CHAPTER 4

INTELLIGENT TESTER-INTELLIGENT SEARCH AGENT
BASED TEST SEQUENCE OPTIMIZATION

4.1 PROBLEM FORMULATION

The objective of test sequence optimization is to reduce the number of test executions. A test sequence indicates the execution of a sequence of states / blocks of statements in the given software. It usually corresponds to a test path. Assume that, a set of test cases is available in the test case repository. The intention of software testing process is to execute these test cases against the software under test to cover all the states in it. There are two cases to deal with:

1. If some of the states in a test path are infeasible, then even an exhaustive testing cannot cover them; and testing time spent on these states is merely wasted without an indication of infeasible code to the tester.

2. In looping constructs, some of the states are repeatedly exercised in different test sequences with different test cases; and testing time is simply spent without any benefit since they will not reveal any new type of errors.

If the Software under Test (SUT) is small and simple, then a human tester can identify these infeasible test sequences and repeated subsequences by applying intelligence. But, if it is large and complex, then automation is
required in the testing process. Although, the testing tools available in the market provide automation, they lack human like intelligence in decision making process and they exhibit less consideration on the amount of time spent on testing process and satisfaction of the test adequacy criteria.

This leads the research on development of a testing approach that couples human like intelligence and automation like a tool. Hence, in the proposed approach, an intelligent agent that has these two features with autonomy, interoperability and social ability is applied to test sequence optimization. The basic reason for applying intelligent agents is that, it boils down to solve a problem that assumes intelligence by applying efficient search methods to a directed data set.

In the proposed approach, the infeasible test sequences and the repeated subsequences are identified using heuristic guided intelligent searching performed by Intelligent Search Agent (ISA); and only the feasible and non-redundant test sequences are stored in the optimal test sequence repository with an indication of infeasible code to the tester. Now, the test cases can be exercised only on the optimal test sequences which lead to reduction in testing time. If the infeasible test sequences are corrected by the developer, then the optimal test sequences repository will be updated automatically by the agent.

4.2 REPRESENTATION OF THE SEARCH SPACE – FORMAL PROBLEM DEFINITION

In the proposed approach, the given Software under Test (SUT) is represented as a search space. Usually, search spaces are represented by graphs. This graphical representation of the Software under Test (SUT) helps to solve the testing problem which requires intelligent searching. Hence, graphs are used to visualize the given Software under Test (SUT).
A program or a software code consists of both sequential statements and control statements. During execution, the next statement in the program follows a sequential statement whereas the statement following a control statement is selected by a conditional jump (Aditya P. Mathur 2008). The statement ordering during the execution of the program is described by an Execution Sequence Graph (ESG) similar to Control Flow Graph (CFG) derived from the State chart of the SUT.

Several research works on software testing have proposed graph-theoretic models of several types of information flows and have used them to define testing techniques (McMinn et al. 2003, Li and Lam 2004). Hence, in the proposed approach, the SUT is converted into a state chart which usually provides an abstract view of the software. This state chart is converted into a graph based network model, in which each state represents a block of executable statements in the software and is shown in Figures 4.1 and 4.2.

Intelligent searching is done on the converted graph using heuristics guided search process. In this thesis, the terms nodes and states are used interchangeably.

```
While <cond> do......................state1
{
  if <cond> ......................state 2
    [...} ......................state 3
  else {...} ......................state 4
  switch (choice)............... state 5
  {
    case 1: continue; ............state 6
    case 2: .... ..................state 7
    if <cond> ................... state 8
      [...] ......................state 9
    else {...} ..................state 10
    case 3: .... ..................state 11
      Break;
  }
  ...
  ..............................state 12
```  

Figure 4.1 Sample skeleton code
4.2.1 Problem Environment

The Software under Test (SUT) is given as input. Let ‘n’ be the cyclomatic complexity value that indicates there are ‘n’ independent test paths in the Software under Test (SUT). Each test path $TP_j$ is the basis for generating test case $TC_j$, where $j=1$ to $n$. Let $TSeq_j$ is a test sequence associated with each test path.

4.2.2 Assumption

Software under Test (SUT) has no syntax errors (Compilation Errors). Software coding standards are followed when developing the SUT.

4.2.3 Objective Criterion

The objective is to generate feasible and non-redundant test sequences by identifying infeasible test sequences and redundant sub-
sequences in the generated test sequences using Intelligent Search Agent that applies intelligent graph searching through the SUT.

### 4.2.4 Formal Optimization Model

The general multi-objective optimization problem for test sequence optimization is given formally as shown below:

Min.

- Test-cost (SUT) \hspace{1cm} (4.1)
- Test-time (SUT) \hspace{1cm} (4.2)
- Length (Test-Sequence) \hspace{1cm} (4.3)

Max.

- State-Coverage (SUT) \hspace{1cm} (4.4)
- Branch-Coverage (SUT) \hspace{1cm} (4.5)
- Statement-Coverage (SUT) \hspace{1cm} (4.6)

The objective functions (4.1) and (4.2) are focusing on minimizing the cost and time associated with the testing process. The objective function (4.3) indicates the optimality requirement criterion in terms of length of the test sequences, stating that the length of the test sequences generated for exercising the SUT should be less. The next set of objective functions (4.4), (4.5) and (4.6) enforces the quality of the software in terms of maximizing coverage based test adequacy criteria. The analysis of the different coverage measures is conducted using the survey of Horgan et al (1994).

### 4.3 RELATED WORK

Lori A.Clarke (1976) presented a system which generates test data and symbolically executes program paths to aid the selection of test data. Linear programming techniques were used in her paper, to generate input data
for paths whose constraints are linear. Symbolic execution is used to generate
the constraints representing the selected path and an inequality solver is used
to solve the linear constraints; if the solver finds a constraint inconsistent with
the previous ones, the path is marked to be non-executable.

Hedley and Hennell (1985) discussed the main causes of infeasible
paths in programs and presented a classification for the causes. Frankl and
Weyuker (1988) discussed the concepts related to un-executable paths and
their consequences on the application of data flow testing criteria; they
presented a heuristics using data flow analysis and symbolic execution
techniques to determine infeasible data flow associations.

Malevris et al (1990) used the number of predicates in a path to
predict infeasibility. They concluded that the greater the number of predicates
in a path, the greater the probability of it being infeasible.

Korel (1990) reduced the problem of finding the input data to a
sequence of sub goals; each one is associated with a path predicate.
According to this approach, each sub-goal is solved by using direct search to
minimize the value of the error’s functions associated with the predicates.

Goldberg et al (1994) used symbolic evaluation to construct a
formula that is solvable if and only if, there are input values which drive the
execution down the path. A theorem-prover is invoked to test the feasibility
of the formula.

Roger and Korel (1996) extended the technique proposed by Korel
(1990) by including data dependence information for path selection in the test
data generation process (the “chaining approach”). If an undesirable execution
flow is observed in any branch and the search process cannot find an input
value to change this execution flow, data flow analysis is applied to identify
the nodes that must be executed prior to reaching the branch to increase the chance of altering the flow execution.

Forgacs and Bertolino (1997) used control flow information and data dependencies to select feasible paths in reaching a given point in the program. The approach proposed by them minimizes the number of predicates which are essential to reach the point by reducing the number of predicates examined during the input data generation.

Bodik et al (1997) used a static branch correlation analysis to define a technique for the identification of infeasible program subpaths and infeasible def-use pairs.

Tracey et al (1998) presented a test-case data generation framework based on optimization technique called Simulated Annealing (SA) and illustrated the application of this framework to test specification failures and exception conditions.

Gupta et al (1998) suggested a method where, if the intended path is not executed, the input is iteratively refined. Two representations are computed for each predicate: the slice - a set of commands that influence the predicate along the intended path; and the predicate’s linear arithmetic representation as a function of input variables. The two representations are used to obtain a desired input by iteratively refining the initial input value. This technique is improved (1999) through a “Unified Numerical Approach” where the choice of input values is done using Least Square Errors techniques. Numerical analysis is applied to identify the infeasible paths when all the predicates in the paths are linear with respect to the input variables.

Paulo et al (2000) proposed a tool and techniques for test data generation and identification of a path's likely unfeasibility in structural
software testing. Their tool was based on the Dynamic Technique and search using Genetic Algorithms. Their work introduced a new fitness function that combines control and data flow dynamic information to improve the process of search for test data. The unfeasibility issue was addressed by monitoring the Genetic Algorithm's search progress.

Wimmel et al (2000) converted the specifications into prepositional logic and then fed it into a solver. Then test sequences are generated by I/O traces represented in the form of a Message Sequence Charts.

Willem et al (2004) developed a Java Path Finder tool for generating test paths by taking the execution traces of the given system.


Turgut et al (2006) proposed a pair wise sequence comparison operation used in bio-informatics for fitness function evaluation in meta-heuristic based approach for structured software testing.

McMinn et al (2006) introduced the species per path approach, for search based software test data generation. In their approach, they first transformed the SUT into a version having multiple paths to the target. Then they identified test data by searching the individual path by dedicated species working in parallel. They analyzed the impact of feasible paths which gave rise to landscapes than the original landscape. The verification and validation of their study indicated the possibility of generating test data for targets using standard evolutionary method which is very troublesome.
José et al (2008) proposed a strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software. In their paper, they provided a methodology for generating test data for the structural unit-testing of object-oriented Java programs. In their methodology, they identified feasible and unfeasible test cases based on the different paths in the SUT since, they define elaborate state scenarios.

Moataz et al (2008) proposed GA based automated test data generation approach for white box testing. Their paper presented an approach to cover multiple target paths. They have designed a GA based test data generator that is able to synthesize multiple test data to cover multiple target paths.

Yan et al (2008) proposed an approach, in which a Flattened Control Flow Graph and a Flattened Control Dependence Graph for the switch-case construct are first presented, and a unified fitness calculation approach based on Alternative Critical Branches is proposed for the switch-case and other constructs. The concept of Alternative Critical Branches is extended from the single critical branch. Experiments on several large-scale open source programs demonstrate that this approach contributes a much better guidance to evolutionary search.

Minh and Hee (2008) proposed a heuristics-based approach to infeasible path detection for dynamic test data generation. Their approach is based on the observation that many infeasible program paths exhibit some common properties. Through realizing these properties in execution traces collected during the test data generation process, infeasible paths can be detected early with high accuracy.
4.4 PROPOSED TEST SEQUENCE OPTIMIZATION FRAMEWORK

4.4.1 Need for Intelligent Agent based Approach in Software Testing

- Existing approaches are human centric.
- Manual testing is error prone and time consuming.
- Automation of the testing process lack optimization criteria and fulfillment of test adequacy criteria.
- Automated tools lack human like intelligence in decision making process.
- Automated software testing tools and techniques usually suffer from a lack of generic applicability and scalability.
- Most of the tools are language or domain dependent and human intervention is inevitable.
- The degree of automation remains at the automated test script level.
- No care / less care on optimized number of test cases.

The related work indicated the application of linear programming techniques, data and control flow analysis, data dependence analysis, static branch correlation analysis, simulated annealing and unified numerical approach to identify infeasible statements in the Software under Test (SUT). But each of these techniques is only partially automated and they did not provide a way to find out the feasible test paths thereby reducing the number of test runs; also, they require the guidance of a human tester during their identification process. If the test sequence selection is guided by a human tester, then by means of human intelligence it will be easier to find out the infeasible statements in the Software under Test (SUT). The problem of cost
and time involved in manual software testing process implies the need for a human like intelligent approach but in an automated way.

Hence, an Intelligent Agent that couples automation and human like intelligence is applied in this proposed approach. The agent learns the Software under Test (SUT) and updates its knowledge source based on the previous decisions and current constraint satisfaction. Hence the developed agent simulates the behavior of a human tester and works in an automated way as a tool. Ultimately, a testing tool is developed that has high level of performance and selects the limited set of efficient test sequences from the SUT.

(a) **Intelligent agents – an introduction**

Intelligent Agents are pieces of software that are designed to make computing and other tasks easier by assisting and acting on behalf of the user. The user can interact with the agent through the user interface while the agent can sense and act according to the condition of the external environment. The agent performs its tasks by taking information from the environment in which it is working (Russel and Norvig 1995).

Agents can be constructed with a wide range of capabilities. In agent-based approach a complex processing function can be broken into several smaller and simpler ones. An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect, what it senses in the future (Russel and Norvig 1995).

As in the earlier work on the application of Intelligent Agents in Software Testing (Dhavachelvan and Uma 2003), agents can be implemented with sophisticated intellectual capabilities such as the ability to reason, learn,
or plan. In addition, intelligent software agents can utilize extensive amounts of knowledge about their problem domain.

Among many type of Intelligent Agents, the proposed approach has focused on the design of Task-Specific Software Agents to perform software testing task. Since testing activity consists of a number of uncertainties, the test agent which is proposed in this research work checks up all the conditions in the SUT in generating optimal test sequences.

4.4.2 Proposed Intelligent Search Agent (ISA) Framework

In the proposed ISA framework shown in Fig. 4.3, the SUT is converted into a state chart and the agent selected a very limited set of test sequences to be executed from the extreme large number (usually infinitely many) of potential ones. The proposed approach demonstrates a way to generate few effective test sequences that are guaranteed to take less time to execute and also satisfies both state and branch coverage based test adequacy criteria using Intelligent Agents.

![Figure 4.3 Optimized Test Sequence Generation Framework](image_url)
The proposed Intelligent Agent namely, the Intelligent Search Agent (ISA) generates the list of test sequences from the given SUT by getting the set of world states in the SUT and the transitions between them as inputs. These set of world states and the interaction between them are stored in the agent’s knowledge source as perception sequences. Then the action of either to include or exclude a state to or from a test sequence is done based on the two novel algorithms namely infeasible test sequence identification and repeated subsequence detection.

The proposed agent is a Search Agent, which has been developed using Java. It takes the SUT as input and produces optimal few test sequences depending upon the sum of the heuristic value associated with each node and the cost associated with each edge. The objective is to cover all the feasible nodes in the SUT without the repetition of subsequences (sequence of states/nodes).

The agent learns the SUT by means of Blackboard (BB) based learning. The decision making process of the agent is guided by a novel utility function called as ‘Happiness Value Calculation Function’. The agent performs intelligent searching through the SUT by means of heuristics guided search algorithms namely Infeasible Test Sequence Identification Algorithm and Redundant Subsequence Detection Algorithm to aid the decision making process.

(a) **Agent descriptor**

The agent descriptions are provided by means of Prometheus Methodology (John et al 2005).

- Name: Intelligent Search Agent (ISA)
- Description: Searching the states and finding the optimal test sequences by considering the heuristic values associated with each node and edge in the graph.
• Lifetime: Till the software became obsolete.
• Initialization: Based on Software Under Test (SUT).
• Incoming and Outgoing messages:
  o get_state(SUT → Agent)
  o Utility_calc(agent) (or) happiness_val_calc(Agent)
  o Check_state(Agent → Agent)
  o Opt_sel_states(agent → repository)

(b) Learning approach of ISA

For an intelligent agent to act according to the problem domain, a learning approach should be devised. An algorithm for learning, when the goal state is well defined involves two important design issues. One is relevance to the goal state (i.e. Measure of the adjacency, closeness) and second is relevance to the goodness criterion (i.e. Minimum distance or time). The proposed agent learns the SUT by means of Blackboard (BB) based learning approach.

As per Nii (1986), the purpose of the Blackboard (BB) is to hold computational and solution state data needed for the knowledge sources. The Blackboard consists of objects from the solution space. The objects on the Blackboard are hierarchically organized into levels of analysis. The objects and their properties define the vocabulary of the solution space (Grady Booch 2003).

As shown in Figure 4.4, the Blackboard framework consists of three elements:

1. Blackboard (BB)
2. Multiple Knowledge Sources (KS)
3. Controller that mediates among these knowledge sources (C)
Figure 4.4 Blackboard Architecture

In Blackboard architecture, a central Blackboard (BB) data structure holds the entire state of a solution. The BB has computational and solution state data needed by the knowledge sources. It consists of objects from the solution space. The objects on the black board are hierarchically organized into levels of analysis. The objects and their properties define the vocabulary of the solution space.

Domain and world knowledge are represented in separate independent Knowledge Sources (KS), which hold computations that respond to changes in BB, and with direct access to it; KS interact through the BB to yield solutions. The knowledge source takes a set of current information on the blackboard and updates it as encoded on its specialized knowledge source. The knowledge sources are domain specific. They have procedures, logical rules and assertion conditions for optimal test sequence generation. They embody two elements namely Preconditions and Actions. Preconditions specify the current state of the blackboard in which the knowledge source shows its interest. Triggering a precondition causes the knowledge source to
focus its attention on this part of the blackboard and then take action by processing its rules or procedural knowledge. The knowledge sources are updated through reasoning by means of forward chaining and verifying by means of backward chaining.

A Control (C) monitors the changes in BB and determines the next action to be executed, plans evaluations and activations according to a strategy that decides which KS is to be used for the next update in the BB. The controller can do the action of using the appropriate knowledge source when it has something to contribute in the processing.

The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure. Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution. These specialized programs are independent of each other. They do not call each other, nor is there a predetermined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of progress. A central control component evaluates the current state of processing and coordinates the specialized programs.

The basic reason for applying BB based learning in the proposed agent is that, the solution generation involves incremental learning process based on the previous state of the solution space and the current state of the problem space. The basic idea behind the solution generation of the proposed approach is that, the solution space has several levels/layers. Initially, the solution space is empty. Then, based on the SUT, the lowest level of the solution space is formed. By means of incremental learning, the next level in the solution space is formed using the previous level of the solution space along with the current state of the system determined by constraint satisfaction. This process is continued till the solution space contains the
expected solution. Since, this process resembles the BB based learning, the proposed agent learns the SUT by means of this learning approach.

As shown in Figure 4.5, the objects such as Blackboard, Knowledge Sources and Controller are created for Test Sequence Optimization problem. Then, identification of domain specific classes and objects is done which helps in specialization of these higher level abstractions.

![Figure 4.5 Blackboard architecture of ISA](image)

Initially, the black board is empty. The knowledge sources (multiple knowledge sources namely KS1, KS2,...KSn) contain the information about the SUT like, the executable statements, states, state transitions, edge costs, node heuristics, happiness values, and similarity measures.

By using these knowledge sources, the lowest level of the BB is formed with all the executable statements in the SUT to form states. The next level in the black board is formed by combining the information from the lowest level and the knowledge sources at that level. Now, the next level in the black board is formed by grouping the block of executable statements taken from the previous level of the BB into executable states / nodes.
The knowledge sources are updated properly by the controller which monitors the activities performed in the BB. The final level in the BB is the formation of optimal test sequences. This is done by applying the infeasible test sequence identification algorithm and the repeated subsequences detection algorithm along with the current state of the progress in the BB.

The learning process is a continuous one, till the final level in the BB is formed. Once, the solution is obtained, the agent stores all the generated optimal test sequences in the optimized test sequence repository. After that, the agent will not go into an inactive state, rather, even during maintenance, if any modifications were done in the software, the agent automatically senses the changes and by using the Controller the corresponding KS’ are modified to reflect the modifications.

Pruning is done by means of a user interface, which is needed only initially. Once the agent learnt the system, it will perform all the activities by its agenda and reduces the amount of user intervention.

(c) Decision-making by ISA

An agent’s preferences between world states is captured by means of an effective decision making process. The decision making process is equipped with a utility function which assigns a single number to express the desirability of a state.

As per the function used in utility value calculation, a nondeterministic action ‘A’ will have possible outcome states Result\_i (A), where the index ‘i’ ranges over the different outcomes. Prior to the execution of ‘A’, the agent calculates the Maximum Expected Utility (MEU) based on the expected utility values associated with all possible outcome states.
To calculate the utility values, the system should be represented as a complete casual model and in the proposed approach, the SUT has been represented as a network model as shown in Figure 4.2. The agent in the proposed approach takes the action of either including or excluding a state to or from a test sequence, by evaluating all possible states emerging from the current state.

The selection of the next best state to transit among the set of possible outcome states, is done based on the utility function called as ‘Happiness Value Calculation Function’. This function calculates the expected utility in terms of ‘Happiness Value’ which in turn is calculated based on the current constraint satisfaction \( h(C_k(n)) \) of all the explored states and the time it takes to transit from the current state to each of the explored frontier nodes \( W_i(e) \).

Before the selection is made, the agent finds the cumulative happiness value of all the nodes along the path till the currently explored nodes have been reached. Then based on this value, the agent chooses the state with Maximum Expected Utility value (MEU) or highest cumulative happiness value. The Maximum Expected Utility value (MEU) is calculated using the formula given in equation (4.7).

\[
\text{MEU} (\text{Result}_i(A)) = \sum_i (H(\sum_k h(C_k(n)), W_i(e)))
\]

(d) **Searching in ISA**

Usually, simple graph searching algorithms perform blind search because they cannot provide information on where the proper solution lies in the solution space. If the solution generation is based on the minimum edge
weight based graph searching technique, then the solution may lead to a wrong path and will never wind up at a goal. Hence, a more intelligent search needs to be constructed using heuristics which are general searching guidelines that guides the solution generation process.

This has indicated that, the use of Artificial Intelligence (AI) in searching the given search space is necessary to have a complete search and to gain insight into the search space in problems like Software Testing. Intelligent search methods can be mixed up with traditional methods by combining them with heuristics and sets of problem-specific rules. Application of proven Artificial Intelligence (AI) based searching methods enables us to add power, elegance and sophistication to the testing tool development.

In the proposed approach, the graph based searching is guided by two important heuristics for infeasible test sequence identification and repeated subsequence detection.

The domain of intelligent search algorithms for general purpose application can be classified into two primary categories: Path Finding and Constraint Satisfaction (Russel and Norvig 1995). These divisions arise based on the type of problem that is being solved.

- Path finding problems are focused on finding the path from some initial state to some final state. When solving this type of problem, the start and end points of the search might be known in advance. Finding an efficient, and possibly optimal, path between the start and end state is the goal.

- Constraint satisfaction problems (CSP) are concerned with taking some initial state and converting it to a final state that
conforms to some predefined constraints. This is the most commonly used problem category in intelligent search algorithms.

The proposed search algorithms for the ISA are belonging to these stated categories namely path finding and constraint satisfaction. Based on constraint satisfaction, the search along the SUT is guided using the expected utility value / happiness value associated with each node. In the proposed approach, the effective search algorithm is achieved by means of this expected utility value /happiness value which is formulated based on the identification of difference between happy and unhappy states. The distinction between happy and unhappy states is identified by means of constraint satisfaction.

4.5 INTERNAL ARCHITECTURE OF INTELLIGENT SEARCH AGENT (ISA)

In the internal architecture as in Figure 4.6, the ISA has a sensor, which is not a hardware sensor rather it is a software code that receives the nodes, edges, the cost associated with each edge and node heuristics along with the SUT as a directed graph. The state module consists of all the states in the system.

Starting from the initial node of the graph, the ISA explores all the frontier nodes from it, using the perception sequences available in the knowledge source. If there is a single frontier node and its constraint is satisfied, then it is assigned with a maximum happiness value. This utility value indicates that the explored node is a feasible node and so, the ISA chooses that node and append it with the current test sequence. If there are multiple frontier nodes, then the ISA decides which frontier to choose by
applying the decision making process based on the Maximum Expected Utility value (MEU) / cumulative happiness value.

The decision making process involves the selection or rejection of a particular node based on its associated happiness value. If the constraint associated with the current node is just a contradictory to the previously satisfied constraints, then it cannot be covered by any of the test data. Hence, the node is marked as infeasible, and its happiness value is set as zero which makes the node to be rejected and put up in the log file. If there is more than one node with same maximum happiness value, then the selection of next best node is done purely arbitrarily. Once the node is selected, it is appended with the feasible test sequence.

**Figure 4.6 Internal Architecture of Intelligent Search Agent**

The decision making process involves the selection or rejection of a particular node based on its associated happiness value. If the constraint associated with the current node is just a contradictory to the previously satisfied constraints, then it cannot be covered by any of the test data. Hence, the node is marked as infeasible, and its happiness value is set as zero which makes the node to be rejected and put up in the log file. If there is more than one node with same maximum happiness value, then the selection of next best node is done purely arbitrarily. Once the node is selected, it is appended with the feasible test sequence.
If the search process has revealed that the SUT has still more nodes unexplored, then the process of node exploration and selection is repeated. Once all the states in the current test sequence are completed and the agent has reached the final node in the SUT, then the test sequence is stored in the test sequence repository. The process is continued till all the nodes have been covered at least once except the infeasible nodes or till a system state of no more frontier nodes exists is reached. Then, these newly formulated test sequences are provided as input to the ISA to detect the repeated subsequences in it. Finally, the optimized test sequence repository contains only feasible and non-redundant test sequences in it.

To achieve this, two sub levels of optimizations are performed for test sequence optimization. They are,

- Infeasible test sequence identification based optimization
- Repeated subsequence detection based optimization

Here the need is to ensure that every state has been visited at least once. By using the infeasible test sequence identification algorithm, only the feasible states are included in the test sequence repository. Since, the exploration of the SUT is continued till no more frontier nodes are to be explored, the state coverage based test adequacy criteria is achieved by ensuring that all the reachable states have been covered at least once. Branch coverage criterion is achieved in the proposed approach, by evaluating the constraints on both true and false cases through branch count measure. The similarity measure is used to find the repetition of subsequences, thus avoiding the time spent on executing the same path again. Hence all the branches and states except the infeasible states are covered using the proposed approach.
4.6 ALGORITHMS FOR TEST SEQUENCE OPTIMIZATION

The algorithms proposed in this framework are:

1. Infeasible Test Sequence Identification Algorithm
2. Repeated Subsequence Detection Algorithm

4.6.1 Heuristics used in Test Sequence Optimization

**Happiness Value** – It is associated with each constraint in the SUT. The utility function / fitness function provides the happiness value for each predicate. It is used for infeasible test sequence identification.

**Similarity measure** – It is used to find the repetition of subsequences, thus avoiding the time spent on executing the same path again.

4.6.2 Infeasible Test Sequence Identification Algorithm

(a) Infeasible paths – an introduction

Consider the following code:

```c
If (a<2)
{ if (a>3)
   { x=x*1; }
}
```

**Figure 4.6.2 (a) Unreachable code 1**
In the code shown in Figure 4.6.2 (a), if ‘a’ is less than 2, then it cannot be greater than 3, and so, $x=x*1$ cannot be reached. Similarly, in Figure 4.6.2 (b), ‘i’ is never less than zero, and so the statement $\text{total} = \text{total} + \text{value}[\text{i}]$ can never be reached. These statements make the path to be infeasible. And, these sorts of infeasible paths will not allow an execution path to be completed.

(b) Functionality of infeasible test sequence identification algorithm

In order to overcome the problem of infeasible paths, execution is not initially forced to exercise the whole test path as in the case of simple testing tools or in the case of traditional searching techniques, rather progressively one decision at a time is done. Once a set of test data has been found that executes the path up to a certain point, next decision is appended at the end of the current path and optimization process is restarted. Once a state is selected, the next state to traverse is determined by the agent automatically by means of constraint value and continue-execution-flag.

At each stage during the search, the current search point is checked to see that it satisfies all types of constraints. Only the search points or nodes which satisfy all the constraints are eligible for selection, otherwise the execution of the problem will result in run-time errors. By means of this, the infeasible paths are identified.
For example, consider the generation of test data that forms the test sequence $TSEQ=(S_1,S_2,S_3,\ldots,S_n)$, Where each ‘$S_i$’ represents a state in the state chart. Initially, optimization is performed to find a set of test data ‘$x_1$’ that executes state ‘$S_1$’. Keeping ‘$S_1$’ as the initial search point, a search is then performed to find a set of test data ‘$x_2$’ which executes $S_1S_2$ and the process continues until a set of test data ‘$x_n$’ is found to execute the complete test sequence ‘$TSEQ$’.

If at any stage, a set of test data cannot be found to execute a current path, then it is clear that, no sets of test data will be found that will execute test sequence ‘$TSEQ$’ and so, ‘$TSEQ$’ becomes an infeasible test sequence; consequently the search is terminated on that sequence. Therefore, this method of step-by-step constraint satisfaction helps in the early determination of states which are not feasible.

The agent finds the infeasible test sequences by using the constraint value of each predicate in a conditional expression. The predicate is converted in the form shown in Figure 4.6.2(c), for all the predicates of the form $<$, $<=$, $>$, $>=$, $==$ and $!=$. The logical operators such as $\&\&$ and $||$ will use the results of either of the above forms consistently.

\[
\begin{align*}
\text{If } (x==y) & \Rightarrow (x-y) \quad // \text{Constraint value of } (x==y) \\
\text{If } (x>y) & \Rightarrow (x-y) \quad // \text{Constraint value of } (x>y)
\end{align*}
\]

**Figure 4.6.2 (c) Conversion of Predicates**

The agent assigns a happiness value to the search point / node in which the constraint is satisfied. If the accumulated happiness value is low, then it is an indication to the tester that, some of the conditions are invalid which in turn leads to an infeasible code / dead code.
The agent selects only the set of test sequences that have the highest happiness values among the set of test sequences and stores them as the optimal test sequences. Then the Continue Execution flag will be reset to false if all the states have been visited at least once with the infeasible code are stored in the log file. If the infeasible code is corrected, the agent then adds it to the optimal test sequence set, if it is needed for the coverage of the code.

(c) **Mathematical model for infeasible test sequence identification**

The objective function is to maximize the happiness value associated with each node in the test sequence construction.

Max. 
\[
f'(TSeq_j) = \sum_i (H(\sum_k h(C_k(X_{ij})), W_i(e)))
\]  
\[(4.8)\]

where 
\(i=1\) to \(m;\) \(j=1\) to \(n;\) \(k=1\) to \(p\) (\(m\)-number of nodes, \(n\)-cyclomatic complexity value, \(p\)-number of predicates at constraint \(k\))

\(TSeq_j\) – Test sequence \(j;\) \(X_{ij}\)- state/node \(i\) in the test sequence \(j;\)
\(e\)- edge in SUT;
\(C_k = (C_1, C_2…,C_p)\) are the constraint values of the predicates at node \(X_{ij}\);
\(H(\sum_k h(C_k(X_{ij})), W_i(e))\) – is the happiness function / Utility function;
\(h(C_k(X_{ij})))\) - happiness value of the predicate \(C_k\) at node \(X_{ij}\);
\(W_i(e)\) – Function to calculate the transition time from state \(x\) to state \(y\)

Sub. to 
\[
h(C_k(X_{ij})) = \begin{cases} 
0 & \text{if constraint } C_k \text{ on } X_{ij} \text{ is not satisfied} \\
\text{Max} & \text{Otherwise}
\end{cases}
\]  
\[(4.9)\]
\[ W_i(e) = \begin{cases} \text{Max} & \text{for node with minimum transition time} \\ 0 & \text{for others} \end{cases} \quad (4.10) \]

\[ \text{Cover}(x) = \begin{cases} \text{false} & \text{if x is uncovered} \\ \text{true} & \text{otherwise} \end{cases} \quad (4.11) \]

\[ \text{Status}(x) = \begin{cases} 1 & \text{if } H(\sum_k h(C_k(X_{ij})), W_i(e)) = \text{MAX} \\ 0 & \text{if Cover}(x) = \text{false (Initialization)} \\ -1 & \text{if } H(\sum_k h(C_k(X_{ij})), W_i(e)) = 0 \end{cases} \quad (4.12) \]

\[ T_{seq_j} = \begin{cases} T_{seq_j} + x & \text{if Status}(x) = 1 \\ T_{seq_j} & \text{otherwise} \end{cases} \quad (4.13) \]

\[ \text{Continue-execute-flag (SUT)} = \begin{cases} 1 & \text{if Cover}(x) = \text{false} \\ 0 & \text{otherwise} \end{cases} \quad (4.14) \]

The objective function (4.8) is to maximize the fitness function \( H(\sum_k h(C_k(X_{ij})), W_i(e)) \). Constraint (4.9) indicates the happiness value associated with node \( X_{ij} \) based on constraint \( C_k \). Calculation of weight value \( W_i(e) \) in terms of time taken to execute the current block is shown in constraint (4.10). The continuation of test sequences generation till no more frontier nodes to explore is given in constraint (4.11). The value of status flag associated with each node that guides the agent about the presence / absence of infeasible code is given by constraint (4.12). The construction of the test sequence based on the Status flag associated with each node is indicated in constraint (4.13). The final constraint (4.14) indicates the status of continuation of exploration and is based on the result of constraint (4.11).
Based on the constraint value \( (c_i(n)) \), the happiness value is calculated as follows:

(i) For \( a > b \), \( c_i(n) = (b-a) \) and if \( c_i(n) < 0 \), \( h(c_i(n)) = \text{MAX} \), otherwise \( h(c_i(n)) = 0 \)  \( (4.15) \)

(ii) For \( a \geq b \), \( c_i(n) = (b-a) \) and if \( c_i(n) \leq 0 \), \( h(c_i(n)) = \text{MAX} \), otherwise \( h(c_i(n)) = 0 \)  \( (4.16) \)

(iii) For \( a < b \), \( c_i(n) = (b-a) \) and if \( c_i(n) > 0 \), \( h(c_i(n)) = \text{MAX} \), otherwise \( h(c_i(n)) = 0 \)  \( (4.17) \)

(iv) For \( a \leq b \), \( c_i(n) = (b-a) \) and if \( c_i(n) \geq 0 \), \( h(c_i(n)) = \text{MAX} \), otherwise \( h(c_i(n)) = 0 \)  \( (4.18) \)

(v) For \( a = b \), \( c_i(n) = (b-a) \) and if \( c_i(n) = 0 \), \( h(c_i(n)) = \text{MAX} \), otherwise \( h(c_i(n)) = 0 \)  \( (4.19) \)

(vi) For \( a \neq b \), \( c_i(n) = (b-a) \) and if \( c_i(n) \neq 0 \), \( h(c_i(n)) = \text{MAX} \), otherwise \( h(c_i(n)) = 0 \)  \( (4.20) \)

(vii) For \( a \text{ OR } b \), \( h(c_i(n)) = h(c_i(a)) + h(c_i(b)) \)  \( (4.21) \)

(viii) For \( a \text{ AND } b \), \( h(c_i(n)) = \text{MIN} (h(c_i(a)), h(c_i(b))) \)  \( (4.22) \)

The happiness value associated with each state decides the next state to choose for its transition. This process is continued until a system state where no more frontier nodes to explore is reached.

**Proposed infeasible test sequence identification algorithm**

1. Convert the SUT into a UML State Chart notation.
2. Translate the State Chart into an Execution Sequence Graph (ESG). This graph is a directed graph in which each node is a
state in the State Chart and edge between the nodes is the state transition between the states.

3. Start from the initial node as the node to explore.

4. To identify the next best node to explore, calculate the edge weight \( w(e) \) and the goodness value of each node \( h(n) \).

5. The Edge Weight \( (w_i(e)) \) is calculated as, \( w_i(e) = \text{Function to calculate the time taken to traverse from the current node to the next node} \)

6. The Goodness criterion (node heuristic) for each node is calculated as, \( h(n) = \text{Happiness Value associated with each constraint } C_i \).

7. Cumulative Happiness value \( (h_v(n)) \) is calculated as,
   \[
   h_v(n) = \sum (h(c_i(n), w_i(e)).
   \]  

8. Based on this \( h_v \), the next state is selected among the set of world states and test data associated with that state is appended with the current test sequence.

9. If \( h(n_i) > h(n_j) \) then \( TSeq_i \leftarrow TSeq_i + n_i \)

10. Else \( TSeq_i \leftarrow TSeq_i + n_j \)

11. If \( h_v(TSeq_{current}) < h_v(TSeq_{rejected}) \) then
    \[
    TSeq_{current} \leftarrow TSeq_{rejected}.
    \]

12. Else \( TSeq_{current} \leftarrow TSeq_{current} \)

13. If the final node is reached, the test sequence is stored in the optimized test sequence repository.

14. Consider the current node as the node to explore. Repeat steps 3 to 11 until we arrived at a state where we don’t have any more frontier states to explore.
4.6.3 Repeated Subsequence Detection Algorithm

(a) Repeated subsequences – an introduction

![Sample Graph with loops](image)

Figure 4.7 Sample Graph with loops

For the graph shown in Figure 4.7, the possible test sequences are derived and are listed as below:

- TSeq1: A→C→F
- TSeq2: A→B→E→F
- TSeq3: A→D→C→F
- TSeq4: A→B→E→A→C→F
- TSeq5: A→B→E→A→B→E→F
- TSeq6: A→B→E→B→E→F
- TSeq7: A→B→E→A→D→C→F
- TSeq8: A→B→E→A→B→E→B→E→F
- TSeq9: A→B→E→B→E→A→D→C→F
  
Etc…

Among the generated test sequences, some of the test sequences are simply the combination of other test sequences thus leads to repeated
subsequences. These test sequences should be identified and marked so that, they will not be included in the optimal test sequences set.

(b) **Functionality of repeated subsequence detection algorithm**

For identifying the repeated subsequences in a test sequence, the agent breaks the given test sequence into a number of subsequences. Where, if there is a repetition for ensuring a branch coverage, then the said sequence is broken in such a way that, a sub sequence before the branch, a subsequence with a branch and a subsequence after the branch is obtained for all the set of available paths. The repeated subsequences are removed and marked as visited since they won’t produce a new coverage factor.

To detect the repeated subsequences, the proposed approach applies Hamming Distance as a measure to find the similarity value. The Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different. It has been applied in network security based problems. Its functionality has been extended in the proposed approach to detect the positions in which two test sequences exactly differing from each other and matching each other.

In the proposed approach, the repeated subsequences are identified by finding the similarity measure based on Hamming distance between the generated test sequences. If two test sequences/paths are of the same length, then the difference between them is calculated. It is the number indicating the places where the test sequences are differing from each other.

If it is zero, then the test sequences are repeating exactly and one of them should be rejected. If they are differing in one or more places, then find the subsequences in it which are simply a repetition of other test sequences.
Only the test sequences which are not yet traversed or not repeated are added to the list.

(c) **Mathematical model of repeated subsequence detection**

The optimization problem is given as,

\[
\text{Min. } f(TSeq_x, TSeq_y) = \sum \text{Similarity}(TSeq_x, TSeq_y) \tag{4.24}
\]

Where \(TSeq_x, TSeq_y\) – Test sequences \(x\) and \(y\) respectively.

Sub. to

\[
\text{Similarity}(TSeq_x, TSeq_y) = \begin{cases} 
0 & \text{if Hamming-Dist}(TSeq_x, TSeq_y) = 0 \\
> 0 & \text{otherwise} 
\end{cases} \tag{4.25}
\]

The objective function (4.24) is to minimize the similarity between two test sequences \(TSeq_x\) and \(TSeq_y\). Constraint (4.25) represents the similarity value calculation based on the Hamming-Distance between two test sequences \(TSeq_x\) and \(TSeq_y\). The Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different.

(d) **Proposed repeated subsequence detection algorithm**

For each generated test sequence a similarity measure is calculated which is acting as a fitness value to find the repeated subsequences.

- Take two test sequences \(TSeq_i\) and \(TSeq_j\).
- Represent the given sequences in terms of 1’s and 0’s based on the presence or absence of states in the test sequences.
- Break the test sequences based on the occurrence of branches.
- Calculate the Fitness Function \( f(x) = \text{similarity}(\text{TSeq}_i, \text{TSeq}_j) \), where similarity \((\text{TSeq}_i, \text{TSeq}_j)\) finds the distance between two test sequences \(\text{TSeq}_i\) and \(\text{TSeq}_j\) which extends the functionality of Hamming distance.

- For the given two test sequences \(\text{TSeq}_i\) and \(\text{TSeq}_j\), the similarity between them is calculated, based on the comparison of \(n\)-order sets of ordered and even the cascaded branches (subsequences) of each test sequence.

- Normalize the symmetric difference between the subsequences. Then the similarity measure is calculated by subtracting a value ‘1’ from the calculated value if there is a repetition of a subsequence.

- Finally, the total similarity between the two test sequences is calculated as the sum of similarities of all \(n\)-order sets. It is the measure indicating the number of places in which the two test sequences are differing with each other.

- If the similarity measure is zero, then the two test sequences are not differing in any of the places and are exactly the same and so, one of them should be rejected. Otherwise, the value of similarity measure is used to find the number of places and the locations in which the two test sequences are differing and resembling each other because of the absence or presence of repeated subsequences respectively.

### 4.7 EXPERIMENTATION AND EVALUATION

To do the experimentations, the proposed Intelligent Search Agent has been coded in Java by taking the guidelines proposed in the existing work (Paolo, 1999, Shoham, 1993) for developing intelligent systems using Java.
4.7.1 Tested Problems

To study the relevance of the proposed approach based on the evaluation of test adequacy criteria, the experimentation process is conducted with fifteen programs written in structured programming language ‘C’ and fifteen programs written in object-oriented programming language C++/Java as shown in Tables 4.1(a) and 4.1(b).

Table 4.1 (a) Tested Programs in ‘C’

<table>
<thead>
<tr>
<th>Case Study #</th>
<th>Structured Programming problems – in C</th>
<th># Methods / LOC (Lines of Code)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Case Study</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Temperature Monitoring System</td>
<td>4 / 1265</td>
</tr>
<tr>
<td>2</td>
<td>Sine Series Calculation</td>
<td>3 / 75</td>
</tr>
<tr>
<td>3</td>
<td>Factorial</td>
<td>3 / 15</td>
</tr>
<tr>
<td>4</td>
<td>Function Point Calculation</td>
<td>4 / 120</td>
</tr>
<tr>
<td>5</td>
<td>Cyclomatic Complexity Calculation</td>
<td>4 / 275</td>
</tr>
<tr>
<td>6</td>
<td>Brick Game</td>
<td>11 / 1075</td>
</tr>
<tr>
<td>7</td>
<td>Binary Search</td>
<td>5/243</td>
</tr>
<tr>
<td>8</td>
<td>Singly Linked List for Country and Capital Matching</td>
<td>12/1429</td>
</tr>
<tr>
<td>9</td>
<td>Find Biggest</td>
<td>6/128</td>
</tr>
<tr>
<td>10</td>
<td>Square root</td>
<td>4/52</td>
</tr>
<tr>
<td>11</td>
<td>Filecopy</td>
<td>5/472</td>
</tr>
<tr>
<td>12</td>
<td>Sum of odd and even numbers</td>
<td>4/69</td>
</tr>
<tr>
<td>13</td>
<td>Fibonacci Series</td>
<td>5/117</td>
</tr>
<tr>
<td>14</td>
<td>Count</td>
<td>14/1504</td>
</tr>
<tr>
<td>15</td>
<td>Device Driver</td>
<td>20 / 1824</td>
</tr>
</tbody>
</table>
The test adequacy criteria are validated on these systems. For demonstration purpose, two of the case studies were discussed in this thesis. The first one, “Temperature Monitoring System” is written in structured programming language ‘C’ and the second system “Coffee/COCOA/Money Lending Machine” is written in Object Oriented programming language ‘Java’.
4.7.2 Case Study 1 – Performance Evaluation of ISA

Here as the first case study, a “Temperature monitoring system” is taken and the optimal test sequences are identified for it. The proposed agent ISA converted the given SUT into the state chart and its corresponding graphical notation.

Step 1: Construction of State Chart

The given SUT is analyzed and a corresponding state chart is built and was checked against the given scenarios. The final one looks as shown in Figure 4.8.

Figure 4.8 State chart of Temperature Monitoring System.
At the end of step 1, the given SUT has been converted into a state chart using UML notation.

**Step 2:** Conversion of State Chart into a Directed graph Notation with edges assigned with weights calculated in terms of their transition times as shown in Figure 4.9.

![Figure 4.9 Graphical representation of the State Chart with Edge Cost](image)

In the Figure 4.9, the following conventions are used: N1 – node1, N2-node2, N3-node3, N4-node4, N5-node5 and N6-node6.

**Step 3:** Generating all possible independent test sequences, with the total cost required for each test sequence.

Then by applying the infeasible test sequence detection algorithm, ISA generated all the feasible paths from the SUT as shown in Table 4.2.
## Table 4.2 The Generated Feasible Paths

<table>
<thead>
<tr>
<th>Test Sequences</th>
<th>Cost</th>
<th>Included Edges</th>
<th>Excluded Edges</th>
<th>Other Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1-&gt;node3-&gt;node5 -node6</td>
<td>8</td>
<td>None</td>
<td>None</td>
<td>node1,node3,node4,node6 = 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None</td>
<td>&lt;node5,node6&gt;</td>
<td>node1,node3,node5,node6 =12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;node3,node5&gt;</td>
<td>&lt;node5,node6&gt;</td>
<td>node1,node2,node3,node6 =14</td>
</tr>
<tr>
<td>node1-&gt;node3-&gt;node4-&gt;node6</td>
<td>9</td>
<td>None</td>
<td>&lt;node5,node6&gt;</td>
<td>node1,node3,node5,node6 =13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None</td>
<td>&lt;node4,node6&gt;</td>
<td>node1,node2,node3,node4,node6 =15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;node3,node4&gt;</td>
<td>&lt;node4,node6&gt;</td>
<td>node1,node3,node4,node5,node6 =16</td>
</tr>
<tr>
<td>node1-&gt;node2-&gt;node5-&gt;node6</td>
<td>12</td>
<td>&lt;node5,node6&gt;</td>
<td>&lt;node3,node5&gt;</td>
<td>node1,node3,node4,node5,node6 =16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;node5,node6&gt;</td>
<td>&lt;node2,node5&gt;</td>
<td>node1,node2,node3,node4,node5,node6 =16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;node2,node5&gt;</td>
<td>&lt;node1,node2&gt;</td>
<td>node1,node2,node3,node4,node5,node6 =16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;node1,node2&gt;</td>
<td>&lt;node3,node5&gt;</td>
<td>node1,node2,node3,node4,node5,node6 =16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

………..
The generated test sequences are listed below:

(a) **Feasible Test Sequences**

1. Read and Check Temp $\rightarrow$ Mode Change $\rightarrow$ User Interaction $\rightarrow$ Display Thru Panel (8)

2. Read and Check Temp $\rightarrow$ Mode Change $\rightarrow$ Perform Routine Activities $\rightarrow$ Display Thru Panel (9)

3. Read and Check Temp $\rightarrow$ Enable Cooler $\rightarrow$ User Interaction $\rightarrow$ Display Thru Panel (12)

4. Read and Check Temp $\rightarrow$ Enable Cooler $\rightarrow$ Perform Routine Activities $\rightarrow$ Display Thru Panel (13)

5. Read and Check Temp $\rightarrow$ Enable Cooler $\rightarrow$ Mode Change $\rightarrow$ User Interaction $\rightarrow$ Display Thru Panel (14)

6. Read and Check Temp $\rightarrow$ Mode Change $\rightarrow$ Perform Routine Activities $\rightarrow$ User Interaction $\rightarrow$ Display Thru Panel (15)

7. Read and Check Temp $\rightarrow$ Enable Cooler $\rightarrow$ Perform Routine Activities $\rightarrow$ User Interaction $\rightarrow$ Display Thru Panel (16)

8. Read and Check Temp $\rightarrow$ Enable Cooler $\rightarrow$ Mode Change $\rightarrow$ Perform Routine Activities $\rightarrow$ Display Thru Panel (20)

9. Read and Check Temp $\rightarrow$ Enable Cooler $\rightarrow$ Mode Change $\rightarrow$ Perform Routine Activities $\rightarrow$ User Interaction $\rightarrow$ Display Thru Panel (22)
(b) **Infeasible Test Sequences:**

10. Read and Check Temp → Enable Cooler → Mode Change → Enable Cooler → Perform Routine Activities → User Interaction → Display Thru Panel

11. Read and Check Temp → Mode Change → Enable Cooler → Perform Routine Activities → Display Thru Panel

12. Read and Check Temp → Mode Change → Enable Cooler → Perform Routine Activities → User Interaction → Display Thru Panel

13. Read and Check Temp → Mode Change → Enable Cooler → User Interaction → Display Thru Panel

14. Read and Check Temp → Enable Cooler → Mode Change → Enable Cooler → Perform Routine Activities → Display Thru Panel

15. Read and Check Temp → Enable Cooler → Mode Change → Enable Cooler → Perform Routine Activities → User Interaction → Display Thru Panel

16. Read and Check Temp → Enable Cooler → Mode Change → Enable Cooler → User Interaction → Display Thru Panel

**Step 4: Rule for inclusion and exclusion of paths**

From the above list of all test sequences, the feasible test sequences are identified by means of constraint satisfaction. By applying the infeasible test sequence identification algorithm it has been identified that; the test sequences from 10 to 16 are infeasible. The reason is, to enable cooler, the constraint ‘cooler=true’ should be satisfied. Once it is satisfied, the ‘Enable
Cooler’ state has been appended to the current test sequence. Then, it explores all the frontier states from ‘Enable Cooler’ state. The agent then calculates the happiness values of all the frontier nodes. Consider the ‘Mode Change’ state which is one of the frontier states of ‘Enable Cooler’ state. This state has been further explored in which, it has a constraint, ‘cooler=false’ for making the system to transit to the same ‘Enable Cooler’ state. Since, the constraint ‘cooler=true’ is already satisfied, it cannot be false (contradictory), to go to the same state. Since, the ‘cooler’ variable cannot be true as well as false in the same sequence, this leads to an infeasible test sequence. Hence, all the transitions going through the states ‘Enable Cooler → Mode Change→Enable Cooler’ and ‘Mode Change→Enable Cooler’, lead to infeasible test sequences and their corresponding happiness values are zero. So, the test sequences from 10 to 16 are named as infeasible and they have been removed from the test sequence repository and only the test sequences from 1 to 9 are stored in the feasible test sequence repository.

From the identified set of feasible test sequences, the redundant subsequence detection algorithm is applied to find out repeated subsequences. By applying the similarity measure heuristic, it has been identified that, the test sequences 8 and 9 have repeated subsequences which have been already covered by other test sequences. Hence, these two test sequences have been removed and the remaining test sequences from 1 to 7 are stored in the optimal test sequence repository. It has also been verified that, all the states and branches have been covered at least once except the infeasible states.

**Step 5:** The optimized test sequences which require only less cost are given in ascending order of their cost associated as below:

1. Read and Check Temp → Mode Change → User Interaction → Display Thru Panel (8)
2. Read and Check Temp → Mode Change → Perform Routine Activities → Display Thru Panel (9)

3. Read and Check Temp → Enable Cooler → User Interaction → Display Thru Panel (12)

4. Read and Check Temp → Enable Cooler → Perform Routine Activities → Display Thru Panel (13)

5. Read and Check Temp → Enable Cooler → Mode Change → User Interaction → Display Thru Panel (14)

6. Read and Check Temp → Mode Change → Perform Routine Activities → User Interaction → Display Thru Panel (15)

7. Read and Check Temp → Enable Cooler → Perform Routine Activities → User Interaction → Display Thru Panel (16)

From the set of all generated test sequences, ISA applied the redundancy detection algorithm and generated the shortest and optimal test sequences. They are listed below:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1→node3→node5→node6</td>
<td>-</td>
</tr>
<tr>
<td>node1→node3→node4→node6</td>
<td>-</td>
</tr>
<tr>
<td>node1→node2→node5→node6</td>
<td>-</td>
</tr>
<tr>
<td>node1→node2→node4→node6</td>
<td>-</td>
</tr>
<tr>
<td>node1→node2→node3→node5→node6</td>
<td>-</td>
</tr>
<tr>
<td>node1→node2→node3→node4→node6</td>
<td>-</td>
</tr>
<tr>
<td>node1→node3→node4→node5→node6</td>
<td>-</td>
</tr>
</tbody>
</table>
Result of case study 1-Test adequacy criteria Vs optimization:
The first study with the “Temperature Monitoring System” validates the test adequacy criteria of branch and state coverage with reduced length of the test sequences. The approach generates the feasible and non-redundant test sequences based on the infeasible test sequences identification and redundant subsequences detection algorithms. Among the total number of feasible test sequences, two test sequences have been excluded due to the coverage of them by other sequences.

Hence, with a fitness function based on state and branch coverage, the ISA algorithms optimized the test sequences. By using these test sequences, one can ensure that a test suite composed of seven test cases (each for one test sequence) covers 96.4% of the code (by excluding infeasible code).

4.7.3 Case Study 2- Performance Comparison of ISA and ACO

For comparison of performance between ACO and ISA, the benchmark problem taken by Li and Lam (2004) namely “Coffee/Cocoa/Money Lending machine” is demonstrated in this thesis, to get the comparison of results. Since, Li and Lam have already proved their work using the said example as the benchmark problem; the same problem has been taken to prove the proposed approach. The various steps discussed in the proposed approach are applied to derive optimal test sequences from the SUT.

(a) ISA Based Test Sequence Optimization

Step 1: Reading of the SUT

The given SUT is analyzed and a corresponding state chart is built and is shown in Figure 4.10 and is checked against the given scenarios. The final state chart for the bench-mark problem looks as below:
Figure 4.10  State chart of a COFFEE/COCOA /Money lending machine

Step 2:  Conversion of SUT into a Graph

The conversion module of the proposed tool converted the SUT into the corresponding state chart and a graph based representation as shown in Figure 4.11.
Step 3: Generation of feasible test sequences

From the SUT, by applying the infeasible test sequence identification algorithm, the test sequences with highest happiness values are obtained. The feasible test sequences are:

1. Test Seq1 = {node 0 → node1 → node5 → node7 → node11}
2. Test Seq2 = { node0 → node1 → node5 → node6 → node7 → node11}
3. Test Seq3 = { node0 → node1 → node2 → node4 → node11}
4. Test Seq4 = { node0 → node1 → node2 → node3 → node4 → node11}
5. Test Seq5 = \{ \text{node0} \rightarrow \text{node1} \rightarrow \text{node8} \rightarrow \text{node10} \rightarrow \text{node11} \}

6. Test Seq6 = \{ \text{node0} \rightarrow \text{node1} \rightarrow \text{node8} \rightarrow \text{node9} \rightarrow \text{node10} \rightarrow \text{node11} \}

**Step 4: Optimized Test Sequences Generation**

It is understood that, even though all these sequences are independent in nature, some of them have repeated subsequences, which increases the cost for executing that path. By applying the repeated subsequence detection algorithm, testing process requires only less number of test inputs that achieves the test adequacy criteria. Hence the optimal test sequences for the given system using ISA is,

1. Test Seq2 = \{ \text{node0} \rightarrow \text{node1} \rightarrow \text{node5} \rightarrow \text{node6} \rightarrow \text{node7} \rightarrow \text{node11} \}

2. Test Seq3 = \{ \text{node0} \rightarrow \text{node1} \rightarrow \text{node2} \rightarrow \text{node3} \rightarrow \text{node4} \rightarrow \text{node11} \}

3. Test Seq6 = \{ \text{node0} \rightarrow \text{node1} \rightarrow \text{node8} \rightarrow \text{node9} \rightarrow \text{node10} \rightarrow \text{node11} \}

The above test sequences are sufficient to test the entire system. In order to consider the branch coverage criterion and to identify the repeated subsequences, the sequences are split up into subsequences. After the split up, additional subsequences are added to achieve branch coverage. They are:

1. subsequence1 = \{ \text{node5} \rightarrow \text{node7} \}

2. subsequence2 = \{ \text{node2} \rightarrow \text{node4} \}

3. subsequence3 = \{ \text{node8} \rightarrow \text{node10} \}
Using the above set of test sequences, the branch coverage based test adequacy criterion is achieved.

(b) ACO Based Test Sequence Optimization

As per (Li H and Lam P, 2004), the optimal test sequences generated are:

\[ Tseq1 = \{node0 \rightarrow node1 \rightarrow node8 \rightarrow node9 \rightarrow node10 \rightarrow node8 \rightarrow node1 \rightarrow node5 \rightarrow node6 \rightarrow node7 \rightarrow node11\} \]

\[ Tseq2 = \{node0 \rightarrow node1 \rightarrow node8 \rightarrow node9 \rightarrow node10 \rightarrow node8 \rightarrow node1 \rightarrow node2 \rightarrow node3 \rightarrow node4 \rightarrow node5 \rightarrow node11\} \]

(c) Result of case study 2 - optimization using ISA Vs. ACO

The second case study “Coffee/COCOA/Money Lending Machine” is used to compare the performance of ACO and ISA. The aim of this study is to estimate the total cost and the time needed for testing using the test sequences generated by ISA and ACO. Also, it compared the length of the generated test sequences and the satisfaction of branch coverage based adequacy criterion by these two approaches.

4.8 PERFORMANCE ANALYSIS

4.8.1 Performance of Ant Colony Optimization (ACO)

When Ant Colony Optimization is applied to generate the optimal test sequences for Coffee/Cocoa/Money lending machine, the following problems are identified.
(a) **Analysis Result 1**

The algorithm proposed by (Li and Lam, 2004) initially had an assumption that the two ants are having started at an initial node, and during random selection of next node, they will go to the same node. The problem here is that, due to the random selection, one cannot expect that the technique will always produce the same next node at different instances.

(b) **Analysis Result 2**

The evaluation states that, most of the nodes are repeating between the two sequences ‘Tseq1’ and ‘Tseq2’ generated by ACO. Even though, these sequences covered all the nodes, they did not consider the branch coverage criterion. Also, the most important criterion specified by Li and Lam (2004), the test sequence with minimum length is not satisfied here.

(c) **Analysis Result 3**

When a transition cost is assigned between states using the time calculation function, each edge is assigned with an edge cost as below:

- node0 – node 1 = 17.3,
- node1–node2=23.1, node1-node5=18.9, node1-node8= 25.8,
- node2 – node3 =21.6, node2-node4 = 41.3
- node3 – node4 =23.9
- node5 – node6 = 23.1, node5 – node7 = 42.6
- node6 – node7 = 20.5
• node8 – node9 = 27.0, node8 – node10 = 48.5
• node9 – node10 = 26.1
• node4–node11=1.0, node7–node11=1.0, node10–node11=1.0

Now, the cost value associated with each edge is used to compare the performance of ISA and ACO. The total cost incurred for the two test sequences is given below:

\[
\text{Cost(Tseq1)}= 17.3+25.8+27+26.1+48.5+25.8+18.9+23.1+20.5+1.0 = 234
\]
\[
\text{Cost(Tseq2)}= 17.3+25.8+27+26.1+48.5+25.8+23.1+21.6+23.9+0+0 = 239.1
\]

Hence the **Total Cost** = 234+239.1 = 473.1

### 4.8.2 Performance of Intelligent Search Agent (ISA)

Performance of ISA is very high and it doesn’t have any of the problems listed above in ACO.

Using ISA, the optimal test sequences are:

1. **Tseq1** = \{ node0 \rightarrow node1 \rightarrow node5 \rightarrow node6 \rightarrow node7 \rightarrow node11 \}
2. **Tseq2** = \{ node0 \rightarrow node1 \rightarrow node2 \rightarrow node3 \rightarrow node4 \rightarrow node11 \}
3. **Tseq3** = \{ node0 \rightarrow node1 \rightarrow node8 \rightarrow node9 \rightarrow node10 \rightarrow node11 \}
The length of the sequences is less when compared to the ACO. As per the discussion, this will cover all the states; the cost required for these paths is calculated as below:

- \( \text{Cost(Tseq1)} = 17.3 + 18.9 + 23.1 + 20.5 + 1 = 80.8 \)
- \( \text{Cost(Tseq2)} = 17.3 + 23.1 + 21.6 + 23.9 + 1 = 86.9 \)
- \( \text{Cost(Tseq3)} = 17.3 + 25.8 + 27 + 26.1 + 1 = 97.2 \)

Hence the \textbf{Total cost} = 80.8 + 86.9 + 97.2 = 264.9

The cost needed for executing the sub paths of the branches is also included to achieve branch coverage criterion. The cost for these additional subsequences is:

- \( \text{Cost(subseq1)} = 42.6 \)
- \( \text{Cost(subseq2)} = 41.3 \)
- \( \text{Cost(subseq3)} = 48.5 \)

Even after the cost of these additional subsequences is added, the total cost using ISA is, \textbf{Total Cost} = 264.9 + 42.6 + 41.3 + 48.5 = 397.3

From the total cost calculation, it has been identified that, the total cost spent for ISA is less when compared to ACO.

The proposed approach has been evaluated using various case studies. The results of these case studies have been compared with the results of ACO based optimization. These results have been indicated in Tables 4.3, 4.4 (a), 4.4 (b) and 4.5.
### Table 4.3 Performance based comparison – ISA Vs. ACO

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Optimal Test sequences</th>
<th>Subsequences</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Major Test sequences</strong></td>
<td><strong>Cost</strong></td>
<td><strong>Test sequences</strong></td>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>ACO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSeq1={node0→node1→node8→node9→node10→node8→node1→node5→node6→node7→node11}</td>
<td>234</td>
<td>No – Identification</td>
<td>NA</td>
</tr>
<tr>
<td>TSeq2={node0→node1→node8→node9→node10→node8→node1→node2→node3→node4→node5→node11}</td>
<td>239.1</td>
<td>Subseq1 = {node5→node7}</td>
<td>42.6</td>
</tr>
<tr>
<td>ISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSeq1= { node0 → node1 → node5 → node6 → node7 →node11}</td>
<td>80.8</td>
<td>Subseq2 = {node2→node4}</td>
<td>41.3</td>
</tr>
<tr>
<td>TSeq2= { node0 → node1 → node2 → node3 → node4 →node11}</td>
<td>86.9</td>
<td>Subseq3 = {node8→node10}</td>
<td>48.5</td>
</tr>
<tr>
<td>TSeq3= { node0 → node1 → node8 →node9 → node10 →node11}</td>
<td>97.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the Table 4.3, it has been identified that, the cost associated with the test sequences generated using ISA is less when compared to ACO based test sequence optimization. Even though, the length of the test sequences is high in ACO when compared to ISA, it lacks the branch coverage criterion. Also, the nodes in the two sequences generated using ACO are simply repeating in exactly the same positions. As the problem complexity increases, the number of states in the SUT also increases, which in turn increases the length of the test sequences in the case of ACO; whereas, the length of the test sequences in ISA has not shown that much level of increase, since it has the identification of repeated subsequences in the generated test sequences. It is shown in the Tables 4.4(a) and 4.4 (b).
Table 4.4(a)  Length of the test sequences  (C)- ISA Vs. ACO

<table>
<thead>
<tr>
<th>Case Study #</th>
<th>Structured Programming problems – in C</th>
<th>Length of Test Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Case Study</td>
<td>No. of States</td>
</tr>
<tr>
<td>1.</td>
<td>Temperature Monitoring System</td>
<td>6</td>
</tr>
<tr>
<td>2.</td>
<td>Sine Series Calculation</td>
<td>12</td>
</tr>
<tr>
<td>3.</td>
<td>Factorial</td>
<td>12</td>
</tr>
<tr>
<td>4.</td>
<td>Function Point Calculation</td>
<td>15</td>
</tr>
<tr>
<td>5.</td>
<td>Cyclomatic Complexity Calculation</td>
<td>6</td>
</tr>
<tr>
<td>6.</td>
<td>Brick Game</td>
<td>15</td>
</tr>
<tr>
<td>7.</td>
<td>Binary Search</td>
<td>12</td>
</tr>
<tr>
<td>9.</td>
<td>Find Biggest</td>
<td>12</td>
</tr>
<tr>
<td>10.</td>
<td>Square root</td>
<td>6</td>
</tr>
<tr>
<td>11.</td>
<td>Filecopy</td>
<td>15</td>
</tr>
<tr>
<td>12.</td>
<td>Sum of odd and even numbers</td>
<td>12</td>
</tr>
<tr>
<td>13.</td>
<td>Fibanacci Series</td>
<td>12</td>
</tr>
<tr>
<td>14.</td>
<td>Count</td>
<td>28</td>
</tr>
<tr>
<td>15.</td>
<td>Device Driver</td>
<td>36</td>
</tr>
</tbody>
</table>
Table 4.4(b) Length of the test sequences (C++ and Java) - ISA Vs. ACO

<table>
<thead>
<tr>
<th>Case Study #</th>
<th>Object Oriented Programming problems – in C++ and Java</th>
<th>Length of Test Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Case Study</td>
<td>No. of States</td>
</tr>
<tr>
<td>1.</td>
<td>Coffee/COCOA/ Money Lending Machine using C++</td>
<td>12</td>
</tr>
<tr>
<td>2.</td>
<td>Stack using Java</td>
<td>6</td>
</tr>
<tr>
<td>3.</td>
<td>Queue using Java</td>
<td>6</td>
</tr>
<tr>
<td>4.</td>
<td>Library Management System using Java</td>
<td>15</td>
</tr>
<tr>
<td>5.</td>
<td>Students Mark Processing System using Java</td>
<td>12</td>
</tr>
<tr>
<td>6.</td>
<td>Banking Transaction System using Java</td>
<td>12</td>
</tr>
<tr>
<td>7.</td>
<td>Shopping Cart using Java</td>
<td>12</td>
</tr>
<tr>
<td>8.</td>
<td>File System Manager using C++</td>
<td>6</td>
</tr>
<tr>
<td>9.</td>
<td>Network Monitor using C++</td>
<td>28</td>
</tr>
<tr>
<td>10.</td>
<td>Examination Workflow system using Java</td>
<td>36</td>
</tr>
<tr>
<td>11.</td>
<td>Quiz using Java</td>
<td>28</td>
</tr>
<tr>
<td>12.</td>
<td>Management Information System using Java</td>
<td>28</td>
</tr>
<tr>
<td>13.</td>
<td>Stock Maintenance using Java</td>
<td>12</td>
</tr>
<tr>
<td>14.</td>
<td>Credit Card Validation using Java</td>
<td>28</td>
</tr>
<tr>
<td>15.</td>
<td>Fraudulent Detection in Banking Transaction / Anti-Money Laundering System using Java</td>
<td>36</td>
</tr>
</tbody>
</table>
As shown in Table 4.5, the initial time taken by ACO and ISA are almost the same. But once the agent has learnt the SUT, the further processing time is tremendously decreased when compared to ACO.

**Table 4.5 Average Time taken during Optimization – ISA Vs. ACO**

<table>
<thead>
<tr>
<th>Average Time Spent for Optimization for all the Case Studies</th>
<th>Average Time Taken (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ISA</td>
</tr>
<tr>
<td>Initial Time Taken – Average Learning time</td>
<td>50</td>
</tr>
<tr>
<td>Average Processing time for optimization</td>
<td>20</td>
</tr>
<tr>
<td>Average Time spent for optimization during maintenance</td>
<td>10</td>
</tr>
<tr>
<td>Average Time spent for optimization during further maintenance process</td>
<td>10</td>
</tr>
</tbody>
</table>

### 4.8.3 Comparison Charts – ISA Vs. ACO

![Figure 4.12 Cost of ACO Vs. ISA](image)

Figure 4.12 Cost of ACO Vs. ISA
The comparison chart shown in Figure 4.12 indicates the cost associated with the test sequences generated using ACO and ISA. It has been identified that, ISA takes only less cost when compared to ACO, and is shown by taking the total cost in Y axis and algorithms in X axis.

Figure 4.13 shows that, as the number of states is increasing in a software system, the length of the test sequences is also increasing, when
optimization is done using ACO. Whereas in ISA, the length is controlled by eliminating the infeasible test sequences and repeated subsequences. So, the length of the test sequences is less in ISA.

The evaluation of the time taken by both ISA and ACO is shown in Figure 4.14. It indicates that, during the initial test run, both the algorithms take at most same amount of time for selection of the test sequences. But during the subsequent runs, ISA takes only less time when compared to ACO. This is because of the fact that, once the agent learnt the system, the subsequent modifications in the code, will be taken care automatically by the agents through their ability to behave dynamically in the changing environment. Hence, the knowledge base is updated automatically and seamlessly through blackboard based learning without any user intervention.

4.9 SUMMARY

This chapter described the application of Intelligent Agents in test sequence optimization. The developed agent namely the Intelligent Search Agent (ISA), couples human like intelligence and automation as a tool in selection and optimization of test sequences in the SUT. In this proposed approach, two mathematical models have been formulated, for infeasible test sequence identification and repeated subsequence detection to achieve test sequence optimization. During this optimization, the search process is equipped with two important heuristics namely ‘Happiness Value’ and ‘Similarity Measure’ to guide intelligent searching.

The results of the proposed approach using Intelligent Search Agent (ISA) have been compared with Ant Colony Optimization (ACO) based approach with respect to time, cost and length of the test sequences. Since, ACO has been a proven technique when compared to other techniques, this
thesis compared the proposed approach with ACO only; and by transitivity it proves that ISA is better than all other approaches.

The proposed Intelligent Search Agent (ISA) is coded in Java and is packed as a tool “Intelligent Tester”, which is provided as part of the major tool “IntelligenTester”. The screenshots and sample source code of the tool are given in Appendix 1.

The evaluation results show that, the test sequences generated by ISA are sufficient for covering the states and branches in the SUT. ACO does not take into consideration the branch coverage criterion and the sequences are simply the repetition of previously visited states. Also, from the evaluations, it is clear that ISA consumes less time and cost and also generates lesser length (reduction of up to 50%) test sequences when compared to ACO based on several bench mark problems. Hence, the proposed test sequence optimization approach proved that, ISA is better in test sequence optimization when compared to ACO. Even though the initial learning time taken by the agent is high; later on, the agent takes only less time to learn and plan any modifications made in the SUT. Thus the reduced set of test sequences using ISA consumes only less time and cost when compared to ACO and saves tremendous amount of resources needed in software testing process.