Teoria da Computação

Frequência

Duração: 2 horas

Universidade da Beira Interior

Quinta-Feira 10 de Janeiro de 2008

É proibido o uso de calculadora e de telemóvel.
Qualquer fraude implica reprovação na disciplina.
Só serão corrigidas as provas **legíveis**.

Relembramos que, na tradição da aritmética de Peano – que seguimos nesta Unidade Curricular, a notação $\mathbb{N}$ refere-se ao conjunto dos naturais incluindo o 0. Referiremo-nos ao conjunto dos naturais sem o 0 ($\{1, 2, 3 \ldots\}$) por $\mathbb{N}^*$.

Os tempos de resolução presentes em cada alínea são meramente indicativos e tem por objectivo ajudar-vos a planear a resolução.

1 Princípios da Teoria da Computação

Exercício 1 *(5 minutos) Classicamente associamos a noção de totalidade (da função sobre o domínio de interesse) à noção de computabilidade/decidibilidade. Um problema é decidível se existe uma processo/função total que calcule a solução ao problema (isto é, que converge para a solução qualquer que seja a instância considerada). Neste cenário, explique e contextualize a noção de parcialidade (i.e. função parcial sobre o domínio de interesse). Que representa ela quando tentamos qualificar a decidibilidade (ou não) dum problema?*

Resposta:

A totalidade de um processo de resolução de um problema P é necessária para garantir que para cada instância possível do domínio de interesse o processo converge para uma solução do problema P. A parcialidade implica que para (pelo menos) uma instância do problema o processo considerado diverge, ou seja não converge para uma solução.

Por outras palavras, em caso de parcialidade há uma instância para a qual processo considerado não encontrará uma solução.

Assim a totalidade é uma característica intrinsecamente ligada a noção de decidibilidade de um problema. Um processo de decisão é forçosamente total.

No caso da parcialidade, a relação é mais subtil. Aquando da definição de um processo p que se pretende de decisão para um problema P, se este for parcial para o domínio P então este não é um processo de resolução

(decisão) para P. Ou seja, p não permite afirmar que P é decidível. Mas isso não significa que P seja indecidível. Significa que p não é solução para P.

Agora, um problema P indecidível implica a inexistência de soluções totais, ou seja qualquer tentativa de resolução levará a processos no mínimo parciais (falar-se-á de problema semi-decidível).

□

2 Técnicas de Demonstração e OCaml

Exercício 2 *(10 minutos) Em qualquer conjunto de n inteiros positivos, existe sempre dois destes inteiros cuja diferença é divisível por $n-1$. Prove esta afirmação usando o princípio da gaiola de pombos e considerando o resto da divisão destes n valores inteiros por $n-1$.*

Resposta:

Sejam $a_1, a_2, \ldots a_n$ os n inteiros sorteados. Por cada a_i $(1 \leq i \leq n)$ consideremos r_i o resto da divisão por $n-1$ (i.e. $r_i = a_i \ mod \ (n-1)$).

Cada r_i definido desta forma pertence necessariamente ao conjunto $\{0, 1, \ldots, n-2\}$. Assim há $n-1$ valores possíveis para os r_i que, por seu turno, estão em numero n.

Usando o princípio da gaiola de pombos, se definirmos os pombos como sendo os r_i e as gaiolas os valores que os r_i podem tomar, então de certeza que haverá uma pelo menos uma gaiola com pelo menos dois pombos diferentes. Ou seja, existem necessariamente $j, k \in \{1, 2, \ldots, n\}$ tais que $j > k$ e $r_j = r_k$.

Logo a_j e a_k devolvem o mesmo resto quando dividido por $n-1$. Assim, $a_j - a_k$ é um múltiplo de $n-1$. Dito por outras palavras, $a_j - a_k$ é divisível por $n-1$.

□

Exercício 3 *As listas, como definidas nativamente em OCaml, são polimórficas. Depois de instanciadas só podem conter elementos de um só tipo. Ou seja, se uma lista contém um inteiro, então todos os outros elementos devem ser inteiros. Imagine que queremos dispor de uma lista que possa conter inteiros mas também reais. Uma forma de ultrapassar este facto é definir um tipo* agregador*, da seguinte forma:*

```
type valor = I of int | R of float
```

Assim, um elemento do tipo valor, ou é inteiro ou é real. Para ter uma lista como desejado basta defini-la como sendo do tipo `(valor list)`*. Uma lista deste tipo contém elementos do tipo valor, ou seja inteiros ou reais, tal como desejado. Um exemplo pode ser*

```
[I 2 ; I 6 ; R 7.9 ; R 9.2 ; I 10]
```

- *(5 minutos) Defina a função merge : int list →float list →valor list que aceita uma lista ordenada de inteiros e uma lista ordenada de reais (ambas de forma crescente) e que junte estas duas listas numa lista ordenada de tipo (*`valor list`*).*

  Por exemplo: merge [2;6;10] [7.9;9.2] deverá devolver [I 2 ; I 6 ; R 7.9 ; R 9.2 ; I 10].

- *(5 minutos) Defina, sem usar a função fold_left , a função recursiva split : valor list →(int list ⋆ float list), a função inversa de merge, que dado uma lista de valores separa os valores inteiros dos valores reais, num par de duas listas e conservando a ordem dos valores originais.*

  Por exemplo: split [I 2 ; I 6 ; R 7.9 ; R 9.2 ; I 10] deverá devolver ([2 ; 6 ; 10],[7.9 ; 9.2])

- *(5 minutos) Dê uma definição alternativa da função split que utilize a função fold_left .*

- *(10 minutos) Demonstre por indução estrutural que*

$$\forall\, lv \in (valor\ list), \forall\, li \in (int\ list), \forall\, lr \in (float\ list),$$
$$split\ lv = (li, lr) \implies length\ lv = length\ li + length\ lr$$

Resposta:

```
open List;;

type valor = I of int | R of float;;

type lista = valor list;;

let rec merge l1 l2 =
  match (l1,l2) with
      [],_ -> map (fun a -> R a) l2
    | _ , [] -> map (fun a -> I a) l1
    | a::ll1,b::ll2 ->
      if (float_of_int a)> b
      then  R b ::(merge l1 ll2)
      else  I a ::(merge ll1 l2)

let rec split_aux l lv =
  let (li,lr) = lv in
  match l with
      [] -> (rev li , rev lr)
    | (R el)::li -> split_aux li (li, el::lr)
    | (I el):: li -> split_aux li (el::li, lr)

let rec split l = split_aux l ([],[])

let split2 l =
  fold_left (fun (a,b) el -> match el with R r -> (a,r::b)| I i -> (i::a,b))
    ([],[]) l
```

Vamos proceder à prova de

$$\forall\, lv \in (valor\ list), \forall\, li \in (int\ list), \forall\, lr \in (float\ list),$$
$$split\ lv = (li, lr) \implies length\ lv = length\ li + length\ lr$$

por indução estrutural sobre a lista lv.

O princípio de indução sobre as listas é o seguinte:

$\forall P:$ *Predicado sobre listas (de tipo A)* se $P([]) \wedge \forall x:$ *elemento de tipo A*, $\forall l:$ *lista de tipo A*, $P(l) \implies P(x :: l)$ então $\forall l:$ *lista de tipo A*, $P(l)$.

Neste caso aqui, para uma lista lv qualquer de elementos de tipo *valor*, P é definido como $\forall\, li \in (int\ list), \forall\, lr \in (float\ list)$,
$split\ lv = (li, lr) \implies length\ lv = length\ li + length\ lr$

- Caso de base. Verificar se $P([])$.

  Este passo de prova é trivial. De facto as únicas listas li e lr possíveis são as listas vazias. (porque $split\ [] = ([], [])$). Neste caso, $length\ [] = 0 = 0 + 0 = length\ [] + length\ []$

- Caso do passo indutivo.

  Vamos considerar uma lista lv, de tipo lista de tipo valor, qualquer.

  A hipótese de indução (designamo-la de HI) é: $\forall\, li \in (int\ list), \forall\, lr \in (float\ list)$,
  $split\ lv = (li, lr) \implies length\ lv = length\ li + length\ lr$

Ou seja, admitimos que para uma lista lv qualquer, *split* lv resulta em duas listas (digamos li e lr) cuja a soma dos comprimentos é igual ao comprimento de lv.

Pretendemos provar que qualquer que seja o elemento a, a lista $a :: lv$ também carece da mesma propriedade.

Vejamos. O elemento a é de tipo *valor* ou seja é necessariamente da forma $R\ r$ ou $I\ i$.

Façamos assim uma analise por caso. caso $a = R\ r$ e caso $a = I\ i$. Se ambos os casos validam $P(a :: lv)$, então o passo indutivo está provado.

- Caso $a = I\ i$. Neste caso $split(a :: lv) = (i :: li, lr)$.
  É fácil de ver que $length(a :: lv) = 1 + lengthlv$
  É igualmente fácil de ver que $length(i :: li) = 1 + lengthli$.
  Temos de provar que $length(a :: lv) = length(i :: li) + lengthlr$. Usando as duas igualdades previamente estabelecidas temos: $1 + lengthlv = 1 + lengthli + lengthlr$. Ora por hipótese de indução temos que $length\ lv = length\ li + length\ lr$. Se substituímos $length\ lv$ por $length\ li + length\ lr$ na igualdade que pretendemos provar obtemos então:
  $1 + lengthli + lengthlr = 1 + lengthli + lengthlr$.
  Esta igualdade é trivialmente verdade. QED.
- Caso $a = R\ r$. Procede-se exactamente da mesma forma com a excepção de que é a componente da lista lr que vai sofrer alterações. Vejamos mesmo assim os detalhes
  Neste caso $split(a :: lv) = (li, r :: lr)$.
  É fácil de ver que $length(a :: lv) = 1 + lengthlv$
  É igualmente fácil de ver que $length(r :: lr) = 1 + lengthlr$.
  Temos de provar que $length(a :: lv) = lengthli + length(r :: lr)$. Usando as duas igualdades previamente estabelecidas temos: $1 + lengthlv = 1 + lengthli + lengthlr$. Ora por hipótese de indução temos que $length\ lv = length\ li + length\ lr$. Se substituímos $length\ lv$ por $length\ li + length\ lr$ na igualdade que pretendemos provar obtemos então:
  $1 + lengthli + lengthlr = 1 + lengthli + lengthlr$.
  Esta igualdade é trivialmente verdade. QED.

No final, tanto o caso de base como o caso indutivo foram provados. Logo,

$$\begin{array}{l}\forall\, lv \in (valor\ list), \forall\, li \in (int\ list), \forall\, lr \in (float\ list),\\ \quad split\ lv = (li, lr) \implies length\ lv = length\ li + length\ lr\end{array}$$

QED.

□

3 Autómatos Finitos e Linguagens regulares

Exercício 4 *(5 minutos) Considere o alfabeto $\Sigma = \{a, b\}$, defina um autómato* determinista *que reconheça a linguagem das palavras sobre Σ que contém pelo menos uma ocorrência de aa mas nenhuma ocorrência de aaa.*

Resposta: O autómato da figura 1 explica-se da seguinte forma.

O caminho entre 1 e 3 representa a ocorrência de dois a's. Do estado 3, o consumo de um b permite a garantir que já houve uma ocorrência de exactamente dois a's. Estando no estado 3, se não houver mais nada no buffer de entrada, então estamos na presença de uma ocorrência de aa que termina a palavra por analisar.

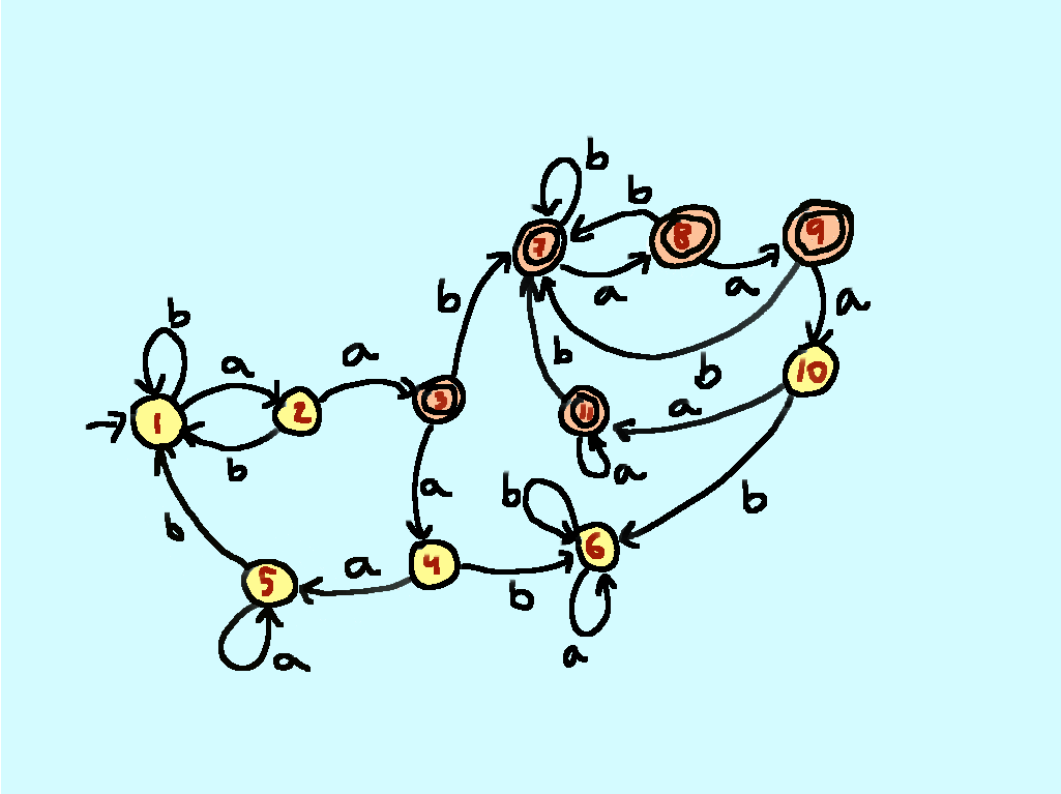

Figura 1: Autómato determinístico que reconhece palavras com pelo menos uma ocorrência de aa e 0 de aaa

Por isso o estado 3 é final. Se o buffer não está vazio e do estado 3 for consumido um a (a transição para o estado 4), então podemos estar na presença de uma ocorrência de aaa. Se do estado 3 for consumido um b então acabamos de confirmar mais uma vez que foi analisado uma ocorrência exacta de aa. Vamos neste caso para uma parte do autómato (que começa no estado 7) que vai agora tentar tratar de recusar aaa e aceitar qualquer outra sequência (já que foi confirmada uma ocorrência de aa).

Mas antes de analisar esta parte, voltamos ao estado 4 para terminar a analise do padrão aa. Recorda-se que no estado 4 podemos estar na presença do padrão aaa. Para confirmar (ou não) a ocorrência exacta de aaa a partir do estado 4 basta olhar para o próximo carácter por consumir. Se não houver mais nada por consumir, então estamos de facto na presença de uma ocorrência de aaa. Por isso o estado 4 não é final. Se do estado 4 for consumido um b, então está igualmente confirmado a presença do padrão aaa. Assim o estado destino é o estado 6 e daí, qualquer que seja o buffer, temos sempre de dar uma resposta negativa. o estado 6 é um estado poço.

Assim, os estados $1, 2, 3, 4, 5$ e 6 se dedicam na procura da primeira ocorrência de aa sempre rejeitando o padrão aaa.

Da mesma forma, os estados $7, 8, 9, 10$ e 11 dedicam-se na aceitação de qualquer palavra (relembra-se, já foi detectado nesta altura o padrão aa) onde não ocorre o padrão aaa. Este padrão ocorre quando estamos nos estados 10 e 6 (reutiliza-se aqui o estado poço), a semelhança dos estados 4 e 6 da parte anterior. Assim, destes estados, só o 10 (e o 6 por razões já expostas) não é final.

Alias, o leitor atento verá que o autómato global esta dividido em três partes, duas delas muito semelhantes.

Os estados $1, 2, 3, 4$, e 5, os estados $7, 8, 9, 10$ e 11, e finalmente o estado poço 6.

□

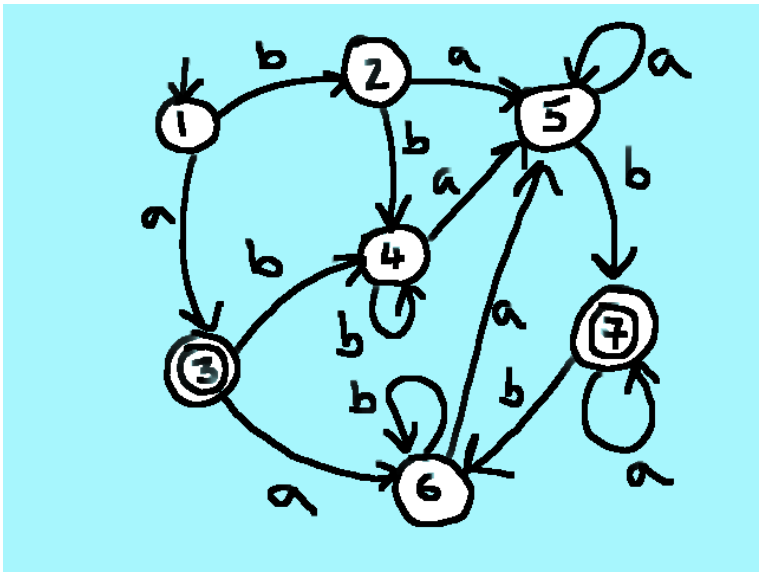

Figura 2: Autómato A_1

Exercício 5 *(10 minutos) Considere o alfabeto* $\Sigma = \{a, b\}$ *e o autómato* A_1 *da figura 2:*

Minimize o autómato A_1.

Resposta:

Vamos proceder, como manda o algoritmo visto nas aulas, por iterações na procura do ponto fixo da expressão $\bigcup_i \sim_i$ (ou seja vamos calcular as relações $\sim_0$, $\sim_1$ etc... até encontrar o primeiro i tal que $\sim_i = \sim_{i+1}$).

- Cálculo da classe de equivalência induzida por $\sim_0$. Vejamos então que estados reconhecem palavras de comprimento menor ou igual a 0 (ou seja ϵ).

Estados	ϵ
1	$\times$
2	$\times$
3	✓
4	$\times$
5	$\times$
6	$\times$
7	✓

  Esta tabela induz assim a partição $\{\{1, 2, 4, 5, 6\},\ \{3, 7\}\}$

- Cálculo da classe de equivalência induzida por $\sim_1$. Vejamos então que estados reconhecem palavras de comprimento menor ou igual a 1 (ou seja ϵ, a, b).

$Estados$	ϵ	a	b
1	$\times$	✓	$\times$
2	$\times$	$\times$	$\times$
3	✓	$\times$	$\times$
4	$\times$	$\times$	$\times$
5	$\times$	$\times$	✓
6	$\times$	$\times$	$\times$
7	✓	✓	$\times$

Esta tabela induz assim a partição $\{\{1\},\ \{2,4,6\},\ \{5\},\ \{3\},\ \{7\}\}$

A partição mudou, por isso ainda temos de considerar a iteração seguinte.

- Cálculo da classe de equivalência induzida por $\sim_2$. Vejamos então que estados reconhecem palavras de comprimento menor ou igual a 2 (ou seja $\epsilon, a, b, aa, ab, ba, bb$).

$Estados$	ϵ	a	b	aa	ab	ba	bb
1	$\times$	✓	$\times$	$-$	$-$	$-$	$-$
2	$\times$	$\times$	$\times$	$\times$	✓	$\times$	$\times$
3	✓	$\times$	$\times$	$-$	$-$	$-$	$-$
4	$\times$	$\times$	$\times$	$\times$	✓	$\times$	$\times$
5	$\times$	$\times$	✓	$-$	$-$	$-$	$-$
6	$\times$	$\times$	$\times$	$\times$	✓	$\times$	$\times$
7	✓	✓	$\times$	$-$	$-$	$-$	$-$

(O símbolo $-$ representa o facto de que não ser necessário explorar esta opção, devido às optimizações possíveis do algoritmo – ver aulas)

Esta tabela induz assim a partição $\{\{1\},\ \{2,4,6\},\ \{5\},\ \{3\},\ \{7\}\}$

A partição desta vez não mudou. Atingimos o ponto fixo. Terminamos. O autómato resultante é apresentado na figura 3.

□

Exercício 6 *(10 minutos) Considere o autómato A_2 da figura 4.*

Remova as ϵ-transições com base na utilização do algoritmo introduzido nas aulas.

Resposta:

O autómato original encontra-se representado na figura 5

Na figura 6 apresentamos destacadas as transições $e \xrightarrow{c} e'$ ($c \in \{a, b\}$) que serão alvo das atenções do algoritmo (de e' sai pelo menos uma transição ϵ).

Para cada uma destas 7 transições, juntamos as transições necessárias para o primeiro passo do algoritmo (ver figura 6).

Calculamos igualmente nesta figura os novos estados iniciais e finais.

Finalmente na figura 7 removemos as transições ϵ agora obsoletas.

□

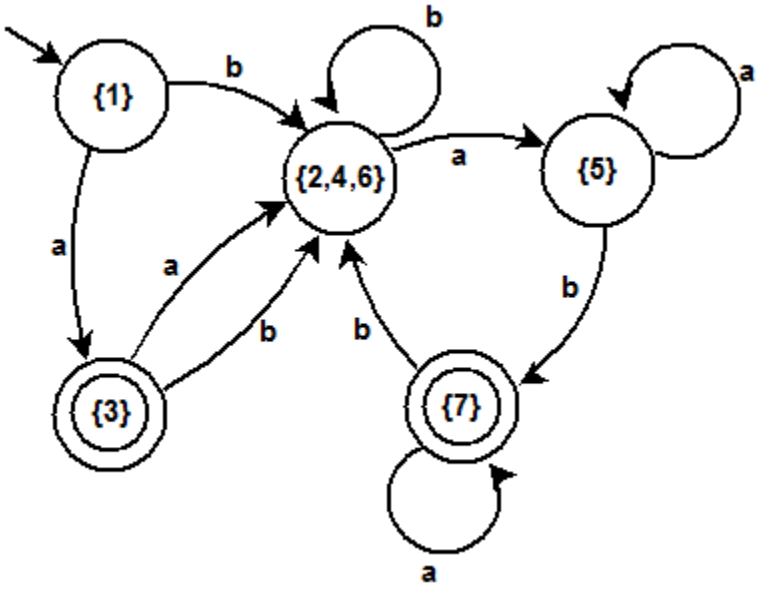

Figura 3: autómato resultante da minimização de A_1

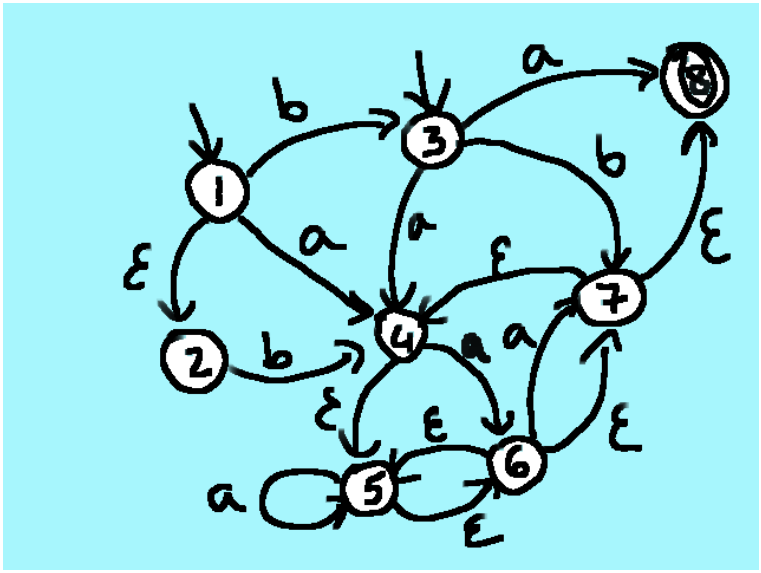

Figura 4: Autómato A_2

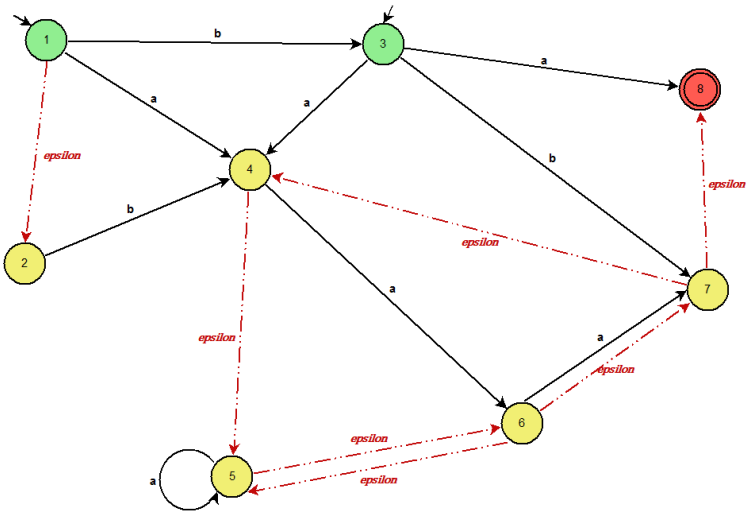

Figura 5: Remoção de transições ϵ - parte 1

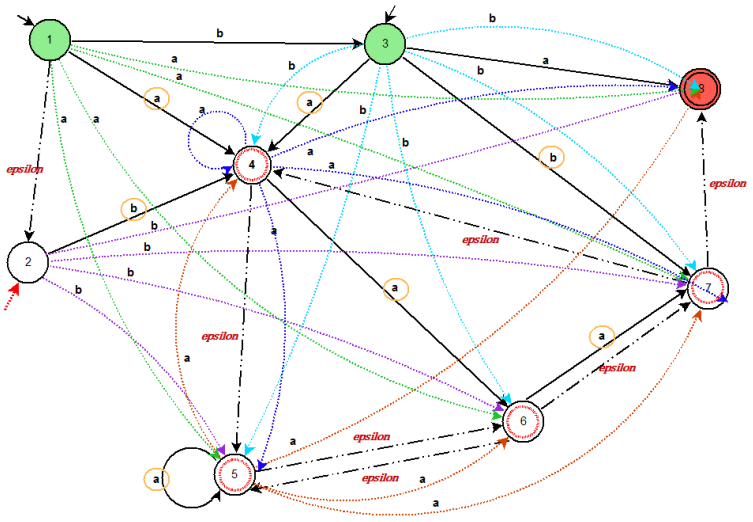

Figura 6: Remoção de transições ϵ - parte 2

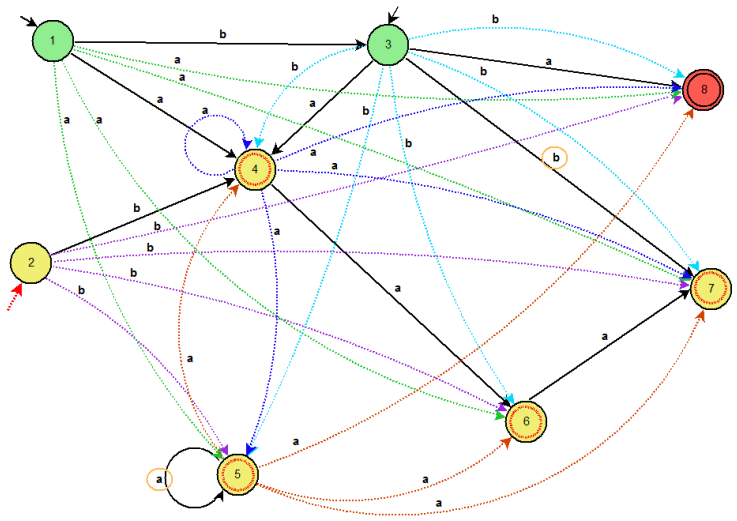

Figura 7: Remoção de transições ϵ - parte 3

Exercício 7 *(15 minutos) Dado o alfabeto* $\Sigma = \{a, b\}$*, demonstre que a linguagem* $\{w \mid w \in \Sigma^\star \ \wedge \ \sharp_a(w) < \sharp_b(w)\}$ *não é regular. Para tal relembra-se que* $\sharp_x(w)$ *é a função que devolve o numero de ocorrências do carácter* x *na palavra* w*.*

Resposta:

Como vimos nas aulas, há duas grandes técnicas para provar a não regularidade. A primeira técnica utiliza directamente o lema de bombeamento. A segunda tenta tirar proveito da estrutura da linguagem alvo e comparar a sua (não) regularidade com a de outras linguagens (conhecidas por serem ou não regulares). Esta comparação, obviamente, assenta numa utilização indirecta do lema de bombeamento (em última instância as linguagens de referências foram desmonstradas não regulares pelo uso do lema).

Neste exercício aqui vamos usar esta segunda abordagem.

Temos então $L = \{w \mid w \in \Sigma^\star \ \wedge \ \sharp_a(w) < \sharp_b(w)\}$.

Vamos proceder por contradição. Assumimos que L é regular. Então a intersecção de L com uma outra linguagem regular deverá ser igualmente regular (ver aulas). Por exemplo, se consideramos a linguagem (demonstrada regular nas aulas) $L' = a^*b^*$, então $L'' = L \cap L'$ deverá ser igualmente regular.

Que linguagem é L''?. É $\{a^n b^m \ \mid \ n < m\}$. Ora, esta linguagem não é regular. Por não ser, então L também não é regular.

Este raciocínio assenta no facto de L'' não ser regular. Como podemos afirmar tal facto? Basta considerar aqui uma utilização simples do lema de bombeamento.

Sabemos que, se admitirmos que L'' seja regular, existe um comprimento l a partir do qual as palavras exibem um padrão especial de repetição (ver definição do lema de bombeamento nos acetatos das aulas).

Se consideramos a palavra $p = \underbrace{aaa\ldots aaaa}_{l+1}\underbrace{bbbbb\ldots bbbbb}_{l+p \ (p>1)}$

Neste caso é facil ver (à luz dos detalhes do lema de bombeamento) que a única possibilidade de arranjar o padrão $p = xvy$ (tal que xv^*y pertença à linguagem L'') é x e v consituidos exclusivamente por ocorrências da letra a. Desta forma é sempre possível repetir os a's de v as vezes que se quiser, para além mesmo do numero de b's. Logo quebrando a regra da definição da linguagem L''.

Logo L'' não é regular.

QED.

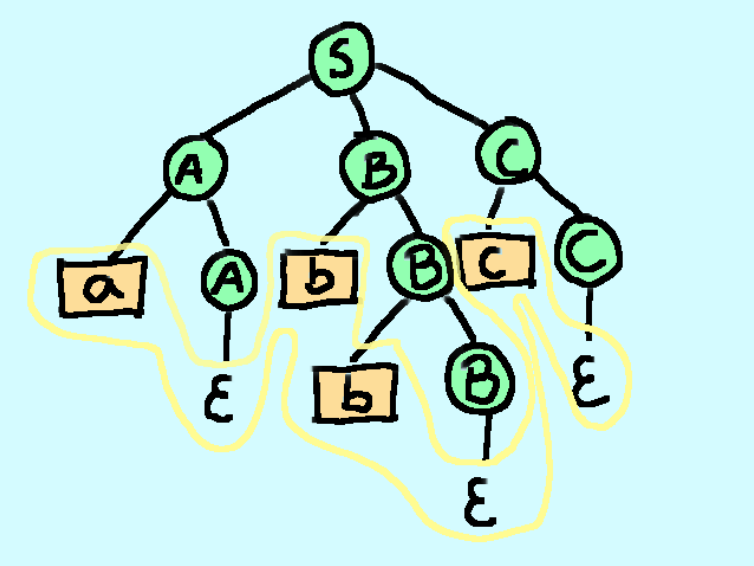

Figura 8: Arvore de derivação de *abbc*

□

4 Gramáticas

Exercício 8 *Considere a gramática cujas as produções são as seguintes:*

$$\begin{array}{lcl} S & \to & ABC \\ A & \to & aA \\ A & \to & \epsilon \\ B & \to & bB \\ B & \to & \epsilon \\ C & \to & cC \\ C & \to & \epsilon \end{array}$$

e a palavra $w = abbc$.

1. *(2 minutos) Dê uma derivação esquerda de* w.
2. *(2 minutos) Dê a árvore de derivação de* w.

Resposta:

Derivação esquerda de *abbc* :

$$S \Longrightarrow ABC \Longrightarrow aABC \Longrightarrow aBC \Longrightarrow abBC \Longrightarrow abbBC \Longrightarrow abbC \Longrightarrow abbcC \Longrightarrow abbc$$

Para a ávrvore de derivação, veja a figura 8.

□