UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO: CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: ESTRUTURA DE DADOS
ALUNO: MARCOS ANTÔNIO DIAS JÚNIOR
EMAIL: belmont.programmer@gmail.com

Relatório TP1 - Avaliação e Satisfatibilidade Booleana

Introdução

Neste trabalho foram propostos dois problemas clássicos de computação: O primeiro foi propor a implementação de um avaliador de expressões booleanas de maneira iterativa - um problema simples. Já o segundo se tratava de um problema clássico de computação: o problema de Satisfatibilidade (SAT). Neste problema o desafio se tratava de implementar um programa que, dado uma expressão booleana e um conjunto de valorações - incluindo quantificadores booleanos -, checava se tal expressão é de fato satisfazível para aquela dada valoração.

Se por um lado o primeiro problema de avaliação é uma proposição simples - que geralmente é implementada com complexidade linear em relação ao número de operadores da entrada -, o problema de SAT por outro lado, se trata de um problema muito mais desafiador. A questão da satisfatibilidade é uma questão cara no ramo da teoria da computação e com aplicações diversas que vão desde a simplificação de circuitos e otimização de sistemas à implicações em Inteligência Artificial e aplicações mais complexas. Trata-se não só de *mais um* problema NP Completo, mas mais especificamente do primeiro problema NP Completo a ser provado como tal - como demonstrado pelo Teorema de Cook em 1971.

Optou-se por implementar a solução aqui apresentada inteiramente em C++, devido a mera preferência de linguagem, uma vez que o uso da STL foi vetado. Também optou-se por uma abordagem de implementação de soluções utilizando-se uma adaptação da notação pós-fixa de expressões da álgebra aritmética para as expressões booleanas, que basicamente se diferem pela existência de operadores básicos unitários (como o “¬”, “NOT” lógico)[1].

Metodologia

Classes e Estrutura de Dados

As únicas estruturas de dados significativas utilizadas foram Filas e Pilhas típicas. Ambas foram implementadas como classes genéricas (utilizando *templates methods* e

[1] Aqui operadores unitários típicos da álgebra aritmética como exponenciação, radiciação, fatoriais e etc não foram tomados por operadores básicos - uma vez que podem se descritos como aninhamento dos operadores de adição, subtração, multiplicação e divisão (que aqui são entendidos por básicos).

template types) e com alocação dinâmica. Uma peculiaridade da classe Fila é que o método "At", típico de vetores e listas ordenadas, foi adicionado a classe. Não havia a real necessidade da implementação deste método neste trabalho - uma vez que cada dado alocado na estrutura fila (neste programa) só é utilizado uma vez - durante sua avaliação. Porém, optou-se por essa implementação de maneira arbitrária apenas para fazer com que fosse possível iterar sobre a fila sem necessariamente ter que esvaziá-la.

A classe Fila foi utilizada para comportar as expressões booleanas em suas formas in-fixa (a forma temporária pela qual a expressão é apresentada na entrada do programa) e pós-fixa, a forma final na qual a expressão é armazenada na estrutura de dados do programa, uma fila de strings. A classe pilha é utilizada apenas no momento da avaliação das expressões pós-fixas: operadores e operandos são empilhados em pilhas distintas durante a avaliação. Os operandos são empilhados em pilhas de inteiros e os operadores em pilhas de strings.

Adicionalmente foram criadas *structs* que são apenas utilizadas na notificação e tratamento de exceções do programa.

Métodos

string2queue: este método basicamente carrega uma fila com strings obtidas pela seccionamento da string de entrada. O método é alimentado com uma string a ser subdivida em múltiplas substrings (usando *whitespaces* como delimitador) e retorna uma fila carregada. Dentro deste método a função **split** é utilizada.

infix2postfix: este método recebe uma string contendo uma expressão infixa e uma fila de strings na qual será armazenada a expressão pósfixa resultante. A implementação deste método segue a lógica do *shunting yard algorithm* (Algoritmo do Desvio Padrão), como apresentado por por Edsger Dijkstra porém utiliza filas como estrutura de dados final e o sistema de prioridades foi remodelados para o da álgebra booleana.

variations: este método recebe a string de valores de variáveis dada na entrada do programa e um carácter a ser substituído por suas variações (neste caso 'a' ou 'e'). Este método foi implementado de maneira recursiva e carrega uma fila de strings com todas as variações possíveis de valores atribuíveis aos caracteres designados. Este método substitui um tipo de quantificador por vez e gera 2^n strings diferentes - onde n é o número de vezes que o caractere de substituição aparece na string original.

calculate: método que recebe 2 operandos e um operador e retorna o cálculo booleano entre eles. Os operandos são passados por booleans e o operador por char. Quando o operador em questão é o de negação, o segundo operando é ignorado - uma vez que o NOT lógico é um operador unitário.

solvePostfix: esta função recebe uma Fila contendo uma expressão pós-fixa e uma string com os valores das variáveis nesta expressão e a avalia com ajuda de uma pilha. Como a expressão está na sua forma pós-fixa a avaliação é bastante intuitiva: ao iterar na expressão o algoritmo empilha operandos até encontrar um operador. Quando encontra um operador o algoritmo desempilha os dois últimos operandos, realiza o cálculo entre eles e empilha o resultado. Quando a pilha de operador chegar ao tamanho 1, a expressão está resolvida.

satA: Este método testa a satisfatibilidade de expressões que contém apenas valores fixos e quantificadores “para todo”. É um método iterativo que recebe uma fila contendo uma expressão booleana in-fixa e uma string de valoração de variáveis e retorna verdadeiro caso a expressão seja satisfazível em todas as variações dos valores de todos os “para todos” na expressão. Objetivamente, o método testa todas as variações possíveis de valores para os “para todos” e retorna falso se pelo menos uma delas falhar ou verdadeiro se todas forem satisfazíveis.

sat: Este é o método de satisfatibilidade mais genérico. Este método gera todas as variações de valoração das variáveis “existe” presentes na entrada e, para cada dessas variações, testa a satisfatibilidade dos “para todos” submetendo-as no método **satA.** Se qualquer das variações de “existe” passar em todos os testes das variações de “para todo” a expressão é considerada satisfazível para aquela valoração de “existe”. Por fim, este método checa a variáveis de *don’t care* através da função **solution** descrita abaixo.

solution: esta função recebe um *array* de strings com valorações que satisfazem a expressão booleana e as compara para encontrar os casos de *don’t care* - ou seja, as variáveis cujos seus valores não alteram o resultado da expressão - e as substituem por ‘a’ na string final.

Procedimentos

Avaliação

O procedimento de avaliação de expressões consiste basicamente em transformar a expressão em pós-fixa, usando uma variação do *shunting yard algorithm* já descrita no método **infix2postfix**, e então resolvê-la com o auxílio de uma pilha. É apenas durante a avaliação que os valores das variáveis indexadas na expressão são de fato substituídos pelos valores fornecidos na entrada. Isso foi uma decisão de projeto escolhida para que não fosse necessário construir uma expressão com os valores a serem usados sempre que fosse necessário avaliar a expressão - uma vez que, no caso de testar satisfatibilidade, a mesma expressão é avaliada várias vezes com múltiplos valores.

Teste de Satisfatibilidade

Para checar se os valores de entrada das variáveis fornecidas são capazes de satisfazer uma expressão, o seguinte procedimento é realizado: no método **sat,** primeiramente são geradas strings que contém todas as possibilidades de valoração possíveis pela substituição dos “existe” na string original - algo muito similar a gerar uma tabela verdade apenas das variáveis representadas por ‘e’ na string original. Todas essas strings são passadas para o método **satA** que, para cada uma dessas variações, gera ainda novas variações obtidas pelas substituições das variáveis ‘a’ restantes. Essas últimas variações são testadas usando o **solvePostfix** até que todas as variações de “para todo” retornem true. Neste caso a expressão é considerada satisfazível. Porém se, para uma dada variação de “existe”, uma das valorações de “para todo” retornar falso, o método interrompe as avaliações dessa variação e passa a testar a próxima variação de existe. Isso porque, uma vez que aquela valoração de “existe” não é verdadeira para todo ‘a’ na expressão, aquela expressão não é satisfazível com aqueles valores. Por tanto, o código

checa se existem outros valores para os quais a expressão seja satisfazível para todo ‘a’ nela.

Análise de Complexidade

Primeiramente há que se ressaltar que uma análise de complexidade feita somente em função do tamanho da entrada (tamanho da expressão) seria plenamente ineficiente e pouco confiável, uma vez que o programa escalona de maneira geralmente linear em relação ao tamanho da entrada porém escalona de forma exponencial em relação ao número de quantificadores. Por exemplo, em termos de complexidade de espaço, o método de avaliação pode receber uma entrada com infinitos operadores NOT e sua alocação adicional de memória se mostrará constante, desde que o número de opeandos se mantenha constante. Igualmente, o método de satisfatibilidade tem crescimento de complexidade de tempo e espaço apenas linear em relação ao tamanho da expressão de entrada, porém cresce de forma exponencial para cada quantificador implementado. Assim uma expressão de tamanho muito curto, porém com muitos operadores, é muito mais complexa que uma expressão significativamente mais longa porém com poucos ou nenhum quantificadores. Por este motivo, todas as análises de complexidade levarão em conta tanto o tamanho da exrpessão quanto o número de quantificadores utilizados.

infix2postfix: Neste método apenas iteramos pela expressão uma vez - num loop que checa caracteres - portanto é um método de complexidade procedural linear em relação ao tamanho da entrada, **O(n).** Essa análise também é válida para análise da complexidade espacial, uma vez que o espaço reservado é igual ao tamanho da entrada. Um detalhe curioso é que, dada a especificação da entrada do trabalho (que prevê espaços em branco entre todos os operandos e operadores da expressão), seria facilmente possível diminuir a alocação de memória deste método. Isso porque fica evidente que, se string tem **n** caracteres, o número de caracteres que não são espaços em brancos são necessariamente **MAX(n/2) +1.**

variations: este é o único método recursivo do código. Este método itera por toda a string original checando se o caractere é o caractere de substituição (quantificador). Porém, para cada caractere de substituição encontrado, ele chama recursivamente a si mesmo duas vezes (porém dessa vez o método só percorre os caracteres que ainda não foram checados da string). Assim temos que, se **k** é o número de quantificadores na string e **n** o tamanho da string, a ordem de complexidade deste método é **O(n * 2^k).** Como este método foi implementado de maneira recursiva, se aproveitando da própria estrutura da pilha de execução, ele possui uma complexidade de espaço tipicamente alta: ao passar por cada um dos **n** caracteres o método chama a si mesmo uma vez passando a string construída até ali como argumento. Porém, se o caractere é um quantificador, o método faz a alteração naquele caractere e chama a si mesmo recursivamente porém passando duas strings diferentes como argumento. Assim a complexidade de espaço dessa função é **O(n^2 * 2^k).**

calculate: método de complexidade constante, independentemente se o operador é unitário ou binário - uma vez que este método simplesmente faz operações booleanas entre uma ou duas variáveis.

solvePostfix: este método itera pela expressão pós-fixa apenas uma vez checando se o caractere é um operador ou operando. Se se tratar de um operando, este operando é empilhado na pilha auxiliar. Se, porém, se tratar de um operador, a devida operação booleana é executada. A complexidade de tempo desse método seria portanto **O(n) + O(k)**, onde **k** é o número de operadores. Logo, a complexidade seria **O(MAX(n, k)),** onde **n** é necessariamente menor que **k**. Portanto, a complexidade de tempo é melhor descrita como **O(n) .** Em termos de complexidade de espaço, esse método se trata de um método bem comportado, uma vez que a única alocação de memória adicional necessária se trata da pilha de operandos - que necessariamente é menor que o tamanho da expressão. Logo, temos que a complexidade espacial desta função é **O(n).**

Mais precisamente, poderia-se dizer que a complexidade espacial é na verdade **O(t)**, onde **t** é o número de operandos na expressão**.** É bastante imediato concluir que o número de operandos é limitado a no máximo **n/2 + 1** - uma vez que todo operando está sofrendo no mínimo uma operação. Porém, como há presença de operadores unitários, não é possível precisar o número de operandos em função do tamanho da entrada. Em essência, um operador de negação, NOT, poderia ser aninhado infinitamente num mesmo operando - sendo, por exemplo, possível gerar expressões algébricas de tamanhos arbitrários com apenas um operando (negando infinitamente uma mesma variável). Ainda assim, a complexidade de espaço do algoritmo ainda seria tão baixa quanto o número de operandos. Porém o ponto aqui é que número de operandos não está em função do tamanho da expressão. Sem conhecer a expressão, sabemos no máximo que o número de operandos é sempre igual ou menor que **n/2 +1.** Ainda assim, ordem de **n/2 +1** ainda é ordem de **n.**

satA: este método executa o método **variations** e em seguida o método **solvepostfix 2^a** vezes**,** onde **a** é o número de quantificadores “para todo”**.** Assim sua complexidade de tempo seria **O(n * 2^a) + O(n * 2^a),** logo a ordem de complexidade é **O((n*2^a)).** Em termos de complexidade de espaço, porém, a cada nova chamada do método **solvePostfix** a memória deste método é liberada e realocada. Assim a complexidade de espaço deste método seria **O(n * 2^a) + O(n)** - portanto, **O(MAX(n*2^a, n)).** Logo, nossa complexidade é claramente **O(n),** considerando que **a** é um número natural**.**

sat: este método funciona de maneira praticamente igual ao método anterior. A diferença é que, no lugar de chamar o método **solvepostfix**, ele chama o método **satA**. O método, portanto, executa a função **variations** e depois executa o método **satA 2^e** vezes - onde **e** é o número de quantificadores “existe” na expressão original. Logo, se o método **satA 2^a** vezes (onde **a** é o número de quantificadores “para todo”), conclui-se que o método **solvepostfix** é executado **2^e * 2^a = 2^k** vezes (onde **k** é o número total de operadores). Assim a complexidade de tempo deste método é **O (MAX(2^k, n*2^k)).** Como esperado, em termos de ocupação de memória, a complexidade dessa função é a soma da complexidade da função **variations** com a função **satA** - uma vez que apenas uma **satA** é alocada na pilha de execução por vez. Portanto a complexidade de espaço seria, respectivamente, **O(n² * 2^e) + O(n² * 2^a).** Logo, **O(MAX(n²*2^e, n²*2^a));**

solution: este método apenas itera sobre os caracteres das strings de valoração que satisfazem a expressão procurando casos de *don't care*. Como neste trabalho não há soluções disjuntas, as comparações são apenas feitas entre a primeira solução encontrada e todas as outras. A complexidade de é **O(v*n)**, onde **v** é o número de soluções válidas. Portanto, ordem de **n** - uma vez que o número de soluções válidas é tipicamente pequeno e nunca é maior que **n.**

string2queue: este método tem a complexidade padrão da separação de uma string em palavras. **O(n)** tanto para complexidade de tempo quanto para o uso de espaço de memória auxiliar.

Considerações sobre a análise assintótica

O recorte do problema apresentado define que o número de quantificadores no problema de satisfatibilidade é no máximo 5. O problema de satisfatibilidade é NP Completo, por tanto não existem métodos garantidos muito mais eficientes do que tentar todas as possíveis possibilidades. Até pode se diminuir ligeiramente o tempo de computação colocando as expressões na forma normal conjuntiva, mas ainda assim isso não torna o programa muito mais eficiente, além de demandar a transformação da expressão. A solução aqui apresentada foi feita para funcionar com qualquer número de quantificadores, em termos algorítmicos. Porém a implementação das estruturas de dados foram escolhidas visando a facilidade de implementação, comparação e legibilidade. Caso a restrição de número de quantificadores fosse retirada e fossem utilizados um número muito alto de quantificadores seria necessário refatorar as estruturas de dados utilizadas (bem como o método **variations**) para aliviar o uso de memória. Todas estas alterações já foram discutidas e explicadas na avaliação assintótica de cada função.

Em termo de complexidade de tempo e espaço o algoritmo desenvolvido escalona tanto com o tamanho da entrada (de maneira linear) quanto com o número de quantificadores (de maneira exponencial). Não me parecesse necessário aprofundar mais na análise assintótica do algoritmo em relação ao número de operadores, uma vez que todas as análises de complexidades apresentadas acima foram feitas em função do tamanho da entrada e do número de quantificadores - evidenciando o quão impactante é o número de quantificadores neste tipo de algorítmo: tão importante ou ainda mais importante que o tamanho da entrada.

Robustez:

Em termos de estrutura de dados, todas elas foram implementadas encapsulando os dados em si. De forma, que o usuário apenas modifica este dados pelos métodos típicos de cada estrutura de dados - não tendo acesso direto aos dados em si, senão por elas. *Throws* são lançados em casos de *null pointer exception* (também conhecido por *segmentation fault* devido a acesso a memória não alocada).

Caso a sintaxe de execução do programa (**argc >= 4**) não seja satisfeita, uma exceção é lançada e uma mensagem de erro auto-explicativa é exibida. O mesmo ocorre quando a expressão contém caracteres inválidos ou não previstos.

Análise Experimental:

Uma vez que as soluções são avaliadas via VPL e a solução aqui apresentada passou em 100% dos testes, considero desnecesaŕio discorrer sobre a eficácia do programa. Porém, algo curioso de se notar foram as tentativas de envio: inicialmente apenas a classe pilha havia sido feita com alocação dinâmica. Isso era suficiente pro código passar em todos os testes do VPL de testes. Porém, em 60% dos casos,o código era interrompido por motivo desconhecido quando rodava no VPL de avaliação. Isso acontecia porque a memória stack é muito curta em relação a memória heap. Assim, quando era necessário alocar filas muito grandes (em alocação estática) o programa tentava acessar memória que ele não conseguiria alocar - porque passava do limite de memória disponível para alocação estática. Isso foi resolvido simplesmente alocando memória dinamicamente na fila - posto que a pilha já foi construída com alocação dinâmica.

Em termos de localidade de refência, o momento mais crítico seria durante a avaliação da expressões - uma vez que as estruturas de dados foram implementadas de forma armazenar tanto a variáveis quanto a expressão de forma contígua. Inicialmente foi implementado um método **buildExpression** que, além de transformar uma expressão in-fixa em uma expressão pós-fixa, também substituia os valores das variaáveis (originalmente einderaçadas por números naturais) por seus respectivos valores dados pela string de valoração fornecida pela entrada do programa. Dessa forma, o programa teria uma ótima localidade de referência no tocante a avaliação de expressões, porém há um *trade-off* de desempenho aqui: Em casos de satisfatibilidade, a mesma expressão é avaliada muitas vezes com múltiplas valorações. Isso tornaria necessário reconstruir a mesma expressão muitas vezes - algo que toma uma quantidade considerável de recursos computacionais. Por tanto, foi decidido que a avaliação de expressões se daria mantendo o endereçamento de variáveis de forma que uma expressão só fosse “construída” uma única vez. O método de avaliação ainda caminha na string de valoração com uma localidade de refência espacial tão boa quanto a disposição das variáveis na expressão permitir, no entanto ele ainda tem que caminhar tanto na fila que comporta a expressão, quanto na string de valoração - diferentemente da solução previamente implementada, **buidExpression**, que apenas iterava na fila de expressão. Essa foi de fato uma escolha de projeto, uma vez que o código pode realmente escalonar com o número de quantificadores - o que tornaria o uso de memória e processamento muito mais ineficiente, caso múltiplas versões de valoração da mesma expressão fossem construídas.

Conclusão

A solução apresentada resolveu tanto o problema de avaliação booleana quanto o problema de satisfatibilidade através da avaliação de expressões pós-fixas - sistema que inclusive é mais natural para avaliação algorítmica computadorizada do que o sistema de notação in-fixa. Em termos de desafio, apesar de se tratar de um problema NP Completo, este trabalho não apresentou grandes desafios nem foi necessário muita pesquisa para desenvolvê-lo - exceto no tocante a conversão in-fixa -> pós-fixa, no qual foi necessário pesquisar o método de *shunting yard* de Dijkstra. No entato, no que é relativo aos algorítmos de avaliação e satisfatibilidade em si, que é o mote deste trabalho, tudo correu de maneira bastante intuitiva, todas as ideias de soluções implementadas se mostraram

rapidamente bastante eficazes e não houveram surpresas negativas quanto às soluções pensadas e implementadas - basicamente tudo se mostrou eficaz na primeira implementação.

No entanto, o maior aprendizado durante a implementação desta solução - e aqui sim houve algo inesperado - foi relativo à diferença de tamanho entre a memória *heap* e a *stack memory*. Mesmo que o código do algoritmo em si foi desenvolvido sem maiores percalços, o escalonamento da entrada foi uma surpresa, uma vez que eu não estava ciente da diferença de tamanho no alocamento de memória. Assim a classe fila precisou ser refatorada para suportar alocamento dinâmico - algo que inicialmente já havia sido feito, porém havia sido descontinuado uma vez que muitos métodos retornavam filas e o programa estava sofrendo com dupla liberação de memória. Durante a refatoração tanto os problemas de liberação de memória quanto os problemas de vazamento de memória foram resolvidos, assim como aproveitou-se para transformar todas as classes em implementações genéricas das mesmas usando *template types* e *template methods* - isso foi feito meramente por exercício, uma vez que a única classe que demandava mais de um tipo na utilização era a classe Pilha.

Por fim, há que considerar o *trade off* de legibilidade por otimização: como foi amplamente discutido na seção de análise de complexidade e assintótica, boa parte dos métodos poderiam funcionar de maneira mais otimizada ou alocar menos memória, porém o preço seria de que o código se tornaria notavelmente mais complexo e menos legível. Todas as possíveis alterações visando tornar o código mais otimizado foram discutidas e explicadas nesta documentação, porém - dado o escopo do projeto - não foi necessário implementá-las. No entanto, se o número de quantificadores ou o tamanho da entrada escalonasse muito, elas se tornariam de fato necessárias para garantir a viabilidade da avaliação.

Bibliografia

- **https://en.wikipedia.org/wiki/Boolean_satisfiability_problem**
- **https://en.wikipedia.org/wiki/Shunting_yard_algorithm**
- **https://www.geeksforgeeks.org/proof-that-sat-is-np-complete/**

Link para o repositório:

https://github.com/trevorbelmont/Boolean-SAT-and-Evaluator.git