Un problema en el espacio **PSPACE**

Marco Bock
Cruz Enrique Borges Hernández

5 de febrero de 2005

Índice

1. Resumen

El teorema de Savitch es una herramienta muy útil y que ha posibilitado un gran avance en el estudio y desarrollo de algoritmos. Intentaremos comprender que es lo que afirma este teorema y pretendemos mostrar algunas de sus aplicaciones a lo largo de este trabajo. Intentaremos también mostrar un problema que se resuelve mediante un algoritmo perteneciente a la clase **PSPACE**-Completo.

Comenzaremos con un resumen de las distintas clases de complejidad y sus relaciones. Una vez encuadrado el teorema de Savitch entraremos a analizar sus consecuencias: la diferencia entre complejidad en espacio y en tiempo, la creación de algoritmos "eficientes" en espacio, y la desaparición del indeterminismo en espacio. Encuadraremos también la clase **PSPACE** y entraremos, de forma somera, sobre la completitud y sus asombrosas cualidades.

Continuaremos con la teoría de grafos. Daremos una idea intuitiva y procederemos a la definición. Presentaremos un algoritmo básico que resuelve el problema de alcanzabilidad, mostrando las distintas alternativas que posee según se usen estructuras tipo pilas o colas o simplemente la aleatoriedad. Insistiremos en este problema, pues es fundamental debido a que se usa en multitud de ocasiones, mostrando otro algoritmo que sigue otra idea distinta que usaremos posteriormente para demostrar el teorema de Savitch. Ilustraremos los resultados con una traza de cada algoritmo y analizaremos la complejidad de éstos.

El interés de estudiar la teoría de grafos proviene pues las máquinas de Turing indeterministas se representan muy bien mediante un tipo especial de grafos, hablamos de los árboles n-áreos. Estudiaremos los árboles, sus particularidades y luego nos centraremos en los árboles n-áreos. Tras ello pasaremos a enunciar y resolver el teorema de Savitch aplicando los algoritmos que aprendimos en el apartado anterior.

Pasaremos luego a comentar algunos aplicaciones directas del teorema de Savitch como son: algoritmos recursivos bien programados y backtracking. Plantearemos un algoritmo genérico para resolver problemas mediante un backtracking y analizaremos la relación entre la forma que tiene un backtracking y el resultado del teorema de Savitch. Tras lo cual pasaremos a mostrar un problema que se resuelve mediante un backtracking. Hablamos de la búsqueda de un camino Hamiltoniano, un problema clásico en **NP**. Intentaremos también relacionar los backtracking con los algoritmos indeterministas como sistema para emular los "guess".

Tras esto pasaremos a la segunda parte del trabajo Tratamos la clase de complejidad **PSPACE** con más profundidad. Más exactamente veremos un ejemplo importante de esta clase: QBF, las fórmulas booleanas cuantificadores. Este problema es **PSPACE**-completo. Indicaremos una idea de la demostración de que QBF esta en **PSPACE** usando el teorema de Savitch visto con anterioridad.

Después veremos un ejemplo de como las formulas booleanas cuantificadores aparecen en la téoria de bases de datos. Al final este trabajo recorreremos algunos otros ejemplos de problemas en **PSPACE** como, por ejemplo juegos. Analizaremos como podemos usar QBF para diseñar estrategias ganadoras en juegos de mesa,

como en GEOGRAPHY. Para finalizar profundizaremos en este juego viendo que es **PSPACE**-completo.

2. Complejidad algorítmica

2.1. Definiciones

En esta sección haremos una introducción a las principales definiciones en complejidad algorítmica y la motivación de dicha clasificación.

Cuando diseñamos un algoritmo para resolver un problema nos encontramos con el dilema de conocer el alcance de éste algoritmo, en definitiva, de saber si es bueno o malo, o incluso mejor o peor que otro. Pero, ¿qué significa que un algoritmo sea bueno o malo? ¿cómo comparamos un algoritmo con otro? La complejidad algorítmica es la rama de las matemáticas que se encarga de estudiar dicho problema.

Generalmente se estudian cuatro parámetro de un algoritmo:

1. Cantidad de espacio (en el peor caso).
2. Cantidad de tiempo (en el peor caso).
3. Cantidad de espacio (en el caso medio).
4. Cantidad de tiempo (en el caso medio).

Nosotros nos centraremos en los peores casos. Ahora bien, cuando decimos cantidad de espacio o tiempo ¿a qué nos referimos exactamente? Sabemos que a todo algoritmo le podemos asociar una máquina de Turing, y aprovechando esto, podemos definir el tiempo como el número de pasos de cálculo que debe realizar dicha máquina para resolver un problema de tamaño n (donde n generalmente es la talla del input). En cuanto al espacio, lo podemos definir como el máximo espacio (memoria) consumido por todas las cintas de trabajo de la máquina de Turing asociada al algoritmo, al resolver un problema de tamaño n.

Atendiendo a estas definiciones surgen inmediatamente estas otras, que clasifican los algoritmos:

1. $\mathbf{DTIME}(f)$[1] = Conjunto de algoritmos para los cuales una máquina de Turing determinística usa, como mucho, $f(n)$ pasos de cálculo para resolver un problema de tamaño n.
2. $\mathbf{NTIME}(f)$ = Conjunto de algoritmos para los cuales una máquina de Turing (determinística o no) usa, como mucho, $f(n)$ pasos de cálculo para resolver un problema de tamaño n.

   Que dan lugar a clases más específicas como:

   a) $\mathbf{P} = \bigcup_{n\in\mathbb{N}} \mathbf{DTIME}(n^k)$

[1] $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ donde n suele ser el tamaño del input.

b) $\mathbf{NP} = \bigcup_{n\in\mathbb{N}} \mathbf{NTIME}(n^k)$[2]

c) $\mathbf{EXTIME} = \mathbf{E} = \bigcup_{n\in\mathbb{N}} \mathbf{DTIME}(k^n)$

d) $\mathbf{NEXTIME} = \mathbf{NE} = \bigcup_{n\in\mathbb{N}} \mathbf{NTIME}(k^n)$

e) $\mathbf{EXPTIME} = \mathbf{EXP} = \bigcup_{n\in\mathbb{N}} \mathbf{DTIME}(2^{n^k})$

f) $\mathbf{NEXPTIME} = \mathbf{NEXP} = \bigcup_{n\in\mathbb{N}} \mathbf{NTIME}(2^{n^k})$

g) etc.

3. $\mathbf{DSPACE}(f)$ = Conjunto de algoritmos para los cuales una máquina de Turing determinística usa $f(n)$ espacio para resolver un problema de tamaño n.

4. $\mathbf{NSPACE}(f)$ = Conjunto de algoritmos para los cuales una máquina de Turing (determinística o no) usa $f(n)$ espacio para resolver un problema de tamaño n.

   Que dan lugar a clases más específicas como:

   a) $\mathbf{LOG} = \mathbf{L} = \mathbf{DSPACE}(\log_2(n))$

   b) $\mathbf{NLOG} = \mathbf{NL} = \mathbf{NSPACE}(\log_2(n))$

   c) $\mathbf{PLOG} = \bigcup_{n\in\mathbb{N}} \mathbf{DSPACE}(\log_2^k(n))$

   d) $\mathbf{NPLOG} = \bigcup_{n\in\mathbb{N}} \mathbf{NSPACE}(\log_2^k(n))$[3]

   e) $\mathbf{PSPACE} = \bigcup_{n\in\mathbb{N}} \mathbf{DSPACE}(n^k)$

   f) $\mathbf{NPSPACE} = \bigcup_{n\in\mathbb{N}} \mathbf{NSPACE}(n^k)$

   g) etc.

2.2. Relaciones entre clases

A simple vista podemos afirmar que las anteriores definiciones presentan algunas relaciones. Por ejemplo es obvio que[4]:

- $\mathbf{DTIME}(f(n)) \subseteq \mathbf{NTIME}(f(n))$
- $\mathbf{DSPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$

[2]Estas clases, junto con la clase **PSPACE**, forman el actual universo de lo tratable o computable.

[3]La segunda parte del presente trabajo consiste en demostrar el teorema de Savitch el cual nos asegura que $\mathbf{DSPACE}(f) = \mathbf{NSPACE}(f^2)$ con lo cual obtenemos que **PLOG**=**NPLOG** y que **PSPACE**=**NPSPACE**.

[4]Este resultado se debe al hecho de que una máquina de Turing determinística no es más que un caso especial de máquina de Turing indeterminística.

Unos resultados muy importantes, son los llamados *Teoremas de jerarquía*, tanto en espacio como en tiempo.

$\mathbf{DTIME}(f) \neq \mathbf{DTIME}(f \log(f))$

$\mathbf{NTIME}(f) \neq \mathbf{NTIME}(f \log(f))$

Si $f \notin O(g)$ entonces

$\mathbf{DSPACE}(f) \neq \mathbf{DSPACE}(g)$

$\mathbf{NSPACE}(f) \neq \mathbf{NSPACE}(g)$

Una prueba de estos resultados se puede encontrar en [1, pag.143-145]

Gracias a estos teoremas podemos realizar una clasificación de los algoritmos en clases, sabiendo además cuando un algoritmo estará en una clase y no en otro, o mejor dicho, cuando dos clases serán disjuntas. Esto nos permite *encuadrar* un algoritmo en una clase y en caso de ser una de las clases **E** o **EXP** o peores, prácticamente abandonar la idea de resolver dicho problema de forma determinística y comenzar a buscar algoritmos que aproximen la solución.

Otras dos relaciones importantes y que no son triviales como las anteriores son:

- $\mathbf{NTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$
- $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{DTIME}(k^{\log(n)+f(n)})$

Una prueba de estas relaciones se puede encontrar en [1, pag. 147].

Como consecuencia de estos resultados obtenemos la siguiente cadena de complejidades:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$$

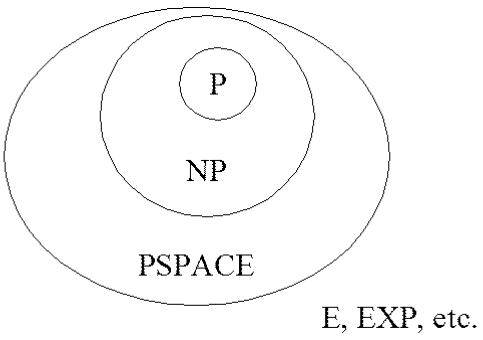

Figura 1: Gráfico con la relación entre las clases centrales de complejidad.

Por los teorema de jerarquía se sabe que alguno de los últimos contenidos es estricto, pero se desconoce cual de ellos es. Es más, cualquier información relevante en la cadena de desigualdades anterior puede llevar consigo resolver la *conjetura de Cook* ($\mathbf{P} \stackrel{?}{=} \mathbf{NP}$), uno de los siete problemas del *Clay Mathematics Institute* que tiene como premio un millón de US$

2.3. Completitud

Es evidente que demostrar que cierto problema[5] pertenece a una determinada clase puede ser una tarea muy dura. Ahora bien, podemos decir que un problema *es tan complejo* como otro problema si podemos transformar uno en el otro en *poco* tiempo[6]. Esto quiere decir que si resolvemos nuestro *problema transformado* habremos resuelto también nuestro problema original.

Se comprueba fácilmente que esta propiedad cumple la propiedad transitiva, con lo cual podemos hacer *cadenas* de problemas. Rápidamente nos viene a la mente la pregunta ¿existirán elementos maximales? La respuesta es afirmativa y a este subconjunto de problemas se les llama *completos*. Son los problemas más *difíciles* de cada clase. Captan la esencia y dificultad intrínseca de cada clase y pueden ser tomados como representantes. Es obvio que su estudio esta fomentado, no sólo por el problema del que proceden, sino por tener la curiosa y sorprendente propiedad de que si se halla un algoritmo *mejor* para resolver dicho problema rápidamente podrá ser usado para mejorar TODOS los algoritmos de dicha clase.

Algunos ejemplos de estas clases son:

- **NP**-completo por estar en ella gran parte de los problemas interesantes, aparte del interés en demostrar la conjetura de Cook.

  Ejemplos: Karp, SAT, juegos como el Tetris o problemas de programación lineal entera.

- **PSPACE**-completo que representa la frontera de lo actualmente computable.

  Ejemplos: El QBF, juegos como el: Go, Socoban, Geography, etc.

[5] Cuando decimos que un problema pertenece a cierta clase de complejidad entendemos que el mejor algoritmo conocido, que resuelve dicho problema, está en esa clase.

[6] Con poco tiempo queremos decir que el algoritmo que realiza dicha transformación pertenezca a lo sumo a la clase **P**.

3. Teorema de Savitch

3.1. Una introducción a la teoría de grafos

Un grafo es una brillante idea matemática que pretende modelizar algunos problemas. Pretende la simplificación y obviar lo superfluo. El ejemplo más simple puede ser el del viajero con un mapa de carreteras que pretende ir de una ciudad a otra y se pregunta cual es la ruta a seguir, si existe. El plano de carreteras es, básicamente, un grafo. Bueno, estrictamente hablando sería más que un grafo, pues para nuestro problema el plano de carretera incluye información que nos es superflua como: el tipo de carretera, la trayectoria que sigue, la distancia, etc. En realidad a nosotros lo único que nos interesa es si dos ciudades del mapa (nodos) tienen una carretera que las una (aristas), y nos importa bien poco el tipo de carretera o la trayectoria que siga. Hemos reducido nuestro problema de encontrar una ruta entre dos ciudades a encontrar un camino a lo largo de un grafo.

Figura 2: Mapa de carreteras y su grafo asociado.

Podemos dar ahora una definición más formal de un grafo:

Definición (Grafo)

Sea V un conjunto finito (de nodos) y $A = \{(x, y)/x, y \in V\}$ conjunto de aristas, donde $(x, y) \in A \Leftrightarrow x$ e y están conectados. Llamaremos grafo al par (V, A)

Existen múltiples formas de dar un grafo. Si pensamos en el conjunto de nodos, en principio pude ser cualquier cosa, pero al ser un conjunto finito podemos hacer un biyección sobre $\mathbb{N}$ y por lo tanto el parámetro fundamental aquí será el cardinal del conjunto, pues mediante esta biyección identificamos cada nodo con un número natural. En cuanto a las aristas, la forma mas común de darlas es mediante una matriz $A \in \mathcal{M}_{n\times n}(\mathbb{Z}_2)$ donde un 1 en la posición (i, j) indica que existe una arista

desde el nodo i al nodo j. Pero ésta no es la forma más eficiente de dar las aristas desde el punto de vista del espacio, pues por lo general el número de 0 en la matrix A (nodos no conectados) es muy superior al número de 1, es lo que matemáticamente se llama una *sparse matrix*. Para dar un método eficiente necesitamos antes dar un par de definiciones.

Definición (Distancia entre nodos)

Sea (V, A) un grafo con $x, y \in V$. Decimos que $dist(x, y) = n$ si para ir de x a y, siguiendo las aristas del grafo, es necesario recorrer al menos n aristas.

Definición (Conjunto de sucesores, predecesores)

Sea (V, A) un grafo. $\forall x \in V$ definimos:

- $(\Gamma_x^+)^k = \{y \in V \ / \ A(x, y) = 1\}$

  Esto es, el conjunto de nodos a los que puedo llegar en, a lo sumo$k$ pasos, desde x.

- $(\Gamma_x^-)^k = \{y \in V \ / \ A(y, x) = 1\}$

  Esto es, el conjunto de nodos desde los que puedo llegar a x en a lo sumo, k pasos.

Con esta forma de dar el conjunto de aristas del grafo minimizamos el espacio en memoria pues sólo almacenamos las relaciones.

Es fácil ver que podemos pasar de A que Γ^+ muy fácilmente. Por ejemplo, dada $A \in \mathcal{M}_{n\times n}(\mathbb{Z}_2)$ y $x \in V$, Γ_x^+ se halla recorriendo la columna o fila de la matriz A correspondiente a x y "guardando" las posiciones en las que encuentre 1. Por el contrario, dado $\Gamma_x^+ \ \forall x \in V$ entonces basta ir recorriendo cada conjunto y realizando la siguiente asignación, $A(x, i) = 1$ si $i \in \Gamma_x^+$.

En los ejemplos siguientes usaremos A, pues simplifica el algoritmo a usar. En otros algoritmos es preferible usar el conjunto de sucesores e incluso en algunos es mejor usar ambos[7].

3.2. El problema de alcanzabilidad

Ahora que tenemos la definición de grafo vamos a profundizar en el problema del viajero. Supongamos que nuestro viajero quiere ir desde el nodo 1 al 6 de la figura[8] 3. ¿Lo podrá hacer pasando por cuatro aristas? ¿Y por sólo 2?

Veamos un algoritmo que nos va a permitir decidir sobre este problema:

[7] De forma teórica, pues en la practica o se tiene uno o el otro, y si es estrictamente necesario se reconstruye el otro.

[8] Debemos usar otro grafo, pues el anteriormente presentado es un *árbol* (Un árbol es un grafo que no tiene circuitos, esto es, sólo va a existir un camino entre dos puntos, y que además conecta a todos los puntos, es decir, es conexo). Este hecho nos deja muy poco juego a la hora de usar el algoritmo.

Algoritmo (Alcanzabilidad)

Sea (Γ_z^+, n) con $z \in \{1, \ldots, n\}$ el conjunto de sucesores y el número de nodos de un grafo (V, A) y sea $(x, y, i) \in V^2 \times \mathbb{N}/x, y, i \leq n$ donde x es el nodo de partida, y el de llegada e i el número de pasos que podemos dar.

Definimos ahora las siguiente variables:

- $P \in \{-1, 0, 1\}^n$ *\ Conjunto de trabajo. *
- $V \in \{0, 1\}^n$ *\ Vector de marcas. *
- $j \in \mathbb{N}$ *\ Distancia. *

Paso. 1 Inicialización:

$j = 0$; $P[x] = V[x] = 1$;

Paso. 2 Mientras $\{\exists z \ / \ P[z] = 1\} \wedge \{j \leq i\} \wedge \{V[y] = 0\}$ haz:

- $P = -P$ *\ Defino el conjunto de trabajo. *
- $\forall \ w \ / \ P[w] = -1$

  2. 1 $P[w] = 0$ *\ Saco un elemento de la cola. *

  2. 2 $\forall z \in \Gamma_w^+$ si $V[z] \neq 1$ entonces $P[w] = V[z] = 1$ *\ Meto el elemento en la cola y lo marco como visitado. *
- $j = j + +$

Paso. 3 Si $V[y] = 1$ entonces y es "alcanzable" desde x.

Este algoritmo trabaja como un "explorador". A cada paso guarda en V los nodos a los que puede llegar en ese momento y coloca en P los nodos de trabajo para, en la siguiente iteración, usarlos como base de sus "exploraciones". El algoritmo concluye cuando encuentra el nodo buscado o se pasa del número de aristas a recorrer.

Veamos una traza de este algoritmo sobre el grafo de la figura 3:

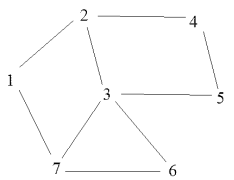

Figura 3: Ejemplo de grafo para el problema de alcanzabilidad.

Traza

Input $= (1, 6, 3)$:

$P = [1, 0, 0, 0, 0, 0, 0]$; $V = [1, 0, 0, 0, 0, 0, 0]$; $j = 0$;

1º Iteración $x = 1$; $0 \leq 3$; $V[6] = 0$;

$P = [-1, 0, 0, 0, 0, 0, 0]$

- $P[1] = 0$
  $\Gamma_1^+ = \{2, 7\}$
  - $V[2] = 0 \Longrightarrow P = [0, 1, 0, 0, 0, 0, 0]$; $V = [1, 1, 0, 0, 0, 0, 0]$
  - $V[7] = 0 \Longrightarrow P = [0, 1, 0, 0, 0, 0, 1]$; $V = [1, 1, 0, 0, 0, 0, 1]$

$j = 1$

2º Iteración $x \in \{2, 7\}$; $1 \leq 3$; $V[6] = 0$;

$P = [0, -1, 0, 0, 0, 0, -1]$

- $P[2] = 0$
  $\Gamma_2^+ = \{3, 4\}$
  - $V[3] = 0 \Longrightarrow P = [0, 0, 1, 0, 0, 0, -1]$; $V = [1, 1, 1, 0, 0, 0, 1]$
  - $V[4] = 0 \Longrightarrow P = [0, 0, 1, 1, 0, 0, -1]$; $V = [1, 1, 1, 1, 0, 0, 1]$
- $P[7] = 0$
  $\Gamma_7^+ = \{3, 6\}$
  - $V[3] = 1$
  - $V[6] = 0 \Longrightarrow P = [0, 0, 1, 1, 0, 1, 0]$; $V = [1, 1, 1, 1, 0, 1, 1]$

$j = 2$

3º Iteración $x \in \{3, 4, 6\}$; $2 \leq 3$; $V[6] = 1$;

Existe un camino entre el nodo 1 y 6 recorriendo menos de 3 aristas.

Observamos en la traza como en cada iteración "crece" el conjunto V, que es terreno visitado por nuestro curioso algoritmo "explorador". Este tipo de recorridos se denomina *Recorridos en amplitud* pues explorar todas las aristas cercanas antes de adentrarse en cada una de ellas. Nótese como el vector P actúa como si fuera una *Cola*[9], motivo por el cual este algoritmo se comporta de esta manera. Más adelante veremos otro ejemplo en el cual en vez de una cola usamos una estructura de *Pila*

Hagamos un estudio de la complejidad algorítmica, tanto en tiempo como en espacio. Comenzaremos con el espacio que presenta más facilidades.

Para la realización del algoritmo observamos que necesitamos definir tres variables auxiliares, dos de ellas pertenecen a $O(n)$ (los dos vectores) y la otra pertenece a $O(\log_2 n)$, donde n es el número de nodos del grafo. También hay que tener en

[9]Una cola es una forma de almacenar datos de forma que se van introduciendo datos en la cola y se sacan de forma que el primero en entrar sea el primero en salir (FIFO según las siglas en inglés). Como ejemplo podemos poner la cola que se forma a la hora de sacar las entradas para un espectáculo.

cuenta que la talla del input pertenece a $O(n^2)$ pues el conjunto de sucesores Γ^+ es de ese orden en el peor caso (Grafo completo[10]). Luego podemos asegurar que es un algoritmo $\in O(n)$ en espacio respecto al número de nodos.

Veamos ahora que sucede con el tiempo. Los parámetros fundamentales aquí serán la longitud del camino buscado, i y el cardinal del número de sucesores, ambas cantidades acotadas por n. El algoritmo realiza un máximo de n iteraciones y en cada una de ellas realiza del orden $O(n)$ operaciones. Luego a primera vista parece que es un algoritmo de $O(n^2)$ en tiempo si estamos considerando un grafo completo. Sin embargo veamos que ambas cosas no se pueden dar simultáneamente.

Supongamos que buscamos el camino entre dos puntos cualesquiera del grafo y nos da igual su longitud, por lo tanto $i = n$. Lo peor que nos podría pasar es que el conjunto de sucesores tuviera por cardinal n, sin embargo esto significaría que existe una arista directa entre ambos nodos y el algoritmo terminaría en un máximo de n pasos.

Supongamos ahora que no existe esa arista. Hemos realizado $n-1$ pasos y ahora se nos pone la cosa cuesta arriba pues hay que analizar los sucesores de $n-2$ nodos, pero si nos damos cuenta, solo queda un nodo por marcar, nuestro objetivo, luego en una iteración más habremos encontrado la solución. Y así sucesivamente.

Con esto demostramos que el algoritmo realiza $O(n)$ iteraciones, por lo que es un algoritmo $O(n)$ en tiempo.

Veamos ahora otro algoritmo que resuelve el problema de la alcanzabilidad que reduce el espacio usado. Más adelante usaremos este algoritmo para demostrar el teorema de Savitch.

Algoritmo (Alcanzabilidad)

Sea (V, A^{11}) un grafo, con $|V| = n$ y $A \in \mathcal{M}_n(\mathbb{Z}_2)$ y sea $(x, y, i) \in V^2 \times \mathbb{N}$

Definimos la función recursiva booleana:

$Path$(x,y,i)

con las siguiente variables:

- $z \in \mathbb{N}$ *\ Auxiliar para recorrer los nodos. *
- $sol \in \mathbb{Z}_2$ *\ Auxiliar para controlar si encontramos un punto medio. *

Paso. 1 Si $i = 1$ entonces $Path = A(x, y)$

en otro caso:

Paso. 2 Inicialización:

$z = 1$; $sol = 0$;

[10]Decimos que un grafo es completo cuando $\forall x, y \in V \ \exists (x, y) \in A$. Estos es, dados dos nodos cualesquiera del grafo, existe una arista que los une.

[11]Para que este algoritmo quede un poco más simplificado asumiremos que la diagonal de la matriz de adyacencia esta formada por 1, aunque en realidad no existan bucles, esto es, caminos de un nodo en si mismo.

Paso. 3 Mientras $\{sol = 0\} \wedge \{z \leq n\}$ haz:

3. 1 Si $z \neq x$ haz

- $sol = Path(x, z, i-1)$
- Si $sol = 1$ entonces $sol = Path(z, y, i-1)$

3. 2 z=z++

Paso. 4 $Path = sol$

Este algoritmo explota la siguiente idea. Para ir de x a y en i pasos probablemente tenga que pasar por otros nodos z antes en, como mucho, $i-1$ pasos. El algoritmo busca estos caminos de forma recursiva. Se pregunta cada vez si existe un camino intermedio de x a z_0 y luego se pregunta lo mismo (llamada recursiva) entre x e z_1. Lo anterior se repite hasta que $i = 1$ momento en el cual podremos averiguar, directamente, si existe o no el camino, simplemente mirando la matriz de adyacencia. En caso afirmativo pasa a explorar la existencia del camino z_0, y de la misma forma. El algoritmo concluye si encuentra la tal z_0 buscada. A este tipo de búsqueda de caminos se les denomina *Recorridos en profundidad* pues el algoritmo intenta alejarse del nodo origen con mucha rapidez.

Veamos ahora una traza de este algoritmo sobre el grafo de la figura 3.

Traza

Input= $(1, 6, 3)$

$i \neq 1$

$z = 1$; $sol = 0$; *\ Inicialización. *

$sol = 0 \wedge z = 1 \leq 7$ *\ Bucle mientras. *

- $z = 1$ $\times$
- $z = 2$ ✓

  Input= $(1, 2, 2)$*\ Llamada recursiva. *

  $i \neq 1$

  $z = 1$; $sol = 0$; *\ Inicialización. *

  $sol = 0 \wedge z = 1 \leq 7$ *\ Bucle mientras. *

  - $z = 1$ $\times$
  - $z = 2$ ✓

    Input= $(1, 2, 1)$*\ Llamada recursiva. *

    $i = 1$

    $sol = 1$ ✓

    Input= $(2, 2, 1)$*\ Llamada recursiva. *

    $i = 1$

    $sol = 1$ ✓

Input= $(2, 6, 2)$*\ Llamada recursiva. *

$i \neq 1$

$z = 1$; $sol = 0$; *\ Inicialización. *

$sol = 0 \wedge z = 1 \leq 7$ *\ Bucle mientras. *

- $z = 1$ ✓
  Input= $(2, 1, 1)$*\ Llamada recursiva. *
  $i = 1$
  $sol = 1$ ✓
  Input= $(1, 6, 1)$*\ Llamada recursiva. *
  $i = 1$
  $sol = 0$ ×
- $z = 2$ ×
- $z = 3$ ✓
  Input= $(2, 3, 1)$*\ Llamada recursiva. *
  $i = 1$
  $sol = 1$ ✓
  Input= $(3, 6, 1)$*\ Llamada recursiva. *
  $i = 1$
  $sol = 0$ ✓

$Path$= 1 y por lo tanto existe un camino entre los nodos 1 y 6.

Hagamos ahora también un análisis de la complejidad de este algoritmo. La complejidad de los algoritmos recursivos se calcula muy fácilmente a partir de ecuaciones en recurrencia. La asociada al tiempo en este problema sería:

$$T(n) = 2T(n-1) = 2^2T(n-2) = \ldots = 2^{n-1}T(1) = 2^n$$

Con lo cual obtenemos que el algoritmo es de orden $O(2^n)$ en tiempo. Llevando un vector de visitados podríamos reducir enormemente la cantidad de pasos, pero intentamos reducir el espacio que utiliza el algoritmo, no el número de operaciones.

Al analizar el espacio observamos que los parámetros importantes son: cuanto espacio requiere cada instancia de la llamada recursiva y cuantas llamadas recursivas se producen simultáneamente. Respecto al primer parámetro observamos que solo usamos dos variables, z que es de tamaño $O(\log_2 n)$ y sol que es de tamaño constante. Para realizar las llamadas recursivas usaremos una $pila$[12]. En la pila introduciremos una copia de las variables z y sol así como el input correspondiente a esa llamada recursiva con lo cual cada elemento de la pila tendrá $O(\log_2 n)$ en espacio pues el

[12]Una pila es una forma de almacenar datos de forma que los introducimos en la pila y los sacamos de forma que el último en entrar sea el primero en salir (LIFO según las siglas en inglés). Como ejemplo podemos poner la pila de platos cuando fregamos la loza.

input es de ese orden. Ahora bien, tras obtener la respuesta de la llamada recursiva, sacamos (borramos) ese elemento de la pila, con lo cual la profundidad (número máximo de elemento que contiene) esta acotada por n. Luego este algoritmo es de orden $O(n \log_2 n)$.

3.3. Árboles n-áreos

Vamos a estudiar ahora un tipo particular de grafos, se trata de los árboles n-áreos. Estudiaremos este tipo de grafos pues son la representación más clara que tenemos para simular como se comporta un algoritmo no determinista.

Definición (Árbol n-áreo)

Sea (V, A) un grafo, decimos que (V, A) es un árbol n-áreo si:

$\forall x \in V, \ \ |\Gamma_x^+| \leq n \ \wedge \ |\Gamma_x^-| \leq 1 \ \wedge \ \exists | x \in V \ / \ |\Gamma_x^-| = 0$

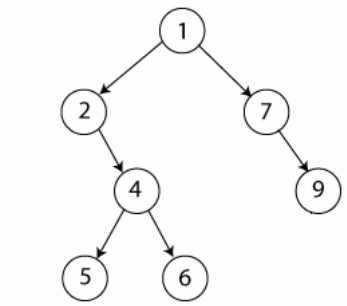

Figura 4: Ejemplo de árbol binario

Vamos a enumerar a continuación unas cuantas propiedades interesantes de este tipo de grafos.

Propiedades

Sea m el número de nodos del árbol, entonces:

1. Al, único, nodo sin predecesores lo llamaremos raíz.
2. Existe hasta un máximo de $n^{\lceil \log_n m \rceil - 1}$ nodos sin sucesores, a los cuales llamaremos hojas.
3. La profundidad máxima del árbol es $\lceil \log_n m \rceil - 1$ donde la profundidad es el número máximo de aristas que hay que recorrer hasta encontrar una hoja.
4. Dado un nodo del árbol, existe un único camino desde la raíz hasta él.

A partir de estas propiedades se puede deducir que los árboles n-áreos son en realidad un subconjunto de los llamados árboles, cuya definición es la siguiente:

Definición (Árbol)

Sea (V, A) un grafo. (V, A) un árbol si:

- Es acíclico, esto es, no existe ningún camino que empiece y termine en el mismo nodo. (El grafo no posea aristas "de más").
- Es conexo, esto es todos los nodos están conectados entre si.

Vamos a revisar ahora la complejidad de nuestro algoritmo de alcanzabilidad cuando se lo aplicamos a un árbol n-áreo. Por un argumento anterior sabíamos que la cota anterior era $O(m \log_2 m)$ donde m era el número de nodos del grafo. Recordamos que el término lineal provenía de la profundidad de la pila, que en un grafo sólo podemos acotarla por m, sin embargo ahora, como el grafo es un árbol n-áreo, sabemos que dicha profundidad no será mayor que $\lceil \log_n m \rceil$ por lo que nuestro algoritmo es del orden $O(\log_2^2 m)$.

3.4. Teorema de Savitch

Veamos ahora como podemos usar este resultado para demostrar el teorema de Savitch.

Si un algoritmo es $\mathbf{NSPACE}(f(n))$ significa que, en algún momento, la función de transición es una correspondencia. Luego, o no tiene imagen, o bien tiene más de una. Representamos cada paso de cálculo como un nodo de un grafo y dos nodos del grafo tienen una arista si puedo ir de uno a otro mediante la aplicación de la función de transición. Supongamos que la función de transición no crea más de c alternativas, tenemos entonces que, a partir de la configuración inicial (que será nuestra raíz), se genera un árbol de $c^{f(n)}$ nodos que tendrá de profundidad $f(n)$ nodos.

Corolario (Teorema de Savitch)

$\mathbf{NSPACE}f(n) \subseteq \mathbf{DSPACE}(f^2(n))\ \forall f(n) \geq \log n$

Demostración

La demostración es muy simple. Buscamos un algoritmo determinista que resuelva nuestro problema en espacio $O(f^2(n))$.

Aplicamos el algoritmo de alcanzabilidad anteriormente descrito al grafo de las configuraciones. Con lo cual tenemos que el orden en espacio es de $O(\log_c^2 c^{f(n)}) = O(f^2(n))$ y conseguimos recorrer todos los nodos del grafo de configuraciones (y obtener la eventual solución) en un espacio no mayor del pedido.

3.5. Una aplicación en la programación

Es común utilizar algoritmos recursivos cuando programamos... y también es muy común leer u oír que se deben evitar dichos algoritmos. Lo que no es tan común es que se explique porqué. La respuesta es bien simple cuando uno conoce el teorema de Savitch. La traza de un algoritmo recursivo es un grafo, donde cada nodo representa el estado de las variables antes de realizar una llamada recursiva, y las aristas representan las relaciones de "padre e hijo" entre las llamadas recursivas. Este grafo es en realidad un árbol, además, con mucha frecuencia, binario.

Viéndolo de esta forma se ve claro que un algoritmo recursivo mal programado puede provocar un desperdicio de espacio considerable. Aquí entra en juego el teorema de Savitch, el cual nos asegura que sea cual sea el algoritmo recursivo, siempre existirá otro de forma que no se desperdicie el espacio y realice la misma tarea. Pero lo mejor de todo es que no solo nos asegura que existe, sino que en la demostración nos dice cómo debemos modificar el algoritmo. Por este motivo muchos compiladores modernos ya realizan la modificación sin que el programador se lo ordene, ni lo note.

Veamos primero que pasaba antiguamente con los algoritmos recursivos. Cuando se producía una llamada recursiva, el procedimiento se paraba y se abría otra instancia de él mismo, que ocupaba otro bloque de memoria y el programa continuaba. Así hasta que se acabara la memoria o el algoritmo comenzara a cerrar instancias de si mismo. Una mala programación provocaría que el número de instancias simultaneas fuera excesivamente grande, aparte del desperdicio de espacio que es abrir una nueva instancia.

La modificación consiste, simplemente, en utilizar una pila externa en la que se guardan las variables de cada llamada y transformar las llamadas recursivas en una iteración. Con lo cual se evita la creación de nuevas instancias, y al guardarse los datos en una pila, nos aseguramos que no se abren instancias innecesaria. Recordemos que para demostrar el teorema de Savitch lo que realizamos fue un recorrido en profundidad de un grafo mediante una pila, algo completamente análogo a lo que estamos realizando ahora. Esta modificación se podría realizar sobre el propio código del programa, pero podría convertir el código en algo confuso.

Veamos, como ejemplo, como quedaría nuestro algoritmo de alcanzabilidad usado para demostrar el teorema de Savitch transformado adecuadamente:

Algoritmo (Alcanzabilidad)

Sea (V, A^{13}) un grafo, con $|V| = n$ y $A \in \mathcal{M}_n(\mathbb{Z}_2)$ y sea $(x, y, i) \in V^2 \times \mathbb{N}$

Definimos la estructura de datos:

$Pila$(u,v,j,w,sol)

Con las siguiente variables:

[13]Para que este algoritmo quede un poco más simplificado asumiremos que la diagonal de la matriz de adyacencia esta formada por 1, aunque en realidad no existan bucles, esto es, caminos de un nodo en si mismo.

- $u, v \in \mathbb{N}$ *\ u, v nodos origen y destino. *
- $j \in \mathbb{N}$ *\ Contador de distancia entre nodos. *
- $w \in \mathbb{N}$ *\ Auxiliar que recorre los nodos. *
- $sol \in \mathbb{Z}_2$ *\ Variable booleana que indica si se ha encontrado solución. *

Como variables del programa tenemos a:

- $x, y, z \in \mathbb{N}$ *\ x, y nodos origen y destino. z auxiliar que recorre los nodos. *
- $i \in \mathbb{N}$ *\ Contador de distancia entre nodos. *
- $sol \in \mathbb{Z}_2$ *\ Variable booleana que indica si se ha encontrado solución. *
- $no\_z \in \mathbb{Z}_2$ *\ Variable booleana que indica si se ha de cambiar de nodo de búsqueda. *
- $P \in Pila$ *\ Variable de tipo pila que almacena las variables locales en cada llamada. *

Como *Pila* es una pila tiene asociada las siguientes funciones:

- $Push(P, x, y, i, z, sol)$ *\ Función que introduce un elemento en la pila. *
- $Pop(P)$ *\ Función que elimina un elemento de la pila. *
- $Leer(P, x, y, i, z, sol)$ *\ Función que copia las variables de la pila a las del programa. *
- $Pila\_vacia(P)$ *\ Función booleana que indica si una pila contiene o no algún elemento. *
- $Mod\_sol(sol)$ *\ Función que modifica la variable *sol* de la pila. *
- $Mod\_z(z)$ *\ Función que modifica la variable z de la pila. *

Paso. 1 *Push(P,x,y,i,1,0)* *\ Inicialización *

Paso. 2 Mientras $\neg$ $Pila\_vacia(P)$ haz:

2. 1 $Leer(P, x, y, i, z, sol)$
2. 2 $No\_z = 0$
2. 3 En caso de que:
   (a) $i = 0$ entonces:
   Si *sol* entonces:
   - $Pop(P)$

- $sol = No\_z = A(x, y)$
- $Mod\_sol(sol)$:
- Mientras sol
  - $Leer(P, x, y, i, z, sol)$
  - Si sol entonces $Pop(P)$

en otro caso:

- $Pop(P)$
- $sol = No\_z = A(x, y)$
- $Mod\_sol(sol)$

(b) $z \leq n$ entonces:
Si $\neg$ sol y $z \neq n$ entonces:

- $Push(P, x, z, i - 1, 1, sol)$
- $No\_z = 1$

en otro caso:

- $Push(P, z, y, i - 1, 1, sol)$
- $No\_z = 1$

(c) $z > n$ entonces $Pop(P)$

2. 4 Si $\neg$ No_z entonces $Mod\_z(z + 1)$

Paso. 3 Si sol entonces "Existe camino"

en otro caso entonces "No existe camino"

Observamos que claramente el algoritmo es mucho más complicado e ininteligible, aunque es más eficiente. Es más, en realidad la verdadera prueba del teorema de Savitch se realiza con este algoritmo, pues se observa que la variable P (la pila) es la variable que marcará el orden en espacio del algoritmo, y ésta siempre se mantiene del orden de $O(\log^2 n)$ pues cada elemento de la pila tiene orden $O(\log n)$ y habrá un máximo de $\log n$ elementos en la pila, con lo cual obtenemos el resultado.

3.6. Backtracking

Veamos ahora un tipo muy especial y útil de algoritmo recursivo, se trata del backtracking. El backtracking es un *algoritmo enumerativo*, se usa cuando no queda más remedio que enumerar todas los casos posibles a la hora de resolver un problema. Es una herramienta muy útil para intentar resolver los problemas indeterministas pues mediante un backtracking simulamos el grafo de configuraciones asociado y lo recorremos (en profundidad) íntegramente de forma ordenada. Digamos que es una manera de "simular" el *guess* típico en los algoritmos indeterminísticos.

Para poder aplicar un backtracking es necesario que nuestro problema cumpla una serie de condiciones, como:

- La solución debe ser un vector $V[k]$, de tamaño finito.
- Dicho vector se puede construir de forma "inductiva", es decir dados los k primeros elementos del vector debemos poder construir el elemento $k+1$.
- Los valores que puede tomar $V[k]$ son conocidos y los denotaremos $P(V,k)$

El algoritmo genérico de un backtracking es el siguiente:

Algoritmo (Backtracking)

Definimos la siguiente función recursiva:

Backtracking(V,k)

Con las siguientes variables:

- $V[\,]$ *\ Vector de tamaño conocido que albergará la solución. *
- $k \in \mathbb{N}$ *\ Variable con la componente del vector de solución *
- $p = P(V,k)$ *\ Variable de conjunto que almacena las posibilidades que puede tomar el vector solución para un determinado valor de k. *
- $v \in P(V,k)$ *\ Variable para almacenar elementos de p. *

Y la siguiente función interna:

- *Es_solucion*(V,k) *\ Función que calcula si un determinado vector cumple el enunciado del problema o no *

Mientras $p \neq \emptyset$ hacer:

Paso. 1 Sea $v \in p$

Paso. 2 $p = p - \{v\}$

Paso. 3 $V[k] = v$

Paso. 4 Si *Es_solución*$(V,k) \ \wedge \ \{k = n\}$ entonces "La solución es $V[\,]$"
en otro caso *Backtracking*$(V, k+1)$

Observamos que al realizar llamadas recursivas usamos una pila como en el resolución del teorema de Savitch. Existen otras formas de recorrer un grafo de configuraciones, por ejemplo los recorridos en amplitud, que surgen de usar una *cola* a la hora de manejar las llamadas recursivas. La única diferencia entre este algoritmo y el anteriormente descrito estriba en que las funciones *Leer*, *Pop*, *Mod_sol* y *Mod_z*, al estar ahora relacionadas con una cola, afectan al primer elemento de la cola, y no al último con antes. Este pequeño cambio provoca que el número de instancias

abiertas recursivamente simultáneamente pueda ser tan grande como el número de nodos, esto es, exponencial con respecto al tamaño de la entrada. Con lo cual este tipo de algoritmos cae con mucha facilidad en la clase **NE** o **NEXP**.

Este tipo de recorridos a través de un grafo no son recomendables para realizar un backtracking, sin embargo para otro tipo de algoritmos pueden ser útiles (basta ver que en el primer algoritmo dado para analizar la *alcanzabilidad* se puede usar una cola para recorrer el grafo y no afecta a la complejidad del algoritmo).

Actualmente hay infinidad de problemas que sólo sabemos resolverlos con un backtracking. El problema de este algoritmo es que generalmente es de orden exponencial en tiempo, no así en espacio, pues la profundidad del árbol suele estar acotada polinomialmente y por el teorema de Savitch, conocemos una forma de recorrer el árbol (alcanzabilidad) en espacio polinomial. Sin embargo, el orden en tiempo depende del problema y viene determinada por el orden al que pertenezca *Es_solucion*. En muchos casos será polinomial y entonces el problema estará en **NP** pero en general será exponencial o expo-potencial (o peor aun incluso) y la clase al la que pertenecerá el backtracking será **PSPACE**.

Se han realizados muchos estudios para mejorar la eficiencia de este algoritmo con ideas como la ramificación y corte, (dejar de visitar casos pues no conducen a buenas soluciones) o algoritmos heurísticos (que nos dicen que valores debemos explorar para alcanzar una buena solución, aunque no tengan unos fundamentos muy sólidos), pero aún así se sigue trabajando para resolver la *conjetura de Cook* o por lo menos conseguir reducir la cantidad de cálculos asociados a este tipo de algoritmos.

3.7. Camino hamiltoniano, el problema del comerciante viajero.

Vamos ahora a enunciar un problema que podemos resolver mediante un backtracking y a realizarle una traza.

Imaginemos que somos un agente comercial de una importante compañía. Se nos ha encargado promocionar unos artículos a lo largo de un país visitando diversas ciudades. Nos interesaría recorrer todas las ciudades pasando sólo una vez por cada una de ellas, para no desperdiciar el tiempo, pero sin dejar de pasar por ninguna. Además, nos interesa que volvamos al punto de partida, pues de todas formas debemos hacerlo para regresar a casa.

Como ya hemos visto con anterioridad el mapa de carreteras entre ciudades es un grafo, y realizar un ruta a lo largo del grafo que cumpla lo anterior se denomina realizar un *circuito hamiltoniano*.

Una manera de resolver este problema es mediante un backtracking usando la idea del algoritmo de alcanzabilidad mostrado al principio del trabajo.

El algoritmo quedaría tal que así:

Algoritmo (Circuito Hamiltoniano)

Sea (V, A) un grafo, con $n = |V|$

Definimos la siguiente función recursiva:

Backtracking(V, k)

Con las siguientes variables:

- $V \in \mathbb{N}^n$ *\ Vector que guarda la ruta. *
- $k \in \mathbb{N}$ *\ Variable con la componente del vector de solución. *
- $P = \Gamma^+_{V[k]}$ *\ Conjunto de sucesores. *
- $v \in \Gamma^+_{V[k]}$ *\ Un elemento del conjunto de sucesores. *

Y la siguiente función interna:

- *Es_solucion*(V, k)
  Si $V[k+1] \neq V[i]\ \forall i \leq k$ entonces *Es_solucion*
  en otro caso $\neg$ *Es_solucion*

Mientras $\{P \neq \emptyset\}\ \wedge\ \{k \leq n\}$ hacer:

Paso. 1 Sea $v \in p$

Paso. 2 $P = P - \{v\}$

Paso. 3 $V[k] = v$

Paso. 4 Si *Es_solución*$(V, k)\ \wedge\ \{k = n\}\ \wedge\ \{1 \in \Gamma^+_{V[n]}\}$ entonces "La solución es V".
en otro caso *Backtracking*$(V, k+1)$

El algoritmo se basa en un principio muy simple, partiendo de la idea del backtracking, va "creando" el camino eligiendo un elemento del conjunto de sucesores en cada paso tal que cumpla todas las propiedades. Es decir, que no se halla visitado con anterioridad. El algoritmo sabe que ha finalizado si consigue completar un vector de n componentes (pues en tal caso habrá visitado todos los nodos del grafo) y desde el último nodo visitado se pueda llegar al primero de un paso.

Existe una variante de este problema consiste en no imponer que se "cierre" el camino. Es decir, permitimos que desde el último nodo no se pueda ir al primero en un paso. Una solución de este problema se realiza con el mismo algoritmo simplemente eliminando la condición $\{1 \in \Gamma^+_{V[n]}\}$ en el paso 4. Sin embargo este problema tiene nombre propio, es la creación de un *árbol generador* y posee algoritmos propios mucho más eficaces que éste.

Vamos a realizar una traza a este algoritmo sobre el grafo de la figura 5.

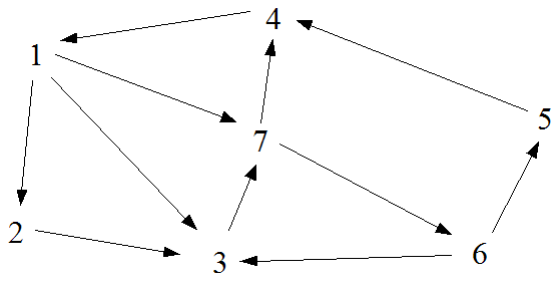

Figura 5: Ejemplo para el problema del comerciante.

Vamos a representar la traza como si fuera un grafo, donde los caminos que podemos formar a lo largo del grafo son los vectores posibles. Observamos que un camino es solución, si y sólo si tiene $n + 1$ nodos (sale del nodo 1 y llega al nodo 1 habiendo pasado por todos los anteriores) Mientras que si llegamos al nodo 1 sin haber pasado antes por todos los demás, o bien repetimos algún nodo (por ejemplo el 3) el camino se cierra.

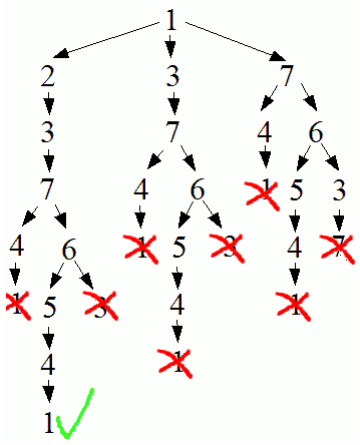

Figura 6: Traza del backtracking sobre el grafo de la figura 5.

4. Problemas en PSPACE-completo

4.1. QBF es un problema PSPACE-completo.

Ahora tratemos un problema en **PSPACE**, concretamente QBF[14]. Veamos que este problema esta **PSPACE**-completo. Pero comencemos con unas definiciones.

Definición (QBF)

Una Fórmula Booleana Cuantificada en forma prenexa es una expresión de la siguiente forma:

$$Q_1 x_1 Q_2 x_2 Q_3 x_3 \dots Q_n x_n \, \Phi$$

donde: $\Phi(x_1, \dots, x_n)$ es una fórmula booleana, $x_i \in \mathbb{Z}_2$ y $Q_i \in \{\forall, \exists\}$ son los cuantificadores.

En la literatura hay diferentes definiciones para QBF[15]. Para nosotros, QBF será el conjunto de las Fórmulas Booleanas Cuantificadores en forma prenexa en donde la fórmula booleana Φ es de cualquier forma.

Nuestro propósito es demostrar que QBF es **PSPACE**-completo. Es decir que tenemos que demostrar dos cosas:

- QBF $\in$ **PSPACE**
- Todo problema en **PSPACE** es reducible (en tiempo polinomial o en espacio logarítmico) a QBF.

Ahora empecemos a mirar las cosas más detenidamente. En primer lugar veamos que QBF es un problema en **PSPACE**.

Sea E una fórmula booleana cuantificada, n el número de variables booleanas y $l = |E|$ la talla de E. Definimos $E_a(x_2, \dots, x_n) = Q_2 x_2 \dots Q_n x_n \ \Phi(a, x_2, \dots, x_n)$ para $a \in \{0, 1\}$. Ahora miramos el siguiente algorítmo:

```
01  BOOLEAN evaluate (E)
02        if ( (cantidad de cuantificadores de E) >= 1 ) then
04          v0 = evaluate(E_0);
06          v1 = evaluate(E_1);
07          if Q_1 = ∀ then v = (v0 and v1);
08          if Q_1 = ∃ then v = (v0 or v1);
09        else v = E
10        return v;
```

[14] Quantified Boolean Formulas

[15] En algunos libros se define QBF como Fórmulas Booleanas Cuantificadas cualquieras, en otros, como las anteriormente definidas, fórmulas en forma prenexa donde Φ es una fórmula booleana cualquiera e incluso algunas veces se le llama QSAT (Quantified SATisfiability) a QBF. Pero siempre es necesario que la fórmula sea satisfactoria.

```
11      end;
```

Este algorítmo funciona recursivamente. Se puede imaginar como un árbol binario completo donde cada variable booleana x_i es un nivel del árbol. La conección de un nodo al próximo significa: si se halla en la rama de la izquierda, la variable toma el valor 1, en cambio, si se halla en la rama de la derecha, toma el valor 0. Entonces, un camino de la raíz del árbol hasta una hoja es una configuración de las variables x_1 hasta x_n. Por eso la profundidad del árbol y del algorítmo es n.

En cada hoja tenemos (fila 9 en el algoritmo) una evaluación de una fórmula booleana con sólo constantes. Esta evaluación necesita espacio polinomial en la talla de la fórmula. Y como la profundidad del árbol es n, este algorítmo necesita espacio polinomial. Por lo tanto QBF $\in$ **PSPACE**.

Observamos que este algoritmo se asemeja bastante a un backtracking, por lo menos en su parte enumerativa, pues presenta algunas diferencias. Éstas proceden de necesitar *TODOS* las posibilidades para dar una respuesta. Decimos que se parece a un backtracking pues la forma de recorrer los nodos del árbol de configuraciones se asemeja a como lo haría un backtracking.

Ahora continuamos con el segundo punto. Sea L un lenguaje cualquiera en **PSPACE** y M una máquina de Turing que decide si un input a con $|a| = n$ es aceptado por L o no. Tenemos que buscar una reduccioón en tiempo polinomial[16] o en espacio logarítmico de este problema sobre QBF. Buscaremos una en tiempo polinomial porque es más fácil[17].

Construimos un grafo que consiste en las configuraciones de M con input a. Como $L \in$ **PSPACE** $\subset$ **EXP** hay a lo sumo 2^{n^k} configuraciones con input a (y también nodos del grafo). Por eso se puede codificar una configuración con un vector en $\{0,1\}^{n^k}$.

Vamos a construir fórmulas booleanas cuantificadas ψ_i sobre:

$$(A, B) = (a_1, \ldots, a_{n^k}, b_1, \ldots, b_{n^k})$$

que serán verdad si y sólo si $A = (a_1, \ldots, a_{n^k})$ y $B = (b_1, \ldots, b_{n^k})$ codifican dos configuraciones donde hay un camino désde A hasta B de longitud a lo sumo 2^i, en el grafo de configuraciones. $\psi_{n^k}(A, B)$ es la fórmula que buscamos donde A es la configuración inicial y B la configuración final aceptadora.

Ahora buscamos resolver ψ_{n^k} en tiempo polinomial. Empezamos con $i = 0$. En este caso $\psi_0(A, B)$ significa que o bien $a_j = b_j$ para $j \in \{1, ..., n^k\}$ o la configuración B continua a A en un paso. Como en la demostración del teorema de Cook (SAT es **NP**-completo[18]) se puede construir ψ_0 como disjunción de $O(n^k)$ claúsuras, cada una con $O(n^k)$ variables.

[16] Karp-reducible

[17] Una reducción en espacio logarítmico es más fuerte pero también más difícil de construir.

[18] Por ejemplo en [3]

Supongamos que tenemos $\psi_i(A, B)$. Continuamos inductivamente con $\psi_{i+1}(A, B)$. Podemos definir:

$$\psi_{i+1}(A, B) := \exists Z[\psi_i(A, Z) \wedge \psi_i(Z, B)], \qquad Z \in \{0, 1\}^{n^k}.$$

Donde Z es una configuración intermedia del camino entre A y B. Pero el número de operaciones de esta expresión crece exponencialmente, como ya hemos visto en 3.2, y por tanto no es la reducción que buscamos (buscamos una reducción en tiempo polinomial).

Definimos ahora:[19]

$$\psi_{i+1}(A, B) := \exists Z \forall X \forall Y[((X = A \wedge Y = Z) \vee (X = Z \wedge Y = B)) \Rightarrow \psi_i(X, Y)]$$

donde X, Y y Z son variables en $\{0, 1\}^{n^k}$. Esta fórmula hace lo que queremos, pero no está en forma prenexa. Sin embargo, es fácil transformarla a forma prenexa. Podemos, con la ayuda de algunas reglas de lógica, poner los cuantificadores al comienzo. Este método emula el que usamos para la demostración del teorema de Savitch. Usamos las variables X, Y como variables auxiliares a modo de "pila" con el fin de reducir el espacio usado.

Ahora miramos el coste de esta construcción. ψ_0 se puede construir en tiempo $O(n^{2k})$. Después añadimos en cada paso una parte de talla $O(n^k)$ y como tenemos n^k pasos el tiempo total es $O(n^{2k})$.

Con lo cual hemos obtenido que cualquier problema en **PSAPACE** puede reducirse a realizar una búsqueda en un grafo de configuraciones mediante QBF. Por lo que podemos asegurar que QBF es **PSCAPE**-Completo.

Con esto completamos la demostración, pero seguiremos ocupándonos todavía un poco más de QBF.

4.2. Preguntas a una base de datos

La pregunta es si este resultado tiene aplicaciones en la práctica. La repuesta es que sí. QBF se usa en bases de datos para realizar preguntas, es decir, realizar una consulta a una base de datos se puede describir como una fórmula booleana cuantificada.

Es más fácil entender este concepto con un ejemplo (figura 7). Una base de datos es un conjunto de tablas (=relaciones) que se relacionan entre ellas. Cada tabla tiene un nombre y una fila con títulos para cada columna.

Una pregunta a esta base de datos podría ser: "Quiero saber los nombres de los alumnos qué tienen un profesor que vive en Burgos". En cálculo proposicional la pregunta tiene la forma siguiente:

$$\{\, x \mid \exists y\, (\, (\, \exists a(\, \text{profesores}(a, \text{“Burgos”}) \,\wedge\, \text{matemáticas}(a, y)\,)\,) \wedge \text{alumnos}(x, b, y)\,)\,\}$$

[19] Recordamos que: $(x \Rightarrow y) \Leftrightarrow (\bar{x} \vee y)$

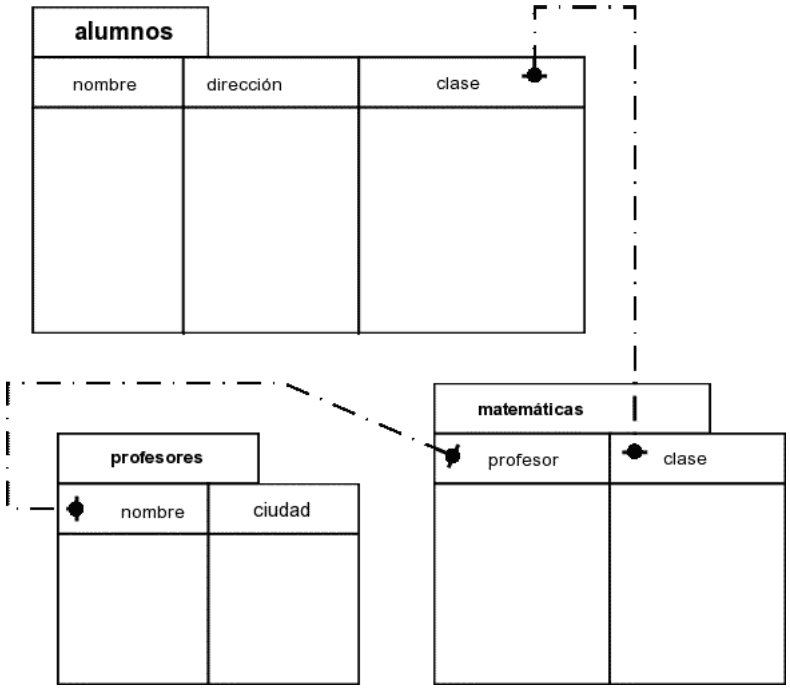

Figura 7: Representación de la base de datos del ejemplo.

Aquí "profesores(a,"Burgos")" es una función con imágen $\{0, 1\}$ que es verdad si lo dicho entre los paréntesises, en una fila de la tabla en la relación profesores.

La repuesta a esta pregunta también es una tabla. Se ve que hallar la respuesta radica en resolver una fórmula booleana cuantificada.

Sabemos entonces, por el apartado anterior, que preguntar a una base de datos, es un problema **PSPACE**. Podemos suponer, y de hecho es así en la mayoría de los casos, que la talla de una base de datos es enorme, en particular mucho más grande que la memoria "rápida" de un ordenador. Pero con este resultado podemos decir que calcular las preguntas a una base de datos pueden tener lugar en la memoria "rápida", pues no usamos más que espacio polinomial en la talla de la entrada, esto es, en la talla de la fórmula dada. Se puede comparar con la diferencia entre el disco duro y la memoria RAM de un ordenador.

4.3. Otros problemas PSPACE-completos

Al final tratamos algunos otros problemas en **PSPACE**, algunos de ellos, también, **PSPACE**-completos. Empezamos con juegos para dos jugadores.

4.3.1. Generalidades

Imagina que estás en jugando con alguien a algún juego. Sería interesante conocer, si existe, una sucesión de "jugadas" tal que uno de los jugadores ganara indepen-

dientemente de las "jugadas" del otro jugador. Dicha posibilidad se podría analizar mediante una fórmula booleana cuantificada.

Usamos los cuantificadores alternados para delimitar las jugadas de las dos personas, así el jugador 1 utilizaría los $\exists$ y el jugador 2 usaría los $\forall$. El jugador 1 ($\exists$) empieza y hace una jugada (elige un valor para x_1). Después el segundo jugador ($\forall$) realiza su jugada (elige un valor para x_2). Así los jugadores continúan realizando sus jugadas (que corresponde con las variables x_{2i+1} para el jugador 1 y x_{2i} para el jugador 2). Suponemos que después de n turnos uno de los jugadores gana. Ilustramos esto de la siguiente forma: el jugador 1 ($\exists$) gana si la fórmula booleana ψ es verdad y el jugador 2 ($\forall$) gana en caso contrario.

Lo anterior es válido para juegos de dos jugadores que alternen sus turnos, conociendo a priori el número máximo de jugadas. Por ejemplo en un juego de tablero (como el ajedrez, GO, tic-tac-toe) en los que las posibilidades son acotadas debido al tablero. El desarrollo de un juego se puede ilustrar como un árbol, pero para ganar se necesita una estrategia y no solamente un camino en el árbol. Esta estrategia es una repuesta en cada turno a la jugada del otro jugador y la talla del algoritmo que genera dicha estrategia es exponencial, generalmente.

Algunos juegos como GO o GEOGRAPHY son **PSPACE**-completos pero otros están en una clase de complejidad más fuerte y es posible resolverlos más rápido y con menos espacio.

4.3.2. GEOGRAPHY

Nos centramos ahora en el juego GEOGRAPHY. Vamos a ver que este juego para dos personas es **PSPACE**-completo. GEOGRAPHY consiste en: Se escoge un conjunto de ciudades (por ejemplo, las ciudades en España). El Jugador I empieza y elige una ciudad de este conjunto, después el otro jugador II tiene que hallar una ciudad cuya primera letra sea la misma que la última letra de la ciudad anterior. Así los jugadores continúan alternándose. Teniendo en cuenta que no está permitido elegir la misma ciudad dos veces, después de un cierto número de turnos algún jugador no puede elegir una ciudad que valga y entonces pierde.

A continuación vamos a pasar GEOGRAPHY al lenguaje de grafos. Observamos que podemos considera las ciudades como los nodos V de un grafo y decimos que existe una arista $a \in E$ de la ciudad A a ciudad B si B empieza con la última letra de A. Luego, obtenemos un grafo dirigido $G = (V, E)$. Este problema se puede generalizar a algunos grafos finitos.

El problema a resolver con GEOGRAPHY es el siguiente: Dado un grafo G con nodo inicial 1, ¿puede ganar el jugador que empieza (jugador I)?

Ahora veamos porque GEOGRAPHY es **PSPACE**-completo. La primera parte funciona como en la demostración de QBF. Se construye el árbol del juego. Esto es un algorítmo recursivo indeterminístico con profundidad a lo más $|V|$ y por el teorema de Savitch necesita espacio polinomial. Cuando un nodo tiene más que dos sucesores so tiene que construir un "árbol binario parcial" y usar esto.

Más interesante es la segunda parte. Buscamos una reducción en tiempo polinomial de un problema cualquiera en **PSPACE** a GEOGRAPHY. Como sabemos que ya existe dicha reducción a QBF solamente necesitamos mostrar que hay una reducción de QBF a GEOGRAPHY en tiempo polinomial. Elegimos una fórmula booleana cuantificador $a \in$ QBF donde $|a| = l$ y n es la cantidad par de variables. Además a empieza con $\exists$[20] y está en forma normal conjuntiva [21]. Necesitamos construir el grafo en tiempo polinomial. La mejor forma de explicar el método es con un ejemplo. Miramos la fórmula siguiente:

$$\exists x \forall y \exists z \forall w \, [\underbrace{(\neg x \vee \neg y)}_{c1} \wedge \underbrace{(y \vee z)}_{c2} \wedge \underbrace{(y \vee \neg z)}_{c3}]$$

El algorítmo construye un grafo (Figura 8): Para cada variable genera un rombo y los conecta uno tras otro. Después del último rombo hay una arista a un nodo nuevo para cada clausura. A partir de estos nodos tenemos una arista al lado convenientes del rombo de cada variable en la clausura.

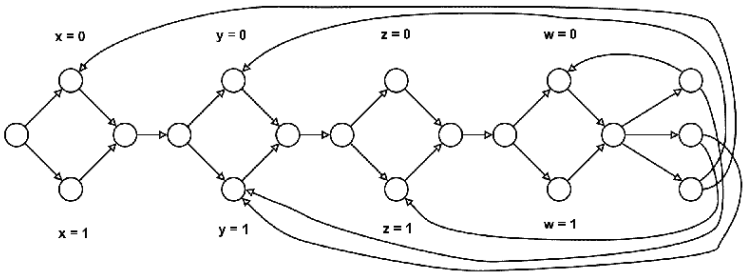

Figura 8: El grafo para la fórmula del ejemplo

Por esto se juego como el siguiente. Tomamos una configuración para las variables de a. En el juego esto significa que usamos un camino hasta el último rombo A en donde marcamos siempre el nodo de enfrente (En el ejemplo: Si $x = 1$ marcamos el nodo $\neg x$).

Después estamos en el nodo A y es el turno de jugador II. Para ganar tiene que hallar un nodo (en el ejemplo: c1, c2 o c3) que solamente tiene aristas a nodos marcados. Pero este es el caso si, con nuestra configuración para las variables, una claúsura de a tiene valor 0. Si este nodo no existe el jugador I va a ganar. Con esta explicación se ve que en general este grafo tiene la propiedad de que a es satisfactorio si y sólo si hay una estrategia para jugador I para ganar.

También es fácil ver que esta construcción del grafo se puede hacer en tiempo polinomial. Hemos probado entonces que GEOGRAPHY es un problema **PSPACE**-completo.

[20] Si la fórmula no tiene esta forma se puede añadir cuantificadores con variables mudas.

[21] Esto no afecta el problema pues hay un algorítmo en tiempo polinomial que transforme una fórmula booleana cualquiera su forma normal conjuntiva.

Referencias

[1] Christos H. Papadimitriou. *"Computacional Complexity"* . Addison-Wesley (1994).

[2] Alonso, Sergio. Apuntes de la asignatura "Modelos matemáticos combinatorio en investigación operativa"perteneciente a la licenciatura de Ciencias y Técnicas Estadísticas de la universidad de La Laguna (2003-2004).

[3] Cook, S.A., "The Complexity of Theorem Proving Procedures", *Proc. 3rd ACM Symp. on Theory of Computing, p. 151-158, 1971*

[4] `http://www.wikipedia.org`

[5] `http://c2.com/cgi/wiki?ExternalizeTheStack`