Trabalho Prático 1

Expressões lógicas e satisfabilidade

Júlio Guerra Domingues (2022431280)

Estruturas de Dados - DCC/UFMG - 2023-2

Introdução

O presente trabalho lida com dois temas relacionados à lógica: avaliação de expressões lógicas e verificação de satisfabilidade. Nele, lidamos com variáveis binárias e operadores lógicos conhecidos, como "E" (∧), "OU" (∨) e "NÃO" (¬). Uma expressão lógica consiste em variáveis sujeitas a esses operadores. São utilizados também dois quantificadores: "EXISTE" (∃) e "PARA TODO" (∀), que se referem aos valores que as variáveis podem assumir.

Para a avaliação de expressões lógicas, o programa recebe uma expressão ϕ(x1, x2, ..., xn), onde x1, x2, ..., xn são as variáveis que aparecem na expressão, e uma valoração para elas. A resposta consiste em substituir as variáveis por seus valores e calcular se a expressão ϕ é verdadeira (1) ou falsa (0). A ordem de precedência das operações é a seguinte: Precedência delimitada por "()" > negação (¬) > conjunção (∧) > disjunção (∨). A saída é um inteiro (0 ou 1) que indica se a expressão é verdadeira ou falsa.

No problema da satisfabilidade, por sua vez, o objetivo é encontrar se há uma valoração que satisfaça a expressão. A entrada inclui a expressão lógica e uma string que codifica os quantificadores para as variáveis. Algumas variáveis podem ser quantificadas com ∃ ou ∀, enquanto outras terão valores pré-estabelecidos. A resposta será verdadeira (1) se uma valoração que satisfaça a expressão existir e falsa (0) caso contrário. Caso exista, a valoração que satisfaz a expressão é apresentada em seguida.

Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema Operacional: MacOS Sonoma 14.0
- Processador: ARM Apple Silicon M2
- Memória RAM: 8,00 GB

Para a avaliação das expressões lógicas, a estrutura de dados escolhida foi a pilha. A pilha é uma escolha adequada para esse cenário devido à natureza das expressões lógicas, que podem conter operadores e operandos aninhados em várias camadas. A pilha permite que o programa avalie a expressão de maneira eficiente, seguindo a precedência dos operadores. Quando um operador é encontrado, ele é empilhado, e quando um operando é encontrado, ele é empilhado para aguardar a operação apropriada. Esse comportamento em camadas e a ordem de processamento "último a entrar, primeiro a sair" da pilha se alinham bem com a lógica da avaliação de expressões. Além disso, a pilha encadeada é eficiente em termos de uso de memória, uma vez que pode ser redimensionada dinamicamente à medida que a expressão é avaliada.

No caso da verificação de satisfabilidade, a estrutura de dados escolhida foi a árvore binária. A escolha da árvore binária é justificada pela natureza hierárquica das expressões lógicas quantificadas, onde os quantificadores "para todo" e "existe" podem estar aninhados em vários níveis. A árvore binária permite que o programa represente a estrutura da expressão de maneira organizada, com cada nó representando uma parte da expressão. Isso facilita a resolução da satisfabilidade da expressão, já que a árvore pode ser percorrida recursivamente para avaliar as combinações possíveis de valoração. Além disso, a árvore binária se encaixa bem com a estrutura recursiva da avaliação da satisfabilidade, tornando o código mais legível e eficiente em termos de tempo, uma vez que evita o retrabalho e o recálculo de subárvores da expressão.

O programa é composto pelas seguintes classes principais:

- Pilha: Classe abstrata que define a interface para pilhas.
- PilhaEncadeada: Classe concreta que implementa uma pilha encadeada.
- TipoCelula: Representa um nó de uma pilha encadeada.
- ArvoreBinaria: Classe que representa a árvore binária.
- TipoNo: Representa um nó de uma árvore binária.

- ExpressaoLogica: Classe que lida com a avaliação de expressões lógicas.
- Satisfaz: Classe que lida com a verificação de satisfabilidade da expressão.

As principais funções relacionadas à resolução do problema são:

- avaliar (na classe ExpressaoLogica): A função "avaliar" percorre a fórmula lógica caracter por caracter, lidando com operadores e variáveis. Ela utiliza duas pilhas para controlar operadores e variáveis, respeitando as precedências. No final, retorna verdadeiro ou falso com base na avaliação da expressão.

- avaliarProximaOperacao (na classe ExpressaoLogica): Quando um operador é encontrado na fórmula, a função "avaliarProximaOperacao" é chamada. Ela lida com a operação atual, desempilhando operadores com alta precedência, aplicando a operação e empilhando o resultado de volta.

- resolve (na classe Satisfaz): A função "resolve" é crucial para avaliar fórmulas com quantificadores existenciais e universais. Ela é chamada recursivamente para cada nó da árvore da fórmula lógica. Ao avaliar, a função lida com quantificadores e tenta encontrar valorações que satisfaçam a fórmula. Se uma solução é encontrada, a função atualiza a valoração e marca o nó como resolvido. Se não, o nó é marcado como resultado falso, indicando que a fórmula não é satisfeita.

- avaliaSatisfaz (na classe Satisfaz): A função "avaliaSatisfaz" inicia o processo de avaliação da satisfação da fórmula lógica. Ela chama a função "resolve" para iniciar a resolução da árvore. Com base no resultado, imprime "true" e a valoração correspondente se a fórmula for satisfeita, ou imprime "false" se a fórmula não for satisfeita para nenhuma valoração. Se "resolve" retornar "false", "avaliaSatisfaz" imprime "false" diretamente.

Foram utilizadas ainda funções para manipulação das estruturas de dados supradescritas (pilha e árvore), além de outros getters e setters para a manipulação de atributos de estruturas privadas.

Análise de Complexidade

Para a avaliação de expressões lógicas, o tamanho máximo da entrada é da ordem de 10^6 caracteres para a *string* que representa a fórmula lógica e no máximo 100 caracteres para a *string* de valoração. Para o problema de satisfabilidade, as restrições limitam o máximo de cinco variáveis quantificadas. Essas informações definem o tamanho das entradas relevantes para a análise de complexidade. A complexidade dos algoritmos é influenciada principalmente pelo tamanho da fórmula lógica, uma vez que a valoração tem um tamanho limitado.

Podemos observar que a avaliação de expressões lógicas da forma aqui implementada - utilizando uma pilha para converter a expressão em notação pós-fixa e avaliá-la - tem complexidade de tempo linear em relação ao tamanho da fórmula (O(n)) e complexidade de espaço no pior caso também linear (O(N)) devido ao uso da pilha.

Análise de complexidade das principais funções da avaliação de expressões lógicas:

- Função "avaliar" (na classe ExpressaoLogica):
    - Complexidade de Tempo: A função apresenta um loop *for* inicial, que itera sobre cada caractere da *string* "fórmula". Ao longo desse laço, a função percorre os caracteres da fórmula, processando-os um a um. Dessa forma, temos que a complexidade de tempo é O(n), em que "n" é o tamanho da fórmula. Em seguida, temos a aplicação repetida da função "avaliarProximaOperacao" em loops *while*, dependente de alguns *ifs*, que também está limitada ao tamanho da pilha, logo, O(n).
    - Complexidade de Espaço: A função usa duas pilhas, uma para operadores e outra para variáveis. A quantidade de elementos empilhados nunca excede o tamanho da fórmula, e, portanto, a complexidade de espaço também é O(n). Os loops *while* não apresentam um impacto adicional à complexidade de espaço, uma vez que eles operam nas mesmas pilhas que já foram analisadas na complexidade de espaço original (O(n)).
- Função "avaliarProximaOperacao" (na classe ExpressaoLogica):
    - Complexidade de Tempo: O loop *while* no interior da função tem um tempo de execução que depende da precedência dos operadores. No pior caso, em

que a pilha de operadores deve ser esvaziada, a função tem complexidade de tempo O(n), onde "n" é o tamanho da fórmula.

- Complexidade de Espaço: A função não alocará espaço adicional, e sua complexidade de espaço é O(1), pois não depende do tamanho da fórmula.

O problema da satisfabilidade, por sua vez, utilizando uma árvore binária para representar a fórmula quantificada, tem uma complexidade de tempo que depende do número de combinações possíveis das variáveis quantificadas. O máximo de cinco variáveis quantificadas resulta em uma complexidade exponencial no pior caso (O(2^5) = O(32)). A complexidade de espaço também é afetada pelo número de ramos na árvore, que pode crescer exponencialmente com o aumento das variáveis quantificadas.

Análise de complexidade das principais funções da verificação de satisfabilidade:

- Função "resolve" (na classe Satisfaz):
  - Complexidade de Tempo: A complexidade de tempo da função "resolve" depende do número de nós na árvore de análise da fórmula. Em cada nó, a função avalia operações ou recursivamente chama outros nós. No pior caso, a árvore completa tem 2^n nós (onde "n" é o número de variáveis). Portanto, a complexidade de tempo é O(2^n) no pior caso.
  - Complexidade de Espaço: A função "resolve" mantém várias chamadas recursivas ativas em uma pilha de chamadas. O espaço ocupado na pilha de chamadas é proporcional à profundidade máxima da árvore da fórmula. No pior caso, a profundidade máxima é "n", onde "n" é o número de variáveis. Portanto, a complexidade de espaço é O(n) no pior caso.
- Função "avaliaSatisfaz" (na classe Satisfaz):
  - Complexidade de Tempo: A função "avaliaSatisfaz" chama a função "resolve", que tem complexidade de tempo O(2^n) no pior caso. Portanto, a complexidade de tempo de "avaliaSatisfaz" também é O(2^n) no pior caso.
  - Complexidade de Espaço: A função "avaliaSatisfaz" não aloca espaço adicional significativo, mas chama "resolve", que pode ter uma pilha de chamadas com profundidade máxima de "n". Portanto, a complexidade de espaço é O(n) no pior caso.

Se a restrição de apenas 5 variáveis quantificadas fosse removida, o número de variáveis quantificadas em uma fórmula poderia ser muito maior. Isso afetaria diretamente a complexidade de tempo e espaço da resolução do problema de satisfabilidade.

- Complexidade de Tempo: Sem a restrição de apenas 5 variáveis, a complexidade de tempo se tornaria exponencial em relação ao número de variáveis quantificadas na fórmula. Se houvesse "m" variáveis quantificadas, a complexidade de tempo seria O(2^m) no pior caso. O algoritmo se tornaria impraticável para um grande número de variáveis quantificadas.

- Complexidade de Espaço: A complexidade de espaço permaneceria O(m), onde "m" é o número de variáveis quantificadas. Isso ocorre porque o espaço ocupado pela pilha de chamadas recursivas ainda seria proporcional ao número de variáveis quantificadas. Portanto, a complexidade de espaço ainda seria linear no número de variáveis quantificadas.

Sumarizando, a avaliação de expressões lógicas tem uma complexidade linear em relação ao tamanho da entrada (até O(10ˆ6)), enquanto o problema de satisfabilidade pode ser muito mais custoso computacionalmente (O(2ˆn)), especialmente se mais variáveis forem quantificadas. A escolha da estrutura de dados e algoritmo está relacionada aos problemas e às limitações de entrada.

Estratégias de Robustez

Algumas estratégias de robustez foram implementadas, de modo a garantir que o programa lide com exceções e erros de forma apropriada, evitando falhas inesperadas e tornando-o mais resistente a entradas incorretas ou inesperadas. Destacam-se as seguintes:

- Tratamento de erros e exceções: Foram adicionadas condições de erro em partes mais susceptíveis do código (notadamente aquelas em que há leitura de entrada), para fornecer mensagens de erro adequadas e informativas e garantir que o programa não falhe inesperadamente.

- Validação de entrada: Garantir que os dados de entrada estejam dentro dos limites esperados, verificando o tamanho das strings, por exemplo. O principal ponto crítico é a leitura dos parâmetros na linha de comando, que devem seguir as especificações do TP.

- Testes abrangentes: Alguns testes foram implementados no Makefile (podem ser verificados ao rodar com *make tests*) e o programa foi testado no VPL no Moodle, o que contribui para sua robustez.

- Validação de argumentos: A função *parse_args* realiza algumas validações dos argumentos fornecidos, verificando, por exemplo, se o número de argumentos está correto e se não há strings vazias.

- Uso de constantes para opções e parâmetros críticos: Algumas constantes foram definidas para representar opções e parâmetros críticos, de modo a facilitar a leitura e evitar erros de digitação e "números mágicos" no código.

Análise Experimental

O programa foi avaliado por meio das ferramentas gprof, Valgrind (cachegrind e callgrind) e a biblioteca Memlog (instrumentações retiradas do código final).

A avaliação de vazamentos de memória com o Valgrind não evidenciou problemas.

```
@juliogdomingues → /workspaces/estruturas_de_dados_ufmg/tp01 (main) $ valgrind --leak-check=full ./bin/tp1.out -a "0 | 1" 01
==31866== Memcheck, a memory error detector
==31866== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31866== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==31866== Command: ./bin/tp1.out -a 0\ |\ 1 01
==31866==
1
==31866==
==31866== HEAP SUMMARY:
==31866==     in use at exit: 0 bytes in 0 blocks
==31866==   total heap usage: 8 allocs, 8 frees, 89,754 bytes allocated
==31866==
==31866== All heap blocks were freed -- no leaks are possible
==31866==
==31866== For lists of detected and suppressed errors, rerun with: -s
==31866== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

A avaliação com o Valgrind Cachegrind demonstrou que a maioria das instruções e dados é buscada com sucesso nos caches nível 1, com baixa taxa de miss no cache de último nível:

O relatório do Valgrind Callgrind evidencia que a função ExpressaoLogica::avaliar() foi a principal chamada durante a execução do programa, apresentando alto número de instruções. Tal informação é compatível com o enunciado do problema, em que provavelmente a tarefa de avaliação seria de fato a mais vezes chamada.

```
==27621== Cachegrind, a cache and branch-prediction profiler
==27621== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==27621== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==27621== Command: ./bin/tp1.out -s (\ (\ 0\ &\ 1\ )\ &\ (\ 2\ &\ 3\ )\ &\ (\ 4\
1e1e1e1
==27621==
--27621-- warning: L3 cache found, using its data for the LL simulation.
1 1111111111
==27621==
==27621== I   refs:      3,467,214
==27621== I1  misses:        2,410
==27621== LLi misses:        2,228
==27621== I1  miss rate:      0.07%
==27621== LLi miss rate:      0.06%
==27621==
==27621== D   refs:      1,320,604  (927,180 rd   + 393,424 wr)
==27621== D1  misses:       18,352  ( 15,428 rd   +   2,924 wr)
==27621== LLd misses:       10,441  (  8,501 rd   +   1,940 wr)
==27621== D1  miss rate:       1.4% (    1.7%     +     0.7%  )
==27621== LLd miss rate:       0.8% (    0.9%     +     0.5%  )
==27621==
==27621== LL refs:          20,762  ( 17,838 rd   +   2,924 wr)
==27621== LL misses:        12,669  ( 10,729 rd   +   1,940 wr)
==27621== LL miss rate:        0.3% (    0.2%     +     0.5%  )
```

```
==28071== I   refs:      2,531,677
==28071== I1  misses:        2,202
==28071== LLi misses:        2,066
==28071== I1  miss rate:      0.09%
==28071== LLi miss rate:      0.08%
==28071==
==28071== D   refs:        798,927  (575,866 rd   + 223,061 wr)
==28071== D1  misses:       17,977  ( 15,153 rd   +   2,824 wr)
==28071== LLd misses:       10,361  (  8,487 rd   +   1,874 wr)
==28071== D1  miss rate:       2.3% (    2.6%     +     1.3%  )
==28071== LLd miss rate:       1.3% (    1.5%     +     0.8%  )
==28071==
==28071== LL refs:          20,179  ( 17,355 rd   +   2,824 wr)
==28071== LL misses:        12,427  ( 10,553 rd   +   1,874 wr)
==28071== LL miss rate:        0.4% (    0.3%     +     0.8%  )
```

O relatório do gprof fornece informações sobre a execução de funções no programa, incluindo o número de chamadas e tempo gasto em cada função. Com os testes feitos, verificamos que as funções mais chamadas estão relacionadas à manipulação da pilha de expressões. As funções não consumiram tempo de processamento significativo, o que provavelmente se deve às limitações do tamanho da entrada e eficiência na implementação.

Conclusões

Ao longo desse projeto, foi desenvolvido um programa em C++ que se destina a avaliar expressões lógicas e verificar a satisfabilidade em expressões quantificadas. No decorrer do trabalho, ficou evidente a importância da seleção criteriosa das estruturas de dados para abordar esses desafios complexos da lógica. A escolha das pilhas para a avaliação de expressões e das árvores binárias para a verificação de satisfabilidade reflete a eficiência dessas estruturas e sua adaptabilidade aos problemas em questão. Além disso, a análise da complexidade de tempo e espaço reforçou a otimização das soluções implementadas.

Trata-se de uma aplicação prática das estruturas de dados na solução de problemas lógicos complexos. Ao final, obtivemos uma compreensão mais profunda das estratégias de programação necessárias para abordar esses desafios e sua relevância em contextos mais amplos de desenvolvimento de software.

Bibliografia

1. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms, fourth edition. MIT Press; 2022. ISBN: 9780262046305.

2. Infix, Prefix, and Postfix Expressions [Internet]. São Paulo: Instituto de Matemática e Estatística, Universidade de São Paulo; [data de acesso: 11 de outubro de 2023]. Disponível em: https://panda.ime.usp.br/panda/static/pythonds_pt/03-EDBasicos/09-ExpressoesInfixaPrefixaPosfixa.html

3. Stacks [Internet]. São Paulo: Instituto de Matemática e Estatística, Universidade de São Paulo; [data de acesso: 11 de outubro de 2023]. Disponível em: https://www.ime.usp.br/~pf/mac0122-2002/aulas/stacks.html