldots, N_l\}$) y p_l es un valor $\mathcal{O}(N_l^2)$ que es igual para todos los vértices que simulan puertas del layer l. (Figura 3.5). En caso que la puerta sea del tipo OR, a los vértices activos (auxiliares) les asignamos coordenadas muy cercanas a $(2u-1, p_l)$ de modo que no molesten en el resto de la construcción, por ejemplo, $(2u - \varepsilon, p_l - \varepsilon)$, $(2u - \varepsilon, p_l + \varepsilon)$, para ε pequeño.

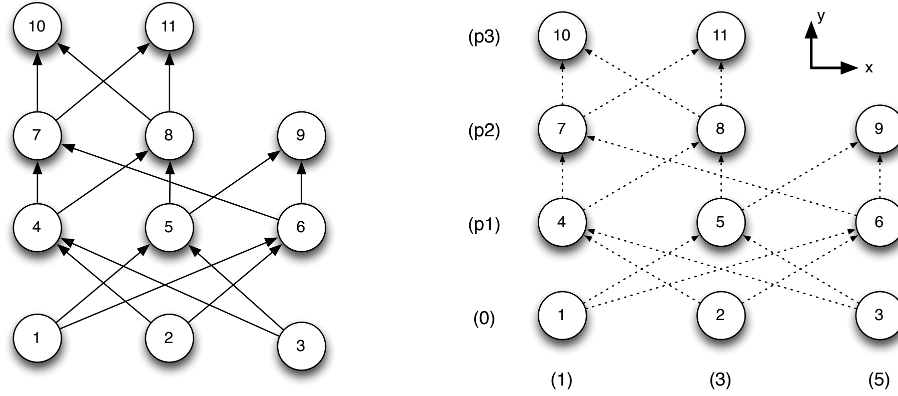

Figura 3.5: Un ejemplo de un circuito y la posición de las puertas en la incrustación plana. Notemos que la coordenada y es aún desconocida para las puertas que no están en la capa de entrada, pero tiene el mismo valor para todas las puertas de una misma capa. Obviamente si aquí simplemente dibujamos los arcos, no obtendremos necesariamente una incrustación plana.

Lo anterior, salvo el calculo de p_l se puede hacer en espacio $\mathcal{O}(\log(n))$, ya que sólo debemos almacenar la enumeración de la puerta que estamos simulando, que se codifica en espacio $\mathcal{O}(\log(n))$ y si es AND u OR $\mathcal{O}(1)$. Para obtener los valores de p_l, vamos a ejecutar la siguiente inducción: Definimos $p_0 = 0$, y luego que hemos definido p_l obtenemos p_{l+1} como sigue:

Etapa 1: Primero vamos a usar el gadget para multiplicar información (Figura 3.3), de manera de poder "separar" los dos outputs de cada puerta del layer l. Recordar que nuestro caso restringido del **CVP** nos asegura que cada puerta tiene dos salidas y dos entradas. Para un vértice que simula una puerta en el layer l, digamos numerada u, dibujamos el gadget de modo que los outputs de éste les asignamos las posiciones $(2u-1, p_l+1)$ y $(2u, p_l+1)$ y posiciones de todos los demás vértices permita que el embedding siga siendo planar, como en la Figura 3.3 pero "vertical". A los outputs del gadget los llamamos u_a y u_b (Figura 3.6).

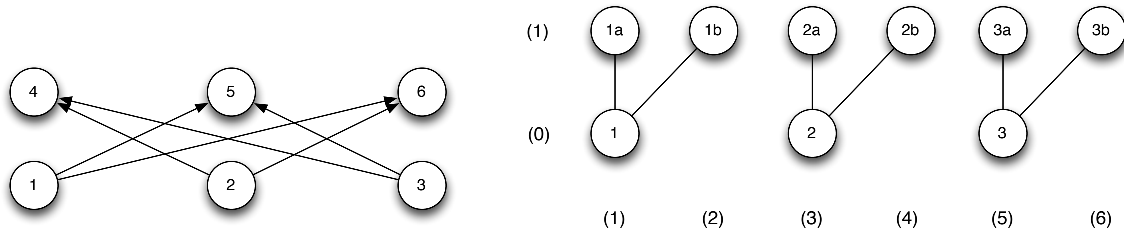

Figura 3.6: Izquierda: Los dos primeros layers del circuito en el ejemplo de la Figura 3.5. Derecha: Esquema de las posiciones de los vértices que simulan el layer input y los dos outputs correspondientes.

Este paso se puede hacer en $\mathcal{O}(\log(n))$, ya que sólo se requiere almacenar un puntero al número que representa el vértice que queremos duplicar u ($\mathcal{O}(\log(n))$) y luego dibujar el gadget ($\mathcal{O}(1)$).

Etapa 2: Sea g_1 el primer vértice en el layer $l+1$, y sean i, j sus inputs, con $i < j$. Entonces, en el grafo que simula el circuito, g_1 va a tener como inputs a i_a y a j_a. En general, si i es un input de una puerta g, en nuestro circuito simulado, el vértice que simula a g tendrá como input a i_a si g es la primera puerta en el layer $l+1$ que tiene a i como input, e i_b de otro modo.

Para mantener la planaridad, nos gustaría que los inputs de la k-ésima puerta del layer $l+1$ estuvieran en la posición $(2k-1, p_{l+1}-1)$ y $(2k, p_{l+1}-1)$. Como normalmente este no es el caso, vamos a tener que introducir un nuevo gadget que nos ayudará a intercambiar las posiciones de los vértices de modo de que al final todos los que simulan inputs de puertas en el layer $l+1$ terminen en las posiciones deseadas.

Vamos a cambiar los lugares donde aparecerían "cruces de cables", construyendo un gadget que usa "semáforos", que dejarán pasar la información en un sentido o en otro, dependiendo de la paridad del paso. La Figura 3.7 muestra este gadget. Los vértices **a** y **b** son los "inputs", **c** y **d** son los "outputs" del gadget. Para simplificar la explicación omitimos que antes y después de cada input y output del gadget conectamos diodos correspondientes.

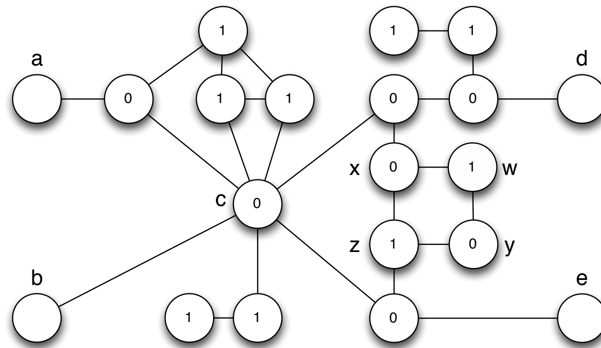

Figura 3.7: Gadget para el 'Cruce de cables' usando 'semáforos'

La clave para lo que sigue es notar que a excepción de los vértices activos estables, necesarios para construir los gadgets, cualquier vértice que simula una puerta lógica que se vuelve activo en algún step, va a comenzar a 'parpadear' entre estados activos y pasivos de ahí en adelante.

En efecto los vértices que simulan inputs luego del primer step van a pasar a ser pasivos (independiente de si inicialmente están activos o pasivos), ya que estos vértices están solo conectados al primer vértice de un diodo (del gadget para multiplicar la información, Figura 3.3). Luego, aquellos inputs inicialmente activos volverán a ser activos en el segundo step (dado que están conectados a sólo un vértice, que en el primer step pasó a ser activo) y así sucesivamente. Inductivamente en la distancia a los vértices que simulan los inputs del circuito, esta situación se repite para los demás vértices que en algún step pasan a estar activos.

Suponemos que al menos uno de los inputs del gadget de Cruce de Cables se vuelve activo en un step par, y entonces por el párrafo anterior, es pasivo en los steps impares, ya que suponemos que el circuito es síncrono y por lo tanto los inputs del gadget están en el mismo

layer. Si por el contrario alguno de los inputs se vuelve activo en un step impar, cambiamos $x_x(0) = x_y(0) = 0$ y $x_z(0) = x_w(0) = 1$, y lo que sigue es análogo.

Tenemos que $x_x(0) = x_y(0) = 1$ y $x_z(0) = x_w(0) = 0$, (Figura 3.7), entonces para todo $k \in \mathbb{N}$:

$$x_x(2k) = x_y(2k) = 1 \ , \ x_z(2k) = x_w(2k) = 0,$$

$$x_x(2k-1) = x_y(2k-1) = 0 \text{ and } x_z(2k-1) = x_w(2k-1) = 1.$$

Sea **i** el vértice conectado a **c,z** y **e** en la Figura 3.7. Notar que **i** se volverá activo si al menos dos de ellos se vuelven activos, es más, como al vértice **e** le sigue un diodo, solamente queda la posibilidad de que simultáneamente estén activos **z** y **c**, y esto sólo puede pasar si **a** está activo en algún step par. En ese caso, luego de 5 steps podemos llevar información desde **a** hasta **e**(recordar que se requiere 4 steps para pasar por un diodo). Más aún, si solo el input **a** (y no el **b**) está activo en un step par, entonces **c** será activo en los steps pares y pasivo en los steps impares, y por lo tanto el vértice que está unido a **c** y **x** nunca pasará a ser activo. Luego, podemos pasar la información de **a** hasta **e** sin 'contaminar' **d** (Figura 3.8). De manera análoga se puede transmitir información desde **b** a **d** sin 'contaminar' **e**.

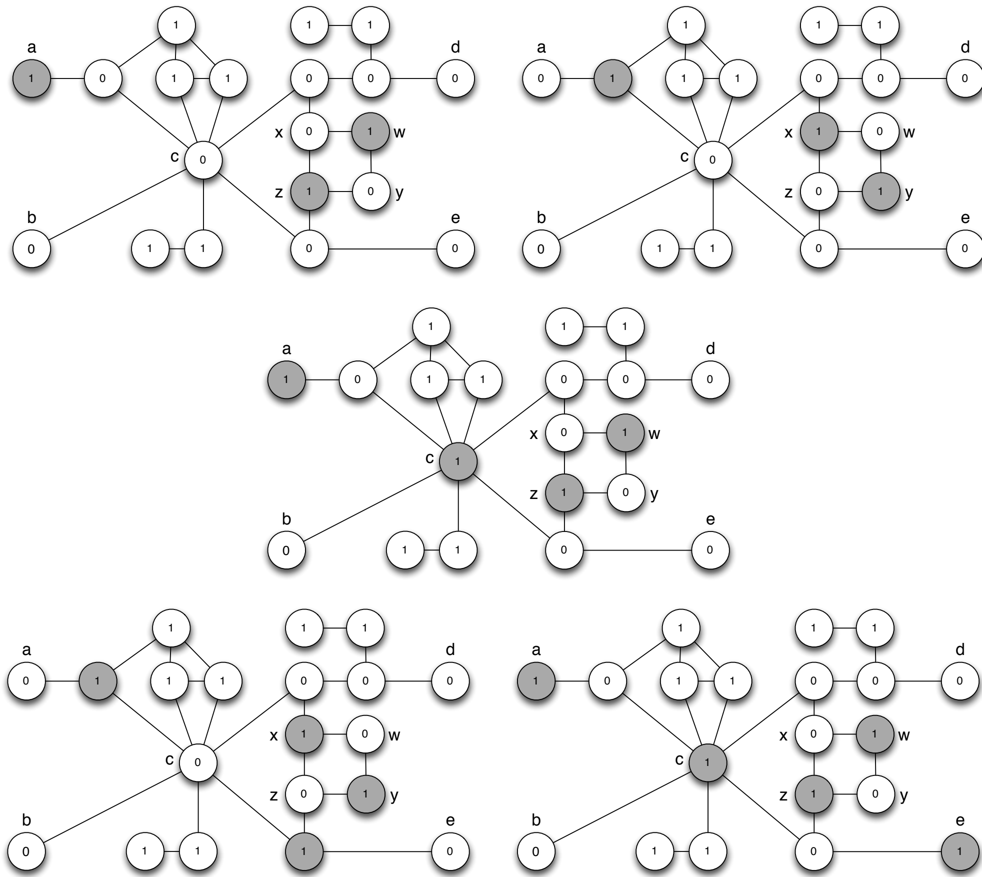

Figura 3.8: Los pasos necesarios para que el gadget de cruce de información transmita desde **a** hacia **e** sin contaminar **d**. Recordar que suponemos que antes de los inputs y después de los outputs irán diodos.

Volviendo con la **Etapa 2**, vamos a usar nuestro gadget para cambiar las posiciones de los vértices que llevan la información del layer l al layer $l+1$. Comenzando con los inputs del primer gadget del layer $l+1$, que llamamos g_1. Sea i_a el menor input (en enumeración) de g_1, si la posición de i_a no es $(1, p_l + 1)$, usamos nuestro gadget de cruce de cables con $\mathbf{a} = (i-1)_b$, $\mathbf{b} = i_a$, $\mathbf{d} = (2(i-1), p_l + 2)$, $\mathbf{e} = (2i-1, p_l + 2)$ y para el resto de los vértices del gadget, coordenadas muy cercanas a $(2i - 3/2, p_l + 3/2)$. Conectamos mediante un cable de largo 10 (Figura 3.4) a todos los vértices con posición $(s, p_l + 1)$ con el fin del cable en $(s, p_l + 2)$, donde $s \in \{1, \ldots, N_l\} \setminus \{i_a, (i-1)_b\}$ (Figura 3.9).

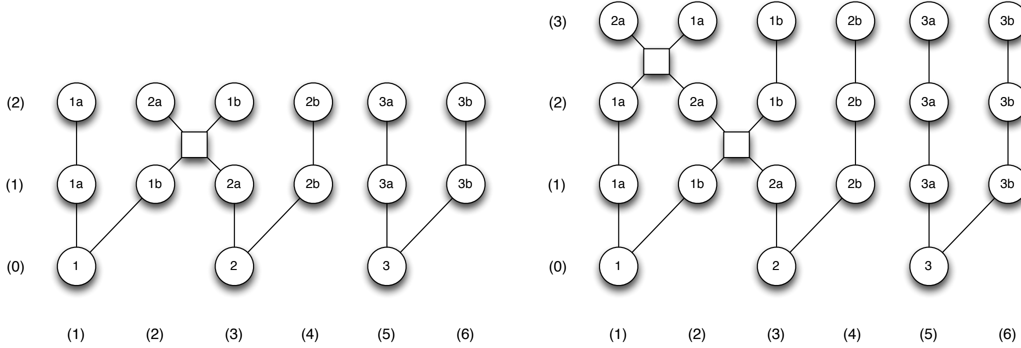

Figura 3.9: Esquema de la etapa 2: el nodo cuadrado representa que ahí va un gadget de cruce de información. Los arcos con extremos con vértices con coordenada y (1) y (2), o (2) y (3), representan cables de largo 10.

En suma, hemos agregado una nueva fila de vértices tal que i_a cambia su coordenada x con $(i-1)_b$ usando el gadget para cruzar cables, y todos los demás quedaron en su posición (Figura 3.9). Repetimos esto hasta que i_a alcance la posición 1, y luego lo repetimos para el otro input j, hasta que alcance la posición 2. Una vez que los dos inputs del primer gate están en su posición correcta, aplicamos lo mismo para la segunda gate del layer $l+1$, etc. (Figura 3.10).

Todo esto se puede hacer en espacio $\mathcal{O}(\log(n))$. En efecto, para un vértice v en el layer $l+1$, necesitamos:

- Identificar sus inputs (si son i_a o i_b), lo que requiere espacio $\mathcal{O}(\log(n))$, ya que necesitamos almacenar cuales son los inputs de v ($\mathcal{O}(\log(n))$) y dos valores booleanos $\mathcal{O}(1)$ para determinar si v es el primero en el layer que tiene cada uno de los inputs.
- Calcular la posición inicial en la que se encuentran los inputs i_x de v, la cual es

$$I[i_x] = 2i + Q[x] + P[i],$$

  donde $Q[x] = -1$ si $x = a$, 0 si $x = b$, y $P[i] = N_a[i] + N_b[i]$, con

$$N_a[i] = |\{j \in \text{layer } l \mid j_a \text{ es input de } u < v \text{ , y } j > i \}|,$$

$$N_b[i] = |\{j \in \text{layer } l \mid j_b \text{ es input de } u < v \text{ , y } j > i \}|.$$

  $P[i]$ se puede calcular en $\mathcal{O}(\log(n))$ ya que sólo necesitamos chequear desde 1 hasta v si alguna puerta tiene uno, dos o ningun input menor que i ($\mathcal{O}(\log(n))$), y $Q(x)$ se calcula usando solo espacio $\mathcal{O}(\log(n))$ usando el punto anterior.

- Construir los gadgets, lo que requiere espacio $\mathcal{O}(\log(n))$ para almacenar los punteros de los inputs y outputs, para asignar coordenadas.

Luego, la **Etapa 2** requiere solo espacio $\mathcal{O}(\log(n))$.

Etapa 3: Finalmente, tenemos que para todo vértice que simula una puerta en el layer $l+1$ tiene sus inputs en las posiciones $2v-1$ y $2v$, por lo que basta dibujar los diodos correspondientes. Esto requiere espacio $\mathcal{O}(1)$, ya que solo necesitamos llevar una cuenta de la paridad (conectamos de dos en dos).

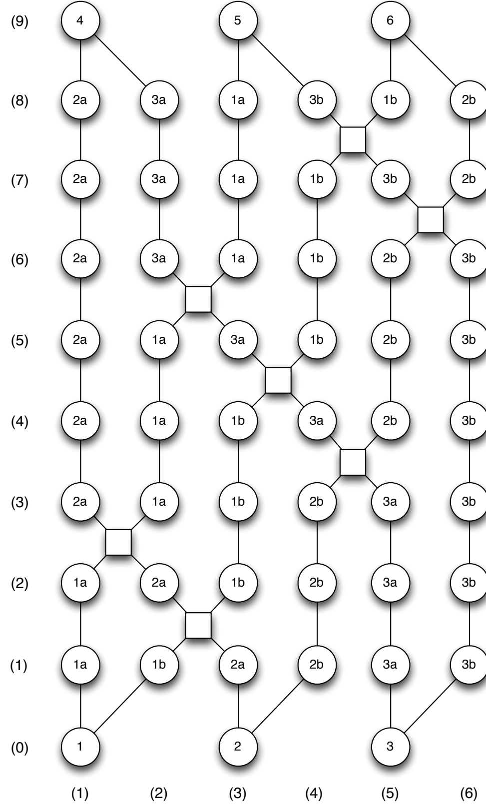

Figura 3.10: Esquema de la incrustación plata que simula las dos primeras capas del circuito en la Figura 3.6. El nodo cuadrado representa el gadget para cruzar cables. Los arcos con extremos con coordenada y diferente y entre (1) y (8) representan cables de largo 10.

Repetimos esto para cada layer del circuito. Entonces, con estos gadgets y esta forma de construir el grafo, dado ϕ un circuito booleano que cumple todas las restricciones de **AM2CVP**, I un input del circuito y s una puerta output, podemos construir un grafo **plano** G simulando las puertas lógicas con los gadgets recién explicados. Definiendo la configuración inicial x como la que tiene activos a los vértices necesarios para los gadgets, activos a los vértices que simulan puertas lógicas con input Verdadero y pasivos al resto, y tomando como v al vértice que simula la puerta lógica s, tenemos que (G, x, v) pertenece a **PER** si y sólo si (ϕ, I, s) pertenece a **AM2CVP**. Y como ya vimos esta reducción se puede hacer en espacio $\mathcal{O}(\log(n))$, por lo que para la regla de la mayoría estricta **PER** es **P**-Duro en la familia de grafos planos. $\square$

Capítulo 4

Complejidad y modos de actualizar

En los capítulos anteriores vimos cómo afecta la complejidad de las redes de autómatas cuando variamos la topología del grafo que define el autómata. La complejidad de una red de autómatas la medimos a través del problema de decisión **PER**. Los cambios en la topología han sido, por ejemplo, restricciones en el grado máximo o trabajar sobre la familia de grafos planos. Todo esto siempre considerando un modo de iterar síncrono.

Ahora, en cambio, estudiaremos qué ocurre con la complejidad de la red de autómatas cuando variamos a otros modos de iterar. Es decir, estudiaremos si es posible establecer si la complejidad depende del modo de iterar. Definiremos entonces problemas de decisión que van en la misma línea que **PER** pero que considere como variable el modo de iterar.

En la próxima sección definiremos problemas de decisión adecuados para éste propósito, y luego, en las secciones siguientes, lo aplicaremos primero a las reglas vistas anteriormente (Bootstrap Percolation y la regla de la mayoría estricta) obteniendo para ambos casos que de algún modo, la complejidad no varía al considerar cambios en el modo de iterar. Finalmente definiremos otra regla, cuya complejidad si será sensible a cambios en el modo de iterar la red.

4.1. Problema de decisión

Como discutimos en el primer capítulo, al momento de definir la complejidad computacional de una red de autómatas, hay varios problemas de decisión que podemos escoger. En nuestro caso, escogimos el problema de decidir cambios en un sólo vértice.

Del mismo modo, cuando se introducen variaciones en el modo de iterar, debemos escoger cómo éstos participan del problema de decisión. En este sentido, trabajaremos con las siguientes variantes de **PER**.

PER-ASYNC Sea $\mathcal{A} = (G = (V, E), \{0,1\}, (f_i : i \in V))$ una red de autómatas, x una configuración de G y $v \in V$ un vértice de G con $x_v = 1$. ¿Existe una palabra w de G tal que $\exists y \in T(x, F, w)$ con $y_v = 1$?, donde F es la función de transición global de $\mathcal{A}$.

PER-ASYNC es el problema de decisión que consiste en determinar si existe una palabra para la cual un vértice determinado cambia de estado en algún momento. Dicho de otro modo, la respuesta al problema de decisión **PER-ASYNC** será negativa si un vértice no cambia independiente del modo en que se actualice la red.

PER-SEC Sea $\mathcal{A} = (G = (V, E), \{0,1\}, (f_i : i \in V))$ una red de autómatas, x una configuración de G, $v \in V$ un vértice de G con $x_v = 1$, y w una palabra de G. ¿Existe $y \in T(x, F, w)$ con $y_v = 1$?, donde F es la función de transición global de $\mathcal{A}$.

PER-SEC por su parte, es el problema que consiste en determinar si un cierto vértice es o no estable para un modo de iterar fijo. Ciertamente si la palabra $w = (V)$, es decir, iteramos todos los nodos en paralelo, tenemos que

$$(\mathcal{A}, x, v, (V)) \in \mathbf{PER-SEC} \iff (\mathcal{A}, x, v) \in \mathbf{PER}.$$

Más aún, en general

$$(\mathcal{A}, x, v) \in \mathbf{PER} \Rightarrow (\mathcal{A}, x, v) \in \mathbf{PER-ASYNC} \Rightarrow \exists w \text{ tal que } (\mathcal{A}, x, v, w) \in \mathbf{PER-SEC}.$$

A continuación expondremos ejemplos donde estudiaremos la complejidad computacional de estos problemas.

4.2. Bootstrap Percolation

Recordemos que Bootstrap Percolation es la regla en que un nodo pasivo pasaba a activo si la mayoría estricta de sus vecinos era activo, y de ahí en adelante se mantiene siempre activo. Veremos que para este caso la complejidad computacional no depende del modo de iterar.

Teorema 4.1 *Si F es Bootstrap Percolation, entonces para toda palabra w*

$$(\mathcal{A}, x, v, w) \in \mathbf{PER-SEC} \iff (\mathcal{A}, x, v) \in \mathbf{PER}.$$

Demostración. Supongamos primero que $(\mathcal{A}, x, v) \in \mathbf{PER}$ y sea ω una palabra de G arbitraria. Tenemos entonces que $\exists T > 0$ tal que $x_v(T) = 1$. Sea $S_k = \{u \in V | \, x_u(k) = 1\}$ el conjunto de todos los vértices activos en el step k para la iteración paralela y $S_k^\omega = \{u \in V | \; x_u^\omega(k) = 1\}$ el de los activos en el step k para la iteración dada por ω.

Notemos que como los vértices activos se mantienen siempre activos, entonces $S_k \subset S_k^{\omega}$, para todo $k \in \mathbb{N}$. En efecto, $S_0 = S_0^{\omega} = \{u \in V |\ x(u) = 1\}$ y si suponemos que $S_{k-1} \subset S_{k-1}^{\omega}$, entonces tendremos que para todo $u \in S_k \backslash S_{k-1}$, la mayoría de los vecinos de u están activos en el step $k-1$ para ambos modos de iterar. Como los nodos activos se mantienen siempre activos, $x_u^{\omega}(k) = 1$, y luego $u \in S_k^{\omega}$. Por lo tanto, $x_v(T) = 1 \Rightarrow v \in S_T \Rightarrow v \in S_T^{\omega} \Rightarrow x_v^{\omega}(T) = 1$ y luego $(\mathcal{A}, x, v, w) \in \mathbf{PER-SEC}$.

Supongamos ahora que $(\mathcal{A}, x, v) \notin \mathbf{PER}$. Por lo visto en el Capítulo 2, se tiene que necesariamente v pertenece a una alianza A de vértices pasivos para x. Por lo tanto, independiente del modo de iterar la red, v siempre será pasivo. En efecto, para una palabra w, sea u es el primer vértice en A en volverse activo, digamos en un tiempo $\bar{t}$. Entonces, en $y = F(x(\bar{t}-1), w[u])$, es decir, en la configuración que había justo antes que u se actualizara, se tiene que u tiene más vecinos activos que pasivos. Como A es una alianza, la mayoría de los vecinos de u también están en A, luego tenemos una contradicción con que u sea el primero de su alianza en volverse activo. Por lo tanto para toda palabra w, $(\mathcal{A}, x, v, w) \notin \mathbf{PER-SEC}$. □

Notemos que el Teorema anterior es válido para las reglas Bootstrap Percolation no estricto y umbral θ, donde las demostraciones son análogas.

Corolario 4.2 *Si F es Bootstrap Percolation*

1. *En la familia de grafos G tales que $\Delta(G) \geq 5$, el problema* **PER-SEC** *es **P**-Completo.*
2. *En la familia de grafos G tales que $\Delta(G) \leq 4$, el problema* **PER-SEC** *está en* **NC**.

4.3. Regla de la mayoría

Recordemos que la regla de la mayoría corresponde a la regla en que tanto vértices activos como pasivos toman el estado de la mayoría de sus vecinos, y mantienen su estado en caso de empate. Para esta regla, tenemos el siguiente resultado:

Teorema 4.3 *Si M es la regla de la mayoría y P es Bootstrap Percolation, entonces*

$$(\mathcal{A} = (G, \{0,1\}, M), x, v) \in \mathbf{PER-ASYNC} \iff (\mathcal{A} = (G, \{0,1\}, P), x, v) \in \mathbf{PER}.$$

Demostración. Sea $\{x(t) : t \geq 0\}$ la trayectoria obtenida a partir de x con la regla Bootstrap Percolation, y sea $\{S_0, ..., S_m\}$ una familia de subconjuntos de V tales que $S_0 = \{u \in V\ :\ x(u) = 1\}$ y $S_{k+1} = \{u \in V\ :\ |N(u) \cap S_k| > |N(u) \cap (V \backslash S_k)|\}$, en otras palabras S_0 es el conjunto de los vértices inicialmente activos, y S_{k+1} es el conjunto de vértices que tiene la mayoría de sus vecinos en S_k. Notemos que $u \in S_{k+1} \backslash S_k \iff x_u(k+1) = 1$ y $x_u(k) = 0$.

$(G, x, v) \in \mathbf{PER}$ significa que existe $T > 0$ tal que $x_v(T) = 1$. Tomando el mínimo T que satisface esto, tenemos que $v \in S_T \backslash S_{T-1}$. Si elegimos entonces la palabra $w = (S_1 \backslash S_0, S_2 \backslash S_1, \ldots, S_0)$ tendremos que si $(\{\overline{x}(t) : t \geq 0\}, w)$ es la trayectoria asíncrona de

x con la regla de la mayoría estricta y palabra w, entonces $\overline{x}_v(1) = 1$ y entonces $(G, x, v) \in$ **PER − ASYNC**.

$(G, x, v) \notin$ **PER** significa que v pertenece a una alianza A de vértices inicialmente pasivos. Luego, no importa como se itere la red, los vértices en A siempre tendrán más vecinos pasivos que activos, y entonces para cualquier palabra w, si $(\{\overline{x}(t) : t \geq 0\}, w)$ es la trayectoria asíncrona de x con la regla de la mayoría estricta y palabra w, entonces $\overline{x}_a(t) = 0$ $\forall t \geq 0$ $\forall a \in A$, y luego $\overline{x}_v(t) = 0$ para cualquier palabra w y todo $t \geq 0$, y entonces $(G, x, v) \notin$ **PER − ASYNC**. □

Corolario 4.4 *Si M es la regla de la mayoría*

1. *En la familia de grafos G tales que $\Delta(G) \geq 5$, el problema* **PER-ASYNC** *es* ***P****-Completo.*
2. *En la familia de grafos G tales que $\Delta(G) \leq 4$, el problema* **PER-ASYNC** *está en* **NC**.

4.4. Redes AND-OR

Sea $G = (V, E)$ un grafo no dirigido, simple, finito y conexo. Suponga que tenemos una partición de V en dos subconjuntos que llamamos **ANDS** y **ORS**, y las siguientes funciones de transición local definidas a partir de ésta:

$$f_v(x) = \begin{cases} 1 & \text{si} \quad \forall u \in N(v) \quad x(u) = 1 \\ 0 & \text{si} \quad \exists u \in N(v) \quad x(u) = 0 \end{cases} \quad \text{si } v \in \textbf{ANDS},$$

$$f_v(x) = \begin{cases} 1 & \text{si} \quad \exists u \in N(v) \quad x(u) = 1 \\ 0 & \text{si} \quad \forall u \in N(v) \quad x(u) = 0 \end{cases} \quad \text{si } v \in \textbf{ORS}.$$

Es decir, un nodo que pertenece a **ANDS** pasará a ser activo si todos sus vecinos son activos, y pasivo en caso contrario; un nodo que pertenece a **ORS** pasará a activo si alguno de sus vecinos es activo, y a pasivo en caso contrario. Llamamos regla AND-OR a la regla definida a partir de estas funciones de transición y la partición dada.

Teorema 4.5 *Sea F la regla AND-OR y x una configuración de G tal que*

$$\{v \in V \mid x_v = 1\} \subset \textbf{ORS}$$

es decir, los nodos inicialmente activos son todos del tipo OR. Entonces:

1. **PER-SEC** *es* **P-Duro.**
2. *Sea W la familia de palabras que satisfacen que $\forall v \in$* **ANDS** *se tiene que*

$$w(v) = w(u) \quad \forall u \in N(u) \cap \textbf{ORS}.$$

Entonces para la familia de palabras W, **PER-SEC** *está en* **NC**.

DEMOSTRACIÓN. (del Teorema 4.5 parte 1.)

Como en los capítulos anteriores, demostraremos la P-Dureza simulando un circuito monótono con una red de autómatas con la regla AND-OR. Recordemos que nuestra versión restringida del **CVP** considera circuitos monótonos con Fan-In 2, Fan-Out 2, síncronos y ordenados lexicográficamente. Para este caso, usaremos también que el circuito es *alternante*, es decir, alterna entre layers de puertas OR y AND, donde el layer input y el output son de puertas OR.

En este caso, nuestro algoritmo no solo construirá un grafo que simule las puertas lógicas del circuito, sino que además irá definiendo la palabra que definirá el modo de iterar la red. Esto se hará de modo que los nodos que simulen puertas del layer i se actualicen antes que los del layer $i+1$ y después que los del layer $i-1$; sin contar los nodos que simulen los inputs y los outputs, que se actualizarán al último y penúltimo, respectivamente. La idea es que la información atraviese todo el circuito, si es el caso, en una sola iteración.

Comenzamos definiendo un diodo, que como antes, nos permitirá transmitir la información en una sola dirección. En Figura 4.1 el nodo con contorno doble (q) representa un nodo perteneciente a **ANDS**, mientras que los otros dos (p y r) pertenecen a **ORS**.

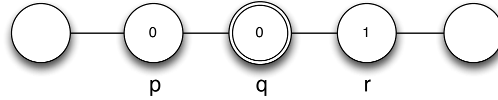

Figura 4.1: Un diodo

Este gadget se define de modo que $w(i) < w(p) < w(q) < w(r) < w(o)$, por lo tanto, un 1 que viene de (i) activará a (p), y entonces (q) tendrá ambos vecinos activos y entonces se activará, con lo que finalmente (r) se mantiene activo y le transmite la información a (o). Si por el contrario, (o) es activo pero (i) no lo es, luego de una iteración sólo (r) se mantendrá activo, ya que (q) requiere ambos vecinos activos para activarse, por lo que la información no logra transmitirse a (i).

Usando este diodo, construimos los gadgets que simulan puertas AND y OR. Sea g una puerta con enumeración u. Si g es una puerta AND, será simulada por el gadget que muestra la Figura 4.2; si es una puerta OR, entonces será simulada por el de la Figura 4.3. Al igual que en la Figura 4.1, los nodos con contorno doble (a, e, f) representan nodos pertenecientes a **ANDS**, y todos los demás son nodos están en **ORS**.

Ambos gadgets se definen de modo que $w(a) = 5u-4$, $w(b) = 5u-3$, $w(c) = w(d) = 5u-2$, $w(e) = w(f) = 5u-1$ y $w(g) = w(h) = 5u$.

Explicaremos como funciona el gadget para simular puertas AND, el funcionamiento del gadget OR es análogo. Si (i_1) e (i_2) son activos en el step $5u-5$, entonces el nodo (a), que recordemos está en ANDS, tendrá todos sus vecinos activos y en el step $5u-4$ se activará, luego en $5u-3$ (b) se mantendrá activo, transmitiendo la información a (o_1) y (o_2) a través de los diodos (c,e,g) y (d,f,h).

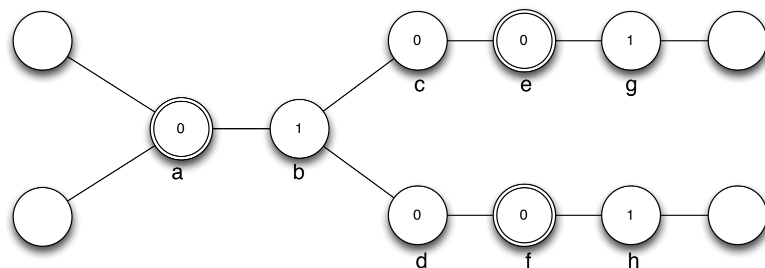

Figura 4.2: Puerta AND

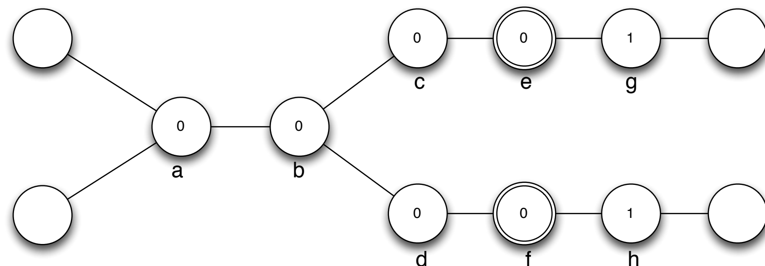

Figura 4.3: Puerta OR

Entonces, dado ϕ un circuito booleano que cumple todas las restricciones anteriormente expuestas, I un input del circuito y s una puerta output, podemos construir un grafo G simulando las puertas lógicas con los gadgets recién explicados. Definiendo la configuración inicial x como la que tiene activos a los vértices necesarios para los gadgets, activos a los vértices que simulan puertas lógicas con input Verdadero y pasivos al resto, tomando como v al vértice que simula la puerta lógica s y definiendo w como la palabra construida a partir de cada puerta, tenemos que (G, x, v, w) pertenece a **PER-SEC** si y sólo si (ϕ, I, s) pertenece a nuestro caso especial del **CVP**.

Esta reducción se puede hacer en espacio $\mathcal{O}(\log(n))$, ya que sólo necesitamos definir qué gadget corresponde con cada puerta ($\mathcal{O}(1)$), almacenar cuatro punteros para determinar adyacencias ($\mathcal{O}(\log(n))$) y almacenar la enumeración de la puerta para construir la palabra ($\mathcal{O}(\log(n))$). Por lo tanto, **PER-SEC** es **P**-Duro. $\square$

Para demostrar la segunda parte del Teorema 4.5, necesitaremos el siguiente lema para el caso en que todos los nodos son del tipo ORS.

Lema 4.6 *Sea G un grafo conexo, F la regla AND-OR en G con particiones* **ANDS** $= \emptyset$ *y* **ORS** $= V(G)$*. Entonces* **PER-SEC** *está en* **NC**.

Demostración. Sea $(\mathcal{A} = (G = (V, E), \{0, 1\}, F), x, v, w)$ una instancia de **PER-SEC**, donde **ORS** $= V$. Suponemos que existe al menos un vértice inicialmente activo, ya que de lo contrario el problema es trivial. Supongamos entonces que hay al menos un vértice inicialmente activo.

Si hay dos vértices inicialmente activos que sean adyacentes, digamos u y r, entonces en a lo más $|V(G)|$ pasos la dinámica entra a un punto fijo en que todos los vértices son activos. En efecto, como u y r se quedan siempre activos, en el primer paso u y todos sus vecinos serán activos, en el segundo paso se mantiene activo u y sus vecinos, y pasan a activos los vecinos de los vecinos; y así sucesivamente. Tendremos entonces que en el paso t todos los elementos en $A(t) = \bigcup_{k=0}^{t} N^k(u)$ serán activos. Como G es conexo $A(|V|) = V$ y luego en el paso $|V|$ todos los vértices de G son activos.

Suponemos entonces que no hay dos vértices que sean inicialmente activos y adyacentes. Supongamos ahora que todos los vértices de G se actualizan en paralelo, es decir, supongamos que $w = (V)$. Sea u un vértice inicialmente activo y definamos el conjunto $B(t) = \bigcup_{k=0}^{\lfloor \frac{t}{2} \rfloor} N^{i-2k}(u)$, es decir, si t es par $B(t)$ es el conjunto de vértices cuya distancia a u es un número par en $[0, t]$. Si t es impar $B(t)$ es el conjunto con distancia impar en $[1, t]$. Tenemos entonces que en el paso t, todos los vértices del conjunto $B(t)$ son activos. En efecto inicialmente $B(0) = \{u\}$ y $x_u = 1$. Si suponemos que en el paso k todos los vértices en $B(k)$ son activos, entonces para todo $r \in B(k+1)$ existe, por definición de $B(\cdot)$, un elemento $t \in B(k)$ adyacente a r. Como todos los nodos se actualizan en paralelo, tendremos que r pasa a ser activo en $k + 1$.

Notemos que en el paso $t > 1$, los vértices en $B(t-1)$ pueden no ser activos. Por ejemplo, como suponemos que los nodos inicialmente activos no tienen vecinos activos, necesariamente todos estos serán pasivos en $t = 1$. Como G es conexo, se tiene que $B(|V|) = V$ y entonces en a lo más $2|V|$ pasos llegamos a un punto fijo donde todos son activos, o bien a un ciclo de largo dos. En efecto, si G es bipartito y todos los vértices inicialmente activos pertenecen a una misma partición de G, la trayectoria en ese caso entra a un ciclo que consiste en una alternación de estados por cada partición. En caso contrario, necesariamente serán activos en un mismo paso dos nodos adyacentes, y por lo visto anteriormente, la dinámica entra a un punto fijo donde todos los nodos son activos luego de a lo más $|V|$ pasos.

Supongamos ahora que no todos los vértices en G se actualizan en paralelo. Recordemos que estamos en el caso en que los nodos activos tienen sólo vecinos pasivos. Si todos los vértices inicialmente activos se actualizan antes que sus vecinos, tendremos que en el primer paso todos los nodos serán pasivos, y luego la trayectoria entra en un punto fijo donde todos los vértices son pasivos.

Sea ahora u un vértice inicialmente activo que tiene un vecino que se actualiza antes que él. Tendremos entonces dos vértices activos adyacentes en el primer paso, y por lo visto anteriormente en a lo más $|V|$ pasos la dinámica entra en un punto fijo donde todos los vértices son activos.

Por último, en el caso que para todo vértice inicialmente activo se cumple que todos sus vecinos se actualizan al mismo tiempo que u, pero no toda la red se itera en paralelo, consideramos C la componente conexa que contiene a u, en el grafo de los nodos que se actualizan al mismo tiempo que u. Por lo dicho anteriormente, en a lo más $|C|$ pasos, la dinámica definida en C entra a un ciclo de largo dos o a un punto fijo. Si se entra a un punto fijo entonces nuevamente tenemos el caso de dos vértices inicialmente activos y adyacentes, por lo que en $|V|$ pasos se llega a un punto fijo. Si por otro lado la dinámica definida en C

entra a un ciclo de largo dos, como no todo el grafo se actualiza en paralelo tenemos que existe un $r \in V \backslash C$ tal que $\exists s \in C$ tal que $rs \in E$. Independiente de como se actualice r, en a lo más dos $|C| + 2$ pasos r y s serán activos y luego, nuevamente, en $|V|$ pasos la dinámica entrará en un punto fijo. Luego, en a lo más $2|V| + 2$ tendremos que todo el grafo pasa a ser activo.

En resumen,

- Si todos los vértices son inicialmente pasivos; o bien no hay vértices inicialmente activos y adyacentes, y todos los vértices inicialmente activos no tienen vecinos activos y se actualizan antes que sus vecinos, entonces la dinámica entra en un punto fijo donde todos los nodos son pasivos
- Si todos los vértices se actualizan en paralelo, el grafo es bipartito y todos los nodos inicialmente activos pertenecen a la misma partición, entonces la dinámica entra en un ciclo de largo dos.
- La dinámica entra en un punto fijo donde todos son activos en cualquier otro caso.

Notemos que en el análisis anterior, si un nodo que no es inicialmente activo pasa a ser activo, entonces no es posible que cuando la trayectoria llegue a un punto fijo este nodo sea pasivo. Definimos entonces el Algoritmo 4, el cual calculará a partir de la matriz de adyacencia asociada a $G = (V, E)$, una configuración x de G, $v \in V$ y una palabra w de G, el estado de cada vértice cuando la trayectoria entra en el punto fijo. Claramente éste algoritmo trabaja más de lo necesario para responder al problema **PER-SEC**, pero el algoritmo de este modo nos será útil para el caso en que no todos los vértices son del tipo ORS.

Notemos que cada paso de este algoritmo se puede ejecutar en tiempo $\mathcal{O}(\log^2(n))$ y $\mathcal{O}(n^2)$ procesadores. En efecto, en el algoritmo los pasos (1) y (7), (9) y (10) se pueden hacer en tiempo constante con $\mathcal{O}(n)$ procesadores, los pasos (2), (3), (4), (6) y (8) se pueden hacer en $\mathcal{O}(\log(n))$ y $\mathcal{O}(n^2)$ procesadores ya que sólo es necesario hacer chequeos locales y luego ejecutar un algoritmo de sumas de prefijos.

Por último, el paso (5) se puede ejecutar en tiempo $\mathcal{O}(\log n)$ y $\mathcal{O}(n^2)$. Comenzamos calculando un árbol recubridor de G, tomando un vértice arbitrario como raíz. A continuación calculamos el nivel (distancia a la raíz) de cada vértice del árbol. Entonces, el conjunto de vértices queda particionado en dos conjuntos disjuntos, dependiendo si el nivel del vértice es par o impar. Finalmente, determinamos que todos los extremos de los arcos de G pertenezcan a subconjuntos diferentes. Encontrar un árbol recubridor de G se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ usando $\mathcal{O}(n^2)$ procesadores [13]. Calcular el nivel de los vértices de un árbol toma tiempo $\mathcal{O}(\log n)$ y requiere $\mathcal{O}(n)$ procesadores [26]. Los demás se puede hacer en tiempo $\mathcal{O}(1)$ y requiere $\mathcal{O}(n^2)$ procesadores.

Algoritmo 4 ORS

Entrada: La matriz A de $n \times n$ que corresponde a la matriz de adyacencia de un grafo no dirigido $G = (V, E)$, una configuración x de G, un vértice $v \in V$ y una palabra w de G.

Salida: Un vector **S** de dimensión n donde $S(u) = 0$ o 1, si cuando la trayectoria entra en un punto fijo u es activo o pasivo, respectivamente; y si la trayectoria entra en un ciclo de largo dos, $S(u) = P$ o I dependiendo si u es activo en los pasos pares o impares, respectivamente.

1: Definir un vector **S** de largo n donde $S(i) \leftarrow 0$, para todo i.
2: Determinar si hay algún nodo inicialmente activo.
   Si $\rightarrow$ (3),
   No $\rightarrow$ (9).
3: Determinar si hay dos nodos inicialmente activos adyacentes
   Si $\rightarrow$ (10),
   No $\rightarrow$ (4).
4: Determinar si todos los nodos se actualizan en paralelo.
   Si $\rightarrow$(5),
   No$\rightarrow$ (8).
5: Determinar si el grafo es bipartito con particiones **1** y **2**, y obtener el vector C donde $C(u) = i$ si u pertenece a la partición i, en caso que el grafo sea bipartito.
   Si el grafo resulta ser bipartito ir a (6).
   Si no, ir a (10).
6: Determinar si todos los vértices inicialmente activos pertenecen a una misma partición.
   Si $\rightarrow$ almacenar en k si es la primera o la segunda partición e ir a (7),
   No$\rightarrow$(10)
7: Asignar $S(u) =$ **P** a los vértices u que pertenezcan a la partición k y $S(u) =$ **I** a los otros.
   Terminar
8: Determinar si existe algún vértice inicialmente activo que tenga un vecino que se actualice antes o al mismo tiempo que él.
   Si $\rightarrow$(10),
   No$\rightarrow$ (9)
9: Asignar $S(u) =$**0** a todos los nodos $u \in V$ y **Terminar.**
10: Asingar $S(u) =$**1** a todos los nodos $u \in V$ y **Terminar.**

Un análisis exhaustivo de la complejidad de este algoritmo se puede encontrar en el anexo. Por lo tanto, para decidir **PER-SEC** en caso que todos los vértices son del tipo ORS, bastará ejecutar el Algoritmo 4 y luego verificar que $S(v) \neq 0$ y aceptar, o rechazar en caso contrario. □

Demostremos ahora el Teorema 4.5.

Demostración. (del Teorema 4.5) Sea G un grafo conexo, F la regla AND-OR en G y w una palabra que pertenece al familia W, es decir todos los vértices de tipo **ANDS** se actualizan en paralelo con sus vecinos de tipo **ORS**. Sea x una configuración donde $x_u = 0$ para todo vértice u en **ANDS** .

Notemos que como todos los vértices **ANDS** son inicialmente pasivos, si dos vértices de este tipo son adyacentes entonces serán pasivos siempre, por los que solo nos podrán aportar algo aquellos nodos **ANDS** que solo tengan vecinos **ORS**. Lo primero que haremos entonces será calcular las componentes conexas de **ORS** en G, usando el Algoritmo 1 definido en el Capítulo 1. A continuación le aplicaremos el Algoritmo 4 a cada subgrafo inducido por cada componente conexa. Claramente si el resultado es que $S(u) = 1$ para los elementos de alguna componente, esto seguirá siendo cierto para el grafo completo.

Sea v un vértice del tipo **ANDS** , tal que $N(v) \subset$ **ORS**, es decir todos sus vecinos son del tipo **ORS**, y además todos sus vecinos son inicialmente activos. Entonces en todos los pasos impares v será activo y en todos los pasos pares lo serán todos los vecinos de v. En efecto, como todos sus vecinos son inicialmente activos y v se actualiza en paralelo con ellos, v se activará en $t = 1$ y como todos sus vecinos son del tipo OR, todos serán activos $t = 2$, y así sucesivamente.

Sea entonces C una componente donde $S(u) = 0$, para todo $u \in C$. Notemos que esto no necesariamente es cierto para la dinámica en el grafo completo. En efecto, si C contiene a solo un vértice inicialmente activo u, como el grafo es conexo, necesariamente u tendrá al menos un vecino v del tipo **ANDS** . Como antes, si este vecino tiene todos sus vecinos del tipo **ORS**, y todos son inicialmente activos, entonces en los pasos pares u será activo y pasivo en los impares. Es decir, cambiaremos $S(u)$ de 0 a P. Por otro lado, si C contiene a más de un vértice y existe un vértice u inicialmente activo en C, tendremos que como $S(u) = 0$ necesariamente todos sus vecinos de tipo **ORS** se actualizan después que él. Si u es vecino de un nodo v del tipo **ANDS** , que nuevamente tiene todos sus vecinos inicialmente activo, entonces u pasará a ser activo en todos los pasos pares, y como vimos en la demostración del Lema 4.6, como no todos los vértices de C se actualizan en paralelo tendremos que en a lo más $|C|$ pasos la componente completa pasará a ser activa. Si lo anterior no se cumple para ningún nodo en C, para cualquier nodo u inicialmente activo todos sus vecinos **ANDS** son adyacentes a otro vecino en **ANDS** , o bien no tienen todos sus vecinos inicialmente activos, entonces ningún vecino de u pasará activo en el primer paso y entonces toda la componente C se quedará pasiva.

Sea v un vértice del tipo **ANDS**, tal que $N(v) \subset$ **ORS** y en cierto paso t se tiene que todos sus vecinos son activos y alguno es del del tipo P. Entonces en todos los pasos impares v será activo y en todos los pasos pares pasivo. En efecto, en los pasos pares todos los vecinos

de v serán activos, pero como v se actualiza al mismo tiempo que sus vecinos, en particular los de tipo P, en los impares v será activo, pero al mismo tiempo sus vértices de tipo P serán pasivos, y luego en los pasos pares v será pasivo. Lo anterior es análogo cuando todos los vecinos de v son activos y alguno es del tipo I, cambiando donde dice par por impar y viceversa.

Por lo tanto, para decidir **PER-SEC** en la familia de palabras W, tenemos que ejecutar el Algoritmo 4 a cada componente conexa y luego verificar si algún vértice u, inicialmente activo y que tenga $S(u) = 1$, tiene un vecino del tipo **ANDS** que tenga todos sus vecinos inicialmente activos. En ese caso se cambia $S(\cdot)$ de 0 a 1 para todos los elementos de la componente que contiene a u. Finalmente, si nuestro nodo objetivo v es del tipo **ORS**, se verifica que $S(v) \neq 0$, y si es del tipo **ANDS** se verifica que $N(v) \subset$ **ORS**, que $\forall u \in N(v)$, $S(u) \neq 0$ y que no existan $u_1, u_2 \in N(v)$ tales que $S(u_1) = P$ y $S(u_2) = I$.

En efecto, si para un nodo del tipo **ANDS** se tienen dos vecinos u_1, $u_2 \in$ **ORS**, con $S(u_1) = I$ y $S(u_2) = P$, entonces v es siempre pasivo porque en todo paso tendrá un vecino pasivo. El algoritmo entonces es el Algoritmo 5.

La correctitud de este algoritmo está dada por la discusión anterior y el Lema 4.6. La complejidad, por su parte, viene dada por las complejidades de los Algoritmos 1 y 4, para los pasos (1) y (2) respectivamente. Los pasos (3), (8), (9) se pueden hacer en tiempo constante con una cantidad $\mathcal{O}(n^2)$ de procesadores. Por último, los pasos (4), (5), (10) y (11) se pueden hacer en tiempo $\mathcal{O}(\log(n))$ usando $\mathcal{O}(n^2)$ procesadores usando el algoritmo de sumas de prefijos. Un análisis exhaustivo de este algoritmo se encuentra en el Anexo.

Por lo tanto, usando el Algoritmo 5 podemos decidir **PER-SEC** en el caso en que la palabra $w \in W$ y los nodos inicialmente activos sean todos del tipo ORS en tiempo $\mathcal{O}(\log^2(n))$ usando $\mathcal{O}(n^2)$ procesadores, y entonces **PER-SEC** en este caso está en **NC**. $\square$

Algoritmo 5 ANDS-ORS

Entrada: La matriz A de $n \times n$ que corresponde a la matriz de adyacencia de un grafo no dirigido $G = (V, E)$, un vector T que contiene el tipo de cada vértice (si es ANDS u ORS), una configuración x de G, un vértice $v \in V$ y una palabra w de G.

Salida: Aceptar si $(G, x, v, w) \in \mathbf{PER} - \mathbf{SEC}$ y rechazar en caso contrario.

1: Calcular las componentes conexas de $G[ORS]$, el grafo inducido por los nodos del tipo ORS, usando el Algoritmo 1.
2: Ejecutar en cada componente de $G[ORS]$ el Algoritmo 4 y guardar los resultados de todas las componentes en un vector S.
3: **si** $\exists u \in \mathbf{ORS}$ tal que $S(u) = 0$ y $x_u = 1$ **entonces**
4:    **si** $\exists p \in N(u) \cap \mathbf{ANDS}$ tal que $\forall q \in N(p), x_q = 1$ **entonces**
5:       Determinar si u es aislado en $G[ORS]$.
      No $\rightarrow$ Definir $S(r) = 1$ para todo elemento r de la componente que contiene a u.
      Si $\rightarrow$ $S(u) = P$.
6:    **fin si**
7: **fin si**
8: Determinar si v pertenece a ORS
Si $\rightarrow$ (9),
No $\rightarrow$ (10)
9: Determinar si $S(v) \neq 0$.
Si $\rightarrow$ (12),
No $\rightarrow$ (13)
10: Determinar si todos los vecinos de v son ORS
Si $\rightarrow$ (11),
No $\rightarrow$ (13)
11: Determinar si existen u_1, $u_2 \in N(v)$ tales que $S(u_1) = P$ y $S(u_2) = I$ o bien si existe $u \in N(v)$ tal que $S(u) = 0$.
Si $\rightarrow$ (13),
No $\rightarrow$ (12)
12: **Aceptar** y **Terminar**.
13: **Rechazar** y **Terminar**.

Conclusión

Hemos discutido sobre la complejidad computacional de redes de autómatas mediante el problema de decisión **PER**. Primero estudiamos algunas variantes de reglas de mayoría en redes de autómatas, y hemos probado que la complejidad de Bootstrap Percolation depende del grado máximo del grafo que se entrega como entrada del problema de decisión. Cuando el grado máximo es pequeño, el problema está en **NC**, y para grados más grandes el problema se vuelve **P**-Completo. Claramente nuestras cotas son estrechas. Nuestras pruebas de **P**-Complitud usan reducciones del caso especial del **CVP**, el cual es **P**-Completo en circuitos arbitrarios.

Una extensión natural de nuestros resultados sería considerar clases especiales de grafos. Una pregunta que surge directamente sería entonces: ¿Qué pasa con Bootstap Percolation **PER** en el caso plano?. Es obvio que los resultados para $\Delta(G) \leq 4$ se mantienen ciertos, por lo que pensamos en el caso en que el grado máximo es al menos 5. Como mencionamos en el primer capítulo, el **CVP** restringido a circuitos monótonos planos está en **NC**, por lo que la técnica de reducción usada para Bootstrap Percolation no sirve directamente.

En ese sentido es que variamos la regla de Bootstrap Percolation, ahora permitiendo que los nodos activos pasen a ser pasivos, es decir, tomando la regla de la mayoría estricta. Para este caso demostramos que aún cuando el grafo es plano el problema es **P**-Completo.

Como dijimos anteriormente, este problema fue estudiado por C. Moore [19], en el caso particular del lattice d dimensional, con $d > 2$, dejando abierto el problema para el caso $d = 2$. En ese contexto, los resultados anteriores nos muestran que si por una parte variamos la regla dejando los 'unos fijos' (i.e. cambiando de la mayoría a Bootstrap Percolation) entonces en lattice de dimensión 2 el problema está en **NC**, ya que en ese caso todos los vértices del grafo tienen grado 4. Por otra parte, si mantenemos la regla de la mayoría pero cambiamos la estructura del grafo, vemos que la planaridad no es suficiente para determinar que el problema está en **NC**.

Por otra parte, hemos discutido como varía la complejidad computacional cuando cambiamos el modo de iterar la red. Notamos que para ciertas reglas, Bootstrap Percolation por ejemplo, la complejidad computacional es invariante con respecto a cambios en el modo de actualizar. Para otras, la complejidad computacional resulta ser sensible al modo de iterar (regla AND-OR). Esto nos lleva a plantear una suerte de *Complejidad dependiente del modo de iterar* para una determinada regla, o si es el caso, una **P**-Dureza *robusta*, en el sentido que con dicha regla se puedan simular circuitos independiente del modo en que se itere.

Algunos problemas abiertos o trabajos futuros

Para Bootstrap Percolation, pensamos que en el caso que el problema esté restringido a la familia de grafos planares, **PER** estará en **NC**, o al menos no es **P**-Completo, independiente del grado del grafo. Recordemos que **PMCVP**, el problema del valor de circuitos monótonos planos, está en **NC**. Si el grafo es plano, con Bootstrap Percolation no hay como simular un circuito no plano sin cruzar información. Dado que los nodos activos se mantienen activos no vemos como podríamos construir semáforos, por ejemplo.

Para el caso de la regla de la mayoría estricta claramente queda aún abierta la conjetura de C. Moore, que dice que en el lattice 2- dimensional el problema está en **NC**. Una forma sería intentar replicar lo hecho para Bootstrap Percolation, pero al no ser siempre activos los nodos inicialmente activos, la técnica de buscar ciclos (alianzas) inicialmente pasivos falla, ya que podría no haber ninguno, pero que se creara uno después de algunas iteraciones.

Finalmente para la regla AND-OR, faltaría probar que para palabras arbitrarias el problema pertenece a **P**, ya que solo probamos la **P**-Dureza del problema. Usar la funciones de energía de Goles [11, 8, 9] para probar que la dinámica entra en un ciclo, como en el capítulo 3, no es una alternativa por ahora, ya que éstas están solo definidas para iteraciones paralelas y secuenciales.

Bibliografía

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, New York, NY, USA, 2009.

[2] J. Balogh and G. Pete. Random disease on the square grid. *Random Structures & Algorithms*, 13:3–4, 1998.

[3] R. Carvajal, M. Matamala, I. Rapaport, and N. Schabanel. Small alliances in graphs. In *MFCS'07*, pages 218–227, 2007.

[4] J. Chalupa, P.L. Leath, and G.R. Reich. Bootstrap percolation on a bethe lattice. *Journal of Physics C: Solid State Physics*, 12:L31–L35, 1979.

[5] A.L. Delcher and S.R. Kosaraju. An nc algorithm for evaluating monotone planar circuits. *SIAM Journal on Computing*, 24:369–375, 1995.

[6] S. Fortune and J. C. Wyllie. Parallelism in random access ma- chines. *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[7] L.M. Goldschlager. A universal interconnection pattern for parallel computers. *J. ACM*, 29(4):1073–1086, 1982.

[8] E. Goles. Comportement oscillatoire d'une famille d'automates cellulaires non uniformes. *Thèse IMAG, Grenoble*, 1980.

[9] E. Goles and S. Martinez. *Neural and automata networks: dynamical behavior and applications.* Kluwer Academic Publishers, 1990.

[10] E. Goles, P. Montealegre-Barba, and I. Todinca. The complexity of the bootstraping percolation and other problems. *Theoretical Computer Science*, 2012.

[11] E. Goles-Chacc, F. Fogelman-Soulie, and D. Pellegrin. Decreasing energy functions as a tool for studying threshold networks. *Discrete Applied Mathematics*, 12(3):261 – 277, 1985.

[12] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits to parallel computation: P-completeness theory.* Oxford University Press, 1995.

[13] J. JáJá. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing

Co., Inc., Redwood City, CA, USA, 1992.

[14] J. Jájá and S.R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3), 1988.

[15] R.E. Ladner. The circuit value problem is log space complete for p. *SIGACT News*, 7(1):18–20, 1975.

[16] S.S. Manna. Abelian cascade dynamics in bootstrap percolation. *Physica A: Statistical Mechanics and its Applications*, 261(3-4):351 – 358, 1998.

[17] W.S. Mcculloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysic*, 5:115–133, 1943.

[18] R.A. Meyers. *Encyclopedia of Complexity and Systems Science - v. 1-10.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[19] C. Moore. Majority-Vote Cellular Automata, Ising Dynamics, and P-Completeness. *Journal of Statistical Physics*, 88:795–805, 1997.

[20] C. Moore. Predicting non-linear cellular automata quickly by decomposing them into linear ones. *Physica D: Nonlinear Phenomena*, 1997.

[21] C. Moore and D. Griffeath. Life without death is p-complete. *Complex Systems*, 10:437–447, 1997.

[22] C. Moore and M.G. Nordahl. Predicting lattice gases is p-complete. *Santa Fe Institute Working Paper 97-04-034.*, 1997.

[23] T. Neary and D. Woods. P-completeness of cellular automaton rule 110. In *Proceedings of the 33rd international conference on Automata, Languages and Programming - Volume Part I*, ICALP'06, pages 132–143, Berlin, Heidelberg, 2006. Springer-Verlag.

[24] I. Rapaport, K. Suchan, I. Todinca, and J. Verstraete. On dissemination thresholds in regular and irregular graph classes. *Algorithmica*, 59:16–34, 2011. 10.1007/s00453-009-9309-0.

[25] M. Treaster, W. Conner, I. Gupta, and K. Nahrstedt. Contagalert: Using contagion theory for adaptive, distributed alert propagation. In *NCA'06*, pages 126–136, 2006.

[26] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20(235–240), 1985.

Apéndice A

Análisis de la Complejidad del Algoritmo 3

Vamos ahora a entregar un análisis detallado de la complejidad del Algoritmo 3, paso a paso. Sea A la matriz de adyacencia del grafo G, $x(0)$ una configuración y v el vértice sobre el cual decidimos.

Paso 1: Calcular $A[0]$, la matriz de adyacencia de $G[0]$ usando $x(0)$ y el algoritmo de sumas de prefijos.

1. Primero calculamos $\overline{x(0)}$ donde

$$\overline{x_i(0)} = \begin{cases} 0 & \text{si} \quad x_i(0) = 1 \\ 1 & \text{si} \quad x_i(0) = 0 \end{cases}$$

   Esto se puede hacer en tiempo paralelo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores (uno por cada nodo), donde el procesador i lee $x_i(0)$ y escribe $1 - x_i(0)$ en el espacio de memoria asignado para almacenar $\overline{x(0)}$

2. Luego, usamos el algoritmo de sumas de prefijos con entrada $\overline{x(0)}$ obteniendo un arreglo s donde

$$s_i = \sum_{j=0}^{i} \overline{x_j(0)}$$

   notemos que s representa las nuevas etiquetas de los vértices de G en el grafo $G[0]$, y s_n es igual al número de vértices de $G[0]$. Como vimos en el primer capítulo, ejecutar el algoritmo de sumas de prefijos requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.

3. Con s, $x(0)$ y A, construimos $A[0]$, la matriz de adyacencia de $G[0]$: Asignamos un espacio de memoria para $A[0]$ de $s_n \times s_n$, relleno de ceros. Para cada $i, j \in \{1, \ldots, n\}$ tal que $x_i(0) = x_j(0) = 0$, si $A_{i,j} = 1$, entonces $(A[0])_{s_i s_j} = 1$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n^2)$ procesadores, asignado un procesador para cada par de vértices del grafo.

Por lo tanto, todo el Paso 1 se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 2: Calcular $A[0,v]$, la matriz de adyacencia de $G[0,v]$, usando $A[0]$ en el Algoritmo 1.

Sea $k = s_n$, donde recordemos s_n es la cantidad de vértices en $G[0]$.

1. Calcular primero C, la salida del Algoritmo 1 con entrada $A[0]$. Esto se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ usando $\mathcal{O}(n^2)$ procesadores. Notemos que C es de largo k.
2. Recordemos que s_v es la etiqueta de v en $A[0]$. Entonces C_{s_v} es la etiqueta de la componente conexa que contiene a v. Calcular entonces $\overline{C}$, un arreglo con las mismas dimensiones que C, tal que $\overline{C}_i = 1$ si $C_i = C_{s_v}$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores (uno por cada nodo de $A[0]$), donde el procesador i lee C_i y escribe 1 o 0, según corresponda, en el espacio de memoria asignado para almacenar $\overline{C}_i$
3. Usar el algoritmo de sumas de prefijos con entrada $\overline{C}$, obteniendo S, donde

$$S_i = \sum_{j=0}^{n} \overline{C}_j$$

   entonces S representa las nuevas etiquetas de los vértices de $G[0]$ en el grafo $G[0,v]$, y S_k es igual al número de vértices de $G[0,v]$. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
4. Con S, $\overline{C}$ y $A[0]$, construimos $A[0,v]$, la matriz de adyacencia de $G[0,v]$: Asignamos un espacio de memoria para $A[0,v]$ de $S_k \times S_k$, relleno de ceros. Para cada $i, j \in \{1, \ldots, S_n\}$ tal que $\overline{C}_i = \overline{C}_j = 1$, si $A[0]_{i,j} = 1$, entonces $(A[0,v])_{S_i S_j} = 1$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n^2)$ procesadores, asignado un procesador para cada par de vértices del grafo.

Por lo tanto, todo el Paso 2 se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores.

Pasos 3,4,5: Determinar si v está aislado en $G[0,v]$ se puede hacer en tiempo $\mathcal{O}(1)$ (secuencial) simplemente verificando que $S_k > 1$.

Paso 6: Calcular B, la salida del Algoritmo 2 en $A[0,v]$. Esto se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ usando $\mathcal{O}(n^2)$ procesadores.

Paso 7: Encontrar los ciclos en $A[0,v]$ a partir de B y almacenarlo en un arreglo C donde $C_i = 1$ si i pertenece a un ciclo en $A[0,v]$

1. Calcular M, el valor máximo en B usando el algoritmo de prefijos con la operación binaria $a + b = max\{a, b\}$. Esto requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
2. Almacenar en N el largo de B. Esto requiere tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^2)$ procesadores. Notemos que $M, N \leq (S_k)^2 \leq n^2$
3. Computar la matriz K de dimensiones $M \times N$, donde $K_{ij} = 1$ si $B_j = i$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^4)$ procesadores.
4. Calcular el arreglo L, donde $L_i = \sum_{j=1}^{N} K_{ij}$, i.e. L_i es el número de arcos en la componente biconexa i. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.
5. Usando L y $A[0,v]$, computar el arreglo C: sea $g = ij$ un arco (i.e. $A[0,v]_{ij} = 1$) con $B_g = p$ y sea $L_p \neq 1$, entonces $C_i = C_j = 1$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^2)$

Por lo tanto el Paso 7 se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^4)$ procesadores.

Paso 8: Calcular D, el vector de los grados de cada vértice en G. Este vector se puede calcular usando el algoritmo de sumas de prefijos sobre cada fila (o columna) de la matriz A, y almacenando el valor del último elemento del vector de salida. Es decir, para este paso requerimos tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 9: Definir $\overline{A}$, la matriz de adyacencia de $\overline{G}$, usando $A[0, v]$, D (paso 8) y C (paso 7). Sea m la dimensión de $A[0, v]$, con una fila y una columna más, correspondiente al vértice ∞. Entonces, para calcular $\overline{A}$ copiamos $A[0, v]$ y $\overline{A}_{i,m+1} = \overline{A}_{m+1,i} = 1$ si $D_i \leq 2$ o $C_i = 1$. Esto requiere tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

Paso 10 Verificar que ∞ no esté aislado: Calcular $\overline{C}$, la salida del Algoritmo 1 en la entrada $\overline{A}$. Se define el arreglo Z por $Z_i = 1$ si $\overline{C}_i = \overline{C}_\infty$ y 0 en otro caso. Luego aplicar suma de prefijos sobre Z, obteniendo un vector s donde s_{m+1} es la cantidad de elementos en la componente que contiene a ∞. Es decir, si $s_{m+1} \neq m + 1$ entonces ∞ es aislado y **aceptar**. Todo esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.

Paso 11: Calcular $\overline{B}$, la salida del Algoritmo 2 en la entrada $\overline{A}$. Esto se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 12: Usando $\overline{B}$, determinar si v y ∞ están en un ciclo.

1. Calcular M, el valor máximo en $\overline{B}$ usando el algoritmo de prefijos con la operación binaria $a + b = max\{a, b\}$. Esto requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
2. Almacenar en N el largo de B. Esto requiere tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^2)$ procesadores. Notemos que $M, N \leq n^2$
3. Computar los arreglos I, V de largo M: Sea $g \in E(\overline{G})$, con $\overline{B}_g = k$. Si $\infty \in g$, entonces $I_k = 1$, y si $v \in g$, entonces $V_k = 1$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n^2)$ procesadores. Notar que V_k (respectivamente I_k) es 1 si v (respectivamente ∞) está en un arco de la componente k.
4. Computar la matriz K de dimensiones $M \times N$, donde $K_{ij} = 1$ si $\overline{B}_j = i$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^4)$ procesadores.
5. Calcular el arreglo L, donde $L_i = \sum_{j=1}^{N} K_{ij}$, i.e. L_i es el número de arcos en la componente biconexa i. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.
6. Calcular $I + V$. Si para algún k, $(I + V)_k \geq 2$ y $L_k > 1$, entonces v y ∞ están en la misma componente no trivial, y por lo tanto en un ciclo. Esto se puede calcular en $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

Entonces, todo el Paso 12 requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^4)$ procesadores.

Sumando todo lo anterior, tenemos en conclusión que el algoritmo completo requiere tiempo $\mathcal{O}(\log^2 n)$ y $\mathcal{O}(n^4)$ procesadores.

Apéndice B

Análisis de la Complejidad del Algoritmo 4

A continuación entregaremos un análisis detallado de la complejidad del Algoritmo 4, paso a paso. Sea A la matriz de adyacencia del grafo G, x una configuración y v el vértice sobre el cual decidimos, y w una palabra de G, donde un vértice i se actualiza antes que j si $w_i < w_j$.

Paso 1: Definir un vector S de largo n, donde $S_i = 0$ para todo i. Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores, simplemente asignando un procesador por vértice del grafo, con un espacio de memoria para cada procesador y haciendo que cada uno escriba un cero en el espacio asignado.

Paso 2: Determinar si hay algún nodo inicialmente activo.

1. Computar el vector Q, que corresponde a la salida del algoritmo de sumas de prefijos con entrada x. Esto toma tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
2. Verificar que $Q_n > 0$, lo que toma tiempo (secuencial) $\mathcal{O}(1)$

Luego, todo el paso dos se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.

Paso 3: Determinar si hay dos nodos inicialmente activos adyacentes.

1. Construir la matriz $\overline{A}$ tal que $\overline{A}_{ij} = \begin{cases} 1 & \text{si} \quad A_{ij} = 1 \quad \text{y} \quad x_i = x_j = 1 \\ 0 & \text{en otro caso} \end{cases}$

   Esto toma tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^2)$ procesadores, ya que simplemente se le asigna un procesador a cada par de vértices, y un espacio de memoria exclusivo donde se escribe la componente correspondiente de $\overline{A}$.
2. Ejecutar el algoritmo de sumas de prefijos para cada fila de $\overline{A}$ obteniendo para la fila i-ésima un vector q^i donde q^i_n es distinto de cero si el vértice i es activo y tiene algún vecino activo, y es igual a cero en caso contrario. Esto toma tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores, ya que tenemos que hacer una suma de prefijos ($\mathcal{O}(\log n)$ con $\mathcal{O}(n)$) por cada fila (n filas).
3. Almacenar cada q^i_n en un vector Q (de largo n). Esto toma tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$

procesadores.

4. Ejecutar el algoritmo de sumas de prefijos sobre Q, obteniendo un vector P. Esto requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
5. Se tiene entonces que $P_n = 0$ si algún vértice activo tiene al menos un vecino activo, es decir, si hay dos vértices adyacentes inicialmente activos. Verificar que $P_n = 0$ requiere tiempo (secuencial) $\mathcal{O}(1)$

Luego todo este paso requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 4: Determinar si todos los nodos se actualizan en paralelo.

1. Sea $\overline{w} = w_1$. Construir el vector Q de dimensión n tal que

$$Q_i = \begin{cases} 1 & \text{si} \quad w_i = \overline{w} \\ 0 & \text{en otro caso} \end{cases}$$

   Esto toma tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.
2. Ejecutar el algoritmo de sumas de prefijos con entrada Q, obteniendo un vector P. Esto toma tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores. Notemos que P_n es la cantidad de vértices que se actualizan al mismo tiempo que 1.
3. Determinar si $P_n = n$. Si es así todos los vértices se actualizan en paralelo. Esto requiere tiempo (secuencial) $\mathcal{O}(1)$.

Luego todo este paso requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.

Paso 5: Determinar si el grafo es bipartito con particiones **1** y **2**, y obtener el vector C donde $C(u) = i$ si u pertenece a la partición i, en caso que el grafo sea bipartito.

1. Calculamos un árbol recubridor de G, lo cual se hace con un algoritmo similar al Algoritmo 1, que se puede ver en [13]. Éste tiene como salida un vector T de dimensión n, donde $T(i) = j$ si j es el padre del vértice i en el árbol recubridor resultante. Si el i es la raíz entonces $T(i) = i$. Esto se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores [13].
2. Calculamos el nivel (distancia a la raíz en el árbol recubridor) de cada vértice del árbol. Obtenemos un vector niv de dimensión n tal que $niv(i) = k$ si el vértice i tiene nivel k. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ y requiere $\mathcal{O}(n)$ procesadores usando un algoritmo que se encuentra en [26].
3. Construimos el vector C de dimensión n, tal que

$$C_i = \begin{cases} 1 & \text{si} \quad niv(i) \text{ es impar} \\ 2 & \text{si} \quad niv(i) \text{ es par} \end{cases}$$

   Esto se puede hacer en tiempo $\mathcal{O}(1)$ y requiere $\mathcal{O}(n)$ procesadores.
4. Determinar si hay un arco que une vértices con nivel de igual paridad. (Si hay arcos que unen vértices de la misma partición)
   (a) Construir la matriz $\overline{A}$ tal que $\overline{A}_{ij} = \begin{cases} 1 & \text{si} \quad A_{ij} = 1 \quad \text{y} \quad C_i = C_j \\ 0 & \text{en otro caso} \end{cases}$
   Esto toma tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n^2)$ procesadores.

(b) Ejecutar el algoritmo de sumas de prefijos para cada fila de $\overline{A}$ obteniendo para la fila i-ésima un vector q^i donde q^i_n es distinto de cero si el vértice i tiene algún vecino j tal que $C_i = C_j$, y es igual a cero en caso contrario. Esto toma tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.

(c) Almacenar cada q^i_n en un vector Q (de largo n). Esto toma tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

(d) Ejecutar el algoritmo de sumas de prefijos sobre Q, obteniendo un vector P. Esto requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.

(e) Se tiene entonces que $P_n \ = 0$ si algún vértice i tiene un vecino j con $C_i = C_j$, es decir, si hay dos vértices adyacentes en la misma partición. Verificar que $P_n \ = 0$ requiere tiempo (secuencial) $\mathcal{O}(1)$

Todo esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.

Luego, todo el Paso 5 se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 6: Determinar si todos los vértices inicialmente activos pertenecen a una misma partición.

1. Calculamos primero q_{tot} que es la última componente de la salida del algoritmo de sumas de prefijos sobre x, es decir q_{tot} es el número de vértices inicialmente activos. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
2. Calculamos luego el vector Q de dimensión n, tal que Q_i vale 1 si $x_i = 1$ y $C_i = 1$, y cero en otro caso. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.
3. Computamos entonces q_1 que es la última componente de la salida del algoritmo de sumas de prefijos sobre Q, es decir q_1 es el número de vértices inicialmente activos en la partición **1**. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
4. Calculamos luego el vector P de dimensión n, tal que P_i vale 1 si $x_i = 1$ y $C_i = 2$, y cero en otro caso. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.
5. Computamos entonces q_2 que es la última componente de la salida del algoritmo de sumas de prefijos sobre P, es decir q_2 es el número de vértices inicialmente activos en la partición **2**. Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
6. Finalmente verificamos si $q_1 = q_{tot}$ o $q_2 = q_{tot}$, es decir, si todos los vértices activos pertenecen a la partición **1** o **2**. Si $q_1 = q_{tot}$ entonces $k = 1$, si $q_2 = q_{tot}$ entonces $k = 2$.

   Esto se puede hacer en tiempo (secuencial) $\mathcal{O}(1)$.

Paso 7: Asignar $S(u) = \mathbf{P}$ a los vértices u que pertenezcan a la partición k y $S(u) = \mathbf{I}$ a los otros. Esto se hace verificando que $C_u = k$, y en ese caso $S_u = P$, y en caso contrario $S(u) = I$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

Paso 8:Determinar si existe algún vértice inicialmente activo que tenga un vecino que se actualice antes o al mismo tiempo que él.

1. Construir R, un vector de tamaño de tamaño n relleno de 0.
2. Para $j = 1 \ldots n$

(a) Sea $\overline{w} = w_j$. Construir el vector Q de dimensión n tal que

$$Q_i = \begin{cases} 1 & \text{si} \quad w_i \leq \overline{w} \\ 0 & \text{en otro caso} \end{cases}$$

Esto toma tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

(b) Ejecutar el algoritmo de sumas de prefijos con entrada Q, obteniendo un vector P. Esto toma tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores. Notemos que P_n es la cantidad de vértices que se actualizan al mismo tiempo que j.

(c) Determinar si $P_n > 1$. Si es así hay dos vértices que se actualizan antes o al mismo tiempo que j. Si además $x_j = 1$, entonces $R_j = 1$. Esto requiere tiempo (secuencial) $\mathcal{O}(1)$.

Todo esto esto requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores

3. Ejecutar el algoritmo de sumas de prefijos con entrada R, obteniendo un vector Y. Si $Y_n \neq 0$ significa que hay un vértice activo que tiene un vecino que se actualiza antes o al mismo tiempo que él. Esto toma tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.

Luego todo este paso requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 9 y 10: Asignar $S(u) = 1$ o $S(u) = 0$ a todos los nodos en V requiere tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

Sumando todo lo anterior, concluimos que este algoritmo requiere tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores.

Apéndice C

Análisis de la Complejidad del Algoritmo 5

A continuación entregaremos un análisis detallado de la complejidad del Algoritmo 5, paso a paso. Sea A la matriz de adyacencia del grafo G (de $n \times n$), x una configuración y v el vértice sobre el cual decidimos, w una palabra de G, donde un vértice i se actualiza antes que j si $w_i < w_j$, y T un vector de tamaño n tal que $T_i = 0$ si $i \in$ **ANDS** y $T_i = 1$ si $i \in$ **ORS**.

Paso 1: Calcular las componentes conexas de $G[ORS]$, el grafo inducido por los nodos del tipo **ORS**, usando el Algoritmo 1.

1. Usamos el algoritmo de sumas de prefijos con entrada T obteniendo un arreglo s donde

$$s_i = \sum_{j=0}^{i} T_j$$

   notemos que s representa las nuevas etiquetas de los vértices de G en el grafo $G[ORS]$, y s_n es igual al número de vértices de $G[ORS]$. Como vimos en el primer capítulo, ejecutar el algoritmo de sumas de prefijos requiere tiempo $\mathcal{O}(\log n)$ con $\mathcal{O}(n)$ procesadores.
2. Con s, T y A, construimos $A[ORS]$, la matriz de adyacencia de $G[ORS]$: Asignamos un espacio de memoria para $A[ORS]$ de $s_n \times s_n$, relleno de ceros. Para cada $i, j \in \{1, \ldots, n\}$ tal que $T_i = T_j = 1$, si $A_{i,j} = 1$, entonces $(A[ORS])_{s_i s_j} = 1$. Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n^2)$ procesadores, asignado un procesador para cada par de vértices del grafo.
3. Calculamos C, la salida del Algoritmo 1 con entrada $A[0]$. Esto se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ usando $\mathcal{O}(n^2)$ procesadores. Notemos que C es de largo s_n y es tal que $C_i = k$ si el vértice i pertenece a la componente conexa enumerada k.

Por lo tanto, todo el Paso 1 se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores.

Paso 2: Ejecutar en cada componente de $G[ORS]$ el Algoritmo 4 y guardar los resultados de todas las componentes en un vector S.

1. Definir S de tamaño s_n inicialmente rellena de ceros.

2. Para $k \in \{1 \dots s_n\}$ Construimos $A[k]$, $x[k]$, $w[k]$ que corresponden a la matriz de adyacencia, la configuración y la palabra de la componente k.
   (a) Definir un vector $Q[k]$ de largo s_n tal que $Q[k]_i = 1$ si $T_i = 1$ y $C_i = k$ (ie. cuando i es de tipo ORS y está en la componente k) y $Q[k]_i = 0$ en otro caso.
   (b) Obtener $s[k]$ el vector salida del algoritmo de sumas de prefijos con entrada $Q[k]$. Notemos que $s[k]$ son las etiquetas de los vértices de $G[ORS]$ en la componente k. Además $s[k]_{s_n}$ es la cantidad de vértices en la componente k de $G[ORS]$.
   Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
   (c) Con s, $s[k]$, T y $A[ORS]$, construimos $A[k]$, la matriz de adyacencia de la componente k de $G[ORS]$: Asignamos un espacio de memoria para $A[k]$ de $s[k]_{s_n} \times s[k]_{s_n}$, relleno de ceros. Para cada $i, j \in \{1, \dots, s_n\}$ tal que $C_i = C_j = k$, si $A[ORS]_{i,j} = 1$, entonces $(A[k])_{s[k]_i s[k]_j} = 1$.
   Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n^2)$ procesadores, asignado un procesador para cada par de vértices del grafo.
   (d) Definir $w[k]$ y $x[k]$ de tamaño $s[k]_n$ tal que si $i \in \{1 \dots n\}$ cumple que $T_i = 1$, $C_{s_i} = k$, entonces $w[k]_{s[k]_{s_i}} = w_i$ y $x[k]_{s[k]_{s_i}} = x_i$.
   Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores.
   (e) Ejecutar el algoritmo Algoritmo 4 con entradas $A[k]$, $x[k]$ y $w[k]$ y almacenar la salida en el vector $S[k]$ de largo $s[k]_{s_n}$.
   Esto se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^2)$ procesadores.

   Luego, todo lo anterior se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^3)$ procesadores.
3. Para $i \in \{1 \dots n\}$ tal que $T_i = 1$, $C_{s_i} = k$, escribimos $S_{s_i} = S[k]_{s[k]_{s_i}}$.
   Esto se puede hacer en tiempo $\mathcal{O}(1)$ con $\mathcal{O}(n)$ procesadores.

Luego todo este paso se puede hacer en tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^3)$.

Pasos 3 - 7.

1. Definir un vector Q de tamaño n, inicialmente en cero.
2. Para cada $i \in \{1, \dots, n\}$ tal que $T_i = 1$, $S_{s_i} = 0$ y $x_i = 1$.
   (a) Definir un vector $Q[i]$ de tamaño n, inicialmente en cero.
   (b) Para cada $j \in \{1, \dots, n\}$ tal que $T_j = 0$ y $A_{ij} = 1$, $T_j = 0$.
      i. Computar P, la salida del algoritmo de sumas de prefijos con entrada la fila j-ésima de A.
      Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
      ii. Computar r un vector tal que $r_k = 1$ si $A_{jk} = 1$ y $x_k = 1$ y $r_k = 0$ en otro caso.
      Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores.
      iii. Computar R la salida del algoritmo de sumas de prefijos con entrada r.
      Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
      iv. Si $R_n = P_n$, entonces $Q[i]_j = 1$. (P_n es el número de vecinos de j y R_n es el numero de vecinos activos de j)
      Esto se puede hacer en tiempo (secuencial) $\mathcal{O}(1)$.

   Todo esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n^2)$ procesadores.

(c) Computar $q[i]$ la salida del algoritmo de sumas de prefijos con entrada $Q[i]$.
Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.

(d) Si $q[i]_n \neq 0$, entonces $Q_i = 1$.

(e) Definir $P[i]$, la salida del algoritmo de sumas de prefijos con entrada la fila i-ésima de $A[ORS]$.
Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.

Todo esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n^3)$ procesadores.

3. Por cada $i \in \{1, \ldots, n\}$ tal que $T_i = 1$, $Q_i = 1$ y $P[i]_{s_n} \neq 0$, entonces para todo $j \in \{1, \ldots, n\}$ tal que $T_j = 1$ y $C_{s_j} = C_{s_i}$ hacemos $S_{s_j} = 1$ y $\overline{S}_{s_j} = 1$.
Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n^2)$ procesadores.
4. Por cada $i \in \{1, \ldots, n\}$ tal que $T_i = 1$ y $Q_i = 1$ y $P[i]_{s_n} = 0$ y hacemos $S_{s_i} = P$.
Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores.

Entonces, todo este paso se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n^3)$ procesadores.

Paso 8: Determinar si $T_v = 1$ se puede hacer en tiempo (secuencial) $\mathcal{O}(1)$.

Paso 9: Determinar si $S_{s_v} \neq 0$ se puede hacer en tiempo (secuencial) $\mathcal{O}(1)$.

Paso 10: Determinar si todos los vecinos de v son ORS

1. Calcular el vector Q de dimensión n tal que $Q_i = 1$ si $A_{iv} = 1$ y $T_i = 1$.
Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores.
2. Computar P, la salida del algoritmo de sumas de prefijos con entrada la fila numero v de A.
Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
3. Computar R, la salida del algoritmo de sumas de prefijos con entrada Q.
Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
4. Si $R_n = P_n$, entonces todos los vecinos de v son **ORS** (P_n es el número de vecinos de v y R_n es el numero de vecinos ORS de v)
Esto se puede hacer en tiempo (secuencial) $\mathcal{O}(1)$

Entonces, todo este paso se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.

Paso 11: Determinar si existen u_1, $u_2 \in N(v)$ tales que $S(u_1) = P$ y $S(u_2) = I$ o bien si existe $u \in N(v)$ tal que $S(u) = 0$.

1. Computar P, la salida del algoritmo de sumas de prefijos con entrada la fila numero v de A.
Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.
2. Calcular el vector Q de dimensión n tal que $Q_i = 1$ si $A_{iv} = 1$ y $S_{s_i} = 1$ o $S_{s_i} = I$.
Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores.
3. Calcular el vector R de dimensión n tal que $R_i = 1$ si $A_{iv} = 1$ y $S_{s_i} = 1$ o $S_{s_i} = P$.
Esto se puede hacer en tiempo $\mathcal{O}(1)$ usando $\mathcal{O}(n)$ procesadores.
4. Computar p, q, r, las ultimas componentes de la salida del algoritmo de sumas de prefijos con entrada P, Q y R respectivamente. Entonces p es el numero de vecinos de v, q es

el numero de vecinos con $S = 1$ o I y r es el numero de vecinos con $S = 1$ o P
Esto se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.

5. Verificar si $p = q$ o $p = r$.

Entonces, todo este paso se puede hacer en tiempo $\mathcal{O}(\log n)$ usando $\mathcal{O}(n)$ procesadores.

Sumando todo lo anterior, concluimos que este algoritmo requiere tiempo $\mathcal{O}(\log^2 n)$ con $\mathcal{O}(n^3)$ procesadores.

Apéndice D

Artículos

D.1. Artículo Publicado en *Theoretical Computer Science*

The complexity of the bootstraping percolation and other problems

Eric Goles[1,2]

Facultad de Ciencias y Tecnología, Universidad Adolfo Ibáñez, Santiago, Chile

Pedro Montealegre-Barba[1,2,3]

Departamento de Ingeniería Matemática, Universidad de Chile, Correo 3, Santiago 170-3, Chile.

Phone number: +56 9 90991631

Ioan Todinca

LIFO, Université d'Orléans, 45067 Orléans Cedex 2, France

Abstract

We study the problem of predicting the state of a vertex in automata networks, where the state at each site is given by the majority function over its neighborhood. We show that for networks with maximum degree greater than 5 the problem is **P**-Complete, simulating a monotone Boolean circuit. Then, we show that the problem for networks with no vertex with degree greater than 4 is in **NC**, giving a fast parallel algorithm. Finally, we apply the result to the study of related problems.

Keywords:
computational complexity, bootstrap percolation, parallel algorithms, P-Completeness, majority functions

Email addresses: `eric.chacc@uai.cl` (Eric Goles), `pmontealegre@dim.uchile.cl` (Pedro Montealegre-Barba), `Ioan.Todinca@univ-orleans.fr` (Ioan Todinca)

[1]This work was partially supported by Fondecyt 1100003 (Eric Goles, Pedro Montealegre-Barba), BASAL-CMM (Eric Goles), and Anillo ACT-88 (Eric Goles, Pedro Montealegre-Barba)

[2]This research was partially done while visiting the LIFO, Université d'Orléans

[3]Corresponding author

Preprint submitted to Theoretical computer science *July 5, 2012*

1. Introduction

Let V a finite set of sites or vertices. An *Automata Network* defined on V is a triple $\mathcal{A} = (G, Q, (f_i : i \in V))$, where $G = (V, E)$ is a simple undirected graph, Q is the set of states, which is assumed to be finite, and $f_i : Q^{|V|} \to Q$ is the transition function associated to the vertex i. The set $Q^{|V|}$ is called the set of configurations, and the automaton's global transition function $F : Q^{|V|} \to Q^{|V|}$, is constructed with the local transition functions $(G, Q, (f_i : i \in V))$ and with some kind of updating rule, for instance a synchronous or a sequential one.

In the special case where $Q = \{0, 1\}$ we say that the state 1 means that the vertex is *active*, while state 0 represents *passive* vertices. If $N(v)$ is the neighborhood of v, i.e. the set of vertices $\{u \mid uv \in E\}$, consider the following transition function:

$$f_i(x) = \begin{cases} 1 & if \quad x_i = 1 \\ 1 & if \quad \sum_{j \in N(i)} x_j > \frac{|N(i)|}{2} \quad and \quad x_i = 0 \\ 0 & if \quad \sum_{j \in N(i)} x_j \leq \frac{|N(i)|}{2} \quad and \quad x_i = 0 \end{cases}$$

In other words, a passive vertex becomes active if the strict majority of its neighbors are active, and thereafter never changes its state. We call *the strict majority rule* to the global transition function given by $(f_i : i \in V)$.

This is a special case of bootstrap percolation, which corresponds to a simple model introduced in the late 1970s to study properties of some magnetic materials [1]. More recently it has been used to model problems like disease spreading [2], spreading of alerts on distributed networks [3], sand pile formation [4], and others [5].

The classical theoretical studies of bootstrap percolation deals with the question of what are the minimum number of active (infected) sites in a graph in order to infect, say with a high probability, the whole structure. The results in this context are tricky and usually restricted to specific families of graphs (lattices, cubic graphs, etc) [6, 7]. Our approach is different and complementary with previous one, since we ask for the possibility that an specific site will be infected given an initial configuration, and if that can be quickly predicted.

One of the first to introduce the computation complexity to cellular automata (CA) and related systems has been C. Moore [8, 9, 10]. This have been done because, from one side, it's very hard to obtain characterizations of the the general dynamical behavior of a CA on time; and from other side,

they are related with parallel processing of information and algorithms, since some CA are able to emulate a universal Turing machine. Then, the idea is try to develop a bridge between this two aspects of the problem: this dynamical nature and its algorithmic capabilities.

The computational complexity of a prediction problem can be defined as the amount of resources, like time or space, needed to predict it. In this case, we consider two fundamental classes: **P**, the class of problems solvable in polynomial time on a serial computer, and **NC**, the class of problems solvable in poly-logarithmic time in a PRAM machine, with a polynomial amount of processors. In other words, **NC** is the class of problems which have a fast parallel algorithm. It is a well known conjecture that **NC** $\neq$ **P**, and if so, there exist "inherently sequential" problems that belong to **P** and do not belong to **NC**. The most likely to be inherently sequential are **P**-Complete problems, to which any other problem in **P** can be reduced (by an **NC**-reduction or a logarithmic space reduction). If any of these problems have a fast parallel algorithm, then **P**=**NC** [11, 8].

One such problem is the Circuit value problem (**CVP**), which consists in predicting the truth value of the output of a Boolean circuit, given the circuit and a truth value of its inputs. This problem is **P**-Complete since any deterministic Turing machine computation of length k can be converted into a Boolean circuit of depth k; thus polynomial time computations are equivalent to polynomial size and depth circuits; a complete analysis of this reduction can be found in [11]. The **CVP** remains **P**-Complete when the circuit is restricted to be monotone (that is, with AND and OR gates but without negation) and all vertices have in degree (fan in) and out degree (fan out) exactly two [11]. We call **M2CVP** this restriction of the **CVP** problem.

Now we give some definitions we need. Consider $G = (V, E)$ an undirected graph with vertex set V and edge set E. An edge between vertices x and y is simply denoted xy. We denote $n = |V|$ and $m = |E|$. Two vertices are *connected* if $u = v$ or there exists a path $P = x_1, x_2, \ldots, x_k$ such that $u = x_1$, $v = x_k$ and $x_i x_{i+1} \in E$, for all $1 \leq i \leq k-1$. This relation is an equivalence relation on V, and hence partitions V into equivalence classes $\{V_j\}_{j=1}^{l}$. The subgraphs $G_j = (V_j, E_j)$, where $E_j = \{xy \in E \mid x, y \in V_j\}$, are called the **connected components** of G. If G has a unique connected component, we say that G is connected.

Let $G = (V, E)$ be a connected graph. We define a relation R on E as follows. Given two edges e and g, eRg if and only if $e = g$ or e and g are in a common simple cycle. It is straightforward to check that R is an

equivalence relation, and hence partitions E into equivalence classes $\{E_i\}_{i=1}^s$. Let $V_i = \{v \in V | v$ is the end point of some edge in $E_i\}$. Then, the subgraphs $G_i = (V_i, E_i)$ are called **biconnected components** of G (See Figure 1).

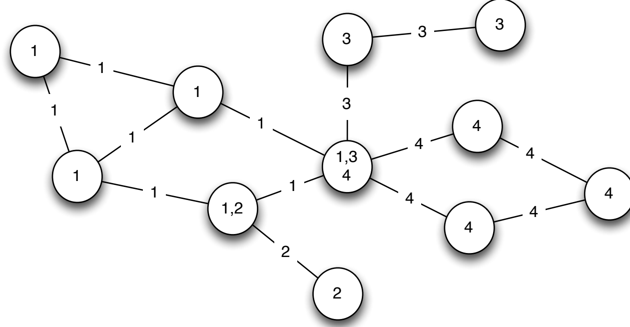

Figure 1: Biconnected components of a graph. Edges with the same label are in the same component, notice that a vertex may appear in several biconnected components.

It is important to remark that there are **NC** algorithms for finding connected and biconnected components of a given graph G [12].

Given $G = (V, E)$, and $v \in V$, the degree $d_G(v)$ of a vertex v is the number of its neighbors in G:

$$d_G(v) = |N(v)|.$$

and the number $\Delta(G)$ is the maximum degree:

$$\Delta(G) = \max_{v \in V}\{d_G(v)\}.$$

As usual, the subscript will be omitted when no confusion is possible.

In the next section, we discuss the complexity of the bootstrap percolation problem using the strict majority rule. In Section 4 we enlarge the results to other types of dynamics (simple majority, threshold rules). We end our paper by some open questions.

2. Complexity of bootstraping percolation

Consider the following problem: given an initial configuration $x(0) \in \{0,1\}^{|V|}$ and a vertex $v \in V$, initially passive ($x_v(0) = 0$), determine if there

exists a time $T > 0$ such that v is active ($x_v(T) = 1$), where $x(t) = F(x(t-1))$ and F is some synchronous global transition function (for example, the strict majority rule). We call this decision problem **PER**.

Our main result is the following:

Theorem 1. *For the strict majority rule:*

1. *On the family of graphs $G = (V, E)$ such that $\Delta(G) \geq 5$, the problem* **PER** *is* ***P****-Complete.*
2. *On the family of graphs $G = (V, E)$ such that $\Delta(G) \leq 4$, the problem* **PER** *is in* ***NC****.*

We prove here the first part of the theorem, the second is postponed to the next section.

Proof. (of Theorem 1.(1))

Notice that since the active vertices remains active, in at most $|V|$ steps, the dynamic gets into a fixed point. Then, **PER** can be decided in $\mathcal{O}(n)$, so is in **P**.

To prove that **PER** is **P**-Complete, we will reduce the restricted case circuit value problem **M2CVP** to **PER**. Since **M2CVP** is **P**-Complete, if the reduction uses only a logarithmic space, then **PER** will be **P**-Complete.

To reduce **M2CVP** to **PER**, we will build, given a monotone circuit, a graph that simulate its gates, where active vertices represent Boolean value *true*, and passive vertices represent value *false*. Since the circuit is monotone, we only have to simulate the AND and OR gates.

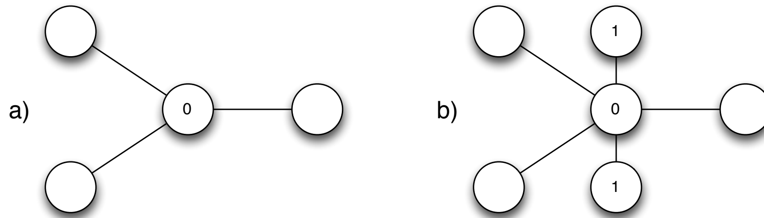

Figure 2: a) The AND gate b) The OR gate

The AND gate (Figure 4 (a)) is simulated by an initially passive middle vertex with degree 3, two of them will be the inputs and the other the output.

By the strict majority rule, this vertex will become active only if two more neighbors become active. In a similar way, the OR gate (Figure 4 (b)) has an initially passive middle vertex, with degree 5 and two neighbors initially active, then, this vertex will become active if any passive neighbor becomes active.

In order to avoid problems with the flow of information, Figure 3 (a) shows the construction of a *diode* which only allows the flow of information in 'one way': If the left vertex is active, then the right vertex will become active. But if the right vertex is active, the left vertex remains in its state. We simplify the notation with an arrow Figure 3(b).

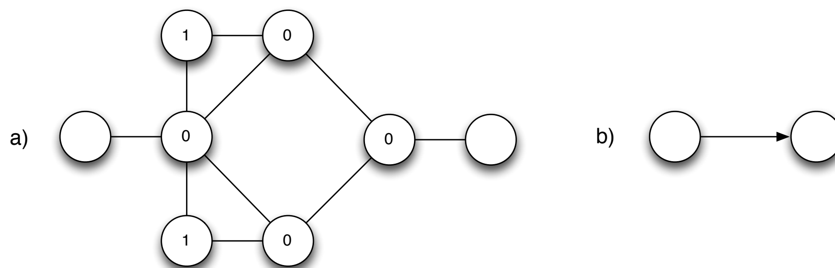

Figure 3: a) The diode b) Symbol of a diode

We modify the original gates, in order to have precise definitions of inputs and outputs. Figure 4 shows the constructions of AND an OR gates with diodes.

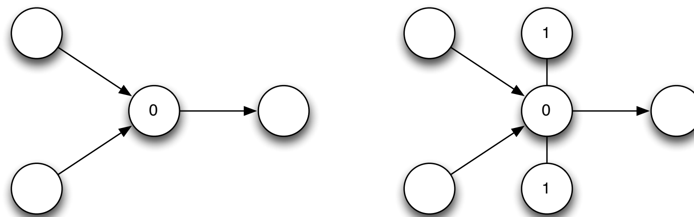

Figure 4: Gates with diodes

Remember that in our restricted version of the circuit value problem **M2CVP**, every gate of the original circuit must have fan-out exactly 2, so

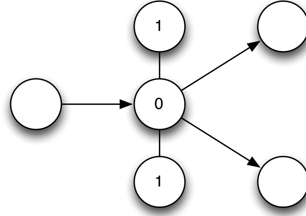

Figure 5: OR gate 'backwards' to multiply the information of an output

we use OR gates 'backwards' in order to multiply the information of the output, as in Figure 5.

Then, given ϕ a monotone Boolean circuit and an input I, we can build a graph G that simulates ϕ in I. Letting active the input nodes that are true, passive the rest, and considering v as the output of the circuit, $(G, x(0), v)$ belongs to **PER** if and only if (ϕ, I) belongs to **M2CVP**.

This reduction can be done by a Turing machine in logarithmic space: reading the input (a monotone circuit) of size n, the machine only has to determine which construction corresponds to each gate ($\mathcal{O}(1)$ space), and then determine the connectivity of every gate ($\mathcal{O}(\log n)$ space). Then, the whole construction requires $\mathcal{O}(\log n)$ space. □

Remark 1. Note that in the construction of the OR gate and the diode, we need vertices with degree 5. We will show that a construction that uses vertices with degree less than 5 is not possible unless $\mathbf{P} = \mathbf{NC}$.

3. Case $\Delta(G) \leq 4$

To prove the second part of the theorem, it will be useful to work with the vertices that are initially passive, so we will call $G[0] = (V[0], E[0])$ the induced subgraph of initially passive vertices of G:

$$\begin{aligned} V[0] &= \{u \in V \mid x_u(0) = 0\} \\ E[0] &= \{uw \in E \mid u, w \in V[0]\} \end{aligned}$$

Definition 1. *A community in G is a subset of nodes $X \subset V$ each of which has at least as many neighbors in X as in $V \backslash X$, i.e. for every $v \in X$, $|N(v) \cap X| \geq |N(v) \cap (V \backslash X)|$.*

Definition 2. *Given an initial configuration $x(0)$, we say that a passive vertex v is stable if $x_v(t) = 0$ $\forall t > 0$.*

Remark 2. A vertex is stable if and only if it belongs to a community X of initially passive vertices.

Remark 3. In the case of $\Delta(G) \leq 4$, every cycle is a community, so every cycle in $G[0]$ is stable.

Notice that even if G is connected, $G[0]$ may not be. Given $v \in V$, let $C[v]$ the connected component of v in $G[0]$, and $G[0, v] = (C[v], E[0, v])$ the subgraph of G induced by $C[v]$, that is:

$$E[0, v] = \{uw \in E \mid u, w \in C[v]\}$$

Lemma 2. *Let G with $\Delta(G) \leq 4$. A vertex v is stable if and only if it exists a path P of $G[0, v]$ that contains v and, if u is one of its ends, then*

1. *u belongs to a cycle in $G[0]$, or*
2. *$d_G(u) \leq 2$.*

Proof. Suppose first that v is stable. We know that at most after $T = |V|$ steps, the dynamic given by x gets into a fixed point, this is $x(T+t) = x(T)$ $\forall t \geq 0$. Let $G'[0, v]$ the connected component of passive vertices of G in the step T that contains v. Notice that $G'[0, v]$ is a subgraph of $G[0, v]$. Take P the longest path in $G'[0, v]$ that contains v. Let u be an end of P. Suppose first that u has only one neighbor in $G'[0, v]$, since u is stable, u has at most one more neighbor in G, hence $d_G(u) \leq 2$. Suppose now that u has more than one neighbor in $G'[0, v]$. Since P is the longest path that contains v in $G'[0, v]$, both neighbors of u must belong to P. Then u belongs to a cycle in $G[0]$.

Let us prove the converse. Let P be a path in $G[0, v]$, with $v \in P$ and its ends satisfying (1) or (2). Let X be the set of vertices of P, plus, for each end-point u of P satisfying condition (1), the corresponding cycle C_u of $G'[0, v]$ containing u. Note that, since $\Delta(G) \leq 4$, X forms a community of initially passive vertices. Indeed, each interior vertex of P (P minus its ends) has at least half of its neighbors in X, and the same holds for the vertices of the cycles added to X. If an endpoint u of P satisfies condition (2), is also has at least half of its neighbors in X, thus X is a community. The proof follows from Remark 2. $\square$

In the introduction we mentioned that finding connected and biconnected components of a given graph can be done in **NC**, both of them in $\mathcal{O}(\log^2 n)$ time, using a total of $\mathcal{O}(n^2)$ processors. The proof of these propositions, which consists in fast parallel algorithms, can be found in [12]. In order to make use of these results, we will need to know the inputs and the outputs of these algorithms, detailed in Algorithms 1 and 2.

Algorithm 1 Connected components

Require: The $n \times n$ adjacency matrix A of an undirected graph.

Ensure: An array D of size n such that $D(i)$ is equal to the smallest vertex in the connected component of i.

Algorithm 2 Biconnected components

Require: The $n \times n$ adjacency matrix A of an undirected connected graph.

Ensure: An array B such that $B(e) = B(g)$ if and only if e and g are in the same biconnected component.

In some steps of our algorithm, we will need to compute the prefix sum of an array. Given a group $G = (X, +)$, the prefix sum of a sequence of numbers $x_0, x_1, ...x_n \in X$ is a second sequence of numbers $y_0, y_1, ..., y_n \in X$, where $y_k = \sum_{i=0}^{k} x_i$. JáJá [12] provides a fast parallel algorithms that uses $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ processors to calculate the prefix sum of a given set.

Now we are ready for proving the second part of the Theorem 1. (2):

Proof. From Lemma 2, to decide **PER** it is enough to determine if v belongs to a path in $G[0, v]$ with ends satisfying (1) or (2). This must be done quickly in parallel, so we can not apply regular search methods like Depth-First Search to achieve this.

We can obtain $G[0, v]$ quickly in parallel with Algorithm 1 and a prefix sum algorithm. First, to build $G[0]$, we calculate its adjacency matrix $A[0]$ from A and the initial configuration $x(0)$: letting $\overline{x} = 1 - x(0)$ and then applying the prefix sum algorithm to it, we obtain a vector s such that s_n will be the number of initially passive vertices, and for any i, j such that $x_i = x_j = 0$, if $A_{i,j} = 1$ then $A[0]_{s_i s_j} = 1$. This can be done in $\mathcal{O}(1)$ time using $\mathcal{O}(n^2)$ processors.

Then, with Algorithm 1, we calculate the connected components of $G[0]$ in $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^2)$ processors. With the array given by Algorithm

1, we build $A[0,v]$ the adjacency matrix of $G[0,v]$. Notice that if v is isolated in $G[0,v]$, then v is stable. Suppose now that v is not isolated in $G[0,v]$.

A biconnected component that contains more than two vertices has the property that any two vertices are on a same cycle, and conversely the vertices of any cycle are in a same biconnected component. It follows that from the array given by Algorithm 2, we can obtain which vertices of $G[0,v]$ satisfy (1). Using Algorithm 2, this can be done quickly in parallel.

To calculate the vertices that satisfy (2) we obtain from A the vector D of dimension $|V|$, where $D_u = d_G(u)$. This can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ processors with a prefix sum algorithm. Then, any vertex u of $G[0,v]$ where $D_u \leq 2$ satisfies (2).

To determine if v belongs to a path with ends satisfying (1) or (2), we build another graph $\overline{G} = (\overline{V}, \overline{E})$ from $G[0,v]$ and a new vertex called ∞, where:

$$\begin{aligned} \overline{V} &= V[0,v] \cup \{\infty\} \\ \overline{E} &= E[0,v] \cup \{u\infty \mid u \in V[0,v] \text{ satisfying (1) or (2)}\} \end{aligned}$$

Notice that v is stable if and only if there are two different paths from v to ∞. Then, v is stable if and only if there is a cycle in $\overline{G}$ that contains v and ∞. In conclusion, the last steps of the algorithm must calculate the biconnected components of $\overline{G}$, and then decide if v and ∞ are in the same component.

The result is Algorithm 3.

The correctness of the algorithm is obtained as follows: if v is stable, then by Lemma 2, v belongs to a path P in $G[0,v]$ which ends satisfy (1) or (2). By definition of $\overline{A}$, the ends of P are connected to the vertex ∞, then, there exists a cycle C that contains v and ∞. Conversely, if there is a cycle in $\overline{A}$ that contains v and ∞, then by definition of $\overline{A}$, there is a path P in $G[0,v]$ which ends satisfy (1) or (2), and then, by Lemma 2, v is stable. In the Appendix there is a deep analysis of the complexity of this algorithm, which concludes that it can be done in a PRAM machine in $\mathcal{O}(\log^2 n)$ time, with $\mathcal{O}(n^4)$ processors. □

4. Related problems

We consider here two other variants of dynamics, the simple (non strict) majority voting and a dynamic with a fixed threshold.

Algorithm 3 PER

Require: The $n \times n$ adjacency matrix A of a undirected graph $G = (V, E)$, an initial configuration $x(0)$ and a vertex $v \in V$

Ensure: Accept if $\{G, x(0), v\}$ belongs to **PER** and Reject otherwise.

1: Calculate $A[0]$, the adjacency matrix of $G[0]$ using $x(0)$ and a prefix sum algorithm.
2: Calculate $A[0, v]$, the adjacency matrix of $G[0, v]$, using $A[0]$ in Algorithm 1
3: **if** v is isolated in $A[0, v]$ **then**
4: Accept, and quit.
5: **end if**
6: Calculate B the output of Algorithm 2 in $A[0, v]$.
7: Calculate the cycles in $A[0, v]$ from B and store in a Boolean array C where $C_u = 1$ iff u belongs to a cycle in $A[0, v]$.
8: Calculate D, the vector of degree of G.
9: Define $\overline{A}$, the adjacency matrix of $\overline{G}$, using $A[0, v]$, D and C.
10: Calculate $\overline{B}$ the output of Algorithm 2 in $\overline{A}$.
11: Using B determine if v and ∞ are in a cycle. Reject if they do, Accept in other case.

4.1. **Simple majority:**

Consider the rule given by the following transition function:

$$f_i(x) = \begin{cases} 1 & if \quad x_i = 1 \\ 1 & if \quad \sum_{j \in N(i)} x_j \geq \frac{|N(i)|}{2} \quad and \quad x_i = 0 \\ 0 & if \quad \sum_{j \in N(i)} x_j < \frac{|N(i)|}{2} \quad and \quad x_i = 0 \end{cases}$$

We call it *the simple majority rule* and we have a similar result:

Theorem 3. *For the simple majority rule:*

1. *On the family of graphs $G = (V, E)$ such that $\Delta(G) \geq 4$, the problem* **PER** *is **P**-Complete.*
2. *On the family of graphs $G = (V, E)$ such that $\Delta(G) \leq 3$, the problem* **PER** *is in **NC**.*

For the proof of this theorem we will need a version of Lemma 2 for this rule.

Lemma 4. *Let G with $\Delta(G) \leq 3$. A vertex v is stable for the non strict majority if and only if exists a path P of $G[0, v]$ that contains v and, if u is one of its ends, then*

1. *u belongs to a cycle in $G[0]$, or*
2. *$d_G(u) = 1$.*

The proof of this lemma is analogous to the proof of Lemma 2.

Proof. (of Theorem 3)

1. Figures 6 show the gadgets used to simulate a monotone circuit with the non strict majority rule. Notice that in this case we only need vertices with degree 4.
2. For the second part, we consider Lemma 4 and modify Algorithm 3 only in the definition of $\overline{A}$: We connect to ∞ vertices in $G[0, v]$ with degree 1 and vertices that belong to cycles in $G[0, v]$.

□

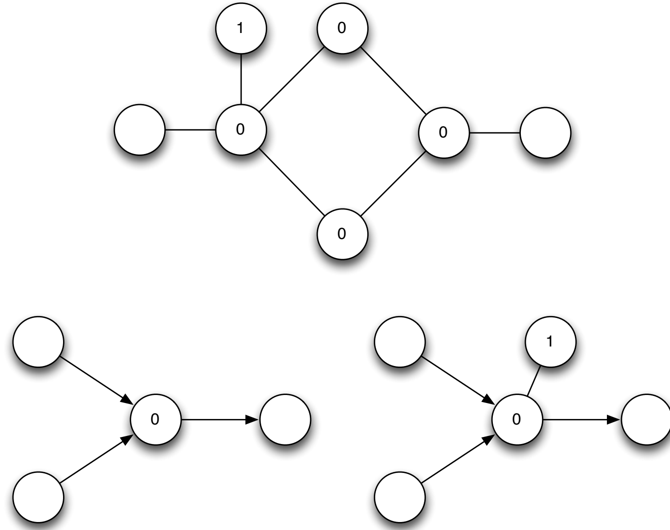

Figure 6: Top: The diode for the non strict case, Bottom: the AND and OR gates for the same rule.

4.2. θ-Rule

Given $\theta > 0$ and $G = (V, E)$ such that $\delta(G) \geq \theta$ (i.e. $d_G(v) \geq \theta \ \forall v \in V$). Consider the rule given by the following transition function:

$$f_i(x) = \begin{cases} 1 & if \quad x_i = 1 \\ 1 & if \quad \sum_{j \in N(i)} x_j > \theta \quad and \quad x_i = 0 \\ 0 & if \quad \sum_{j \in N(i)} x_j \leq \theta \quad and \quad x_i = 0 \end{cases}$$

We call it *the* θ*-rule* and the result in this case is:

Theorem 5. *For the* θ*-rule:*

1. *On the family of graphs* $G = (V, E)$ *such that* $\Delta(G) \geq \theta + 3$*, the problem* **PER** *is* ***P****-Complete.*
2. *On the family of graphs* $G = (V, E)$ *such that* $\Delta(G) \leq \theta + 2$*, the problem* **PER** *is in* ***NC****.*

Remark 4. Notice that in this case, a passive node becomes active if it has at least $\theta + 1$ active neighbors. Then, in the case of $\Delta(G) \leq \theta + 2$, a cycle in $G[0]$ will be stable.

For the proof of theorem 5 we will also need a version of Lemma 2 for this rule.

Lemma 6. *Let G with $\Delta(G) \leq \theta + 2$. A vertex v is stable for the non strict majority if and only if exists a path P of $G[0, v]$ that contains v and, if u is one of its ends, then*

1. *u belongs to a cycle in $G[0]$, or*
2. *$d_G(u) \leq \theta + 1$.*

The proof of this lemma is analogous to the proof of Lemma 2.

Proof. (of Theorem 5)

1. Like in the other cases, we will simulate a monotone Boolean circuit with a graph G. Since $\delta(G) \geq \theta$, we have to change the constructions of theorem 1(1), in order to obtain vertices with degree at least θ. The solution is to take the gadgets of Theorem 1.(1), and connect the vertices to p other active vertices, that are connected with each other, where p is some convenient number (obviously depending on θ). In other words, we connect every vertex of the gadgets to every vertex of a complete graph of active vertices.
   Figure 7 shows how to simplify the notation. We represent the whole complete subgraph K and the $|K|$ connections by a squared vertex with the number $|K|$ inside.

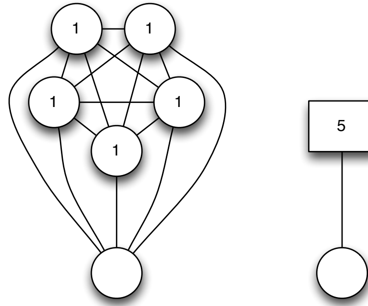

Figure 7: We represent a complete subgraph K, in this case K_5, and the $|K|$ connections with a single squared vertex with the number $|K|$ inside with one edge.

   Using this, it is easy to build the new gadgets, shown in Figure 8. Notice that in the construction of the OR gate and the diode, the degree required is $\theta + 3$.

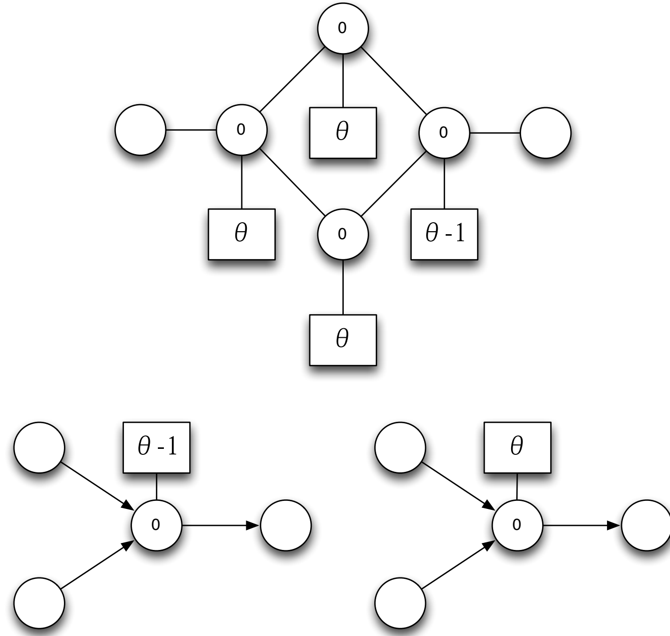

Figure 8: Top: the Diode for the θ rule, Bottom: The AND and OR gates for the same rule.

2. In this part, we consider Lemma 6 and modify Algorithm 3 only in the definition of $\overline{A}$: We connect to ∞ vertices in $G[0, v]$ with degree $\leq 1+\theta$ and vertices that belong to cycles in $G[0, v]$.

□

5. Conclusion

We have discussed several variants of majority voting rules on Boolean automata networks, and we have shown that the complexity of the bootstrap percolation problem depends on the maximum degrees of the input graphs. When this maximum degree is small, the problem is in **NC**, and for large degrees the problem becomes **P**-Complete. Clearly, our bounds are tight. Our **P**-completeness proofs use a reduction from a special case of **CVP** problem, which is **P**-Complete on arbitrary circuits.

A natural extension of our results would be to consider special graph classes.

A straightforward question is: What happens with bootstrap percolation **PER** in the planar case? It is obvious that the results for $\Delta(G) \leq 4$ remains true. So we are thinking of the case when the max degree is at least 5. We point out that monotone circuit value problem **MCVP** becomes **NC** when restricted to planar Boolean circuits, so we cannot use our reduction technique for the planar case.

References

[1] J. Chalupa, P. L. Leath, G. R. Reich, Bootstrap percolation on a bethe lattice, J Phys C 12 (1979) L31–L35.

[2] J. Balogh, G. Pete, Random disease on the square grid, Random Struc. & Alg 13 (1998) 3–4.

[3] M. Treaster, W. Conner, I. Gupta, K. Nahrstedt, Contagalert: Using contagion theory for adaptive, distributed alert propagation., in: NCA'06, 2006, pp. 126–136.

[4] S. Manna, Abelian cascade dynamics in bootstrap percolation, Physica A: Statistical Mechanics and its Applications 261 (3-4) (1998) 351 – 358. doi:10.1016/S0378-4371(98)00346-X.

[5] R. A. Meyers, Encyclopedia of Complexity and Systems Science - v. 1-10, 1st Edition, Springer Publishing Company, Incorporated, 2009.

[6] I. Rapaport, K. Suchan, I. Todinca, J. Verstraete, On dissemination thresholds in regular and irregular graph classes, Algorithmica 59 (2011) 16–34, 10.1007/s00453-009-9309-0.
URL `http://dx.doi.org/10.1007/s00453-009-9309-0`

[7] R. Carvajal, M. Matamala, I. Rapaport, N. Schabanel, Small alliances in graphs, in: MFCS'07, 2007, pp. 218–227.

[8] C. Moore, Majority-Vote Cellular Automata, Ising Dynamics, and P-Completeness, Journal of Statistical Physics 88 (1997) 795–805.

[9] C. Moore, M. G. Nordahl, Predicting lattice gases is p-complete, Santa Fe Institute Working Paper 97-04-034.

[10] D. Griffeath, C. Moore, Life without death is p-complete, Complex Systems 10 (1997) 437–447.

[11] R. Greenlaw, H. Hoover, W. Ruzzo, Limits to parallel computation: P-completeness theory, Oxford University Press, 1995.

[12] J. JáJá, An introduction to parallel algorithms, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[13] E. Goles, S. Martinez, Neural and automata networks: dynamical behavior and applications, Kluwer Academic Publishers, 1990.

[14] J. Jájá, S. Kosaraju, Parallel algorithms for planar graph isomorphism and related problems, IEEE Transactions on Circuits and Systems 35 (3).

[15] A. L. Delcher, S. R. Kosaraju, An nc algorithm for evaluating monotone planar circuits, SIAM J. Comput. 24 (1995) 369–375. doi:10.1137/S0097539792226278.

Appendix A. Analysis of the complexity of Algoritm 3

Now we will give a detailed analysis of the complexity of Algorithm PER, step by step:

Step 1: Calculate $A[0]$, the adjacency matrix of $G[0]$ using $x(0)$ and a prefix sum algorithm.

1. We first calculate $\overline{x(0)}$ where

$$\overline{x_i(0)} = \begin{cases} 0 & if \quad x_i(0) = 1 \\ 1 & if \quad x_i(0) = 0 \end{cases}$$

   This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n)$ processors.
2. Then we use the prefix sum algorithm on $\overline{x(0)}$ to calculate an array s, where

$$s_i = \sum_{j=0}^{i} \overline{x_j(0)}$$

   then s represents the new labels of the vertices of G in the graph $G[0]$, and s_n equals the number of vertices of $G[0]$. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n)$ processors.

3. With s, $x(0)$ and A, build $A[0]$, the adjacency matrix of $G[0]$: The dimensions of $A[0]$ are $s_n \times s_n$. For any i, j such that $x_i(0) = x_j(0) = 0$, if $A_{ij} = 1$ then $(A[0])_{s_i s_j} = 1$, otherwise $(A[0])_{s_i s_j} = 0$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^2)$ processors.

Step 2: Calculate $A[0, v]$, the adjacency matrix of $G[0, v]$, using $A[0]$ in Algorithm 1

1. First calculate C, the output of Algorithm 1 on input $A[0]$. This can be done in $\mathcal{O}(\log^2 n)$ time with $\mathcal{O}(n^2)$ processors.
2. Remember that s_v is the label of v in $A[0]$. Then C_{s_v} is the label of the connected component that contains v. Calculate $\overline{C}$, an array with the same dimensions as C such that $\overline{C}_i = 1$ if $C_i = C_{s_v}$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n)$ processors.
3. Use the prefix sum algorithm on $\overline{C}$, obtaining S, where

$$S_i = \sum_{j=0}^{n} \overline{C}_j$$

then S represents the new labels of the vertices of $G[0]$ in the graph $G[0, v]$, and S_n equals the number of vertices of $G[0]$. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n)$ processors.
4. With S, $\overline{C}$ and $A[0]$, build $A[0, v]$, the adjacency matrix of $G[0, v]$: The dimensions of $A[0, v]$ are $S_n \times S_n$, and if $A[0]_{ij} = 1$, with $\overline{C}_i = \overline{C}_j = 1$, then $(A[0, v])_{S_i S_j} = 1$, and $(A[0, v])_{S_i S_j} = 0$ in the other case. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^2)$ processors.

Step 3,4,5: Determine if v is isolated; can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n)$ processors.

Step 6: Calculate B the output of Algorithm 2 in $A[0, v]$. This can be done in $\mathcal{O}(\log^2 n)$ time with $\mathcal{O}(n^2)$ processors.

Step 7: Calculate the cycles in $A[0, v]$ from B and save it in an array C where $C_u = 1$ if u belongs to a cycle in $A[0, v]$.

1. Calculate M, the maximum value in B and N the length of B. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n)$ processors. Note that $M, N \leq n^2$.

2. Compute the matrix K of dimensions $M \times N$, where $K_{ij} = 1$ if $B_j = i$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^4)$ processors.
3. Calculate the array L where $L_i = \sum_{j=1}^{N} K_{ij}$, i.e. L_i is the number of edges in component i. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n^2)$ processors.
4. Compute the array C: let $g = uv$, (i.e. $A[0,v]_{u,v} = 1$) with $B_g = k$ and $L_k \neq 1$ then $C_u = C_v = 1$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^2)$ processors.

Step 8: Calculate D, the vector of degrees of G. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n^2)$ processors.

Step 9: Define $\overline{A}$, the adjacency matrix of $\overline{G}$, using $A[0,v]$, D and C. Let m the dimension of $A[0,v]$. $\overline{A}$ has dimensions $m+1 \times m+1$, because $\overline{A}$ is the same as $A[0,v]$ with one more row and column, corresponding to the vertex ∞. Then, to calculate $\overline{A}$, we copy $A[0,v]$ and $\overline{A}_{i,m+1} = \overline{A}_{m+1,i} = 1$ if $D_i \leq 2$ or $C_i = 1$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^2)$ processors.

Step 10: Calculate $\overline{B}$ the output of Algorithm 2 in $\overline{A}$. This can be done in $\mathcal{O}(\log^2 n)$ time with $\mathcal{O}(n^2)$ processors.

Step 11: Using B, determine if v and ∞ are in a cycle.

1. Calculate M, the maximum value in B and N the length of B. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n)$ processors. Note that $M, N \leq n^2$.
2. Compute arrays I, V with length M: Let $g \in E(\overline{G})$, with $B(g) = k$. If $\infty \in g$, then $I_k = 1$, and if $v \in g$, then $V_k = 1$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^2)$. Notice that V_k (resp. I_k) is 1 if v (resp. ∞) is in a edge in component k.
3. Calculate the matrix K of dimensions $M \times N$, where $K_{ij} = 1$ if $B_j = i$. This can be done in $\mathcal{O}(1)$ time with $\mathcal{O}(n^4)$ processors.
4. Compute the array L where $L_i = \sum_{j=1}^{N} K_{ij}$, i.e. L_i is the number of edges in component i. This can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n^2)$ processors.
5. Calculate $I + V$. If for some k, $(I+V)_k \geq 2$ and $L_k > 1$ then v and ∞ are in the same (non trivial) component, and so in a cycle.

In conclusion, the whole algorithm can be executed in $\mathcal{O}(\log^2 n)$ time, with $\mathcal{O}(n^4)$ processors.

D.2. Artículo enviado a *Advances in Applied Mathematics*

The complexity of majority rule on planar graphs

Eric Goles[1,2]

Facultad de Ciencias y Tecnología, Universidad Adolfo Ibáñez, Santiago, Chile

Pedro Montealegre-Barba[1,2]

Departamento de Ingeniería Matemática, Universidad de Chile, Correo 3, Santiago 170-3, Chile.

Abstract

We study the complexity of the majority rule on planar automata networks. We reduce a special case of the Monotone Circuit Value Problem to the prediction problem of determining if a vertex of a planar graph will change its state when the network is updated with the majority rule.

Keywords:
Automata networks, computational complexity, Majority, P-Completeness, NC, planar graph

---

1. Introduction

Let $G = (V, E)$ be a simple undirected finite graph, where V is the set of vertices and E the set of edges. An *Automata Network* is a triple $\mathcal{A} = (G, Q, (f_i : i \in V))$, where, Q is the finite set of states and $f_i : Q^{|V|} \to Q$ is the transition function associated to the vertex i. The set $Q^{|V|}$ is called the set of configurations, and the automaton's global transition function $F : Q^{|V|} \to Q^{|V|}$, is constructed from the local functions $(G, Q, (f_i : i \in V))$ such that $(F(x))_i = f_i(x)$.

Email addresses: `eric.chacc@uai.cl` (Eric Goles), `pmontealegre@dim.uchile.cl` (Pedro Montealegre-Barba)

[1]This work was partially supported by Fondecyt 1100003 (Eric Goles, Pedro Montealegre-Barba), BASAL-CMM (Eric Goles), and Anillo ACT-88 (Eric Goles, Pedro Montealegre-Barba)

[2]This research was partially done while visiting the LIFO, Université d'Orléans

In the special case where $Q = \{0, 1\}$ we say that the state 1 means that the vertex is *active*, while state 0 represents *passive* vertices. Let $N(v)$ be the neighborhood of v, i.e. the set of vertices $\{u \mid uv \in E\}$, consider the following transition function:

$$f_i(x) = \begin{cases} 0 & if \quad \sum_{j \in N(i)} x_j < \frac{|N(v)|}{2} \\ x_i & if \quad \sum_{j \in N(i)} x_j = \frac{|N(v)|}{2} \\ 1 & if \quad \sum_{j \in N(i)} x_j > \frac{|N(v)|}{2} \end{cases}$$

In other words, a vertex takes the state of the majority of its neighbors, and in case of a tie, keeps its state. We call *the majority rule* to the global transition function given by $(f_i : i \in V)$.

In this paper we will study the computational complexity of predicting the state of a vertex in a given graph from an initial configuration. Let G be a graph, and x a configuration of G. A trajectory obtained from x is the set $\{x(t) : t \geq 0\}$ where $x(0) = x$, F is the majority rule, and $x(t+1) = F(x(t))$. We define **PER** as the decision problem which consists in predicting if a single vertex in a graph will change its state, in the trajectory given by some initial configuration. Formally:

PER: Let $G = (V, E)$ be an undirected graph, $x \in \{0, 1\}^{|V|}$ an initial configuration of G, $v \in V$ a vertex initially passive ($x_v = 0$), and F some global transition function. Determine if there exists $T > 0$ such that $X(T)_v = 1$.

The computational complexity of a decision problem can be defined as the amount of resources, like time or space, needed to predict it. In this case, we consider two fundamental classes: **P**, the class of problems solvable in polynomial time on a serial computer; and **NC**, the class of problems solvable in poly-logarithmic time with a polynomial amount of processors in a PRAM machine. It is easy to prove that **NC** $\subset$ **P**. Informally **NC** is known as the class of problems which have a fast parallel algorithm [1]. It is a well known conjecture that **NC** $\neq$ **P**, and if so, there exist "inherently sequential" problems, this is, that belong to **P** and do not belong to **NC**. The most likely to be inherently sequential are **P**-Complete problems, to which

any other problem in **P** can be reduced by an **NC**-reduction or a logarithmic space reduction. If any of these problems have a fast parallel algorithm, then **P**=**NC** [1, 2].

One such problem is the *Circuit Value Problem* (**CVP**), which consists in predicting the truth value of an output of a Boolean circuit, given a truth value of its inputs. This problem is **P**-Complete since any deterministic Turing machine computation of length k can be converted into a Boolean circuit of depth k; thus polynomial time computations are equivalent to polynomial size and depth circuits; a complete analysis of this reduction can be found in [1].

We define the *layer* of a gate v, denoted $layer(v)$, to be zero for input gates and for the other gates, is the length of the longest path from an input to v. A circuit is *synchronous* if all inputs to a gate v come from vertices at layer $[\text{layer}(v) - 1]$. The **CVP** remains **P**-Complete when the circuit is restricted to be synchronous, monotone (that is, with AND and OR gates but without negation) and all vertices have in degree (fan in) and out degree (fan out) exactly two, with the obvious exceptions of the input with in degree zero, and the outputs with out degree zero [1]. We call **S2MCVP** this restriction of the **CVP**.

A different case is when we restrict the circuit to be planar, because *Planar monotone circuit value problem* or **PMCVP**, the restriction of **CVP** to a planar and monotone circuit, is in NC [3].

C. Moore in [2] studied the complexity of the majority rule in the d-dimensional lattice, with $d \geq 3$. He proved that in that kind of graphs **PER** is **P**-Complete, but leaves as an open question what happens in the 2-dimensional lattice.

In a previous work [4] we studied the problem **PER** with the bootstrap percolation rule, this is, passive vertices become active if the majority of its neighbors are active, but once a vertex is active, it never changes again. We found that the complexity of this problem depends on the characteristics of the graph, moreover, the problem is **P**-Complete in the general case, but in **NC** if we restrict the vertices to have degree less or equals than 4. In that paper we left as an open question what happens in the case when the graph is restricted to a planar one.

We approach this open problem proving that, for the majority rule and the family of planar graphs, **PER** is **P**-Complete. Then the main theorem of this article is:

Theorem 1. *Over the family of planar graphs, the problem* **PER** *associated to the majority rule is* **P**-*Complete.*

In the following sections we will give a proof of this theorem, which follows from three lemmas. First, in section 2, we will show the membership in **P** for **PER** with the majority rule. Then, we will give a proof of the **P**-Hardness of **PER** with the majority rule. Finally, we will show a **P**-Hardness proof for the planar case. At the end we give some conclusions.

2. The majority rule is in P.

To prove the membership in **P** of the majority rule, we notice that from any configuration x of G, the next state of any vertex can be calculated in $\mathcal{O}(n)$ time, and then the whole next configuration $F(x)$ can be calculated in at most $\mathcal{O}(n^2)$ time. Let $\{x(t) : t \geq 0\}$ a trajectory obtained from a configuration x. We say that the dynamic of x enters into a cycle if $\exists t_1, t_2 \geq 0$, $t_1 < t_2$ such that $x(t_1) = x(t_2)$, and the length of the cycle is $t_2 - t_1$.

To decide **PER**, we should prove that for any configuration the dynamic of x enters into a cycle of polynomial length in at most a polynomial number of steps.

Lemma 2. *For the majority rule,* **PER** *is in* **P**.

Proof. We will use the results obtained in [5, 6]. Let $\eta : \mathbb{R} \to \{0, 1\}$ the threshold function

$$\eta(x) = \begin{cases} 1 & if \quad x \geq 0 \\ 0 & if \quad x < 0 \end{cases}$$

we can characterize the majority rule using this function:

$$x_v(t+1) = \begin{cases} \eta\left(\sum_{u \in N(v)} x_u(t) + x_v(t) - \frac{|N(v)|+1}{2}\right) & if \quad |N(v)| \text{ is even} \\ \eta\left(\sum_{u \in N(v)} x_u(t) - \frac{|N(v)|}{2}\right) & if \quad |N(v)| \text{ is odd} \end{cases}$$

If $\overline{\eta} : \mathbb{R}^{|V|} \to \{0, 1\}^{|V|}$ is the multidimensional threshold function: $\overline{\eta}((x_i)_{i \in V}) = (\eta(x_i))_{i \in V}$, then $x(t+1) = \overline{\eta}(Ax(t) - b)$ where $A = (a_{uv})$ is a square matrix of order $|V|$ and b is a vector of size $|V|$ with

$$a_{uv} = \begin{cases} 1 & if & uv \in E \\ 1 & if & i = j \ \wedge \ |N(v)| \text{ is even} \\ 0 & & \text{otherwise} \end{cases}$$

and

$$b_v = \begin{cases} \frac{|N(v)|+1}{2} & if \quad |N(i)| \text{ is even} \\ \frac{|N(v)|}{2} & if \quad |N(i)| \text{ is odd} \end{cases}$$

Using this characterization, let $\{x(t) : t \geq 0\}$ be a trajectory of the majority rule with initial condition $x(0)$, and define the following functional for $t \geq 1$:

$$\begin{aligned} E(x(t)) &= \sum_{v \in V} (2b_v - \sum_{u \in V} a_{vu})(2x_v(t) + 2x_v(t-1) - 2) \\ &\quad -(2x_v(t) - 1) \sum_{u \in V} a_{vu}(2x_u(t-1) - 1) \end{aligned}$$

We have that this functional satisfies

$$\begin{aligned} \Delta_t E &= E(x(t)) - E(x(t-1)) \\ &= -4 \sum_{v \in V} (x_v(t) - x_v(t-2))(\sum_{u \in V} a_{vu} x_u(t-1) - b_v) \leq 0 \end{aligned}$$

and then is a strictly decreasing Lyapunov function. Hence, if $\{x(t) : t \in [t_1, t_2]\}$ is a cycle, then $E(x(t))$ is necessarily constant for any $t \in [t_1, t_2]$. Actually, since for any $x \in \{0,1\}^{|V|}$ and $v \in V$ we have $\sum_{u \in V} a_{vu} x_u \neq b_v$, then the cycles are only fixed points and/or cycles of length two, i.e. $t_2 = t_1$ or $t_2 = t_1 + 2$.

Then we have that for any initial configuration, the dynamics necessarily enters in a cycle of length at most 2. We must prove now that the entrance occurs in a step that is polynomial on the size of the graph. Let again $\{x(t) : t \geq 0\}$ be a trajectory of some rule with initial condition $x(0)$. We can define its transient length [5, 6] by:

$$\tau(x) = \max\{t \geq 0 : x(t) \text{ enters a cycle for the first time}\}$$

and the transient lenth for the Automata Network is the greatest of these values:

$$\tau(A, b) = \max\{\tau(x(0)) : x(0) \in \{0,1\}^{|V|}\}.$$

Also by a result in [5, 6] we have that $\tau(A, b) \leq |V|^3$ for the majority rule. This tells us that in a number of steps that is polynomial in the size of the input, the trajectory enters into a cycle of length 2. Since any iteration can be simulated in polynomial time, **PER** is in P. □

3. The majority rule is P-Hard

We will give now a proof of **P**-Hardness of the majority rule, reducing it to a restricted case of **CVP**. Clearly proving the **P**-Completeness in the family of planar graphs is stronger than proving it over any graph. Moreover, the **P**-Hardness of **PER** with the majority rule follows from the result of C. Moore in [2] who proved that for the majority rule **PER** is **P**-Complete in the d-dimensional lattice, for d grater than 3. We will provide our own proof of the **P**-Hardness, since it will help us to explain better the planar case.

Lemma 3. *For the majority rule* **PER** *is* **P**-*Hard*

Proof. To prove that for the majority rule **PER** is P-Hard, we will reduce **S2MCVP** to it. Since **S2MCVP** is P-Complete, if the reduction uses only a logarithmic space, then **PER** would be P-Complete.

To reduce **S2MCVP** to **PER**, we will build, given a monotone circuit, a graph that simulates its gates, where active vertices represent truth, and passive vertices represent false. Since the circuit is monotone, we only have to simulate the AND and OR gates.

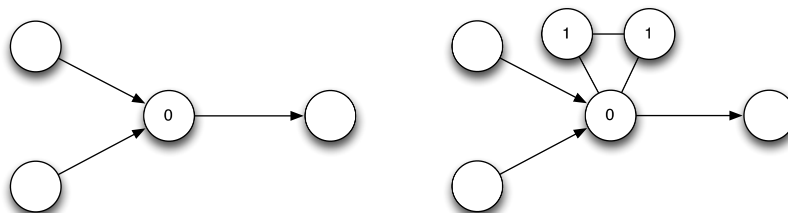

Figure 1: a) The AND gate b) The OR gate

The AND gate (Figure 1 (a)) is simulated by an initially passive middle vertex with degree 3, two of them will be the inputs and the other the output. By the majority rule, this vertex will become active only if two more neighbors become active. In a similar way, the OR gate (Figure 1 (b)) has an initially passive middle vertex, with degree 5 and two neighbors initially active, then, this vertex will become active if any passive neighbor becomes active. Connections between active vertices are used to keep them active.

In order to avoid problems with the flow of information, Figure 2 (a) shows the construction of a 'diode' which only allows the flow of information

in 'one way': If the left vertex is active, then the right vertex will become active. But if the right vertex is active, the left vertex remains in its state. We simplify the drawing with an arrow Figure 2 (b). In both AND and OR gates (Figure 1), the diode is included.

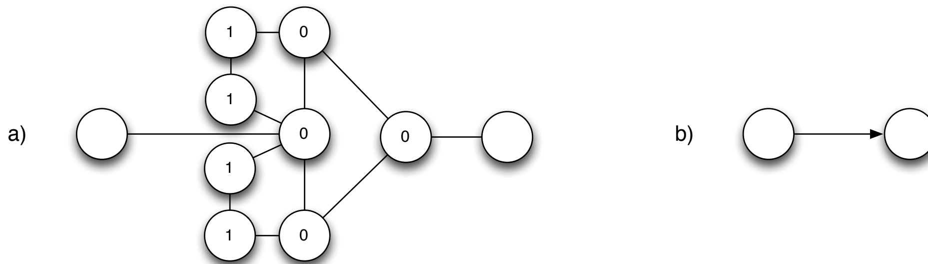

Figure 2: a) The diode b) Simbol of a diode

Remember that in our restricted version of the circuit value problem **S2MCVP**, every gate of the original circuit must have fan-out exactly 2, so we use OR gates 'backwards' in order to multiply the information of the output, as in Figure 3. Finally, we connect the gates with 'wires' (Figure 4)

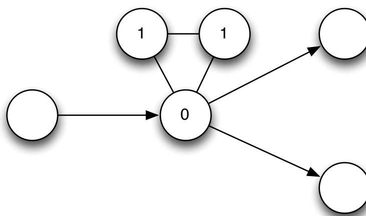

Figure 3: OR gate 'backwards' to multiply the information of an output

that corresponds to initially passive vertices with degree 3 and one neighbor initially active, so they stay passive unless another neighbor becomes active. Then, with these constructions, given ϕ a monotone Boolean circuit and I an input, we can build a graph G that simulates ϕ in I. Letting active the input nodes that are true, passive the rest, and considering v the vertex that simulates the state of the output o of the circuit, (G, x, v) belongs to **PER** if and only if (ϕ, I, o) belongs to **S2MCVP**.

This reduction can be done by a Turing machine in logarithmic space: reading the input (a monotone circuit) of size n, the machine only has to

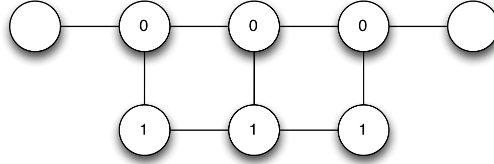

Figure 4: A wire of length 3

determine which construction corresponds to each gate ($\mathcal{O}(1)$ space), and then determine the connectivity of every gate ($\mathcal{O}(\log n)$ space). Then, the whole construction requires $\mathcal{O}(\log n)$ space.

Then for the majority rule **PER** is **P**-Hard. □

4. P-Hardness of the majority rule for planar graphs.

Despite of the constructions of Lemma 3 that are all planar, the input graph may be not be a planar one. Moreover, as we had said in the introduction, the monotone circuit value problem restricted to the family of planar graphs **PMCVP** is not likely to be a P-Complete problem, since there is a NC-Algorithm that solves it [3]. Then we should build a planar graph that simulates a not necessarily planar circuit.

Let (ϕ, x) an instance of our special case of **S2MCVP**, this is, ϕ is a monotone, in degree two, out degree two, layered synchronous circuit, where x is an input of ϕ. We will need that our circuit is lexicographically ordered, which means that the gates of the circuit are numbered from 1 to n, where n is the number of gates, and it's coded in order that if n_1, n_2 are the numberings of the gates in layers l_1 and l_2 respectively, then $l_1 < l_2 \Rightarrow n_1 < n_2$. **CVP** with this restrictions still is **P**-Complete [1].

We have that the gates in the input layer (layer 0) are numbered from 1 to n_0, the gates in the layer 1 are numbered from $n_0 + 1$ to n_1 and so on. We define $N_l = n_l - n_{l-1}$, the number of gates in the layer l.

Lemma 4. *Over the family of planar graphs, the problem* **PER** *associated to the majority rule is* **P**-*Hard.*

Proof. In this case the P-Hardness also is obtained reducing a circuit, but this time we must guarantee that the resulting graph is planar. To ensure this, in our reduction algorithm we also give the positions of every vertex in the XY plane in order to obtain a planar embedding.

We will first assign positions in the XY plane to the vertices that will simulate the gates of the circuit, which we know, by the proof of Lemma 3, that are: a passive vertex joined with two initially active and stable vertices if the gate is of type OR, and just a passive vertex if the gate is of type AND (Figure 1). In both cases the initially active vertex is assigned to coordinates $(2u-1, p_l)$, where u is the numbering of the simulated gate, relatively to its layer l (then $u \in \{1, \ldots, N_l\}$) and p_l is some number bounded $\mathcal{O}(N_l^2)$ which is the same for every gate in layer l (Figure 5). In the case that the gate is of type OR, we give positions to the initially active vertices positions very close to $(2u-1, p_l)$ in the same layout of the Figure 1, for example $(2u-\epsilon, p_l-\epsilon)$, $(2u-\epsilon, p_l+\epsilon)$, for some small ϵ.

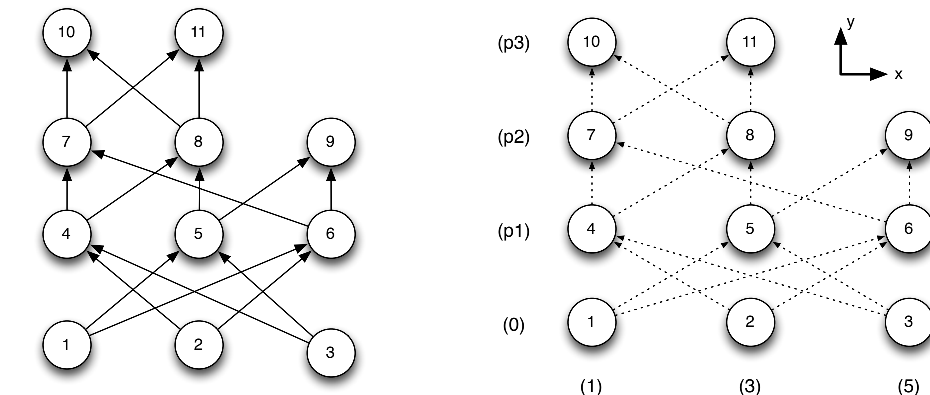

Figure 5: An example of a circuit and the position of the gates in the planar embedding. Notice that the y coordinate is unknown yet for the vertices that do not represent gates in the input layer, but is the same for every vertex which represent gates in the same layer. If we just draw the edges we clearly won't obtain a planar embedding.

This can be done in $\mathcal{O}(\log(n))$ space, since we just need to keep the numbering of the gate that we are simulating ($\mathcal{O}(\log(n))$ space) and its type ($\mathcal{O}(1)$ space). To obtain the values of p_l, we will execute the following induction: We define $p_0 = 0$, and then that we have defined p_l we obtain p_{l+1} by the following steps.

Step 1 First we use the gadget for multiplying the information (Figure 3), in order to be able to "separate" the two outputs of every gate in the layer l. Remember that our case of CVP ensures us that every gate has exactly two inputs and two outputs. For a vertex that simulates a gate of the layer l, say numbered u, we draw the gadget in order that the positions of the vertices

is planar, like in figure 3, and the outputs are in position $(2u-1, p_l+1)$ and $(2u, p_l+1)$. We will call these new vertices u_a and u_b.(Figure 6).

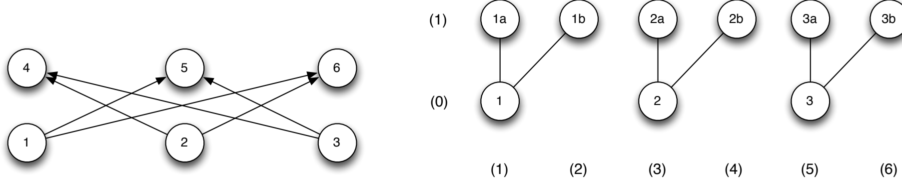

Figure 6: Left: The two first layers of the circuit of the example in Figure 5. Right: Scheme of the positions of the vertices that simulates the input layer and the positions of their two outputs.

Step 2 Let g_1 the first gate in the layer $l+1$, and let i and j, its inputs, with $i < j$. Then, g_1 will have as inputs i_a and j_a in the simulated circuit. In general, if i is an input of a gate g, in our simulated circuit the vertex that simulates g will have as input i_a if g is the first gate in the layer $l+1$ which has i as input, and i_b otherwise.

Too keep planarity, we would like that the inputs of the gate k-th gate of the layer $l+1$ has positions $(2k-1, p_{l+1}-1)$ and $(2k, p_{l+1}-1)$. Since normally this is not the case, we will introduce a new gadget that will help us to shift the positions.

We are going to change the places where there are 'crossing cables', building a gadget that uses 'traffic lights', letting the information pass in one way or another depending in the parity of the step. Figure 7 shows this gadget. Vertices **a** and **b** are the 'inputs' , **d** and **e** are the 'outputs' of the gadget. Before **a** and **b** and after **d** and **e** there are diodes, that are omitted for simplicity.

The key here is that the active vertices that simulate inputs will become passive, since they are only connected only to the first vertex of a diode of the gadget for multiplying info (Figure 3), in the first step all become passive, in the second the initially active become active again, and so on. More over, with exceptions of the initially active nodes that are used to build the gadgets, every vertex that simulates a gate and becomes active at some step, begin to blink between active and passive states.

Suppose then that at least one of the inputs of the crossing cable gadget become active in an even step (and then is passive in odd steps, we are supposing that the inputs of the gadgets are synchronized). If they become

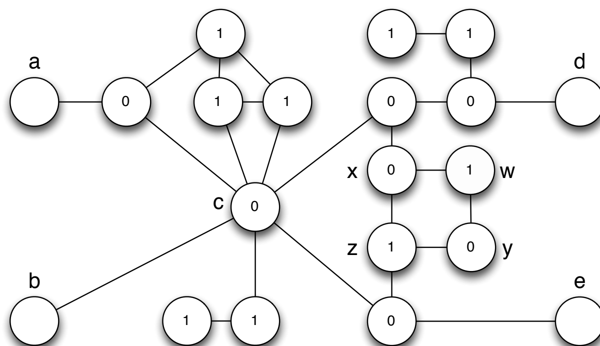

Figure 7: Gadget for 'Crossing cables' ussing 'Traffic Lights'

active in an even step, we change $x_x(0) = x_y(0) = 0$ and $x_z(0) = x_w(0) = 1$ and the rest is analogous.

We have that $x_x(0) = x_y(0) = 1$ and $x_z(0) = x_w(0) = 0$, (Figure 7), then for every $k \in \mathbb{N}$

$$x_x(2k) = x_y(2k) = 1 \text{ , } x_z(2k) = x_w(2k) = 0,$$

$$x_x(2k-1) = x_y(2k-1) = 0 \text{ and } x_z(2k-1) = x_w(2k-1) = 1.$$

Let **i** the vertex connected to **c**, **z** and **e**, in Figure 7. Notice that it will become active only if two of them are active. Since **z** is active only in odd steps, and between **i** and **e** we have a diode, the only way that **i** becomes active is that **z** and **c** are active at the same step, and this can only happen if **a** is active at some even step. Then, we can take the information from **a** to **e** without any "contamination" to **d** (Figure 8). We can cross information from **b** to **d** in a similar fashion.

Back in **step 2**, we are going to use our gadget to change positions of the vertices that carry the information from layer l to layer $l+1$. Starting with the inputs of the first gate of the layer $l+1$, which we called g_1. Let i_a the smallest input of g_1, if the position of i_a isn't $(1, p_l + 1)$, we use our crossing cables gadget (Figure 7) with $\mathbf{a} = (i-1)_b$, $\mathbf{b} = i_a$, $\mathbf{d} = (2(i-1), p_l + 2)$ and $\mathbf{e} = (2i-1, p_l + 2)$ and the rest of the vertices of the gadget very close of $(2i - 3/2, p_l + 3/2)$. We connect with a wire of length 13 (Figure 4) between every vertex with position $(x, p_l + 1)$ to $(x, p_l + 2)$, where $x \in \{1, \ldots, N_l\} \backslash \{i_a, (i-1)_b\}$ (Figure 9).

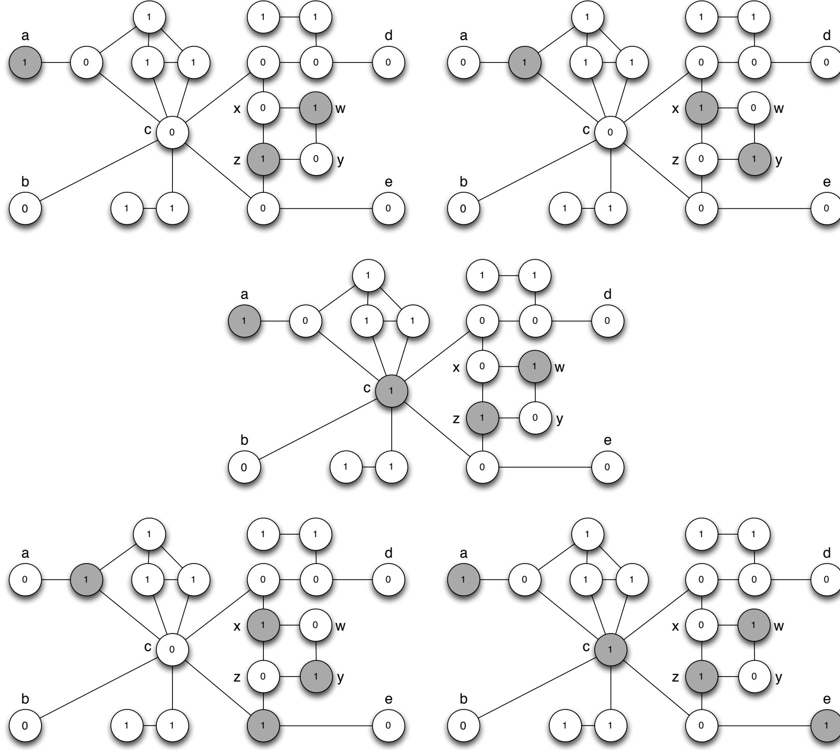

Figure 8: Steps for the crossing information gadget from **a** to **e**. Notice that before **a** and **b** and after **d** and **e** there are diodes, which can be crossed in 4 steps. Then the whole gadget is crossed in 13 steps.

Summing-up, we just add a new row of vertices such that i_a changes x coordinate with $(i-1)_b$ using the crossing cable gadget, and everyone else keeps their position (Figure 9). We repeat this until i_a reaches position 1, and then repeat the same for the other input j until it reaches position 2. Once we have both inputs of the first layer in their correct positions, apply the same to the second vertex in the layer $l+1$, and so on (Figure 9).

This can be done in $\mathcal{O}(log(n))$ space. For a vertex v in the layer $l+1$ we need to:

- Identify its inputs (if they are i_a or i_b), which requires $\mathcal{O}(log(n))$ space, since we only need to save its inputs i, j ($\mathcal{O}(log(n))$) and a flag $\mathcal{O}(1)$ to determine if v is the first vertex in the layer which has this inputs.
- Calculate the initial position of an input i_x of v, which is $I[i_x] = 2i +$

$Q[x] + P[i]$, where $Q[x] = -1$ if $x = a$ and 0 if $x = b$, and $P[i] = N_a[i] + N_b[i]$, with

$$N_a[i] = |\{j \in \text{layer } l \mid j_a \text{ is input of } u < v \text{ , and } j > i\}|$$

$$N_b[i] = |\{j \in \text{layer } l \mid j_b \text{ is input of } u < v \text{ , and } j > i\}|$$

$P[i]$ can be calculated in $\mathcal{O}(log(n))$ since we just need to check from 1 to v if any gate has none, one or both inputs smaller than i, and $Q[x]$ can be calculated with the identification done before.

- Build the gadgets from the initial position into the final one, which is $2v - 1$ or $2v$ depending if we are moving the first or the second input. This requires $\mathcal{O}(log(1))$ since we just need to keep a counter for the number of times that we must build our gadgets.

Then step 2 requires only $\mathcal{O}(log(n))$ space.

Step 3 Finally, we have that for every vertex v that simulates a gate in layer $l + 1$ has its inputs in positions $2v - 1$ and $2v$, we just draw the corresponding diodes (Figure 10). This requires $\mathcal{O}(1)$ space since we just need to take a counter for parity.

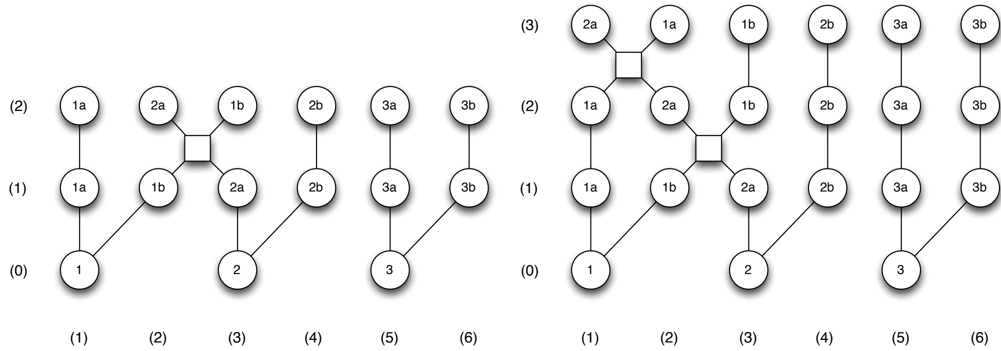

Figure 9: Scheme of Step 2: Squared node represents the crossing cables gadget. Edges between vertices of y-coordinate (1) and (2), and y coordinate (2) and (3) represent wires of length 13.

We repeat this for every layer of the circuit. Then, with these constructions and gadgets, given ϕ a monotone Boolean circuit and I an input, we can build a planar graph G that simulates ϕ in I. Letting active the input

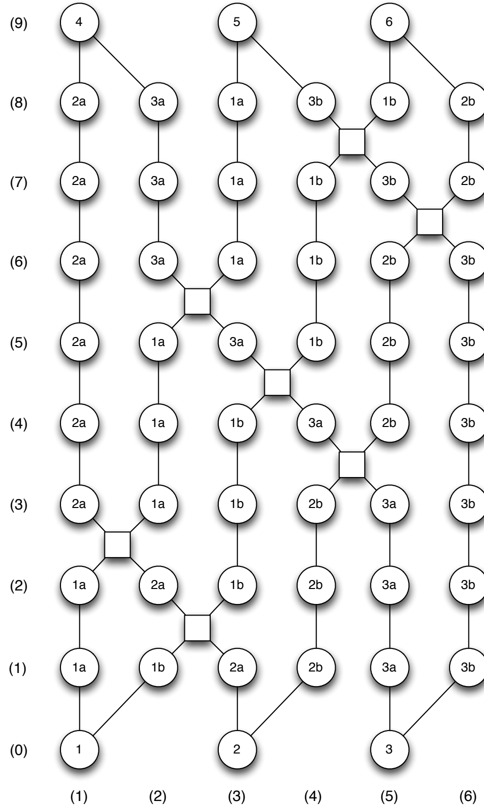

Figure 10: Scheme of the planar embedding that simulates the two first layers of a circuit in Figure 6. Squared node represents the crossing cables gadget. Edges between vertices with different y-coordinate grater that 1 represent wires of length 13.

nodes that are true, passive the rest, and considering v the vertex that simulates the state of the output o of the circuit, (G, x, v) belongs to **PER** if and only if (ϕ, I, o) belongs to **S2MCVP**.

Since this reduction can be done in $\mathcal{O}(\log n)$ space, for the majority rule **PER** is **P**-Hard on the family of planar graphs. □

5. Conclusions

We have discussed about the majority rule in planar graphs, proving that the problem is **P**-Complete. The membership in **P** was proven defining a decreasing energy function, which ensure us that the dynamics are going to converge into cycles of length at most two in a polynomial number of steps.

Then, using some gadgets, where the most important one is the crossing cables gadget with *traffic lights*, we were able to simulate a monotone circuit with the majority rule in a planar graph.

On the other hand, C. Moore studied this rule in the d-dimensional lattice [2]. He proved that for $d \geq 3$ the problem is **P**-Complete, but leaves open the case of $d = 2$, conjecturing membership in **NC**. Clearly, two dimensional lattice is a planar graph, and in this context our result shows that the planarity alone is not enough to classify the problem into **NC**. If the conjecture is true we may consider more specific cases, like restrict the degree of the graph.

In a previous work [4] we have studied the complexity of bootstrap percolation, which is the majority rule but when *active* vertices remains always *active*. In that case we proved that for the family of graphs with degree strictly lower than 5 the rule is in **NC**, so for the two dimensional lattice the particular case majority (Bootstrap Percolation) is in **NC**.

These approaches enclose the conjecture of C. Moore, showing that both the topological properties of the lattice (for example the restrictions on degree and regularity) and the symmetry of the rule for both states (zeros and ones may change) are relevant to study this problem.

More technical details about previous approaches and some generalizations to other local functions can be seen in [7].

References

[1] R. Greenlaw, H. Hoover, W. Ruzzo, Limits to parallel computation: P-completeness theory, Oxford University Press, 1995.

[2] C. Moore, Majority-Vote Cellular Automata, Ising Dynamics, and P-Completeness, Journal of Statistical Physics 88 (1997) 795–805.

[3] A. L. Delcher, S. R. Kosaraju, An nc algorithm for evaluating monotone planar circuits, SIAM J. Comput. 24 (1995) 369–375. doi:10.1137/S0097539792226278.

[4] E. Goles, P. Montealegre-Barba, I. Todinca, The complexity of the bootstraping percolation and other problems, Theoretical Computer Science (0) (2012) –.

[5] E. Goles-Chacc, F. Fogelman-Soulie, D. Pellegrin, Decreasing energy functions as a tool for studying threshold networks, Discrete Applied Mathematics 12 (3) (1985) 261 – 277.

[6] E. Goles-Chacc, Comportement oscillatoire d'une famille d'automates cellulaires non uniformes, Thèse IMAG, Grenoble.

[7] P. Montealegre-Barba, Redes de autómatas y complejidad computacional, Departamento de Ingeniería Matemática, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, 2012.

[8] J. JáJá, An introduction to parallel algorithms, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.