Complexidade de Espaço

Marlon Henry Schweigert
Centro de Ciências Tecnológicas
Universidade do Estado de Santa Catarina
Joinville, Brasil
marlon.henry@edu.udesc.br

Alexandre Mendonça Fava
Centro de Ciências Tecnológicas
Universidade do Estado de Santa Catarina
Joinville, Brasil
alexandre.fava@edu.udesc.br

Space Complexity is one of the most recurring problems in solving computational problems, along with the Time Complexity. The similarities between these two fundamental complexity is such that both use asymptotic notation. In addition to notation, space and time classes have similar names relative to their definition. In the middle of this article, there are also some algorithms that will serve as example to illustrate how time complexity is measured. Finally, it will be mentioned the notation and complexity of space, the Savitch theorem that relates two classes P-space and NP-space. Briefly this article brings concepts of definition, measurement, classes of space complexity, as well as proof of the Savitch theorem.

Complexidade de Espaço é um dos problemas recorrentes na resolução de problemas computacionais, junto a Complexidade de Tempo. As similaridades entre essas duas complexidades fundamentais é tamanha que ambas utilizam notação assintótica. Além da notação, as classes de espaço e tempo possuem nomes similares relativos a sua definição. No decorrer do artigo, existem também alguns algoritmos que servirão de ilustração para exemplificar como a complexidade de tempo é medida. Por fim será abordado além da notação e das complexidades de espaço, o Teorema de Savitch que relaciona duas classes de espaço P-SPACE e NP-SPACE. Resumidamente este artigo trás conceitos de definição, medição, classes da complexidade de espaço, além da prova do Teorema de Savitch.

1 Conceitos

A compreensão dos conceitos básicos ajudará nos tópicos futuros abordados, pois embora esse seja um artigo de cunho didático (seminário) ele trará consigo alguns conceito e terminologias técnicas que são de fundamental conhecimento para a compreensão deste documento.

Saber acerca das Máquinas de Turing é fundamental para compreender alguns conceitos da complexidade de espaço, além disto o leitor deve estar familiarizado com a notações assintóticas, muito utilizada em complexidade de tempo. Saber também as classes de tempo ajudaria a compreender as classes de espaço, todavia este não se faz um requisito essencial. Assume-se que o leitor também esteja familiarizado com provas de teoremas e algoritmos.

1.1 Complexidade de Espaço

Complexidade de espaço de uma Máquina de Turing está relacionado a entregar uma função f a qual diga quantas células serão necessárias para computar algum problema. Em outras palavras, ao ser inserida uma palavra na fita da Máquina de Turing, a quantidade de células a mais que serão usadas da fita irá determinar a complexidade de espaço.

Se a complexidade da máquina M com entrada w, sendo M uma máquina de turing decisora, podemos afirmar que o tamanho máximo utilizável da fita será f(n) para computar w. Caso a máquina M seja uma máquina não determinista, definimos f(n) retornará o número máximo de células de todos os ramos. Ou seja, o pior caminho de computação.

Submitted to:
V SUCU

1.2 Exemplo de Complexidade de Espaço

L = {W | W respeita (10)* ou (01)*}.

Exemplos de entradas da linguagem L:

- 101010
- 0101

Seja a MT ML decisora de L. A sua programação segue o seguinte algoritmo:

1. Vá um passo a frente
2. Repita até encontrar _:
   (a) Dado o valor da cabeça na posição atual, verifique se a posição anterior possui um valor diferente.
   (b) Caso seja diferente continue, caso seja igual, recuse.
   (c) Avance duas casas.
3. Ao encontrar _, verifique se a última casa é 1. Se for 1 escreva três zeros no final e aceite.
4. Ao encontrar _, verifique se a última casa é 0. Se for 0 aceite.

O seu pior caso pode ser considerado pela fórmula $f(n) = n+3$, visto que a entrada pertencerá a L, logo a máquina escreverá três zeros ao final.

O seu melhor caso pode ser considerado pela fórmula $g(n) = n$, visto caso a entrada não pertença a L, será recusado antes de escrever três zeros ao final.

Para as seguintes entradas temos as seguintes estimativas:

Tabela 1: Exemplos de entrada para ML

w	#w	f(n)	Espaços utilizados por ML
1010	4	7	4
0101	4	7	7
101010	6	9	6

1.3 Notação de Complexidade de Espaço

Assim como na complexidade de tempo, usaremos as principais notações onde elas são[6]: $O(f(n))$ (*big o*), $\Theta(f(n))$, $\Omega(f(n))$. De forma simplista, podemos dizer que as notações de complexidade de espaço podem ser interpretadas da seguinte forma:

- $O(f(n))$ refere-se ao pior caso.
- $\Theta(f(n))$ refere-se ao caso médio, ou onde o pior e melhor caso são iguais.
- $\Omega(f(n))$ refere-se ao melhor caso.

1.4 Exemplos de notação de complexidade de espaço

Utilizando o exemplo da máquina ML definido na seção 1.2, temos o melhor caso definido $g(n) = n$ e o pior caso definido por $f(n) = n+3$. Nesse exemplo, $n+3 = O(n)$ e $n = \Omega(n)$, logo o melhor e pior caso são iguais. Por esse motivo, a complexidade dessa máquina de turing será $\Theta(n)$, pois $O(n) = \Omega(n)$[5].

Nem sempre é fácil definir a função da complexidade média. Nesse caso conseguimos decidir facilmente pois a complexidade do pior caso e do melhor caso são iguais, logo seu caso médio pode ser unicamente igual ao pior e melhor caso, ao mesmo tempo. Porém em outros casos, podemos ter complexidades diferentes. Em casos onde o melhor e pior caso são diferentes, devemos ter a função que estima a média de todos os casos. Torna-se difícil decidir a complexidade média caso as possibilidades de entrada sejam extensas, ou até mesmo infinitas[3].

2 Medição

A medição da complexidade da espaço assemelhasse da forma como é medida a complexidade de tempo, tanto até que ambas possuem a mesma representação assintótica e de entrada: T(n). Vale destacar que em um algoritmo hipotético sua complexidade de tempo pode ser T(n)=O(n^2) e sua complexidade de espaço T(n)=O(n), ou o contrário. Tudo depende depende do comportamento do algoritmo, em alguns casos sua complexidade de tempo e espaço podem ser iguais, ou completamente diferentes.

Na complexidade de tempo a sigla T designa "função de complexidade de tempo", e na complexidade de espaço T é chamado de "função de complexidade de espaço", onde a representação dos dados é importante para se determinar o espaço utilizado:

- Se os dados possuem representação natural, (ex. matriz) considera-se apenas o espaço extra utilizado pelo algoritmo;
- Se os dados podem ser representados de várias formas (ex. grafo) deve-se considerar o espaço utilizado por sua representação (matriz ou lista encadeada).

Normalmente as complexidades de espaço tendem a serem requisitadas em casos de algoritmos **not-in-place** ou **out-of-place**, pois em casos mais comuns de **in-place** a complexidade de espaço tende a ser O(1). Vale lembrar que **in-place** são denominados os algoritmos que trabalham com suas entradas sem utilizar um estrutura de dados auxiliar. Para mostrar de maneira prática como é medida a complexidade de espaço, iremos utilizar de dois exemplos, **in-place** e **not-in-place**.

Figura 1: Função not-in-place.

```
ACHA-MAIOR (v: vetor de double) {
  pos ← 0
  maior ← 1
  for i ← 2 to v.length {
    if (v[i] > v[maior])
       maior ← i
  }
  return maior
}
```

2.1 Exemplo: in-place

O função presente na figura 1, escrita em pseudo-código tem como objetivo encontrar o maior valor de um determinado vetor de tamanho n.

Ao calcularmos a complexidade de tempo obtém-se 3n+1, porém as definições da própria complexidade permitem "arredondar"este valor, finalizando-o em T(n)=O(n).Entretanto sua complexidade de espaço é O(1), pois nenhum espaço a mais é requisitado. Mesmo que alguns espaços na memória sejam criados para armazenar o maior número (por exemplo), estes espaços requisitados são pequenos demais para serem considerados, ou seja, da mesma forma que a complexidade de tempo "arredondou"seu valor descartando desnecessidades, a complexidade de espaço faz o mesmo.

2.2 Exemplo: not-in-place

O programa presente na figura 2, escrito em linguagem C, é um algoritmo menos conhecido de ordenação denominado de Counting Sort.

Ao calcularmos a complexidade de tempo obtém-se O(n+k), onde n representa o tamanho do vetor de entrada que será ordenado e k representa o tamanho do vetor auxiliar que foi criado. Neste método de ordenação, o algoritmo além de ter que percorrer seu vetor de entrada de ponta à ponta, ele ainda é obrigado a percorrer o vetor auxiliar criado, para finalizar sua ordenação. É justamente por criar um vetor auxiliar que este é um algoritmo **not-in-place**, possuindo sua complexidade de espaço T(n)=O(k), onde k representa o maior valor encontrado no vetor.

3 Classes da complexidade de espaço

Estimamos a complexidade de espaço por duas categorias[4]:

- A classe P é o conjunto de todas as linguagens decidíveis deterministicamente em tempo polinomial.
- A classe NP é o conjunto de todas as linguagens decidíveis não deterministicamente em tempo polinomial.
- A classe P-SPACE é o conjunto de todas as linguagens decidíveis deterministicamente em espaço polinomial.
- A classe NP-SPACE é o conjunto de todas as linguagens decidíveis não deterministicamente em espaço polinomial.
- A classe D-SCAPE é um recurso computacional descrevendo a disponibilidade de memória para uma máquina de Turing.

Figura 2: Função not-in-place.

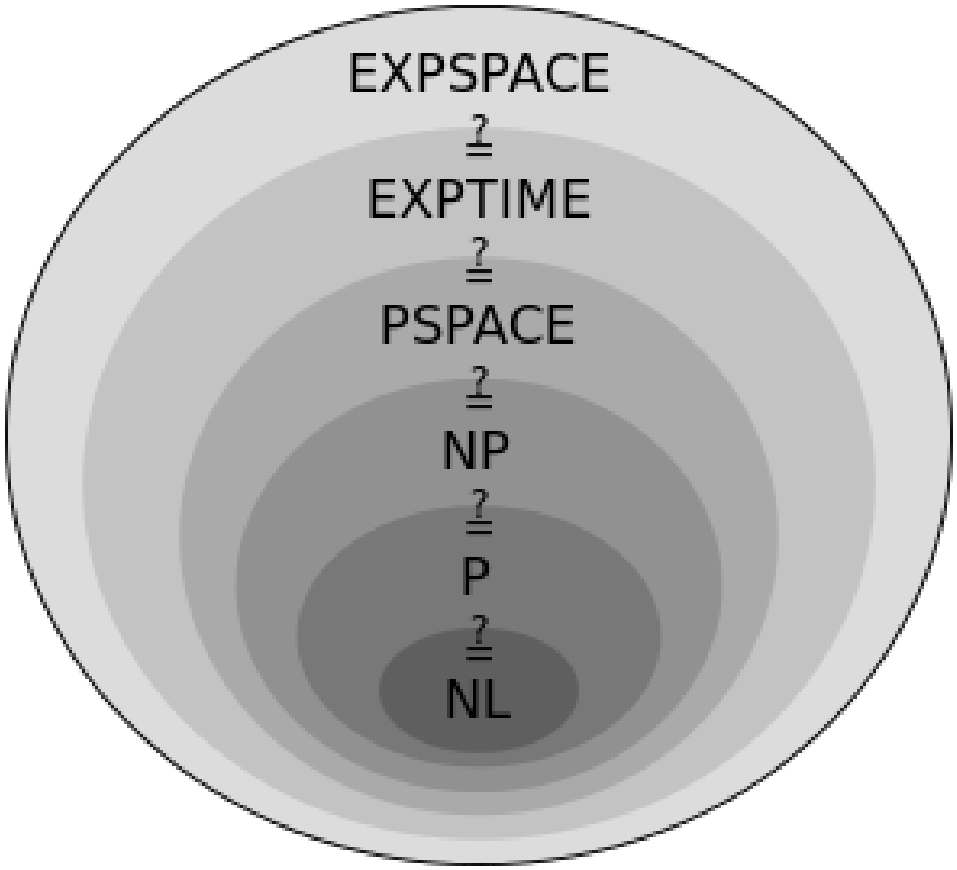
```c
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include <ctype.h>
# define MAX 100001

struct data
{
    int number;
    char key[100];
} DataBase[MAX], VectorSort[MAX];

int CounterNum[MAX];
int size = 0;

int main (void)
{
    int i = 0;

    while (scanf("%d%s", &DataBase[size].number, DataBase[size].key) >= 1)
        size++;

    int aux[2] = {0, 0};
    for (i = 0; i <= size; i++)
        aux[DataBase[i].number]++;

    aux[1] += aux[0];

    for (i = size - 1; i >= 0; i--)
        VectorSort[--aux[DataBase[i].number]] = DataBase[i];

    for (i = 0; i < size; i++)
        printf("Number: %d  ---  Key: %s\n", VectorSort[i].number, VectorSort[i].key);

    return 0;
}
```

- A classe EXP-SPACE é o conjunto de todos os problemas de decisão solúveis por uma máquina de Turing determinística em espaço O(2p(n)) onde p(n) é uma função polinomial de n. (Alguns autores restringem p(n) para uma função linear, mas a maioria chama a classe resultante de ESPACE.) Se, por outro lado, nós usamos uma máquina não determinística, teremos a classe NEXP-SPACE, que é igual a EXP-SPACE pelo Teorema de Savitch que será explicado na próxima secção.
- A Classe N-SPACE(f(n)) é um conjunto de problemas de decisão que podem ser resolvido por uma máquina de Turing não-determinística usando espaço O(f(n)), e tempo ilimitado.

Tabela 2: Classes de Complexidade de Espaço

Classe de Complexidade	Modelo de Computação	Limitação de Recursos
D-SPACE(f(n))	Máquina de Turing Determinística	Espaço f(n)
L	Máquina de Turing Determinística	Espaço O(log n)
P-SPACE	Máquina de Turing Determinística	Espaço poly(n)
EXP-SPACE	Máquina de Turing Determinística	Espaço 2poly(n)
N-SPACE(f(n))	Máquina de Turing Não-Determinística	Espaço f(n)
NL	Máquina de Turing Não-Determinística	Espaço O(log n)
NP-SPACE	Máquina de Turing Não-Determinística	Espaço poly(n)
NEXP-SPACE	Máquina de Turing Não-Determinística	Espaço 2poly(n)

Figura 3: Diagrama das classes de complexidade de espaço[1].



Existem outras classes que são definidas utilizando outras alterações de máquina de turing, como as seguintes:

- BPP, ZPP e RP em máquinas de turing probabilísticas.
- AC e NC em circuítos boolianos.
- BQP e QMA em máquinas de turing quânticas.

Existem outras classes que são formadas conforme as características do problema. São alguns exemplos[1]:

- *#P* que é uma classe importante que inclui problemas de contagem, a qual não é um problema de decisão.
- *IP* e *AM* são classes de prova iterativa.
- *ALL* é a classe de todos os problemas de decisão.

4 Relação entre as classes de espaço

Para dar início a esta secção, podemos começar apresentando uma das relações mais conhecidas quando se fala de complexidade de espaço: P-SPACE = NP-SPACE. Todavia, mesmo com essa informação a pergunta de um milhão de dólares ainda continua sem resposta: P=NP?

Por tal razão, ao invés de ficarmos discorrendo de forma leviana, por cada uma das complexidades de espaços, iremos nesta secção fornecer de forma detalhada prova que implica que P-SPACE = NP-SPACE, prova essa encontrada através do "Teorema de Savitch"proposta por Walter Savitch em 1970.

A demonstração de Walter mostra que máquinas deterministas podem simular máquinas não deterministas usando uma pequena quantidade de espaço. Segue-se o teorema abaixo:

Teorema 4.18. *Seja* $f : \mathbb{N} \to \mathbb{R}^+$ *tal que* $f(n) \geq n$. *Então*

$$nspace(f(n)) \subseteq space(f(n)^2).$$

A prova se inicia assumindo L $\in$

NSPACE(f(n)) e N uma máquina de Turing não-determinista que decide L tal que NSPACE$_N(n) = O(f(n))$.

Considere-se a máquina de Turing igual a N mas que imediatamente antes de transitar para o estado de aceitação limpa a sua fita de entrada e posiciona o sensor na célula mais à esquerda. Para não perdemos generalidade iremos designar essa máquina também por N.

Considere-se o seguinte algoritmo atingível$_e$, em que e é um natural, que recebe duas configurações c1 e c2 de N e um número natural t que é uma potência de 2 e devolve verdadeiro se numa árvore de computação de N há um ramo que evolui de c1 até c2 em t ou menos passos. Se não evolui, atingível$_e$ devolve falso:

se $t = 1$ então
    se $c_1 = c_2$ ou $c_1 \to_N c_2$ então
        devolve *verdadeiro*
    caso contrário
        devolve *falso*
    fim se
fim se
se $t > 1$ então
    para cada configuração c de N usando espaço e
        se $atingivel_e(c_1, c, \frac{t}{2}) = verdadeiro$ e $atingivel_e(c, c_2, \frac{t}{2}) = verdadeiro$
então
            devolve *verdadeiro*
        fim se
    fim para cada
fim se
devolve *falso*

Considere-se a máquina de Turing M que ao receber uma palavra de entrada w de tamanho n, faz:

$e := 1$
repete
    $r := atingivel_e(c^w_{inicial}, c_{aceitacao}, T^w)$
    se $r = verdadeiro$ então
        termina aceitando w
    caso contrário
        $eusado := falso$
        para cada configuração c de N usando espaço $e + 1$
            se $atingivel_{e+1}(c^w_{inicial}, c, T^w) = verdadeiro$ então
                $eusado := verdadeiro$
                break
            fim se
        fim para cada
        se $eusado = verdadeiro$ então
            $e := e + 1$
        caso contrário
            termina rejeitando w
        fim se
    fim se
fim repete

onde $c^w_{inicial}$ é a configuração inicial de N para w, $c_{aceitacao}$ é a configuração de aceitação de N, e T^w é o número máximo de configurações de N usando $nspace_N$ (|w|).

Então:

(1) Lac(M) = L. Observa-se que w Lac(M) sse M aceita w sse existe uma evolução não-determinista de N desde a sua configuração inicial para w até à sua configuração de aceitação sse N aceita w sse w L.

(2) Lrj(M) = L. Observa-se que w Lrj(M) sse M rejeita w sse não existe uma evolução não-determinista de N desde a sua configuração inicial para w até à sua configuração de aceitação sse N rejeita w sse w / L sse w L.

(3) spaceM(n) = O(f(n) 2). Observe-se que de cada vez que atingível se chama a si próprio recursivamente ele guarda c1, c2 e t na pista. Observe-se que para guardar estes valores ele precisa de O(nspaceN (n)) células no máximo. Por outro lado, dado que em cada passo da recursão t é dividido a metade, então a recursão irá ser feita O(log(T w)) vezes, isto é, O(log(2nspaceN (n))) vezes.

Assim o número máximo de células usadas por M é O(nspace2 N (n)), isto é, O(f 2 (n)). Assim L space(f(n) 2).

Este resultado permite-nos identificar classes de complexidade espaciais deterministas e não-deterministas.

5 Conclusão

Embora a complexidade de espaço não seja tão citada, em relação a complexidade de tempo, é notável que ambos os recursos (tempo e espaço) são importante e devem ser levados em consideração no momento de formular um programa. Cotidianamente, se preocupa mais com tempo de execução (complexidade de tempo) e espaço de armazenamento, porém este espaço de armazenamento não é considerado como sendo complexidade de espaço, como fora visto nas demais secções deste texto.

A complexidade de espaço não possui tantos problemas em aberto como a complexidade de tempo, gerando ainda mais desinteresse por esse assunto. Porém, é notável que os programadores ainda levam em consideração o espaço em seus programas, pois ninguém quer ser vítima de um buffer overflow.

Referências

[1] fonte pública (2017): *Complexidade Computacional*. Available at `https://pt.wikipedia.org/wiki/Complexidade_computacional`.

[2] Michael Sipser (2006): *Introduction to the theory of computation*, second edition edition. Thomson Course Technology.

[3] UDESC (2017): *Complexidade de Algoritmos*. Available at `https://buchinger.github.io/CAL/index.html`.

[4] UFPR: *Teoria da Computação*. Available at `http://www.dainf.ct.utfpr.edu.br/~murilo/teoria/notes/TCparte3.pdf`.

[5] UFPR (2017): *Complexidade Computacional*. Available at `http://www.dainf.ct.utfpr.edu.br/~murilo/teoria/notes/TCparte3.pdf`.

[6] USP: *Notação Assintótica*. Available at `https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/Oh.html`.