SOBRE A COMPLEXIDADE DE JOGOS COMBINATÓRIOS

Danilo Artigas da Rocha

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Celina Miraglia Herrera de Figueiredo, D.Sc.

Prof. Mario Roberto Folhadela Benevides, Ph.D.

Prof. Luerbio Faria, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

FEVEREIRO DE 2006

ROCHA, DANILO ARTIGAS DA

Sobre a Complexidade de Jogos Combinatórios [Rio de Janeiro] 2006

XI, 85 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1 - Complexidade computacional

2 - Jogos combinatórios

I. COPPE/UFRJ II. Título (série)

À Maria, minha mãe.

Agradecimentos

À minha namorada Letícia, em um primeiro instante por ter conseguido me dividir com uma dissertação e no momento final por ter perdido completamente o namorado, mas ainda assim, conseguiu me dar carinho e apoio para continuar. Obrigado pela paciência e atenção, Eu te amo!

Aos meus pais, vocês que jamais entenderam porque tanto sofrimento por causa de uns "joguinhos", mas me deram muito amor, apoio e incentivo.

À Celina, agradeço por ter me dado liberdade na escolha deste tema e, principalmente, por desde a minha graduação, ajudar a atribuir sentido as minhas idéias e me exigir disciplina. Eu precisava!

Aos Profs. Luerbio e Mario, por terem aceitado o convite para fazer parte desta banca, suas recomendações contribuíram significativamente para a qualidade deste trabalho.

Aos Profs. Fábio Protti, Luerbio e Jayme, pelos excelentes cursos ministrados, em particular pelo curso de complexidade de algoritmos, o qual foi decisivo na escolha do tema da dissertação.

À Dona Margarida, pela eterna simpatia, e pela excelência dos serviços prestados aos usuários da biblioteca do IM.

A Rafael Bernardo e Rodrigo Hausen, por discutirem sobre o meu assunto de pesquisa, pelas constantes piadas (mesmo pelas sem graça), por me fazer pensar nos mais estranhos assuntos, pelas dicas de LaTeX (seria muito difícil

cumprir o prazo sem vocês) e principalmente pelo apoio ao longo de todo o mestrado.

Aos amigos Fabiano, Noemi, Priscila e Raquel, por ouvirem minhas aflições e medos e compartilharem um pouco de suas angústias comigo.

A Bel, Débora e Luciane, por me fazerem pensar em aplicações sociais dos meus objetos de estudo, e pela grande amizade, eu adoro vocês!

A Bernardo, Ivomar, Pedrinho e Marina, por terem se tornado parceiros não apenas no Fundão, mas também fora dele.

Aos amigos de outras datas, Bia, Caique, Cecília, Fábio Matema, Luciana e Victor.

Às novas amizades formadas, Bruno Carioca, Elias Barenba, Fernanda Nocchi, Mitre Dourado e Thiago Mineiro.

A Miguel Eduardo Bruno, pelo apoio.

Por fim, agradeço a CAPES pela ajuda financeira.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

SOBRE A COMPLEXIDADE DE JOGOS COMBINATÓRIOS

Danilo Artigas da Rocha

Fevereiro/2006

Orientadora: Celina Miraglia Herrera de Figueiredo
Programa: Engenharia de Sistemas e Computação

Jogo combinatório é um jogo de dois jogadores, I e II, onde cada jogador faz um lance por vez e I é o primeiro a jogar. As regras de movimentação, bem como as de término do jogo, são bem definidas. A todo momento ambos os jogadores possuem total informação sobre quais os possíveis próximos lances do oponente (não há elementos secretos ou aleatórios). São exemplos populares de jogos combinatórios: Xadrez, Dama e GO. Neste trabalho estudamos a complexidade de jogos combinatórios com particular interesse na classe $PSPACE$, classe que concentra um grande número de jogos combinatórios. Para estudar esta classe exibimos resultados clássicos sobre complexidade de espaço. Consideramos os jogos GEOGRAPHY, HEX, GO e G_1 como nossos exemplos de jogos combinatórios difíceis e desenvolvemos a teoria tendo esses jogos como base. Para tal, consideramos como modelos de computação: Máquina de Turing determinística, não determinística e alternada, onde a última é uma generalização das duas primeiras.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ON THE COMPLEXITY OF COMBINATORIAL GAMES

Danilo Artigas da Rocha

February/2006

Advisor: Celina Miraglia Herrera de Figueiredo
Department: Systems Engineering and Computer Science

A combinatorial game is a game with two players, I and II, where each player makes a move at a time, and I is the first to play. The rules of the game as well as the rule to finish the game are well defined. At each time, both players have all knowledge about the subsequent movements of the other player (there are no secret nor random elements). Chess, Checkers and GO are popular examples of combinatorial games. In this work, we study the complexity of combinatorial games. The $PSPACE$ class is studied as a space complexity class containing most of the combinatorial games. We consider the games: GEOGRAPHY, HEX, GO and G_1 as examples of hard combinatorial games. We consider as models of computation the following variations of Turing machines: deterministic, nondeterministic and alternating, in which the latter is a generalization of the two former ones.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Complexidade de jogos combinatórios

Um *jogo combinatório* é um jogo de dois jogadores, I e II, onde cada jogador faz um lance por vez e I é o primeiro a jogar. As regras de movimentação, bem como as de término do jogo, são bem definidas. A todo momento ambos os jogadores possuem total informação sobre quais os possíveis próximos lances do oponente (não há elementos secretos ou aleatórios). São exemplos populares de jogos combinatórios: Xadrez, Dama, HEX, GO e GEOGRAPHY.

Existem problemas de diferentes naturezas associados aos jogos combinatórios. Uma ampla abordagem sobre o assunto está presente em [2]. Nosso interesse será estudar a *complexidade* de jogos combinatórios. São ótimas referências para o estudo de complexidade [1, 6, 11].

Para classificarmos a complexidade de jogos combinatórios, precisamos associar a cada jogo um problema de decisão. O problema de decisão associado será:

Dados: As regras e uma instância qualquer de um jogo combinatório.

Pergunta : O jogador I vence a partida?

O problema de decisão definido no parágrafo anterior sugere uma pergunta: O que é uma instância de um jogo combinatório? Para entendermos a importância dessa pergunta, notemos que, dada uma posição qualquer do jogo de xadrez no seu tabuleiro 8×8, sabemos responder se as Brancas (nosso jogador I) vencem a partida com um algoritmo $O(1)$. Isto é possível pois temos um número fixo de peças e um tabuleiro de tamanho fixo. Logo, temos um limite superior constante para o número de posições distintas. Podemos portanto decidir se as Brancas vencem o jogo fazendo um "backtracking" que terá tempo constante.

Trabalharemos portanto com generalizações das versões originais dos jogos. Ao invés de tabuleiros de tamanho fixo usaremos tabuleiros finitos mas uma instância genérica será: Um tabuleiro $n \times n$ e uma posição qualquer nesse tabuleiro. Para alguns jogos, como GO, essa generalização será bem natural, não altera as regras do jogo. Para o Xadrez (ver [5, 17]) uma generalização altera toda sua essência.

Estudando a complexidade de jogos combinatórios vimos a importância da classe $PSPACE$, a classe dos problemas decididos utilizando espaço polinomial. Um grande número de jogos são *completos* para esta classe, encontramos alguns exemplos em [4, 14]. Em linhas gerais, todos os jogos com limite polinomial para o número de jogadas são $PSPACE$-completos.

Seja ξ uma classe de complexidade. Um problema Π é ξ-*completo* se:

(i) $\Pi \in \xi$;

(ii) Para todo problema $\Pi' \in \xi$. Π' se transforma polinomialmente a Π (abreviadamente $\Pi' \propto \Pi$).

Se Π satisfaz o último item, Π é denominado ξ-*difícil*.

Além de tratar sobre a classe $PSPACE$ e sua relação com os jogos combinatórios, fizemos uma abordagem sobre complexidade de espaço. Como este tipo de complexidade não é tão comumente estudada quanto a complexidade de tempo, exibimos seus resultados iniciais e algoritmos básicos.

Por fim, exibimos uma generalização dos modelos usuais de máquina de Turing. Essa máquina generalizada chama - se ATM [16], *máquina de Turing alternada*, e é um poderoso instrumento para a análise de jogos combinatórios. Relacionaremos a computação desta máquina com a DTM e a $NDTM$.

A ATM pode ser usada para provar resultados de **E**-completude, onde **E** é a classe dos problemas resolvidos em tempo exponencial. Chamamos a atenção para o fato que **E** contém a classe $PSPACE$. Em particular estudaremos G_1, um jogo sobre fórmulas booleanas que é **E**-completo. A definição do jogo e o resultado de completude estão em [15].

Infelizmente, o tempo para conclusão da dissertação se esgotou e não conseguimos suficiente domínio sobre as ferramentas necessárias para a prova de completude de G_1. Exibiremos que G_1 pertence a **E** e daremos a idéia da demonstração que G_1 é **E**-difícil.

Nossa contribuição com este trabalho encontra - se na exposição unificada dos temas complexidade de espaço e jogos combinatórios. E, principalmente, enriquecendo as demonstrações presentes na literatura, seja com a exibição de mais detalhes ou adaptando algumas idéias existentes a novos contextos ou criando demonstrações novas para resultados conhecidos.

1.2 Organização da dissertação

Esse texto é um survey que utiliza como fontes básicas 2 livros: Aho, Hopcroft e Ullman [1] e Papadimitriou [11]. As definições e notações seguem, predominantemente, o livro [11]. O uso das demais referências é explicitado a seguir quando descrevemos a organização e o conteúdo de cada capítulo.

No capítulo 2 introduzimos as notações e definições principais a serem utilizadas neste trabalho. São definidas máquinas de Turing, classes de complexidade e problemas completos. Além disso são feitos os resultados básicos para o estudo de complexidade. As referências para este capítulo são [1, 6, 11].

No capítulo 3 definimos a classe $PSPACE$ e seus problemas básicos, e também exibimos jogos combinatórios completos para tal classe. Em particular, tratamos os jogos GEOGRAPHY [8, 11, 14] e o HEX [4].

No capítulo 4 voltamos nossas atenções para jogos combinatórios com complexidade superior a $PSPACE$, considerando dois jogos: GO e G_1. Ao tratar do GO estamos exibindo um problema $PSPACE$-difícil mas que não sabemos se é $PSPACE$. A referência desta parte é [8]. Na segunda parte exibimos a ATM, uma generalização de $NDTM$ e ferramenta útil para a prova de **E**-completude de G_1. A referência sobre ATM é [16] e sobre G_1 utilizamos [15].

Por fim, no capítulo 5 comentamos nossas conclusões sobre a dissertação e quais são os planos para o futuro.

Capítulo 2

Definições e resultados iniciais

2.1 Máquinas de Turing

Inicialmente, precisamos definir nosso modelo de computação. Ele será o mais simples possível, e ainda assim, terá o mesmo poder computacional que os modelos mais complexos de computação. Adotaremos como modelo a *máquina de Turing.*

Definição: Uma *máquina de Turing determinística* M (com k fitas), abreviadamente *DTM*, é uma quádrupla ordenada $M = (Q, \Sigma, \delta, q_0)$. Consiste de um controlador de estados, k fitas infinitas e k cabeças leitoras. As fitas estão divididas em infinitas células. Cada cabeça leitora percorre uma fita distinta, cada cabeça pode ler ou escrever sobre uma célula por vez.

Além dessas características gerais, uma *DTM* M possui:

- Q, um conjunto finito de estados.
- Σ, um conjunto finito de símbolos de M. Dentre os símbolos de Σ destacam - se b e $\triangleright$, respectivamente, o símbolo branco (as células que

não foram utilizadas nenhuma vez contém esse símbolo) e o símbolo inicial (todas as fitas são inicializadas com esse símbolo).

- Uma *função de transição*, δ: $Q \times \Sigma^k \rightarrow (Q \cup \{q_Y, q_N, q_H\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$. Onde q_Y, q_N, q_H são chamados estados de parada, e representam respectivamente, o *estado de aceitação*, o *estado de rejeição* e o *estado de parada*. Essa função caracteriza o "programa" de M. Ao ler uma $(k+1)$-upla $(q, \sigma_1, ..., \sigma_k)$, δ indica qual símbolo deve ser escrito na atual posição de cada uma das k cabeças leitoras e para qual lado cada uma delas deve se mover ($\leftarrow$, mover para esquerda; $\rightarrow$, mover para a direita), ou se devem ficar paradas ($-$). A essa alteração feita na fita e no posicionamento das cabeças leitoras chamamos de *movimento* ou *passo* da DTM. Para qualquer estado q, $\delta(q, \triangleright) = (q', \triangleright, \rightarrow)$, onde q' depende apenas de δ. Isso quer dizer que $\triangleright$ não pode ser ultrapassado à esquerda e nem pode ser apagado.

- q_0 é o estado inicial.

Seja x a entrada de M, definimos por $M(x)$ a saída de M. Notemos que δ está bem definida para toda $k+1$-upla $(q, \sigma_1, ..., \sigma_k) \in Q \times \Sigma^k$, logo M só pára se atingir alguns dos três estados de parada. Se M entra no estado q_Y, dizemos que M aceita x (usaremos apenas aceita, omitindo o x) e $M(x) =$ SIM. Se M entra no estado q_N, dizemos que M rejeita, $M(x) =$ NÃO. Se M entra no estado q_H, convencionaremos como $M(x)$, a sequência de símbolos contida na k-ésima fita de M.

M é inicializada no estado q_0; com $\triangleright x$ na primeira fita e $\triangleright$ nas demais fitas. Cada cabeça leitora começa sobre um $\triangleright$ distinto.

Uma DTM M *decide* uma linguagem L se $M(x) =$ SIM para todo $x \in L$ e $M(x) =$ NÃO para todo $x \notin L$.

A literatura sobre o assunto diverge muito quanto ao modelo padrão de máquina de Turing. Notemos que a máquina de Turing com uma fita é um caso particular da máquina com k fitas. Ainda neste capítulo (proposição 2.2 e seus corolários subsequentes) veremos que o modelo adotado não interfere no estudo de complexidade dos problemas.

Definição: Uma *máquina de Turing não determinística* (com k fitas), abreviadamente $NDTM$, é uma quádrupla ordenada $M = (Q, \Sigma, \delta, q_0)$.

Todos os componentes são idênticos aos da DTM, exceto a função de transição δ. Na $NDTM$, δ não representará uma função, mas uma relação $\delta \subset (Q \times \Sigma^k) \times [(Q \cup \{q_Y, q_N, q_H\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k]$. Ou seja, dados uma $(k+1)$-upla ordenada, consistindo de um estado e k símbolos, podem haver mais que uma próxima computação possível. A definição de *movimento* da $NDTM$ é análoga ao da DTM.

Seja x a entrada de uma $NDTM$ M. M *aceita* x se existe alguma computação, iniciada no estado q_0 que alcance o estado q_Y. Do contrário, se nenhuma computação de M alcança os estados q_Y ou q_H, então M *não aceita* x. Se a computação de M entra no estado q_H, então a sequência de símbolos contida na última fita será a saída de M. Em analogia com as $DTMs$ a saída de uma $NDTM$ será chamada de $M(x)$.

Note a assimetria entre aceitação e não aceitação de uma $NDTM$. Neste modelo o estado q_N não é tão importante como em uma DTM, em uma $NDTM$ se uma sequência de computação alcança o estado q_N não significa que a máquina não aceitará a entrada, uma entrada só não é aceita se nenhuma sequência de computação puder alcançar o estado q_Y ou q_H.

Uma $NDTM$ M decide uma linguagem L se para todo x em Σ^*, a seguinte propriedade é verdadeira: $x \in L$ se, e somente se, M aceita x.

Ao longo do texto, quando citarmos máquina de Turing, estaremos nos referindo a qualquer um dos dois modelos. Quando nos expressarmos dessa forma mais genérica, nosso interesse será em características comuns a ambos os modelos.

Definição: A *configuração* de uma máquina de Turing M com k fitas é uma $(2k+1)$-upla ordenada $(q, u_1, w_1, ..., u_k, w_k)$, onde q representa o estado de M; $u_j w_j$ representa a palavra da fita j, a cabeça leitora na fita j está posicionada no último símbolo de u_j, os demais símbolos de u_j representam a sequência à esquerda da cabeça leitora, e os símbolos de w_j representam os símbolos à direita da cabeça leitora. Se a máquina de Turing M passa da configuração C a C' em um único passo denotamos que $C \vdash_M C'$. Se M passa da configuração C a C' em r passos denotamos que $C \vdash_M^r C'$. Se existe alguma sequência de movimentos tal que, partindo da configuração C, M alcança C', denotamos que $C \vdash_M^* C'$.

Definição: Uma máquina de Turing M com k fitas e *com entrada e saída*, utiliza a fita 1 como fita de entrada, ou seja, no início da computação a fita 1 é inicializada com $\triangleright x$; utiliza a fita k como fita de saída, ou seja, a fita k é utilizada apenas para escrever a saída; as demais $k-2$ fitas são denominadas fitas de trabalho. A fita 1 é uma fita somente de leitura, a fita k é uma fita somente de escrita e as demais fitas são de leitura e escrita.

Toda a nossa teoria será desenvolvida utilizando máquinas com entrada e saída. A partir deste ponto, quando nos referirmos a uma máquina de Turing (determinística ou não determinística), ou abreviadamente DTM ($NDTM$), estaremos nos referindo a uma máquina de Turing M com k fitas e com entrada e saída.

2.2 Classes de complexidade

Definiremos agora o que são *classes de complexidade*. Este é o conceito adotado para qualificar os problemas quanto a sua dificuldade. Em linhas gerais, as classes de complexidade são determinadas pelo modelo de máquina de Turing adotado, e para qual aspecto da computação voltamos nossas atenções. As classes que classificam os problemas quanto ao número de movimentos executados pela máquina, são as classes de *tempo*. E as que classificam os problemas quanto a quantidade de células distintas utilizadas na computação, são classes de *espaço*.

Definição: Seja uma função $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$. Uma linguagem L pertence à classe $SPACE(f(n))$ se existe uma DTM M com *complexidade de espaço* $f(n)$ que decide L, i.e., a maior quantidade de células distintas usadas por alguma de suas $k-2$ fitas de trabalho de M para computar qualquer entrada de tamanho n é $f(n)$.

É importante observar que as fitas de entrada e de saída da DTM não são consideradas para a classificação da complexidade, portanto, existem classes onde $f(n) < n$, a mais relevante delas é $SPACE(\log n)$. Se a máquina não é de entrada e saída, consideramos todos as suas fitas como fita de trabalho, logo, não faz sentido falar sobre $SPACE(f(n))$, onde $f(n) < n$.

Definimos de forma análoga a classe $NSPACE(f(n))$, a única diferença é que a máquina será uma $NDTM$.

Definição: Uma linguagem L pertence a classe $TIME(f(n))$ se existe uma máquina de Turing M com *complexidade de tempo* $f(n)$ que decide L, i.e., o maior número de movimentos gastos por M para computar $M(x)$, para qualquer x onde $|x| = n$, é $f(n)$.

As definições de complexidade de tempo de uma $NDTM$ e de uma DTM são diferentes, logo a definição de $NTIME$ também será diferente.

Definição: Uma linguagem L pertence a classe $NTIME(f(n))$ se existe uma máquina de Turing M com *complexidade de tempo* $f(n)$ que decide L, i.e., seja $C_0 = (q_0, \triangleright, x, \triangleright, b, ..., \triangleright, b)$, a configuração inicial de M, se $C_0 \vdash_M^t C$, onde C é uma configuração utilizada na computação de x, então $t \leq f(n)$.

Definição: Seja $L \subseteq \Sigma^*$ uma linguagem. O *complemento* de L, denotado por $\overline{L}$, é a linguagem $\Sigma^* \setminus L$. Seja ξ uma classe de complexidade. O complemento de ξ, denotado por $co\xi$, é a classe $\{\overline{L} \mid L \in \xi\}$.

Claramente se ξ é uma classe determinística (em tempo ou espaço), então $\xi = co\xi$; as classes determinísticas são fechadas para o complemento. Seja M uma máquina determinística que decide L com um determinado limite de tempo (respectivamente espaço). Para decidirmos $\overline{L}$ com o mesmo limite de tempo (respectivamente espaço) utilizamos M', uma máquina que tem o funcionamento igual a M, mas atribui as respostas invertidas, M' responde SIM se M responde NÃO e vice e versa.

A próxima proposição é o início de uma série de resultados que nos ajudará a entender como se relacionam as diversas classes de complexidade.

Proposição 2.1. $TIME(f(n)) \subseteq SPACE(f(n))$

Prova: Se a cada novo passo de uma DTM M, M se deslocar para uma célula distinta sobre uma mesma fita, então ao final de $f(n)$ movimentos M terá utilizado espaço $f(n)$. Logo, uma máquina M que utilize tempo $f(n)$ conseguirá usar no máximo espaço $f(n)$. □

Antes de estudarmos as principais classes de interesse deste trabalho precisamos introduzir uma notação:

Definição: $SPACE(O(f(n))) = \bigcup_{c>0} SPACE(c.f(n))$

A definição de $TIME(O(f(n)))$ é análoga a anterior.

Nesse trabalho daremos atenção especial a classe $PSPACE$, classe dos problemas resolvidos por $DTMs$ que utilizam espaço polinomial. Formalmente:

$$\mathbf{PSPACE} = \bigcup_{\mathbf{i>0}} \mathbf{SPACE(O(n^i))} \tag{2.1}$$

Analogamente definimos:

$$\mathbf{NPSPACE} = \bigcup_{\mathbf{i>0}} \mathbf{NSPACE(O(n^i))} \tag{2.2}$$

As classes com limite polinomial em tempo, P e NP, são definidas de forma análoga.

Também temos interesse na classe **E**, classe dos problemas resolvidos em tempo exponencial:

$$\mathbf{E} = \bigcup_{c\geq 1} \mathbf{TIME(O(c^n))} \tag{2.3}$$

Exibiremos agora alguns resultados comparando os diversos modelos de máquina de Turing.

É importante avisar que os resultados a seguir foram extraídos de [1] e os detalhes foram desenvolvidos por mim.

Proposição 2.2. *Dada uma DTM M com k fitas, complexidade de tempo $T(n)$ e entrada x, $|x| = n$. Podemos construir uma DTM M' com uma única fita e complexidade de tempo $O(T^2(n))$ tal que M aceita x se, e somente se, M' aceita x.*

Prova: Seja $M = (Q, \Sigma, \delta, q_0)$, $Q \cap \Sigma = \varnothing$. Construiremos a máquina $M' = (Q', \Sigma', \delta', q_0')$ tal que: Os elementos de Q' são $(2k+1)$-uplas, $Q' =$

$(Q \cup \overline{Q}) \times (\Sigma \times D)^k$, onde $\overline{Q} = \{\overline{q} | q \in Q\}$ e $D = \{\leftarrow, \rightarrow, -, \divideontimes\}$; Σ' é um conjunto de $2k$-uplas, $\Sigma' = (\Sigma \times \{b, \triangle\})^k$; δ' e q'_0 possuem suas definições usuais.

As i-ésimas coordenadas ímpares dos elementos de Σ' possuem o mesmo conjunto de símbolos de Σ e representam símbolos na i-ésima fita de M. As i-ésimas coordenadas pares possuem ou o símbolo b (caracter branco), ou o marcador especial $\triangle$ que serve para determinar a posição da cabeça leitora na i-ésima fita de M. Se na i-ésima fita de M a cabeça leitora estiver sobre a célula j, escaneando o símbolo φ, então na j-ésima célula de M' o símbolo de coordenada $2i - 1$ será φ, e o da coordenada $2i$ será $\triangle$. Nas demais células de M' teremos b na coordenada $2i$.

Supondo que a cabeça leitora de M' se encontra no início da célula que contém a cabeça leitora mais à esquerda de M, M' simula um movimento de M da seguinte forma:

(i) M' inicia um movimento no estado $(q, \sigma_1, \divideontimes, ..., \sigma_k, \divideontimes)$, onde $q \in Q$, $\sigma_i \in \Sigma$, para $i = 1, ..., k$.

(ii) A cabeça de M' move - se para a direita até passar por todos os k marcadores $\triangle$. Sempre que M' passar por um marcador $\triangle$ posicionado em $2l$, M' salva o símbolo da coordenada $2l - 1$ da célula na $2l$-ésima coordenada de Q', e substitui o símbolo $\divideontimes$ da $2l + 1$-ésima coordenada de Q' por $-$. Quando passar pelo último marcador $\triangle$, Q' terá registrado todos os símbolos escaneados pelas cabeças leitoras de M. M' reconhecerá que passou por todos os marcadores quando trocar todos os seus símbolos $\divideontimes$ em Q' por $-$.

(iii) Nesse momento, conhecendo os valores de q e de cada σ_i, o estado de M' se transformará em $\overline{y}$, onde $\delta(q, \sigma_1, ..., \sigma_k) = y = (w_1, w_2, ...,$

$w_{2k+1})$ e $\overline{y} = (\overline{w_1}, w_2, ..., w_{2k+1})$, onde $(q, \sigma_1, ..., \sigma_k) \in Q \times \Sigma^k$. Agora o estado de M' contém a informação sobre o próximo movimento de cada uma das cabeças leitoras de M.

Se a primeira coordenada do estado de M' pertence a Q, a cabeça leitora de M' estará se movendo para a direita, com o objetivo de ler os símbolos. Quando a primeira coordenada pertence a $\overline{Q}$, a cabeça de M' está se movimentando para a esquerda para escrever os novos símbolos.

(iv) Uma vez reconhecido o próximo movimento de M, M' move sua cabeça para a esquerda até passar por todos os marcadores $\triangle$. Cada vez que a cabeça de M' passar por um marcador $\triangle$ posicionado na coordenada $2i$, M' altera o símbolo da coordenada $2i - 1$ substituindo - o pelo símbolo na posição $2i$ do vetor estado de M'. M' deve agora deslocar o marcador $\triangle$. Uma vez alcançada uma célula com marcadores, M' consulta seu estado para saber em que direção os marcadores devem ser movidos. Primeiramente M' vai para a direita e escreve os marcadores que precisam ir para a direita. Após alterar a posição de um marcador M' substitui, em seu vetor estado, o símbolo que indicava a direção por $\divideontimes$. O símbolo $\divideontimes$ serve para registrar quais marcadores já tiveram sua posição alterada. Deslocamos depois os marcadores que não devem mudar de posição (ou seja, não mudamos em nada o conteúdo da célula, mas alteramos o estado de M' substituindo os indicadores de direção por $\divideontimes$). Por fim alteramos os marcadores que serão deslocados para esquerda, este procedimento é análogo ao deslocamento para a direita.

(v) M' é construída de tal forma que sua cabeça leitora não se desloca no movimento em que altera um marcador de posição. Logo, pelo item

(iv), ao final desse processo a cabeça leitora de M' estará na mesma posição do marcador $\triangle$ mais a esquerda. M' reconhece o final do processo quando tiver k símbolos $\divideontimes$ em seu vetor estado.

Com o procedimento acima M' simulará um movimento de M. Após um movimento a cabeça de M' estará na mesma posição do marcador mais a esquerda de M'. Podemos portanto reiniciar o ciclo e executar os movimentos seguintes.

Como o espaço utilizado é menor ou igual ao tempo gasto, a quantidade máxima de células distintas utilizadas em uma mesma fita de M, é $T(n)$. Logo, na simulação de um movimento de M utilizamos $O(T(n))$ movimentos de M'. Consequentemente gastamos $O(T^2(n))$ movimentos com todo o processo. □

Na proposição anterior descrevemos todos os detalhes de um algoritmo para máquina de Turing, a partir de agora os algoritmos serão descritos em nível mais alto. Não manipularemos mais as cabeças das máquinas de Turing diretamente, mas sim as estruturas utilizadas no algoritmo.

Corolário 2.3. *Dada uma DTM M com k fitas, complexidade de espaço $S(n) \geq n$ e entrada x, $|x| = n$, então podemos construir uma DTM M' com uma fita e complexidade de espaço $S(n)$ tal que M aceita x se, e somente se, M' aceita x.*

Prova: Claramente na simulação feita acima, M' utilizará a mesma quantidade de células que a maior fita de M. □

Corolário 2.4. *Dada uma NDTM M com k fitas, complexidade de tempo $T(n)$ e entrada x, $|x| = n$, então podemos construir uma NDTM M' com uma única fita e complexidade de tempo $O(T^2(n))$ tal que M aceita x se, e somente se, M' aceita x.*

Prova: A máquina M' que simula M é idêntica a da proposição 2.2. A única diferença é que M é não determinística, portanto, construindo a máquina M' de forma que δ' simule o não determinismo de δ o corolário é válido. Note que o único passo não determinístico dessa simulação é a consulta a δ assim que M' acabou de armazenar em Q' os k símbolos sobre as cabeças leitoras de M, basta portanto construir M' de forma que δ' consiga escolher qualquer elemento possivelmente escolhido por δ. □

Corolário 2.5. *Dada uma NDTM M com k fitas, complexidade de espaço $S(n) \geq n$ e entrada x, $|x| = n$, então podemos construir uma NDTM M' com uma única fita e complexidade de espaço $S(n) \geq n$, tal que M aceita x se, e somente se, M' aceita x.*

Um dos objetivos principais desta dissertação é comparar o poder computacional de diferentes classes, o próximo resultado obedece a tal fim.

Nossa meta a seguir é decidir qual deve ser a relação entre duas funções $f(n)$ e $g(n)$, de tal forma que, $SPACE(f(n))$ é um subconjunto próprio de $SPACE(g(n))$.

Definição: Uma função é dita *espaço construtível* se existe uma DTM M tal que dada uma entrada de tamanho n, M irá marcar a $S(n)$-ésima célula de alguma de suas fitas sem usar mais que $S(n)$ células das demais fitas.

Teorema 2.6 (Hierarquia de espaço). *Sejam $S_1(n)$ e $S_2(n)$ duas funções* espaço construtíveis, *$S_1(n) \geq n$ e $S_2(n) \geq n$, tais que*

$$\liminf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0 \tag{2.4}$$

Então, existe uma linguagem L aceita por uma DTM com complexidade de espaço $S_2(n)$, mas não é aceita por nenhuma DTM com complexidade de espaço $S_1(n)$.

Prova: Inicialmente, observemos que, pelo corolário 2.3, se existe uma DTM com complexidade de espaço $S_1(n)$ que aceita uma linguagem L, então existe uma DTM com uma fita e complexidade de espaço $S_1(n)$ que aceita L. Logo, se provarmos que L não é aceita por uma DTM com uma fita e complexidade de espaço $S_1(n)$, então concluiremos que nenhuma DTM com complexidade de espaço $S_1(n)$ aceita L.

Seja M_0 uma DTM com 5 fitas e complexidade de espaço $S_2(n)$ (sendo uma de entrada e outra de saída), com entrada x de tamanho n, descrevemos a seguir a computação de M_0:

(i) M_0 marca $S_2(n)$ células em cada fita de trabalho.

(ii) Se x não é a codificação de alguma DTM com uma fita, então M_0 rejeita x.

(iii) Do contrário seja M a DTM representada por x. M_0 determina $|\Sigma|$ (tamanho do alfabeto de M) e $|Q|$ (número de estados), M_0 remarcará na sua segunda fita $S_1(n)$ blocos de $\lceil \log |\Sigma| \rceil$, cada bloco será separado por um marcador, usando portanto um total de $S_1(n)$ ($\lceil \log |\Sigma| \rceil$ + 1) células. Cada símbolo de M será codificado em binário e os blocos desta fita simulam as células de M.

(iv) Na terceira fita, M_0 delimita um bloco de $\lceil \log |Q| \rceil + \lceil S_1(n) \rceil + \lceil \log |\Sigma| \rceil$ $S_1(n)$ células, essa fita será usada como um contador para contarmos até $|Q|$ $S_1(n)$ $|\Sigma|^{S_1(n)}$, o número de configurações distintas de M.

(v) M_0 simulará M usando a fita 1 (fita de entrada) para consultar os movimentos de M, a fita 2 simulará a fita de M, a fita 3 contará os movimentos de M e a fita 4 será usada para guardar o estado de M.

(vi) Se M aceitar, M_0 rejeita. M_0 aceita se M rejeitar, se a simulação de M precisar de mais espaço que o reservado na fita 2 ou se o número de movimentos exceder o limite do contador (fita 3).

Suponha que exista M', uma DTM com 1 fita e complexidade de espaço $S_1(n)$, que aceite a mesma linguagem que M_0, e seja x a codificação de M'. Criaremos uma entrada w, $|w| = n$, completando x com 1's no final de tal forma que $S_2(n) \geq max\{S_1(n)(\lceil \log |Q| \rceil + 1), \lceil \log |Q| \rceil + \lceil S_1(n) \rceil + \lceil \log |\Sigma| \rceil S_1(n)\}$, de (2.4) sabemos que tal n existe.

Com as condições acima podemos realizar a simulação da computação de $M_0(w)$. Mas as saídas de M' e M_0 para a entrada w são diferentes, pois:

Se M' aceitar w, M_0 rejeita. M_0 aceita w se M' não aceitar, o que é um absurdo pois M' e M_0 aceitam a mesma linguagem. Logo, não existe tal máquina M'.□

Corolário 2.7. *Seja $S(n) \geq n$ uma função espaço construtível. A classe $SPACE(S(n))$ é um subconjunto próprio de $SPACE$ $(S(n) \log n)$.*

É importante observar que o corolário 2.3 foi construído para DTMs com complexidade de espaço maior ou igual a n. Isso ocorre porque conforme nossa definição, uma DTM M de uma única fita possui complexidade de espaço ao menos n. Na literatura pode - se encontrar máquinas de uma fita que possuem fita de entrada. Para tais máquinas os corolários 2.5, 2.7 e o teorema 2.6 são válidos sem precisar da restrição $S(n) \geq n$.

2.3 Problemas completos

Definiremos agora o conceito de *redução*, este é um conceito utilizado para comparar os problemas de decisão quanto a sua dificuldade. Intuitivamente,

se conseguirmos transformar "eficientemente" qualquer instância φ de um problema A numa instância γ de um problema B, de forma que a resposta de γ seja a mesma de φ, se soubermos resolver o problema B saberemos resolver A. O algoritmo para A é a composição da transformação de A para B com algoritmo para resolver B. Logo, como queremos relacionar a dificuldade de A com a de B, é importante que essa transformação seja eficiente. Resta ainda definir o conceito de transformação (e algoritmo) "eficiente", dizemos que um algoritmo é *eficiente* se podemos computá - lo com uma DTM em tempo polinomial.

Esse trabalho é predominantemente voltado para complexidade de espaço, é usual na literatura o uso de algoritmos $SPACE(O(\log n))$ para estudo de tais classes de complexidade, conforme [9, 11]. Sabemos de [11] que toda redução logarítmica em espaço é polinomial em tempo, mas não conhecemos nenhuma redução polinomial em tempo que não seja logarítmica em espaço. Como temos interesse em classes de complexidade superiores a classe P, o uso de transformação tempo polinomial será suficiente para nossos estudos. Para tipos mais restritos, ver [10].

Formalmente, um problema Π se *reduz (em tempo polinomial)* a um problema Π' (abreviadamente $\Pi \propto \Pi'$) se existe um algoritmo tempo polinomial R tal que para toda instância x de Π: x é verdadeira para Π se, e somente se, a instância $R(x)$ é verdadeira para Π'. Essa redução também pode ser chamada de *transformação polinomial* ou *redução de Karp*, originalmente definida por Karp em [7].

Proposição 2.8. *Sejam* Π_1, Π_2 *e* Π_3 *problemas de decisão. Se* $\Pi_1 \propto \Pi_2$ *e* $\Pi_2 \propto \Pi_3$, *então* $\Pi_1 \propto \Pi_3$

Prova: Ver [6], capítulo 2. □

Definição: Sejam Π_L e $\Pi_{L'}$ os problemas de decisão das linguagens L e L' respectivamente. Se $\Pi_L \propto \Pi_{L'}$, então dizemos que $L \propto L'$.

Junto com o conceito de redução e a proposição anterior, duas questões tornam - se naturais: Existe alguma linguagem L tal que, para toda linguagem L' pertencente a uma classe ξ, $L' \propto L$? Caso exista tal L, pode L pertencer a ξ?

A primeira pergunta nos fornece duas informações: A primeira vem da definição de $\propto$, caso tal L exista saberemos decidir qualquer linguagem em ξ se soubermos decidir L. Logo, em certo sentido, podemos interpretar que decidir L é ao menos tão difícil quanto decidir qualquer linguagem em ξ, por isso adotaremos o termo *ξ-difícil* para L. A segunda informação, obtida da proposição 2.8, é que se uma linguagem L^* é tal que $L \propto L^*$. Então, L^* também é ξ-difícil.

A segunda pergunta é interessante pois: Se decidir uma linguagem qualquer em ξ é tão difícil quanto decidir L, e além disso, L pertence a ξ, então para sabermos o quão difícil é decidir um problema qualquer de ξ, basta sabermos o quão difícil é decidir L. Nesse caso L ganha o termo de *ξ-completo.* Formalmente:

Seja ξ uma classe de complexidade. Um problema Π é *ξ-completo* se satisfaz as duas condições seguintes:

(i) $\Pi \in \xi$;

(ii) $\forall\, \Pi' \in \xi$, $\Pi' \propto \Pi$.

Este último item caracteriza um problema como *ξ-difícil.*

Os resultados a seguir foram extraídos de [11].

2.4 Alcançabilidade

Nessa seção citaremos um dos mais básicos problemas em grafos, o problema da alcançabilidade (denotaremos por ALCANÇABILIDADE):

Instância: Um digrafo $D(V, E)$ e vértices $x, y \in V(D)$.

Questão: Existe algum caminho de x a y?

Esse problema é resolvido por algoritmos básicos de busca, a mais notável delas é a busca em profundidade, algoritmo de complexidade $O(n^2)$, onde $|V| = n$.

Se a resposta para uma dada instância for sim, dizemos que x *alcança* y em D.

Veremos como modelar uma computação não determinística através de um digrafo. É devido a essa modelagem que o problema de alcançabilidade ganhará grande importância. Os detalhes e vantagens dessa modelagem serão explorados nas seções seguintes. Antes porém, veremos como funciona uma $NDTM$. Como exemplo descreveremos um algoritmo não determinístico para resolver o problema de alcançabilidade.

Antes de exibir o algoritmo para o problema de ALCANÇABILIDADE, informamos que neste trabalho será adotada a codificação binária para representar as estruturas utilizadas nos algoritmos. Por exemplo, os n vértices de um grafo serão codificados de 1 a n em binário e um natural n será representado por sua codificação binária.

Lema 2.9. *ALCANÇABILIDADE é computado por uma NDTM em espaço* $O(\log n)$.

Prova: Exibiremos uma máquina não determinística que decide se x alcança y, num digrafo D, com n vértices, com a complexidade especificada.

Tomemos uma máquina de Turing M com 3 fitas (além das de entrada e de saída). As 3 fitas de trabalho serão usadas da seguinte forma:

Na fita 1, a qual chamaremos de principal, escreveremos um vértice de D, inicialmente x, e na fita 2 "adivinhamos" um vértice w de D e verificamos se $(x, w) \in E(D)$, se $(x, w) \notin E(D)$, então M rejeita a entrada. Do contrário, verifique se $w = y$, caso a igualdade valha, M aceita a entrada. Caso contrário, apague x da primeira fita e escreva w, apague w da fita 2 e repita o processo.

Para ter certeza que o algoritmo pára, a terceira fita de trabalho funciona como contador, cada vez que mudarmos de vértice incrementamos o contador de uma unidade, se esse contador ultrapassar o limite de n unidades significa que a computação repetiu vértices, portanto M rejeita.

O algoritmo descrito funciona pois como "adivinhamos" os vértices certos, se houver caminho de x a y adivinharemos toda a sequência de vértices do caminho. Caso não exista caminho, o algoritmo irá parar ou por adivinhar um vértice que não é alcançável a partir do vértice corrente, ou por exceder o limite do contador. □

2.5 TIME vs. NSPACE

Esta é a primeira seção onde compararemos classes de diferentes naturezas. Nossa meta é determinar limites inferiores para a simulação de uma DTM limitada em tempo através de uma $NDTM$ limitada em espaço. Antes porém é necessário introduzir algumas definições:

Definição: O *digrafo de configuração*, $G(M, x)$, de uma máquina M com entrada x é um digrafo que possui como conjunto de vértices, V, todas as configurações possíveis de M, e cujo conjunto de arestas, E, contém para cada v e w pertencentes a V uma aresta orientada de v a w se $v \vdash_M w$.

Notemos que para uma máquina M com entrada e saída, $w_1 u_1$ será sempre $\triangleright x$, onde x é a entrada. A única alteração na fita 1 (de entrada) ao longo da computação é o posicionamento da cabeça leitora, usaremos um contador i para determinar seu posicionamento. Além disso para uma máquina que decide uma

linguagem, a fita de saída é usada uma única vez, ao final da computação para escrevermos SIM ou NÃO, logo esta fita é indiferente na contagem do número total de configurações. Com isso, a configuração de M pode ser representada pela $(2k-2)$-upla ordenada, $(q, i, u_2, w_2, ..., u_{k-1}, w_{k-1})$.

Seja $M \in SPACE(f(n))$. Com a representação da configuração de M por esta $(2k-2)$-upla ordenada, obtemos o seguinte limite superior para o número de configurações distintas de M: Temos $|Q|$ valores distintos para o estado q; temos $n+1$ valores distintos para o contador i, onde $|x| = n$ é o tamanho da entrada; temos $|\Sigma|^{f(n)}$ possibilidades para cada uma das $(2k-2)$-uplas restantes, dado que Σ é o alfabeto de M e $f(n)$ é o comprimento máximo de uma fita. Como $|Q|$ e $|\Sigma|$ são constantes, temos não mais que $nc_1^{f(n)}$ configurações distintas para M, para alguma constante c_1 tal que $nc_1^{f(n)} \leq c_1^{\log n + f(n)}$.

Proposição 2.10. $NSPACE(f(n)) \subseteq TIME(c^{\log n + f(n)})$

Prova: Seja M uma $NDTM$ com complexidade de espaço $f(n)$. Simularemos M com uma DTM M' que use tempo $c^{\log n + f(n)}$.

Claramente, seja L uma linguagem, decidir se $x \in L$ é equivalente a decidir se existe um caminho da configuração inicial a uma configuração de parada com SIM em $G(M, x)$, onde $|x| = n$ e $G(M, x)$ é um digrafo com no máximo $c_1^{\log n + f(n)}$ vértices. Para esse problema conhecemos o algoritmo de busca em profundidade que resolve o problema com um número de iterações quadrático, no tamanho da entrada.

Precisamos portanto de $c_2 c_1^{2(\log n + f(n))}$ passos, para alguma constante c_2, para decidir se $x \in L$. Tomando $c = c_2 c_1^2$, $c^{(\log n + f(n))}$ é um limite superior para o número de passos da simulação e suficiente para demonstrar a proposição.

Nos resta ainda explicitar como utilizar a busca em profundidade nesse contexto. Podemos utilizá - la da forma usual escrevendo a matriz de adjacência de $G(M, x)$ em alguma fita de M', ou então, não construímos a matriz de adjacência e sempre que quisermos decidir se (C, C') é aresta de $G(M, x)$ olhamos os símbolos

sobre cada uma das cabeças leitoras na configuração C e consultamos δ (função de transição de M) para verificar se $C \vdash_M C'$. Para descobrir o símbolo sobre a cabeça leitora na fita de entrada utilizamos uma nova fita como contador para descobrirmos o símbolo na i-ésima posição. □

A técnica utilizada na demonstração anterior é chamada de *método da alcançabilidade*, além do resultado obtido, essa técnica voltará a ser usada em resultados ainda mais fortes.

Como consequência imediata da proposição anterior temos NSPACE($f(n)$) $\subseteq$ SPACE($c^{\log n + f(n)}$) o que nos faz pensar: "Seria esse o melhor limite superior, em espaço, para simulação de máquinas não determinísticas, utilizando máquinas determinísticas? " Veremos agora que essa simulação precisa ser apenas quadrática. Para tal fim, descreveremos uma *DTM* que decide ALCANÇABILIDADE de forma mais eficiente em espaço. O resultado esperado será consequência do uso desta máquina, e do método da alcançabilidade.

2.6 $PSPACE = NPSPACE$

Nesta seção veremos que para máquinas de Turing com limite polinomial em espaço, determinismo e não determinismo são equivalentes.

Teorema 2.11. *ALCANÇABILIDADE* $\in$ $SPACE(O(\log^2 |V|))$

Prova: Inicialmente definiremos um novo termo: Sejam G, x, y e i , respectivamente, um digrafo, dois vértices deste digrafo e um inteiro não negativo. **CAMINHO**(x,y,i) será verdadeiro se existir um caminho de x a y, em G, com comprimento no máximo 2^i e falso caso contrário.

Como um caminho em um digrafo tem comprimento no máximo n, onde $|V| = n$, notemos que o problema de alcançabilidade em G é equivalente ao problema de decidir CAMINHO(x,y,$\lceil \log n \rceil$).

Para resolvermos CAMINHO(x,y,$\lceil \log n \rceil$), descreveremos uma máquina de Turing M com 3 fitas (incluindo as fitas de entrada e saída). Na fita de entrada, M contém a matriz de adjacência de G. Na fita de trabalho escreveremos, codificado em binário, alguns ternos da forma (x,y,i). Inicializamos a fita de trabalho com o terno (x,y,$\lceil \log n \rceil$).

Necessitamos agora de um procedimento para decidir CAMINHO(x,y,i). Se $i = 0$, ou seja, y está a uma distância de comprimento máximo 1 de x, consultamos a matriz de adjacências e verificamos se os vértices são iguais ou adjacentes. Se $i > 0$, adotaremos um algoritmo recursivo, para decidir CAMINHO(x,y,i), ver algoritmo 1.

Algoritmo 1 CAMINHO(x,y,i)

Procedure CAMINHO(x,y,i)
alcance $\leftarrow$ FALSO;
for all $z \in V(G)$ **do**
  if (CAMINHO(x,z,$i-1$) $\wedge$ CAMINHO(z,y,$i-1$)) **then**
    alcance $\leftarrow$ VERDADEIRO;
return *alcance*;

A idéia do algoritmo é simples: Dois vértices x e $y \in V(G)$, tais que x alcança y, e se encontram a uma distância de 2^i, possuem um ponto médio z que dista de 2^{i-1} de cada um deles. Descreveremos agora uma implementação desse algoritmo eficiente em espaço.

Sejam x,$y \in V(G)$, tais que (x,y,i) é o último terno da direita na fita de trabalho. Para decidir CAMINHO(x,y,i) geramos os vértices z um após o outro, apagando e reutilizando espaço. Cada vez que um novo z for gerado adicionaremos o terno (x,z,$i-1$) e trataremos recursivamente o problema CAMINHO(x,z,$i-1$). Se este for falso, então apagamos o terno da fita de trabalho e passamos ao próximo z. Senão apagamos esse terno e passaremos ao terno (z,y,$i-1$) (usamos o próximo terno à esquerda na fita para consultar y). Se obtivermos falso

para CAMINHO(z,y,$i-1$) apagamos o novo terno e tentamos um novo z. Senão comparamos (z,y,$i-1$) com (x,y,i), o terno à esquerda, e concluímos que é a segunda chamada recursiva, logo, CAMINHO(x,y,i) é verdadeiro.

O algoritmo é iniciado escrevendo (x,y,$\lceil \log n \rceil$) na fita de trabalho, depois disso executamos o procedimento descrito no parágrafo anterior.

Claramente o algoritmo acima está correto. Como cada novo terno escrito na fita tem o valor de i uma unidade menor que o terno anterior, teremos ao longo do processo no máximo $\lceil \log n \rceil + 1$ ternos. Cada um deles com comprimento máximo $3 \log n$, portanto o espaço gasto é $O(\log^2 n)$ o que conclui a demonstração. □

Corolário 2.12. *$NSPACE(f(n)) \subseteq SPACE(O(f^2(n)))$, para $f(n) \geq \log n$.*

Prova: Para simularmos M, uma *NDTM* que opera em espaço $f(n)$, utilizamos uma *DTM* M' e rodamos o algoritmo da demonstração anterior em $G(M,x)$, $|x| = n$. No novo algoritmo, consultaremos a entrada unicamente para testar adjacência entre vértices C e C' quando $i = 0$. Efetuamos essa operação verificando, através da função de transição da máquina M, se C alcança C'. Como $G(M,x)$ possui $c^{f(n)}$ vértices, para alguma constante c, precisaremos de espaço $O(f^2(n))$. □

O resultado a seguir é um dos mais importantes do capítulo. Veremos que se uma computação utiliza espaço polinomial, não importa se ela foi realizada por uma máquina determinística ou não determinística. Tal resultado é consequência imediata do corolário anterior.

Corolário 2.13. *$PSPACE = NPSPACE$.*

2.7 $NSPACE(O(f(n))) = coNSPACE(O(f(n)))$

Estudaremos agora o problema de contar quantos vértices alcançamos em um digrafo G a partir de um vértice x, este problema é uma extensão da ALCANÇABILIDADE e também pode ser resolvido em espaço $O(\log n)$ utilizando uma NDTM.

Além da baixa complexidade de espaço, notemos que determinar o número de vértices não alcançáveis em G, a partir de x, é equivalente a subtrair o total de vértices da quantidade de vértices alcançáveis. Essa facilidade em que calculamos o complemento é a grande importância deste problema de contagem.

Teorema 2.14. *Dado um digrafo G e um vértice x, o número de vértices alcançáveis em G a partir de x pode ser calculado em espaço $O(\log n)$.*

Prova: Esse problema é resolvido através do algoritmo 2. Antes porém definiremos o conjunto $S(k)$ como conjunto dos vértices que distam no máximo k unidades de x. Claramente a solução do problema é o valor de $|S(n-1)|$.

No algoritmo usaremos o termo booleano $G(v,u)$ que é verdadeiro se $v = u$, ou $(v,u) \in E(G)$, e falso caso contrário.

Nosso algoritmo consiste em 4 laços:

No primeiro laço do algoritmo executamos um procedimento recursivo para calcular o valor de $|S(k)|$, para $k = 1, ..., n-1$. Porém não sabemos como computar o valor de $|S(k)|$ a partir de $|S(k-1)|$, essa é a função do 2º laço.

No 2º laço criamos um contador l que conta os membros de $S(k)$ para todo k. Temos porém um novo problema, não sabemos decidir se $u \in S(k)$. Resolveremos esse problema no 3º laço.

No 3º laço, para cada $v \in S(k-1)$, verificamos se $G(v,u)$. Caso exista algum v que atenda $G(v,u)$, concluímos que $u \in S(k)$. A variável *resposta* funciona como controladora desse fato. Ainda nos resta um último problema nesse 3º laço, não sabemos decidir se $v \in S(k-1)$. Para isso usaremos um laço final.

No 4º laço decidimos se $v \in S(k-1)$. Este procedimento é a parte não determinística do algoritmo. Esta etapa é parecida com o algoritmo de alcançabilidade não determinístico descrito no lema 2.9. adivinharemos uma sequência de $k-1$ vértices, onde os vértices da sequência constituem um caminho de x a v. Notemos que os vértices da sequência não precisam ser distintos e um vértice pode ser o sucessor dele mesmo nesta sequência.

Algoritmo 2

$|S(0)| \leftarrow 1$; $\{1^o\ laço\}$

for $k = 1,...,n$ **do**

compute $|S(k)|$ usando $|S(k-1)|$;

$l \leftarrow 0$; $\{2^o\ laço\}$

for all $u \in V$ **do**

if $u \in S(k)$ **then**

$l \leftarrow l + 1$;

resposta $\leftarrow$ falso; $\{3^o\ laço\}$

for all $v \in V$ **do**

if $(\ v \in S(k-1) \wedge G(v,u))$ **then**

resposta $\leftarrow$ verdade;

Return *resposta*;

$w_0 \leftarrow x$; *pertinencia* $\leftarrow$ falso; $\{4^o\ laço\}$

for $p = 1,...,\ k-1$ **do**

"adivinhe"um nó w_p;

if $G(w_{p-1}, w_p) =$ falso **then**

return *pertinencia*;

if $w_{k-1} = v$ **then**

pertinencia $\leftarrow$ verdadeiro;

return *pertinencia*;

Concluímos assim a descrição do algoritmo. Notemos que tudo o que precisamos guardar são os valores de k, $|S(k-1)|$, l, u, v, p, w_p, w_{p-1}. Claramente, com codificação binária, uma máquina com 8 fitas (sem contar a de entrada e a de saída) usaria espaço $O(\log n)$ para realizar este cômputo.

Provaremos agora que o algoritmo funciona, para tal fim mostraremos que $|S(k)|$ é calculado corretamente para todo k. Faremos uma indução onde precisamos provar que, para cada k, o contador l só é incrementado se, e somente se, o vértice u corrente $\in S(k)$. Para $k = 0$ temos $|S(k)| = 1$, logo a base está correta. Para um k qualquer, todos os vértices de $S(k-1)$ são verificados. Se l foi incrementado, então $G(v,u)$ é verdadeiro, para algum $v \in S(k-1)$, portanto $u \in S(k)$. E, se $u \in S(k)$, a variável *resposta* guarda a informação se $G(v,u)$ é verdadeira para algum $v \in S(k-1)$. Como é esta variável que decide o incremento de l nossa prova esta concluída. □

Corolário 2.15. *$NSPACE(O(f(n))) = coNSPACE(O(f(n)))$, para $f \geq \log n$.*

Prova: Seja $L \in NSPACE(O(f(n)))$, uma linguagem decidida por M, uma $NDTM$ que utiliza espaço $O(f(n))$.

Iremos construir uma máquina não determinística $\overline{M}$ com espaço $O(f(n))$ que decida $\overline{L}$.

Para uma dada entrada x, $\overline{M}$ irá rodar o algoritmo anterior em $G(M,x)$, usando a função de transição de M para decidir se uma configuração é adjacente a outra. Se $\overline{M}$ verificar que alguma configuração de aceitação $\eta \in S(k)$, para qualquer k, $\overline{M}$ pára e rejeita, mas se ao final do cálculo de $S(n-1)$ nenhuma configuração de aceitação foi encontrada, $\overline{M}$ aceita. Lembrando que $|V(G(M,x))| = c^{\log n + f(n)}$ e o algoritmo anterior utiliza espaço $O(\log n)$ nosso resultado está demonstrado. □

Capítulo 3

Classe $PSPACE$ e jogos combinatórios

Exibiremos neste capítulo uma visão geral sobre a classe $PSPACE$. Mostraremos como resolver problemas através de algoritmos polinomiais em espaço e definiremos os problemas completos básicos para essa classe. Exibiremos, através dos problemas utilizados, o por que da importância desta classe para o estudo de complexidade de jogos combinatórios e por fim exibiremos dois jogos combinatórios $PSPACE$-completos.

3.1 Jogos combinatórios

Um dos objetivos deste texto é a analise de complexidade de *jogos combinatórios*. Veremos a seguir o quão interessantes são esses problemas para o estudo de classes superiores a NP. Segue abaixo a definição de um problema deste tipo.

Definição: Um *jogo combinatório* é um jogo com 2 jogadores, I e II, que atende as seguintes condições:

- Os jogadores jogam alternadamente e cada jogador executa um lance por vez.

- A qualquer momento o conjunto de jogadas legais está bem definido e é do conhecimento de todos, cabendo ao jogador a escolha de sua jogada.
- Cada jogador tem total informação sobre os próximos lances que podem ser executados pelo oponente.
- Há condições bem definidas e previamente conhecidas que determinam quando o jogo acabou, assim como seu resultado final.

Convencionaremos que o jogador I sempre será o primeiro a jogar.

Os problemas relacionados a jogos combinatórios serão da seguinte forma: Dados um jogo G e uma posição π. Existe uma estratégia vencedora para o jogador I no jogo G iniciado na posição π?

A partir deste ponto quando tivermos que falar sobre a complexidade do problema de decisão associado ao jogo G, utilizaremos o termo complexidade do jogo.

Para provar que um problema é $PSPACE$, precisamos exibir um algoritmo para solucionar o problema que utilize espaço polinomial. Para os problemas de jogos combinatórios, esse algoritmo decide se a posição dada é vitoriosa para I, ou seja, independente das respostas de II o jogador I vence a partida, lembrando que I é quem faz o lance inicial. Os algoritmos $PSPACE$ para jogos utilizam o conceito de *árvore do jogo.*

Definição: Seja G um jogo, ϖ_0 a posição inicial (posição fornecida na instância do problema) de G e (ϖ, j) um par ordenado, onde ϖ representa uma posição de G e j o jogador que tem a vez em ϖ. A *árvore do jogo* de G, denotada por T_G, é uma árvore enraizada em $(\varpi_0$, I) com vértices (ϖ, j). Cada (ϖ, j) possui como descendentes todos os possíveis pares (ϖ', j'), onde $j \neq j'$ e ϖ' é obtida da posição ϖ com um único lance de j, considerando - se que (ϖ, j) foi obtida através de uma sequência de jogadas representada pelo caminho de $(\varpi_0$, I) a (ϖ, j) em T_G.

A árvore do jogo contém todas as possíveis sequências de jogadas de G representadas pelos caminhos maximais da árvore. Note que um mesmo elemento (ϖ, j)

pode aparecer mais de uma vez em T_G, isto significa que (ϖ, j) pode ser obtida por diferentes sequências de jogadas em G, a partir da posição ϖ_0.

Cada vértice v de T_G terá associado um valor booleano. O vértice v será verdadeiro se o jogador I vence o jogo em v independente do jogador II, e falso caso contrário. Notemos que as folhas da árvore representam posições terminais de G, para estas posições os valores booleanos são definidos pela regra de G. Um vértice interno v, onde I tenha a vez, será verdadeiro se possuir algum descendente w em T_G tal que w é verdadeiro. Um vértice interno v, onde II tem a vez, é verdadeiro se todos os seus descendentes em T_G são verdadeiros. Esta relação nos fornece um procedimento recursivo para decisão de G utilizando T_G. Chamaremos a função que atribui um valor booleano a cada vértice de μ.

A resposta do problema de decisão associado a um jogo combinatório G corresponde ao valor booleano da raiz de T_G. Para determinar este valor faremos um algoritmo de backtracking em T_G. Este algoritmo recursivo é descrito a seguir.

Inicializamos o algoritmo introduzindo a raiz de T_G em uma pilha P e decidimos o seu valor. Para decidir $\mu(v)$ de um vértice v qualquer devemos, inicialmente, verificar se este vértice é uma folha. Em caso positivo, o vértice é terminal e decidimos seu valor utilizando as regras do jogo. Se v não for folha, então inserimos em P um vértice w, descendente de v em T_G. Decidimos o valor de $\mu(w)$, removemos w de P e informamos o valor de $\mu(w)$ a v. Dependendo de $\mu(w)$ e de qual jogador tem a vez em v, podemos decidir $\mu(v)$ (se $\mu(w) = 1$ e I tem a vez em v, sabemos que $\mu(v) = 1$; ou se $\mu(w) = 0$ e II tem a vez em v, sabemos que $\mu(v) = 0$). Se $\mu(w)$ não for suficiente para determinar $\mu(v)$, então inserimos em P o vértice w', outro descendente de v em T_G. Repetimos a operação feita para w, e procedemos desta forma para todos os descendentes de v em T_G. Embora o algoritmo possa decidir $\mu(v)$ sem decidir μ para todos os descendentes de v, sempre saberemos decidir $\mu(v)$ após decidirmos μ de todos os descendentes de v (se I tem a vez em v e o valor de μ dos seus descendentes não decidiu $\mu(v)$, então v não possui um

descendente y, tal que $\mu(y) = 1$. Logo, $\mu(v) = 0$. Para vértices v onde II tem a vez a argumentação é análoga).

O algoritmo acima resolve o jogo G pois verifica todas as possíveis sequências de jogadas em G. Note que a pilha P contém o caminho correntemente verificado em T_G, logo um limite superior para o número de vértices em P é igual a altura h da árvore. Este algoritmo só precisa armazenar o conteúdo de P, logo sua complexidade de espaço é $O(h)$.

É importante observar que na análise de complexidade feita no parágrafo anterior, supomos que o valor μ de uma folha de T_G é previamente fornecido. Embora pela definição de jogo combinatório a regra de término é clara, precisamos sempre analisar a complexidade de obtenção de μ das folhas de T_G, antes de determinar a complexidade de espaço.

Os jogos $PSPACE$ exibidos neste trabalho possuem limite polinomial (no tamanho da entrada) para o número de jogadas. Logo, sua árvore do jogo possui altura polinomial.

Para provar que um jogo é $PSPACE$-difícil, usaremos reduções dos problemas básicos de $PSPACE$ apresentados a seguir.

3.2 Problemas básicos

Assim como no caso dos problemas NP-completos teremos um problema $PSPACE$-completo base, o qual usaremos para fazer reduções para os demais problemas $PSPACE$-difíceis. O nosso problema base é o seguinte:

Instância: n variáveis booleanas, n quantificadores Q_i ($\forall$ ou $\exists$) e uma fórmula booleana ϕ

Pergunta: Seja Q_i o quantificador da variável x_i, ϕ é verdadeira?

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi?$$

Onde Q_i é um quantificador ($\forall$ ou $\exists$).

Esse problema é o QBF (quantified boolean formula). Não exibiremos a demonstração de $PSPACE$-completude deste problema, mas ele foi originalmente definido e provado por Meyer e Stockmeyer em [10].

Outro problema $PSPACE$-completo que será útil na análise de jogos combinatórios é o QSAT:

Instância: n variáveis booleanas, m cláusulas e ϕ uma fórmula na forma normal conjuntiva.

Pergunta: Existe uma atribuição para a variável x_1, tal que para toda atribuição da variável x_2 e assim sucessivamente até x_n, tal que a fórmula ϕ é verdadeira?

$$\exists x_1 \forall x_2 \ldots Q_n x_n \phi ?$$

Onde,

$$Q_i = \begin{cases} \exists & ,i \text{ é ímpar.} \\ \forall & ,i \text{ é par.} \end{cases}$$

Assim como o SAT algumas variações do QSAT também são $PSPACE$-completas, dentre elas o 3QSAT onde cada cláusula possui 3 literais:

Um exemplo de pergunta para o 3QSAT com 3 variáveis e 3 cláusulas é:

$$\text{“ } \exists x_1 \forall x_2 \exists x_3 (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_1 \vee x_3) \text{ ? ”}$$

Observe que a ordem das variáveis quantificadas altera o resultado final do QSAT. Como exemplo, seja o QSAT com 2 variáveis e 2 cláusulas. Seja $\Phi = (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2})$, as perguntas “$\exists x_1 \; \forall x_2 \; \Phi$?” e “$\forall x_2 \; \exists x_1 \; \Phi$?” possuem diferentes respostas.

A fórmula Φ só é verdadeira para as seguintes atribuições: $x_1 = 1$ e $x_2 = 1$; ou $x_1 = 0$ e $x_2 = 0$. Portanto, para todo x_2 existe x_1 que torne Φ verdadeira. No entanto, **não** existe x_1 tal que, para todo x_2 a fórmula Φ é verdadeira.

Conhecemos um bom número de problemas que são $PSPACE$-completos, em particular um grande número de jogos combinatórios são $PSPACE$-completos e é para esse grupo de problemas que voltaremos nossas atenções.

O QSAT é um problema de grande interesse em nossos estudos, uma vez que a disposição dos seus quantificadores refletem a disputa de um jogo combinatório. Quando perguntamos se existe uma estratégia vencedora para o jogador I, o que desejamos saber é:

Existe uma jogada para I tal que, **para toda** jogada de II, **existe** uma jogada para I tal que, ... o jogador I vence a partida?

Exibiremos a seguir o primeiro jogo $PSPACE$-completo deste trabalho.

3.3 GEOGRAPHY

O GEOGRAPHY é um jogo sobre um conjunto de cidades de grande importância para este tema. Pois além de ser um exemplo de jogo $PSPACE$-completo, sua estrutura simples será útil para análise de outros problemas como veremos no próximo capítulo. A referência para esta seção é [14], que também é a referência original sobre o problema.

Dada uma cidade inicial, o jogador I deverá citar uma cidade cuja primeira letra é a última letra da cidade inicial. A cada jogada, o jogador que tiver a vez deve citar uma cidade cuja a primeira letra é a última letra da cidade anterior, sendo que nenhuma cidade pode ser citada mais de uma vez e a cidade inicial não pode ser citada. O jogo acaba quando um jogador x não puder citar mais nenhuma cidade. O jogador x é considerado o perdedor.

Temos portanto o seguinte problema de decisão para o GEOGRAPHY:

Instância : Um conjunto de cidades e uma cidade inicial.

Questão : Existe uma estratégia vencedora para I?

Construiremos o digrafo $D(V, E, s)$ onde V é o conjunto das cidades, $E = \{(v,w) \mid$ a última letra de v é a primeira de $w\}$ e s é a cidade inicial.

Com esse digrafo, um jogo equivalente ao GEOGRAPHY seria: O jogador I escolhe uma cidade de V adjacente a inicial, e a cada jogada o jogador que tiver a vez escolhe uma cidade adjacente a última cidade escolhida e que ainda não tenha sido visitada. O jogo termina quando um jogador x escolhe uma cidade que é um sumidouro em D ou uma cidade onde todas as suas adjacentes em $V \setminus \{s\}$ já foram escolhidas, nesse caso o jogador x é decretado vencedor.

Teremos portanto um novo problema de decisão equivalente ao GEOGRAPHY:

Instância : Um digrafo $D(V, E, s)$

Questão : Existe uma estratégia vencedora para I?

Quando falarmos do GEOGRAPHY estaremos nos referindo a esse segundo problema. Essa modelagem por digrafos nos permite usa - lo com mais facilidade que a versão original.

Teorema 3.1. *GEOGRAPHY é PSPACE-completo.*

A prova de $PSPACE$-completude do GEOGRAPHY possui 2 etapas.

3.3.1 GEOGRAPHY é *PSPACE*

Provaremos que existe um algoritmo para solucionar GEOGRAPHY que utiliza espaço polinomial no número de vértices do jogo.

Dois pontos importantes devem ser notados:

(i) O jogo dura no máximo $|V|$ jogadas.

(ii) Dado o *estado* do jogo (um caminho do vértice inicial até o último escolhido e a indicação de quem deverá executar o próximo lance) conseguimos determinar em espaço polinomial todos os possíveis próximos lances e estados do jogo, ou se não houver nenhum lance que possa ser executado sabemos quem foi o vencedor.

Para estabelecer item (ii), basta olharmos os vértices adjacentes ao último jogado que ainda não foram jogados, ou se todos tiverem sido jogados informar que o jogador que tem a vez perde naquela posição.

Com os ítens anteriores, observamos que podemos resolver o GEOGRAPHY percorrendo sua a árvore do jogo, conforme descrito na seção 3.1. Tal algoritmo usará espaço $O(|V|)$, correspondente a altura da árvore do jogo de GEOGRAPHY. É importante notar que não podemos armazenar essa árvore pois ela possui tamanho exponencial.

3.3.2 QSAT $\propto$ GEOGRAPHY

Provaremos agora que o GEOGRAPHY é $PSPACE$-difícil. Para tal fim, provaremos que QSAT $\propto$ GEOGRAPHY.

Utilizaremos uma variação do QSAT onde o número de variáveis, n, é par. Essa variação também é $PSPACE$-completa, pois dada uma instância de QSAT, basta adicionarmos a cláusula $(x_{n+1} \vee \neg x_{n+1})$ à fórmula ϕ que tenha n ímpar. Dessa forma, vemos que o QSAT se reduz ao QSAT com n par.

Para reduzir o QSAT ao GEOGRAPHY faremos a seguinte transformação:

Para cada variável do QSAT criaremos um *diamante* de escolha conforme figura 3.1. As laterais de um diamante serão denominadas *vértices literais.*

Dada uma instância do QSAT criaremos um digrafo $G(V, E, s)$, onde o sumidouro do diamante de x_i será ligado a fonte do diamante de x_{i+1}, para todo i de 1 a $n-1$. O sumidouro de x_n será ligado a m *vértices cláusula*, onde m é o número de cláusulas em ϕ. Cada um desses vértices corresponderá a uma cláusula. De cada vértice cláusula partem arcos a literais que estão nela contidos. Mas antes de

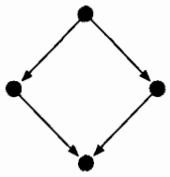


Figura 3.1: diamante

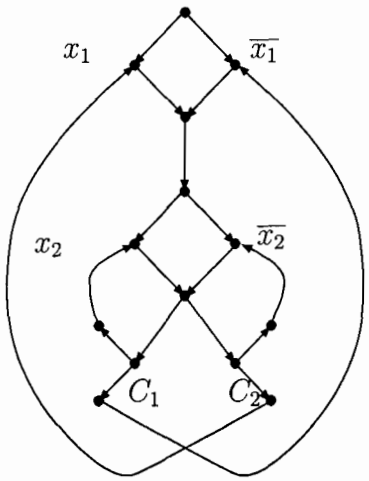


Figura 3.2: Instância de GEOGRAPHY correspondente a instância de QSAT $\exists x_1$ $\forall x_2$ $(\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2})$

alcançar o literal, cada um desses arcos vai para um vértice com grau de entrada 1 e grau de saída 1, esses vértices serão chamados de triviais conforme o exemplo da figura 3.2.

E finalmente:

A sentença quantificada é uma instância sim para o QSAT se, e somente se, I possui uma estratégia vencedora para a instância de GEOGRAPHY.

Prova: ($\Rightarrow$) I percorrerá os diamantes de escolha das variáveis existenciais e II das variáveis universais. A estratégia para I consiste em jogar sobre literais

verdadeiros e interpretar as jogadas de II como se este estivesse escolhendo literais verdadeiros. A cada jogada I escolhe o valor da variável de forma a tornar ϕ verdadeira, de acordo com as atribuições de valores já feitas. Ao chegar no final do diamante x_n estará na vez de II, este escolhe uma cláusula e se desloca para o vértice correspondente. Para qualquer cláusula que este jogador escolha, como QSAT é verdadeiro, sabemos que I sempre pode escolher vértices que tornem ϕ verdadeira para qualquer combinação de respostas de II. Logo, sempre haverá ao menos um literal verdadeiro por cláusula, I jogará no vértice trivial que se encontra no caminho entre o vértice cláusula e algum vértice literal verdadeiro daquela cláusula. II ficará sem jogadas.

($\Leftarrow$) Como I ganha independente das respostas de II, existe uma escolha para I no diamante x_1 tal que, para qualquer jogada de II no diamante x_2, existe uma escolha para I no diamante x_3 e assim sucessivamente, de forma que I deixa II sem jogadas. Tomemos os vértices literais utilizados por II como literais verdadeiros, os literais que foram utilizados por I para vencer também devem ser considerados verdadeiros. Como ao final do jogo II ficou sem jogadas, então para todo vértice cláusula havia ao menos um vértice literal verdadeiro. Dessa forma concluímos que a instância do QSAT é verdadeira. □

Note que dada uma instância de QSAT construímos, trivialmente, a instância de GEOGRAPHY. Basta para cada cláusula e variável gerar os vértices correspondentes e determinar as arestas do grafo. Esta transformação é claramente polinomial.

3.4 HEX

Mostraremos agora um resultado de $PSPACE$-completude para um jogo mais conhecido, o HEX:

Instância : Grafo $G(V, E)$ e vértices $\{s, t\} \in V$.

Questão : Seja o seguinte jogo: Posições são partições de V em 3 conjuntos $<V_1, V_2, V_3>$ tal que $\{s, t\} \subset V_3$; $\bigcup_{i=1}^{3} V_i = V$ e $V_i \cap V_j = \varnothing$, $\forall\ i \neq j$. A posição inicial é $<\varnothing, \varnothing, V>$ e as posições finais têm a seguinte forma $<V_1, V_2, \{s, t\}>$. Em uma posição $<V_1, V_2, V_3>$, se for sua vez, I escolhe um vértice $u \in V_3 \setminus \{s, t\}$ e troca a posição para $<V_1 \cup \{u\}, V_2, V_3 \setminus \{u\}>$. Analogamente II fará uma escolha $w \in V_3 \setminus \{s, t\}$ e alterará a posição parcial do jogo para $<V_1, V_2 \cup \{w\}, V_3 \setminus \{w\}>$. I ganha o jogo se o subgrafo induzido por $V_1 \cup \{s, t\}$ contém um caminho de s a t. Existe uma estratégia vencedora para I?

Este foi um dos primeiros jogos combinatórios $PSPACE$-completo conhecido. Sua demonstração se deve a Even e Tarjan [4], esta será também a fonte básica desta seção.

3.4.1 HEX é *PSPACE*

O algoritmo espaço polinomial para solucionar o HEX é parecido com o algoritmo para o GEOGRAPHY. Ele consiste em fazer uma busca em todos os caminhos da árvore do jogo de HEX.

Como HEX possui no máximo $|V|$ jogadas precisamos de espaço $O(|V|)$ para computar o algoritmo baseado na árvore do jogo, e dada uma posição terminal sabemos decidir em espaço polinomial se ela é verdadeira ou falsa, então o algoritmo é $PSPACE$.

3.4.2 QBF $\propto$ HEX

Redução

Sem perda de generalidade utilizaremos nessa redução uma versão do QBF na qual $Q_1 = Q_n = \exists$ onde x_1 e x_n são, respectivamente, a primeira e a última variável, ϕ está na forma normal conjuntiva e possui k cláusulas.

Lembre que o QBF, ao contrário do QSAT, não possui atribuição fixa para os quantificadores Q_i de suas variáveis (em particular esta versão possui fixos $Q_1 = Q_n = \exists$), portanto trataremos uma instância qualquer do QBF como um bloco (onde este bloco pode ter qualquer tamanho entre 1 e n) de variáveis consecutivas quantificadas com $\exists$, seguidas de um bloco de variáveis consecutivas quantificadas com $\forall$ e assim sucessivamente. Lembrando apenas que o primeiro e o último bloco são quantificados com $\exists$.

Dada uma instância do QBF o grafo que construiremos para o HEX possui 3 pedaços distintos: Uma árvore estritamente binária (ou seja, uma árvore binária, onde todos os seus vértices possuem 0 ou 2 descendentes) que representa uma conjunção, uma árvore de três níveis que representa as disjunções e um cubo onde decidiremos os valores das variáveis. Nos referiremos a uma árvore estritamente binária como árvore B.

Adotaremos a terminologia que I ganhou o jogo, se o jogador I conseguiu formar um caminho de s a t, com os vértices por ele escolhidos. E utilizaremos que II ganhou o jogo, se o jogador II conseguiu impedir todas as possibilidades do jogador I formar um caminho de s a t, também utilizaremos os termos bloquear e cortar para tal impedimento.

Descreveremos a seguir como seria o jogo em cada estrutura da construção geral isoladamente. Depois explicitaremos a construção geral e usaremos as estratégias desenvolvidas para as estruturas isoladamente.

Considere uma árvore B na figura 3.3 (observe que quando nos referimos a árvore, estamos desconsiderando os vértices s e t, e as arestas determinadas por estes vértices). O jogador I só vencerá o HEX jogado nesta estrutura se todos as folhas da árvore estiverem ligados a t. Para vencer o jogador I deve adotar a seguinte estratégia:

Inicialmente jogue na raiz. Em jogadas posteriores, marque um vértice filho do vértice que jogou na última vez em que teve a vez, este filho deve ser a raiz de uma árvore que não possua vértices utilizados por II.

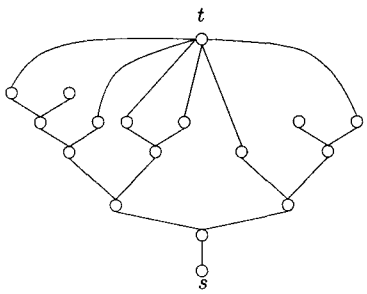


Figura 3.3: Árvore binária(conjunção)

Se alguma folha da árvore B não for adjacente a t, II ganha com a seguinte estratégia:

Se algum vértice que ainda não foi utilizado corta todos os caminhos remanescentes de s a t jogue - o. Senão, jogue em um dos dois filhos do último vértice escolhido por I. Descubra qual deles é raiz de uma árvore onde alguma folha não está conectada a t, jogue no outro filho.

Vamos agora provar por indução que as estratégias de ambos os jogadores funcionam:

- Para o jogador I:

  Hipótese de Indução: Seja T uma árvore B com n vértices e com todas as folhas conectadas a t. Então, I ganha o jogo com a estratégia fornecida.

  Base: $n = 1$. Claramente I vence o jogo.

  Supondo a hipótese verdadeira para todo $m < n$:

  Passo: Dada uma árvore B, $T(V, E)$, com $|V| = n$ vértices. Notemos inicialmente o seguinte fato: Para cada folha v no maior nível de T, a folha w, irmã de v, também está no maior nível de T. Isso é verdade pois cada vértice de uma árvore B possui ou 2 filhos, ou nenhum filho, portanto v possui irmão w em T. Além disso v está no maior nível de T, o que implica que w é uma folha.

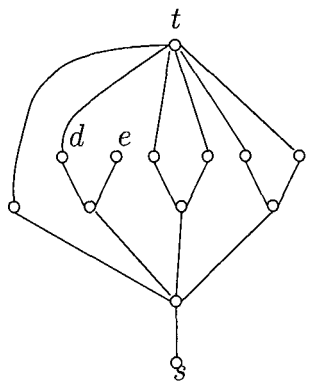


Figura 3.4: Árvore de três níveis (disjunção)

Sejam v e w duas folhas irmãs no maior nível de T. Tomemos a árvore induzida $T' = T[V\backslash\{v, w\}]$ e conectemos todas as suas folhas a t, pela hipótese de indução sabemos que I consegue formar um caminho até alguma folha f de T'. Em T, ou bem f é uma folha e o jogo está vencido; ou bem f é pai de v e w. Portanto, pela estratégia definida, quando I jogar em f a árvore enraizada neste vértice não possui nenhum vértice jogado por II. Logo, II só poderá jogar em um dos filhos de f, consequentemente I vence jogando no outro filho de f.

- Para o jogador II:

  A argumentação é parecida com a anterior, notando que agora uma árvore B possui alguma folha f que não está conectada a t. Deve - se provar que II é capaz de cortar todos os caminhos da raiz a folhas da árvore, exceto o caminho até f.

Consideraremos agora o grafo da figura 3.4, representamos por V seu conjunto de vértices. O grafo $G[V\backslash\{s, t\}]$ (onde $G[U]$, $U \subset V$, representa o subgrafo de G induzido pelo conjunto de vértices U) é uma árvore de 3 níveis (raiz, filho e neto) onde um filho da raiz é uma folha, e os demais filhos são adjacentes a 2 netos da

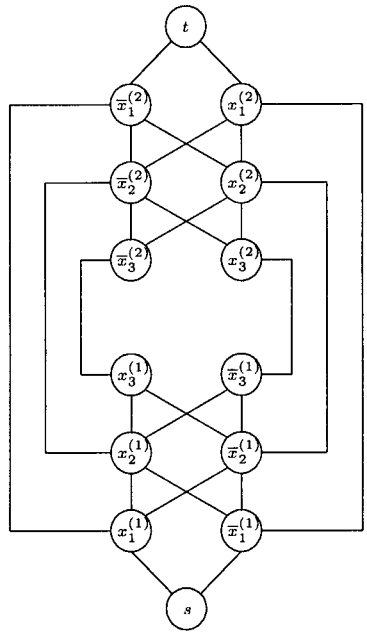


Figura 3.5: Cubo

raiz. Chamaremos os netos irmãos de d e e, convencionando que, para uma dupla de irmãos d e e se apenas um desses vértices estiver conectado a t, esse vértice será o d. I possuirá uma estratégia vencedora nessa árvore se ao menos um dos netos e for adjacente a t. A estratégia para ambos os jogadores é a seguinte:

I joga na raiz e II joga na folha filha da raiz (é fácil ver que esses lances são forçados). Agora I joga em outro filho, se os dois netos adjacentes a este estiverem conectados a t, II não conseguirá obstruir o caminho de I. Do contrário, II joga em d. Para os demais filhos em que I jogar, II sempre responderá em d.

O último pedaço dessa construção é o cubo, exibido na figura 3.5, é através dele que decidimos os valores das variáveis.

O cubo consiste de um conjunto de quadras da forma $\{x_i^{(1)}, \overline{x}_i^{(1)}, x_i^{(2)}, \overline{x}_i^{(2)}\}$.

Escolhendo vértices em ordem crescente de índices, o primeiro jogador pode jogar de forma a ocupar $\{x_i^{(1)}, x_i^{(2)}\}$ ou $\{\overline{x}_i^{(1)}, \overline{x}_i^{(2)}\}$ para cada quadra. Note que

todos os lances do outro jogador são forçados dentro do cubo, por exemplo, se I joga em $x_1^{(1)}$, II deve jogar em $\overline{x}_1^{(2)}$, em seguida I joga forçadamente em $x_1^{(2)}$ e II responde em $\overline{x}_1^{(1)}$. Com isso I determina qual a atribuição de x_1, no caso citado x_1 é verdadeira.

Na construção completa nós usaremos o cubo para representar as variáveis com alguns vértices extras chamados de *quantificadores*, a presença de um quantificador entre duas quadras, representantes das variáveis x_i e x_{i+1}, significa que o quantificador de x_i é diferente de x_{i+1}. Usaremos a árvore B para representar a conjunção com uma cláusula por folha, cada uma das folhas dessa árvore é raiz de uma árvore de três níveis que representa uma cláusula. A árvore B possui $3n$ folhas extras, cada uma dessa folhas será raiz de uma cláusula $(x_i \vee \overline{x}_i)$, sendo 3 cláusulas para cada valor de i distinto.

A construção completa possuirá os seguintes vértices:

s, t;

$x_i^{(1)}, \overline{x}_i^{(1)}, x_i^{(2)}, \overline{x}_i^{(2)}$ para $1 \leq i \leq n$, chamados vértices variáveis;

q_i para cada $Q_i \neq Q_{i+1}$, chamado quantificadores;

$2(k+3n)-1$ vértices formando uma árvore B com $k+3n$ folhas, essas folhas serão denominadas de l_i, para $1 \leq i \leq k+3n$;

$l_j(0)$, $1 \leq j \leq k+3n$;

$y(0,j), y(a,j), y(b,j)$ para cada literal y na cláusula j, $1 \leq j \leq k$ (esses vértices em conjunto com $l_j(0)$ e l_j, formam uma árvore de três níveis que representa a j-ésima cláusula);

$x_i(0, k+i), x_i(a, k+i), x_i(b, k+i)$ para $1 \leq i \leq n$.

$x_i(0, k+n+i), x_i(a, k+n+i), x_i(b, k+n+i)$ para $1 \leq i \leq n$.

$x_i(0, k+2n+i), x_i(a, k+2n+i), x_i(b, k+2n+i)$ para $1 \leq i \leq n$.

E também apresentará, além das arestas de B as seguintes arestas:

$(s, x_1^{(1)})$, $(s, \overline{x}_1^{(1)})$, $(t, x_1^{(2)})$, $(s, \overline{x}_1^{(2)})$.

$(x_i^{(2)}, x_{i+1}^{(2)})$, $(x_i^{(2)}, \overline{x}_{i+1}^{(2)})$, $(\overline{x}_i^{(2)}, x_{i+1}^{(2)})$, $(\overline{x}_i^{(2)}, \overline{x}_{i+1}^{(2)})$, para $1 \leq i < n$.

$(x_i^{(1)}, \overline{x}_i^{(2)})$, $(\overline{x}_i^{(1)}, x_i^{(2)})$, para $1 \leq i \leq n$.

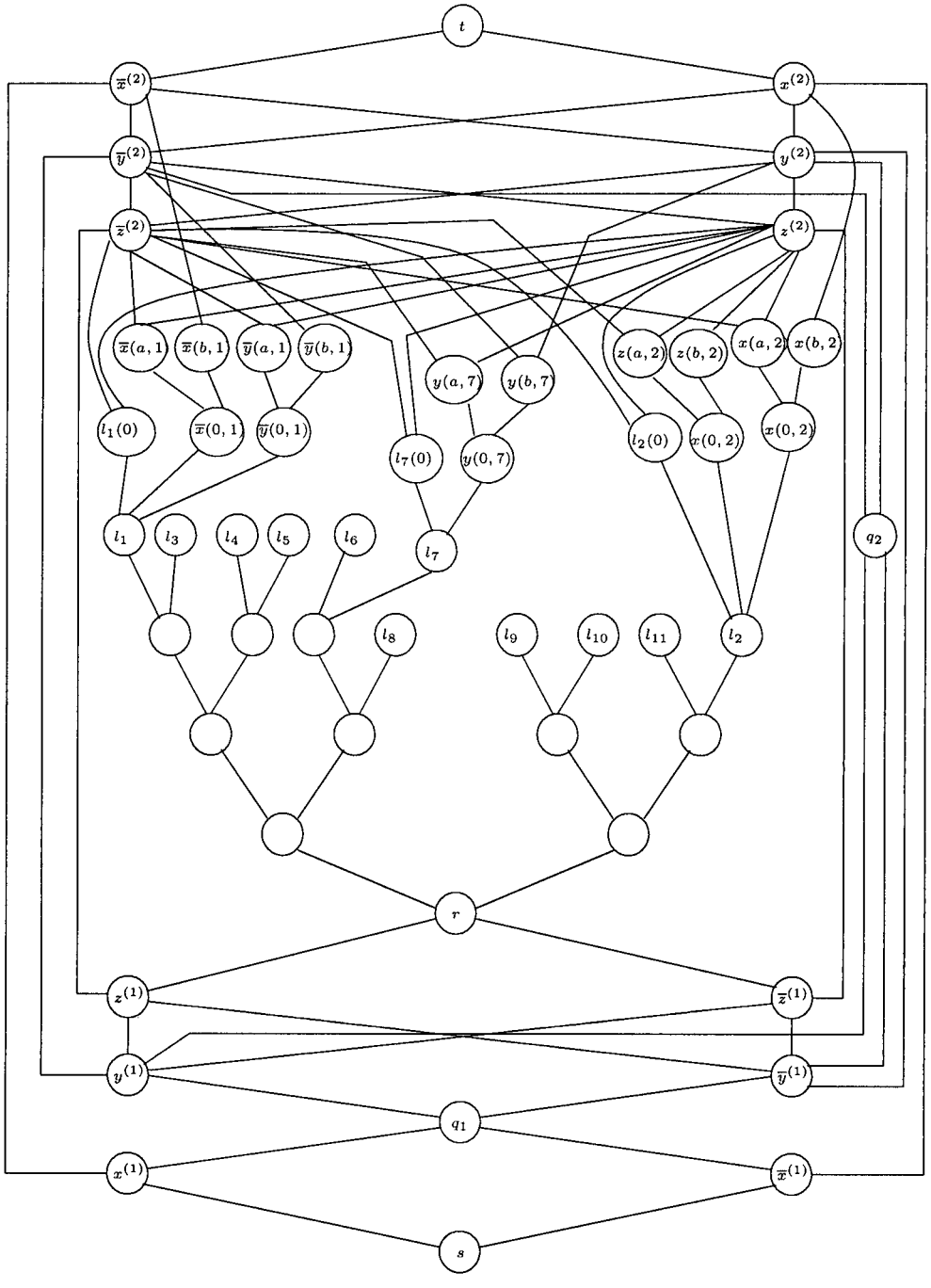


Figura 3.6: Grafo do jogo para a instância $\exists x\ \forall y\ \exists z\ ((\overline{x} \vee \overline{y}) \wedge (x \vee z))$ com algumas árvores de 3 níveis omitidas.

$(x_n^{(1)}, r), (\overline{x}_n^{(1)}, r)$ onde r é a raiz de B;

$(x_i^{(1)}, x_{i+1}^{(1)}), (x_i^{(1)}, \overline{x}_{i+1}^{(1)}), (\overline{x}_i^{(1)}, x_{i+1}^{(1)}), (\overline{x}_i^{(1)}, \overline{x}_{i+1}^{(1)})$ se $Q_i \neq \exists$ ou $Q_{i+1} \neq \forall$;

$(x_i^{(1)}, q_i), (\overline{x}_i^{(1)}, q_i)$ se $Q_i \neq Q_{i+1}$;

$(q_i, x_{i+1}^{(1)}), (q_i, \overline{x}_{i+1}^{(1)})$ se $Q_i = \exists$, $Q_{i+1} = \forall$;

$(q_i, x_i^{(2)}), (q_i, \overline{x}_i^{(2)})$ se $Q_i = \forall$, $Q_{i+1} = \exists$;

$(l_j, l_j(0))$ para $1 \leq j \leq k + 3n$;

$(l_j(0), x_n^{(2)})$, $(l_j(0), \overline{x}_n^{(2)})$, $1 \leq j \leq k + 3n$.

$(l_j, y(0,j))$, $(y(0,j), y(a,j))$, $(y(0,j), y(b,j))$, $(y(a,j), x_n^{(2)})$, $(y(a,j), \overline{x}_n^{(2)})$, $(y(b,j), y^{(2)})$ para cada literal y na cláusula j, $1 \leq i \leq k$;

$(l_{k+hn+i}, x_i(0, k+hn+i)), (x_i(0, k+hn+i), x_i(a, k+hn+i)), (x_i(0, k+hn+i), x_i(b, k+hn+i)), (x_i(a, k+hn+i), x_n(2)), (x_i(a, k+hn+i), \overline{x}_n^{(2)}), (x_i(b, k+hn+i), x_i^{(2)}), (x_i(b, k+hn+i), \overline{x}_i^{(2)})$ para $1 \leq i \leq n, 0 \leq h \leq 2$.

A figura 3.6 mostra uma construção completa para a seguinte instância: $\exists x\ \forall y\ \exists z\ ((\overline{x} \vee \overline{y}) \wedge (x \vee z))$, onde, para uma maior clareza na construção, apenas as árvores de três níveis enraizadas em l_1, l_2, l_7 são exibidas.

Para melhor compreensão desta transformação é importante mostrar um fato sobre árvores estritamente binárias.

Proposição 3.2. *Toda árvore estritamente binária com l folhas possui $2l$ - 1 vértices.*

Prova: Provaremos por indução no número de folhas. Para $l = 2$ o resultado é trivial. Suponhamos a hipótese verdadeira para todo $l' < l$. Dada uma árvore com l folhas, tomemos a sua folha no nível mais alto. Removemos esta folha e seu irmão (o qual já comentamos que existe). A nova árvore tem $l - 1$ folhas e é estritamente binária, pela hipótese de indução esta árvore possui $2(l-1) - 1 = 2l$ - 3 vértices. Logo, a árvore com l folhas possui $2l$ - 1 vértices. □

Esta proposição serve para provar que é sempre possível construir a árvore B mencionada na construção completa. E mais ainda, não importa qual o seu

formato, ela sempre terá a mesma quantidade de vértices e as estratégias dos jogadores nessa estrutura será sempre aquela definida anteriormente.

Observe ainda que a estrutura utilizada para representar um mesmo bloco de variáveis é exatamente um cubo. O quantificador q_i é a única ligação, na parte inferior do cubo, entre as quadras que representam x_i e x_{i+1}, quando $Q_i = \exists$ e $Q_{i+1} = \forall$. E, quando $Q_i = \forall$ e $Q_{i+1} = \exists$, as quadras de x_i e x_{i+1} se relacionam normalmente (como quadras consecutivas em um cubo) e além disso, a parte inferior e a parte superior de x_i possuem arestas até q_i.

Observe também que as cláusulas extras (com árvores de 3 níveis enraizadas nas folhas l_{k+1} até l_{k+3n}) possuem uma construção diferente das demais cláusulas.

A construção completa do grafo determina portanto uma transformação de fórmulas em grafos, onde número de vértices é da mesma ordem de grandeza do número de cláusulas mais o número de variáveis. A criação do cubo, da árvore B e das árvores de três níveis são claramente polinomiais, logo a transformação completa também é polinomial.

Nos resta provar agora que a instância do QBF é verdadeira se, e somente se, I possui uma estratégia vencedora no grafo obtido com a transformação.

Primeiramente descreveremos uma *forma normal*, conjunto das melhores estratégias de I e II, depois mostraremos que nenhum dos dois jogadores consegue alterar o resultado final do jogo caso desvie da forma normal.

Forma normal

Inicialmente I tem a iniciativa, ele joga no cubo escolhendo vértices que vão representar o valor do primeiro bloco de variáveis existenciais, I joga no cubo começando da quadra de x_1 e depois de acabar com a primeira quadra, joga nas variáveis x_i em ordem crescente de i. As respostas de II são forçadas. Terminado o primeiro bloco de variáveis existenciais I joga no primeiro quantificador

(o que representa o início do primeiro bloco de variáveis universais). II passará a ter a iniciativa e decidirá qual o valor das variáveis dentro do bloco universal, II também faz essas atribuições em ordem crescente das variáveis, I deverá responder as jogadas de II na mesma quadra em que II jogou de forma a confirmar a escolha de II, isto é, se II joga em $x_i^{(1)}$ (respectivamente $x_i^{(2)}$), então I deverá jogar em $\overline{x}_i^{(1)}$ (respectivamente $\overline{x}_i^{(2)}$). Ao final do bloco de variáveis universais, II jogará no próximo quantificador, esta jogada dará a iniciativa a I. O jogo continuará dessa forma até que o cubo seja completamente usado, quando isso acontecer I joga na raiz r da árvore B. Note que há um caminho, constituído por vértices escolhidos por I, de s a r. O jogo procede na árvore B conforme definido anteriormente. I consegue formar caminho até uma folha f de B, porém, é II quem decide até qual folha I formará o caminho. A folha f é raiz de uma árvore de 3 níveis representando uma cláusula C. A construção da instância ocorre de forma que, se C for verdadeira para a atribuição das variáveis determinada enquanto o jogo ocorria no cubo (considerando os vértices escolhidos por I como literais verdadeiros), então a árvore de três níveis que representa C terá alguma de suas folhas e conectada a um vértice v escolhido por I na parte superior do cubo, onde v se encontra num caminho de vértices escolhidos por I até o vértice t. Desta forma I ganha o jogo se, e somente se, Φ é verdadeira dada a atribuição de valores obtidos jogando no cubo.

Detalhadamente, a forma normal consiste na seguinte estratégia vencedora para cada um dos jogadores:

- Para I:

  Se a primeira quadra desocupada corresponde a uma variável existencial, escolha um valor para a variável representada por esta quadra que torne a fórmula verdadeira e jogue os dois vértices correspondentes dentro da quadra nos 2 próximos lances. As respostas de II serão forçadas.

Se a próxima quadra desocupada corresponde a $\forall x_{i+1}$ e a anterior correspondia a $\exists x_i$, jogue em q_i. E então responda as jogadas de II dentro do bloco universal, de forma a confirmar a escolha de II e sem permitir que II jogue em 2 vértices de uma mesma horizontal do cubo. Por exemplo, se II joga em $x^{(1)}$, responda em $\overline{x}^{(1)}$.

Quando o cubo estiver preenchido, utilizando a estratégia já conhecida, forme um caminho em B da sua raiz até alguma de suas folhas, folha a qual é raiz de uma árvore de 3 níveis que não contém nenhum movimento de II. E então jogue na árvore de 3 níveis com a estratégia já conhecida (notando que uma folha desta árvore não está diretamente conectada a t, mas sim a um vértice que esteja num caminho, de vértices escolhidos por I, até t).

- Para II:

  Se a primeira quadra desocupada corresponde a uma variável universal, escolha um valor para a variável representada por esta quadra que torne a fórmula falsa e jogue os dois vértices correspondentes dentro da quadra nos 2 próximos lances.

  Se a próxima quadra desocupada corresponde a $\exists x_{i+1}$ e a anterior correspondia a $\forall x_i$, jogue em q_i. E então responda (de forma forçada) as jogadas de I no próximo bloco de variáveis.

  Quando o cubo estiver preenchido. Utilizando a estratégia já conhecida, bloqueie todos os caminhos em B exceto um que conduza a uma folha que representa uma cláusula falsa. E então jogue na árvore de 3 níveis enraizada nessa folha.

Mostraremos agora que a alteração das jogadas não altera o resultado do jogo.

II não pode vencer se a fórmula for verdadeira

Prova: Se II desviar da forma normal no cubo, enquanto estiver jogando dentro de um bloco de variáveis existenciais, então II perde no próximo lance de I. Se o desvio for dentro de um bloco de variáveis universais, então I deverá atribuir aleatoriamente valores para as variáveis dentro do bloco universal. Todas as jogadas de II são forçadas (impedindo que I forme caminho até t com os vértices jogados). No fim desse bloco, o jogador I joga no quantificador seguinte e ganha o jogo, formando um caminho que utiliza os vértices já jogados por I e o quantificador. Se II deixar de jogar num quantificador universal, então I joga nesse quantificador e ganha. Se alguma das respostas de II tiver sido escolhida enquanto II desviou da forma normal, o jogo volta a forma normal.

As estratégias anteriormente descritas garantem a vitória para I se II desviar o jogo na árvore B ou em alguma das árvores de três níveis. □

I não pode vencer se a fórmula for falsa

Prova: Se o primeiro desvio de I for dentro de um bloco de variáveis universais no cubo, então II deve jogar de forma a bloquear alguma horizontal da quadra em que estiver jogando. Se a parte inferior do cubo for bloqueada, então todos os caminhos de s a t serão bloqueados e II terá vencido. Note que todos os caminhos de s a t entre vértices de quadras abaixo da quadra bloqueada foram cortados durante a forma normal. Se a parte superior for bloqueada, II ainda terá que bloquear alguns caminhos que passam através dos vértices das árvores de três níveis. É importante notar que, até II fazer seu próximo lance para bloquear esses caminhos, é possível I faça mais um lance sem seguir a forma normal, totalizando 2 lances fora da forma normal.

Se o desvio de I for dentro do bloco existencial, então a estratégia de II continuará sendo tentar cortar o grafo na parte inferior de uma quadra, ou no próximo quantificador (ou na raiz de B), se a última quadra jogada foi a última do bloco.

Se isso não puder ser feito, II tentará cortar a parte de cima do cubo nos dois vértices superiores da quadra. Se o desvio de I ocorreu quando deveria jogar o primeiro vértice da quadra, então I pode evitar que II realize o objetivo citado, para isso I precisa voltar a jogar na forma normal, a única diferença é que II terá a iniciativa dentro do bloco existencial. O jogo continuará dessa forma até que I jogue novamente fora da forma normal, nesse caso II conseguirá cumprir alguma das metas citadas acima, pois ele faz o primeiro lance em cada quadra, ou a quadra ou quantificador onde I fez o seu movimento não normal será alcançado retornando o jogo ao normal; ou o bloco existencial chega ao fim e então II joga no próximo quantificador (que também pode ser a raiz de B) e ganha o jogo pois este quantificador corta todos os caminhos não bloqueados de s a t. Em todos esses casos, ou o jogo retorna ao normal ou II vence imediatamente, bloqueando a parte inferior do cubo, ou II bloqueia a parte de cima do cubo. Note que até o jogador II bloquear uma horizontal superior do cubo, o jogador I pode fazer até 2 lances fora da forma normal. Portanto, até que o jogador II venha jogar novamente, I pode fazer mais um lance fora da forma normal, totalizando 3 movimentos fora da forma normal.

Nos resta provar que II pode bloquear todos os caminhos de s a t no caso em que ele cortou a parte de cima do cubo, jogando algum par $x_q^{(2)}$, $\overline{x}_q^{(2)}$, ainda que I faça até 3 movimentos fora da estratégia normal.

Primeiramente notemos que cortar a parte superior do cubo (jogar em $x_q^{(2)}$, $\overline{x}_q^{(2)}$) não garante a vitória. Pois os vértices da forma $y(b,j)$ podem estar conectados a vértices localizados acima da horizontal cortada. Como II jogou na forma normal sabemos que os caminhos de s a t, utilizando $(x_i^{(1)}, \overline{x}_i^{(2)})$ ou $(\overline{x}_i^{(1)}, x_i^{(2)})$, estão bloqueados para todo $i < q$.

Portanto, chamaremos os vértices $y(0,j)$, $y(b,j)$ de *especiais* e decidiremos a estratégia de II de acordo com o seguinte: "II usou no mínimo dois dos seus movimentos fora da forma normal para jogar em vértices especiais?" Caso não, todos os caminhos de s a t são trivialmente cortados jogando de forma complementar nos

pares $y(0,j)$,$y(b,j)$. Isto é, se I jogar em algum desses vértices II joga no outro, isso é feito para todo j.

No caso de I ter usado dois dos seus movimentos fora da forma normal para jogar em vértices especiais. O jogador II tem bloquear as possibilidades do jogador I formar um caminho de s a t. Sabemos que esse caminho não poderá ser formado apenas por vértices do cubo e quantificadores, pois para variáveis x_i onde $i < q$, esses caminhos foram bloqueados por II jogando na forma normal. Para $i \geq q$ esses caminhos foram bloqueados pois eles precisariam de um dos vértices $x_q^{(2)}$ ou $\overline{x}_q^{(2)}$. Logo, a única possibilidade de I formar um caminho de s a t é utilizando os vértices da árvore B.

Pelo parágrafo anterior, II tem como estratégia cortar a árvore B pela raiz, se isso não for possível II utilizou seu terceiro movimento fora da forma normal jogando na raiz. Como I jogou na raiz, II deve determinar um caminho na árvore B que leve a uma folha *morta* de B, isto é l_{k+q}, l_{k+n+q}, l_{k+2n+q}, neste caso escolha um caminho em que a árvore de três níveis, enraizada na folha morta, não tenha nenhum movimento de I (isto é possível pois apenas 2 lances de I foram feitos em vértices especiais, o outro foi feito na raiz da árvore B, logo alguma dessas folhas é raiz de uma árvore sem jogadas de I). Note que as folhas mencionadas são raízes de árvores de 3 níveis conectadas a $x_q^{(2)}$, $\overline{x}_q^{(2)}$, além de $x_n^{(2)}$, $\overline{x}_n^{(2)}$. Isto justifica o nome utilizado para tais folhas de B. □

Desta forma notamos que esta instância do HEX é equivalente a instância dada de QBF.

Capítulo 4

Além de $PSPACE$

Neste capítulo serão exibidos jogos combinatórios que correspondem a problemas de complexidade superior a $PSPACE$. O primeiro deles é o GO e o segundo um jogo sobre fórmulas booleanas.

O primeiro jogo não sabemos classificar sua complexidade ao certo, mas sabemos que ele é ao menos tão difícil quanto qualquer problema $PSPACE$, pois o GO é $PSPACE$-difícil. Usaremos esse jogo para caracterizar quais tipos de jogos são resolvidos por uma DTM que use espaço polinomial. Além disso, para a redução do QSAT para o GO usaremos uma nova técnica, faremos uma transformação intermediária para o GEOGRAPHY.

O segundo jogo é completo para tempo exponencial, este será útil pois faremos sua análise utilizando uma máquina de Turing alternada (ATM). Embora não tenhamos utilizado este tipo de máquina antes ela é muito comum no estudo de complexidade de jogos combinatórios, o motivo será exibido a seguir.

4.1 GO

Como vimos para o caso do GEOGRAPHY e do HEX, se tivermos um limite polinomial para o número de jogadas de um jogo combinatório, obtemos um algoritmo $PSPACE$ para solucionar o jogo utilizando a árvore do jogo.

Estudaremos agora o GO, ele destaca - se dos demais já estudados pois não há um limite polinomial conhecido para o número de jogadas, logo a árvore do jogo poderá ter altura exponencial, o que inviabiliza o uso de tal árvore para a construção do algoritmo espaço polinomial.

Embora não possamos utilizar a árvore do jogo, poderíamos tentar outra técnica para resolver o jogo, porém nenhum algoritmo espaço polinomial é conhecido.

Na verdade, adotaremos em nossa generalização apenas um subconjunto do conjunto de regras do GO. Embora as regras utilizadas sejam suficientes para o nosso propósito, a principal consequência da omissão de algumas regras é que poderíamos ter repetições infinitas de jogadas. O conjunto original de regras possui regras para restringir repetições. Estudos de complexidade para esse grupo de regras foram desenvolvidos em [12]. A formulação que adotaremos do problema é a mesma feita em [8], esta também será a referência básica sobre o jogo.

Apesar de não sabermos se o GO é $PSPACE$, conhecemos uma redução do QSAT para o GO, o que o torna $PSPACE$-difícil. Essa redução consiste em transformar uma dada instância de QSAT em uma instância de GEOGRAPHY e então transformá - la numa instância do GO.

4.2 Resultados sobre GEOGRAPHY

Conforme mencionado utilizaremos o GEOGRAPHY para obter a transformação do QSAT no GO. No entanto, o GEOGRAPHY conforme foi definido não parece tão interessante para obtermos a redução para um jogo de tabuleiro. Preci-

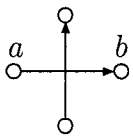


Figura 4.1: Cruzamento

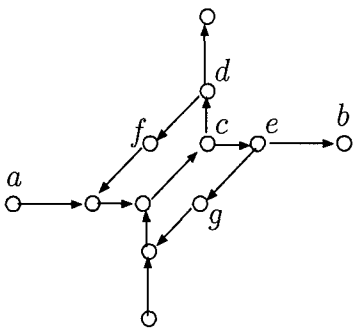


Figura 4.2: Estrutura que substitui o cruzamento.

saremos portanto de uma versão restrita do GEOGRAPHY. Provaremos que estas versões restritas continuam $PSPACE$-completas.

Definição: *GEOGRAPHY PLANAR* é o GEOGRAPHY jogado sobre um grafo planar.

Teorema 4.1. *GEOGRAPHY PLANAR é PSPACE-completo*

Prova: Seja D a instância de GEOGRAPHY obtida com a redução do QSAT apresentada na seção 3.3.2. Notemos que só há cruzamentos entre arcos de D que saem dos vértices consecutivos aos vértices cláusulas. Logo, os cruzamentos possuem o formato da figura 4.1.

Para provar que GEOGRAPHY PLANAR é $PSPACE$-completo trocamos as arestas que se cruzam em D pela construção da figura 4.2.

Claramente, se o último vértice jogado na figura 4.1 foi a e um jogador L pretende jogar b, então seu oponente R não poderá evitar que essa jogada ocorra

no grafo da figura 4.2. Pois, se L joga na direção de b, a primeira opção de desvio do jogador R é no vértice c. Se R jogar em d, o jogador L vence o jogo jogando em f. Logo, quando c for o último vértice jogado, R deverá jogar em e. Alcançado o vértice e, será a vez de L. Se L jogar em g, então perderá o jogo na próxima jogada, assim que R jogar no sucessor de g. Portanto L é forçado a jogar em b. Consequentemente a única opção para ambos os jogadores é jogar até alcançar o vértice b, onde será a vez de R jogar. É isso que ocorre jogando no grafo original (figura 4.1).

Analogamente se um jogador pretende atravessar o cruzamento de baixo para cima, então não pode ter seu trajeto alterado.

Logo um jogador vence o jogo da instância obtida de GEOGRAPHY PLANAR se, e somente se, o vencesse na instância correspondente de GEOGRAPHY. □

Definição: Seja D um digrafo. O *grau* de um vértice v de D (abreviadamente $d(v)$) é igual ao grau de saída de v mais o grau de entrada de v.

Teorema 4.2. *GEOGRAPHY PLANAR com vértices de grau no máximo três (GP3) é PSPACE-completo.*

Prova: Note que o vértice inferior do último diamante (definido na seção 3.3) de GEOGRAPHY PLANAR é o único vértice de GEOGRAPHY PLANAR com grau de entrada e grau de saída maiores que 1, chamaremos esse vértice de *vértice de escolha.* Acrescentando alguns vértices a instância de GEOGRAPHY PLANAR transformaremos este vértice num vértice com grau de entrada 2 e grau de saída 1, conforme a figura 4.3. Vértices extras serão incluídos para preservar a escolha do jogador II.

Para os demais vértices com grau superior a 3, basta substituirmos por novas estruturas conforme especificado na figura 4.4.

Para cada vértice v substituído acrescentamos ao digrafo original $O(d(v))$ vértices, logo o acréscimo total de vértices é $O(|V|^2)$. Logo, computamos essa transformação em tempo polinomial.

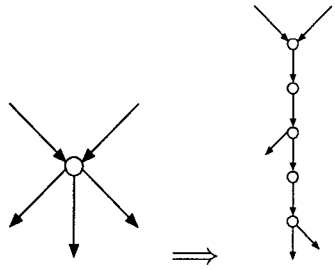


Figura 4.3: Transformação do vértice de escolha

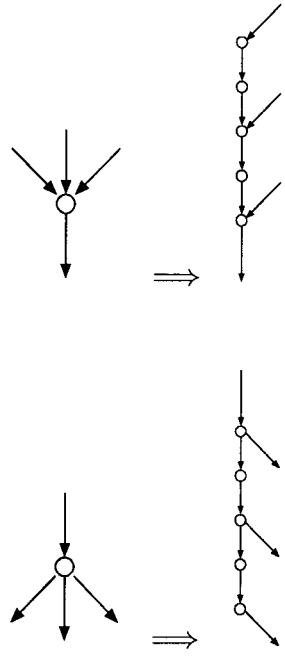


Figura 4.4: Transformação de vértices com grau superior a 3

Claramente, com a nova estrutura o jogador I vencerá o jogo se, e somente se, o vencesse na estrutura original. □

4.3 GO é $PSPACE$-difícil

Com os resultados obtidos na seção anterior estamos aptos a obter o nosso resultado principal. A idéia para a obtenção do resultado é construir dentro de um tabuleiro de GO uma instância de $GP3$ de forma que um jogador vence a instância de $GP3$ se, e somente se, vence a instância de GO obtida da instância de $GP3$.

4.3.1 Regras do GO

GO é um jogo de dois jogadores, Negras e Brancas, jogado num tabuleiro que é uma grade de 19×19 lugares chamados *pontos*, a figura 4.5 mostra o tabuleiro de GO. Um jogador faz uma *jogada* colocando uma pedra de sua própria cor em um ponto vazio do tabuleiro, as Negras são as primeiras a jogar, as jogadas alternam entre os jogadores e um jogador pode *passar* (ceder a vez ao oponente) a qualquer momento. O jogo termina quando ambos os jogadores passam.

Com a evolução do jogo as pedras formam *grupos*. Um grupo negro (respectivamente branco) é um conjunto conexo maximal de pedras de negras (respectivamente brancas). Um grupo de pedras de uma mesma cor estará *cercado* se ele não é adjacente a nenhum ponto vazio. Após cada jogada das Negras (respectivamente Brancas), todos os grupos brancos (respectivamente negros) cercados são removidos, seguidos de todos os grupos negros (respectivamente brancos) cercados. As pedras que forem removidas por ficarem cercadas são denominadas pedras *capturadas*.

No final do jogo todas as pedras *mortas* são removidas do tabuleiro, onde uma pedra é dita *morta* se ela pode ser cercada independente de qualquer tentativa de

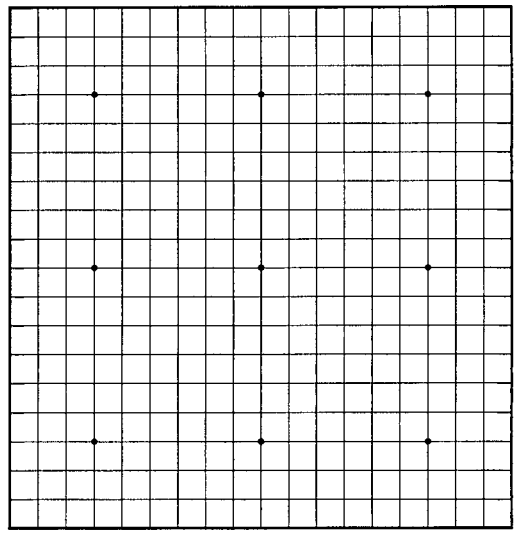


Figura 4.5: Tabuleiro de GO

salvá - la. Um ponto vazio é denominado *território negro* se não existir nenhum caminho, formado apenas por pontos vazios, entre o ponto e uma pedra branca. Definimos *território branco* de forma análoga. Observamos que a classificação dos territórios (negro ou branco), só ocorre depois da remoção de pedras mortas.

A pontuação das Negras é o total de territórios negros menos o número de peças negras capturadas, analogamente definimos a pontuação das Brancas. O jogador que possuir maior pontuação vence a partida.

Exemplo: A figura 4.6 mostra uma posição de um jogo de GO num tabuleiro 5 $\times$ 5. Supondo que essa é uma posição final do jogo (os dois jogadores passaram em seus últimos lances) e nenhuma pedra foi capturada no decorrer do jogo, então as Negras possuem 10 territórios (à esquerda das pedras negras) e as Brancas 5 (à direita das pedras brancas). Logo, as Negras venceram.

Observações: Notemos que determinados grupos não podem ser capturados, por exemplo, se um grupo cerca dois pontos vazios isolados, onde cada um desses

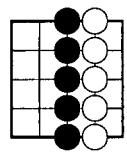


Figura 4.6: Negras venceram a partida.

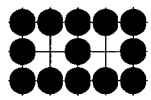


Figura 4.7: Grupo que não pode ser capturado (dois olhos).

pontos recebe o nome de *olho*, o grupo não pode ser capturado. Pois para cercar este grupo, o adversário precisaria ocupar os dois olhos simultaneamente, como ele não pode fazer isso o grupo é incapturável. Damos um exemplo de grupo incapturável na figura 4.7. Chamaremos um grupo conexo a dois olhos de sua mesma cor de *grupo seguro.*

A nossa construção a seguir é baseada numa disputa onde um dos jogadores tenta conectar um grande grupo de peças a dois olhos enquanto o oponente tentará cortar essa possibilidade.

4.3.2 Posição de GO

Conhecidas as regras do jogo, descreveremos o problema de decisão associado ao GO:

Instância: Um tabuleiro $n \times n$, uma distribuição de peças sobre este tabuleiro e a indicação de qual jogador tem a vez.

Pergunta: Negras vencem o jogo?

Exibiremos agora uma redução do QSAT ao GO utilizando o $GP3$ como problema intermediário.

A idéia é construir uma posição onde os jogadores simulariam uma partida de $GP3$. Brancas possuem um conjunto de territórios *garantidos* (definiremos este termo em momento oportuno, informalmente, um território branco está garantido se independente das jogadas das Negras, ao término do jogo o território continuará branco), enquanto as Negras possuem apenas 2 territórios garantidos. As Brancas possuem um grande grupo de pedras que está quase cercado, este grupo de pedras possui uma ligação com a região do tabuleiro onde simularemos o $GP3$. Existem poucas casas livres no tabuleiro, de tal forma que, a captura do grande grupo branco significaria vitória das Negras; enquanto se Brancas conectarem esse grupo a dois olhos, Brancas vencem a partida. Chamaremos o conjunto de territórios brancos garantidos de *Território branco* e o grande grupo de pedras brancas chamaremos de *grande grupo branco.*

A construção segue o esquema apresentado na figura 4.8. Ao longo da especificação, citaremos apenas o estado de parte do tabuleiro, as partes do tabuleiro que não forem especificadas possuem pedras negras.

Dois fatos são importantes nessa construção: O primeiro deles é que o Território branco está garantido, ou seja, ele é cercado por um grupo seguro e a cada duas colunas vazias ele possui uma coluna de pedras brancas que vão até o penúltimo ponto vazio antes da outra borda do grupo. Dessa forma, as pedras negras que forem postas dentro desta região serão consideradas pedras mortas, portanto, as Negras são incapazes de diminuir a pontuação das Brancas naquela região (conforme figura 4.9). Outro fator importante é que todas as pedras negras estão ligadas a dois olhos, os dois únicos territórios negros. Logo, as Brancas não tem chance de capturar nenhum território negro, além disso, o grande grupo branco é atravessado por pedras negras a cada duas linhas. Logo, se as Brancas

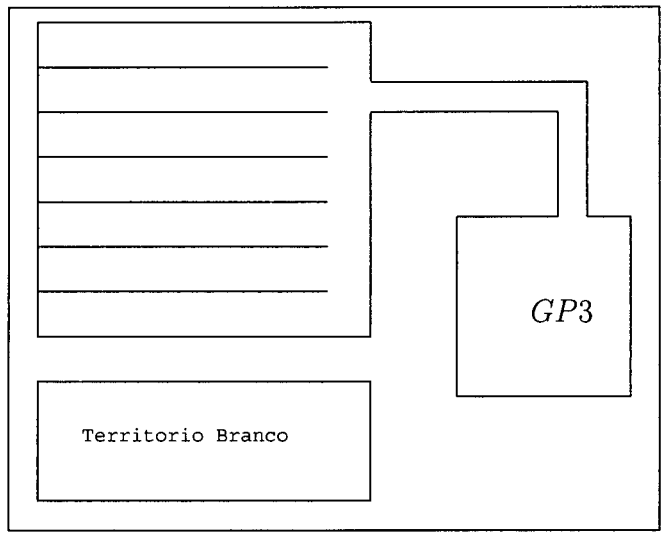


Figura 4.8: Estrutura da posição de GO

perderem este grupo não terão chance de recapturar o território. A figura 4.10 mostra o grande grupo branco e a sua ligação com a instância de $GP3$.

Dada uma instância do QSAT com número par de variáveis, construímos a instância do $GP3$ conforme especificado anteriormente. Agora transformaremos essa instância numa instância de GO, devemos portanto observar alguns detalhes sobre o grafo de $GP3$.

Transformaremos cada vértice e arco de $GP3$ em uma estrutura específica no GO. Notemos que existem 5 tipos de vértices diferentes no $GP3$ (figura 4.12), as respectivas transformações de cada um desses vértices estão na figura 4.13.

Cada uma das estruturas diferentes será ligada por *condutores* (figura 4.11).

Construiremos a posição de GO de tal forma que Brancas jogam primeiro, e sempre que chegarmos a uma construção que represente um vértice do grafo, será a vez das brancas. Notemos que o uso de vértices triviais após o uso de vértices de escolha, para trocar a iniciativa do jogo, não será necessário, pois os vértices de escolha receberão estruturas distintas de acordo com o jogador que tiver a vez.

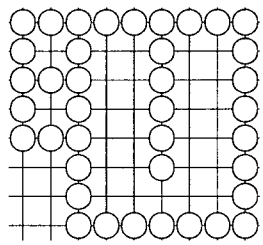


Figura 4.9: Território branco.

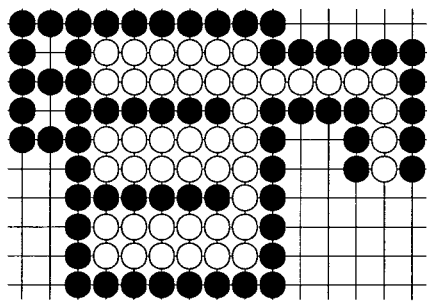


Figura 4.10: Grupo branco e o início de $GP3$.

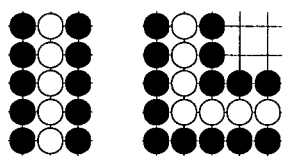


Figura 4.11: Condutores

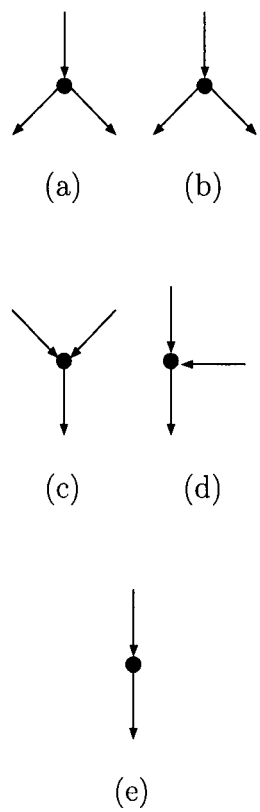


Figura 4.12: (a)Escolha de I (b)Escolha de II (c)Junção (d)Teste (e)Trivial

Logo, trataremos os vértices triviais como arcos e na instância do GO eles serão substituídos por condutores. Consequentemente, omitiremos a transformação dos vértices triviais na figura 4.13.

Os vértices de junção e teste diferem - se pelo fato destes estarem presentes apenas nas laterais dos diamantes, enquanto aqueles simbolizam todos os demais vértices no grafo com grau de entrada 2 e saída 1.

Demonstraremos que com a construção feita, os jogadores estão jogando o $GP3$ dentro do tabuleiro de GO. Para isso observaremos cada pedaço da construção separadamente, lembrando sempre que Brancas movem primeiro.

Na estrutura (a) temos:

(i) *Brancas jogam em 1 ou perdem imediatamente.*

Prova: Claramente se as Negras jogam em 1 capturam o grupo branco. □

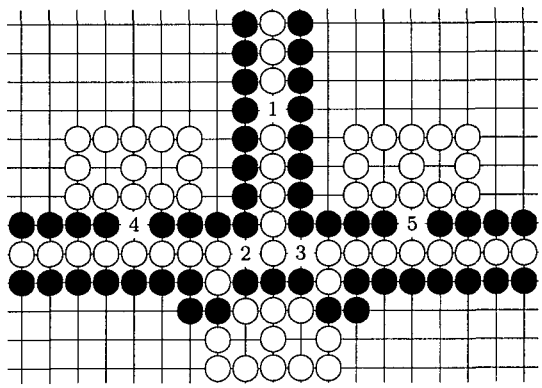
(ii) *Após a jogada das Brancas em 1, Negras jogam em 2 ou 3 ou perdem em 2 lances.*

Prova: Negras deverão deixar 4 ou 5 vazias, suponhamos que 4 esteja vazia, Brancas jogam em 2 e fazem uma ameaça dupla: Jogar em 4 e vencer a partida salvando o seu grupo, pois o conectaria aos dois olhos. Ou jogar em 3 e capturar o grupo de três pedras negras, e no próximo lance conectariam inevitavelmente seu grupo de pedras a dois olhos e venceriam a partida. □

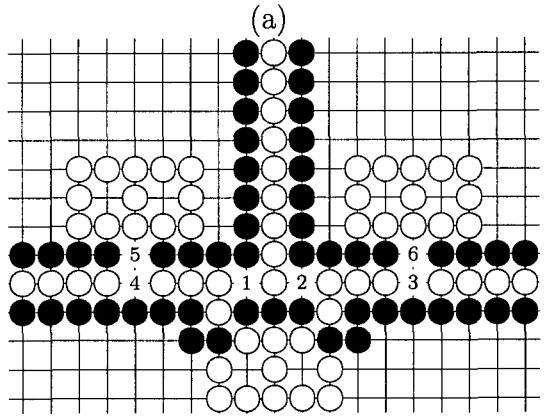
(iii) *Após a jogada de Negras em 2 (respectivamente 3) Brancas devem responder em 3 (respectivamente 2) ou perdem.*

Prova: Do contrário, suponha que as Negras tenham jogado 2 e as Brancas não responderam em 3, então quando as Negras jogarem em 3 o grupo branco estará cercado. □

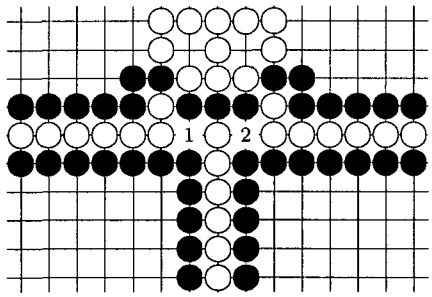
(iv) *Se as brancas jogaram em x (onde $x \in \{2, 3\}$) Negras devem responder em $x + 2$.*

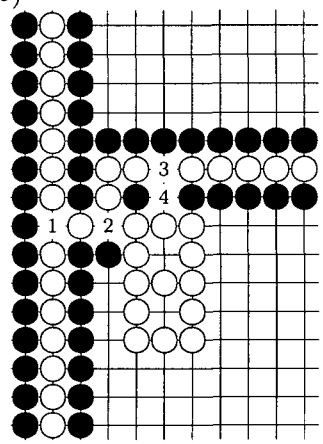


(a)



(b)





(c) (d)

Figura 4.13: (a)Escolha de I (b)Escolha de II (c)Junção (d)Teste

Prova: Se as Brancas jogarem em $x + 2$ conectarão seu grupo a dois olhos. □

Notemos que o único lance não forçado foi primeiro lance das Negras, onde puderam optar entre jogar em 2 ou 3.

Na estrutura (b) temos:

(i) *Brancas jogam em 1 ou 2, senão perdem em dois lances.*

Prova: 3 ou 4 deverão ficar vazias. Suponhamos que 3 esteja vazia, se as Negras jogam em 1, as Brancas são obrigadas a jogar em 2. As Negras jogam em 3 e ganham o jogo. □

(ii) *Negras respondem a jogada das Brancas em 1 (respectivamente 2) jogando em 2 (respectivamente 1) ou perdem imediatamente.*

Prova: Do contrário as brancas capturam o grupo de três pedras Negras e inevitavelmente conectam o seu grande grupo a dois olhos. □

(iii) *Brancas respondem a jogada Negra em 1 (respectivamente 2) jogando em 3 (respectivamente 4) ou perdem imediatamente.*

Prova: Do contrário Negras jogam em 3 (respectivamente 4) e ganham. □

(iv) *Negras respondem a jogada das Brancas em 4 (respectivamente 3), jogando em 5 (respectivamente 6) ou perdem.*

Prova: Este ítem está claro. □

Devemos notar aqui que o único lance não forçado é o primeiro lance feito pelas Brancas, elas podem optar entre 1 ou 2.

Na estrutura (c) temos:

(i) *Em c todos os lances são forçados.*

Prova: Brancas impedem a captura de seu grande grupo e as Negras impedem a captura do seu grupo de 3 pedras que representaria a perda do jogo. □

Na estrutura (d) temos:

(i) *Se o jogo entra em (d) pela parte de cima da estrutura, então Brancas jogam em 1, Negras em 2 e então o jogo sai da estrutura (d).*

Prova: Os lances iniciais das Brancas em 1 e o das Negras em 2 são forçados, após esses lances Negras estão ameaçando capturar o grupo branco com um único lance no bloco de junção, logo as Brancas deverão jogar no bloco de junção. □

(ii) *Se o jogo entra pela direita de (d) já tendo passado pelo topo então Negras vencem.*

Prova: Brancas são forçadas a jogar em 3,como 2 possui uma pedra negra, as Negras vencem jogando em 4. □

(iii) *Se o jogo entra pela direita de (d) sem ter passado pelo topo então Brancas vencem.*

Prova: As Brancas jogam em 3 e ameaçam conectar seu grande grupo a dois olhos jogando em 4 ou 2, e as Negras não poderiam evitar a ameaça dupla. □

Notemos que para cada variável um jogador teve de optar se tomaria o caminho da direita ou o da esquerda. Após terminadas as variáveis as Brancas escolhem uma cláusula, e dentro dessa cláusula as Negras indicam o caminho até uma estrutura de teste. Ao final desse caminho elas ganham se o jogo já tiver passado por lá, e perdem caso o contrário.

Antes da conclusão final é preciso observar que, ainda que as Brancas conectem seu grande grupo a dois olhos, pelas regras do GO as Negras ainda podem

fazer jogadas. As Brancas podem inclusive perder algumas pedras na região do tabuleiro onde o $GP3$ foi simulado. Entretanto, daremos dimensões as regiões do tabuleiro de forma a evitar que uma alteração do resultado ocorra. Note que a representação de cada um dos vértices de $GP3$ no tabuleiro de GO tem dimensões inferiores a 20×20, portanto construiremos a região do tabuleiro onde $GP3$ é simulado com dimensões $20|V| \times 20|V|$, onde V é o conjunto de vértices de $GP3$. O Território branco terá dimensões $45|V| \times 30|V|$ (esse número não conta os dois olhos conexos as pedras do território branco) e o retângulo do grande grupo branco terá dimensões $45|V| \times 60|V|$.

Com as dimensões de cada região do tabuleiro vemos que o Território branco é maior que o dobro da região onde o $GP3$ é jogado, portanto a única forma das Negras obterem uma pontuação maior que as Brancas é capturando todo o grande grupo branco. E como já foi mencionado antes se as Negras capturarem o grande grupo branco, as Brancas não podem recapturá - lo, logo as Negras vencem.

Com a descrição acima vemos que Negras e Brancas simularam uma partida de $GP3$ num tabuleiro de GO onde as Negras simularam o jogador I e as Brancas o II, logo Negras vencem se, e somente se o jogador I vence o $GP3$.

Vimos também que a dimensão do tabuleiro de GO é $O(V)$, claramente a construção de cada uma das regiões do tabuleiro é polinomial e a construção do instância completa de GO também é.

Logo QSAT $\propto$ GO e portanto GO é $PSPACE$-difícil.

4.4 *ATM*

Nessa seção exibiremos um novo modelo de máquina de Turing, a *máquina de Turing alternada* (ATM). Um dos primeiros, e possivelmente o mais relevante, artigos em revista sobre o assunto foi [16]. Nossas definições e resultados iniciais vem desta referência.

Definição: Um *circuito lógico* é uma árvore $T(V,E)$ onde os vértices V são denominados *portas*. Cada porta i possui um valor $s(i)$ associado a ela, onde $s(i) \in \{1,0,\vee,\wedge\} \cup \{x_1,\overline{x_1},x_2,\overline{x_2},...\}$. Seja i uma porta se $s(i) \in \{1,0,x_1,\overline{x_1},x_2,\overline{x_2},...\}$, então i não possui descendentes; se $s(i) \in \{\vee,\wedge\}$, então i possui de 1 a $|V|-1$ descendentes. As folhas de T são denominadas *entradas* do circuito e a raiz r de T a *saída* do circuito.

Denominamos de conjunto de variáveis (booleanas) do circuito o conjunto $\{x_1,x_2,...\}$. Dada uma atribuição de verdade para as variáveis do circuito, temos: Cada porta do circuito possui um valor booleano determinado pela função $\mu : V \to \{0,1\}$. $\mu(i)$ é decidido recursivamente, se $s(i) \in \{x_1,\overline{x_1},x_2,\overline{x_2},...\}$, então $\mu(i)$ terá seu valor 0 ou 1 determinado pela atribuição de verdade das variáveis. Se $s(i)=1$, então $\mu(i)=1$; se $s(i)=0$, então $\mu(i)=0$. Se $s(i)=\vee$, então $\mu(i)=1$ se existe algum j descendente de i tal que $\mu(j)=1$. Se $s(i)=\wedge$, então $\mu(i)=1$ se para todo descendente j de i, $\mu(j)=1$. O valor booleano do circuito é $\mu(r)$ (valor booleano da saída).

Usualmente utilizaremos apenas a palavra circuito para nos referirmos a um circuito lógico.

Observamos que a definição de circuito lógico aqui adotada não é tão comum na literatura, para uma definição mais geral veja [11] (capítulo 4).

A ATM é uma generalização da $NDTM$. A ATM possui todos os componentes de uma $NDTM$, a grande diferença é a divisão do conjunto de estados Q em dois conjuntos disjuntos: U, conjunto de *estados universais*; e $Q \setminus U$, o conjunto de *estados existenciais*.

Seja N uma $NDTM$. Seja T_N a árvore cujos caminhos maximais são todas as possíveis sequências de computação de N com entrada x, denominaremos esta árvore de *árvore de computação*. Sabemos que N aceita x, se existe algum caminho maximal de T_N que termine numa configuração de aceitação.

Pensando nessa árvore como um circuito, onde as folhas representam as variáveis do circuito, os vértices internos representam operadores e a raiz é a saída do

circuito. Notemos que, para simular uma computação de uma $NDTM$ todos os vértices internos do circuito devem representar uma disjunção. Dessa forma basta que uma folha tenha valor 1 para que o circuito seja verdadeiro.

Os dois parágrafos anteriores fornecem visões diferentes sobre a computação de uma $NDTM$. Uma abordagem trata da alcançabilidade e a outra como circuitos lógicos. Baseado na segunda abordagem foi criado o conceito de ATM M. Ao invés de tratarmos todos os vértices internos de T_M como existenciais, adotaremos alguns vértices internos como universais. Lembrando que os vértices representam configurações, iremos atribuir valores booleanos às configurações. Configurações com estado q_Y possuem valor 1 (verdadeira), as com estado q_N possuem valor 0. Uma configuração existencial $C_\vee$ possui valor 1 se existe alguma configuração C' tal que $C_\vee \vdash_M C'$ ($\vdash_M$ possui o mesmo significado dos capítulos anteriores) e C' vale 1. Uma configuração universal $C_\wedge$ é verdadeira se para toda configuração C'' tal que $C_\wedge \vdash_M C''$, C'' é verdadeira. M aceita uma entrada x se a configuração inicial for verdadeira.

Temos agora um entendimento completo de como essa generalização nos pode ser útil. A diferença entre os dois tipos configurações da ATM é semelhante a diferença entre as posições em que o jogador I possui a vez e as que o jogador II possui a vez num jogo combinatório.

4.4.1 Definição de ATM

Faremos agora a definição formal de ATM:

Definição: Uma *máquina de Turing alternada* (com k fitas), abreviadamente ATM, é uma quíntupla ordenada $M = (Q, U, \Sigma, \delta, q_0)$.

- Q, o conjunto finito de estados.
- U, o conjunto finito de *estados universais.*
- Σ, o conjunto finito de símbolos de M.

- δ, uma função de transição.

- q_0 é o estado inicial.

Notemos que o conjunto de estados universais U também informa outro elemento importante de M, o conjunto de *estados existenciais* $Q \setminus U$.

Os demais elementos de M são iguais aos elementos de uma $NDTM$.

Assim como as demais máquinas de Turing, a ATM pode ter múltiplas fitas, usualmente adotaremos k. Nosso modelo padrão de ATM terá $k-2$ fitas de trabalho (de leitura e escrita), uma fita de entrada (apenas leitura) e uma de saída (apenas escrita).

Definição: A *configuração* $C = (q, u_1, w_1, ..., u_k, w_k)$, de uma ATM é análoga a de uma $NDTM$ na seção 2.1.

Se o estado de uma configuração é universal (respectivamente existencial) dizemos que esta é uma *configuração universal* (respectivamente *existencial*). Se o estado de uma configuração é q_Y, dizemos que esta é uma *configuração de aceitação.* Se o estado de uma configuração é q_N, dizemos que esta é uma *configuração de rejeição.*

A cada configuração atribuiremos um valor booleano. Denominaremos de *status* a função $l{:}C \rightarrow \{0,1\}$ que determina se uma configuração aceita ou rejeita a entrada.

Em uma ATM, para qualquer configuração C onde o estado de C não pertence a $\{q_Y, q_N\}$, existe ao menos uma configuração C' tal que $C \vdash_M C'$. E, se o estado da configuração C é q_Y ou q_N, então não existe C' tal que $C \vdash_M C'$, também chamaremos essas configurações de *terminais* (não possuem sucessor).

Descreveremos intuitivamente a computação de uma ATM M. Suponhamos que uma configuração C de M possua i configurações sucessoras: β_1, ..., β_i. Podemos ver o passo seguinte a configuração C como a inicialização de i novos processos: β_1, ..., β_i. Cada processo β assim que descobre seu *status*, informa para C. De

acordo com o status de seus descendentes C decide o seu próprio status. O interessante é que C não precisa que a computação de todos os seus descendentes termine. Pois dado C existencial, se algum β (sucessor de C) tem status 1, então $l(C) = 1$. Por outro lado, dado C universal, se algum β (sucessor de C) tem status 0, então $l(C) = 0$. Dessa forma obtemos um procedimento recursivo para a computação de M.

Formalmente:

$$l(C) = \begin{cases} \bigvee_{C \vdash_M C'} l(C') & , C \text{ é existencial} \\ \bigwedge_{C \vdash_M C'} l(C') & , C \text{ é universal.} \\ 1 & , \text{o estado de } C \text{ é } q_Y. \\ 0 & , \text{o estado de } C \text{ é } q_N. \end{cases}$$

A aceitação de uma entrada x por uma ATM M está vinculada ao status da configuração inicial de M:

Definição: Uma ATM M *aceita* x se, e somente se, a configuração inicial C_0, $C_0 = (q_0, \triangleright, x, \triangleright, b, ..., \triangleright, b)$, é tal que $l(C_0) = 1$.

Notemos que uma DTM é uma ATM M, onde para toda configuração C de M existe no máximo uma configuração C' tal que $C \vdash_M C'$. Uma $NDTM$ é uma ATM onde $U = \varnothing$.

4.4.2 Complexidade de uma ATM

Assim como as demais máquinas de Turing estudadas, a ATM também determina classes de complexidade de tempo e espaço. As classes de complexidade para ATM são denominadas $ATIME$ e $ASPACE$.

Definição: Seja M uma ATM e C_0 sua configuração inicial. M aceita uma entrada x em *tempo* t, se $l(C_0) = 1$ e apenas os valores $l(C)$ das configurações C que estão a no máximo t passos de distância de C_0 ($C_0 \vdash_M^t C$) são utilizadas para a atribuição deste resultado.

Em termos do circuito formado: Consideraremos apenas a parte do circuito que possui altura $t + 1$.

Definição: Seja M uma ATM e C_0 sua configuração inicial. M aceita uma entrada x em *espaço* s, se $l(C_0) = 1$ e apenas as configurações que utilizam espaço máximo s (analogamente a definição do capítulo 1, o espaço utilizado pela configuração de uma máquina de Turing é o número de células distintas utilizadas pela maior fita) são utilizadas para a atribuição deste resultado.

Em analogia com as demais máquinas de Turing, sejam $T(n)$ e $S(n)$ funções. Uma *ATM M aceita x em tempo $T(n)$* (respectivamente espaço $S(n)$), onde $n = |x|$ se para tal x ela utiliza tempo máximo $T(n)$ (respectivamente espaço $S(n)$). O conjunto dos elementos x, aceitos por uma ATM M em tempo $T(n)$, constituem a linguagem da máquina, $L(M)$, para tal tempo de computação.

Notemos a analogia entre ATMs e as demais máquinas de Turing quanto aceitação de tempo e espaço. Definiremos as classes de complexidade alternada de forma análoga a que foi feita na seção 2.2. Seja uma função f:$\mathbb{N} \rightarrow \mathbb{N}$. Nossas classes serão denominadas de $ATIME(f(n))$, para complexidade de tempo e $ASPACE(f(n))$ para o espaço.

Proposição 4.3. *Dada uma ATM M com k fitas, complexidade de espaço $S(n) \geq n$ e entrada x, $|x| = n$, então podemos construir uma ATM M' com uma fita e complexidade de espaço $S(n)$ tal que M aceita x se, e somente se, M' aceita x.*

Prova: Demonstração análoga a do corolário 2.3. □

Dois resultados comparando ATMs com DTMs, exibidos em [16], refletem a força do uso dessa máquina para o estudo da complexidade de jogos combinatórios:

Teorema 4.4. $PSPACE = \bigcup_{k \geq 1} ATIME(n^k)$

Não faremos a demonstração deste resultado, ver [16], mas ele mostra a conexão, exibida no capítulo 3, entre jogos de tempo limitado por um polinômio e a classe $PSPACE$.

Nosso maior interesse nessa seção é comparar a classe **E** com classes de complexidade alternada. Para obter nosso resultado principal precisamos desenvolver os dois lemas seguintes.

Lema 4.5. *Se* $S(n) \geq \log n$, *então*

$$ASPACE(S(n)) \subseteq \bigcup_{k>0} TIME(k^{S(n)})$$

Prova: Seja M uma ATM que utiliza espaço $S(n)$, onde n é o tamanho da entrada x. Seja C_M uma configuração de M. Sabemos que o espaço utilizado por M é menor ou igual a $S(n)$. Como o número de símbolos de $Q \cup \Sigma$ é constante, existe uma constante c_M, dependente apenas de M, tal que $c_M^{S(n)}$ é um limite superior para o número de configurações distintas de M. Construiremos agora uma máquina M' para simular M.

M' escreve na fita todas as $c_M^{S(n)}$ configurações de M e atribui um valor a cada uma delas, inicialmente cada uma recebe ε. O algoritmo consiste em caminhar sobre a fita e decidir os valores das configurações que podemos decidir. Por exemplo, na primeira vez que passarmos sobre a fita atribuímos o valor 0 a configurações cujo estado é q_N e 1 se for q_Y, as demais permanecem inalteradas. Na próxima vez, passamos novamente sobre todas as configurações da fita. Para cada configuração verificamos suas sucessoras, utilizando a definição de $l(C)$, poderemos decidir o valor de algumas antecessoras das configurações anteriormente decididas. Nas próximas iterações repetiremos este processo. Com o procedimento descrito, a cada nova iteração temos que passar por toda fita, para cada configuração de M. Como M aceita em espaço $S(n)$ o maior caminho de computação tem comprimento máximo $c_M^{S(n)}$. Logo, M' deve passar $O(c_M^{S(n)})$ vezes por uma fita de tamanho $S(n).O(c_M^{S(n)})$, para cada uma das $c_M^{S(n)}$ configurações. Portanto, M' usa tempo $k^{S(n)}$, para alguma constante k. □

Definição: Seja M uma máquina de Turing de uma única fita. A *palavra de uma configuração* C de M, $C = (q, u, w)$, é a seguinte codificação ω. Sejam $u = u_1u_2...u_i$ e $w = w_1w_2...w_j$, então $\omega(C) = u_1u_2...u_{i-1}qu_iw_1w_2...w_j$.

Definição: Uma *função parcial* é uma função que não está definida para alguns elementos do domínio.

Lema 4.6. *Se* $T(n) \geq n$ *e* $c > 0$, *então*

$$TIME(T(n)) \subseteq ASPACE(c.\log(T(n)))$$

Prova: Seja M uma DTM que computa x, $|x| = n$, em tempo $T(n)$. Vamos supor que M possui uma única fita, portanto, pela proposição 2.2 adotaremos que M utiliza tempo $O(T^2(n))$. Faremos a seguinte alteração na máquina M, vamos supor que M só pára após $T(n)$ passos, independente da entrada x. Uma vez que M entre em q_Y, M ficará repetindo a mesma configuração até o momento $T(n)$. Essa nova máquina M aceita uma entrada x se entra no estado q_Y até o momento $T(n)$.

Seja $\Delta = Q \cup \Sigma$. Representaremos uma configuração C da máquina M por sua palavra $\omega(C)$, conforme definido anteriormente. Uma computação de M é uma sequência de configurações C_0,C_1,...,$C_{T(n)}$.

Chamaremos de $\lambda_{t,j}$ o símbolo na j-ésima posição de $\omega(C_t)$. Sem perda de generalidade adotaremos $\omega(C_0) = \triangleright q_0xbbb....$ Notemos que $\lambda_{t,j} \in \Delta$. Como a máquina M é determinística, a cada iteração um símbolo na posição j só será alterado se for alterado pela cabeça leitora, se a cabeça leitora se deslocar para a posição j (vindo da posição $j-1$ ou $j+1$) ou se a cabeça leitora sair da posição j, portanto $\lambda_{t,j}$ pode ser determinado consultando $\lambda_{t-1,j-1}$, $\lambda_{t-1,j}$, $\lambda_{t-1,j+1}$, $\lambda_{t-1,j+2}$. Logo, existe uma função parcial $ADJ_M : \Delta^4 \rightarrow \Delta$, onde:

$$\lambda_{t,j} = ADJ_M(\lambda_{t-1,j-1}, \lambda_{t-1,j}, \lambda_{t-1,j+1}, \lambda_{t-1,j+2}).$$

Construiremos uma ATM M' para simular a máquina M. A computação de M' consiste no seguinte processo recursivo: Inicialmente M' "adivinha" inteiros t e j, $0 \leq t, j \leq T(n)$ tais que $\lambda_{t,j} = q_Y$.

A partir de então M' precisa decidir se para um dado símbolo σ, $\lambda_{t,j} = \sigma$. Sabemos responder facilmente a essa questão nos casos em que $j = 0$ ($\lambda_{t,0} = \triangleright$) e para o caso $t = 0$ ($\lambda_{0,j}$ é o j-ésimo símbolo de $\omega(C_0)$).

Para o caso geral, $\lambda_{t,j} = \sigma$. M' adivinha 4 símbolos σ_{-1}, σ_0, σ_1 e σ_2. Se $\sigma \neq ADJ_M(\sigma_{-1}, \sigma_0, \sigma_1, \sigma_2)$, então M' rejeita. Senão M' escolhe k, $-1 \leq k \leq 2$, e repete o procedimento para verificar se $\lambda_{t-1,j+k} = \sigma_k$, para $t, j \geq 1$.

Para realizar a computação anterior M' só precisará armazenar os valores de t e j. Como $M \in TIME(O(T^2(n)))$, t e j são limitados por $O(T^2(n))$, logo para qualquer $c > 0$, existe um inteiro b tal que, se utilizarmos notação b-ária para os inteiros t e j, espaço $c.\log T(n)$ será suficiente para armazená - los. Portanto M' $\in ASPACE(c.\log T(n))$, para qualquer constante $c > 0$. □

Antes dos demais resultados, analisaremos o funcionamento de uma ATM utilizando o algoritmo de simulação descrito na demonstração do lema 4.6. Notemos que há momentos que a ATM deve "adivinhar" um valor, em linhas gerais isso significa que a computação está num estado existencial. Logo, basta que uma configuração subsequente seja verdadeira. Este "adivinhar" representa que, dentre as possíveis próximas configurações, a máquina passa para uma configuração seguinte verdadeira. Em outra etapa a máquina "escolhe" o valor de k, nesse momento a máquina está num estado universal. Não importa qual o valor de k escolhido, a computação deve ser verdadeira para qualquer um deles. Vimos portanto a diferença de procedimento da ATM quando está em um estado existencial, ou quando está num estado universal e como construir uma ATM.

Com os lemas anteriores demonstraremos o teorema seguinte:

Teorema 4.7. *Se* $S(n) \geq \log n$, *então*

$$ASPACE(S(n)) = \bigcup_{k>0} TIME(k^{S(n)})$$

Prova: Esse resultado decorre da combinação direta dos lemas 4.5 e 4.6. □

Corolário 4.8. $ASPACE(n) = \boldsymbol{E}$

O corolário anterior é facilmente obtido a partir do teorema 4.7. A partir de agora ao invés de provar que um problema é **E**-completo (através de redução polinomial), provaremos que o problema é completo para a classe $ASPACE(n)$.

Como procedemos com as classes de complexidade determinísticas e não determinísticas, após estudar as equivalências e resultados básicos das classes, estudaremos algum problema que pertença a classe. Nosso problema escolhido foi um jogo combinatório definido em [15], este também será a referência base até o final deste capítulo.

4.5 Um jogo exponencial

Essa seção tem o objetivo de apresentar um jogo, sobre fórmulas proposicionais, que é completo em tempo exponencial. Esse jogo não possui um nome em particular, manteremos portanto o nome atribuído no primeiro artigo em que esse jogo foi mencionado [15]. Nosso jogo será denominado de G_1.

Definição: Uma sentença está na forma $\overline{4}$-CNF se representa uma disjunção de cláusulas contendo apenas conjunções, onde cada cláusula tem no máximo 4 literais.

Notemos a analogia da notação, usamos 4-CNF para sentenças com cláusulas contendo exatamente 4 literais.

4.5.1 Regras de G_1

Dada uma sentença $\Phi(X, Y, t)$ na forma $\overline{4}$-CNF. Dividimos seu conjunto de variáveis booleanas em 3 conjuntos disjuntos: X, Y, t, onde representamos por

t o conjunto unitário $\{t\}$. Os jogadores jogam alternadamente e o jogador I é o primeiro a jogar. É fornecida uma atribuição inicial α para as variáveis de $X \cup Y$, convencionaremos que inicialmente t vale ε. Na sua vez o jogador I altera o a atribuição da variável t para 1 (verdade) e atribui valores para todas as variáveis em X. Na sua vez o jogador II, altera a atribuição da variável t para 0 e atribui valores as variáveis em Y. Um jogador perde se ao final de seu turno Φ for falsa.

Exemplo: Seja $\Phi = (t \vee x) \wedge (\bar{t} \vee y)$. E a atribuição inicial para (X, Y, t) é (1, 1, ε).

Em sua primeira jogada o jogador I altera o valor de t para 1 (isso é obrigatório) e altera o valor de x para 0, Φ é verdadeira para a atribuição corrente. II deverá fazer $t = 0$ e não importa qual o valor que ele escolha para y, a primeira cláusula de Φ é falsa. Logo, I venceu a partida.

Notemos que a variável t serve para determinar qual jogador possui a vez. Quando $t = 0$ ou ε (o que simboliza a posição inicial), é a vez de I e quando $t = 1$ é a vez de II. Criaremos portanto o conceito de *posição* como a quádrupla (Φ, X, Y, t). Citaremos apenas informalmente o termo posição ao longo do texto. Dessa forma não iremos explicitar a quádrupla a qual a posição se refere, embora forneceremos informações suficientes para a construção da mesma.

Com as regras acima podemos definir o problema de decisão associado a G_1:

Instância: Uma fórmula booleana $\Phi(X, Y, t)$ em $\overline{4}$-CNF e uma atribuição de verdade α para as variáveis de $X \cup Y$ e $t = \varepsilon$. I é o primeiro a jogar.

Questão: Existe uma estratégia de G_1 vencedora para I?

4.5.2 G_1 é E-completo

A prova que G_1 é E-completo pode ser dividida em duas partes. Primeiro, estabeleceremos que G_1 pode ser resolvido em tempo exponencial.

Teorema 4.9. $G_1 \in \boldsymbol{E}$

Prova: O algoritmo para resolver G_1 é baseado no seguinte fato: Dada uma instância $(\Phi(X,Y,\varepsilon),\alpha,\varepsilon)$, onde α é uma atribuição para $X \cup Y$ e seja p o número de posições possíveis para fórmula Φ fixa. Se após p iterações o jogo não acabou, então as posições estão repetindo e o jogador I não conseguiu vencer. Logo, essa instância é falsa. Notemos que $p \leq 2^{|\Phi|}$, onde $|\Phi|$ é o número de literais em Φ.

A idéia desse algoritmo é determinar o conjunto de posições para as quais I obtém vitória em no máximo p passos. O algoritmo é iterativo, determina primeiro as posições *finais* de vitória para I (Posições em que o jogador II já alterou Y e t, mas Φ ficou falsa). O próximo passo é determinar as posições em que I vence com 1 passo (II joga e perde). Depois, as posições onde I vença com 2 passos (I joga e II fica numa posição onde joga e perde). E assim sucessivamente até que determinemos todas as posições em que I vence com no máximo p passos. Após determinado este conjunto, verificamos se a posição inicial é vitoriosa para I checando se ela alcança alguma posição do conjunto em um único lance.

Notemos que para uma fórmula Φ fixa, as atribuições definem completamente as posições do jogo. Sejam π_α e π_β as posições representadas pelas atribuições α e β. Definiremos que uma atribuição α *alcança* β, quando a posição π_α alcança π_β em um único movimento.

O algoritmo exponencial para G_1 inicialmente escreve todas as possíveis atribuições de (X,Y,t), nessas atribuições t vale 0 ou 1. Como as possíveis atribuições tem tamanho $O(|\Phi|)$, esta etapa do algoritmo utiliza $O(|\Phi|.p)$ iterações. A próxima etapa é assinalar quais dessas atribuições podem ser uma posição final, ou seja, $t = 0$ e a atribuição de X, Y é falsa. Nessa etapa são gastos $O(|\Phi|.p)$ iterações, pois cada uma das p atribuições tem tamanho $O(|\Phi|)$.

Na próxima iteração marcamos quais das atribuições não marcadas, com $t = 1$ só alcançam atribuições marcadas. Essas são as posições em que I vence com um único passo (II tem a vez e perde para qualquer movimento que faça). Para cada uma das $O(p)$ atribuições não marcadas, gastaremos $O(|\Phi|.p)$ passos verificando se ela alcança alguma não marcada. Senão, marcamos esta atribuição.

Na próxima iteração marcamos quais das atribuições não marcadas, e com $t = 0$ alcançam alguma das configurações marcadas. Essas são as posições onde I vence com dois passos. Para cada uma das $O(p)$ configurações desmarcadas gastaremos $O(|\Phi|.p)$ passos.

Procederemos dessa forma, tratando de forma diferenciada atribuições que representam posições onde I tem a vez e as que II tem a vez. Faremos isso até a p-ésima iteração. Depois decidimos se a instância inicial $(\Phi, X, Y, \varepsilon)$ é vitoriosa para I, verificando se esta alcança em um único passo alguma das configurações assinaladas. Claramente esse algoritmo utiliza um número polinomial em $O(|\Phi|.p)$ de passos. Logo, $G_1 \in \mathbf{E}$ □

A segunda parte da prova de **E**-completude de G_1 consiste na prova que $\mathbf{E} \propto G_1$. Infelizmente, o tempo da dissertação se esgotou e nós não exibiremos esta demonstração, mas falaremos a idéia dessa demonstração que pode ser encontrada em [15].

A idéia é provar que qualquer linguagem em **E** se reduz a G_1. Pelo corolário 4.8 é equivalente provar que qualquer linguagem em $ASPACE(n)$ se reduz a G_1. Assim como foi feito por Cook [3](ver [1]) associaremos a computação de uma máquina M, que decida uma linguagem em $ASPACE(n)$, uma fórmula booleana F. Essa fórmula só será verdadeira se a atribuição de suas variáveis representar a codificação de configurações consecutivas de M. Associaremos a G_1 uma fórmula F' em $\overline{4}$-CNF que foi construída de acordo com F. E finalmente, a disputa entre os dois jogadores para manter F' verdadeira simulará a computação de M.

Capítulo 5

Conclusão

Essa dissertação é uma exposição de resultados selecionados envolvendo dois assuntos, jogos combinatórios e complexidade computacional. Interpretamos este trabalho como uma tentativa de ligação entre os dois temas.

Duas propostas de estudo futuro decorreram da elaboração da dissertação. Uma é concluir o entendimento sobre a completude de G_1. A outra é estudar o enfoque dado aos jogos combinatórios em [13], onde os jogos são analisados de forma geral. Em [13] é apresentada uma nova estrutura que represente qualquer jogo, e resultados gerais sobre jogos combinatórios são obtidos a partir desta estrutura. Esse enfoque é bem diferente da abordagem que demos ao assunto na presente dissertação, principalmente porque tratamos de jogos específicos, e também porque voltamos nossas atenções para os aspectos computacionais de um jogo. Além da referência mencionada, indicamos [2] que é uma das mais completas referências sobre o assunto.

Podemos dividir as demonstrações aqui presentes em dois grupos: As que são puramente sobre complexidade (resultados sobre as máquinas de Turing, comparações entre as classes) e as utilizadas para problemas de jogos.

Embora não tenhamos voltado nossas atenções para problemas em aberto, o segundo grupo de demonstrações constituem um conjunto de técnicas para classi-

ficação de jogos combinatórios. O conhecimento de tais técnicas nos fornece uma perspectiva de trabalhos futuros classificando jogos quanto a sua complexidade computacional de tempo e de espaço.

Bibliografia

[1] AHO, A. V., HOPCROFT, J. E., ULLMAN, J. D., *The Design and Analysis of Computer Algorithms.* 1 ed. Addison-Wesley, 1974.

[2] BERLEKAMP, E. R., CONWAY, J. H., GUY, R. K., *Winning Ways For Your Mathematical Plays.* 1 ed. Academic Press, 1982.

[3] COOK, S., "The complexity of theorem proving procedures", In: *Proc. 3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158, 1971.

[4] EVEN, S., TARJAN, R., "A Combinatorial Problem Which is Complete in Polynomial Space", *Journal of the ACM*, v. 23, n. 4, pp. 710–719, 1976.

[5] FRAENKEL, A. S., LICHTENSTEIN, D., "Computing a Perfect Strategy for $n \times n$ Chess Requires Time Exponential in n", *Journal of Combinatorial Theory, Series A*, v. 31, n. 2, pp. 199–214, 1981.

[6] GAREY, M. R., JOHNSON, D. S., *Computers and Intractability.* New York, W. H. Freeman and Company, 1979.

[7] KARP, R. M., "Reducibility among combinatorial problems", In: *Complexity of Computer Computations* (MILLER, R. E., THATCHER, J. W., eds.), pp. 85–104, Plenum Press, 1972.

[8] LICHTENSTEIN, D., SIPSER, M., "GO is Polynomial-Space Hard", *Journal of the ACM*, v. 27, pp. 393–401, 1980.

[9] MEYER, A. R., STOCKMEYER, L. J., "The equivalence problem for regular expressions with squaring requires exponential space", In: *13th Annual IEEE Symposium on Switching and Automata Theory*, pp. 125–129, 1972.

[10] MEYER, A. R., STOCKMEYER, L. J., "Word problems requiring exponential time", In: *Proc. 5th Annual ACM Symposium on Theory of Computing*, v. 48, pp. 1–9, 1973.

[11] PAPADIMITRIOU, C. H., *Computational Complexity*. 1 ed. Addison-Wesley, 1994.

[12] ROBSON, J. M., "The complexity of go", In: *Proc. Information Processing*, pp. 413–417, 1983.

[13] SALDANHA, N. C., *Tópicos em Jogos Combinatórios*. 18° Colóquio Brasileiro de Matemática, 1 ed. Rio de Janeiro, IMPA, 1991.

[14] SCHAEFER, T. J., "On The Complexity of Two-Person Perfect-Information Games", *Journal of Computer and System Sciences*, v. 16, n. 2, pp. 185–225, 1978.

[15] STOCKMEYER, L. J., CHANDRA, A. K., "Provably Difficult Combinatorial Games", *SIAM Journal on Computing*, v. 8, pp. 151–174, 1979.

[16] STOCKMEYER, L. J., CHANDRA, A. K., KOZEN, D. C., "Alternation", *Journal of the ACM*, v. 28, n. 1, pp. 114–133, 1981.

[17] STORER, J. A., "On the Complexity of Chess", *Journal of Computer and System Sciences*, v. 27, n. 1, pp. 77–100, 1983.