Search Techniques To Contain Combinatorial Explosion in Artificial Intelligence

Ajay Kumar Gaur
Research Scholar Singhania University Rajsthan

Abstract

This paper reviews the current literature regarding the artificial intelligence techniques to contain combinatorial explosion as well as some methods used to optimize the final solution. Through the analysis of the specific problem and the previous work in the literature, this paper will establish the scholarly base for the research methodology used in this thesis. Evaluation of various search algorithms will assist in the development of a algorithm to determine the best technique to solve the problem of combinatorial Explosion.

1 Introduction

Research in Artificial Intelligence (AI) has a long tradition. The first paper attributed to the field was published by Warren McCulloch and Walter Pitts in 1943 [1], and the term "artificial intelligence" was proposed and agreed at the famous Dartmouth Workshop held in 1956.

What is Intelligence?

The notion of human Intelligence is very complex; it comprises the following (and possibly many other) capabilities:

- Understanding meaning of symbols, words, text, data, images, utterances
- Learning (acquiring knowledge) from data, text, images as well as from own behaviour and behaviour of others and learning by discovery
- Analysing (deconstructing) complicated situations
- Making choices (decisions) under conditions of variety and uncertainty and therefore solving incompletely specified problems and achieving goals under conditions of the occurrence of frequent unpredictable events
- Interacting (communicating) with other actors in the environment, which include intelligent creatures and machines
- Autonomously adapting to changes in the environment
- Creating (constructing) new concepts, principles, theories, methods, artefacts, models, literature, art
- Setting and achieving goals by competing and/or cooperating with others

An important part of human intelligence is to strive to create Artificial Intelligence.

"Artificial" means man-made rather than natural. Artificial Intelligence is supposed to be man-made intelligence, designed and implemented in computer software and built into art effects such as robots or intelligent machines [3]. Historically artificial intelligence programs appeared in various disguises such as universal problem solvers , expert systems , and neural networks .

3.SEARCH METHODS IN ARTIFICIAL INTELLIGENCE :

Search is inherent to the problems and methods of artificial intelligence (AI) [4]. 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.

3.1 Uninformed Search Methods

3.1.1 Search Strategies

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

1) 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.

2) a set of operators which defines possible transformations of states and

3) a control strategy which determines how transformations amongst states are to take place by applying operators.

Reasoning from a current state in search of a state which is closer to a goal state is known as forward reasoning. Reasoning backwards to a current state from a goal state is known as backward reasoning. As such it is possible to make distinctions between bottom up and top down approaches to problem solving. Bottom up is often "goal directed" -- that is reasoning backwards from a goal state to solve intermediary sub-goal states. Top down or data-driven reasoning is based on simply being able to get to a state which is defined as closed to a goal state than the current state. Often application of

operators to a problem state may not lead directly to a goal state and some backtracking may be necessary before a goal state can be found (Barr & Feigenbaum, 1981).

3.1.2 State Space Search

Exhaustive search of a problem space (or search space) is often not feasible or practical due to the size of the problem space. In some instances it is however, necessary. More often, we are able to define a set of legal transformations of a state space (moves in the world of games) from which those that are more likely to bring us closer to a goal state are selected while others are never explored further. This technique in problem solving is known as split and prune. In AI the technique that emulates split and prune is called generate and test [4]. The basic method is:

Repeat
    Generate a candidate solution
    Test the candidate solution
Until a satisfactory solution is found, or
    no more candidate solutions can be generated:
If an acceptable solution is found, announce it;
    Otherwise, announce failure.

Good generators are complete, will eventually produce all possible solutions, and will not suggest redundant solutions. They are also informed; that is, they will employ additional information to limit the solutions they propose.

Means-ends analysis is another state space technique whose purpose is, given an initial state to reduce the difference (distance) between a current state and a goal state. Determining "distance" between any state and a goal state can be facilitated difference-procedure tables which can effectively prescribe what the next state might be. To perform means-ends analysis:

Repeat
    Describe the current state, the goal state, and the difference between the two.
Use the difference between the current state and goal state, possibly with the description of the current state or goal state, to select a promising procedure.
Use the promising procedure and update the current state.
Until the GOAL is reached or no more procedures are available
If the GOAL is reached, announce success; otherwise, announce failure.

The technique of problem reduction is another important approach to AI problems. That is, to solve a complex or larger problem, identify smaller manageable problems (or subgoals) that you know can be solved in fewer steps. steps.

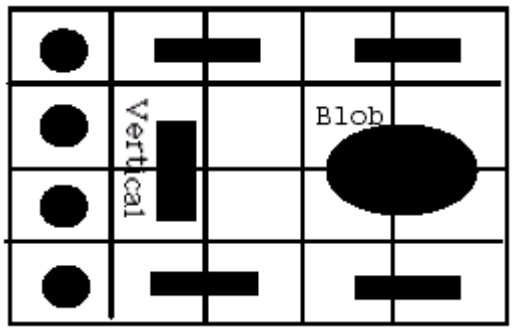


Figure 1 : Problem Reduction and The Sliding Block Puzzle Donkey

This sliding block puzzle has been known for over 100 years[4]. The object is to be able to bypass the Vertical bar with the Blob and place the Blob on the other side of the Vertical bar . The Blob occupies four spaces and needs two adjacent vertical or horizontal spaces in order to be able to move while the Vertical bar needs two adjacent empty vertical spaces to move left or right, or one empty space above or below it to move up or down. The Horizontal bars can move to any empty square to the left or right of them, or up or down if there are two empty spaces above or below them. Likewise, the circles can move to any empty space around them in a horizontal or vertical line. A relatively uninformed state space search can result in over 800 moves for this problem to be solved, with plenty of backtracking necessary. By problem reduction, resulting in the subgoal of trying the get the Blob on the two rows above or below the vertical bar, it is possible to solve this puzzle in just 82 moves!

Another example of a technique for problem reduction is called And/Or Trees. Here the goal is to find a solution path to a given tree by applying the following rules:

A node is solvable if --

1. it is a terminal node (a primitive problem),
2. it is a nonterminal node whose successors are AND nodes that are all solvable, OR
3. it is a nonterminal node whose successors are OR nodes and least one of them is solvable.

Similarly, a node is unsolvable if –

1. it is a nonterminal node that has no successors)a nonprimitive problem to which no operator applies),
2. it is a nonterminal node whose successors are AND nodes and at least one of them is unsolvable, or
3. it is a nonterminal node whose successors are OR nodes and all of them are unsolvable.

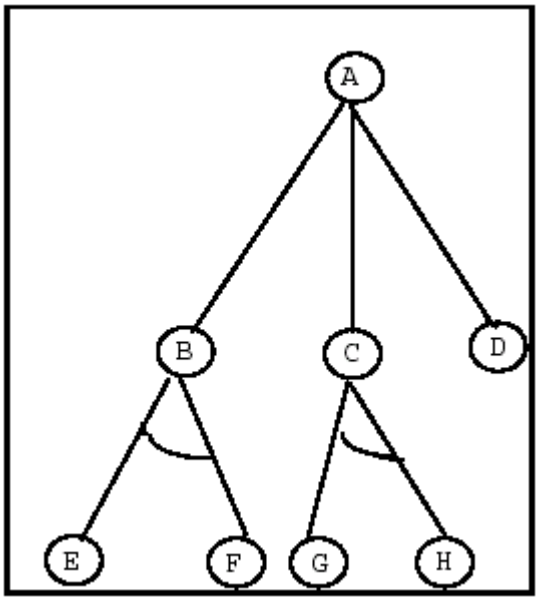


Figure 2: And/Or Tree

In this figure nodes B and C serve as exclusive parents to subproblems EF and GH respectively. One of viewing the tree is with nodes B, C, and D serving as individual, alternative subproblems . Solution paths would therefore be: {A-B-E}, {A-B-F}, {A-C-G}, {A-C-H}, and {A,D}.

In the special case where no AND nodes occur, we have the ordinary graph occurring in a state space search. However the presence of AND nodes distinguishes AND/OR Trees (or graphs) from ordinary state structures which call for their own specialized search techniques (Nilsson, 1971).

3.1.2 .1 Depth First Search :

The Depth First Search (DFS) is one of the most basic and fundamental Blind Search Algorithms. It is for those who want to probe deeply down a potential solution path in the hope that solutions do not lie too deeply down the tree. That is "DFS is a good idea when you are confident that all partial paths either reach dead emds or become complete paths after a reasonable number of steps. In contrast, "DFS is a bad idea if there are long paths, even infinitely long paths, that neither reacj dead emds nor become complete paths (Winston, 1992).

To conduct a DFS:

(1) Put the Start Node on the list called OPEN.
(2) If OPEN is empty, exit with failure; otherwise continue.
(3) Remove the first node from OPEN and put it on a list called CLOSED. Call this node n.
(4) If the depth of n equals the depth bound, go to (2); Otherwise continue.
(5) Expand node n, generating all successors of n. Put these (in arbitrary order) at the beginning of OPEN and provide pointers back to n.
(6) If any of the successors are goal nodes, exit with the solution obtained by tracing back through the pointers; Otherwise go to (2).

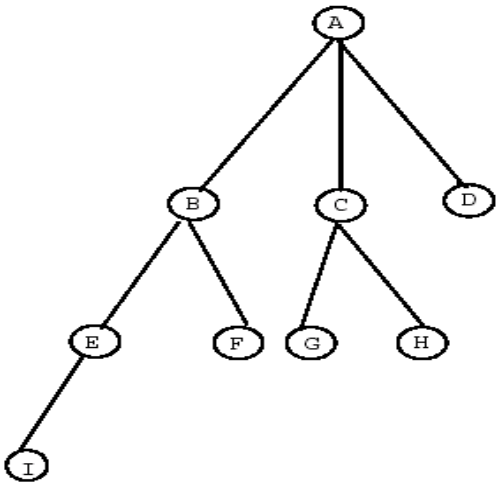


Figure 3: Tree Searching Example of Depth First Search and Breadth First Search

DFS always explores the deepest node first. That is, the one which is farthest down from the root of the tree. To prevent consideration of unacceptably long paths, a depth bound is often employed to limit the depth of search. DFS would explore the tree in Figure 5 in the order: A-B-E-I-F-C-G-H-D.

DFS with Iterative Deepening remedies many of the drawbacks of the DFS and the Breadth First Search. The idea is to perform a level by level DFS. It starts with a DFS with a depth bound of 1. If a goal is not found, then it performs a DFS with depth bound of 2. This continues, with the depth bound increasing by one with each iteration, although with each increase in depth the algorithm must re-perform its DFS to the prescribed bound. The idea of Iterative Deepening is credited to Slate and Adkin (1977) with their work on the Northwestern University Chess Program. Studies of its efficiency have been carried out by Korf (1985).

3**.1.2 .2 Breadth First Search:**

Breadth First Search always explores nodes closest to the root node first, thereby visiting all nodes of a given length first before moving to any longer paths. It pushes uniformly into the search tree. Breadth first search is most effective when all paths to a goal node are of uniform depth. It is a bad idea when the branching factor (average number of offspring for each node) is large or infinite. Breadth First Search is also to be preferred over DFS if you are worried that there may be long paths (or even infinitely long paths) that neither reach dead ends or become complete paths (Winston, 1992). For the tree in Figure 5 Breadth First Search would proceed alphabetically.

The algorithm for Breadth First Tree Search is:

(1) Put the start node on a list called OPEN.
(2) If OPEN is empty, exit with failure; Otherwise continue.
(3) Remove the first node on OPEN and put it on a list called CLOSED; Call this node n;
(4) Expand node n, generating all of its successors. If there are no successors, go immediately to (2). Put the successors at the end of OPEN and provdie pointers from these successors back to n.
(5) If any of the successors are goal nodes, exit with the solution obtained by tracing back through the pointers; Otherwise go to (2)

3.1.3 Bidirectional Search

To this point all search algorithms discussed (with the exception of means-ends analysis and backtracking) have been based on forward reasoning. Searching backwards from goal nodes to predecessors is relatively easy. Pohl (1969, 1971) combined forward and backward reasoning into a technique called bidirectional search. The idea is to replace a single search graph, which is likely to grow exponentially, with two smaller graphs -- one starting from the initial state and one starting from the goal. The search is approximated to terminate when the two graphs intersect This algorithm is guaranteed to find the shortest solution path through a general state-space graph. Empirical data for randomly generated graphs showed the Pohl's algorithm expanded only about 1/4 as many nodes as unidirectional search (Barr and Feigenbaum, 1981). Pohl also implemented heuristic versions of this algorithm.

3.2 Heuristic or informed methods.

The problem with all Uninformed Search Methods algorithms is that their time complexities grow exponentially with problem size. This problem is called combinatorial explosion.

The combinatorial explosion results in limited size of problems that can be solved with with Uninformed Search Methods techniques. For example while eight-puzzle with 10^5 states is easily solved by Uninformed Search Methods the fifteen-puzzle contains over 10^{13} states, and hence cannot be solved with Uninformed Search Methods techniques on current machines. Even faster machines cannot have significant impact on this problem since the 5 * 5 twenty-four puzzle contains almost 10^{25} states.

George Polya, via his wonderful book "How To Solve It" (1945) may be regarded as the "father of heuristics". In essence Polya's effort focused on problem-solving, thinking and learning. He developed a short "heuristic Dictionary" of heuristic primitives. Polya's approach was both practical and experimental. He sought to develop commonalities in the problem solving process through the formalization of observation and experience.

Present-day researches notions of heuristics are somewhat distant from Polya's (Bolc and Cytowski, 1992). Tendencies are to seek formal and rigid algorithmic solutions to specific problem domains rather than the development of general approaches which could be appropriately selected and applied for specific problems.

The goal of a heuristic search is to reduce the number of nodes searched in seeking a goal. In other words, problems which grow combinatorially large may be approached. Through knowledge, information, rules, insights, analogies, and simplification in addition to a host of other techniques heuristic search aims to reduce the number of objects examined. Heuristics do not guarantee the achievement of a solution, although good heuristics should facilitate this. Heuristic search is defined by authors in many different ways:

• it is a practical strategy increasing the effectiveness of complex problem solving (Feigenbaum, Feldman, 1963) [5]
• it leads to a solution along the most probable path, omitting the least promising ones (Amarel, 1968) [6]
• it should enable one to avoid the examination of dead ends, and to use already gathered data (Lenat, 1983) . [7]

The points at which heuristic information can be applied in a search include:

1. Deciding which node to expand next, instead of doing the expansions in a strictly breadth-first or depth-first order;
2. In the course of expanding a node, deciding which successor or successors to generate -- instead of blindly generating all possible successors at one time, and
3. Deciding that certain nodes should be discarded, or pruned, from the search tree.

Bolc and Cytowski (1992) add:

... use of heuristics in the solution construction process increases the uncertainty of arriving at a result ... due to the use of informal knowledge (rules, laws, intuition, etc.) whose usefulness have never been fully proven. Because of this, heuristic methods are employed in cases where algorithms give unsatisfactory results or do not guarantee to give any results. They are particularly important in solving very complex problems (where an accurate algorithm fails), especially in speech and image recognition, robotics and game strategy construction. ...

Heuristic methods allow us to exploit uncertain and imprecise data in a natural way. ... The Main objective of heuristics is to aid and improve the effectiveness of an algorithm solving a problem.

Most important is the elimination from further consideration of some subsets of objects still not examined. ..."

Most modern heuristic search methods are expected to bridge the gap between the completeness of algorithms and their optimal complexity (Romanycia and Pelletier, 1985)[8]. Strategies are being modified in order to arrive at a quasi-optimal -- instead of an optimal -- solution with a significant cost reduction (Pearl, 1984).

Games, especially two-person, zero-sum games of perfect information like chess and checkers have proven to be a very promising domain for studying and testing heuristics.

Following is a list of heuristic search techniques[9].

Pure Heuristic Search,
A* Search Algorithm,
Algorithm Iterative-Deepening A*,
Depth-First Branch-And-Bound
Heuristic Path Algorithm,
Recursive Best-First Search

3.2. 1 Pure Heuristic Search

The simplest of heuristic search algorithms, the pure heuristic search, expands nodes in order of their heuristic values h(n). It maintains a closed list of those nodes that have already been expanded, and a open list of those nodes that have been generated but not yet been expanded. The algorithm begins with just the initial state on the open list. At each cycle, a node on the open list with the minimum h(n) value is expanded, generating all of its children and is placed on the closed list. The heuristic function is applied to the children, and they are placed on the open list in order of their heuristic values. The algorithm continues until a goal state is chosen for expansion.

In a graph with cycles, multiple paths will be found to the same node, and the first path found may not be the shortest. When a shorter path is found to an open node, the shorter path is saved and the longer one is discarded. When a shorter path to a closed node is found, the node is inaved to open and the shorter path is associated with it. The main drawback of pure heuristic search is that since it ignores the cost of the path so far to node n, it does not find optimal solutions.

Breadth-first search, uniform-cost search, and pure heuristic search are all special cases of a more general algorithm called best-first search. In each cycle of a best-first search, the node that is best according to some cost function is chosen for expansion. These best-first search algorithms differ only in their cost functions the depth of node n for breadth-first search, g(n) for uniform-cost search h(n) for pure heuristic search.

3.2. 2 A* Algorithm

The A* algorithm combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions. A* algorithm is a best-first search algorithm in which the cost associated with a node is f(n) = g(n) + h(n), where g(n) is the cost of the path from the initial state to node n and h(n) is the heuristic estimate or the cost or a path from node n to a goal. Thus, f(n) estimates the lowest total cost of any solution path going through node n. At each point a node with lowest f value is chosen for expansion. Ties among nodes of equal f value should be broken in favour of nodes with lower h values. The algorithm terminates when a goal is chosen for expansion.

A* algorithm guides an optimal path to a goal if the heuristic function h(n) is admissible, meaning it never overestimates actual cost. For example, since airline distance never overestimates actual highway distance, and manhatten distance never overestimates actual moves in the gliding tile.

For Puzzle, A* algorithm, using these evaluation functions, can find optimal solutions to these problems. In addition, A* makes the most efficient use of the given heuristic function in the following sense: among all shortest-path algorithms using the given heuristic function h(n). A* algorithm expands the fewest number of nodes.

The main drawback of A* algorithm and indeed of any best-first search is its memory requirement. Since at least the entire open list must be saved, A* algorithm is severely space-limited in practice, and is no more practical than breadth-first search on current machines. For example, while it can be run successfully on the eight puzzle, it exhausts available memory in a matter of minutes on the fifteen puzzle.

3.2. 3 Iterative Deepening A* (IDA*) Search

Just as iterative deepening solved the space problem of breadth-first search, iterative deepening A* (IDA*) eliminates the memory constraints of A* search algorithm without sacrificing solution optimality. Each iteration of the algorithm is a depth-first search that keeps track of the cost, f(n) = g(n) + h(n), of each node generated. As soon as a node is generated whose cost exceeds a threshold for that iteration, its path is cut off, and the search backtracks before continuing. The cost threshold is initialized to the heuristic estimate of the initial state, and in each successive iteration is increased to the total cost of the lowest-cost node that was pruned during the previous iteration. The algorithm terminates when a goal state is reached whose total cost dees not exceed the current threshold.

Since Iterative Deepening A* performs a series of depth-first searches, its memory requirement is linear with respect to the maximum search depth. In addition, if the heuristic function is admissible, IDA* finds an optimal solution. Finally, by an argument similar to that presented for DFID, IDA* expands the same number of nodes, asymptotically,

as A* on a tree, provided that the number of nodes, asymptotically, as A* on a tree, provided that the number of nodes grows exponentially with solution cost. These costs, together with the optimality of A*, impy that IDA* is asymptotically optimal in time and space over all heuristic search algorithms that find optimal solutions on a tree. Additional benefits of IDA* are that it is much easier to implement, and often runs faster than A*, since it does not incur the overhead of managing the open and closed lists.

3.2.4 Depth-First Branch-And-Bound Search

For many problems, the maximum search depth is known in advance or the search is finite. For example, consider the traveling salesman problem (TSP) of visiting each of the given set of cities and returning to the starting city in a tour of shortest total distance. The most natural problem space for this problem consists of a tree where the root node represents the starting city, the nodes at level one represent all the cities that could be visited first, the nodes at level two represent all the cities that could be visited second, etc. In this tree, the maximum depth is the number of cities, and all candidate solution occurs at this depth. In such a space, a simple depth-first search guarantees finding an optional solution using space that is only linear with repsect to the number of cities.

The idea of depth-first branch-and-bound (DFBnB) Search is to make this search more efficient by keeping track of the lowest-cost solution found so far. Since the cost of a partial tour is the sum of the costs of the edges traveled so far, whenever a partial tour is found whose cost equals or exceeds the cost of the best complete tour found so far, the branch representing the partial tour can be pruned, since all its descendents must have equal or greater cost. Whenever a low-cost complete tour is found, the cost of the best tour is updated to this low cost. In addition, an admissible heuristic function such as the cost of the minimum spanning tree of the remaining unvisited cities, can be added to the cost so far of a partial tour to increase the amount of prunin. Finally, by carefully ordering the children of a given node from smallest to largest estimated total cost, a lower-cost solution can be found more quickly, further improving the pruning efficiency.

Interestingly, IDA* and DFBnB exhibit complementary behavior. Both are guaranteed to return an optimal solution cost, and increase in each iteration until it reaches the optimal cost. In DFBnB, the cost of the best solution found so far is always an upper bound on the optimal solution cost and decreases until it reaches the optimal cost.

While IDA* never expands any nodes whose cost exceeds the optimal cost, its overhead cosists of expanding some nodes more than once. While DFBnB never expands any node more than once its overhead consists of expanding some nodes whose cost exceed the optimal cost. For problems whose search trees are of bounded depth, or for which it is easy to construct a good solution such as the TSP, DFBnB is usually the algorithm of choice for finding an optimal solution. For problems with infinite search trees or for which it is difficult to construct a low-cost solution, such as the sliding-tile puzzles or Rubik's Cube, IDA* is usually the best choice.

3.2.5 Heuristic Path Search Algorithm

Since the complexity of finding optimal solutions to these problems is generally exponential in practice, in order to solve significantly larger problems, the optimality requirement must be released. An early approach to this problem was the heuristic path algorithm (HPA).

Heuristic path algorithm is a best-first search algorithm, where the figure of merit of node n is $f(n) = (1-w)*g(n)+w*h(n)$. Varying w produces a range of algorithms from uniform-cost search (w=0) through A* (w=1/2) to pure heuristic search (w=1). Increasing w beyond ½ generally decreases the amount of computation while increasing the cost of the solution generated. The trade off is often quite favorable, with small increases in solution cost yielding huge savings in computation. Moreover, it shows that the solutions found by this algorithm are guaranteed to be no more than a factor of w/(1-w) greater than optimal, but often are significantly better.

3.2. 6 Hill Climbing

Hill climbing is a depth first search with a heuristic measurement that orders choices as nodes are expanded. The heuristic measurement is the estimated remaining distance to the goal. The effectiveness of hill climbing is completely dependent upon the accuracy of the heuristic measurement.

To conduct a hill climbing search:

Form a one-element queue consisting of a zero-length path that contains only the root node.
Repeat
Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
Reject all new paths with loops.
Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal.
Until the first path in the queue terminates at the goal node or the queue is empty
If the goal node is found, announce success, otherwise announce failure.

Winston (1992) lucidly explains the potential problems affecting hill climbing. They are all related to issue of local "vision" versus global vision of the search space. The foothills problem is particularly subject to local maxima where global ones are sought, while the plateau problem occurs when the heuristic measure does not hint towards any significant gradient of proximity to a goal. The ridge problem illustrates just what it's called: you may get the impression that the search is taking you closer to a goal state, when in fact you travelling along a ridge which prevents you from actually attaining your goal.

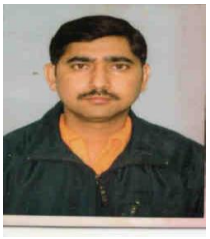
3.3 Complexity Of Finding Optimal olutions[9]:

The time complexity of a heuristic search algorithm depends on the accuracy of the heuristic function. For example, if the heuristic evaluation function is an exact estimator, then A* search algorithm runs in linear time, expanding only those nodes on an optimal solution path. Conversely, with a heuristic that returns zero everywhere, A* algorithm becomes uniform-cost search, which has exponential complexity.

In general, the time complexity of A* search and IDA* search is an exponential function of the error in the heuristic function. For example, if the heuristic has constant absolute error, meaning that it never underestimates by more than a constant amount regardless of the magnitude of the estimate, then the running time of A* is linear with respect to the solution cost. A more realistic assumption is constant relative error, which means that the error is a fixed percentage of the quantity being estimated. The base of the exponent, however, is smaller than the brute-force branching factor, reducing the asymptotic complexity and allowing larger problems to be solved. For example, using appropriate heuristic functions, IDA* can optimally solve random instance of the twenty-four puzzle and Rubik's Cube.

REFERENCES

1.McCulloch, W.S. and Pitts,W. "A Logical Calculus of the Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics, Volume 5, 1943, pp 115-137.
2.Rzevski, G., Skobelev, P., "Emergent Intelligence in Large Scale Multi-Agent Systems". International Journal of Education and Information Technology, Issue 2,Volume 1, 2007, pp 64-71.
3.Rzevski, G (ed), "Mechatronics: Designing Intelligent Machines", Butterworth Heinemann, 1995.
4. Artificial Intelligence: Search Methods by D. Kopec and T.A. Marsland
5. Feigenbaum, E , Feldman, J., Computers and Thought New York: McGraw-Hill, 1963.
6. Amarel, S. On representation of problems of reasoning about actions. Machine Intelligence, 1968, No 3, pp131-171.
7. Lenat, D. (1983) Theory formation by heuristic search. In: Search and Heuristics, J. Pearl (Ed.) New York: , North Holland, 1983.
8. Romanycia, M, Pelletier, F. What is heuristic? Computer Intelligence, 1985, No. 1, pp. 24-36.
9. Heuristic Search Methods for Combinatorial Programming Problems, Jacques A. Ferland,and Daniel Costa,March 2001.



Author

Ajay Kumar Gaur received the M. Sc(Physics, Computer Science) and M.Tech degrees in computer science and Engineering from Rohilkhand University and AAIDU Allahabad, respectively. Presently Pursuing Ph.D from Sighania University and working as a Assistant professor in KIT Kanpur.