**COMBINATORIAL EXPLOSION**

**Abstract**:

Of the various approaches adopted to tackle problems that can be solved by computational power, enumeration and constraint-checking is intuitive and simplest. Enumeration is directly dependent on the input size. With increasing input size, enumerated scenarios that are to be checked for constraint satisfaction also grows. For some problems the growth is linear or sub-linear and for others the growth is exponential. In many cases this increasing enumerated scenarios cause combinatorial expansion i.e, the exponential increase in the search space for many problems. The inefficiency induced due to the sheer number of combinations to be considered in order to produce a (optimal or complete) solution is called combinatorial explosion. In this paper, combinatorial manifestations in several domains are presented and discussed. Some intuitive and novel ways, often domain dependent, that are employed to avoid this problem will be presented. Finally, the impact of computation technology and heuristics for approximation will be presented.

**Introduction**:

Search is inherent to the problems and methods of artificial intelligence (AI). That is because AI problems are intrinsically complex. Efforts to solve problems with computers which humans can routinely solve by employing innate cognitive abilities, pattern recognition, perception and experience, invariably must turn to considerations of search. All search methods essentially fall into one of two categories:

(a) Exhaustive (blind) or uninformed methods and

(b) Heuristic or informed methods.

All search methods in computer science share in common three necessities:

(a) a world model or database of facts based on a choice of representation providing the current state, as well as other possible states and a goal state.

(b) Set of operators which defines possible transformations of states and

(c) Control strategy which determines how transformations amongst states are to take place by applying operators

Exhaustive search of a problem space (or search space) is often not feasible or practical due to the size of the problem space. This can be best visualized as follows:

Consider each state to be a node of a tree. The different actions that can be taken from the node form the children of the node. Each of these children in turn spawns children that correspond to the actions that can be taken from their respective states and so on. Thus starting with one (or few) states, we end up with an exponential number of nodes. This exponential number is dependent on the branching factor and the height of the tree. Branching factor stands for the number of decisions at each stage and the height stands for the number of stages that we have decide at. The leaf nodes of the tree all stand for a sequence of actions which could be a possible solution to the problem posed. Regardless of the way we traverse this tree if all the solutions are to be found then all legal action-sequences, that satisfy the constraint, have to be visited. This makes the problem search space exponential in nature.

Let us consider a typical search problem. Assume that we have set S with N variables. Say each of these variables can take values from the set of domains S’. Thus a variable Vi can only take values from the domain Di. The sets and domains are notated below.

S = {V1, V2, … , V­­N}

S’ = {D1 , D2 , …, DN}

Say we are to find values for V1 to VN such that Vi < Vi+1 for i = 1, 2, …, N-1. That is the values in order form a increasing order. The usual enumeration and constraint checking approach solves this problem by enumerating each of the possible value sequences and checking if the above constraint holds.

Say the cardinality(Di) = K.

Thus the total number of value-sequences or value-orders to be enumerated is KN. Let us see some numbers to better understand the increase in the number of value-orders to be enumerated. Let the total number of enumerations be #(N,K). For K = 10, we have:

#(2,10) = 100

#(3,10) = 1000

#(4,10) = 10000

#(5,10) = 100000

#(6,10) = 1000000

Hence, when an extra variable is added to the problem, the number of enumerations becomes K times the previous value. Thus increase in the size of n leads to exponential increase in the size of the enumerations. Thus for N = 20, #(20,10) = 1020. So if we are to solve this problem with a computer that can execute 1010 instructions per second (optimistic expectation), it would still take 1010 seconds (317 years) to produce all the value-orders. So the problem became practically incomputable just by changing the value of N from 2 to 20. We notice that this happens due to huge number of combinations that were to be considered. This is called combinatorial explosion. Thus, Combinatorial Explosion is broadly defined as problem that the number of combinations that one has to examine grows exponentially, so fast that even the fastest computers will require an intolerable amount of time to examine them.

We see combinatorial explosion in all walks of the digital life. Circuit design testing faces this problem, as the search space increases by a factor of 2 with every added input taking values ‘low’ and ‘high’. The number of specialized communication channels that serve as the media for point to point communication grows exponentially with the number of users. This grows in the order of (nC2). Search space enumeration, as we have seen above, runs into combinatorial explosion. Software testing has a similar hurdle [2] as each branch in the code provides two paths for the control flow.

**Avoiding Combinatorial Explosion:**

Many heuristic functions exist that reduce the search space enormously. Also various techniques that employ partial constraint checking also help reduce the huge space of the enumerated combinations. Most of these techniques are domain dependent. Techniques like alpha-beta pruning are applicable to most of the search problem involving zero-sum competitive games.

For the example search problem provided in introduction, some techniques to avoid combinatorial explosion are as follows:

(a) **Test and Generate**: Without actually computing all the values of an order sequence and finally checking for constraint satisfaction, incremental constraint checking could be used. For example, after values V1 and V2 have been chosen, validate all the constraints that use only V1 and V2. Here the partial constraint is V1 < V2. If this is not satisfied, then any values for V3 to VN will not satisfy the total constraint. This way the search space can be reduced drastically. Some statistics employing the above partial constraint checker are provided below.

|  |  |  |
| --- | --- | --- |
| #(N,K) | Without partial constraint checking | With partial constraint checking |
| #( 6, 6) | 93,311 | 631 |
| #(15, 15) | 437,893,890,380,859,375 | 917,477 |

(b) **Propagate and Distribute**: Even with test and generate, we notice that the number of nodes to be considered is still high. Propagate and Distribute aims at reducing the number of nodes to be searched even further. The idea is to make the tests active instead of being passive and just checking if the constraint is satisfied or violated; rather the tests are made to propagate constraints. In the above example of #(15, 15), both V1 and V2 take values in {1,...,15}.

Furthermore we have the constraint that: V1 < V2

This means that the value of V2 must be at least 1 greater than that of V1. Therefore V1 must actually take values in {1,...,14} and V2 in {2,...,15}. This is repeated with the same reasoning with V2 and V3, etc, until V14 and V15, at which point we obtain the conclusion that V15 can take only the values in {15}. In other words the only possible value forV15 is 15. By iterating this process we deterministically arrive at the conclusion that V1 = 1, V2 = 2, ..., V15 = 15. Thus, the search tree now contains only one node.

In general, however, search may still needs to be performed, but the idea is to first derive as much as possible through deterministic inference (forward or backward) using the available constraints and only then make a non-deterministic choice if still necessary. This is also the general method of constraint programming which is often paraphrased as `propagate and distribute' [3]. A propagation step restricts the set of possible solutions using simple, deterministic inference. A distribution step performs a non-deterministic case distinction and should only be considered when no further inferences are possible through propagation alone. In this fashion, the search tree requires much fewer choice points: propagation is said to `prune' the search tree.

**Combinatorial Explosion in Software Testing:**

Testing consumes significant amounts of resources in development projects, some estimates projecting the testing time and effort to be half of that of the total project. For an activity taking up so much time and effort, testing does not guarantee strictly bug-free code in many cases. This is because of the huge state space of inputs that need to be tested. Combinatorial explosion is a frequently occurring problem in testing. One instance of combinatorial explosion in testing is when systems under test have several parameters, each with so many possible values that testing every possible combination of parameter values is infeasible. Another instance of combinatorial explosion in testing may occur for configurable systems. When systems under test have many configuration parameters, each with several possible values, testing each configuration is infeasible. Examples of configuration parameters are versions of a specific software or hardware module, different types of software or hardware modules, and number of logical or physical entities included in a computer system.

In the following sections we consider a detailed analysis of combinatorial explosion in software testing and its implications. We will look into some methods to reduce the effect of combinatorial explosion and how well these methods fit into the software testing approaches currently in use.

**Combinatorial Strategies:**

Apart from the methods and heuristics that we have seen to counter combinatorial explosion, a family of methods called combinational strategies is useful, particularly in testing. The fundamental property of combination strategies is their ability to select a subset of all possible combinations such that a coverage criterion is satisfied. Thus, Combination strategies are test case selection methods designed to handle combinatorial explosion. Combinatorial explosion can occur in testing when a test object can be described by a number of parameters, each with many possible values. Every complete combination of parameter values is a potential test case. Due to its generality, combinatorial explosion is a common problem in practical testing. The effect of combinatorial explosion is that it is infeasible to test every possible combination of parameter values. Combination strategies handle the combinatorial explosion by identification of a tractable subset of all combinations. This selection is based on coverage, which enables objective evaluation of the selected combinations.

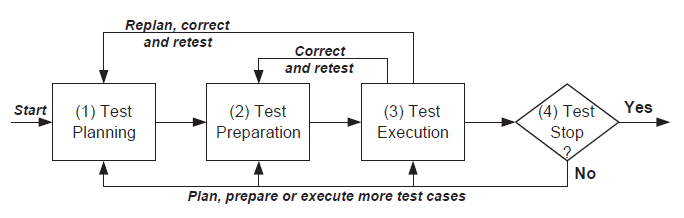
The automatic generation of test plans for technical systems becomes more and more important, especially since the producer of a technical product can be made liable for any damage that is caused by the product. Test generation and quality control is not only important for new products. Systems that have been repaired or maintained must also be tested again. A test plan is a sequence of tests (or measurements) being used to indicate that the behavior of a system is correct with respect to a formal speciﬁcation. The composition of tests must consider several criteria - First of all, the test plan must be complete in some sense, i.e. if there is a fault in a component, it should be detected by at least one test. In most cases, non-trivial assumptions must be made to guarantee completeness: the fault model completeness assumption, the single-fault assumption and the non-intermittency assumption. Thus, the challenge for Combination strategies is to select a subset of the test cases that will produce the same result as if the whole search space is tested.

**Input Parameter Model:**

Under Combination strategies the system under test is represented as an input parameter model (IPM). As an example, consider testing function *int index(element, vector)* which returns the index of the element in the vector. An IPM of the index function consists of two parameters, one representing different elements and the other representing different vectors. An alternative IPM could use one parameter to represent possible results, for instance, element found first in vector, element found last in vector, element not found at all, etc. A second parameter could be used to represent the size of the vector, for instance, zero elements, one element, and several elements. We notice that the IPM depends on the tester and the approach followed. For now, let us not consider the advantages and disadvantages of each type of IPM’s.

**Testing:**

Testing is the activity in which test cases are identified, prepared and executed. A test case contains, at least, some input and expected results. The software unit (e.g., module, program or system) under test is called the test object. During the execution of a test case, the test object is stimulated by the input of the test case and reacts to this by producing actual results. The expected results of the test case is compared with the actual results produced by the test object and a test result, that is, either pass or fail, is produced. A failure is a deviation of the delivered service of the software unit from fulfilling the system function. A failure is normally detected by a difference between expected and actual result. An error is the part of the system state that is liable to lead to a subsequent failure. Finally, a fault in the general sense is the adjudged or hypothesized cause of an error. A set of test-cases is called a test-suite.



**Fig 1.a – General flow of testing**

The general workflow of testing can be seen in fig 1.a. Step 1 of any test process is to plan the forthcoming activities. The planning includes, at least, identifying the tasks to be performed, estimating the amount of resources needed to perform the tasks, and making financial and time budgets. Step 2 is to make any preparations needed for the upcoming test execution. Important tasks during the preparation step are to select and document the test cases. In step 3, the test cases are executed and test results are collected. These results are then analyzed in step 4 in order to determine whether or not more testing is needed. If more testing is needed feedback loops allow for returning to any of the previous steps depending on the amount of work needed. Also, feedback loops from step 3 allow for correction and re-execution of failed test cases. During testing it is important to use test cases with both valid and invalid values. Valid values are values within the normal operating ranges of the test object and correspondingly invalid values are values outside the normal operating ranges. Testing using invalid values in the test cases, called negative testing, is used to test error or exception handling mechanisms.

The quality of the software and testing depend considerably on the quality of requirements. Different combination strategies are used based on the quality of requirements. Broadly, combination strategies are classified into deterministic and non-deterministic categories. Non-deterministic strategies employ randomness in testing. These strategies, hence, produce different solutions to same test objects at different times. The deterministic combination strategies always produce the same solution for a given problem. Deterministic combination strategies are further divided into instant and iterative combination strategies. The instant combination strategies produce all combinations in an atomic step while the iterative combination strategies build the solution one combination at a time.

**Test cases and Coverage:**

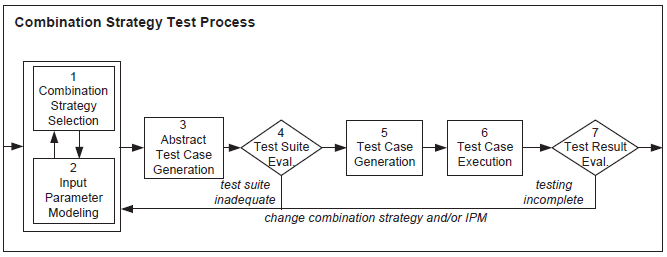
Based on the IPM, combination strategies generate abstract test cases. Abstract test cases are combinations of IPM parameter values consisting of one value from each IPM parameter. Abstract test cases may be translated, in a post-processing step, into actual inputs of the test object. Coverage is a key factor when deciding which combination strategy to use. Different combination strategies support different levels of coverage with respect to the IPM. The level of coverage affects the size of the test suite and the ability to detect certain types of faults. 1-wise (also known as each-used) coverage is the simplest coverage criterion. 1-wise coverage requires that every value of every IPM parameter is included in at least one test case in the test suite. 2-wise (also known as pair-wise) coverage requires that every possible pair of values of any two IPM parameters is included in some test case. Note that the same test case can often cover more than one unique pair of values. A natural extension of 2-wise coverage is t-wise coverage, which requires every possible combination of values of t IPM parameters to be included in some test case in the test suite. The most thorough coverage criterion, N-wise coverage, requires a test suite to contain every possible combination of the IPM parameter values in the IPM. Base choice coverage is an alternative coverage criterion partly based on semantic information. One base value is selected from each parameter. The base value may, for instance, be selected based on the most frequently used value of each parameter. The combination of base values is called the base test case. Base choice coverage requires every value of each IPM parameter to be included in a test case in which the rest of the values are base values. Further, the test suite must also contain the base test case.

Model-based test case selection methods are categorized into control-flow based and data-flow based test case selections. In both these cases, models (graphs) are derived from the source code. This has two important implications for testing. First, modeling (and hence testing) cannot be started before the source code is finished. Second, the models can be generated automatically. With automation, the potential gain could be that the cost of generating several abstract test suites would be small. Also, costly test activities can be postponed until it is certain that these activities should be executed. Moreover, the same IPM can be input to several combination strategies supporting different coverage levels thus reusing the test model for various test objects.

**Selecting Combination Strategies:**

There are 15 major combination strategies that have been identified from survey of relevant literature []. Some sources of literature point that using a combined approach of several combination strategies could be more useful than using a single combination strategy. Thus the sheer number of combination of combination strategies is significant. This topic will be discussed in more detail in the results section that will be added in the next paper submission. However, in short which combination strategy to apply may depend on several properties, like associated coverage metric, size of generated test suite, and types of faults that the combination strategy targets.

**Detailed Combination Strategy Test Process:**

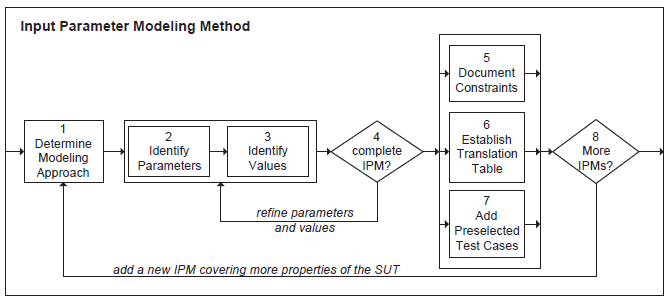


**Fig 1.b**

Figure 1.b shows a test process specifically designed for the use of combination strategies. The planning step of the generic test process is omitted in the combination strategy test process. This is because the planning step is general in the sense that the planning decisions made govern the rest of the testing activities, for instance which test case selection methods to use. One of the several implications of this is that instructions on how to use a specific test case selection method do not need to take planning into account. Thus, it is beneficial to keep planning and test case selection independent. Apart from the absence of a planning activity, the main difference between the combination strategy test process and the generic test process is in the test preparation activity of the generic test process. In the combination strategy test process, this activity is refined to satisfy the requirements from combination strategies. Steps (1)-(5) in the combination strategy test process are all specific to combination strategies. Step (1) is to select a combination strategy to use. Step (2) is to construct an IPM. There is a bidirectional dependence between steps (1) and (2), that is, the results of one step may affect the other step. For instance, if the combination strategy base choice is selected, one value for each IPM parameter in the IPM should be marked as the base choice. In a similar way, if the result of input parameter modeling is two or more IPMs, it may be favorable to use different combination strategies for the different IPMs. Hence, the combination strategy test process should support multiple iterations between the two steps choice of combination strategies and creation of an IPM. The arrows between the two steps provide this possibility. Fig 1.c gives insights into how the sub-process of choosing an IPM. Step (3) is the generation of abstract test cases. In this step, the selected combination strategies are applied to the created IPM. The result of this step is an abstract test suite. This step is possible to automate as most combination strategies can be expressed as a step by step algorithm. The selection of test cases is often influenced by the time available for test case execution. In step (4) the abstract test suite is evaluated. The evaluation may, for instance, focus on the size of the test suite and indirectly consider the testing time.

If the abstract test suite is too large the test process may go through steps one and two to try to reduce the size of the test suite. The advantage with this approach is that the costly parts of test case development, that is, identification of expected results and documentation of test cases are postponed until it is certain that the test cases will actually be used. In step (5), “test case generation", the abstract test cases are transformed into executable test cases. This step consists of more than three tasks. The first task is the identification of actual test case inputs to the test object. The abstract test cases are converted into real test case inputs through some mapping function that is established during the input parameter modeling. The second task is to identify the expected result for the specific input and the third task is to document the test case in a suitable way. If the test execution is to be automated then test programs are to be written under this step. For manual test execution, test case instructions are to be documented. However, automation of this step is not straight forward. All three of the test generation tasks are difficult to automate. Identification of actual test case inputs can be automated but it requires that the function mapping IPM parameter values to actual inputs be formalized.

Automation of the identification of the expected results may possibly be the most difficult task to automate as the specification needs to be semantically exact and machine readable and hence expressed in some formal language. Lastly, automatic documentation of test cases requires code generation, which also requires semantically exact specifications. Unless these requirements are satisfied, the test generation step is likely to be relatively expensive due to much manual intervention. Step (6) of the combination strategy test process is test case execution. As the name implies, the test cases are executed and the results recorded. There are no differences in this step compared to the corresponding step in the generic test process. The final step (7) is the test stop decision. Again, it is a copy of the corresponding test stop decision step in the generic test process. Steps (6) and (7) are included in the combination strategy test process to indicate the opportunity for combination strategy re-selection and input parameter re-modeling should the test results be unsatisfactory.



**Fig 1.c IPM method**

The six combination strategies namely - Each Choice (EC), Base Choice (BC), Orthogonal Arrays (OA), In Parameter Order (IPO), Automatic Efficient Test Generator (AETG), and All Combinations (AC) - which were obtained from earlier works were run standard tests to identify their impact on and synergy with existing testing methods. The results of the tests will be presented in the next paper submission. Briefly, the results focus on the following properties:

1. Coverage Criteria
2. Test Suite Size
3. Algorithm Complexity
4. Fault types
5. Number of faults

**Conclusion:**

As seen with some examples, it is hard to find techniques for search space reduction or tree pruning that are applicable to all problems. However, domain specific heuristics and methods often scale down the problem and make it practically feasible to compute. Approximation and Monte-Carlo methods are other approaches through which combinatorial explosion can be avoided. The effect of parallel computation also helps in the above, though no algorithms use this explicitly.

Regarding combinatorial explosion in software testing - we have seen that stating the requirements itself is a difficult task owing to the different ways in which a single test object can be interpreted. It is understood that software testing can never be fully automated, however, with possible automation the cost spent on testing could be lowered while increasing the reliability of software. A detailed testing process using combination strategies was looked at. The test object for which experiments shall be run using the six models previously stated, along with the results will be presented in the next-paper submission.

**References**:

[1] *Handling Combinatorial Explosion in Software Testing* by Mats Grindal, Disseratation

http://www.diva-portal.org/smash/get/diva2:17568/FULLTEXT01.pdf

[2] *Search Techniques to Contain Combinatorial Explosion in Artificial Intelligence*, Ajay Kumar Gaur, International Journal of Engineering Research and Technology, September 2012.

[3] *Using formal methods to derive test frames in category partition testing*, P. E. Ammann and A. J. Offutt, In Proceedings of Ninth Annual Conference on Computer Assurance (COMPASS'94), Gaithersburg MD, pages 69-80. IEEE Computer Society Press, Jun. 1994

[4] *A practical strategy for testing pair-wise coverage of network interfaces*, A. W. Williams and R. L. Probert, In Proceedings of Seventh International Symposium on Software Reliability Engineering (ISSRE96), 1996.

[5] *In-parameter-order: A test generation strategy for pair-wise testing*, Y. Lei and K. C. Tai, In Proceedings of Third IEEE High Assurance Systems Engineering Symposium, November 1998.

[6] *Orthogonal Latin Squares: An application of experiment design to compiler testing*. R. Mandl, In Communications of the ACM, 28(10):1054-1058, October 1985.

[7]*Avoiding Combinatorial Explosion in Automatic Test Generation: Reasoning about Measurements is the Key*, Harald Lange, Ralf Möller, Bernd Neumann, in proceedings: KI'96, Dresden, September 1996.

[8] *Experimental evaluation of algorithms for solving problems with combinatorial explosion*, Marco Gavanelli, Toni Mancini, RCRA international workshop 2012.