A Primer in – SBSE - Search Based Software Engineering

This material was kindly provided by Mark Harman, King’s College London
Does this look familiar? You have to change and test it!
In SBSE we apply search techniques to search large search spaces, guided by a fitness function that captures properties of the acceptable software artefacts we seek.
In SBSE we apply search techniques to search large search spaces, guided by a fitness function that captures properties of the acceptable software artefacts we seek.

Genetic Algorithms, Hill climbing, Simulated Annealing, Random, Tabu Search, Estimation of Distribution Algorithms, Particle Swarm Optimization, Ant Colonies, LP, Genetic programming, Greedy
Checking vs Generating

Search Based Software Engineering
Write a method to determine which is the better of two solutions

Conventional Software Engineering
Write a method to construct a perfect solution
Check vs Generating

Search Based Software Engineering
Write a method to determine which is the better of two solutions

Conventional Software Engineering
Write a method to construct a perfect solution
Checking vs Generating

Search Based Software Engineering

Write a *method* to determine which is the better of two solutions

Conventional Software Engineering

Write a method to construct a perfect solution
Checking vs Generating

Search Based Software Engineering
Write a **fitness function (also said cost function)** to determine which is the better of two solutions

Conventional Software Engineering
Write a method to construct a perfect solution
Search Based Software Engineering
Write a fitness function to guide a search a solution

Conventional Software Engineering
Write a method to construct a perfect solution
Checking vs Generating

Search Based Software Engineering
Write a *fitness function* to guide *automated search*

Conventional Software Engineering
*Write a method to construct a perfect solution*
Hill Climbing

What
8 Queens
Scale up
Two visions
Evolutionary Algor.
Hill Climbing
Applications
Ingredients
Conclusions
Hill Climbing

What
- 8 Queens
- Scale up
- Two visions

Evolutionary Algor.
- Hill Climbing

Applications
- Ingredients
- Conclusions

SBSE

ET

Conclusion

INF6305 - SBSE Primer - Hiver/Winter 2009
Hill Climbing

SBSE

- What
- 8 Queens
- Scale up
- Two visions
- Evolutionary Algor.
  - Hill Climbing
- Applications
- Ingredients
- Conclusions

ET

Conclusion

INF6305 - SBSE Primer - Hiver/Winter 2009
bP = null;

while(searching())
{
    p = selectNext();
    if(p.isBetter(bP))
    {
        bP = p;
    }
}
Genetic Algorithms

- Gene
- Chromosome
- Population
Evolutionary Algorithms

- Selection
- Insertion
- Recombination
- Mutation
- Fitness evaluation

SBSE
- What
- 8 Queens
- Scale up
- Two visions
- Evolutionary Algor.
- Applications
- Ingredients
- Conclusions

ET

Conclusion

INF6305 - SBSE Primer - Hiver/Winter 2009
Genetic Algorithms (GAs)

- Come from an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems.

- applied to different domains such as machine and robot learning, economics, operational research, ecology, studies of evolution, learning and social systems

- Useful when:
  - The search space is large
  - No mathematical analysis available
  - Problem finding the solution of which is NP-Hard or NP-Complete

- Reference:
**How does a GA work?**

A finite set of strings, array, …, the genome

Crossover operator: produces individuals from two parents, exchanging parts of their genomes.

Mutation operator: randomly modifies an individual's genome.

Selection operator: selects individuals for the reproduction.

**8 Queens**

**Evolutionary Algor.**

**Conclusion**

**Scale up**

**Two visions**

**Evolutionary Algor.**

**Applications**

**Ingredients**

**Conclusions**

**SBSE**

- What
- 8 Queens
- Scale up
- Two visions
- Evolutionary Algor.
- GA

**ET**

**Conclusion**
Initialize population $P[0]$;

generation=0;

while (generation < max_number_of.generations)
    Evaluate $P[generation]$;
    generation=generation+1;
    Select $P[generation]$ from $P[generation-1]$;
    Crossover $P[generation]$;
    Mutate $P[generation]$;
end while
Evolutionary Testing: HowTo

- Search space: the system under test (SUT) input domain(s)
- Test objective needs to be defined numerically and transformed in a fitness function
- Fitness is computed by monitoring program execution results (output, performance, etc.)
- Iterative procedure
Evolutionary Testing

Start with a set of randomly generated test input data and ..
Evolutionary Testing

Execute on test data to gather information

ET HowTo

ET

Motivation

SBSE

GA Encoding

Structural Test.

Buffer Overflow

Mutation Test.

Conclusion

Insertion

Mutation

Recombination

Selection

Test cases

Fitness evaluation

End?

Execution
Monitored value are used to compute the fitness
[Wegener, 97]
Applying GA to Testing

- The genome encodes a test suite
  - Starting from randomly generated test cases – test input data
  - Evolution towards a testing objective, indicated in the fitness function
  - Known applications
    - Structural testing
    - Safety testing
      - buffer overflow
    - Stress testing
    - Temporal behavior testing
    - Mutation testing
## Genome encoding

<table>
<thead>
<tr>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
<th>...</th>
<th>...</th>
<th>$I_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{x1}$</td>
<td>&lt;24.5&gt;</td>
<td>&lt;11.3&gt;</td>
<td>&lt;abc&gt;</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$T_{x5}$</td>
<td>&lt;7.6&gt;</td>
<td>&lt;4.7&gt;</td>
<td>&lt;def&gt;</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Typical Parameter Settings**

- $\text{dimpop} = 70$
- $\text{ngen} = 500$
- $\text{Pmutation} = 0.01$

** creep mutator **

INF6305 - SBSE Primer - Hiver/Winter 2009
whole arithmetical crossover

one-point crossover

\[
\text{Child 1} = \alpha \cdot x + (1-\alpha) \cdot y \\
\text{Child 2} = \alpha \cdot y + (1-\alpha) \cdot x
\]
- **Aim**
  - Generate a set of test data to cover structural properties automatically

- **Coverage-oriented approaches**
  - Fitness proportional to the number of CFG nodes covered

- **Distance-oriented approaches**
  - Test coverage criteria is partitioned in sub-goals
  - A fitness function for each sub goal
Coverage-Oriented

- Fitness of individual for different coverage criteria
  - % of covered statements
  - % of covered branches
  - Path coverage: 1/overall execution of path

- Good results, better than random testing

Individual 1: fitness=2 (2 branches)
Individual 2: fitness=3 (3 branches)
Robustness Testing

- **Motivation**
  - Lack of testing for error handling and exceptions (developers tend to test normal behavior)

- **Objective**
  - Generate test data in order to raise exceptions
  - Individuals: sequence of API calls with parameter values
  - Fitness function considers:
    - Return status of API calls (ok, nok, exception)
    - Characteristics of the sequence (e.g. length)
Safety Testing – Speed limit

- **Objective**
  - Safety constraints are specified, and they should never be violated

- **Test**
  - Generate test data to violate safety constraint
  - Fitness function: distance from violating safety condition
  - See Tracey et al. work, University of York

Generated Test data

\[
F = 100 - \text{speed} + K
\]

K: smallest arithmetic step size

If \( F \leq 0 \) \( \rightarrow \) test successful, safety condition violated
Safety/Security Testing
Safety Testing - Buffer Overflow

- Unwanted condition about 50% of detected vulnerabilities – CERT
- Unsafe condition: BUFFER OVERRUN
  - crash is not required
- Implicit safety constraints

```c
1: int main (int argc, char **argv){
2:     char buff[5];
3:     int how_many = atoi(argv[1]), p=0, w=0;
4:     strcpy(buff,argv[0]);
5:     if (argc==3){
6:         for (p=0; p< how_many; p++)
7:             strcat(buff,argv[2]);
8:     } else if (argc>3){
9:         for (p=0; p< how_many; p++)
10:            for (w=2; w< argc; w++)
11:                strcat(buff,argv[w]);
12:     printf("Buffer: %s\n",buff);
13:     exit(0);
}
```

INF6305 - SBSE Primer - Hiver/Winter 2009
The factors of the fitness function are:

1. **Statement Coverage**, code coverage contributes to increase the likelihood that an exception is raised
2. **Vulnerable Statement Coverage** quickly drives towards buffer overflow detection
3. Max nesting: target deeper statements
4. Distance from buffer limit

\[
\text{Fitness}(g) = w_1 \text{sc}_g + w_2 \log(k_g)\nu\text{cov}_g + \\
\quad w_3 \text{nesting}_g + w_4 \max_i \left[ \min_j \left( L_{ij}(g) - SB_i \right) \right]
\]
But ..
  - It really favor crashes …
  - An overrun is just enough for an unsafe programming pattern …
  - It is not amenable for complete automation ….  
    - weight are tuned by hand …. 

But with a minor change:

\[
\text{Fitness}(g) = w_1 \text{sc}_g + w_2 \log(k_g) \nu \text{cov}_g + \\
w_3 \text{nesting}_g + w_4 \max_i \left[ \min_j \left( L_{ij}(g) \right) \right]
\]
The Linear Problem

\[
\max_{w_1, w_2, w_3, w_4} \left\{ \sum_{g \in \text{Population}} \text{Fitness}(g) \right\}
\]

subject to

\[
0 \leq w_i \leq 1 \quad i = 1, 2, 3, 4
\]

\[
w_1 + w_2 + w_3 + w_4 = 1
\]

\[
\forall g \in \text{Population} : \\
\text{Fitness}(g) \geq 0
\]

\[
w_1 \text{sc}_g \leq w_2 \log(k_g) \cdot \text{cov}_g
\]

\[
w_2 \log(k_g) \cdot \text{cov}_g \leq w_3 \text{nesting}_g
\]

\[
w_3 \text{nesting}_g \leq w_4 \max_i \left\{ \min_j L_{i,j}(g) \right\}
\]