

# Otimização Combinatória em testes

Rafael Grisotto e Souza - RA 192765

Vinicius

Paulo Henrique Carvalho de Moraes - RA 192877

11 de maio de 2018

## **Questão 1.**

Resumo de Search-Based Software Testing. <https://ieeexplore.ieee.org/document/5954405/>

Descrever EvoSuite (meta heurística) <http://www.evosuite.org/publications/>

Alternativas para melhorar os resultados: meta heurística a, b, c, ...

Trabalho 2 começa aqui: [1] Projetar e rodar experimento com todas ou um subconjunto e comparar.

## Notes

■ citar!! . . . . .	3
■ Expandir para como ele gera os testes . . . . .	3
■ explicar melhor . . . . .	3
■ Precisa falar das metaheurísticas, quais iremos usar, algum por que de cada uma e depois dizer como elas estão organizadas na seção . . . . .	4
■ Esta quase completa, para mim falta apenas colocar mais variações . . . . .	4
■ Deixar mais fluido o texto aqui . . . . .	4
■ Falta bastante coisa do graps, principalmente sobre suas variações . . . . .	5
■ A descrição do BRKGA esta a mais incompleta. Precisa adicionar bastante coisa e descrever melhor . . . . .	6
■ Aqui precisamos apenas deixar fluido. acho que não precisa de mais conteúdo . . . . .	6

# 1 Referencial EvoSuite

EvoSuite é uma ferramenta de geração de suítes de testes com alta cobertura e produz asserções. O EvoSuite usa uma abordagem baseada em pesquisa, integrando técnicas atuais tais como hybrid search, dynamic symbolic execution e testability transformation. [citar!!](#)

**Geração de suítes de testes:** EvoSuite usa abordagem de busca evolucionária que evolui os conjuntos de casos de testes de acordo com a um critério de cobertura. A otimização prioriza o critério de cobertura ao invés objetivos locais que não resultam em diversidade influenciada pela ordem, dificuldade ou inviabilidade de objetivos individuais.

Otimizando com critério de convergência do que por convergência individual consegue resultados que não são influenciados pela ordem ou pela dificuldade ou inflexibilidade de convergência individuais. Estes são alguns dos problemas, pois alguns objetivos são mais difíceis que outros e existe o recurso limitado de tempo para testes. [Expandir para como ele gera os testes](#)

**Mutações baseadas em gerações de acerssões:** EvoSuite utiliza teste de mutação para produzir um conjunto reduzido de que maximiza a inclusão de defeitos em uma classe que são revelados pelos casos de testes. Isso ajuda dando feedback aos desenvolvedores de quais defeitos devem ser tratados

No *EvoSuite* uma solução candidata consistem de um número variável de casos de teste individuais. Onde, cada um destes casos de teste é uma sequencia de chamadas do método repetidas com a variação do comprimento e aperfeiçoando a unidade em teste e definido objetos complexos para fazer isso.

O *fitness* é calculado por um critério de convergência por completo. [explicar melhor](#)

## 2 Meta-Heurísticas

Meta-heurística é um subcampo primário da otimização estocástica, a qual consiste de algoritmos e técnicas que empregam algum grau de aleatoriedade para encontrar a solução ótima (ou a melhor possível) para problemas reconhecidamente difíceis. Luke [7] ressalva que meta-heurística é um termo que pode gerar desentendimentos uma vez que não se trata de heurística sobre (ou para) heurísticas, o que não é necessariamente verdade para todos os algoritmos.

### 2.1 GA

Um algoritmo genético simula a seleção natural, onde cada indivíduo compete com os outros para sobreviver com base em sua aptidão (*fitness*). Os indivíduos que sobrevivem ao passo de seleção passam por operações genéticas (cruzamentos e mutações), simulando o que ocorre na biologia, para assim criar uma nova população.

### 2.2 Método de seleção de indivíduos

Torneio funciona que até que se tenha cromossomos suficientes para fazer a recombinação, nesse caso 4, dois cromossomos são escolhidos aleatoriamente da população, usando uma distribuição de probabilidade uniforme, e o de melhor *fitness* é selecionado para o processo de recombinação.

### 2.3 Método de recombinação

Método de recombinação o operador *C1* são dados dois pais,  $p_1$  e  $p_2$ , é escolhido um ponto de corte até o qual o filho será igual ao  $p_1$ . Então, os elementos que faltarem no cromossomo filho (todos os clientes faltantes até o ponto de corte) são inseridos no filho seguindo a ordem em que aparecem em  $p_2$ .

Escolhemos o operador de recombinação do tipo *crossover OX*, por se tratar de um cruzamento para codificação de permutação que, além de ser simples, exige um baixo custo computacional quando comparado com os outros esquemas de recombinação para a representação de permutação, desta maneira um maior número de recombinações são possíveis para um tempo fixo de processamento da simulação. Por fim, foram utilizada a seleção aleatória de dois indivíduos da população para a utilização no processo de recombinação.

A recombinação de dois indivíduos  $A$  e  $B$  é feita usando crossover de 2 pontos, selecionando-se aleatoriamente 2 locus nas posições  $L$  e  $R$ , com  $L < R$ , para serem os pontos da troca. O primeiro indivíduo é gerado a partir da fusão entre as posições  $[0, L]$  e  $]R, size(A) - 1]$  de  $A$  e as posições  $]L, R]$  de  $B$ , já o segundo indivíduo é gerado a partir das posições  $[0, L]$  e  $]R, size(B) - 1]$  de  $B$  e as posições  $]L, R]$  de  $A$ .

Após o crossover, é necessário fazer a correção de possíveis indivíduos inválidos segundo as restrições do problema.

#### 2.3.1 Método de mutação

Para uma população, são visitados todos os cromossomos e cada um será mutado se for escolhido um número aleatório maior que a taxa de mutação utilizada. Testamos como taxa de mutação os valores  $1/m$  (constava no artigo como o valor ideal para mutações) e  $2/m$ , sendo  $m$  o tamanho da instância de entrada.

Após a mutação, também é necessário fazer correções, pois pode haver indivíduos inválidos devido às restrições do problema.

Precisa falar das metaheurísticas, quais iremos usar, algum por que de cada uma e depois dizer como elas estão organizadas na seção

Esta quase completa, para mim falta apenas colocar mais variações

Deixar mais fluido o texto aqui

## 2.4 Uniform Crossover

Método de recombinação. O novo método de combinação utilizado foi a recombinação uniforme, ou *Uniform Crossover*. Nessa recombinação, em vez de escolher aleatoriamente dois pontos de cruzamento, a cada locus, um pai é escolhido aleatoriamente para ter seu alelo copiado; o alelo deste pai é copiado para o primeiro descendente enquanto o alelo do outro pai do mesmo locus é copiado para o segundo descendente.

Da mesma maneira que no método de recombinação de dois pontos, após a criação da nova geração através desta recombinação, todos os descendentes são verificados e, para todo par de locus consecutivos com alelos iguais a 1, um destes locus é escolhido aleatoriamente e seu alelo alterado para 0.

## 2.5 Steady-state

Tipicamente, a execução de um algoritmo genético é dividido em gerações que são substituídas (quase que por completo) a cada iteração do algoritmo. No caso do algoritmo genético *steady state* apenas alguns indivíduos são substituídos ao final de cada iteração.

Neste processo, dois pais devem ser selecionados da população atual e um filho deve ser gerado pelo processo de *crossover*. Em seguida, o pior indivíduo dentre ambos os pais e filho é removido e os outros dois são realocados (se necessário) na população. Com isso, o conceito de gerações passa a não fazer sentido pois, neste caso, tem-se apenas um filho gerado enquanto o resto da população permanece constante.

### 2.5.1 Manutenção de diversidade

Para manter os indivíduos diversos e evitar a convergência muito cedo, foi usada uma função que diversifica a população após a seleção de pais, criação e mutação da geração atual.

Neste caso, realizam-se testes 2 a 2 em cada indivíduo e se os indivíduos possuem um número maior que o valor pré-definido (0.5%, por exemplo) de alelos iguais, o primeiro dos dois sofre uma mutação aleatória em um locus.

## 2.6 Cruzamento

Dois indivíduos,  $I_1$  e  $I_2$ , são escolhidos aleatoriamente, usando uma distribuição de probabilidade uniforme, do conjunto de cromossomos previamente selecionados para o cruzamento. Seja  $N$  o número de loci e  $0$  a  $N - 1$  os índices desses loci. Dois pontos de cruzamento  $0 \leq p_1 < p_2 \leq N - 1$  são escolhidos usando uma distribuição de probabilidade uniforme para cada recombinação. São criados dois novos indivíduos, o primeiro recebe os alelos de  $p_1$  a  $p_2$  do indivíduo  $I_1$ . Se o filho não tiver  $p$  alelos 1, alelos 1 são escolhidos aleatoriamente, usando uma distribuição de probabilidade uniforme, dos intervalos  $[0, p_1)$  e  $(p_2, N]$  do indivíduo  $I_2$ , até que o filho seja viável. Se após isso o filho ainda não for viável, são escolhidos alelos 1 do intervalo  $[p_1, p_2]$  do indivíduo  $I_2$  até que o filho seja viável. O mesmo ocorre para o segundo filho invertendo a ordem dos pais. Note que não há cromossomos infactíveis gerados na recombinação.

## 2.7 GRASP

O GRASP (Greedy Randomized Adaptive Search Procedure) para problemas de otimização combinatória foi introduzida por Feo e Resende [3]. O GRASP tem sua iteração primeiramente a construção de uma solução inicial a partir de uma heurística construtiva gulosa e aleatória. Depois, é realizado uma busca local com intenção de explorar a vizinhança da solução inicial até atingir um

Falta bastante coisa do graps, principalmente sobre suas variações

mínimo local do espaço de soluções. Após um critério de parada como quantidade de iterações ou tempo de processamento o GRASP devolve a melhor solução encontrada e termina sua execução.

### 2.7.1 Busca Local

Método de Busca Local: regula o tipo de busca realizada na fase de busca local. Aceita valores *'best improving'* ou *'first improving'*. O primeiro busca por toda a vizinhança a procura da melhor melhoria local do valor objetivo possível, enquanto o segundo procura apenas pela primeira melhoria no valor objetivo;

### 2.7.2 Heurística Construtiva

É da natureza das heurísticas de busca locais a dependência da solução inicial e da dimensão da vizinhança. Assim, o uso de uma heurística construtiva, a fim de gerar uma solução inicial de qualidade, facilita o desempenho da busca local. Como a Busca Tabu tem como essência a busca local foi utilizada uma heurística construtiva.

A heurística construtiva utilizada foi onde os elementos são selecionados a partir de uma lista restrita de candidatos (LRC). A LRC é constituída por todos os elementos que geram melhora a função objetivo quando adicionados a solução. Ao termino de cada iteração da heurística construtiva a LRC é atualizada uma vez que o vetor-solução também foi alterado.

A construção do vetor-solução pode ter mecanismos que garantem a factibilidade durante a busca ou ainda aplicar mecanismos de reparo, para garantir a factibilidade. Como as técnicas de reparo perdem parte da qualidade da solução gerada, a abordagem desse trabalho utiliza somente opções de construção factíveis. A heurística construtiva é executada enquanto existem elementos na LRC.

## 2.8 BRKGA

O BRKGA (biased random-key genetic algorithm)[6] é uma metaheurística para encontrar uma solução ótima ou próxima da ótima. É uma variação do RKGA (random-key genetic algorithms) [2] e se obtém uma solução viável para o problema através da decodificação de uma solução codificada. A solução codificada são vetores de chaves aleatórias em um intervalo de números reais contínuo  $(0, 1]$  e a decodificação é uma etapa que mapeia um vetor de chaves aleatórias numa solução do problema de otimização e calcula o seu custo. Note que o BRKGA tem como seu algoritmo sendo independente do problema.

A descrição do BRKGA esta a mais incompleta. Precisa adicionar bastante coisa e descrever melhor

De forma resumida, a definição de algumas características do BRKGA:

- Codificação de uma solução com chaves aleatórias em intervalo contínuo  $[0, 1]$
- Geração de população inicial com  $p$  vetores de  $n$  chaves aleatórias
- Método de seleção de indivíduos é ordenado pelo *fitness* e gerado dois grupos, elite e não-elite. Após, é escolhido um pai do conjunto elite e o outro é escolhido do conjunto não-elite.

## 2.9 Busca Tabu

A Busca Tabu é uma busca local que permite movimentos que podem não melhorar a solução atual. Para evitar ciclos, existe um mecanismo chamado lista tabu onde os últimos movimentos realizados ficam armazenados por um dado número de iterações, chamado *tenure*. Os movimentos que estão na lista de tabus ficam proibidos, a menos que um dado critério de aspiração seja satisfeito. Caso um critério de aspiração seja satisfeito o movimento é permitido. A heurística foi originalmente

Aqui precisamos apenas deixar fluido. acho que não precisa de mais conteúdo

proposta por Glover [5] e nos últimos 20 anos centenas de artigos utilizam Busca Tabu em diversos problemas de combinatória e mostrando ser bastante eficaz com resultados bem próximos do ótimo ou conseguindo o ótimo [4].

Na Busca Tabu precisamos definir muito bem o espaço de busca e a estrutura da vizinhança. Definir os critérios da restrição de movimentos. Pode ser complicado deixar a metaheurística suficientemente genética para ser usada no framework EvoSuite, porém podemos pensar em casos não muito gerais e aplicar mecanismos de busca por intensificação e por diversidade.

### 2.9.1 Busca *Tenure*

Determina a quantidade de iterações que um movimento permanece tabu, sabendo que em cada iteração um número constante de movimentos são inseridos e retirados da lista Tabu. É dado em relação ao tamanho do problema, ou seja, recebe valores no intervalo  $[0, 1]$ , e cada movimento permanece no tabu por  $tenure * n$  iterações;

### 2.9.2 Intensificação

Implementamos apenas uma intensificação por vizinhança, no qual são explorados todos os movimentos possíveis (inserção, remoção e troca) com quaisquer três elementos. Como este é um método alternativo, e não temos certeza de sua eficácia, habilitamos a possibilidade de desativar esta intensificação como parâmetro. Aceita 'sim' ou 'não';

### 2.9.3 Método de Busca Local

Regula o tipo de busca realizada na fase de busca local. Aceita valores '*best improving*' ou '*first improving*'. O primeiro busca por toda a vizinhança a procura da melhor melhoria local do valor objetivo possível, enquanto o segundo procura apenas pela primeira melhoria no valor objetivo. Caso tal melhora não exista, o movimento que menos piora o valor objetivo é aplicado em ambos os casos;

- *Best Improving*: onde toda a exploração da vizinhança de uma solução é dada até a exaustão, ou seja, todos os elementos são analisados até que o mínimo local seja encontrado. Como a busca está restrita a vizinhança da solução o mínimo encontrado é local, e não se tem garantida quanto a otimalidade global. No mesmo intuito de buscar por soluções sempre factíveis, a fim de não aplicar operadores de reparo nas soluções, a técnica de *best improving* só analisa a vizinhança factível. Três tipos de análise foram utilizadas para a busca: inserção, remoção e troca de entradas do vetor-solução, todas feitas respeitando os critérios de factibilidade.
- *First Improving*: a busca na vizinhança é feita usando as técnicas de inserção, remoção e troca. Todas as opções factíveis para as três técnicas são colocadas em uma lista e um dos elementos dessa lista é selecionado de forma aleatória, assim que a primeira melhora é encontrada o processo de busca nessa vizinhança para e a lista é atualizada para contemplar as novas opções factíveis.

### 2.9.4 Surrogate Objective

Ao resolver um problema por meta-heurísticas, ou ainda, heurísticas, é imprescindível a escolha de uma boa função de avaliação da solução. Por vezes, a melhor escolha para a função de avaliação não é a função objetivo, seja porque o cálculo da função objetivo tem alto custo computacional ou ainda porque ela não trás informações precisas sobre a solução que está sendo tratada. Nesses

casos é interessante o uso de uma função de avaliação diferente, conhecida também como *surrogate objective*.

Para esse trabalho foi utilizada uma função de avaliação alternativa, onde ao invés de calcular  $x'Ax$ , usa-se o valor do benefício possível de cada entrada para calcular o *surrogate objective*. Ou seja, para o elemento  $x_i$ , candidato a participar da solução, sua contribuição é calculada como  $\sum_{j=1}^n A_{i,j} + A_{j,i}$ , onde  $n$  é a dimensão da matriz  $A$ .

### 2.9.5 Critério de Aspiração

Determinar o tamanho da lista tabu e quais níveis de restrição utilizar não é uma tarefa simples. Assim, é possível que existam ações proibidas pela lista tabu que ajudariam a melhorar a solução corrente. Portanto, criar mecanismos que permitam violar a lista tabu são importantes, esses mecanismos são denominados critérios de aspiração.

O critério de aspiração desenvolvido nesse trabalho é baseado na análise do benefício que os elementos da lista tabu poderiam gerar para a solução corrente. Ou seja, caso haja algum elemento na lista tabu que apresente a melhor contribuição, entre os elementos dentro e fora da lista, ele será utilizado para compor a solução.

### 2.9.6 Diversificação por Reinício

Envolve forçar alguns componentes que são raramente utilizados na solução e reiniciar o processo de busca a partir deste ponto. O critério para reiniciar a busca foi definido com  $r$  iterações sem melhoria da solução corrente. Esse parâmetro foi deixado para ser ajustado pelo *irace*. Seja  $t$  o *tenure*, quando a busca é reiniciada são colocados os  $t$  movimentos mais frequentes no tabu, colocando primeiro o menos frequente e por o último o mais frequente. A frequência é calculada somando 1 cada vez que o movimento é realizado pelo operador de busca local. A frequência não é reiniciada quando a busca recomeça. Quando a busca reinicia uma nova solução inicial é construída, a heurística construtiva é escolhida aleatoriamente sendo que as duas tem a mesma probabilidade de serem escolhidas.

### 2.9.7 Diversificação Continua

Integra diversificação considerando diretamente no processo normal de busca. Conseguindo isso a partir do calculo da influencia dos possíveis movimentos adicionando ao objetivo um termo pequeno relacionado a frequência dos componentes.

### 2.9.8 Descrição da lista Tabu

Tabu é uma memória de curto prazo usado pelo operador de busca local para evitar de repetir movimentos ou desfazer movimentos que foram feitos a no máximo um dado número de iterações. Neste trabalho, a lista tabu foi implementada como um fila duplamente terminada. Um número fixo de elementos é permitido estar na fila a cada instante, sendo assim, cada elemento pode ficar por um número determinado de iterações da busca local, pois a cada iteração dois novos elementos entram na lista. Toda vez que um elemento é adicionado ou removido da solução corrente, esse elemento é adicionado na lista tabu. A cada iteração, se apenas uma das operações é a realizada, inserção ou remoção, um elemento que não faz parte do problema é adicionado na lista tabu para assegurar que os dois elementos mais antigos sejam removidos. O número de iterações que o elemento fica na lista tabu é chamado de *tenure*. Para assegurar que a lista sempre tem um número fixo de elementos, a fila foi inicializada com elementos que não fazem parte do problema. Nós percebemos que o tamanho da *tenure* influencia muito na qualidade da solução encontrada pela



busca tabu. Por meio de tentativas, nós chegamos a dois valores que se saíram melhor para a maioria dos casos, 20% e 30% do número de variáveis de decisão definem o tamanho da lista de tabus. Como a cada iteração dois elementos entram e os dois mais antigos saem da lista tabu, o *tenure* dessa lista é a metade de seu tamanho.