Diagonalização e Complexidade Espacial

João Arthur Thiago Emmanuel

Coordenação de pós-graduação em Informática - COPIN

05/08/2007

Roteiro

Teoria da Complexidade

- Separar classes de complexidade
  - Exibir uma máquina em uma classe que gera resposta diferente de **todas** as máquinas de outra classe
- Mostrar a presença de funções não-computáveis
- Provar os teoremas de hierarquia entre classes

Diagonalização é uma técnica apropriada para tais atividades!

Teoria da Complexidade

- Separar classes de complexidade
  - Exibir uma máquina em uma classe que gera resposta diferente de **todas** as máquinas de outra classe
- Mostrar a presença de funções não-computáveis
- Provar os teoremas de hierarquia entre classes

Diagonalização é uma técnica apropriada para tais atividades!

Histórico

- Georg Cantor, 1873
- Argumento da diagonalização
- Provar que o conjunto dos números reais é infinito e não é enumerável

Argumento da diagonalização - Parte 1

- Assuma, por contradição, que o intervalo $[0, 1]$ é infinito enumerável
- Um conjunto é enumerável se ele é equinúmero com o conjunto dos naturais
- Podemos enumerar todos os números deste intervalo através de uma sequência (r1, r2, r3, . . .)
- Cada número pode ser representado como uma expansão decimal

Argumento da diagonalização - Parte 2

- Construir um número dentro do intervalo $[0, 1]$ considerando a diagonal

$r_1 = 0,\ \underline{\mathbf{5}}\ 1\ 0\ 5\ 1\ 1\ 0\ \ldots$
$r_2 = 0,\ 4\ \underline{\mathbf{1}}\ 3\ 2\ 0\ 4\ 3\ \ldots$
$r_3 = 0,\ 8\ 2\ \underline{\mathbf{4}}\ 5\ 0\ 2\ 6\ \ldots$
$r_4 = 0,\ 2\ 3\ 3\ \underline{\mathbf{0}}\ 1\ 2\ 6\ \ldots$
$r_5 = 0,\ 4\ 1\ 0\ 7\ \underline{\mathbf{2}}\ 4\ 6\ \ldots$
$r_6 = 0,\ 9\ 9\ 3\ 7\ 8\ \underline{\mathbf{3}}\ 8\ \ldots$
$r_7 = 0,\ 0\ 1\ 0\ 5\ 1\ 3\ \underline{\mathbf{5}}\ \ldots$
$\ldots$

  - Se o k-ésimo dígito de *rk* é 5, então o k-ésimo dígito de *x* é 4
  - c. c., o k-ésimo dígito de *x* é 5

$$x = 0,4555554\ldots$$

Argumento da diagonalização - Parte 3

- x é um número real dentro do intervalo $[0, 1]$ (expansões decimais representam números reais)
- Deve existir uma sequência $rn = x$
- No entanto, por causa do critério de escolha, x difere na n-ésima posição de rn, então x não está na sequência (r1, r2, r3 . . .)
- Esta sequência, portanto, não é uma enumeração do conjunto de todos os reais no intervalo $[0, 1]$

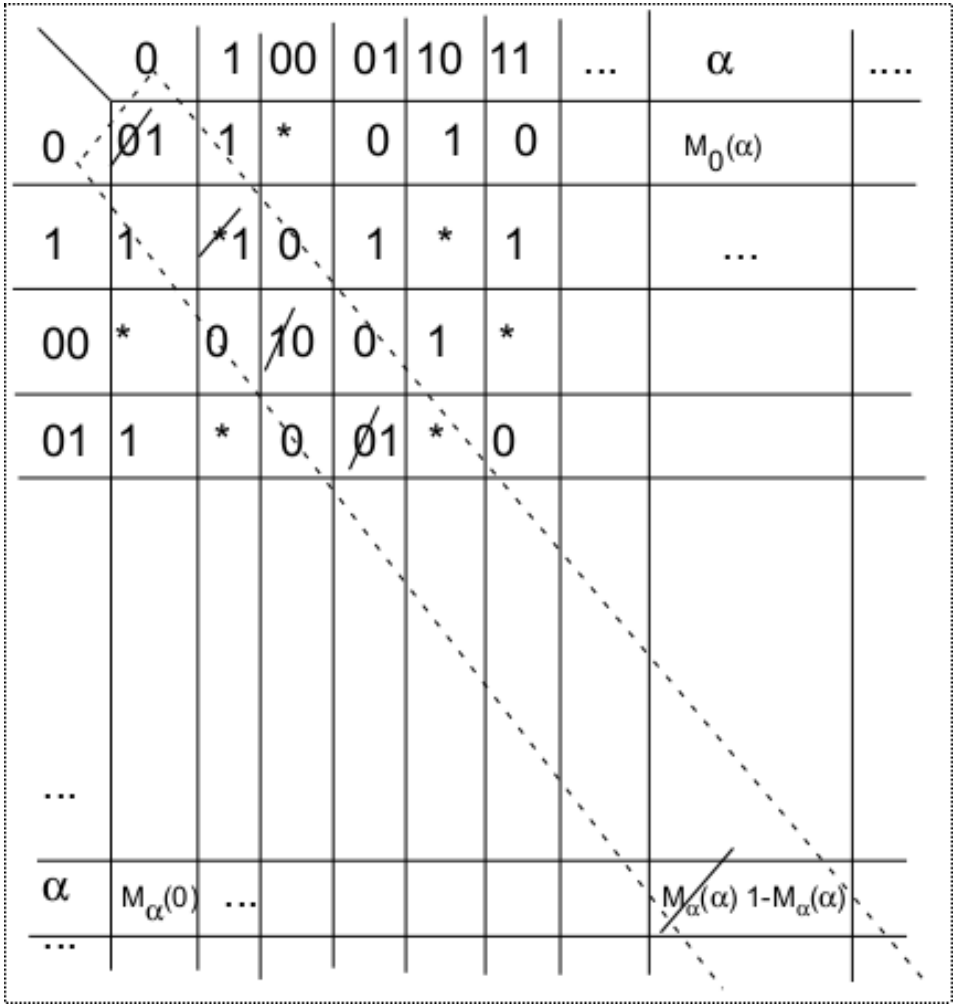
Existem funções não computáveis

- Teorema: Existe uma função $UC : \{0,1\}^* \rightarrow \{0,1\}$ que não é computável por uma MT
- Definição:
  - UC: para todo $\alpha \in \{0,1\}^*$, seja $M(\alpha)$ uma MT representada por α. Se em uma entrada α, M pára com um número finito de passos e computa 1, então $UC(\alpha) = 0$, c.c., $UC(\alpha) = 1$

  Por contradição...

  - Existe uma MT M tal que $M(\alpha) = UC(\alpha)$ para todo $\alpha \in \{0,1\}^*$
  - Em um caso particular $M(M) = UC(M)$, mas isso é impossível pela definição de UC, uma vez que:
    - Se $UC(M) = 1$ então $M(M)$ não pode ser igual a 1 e vice-versa.

Existem funções não-computáveis



Existem funções não-computáveis

Problema da Parada

Time Hierarchy Theorem - Background

- **Aumentar o tempo de computação equivale a aumentar o número de linguagens reconhecidas?**
- *Time-constructible functions*: É uma função f tal que dada uma entrada 1^n, existe uma MT que gera a representação binária de $f(n)$ em tempo $O(f(n))$.
  - $f(n) = nlogn$
  - $f(n) = n^2$

Time Hierarchy Theorem

- Teorema: Sejam f e g funções *time-constructible* tal que $f(n)logf(n) = O(g(n))$, então:

$$DTIME(f(n)) \subsetneq DTIME(g(n))$$

- Vamos provar que $DTIME(n) \subsetneq DTIME(n^{1.5})$

Time Hierarchy Theorem - Prova

- Considere a seguinte MT D: Para uma entrada x, execute por $|x^{1.4}|$ passos a MTU para simular Mx em x. Se Mx gera saída dentro deste intervalo de tempo então a saída será negada (i.e., saída $1 - Mx(x)$). Caso contrário, a saída será 0.
- Por definição, D pára em $|x^{1.4}|$ passos e, por isso, a linguagem L decidida por D está em $DTIME(n^{1.5})$
- Queremos mostrar agora que $L \notin DTIME(n)$

Time Hierarchy Theorem - Prova

- Por contradição, assumimos que existe uma MT M que decide L mas executa em tempo cn para entradas de tamanho n. Então para todo $x \in \{0,1\}^*$, $M(x) = D(x)$.
  - O tempo para simular uma MT na MTU é de $c'c|x|log|x|$
  - Existe um número n0 tal que para todo $n > n0$, $n^{1.4} > c'c|x|log|x|$
  - A máquina terá tempo de terminar a simulação em $|x^{1.4}|$, mas por definição temos:

$$D(x) = 1 - M(x) = M(x)$$

Oráculos

- Diagonalização baseia-se em duas propriedades das MT:
  - Efetiva representação de MT em strings
  - Capacidade de uma MT simular a outra sem grande *overhead*
- Oráculos são MT que possuem acesso a um "oráculo" que resolve "magicamente" um problema de decisão em uma linguagem $O \subseteq \{0,1\}^*$

Oráculos

- Possuem uma fita especial na qual podem escrever uma string $q \in \{0,1\}^*$ e em um momento futuro perguntar se q está presente em O.
- Se O não pode ser decidida em tempo polinomial, então este oráculo adicionou poder à MT.
  - A computação da *query* é feita em 1 passo

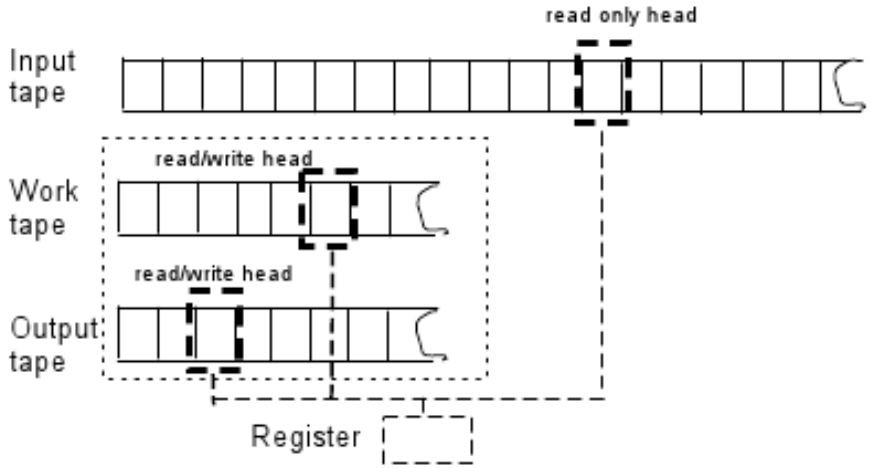
Oráculos - Importância

- Resulta em MT em que $P = NP$ e MT tais que $P \neq NP$
- Podemos concluir que para responder a pergunta $P =? NP$ não basta basear-se somente nas duas propriedades
  - Resultado de relativização *(relativizing results)*
- Se $P = NP$ ou $P =? NP$ for verdadeiro, não é um resultado de relativização

Considerações

- Prova que se $P \neq NP$, então existe uma linguagem $L \in (NP - P)$ que não é NP-Completa. (Teorema de Ladner)
- Usa a representação de máquinas em *strings*
- Diagonalização em si não é capaz de resolver $P =? NP$ sozinha
- Ainda é usada como ferramenta para provas de teoremas

Space-bounded computation



- $SPACE(S(n)) = \{L(M) | M$ is a deterministic multitape TM running in space $S(n)\}$
- $NSPACE(S(n)) = \{L(M) | M$ is a nondeterministic multitape TM running in space $S(n)\}$
  - Número de células ocupadas
    - $c \bullet S(n)$

Relação Espaço x Tempo

Existe algume relação entre a complexidade temporal e espacial de um algoritmo ?

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

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

Relação Espaço x Tempo

Existe algume relação entre a complexidade temporal e espacial de um algoritmo ?

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

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

Relação Espaço x Tempo

Existe algume relação entre a complexidade temporal e espacial de um algoritmo ?

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

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

$DTIME(S(n)) \subseteq SPACE(S(n))$

$DTIME(S(n)) \subseteq SPACE(S(n))$

- Espaço Máximo ocupado por $DTIME(S(n))$
  - Uma célula ocupada em cada passo
  - Sem reuso das células

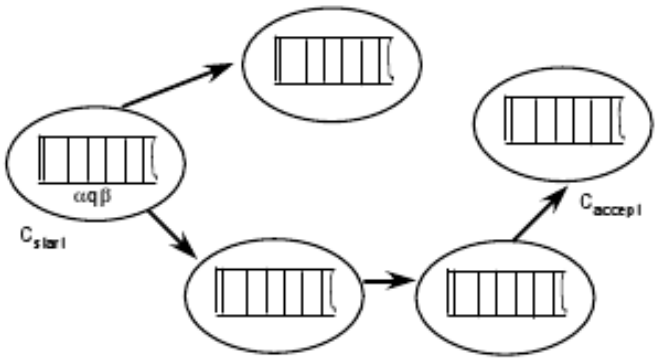
$SPACE(S(n)) \subseteq NSPACE(S(n))$

$SPACE(S(n)) \subseteq NSPACE(S(n))$

- Máquina Determinista X Máquina N-Determinista

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

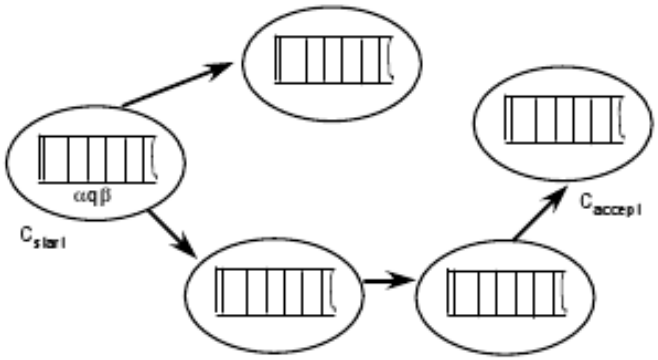
$NSPACE(S(n)) \subseteq DTIME(2^{O(S(n))})$



- configuration path
  - $c \bullet S(n)$ bits por nó
  - Número máximo de nós dado por $2^{cS(n)}$

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

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



- Prova
  - Enumeração das configurações
    - Número máximo de nós dado por $2^{O(S(n))}$
  - Conectividade entre estado inicial e de aceitação
    - Busca em largura $O(|E| + |V|)$

Classes de Complexidade

- $PSPACE = U_{c>0} SPACE(n^c)$
  - $3SAT \in PSPACE$
    - Enumeração das possibilidades
    - Reuso de espaço
  - $NP \subseteq PSPACE$
- $NPSPACE = U_{c>0} NSPACE(n^c)$
- $L = SPACE(logn)$
  - e.x $EVEN = \{$x: x has even number of $1s\}$
  - $3SAT \in L$ (Problema sem resposta)
  - equivalente a pergunta $NP = L$
- $NL = NSPACE(logn)$
  - $PATH \in L$ (Problema sem resposta)
  - equivalente a pergunta $L = NL$

Classes de Complexidade

- $PSPACE = U_{c>0} SPACE(n^c)$
  - $3SAT \in PSPACE$
    - Enumeração das possibilidades
    - Reuso de espaço
  - $NP \subseteq PSPACE$
- $NPSPACE = U_{c>0} NSPACE(n^c)$
- $L = SPACE(logn)$
  - e.x $EVEN = \{$x: x has even number of $1s\}$
  - $3SAT \in L$ (Problema sem resposta)
  - equivalente a pergunta $NP = L$
- $NL = NSPACE(logn)$
  - $PATH \in L$ (Problema sem resposta)
  - equivalente a pergunta $L = NL$

Classes de Complexidade

- $PSPACE = U_{c>0} SPACE(n^c)$
  - $3SAT \in PSPACE$
    - Enumeração das possibilidades
    - Reuso de espaço
  - $NP \subseteq PSPACE$
- $NPSPACE = U_{c>0} NSPACE(n^c)$
- $L = SPACE(logn)$
  - e.x $EVEN = \{$x: x has even number of 1$s\}$
  - $3SAT \in L$ (Problema sem resposta)
  - equivalente a pergunta $NP = L$
- $NL = NSPACE(logn)$
  - $PATH \in L$ (Problema sem resposta)
  - equivalente a pergunta $L = NL$

Classes de Complexidade

- $PSPACE = U_{c>0} SPACE(n^c)$
  - $3SAT \in PSPACE$
    - Enumeração das possibilidades
    - Reuso de espaço
  - $NP \subseteq PSPACE$
- $NPSPACE = U_{c>0} NSPACE(n^c)$
- $L = SPACE(logn)$
  - e.x $EVEN = \{$x: x has even number of $1s\}$
  - $3SAT \in L$ (Problema sem resposta)
  - equivalente a pergunta $NP = L$
- $NL = NSPACE(logn)$
  - $PATH \in L$ (Problema sem resposta)
  - equivalente a pergunta $L = NL$

Classes de Complexidade

- $PSPACE = U_{c>0} SPACE(n^c)$
  - $3SAT \in PSPACE$
    - Enumeração das possibilidades
    - Reuso de espaço
  - $NP \subseteq PSPACE$
- $NPSPACE = U_{c>0} NSPACE(n^c)$
- $L = SPACE(logn)$
  - e.x $EVEN = \{$x: x has even number of $1s\}$
  - $3SAT \in L$ (Problema sem resposta)
  - equivalente a pergunta $NP = L$
- $NL = NSPACE(logn)$
  - $PATH \in L$ (Problema sem resposta)
  - equivalente a pergunta $L = NL$

Completude

- $P \neq PSPACE$ (Problema sem resposta)
- Se $P = PSPACE$ então $P = NP$

PSPACE-hard ..

- A language A is *PSPACE-hard* if for every $L \in PSPACE$, $L \leq_p A$. If in addition $A \in PSPACE$ then A is *PSPACE-complete*

Completude

- $P \neq PSPACE$ (Problema sem resposta)
- Se $P = PSPACE$ então $P = NP$

PSPACE-hard ..

- A language A is *PSPACE-hard* if for every $L \in PSPACE$, $L \leq_p A$. If in addition $A \in PSPACE$ then A is *PSPACE-complete*

Completude

- $P \neq PSPACE$ (Problema sem resposta)
- Se $P = PSPACE$ então $P = NP$

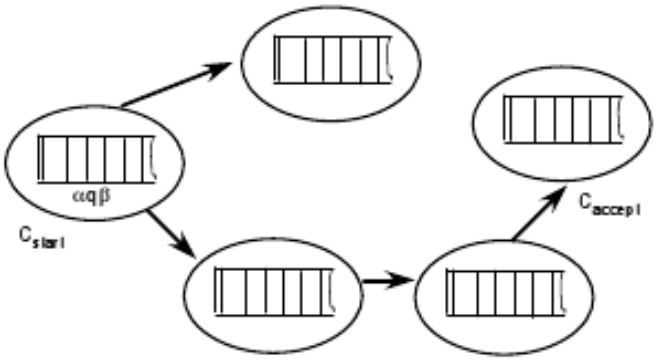
PSPACE-hard ..

- A language A is *PSPACE-hard* if for every $L \in PSPACE$, $L \leq_p A$. If in addition $A \in PSPACE$ then A is *PSPACE-complete*

Teorema de Savitch

- For any space-constructible $S : N \longrightarrow N$ with $S(n) \geq logn$, $NSPACE(S(n)) \subseteq SPACE(S(n)^2)$

Roteiro da prova: *configuration path* ..

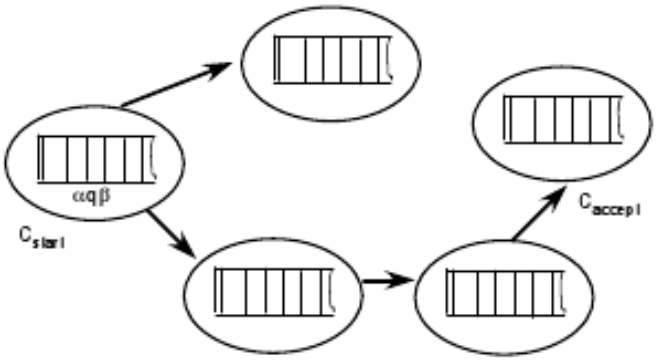


- Cada vértice pode ser descrito usando $cS(n)$ bits
- O grafo tem no máximo $2^{cS(n)}$ vértices

Teorema de Savitch

- For any space-constructible $S : N \longrightarrow N$ with $S(n) \geq logn$, $NSPACE(S(n)) \subseteq SPACE(S(n)^2)$

Roteiro da prova: *configuration path* ..

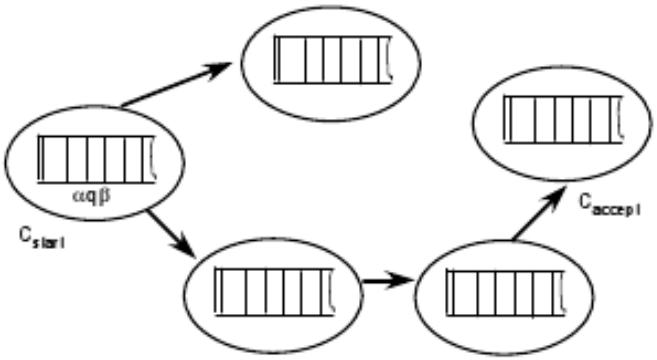


- Cada vértice pode ser descrito usando $cS(n)$ bits
- O grafo tem no máximo $2^{cS(n)}$ vértices

Teorema de Savitch

- For any space-constructible $S : N \longrightarrow N$ with $S(n) \geq logn$, $NSPACE(S(n)) \subseteq SPACE(S(n)^2)$

Roteiro da prova: *configuration path* ..



- Cada vértice pode ser descrito usando $cS(n)$ bits
- O grafo tem no máximo $2^{cS(n)}$ vértices

Teorema de Savitch - Algoritmo

Input: $G = (V, E)$, $s, t \in V$, $k \in \mathbb{N}$ (k a power of 2)
Output: TRUE if $d_G(s, t) \leq k$, FALSE otherwise

```
1: if k = 1 then
2:    return TRUE iff (s, t) ∈ E
3: else
4:    for all w ∈ V do
5:       b1 ← REACH(G, s, w, k/2)
6:       b2 ← REACH(G, w, t, k/2)
7:       if b1 ∧ b2 then
8:          return TRUE
9:       end if
10:   end for
11:   return FALSE
12: end if
```

- Se $k = 1$, requer $O(cS(n))$ bits ou $log|V|$ bits
- Para o comando *for*:
  - Cada chamada recursiva requer 2 vértices, $log|V|$
  - O valor $k/2$, que ocupa não mais que $logk$ bits
  - No total, para o comando *for*, $O(logk + log|V|)$

Teorema de Savitch - Algoritmo

```
Input:  G = (V, E), s, t ∈ V, k ∈ ℕ (k a power of 2)
Output: TRUE if d_G(s, t) ≤ k, FALSE otherwise
 1: if k = 1 then
 2:   return TRUE iff (s, t) ∈ E
 3: else
 4:   for all w ∈ V do
 5:     b_1 ← REACH(G, s, w, k/2)
 6:     b_2 ← REACH(G, w, t, k/2)
 7:     if b_1 ∧ b_2 then
 8:       return TRUE
 9:     end if
10:   end for
11:   return FALSE
12: end if
```

- Se $k = 1$, requer $O(cS(n))$ bits ou $log|V|$ bits
- Para o comando *for*:
  - Cada chamada recursiva requer 2 vértices, $log|V|$
  - O valor $k/2$, que ocupa não mais que $logk$ bits
  - No total, para o comando *for*, $O(logk + log|V|)$

Teorema de Savitch - Algoritmo - cont

```
Input:  G = (V, E), s, t ∈ V, k ∈ ℕ (k a power of 2)
Output: TRUE if d_G(s, t) ≤ k, FALSE otherwise
 1: if k = 1 then
 2:   return TRUE iff (s, t) ∈ E
 3: else
 4:   for all w ∈ V do
 5:     b_1 ← REACH(G, s, w, k/2)
 6:     b_2 ← REACH(G, w, t, k/2)
 7:     if b_1 ∧ b_2 then
 8:       return TRUE
 9:     end if
10:   end for
11:   return FALSE
12: end if
```

- Por definição, uma chamada recursiva tem a complexidade dada $S(|V|, k/2)$
- Como o espaço da primeira chamada recursiva pode ser reutilzado na segunda, a complexidade total $S(|V|, k)$ é definida recursivamente:
  - $log|V|$, se $k = 1$
  - $log|V| + S(|V|, k/2)$, caso contrário

Teorema de Savitch - Algoritmo - cont

```
Input:  G = (V, E), s, t ∈ V, k ∈ ℕ (k a power of 2)
Output: TRUE if d_G(s, t) ≤ k, FALSE otherwise
 1: if k = 1 then
 2:    return TRUE iff (s, t) ∈ E
 3: else
 4:    for all w ∈ V do
 5:       b1 ← REACH(G, s, w, k/2)
 6:       b2 ← REACH(G, w, t, k/2)
 7:       if b1 ∧ b2 then
 8:          return TRUE
 9:       end if
10:    end for
11:    return FALSE
12: end if
```

- Se existir um caminho de tamanho k, no máximo temos que $k \leq |V|$
- Assim, a chamada para REACH no máximo tomará, $S(|V|, 2|V|) = log2^{O(S(n)} xlog2^{O(S(n)}$

Devido a este teorema temos que:

$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE = NSPACE$