Module-02 Informed Search Strategies

Greedy best-first search

- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search.

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$. Were, h(n)= estimated cost from node n to the goal.

Here we have **line distance** heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest.

For example, $h_{SLD}(In(Arad))=366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Figure 3.23 shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.

The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara.

The next node to be expanded will be Fagaras because it is closest.

Fagaras in turn generates Bucharest, which is the goal.

For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.

It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can.

Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search.

EX: Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end. The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras.

The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop. (The graph search version *is* complete in finite spaces, but not in infinite ones.)

The worst-case time and space complexity for the tree version is $O(b_m)$, where$m$is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

The greedy best first algorithm is implemented by the priority queue.

Time Complexity: The worst-case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst-case space complexity of Greedy best first search is O(bm).

Where, m is the maximum depth of the search space.

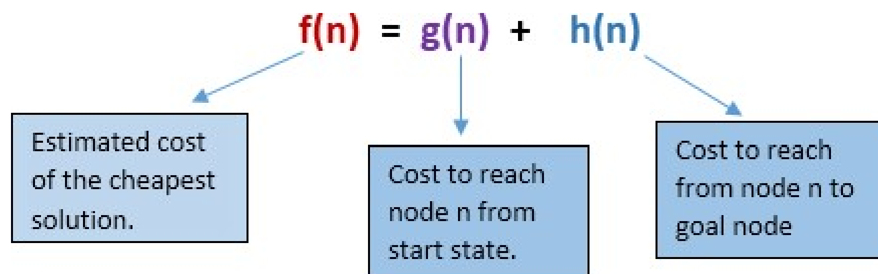
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

A*search: Minimizing the total estimated solution cost

- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence, we can combine both costs as following, and this sum is called as a **fitness number.**



Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n. Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$.
A∗ search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses $g + h$ instead of g.

Conditions for optimality: Admissibility and consistency
The first condition we require for optimality is that $h(n$ ADMISSIBLE $)$ be an **admissible heuristic**.
An admissible heuristic is one that *never overestimates* the cost to reach the goal.
Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.

An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate

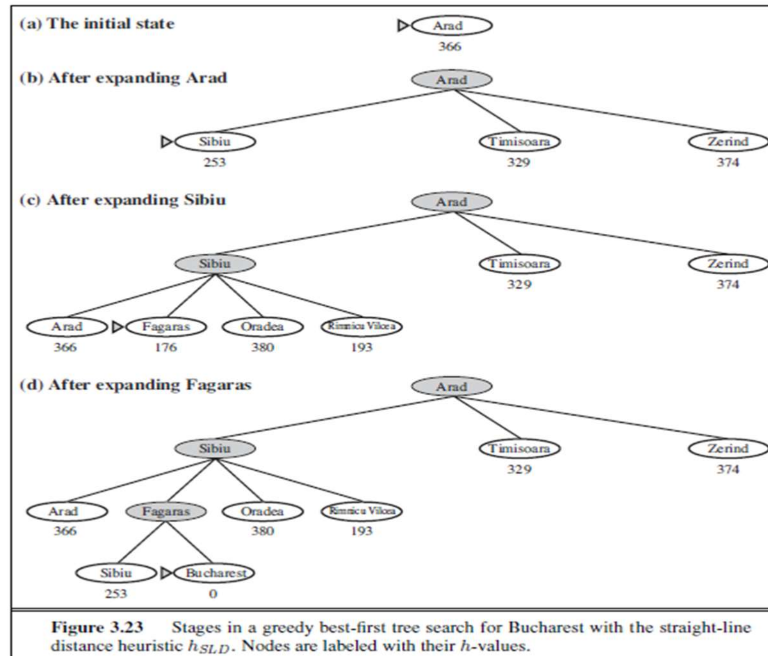


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

In below Figure 3.24, we show the progress of an A∗ tree search for Bucharest. The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 3.22. Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A∗ to graph search.

A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

$$h(n) \leq c(n, a, n') + h(n') .$$

This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides.

Here, the triangle is formed by n, n', and the goal G_n closest to n. For an admissible heuristic, the inequality makes perfect sense: if there were a route from n to G_n via n' that was cheaper than $h(n)$, that would violate the property that $h(n)$ is a lower bound on the cost to reach G_n. It is fairly easy to show that every consistent heuristic is also admissible. Consider, for example, h_{SLD}. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance and that the straight-line distance between n and $n_$ is no greater than $c(n, a, n_)$.Hence, h_{SLD} is a consistent heuristic.

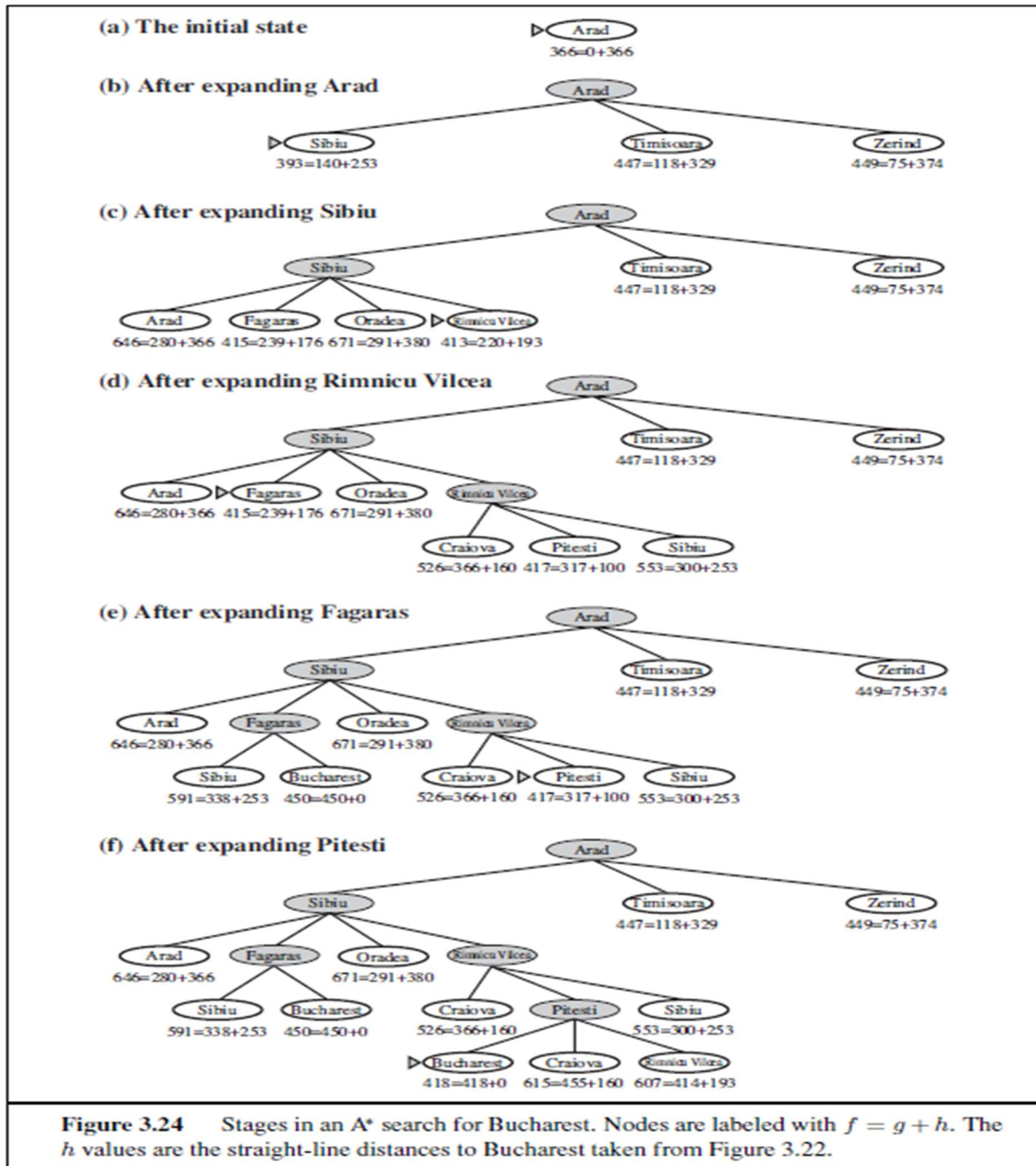


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Optimality of A*

A∗ has the following properties: the tree-search version of A∗ is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent.

The argument essentially mirrors the argument for the optimality of uniform-cost search, with g replaced by f—just as in the A∗ algorithm itself.

The first step is to establish the following: if h(n) is consistent, then the values of f(n) along any path are nondecreasing. The proof follows directly from the definition of consistency.

Suppose n is a successor of n; then g(n') = g(n) + c(n, a, n') for some action a, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) .$$

The next step is to prove that whenever A∗ selects a node n for expansion, the optimal path to that node has been found. Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n, by the graph separation property of Figure 3.9; because f is nondecreasing along any path, n would have lower f-cost than n and would have been selected first.

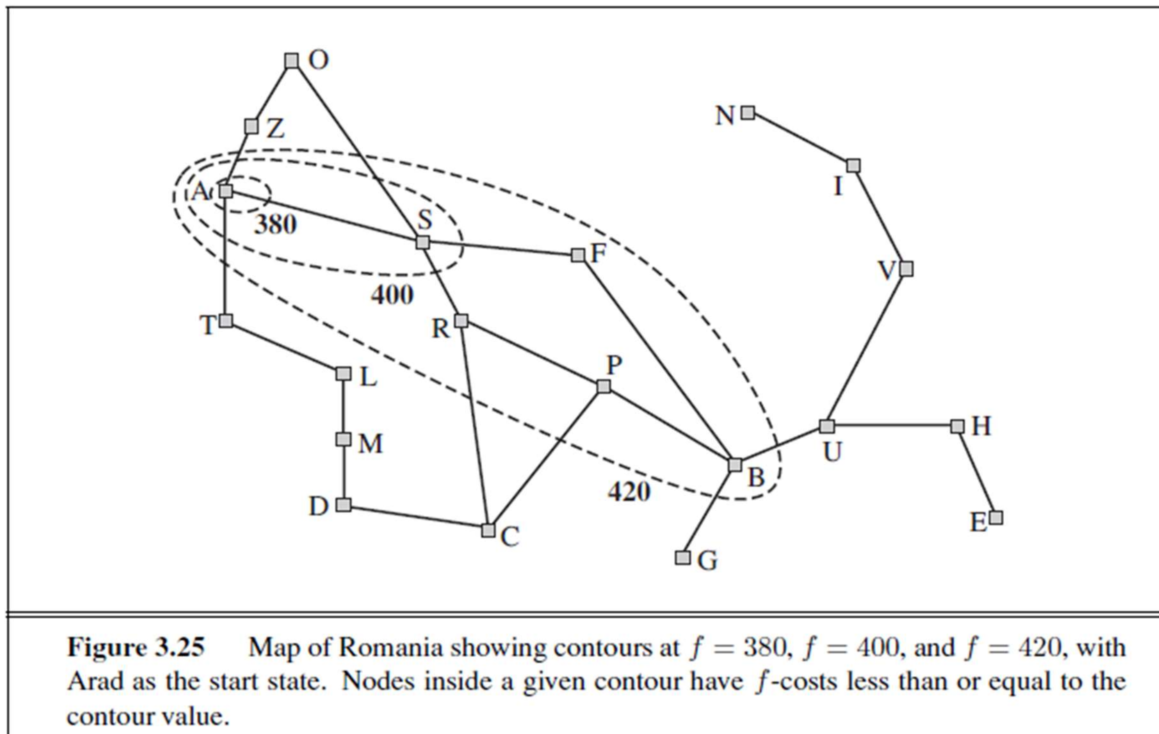


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

From the two preceding observations, it follows that the sequence of nodes expanded by A∗ using GRAPH-SEARCH is in nondecreasing order of f(n). Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h = 0) and all later goal nodes will be at least as expensive.

The fact that f-costs are nondecreasing along any path also means that we can draw contours in the state space, just like the contours in a topographic map. Figure 3.25 shows an example. Inside the contour labeled 400, all nodes have f(n) less than or equal to 400, and so on. Then, because A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

With uniform-cost search (A∗ search using h(n)=0), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path.

If C∗ is the cost of the optimal solution path, then we can say the following:

• A∗ expands all nodes with f(n) < C∗.

• A∗ might then expand some of the nodes right on the "goal contour" (where f(n) = C∗) before selecting a goal node. Completeness requires that there be only finitely many nodes with cost less than or equal to C∗, a condition that is true if all step costs exceed some finite and if b is finite. Notice that A∗ expands no nodes with f(n) > C∗—for example, Timisoara is not expanded in Figure 3.24 even though it is a child of the root. We say that the subtree below

Timisoara is **pruned**; because hSLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI. One final observation is that among optimal algorithms of this type— algorithms that extend search paths from the root and use the same heuristic information—A∗ is optimally efficient for any given consistent heuristic. That is, no other optimal algorithm is guaran- teed to expand fewer nodes than A∗ (except possibly through tie-breaking among nodes with f(n) = C∗). This is because any algorithm that does not expand all nodes with f(n) < C∗ runs the risk of missing the optimal solution.

That A∗ search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A∗ is the answer to all our searching needs. The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution. The details of the analysis are beyond the scope of this book, but the basic results are as follows. For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the the absolute error or the relative error of the heuristic. The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and the relative error is defined as $\varepsilon \equiv (h^* - h)/h^*$.

The complexity results depend very strongly on the assumptions made about the state space. The simplest model studied is a state space that has a single goal and is essentially a tree with reversible actions. (The 8-puzzle satisfies the first and third of these assumptions.) In this case, the time complexity of A∗ is exponential in the maximum absolute error, that is, O(b^Δ). For constant step costs, we can write this as O(b^d), where d is the solution depth. For almost all heuristics in practical

use, the absolute error is at least proportional to the path cost $h*$, so is constant or growing and the time complexity is exponential in d. We can also see the effect of a more accurate heuristic: $O(b\hat{}d) = O((b\hat{})\hat{}d)$, so the effective branching factor (defined more formally in the next section) is b^. When the state space has many goal states—particularly nearoptimal goal states—the search process can be led astray from the optimal path and there is an extra cost proportional to the number of goals whose cost is within a factor of the optimal cost. Finally, in the general case of a graph, the situation is even worse. There can be exponentially many states with $f(n) < C*$ even if the absolute error is bounded by a constant. For example, consider a version of the vacuum world where the agent can clean up any square for unit cost without even having to visit it: in that case, squares can be cleaned in any order. With N initially dirty squares, there are 2N states where some subset has been cleaned and all of them are on an optimal solution path—and hence satisfy $f(n) < C*$—even if the heuristic has an error of 1. The complexity of A∗ often makes it impractical to insist on finding an optimal solution. One can use variants of A∗ that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search. In Section 3.6, we look at the question of designing good heuristics. Computation time is not, however, A∗'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A∗ usually runs out of space long before it runs out of time. For this reason, A∗ is not practical for many large-scale problems. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time. We discuss these next.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

Figure 3.26 The algorithm for recursive best-first search.

3.6 HEURISTIC FUNCTIONS

8-puzzle was one of the earliest heuristic search problems.

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means that an exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states. A graph search would cut this down by a factor of about 170,000 because only 9!/2 = 181, 440 distinct states are reachable. (See Exercise 3.4.) This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13}, so the next order of business is to find a good heuristic function. If we want to find the shortest solutions by using A∗, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

• $h1$ = the number of misplaced tiles. For Figure 3.28, all of the eight tiles are out of position, so the start state would have $h1 = 8$. $h1$ is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

• $h2$ = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. $h2$ is also admissible because all any

move can do is move one tile one step MANHATTAN DISTANCE closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

$h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$.

As expected, neither of these overestimates the true solution cost, which is 26.

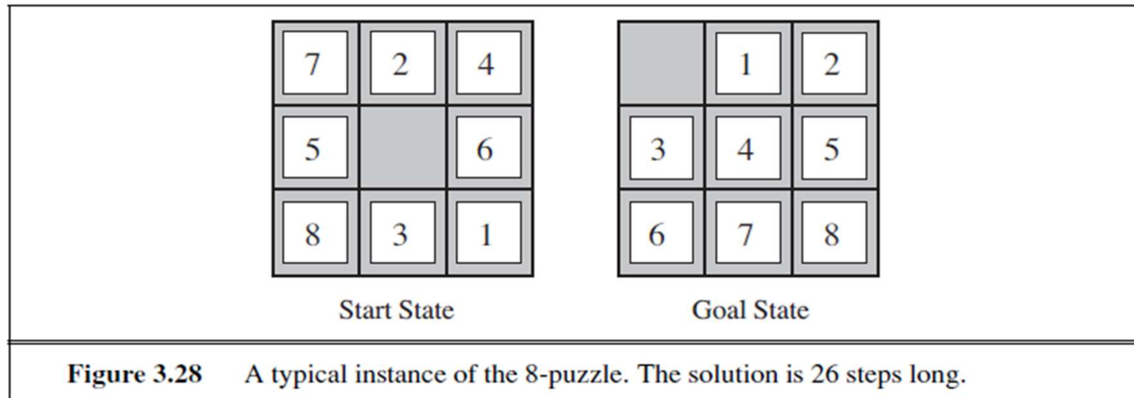


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

3.6.1 The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristic is the effective branching factor b∗. If the total number of nodes generated by A∗ for a particular problem is N and the solution depth is d, then b∗ is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes. Thus, $N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$.

For example, if A∗ finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. (The existence of an effective branching factor follows from the result, mentioned earlier, that the number of nodes expanded by A∗ grows exponentially with solution depth.) Therefore, experimental measurements of b∗ on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b∗ close to 1, allowing fairly large problems to be solved at reasonable computational cost

To test the heuristic functions h1 and h2, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with iterative deepening search and with A∗ tree search using both h1 and h2. Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor. The results suggest that h2 is better than h1, and is far better than using iterative deepening search. Even for small problems with d = 12, A∗ with h2 is 50,000 times more efficient than uninformed iterative deepening search.

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

One might ask whether h2 is always better than h1. The answer is "Essentially, yes." It is easy to see from the definitions of the two heuristics that, for any node n, h2(n) ≥ h1(n). We thus say that h2 dominates h1. Domination translates directly into efficiency: A∗ using h2 will never expand more nodes than A∗ using h1 (except possibly for some nodes with f(n) = C∗). The argument is simple. Recall the observation on page 97 that every node with f(n) < C∗ will surely be expanded. This is the same as saying that every node with h(n) < C∗ − g(n) will surely be expanded. But because h2 is at least as big as h1 for all nodes, every node that is surely expanded by A∗ search with h2 will also surely be expanded with h1, and h1 might cause other nodes to be expanded as well. Hence, it is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

3.6.2 Generating admissible heuristics from relaxed problems

We have seen that both h1 (misplaced tiles) and h2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h2 is better. How might one have come up with h2? Is it possible for a computer to invent such a heuristic mechanically? h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h1 would give the exact number of steps in the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a

relaxed problem. The state-space graph of the relaxed problem is a super graph of the original state space because the removal of restrictions creates added edges in the graph. Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have better solutions if the added edges provide short cuts. Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem. Furthermore, because the derived heuristic is an exact cost for the

relaxed problem, it must obey the triangle inequality and is therefore consistent (see page 95). If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.11 For example, if the 8-puzzle actions are described as A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank, we can generate three relaxed problems by removing one or both of the conditions:
(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B.

From (a), we can derive h2 (Manhattan distance). The reasoning is that h2 would be the proper score if we moved each tile in turn to its destination. The heuristic derived from (b) is discussed in Exercise 3.31. From (c), we can derive h1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step. Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially without search, because the relaxed rules allow the problem to be decomposed into eight independent sub problems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.12 A program called ABSOLVER can generate heuristics automatically from problem definitions, using the “relaxed problem” method and various other techniques (Prieditis, 1993). ABSOLVER generated a new heuristic for the 8-puzzle that was better than any pre-existing heuristic and found the first useful heuristic for the famous Rubik’s Cube puzzle. One problem with generating new heuristic functions is that one often fails to get a single “clearly best” heuristic. If a collection of admissible heuristics h1 ...hm is available for a problem and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining $h(n) = \max\{h1(n),...,hm(n)\}$.

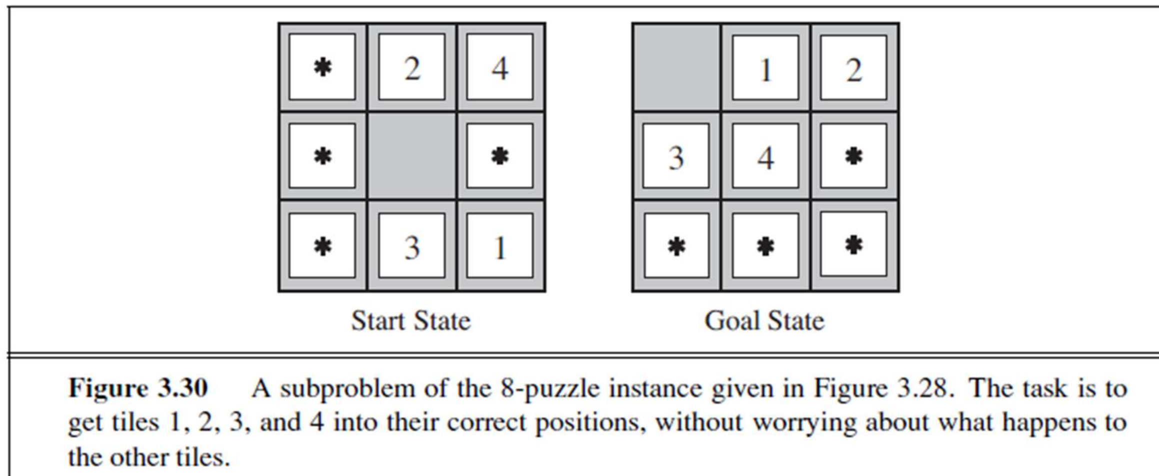


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

3.6.3 Generating admissible heuristics from subproblems: Pattern databases

Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem. For example, Figure 3.30 shows a subproblem of the 8-puzzle instance in Figure 3.28. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be more accurate than Manhattan distance in some cases. The idea behind pattern databases is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank. (The locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward the cost.) Then we compute an admissible heuristic hDB for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching back13 from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances. The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000. One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be added, since the two subproblems seem not to overlap. Would this still give an admissible

heuristic? The answer is no, because the solutions of the 1-23-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-78, and vice versa. But what if we don't count those moves? That is, we record not the total cost of solving the 1-2- 3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind disjoint pattern databases. With such databases, it is possible to solve random 15-puzzles in a few milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with the use of Manhattan distance. For 24-puzzles, a speedup of roughly a factor of a million can be obtained. Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time. For a problem such as Rubik's Cube, this kind of subdivision is difficult because each move affects 8 or 9 of the 26 cubies. More general ways of defining additive, admissible heuristics have been proposed that do apply to Rubik's cube (Yang et al., 2008), but they have not yielded a heuristic better than the best nonadditive heuristic for the problem.

3.6.4 Learning heuristics from experience

A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the state at node n. How could an agent construct such a function? One solution was given in the preceding sections—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. "Experience" here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function h(n) that can (with luck) predict solution costs for other states that arise during search. Techniques for doing just this using neural nets, decision trees, and other methods are demonstrated in Chapter 18. (The reinforcement learning methods described in Chapter 21 are also applicable.) Inductive learning methods work best when supplied with features of a state that are relevant to predicting the state's value, rather than with just the raw state description. For example, the feature "number of misplaced tiles" might be helpful in predicting the actual distance of a state from the goal. Let's call this feature x1(n). We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when x1(n) is 5, the average solution cost is around 14, and so on. Given these data, the value of x1 can be used to predict h(n). Of course, we can use several features. A second feature x2(n)

might be "number of pairs of adjacent tiles that are not adjacent in the goal state." How should $x1(n)$ and $x2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

$h(n) = c1x1(n) + c2x2(n)$.

The constants $c1$ and $c2$ are adjusted to give the best fit to the actual data on solution costs. One expects both $c1$ and $c2$ to be positive because misplaced tiles and incorrect adjacent pairs make the problem harder to solve. Notice that this heuristic does satisfy the condition that $h(n)=0$ for goal states, but it is not necessarily admissible or consistent.

.

Chapter-02: Introduction to Machine Learning

What is Machine Learning

In the real world, we are surrounded by humans who can learn everything from their experiences with their learning capability, and we have computers or machines which work on our instructions.

But can a machine also learn from experiences or past data like a human does? So here comes the role of **Machine Learning.**

Introduction to Machine Learning

A subset of artificial intelligence known as machine learning focuses primarily on the creation of algorithms that enable a computer to independently learn from data and previous experiences. Arthur Samuel first used the term "machine learning" in 1959.

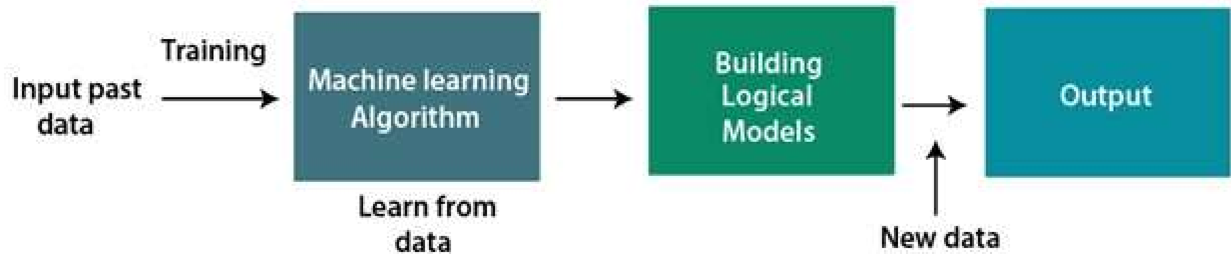
It could be summarized as follows:

- Without being explicitly programmed, machine learning enables a machine to automatically learn from data, improve performance from experiences, and predict things.
- Machine learning algorithms create a mathematical model that, without being explicitly programmed, aids in making predictions or decisions with the assistance of sample historical data, or training data.
- For the purpose of developing predictive models, machine learning brings together statistics and computer science.
- Algorithms that learn from historical data are either constructed or utilized in machine learning. The performance will rise in proportion to the quantity of information we provide.

A machine can learn if it can gain more data to improve its performance.

How does Machine Learning work

- A machine learning system builds prediction models, learns from previous data, and predicts the output of new data whenever it receives it.
- The amount of data helps to build a better model that accurately predicts the output, which in turn affects the accuracy of the predicted output.
- Let's say we have a complex problem in which we need to make predictions.
- Instead of writing code, we just need to feed the data to generic algorithms, which build the logic based on the data and predict the output.
- Our perspective on the issue has changed as a result of machine learning.
- The Machine Learning algorithm's operation is depicted in the following block diagram:



Features of Machine Learning:

- Machine learning uses data to detect various patterns in a given dataset.
- It can learn from past data and improve automatically.
- It is a data-driven technology.
- Machine learning is much similar to data mining as it also deals with the huge amount of the data.

Need for Machine Learning

- The demand for machine learning is steadily rising.
- Because it is able to perform tasks that are too complex for a person to directly implement, machine learning is required.
- Humans are constrained by our inability to manually access vast amounts of data; as a result, we require computer systems, which is where machine learning comes in to simplify our lives.
- By providing them with a large amount of data and allowing them to automatically explore the data, build models, and predict the required output, we can train machine learning algorithms.
- The cost function can be used to determine the amount of data and the machine learning algorithm's performance.
- We can save both time and money by using machine learning.

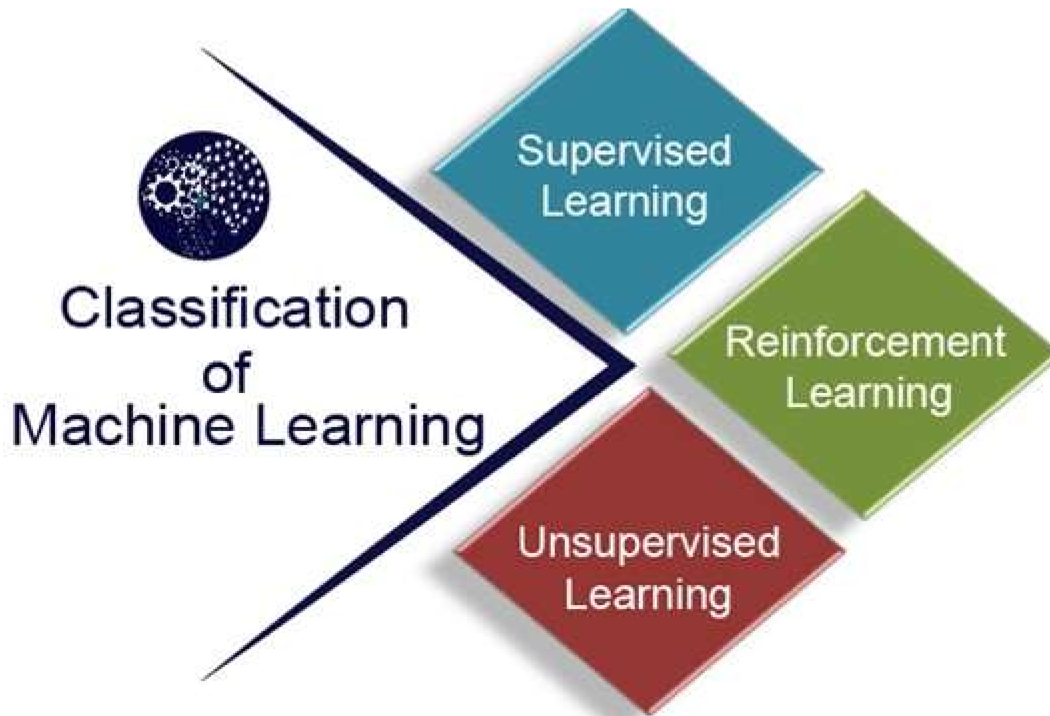
Following are some key points which show the importance of Machine Learning:

- Rapid increment in the production of data
- Solving complex problems, which are difficult for a human
- Decision making in various sector including finance
- Finding hidden patterns and extracting useful information from data.

Classification of Machine Learning

At a broad level, machine learning can be classified into three types:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning



1) Supervised Learning

In supervised learning, sample labeled data are provided to the machine learning system for training, and the system then predicts the output based on the training data.

- The system uses labeled data to build a model that understands the datasets and learns about each one.
- After the training and processing are done, we test the model with sample data to see if it can accurately predict the output.
- The mapping of the input data to the output data is the objective of supervised learning. The managed learning depends on oversight, and it is equivalent to when an understudy learns things in the management of the educator.
- Spam filtering is an example of supervised learning.

Supervised learning can be grouped further in two categories of algorithms:

1. **Classification**
2. **Regression**

2) Unsupervised Learning

- Unsupervised learning is a learning method in which a machine learns without any supervision.
- The training is provided to the machine with the set of data that has not been labeled, classified, or categorized, and the algorithm needs to act on that data without any supervision.
- The goal of unsupervised learning is to restructure the input data into new features or a group of objects with similar patterns.
- In unsupervised learning, we don't have a predetermined result. The machine tries to find useful insights from the huge amount of data.

It can be further classifieds into two categories of algorithms:

1. **Clustering**
2. **Association**

3) Reinforcement Learning

- Reinforcement learning is a feedback-based learning method, in which a learning agent gets a reward for each right action and gets a penalty for each wrong action.
- The agent learns automatically with these feedbacks and improves its performance.
- In reinforcement learning, the agent interacts with the environment and explores it.
- The goal of an agent is to get the most reward points, and hence, it improves its performance.
- The robotic dog, which automatically learns the movement of his arms, is an example of Reinforcement learning.

MACHINE LEARNING EXPLAINED

- Machine learning is an important sub-branch of Artificial Intelligence (AI).
- A frequently quoted definition of machine learning was by Arthur Samuel, one of the pioneers of Artificial Intelligence.
- He stated that "Machine learning is the field of study that gives the computers ability to learn without being explicitly programmed."
- The key to this definition is that the systems should learn by itself without explicit programming.
- How is it possible? It is widely known that to perform a computation, one needs to write programs that teach the computers how to do that computation.
- In conventional programming, after understanding the problem, a detailed design of the program such as a flowchart or an algorithm needs to be created and converted into programs using a suitable programming language.
- This approach could be difficult for many real-world problems such as puzzles, games, and complex image recognition applications.
- Initially, artificial intelligence aims to understand these problems and develop general purpose rules manually.

- Then, these rules are formulated into logic and implemented in a program to create intelligent systems.
- This idea of developing intelligent systems by using logic and reasoning by converting an expert's knowledge into a set of rules and programs is called an expert system.
- An expert system like MYCIN was designed for medical diagnosis after converting the expert knowledge of many doctors into a system.
- However, this approach did not progress much as programs lacked real intelligence.
- The word MYCIN is derived from the fact that most of the antibiotics' names end with 'mycin'.
- The above approach was impractical in many domains as programs still depended on human expertise and hence did not truly exhibit intelligence.
- Then, the momentum shifted to machine learning in the form of data driven systems. The focus of AI is to develop intelligent systems by using data-driven approach, where data is used as an input to develop intelligent models.
- The models can then be used to predict new inputs.
- Thus, the aim of machine learning is to learn a model or set of rules from the given dataset automatically so that it can predict the unknown data correctly.
- As humans take decisions based on an experience, computers make models based on extracted patterns in the input data and then use these data-filled models for prediction and to take decisions. For computers, the learnt model is equivalent to human experience. This is shown in Figure 1.2.

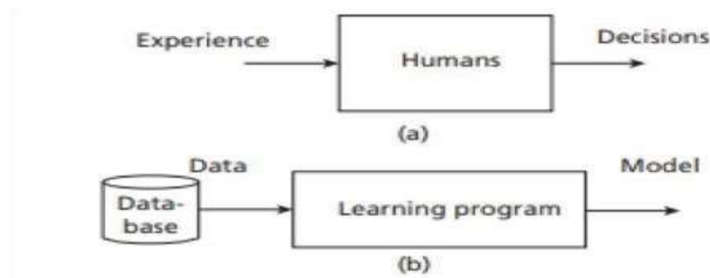


Figure 1.2: (a) A Learning System for Humans (b) A Learning System for Machine Learning

MACHINE LEARNING IN RELATION TO OTHER FIELDS

Machine learning uses the concepts of Artificial Intelligence, Data Science, and Statistics primarily. It is the resultant of combined ideas of diverse fields.

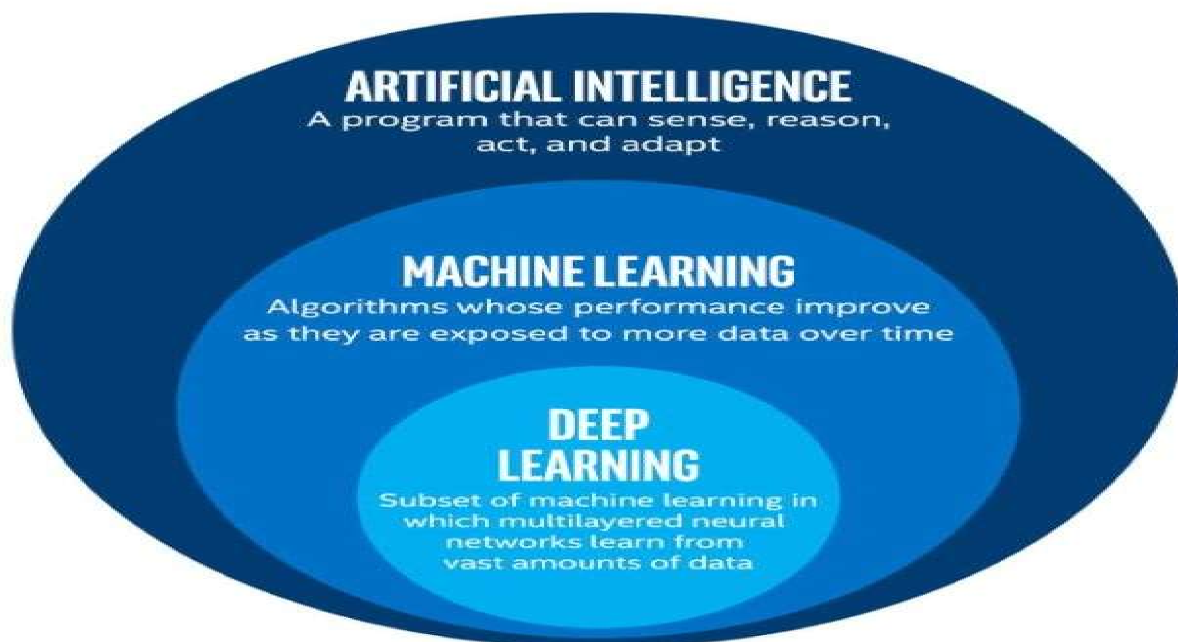
Machine Learning and Artificial Intelligence

Machine learning is an important branch of AI, which is a much broader subject. The aim of AI is to develop intelligent agents. An agent can be a robot, humans, or any autonomous systems. Initially, the idea of AI was ambitious, that is, to develop intelligent systems like human beings. The focus was on logic and logical inferences. It had seen many ups and downs. These down periods were called AI winters.

The resurgence in AI happened due to development of data driven systems. The aim is to find relations and regularities present in the data. Machine learning is the subbranch of AI, whose aim is to extract the patterns for prediction. It is a broad field that includes learning from examples and other areas like reinforcement learning.

The relationship of AI and machine learning is shown in Figure 1.3. The model can take an unknown instance and generate results.

Figure 1.3: Relationship of AI with Machine Learning



- Deep learning is a subbranch of machine learning.
- In deep learning, the models are constructedusing neural network technology.
- Neural networks are based on the human neuron models.
- Many neurons form a network connected with the activation functions that trigger further neurons to perform tasks.

Topic-02: Understanding Data

WHAT IS DATA?

- All facts are data. In computer systems, bits encode facts present in numbers, text, images, audio, and video.
- Data can be directly human interpretable (such as numbers or texts) or diffused data such as images or video that can be interpreted only by a computer.
- Data is available in different data sources like flat files, databases, or data warehouses. It can either be an operational data or a non-operational data.
- Operational data is the one that is encountered in normal business procedures and processes. For example, daily sales data is operational data, on the other hand, non-operational data is the kind of data that is used for decision making.
- Data by itself is meaningless. It has to be processed to generate any information. A string of bytes is meaningless. Only when a label is attached like height of students of a class, the data becomes meaningful.
- Processed data is called information that includes patterns, associations, or relationships among data. For example, sales data can be analyzed to extract information like which product was sold larger in the last quarter of the year.

Elements of Big Data

Data whose volume is less and can be stored and processed by a small-scale computer is called ‘small data’. These

data are collected from several sources, and integrated and processed by a small-scale computer. Big data, on the other hand, is a larger data whose volume is much larger than ‘small data’ and is characterized as follows:

1. Volume – Since there is a reduction in the cost of storing devices, there has been a tremendous growth of data. Small traditional data is measured in terms of gigabytes (GB) and terabytes (TB), but Big Data is measured in terms of petabytes (PB) and exabytes (EB). One exabyte is 1 million terabytes.
2. Velocity – The fast arrival speed of data and its increase in data volume is noted as velocity. The availability of IoT devices and Internet power ensures that the data is arriving at a faster rate. Velocity helps to understand the relative growth of big data and its accessibility by users, systems and applications.
3. Variety – The variety of Big Data includes:

- Form – There are many forms of data. Data types range from text, graph, audio, video, to maps. There can be composite data too, where one media can have many other sources of data, for example, a video can have an audio song.
- Function – These are data from various sources like human conversations, transaction records, and old archive data.
- Source of data – This is the third aspect of variety. There are many sources of data. Broadly, the data source can be classified as open/public data, social media data and multimodal data.
- Some of the other forms of Vs that are often quoted in the literature as characteristics of big data are:

4. Veracity of data – Veracity of data deals with aspects like conformity to the facts, truthfulness,

believability, and confidence in data. There may be many sources of error such as technical errors,

typographical errors, and human errors. So, veracity is one of the most important aspects of data.

5. Validity – Validity is the accuracy of the data for taking decisions or for any other goals that are needed

by the given problem.

6. Value – Value is the characteristic of big data that indicates the value of the information that is extracted

- from the data and its influence on the decisions that are taken based on it. Thus, these 6 Vs are helpful to characterize the big data. The data quality of the numeric attributes is determined by factors like precision, bias, and accuracy.
- Precision is defined as the closeness of repeated measurements. Often, standard deviation is used to measure the precision.
- Bias is a systematic result due to erroneous assumptions of the algorithms or procedures. Accuracy is the degree of measurement of errors that refers to the closeness of measurements to the true value of the quantity. Normally, the significant digits used to store and manipulate indicate the accuracy of the measurement.

Types of Data

In Big Data, there are three kinds of data. They are structured data, unstructured data, and semi structured data.

Structured Data

In structured data, data is stored in an organized manner such as a database where it is available in the form of a table. The data can also be retrieved in an organized manner using tools like SQL. The structured data

frequently encountered in machine learning are listed below:

Record Data A dataset is a collection of measurements taken from a process. We have a collection of objects in a

dataset and each object has a set of measurements. The measurements can be arranged in the form of a matrix.

Rows in the matrix represent an object and can be called as entities, cases, or records. The columns of the dataset

are called attributes, features, or fields. The table is filled with observed data. Also, it is better to note the general

jargons that are associated with the dataset. Label is the term that is used to describe the individual observations.

Data Matrix It is a variation of the record type because it consists of numeric attributes. The standard matrix

operations can be applied on these data. The data is thought of as points or vectors in the multidimensional space

where every attribute is a dimension describing the object.

Graph Data It involves the relationships among objects. For example, a web page can refer to another web page.

This can be modeled as a graph. The modes are web pages and the hyperlink is an edge that connects the nodes.

Ordered Data Ordered data objects involve attributes that have an implicit order among them. The examples

of ordered data are:

- Temporal data – It is the data whose attributes are associated with time. For example, the customer purchasing patterns during festival time is sequential data. Time series data is a special type of sequence data where the data is a series of measurements over time.
- Sequence data – It is like sequential data but does not have time stamps. This data involves the sequence of words or letters. For example, DNA data is a sequence of four characters – A T G C.
- Spatial data – It has attributes such as positions or areas. For example, maps are spatial data where the points are related by location.

Unstructured Data

Unstructured data includes video, image, and audio. It also includes textual documents, programs, and blog data. It is estimated that 80% of the data are unstructured data.

Semi-Structured Data

Semi-structured data are partially structured and partially unstructured. These include data like XML/JSON data, RSS feeds, and hierarchical data.

Data Storage and Representation

Once the dataset is assembled, it must be stored in a structure that is suitable for data analysis. The goal of data storage management is to make data available for analysis. There are different approaches to organize and

manage data in storage files and systems from flat file to data warehouses. Some of them are listed below:

Flat Files These are the simplest and most commonly available data source. It is also the cheapest way of organizing the data. These flat files are the files where data is stored in plain ASCII or EBCDIC format. Minor changes of data in flat files affect the results of the data mining algorithms.

Hence, flat file is suitable only for storing small dataset and not desirable if the dataset becomes larger.

Some of the popular spreadsheet formats are listed below:

• CSV files – CSV stands for comma-separated value files where the values are separated by commas. These

are used by spreadsheet and database applications. The first row may have attributes and the rest of the rows

represent the data.

• TSV files – TSV stands for Tab separated values files where values are separated by Tab. Both CSV and

TSV files are generic in nature and can be shared. There are many tools like Google Sheets and Microsoft Excel to process these files.

BIG DATA ANALYTICS AND TYPES OF ANALYTICS

- The primary aim of data analysis is to assist business organizations to take decisions. For example, a business organization may want to know which is the fastest selling product, in order for them to market activities.
- Data analysis is an activity that takes the data and generates useful information and insights for assisting the organizations.
- Data analysis and data analytics are terms that are used interchangeably to refer to the same concept.
- However, there is a subtle difference. Data analytics is a general term and data analysis is a part of it. Data analytics refers to the process of data collection, preprocessing and analysis. It deals with the complete cycle of data management. Data analysis is just analysis and is a part of data analytics. It takes historical data and does the analysis.

Data analytics, instead, concentrates more on future and helps in prediction.

There are four types of data analytics:

1. Descriptive analytics
2. Diagnostic analytics
3. Predictive analytics
4. Prescriptive analytics

BIG DATA ANALYSIS FRAMEWORK

For performing data analytics, many frameworks are proposed. All proposed analytics frameworks have some common factors. Big data framework is a layered architecture. Such an architecture has many advantages such as genericness.

A 4-layer architecture has the following layers:

1. Date connection layer
2. Data management layer
3. Data analytics later

4. Presentation layer

- Data Connection Layer It has data ingestion mechanisms and data connectors. Data ingestion means taking raw data and importing it into appropriate data structures. It performs the tasks of ETL process. By ETL, it means extract, transform and load operations.

  Data Management Layer It performs preprocessing of data. The purpose of this layer is to allow parallel execution of queries, and read, write and data management tasks. There may be many schemes that can be implemented by this layer such as data-in-place, where the data is not moved at all, or constructing data repositories such as data warehouses and pull data on-demand mechanisms.
- Data Analytic Layer It has many functionalities such as statistical tests, machine learning algorithms to understand, and construction of machine learning models. This layer implements many model validation mechanisms too.
- The processing is done as shown in Box 2.1.
- Presentation Layer It has mechanisms such as dashboards, and applications that display the results of analytical engines and machine learning algorithms.
- Thus, the Big Data processing cycle involves data management that consists of the following steps.

1. Data collection
2. Data preprocessing
3. Applications of machine learning algorithm
4. Interpretation of results and visualization of machine learning algorithm

- This is an iterative process and is carried out on a permanent basis to ensure that data is suitable for data mining.
- Application and interpretation of machine learning algorithms constitute the basis for the rest of the book. So, primarily, data collection and data preprocessing are covered as part of this chapter.

Data Collection

The first task of gathering datasets is the collection of data. It is often estimated that most of the time is spent for collection of good quality data. A good quality data yields a better result. It is often difficult to characterize a 'Good data'. 'Good data' is one that has the following properties:

1. Timeliness – The data should be relevant and not stale or obsolete data.

2. Relevancy – The data should be relevant and ready for the machine learning or data mining algorithms. All

the necessary information should be available and there should be no bias in the data.

3. Knowledge about the data – The data should be understandable and interpretable, and should be self-

sufficient for the required application as desired by the domain knowledge engineer.

Broadly, the data source can be classified as open/public data, social media data and multimodal data.

1. Open or public data source – It is a data source that does not have any stringent copyright rules or

restrictions. Its data can be primarily used for many purposes. Government census data are good

examples of open data:

• Digital libraries that have huge amount of text data as well as document images • Scientific domains

with a huge collection of experimental data like genomic data and biological data

• Healthcare systems that use extensive databases like patient databases, health insurance data, doctors'

information, and bioinformatics information

2. Social media – It is the data that is generated by various social media platforms like Twitter,

Facebook, YouTube, and Instagram. An enormous amount of data is generated by these platforms.

Data Preprocessing

In real world, the available data is ’dirty’. By this word ’dirty’, it means:

• Incomplete data • Inaccurate data

• Outlier data • Data with missing values

• Data with inconsistent values • Duplicate data

- Data preprocessing improves the quality of the data mining techniques. The raw data must be preprocessed to give accurate results. The process of detection and removal of errors in data is called data cleaning.
- Data wrangling means making the data processable for machine learning algorithms. Some of the data errors include human errors such as typographical errors or incorrect measurement and structural errors like improper data formats. Data errors can also arise from omission and duplication of attributes.
- Noise is a random component and involves distortion of a value or introduction of spurious objects. Often, the noise is used if the data is a spatial or temporal component. Certain deterministic distortions in the form of a streak are known as artifacts.
- Consider, for example, the following patient Table 2.1. The ‘bad’ or ‘dirty’ data can be observed in this table.

Table 2.1: Illustration of ‘Bad’ Data

Patient ID	Name	Age	Date of Birth (DoB)	Fever	Salary
1.	John	21		Low	–1500
2.	Andre	36		High	Yes
3.	David	5	10/10/1980	Low	" "
4.	Raju	136		High	Yes