Teste de Software Baseado em Meta-heurísticas

Paulo Henrique Carvalho de Morais
Instituto de Computação
Universidade Estadual de Campinas
Campinas, Brasil
ra192877@unicamp.br

Rafael Grisotto e Souza
Instituto de Computação
Universidade Estadual de Campinas
Campinas, Brasil
rafaelgrisotto@students.ic.unicamp.br

Vinícius Loti de Lima
Instituto de Computação
Universidade Estadual de Campinas
Campinas, Brasil
ra209829@ic.unicamp.br

***Abstract*—Nesse artigo são apresentados alguns conceitos de teste de software baseados busca, principalmente com o uso de meta-heurísticas. Mais ainda, é dada uma breve introdução ao *EvoSuite*, uma ferramenta para geração automatizada de casos de teste, assim como uma introdução á algumas meta-heurísticas que podem ser implementadas dentro do *EvoSuite* para realizar as buscas dos casos de teste.**

Index Terms*—software, teste, busca, meta-heurística, *EvoSuite

I. Introdução

Em desenvolvimento de softwares, etapas de teste do software são essenciais para garantir um produto final de qualidade. A realização de testes busca minimizar os erros no software. No entanto, nem sempre é possível eliminar todos os erros de um software por meio de testes. A minimização dos erros é um fator dependente dos casos de teste gerados. Publicações em "Teste de Software Baseado em Busca" tiveram início em 1976 com [18]. Foi depois de vários anos, em 1990, que Korel [2], [3] deu continuidade a essa linha de pesquisa. Em testes de software baseados em busca, a geração de casos de testes por meio de algoritmos de busca, tal que as entradas são geradas aleatoriamente até que os objetivos do teste sejam satisfeito. Uma busca aleatória pode não ser muito eficiente para geração de casos de teste, quando o espaço de busca é muito maior do que o espaço de soluções viáveis [17]. O ideal é utilizar algoritmos mais guiados, assim como meta-heurísticas, onde existe uma função chamada de função de fitness (*fitness function*), que é dependente do problema sendo resolvido, e que serve para guiar a busca por soluções, retornando valores que indicam a qualidade de cada solução encontrada.

Alguns exemplos de aplicações de funções de fitness, presentes em [17]:

- Testes temporais: busca aproximar os tempos de execução para o melhor e pior caso para componentes de um sistema. Uma forma de modelar esse problema como teste baseado em buscas, é definindo a função de fitness como sendo o tempo de execução do componente, para uma dada entrada. Para melhorar a qualidade dos atributos encontrados pela busca, pode ser utilizada análise estática.
- Testes funcionais: busca testar a funcionalidade de componentes de um sistema. Um exemplo de teste funcional baseado em busca é dado em [4], [5], onde foi testado um sistema de estacionamento que identifica uma vaga de estacionamento e automaticamente manobra o carro para estacioná-lo na vaga sem causar colisões. A função de fitness usada foi então a distância mínima até um ponto de colisão, onde o objetivo era então minimizar essa distância, para detectar possíveis falhas no sistema.
- Testes estruturais: foi a origem de testes baseados em busca, apresentada em [18], e é a área de aplicação que mais vem atraindo atenção em testes baseados em busca. Alguns dos testes estruturais aos quais já foram desenvolvidas funções de fitness na literatura, é cobertura de caminhos (*path coverage*, cobertura de ramos (*branch coverage*) e cobertura de fluxo de dados (*data flow coverage*). Para testes de cobertura de ramos, por exemplo, pode ser utilizada uma função de fitness dada pela normalização da distância de um dado nó da árvore do programa até o primeiro nó onde ocorre a divergência do possível caminho até o nó dado.

Em 2011, [9] propuseram a ferramenta *EvoSuite*, uma ferramenta para geração automatizada de casos de teste que utiliza algoritmos de busca. A meta-heurística utilizada pelos autores para os experimentos foi um algoritmo genético. O objetivo do artigo presente é apresentar algumas informações relevantes da ferramenta *EvoSuite* e de meta-heurísticas que podem ser utilizadas juntamente com o *EvoSuite* para geração de casos de teste, para em uma pesquisa futura, realizar testes de qual é o comportamento do *EvoSuite* ao se utilizar diferentes heurísticas. O artigo está organizado da seguinte forma, na presente Seção foi dada uma introdução, na Seção II é apresentado um referencial do *EvoSuite* e na seção III é apresentado um referencial de meta-heurísticas.

II. Referencial EvoSuite

EvoSuite é uma ferramenta que automatiza a geração de suítes de testes com alta taxa de cobertura, pequenas o suficiente e fornece asserções. Ele gera as suítes de testes tantos para classes individuais quanto para projetos inteiros, sem requerer comandos complicados adicionais. Esta ferramenta é livre, e pode ser usada em linha de comando ou como plugins

para da plataformas de desenvolvimento como por exemplo o Eclipse[1] [9].

O *EvoSuite* utiliza *whole test suite generation* e *Mutation-based assertion generation* para alcançar os seus objetivos eficientemente, os quais serão descritos a seguir serão descritos a seguir:

A. Whole Test Suite Generation

Uma abordagem comum na literatura é gerar um caso de teste para cada meta individuais de cobertura (ex, *branches in branch coverage*), e então os combinam em uma única suíte de teste. Porém, o tamanho resultante da suíte de testes é difícil de predizer pois um caso de teste gerado para uma meta pode implicitamente também cobrir qualquer quantidade de metas de cobertura à frente. Ou seja, a ordem em que cada meta é escolhida pode desempenhar um papel importante, pois pode haver dependências entre elas. Há outros problemas quando se considera uma meta de cada vez, como metas quem podem ser mais difíceis de cobrir do que outras, ou até mesmo serem inviáveis [10].

Para superar esses problemas, *whole test suite generation* é uma abordagem que não produz casos de testes para cada meta individual de cobertura, ao invés disso foca em suítes de testes visando um completo critério de cobertura. Otimizando em relação em critério de cobertura ao invés de metas individuais cobertura tem resultados que não são influenciados pela ordem, pela dificuldade nem pela inviabilidade dos casos de testes [9].

EvoSuite implementa *whole test suite generation* como um algoritmo genético, onde uma suíte de testes é considerada como uma solução candidata. Cada suíte de teste consiste de um número variável de casos de testes, que por sua vez, consiste em uma sequência de instruções. O crossover entre duas suítes de testes são trocados casos de testes entre si baseado em uma escolha aleatória da posição do crossover. A mutação de uma suíte de teste adiciona novos casos de testes, ou modifica testes individuais. Já a mutação de casos de testes são adicionados, removidos ou mudados instruções ou parâmetros.

O *fitness* dos indivíduos é calculado com base a um critério de cobertura, no caso o *EvoSuite* usa *branch coverage* como critério de teste. Para que seja realizada a evolução são recompensados aqueles indivíduos o quais possuem uma melhor cobertura. Quanto maior o tamanho das suítes de testes maior a chance de cobrir as metas de cobertura, e o *EvoSuite* permite que a busca aprofunde em sequencias longas, porém é aplicado várias técnicas de *bloat control*. Essa técnicas de *bloat control* assegura que indivíduos não fiquem excecivamente muito grandes, com isso, no fim da busca as suítes de testes são minimizadas tal que apenas instruções contribuindo para a cobertura permaneçam.

B. Mutation-Based Assertion Generation

Os casos de testes precisam de algum tipo de testes de oraculo manual para que assim o engenheiro de software verifique e capture a corretude da unidade sobre testes que não se pode detectar com oráculos automatizados. No caso de testes unitários esses oráculos manuais geralmente são asserções. Dado um teste de unidade gerado automaticamente, existe um número finito de código que pode ser gerado uma asserção. Porém mesmo que seja limitado, mostrar todas as asserções para o desenvolvedor pode ser problemático, pois para uma falha seja detectada o desenvolvedor precisa verificar a corretude das asserções, e muitas delas podem ser irrelevantes. Semelhante a isso, um caso de teste pode falhar em um ponto posterior, indicando falha de regressão, quando de fato ele possui muitas asserções e isso causa várias restrições, assim conduzindo a falsos alarmes [9].

Para determinar a importância e efetividade das asserções, o *EvoSuite* aplica testes de mutação. Em testes de mutação, defeitos (mutantes) são semeados no programa testados nas suítes de testes. Quanto mais defeitos detectados com o menor número de asserções melhor a qualidade do caso de teste, entretanto, se os defeitos não forem encontrados significa que o caso de teste precisa de melhoras. Após a geração dos casos de testes, o *EvoSuite* roda cada caso de teste no programa sem modificações assim como nos mutantes criados, assim analisando quais defeitos são encontrados pelas asserções. Após isto o *EvoSuite* seleciona as asserções suficientes para encontrar a maior quantidade de defeitos no programa em análise.

C. Resultados do EvoSuite

Em [8] é mostrado que a implementação de *whole test suite generation* como método de busca no *EvoSuite* apresenta um melhoramento significante em comparação com outros métodos de busca anteriores. O *EvoSuite* foi testado sobre 5 bibliotecas de código aberto e uma da indústria assim totalizando 1308 classes, onde a API foi chamada 727 classes públicas (as classes restantes são privadas ou anonimas). A média de cobertura obtida foi superior em relação com a abordagem padrão de um único critério de cobertura por vez.

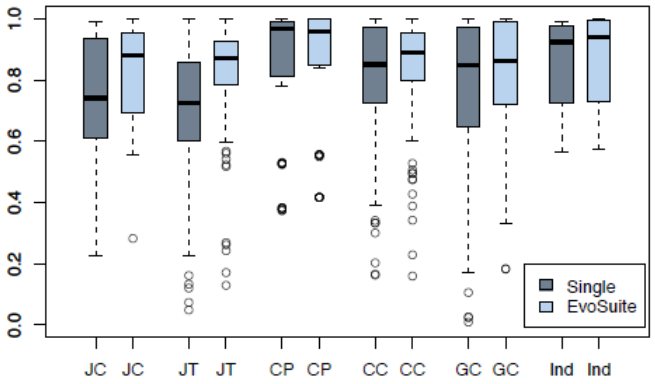

Fig. 1. Média de cobertura: Mesmo com limite de evolução de 1 000 000 instruções,*EvoSuite* alcançou cobertura superior [8]

Já em [11], foram testados 100 projetos de código aberto selecionados aleatoriamente no *EvoSuite*, a fim de avaliar

[1]http://www.evosuite.org/downloads/

a escalabilidade quando existem vários testes. Os autores mostram que o *EvoSuite* é capaz de exercitar oráculos automatizados e produzir alta cobertura de testes ao mesmo tempo. Nos experimentos foram encontrados 32594 falhas distintas em 8844 classes, tal que 1694 dessas falhas podem ser consideradas como falhas reais [11].

III. Meta-Heurísticas

Meta-heurística é um subcampo primário da otimização estocástica, que consiste de algoritmos e técnicas que empregam algum grau de aleatoriedade para encontrar a solução ótima (ou a melhor possível) para problemas reconhecidamente difíceis. [16] [16] ressalva que meta-heurística é um termo que pode gerar desentendimentos uma vez que não se trata de heuristica sobre (ou para) heurísticas, o que não é necessariamente verdade para todos os algoritmos.

As próximas secções descrevem as meta-heurísticas que iremos utiliza ou tentar utilizar no problema de teste de software com o *Evosuite*.

A. Colônia de formigas

Uma meta-heurística bio-inspirada que pode ser útil para o nosso projeto é Colônia de Formigas [6] (do inglês *Ant Colony Optimization*) que surgiu da observação do comportamento das formigas reais e incluso técnicas de busca local posteriormente. Utiliza algoritmo guloso para para construir a solução inicial e para soluções viáveis. Outro ponto importante é a quantidade de feromônio depositado pelas formigas que dá a visibilidade que é a capacidade dada às formigas de observar os elementos que compõem a solução ou o caminho que elas estão percorrendo.

Na Figura 2 mostra todo o processo da meta-heurística colônia de formigas. Basicamente a busca das formigas por um caminho fica em um laço pela quantidade dada de feromônio e caso encontre uma solução que seja o menor caminho, então para.

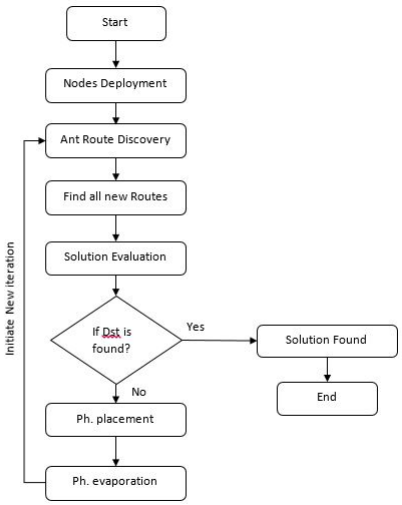

Fig. 2. Fluxo completo de todo o processo da meta-heurística colônia de formigas. [19]

B. Algoritmo Genético

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.

1) 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) Método de crossover: Operador de *crossover* $C1$ 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.

Operador de *crossover* 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.

Outro método de *crossover* de dois indivíduos A e B é usando *crossover* de dois pontos, que é feito 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.

Por fim, o *Uniform Crossover* 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 *crossover* 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.

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

3) 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. Existe um valor $1/m$ [12] que diz ser um valor ideal para mutações, onde m é o tamanho da instância de entrada.

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

4) 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.

5) Manutenção de diversidade: Para manter os indivíduos diversos e evitar a convergência muito cedo, pode ser usado 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 possuírem 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.

C. BRKGA

O BRKGA (biased random-key genetic algorithm) [14] é uma metaheuristica para encontrar uma solução ótima ou próxima da ótima. É uma variação do RKGA (random-key genetic algorithms) [1] 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 continuo $(0, 1]$ 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.

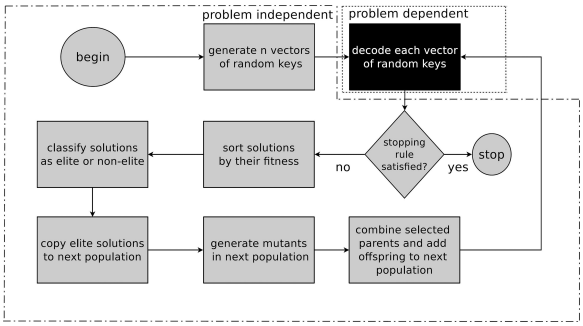

Fig. 3. Imagem mostrando o fluxo da meta-heurística BRKGA. [15]

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

- Codificação de uma solução com chaves aleatórias em intervalo continuo $[0, 1]$
- 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. Com estes é gerado a próxima geração.

Todas as variações descritas no algoritmo genético III-C são possíveis de se aplicadas no BRKGA.

D. GRASP

O GRASP (Greedy Randomized Adaptive Search Procedure) para problemas de otimização combinatória foi introduzida por Feo e Resende [7]. 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 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.

Na Figura 4 temos todo o processo da meta-heurística GRASP. Primeiro existe a fase de construção que pode ser gulosa ou aleatória, após o método de busca local e fica neste processo até um critério de parada seja satisfeito.

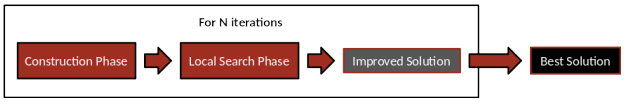

Fig. 4. Fluxo de todo o processo da meta-heurística GRASP. [15]

Algumas variações que podem ser aplicadas:

1) **Reactive GRASP**: A próxima escolha entre aleatório ou guloso fica definida pelas iterações anteriores e não originalmente que é sempre aleatória;
2) **POP in Construction**: Aplicando busca local durante a fase de construção com o principio de que boas soluções estão mais provavelmente próximas a melhores soluções.
3) **Busca Tabu**: Podemos usar a Busca Tabu como processo de busca local do GRASP.

E. 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 proposta por [13] [13] 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 [12].

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 meta-heurística suficientemente genética para ser usada no *EvoSuite*, porém podemos pensar em casos não muito gerais e aplicar mecanismos de busca por intensificação e por diversidade.

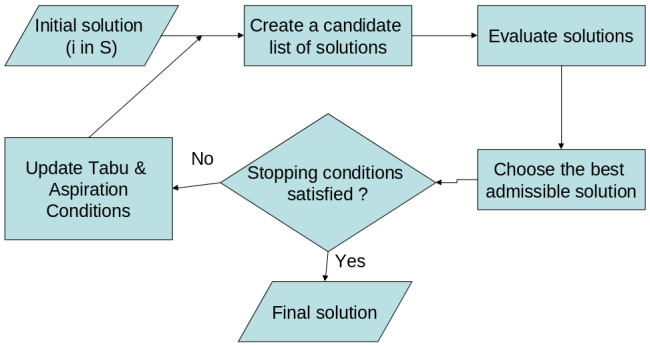

Fig. 5. Imagem ilustrativa de todo o processo da meta-heurística Busca Tabu. [10]

1) Intensificação: Uma possível intensificação por vizinhança é que no qual são explorados todos os movimentos possíveis (inserção, remoção e troca) com quaisquer três elementos.

2) Método de Busca Local:

- *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.
- *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.

3) 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 calculo 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*.

4) Diversificação por Reinício: Envolve forçar alguns componentes que são raramente utilizados na solução e reiniciar o processo de busca a partir deste ponto. 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.

REFERENCES

[1] James C Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, 6(2):154–160, 1994.

[2] Korel Bogdan. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[3] Korel Bogdan. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[4] Oliver Bühler and Joachim Wegener. Evolutionary functional testing of an automated parking system. *International Conference on COmputer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis*, 01 2003.

[5] Oliver Bühler and Joachim Wegener. Evolutionary functional testing. *Comput. Oper. Res.*, 35(10):3144–3160, October 2008.

[6] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages 1470–1477. IEEE, 1999.

[7] Thomas A Feo and Mauricio GC Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

[8] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *Quality Software (QSIC), 2011 11th International Conference on*, pages 31–40. IEEE, 2011.

[9] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[10] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[11] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[12] Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[13] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[14] José Fernando Gonçalves and Mauricio GC Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, 2011.

[15] José Fernando Gonçalves, Mauricio GC Resende, and Rodrigo F Toso. An experimental comparison of biased and unbiased random-key genetic algorithms. *Pesquisa Operacional*, 34(2):143–164, 2014.

[16] Sean Luke. *Essentials of metaheuristics*, volume 113. Lulu Raleigh, 2009.

[17] P. McMinn. Search-based software testing: Past, present and future. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[18] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[19] Anand Nayyar and Rajeshwar Singh. Simulation and performance comparison of ant colony optimization (aco) routing protocol with aodv, dsdv, dsr routing protocols of wireless sensor networks using ns-2 simulator. *American Journal of Intelligent Systems*, 7(1):19–30, 2017.