INSTITUTO DE MATEMÁTICA E ESTATÍSTICA - IME
UNIVERSIDADE DE SÃO PAULO - USP

TRABALHO DE FORMATURA SUPERVISIONADO

Otimização de suítes de Teste através da ferramenta AdaptSuite

Autor: André Pestana
andre.pestana@usp.br
Orientador: Prof. Dr. Alfredo Goldman
Co-Orientador: Renan de Melo Oliveira

São Paulo, Fevereiro de 2016

They say that time solves everything. The question is: How much time?

Alice in Wonderland

Resumo

A realização de testes de sistemas, embora seja algo crucial, é reconhecida como uma atividade cara. Sistemas grandes tendem a possuir muitos testes e testá-los em sua totalidade, além de custoso, é demorado. A proposta desse trabalho é a criação da ferramenta AdaptSuite, capaz de escolher o melhor conjunto de testes para rodar dentro de um tempo limite, baseada em parâmetros. Tal abordagem tem com objetivo possibilitar uma rápida detecção de erros, assim como a confirmação sobre a correção deles, garantindo a qualidade do software sem que seja necessária a execução imediata de todos os testes a cada alteração.

Sumário

Lista de Figuras

1 Introdução

Quando se desenvolve um sistema, é necessário garantir que ele não apresenta falhas e realiza corretamente as funções a que fora designado. Tal processo recebe o nome de validação [1].

O problema que essa etapa de validação apresenta é que o número de testes necessários para se validar um sistema cresce com seu tamanho [2]. Para sistemas muito grandes ou prestes a serem lançados, a execução completa dos testes pode demandar muito tempo.

Uma alternativa para agilizar o processo de validação é usando *Testes Automatizados*, que nada mais são do que programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos [3]. Porém, embora sejam mais rápidos, executar todo o conjunto de testes automáticos ainda pode ser algo demorado [4]. A solução proposta nesse projeto se encaixa nesse contexto, pois ela irá executar apenas um subconjunto de testes automatizados para serem executados dentro de um tempo limite.

Quando uma parte do código é modificada ou, quando se insere algo novo, é necessário verificar através dos testes se essas alterações continuam permitindo que o programa seja executado normalmente [4]. Entretanto, rodar todos os testes pode ser demorado [5]. Além do mais, nem todos os testes interagem com o código modificado, tornando desnecessário sua execução no processo de validação da alteração.

Em uma abordagem mais tradicional, em que grande parte dos testes é deixada para o final [1], a preocupação se encontra em realizar a validação de forma eficiente. Acontece que, as etapas anteriores de especificação e programação costumam atrasar e gastar mais recursos do que inicialmente planejado [6]. Nesse cenário, quando a etapa de validação do sistema chega, o tempo e o dinheiro disponíveis são limitados [4], fazendo com que, mesmo que se deseje garantir a qualidade total do software, a total execução dos testes se torne algo potencialmente inviável.

A correção de *bugs* encarece conforme as etapas do projeto avançam [1]. Em particular, corrigir erros quando o sistema já está em produção é muito caro (Figura1).

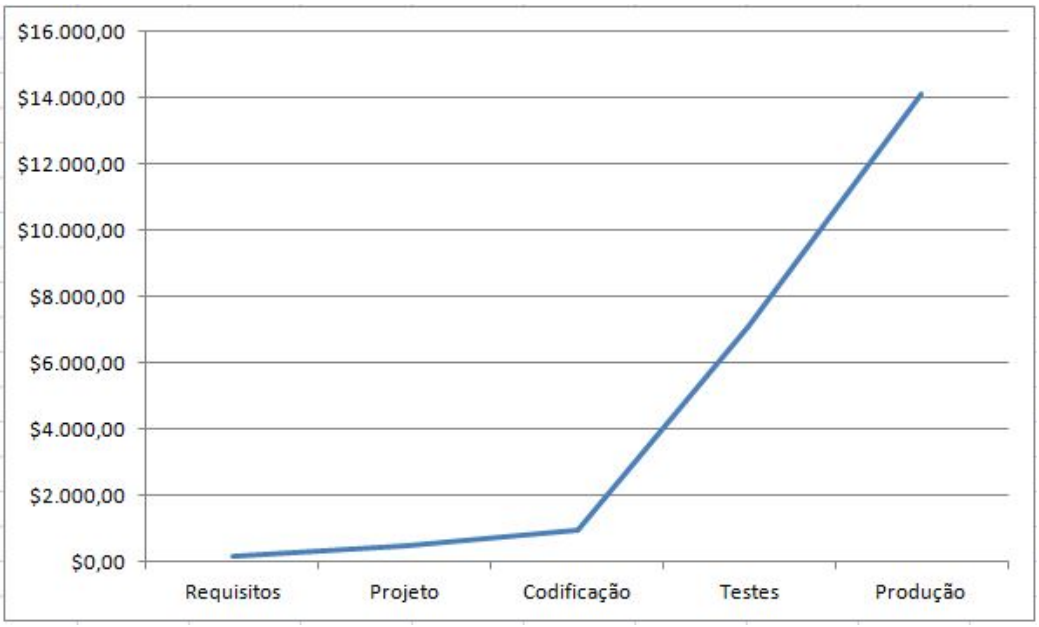

Figura 1: Custo de correção de um *bug* de acordo com a etapa do processo (adaptado de Pressman [1])

Mesmo quando se pensa em projetos construídos usando uma metodologia ágil, em que os testes são uma parte fundamental do desenvolvimento do sistema e não apenas uma etapa final de validação, rodar todos os testes ainda pode ser custoso.

Basta pensar na curva de adoção de tecnologia de Rogers [7] (Figura2) que descreve como um produto, inicialmente, é usado por poucos "inovadores"(*Early Adopters*) até que seu valor seja reconhecido e o número de usuários cresça drasticamente (*Early Majority*). O problema desse crescimento repentino é que a manutenção e a detecção de erros de um sistema são proporcionais ao número de usuários [1].

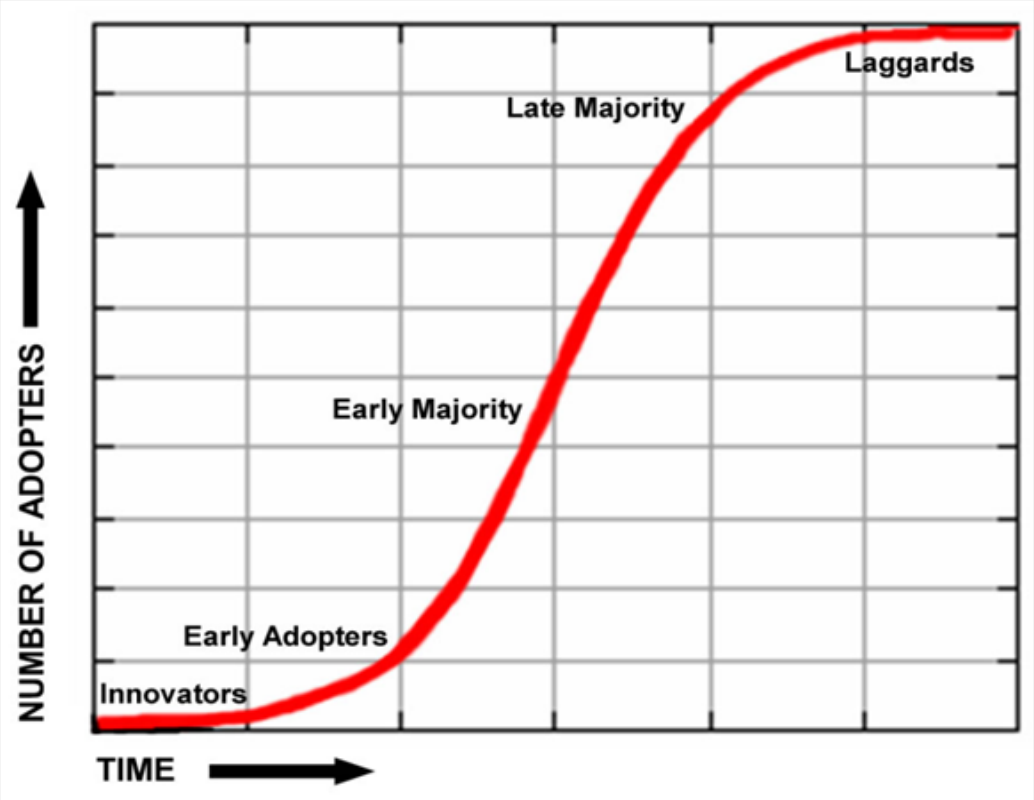

Figura 2: Curva de adoção de tecnologia descrita por Rogers [7]

Nesse cenário, a equipe possui menos tempo e menos recursos, tanto humano quanto financeiro, para os testes. Além do mais, quando se corrige um *bug*, o ideal seria rodar apenas os testes pertinentes ao código alterado, seguindo a ideia de desenvolvimento ágil.

Tendo em vista os cenários apresentados acima, fica claro que testes de sistema, ao mesmo tempo em que são essenciais, podem também ser caros e demorados. Dada a motivação apresentada, esse trabalho propõe-se a criar uma forma para agilizar o processo de validação com a criação de uma ferramenta Java, de nome AdaptSuite, que consiga encontrar o melhor conjunto de testes JUnit [8] para serem executados em um tempo limite determinado pelo usuário.

Existem várias formas para a escolha do conjunto adequado de testes. A solução proposta determina esse conjunto resolvendo o Problema da Mochila [12], tendo em vista que cada teste possui um valor e um tempo de execução.

Esse trabalho segue a seguinte estrutura: primeiro serão mostrados outros métodos existentes que tentam resolver o problema de encontrar um

conjunto de testes adequado. Em seguida, será apresentado o algoritmo do problema da mochila e como ele foi usado para resolver o problema proposto. Após a explicação do algoritmo, o trabalho mostrará como usar a ferramenta AdaptSuite na prática. Por fim, a ferramenta será comparada com outras duas soluções, a saber, a solução gulosa e a solução que escolhe os testes aleatoriamente, sendo apresentados os resultados e as conclusões finais.

Durante o trabalho, o termo suíte ou suíte de testes será usado como substantivo coletivo de testes, conforme definido pelo JUnit [8].

2 Soluções Existentes

O problema que a ferramenta AdaptSuite tenta resolver é conhecido pelo nome *Test Case Priorization* [9] ou pela sigla *TCP*. Nessa área são estudadas técnicas que tentam otimizar a fase de validação, escolhendo um subconjunto de testes de um programa, mas sem que se perca a eficiência na detecção de erros [5] [4] [9] [10].

Uma área relacionada à *TCP* é a de *Fault Localization* [10]. Essa área tenta encontrar a exata localização do erro no código usando os resultados dos testes. Como uma técnica de *TCP* sempre executa um subconjunto dos testes disponíveis, pode-se querer saber se esse subconjunto consegue encontrar a exata localização do erro de forma eficiente usando uma técnica de *Fault Localization* [10]. Porém, esse trabalho não tem como objetivo testar a eficiência da AdaptSuite em um cenário de *Fault Localization*.

Na área de *TCP*, as técnicas usadas para se resolver esse problema baseiam-se em dois parâmetros: o número de falhas ou na cobertura de um determinado teste, mas raramente os dois ao mesmo tempo [5] [9]. O AdaptSuite, por sua vez, foi desenvolvido visando usar esses dois parâmetros ao mesmo tempo, o que o torna diferente dos métodos tradicionais de "Test Case Priorization".

Como exemplo de metodologia baseada apenas no histórico de detecção de falhas, tem-se o "Average of the Percentage of Faults Detected"(APFD)[11], definida como:

Definição Seja T uma suíte com n testes que detecta m falhas. Define-se TFi como o número do teste, em ordem de execução, que revelou a i-ésima falha. A $APFD$ de T é definida como:

$$APFD(T) = 1 - \frac{TF1 + TF2 + ... + TFm}{nm} + \frac{1}{2n}$$

Assim, uma suíte de testes ótima é aquela que maximiza o valor da APFD[5].

Quando se trata de uma metodologia baseada apenas na cobertura do código, é importante definir a granularidade da técnica, ou seja, qual o nível de cobertura em questão. Uma técnica de granularidade fina usa os valores de cobertura em linhas de código. Por sua vez, técnicas de granularidade grossa usam a cobertura com base no número de classes executadas [10].

A *Total Function Test Case Priorization* é um exemplo de técnica baseada em cobertura que usa granularidade grossa. Essa técnica ordena em ordem decrescente os testes de acordo com seus valores de cobertura e os escolhe até a suíte estar completa [10].

Além de usar os dois parâmetros na escolha da suíte ótima, a AdaptSuite se diferencia dos tradicionais modelos de *TCP* por usar um tempo limite máximo para o tamanho da suíte. Poucas técnicas propostas abordam o *TCP* usando um tempo máximo para a suíte ótima [4].

3 Problema da Mochila

3.1 Descrição do Problema

Como já mencionado, o AdaptSuite possui um recurso de capacidade finita, que no caso, é o tempo máximo para se rodar os testes, em milissegundos, e deve preenchê-lo escolhendo dentre uma quantidade finita de opções, os testes, cada qual com seu próprio valor e tempo de execução. A abstração desse problema nada mais é do que um clássico problema matemático de otimização: o *Problema da Mochila* [12].

Esse problema pode ser definido como uma combinação de decisões binárias cujo objetivo é maximizar o valor do conteúdo da mochila. Cada decisão binária se resume a atribuir um valor 1 ou 0 para uma determinada variável, ou, no caso do Problema da Mochila, decidir se determinado item é ou não escolhido. Portanto, o mesmo é representado pelo seguinte problema de programação linear inteira [12]:

maximize $\sum_{j=1}^{n} p_j * x_j$

sujeito à $\sum_{j=1}^{n} w_j * x_j \leq C$

em que, $x_j \in \{0, 1\}$, n = número de itens aptos a serem escolhidos, $w_j \geq 0$ o peso do item j e $p_j \geq 0$ o valor do item j.

A definição formal do problema é dada a seguir [12]:

Definição Considere um conjunto de N itens j, cada qual com um valor $p_j \geq 0$ e peso $w_j \geq 0$, e capacidade C. Deseja-se escolher um subconjunto de N tal que o valor total desse subconjunto seja máximo e o peso total não exceda C.

A primeira conclusão que se pode tirar dessa definição é que se todos os itens couberem na mochila, a solução ótima consiste em escolher todos os itens.

O algoritmo usado pelo AdaptSuite baseia-se na resolução do *Problema da Mochila* usando programação dinâmica que nada mais é do que uma técnica para se encontrar a resolução de um problema grande encontrando a solução ótima de vários subproblemas derivados do problema original [12]. Para isso, foi considerada a seguinte propriedade:

Propriedade 3.1. *Considere a solução ótima para o problema da mochila. Caso um item r seja removido, o subconjunto restante será a solução ótima para o subproblema definido por* $C' = C - w_r$ *e* $N \backslash \{r\}$.

Com essa definição, a solução usando programação dinâmica do problema da mochila é dada a seguir:

Assuma que a solução ótima já foi calculada para um subconjunto de capacidade Y. Adiciona-se mais um i nesse subconjunto e verifica-se se a solução precisa ser atualizada usando a seguinte recorrência:

moch[i,Y] = moch[i-1, Y] se $w_i > Y$
moch[i,Y] = max{moch[i-1, Y], moch[i-1, Y -w_i] + p_i} se $w_i < Y$

O caso $w_i > Y$ indica que a sub-mochila atual é muito pequena para conter i, portanto, a solução ótima para esse subproblema é a sub-mochila atual.

No caso $w_i < Y$, o item de fato cabe na sub-mochila. Nesse caso, deve-se comparar as seguintes possibilidades e escolher aquela que apresentar um valor maior:

- i não entra na mochila e o valor da solução atual se mantém.
- i entra na mochila, mas diminui a capacidade para os $\{1, 2, ..., i-1\}$ itens restantes de Y para $Y - w_i$. Obviamente, essa capacidade restante deve ser preenchida da melhor forma possível.

Essa recorrência produz uma matriz, representada na recorrência por *moch*, com número de linhas igual ao número de elementos disponíveis e número de colunas igual à capacidade máxima da mochila. Um elemento mochi[i,Y] indica o valor da mochila ótima considerando o subconjunto $\{1, 2, i\}$ e capacidade Y.

3.2 Aplicando o Problema da Mochila ao Problema

Com a explicação formal do *Problema da Mochila*, pode-se então definir o problema que a AdaptSuite tenta resolver da seguinte maneira:

Definição Considere N testes a serem escolhidos. Seja V_T o peso de um teste (a ser definido no próximo capítulo) T, Δ_T o tempo da última execução desse mesmo teste e C o tempo máximo para se rodar todos os testes. A suíte ótima contém $k \leq N$ testes tais que $\Delta_{T1} + \Delta_{T2} + ... + \Delta_{Tk} \leq C$ e $V_{T1} + V_{T2} + ... + V_{Tk}$ é máximo para qualquer outro subconjunto de N.

Existe a possibilidade da ferramenta rodar o chamado "caso trivial"[12], no qual o mesmo irá simplesmente rodar todos os testes disponíveis, sem realizar o cálculo do problema da mochila. O sistema irá executar o chamado "caso trivial"quando:

- For a primeira vez que o programa é executado. Nesse cenário, é necessário rodar o caso trivial pois a ferramenta não tem nenhuma informação sobre os conjunto de testes. No final da primeira execução a ferramenta irá armazenar as informações importantes e irá usá-las nas próximas execuções.
- O tempo limite para a execução dos testes for maior do que a soma da execução de todos os testes. Nesse caso, se a mochila tem capacidade de armazenar todos os itens, a mochila ótima é obtida coletando todos os itens.

3.3 Algoritmo

A implementação da ferramenta para o algoritmo descrito acima é dada a seguir [13]:

```
private void testsValue (List<TestData> testData, int N, Long C) {

   Double a, b;

   for (int Y = 0; Y <= C; Y++) {
      //Matriz que armazena os valores das sub-mochilas otimais.
      moch[0][Y] = 0.0;
      for (int i = 1; i <= N; i++) {
         //Valor da mochila atual.
         a = moch[i-1][Y];
```

```
            // Definindo o tempo do teste.
            Long lastExecutionY =
                testData.get(i-1).getLastExecutionY();

            //Definindo variáveis para o cálculo do peso do teste.
            Long testFailures = testData.get(i-1).getFailures() > 0 ?
                testData.get(i-1).getFailures() : 1L;
            Double coverage = testData.get(i-1).getLineCoverage() > 0
                ? testData.get(i-1).getLineCoverage() : 1.0;
            Long lastExecution = testData.get(i-1).getLastExecuted() >
                0 ? testData.get(i-1).getLastExecuted() : 1L;
            Double frequency = testData.get(i-1).getFrequency() > 0 ?
                testData.get(i-1).getFrequency() : 1.0;
            Double failFrequency =
                testData.get(i-1).getFailFrequency() > 0 ?
                testData.get(i-1).getFailFrequency() : 1.0;

            //Caso 1 da recorrência
            if (lastExecutionY.intValue() > Y)
               b = 0.0;
            //Caso 2 da recorrência
            else
               b = moch[i-1][Y - lastExecutionY.intValue()] +
               ( testFailures.doubleValue() * coverage *
                   lastExecution.doubleValue() * frequency);

            //Escolhendo a sub-mochila otimal para o subproblema de
                tamanho i
            moch[i][Y] = Max(a, b);
         }
      }
   }
```

Esse algoritmo necessita de um espaço para a matriz de tamanho $N * C$ e sua complexidade é $\Theta(NC)$ [13].

Entretanto, essa solução apenas calcula o valor da mochila ótima. Para saber quais itens fazem parte dessa mochila, é necessário o uso de mais um algoritmo:

```
private void chooseTests (List<TestData> testData, int N, Long C) {

   Long Y = C;
```

```
   for (int i = N; i >= 1; i--)
      /*Mochila considerando o item é igual
      * à mochila desconsiderando o item =>
      * item não está na mochila otimal */
      if(knaspackTabble[i][Y.intValue()] ==
          knaspackTabble[i-1][Y.intValue()])
         chosenTests[i-1] = false;
      /*Mochila considerando o item é diferente
      * da mochila desconsiderando o item =>
      * item está na mochila otimal */
      else {
         Long executionTime =
             testData.get(i-1).getLastExecutionTime();
         chosenTests[i-1] = true;
         Y -= executionTime;
      }
}
```

Esse algoritmo verifica se a mochila ótima levando em consideração um item i é igual a submochila ótima sem esse item. Caso afirmativo, significa que o mesmo não faz parte da mochila ótima. Caso negativo, esse item faz parte da mochila ótima e na próxima iteração, a capacidade a ser considerada será a capacidade atual Y subtraído do peso w_i do teste que faz parte da mochila ótima [12] [13]

Para se saber todos os itens da mochila ótima, basta começar a comparação com o último item e com a capacidade máxima. Nesse caso, o algoritmo necessita de um espaço para armazenar os itens escolhidos de tamanho N e a complexidade de execução é $\Theta(N)$ [13].

4 Peso do Teste

Como mencionado, o AdaptSuite usa o histórico de falhas e a cobertura do teste para determinar sua qualidade. Além desses valores, a ferramenta ainda faz uso de outras variáveis referentes ao histórico de execuções do teste. A descrição de todas as variáveis armazenadas pela ferramenta é dada a seguir.

4.1 Quantidade de Falhas

Os testes possuem uma propriedade relacionada às falhas ou erros ocorridos durante uma execução. Segundo a definição dada pelo JUnit[8], falhas ocorrem quando o valor da operação difere do valor esperado em um comando

assertion. Os erros, por sua vez, acontecem quando o teste não consegue terminar ou quando exceções são lançadas durante a execução.

O valor inicial dessa propriedade P é 1 e para cada execução em que ocorre um número de falhas $F \geq 1$ ou um número de erros $E \geq 1$, atualiza-se P como $P = P + E + F$.

4.2 Cobertura do Código

O AdaptSuite obtém esses valores de relatórios em formato html gerados pela ferramenta Eclemma [14]. Desse relatório, o valor de cobertura é calculado pela razão entre o número de linhas executadas pelo teste e o número de linhas total do programa.

Caso um teste não possua um relatório gerado pelo Eclemma, a AdaptSuite sempre irá atribuir valor de cobertura igual à 100%. Por esse motivo recomenda-se que todos os testes possuam um relatório de cobertura antes de usar a AdaptSuite.

4.3 Última Execução

Esse parâmetro se refere à última vez que o AdaptSuite escolheu o teste. Caso o teste seja escolhido, o valor dessa variável é zerado. Após a execução da suíte de testes, o programa irá somar 1 ao valor dessa variável em todos os testes.

4.4 Frequência

Por fim, a ferramenta também leva em conta a frequência com que um determinado teste é executado. Quanto menor o valor dessa variável, maior a chance do teste correspondente ser escolhido.

4.5 Prioridades

Para cada uma das variáveis descritas acima, o usuário pode passar uma prioridade de 0 a 5, sendo que quanto maior esse valor, maior será a prioridade da variável associada.

Caso uma variável receba prioridade 0, ela será desconsiderada no cálculo do peso do teste. Caso nenhum valor seja definido, será atribuído prioridade 1 para todas as variáveis.

4.6 Cálculo

Utilizando os valores das variáveis descritas acima, bem como as prioridades das mesmas, a definição de "peso"de um teste usada pela ferramenta é dada logo abaixo:

Definição Seja T um teste qualquer, E_T a quantidade de erros referente ao teste T, C_T sua cobertura, L_T a última vez que o teste foi executado, F_T e P_1, P_2, P_3, P_4 os pesos desses parâmetros que podem ser modificados pelo usuário dentro do intervalo [0,5]. O valor V_T associado a esse teste é definido como:

$$V_T = (P_1 * E_T) * (P_2 * C_T) * (P_3 * L_T) * (P_4 * \frac{1}{F_T})$$

4.7 Tempo de Execução

Embora esse valor não seja usado para se determinar o peso do teste, a ferramenta armazena o tempo gasto para rodar o teste na última vez que ele foi executado. Esse valor é então usado para se garantir que o tempo de execução da suíte de testes não ultrapasse o tempo determinado pelo usuário.

4.8 Novos Testes

No decorrer da programação de um sistema, novos testes são criados para que novas partes do sistema sejam validadas. Embora a ferramenta detecte quando novos testes são criados, ela não terá nenhuma informação gravada referente a eles pelo fato de que as variáveis importantes para o cálculo do peso do teste e o tempo de execução só são salvas no final da execução ferramenta, mas a AdaptSuite nunca executou esses testes.

Para resolver esse problema, sempre que um ou mais testes forem adicionados ao sistema, a ferramenta atribuir-lhes-á peso 1 e tempo de execução 0 na próxima execução, fazendo com que eles sejam obrigatoriamente escolhidos para a suíte ótima, afinal, pensando no *Problema da Mochila*, escolher esses novos itens aumenta o valor da mochila sem aumentar peso atual da mesma, fazendo com que sempre valha a pena escolhê-los.

5 Armazenamento

Todas os valores das variáveis apresentadas na seção anterior são armazenados em um arquivo no formato *csv*, gerado com o auxílio da biblioteca *opencsv* [15]. O armazenamento desses valores em um arquivo é importante para que se possa utilizá-los nas execuções seguintes. Em especial, só é possível usar

os valores referentes à última execução do teste e a frequência com que o mesmo é selecionado pela ferramenta a partir desse arquivo.

Cada elemento do arquivo csv gerado é indexado pelo nome do teste, com cada coluna contendo o valor de uma variável. O arquivo propriamente dito é salvo no diretório em que se encontra o programa, com o nome "AdaptSuite.csv".

5.1 Usando o Eclemma

Como descrito previamente, os valores de cobertura são obtidos através de relatórios no formato "html", gerados pela ferramenta Eclemma. Esses arquivos devem ficar em um diretório de nome *Cov*, no mesmo nível da pasta *source* do projeto. Tais relatórios devem ser nomeados com o mesmo nome da classe de teste que os originou. Ou seja, o relatório de cobertura referente à classe $SampleTest$ deve receber o nome $SampleTest.html$.

Para se criar um relatório em formato html à partir dessa ferramenta, partindo do pressuposto de que a ferramenta já está instalada no Eclipse, basta selecionar o arquivo de teste desejado com o botão direito e escolher a opção "Coverage As". Com isso, a ferramenta irá executar o teste de cobertura do arquivo.

Após executado o teste de cobertura, o usuário deverá clicar na aba "Coverage"e, com o botão direito, selecionar a pasta que representa o nome do projeto e escolher a opção "Export Session"(Figura3).

Nessa nova janela que se abriu, o usuário deverá então escolher a opção "Coverage Report", dentro do menu "Java"e clicar "Next". Por fim, basta selecionar onde esse relatório será salvo e finalizar a operação.

No final desse processo, um arquivo de nome "index.html"será criado no diretório selecionado. O usuário deverá então renomeá-lo para o nome da classe de teste que se refere e salvar esse arquivo dentro do diretório "Cov".

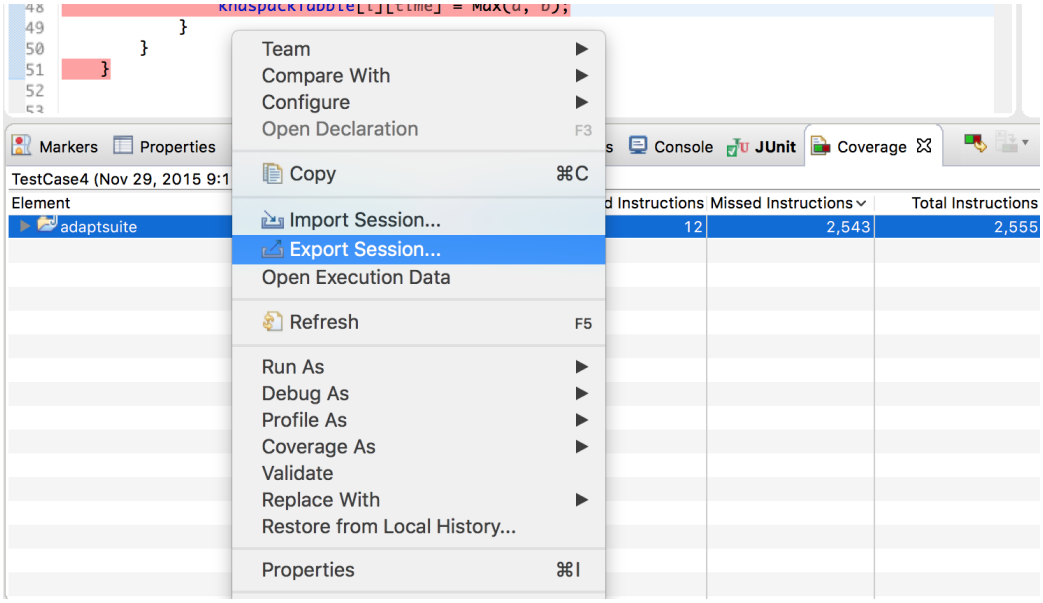

Figura 3: Exportando relatório de cobertura de um teste do projeto adaptsuite [14]

6 Uso da Ferramenta

Para que seja possível usar a ferramenta a partir do Eclipse, deve-se criar uma classe de teste JUnit e importar a ferramenta com o comando:

```
import main.java.org.AdaptSuite.suite.AdaptSuiteBuilder;
```

Após a importação da biblioteca, cria-se uma classe com método único seguindo o exemplo:

```
public class Seconds5suite {
   public static Testsuíte suite() {
      return new AdaptSuiteBuilder().sec(5).build();
   }
}
```

Em que:

- Testsuíte é o objeto JUnit responsável por agrupar e executar um conjunto de testes [8].

- AdaptSuiteBuilder é o objeto que instancia a ferramenta.
- sec(n) é o tempo máximo, em segundos, para se rodar os testes, ou seja, o tempo máximo da suíte ótima. No exemplo, a suíte terá no máximo 5 segundos de duração.

A ferramenta permite que seja definido o tempo máximo da suíte na ordem de milissegundos, segundos e minutos. A sintaxe para se definir a suíte com essas ordens de grandeza é dada por:

```
return new AdaptSuiteBuilder().mili(n).build();
return new AdaptSuiteBuilder().sec(n).build();
return new AdaptSuiteBuilder().min(n).build();
```

6.1 Definindo Prioridades

Por padrão, o sistema irá atribuir uma prioridade uniforme para todos os critérios, ou seja, o sistema assume que todos os parâmetros são igualmente importantes para a detecção de erros do sistema a ser testado.

Entretanto, para um sistema em particular, um ou mais parâmetros podem ser muito mais importantes do que os outros. Nesse caso o usuário pode atribuir valores de 0 a 5 para cada um dos critérios avaliados pela AdaptSuite no momento em que a ferramenta é chamada.

Para definir prioridades para os parâmetros, basta criar um objeto do tipo Map<String, Long> e passá-lo como argumento na chamada principal, conforme o exemplo:

```
public class Seconds5suite {
   public static Testsuíte suite() {
      Map <String, Long> relevance = new HashMap<String, Long>();

      relevance.put(AdaptSuiteBuilder.getErrorConstant(), 2L);
      relevance.put(AdaptSuiteBuilder.getFrequencyConstant(), 4L);
      relevance.put(AdaptSuiteBuilder.getLastExecutionConstant(),
          0L);

      return new AdaptSuiteBuilder().sec(5).build(relevance);
   }
}
```

No exemplo mostrado acima, o programa irá atribuir prioridade 2 para a quantidade de erros, prioridade 4 para a frequência com que o teste é escolhido pela ferramenta e excluirá a variável referente à última vez que o teste

foi executado do cálculo do peso do teste. As outras variáveis continuarão com prioridade 1.

O motivo para a prioridades serem do tipo *Long* é simplesmente porque as variáveis com valores inteiros, bem como o tempo de execução são armazenados como objetos do tipo *Long*.

Caso o programa receba alguma variável com prioridade menor que 0, será atribuída prioridade 0. Usando essa mesma lógica, caso o programa receba alguma variável com prioridade maior que 5, a mesma será considerada como prioridade 5.

As constantes disponíveis para serem usadas como parâmetros no objeto Map são:

```
AdaptSuiteBuilder.getErrorConstant()
AdaptSuiteBuilder.getCoverageConstant()
AdaptSuiteBuilder.getLastExecutionConstant()
AdaptSuiteBuilder.getFrequencyConstant()
```

7 Comparando a Eficiência da AdaptSutie

A eficiência da AdaptSuite foi testada nos seguintes cenários:

- Detecção de um teste que não apresentava erro até então, mas que irá falhar na próxima execução.
- Garantir que um teste defeituoso irá rodar sem apresentar erro após a devida correção no código.

Para isso, a mesma fora comparada com outras duas soluções: o algoritmo guloso que ordena os testes de acordo com seu peso e escolhe os testes de maior valor até a suíte ser preenchida, e o algoritmo aleatório, que tenta simular o comportamento humano que não usa nenhum parâmetro, apenas olhando para o tempo de execução do teste e verificando se o mesmo cabe na suíte de testes.

A escolha do algoritmo aleatório se deu por ser um dos mais usados para se resolver o problema de *TCP*. Embora ele não seja muito bom para se resolver um problema que envolva apenas o *TCP*, essa solução se mostra eficiente quando se tenta resolver o problema da *Fault Localization* [4] [10].

7.1 Preparação

Para todos os algoritmos, em ambos os casos descritos, foram criados 20 testes, sendo que o mais rápido deles demorava 250 milissegundos e cada teste

subsequente era 250 milissegundos mais demorado que o anterior, fazendo com que o 20º teste tivesse duração de 5 segundos.

Para se recriar o cenário em que a ferramenta já estava sendo utilizada a algum tempo, para cada teste de tamanho t, tanto a ferramenta como o algoritmo guloso foram submetidos a encontrar 10 vezes suítes de tamanho $2t$. Esse procedimento não foi realizado com a solução aleatória, pois a mesma não leva em consideração execuções passadas na sua execução atual.

7.2 Metodologia

Em ambos os casos comparativos, quando se tentava encontrar a suíte mínima que escolhesse o teste t de duração w_t, o primeiro tamanho de suíte a ser testado era justamente w_t. Caso o algoritmo não escolhesse tal teste, a próxima suíte a ser testada teria duração w_t+250 milissegundos. Tal processo era repetido até enfim encontrar uma suíte de tamanho C que finalmente executasse o teste t. Por fim, foi considerado que todos os testes possuíam a mesma taxa de cobertura.

Essa metodologia foi usada nas execuções da AdaptSuite, do algoritmo guloso e na solução aleatória. No caso da solução aleatória, a suíte mínima foi encontrada 5 vezes para cada teste t de duração Δ_t e uma média de tempo foi tirada dessas 5 suítes.

7.3 Detectando um Erro

Nesse caso, foi considerado um cenário em que todos os testes rodavam sem apresentar erro. Porém, a próxima execução de um determinado teste irá apresentar um erro e o programador não está ciente disso. Deseja-se saber qual o tamanho mínimo da suíte para que tal teste defeituoso seja executado.

Os resultados desse caso são apresentados abaixo:

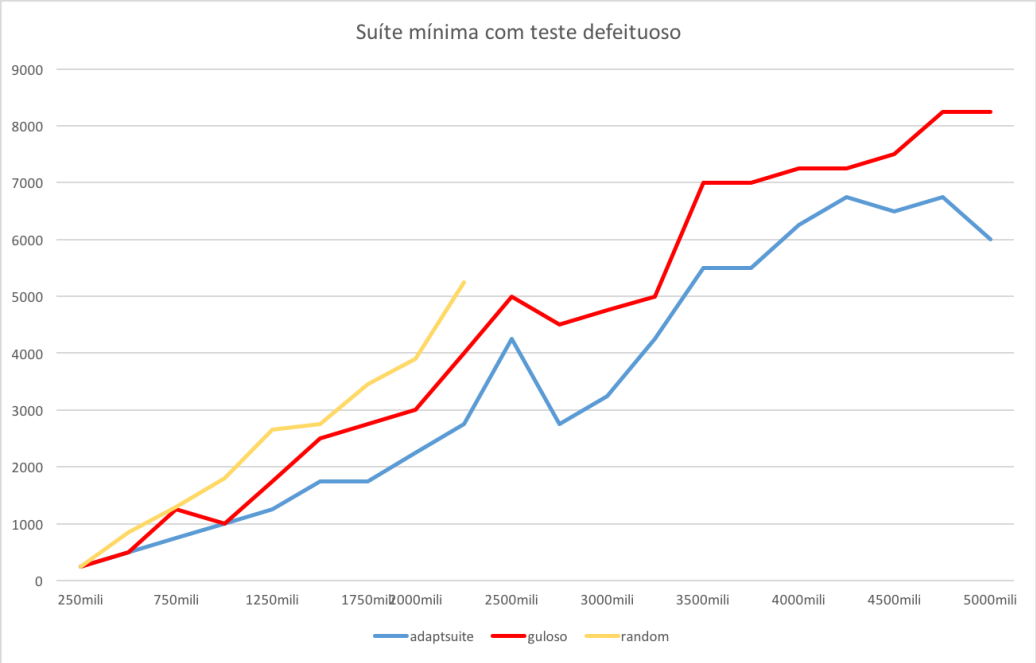

Figura 4: Suíte mínima que executa o teste defeituoso

Pelo gráfico, pode-se observar que de fato a AdaptSuite consegue encontrar o teste defeituoso mais rapidamente do que o algoritmo guloso. Uma propriedade interessante é que a AdaptSuite sempre encontra o teste defeituoso de duração Δ_t em uma suíte não maior que $2\Delta_t$.

Outro ponto interessante é que tanto a AdaptSuite quanto o solução gulosa foram muito melhores que a solução aleatória. De fato, a mesma teve de ser considerada apenas até metade dos casos de testes pois, à partir desse ponto, sua suíte mínima ficava extremamente mais demorada, fazendo com que no final suas suítes mínimas estivessem em uma escala totalmente diferente das outras duas soluções. De fato, as soluções da AdaptSuite e guloso apresentaram um comportamento próximo ao linear, enquanto o aleatório apresentou um comportamento próximo ao exponencial.

7.4 Verificando a Correção

Aqui, foi considerado um cenário em que um teste apresentou um defeito, a correção foi feita, e agora deseja-se verificar que a correção de fato faz com que o teste seja executado sem apresentar um erro. O objetivo, novamente, é encontrar o tamanho mínimo da suíte que irá executar determinado teste.

Os resultados para esse segundo caso são apresentados logo abaixo:

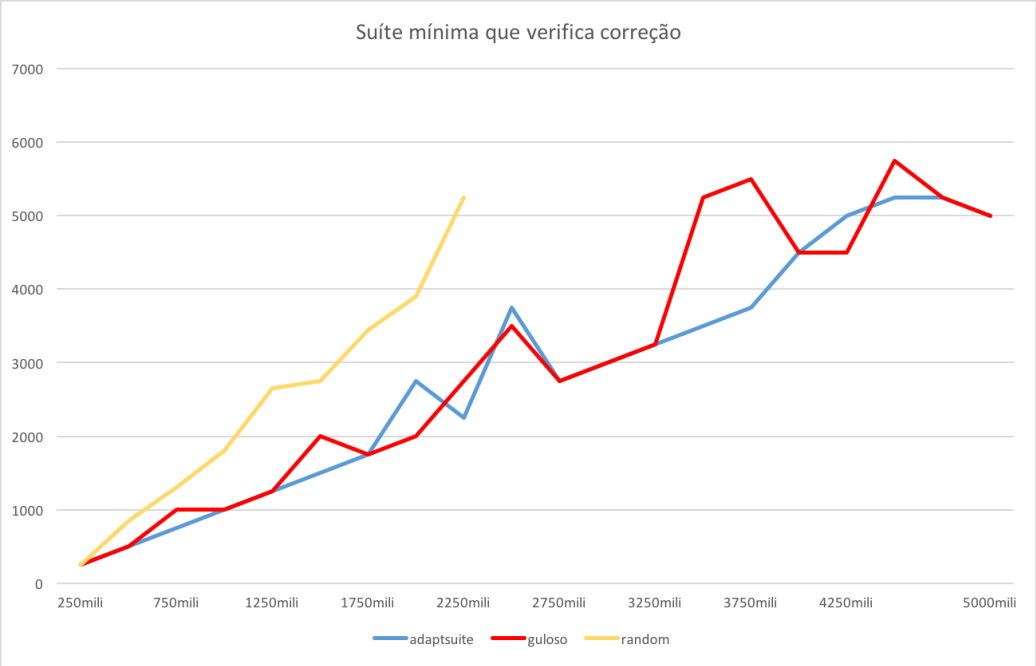

Figura 5: Suíte mínima que verifica a correção do teste defeituoso

Pelos resultados, tanto a AdaptSuite quanto o algoritmo guloso possuem perfomances semelhantes, entretanto, ambos se comportam de maneira mais eficiente nesse caso do que na situação em que desejasse encontrar o teste defeituoso pela primeira vez, mantendo, portanto, a propriedade de encontrar um teste de duração Δ_t em uma suíte de no máximo $2\Delta_t$.

Novamente, a solução aleatória se mostrou muito ineficiente, pois a mesma não leva em conta nenhum fator a não ser a duração do teste. Por esse motivo, os resultados apresentados em ambos as situações foram iguais.

8 Conclusão e Próximos passos

Pelos resultados apresentados, a ferramenta AdaptSuite consegue detectar erros e verificar que os mesmos foram corrigidos de maneira eficiente. De fato, a mesma apresentou uma propriedade interessante de executar testes de duração Δ_t em suítes não maiores do que $2\Delta_t$, o que vai ao encontro com a ideia apresentada no começo de que testes são demorados e de que a

AdaptSuite oferecia uma solução que garantisse a qualidade de um software e economizasse o tempo necessário para tal.

Outro ponto importante apresentado pela ferramenta é que ela se comporta de maneira mais eficiente do que o algoritmo guloso, que sempre escolhe os testes mais valiosos, para se detectar um erro. Com esse resultado, fica evidente que o melhor conjunto de testes não é aquele que contém os melhores testes, como poderia se imaginar inicialmente, ao ser apresentado ao problema pela primeira vez.

Finalmente, uma última conclusão é que, independente da abordagem usada ser a solução ótima do Problema da Mochila ou o algoritmo guloso, eles sempre serão muito melhores do que executar os testes sem parâmetro algum. Talvez esse seja o motivo que faz com que validar um sistema seja tão caro e demorado: Sem uma métrica clara que define o quão bom é um teste, a equipe de teste tem de encontrar um erro quase que às cegas. Nesse caso, apenas executando todos os testes irá garantir a qualidade do sistema.

No ponto em que o projeto se encontra, seria interessante criar um sistema que usasse os conceitos do AdaptSuite e relatar como ela melhorou ou piorou o desenvolvimento em relação à um processo mais tradicional.

Outro próximo passo interessante seria resolver esse mesmo problema, mas considerando cada variável como um elemento de um plano distinto, ou seja, considerar o peso de um teste como um ponto em $\mathbb{R}^4$ ao invés de um ponto em $\mathbb{R}$, como adotado nesse trabalho. Nesse caso, talvez seja mais interessante usar um algoritmo de aproximação para o *Problema da Mochila Multidimensional*

Por fim, um importante futuro trabalho usando a AdaptSuite seria a análise dos efeitos das suítes escolhidas nas técnicas de *Fault Localization*. Tal trabalho deveria responder a seguinte pergunta: "Uma suíte ótima escolhida pela AdaptSuite que detecta um ou mais erros consegue encontrar a exata localização desses erros usando uma técnica de *Fault Localization* eficientemente?".

9 Áreas Correlacionadas

A área de maior influência do AdaptSuite é a de *Test Case Priorization (TCP)*, pois a ferramenta propõe uma solução para o problema fundamental dessa área que agrega os parâmetros mais comumente usados nessa área, sendo eles, o histórico de falhas de um teste e a cobertura do mesmo, sendo umas das poucas técnicas que o fazem levando em consideração um tempo limite máximo para a execução da suíte ótima. A ferramenta ainda usa alguns conceitos básicos de Machine Learning, pois a mesma usa a informação de

cada execução para atualizar sua base de dados e se adaptar para a próxima execução.

Duas outras áreas relacionadas são as de Teste de Software e de Qualidade de Software, pois a AdaptSuite consegue detectar erros e verificar que os mesmos foram corrigidos em um teste de duração t com uma suíte de tamanho no máximo $2t$.

Por fim, a terceira área de influência é a de Desenvolvimento Ágil, tendo em vista que o AdaptSuite permite uma resposta rápida para quando uma modificação no código é realizada.

10 Parte Subjetiva

10.1 Concepção da Ideia

A inspiração para esse projeto veio do aluno de doutorando do professor Alfredo Goldman, Renan de Oliveira, que desejava construir uma ferramenta em Java capaz de otimizar a escolha de testes, fazendo com que os testes mais importantes fossem executados primeiramente.

Inicialmente, eu aceitei esse desafio pois gostava da área de testes de software, mas conforme eu ia estudando sobre o assunto, descobrimos juntos que essa era uma área muito nova e de pouca pesquisa, portanto, não havia muitas obras na literatura que propusesse uma solução definitiva sobre esse problema. Muitos deles apenas ressaltavam a importância de se estudar uma solução e como a etapa de testes costuma ser custosa dentro processo de desenvolvimento de software.

Com essa base de estudos, decidi juntar as métricas mais importantes para um teste, sua capacidade de detectar falhas e sua cobertura, e desenvolvi uma solução que abrangesse esses dois conceitos.

Foi nesse ponto que meu interesse pelo projeto mudou, pois eu percebi que esse projeto seria capaz de juntar um problema teórico, que é o Problema da Mochila, com um problema prático, que é o custo e o tempo necessário para se garantir a qualidade de um sistema. Essa ligação entre o mundo teórico e o mundo prático era justamente o que mais me atraía na graduação e desenvolver algo em uma área pouco explorada usando essa abordagem era o que motivava.

10.2 Dificuldades

Durante esse ano, com toda certeza meu maior desafio foi conseguir conciliar meu tempo entre o TCC e meu estágio no Facebook. Ambas as organizações,

USP e Facebook, exigem um alto grau de qualidade e dedicação. No fim, acabou sendo uma experiência muito positiva pois eu acredito que consegui cumprir meus objetivos tanto na universidade como na empresa.

Embora, como mencionado, o estágio tenha consumido muito do meu tempo hábil, ele também contribui para que eu evoluísse tecnicamente em termos de qualidade de programação. Quando eu trabalhava no TCC, eu sentia que eu conseguia produzir um código de muito mais qualidade e em um tempo muito mais curto do que no ano passado, por exemplo.

10.3 Matérias Importantes

- **MAC0342 - Laboratório de Programação Extrema:** Importante por apresentar o conceito de teste automatizado e como testes são importantes para garantir a qualidade de um projeto desenvolvido a partir de uma metodologia ágil.
- **MAC0338 - Análise de Algoritmos:** Matéria que me apresentou o problema da mochila, usado para resolver o problema proposto nesse trabalho.
- **MAC0332 - Engenharia de Software:** Mostrou como os testes são utilizados em métodos de desenvolvimento mais tradicionais e como muitos projetos de software em geral não os utilizam.
- **MAC0444 - Sistemas Baseados em Conhecimento:** Importante por explicar como sistemas podem usar execuções passadas para melhorar a eficiência das execuções futuras.
- **MAC0340 - Laboratório de Engenharia de Software:** Apresentou uma novo metodologia para a criação de software, baseada em especificações formais e como os testes podem ser criados automaticamente a partir dessa especificação.

10.4 GitHub

O código da ferramenta pode ser encontrado no seguinte repositório dentro do GitHub: `https://github.com/adaptsuite/adaptsuite`

Referências

[1] Pressman, R. S. *Engenharia de Software - Uma Abordagem Profissional.* AMGH Editora, 7 edição, 2011.

[2] Wu, K., Fang, C., Chen, Z., Zhao, Z. *Understanding the effects of changes on the cost-effectiveness of regression testing techniques.* Test Case Prioritization Incorporating Ordered Sequence of Program Elements, 2012.

[3] Bernardo P. C., Kon F. *A Importância dos Testes Automatizados - Controle ágil, rápido e confiável de qualidade.* Engenharia de Software Magazine, 2008

[4] Walcott, K. R., Sofia, M. L., Kapfhammer, G. M. *Time-Aware Test Suite Priorization* Proceedings of the 2006 ACM SIGSOFT International Symposium of Software Testing and Analysis. ACM Press, Nova York, 2006.

[5] Yu, Y. T., Lau, M. F. *Fault-based test suite prioritization for specification-based testing*, volume 54. Information and Software Technology, 2012.

[6] Mens, T., Demeyer S. *Software Evolution.* Editora Springer, 1 edição, 2008.

[7] Rogers, E. M. *Diffusion of innovations.* New York Free Press 5.ed., 2003

[8] Massol, V., Husted T. *JUnit em Ação.* Editora Ciência Moderna, 1 edição, 2005.

[9] Catal, C., Mishra, D. *Test case prioritization: a systematic mapping study.* Springer Science+Business Media, 2012.

[10] Jiang, B., Zhang, Z., Tse, T. H., Chen, T. Y. 33rd Anual IEEE International Computer Software and Applications Conference, 2009.

[11] Rothermel, G., Untch, R. H., Chu, C., Harrold, M.J. *Prioritizing test cases for regression testing*, volume 27. IEEE Transactions on Software Engineering, 2001.

[12] Kellerer, H., Pferschy, U., Pisinger, D. *Knapsack Problems* Editora Springer, 1 edição, 2004

[13] Fernandes, C. G., Feofiloff P., Pina, J. C. *MAC0338 - 1 semestre de 2013* `http://www.ime.usp.br/~cris/mac338/slides/aula16.pdf` Acessado em 10 de fevereiro de 2015.

[14] Eclemma official site http://eclemma.org/ Acessado em 20 de abril de 2014.

[15] Smith, G OpenCSV Main Page. http://opencsv.sourceforge.net/. Acessado em 22 de setembro de 2015.