Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

117536 - Projeto e Análise de Algoritmos
Turma: B

Relatório sobre **Complexidade do Bubblesort**

Camila F. T. Pontes - 15/0156120
Diogo C. Ferreira - 11/0027931

6 de dezembro de 2019

1 Introdução

O problema de ordenação de sequências numéricas surge frequentemente em aplicações computacionais. Este problema pode ser definido formalmente da seguinte maneira [1]:

- **Entrada:** uma sequência de n números, $a_1, a_2, ..., a_n$
- **Saída:** uma permutação (reordenação) da sequência de entrada, $a'_1, a'_2, ..., a'_n$ tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$

Existem diversos algoritmos de ordenação que resolvem o problema apresentado acima, *e.g.* *insertion sort*, *selection sort*, *merge sort*, *quicksort*, dentre outros. Neste trabalho, vamos analisar um dos algoritmos de ordenação mais simples, o *bubblesort*. A ideia geral do *bubblesort* é percorrer diversas vezes o vetor de entrada e, a cada passagem, mover para o final da porção ainda não ordenada o maior elemento. Uma implementação recursiva desse algoritmo é apresentada a seguir (Algoritmo 1).

Algorithm 1 Implementação recursiva do *bubblesort*

```
function BUBBLESORT(int array A, int n)
    if n = 1 then
        return
    end if
    for i ← 0 to n − 1 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
        end if
    end for
    bubblesort(A,n − 1)
end function
```

Uma das formas de comparar o desempenho do *bubblesort* com o desempenho de outros algoritmos de ordenação é através de uma análise de complexidade. A complexidade do *bubblesort* é de ordem quadrática ($O(n^2)$), *i.e.*, no pior caso, são feitas n^2 trocas durante a ordenação, onde n é o número de elementos do vetor de entrada. Neste trabalho, um assistente de prova será utilizado para provar a complexidade do *bubblesort*. A prova será realizada utilizando o *Prototype Verification System* (PVS) [2].

Os principais objetivos deste trabalho são:

- Implementar uma versão recursiva do *bubblesort* com um contador para o número de trocas realizadas durante a ordenação;
- Provar a complexidade assintótica do *bubblesort* utilizando o PVS.

2 Apresentação do Problema

O problema em questão consiste na análise da complexidade assintótica do algoritmo Bubblesort. A resolução proposta foi decomposta em três partes:

na elaboração de funções auxiliares que possibilitem a realização da análise; na elaboração e prova de lemas que dizem respeito a análise da complexidade do algoritmo; e por fim, na elaboração e prova de lemas que garantam a equivalência entre as funções auxiliares e as funções originalmente implementadas.

2.1 Funções auxiliares

Para a realização da análise assintótica do Bubblesort, inicialmente foram definidas três funções auxiliares com o objetivo de rastrear as contagens do número total de comparações realizadas pelo algoritmo após retornar a lista ordenada.

- `bubbling_count`: recebe uma lista l, um natural n (equivalentes aos parâmetros da função `bubbling` original) e um contador *count*, que é incrementando quando alguma comparação for realizada. Seu valor de retorno é o par $(l, count)$ com a lista e o contador devidamente atualizados.

- `bubblesort_aux_count`: recebe uma lista l, um natural n (equivalentes aos parâmetros da função `bubblesort_aux` original) e um contador *count*, que é passado para a função `bubbling_count` chamada internamente. Seu valor de retorno é o par $(l, count)$ com a lista e o contador devidamente atualizados.

- `bubblesort_count`: recebe uma lista l, um natural n, equivalentes aos parâmetros da função `bubblesort` original. Ela chama `bubblesort_aux_count` da mesma forma que `bubblesort` chama `bubblesort_aux`, mas com o parâmetro do contador iniciando em 0. Seu valor de retorno é o par $(l, count)$ com a lista ordenada e o contador atualizados com o número total de comparações realizadas pelo algoritmo.

2.2 Lemas

Como o objetivo é a análise da complexidade assintótica do algoritmo Bubblesort, dividimos a análise com base nas funções que compõem a implementação. Para cada uma das unções auxiliares elaboramos ao menos dois lemas: um para analisar seu comportamento assintótico e um segundo para garantir sua equivalência com a função correspondente original.

Lemas utilizados na prova da complexidade de `bubbling_count`:

1. `bubbling_counts_n`: afirma que `bubbling_count` realiza exatamente n comparações, onde n é o tamanho da lista de entrada, e que, portanto, sua complexidade é linear

2. `bubbling_equiv`: afirma que `bubbling` e `bubbling_count` são equivalentes

3. `bubbling_length`: afirma que a função `bubbling_count` não altera o tamanho da lista de entrada

Lemas utilizados na prova da complexidade de `bubblesort_aux_count`:

4. `bubaux_counts_n2`: afirma que `bubblesort_aux_count` realiza exatamente $n(n+1)/2$ comparações, onde n é o tamanho da lista de entrada, e que, portanto, sua complexidade é quadrática

5. `bubblesort_aux_equiv`: afirma que `bubblesort_aux` e `bubblesort_aux_count` são equivalentes

Lemas utilizados na prova da complexidade de `bubblesort_count`:

6. `bubblesort_counts_n2`: afirma que `bubblesort_count` realiza exatamente $n(n-1)/2$ comparações, onde n é o tamanho da lista de entrada, e que, portanto, sua complexidade é quadrática

7. `bubblesort_equiv`: afirma que `bubblesort` e `bubblesort_count` são equivalentes

2.3 Análise assintótica

As provas podem ser verificadas por completo através do PVS a partir dos arquivos que acompanham este projeto. O arquivo `bubblesort2.pvs` contem a implementação das funções originais fornecidas pelo professor da disciplina, bem como a implementação das funções auxiliares e dos lemas elaborados pelo grupo. Apresentaremos a seguir um detalhamento dos pontos que consideramos fundamentais na realização das provas. Adotaremos a seguinte notação para as fórmulas nesta Seção: se l é uma lista, $|l|$ indica o seu comprimento. O i-ésimo elemento de l é dado por l_i e uma lista $l_i : l'_i$ possui o elemento l_i na cabeça e a lista l'_i na cauda. Analogamente, P_i denota o i-ésimo elemento de um par ordenado $P = (a, b)$ e $f(x)_i$ o i-ésimo elemento de uma função $f(x) = (a, b)$ que retorna um par.

2.3.1 *Bubbling*

Lema 1: `bubbling_counts_n`: seja l uma lista de números naturais, e lembrando que `bubbling_count` é uma função que retorna um par (l, c), onde c denota o número de comparações realizadas ao todo na chamada

$$\forall_{l,n,c} bubbling\_count(l, c, n)_2 = c + n \qquad , n < |l|, c \in \mathbb{N}$$

Estratégia da prova: indução forte sobre $|l|$. A função `bubbling_count` faz chamadas recursivas sobre l, e seria correspondente às linhas 5 a 9 do Algoritmo 1. Cada chamada é feita sobre uma lista menor mas a lista não é dividida exatamente conforme a definição recursiva de l. Portanto precisamos de uma medida alternativa que seja relacionada com a estrutura sobre a qual queremos realizar a indução (Figura 1).

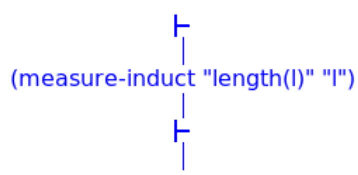

Figura 1: A prova do lema de complexidade de `bubbling_count` foi realizada por indução forte sobre $|l|$.

Após a expansão da definição da definição de `bubbling_count`, precisamos provar 3 casos: o caso em que $n = 0$ (trivial, pois $c = c + n = c + 0 = c$), o caso em que $l_i > l_{i+1}$ e o caso em que $l_i \leq l_{i+1}$. A prova destes dois últimos casos é muito semelhante, e varia essencialmente em como a hipótese de indução (*h.i.*) será instanciada (Figura 2). No primeiro caso, a próxima chamada de `bubbling_count` será realizada sobre uma lista com a forma $l_i : l_{i+2} : l'_{i+2}$, portanto essa deve ser a instanciação da *h.i.*.

```
{-1}  car(x) > car(cdr(x))
[-2]  (H.I.)
  |-------
{1}   1 + bubbling_count(cons(car(x), cdr(cdr(x))), c, n - 1)`2 = c + n
{2}   n = 0

Rule? (inst -2 "cons(car(x), cdr(cdr(x)))")
```

No segundo caso, a próxima chamada `bubbling_count` será propriamente sobre a cauda de l_i, portanto a $h.i.$é instanciada como l'_i:

```
[-1]  (H.I.)
  |-------
{1}   car(x) > car(cdr(x))
{2}   1 + bubbling_count(cdr(x), c, n - 1)`2 = c + n
{3}   n = 0

Rule? (inst -1 "cdr(x)")
```

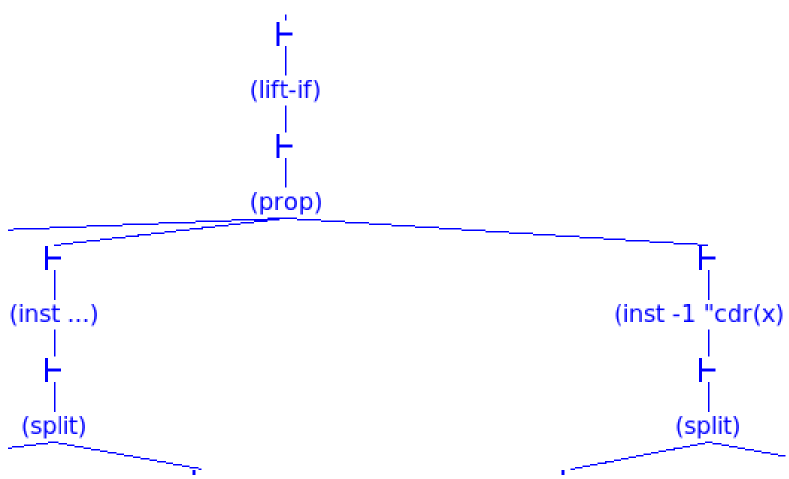

Figura 2: Casos da prova após a expansão da definição de `bubbling_count`. O ramo omitido mais à esquerda é o caso base.

O restante da prova consiste em algumas instanciações adicionais de c e/ou n, na expansão da definição de `length` e inferências sobre o tipo de

n. Felizmente existe a restrição de que $n < |l|$, e o comando `(typepred n)` realiza essa inferência. O uso de `(grind)` após `(typepred n)` foi basicamente um atalho para `(expand list2finseq)` seguido de `(assert)`, ou de alguma resolução dos casos distintos de `length`, que surge a partir da definição de `list2finseq` (Figura 3).

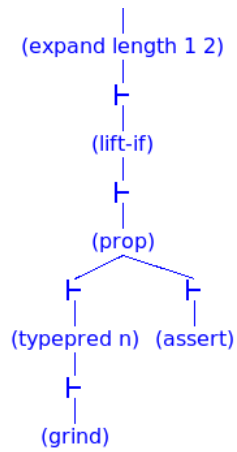

Figura 3: Para finalizar os ramos, foi necessário expandir a definição de `length` e realizar uma inferência sobre o tipo de n.

Dizemos então que `bubbling_count` tem complexidade linear, $O(n)$. Como $n < |l|$ por definição, então a função também é $O(|l|)$. Resta então mostrar que o mesmo se aplica a função original, `bubbling`, e isto foi feito mostrando que `bubbling` e `bubbling_count` são equivalentes:

Lema 2: `bubbling_equiv`:

$$\forall_{l,n,c} bubbling(l,n)_1 = bubbling\_count(l,c,n)_1 \qquad , n < |l|, c \in \mathbb{N}$$

Estratégia da prova: indução forte sobre $|l|$, pelo mesmo motivo que no lema 1 já que a prova se refere à mesma função, `bubbling_count`, que divide a lista da mesma forma que `bubbling`. A diferença principal entre prova deste lema e a do lema 1 é que é necessário expandir as definições de ambas as funções, `bubbling` e `bubbling_count` antes de instanciar a *h.i.*. As instanciações da *h.i.*foram realizdas como na prova do lema 1 (Figura 4).

Figura 4: A prova do lema de equivalência entre `bubbling` e `bubbling_count` requer a expansão de ambas as funções, mas segue de forma semelhante ao lema de complexidade.

Lema 3: `bubbling_length`:

$$\forall_{l,n,c} |bubbling\_count(l, c, n)_1| = |l| \qquad , n < |l|, c \in \mathbb{N}$$

Estratégia da prova: indução forte sobre $|l|$, pelo mesmo motivo que no lema 1 já que a prova se refere à mesma função, `bubbling_count`. Este lema não está diretamente relacionado com a analise da complexidade ou com a equivalência entre as funções. Contudo, ele foi utilizado em alguns pontos das provas subsequentes e, por se tratar de um lema relativamente longo de ser provado, decidimos enunciá-lo separadamente. O objetivo deste lema é mostrar que `bubbling_count` retorna uma lista do mesmo tamanho que a lista de entrada, e a prova também consiste em mostrar que isso é verdade para cada um dos casos da função `bubbling_count`.

2.3.2 *Bubblesort_aux*

Lema 4: `bubaux_counts_n2`:

$$\forall_{l,n,c} bubblesort\_aux\_count(l, c, n)_2 = c + \frac{n^2 + n}{2} \qquad , n < |l|, c \in \mathbb{N}$$

Estratégia da prova: indução forte sobre $|l|+n$. A função `bubblesort_aux_count` é uma função que faz chamadas recursivas sobre l, e seria correspondente ao próprio Algoritmo 1 menos as linhas 5 a 9 (que correspondem melhor com a função `bubbling`). A principal questão aqui é que $|l|$ se mantém constante ao longo das chamadas de `bubblesort_aux_count`, portanto uma indução sobre $|l|$ não geraria uma *h.i.*que pudesse ser instanciada de maneira apropriada. Uma segunda opção seria realizar a indução sobre n, uma vez que ele é o parâmetro que varia ao longo das chamadas recursivas da função. No entanto, como n é dependente de $|l|$, foi necessário que a indução fosse realizada sobre ambos os parâmetros simultaneamente, daí a escolha de $|l| + n$:

```
  |-------
{1}   FORALL (l: list[nat], n: below[list2finseq(l)`length]) (c: nat):
```

```
    bubblesort_aux_count(l, c, n)`2 = c + ((n ^ 2 + n) / 2)

Rule? (measure-induct "length(l) + n" ("l" "n"))
```

A expansão da definição da definição de `bubbling_count` após a escolha da estratégia de prova, resulta em dois casos: o caso base em que $n = 0$ é trivial, pois consiste em provar que $c = c + \frac{n^2+n}{2}$. No segundo caso, precisamos provar o sequente:

```
[-1]  (H.I.)
  |-------
{1}   x_2 = 0
{2}   bubblesort_aux_count(bubbling_count(x_1, c, x_2)`1,
                           bubbling_count(x_1, c, x_2)`2, x_2 - 1)`2
       = ((x_2 ^ 2 + x_2) / 2) + c

Rule? (inst -1 "bubbling_count(x_1, c, x_2)`1" "x_2-1")
```

O que permite a instanciação da hipótese de indução da seguinte forma:

$$
\begin{aligned}
bubblesort\_aux\_count(l, c, n)_2 = c + ((n^2 + n)/2) \qquad\qquad & (h.i.) \\
l = bubbling\_count(x_1, c, x_2)_1 & \\
c = c & \\
n = x_2 - 1 &
\end{aligned}
$$

A partir deste ponto, a prova novamente se ramifica em duas partes devido a estratégias ser indução sobre $|l| + n$. No primeiro ramo, que decorre do do parâmetro n, podemos chamar o lema 1 para mostrar que as chamadas de `bubbling_count` incrementam n de forma linear (Figura 5, ramo da esquerda). O segundo ramo decorre consiste em garantir que estamos instanciando a *h.i.*com valores estritamente menores do que aqueles no consequente, e prova disso utiliza o resultado do lema 3 que mostra que $|bubbling\_count(l, c, n)| + n - 1 < |l| + n$, com os valores devidamente instanciados (Figura 5, ramo da direita).

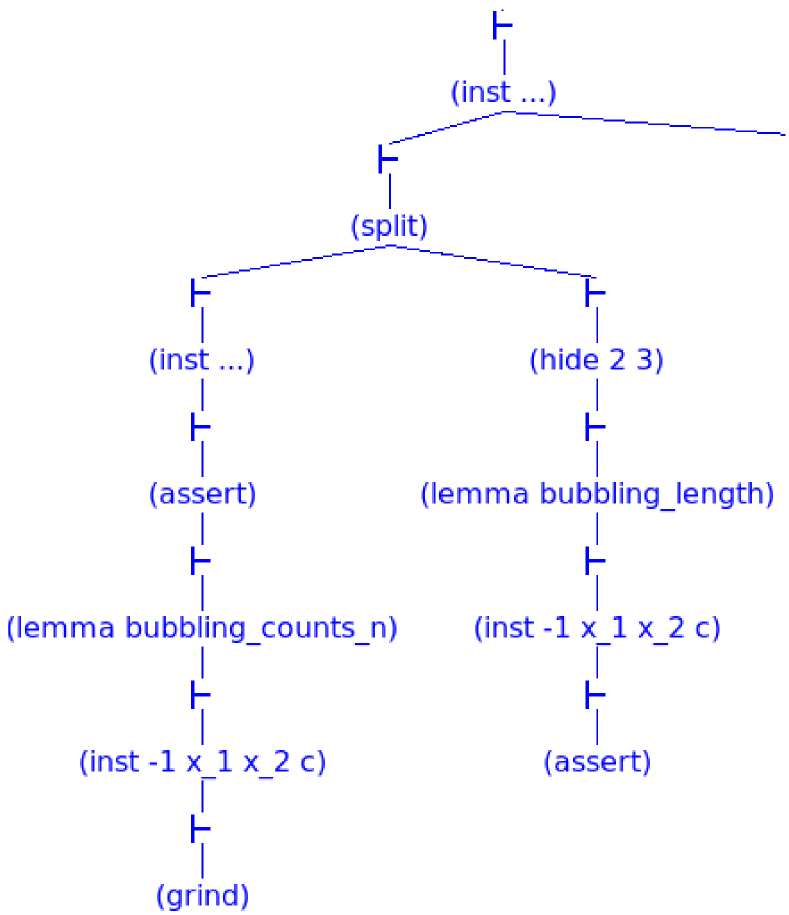

Figura 5: Para finalizar análise da complexidade de `bubblesort_aux` usamos os lemas 1 e 3 provados anteriormente. O ramo oculto ainda mais à direita se refere ao caso base.

Com isto, temos que a função `bubblesort_aux_count` realiza $\frac{n^2+n}{2}$ comparações e portanto é $O(n)$ Da mesma forma que com `bubbling`, pela restrição de $n < |l|$, então também é $O(|l|)$. Aqui ainda precisamos mostrar a equivalência entre os `bubblesort_aux` com e sem contador, conforme realizado a partir do lema a seguir.

Lema 5: `bubblesort_aux_equiv`:

$$\forall_{l,n,c} bubblesort\_aux(l,c,n)_1 = bubblesort\_aux\_count(l,c,n)_1 \qquad , n < |l|, c \in \mathbb{N}$$

Estratégia da prova: indução forte sobre $|l|+n$, pela mesma razão que no lema 4, já também se refere à função `bubblesort_aux_count` e `bubblesort_aux` apresenta as mesmas características.

A prova deste lema também requer a expansão da definição de ambas as funções, `bubblesort_aux_count` e `bubblesort_aux`. Já que essas elas, chamam suas respectivas funções `bubbling`, podemos instanciar a *h.i.*com a chamada interna de `bubbling` de cada uma delas, e completar a prova a partir do lema 2. Da mesma forma que no lema de complexidade 4, esta prova também precisa garantir que a instanciação da *h.i.*foi realizada de maneira adequada, o que também pode ser mostrado usando o resultado do lema 3 sobre o tamanho de l (Figura 6).

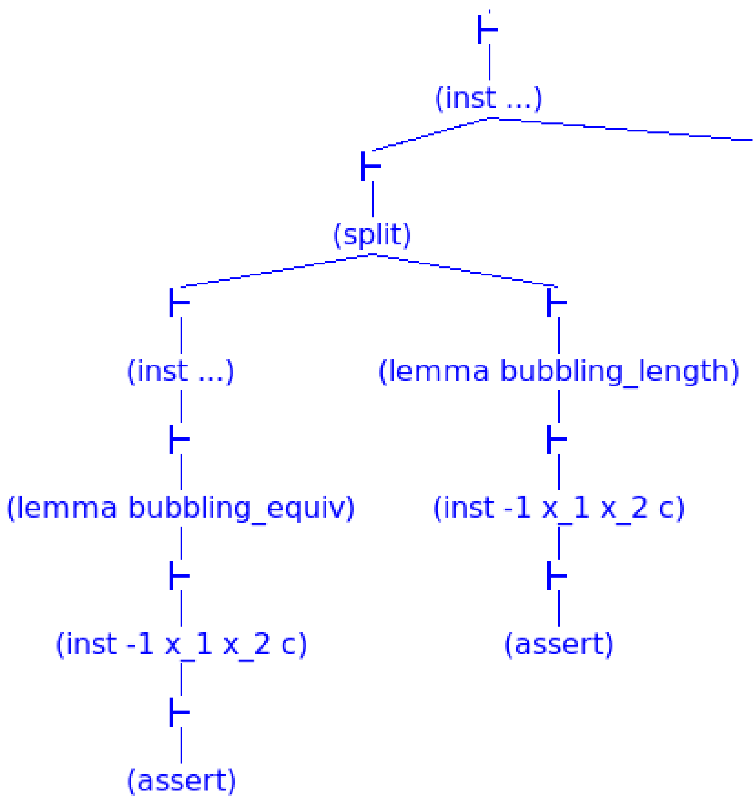

Figura 6: Para finalizar análise de equivalência `bubblesort_aux` entre `bubblesort_aux_count` usamos os lemas 2 e 3 provados anteriormente. O ramo oculto mais à direita se refere ao caso base.

2.3.3 *Bubblesort*

Lema 6: `bubblesort_counts_n2`:

$$\forall_l bubblesort\_count(l)_2 = \frac{|l|^2 - |l|}{2}$$

Estratégia da prova: direta, a partir da aplicação do lema 4. O objetivo da função `bubblesort_count` é encapsular `bubblesort_aux_count`, garantindo que esta seja chamada com os parâmetros corretos. A prova deste lema consiste na expansão da definição de `bubblesort_count` após instanciar l para uma lista qualquer. Nos deparamos então com os dois casos presentes na definição de `bubblesort_count`: o primeiro caso, em que a lista é vazia, é trivial, pois $\frac{|l|^2-|l|}{2} = 0$. No segundo caso, podemos evocar o lema 4 e instanciá-lo adequadamente pois sabemos exatamente os valores de c e n que serão passados para `bubblesort_aux_count`, e como eles se relacionam com $|l|$. O comando `(grind)` utilizado nos ramos desta prova foram utilizados para realizar as expansões da definição de `length` e fazer as simplificações algébricas apropriadas (Figura 7).

Lema 7: `bubblesort_equiv`:

$$\forall_l bubblesort(l)_1 = bubblesort\_count(l)_1 = \frac{|l|^2 - |l|}{2}$$

Estratégia da prova: direta, a partir da aplicação do lema 5. De forma semelhante ao lema anterior, a prova foi feita a partir da aplicação do lema 5. A expansão das definições de ambas `bubblesort` e `bubblesort_count` revela a chamada das funções `bubblesort_aux` e `bubblesort_aux_count` e a igualdade é estabelecida por meio da instanciação do lema. Novamente, o comando `(grind)` foi utilizado para fazer as simplificações algébricas apropriadas e completar a prova (Figura 8).

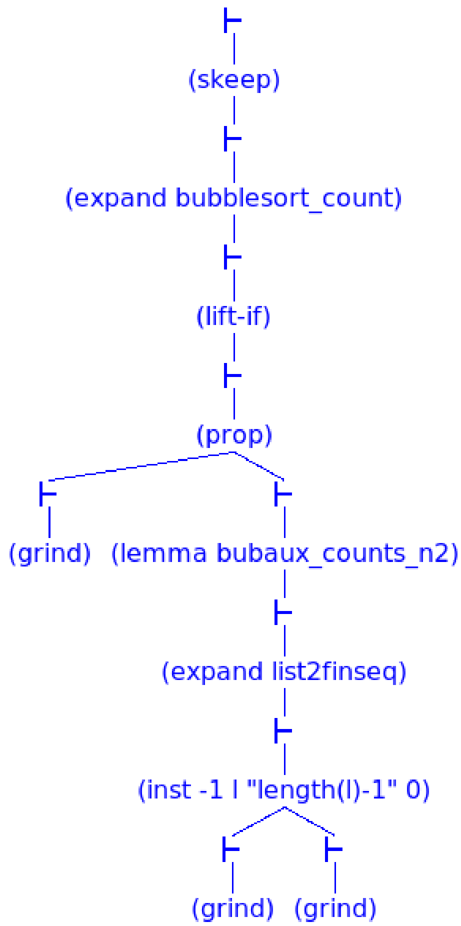

Figura 7: A prova do lema de complexidade do `bubblesort_count` é direta a partir da aplicação do lema 4, pois sabemos os valores exatos de n e c, para uma dada lista l, que serão passados para `bubblesort_aux`.

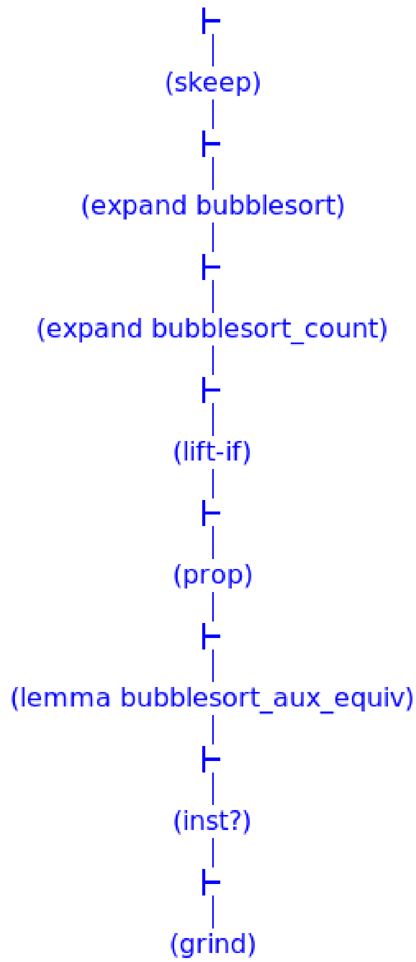

Figura 8: A prova de equivalência entre `bubblesort` e `bubblesort_count` é direta a partir da aplicação do lema 5.

2.4 TCCs

Ao final das provas dos lemas elaborados ainda restaram alguns TCCs (*Type-Correctness Conditions*) que não puderam ser verificados automaticamente pelo PVS. Isto foi parcialmente resolvido após uma mudança na ordem em que os lemas foram enunciados no arquivo de entrada. Faltou, contudo, realizar a prova de TCCs relacionados com a função `bubbling`, e estes precisaram ser provados manualmente, pois necessitaram de uma prova um pouco mais elaborada, embora ainda relativamente curta. A título de exemplo, apresentaremos o `bubbling_TCC3`, mas os outros consistiam de sequentes semelhantes relacionados ao comprimento de partes das listas.

```
  |-------
{1}   FORALL (l: list[nat], (n: below[list2finseq[nat](l)`length])):
        car(l) > car(cdr(l)) AND NOT n = 0 IMPLIES
         n - 1 >= 0 AND n - 1 < list2finseq[nat]
               (cons[nat](car[nat](l), cdr[nat](cdr[nat](l))))`length
```

A prova deste sequente foi direta e se deu principalmente pela expansão de `list2finseq`, e posteriormente a manipulação das definições de `length` que surgiram (Figura 9).

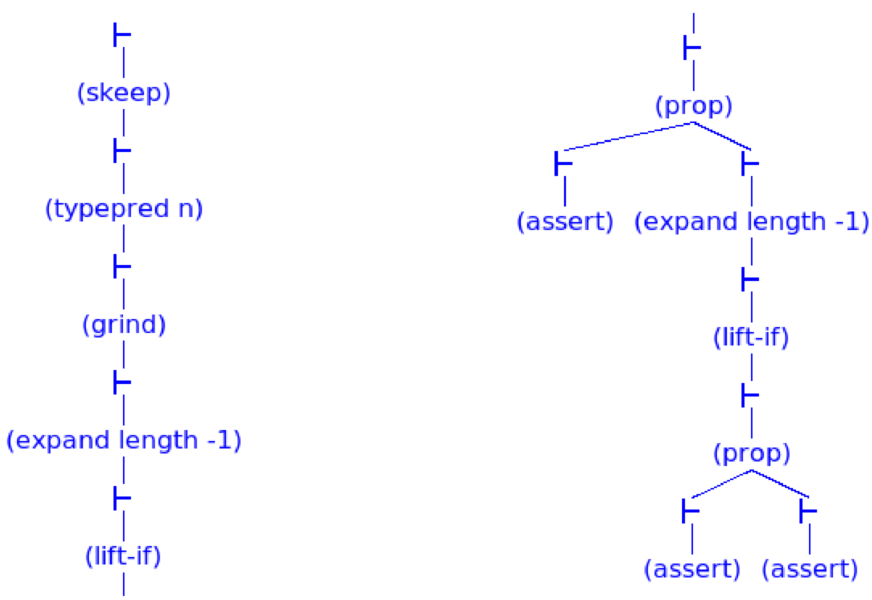

Figura 9: Prova do TCC `bubbling_TCC3` que não pode ser completado automaticamente pelo PVS.

3 Conclusão

Neste trabalho, foi realizada uma prova assistida da complexidade quadrática do algoritmo de ordenação *bubblesort* utilizando o PVS. A prova foi realizada analisando uma contagem do número de comparações realizadas por uma implementação recursiva do algoritmo *bubblesort*, através da função implementada `bubblesort_count`, que depende da prova de complexidade das funções internas que são chamadas por ela, *bubblesort_aux* e *bubbling*.

Notamos que, embora as provas destas duas seja em função do parâmetro n, ao serem chamadas por *bubblesort* sabemos exatamente o valor inicial de n que será passado para elas como argumento. Por este motivo, foi possível concluir a prova final da complexidade de *bubblesort* em função do comprimento da lista de entrada, $|l|$.

Por fim, foi mostrado também que a implementação do *bubblesort* com o contador é equivalente à sua versão com adição do contador do número de trocas. Sendo assim, concluímos que o algoritmo *bubblesort* possui complexidade assintótica $\theta(|l|^2)$, uma vez que obtivemos que ele realiza exatamente $\frac{|l|^2-|l|}{2}$ operações de comparação.

Referências

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.