BCC202 – Estruturas de Dados I

Departamento de Computação - Universidade Federal de Ouro Preto - MG
Professora: **Karla A S Joriatti** (`www.decom.ufop.br/`)

Trabalho Prático I (TP I) - Alocação Dinâmica, TAD e Recursão

- Submissão com data e hora de entrega disponíveis na plataforma da disciplina. O que vale é o horário do Moodle, e não do *seu*, ou do *meu* relógio!!!

- Clareza, identação e comentários no código também vão valer pontos. Por isso, escolha cuidadosamente o nome das variáveis e torne o código o mais legível possível.

- O padrão de entrada e saída deve ser respeitado exatamente como determinado no enunciado. Parte da correção é automática, não respeitar as instruções enunciadas pode acarretar em perda de pontos.

- Durante a correção, os programas serão submetidos a vários casos de testes, com características variadas.

- A avaliação considerará o tempo de execução e o percentual de respostas corretas.

- Eventualmente serão realizadas entrevistas sobre o trabalho para complementar a avaliação;

- O trabalho é em grupo de até 2 (duas) pessoas.

- Será aceito trabalhos após a data de entrega, todavia com um decréscimo de 0,05 a cada 10min.

- Os códigos fonte serão submetidos a uma ferramenta de detecção de plágios em software.

- Códigos cuja autoria não seja do aluno, com alto nível de similaridade em relação a outros trabalhos, ou que não puder ser explicado, acarretará na perda da nota.

- Códigos ou funções prontas específicas de algoritmos para solução dos problemas elencados não são aceitos

- Não serão considerados algoritmos parcialmente implementados.

- Procedimento para a entrega:.

1. Submissão: via **Moodle**.
2. Os nomes dos arquivos e das funções devem ser especificados considerando boas práticas de programação.
3. Funções auxiliares, complementares aquelas definidas, podem ser especificadas e implementadas, se necessário.
4. A solução deve ser devidamente modularizada e separar a especificação da implementação em arquivos *.h* e *.c* sempre que cabível.
5. Os arquivos a serem entregues, incluindo aquele que contém *main()*, devem ser compactados (*.zip*), sendo o arquivo resultante submetido via **Moodle**.
6. Você deve submeter os arquivos *.h*, *.c* e o *.pdf* (relatório) na raiz do arquivo *.zip*. Use os nomes dos arquivos *.h* e *.c* exatamente como pedido.
7. Caracteres como acento, cedilha e afins não devem ser utilizados para especificar nomes de arquivos ou comentários no código.

- Bom trabalho!

O Problema da Satisfatibilidade Booleana

Uma fórmula booleana é formada a partir de variáveis, as quais podem ser valoradas como **TRUE** ou **FALSE** (1 e 0), parênteses, e conectivos lógicos $\vee$ (**OR**), $\wedge$(**E**), $\neg$ (**NOT**), entre outros. Uma fórmula é dita **satisfazível** se para alguma valoração de suas variáveis a fórmula é **VERDADEIRA**. Por exemplo, a fórmula abaixo é satisfazível, pois a valoração da variável a como **TRUE** torna a fórmula verdadeira.

$$a \vee \neg b \wedge c \tag{1}$$

Baseando-se nesses conceitos, o problema de achar uma valoração para as variáveis que torne uma fórmula verdadeira é conhecido como o **problema da satisfatibilidade booleana**. Esse é um notório problema pertencente à classe de problemas NP-Completo, ou seja, ainda não existe um algoritmo em tempo polinomial que o resolva. É dito que uma fórmula está na **forma normal conjuntiva** quando é uma conjunção ($\wedge$) de disjunções ($\vee$). Além disso, é possível especificar o número de variáveis em uma cláusula, o que é conhecido como **Forma Normal k-Conjuntiva**. Neste trabalho, você deverá apresentar um algoritmo para encontrar valorações para a Forma Normal 3-Conjuntiva (**3-CNF**), onde a quantidade de variáveis em uma cláusula é exatamente 3. Abaixo é possível observar um exemplo de uma fórmula em 3-CNF.

$$(\neg a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (a \vee b \vee \neg c) \tag{2}$$

Para satisfazer uma fórmula em 3-CNF é necessário que todas as suas cláusulas sejam verdadeiras. Portanto, seu trabalho é, dado um conjunto de cláusulas, encontrar uma valoração para as variáveis da fórmula que tornem todas as cláusulas verdadeiras.

Imposições e comentários gerais

Neste trabalho, as seguintes regras devem ser seguidas:

- Seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada)
- Um grande número de *Warnings* ocasionará a redução na nota final.

O que deve ser entregue

- Código fonte do programa em C (**bem identado e comentado**).
- Documentação do trabalho (relatório[1]). A documentação deve conter:
  1. **Implementação:** descrição sobre a implementação do programa. Não faça "*print screens*" de telas. Ao contrário, procure resumir ao máximo a documentação, fazendo referência ao que julgar mais relevante. É importante, no entanto, que seja descrito o funcionamento das principais funções e procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Muito importante: os códigos utilizados na implementação devem ser inseridos na documentação.
  2. **Impressões gerais**: descreva o seu processo de implementação deste trabalho. Aponte coisas que gostou bem como aquelas que o desagradou. Avalie o que o motivou, conhecimentos que adquiriu, entre outros.
  3. **Análise**: deve ser feita uma análise dos resultados obtidos com este trabalho.
  4. **Conclusão**: comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
  5. **Formato:** PDF.

[1]Exemplo de relatório: `https://www.overleaf.com/latex/templates/modelo-relatorio/vprmcsdgmcgd`.

Como deve ser feita a entrega

Verifique se seu programa compila e executa na linha de comando antes de efetuar a entrega. Quando o resultado for correto, entregue via *Moodle* até a data disponível na plataforma de entraga um arquivo **.ZIP** com o nome e sobrenome do aluno. Esse arquivo deve conter: (i) os arquivos *.c* e *.h* utilizados na implementação, (ii) instruções de como compilar e executar via terminal, e (iii) o relatório em **PDF**.

Detalhes da implementação

Para atingir o seu objetivo, você deverá construir um Tipo Abstrato de Dados `Formula` como representação de uma fórmula em 3-CNF. O TAD deverá implementar, pelo menos, as seguintes operações:

1. `criaFormula`: aloca um TAD `Formula`.
2. `destroiFormula`: desaloca um TAD `Formula`.
3. `adicionaClausula`: adiciona uma cláusula lida ao TAD `Formula`.
4. `imprimeFormula`: função que imprime o TAD `Formula` de acordo com o formato de uma fórmula em 3-CNF.
5. `solucaoFormula`: função que tenta encontrar uma valoração para as variáveis da fórmula que a tornem verdadeira. Essa função deve ser implementada de forma **recursiva**, utilizando conceitos de *backtracking*.

Fica a cargo do aluno a implementação do TAD `Formula`, porém é sugerida a criação de um TAD `Clausula` para simplificar a implementação do TAD `Formula`. Caso contrário, as cláusulas dentro do TAD `Formula` devem ser implementadas como um **vetor alocado dinamicamente**.

O TAD deve ser implementado utilizando a separação da interface no *.h* e implementação *.c* discutida em sala, bem como as convenções de tradução. Caso a operação possa dar errado, devem ser definidos retornos com erro, tratados no corpo principal.

O código-fonte deve ser modularizado corretamente em três arquivos: *tp.c*, *formula.h* e *formula.c*. O arquivo *tp.c* deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo *formula.h*. A separação das operações em funções e procedimentos está a cargo do aluno, porém, **não deve haver acúmulo** de operações dentro de uma mesma função/procedimento.

Entrada

A entrada é dada por meio do terminal e é composta de vários conjuntos de teste. Para facilitar, a entrada será fornecida por meio de arquivos. [2] A primeira linha de um conjunto de testes contém dois inteiros N e M que representam, respectivamente, o número de variáveis e o número de cláusulas em uma fórmula. Portanto, as M linhas seguintes contêm, cada uma, a descrição de uma cláusula. Uma cláusula é descrita por três inteiros X, Y e Z que representam as variáveis da cláusula, onde um inteiro negativo é a negação da variável. Cada inteiro deve variar de 1 a 26, representando as letras do alfabeto (a, b, c, ...), podendo ou não estar negado.

$$1 \quad -2 \quad 3 \quad \text{é equivalente a cláusula} \quad a \vee \neg b \vee c \tag{3}$$

Saída

Para cada conjunto de teste da entrada, seu programa deve produzir até $n+1$ linhas de saída caso a fórmula seja satisfazível, onde n é o número de variáveis da fórmula. A primeira linha deve conter a impressão da fórmula no formato 3-CNF. As próximas n linhas devem conter as variáveis e as valorações que tornam a fórmula verdadeira. Caso a fórmula seja insatisfazível, uma mensagem informando ao usuário sobre o problema deve ser apresentada. **Siga exatamente o modelo de saída para impressões, visando que não ocorram erros durante a correção automática.**

Não é preciso apresentar todas as soluções, somente a primeira válida encontrada!

[2] Para usar o arquivo como entrada no terminal, utilize `./executavel < nome_do_arquivo_de_teste`.

Exemplo de um caso de teste

Exemplo da saída esperada dada uma entrada:

Entrada	Saída
3 3 1 2 -3 -1 -2 -3 -1 2 3	Formula: (a ∨ b ∨ ¬c) ∧ (¬a ∨ ¬b ∨ ¬c) ∧ (¬a ∨ b ∨ c) Valoracao: a = True b = True c = False

Entrada	Saída
3 8 1 2 3 1 2 -3 1 -2 3 1 -2 -3 -1 2 3 -1 2 -3 -1 -2 3 -1 -2 -3	Formula: (a ∨ b ∨ c) ∧ (a ∨ b ∨ ~c) ∧ (a ∨ ¬b ∨ c) ∧ (a ∨ ¬b ∨ ¬c) ∧ (¬a ∨ b ∨ c) ∧ (¬a ∨ b ∨ ¬c) ∧ (¬a ∨ ¬b ∨ c) ∧ (¬a ∨ ¬b ∨ ¬c) Formula insatisfazivel!

Como uma fórmula pode ter diferentes valorações que a tornem verdadeira, para manter os resultados obtidos iguais aos dos testes, é indicado que toda variável seja inicialmente valorada como True e posteriormente como False.

Diretivas de Compilação

As seguintes diretivas de compilação devem ser usadas (essas são as mesmas usadas no run.codes).

```
$ gcc -c formula.c -Wall
$ gcc -c tp.c -Wall
$ gcc formula.o tp.o -o exe
```

Avaliação de *leaks* de memória

Uma forma de avaliar se não há *leaks* de memória é usando a ferramenta *valgrind*. O *valgrind* é um *framework* de instrumentação para análise dinâmica de um código e é muito útil para resolver dois problemas em seus programas: **vazamento de memória e acesso a posições inválidas de memória** (o que pode levar a *segmentation fault*). Um exemplo de uso é:

```
1 gcc -g -o exe *.c -Wall
2 valgrind --leak-check=full -s ./exe < casoteste.in
```

Espera-se uma saída com o fim semelhante a:

```
1 ==xxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Para instalar no Linux, basta usar: `sudo apt install valgrind`.

Referências

[1] CORMEN, Thomas H.; LEISERSON, Charles E.. *lgoritmos: Teoria e Prática. 4. ed.* Rio de Janeiro: GEN LTC, 2024. E-book. p.751. ISBN 9788595159914. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788595159914/. Acesso em: 03 out. 2025.