aCOMBINATORIAL 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.

**Automatic Test Generation:**

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.

Several approaches to avoid the combinatorial explosion like half-paths, measurements with half-paths, blocking cycles will be discussed in the next paper submission.

**Effect of Computational Capacity and Heuristics**:

**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. These will be talked about in the next paper-submission.

**References**:

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

[2] *Handling Combinatorial Explosion in Software Testing* by Mats Grindal

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

### [3] [Constructive Disjunction in Oz - Mozart Programming System](http://mozart.github.io/publications/abstracts/CDinOz.html).

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

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