Automatic generation of GUI test cases using Ant Colony Optimization and Greedy algorithm.

Jose Rodriguez[1] and Glen D. Rodriguez[23]

[1] Universidad Nacional Mayor de San Marcos, Lima, Peru,
ingpibe@gmail.com,
WWW home page: http://www.unmsm.edu.pe
[2] Universidad Nacional de Ingenieria, Lima, Peru,
grodriguez@uni.edu.pe,
WWW home page: http://www.uni.edu.pe
[3] Universidad Ricardo Palma, Lima, Peru,
glen.rodriguez@gmail.com,
WWW home page: http://www.urp.edu.pe

Abstract. The increasing complexity of new applications means GUIs are also getting more complex, and generating tests cases manually for them becomes harder. Generating automatic, good quality GUI test cases is a growing concern in application testing. Actions performed by the user on the GUI can be regarded as events, which can be performed in sequences, forming a graph of event sequences, and therefore multiple execution paths or routes, known as test cases, are possible. The quality of a set of test cases is measured by the coverage criteria (all actions or events must be performed at least one time in the set), which depend on the length and partial coverage of each execution path. Finding feasible paths and complying with the coverage criteria is a highly combinatorial problem. For such problems, due to high computing power that it would take to find an exact solution, it is well justified to use heuristics and metaheuristics algorithms, allowing us to find approximate solutions of good quality. Those methods have been successfully used in chemistry, physics, biology, and recently, in software engineering. In this paper, the use of a metaheuristic known as Ant Colony Optimization Algorithm (ACO) for generating test cases is proposed. The ACO metaheuristic has been adapted in order to find individual routes that could lead to a set of test cases of good quality. A individual test, path or route is desirable if it is long (it tests a lot of events or actions) and do not share events (or share few events) with other paths. After a appropiate number of candidate test cases are generated, we express the problem of generating a set of test cases as a set covering problem and then we apply a greedy algorithm to solve it. The result is a set of paths (test cases) with full covering of events with small number of test cases. We present also a problem solved by our method, generating test cases for Windows Wordpad, and discuss the results.

Keywords: GUI testing, automated test case generation, Ant Colony Optimization (ACO), event-flow model, set covering problems, greedy algorithm, shortest path

1 Introduction

Software application are growing in complexity year after year. Graphical User Interface (GUI) grows both regarding the number of objects subject to user interaction (buttons, menu options, mouse-triggered events, etc.) and regarding dependencies between GUI objects status. Traditionally this true only for desktop-based applications, but the coming-of-age of AJAX technologies means that this is also true for many web based applications.

GUI testing is an important task in software engineering [20]. A study says that in average GUI development takes up to 48% of source code and 50% of time in the implementation phase [16]. A bad implemented GUI can also impact software reliability; this has been studied in smartphone applications [12]. For a complex GUI, manual generation of test cases and their maintenance, evaluation, and conformance to coverage criteria are very time consuming [13]. According some studies an average of 6% of all bugs found in a system are due to GUI errors [4]. Generating automatic, good quality GUI test cases is a growing concern in application testing, and it s popularity is increasing [9]. If you add the fact that most teams have very strict time and money constraints, and in practice the quality of testing is an afterthought.

To help testers, in the last 15 years there has been efforts in order to automate the generation and execution of GUI test cases; those efforts are oriented towards proposing models for GUI interactions, generating test cases based in one of those models and search algorithms, and developing tools for the task. This papers deals basically with the automatic generation problem. No new model is proposed, but a new approach to search and a tool are presented.

This paper is organized as follows. In section 2, a revision of recent research is shown. In section 3, the modeling of the GUI as a graph, and of each test case as a sequence of events mapped into the graph is explained. In section 4, an algorithm based in Ant Colony Optimization (ACO) and a Greedy algorithm is presented. In section 5, a test problem is presented, and the results are shown and discussed.

2 Revision of related research

According to [14], a GUI can be defined as a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI.

There are two high level models for GUI testing. The oldest is the finite state machine model, proposed first by Hu for any kind of software [5]. Later, it was adapted to user interfase testing, for example on the variable finite state machine (VFSM) model [18], which uses the W_p algorithm proposed by Fujiwara et al [10] to generate test cases. The disadvantages of these model and the algorithms associated are: it does not discard the NULL transitions (transitions which are back to the originating state) leading to potentially repeated identical tests differing only by a NULL [18]; and it is not scalable [20].

A recent research uses generated test cases and log files from running those test, and then does GUI bug mining in order to find new errors (results with behaviour similar to known errors). It can be considered an new empirical based approach to finite state machine model [11]

The second model is based in graphs and called event-flow model [13][14]. The model is presented in detail on section 3 of this paper. After modeling the desired GUI (as it is supossed to be) as a graph, a artificial intelligence technique called Planning. To avoid defining too much operators that could slow down AI planning, events are classified into menu-open events, unrestricted-focus events, restricted-focus events and system-interaction events; later planning operators are reduced to system-interaction operators or abstract operators, according to which events they are related to. The tester choose an initial state and a goal state, and the AI planner suggest one or more sequences of events leading from the initial to the goal state [13].

The final version of the event-flow model starts with encoding each event in term of preconditions and effects, and then created the direct graph for the GUI [14]. Because GUIs are hierarchical, the graph is not fully connected (most graphs are sparse), and this hierarchy may be exploited to identify groups of GUI events that may be modelled in isolation (for example: modal windows). Test case generation was done using a graph-traversal algorithm.

A more recent research have been done by Bauersfeld et al [2][3], that uses Memon's event-flow model and generates one optimal sequence of events by Ant Colony Optimization (ACO). Here, the goal of ACO is to find a sequence of events that maximize the amount of method calls in the source code associated with the navigation across the GUI corresponding to the sequence of events. The implicit goal is testing as many methods as possible with one sequence of events.

Actual techniques used in practice to testing GUIs are incomplete and not fully automatic. Regarding tools, there are some like HP WinRunner and its successors with some degree of automatization, particulary in test case recording and execution. Hp WinRunner generates a GUI map automatically or by monitoring user's interaction with the software, it checks for good GUI design practices (example: mnemonics for all buttons or menu options, labels not too large) but the tester needs to create a TSL (Test Scriptling Language) script to define and run the test cases[15]. Ultimately WinRunner is more oriented to supporting functional and regression testing; good coverage criteria is left as testers' reponsibility at the time they create the TSL scripts.

Other tool is jfcUnit, an extension to the popular testing framework JUnit for testing Java Swing applications. It can obtain handles on Windows/Dialogs opened by the Java code, locate components within a component hierarchy, raise events on the found components (e.g. clicking a button, typing text in a textField component) and check if the result of these events are as expected or not. The testers, too, have to design the test by hand, and it is their responsability to meet the coverage criteria.

Abbot also was created for testing Java Swing applications. It is oriented to regresssion testing: by recording user's interaction with the GUI and replaying it later, it can detect undesired changes in the GUI state machine due to modifications on the source code. It is also inspired by JUnit, and it supports scripts with a function similar to jfcUnit. The test case design is manual [19]. Pounder is another tool similar tojfcUnit and Abbot [17]. None of these tools can generate a set of test cases, at most they can run the test cases designed by the humans.

There are some studies regarding improving test cases or modifying them after a software change, but their relation with this research is weak. The exploration of the event search space should be extensive for test case generation, but it can be local for test case improvement or modification.

The research presented here can be considered both a modification and an extension of research mentioned previuosuly by Memon [13][14] and Bauersfeld [2][3]. We use Memon's event-flow model but not its planning approach. We use ACO as Bauersfeld did, but instead of focusing in a single sequence of events, we contemplate a set of sequence with the goal of achiving complete test coverage of GUI's components (widgets) with the minimum amount of sequences. The generation of an optimal sequence on Bauersfeld's approach is the unique goal of the ACO algorithm, but in creating a set of sequences, non repetition of events is also considered in our approach. Our final goal it is not testing many methods with onley one test (sequence of events); it is testing all components or widgets in a GUI at least once using a set of test cases of minimum size.

3 Event-flow model used in this research

The event-flow model can represent all possible sequences of events in a GUI as a graph. The nodes represent an event e in the GUI. An event is a response of the system to a user interaction (a click on a button triggers an onClick event). Some events (at least one) can be executed directly after the application launched, and are called initial events. An directed edge (e, e') between two events e, e' states that the event e' can be executed immediately after the event e. Conversely, if there is no directed edge between events e, e' then event e' cannot be executed immediately after event e [1]. An example of a GUI and its corresponding graph is shwon in fig.1. In that figure, it can be apreciated that some nodes have two way links (for example, File and New) but others are one way only (Send to $\rightarrow$ Routing R.). Some case, as Edit and Copy, are two way as a fiction (after Copy, most software automatically goes bak to File, or exit the menu). Inital events those available at the GUI's start, as File, Edit, etc. in most text editors.

In this paper, a simplified event-flow model is used. The main difference with the complete model [14] is that different states are not considered. Therefore the graph is always the same, and the ability to reach an event depends only on the actual event, not on past events in the sequence.

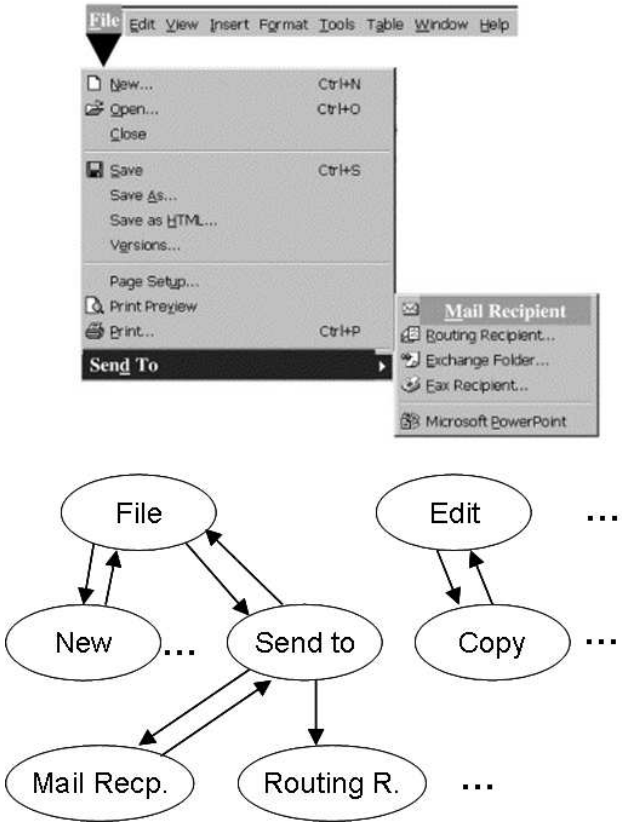

Fig. 1. A GUI and its corresponding event-flow graph

4 Algorithm

The problem of generating a good test case can be defined as: given a event-flow graph representing a GUI, find the smallest set of paths in the graph such as a coverga criteria is met, where each path represent a event sequence valid on the GUI. The coverage criteria is that each event appears as part of al least c paths, with c a predefined integer. In this paper, $c = 1$ as our first research effort in this field.

The problem can be divided into 2 subproblems. Looking at the final result, the last subproblem it is easily described as a set coverage problem. Set covering is defined as: given a collection S of sets over a universe U, find the minimum-cardinality set cover, where a set cover C $\subseteq$ S is a subcollection of the sets whose union is U [21]. Set coverage is a NP-complete problem, so it is very hard to find the best solution; but it is possible to quickly find close to best solutions by heuristics and meta-heuristics. A greedy heuristic has been chosen by its simplicity and speed, and because it has been proved that a greedy algorithm provides good approximations to the optimal solution for both basic set covering and more general weighted set covering[6]. Greedy heuristics are a family of contructive heuristics, that is, they build a solution step by step, making the locally optimal choice at each step with the hope of finding a global optimum. Regarding the greedy algorithm set covering, the candidate set of solutions S is finite and its elements should be valid paths in the event-flow graph.

Therefore, the previous subproblem it is how to generate good paths (elements of S) for the greedy algorithm, because it is not practical to generate all possible paths in the praph (slow to generate, and generates large S which occupies much memory and takes more time for greedy heuristics). The logical procedure would be to generate a not-so-large number of paths , where the nodes common to any two different paths is small in average (this last criteria should lead to a covering of U with the least number of paths). With these ideas in mind, a modified Ant Colony Optimization (ACO) is proposed here. ACO is a metaheuristic born originally to find optimal path in a graph [7] and it has been applied to many kinds of problemas such as traveling salwesman, vehicle routing, scheduling, etc. [8]. The goal here is to obtain many paths, not only one optimal path; because ACO is very good for find paths in graph problems, it looked promising.

ACO is a metaheuristic that is inspired by the pheromone trail laying and following behavior of some ant species. Artificial ants in ACO are stochastic solution construction procedures that build candidate solutions for the problem instance under concern by exploiting artificial pheromone information that is adapted based on the search experience of the ants and on available heuristic information [8]. The basic loop on ACO is: while some termination condition is not met do (a) Construct Ant solutions, (b) Apply local search (optional) and (c) Update pheromones [8].

Before presenting the algorithm, the variables, their definitions and formulae are explained in table 1.

Before continuing, an important observation must be made. The main requirement for this ACO+Greedy method is to have the desired event-flow model at the start; that is, the software engineer must write down the expected behaviour of the GUI (transitions between events allowed and transitions not allowed) as a graph. ACO+Greedy then can be used to generate the set of test cases required to test all events at least once. There is no guarantee to test all transitions, but if we suppose that each event (representing a GUI component or widget) is associated with a method call, then all methods associated with GUI

Table 1. Variables and their meaning

NC	counter for the ACO iteration
NC_{max}	maximum number of iterations
N	Number of events on the GUI (number of nodes in the graph)
m	number of ants
τ	pheromone matrix
S	list of partial solutions
TC	set of test cases
LT_k	Tabu list, nodes already visited for ant k in actual iteration
LO	List of nodes not yet visited
LND	List of destination nodes
LG	Global list of nodes already visited
L_k	length of actual path
d	distance matrix (1=there is edge from node i to node j, 0 otherwise)
η	visibility matrix ($1/d_{ij}$ if $d_{ij} > 0$, 0 otherwise)
$\Delta\tau_{ij}^m$	amount of pheromone deposited by ant m on edge i→j
ρ	persistency coefficient for pheromones
$\tau_{ij} = \rho.\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}$	Actualization of pheromone
$P_{ij} = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{j \notin LT} \tau_{ij}^{\alpha}\eta_{ij}^{\beta}}$	Probablity of the ant going from node i to j
α	coefficient for the pheromone
β	coefficient for the distance
Q	pheromone constant for one ant, Q/L_k is deposited along the path
LI	list of initial nodes (initial events)
$S1$	adjacency matrix of S, $S1_{ij} = 1$ if path i does contain node j, 0 otherwise

input are tested at least once. And the probability of catching a transtion error (transition allowed in the GUI design but not implemented in the source code, example: a text editor is supposed to have disabled the Paste widget disabled at its start, but after clicking the Copy widget, Paste should be enabled, but the programmer forgot to code it) is greater if we run through a set of test cases instead of only one test (which is the approach of previous research [13][14][2][3]).

We start with the graph, the initial nodes or events and parameters for the ACO. The logic of this algorithm is: at first many paths (sequences of events) are created, then any repeated path is deleted, and then a covering for the set of all events is generated. The elements of the set covering are the test cases. as seen in algorithm 1 (ACO+Greedy).

Algorithm 1 `ACO+Greedy`

Require: $d, \rho, \alpha, \beta, LI$
1: S=ACO($d, \rho, \alpha, \beta, LI$)
2: Eliminate repetitions in S
3: TC=Greedy(S)
4: **return** TC

The algorithm 2 (ACO) is a modification of the ACO for finding the shortest path between an origin node and a destination node. The test case generation needs not the best path, it needs many paths which could cover the set U of all events (nodes). This is achieved by not having a destination node. The algorithm starts with matrix d (representing the event-flow graph) and a list LI (list of initial nodes); it tries to generate a path starting from a random node e $\in$ LI, and tries to create a path as long as possible (see loop at lines 11–22) without re-visiting any event already visited by any ant in the actual iteration (until it is impossible to reach a new node directly from the actual node) using a Tabu List (LT_k) for keeping tabs on already-visited nodes, and then actualizes the pheromeone matrix . The next event is chosen randomly but using probability matrix P, which depends on the pheromone matrix and the distances (see table 1). The next ant does the same (see loop at lines 3–25), until no node is left unvisited or until all further paths are blocked. In the first case, it skips to the next iteration (see lines 7–8). In the last case, it ignores the tabu list and starts creating paths between a random initial node and a random non-covered node using Dijkstra shortest path algorithm (see lines 18–20). After all ants are done, it evaporates some ratio $1-\rho$ of pheromones (see line 27) and it goes to the next iteration. All the paths from all the ants on all iterations all saves in list S (see line 28).

Algorithm 3 (Greedy) starts by converting S into its adjacency matrix S1. Information about order in the event sequence is lost, but it is not necessary for solving the covering set problem. An example of the original S and its matrix form is shown in figure 2. In that figure, should S has only the four paths and the GUI only the five events shown, the optimal covering would be the set

Algorithm 2 `ACO`

Require: $d, \rho, \alpha, \beta, LI$

for $NC = 1; NC <= NC_{max}; NC++$ (Iterations) **do**
  LT=$\emptyset$, S=$\emptyset$
  for $k = 1; k <= m; k++$ (Ants) **do**
    Update LO=LO-{nodes visited in this iteration}
    Choose node e at random from LI
    if LO=$\emptyset$ **then**
      Update LG
      Break from inner FOR loop
    else
      j=1
      while $j <= N$ **do**
        if Is there at least one node $e' \notin LT_k$ with an edge $e \rightarrow e'$ **then**
          Select next node e at random using P, from all e' candidates
          if $e \in LO$ update LND
          j=j+1
          Update P
          Update LT_k
          if LND=$\emptyset$ **then**
            Use Dijkstra shortest path to create the path from actual node e to a random node in LO, ignoring tabu lists; update j if new nodes are visited
          end if
        end if
      end while
      Update LG
    end if
  end for
  Calculate path length L_k
  Update τ, evaporating pheromones
  S=S $\cup$ Path ant 1 $\cup$ Path ant 2 $\cup$ ... $\cup$ Path ant m
end for
return S

$\{path_1, path_2\}$ or the set $\{path_3, path_w\}$, both with cardinality 2. The greedy heuristic is an step by step procedure; each step takes the path p_i with the maximum covering over the non-covered events, then set asides that path, and eliminate the columns of matrix S1 whic are covered by p_i, thus eliminating those events from the list of non-covered events. It continues until there are no more non-covered events.

S={ {e_1, e_{N-1}}, {e_3,e_N,e_2},{e_2,e_1,e_N}, ..., {e_3,e_{N-1}} }
$path_1$: e_1→ e_{N-1}
$path_2$: e_3 → e_N → e_2

Path\Event (node)	e_1	e_2	e_3	…	e_{N-1}	e_N
$path_1$	1	0	0		1	0
$path_2$	0	1	1		0	1
$path_3$	1	1	0		0	1
…						
$path_w$	0	0	1		1	0

Fig. 2. A set of paths S and its matrix form

5 Results and analysis

A set of test cases was created for the GUI of MS Windows Wordpad. Wordpad was chosen because, as mentioned in section 4, the main requirement for this method is to design the expected or correct event-flow model (graph), a task for the human software engineer, and Wordpad have a medium-sized GUI that is not so complex that creating its graph become burdensome and error-prone for the human engineer. The number of events are 73, and the number of directed edges is 154. The algorithm has been proved with different combinations of values of α(0, 0.2, 0.4, 0.6, 0.8, 1) and β(0, 0.2, 0.4, 0.6, 0.8, 1, 1.4, 1.8, 2.2, 2.6, 3), for a total of 66 combinations, and 35 ants per iteration were used. Each comination was tested 5 times in order to account for the random nature of the ACO. The quality of the best solution for each combination is shown in table 2 and quality of the average solution is shown in table 3. As it can noted, the best combination is $\alpha = 1$ and $\beta = 0.4$ with an average solution of size 27 and a best solution of size 26. Both are the best results between the results of all 66 combinations. The processing time was less than 1 minute in a conventional desktop PC.

Algorithm 3 `Greedy`

Require: S

TC=$\emptyset$
Create $S1$ (Conectivity matrix of S)
nr= number of rows on S1
nc= number of columns on S1
while $max(S1_{ij} > 0)$ **do**
  $i = Argmax \sum_j S1_{ij}$
  $TC = TC \cup$ i-th element from S
  for kr=1;kr¡=nr;kr++ **do**
    for kc=1;kc¡=nc;kc++ **do**
      if $S1_{i,kc} = 1$ **then**
        $S1_{kr,kc} = 0$
      end if
    end for
  end for
end while
return TC

Table 2. Best solution (minimum cardinality of set of test cases)

		α					
		0	0.2	0.4	0.6	0.8	1.0
β	0	28	27	27	26	27	27
	0.2	28	27	27	27	26	28
	0.4	28	27	26	26	28	26
	0.6	27	27	27	27	26	27
	0.8	27	27	26	27	26	27
	1	27	27	26	26	27	27
	1.4	27	27	27	26	26	27
	1.8	27	27	26	27	26	27
	2.2	28	27	26	26	27	27
	2.6	27	27	27	28	27	27
	3	28	27	27	27	27	27

Table 3. Average solution (average cardinality of set of test cases)

		α					
		0	0.2	0.4	0.6	0.8	1.0
β	0	29	28	28.2	27.6	27.6	28.2
	0.2	28.6	27.6	28	29.2	28	28.6
	0.4	28.8	28.2	27.6	27.8	29	27
	0.6	29	28.4	28.2	28.8	28	28.4
	0.8	29	28.2	27.6	27.8	27.2	28.2
	1	28	28.6	28.4	27.2	27.8	28.4
	1.4	28.6	27.8	28	27.2	27.8	28.2
	1.8	28.4	28.6	27.4	28	28	28.6
	2.2	28.6	28.8	27.8	27.2	28.2	28.4
	2.6	27.8	28	27.8	28.4	28	28
	3	28.8	28	28.2	27.8	28.2	28

6 Conclusions

ACO proves to be a powerful approach to create a candidate set for converting the GUI automatic test case generation into a manageable set covering problem, because the candidate paths are good paths with less overlaping. The number of ants should be at least half the number of events, and the best combination of parameters α and β was found experimentally. The speed of this algorithm is very good and it is far fastest than human test cases generation, but the experiment done is not very complex. A more complex experiment should be done in the future for better validation of this approach.

Other papers suggest that each event should be covered at least by 5 different test cases. This algorithm may adapt to that problem, and it is an research opportunity. By now, our software tools read data from a text file representing the adjacency matrix. In the future, some graphical way for entering the event-flow graph could be developed. The dependency of one event on another events (states) has not been studied here, and it is another research challenge ahead.

References

1. S. Arlt, I. Banerjee, C. Bertolini, A. M. Memon, and M. Schäf. Grey-box GUI testing: Efficient generation of event sequences. *CoRR*, abs/1205.4928, 2012.
2. S. Bauersfeld, S. Wappler, and J. Wegener. An approach to automatic input sequence generation for gui testing using ant colony optimization. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, pages 251–252, New York, NY, USA, 2011. ACM.
3. S. Bauersfeld, S. Wappler, and J. Wegener. A metaheuristic approach to test sequence generation for applications with a gui. In M. Cohen and M. Cinnide, editors, *Search Based Software Engineering*, volume 6956 of *Lecture Notes in Computer Science*, pages 1173–187. Springer Berlin Heidelberg, 2011.
4. P. Brooks, B. Robinson, and A. Memon. An initial characterization of industrial graphical user interface systems. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 11–20. IEEE, 2009.
5. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978.
6. V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.
7. M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2. IEEE, 1999.
8. M. Dorigo and T. Stützle. Ant colony optimization: overview and recent advances. *Handbook of metaheuristics*, pages 227–263, 2010.
9. E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.
10. S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, June 1991.
11. C. Hu and I. Neamtiu. Automating GUI testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

12. A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile OSes: A case study with android and symbian. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 249–258. IEEE, 2010.
13. A. Memon, M. Pollack, and M. Soffa. Hierarchical GUI test case generation using automated planning. *Software Engineering, IEEE Transactions on*, 27(2):144–155, 2001.
14. A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test. Verif. Reliab.*, 17(3):137–157, Sept. 2007.
15. Mercury Interactive. WinRunner Users Guide Version 7.6. `http://www.cbueche.de/WinRunner\%20User\%20Guide.pdf`, 2003.
16. B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems CHI 1992*, New York, NY, USA, 1992. ACM.
17. M. Pekar. Pounder. `http://pounder.sourceforge.net/`, 2002.
18. R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, FTCS '97, pages 80–88, Washington, DC, USA, 1997. IEEE Computer Society.
19. T. Wall. Getting Started with the Abbot Java GUI Test Framework. `http://abbot.sourceforge.net/doc/overview.shtml`, 2011.
20. L. White, H. Almezen, and N. Alzeidi. User-based testing of GUI sequences and their interactions. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ISSRE '01, pages 54–63, Washington, DC, USA, 2001. IEEE Computer Society.
21. N. E. Young. Greedy set-cover algorithms. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.