Especificação do Projeto de Introdução à Computação (IF668)

Fernando Castor
Centro de Informática
Universidade Federal de Pernambuco

3 de junho de 2018

1 Introdução

O objetivo deste trabalho é praticar a escrita de funções e programas em Javascript, fazendo com que os alunos de primeiro período do curso de Ciência da Computação ganhem alguma familiaridade com uma linguagem bastante útil na prática e que, apesar da semelhança sintática, tem diferenças consideráveis com relação a Java. Outro objetivo do projeto é apresentar aos alunos o problema SAT e a noção de intratabilidade. SAT é um dos problemas mais importantes da computação e a experiência de implementar uma solução, ainda que ineficiente, para tal problema, é bastante enriquecedora.

Você deve gastar pelo menos uma hora lendo este documento para se certificar de que entendeu completamente o que é pedido, de forma a não perder tempo com idas e vindas ao documento e ter certeza de que de fato está entregando o que está especificado. Leia atentamente o texto, destacando partes importantes, anotando as dúvidas para saná-las o quanto antes. Além disso, para entender este projeto, será necessário ler documentação adicional indicada a seguir. A leitura de tal documentação é **fundamental**.

Ao longo do documento, as tarefas que você **tem que** realizar são indicadas da seguinte maneira:

Tarefa 0: *ler o documento com bastante cuidado.*

2 Visão Geral do Projeto

Neste projeto, você deverá construir um programa capaz de verificar se uma fórmula escrita em lógica proposicional é **satisfatível**. Uma fórmula lógica é satisfatível se é possível encontrar uma atribuição de valores `V` e `F` para suas variáveis de modo a tornar essa fórmula lógica verdadeira. A seguir são apresentadas duas fórmulas, uma satisfatível e outra não-satisfatível:

$$(A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (B \vee \neg A \vee \neg C) \quad (1)$$

$$(\neg A \vee \neg C) \wedge (A \vee \neg B) \wedge (B \vee C) \wedge (\neg B \vee C) \wedge (B \vee \neg C) \quad (2)$$

A primeira é satisfatível se a A, B e C forem atribuídos os valores `F`, `V` e `F`, respetivamente. Há outras atribuições que também tornam a fórmula verdadeira. Já para a segunda, não há atribuição de valores que torne a fórmula verdadeira (verifique isso!). Este problema é conhecido como SAT e é um problema difícil na prática. Programas cujo objetivo é resolver instâncias de SAT são conhecidos como SAT *solvers*. SAT é o que chamamos de um problema **NP-completo**. Para saber mais sobre problemas NP-completos, você deve **obrigatoriamente** ler os dois textos referenciados abaixo. O primeiro é um email bastante didático escrito pelo professor Jorge Stolfi do IC-UNICAMP enquanto o segundo é um tutorial sobre problemas NP-completos escrito pelo professor Paulo Feofiloff do IME-USP. Não prossiga com a leitura deste documento enquanto não tiver **lido os dois textos**.

- `https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/stolfi-on-NPcomplete.html`
- `https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html`

A fórmula a ser verificada será lida a partir de um arquivo texto (com extensão `.cnf`) codificado de maneira pré-definida. As fórmulas contém apenas variáveis lógicas e os operadores AND ($\wedge$), OR ($\vee$) e NOT ($\neg$), sem implicação. As fórmulas estão na forma normal conjuntiva (*Conjunctive Normal Form* – CNF[1]). Isso significa que estão organizadas em cláusulas onde variáveis lógicas (negadas ou não) são conectadas por operadores OR. As cláusulas, por sua vez, são conectadas apenas por operadores AND. Não há limite para a quantidade de variáveis em cada cláusula nem para a quantidade de cláusulas na fórmula. As duas fórmulas apresentadas anteriormente estão na CNF. A fórmula abaixo também está:

[1] https://en.wikipedia.org/wiki/Conjunctive_normal_form

$$(\neg A \vee \neg C \vee D \vee \neg E) \wedge (A \vee \neg B \vee F) \wedge (B \vee C) \wedge (\neg D) \tag{3}$$

O projeto deverá ser desenvolvido na linguagem Javascript e rodar no Node.js[2]. Não é permitido implementar o projeto em nenhuma outra linguagem. Para desenvolver o projeto, você deverá usar como base o arquivo `sat.js`, disponível no repositório

- `https://github.com/fernandocastor/sat`

Tal arquivo fornece a estrutura geral que seu SAT solver deve apresentar. As funções que se encontram nesse arquivo, conforme indicam os comentários, precisam ser implementadas. No repositório também estão disponíveis exemplos que podem ser usados para testar seu SAT *solver*.

Tarefa 1: *obter os arquivos* `sat.js` *e os vários exemplos em formato* `.cnf` *a partir do endereço* `https://github.com/fernandocastor/sat`

3 Formato da Entrada

Abaixo é apresentado um exemplo de como é a entrada a ser processada pelo seu SAT *solver*.

```
c Exemplo simples de fórmula lógica na CNF.
p cnf 5 5
c Satisfatível
1 2 3 4 5 0
-1 -2 -3 -4 -5 0
1 -2
3 -4 5 0
5 0
-1 2 0
```

Este é um arquivo `.cnf`. Linhas que começam com '`c`' são comentários e devem ser ignorados. Além disso, cada arquivo `.cnf` deve conter uma linha de 'problema'. Sua sintaxe é

```
p cnf #vars #clausulas
```

onde `#vars` é o número de variáveis e `#clausulas` é o número de cláusulas. Para esse exemplo, a linha de problema é `p cnf 5 5`, o que significa que a fórmula que ele representa tem cinco cláusulas e cinco variáveis. Como explicado na Seção 2, uma fórmula na CNF (*Conjunctive Normal Form*) consiste em cláusulas conectadas por operadores lógicos $\wedge$. Cada cláusula, por sua vez, consiste em sequências de variáveis, negadas ou não, conectadas por operadores lógicos $\vee$ (veja os exemplos na Seção 2). Devido a essa estrutura, arquivos `.cnf` estão organizados da seguinte maneira:

1. Variáveis são representadas por números consecutivos começando em 1. O exemplo anterior tem cinco variáveis: 1, 2, 3, 4 e 5. Números negativos representam variáveis que estão aparecendo negadas (o operador '$-$' representa o '$\neg$').

2. Cláusulas são sequências de variáveis, negadas ou não, finalizadas pelo número 0. Usualmente, cláusulas são organizadas uma por linha, mas não necessariamente, como mostrado no exemplo acima, onde a cláusula `1 -2 3 -4 5` está quebrada em duas linhas.

3. Não é necessário indicar explicitamente o '$\wedge$' ou '$\vee$'. Variáveis em uma mesma cláusula são conectadas implicitamente por '$\vee$' enquanto cláusulas diferentes são conectadas implicitamente por '$\wedge$'.

Tarefa 2: Implemente a função `readFormula`. Essa função deve receber o nome de um arquivo `.cnf` e devolver um objeto com dois atributos, `clauses` e `variables`. O primeiro deve guardar um array de cláusulas, onde cada cláusula é ela própria representada como um array (que pode ser de números ou de strings). Já o atributo `variables`, como o nome indica, deve guardar um array contendo uma quantidade de posições igual ao número de variáveis, onde a posição 0 corresponde à variável 1, a posição 1 à variável 2, a posição 2 à variável 3, etc. O valor guardado em cada posição pode ser uma atribuição inicial de valores a essas variáveis, por exemplo, 0 ou `false` (indicando F). Além disso, essa função deve ler a linha de problema da fórmula e verificar se essa fórmula realmente tem as quantidades de variáveis e cláusulas especificadas na linha de problema. Para o exemplo acima, essa função pode devolver um objeto como o seguinte:

[2] https://nodejs.org

```
{ clauses: [[ '1', '2', '3', '4', '5' ],
  [ '-1', '-2', '-3', '-4', '-5' ],
  [ '1', '-2', '3', '-4', '5' ],
  [ '5' ],
  [ '-1', '2' ]],
  variables: [0, 0, 0, 0, 0]
}
```

Para realizar esta tarefa, sugere-se que sejam criadas três funções auxiliares: `readClauses`, que extrai as cláusulas do arquivo `.cnf` e produz um array como o armazenado no atributo `clauses` acima, `readVariables`, que processa as cláusulas e identifica todas as variáveis (ignorando se aparecem negadas ou não) e produz como resultado um array como o armazenado no atributo `variables` acima, e `checkProblemSpecification`, que checa se as cláusulas e variáveis batem com a linha de problema.

4 Atribuindo Valores

Este SAT *solver* deve resolver o problema SAT usando uma estratégia de força bruta. Isso significa que deve testar todas as possíveis atribuições de valores `V` e `F` às variáveis da fórmula, verificando para cada uma delas se o resultado da avaliação da fórmula é verdadeiro. Essa estratégia não é nada eficiente, mas tem a vantagem de ser simples de implementar. Para esta seção, uma função a mais precisará ser implementada. Ela é responsável por gerar uma atribuição de valores booleanos para as variáveis da fórmula. Uma atribuição é um array onde cada posição corresponde a um valor (`V` ou `F`) atribuído a uma variável. A maneira como os valores `V` e `F` são representados é deixada a seu critério. No exemplo da seção anterior, usamos o número 0 para representar `F`, mas seria igualmente válido usar `false`.

Tarefa 3: Implemente a função `nextAssignment`. Essa função recebe como entrada o array contendo a atribuição atual de valores às variáveis da fórmula e devolve um novo array (ou o mesmo array, modificado) com uma nova atribuição de valores a essas variáveis. Se você perceber que fazer a atribuição de valores *in-place*[3] pode ser mais eficiente, você pode adotar essa estratégia (ao invés de devolver uma nova atribuição). A função `nextAssignment` deve funcionar de modo que sempre produza atribuições diferentes de valores, para um determinado número de variáveis. Observe o exemplo a seguir, usando a linha de comando do Node.js para ilustrar e com os valores `V` e `F` representados por `true` e `false`, respectivamente:

```
> nextAssignment([false, false])
[false, true]
> nextAssignment([false, true])
[true, false]
> nextAssignment([true, false])
[true, true]
> nextAssignment([true, true])
[false, false]
```

Neste exemplo, todas as possíveis combinações de valores booleanos para esse array representando uma fórmula com apenas duas variáveis foram geradas.

5 Verificando a Satisfabilidade

Uma vez munidos das cláusulas, das variáveis e de uma atribuição de valores para estas últimas, podemos verificar se a fórmula é ou não satisfeita por essa atribuição de valores às suas variáveis, ou seja, se a avaliação da fórmula produz `V` ou não. Se for o caso, o programa termina, informa o usuário que a fórmula é satisfeita e qual foi a atribuição de valores identificada. Caso contrário, obtém a próxima atribuição usando `nextAssignment` e repete o procedimento até que a fórmula seja satisfeita ou todas as atribuições sejam verificadas. Nesta segunda situação, o programa informa o usuário que a fórmula não é satisfatível.

Tarefa 4: Implemente a função `doSolve`. Essa função recebe como entrada o array contendo a atribuição inicial de valores às variáveis da fórmula e outro array contendo as cláusulas. A função deve funcionar em um `loop` onde, enquanto a fórmula não for satisfeita nem todas as atribuições tiveram sido testadas, ela verifica se a atribuição atual satisfaz ou não a fórmula e, se não satisfizer, pega a próxima atribuição de valores e tenta novamente. Seu resultado é um objeto com dois atributos, (i) `isSat`, que guarda um valor booleano indicando se a fórmula foi ou não satisfeita, e (ii) `satisfyingAssignment`, que guarda `null` se a fórmula não foi satisfeita ou um array contendo a atribuição que a satisfaz, caso contrário.

[3] Ou seja, modificando diretamente o que já está no array recebido como argumento, ao invés de criar um novo e devolvê-lo.

6 Agrupando Tudo e Otimizações

Seu programa deve ser organizado como um módulo que oferece externamente apenas uma função, chamada `solve`. O esqueleto do projeto que está disponível no repositório já está organizado desta forma. É necessário, porém, implementar a função `solve` para que utilize as outras funções que você construiu.

Tarefa 5: Implemente a função `solve`. Esta função deve receber como argumento o nome do arquivo `.cnf` a ser processado e fornecer como resultado o mesmo objeto produzido pela função `doSolve` (Seção 5). Ela deve lançar mão das funções especificadas nas tarefas anteriores para realizar seu trabalho.

As Tarefas 0-5 consistem na parte básica do projeto. Há dois desafios adicionais. Recomenda-se que cada equipe tente fazer pelo menos um deles.

Desafio 1: Modifique seu programa para que a atribuição de valores às variáveis seja incremental e que apenas partes que mudaram na atribuição sejam refeitas. Desta forma, evita-se atribuir novamente os valores de todas as variáveis quando apenas algumas poucas mudam de valor entre duas atribuições subsequentes. Mostre que (se?) o desempenho do seu programa mudou em decorrência dessa otimização. Se possível, tente gerar novamente apenas a parte da atribuição que muda. Indique claramente as partes do seu programa que precisaram mudar para satisfazer esse requisto. Uma dica: é mais fácil cumprir este desafio se a função `doSolve` for recursiva do que se ela for iterativa.

Desafio 2: Traduza para Javascript o SAT *solver* desenvolvido por Sahand Saba, fortemente baseado no programa SAT0W desenvolvido por Donald Knuth[4]. O *solver* de Saba, escrito em Python, está disponível no endereço

`http://sahandsaba.com/understanding-sat-by-implementing-a-simple-sat-solver-in-python.html`

Sua tradução deve incluir uma função `solve` que funciona conforme a função da Tarefa 5, em termos de entradas e resultado, mas pode ser implementada como achar mais conveniente.

7 Avaliação e Logística

O projeto representará 50% da nota de cada aluno na disciplina. Não entregar o projeto implica imediatamente em nota 0 (ZERO) para os membros da equipe que não entregou. A avaliação se dará através de apresentação do projeto desenvolvido para o professor e para os monitores. Ausência no momento da apresentação do projeto implica em nota 0 (ZERO) para todos os membros faltosos da equipe, exceto para os casos previstos para situações onde é aceitável solicitar segunda chamada.

A nota máxima que pode ser obtida no projeto é igual a 10 * N, onde N é o número de membros da equipe. Essa nota é atribuída pelo professor principalmente com base na apresentação e a distribuição dos pontos entre os membros da equipe (caso haja mais de um) é de responsabilidade dos mesmos. A nota máxima que cada membro da equipe pode obter é igual a 10. Se a equipe não realizou nenhum dos desafios, a nota máxima para cada membro é 9,0. Se a equipe se recusar a realizar a distribuição dos pontos, automaticamente a nota 0,0 (zero) é atribuída aos membros da equipe.

Algumas considerações adicionais, relativas à logística do projeto:

- O trabalho poderá ser feito em grupos de no máximo dois componentes
- O código completo do programa deverá estar disponível em um repositório no Github (`http://www.github.com`). Deve ser enviado um **email** para o professor com assunto **PROJETO IF668** até as **23h59 do dia 24/06/18** informando o endereço do repositório. Programas não entregues no horário serão penalizados com a perda de pontos, à taxa de 2,0 pontos perdidos para cada 24h de atraso. Essa data pode ser modificada. Se for o caso, os alunos serão avisados com antecedência.
- Trabalhos copiados de colegas ou da internet, sejam trechos ou a totalidade, serão prontamente anulados (todos os envolvidos).
- A apresentação dos projetos será no **dia 25/06 impreterivelmente**. Todos os membros devem saber todos os detalhes da implementação. O professor escolherá na hora a quem dirigirá cada pergunta durante a apresentação. Novamente, essa data pode ser modificada.

[4] `http://www-cs-faculty.stanford.edu/~knuth/programs/sat0w.w`