Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação 2 de outubro de 2019

117536 - Projeto e Análise de Algoritmos

Prof. Flávio L. C. de Moura*

Projeto - 2019/2 (Turma B)

Introdução

A construção de programas corretos e eficientes é um ponto central na Ciência da Computação. Mas como garantir que um programa está correto? E como medir sua eficiência? Ao longo da primeira metade deste curso, vimos os formalismos teóricos necessários para medir a eficiência de um programa por meio da análise assintótica da função custo, parametrizada pelo tamanho n da entrada, que pode então ser classificada de acordo com algumas classes de complexidade. A correção de um programa, por sua vez, é estabelecida via uma série de propriedades que o programa deve satisfazer.

As provas em papel e lápis normalmente são suficientes para a análise de programas simples, mas podem esconder erros no caso de programas mais complexos. De fato, alguns exemplos famosos de erros envolvendo sistemas críticos incluem:

1. **Pentium FDIV**: Um erro na construção da unidade de ponto flutuante do processador Pentium da Intel causou um prejuízo de aproximadamente 500 milhões de dólares para a empresa que se viu forçada a substituir os processadores que já estavam no mercado em 1994.

2. **Therac-25**: Uma máquina de radioterapia controlada por computador causou a morte de pelo menos 6 pacientes entre 1985 e 1987 por overdose de radiação.

3. **Ariane 5**: Um foguete que custou aproximadamente 7 bilhões de dólares para ser construído explodiu no seu primeiro voo em 1996 devido ao reuso sem verificação apropriada de partes do código do seu predecessor.

Neste contexto, a utilização de métodos formais na construção de programas é cada vez mais comum:

*contato@flaviomoura.mat.br

1. A Intel e AMD utilizam assistentes de provas na verificação de processadores.

2. A Microsoft utiliza métodos formais na verificação de programas e drivers.

3. CompCert: Compilador C verificado em Coq.

4. A AirBus e a NASA utilizam assistentes de provas na verificação de programas de aviação.

5. A Toyota utiliza métodos formais na verificação de sistemas híbridos de controle.

6. A linha 14 do metrô de Paris é totalmente controlada por um programa de computador verificado formalmente.

Apesar da utilização cada vez mais frequente de métodos formais na construção de programas, esta não é uma tarefa fácil. Intuitivamente, um programa é correto se faz exatamente o que se propõe em tempo e espaço finitos. Por exemplo, um programa P que ordena listas de números naturais em ordem crescente é correto se, para qualquer lista l dada, o resultado retornado por P após um tempo finito é uma lista contendo exatamente os elementos de l ordenados de forma crescente.

Descrição do projeto

A proposta deste projeto é formalizar a complexidade de tempo no pior caso, e se possível, a correção de um algoritmo de sua preferência no assistente de provas PVS (`http://pvs.csl.sri.com`). Para os alunos que não têm experiência prévia com o PVS, sugerimos a formalização do algoritmo *bubble sort* ou *merge sort* a partir da formalização disponibilizada no GitHub como detalhado posteriormente.

Utilizaremos como exemplo, a formalização da correção e da complexidade de tempo de uma versão recursiva de *insertion sort*. Este exemplo será desenvolvido em detalhes durante as aulas. Neste caso, considerando que a lista vazia já está ordenada por definição, para ordenarmos listas não nulas, precisamos inserir o primeiro elemento de `l`, denotado por `car(l)`, na versão ordenada da cauda de `l`, denotada por `cdr(l)`:

```
insertion_sort(l): RECURSIVE list[nat] =
IF null?(l) THEN null ELSE
insert(car(l), insertion_sort(cdr(l)))
```

```
ENDIF
MEASURE length(l)
```

onde a função `insert` é definida por:

```
insert (x, l): RECURSIVE list[nat] =
IF null?(l) THEN cons(x,null)
ELSIF x<= car(l) THEN cons(x,l)
ELSE cons(car(l), insert(x,cdr(l)))
ENDIF
MEASURE length(l)
```

A correção de *insertion sort*

Observe que a função `insert` é construída de forma a preservar a ordenação após a inserção. Este comportamento de `insert` pode ser representado por meio do seguinte lema:

```
insert_in_sorted_preserves_sort : LEMMA
FORALL (l: list[nat], x: nat):
   sorted?(l) IMPLIES sorted?(insert(x,l))
```

onde `sorted?` é o predicado que captura o fato de uma lista estar ordenada:

```
sorted?(l:list[nat]) : RECURSIVE boolean =
   CASES l OF
 null: TRUE,
 cons(h,tl): CASES tl OF
     null: TRUE,
     cons(hh,ttl): (h <= hh) AND sorted?(tl)
     ENDCASES
   ENDCASES
   MEASURE length(l)
```

Parte da prova da correção deste algoritmo consiste em provar que `insertion_sort` gera uma lista ordenada para qualquer lista dada como entrada:

```
insertion_sort_sorts: LEMMA
     FORALL (l:list[nat]): sorted?(insertion_sort(l))
```

Este lema é provado por indução na estrutura da lista `l`, via o comando `(induct "l")`. A prova é dividida em dois casos:

1. No primeiro caso, a lista `l` é vazia e o resultado é imediato, dadas as definições de `insertion_sort` e `sorted?`.

2. No segundo caso, a lista `l` é não vazia, onde o primeiro elemento é denotado por `car(l)`, e a cauda, isto é, todos os outros elementos exceto o primeiro, é denotada por `cdr(l)`. Temos por hipótese de indução que `sorted?(insertion_sort(cdr(l)))`, e precisamos mostrar que `sorted?(insertion_sort(cons(car(l),cdr(l))))`. Neste momento, podemos aplicar a definição de `insertion_sort`, via o comando `(expand "insertion_sort")`, obtendo
   `sorted?(insert(car(l),insertion_sort(cdr(l))))`
   como novo objetivo a ser provado, e concluímos com a aplicação do lema `insert_in_sorted_preserves_sort`, via comando
   `(lemma "insert_in_sorted_preserves_sort")`.

A segunda parte da prova da correção consiste em provar que `insertion_sort(l)` gera como saída uma permutação de `l`, mas esta etapa não será descrita aqui.

Análise assintótica do pior caso de *Insertion Sort*

Nesta seção veremos os passos necessários para provar que a complexidade de tempo de `insertion_sort`, no pior caso, é quadrática no tamanho da entrada. Para isto, utilizaremos a notação assintótica estudada no curso. Sabemos que, se $f(n)$ e $g(n)$ são funções dos naturais nos reais não-negativos, então dizemos que $g(n) = O(f(n))$, se existirem constantes positivas c e n_0 tais que

$$g(n) \leq c.f(n), \forall n \geq n_0. \tag{1}$$

Em outras palavras, o conjunto $O(f(n))$ é definido por $\{g(n)$: existem constantes positivas c e n_0 tais que $g(n) \leq c.f(n), \forall n \geq n_0\}$. A correspondente definição em PVS pode ser dada como a seguir:

```
Omicron( f : [nat -> nonneg_real ]) :
 setof [[nat -> nonneg_real]] =
 { g : [nat -> nonneg_real] |
   EXISTS ( c2 : nonneg_real, n0 : nat ) :
     FORALL (n : nat |  n >= n0 ) : g(n) <= c2 * f(n)}
```

O tamanho da entrada é dado pelo número de elementos da lista a ser ordenada. Assim, queremos mostrar um lema da forma:

```
insertion_sort_is_quadratic: LEMMA
 member(LAMBDA(n:nat): T_insertion_sort(l),
 Omicron(LAMBDA(n:nat):length(l)^2 ))
```

onde `T_insertion_sort(l)` computa o número de comparações feitas por `insertion_sort(l)`. Como construir a função `T_insertion_sort(l)`? Adicionaremos um contador à função `insertion_sort` como a seguir:

```
cinsertion_sort(l: list[nat]): RECURSIVE [list[nat],nat] =
IF null?(l) THEN (null,0)
  ELSE cinsert(car(l), cinsertion_sort(cdr(l)))
ENDIF
MEASURE length(l)
```

A função `insertion_sort` com um contador, aqui denotada por `cinsertion_sort`, recebe uma lista de naturais como argumento, e retorna um par cujo primeiro elemento é a versão ordenada da lista original, e o segundo elemento é um natural que corresponde ao número de comparações realizadas para ordenar a lista dada. Desta forma, `T_insertion_sort(l)` é dada por `cinsertion_sort(l)‘2`. Assim, se a lista de entrada é vazia então a saída é o par `(null,0)`, ou seja, foram realizadas 0 comparações para ordenar a lista de entrada. Quando a lista `l` é não vazia, então `cinsertion_sort` vai inserir o primeiro elemento de `l` na versão ordenada da cauda via a função `cinsert` que corresponde à função de inserção com um contador:

```
cinsert (x, lc): RECURSIVE [list[nat],nat] =
IF null?(lc‘1) THEN (cons(x,lc‘1),lc‘2)
ELSIF x <= car(lc‘1) THEN (cons(x,lc‘1), lc‘2 + 1)
ELSE LET lcaux = cinsert(x,(cdr(lc‘1),lc‘2)) IN
(cons(car(lc‘1), lcaux‘1), lc‘2 + 1)
ENDIF
MEASURE length(lc‘1)
```

A função `cinsert` recebe uma par contendo um número natural `x` e um par `lc` contendo uma lista e um natural, respectivamente representados por `lc‘1` e `lc‘2`, e retorna um par contendo a nova lista obtida após a inserção do novo elemento `x`, e o contador `lc‘2` que corresponde ao número de comparações realizadas até então. Desta forma, se `lc‘1` é a lista vazia então obtemos `(cons(x,lc‘1),lc‘2)`, isto é, a lista unitária contendo apenas o elemento `x`, e o número `lc‘2` de comparações realizadas. Se `lc‘1` não for vazia, então precisamos comparar `x` com o primeiro elemento da lista. Se `x` for menor ou igual a `car(lc‘1)` então retornamos o par `(cons(x,lc‘1), lc‘2 + 1)`, *i.e.* inserimos `x` antes da primeira posição de `lc‘1`, e incrementamos o contador em 1. Caso contrário, ou seja, quando `x` é estritamente maior do que `car(lc‘1)` então denotamos por `lcaux` o par contendo a lista resultante da inserção de `x` em `cdr(lc‘1)` e o número de comparações feitas até este ponto. O resultado neste subcaso é dado por `(cons(car(lc‘1), lcaux‘1), lc‘2 + 1)`, ou seja, a lista resultante tem `car(lc‘1)` como primeiro elemento, e cauda `lcaux‘1`, e o número de comparações é incrementado em 1.

Para provarmos o lema `insertion_sort_is_quadratic` precisamos estabelecer uma cota superior para o número de comparações acumuladas no contador:

```
cinsertion_bound_on_comparisons: LEMMA
cinsertion_sort(l)‘2 <= ((length(l))^2 + length(l))/2
```

Outros lemas adicionais podem ser necessários para a conclusão da prova de `insertion_sort_is_quadratic`, mas faremos este trabalho durante as próximas aulas. Também é importante mostrar a equivalência entre as funções `insertion_sort` (resp. `insert`) e `cinsertion_sort` (resp. `cinsert`), por exemplo.

Etapas do projeto

O trabalho, que possui duas etapas, poderá ser realizado individualmente ou em duplas. Os grupos deverão ser formados no GitHub a partir do link
`https://classroom.github.com/g/8yyzdBL0`
Os grupos podem ser formados até *[2019-10-08 Ter 23:00]*.

Formalização do algoritmo (Peso 6.0)

Nesta etapa os grupos devem construir a formalização da correção e complexidade temporal do algoritmo selecionado. Os arquivos PVS relacionados com a formalização devem estar disponíveis no repositório do GitHub até *[2019-12-05 Qui 23:59]*.

Relatório (Peso 4.0)

Cada grupo de trabalho devera entregar um relatório inédito em formato pdf, preferencialmente em LaTeXaté o dia *[2019-12-05 Qui 23:59]*. O arquivo pdf do relatório também deverá estar no repositório GitHub.

Referências

[AdM17] M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs*. Undergraduate Topics in Computer Science. Springer, 2017.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms | Introduction to Design and Analysis*. Addison-Wesley, 1999.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, second edition, 2001.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume Volume 3 of The Art of Computer Programming. Reading, Massachusetts: Addison-Wesley, 1973. Also, 2nd edition, 1998.

[Lev12] A. Levitin. *Introduction to the Design & Analysis of Algorithms*. Pearson, third edition, 2012.