

This is a study of the kinds of knowledge that are required for solving problems in the world.

- **Ontology**

Ontology is the study of the kinds of things that exist. In AI, the programs and sentences deal with various kinds of objects, and we study what these kinds are and what their basic properties are. Emphasis on ontology begins in the 1990s.

- **Heuristics**

A heuristic is a way of trying to discover something or an idea imbedded in a program. The term is used variously in AI. *Heuristic functions* are used in some approaches to search to measure how far a node in a search tree seems to be from a goal. *Heuristic predicates* that compare two nodes in a search tree to see if one is better than the other, i.e. constitutes an advance toward the goal, may be more useful.

- **Genetic Programming**

Genetic programming is a technique for getting programs to solve a task by mating random Lisp programs and selecting fittest in millions of generations.

Search and Control Strategies:

Problem solving is an important aspect of Artificial Intelligence. A problem can be considered to consist of a goal and a set of actions that can be taken to lead to the goal. At any given time, we consider the state of the search space to represent where we have reached as a result of the actions we have applied so far. For example, consider the problem of looking for a contact lens on a football field. The initial state is how we start out, which is to say we know that the lens is somewhere on the field, but we don't know where. If we use the representation where we examine the field in units of one square foot, then our first action might be to examine the square in the top-left corner of the field. If we do not find the lens there, we could consider the state now to be that we have examined the top-left square and have not found the lens. After a number of actions, the state might be that we have examined 500 squares, and we have now just found the lens in the last square we examined. This is a goal state because it satisfies the goal that we had of finding a contact lens.

Search is a method that can be used by computers to examine a problem space like this in order to find a goal. Often, we want to find the goal as quickly as possible or without using too many resources. A problem space can also be considered to be a search space

because in order to solve the problem, we will search the space for a goal state. We will continue to use the term search space to describe this concept. In this chapter, we will look at a number of methods for examining a search space. These methods are called search methods.

- The Importance of Search in AI
 - It has already become clear that many of the tasks underlying AI can be phrased in terms of a search for the solution to the problem at hand.
 - Many goal based agents are essentially problem solving agents which must decide what to do by searching for a sequence of actions that lead to their solutions.
 - For production systems, we have seen the need to search for a sequence of rule applications that lead to the required fact or action.
 - For neural network systems, we need to search for the set of connection weights that will result in the required input to output mapping.
- Which search algorithm one should use will generally depend on the problem domain? There are four important factors to consider:
 - Completeness – Is a solution guaranteed to be found if at least one solution exists?
 - Optimality – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?
 - Time Complexity – The upper bound on the time required to find a solution, as a function of the complexity of the problem.
 - Space Complexity – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.


Preliminary concepts

- Two varieties of **space-for-time** algorithms:
 - Input enhancement — preprocess the input (or its part) to store some info to be used later in solving the problem
 - Counting for sorting
 - String searching algorithms
 - Prestructuring — preprocess the input to make accessing its elements easier
 - Hashing

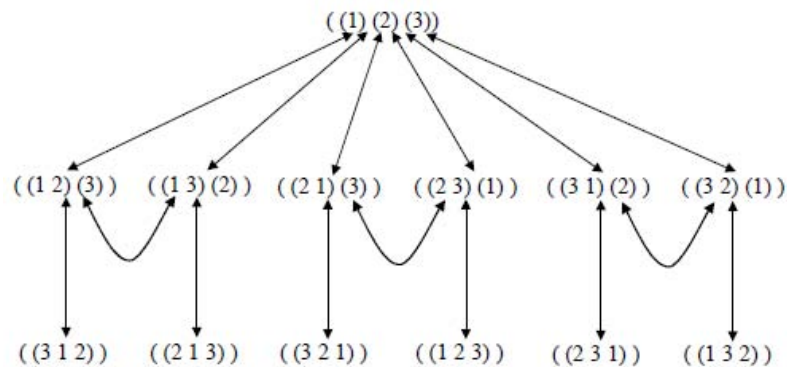
- Indexing schemes (e.g., B-trees)
- **State Space Representations:** The state space is simply the space of all possible states, or configurations, that our system may be in. Generally, of course, we prefer to work with some convenient representation of that search space.
- There are two components to the representation of state spaces:
 - Static States

e.g.  ((1 2) (3))

- Transitions between States

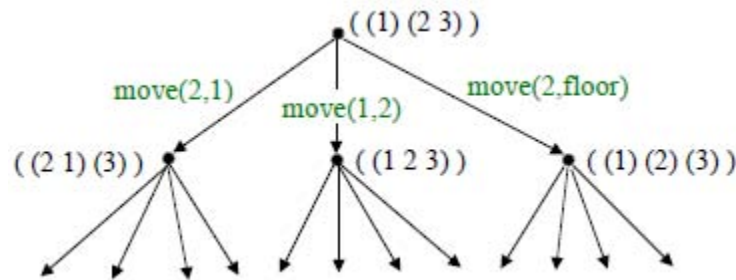
e.g.  move(1, 2)

- **State Space Graphs:** If the number of possible states of the system is small enough, we can represent all of them, along with the transitions between them, in a state space graph, e.g.



- **Routes through State Space:** Our general aim is to search for a route, or sequence of transitions, through the state space graph from our initial state to a goal state.
- Sometimes there will be more than one possible goal state. We define a goal test to determine if a goal state has been achieved.
- The solution can be represented as a sequence of link labels (or transitions) on the state space graph. Note that the labels depend on the direction moved along the link.
- Sometimes there may be more than one path to a goal state, and we may want to find the optimal (best possible) path. We can define link costs and path costs for measuring the cost of going along a particular path, e.g. the path cost may just equal the number of links, or could be the sum of individual link costs.

- For most realistic problems, the state space graph will be too large for us to hold all of it explicitly in memory at any one time.
- **Search Trees:** It is helpful to think of the search process as building up a search tree of routes through the state space graph. The root of the search tree is the search node corresponding to the initial state.
- The leaf nodes correspond either to states that have not yet been expanded, or to states that generated no further nodes when expanded.



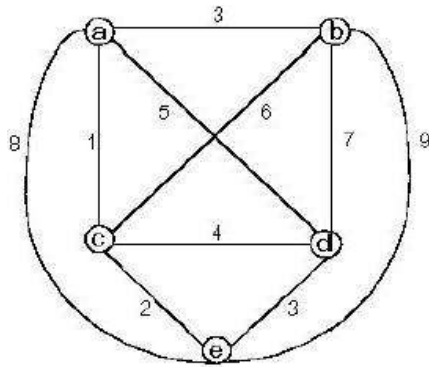
- At each step, the search algorithm chooses a new unexpanded leaf node to expand. The different search strategies essentially correspond to the different algorithms one can use to select which is the next node to be expanded at each stage.

Examples of search problems

- **Traveling Salesman Problem:** Given n cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city.
- A lower bound on the length l of any tour can be computed as follows
 - ✓ For each city i , $1 \leq i \leq n$, find the sum s_i of the distances from city i to the two nearest cities.
 - ✓ Compute the sum s of these n numbers.
 - ✓ Divide the result by 2 and round up the result to the nearest integer

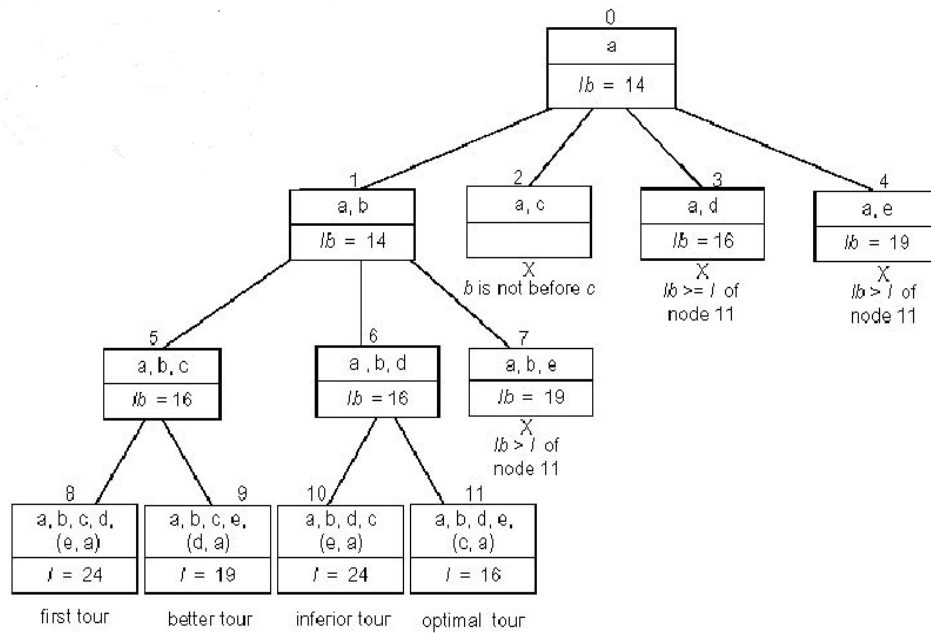
$$lb = s / 2$$

- The lower bound for the graph shown in the Fig 5.1 can be computed as follows:



$$lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 = 14.$$

- For any subset of tours that must include particular edges of a given graph, the lower bound can be modified accordingly. E.g.: For all the Hamiltonian circuits of the graph that must include edge (a, d), the lower bound can be computed as follows:
 $lb = [(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)] / 2 = 16.$
- Applying the branch-and-bound algorithm, with the bounding function $lb = s / 2$, to find the shortest Hamiltonian circuit for the given graph, we obtain the state-space tree as shown below:
- To reduce the amount of potential work, we take advantage of the following two observations:
 - ✓ We can consider only tours that start with a.
 - ✓ Since the graph is undirected, we can generate only tours in which b is visited before c.
- In addition, after visiting $n - 1$ cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one is shown in the Fig 5.2



Root node includes only the starting vertex a with a lower bound of

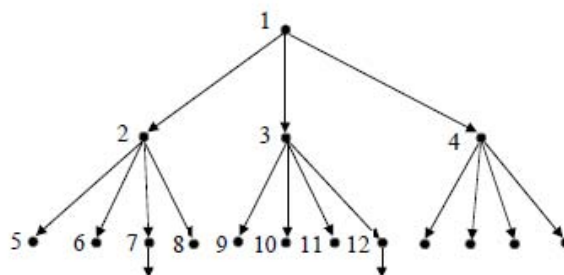
$$lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 = 14.$$

- Node 1 represents the inclusion of edge (a, b)
 $lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 = 14.$
- Node 2 represents the inclusion of edge (a, c). Since b is not visited before c, this node is terminated.
- Node 3 represents the inclusion of edge (a, d)
 $lb = [(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)] / 2 = 16.$
- Node 4 represents the inclusion of edge (a, e)
 $lb = [(1 + 8) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 8)] / 2 = 19.$
- Among all the four live nodes of the root, node 1 has a better lower bound. Hence we branch from node 1.
- Node 5 represents the inclusion of edge (b, c)
 $lb = [(1 + 3) + (3 + 6) + (1 + 6) + (3 + 4) + (2 + 3)] / 2 = 16.$
- Node 6 represents the inclusion of edge (b, d)
 $lb = [(1 + 3) + (3 + 7) + (1 + 2) + (3 + 7) + (2 + 3)] / 2 = 16.$
- Node 7 represents the inclusion of edge (b, e)
 $lb = [(1 + 3) + (3 + 9) + (1 + 2) + (3 + 4) + (2 + 9)] / 2 = 19.$

- Since nodes 5 and 6 both have the same lower bound, we branch out from each of them.
- Node 8 represents the inclusion of the edges (c, d), (d, e) and (e, a). Hence, the length of the tour,
 $l = 3 + 6 + 4 + 3 + 8 = 24$.
- Node 9 represents the inclusion of the edges (c, e), (e, d) and (d, a). Hence, the length of the tour,
 $l = 3 + 6 + 2 + 3 + 5 = 19$.
- Node 10 represents the inclusion of the edges (d, c), (c, e) and (e, a). Hence, the length of the tour,
 $l = 3 + 7 + 4 + 2 + 8 = 24$.
- Node 11 represents the inclusion of the edges (d, e), (e, c) and (c, a). Hence, the length of the tour,
 $l = 3 + 7 + 3 + 2 + 1 = 16$.
- Node 11 represents an optimal tour since its tour length is better than or equal to the other live nodes, 8, 9, 10, 3 and 4.
- The optimal tour is $a \rightarrow b \rightarrow d \rightarrow e \rightarrow c \rightarrow a$ with a tour length of 16.

Uniformed or Blind search

- **Breadth First Search (BFS):** BFS expands the leaf node with the lowest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple queue (“first in, first out”).



- This is guaranteed to find an optimal path to a goal state. It is memory intensive if the state space is large. If the typical branching factor is b , and the depth of the shallowest goal state is d – the space complexity is $O(b^d)$, and the time complexity is $O(b^d)$.
- BFS is an easy search technique to understand. The algorithm is presented below.

```

breadth_first_search ()

{

    store initial state in queue Q

    set state in the front of the Q as current state ;

    while (goal state is reached OR Q is empty)

    {

        apply rule to generate a new state from the current

        state ;

        if (new state is goal state) quit ;

        else if (all states generated from current states are

        exhausted)

        {

            delete the current state from the Q ;

            set front element of Q as the current state ;

        }

        else continue ;

    }

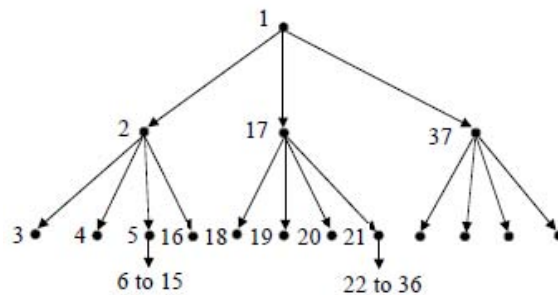
}

```

- The algorithm is illustrated using the bridge components configuration problem. The initial state is PDFG, which is not a goal state; and hence set it as the current state. Generate another state DPFG (by swapping 1st and 2nd position values) and add it to

the list. That is not a goal state, hence; generate next successor state, which is FDPG (by swapping 1st and 3rd position values). This is also not a goal state; hence add it to the list and generate the next successor state GDFP.

- Only three states can be generated from the initial state. Now the queue Q will have three elements in it, viz., DPFG, FDPG and GDFP. Now take DPFG (first state in the list) as the current state and continue the process, until all the states generated from this are evaluated. Continue this process, until the goal state DGPF is reached.
- The 14th evaluation gives the goal state. It may be noted that, all the states at one level in the tree are evaluated before the states in the next level are taken up; i.e., the evaluations are carried out breadth-wise. Hence, the search strategy is called breadth-first search.
- **Depth First Search (DFS):** DFS expands the leaf node with the highest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple stack (“last in, first out”).



- This is not guaranteed to find any path to a goal state. It is memory efficient even if the state space is large. If the typical branching factor is b , and the maximum depth of the tree is m – the space complexity is $O(bm)$, and the time complexity is $O(b^m)$.
- In DFS, instead of generating all the states below the current level, only the first state below the current level is generated and evaluated recursively. The search continues till a further successor cannot be generated.
- Then it goes back to the parent and explores the next successor. The algorithm is given below.

```
depth_first_search ()
```

```
{
```

```
    set initial state to current state ;
```

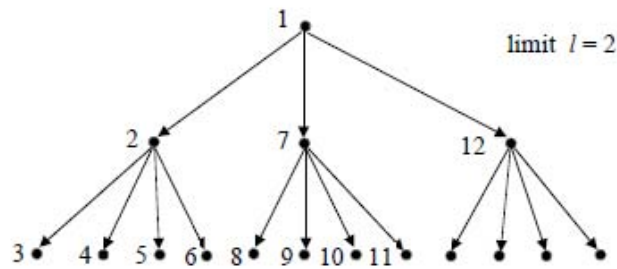
```
    if (initial state is current state) quit ;
```

```

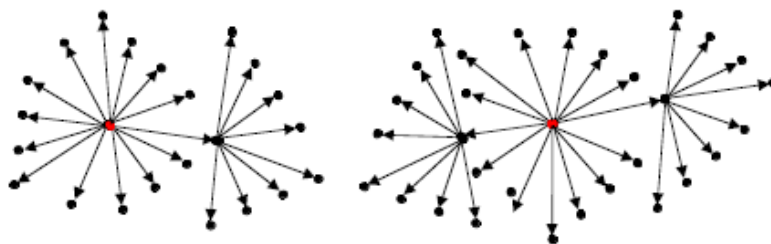
else
{
    if (a successor for current state exists)
    {
        generate a successor of the current state and
        set it as current state ;
    }
    else return ;
    depth_first_search (current_state) ;
    if (goal state is achieved) return ;
    else continue ;
}
}

```

- Since DFS stores only the states in the current path, it uses much less memory during the search compared to BFS.
- The probability of arriving at goal state with a fewer number of evaluations is higher with DFS compared to BFS. This is because, in BFS, all the states in a level have to be evaluated before states in the lower level are considered. DFS is very efficient when more acceptable solutions exist, so that the search can be terminated once the first acceptable solution is obtained.
- BFS is advantageous in cases where the tree is very deep.
- An ideal search mechanism is to combine the advantages of BFS and DFS.
- **Depth Limited Search (DLS):** DLS is a variation of DFS. If we put a limit l on how deep a depth first search can go, we can guarantee that the search will terminate (either in success or failure).



- If there is at least one goal state at a depth less than l , this algorithm is guaranteed to find a goal state, but it is not guaranteed to find an optimal path. The space complexity is $O(bl)$, and the time complexity is $O(b^l)$.
- **Depth First Iterative Deepening Search (DFIDS):** DFIDS is a variation of DLS. If the lowest depth of a goal state is not known, we can always find the best limit l for DLS by trying all possible depths $l = 0, 1, 2, 3, \dots$ in turn, and stopping once we have achieved a goal state.
- This appears wasteful because all the DLS for l less than the goal level are useless, and many states are expanded many times. However, in practice, most of the time is spent at the deepest part of the search tree, so the algorithm actually combines the benefits of DFS and BFS.
- Because all the nodes are expanded at each level, the algorithm is complete and optimal like BFS, but has the modest memory requirements of DFS. Exercise: if we had plenty of memory, could/should we avoid expanding the top level states many times?
- The space complexity is $O(bd)$ as in DLS with $l = d$, which is better than BFS.
- The time complexity is $O(b^d)$ as in BFS, which is better than DFS.
- **Bi-Directional Search (BDS):** The idea behind bi-directional search is to search simultaneously both forward from the initial state and backwards from the goal state, and stop when the two BFS searches meet in the middle.



- This is not always going to be possible, but is likely to be feasible if the state transitions are reversible. The algorithm is complete and optimal, and since the two

search depths are $\sim d/2$, it has space complexity $O(b^{d/2})$, and time complexity $O(b^{d/2})$. However, if there is more than one possible goal state, this must be factored into the complexity.

- **Repeated States:** In the above discussion we have ignored an important complication that often arises in search processes – the possibility that we will waste time by expanding states that have already been expanded before somewhere else on the search tree.
- For some problems this possibility can never arise, because each state can only be reached in one way.
- For many problems, however, repeated states are unavoidable. This will include all problems where the transitions are reversible, e.g.

$((1\ 2\ 3)) \rightarrow ((1)(2\ 3)) \rightarrow ((1\ 2\ 3)) \rightarrow ((1)(2\ 3)) \rightarrow ((1\ 2\ 3)) \rightarrow \dots$

- The search trees for these problems are infinite, but if we can prune out the repeated states, we can cut the search tree down to a finite size, We effectively only generate a portion of the search tree that matches the state space graph.
- **Avoiding Repeated States:** There are three principal approaches for dealing with repeated states:
 - Never return to the state you have just come from
The node expansion function must be prevented from generating any node successor that is the same state as the node's parent.
 - Never create search paths with cycles in them
The node expansion function must be prevented from generating any node successor that is the same state as any of the node's ancestors.
 - Never generate states that have already been generated before
This requires that every state ever generated is remembered, potentially resulting in space complexity of $O(b^d)$.
- **Comparing the Uninformed Search Algorithms:** We can now summarize the properties of our five uninformed search strategies:

Strategy	Complete	Optimal	Time Complexity	Space Complexity
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
DLS	If $l \geq d$	No	$O(b^l)$	$O(bl)$
DFIDS	Yes	Yes	$O(b^d)$	$O(bd)$
BDS	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

- Simple BFS and BDS are complete and optimal but expensive with respect to space and time.
- DFS requires much less memory if the maximum tree depth is limited, but has no guarantee of finding any solution, let alone an optimal one. DLS offers an improvement over DFS if we have some idea how deep the goal is.
- The best overall is DFID which is complete, optimal and has low memory requirements, but still exponential time.

Informed search

- Informed search uses some kind of evaluation function to tell us how far each expanded state is from a goal state, and/or some kind of heuristic function to help us decide which state is likely to be the best one to expand next.
- The hard part is to come up with good evaluation and/or heuristic functions. Often there is a natural evaluation function, such as distance in miles or number objects in the wrong position.
- Sometimes we can learn heuristic functions by analyzing what has worked well in similar previous searches.
- The simplest idea, known as greedy best first search, is to expand the node that is already closest to the goal, as that is most likely to lead quickly to a solution. This is like DFS in that it attempts to follow a single route to the goal, only attempting to try a different route when it reaches a dead end. As with DFS, it is not complete, not optimal, and has time and complexity of $O(b^m)$. However, with good heuristics, the time complexity can be reduced substantially.
- **Branch and Bound:** An enhancement of backtracking.
- Applicable to optimization problems.

- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution).
- Uses the bound for:
 - Ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far.
 - Guiding the search through state-space.
- The search path at the current node in a state-space tree can be terminated for any one of the following three reasons:
 - The value of the node’s bound is not better than the value of the best solution seen so far.
 - The node represents no feasible solutions because the constraints of the problem are already violated.
 - The subset of feasible solutions represented by the node consists of a single point and hence we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.
- **Best-First branch-and-bound:**
 - A variation of backtracking.
 - Among all the nonterminated leaves, called as the live nodes, in the current tree, generate all the children of the most promising node, instead of generation a single child of the last promising node as it is done in backtracking.
 - Consider the node with the best bound as the most promising node.
- **A* Search:** Suppose that, for each node n in a search tree, an evaluation function $f(n)$ is defined as the sum of the cost $g(n)$ to reach that node from the start state, plus an estimated cost $h(n)$ to get from that state to the goal state. That $f(n)$ is then the estimated cost of the cheapest solution through n .
- A* search, which is the most popular form of best-first search, repeatedly picks the node with the lowest $f(n)$ to expand next. It turns out that if the heuristic function $h(n)$ satisfies certain conditions, then this strategy is both complete and optimal.
- In particular, if $h(n)$ is an admissible heuristic, i.e. is always optimistic and never overestimates the cost to reach the goal, then A* is optimal.

- The classic example is finding the route by road between two cities given the straight line distances from each road intersection to the goal city. In this case, the nodes are the intersections, and we can simply use the straight line distances as $h(n)$.
- **Hill Climbing / Gradient Descent:** The basic idea of hill climbing is simple: at each current state we select a transition, evaluate the resulting state, and if the resulting state is an improvement we move there, otherwise we try a new transition from where we are.
- We repeat this until we reach a goal state, or have no more transitions to try. The transitions explored can be selected at random, or according to some problem specific heuristics.
- In some cases, it is possible to define evaluation functions such that we can compute the gradients with respect to the possible transitions, and thus compute which transition direction to take to produce the best improvement in the evaluation function.
- Following the evaluation gradients in this way is known as gradient descent.
- In neural networks, for example, we can define the total error of the output activations as a function of the connection weights, and compute the gradients of how the error changes as we change the weights. By changing the weights in small steps against those gradients, we systematically minimize the network's output errors.

Searching And-Or graphs

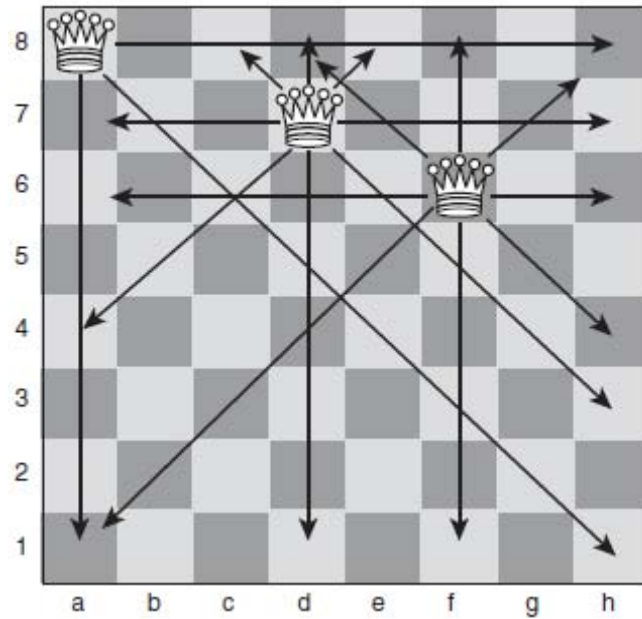
- The DFS and BFS strategies for OR trees and graphs can be adapted for And-Or trees
- The main difference lies in the way termination conditions are determined, since all goals following an And node must be realized, whereas a single goal node following an Or node will do
- A more general optimal strategy is AO* (O for ordered) algorithm
- As in the case of the A* algorithm, we use the open list to hold nodes that have been generated but not expanded and the closed list to hold nodes that have been expanded
- The algorithm is a variation of the original given by Nilsson
- It requires that nodes traversed in the tree be labeled as solved or unsolved in the solution process to account for And node solutions which require solutions to all successors nodes.
- A solution is found when the start node is labeled as solved

- The AO* algorithm
 - Step 1: Place the start node s on open
 - Step 2: Using the search tree constructed thus far, compute the most promising solution tree T_0
 - Step 3: Select a node n that is both on open and a part of T_0 . Remove n from open and place it on closed
 - Step 4: If n is a terminal goal node, label n as solved. If the solution of n results in any of n 's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where T_0 is the solution tree. Remove from open all nodes with a solved ancestor
 - Step 5: If n is not a solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of n 's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvable ancestors
 - Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one subproblem, generate their successors to give individual subproblems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h^* for each newly generated node and place all such nodes that do not yet have descendants on open. Next recompute the values of h^* at n and each ancestor of n
 - Step 7: Return to step 2
- It can be shown that AO* will always find a minimum-cost solution tree if one exists, provided only that $h^*(n) \leq h(n)$, and all arc costs are positive. Like A*, the efficiency depends on how closely h^* approximates h

Constraint Satisfaction Search

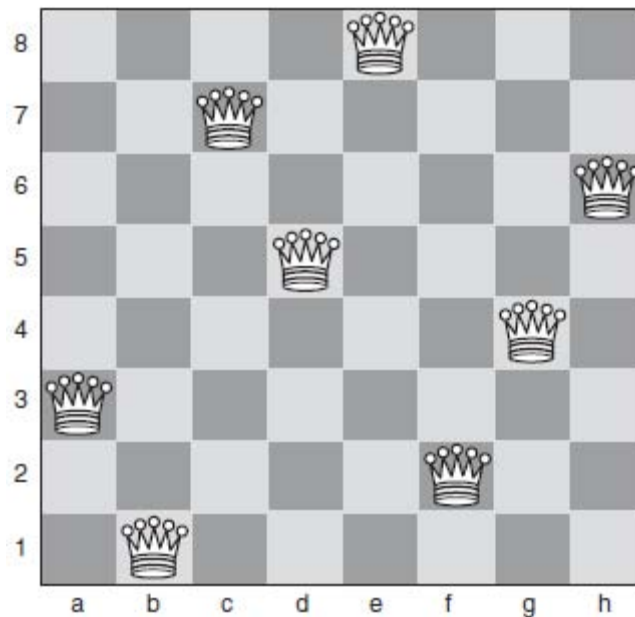
- Search can be used to solve problems that are limited by constraints, such as the eight-queens problem. Such problems are often known as Constraint Satisfaction Problems, or CSPs. I

- In this problem, eight queens must be placed on a chess board in such a way that no two queens are on the same diagonal, row, or column. If we use traditional chess board notation, we mark the columns with letters from a to g and the rows with numbers from 1 to 8. So, a square can be referred to by a letter and a number, such as a4 or g7.
- This kind of problem is known as a constraint satisfaction problem (CSP) because a solution must be found that satisfies the constraints.
- In the case of the eight-queens problem, a search tree can be built that represents the possible positions of queens on the board. One way to represent this is to have a tree that is 8-ply deep, with a branching factor of 64 for the first level, 63 for the next level, and so on, down to 57 for the eighth level.
- A goal node in this tree is one that satisfies the constraints that no two queens can be on the same diagonal, row, or column.
- An extremely simplistic approach to solving this problem would be to analyze every possible configuration until one was found that matched the constraints.
- A more suitable approach to solving the eight-queens problem would be to use depth-first search on a search tree that represents the problem in the following manner:
 - The first branch from the root node would represent the first choice of a square for a queen. The next branch from these nodes would represent choices of where to place the second queen.
 - The first level would have a branching factor of 64 because there are 64 possible squares on which to place the first queen. The next level would have a somewhat lower branching factor because once a queen has been placed, the constraints can be used to determine possible squares upon which the next queen can be placed.
 - The branching factor will decrease as the algorithm searches down the tree. At some point, the tree will terminate because the path being followed will lead to a position where no more queens can be placed on legal squares on the board, and there are still some queens remaining.



In fact, because each row and each column must contain exactly one queen, the branching factor can be significantly reduced by assuming that the first queen must be placed in row 1, the second in row 2, and so on. In this way, the first level will have a branching factor of 8 (a choice of eight squares on which the first queen can be placed), the next 7, the next 6, and so on.

- The search tree can be further simplified as each queen placed on the board “uses up” a diagonal, meaning that the branching factor is only 5 or 6 after the first choice has been made, depending on whether the first queen is placed on an edge of the board (columns a or h) or not.
- The next level will have a branching factor of about 4, and the next may have a branching factor of just 2, as shown in Fig 6.1.
- The arrows in Fig 6.1 show the squares to which each queen can move.
- Note that no queen can move to a square that is already occupied by another queen.



- In Fig 6.1, the first queen was placed in column a of row 8, leaving six choices for the next row. The second queen was placed in column d of row 7, leaving four choices for row 6. The third queen was placed in column f in row 6, leaving just two choices (column c or column h) for row 5.
- Using knowledge like this about the problem that is being solved can help to significantly reduce the size of the search tree and thus improve the efficiency of the search solution.
- A solution will be found when the algorithm reaches depth 8 and successfully places the final queen on a legal square on the board.
- A goal node would be a path containing eight squares such that no two squares shared a diagonal, row, or column.
- One solution to the eight-queens problem is shown in above Fig .
- Note that in this solution, if we start by placing queens on squares e8, c7, h6, and then d5, once the fourth queen has been placed, there are only two choices for placing the fifth queen (b4 or g4). If b4 is chosen, then this leaves no squares that could be chosen for the final three queens to satisfy the constraints. If g4 is chosen for the fifth queen, as has been done in Fig 6.2, only one square is available for the sixth queen (a3), and the final two choices are similarly constrained. So, it can be seen that by applying the

constraints appropriately, the search tree can be significantly reduced for this problem.

- Using chronological backtracking in solving the eight-queens problem might not be the most efficient way to identify a solution because it will backtrack over moves that did not necessarily directly lead to an error, as well as ones that did. In this case, nonchronological backtracking, or dependency-directed backtracking could be more useful because it could identify the steps earlier in the search tree that caused the problem further down the tree.

Forward Checking

- In fact, backtracking can be augmented in solving problems like the eightqueens problem by using a method called forward checking.
- As each queen is placed on the board, a forward-checking mechanism is used to delete from the set of possible future choices any that have been rendered impossible by placing the queen on that square.
- For example, if a queen is placed on square a1, forward checking will remove all squares in row 1, all squares in column a, and also squares b2, c3, d4, e5, f6, g7, and h8.
- In this way, if placing a queen on the board results in removing all remaining squares, the system can immediately backtrack, without having to attempt to place any more queens.
- This can often significantly improve the performance of solutions for CSPs such as the eight-queens problem.

Most-Constrained Variables

- A further improvement in performance can be achieved by using the most-constrained variable heuristic.
- At each stage of the search, this heuristic involves working with the variable that has the least possible number of valid choices.

- In the case of the eight-queens problem, this might be achieved by considering the problem to be one of assigning a value to eight variables, a through h. Assigning value 1 to variable a means placing a queen in square a1.
- To use the most constrained variable heuristic with this representation means that at each move we assign a value to the variable that has the least choices available to it. Hence, after assigning a = 1, b = 3, and c = 5, this leaves three choices for d, three choices for e, one choice for f, three choices for g, and three choices for h. Hence, our next move is to place a queen in column f.
- This heuristic is perhaps more clearly understood in relation to the mapcoloring problem. It makes sense that, in a situation where a particular country can be given only one color due to the colors that have been assigned to its neighbors, that country be colored next.
- The most-constraining variable heuristic is similar in that it involves assigning a value next to the variable that places the greatest number of constraints on future variables.
- The least-constraining value heuristic is perhaps more intuitive than the two already presented in this section.
- This heuristic involves assigning a value to a variable that leaves the greatest number of choices for other variables.
- This heuristic can be used to make n-queens problems with extremely large values of n quite solvable.

Example: Cryptographic Problems

- The constraint satisfaction procedure is also a useful way to solve problems such as cryptographic problems. For example:

FORTY

+ TEN

+ TEN

SIXTY

Solution:

29786

+ 850

+ 850

31486

- This cryptographic problem can be solved by using a Generate and Test method, applying the following constraints:
 - Each letter represents exactly one number.
 - No two letters represent the same number.
- Generate and Test is a brute-force method, which in this case involves cycling through all possible assignments of numbers to letters until a set is found that meets the constraints and solves the problem.
- Without using constraints, the method would first start by attempting to assign 0 to all letters, resulting in the following sum:

00000

+ 000

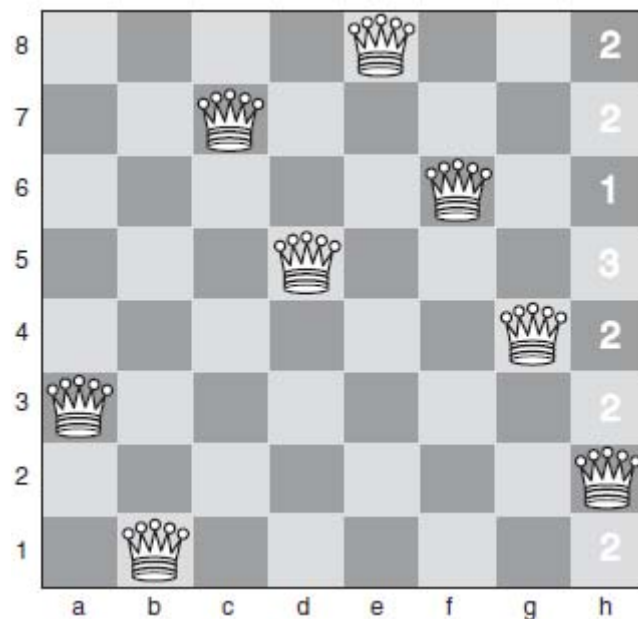
+ 000

00000

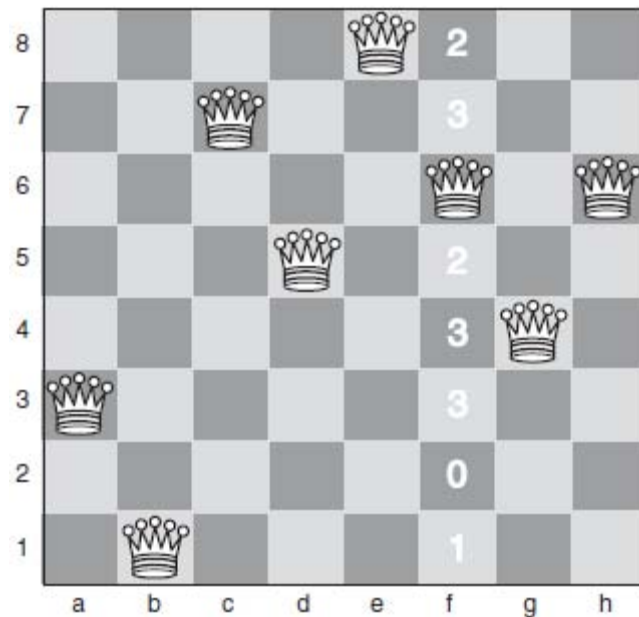
- Although this may appear to be a valid solution to the problem, it does not meet the constraints laid down that specify that each letter can be assigned only one number, and each number can be assigned only to one letter.
- Hence, constraints are necessary simply to find the correct solution to the problem. They also enable us to reduce the size of the search tree.
- In this case, for example, it is not necessary to examine possible solutions where two letters have been assigned the same number, which dramatically reduces the possible solutions to be examined.

Heuristic Repair

- Heuristics can be used to improve performance of solutions to constraint satisfaction problems.
- One way to do this is to use a heuristic repair method, which involves generating a possible solution (randomly, or using a heuristic to generate a position that is close to a solution) and then making changes that reduce the distance of the state from the goal.
- In the case of the eight-queens problem, this could be done using the minconflicts heuristic.
- To move from one state to another state that is likely to be closer to a solution using the min-conflicts heuristic, select one queen that conflicts with another queen (in other words, it is on the same row, column, or diagonal as another queen).
- Now move that queen to a square where it conflicts with as few queens as possible. Continue with another queen. To see how this method would work, consider the starting position shown in Fig 6.3.



- This starting position has been generated by placing the queens such that there are no conflicts on rows or columns. The only conflict here is that the queen in column 3 (on c7) is on a diagonal with the queen in column h (on h2).
- To move toward a solution, we choose to move the queen that is on column h.
- We will only ever apply a move that keeps a queen on the same column because we already know that we need to have one queen on each column.
- Each square in column h has been marked with a number to show how many other queens that square conflicts with. Our first move will be to move the queen on column h up to row 6, where it will conflict only with one queen. Then we arrive at the position shown in below Fig
- Because we have created a new conflict with the queen on row 6 (on f6), our next move must be to move this queen. In fact, we can move it to a square where it has zero conflicts. This means the problem has been solved, and there are no remaining conflicts.
- This method can be used not only to solve the eight-queens problem but also has been successfully applied to the n-queens problem for extremely large values of n. It has been shown that, using this method, the 1,000,000 queens problem can be solved in an average of around 50 steps.
- Solving the 1,000,000-queens problem using traditional search techniques would be impossible because it would involve searching a tree with a branching factor of 10^{12} .



Local Search and Metaheuristics

- Local search methods work by starting from some initial configuration (usually random) and making small changes to the configuration until a state is reached from which no better state can be achieved.
- Hill climbing is a good example of a local search technique.
- Local search techniques, used in this way, suffer from the same problems as hill climbing and, in particular, are prone to finding local maxima that are not the best solution possible.
- The methods used by local search techniques are known as metaheuristics.
- Examples of metaheuristics include simulated annealing, tabu search, genetic algorithms, ant colony optimization, and neural networks.
- This kind of search method is also known as local optimization because it is attempting to optimize a set of values but will often find local maxima rather than a global maximum.
- A local search technique applied to the problem of allocating teachers to classrooms would start from a random position and make small changes until a configuration was reached where no inappropriate allocations were made.
- **Exchanging Heuristics**
 - The simplest form of local search is to use an exchanging heuristic.

- An exchanging heuristic moves from one state to another by exchanging one or more variables by giving them different values. We saw this in solving the eight-queens problem as heuristic repair.
- A k-exchange is considered to be a method where k variables have their values changed at each step.
- The heuristic repair method we applied to the eight-queens problem was 2-exchange.
- A k-exchange can be used to solve the traveling salesman problem. A tour (a route through the cities that visits each city once, and returns to the start) is generated at random. Then, if we use 2-exchange, we remove two edges from the tour and substitute them for two other edges. If this produces a valid tour that is shorter than the previous one, we move on from here. Otherwise, we go back to the previous tour and try a different set of substitutions.
- In fact, using $k = 2$ does not work well for the traveling salesman problem, whereas using $k = 3$ produces good results.
- Using larger numbers of k will give better and better results but will also require more and more iterations.
- Using $k = 3$ gives reasonable results and can be implemented efficiently. It does, of course, risk finding local maxima, as is often the case with local search methods.

- **Iterated Local Search**

- Iterated local search techniques attempt to overcome the problem of local maxima by running the optimization procedure repeatedly, from different initial states.
- If used with sufficient iterations, this kind of method will almost always find a global maximum.
- The aim, of course, in running methods like this is to provide a very good solution without needing to exhaustively search the entire problem space.
- In problems such as the traveling salesman problem, where the search space grows extremely quickly as the number of cities increases, results can be generated that are good enough (i.e., a local maximum) without using many iterations, where a perfect solution would be impossible to find (or at least it would be impossible to guarantee a perfect solution even one iteration of local search may happen upon the global maximum).

- **Tabu Search**

- Tabu search is a metaheuristic that uses a list of states that have already been visited to attempt to avoid repeating paths.
- The tabu search metaheuristic is used in combination with another heuristic and operates on the principle that it is worth going down a path that appears to be poor if it avoids following a path that has already been visited.
- In this way, tabu search is able to avoid local maxima.

Simulated Annealing

- Annealing is a process of producing very strong glass or metal, which involves heating the material to a very high temperature and then allowing it to cool very slowly.
- In this way, the atoms are able to form the most stable structures, giving the material great strength.
- Simulated annealing is a local search metaheuristic based on this method and is an extension of a process called metropolisMonteCarlo simulation.
- Simulated annealing is applied to a multi-value combinatorial problem where values need to be chosen for many variables to produce a particular value for some global function, dependent on all the variables in the system.
- This value is thought of as the energy of the system, and in general the aim of simulated annealing is to find a minimum energy for a system.
- Simple Monte Carlo simulation is a method of learning information (such as shape) about the shape of a search space. The process involves randomly selecting points within the search space.
- An example of its use is as follows: A square is partially contained within a circle. Simple Monte Carlo simulation can be used to identify what proportion of the square is within the circle and what proportion is outside the circle. This is done by randomly sampling points within the square and checking which ones are within the circle and which are not.
- Metropolis Monte Carlo simulation extends this simple method as follows: Rather than selecting new states from the search space at random, a new state is chosen by making a small change to the current state.

- If the new state means that the system as a whole has a lower energy than it did in the previous state, then it is accepted.
- If the energy is higher than for the previous state, then a probability is applied to determine whether the new state is accepted or not. This probability is called a Boltzmann acceptance criterion and is calculated as follows: $e(-dE/T)$ where T is the current temperature of the system, and dE is the increase in energy that has been produced by moving from the previous state to the new state.
- The temperature in this context refers to the percentage of steps that can be taken that lead to a rise in energy: At a higher temperature, more steps will be accepted that lead to a rise in energy than at low temperature.
- To determine whether to move to a higher energy state or not, the probability $e(-dE/T)$ is calculated, and a random number is generated between 0 and 1. If this random number is lower than the probability function, the new state is accepted. In cases where the increase in energy is very high, or the temperature is very low, this means that very few states will be accepted that involve an increase in energy, as $e(-dE/T)$ approaches zero.
- The fact that some steps are allowed that increase the energy of the system enables the process to escape from local minima, which means that simulated annealing often can be an extremely powerful method for solving complex problems with many local maxima.
- Some systems use $e(-dE/kT)$ as the probability that the search will progress to a state with a higher energy, where k is Boltzmann's constant (Boltzmann's constant is approximately 1.3807×10^{-23} Joules per Kelvin).
- Simulated annealing uses Monte Carlo simulation to identify the most stable state (the state with the lowest energy) for a system.
- This is done by running successive iterations of metropolis Monte Carlo simulation, using progressively lower temperatures. Hence, in successive iterations, fewer and fewer steps are allowed that lead to an overall increase in energy for the system.
- A cooling schedule (or annealing schedule) is applied, which determines the manner in which the temperature will be lowered for successive iterations.
- Two popular cooling schedules are as follows:

$$T_{\text{new}} = T_{\text{old}} - dT$$

$$T_{\text{new}} = C \times T_{\text{old}} \text{ (where } C < 1.0\text{)}$$

- The cooling schedule is extremely important, as is the choice of the number of steps of metropolis Monte Carlo simulation that are applied in each iteration.
- These help to determine whether the system will be trapped by local minima (known as quenching). The number of times the metropolis Monte Carlo simulation is applied per iteration is for later iterations.
- Also important in determining the success of simulated annealing are the choice of the initial temperature of the system and the amount by which the temperature is decreased for each iteration.
- These values need to be chosen carefully according to the nature of the problem being solved. When the temperature, T , has reached zero, the system is frozen, and if the simulated annealing process has been successful, it will have identified a minimum for the total energy of the system.
- Simulated annealing has a number of practical applications in solving problems with large numbers of interdependent variables, such as circuit design.
- It has also been successfully applied to the traveling salesman problem.
- **Uses of Simulated Annealing**
 - Simulated annealing was invented in 1983 by Kirkpatrick, Gelatt, and Vecchi.
 - It was first used for placing VLSI* components on a circuit board.
 - Simulated annealing has also been used to solve the traveling salesman problem, although this approach has proved to be less efficient than using heuristic methods that know more about the problem.
 - It has been used much more successfully in scheduling problems and other large combinatorial problems where values need to be assigned to a large number of variables to maximize (or minimize) some function of those variables.

Real-Time A*

- Real-time A* is a variation of A*.
- Search continues on the basis of choosing paths that have minimum values of $f(\text{node}) = g(\text{node}) + h(\text{node})$. However, $g(\text{node})$ is the distance of the node from the current node, rather than from the root node.
- Hence, the algorithm will backtrack if the cost of doing so plus the estimated cost of solving the problem from the new node is less than the estimated cost of solving the problem from the current node.

- Implementing real-time A* means maintaining a hash table of previously visited states with their $h(\text{node})$ values.

Iterative-Deepening A* (IDA*)

- By combining iterative-deepening with A*, we produce an algorithm that is optimal and complete (like A*) and that has the low memory requirements of depth-first search.
- IDA* is a form of iterative-deepening search where successive iterations impose a greater limit on $f(\text{node})$ rather than on the depth of a node.
- IDA* performs well in problems where the heuristic value $f(\text{node})$ has relatively few possible values.
- For example, using the Manhattan distance as a heuristic in solving the eight-queens problem, the value of $f(\text{node})$ can only have values 1, 2, 3, or 4.
- In this case, the IDA* algorithm only needs to run through a maximum of four iterations, and it has a time complexity not dissimilar from that of A*, but with a significantly improved space complexity because it is effectively running depth-first search.
- In cases such as the traveling salesman problem where the value of $f(\text{node})$ is different for every state, the IDA* method has to expand $1 + 2 + 3 + \dots + n$ nodes = $O(n^2)$ where A* would expand n nodes.

Propositional and Predicate Logic

Logic is concerned with reasoning and the validity of arguments. In general, in logic, we are not concerned with the truth of statements, but rather with their validity. That is to say, although the following argument is clearly logical, it is not something that we would consider to be true:

All lemons are blue

Mary is a lemon

Therefore, Mary is blue

This set of statements is considered to be valid because the conclusion (Mary is blue) follows logically from the other two statements, which we often call the premises. The reason that validity and truth can be separated in this way is simple: a piece of a reasoning is considered

to be valid if its conclusion is true in cases where its premises are also true. Hence, a valid set of statements such as the ones above can give a false conclusion, provided one or more of the premises are also false.

We can say: a piece of reasoning is valid if it leads to a true conclusion in every situation where the premises are true.

Logic is concerned with truth values. The possible truth values are true and false. These can be considered to be the fundamental units of logic, and almost all logic is ultimately concerned with these truth values.

Logic is widely used in computer science, and particularly in Artificial Intelligence. Logic is widely used as a representational method for Artificial Intelligence. Unlike some other representations, logic allows us to easily reason about negatives (such as, “this book is not red”) and disjunctions (“or”—such as, “He’s either a soldier or a sailor”).

Logic is also often used as a representational method for communicating concepts and theories within the Artificial Intelligence community. In addition, logic is used to represent language in systems that are able to understand and analyze human language.

As we will see, one of the main weaknesses of traditional logic is its inability to deal with uncertainty. Logical statements must be expressed in terms of truth or falsehood—it is not possible to reason, in classical logic, about possibilities. We will see different versions of logic such as modal logics that provide some ability to reason about possibilities, and also probabilistic methods and fuzzy logic that provide much more rigorous ways to reason in uncertain situations.

Logical Operators

- In reasoning about truth values, we need to use a number of operators, which can be applied to truth values.
- We are familiar with several of these operators from everyday language:

I like apples and oranges.

You can have an ice cream or a cake.