Radix Sort

Gabriel Alves 17/0033813,
Henrique Mariano 17/0012280,
Matheus Breder 17/0018997,
Yuri Serka 17/0024385.

22 de novembro de 2018

Resumo

Este trabalho explorou as noções do príncipio da indução matemática para a resolução de problemas que envolvem a prova de algoritmos. Abordamos portanto três problemas neste trabalho:

d_digits_gt :

$$\forall n : 10^{n\_digits(n)} > n$$

merge_permutes :

$$\forall l1, l2 \text{ listas de naturais e } \forall d \in \mathbb{N} :$$
$$permutations(append(l1, l2), merge(l1, l2, d))$$

merge_sort_d_sorts :

$$\forall l \text{ lista de naturais e } \forall d \in \mathbb{N} :$$
$$is\_sorted\_ud?(l, d) \implies is\_sorted\_ud?(merge\_sort(l, d), d + 1)$$

Os seguintes lemas "*n_digits*", "*permutations*", "*append*", "*merge*", "*merge_sort*", e "*is_sorted_ud*?" estão presentes nos arquivos de teoria que foram disponibilizados, radix_sort.pvs e sorting.pvs.

O PVS (Prototype Verification System) foi utilizado para construir as provas dos três problemas acima apresentados, utilizando-se de lemas presentes na teoria citada, da biblioteca de apoio nasalib e do próprio documento no pvs (prelude-file).

1 Introdução

Para este trabalho, todas as questões propostas foram provadas utilizando Indução Estrutural, que será explicada na próxima seção, e com o auxilio do assistente de provas PVS. Vimos como pode ser complicado provar a estabilidade de um algoritmo, e como os conceitos estudados na matéria podem ser aplicados na vida cotidiana de um pesquisador da área de ciência da computação.

A prova de um algoritmo refere-se a provar matematicamente que este algoritmo irá se comportar de uma maneira esperada nos casos determinados, por exemplo: Um algoritmo de ordenação sempre irá ordenar corretamente qualquer lista que a ele for submetida, sem haver perda de termos ou qualquer termo fora de ordem.

2 Indução Estrutural

A indução estrutural pode ser entendida como uma generalização da indução matemática convencional, mas é comumente usada para provar conjuntos definidos recursivamente. A semelhança vem pelo fato de que o passo base é mostrar que uma propriedades vale para todos os casos da definição da recursão, ou seja os casos base que são os objetos mais simples do conjunto, já o passo indutivo consiste em verificar se a propriedade vale para objetos mais complexos, estes que são construídos a partir dos mais simples.

Para este trabalho foi utilizado nas questões 2 e 3 a Indução na estrutura da lista, que serão explicadas nas seções 4, e 5, respectivamente. Para a questão 1, que está na seção 3 foi utilizado a Indução nos naturais.

3 Dez elevado ao número de dígitos de qualquer número natural, é sempre maior que este próprio número

Com o intuito de provar propriedades das casas decimais, adequadas a ordenação por um termo "d", por um algoritmo que trabalha ordenando os termos de uma lista segundo uma posição "d". Temos a seguinte proposição:

$$\forall n : 10^{n\_digits(n)} > n$$

n_digits é uma função que retorna a quantidade de algarismos que um determinado número natural possui.

Para construir a prova dessa propriedade, foi utilizada a indução forte em n. Isto pode ser realizado no pvs por meio do comando:

$$(measure - induct + \quad "n"("n"))$$

Que retorna ao usuário, dado a conjectura acima, a seguinte fórmula de indução forte (no formato de Gentzen):

Dessarte temos o antecedente, representando os

$$K < N$$

termos válidos da indução forte:

$$\forall y \in \mathbb{N} : y < X \implies 10^{n\_digits(y)} > y$$

E o consequente, representando o termo N, que deve ser provado como também válido:

$$10^{n\_digits(X)} > X$$

Desse modo, utilizando os comandos de prova do pvs, podemos construir a seguinte prova:

Para verificar as propriedades de n_digits devemos expandí-lo, assim poderemos verificar todos os casos da função n_digits, de acordo com uma entrada, e provar a nossa proposição em cada um destes casos. Assim aplicamos:

$$(expand \quad "n\_digits"\ 1)$$

aonde 1 indica que a fórmula sera aplicada em nosso consequente. Para organizar os ”IF-ELSE”que aparecem com a expansão da função, utilizamos

$$(lift-if)$$

. No pvs, as negações não são tratadas diretamente. Quando uma negação aparece a mesma é enviada ao lado oposto em que estava , segundo as propriedades de Gentzen. Também podemos entender um if como uma implicação (com certas ressalvas). De qualquer forma, para tratar os resultados de nosa expansão em ”n_digits”utilizaremos o comando

$$(prop)$$

que irá aplicar as regras: disjunção e conjunção à esquerda e à direita, implicação à esquerda e à direita. Além das regras de conclusão: Absurdo a esquerda e axioma.

Assim aplicando o prop chegamos a duas sub-árvores. Uma delas nos apresenta no antecedente o termo

$$X < 10$$

e no consequente o termo

$$10^1 > X$$

. Isso poderia ser, aos olhos humanos, facilmente enxergado como um axioma para Gentzen. Desse modo, poderíamos aplicar diversos comandos ”expand”, ”lift”, ”prop”e ”assert”, de modo a obter proposições que o pvs pudesse reconhecer como axiomas com mais facilidade. Essas etapas são automaticamente realizadas aplicando o comando

$$(grind)$$

, que também aplica as regras de axioma e absurdo à esquerda. Neste caso, o pvs foi capaz de manipular adequadamente as duas fórmulas de modo a fechar a sub-árvore em questão (por meio de um axioma, encontrado através da manipulação dos dois termos acima citados).

Na segunda sub-árvore obtemos um sequente um pouco mais complicado. Que possui no antecedente:

$$\forall y \in \mathbb{N} : y < X \implies 10^{n\_digits(y)} > y$$

e no consequente:

$$X < 10$$

$$10^{(1+n\_digits(ndiv(X,10)))} > X$$

ndiv é uma função que retorna a divisão de um determinado número natural (X) por um determinado número natural (10).

Desse modo, de acordo com as regras de substituição em Gentzen, podemos instanciar o nosso para todo, de modo a considerar uma variável específica. Neste caso, como o nosso interesse é chegar a um axioma, observando o consequente, podemos chegar a conclusão que a melhor instanciação é a substituição de y por ndiv(X,10). Isto pode ser realizado por meio da regra:

$$(inst\ \text{-}1 \quad \text{”}ndiv(X,10)\text{”})$$

Aonde -1”indica a única fórmula no antecedente a ser instanciada, tal como a apresentamos.

A partir deste instanciação, obtemos a mesma fórmula no antecedente, porém para o termo específico "ndiv(X,10)". Desse modo podemos aplicar o comando

$$split$$

para tratar uma implicação à esquerda. Este comando resulta em dois ramos, tal como na regra de Gentzen para a implicação à esquerda.

Em um dos ramos obtemos, um antecedente vazio, e no consequente:

$$ndiv(X,10) < X$$

$$X < 10(1)$$

$$10^{1+n\_digits(ndiv(X,10)))} > X$$

Dessa forma, podemos aplicar o comando

$$typepred"ndiv(X,10)"$$

que fará com que o pvs busque o retorno da função ndiv, para os valores X e 10. Com isso, nós chegamos a um novo sequente, com o mesmo consequente anterior, mas com um antecedente da forma:

$$X = 10 * ndiv(X,10) + rem(10)(X)(2)$$

$$ndiv(X,10) <= X(3)$$

Aonde rem indica o resto da divisão de X por 10.

E dessa forma, ao aplicar o comando

$$(assert)$$

para verificar um axioma, o pvs é capaz de perceber que ndiv(X,10), devido a proposição (2) é sempre menor do que X. Dessa forma, tem-se um axioma com (1), presente no consequente.

Já no segundo ramo, nós obtemos:

no antecedente:

$$10^{n\_digits(ndiv(X,10))>ndiv(X,10)}$$

no consequente:

$$X < 10$$

$$10^{1+n\_digits(ndiv(X,10))} > X$$

Com isso podemos aplicar

$$(grind)$$

, que após aplicado nos deixou o sequente:

no antecedente:

$$10 * expt(10, n\_digits(ndiv(X,10)) - 1) > ndiv(X,10)$$

no consequente:

$$X < 10$$

$$100 * expt(10, n\_digits(ndiv(X,10)) - 1) > X$$

Assim novamente, e pelas mesmas razões, podemos aplicar o comando:

$$typepred"ndiv(X,10)"$$

, o que nos leva ao seguinte sequente:
no antecedente:

$$X = 10 * ndiv(X,10) + rem(10)(X)$$

$$10 * expt(10, n\_digits(ndiv(X,10)) - 1) > ndiv(X,10)$$

no consequente:

$$X < 10$$

$$100 * expt(10, n\_digits(ndiv(X,10)) - 1) > X$$

dessa maneira, para facilitar a visualização, como também a identificação de um axioma pelo pvs, aplicamos o comando

$$(name - replace"kexpt(10, n\_digits(ndiv(X,10)) - 1)")$$

, que irá deixar o nosso sequente da forma: no antecedente:

$$X = 10 * ndiv(X,10) + rem(10)(X)$$

$$10 * k > ndiv(X,10)(4)$$

no consequente:

$$X < 10$$

$$100 * k > X$$

Deste modo, e neste caso, o pvs considerará rem(10)(X) como zero, considerando portanto

$$X = 10 * ndiv(X,10)$$

, o que implica que

$$100 * k > 10 * ndiv(X,10)$$

, o que permite ao pvs recohecer um axioma com a proposição (4) no antecedente. Logo fechamos todos os ramos da árvore de prova, e portanto termos uma prova da proposição dada nesta sessão 3.

4 União de duas listas com append ou merge

Ao unir duas listas com o algoritmo append, e ao unir duas listas com o algoritmo merge, sempre teremos uma permutação dos termos de ambas as listas, com estes termos preservados (não há perca de termos, nem adição, termos presentes das duas listas, são os mesmos na lista posterior).

Esta conjectura é escrita matemáticamente como:

$$\forall l1, l2, \quad \text{listas de naturais} \quad e \quad \forall d \in \mathbb{N}: \quad permutations(append(l1, l2), merge(l1, l2, d))$$

Sendo "permutations"uma função booleana que retorna se uma lista, é uma permutação da outra, ou não.

Assim, podemos fazer uma indução forte na soma dos tamanhos das listas l1 e l2. A escolha de fazer uma indução na soma dos tamanhos de ambas as listas se deve

ao fato de que na recursão este é o valor alterado a cada passo, indicando portanto o MEASURE do algoritmo merge. Assim, no pvs fazemos:

$$(measure-induct+"length(l1)+length(l2)" \quad ("l1" \quad "l2"))$$

Nos dando portanto o seguinte sequente:

no antecedente:

$$\forall l1, l2 \quad \text{listas de naturais} : \quad \forall d \in \mathbb{N}$$

$$length(Y\_1)+length(Y\_2) < length(X!1)+length(X!2) \implies permutations(append(Y\_1,Y\_2),merge(Y\_1,Y\_2,d))$$

no consequente:

$$\forall d \in \mathbb{N} : permutations(append(X!1,X!2),merge(X!1,X!2,d))$$

Assim, é possível aplicar a indução forte na soma dos tamanhos das listas.

Em nossa primeira abordagem na prova expandimos o merge, com isso tentamos seguir na prova abrindo em vários ramos utilizando o comando prop.

O primeiro ramo é trivial pois, temos como hipótese a primeira lista x!1 como null, e no consequente temos que permutations(append(x!1, x!2), append(x!1, x!2)), como nossa lista x!1 é vazia temos por definição do append que append(null, x!2) = x!2, logo reescrevemos permutations como permutations(x!2, x!2) e por definição de permutations temos que a permutação de duas listas iguais é sempre verdadeiro, e portanto fechamos esse ramo, pois no antecedente temos que o permutations(append(y_1, y_2), merge(y_1, y_2, d)) é sempre verdade e no consequente temos que permutations(append(x!1, x!2), append(x!1, x!2)) é sempre verdade. O segundo ramo é semelhante ao primeiro pois temos como hipótese que a segunda lista x!2 é vazia.

Assim, chegamos ao quarto ramo, em uma primeira abordagem tentamos instanciar nossa hipótese length(y_1) + length(y_2) ¡ length(x!1) + length(x!2) IMPLIES permutations(append(y_1, y_2), merge(y_1, y_2, d)) com listas genéricas x!1 e x!2, porém a medida em que avançavamos na prova não era possível concluirmos. Desta forma, voltamos na prova e tivemos a ideia de instanciar nossa hipótese com uma lista genérica x!1 e a segunda lista como a cauda de x!2, dessa maneira nossa hipótese ficou com a forma length(x!1) + length(cdr(x!2)) ¡ length(x!1) + length(x!2) IMPLIES permutations(append(x!1, cdr(x!2)), merge(x!1, cdr(x!2), d)), com essa instanciação podemos ver que length(x!1) + length(cdr(x!2)) ¡ length(x!1) + length(x!2) é verdade pois, a soma do tamanho das listas de x!1 e x!2 sempre será maior que a soma do tamanho de x!1 com a cauda de x!2.

Com isso, expandimos o permutations e assim obtemos nossa hipótese como FORALL (x: nat): occurrences(append(x!1, cdr(x!2)))(x) = occurrences(merge(x!1, cdr(x!2), d))(x) e dessa maneira abrimos a prova em 2 ramos, com o comando prop, em nosso primeiro ramo utilizamos o lemma occurrences of app, nesse lemma temos que as ocorrências de append(l1, l2) são iguais as somas das ocorrências de l1 e l2, dessa forma instanciamos nosso lemma com duas listas genéricas x!1 e x!2, obtendo occurrences(append(x!1, x!2))(x) = occurrences(x!1)(x) + occurrences(x!2)(x), em nosso consequente temos occurrences(append(x!1, x!2))(x) = occurrences(cons(car(x!2), merge(x!1, cdr(x!2), d)))(x).

No primeiro, temos para fechar a prova o comando (grind), ele basicamente percorreu a lista, ou seja, foi contando as ocorrências sempre da cauda da lista que era contruída a partir da cabeça de x!1 e com o append da cauda de x!1 com a cauda de x!2. Nisso o PVS achou a igualdade necessária para fechar o ramo da prova, que também é conseguida após várias skolimizações, instanciações e simplificações de if-else.

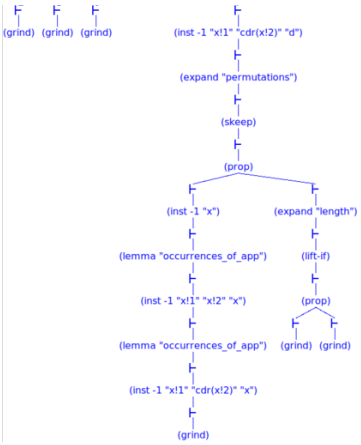

Figura 1: Parte 2 da Árvore da prova.

No segundo ramo temos length(cdr(x!2)) + length(x!1) ¡ length(x!1) + length(x!2) no consequente, temos que essa fórmula é sempre verdade, como não há fórmulas no sequente expandimos o length e depois damos prop para separar em 2 casos, no primeiro caso temos que null?(cdr(x!2)) como hipótese logo trivialmente temos nossas proposição como verdade, no segundo caso temos algo semelhante.

5 dada uma lista ordenada em d, merge sort é correto para ordenar d + 1

Assim, podemos também verificar a conjectura apresentada abaixo, a qual nos aproximaria mais um passo de provar que o algoritmo radix_sort é correto. Então dada a conjectura:

$$\forall l \text{ uma lista de naturais} \quad e \quad \forall d \in \mathbb{N}:$$

$$is\_sorted\_ud?(l,d) \implies is\_sorted\_ud(merge\_sort(l,d),d+1)$$

is_sorted_ud? é uma função booelana que nos indica se a lista esta ordenada, ou não, em d. Explicamos aqui que a função is_sorted_ud? começa a contar a partir do natural 1 (a casa dos decimais é denominada por d = 1 e assim por diante). Enquanto a função merge_sort começa a contar a partir do natural 0 (a casa dos decimais é denominada por d = 0 e assim por diante). Logo o que nossa conjectura nos indica é: Se temos uma lista ordenada na casa d de seus naturais, ao aplicar o algoritmo merge_sort, teremos uma lista ordenada na casa d + 1 (indicando a casa dos decimais como d = 1, de acordo com o algoritmo is_sorted_ud?).

Desse modo, podemos então construir uma prova por meio de indução forte para esta conjectura, no tamanho da lista l. O tamanho da lista l foi o escolhido, pois é a grandeza que se altera a cada passo da recursão, sendo portanto definido como MEASURE para merge_sort. Desse modo aplicando a indução forte no pvs por meio do comando:

$$(measure-induct+ \quad "length(l)" \quad ("l"))$$

Chegamos ao seguinte sequente, que possui no antecedente:

$$\forall y \text{ uma lista de naturais} \quad e \quad \forall d \in \mathbb{N}:$$

$$length(y) < length(x) \implies$$
$$is\_sorted\_ud?(y,d) \implies is\_sorted\_ud(merge\_sort(y,d),d+1)$$

E no consequente:

$$is\_sorted\_ud?(x,d) \implies is\_sorted\_ud?(merge\_sort(x,d),d+1)$$

Assim no pvs devemos fazer duas instanciações, após aplica o skeep: Para x e para d, na fórmula do antecedente. Após feitas as instanciações, o único passo possível é expandir o merge_sort e aplicar o lift-if para reorganizar a prova. Nota-se que, na visão matemática tradicional, o que estamos fazendo aqui é generalizar o a hipótese de indução para listas de tamanho inferior a X, e executar a operação "merge_sort"no nosso passo indutivo, olhando portanto para a sua definição recursiva.

Deste ponto, podemos utilizar o lema "merge_preserves_sort"por meio do comando:

$$(lemma"merge\_preserves\_sort")$$

Este lema nos indica que dadas duas listas ordenadas no d'ésimo dígito, ao aplicar o merge em ambas as listas no d'ésimo dígito, teremos como resultado uma lista ordenada. Isto é exatamente o que precisamos, pois queremos provar que merge_sort preserva a ordenação no d'ésimo dígito.

Neste ponto, é importante realizar uma instanciação correta no que é gerado no pvs, a partir do chamado do lema acima citado. Assim devemos aplicar a seguinte instanciação:

$$(inst \quad -1 \quad "prefix(x, floor(length(x)/2))" \quad "suffix(x, floor(length(x)/2))")$$

Pois são as duas listas sob as quais estamos aplicando o merge, segundo a definição recursiva de merge_sort, neste caso, abrimos dois ramos, um deles é trivial e é facilmente fechado. A partir deste ponto, no ramo restante, podemos aplicar o comando "prop"que irá subdivir a árvore para cada caso, segundo a expansão de merge_sort. O que, na matematica tradicional significa considerar cada condição da recursão de merge_sort. Assim obtemos outros 16 ramos, destes, 11 ramos são triviais, enquanto outros 5, ainda precisam ser provados.

temos a àrvore:

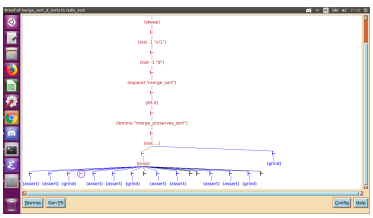

Figura 2: Estado em que a árvore foi deixada.

A partir deste ponto a equipe não conseguiu avanços.

6 Conclusão

Vemos neste trabalho a grande dificuldade advinda da tentativa de comprovar a corretude de algoritmos, como também definir lemas corretos que auxiliem nas respectivas

provas. Com isso, percebemos que assitentes de prova como o PVS, aliados ao bom e velho papel, lápiz e borracha, são ferramentas extremamente úteis na construção de provas para a corretude de algoritmos, tornando, quando alcançada a perícia necessária, muito mais prática a criação de provas que afiram a corretude de algoritmos, auxiliando portanto na criação de sistemas de software muito mais robustos, e confiáveis. Tarefa tão necessária em alguns sistemas complexos que executam papéis cruciais em artefatos e outros sistemas como: Aviões, foguetes, usinas nucleares, e mercados financeiros.