Guia 5 Complejidad

Probar que los siguientes lenguajes estan en PSPACE

Clique

Se puede resolver usando backtracking generando cada subconjunto de vertices posible y viendo si forman un subconjunto completo de k nodos.

La altura del arbol de backtracking es k <= n. (Porque a lo sumo forma conjuntos con k nodos).

En cada nodo del arbol de backtracking esta el grafo, el subconjunto de nodos y ponele que el contador que dice cuantos nodos tiene.

Al subconjunto lo puedo acotar por el tamaño del grafo. Y al contador por un numero k menor o igual a n. Entonces cada nodo tiene tamaño O(2.|G| + n).

Cuando hacemos backtracking hay en memoria una rama de ejecucion nada mas asi que como la altura del arbol es k, hay a lo sumo k nodos (k < n):

Complejidad total: O(n (2|G| + n)) = O(nˆ3) que es PSPACE

Formula mas chica

Puedo armar con backtracking todas las posibles formulas de tamaño 0 a ϕ Y para cada una verificar si es o no equivalente a ϕ Para hacer eso tengo que probar todas las asignaciones posibles, pero eso es PSPACE.

Cada nodos del arbol de backtracking tendria una formula que esta acotada por el tamaño de ϕ. La altura del arbol es a lo sumo k entonces:

Complejidad espacial: $O(|\phi|^2)$

TATETI

Se puede hacer backtracking explorando todas las jugadas. Para que haya estrategia ganadora el algoritmo chequea que existe jugada para el jugador uno (ganadora) tal que para toda jugada del jugador 2 exista otra jugada para el jugador 1 ganadora, y asi hasta que el jugador 2 pierda.

Cada nodo del arbol de backtracking tiene la matriz. Tiene a lo sumo altura kˆ2, se pueden jugar todas las celdar.

Complejidad espacial: $O(k^4)$

2 Probar que PSPACE está cerrado por complemento.

Tomemos un L cualquiera que esta en PSPACE, es decir una maquina M lo reconoce con espacio polinomial

puedo definir a la maquina que reconoce L^c como:

```
M_c(x):
    return not M(x)
```

Lo unico que hace es negar el bit de salida, asi que no usa mas espacio que M.
$\Rightarrow L^c$ es PSPACE

3 Probar que $NP \subseteq PSPACE$

Si un lenguaje esta en NP hay una maquina N no deterministica que lo decide.

N tiene un arbol de computo que represeenta todos sus posibles computos. Con este arbol podria simular deterministicamente a la maquina no deterministica N recorriendo cada computo.

Como el tamaño de cada computo tiene altura polinomial puedo recorrer todo el arbol con espacio polinomial (al recorrer un arbol solo se guarda de a una rama)

8 Probar que la relacion $\leq_L$ es transitiva

Quiero ver:

$$A \leq_L B \wedge B \leq_L C \Rightarrow A \leq_L C$$

Es decir:

$$x \in A \iff f(x) \in B \wedge x \in B \iff g(x) \in C \Rightarrow x \in A \iff h(x) \in C$$

Desarrollo la hipotesis:

$$(x \in A \iff f(x) \in B \wedge x \in B \iff g(x) \in C) \Rightarrow (x \in A \iff f(x) \in B) \Rightarrow (x \in A \iff g(f(x)) \in C)$$

Entonces existe la funcion h(x) = g(f(x)). Esta funcion es computable implicitamente en L porque f y g lo son y la composicion de dos funciones computables implicitamente en L es computableme implicitamente en L.

9 Sea $\mathcal{L}$ el lenguaje de todas las expresiones con parentesis bien formadas. Probar que $\mathcal{L} \in L$

Para esto basta con dar un pseudocodigo que use espacio log.

```
def balanced(x):
    abiertos = 0
    for c in x:
        if c == "(":
            abiertos += 1
        elif c == ")":
            if abiertos == 0:
                return False
            abiertos -= 1

    if abiertos == 0:
        return True
    else:
        return False
```

Como las operaciones aritmeticas usan espacio logaritmico por lo unico que hay que preocuparse es el contador. Como va a ser a lo sumo |x| (por ejemplo una cadena de solo parentesis que abren) el contado esta acotado por:

$O(log(|x|))$ pues los numeros se representan en base 2.

10 Probar que 2-COLOREO esta en NL

CONSULTAR NI IDEA COMO HACER MAQUINAS NO DETERMINISTICAS

```
nodo = -1
c = 0
mientras c < |V|:
    nodo = inventar un nodo no deterministicamente
    nodo.color = inventarle un color no deterministicamente (color es 0 o 1)
    para cada v en vecindad(nodo):
        v.color = adivinar el color no deterministicamente
        si v.color == nodo.color:
            devolver q_no
devolver q_si
```

11 Probar que son LN-COMPLETOS

a)

$$SCC = \{G : \text{G es fuertemente conexo}\}$$

certificado: Los caminos de cada nodo a todos los nodos en el grafo. Son n caminos de longitud a lo sumo n para cada nodo $\rightarrow O(n^3)$

verificador: Por cada nodo chequea si puede llegar a todos, recorriendo los caminos especificados en el certificado (podriamos imaginar un certificado de pinta {1:[[(1,2)],[(1,2),(2,5),(5,3)],...],...} el 1 tiene camino directo a 2, el uno para llegar a 3 hace 1->2->5->3).

Para hacer esto tiene que guardarse dos nodos, un "actual" que es el que usa para ir avanzando y un destino que es el que usa para chequear que efectivamente llego al que estaba buscando, asi para cada nodo chequear que puede llegar a cualquier otro. Esto usa espacio O(log n) pues solo guarda dos nodos que se codifican como numeros. La cinta del certificado ademas se lee una unica vez.

NL-Completo

Voy a reducrir PATH a SCC:

$$\langle g, s, t\rangle \in PATH \iff f(\langle g, s, t\rangle) \in SCC$$

f tiene que ser computable implicitamente en L.

f va a tomar una instancia de $PATH$ y la va a devolver un G' de la siguiente manera:

- G' va a ser igual a G pero con dos nodos nuevos t_{out} y s_{in}
- Ademas va a tener las aristas $t \rightarrow t_{out}$ y $s_{in} \rightarrow s$
- Para todo nodo v del grafo se van a agregar las aristas $t_{out} \rightarrow v$ y $v \rightarrow s_{in}$

Con esta reduccion cualquier nodo se va a poder "meter" al camino por s_in y a la salida del camino puede ir a cualquier otro por t_{out} (solo si hay camino de s a t, sino llega a t cuando entra a s no puede ir a cualquier otro)

Construir este grafo es computable implicitamente en L, es copiar el grafo y agregar cosas no usamos nada en la cinta de trabajo. A lo sumo iterar por los nodos o algo por el estilo pero se hace con espacio logaritmico.

Veamos que vale:

$\Rightarrow$ Dada una instancia de PATH:

Como ya explique arriba, dada una instancia de PATH si la transformo con f , G' va a ser fuertemente conexo. Si hay un camino de s a t cualquier nodo puede entrar a s_{in} como hay camino puede llegar a t_{out} y de ahi llegar a cualquier otro nodo. (si no hubiera camino no habria forma de llegar de s a t asi que no seria fuertemente conexo) $\Rightarrow$ G' es fuertemente conexo.

$\Leftarrow$ Dada una instancia G' de SCC

Si la instancia G' esta en SCC es fuertemente conexo, como ya explique G' es fuertemente conexo si solo si G tiene camino de S a T porque sino no se podria llegar a cualquier nodo.

b)

NFA-NO-VACIO = $\{\langle A \rangle$: A es un automata no deterministico que reconoce un lenguaje no vacio$\}$

Pertenece a NL?

Certificado: El camino que hace el automate de q_0 a algun q_f

Verfificador: Ir "caminando" el automata (lo pienso como un grafo, la unica diferencia es que hay un inicial, estado finales y transiciones. Recorrerlo es igual). Para caminarlo leo una unica vez el certificado avanzandoi un nodo "actual" al siguiente que indique el certificado. Verificando que existan las aristas que los conectan, que el nodo inicial sea q_0 y que el ultimo sea algun estado final. Esto ocupa espacio log, solo guardo el nodo que pivotea.

hardness:

$$\langle g, s, t \rangle \in PATH \iff f(\langle g, s, t \rangle) \in \text{NFA-NO-VACIO}$$

funcion de reduccion: Copia el grafo como automata, el estado inicial es el nodo s q_s, el conjunto de estados finales solo tiene a q_t y en cada transicion se consume un 1 (podria ser cualquier cosa).

Veamos que vale:

$\Rightarrow$

$\langle g, s, t \rangle \in PATH \Rightarrow$ En A se puede llegar desde q_s (estado inicial) hasta q_t (estado final) $\Rightarrow A \in$ NFA-NO-VA

$\Leftarrow$

$A \in$ NFA-NO-VACIO $\Rightarrow$ Se llega desde el inicial (q_s) hasta el unico estado final (q_t) $\Rightarrow$ Analogamente se llega

12

$$2 - SAT \in NL$$

Como NL = coNL, si 2-SAT esta en coNL esta en NL.

2-SAT $\in$ coNL $\iff$ $\neg$ 2-SAT $\in$ NL

Es decir si dada ϕ, $\neg\phi$ es satisfacible

Notar que si *phi* es 2-SAT tiene pinta (algo $\vee$ algo) $\wedge$ (algo $\vee$ algo).

Si negamos algo de esa pinta usando de morgan llegamos a: ($\neg$ algo $\wedge$ $\neg$ algo) $\vee$ ($\neg$ algo $\wedge$ $\neg$ algo)

Con esto en mente bastaria con que una de esas conjunciones sea verdadera para que toda la formula sea verdadera. (porque es una cadena de $\vee$).

Certificado: La asignacion de las dos variables x_1 y x_2 que hagan que alguna conjuncion sea verdadera. (esto implica que toda la formula se vuelve verdadera)

Verificador: Nos copiamos del certificado a la cinta de trabajo las asignaciones ya que solo se puede leer una vez el certificado. Esto va a ocupar espacio logaritmico porque solo es guardar una cantidad acotada de numeros, es decir O(log n). Recorre la formula y en cada conjuncion chequea si asignando x_1 y x_2 se vuelve verdadera. Solo es neceario traerse de a una conjuncion y como cada una tiene a lo sumo 2 literales sigue siendo O(log n).

13 Probar que $NL \subseteq P$

Por definicion $NL = NSPACE(logn)$

Recordar que:

$$NSPACE(S(n)) \subseteq DTIME(2^{O(S(n))})$$

Entonces:

$$NL = NSPACE(logn) \subseteq DTIME(2^{logn}) \subseteq P$$