



# **Lectures in Artificial Intelligence (AI301)**

Faculty of Computers and  
Information

**2024 / 2025**

## Contents

**The course covers three major topics:**

### **Chapter 1: Search**

- Graph search
- Game Search
- Constraint Satisfaction

### **Chapter 2: Introduction to Machine Learning**

- What is machine learning?
- Machine learning algorithms
- Naïve Bayes classification
- Linear regression
- Logistic regression
- Nearest Neighbors
- Support vector machine (SVM)
- Decision Trees
- Random Forest
- Principal Components Analysis (PCA)
- K-means Clustering
- Neural Networks
- Deep learning



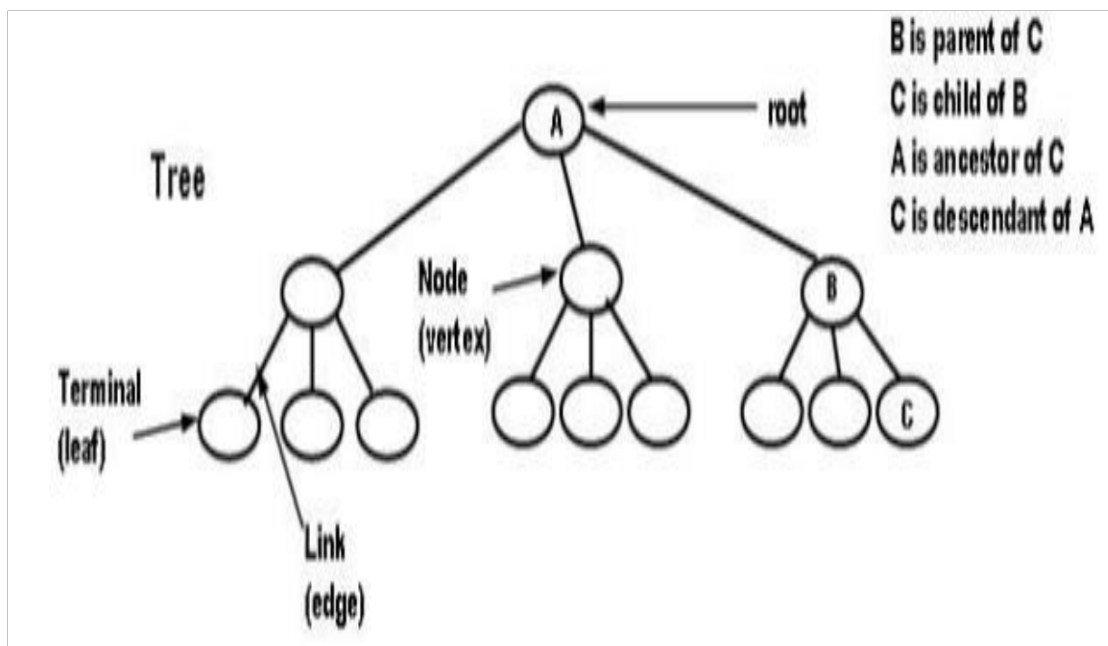
# CHAPTER 1

## SEARCH

### 1.1. Search

Search plays a key role in many parts of AI. These algorithms provide the conceptual backbone of almost every approach to the systematic exploration of alternatives. We will start with some background, terminology and basic implementation strategies and then cover four classes of search algorithms, which differ along two dimensions: First, is the difference between uninformed (also known as blind) search and then informed (also known as heuristic) searches. Informed searches have access to task-specific information that can be used to make the search process more efficient. The other difference is between any path searches and optimal searches. Optimal searches are looking for the best possible path while any-path searches will just settle for finding some solution.

The search methods we will be dealing with are defined on trees and graphs, so we need to fix on some terminology for these structures:

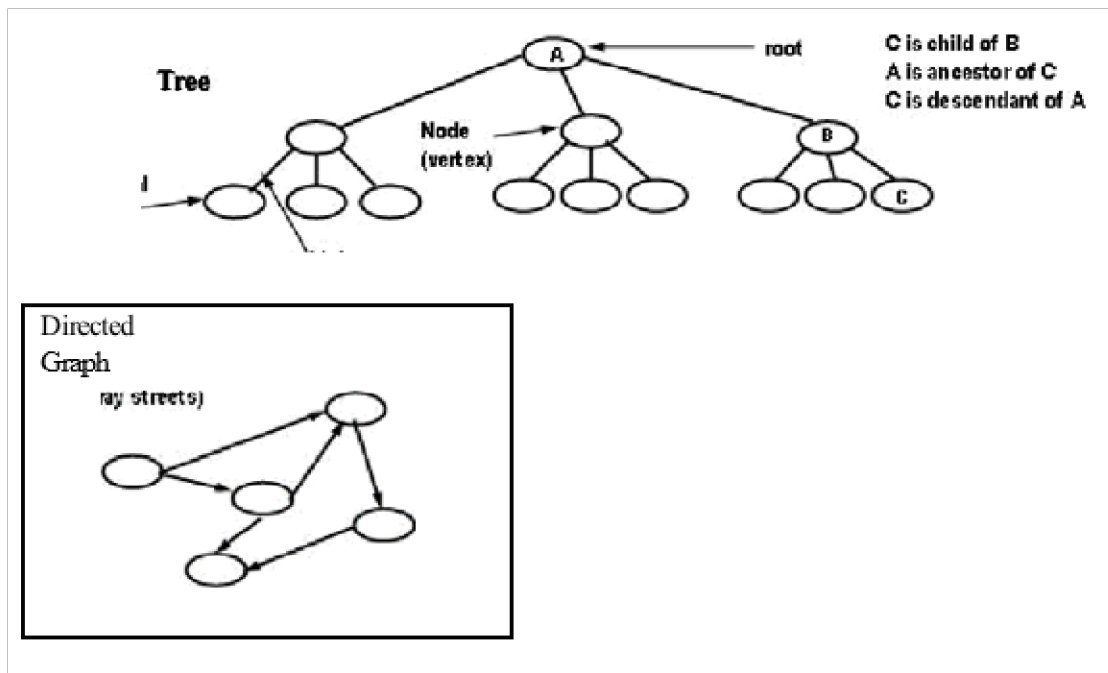


- A **tree** is made up of **nodes** and **links** (circles and lines) connected so that there are no loops (cycles). Nodes are sometimes referred to as vertices

and links as edges (this is more common in talking about graphs).

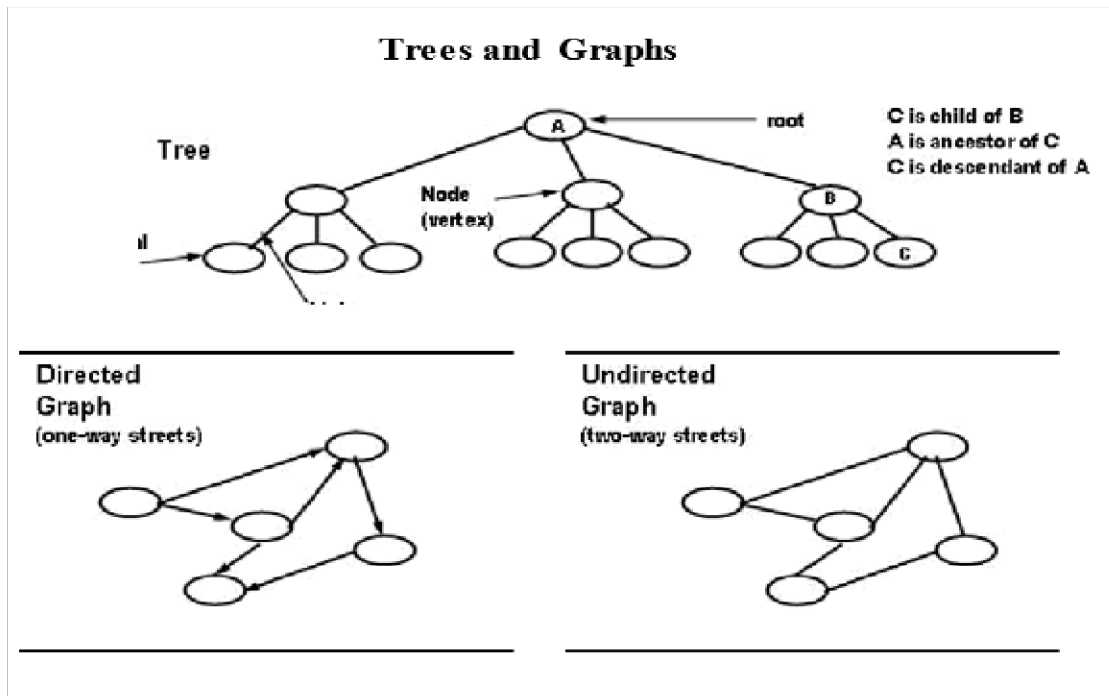
- A tree has a **root** node (where the tree "starts"). Every node except the root has a single **parent (direct ancestor)**. More generally, an ancestor node is a node that can be reached by repeatedly going to a parent node. Each node (except the terminal (**leaf**) nodes) has one or more children (**direct descendants**). More generally, a **descendant** node is a node that can be reached by repeatedly going to a child node.

A graph is also a set of nodes connected by links but where loops are allowed and a node can have multiple parents. We have two kinds of graphs to deal with: directed graphs, where the links have direction (akin to one-way streets).

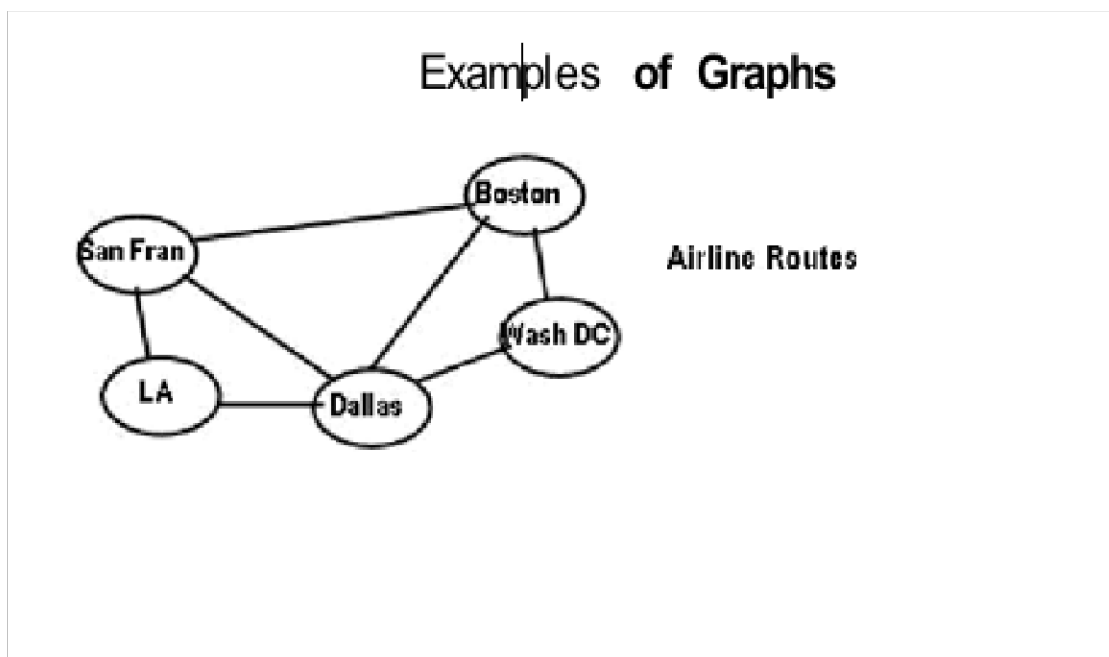


And, **undirected** graphs where the links go both ways. You can think of an undirected graph as shorthand for a graph with directed links going each way between connected nodes.

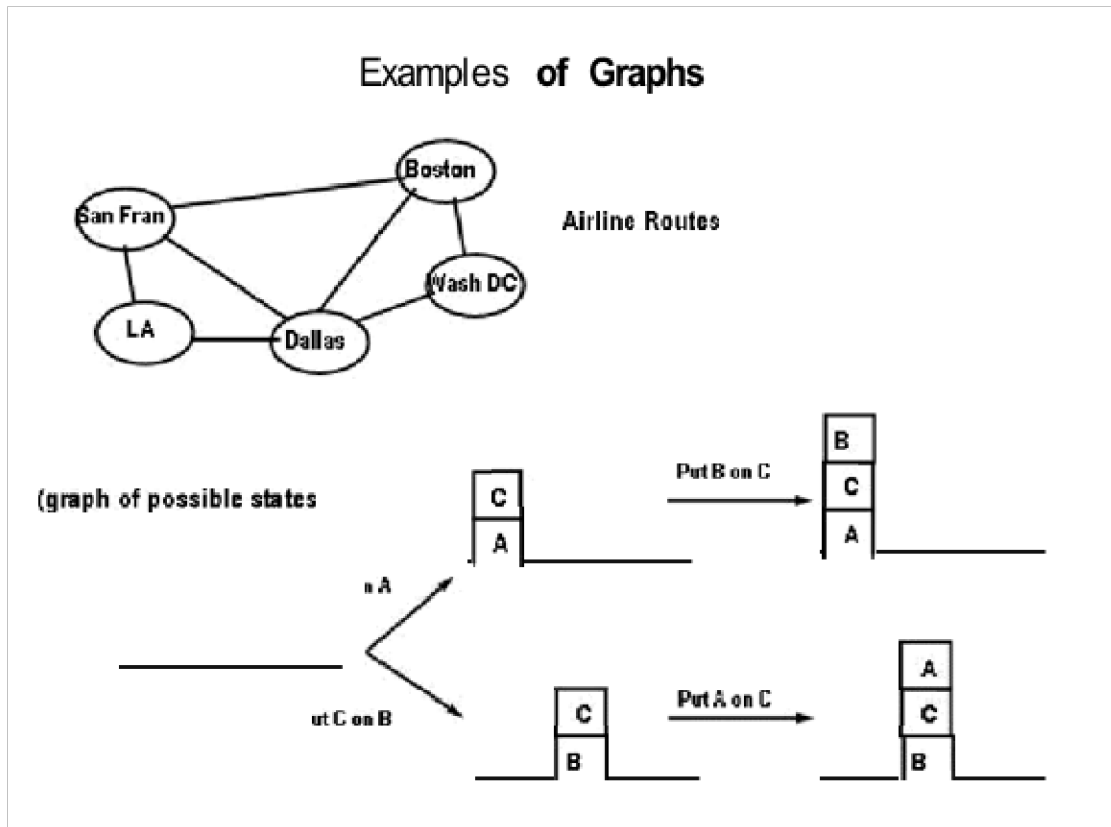
Graphs are everywhere; for example, think about road networks or airline routes or computer networks. In all of these cases we might be interested in finding a path through the graph that satisfies some property. It may be that any path will do or we may be interested in a path having the fewest "hops" or a least cost path assuming the hops are not all equivalent, etc.



Graphs are everywhere; for example, think about road networks or airline routes or computer networks. In all of these cases we might be interested in finding a path through the graph that satisfies some property. It may be that any path will do or we may be interested in a path having the fewest "hops" or a least cost path assuming the hops are not all equivalent, etc.



However, graphs can also be much more abstract. Think of the graph defined as follows: the nodes denote descriptions of a state of the world, e.g., which blocks are on top of what in a blocks scene, and where the links represent actions that change from one state to the other.



A path through such a graph (from a start node to a goal node) is a "plan of action" to achieve some desired goal state from some known starting state. It is this type of graph that is of more general interest in AI.

One general approach to problem solving in AI is to reduce the problem to be solved to one of searching a graph. To use this approach, we must specify what are the states, the actions and the goal test.

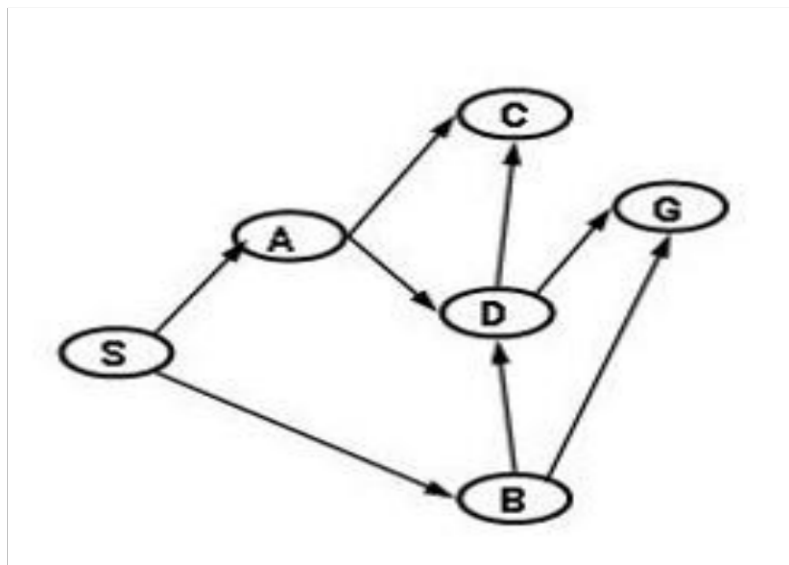
A state is supposed to be complete, that is, to represent all (and preferably only) the relevant aspects of the problem to be solved. So, for example, when we are planning the cheapest round-the-world flight plan, we don't need to know the address of the airports; knowing the identity of the airport is enough. The address will be important, however, when planning how to get from the hotel to the airport. Note that, in general, to plan an air route we need to know the airport, not just the city,

since some cities have multiple airports.

We are assuming that the actions are **deterministic**, that is, we know exactly the state after the action is performed. We also assume that the actions are **discrete**, so we don't have to represent what happens while the action is happening. For example, we assume that a flight gets us to the scheduled destination and that what happens during the flight does not matter (at least when planning the route).

Note that we've indicated that (in general) we need a test for the goal, not just one specific goal state. So, for example, we might be interested in any city in Germany rather than specifically Frankfurt. Or, when proving a theorem, all we care is about knowing one fact in our current data base of facts. Any final set of facts that contains the desired fact is a proof.

In principle, we could also have multiple starting states, for example, if we have some uncertainty about the starting state. But, for now, we are not addressing issues of uncertainty either in the starting state or in the result of the actions.



Note that trees are a subclass of directed graphs (even when not shown with arrows on the links). Trees don't have cycles and every node has a single parent (or is the root). Cycles are bad for searching, since, obviously, you don't want to go round and round getting nowhere.

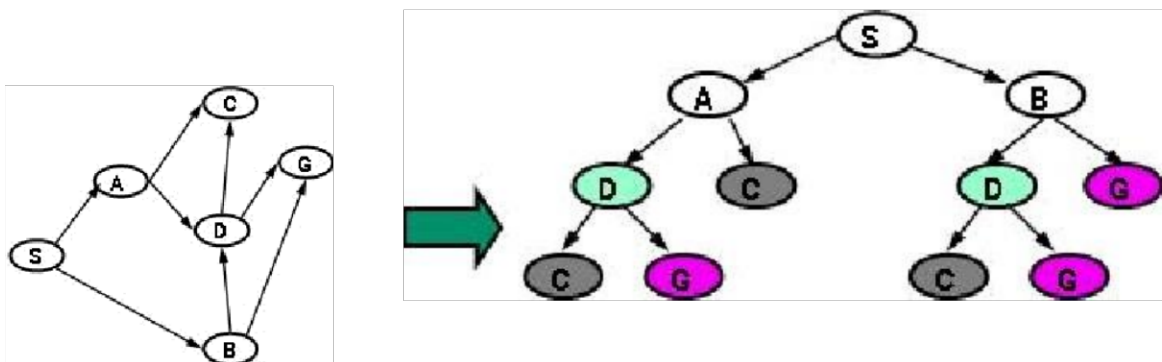
When asked to search a graph, we can construct an equivalent problem of

searching a tree by doing two things: turning undirected links into two directed links; and, more importantly, making sure we never consider a path with a loop or, even better, by never visiting the same node twice.

You can see an example of this converting from a graph to a tree here. If we assume that S is the start of our search and we are trying to find a path to G, then we can walk through the graph and make connections from every node to every connected node that would not create a cycle (and stop whenever we hit G). Note that such a tree has a leaf node for every non-looping path in the graph starting at S.

Also note, however, that even though we avoided loops, some nodes (the colored ones) are duplicated in the tree, that is, they were reached along different non-looping paths. This means that a complete search of this tree might do extra work.

The issue of how much effort to place in avoiding loops and avoiding extra visits to nodes is an important one that we will revisit later when we discuss the various search algorithms.



One important distinction that will help us keep things straight is that between a **state** and a search **node**.

A state is an arrangement of the real world (or at least our model of it). We assume that there is an underlying "real" state graph that we are searching (although it might

not be explicitly represented in the computer; it may be implicitly defined by the actions). We assume that you can arrive at the same real world state by multiple routes, that is, by different sequences of actions.

A search node, on the other hand, is a data structure in the search algorithm, which constructs an explicit tree of nodes while searching. Each node refers to some state, but not uniquely. Note that a node also corresponds to a path from the start state to the state associated with the node. This follows from the fact that the search algorithm is generating a tree. So, if we return a node, we're returning a path.

So, let's look at the different classes of search algorithms that we will be exploring. The simplest class is that of the uninformed, any-path algorithms. In particular, we will look at depth-first and breadth-first search. Both of these algorithms basically look at all the nodes in the search tree in a specific order (independent of the goal) and stop when they find the first path to a goal state.

<b>Classes of Search</b>		
<b>Class</b>	<b>Name</b>	<b>Operation</b>
<b>Any Path Uninformed</b>	<b>Depth-First Breadth-First</b>	Systematic exploration of whole tree until a goal node is found

The next class of methods are informed, any-path algorithms. The key idea here is to exploit a task specific measure of goodness to try to either reach the goal more quickly or find a more desirable goal state.

<b>Classes of Search</b>		
<b>Class</b>	<b>Name</b>	<b>Operation</b>
<b>Any Path Uninformed</b>	<b>Depth-First Breadth-First</b>	Systematic exploration of whole tree until a goal node is found
<b>Any Path Informed</b>	<b>Best-First</b>	Uses heuristic measure of goodness of a state, s estimated distance to goal



Next, we look at the class of uninformed, optimal algorithms. These methods guarantee finding the "best" path (as measured by the sum of weights on the graph edges) but do not use any information beyond what is in the graph definition.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, $s$ estimated distance to goal
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. *Finds "shortest" path

Finally, we look at informed, optimal algorithms, which also guarantee finding the best path but which exploit heuristic ("rule of thumb") information to find the path faster than the uninformed **methods**.

Classes of Search		
Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, $s$ estimated distance to goal
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. *Finds "shortest" path
Optimal	A*	Uses path "length" measure and heuristic Find "shortest" path



The search strategies we will look at are all instances of a common search algorithm, which is shown here. The basic idea is to keep a list (Q) of nodes (that is, partial paths), then to pick one such node from Q, see if it reaches the goal and otherwise extend that path to its neighbors and add them back to Q. Except for details, that's all there is to it.

Note, by the way, that we are keeping track of the states we have reached (visited) and not entering them in Q more than once. This will certainly keep us from ever looping, no matter how the underlying graph is connected, since we can only ever reach a state once. We will explore the impact of this decision later.

## Simple Search Algorithm

- A search node is a path from some state X to the start state, e g, XBAS)
- The state of a search node is the most recent state of the path, e g. X.
- Let Q be a list of search nodes, e g (X B A S) (C B A S) .
- Let S be the start state.
  1. Initialize Q with search node (S) as only entry; set Visited = ( S )
  2. If Q is empty, fail. Else, pick sane search node N from Q
  3. If state(N) is a goal, return N (we've reached the goal)
  4. (Otherwise) Remove N from Q
  5. Find all the descendants of state (N) not in Visited and create all the one- step extensions of N to each descendant.
  6. Add the extended paths to Q; add children of state(N) to Visited
  7. Go to step 2.

### Critical decision:

Step 2: Picking N from Q

Step 6: adding extensions of N to Q

The key questions, of course, are which entry to pick off of Q and how precisely to add the new paths back onto Q. Different choices for these operations produce the various search strategies.

At this point, we are ready to actually look at a specific search. For example, depth-first search always looks at the deepest node in the search tree first. We can get that behavior by:

- Picking the first element of Q as the node to test and extend.

- Adding the new (extended) paths to the FRONT of Q, so that the next path to be examined will be one of the extensions of the current path to one of the descendants of that node's state.

One good thing about depth-first search is that Q never gets very big. We will look at this in more detail later, but it's fairly easy to see that the size of the Q depends on the depth of the search tree and not on its breadth.

Breadth-first is the other major type of uninformed (or blind) search. The basic approach is to once again pick the first element of Q to examine BUT now we place the extended paths at the back of Q. This means that the next path pulled off of Q will typically not be a descendant of the current one, but rather one at the same level in tree.

Note that in breadth-first search, Q gets very big because we postpone looking at longer paths (that go to the next level) until we have finished looking at all the paths at one level.

We'll look at how to implement other search strategies in just a bit. But, first, let's look at some of the more subtle issues in the implementation.

One subtle point is where in the algorithm one tests for success (that is, the goal test). There are two plausible points: one is when a path is extended and it reaches a goal, the other is when a path is pulled off of Q. We have chosen the latter (testing in step 3 of the algorithm) because it will generalize more readily to optimal searches. However, testing on extension is correct and will save some work for any-path searches.

At this point, we need to agree on more terminology that will play a key role in the rest of our discussion of search.

Let's start with the notion of **Visited** as opposed to **Expanded**. We say a state is visited when a path that reaches that state (that is, a node that refers to that state) gets added to Q. So, if the state is anywhere in any node in Q, it has been visited. Note that this is true even if no path to that state has been taken off of Q.

A state  $M$  is **Expanded** when a path to that state is pulled off of  $Q$ . At that point, the descendants of  $M$  are visited and the paths to those descendants added to the  $Q$ .

In our description of the simple search algorithm, we made use of a Visited list. This is a list of all the states corresponding to any node ever added to  $Q$ . As we mentioned earlier, avoiding nodes on the visited list will certainly keep us from looping, even if the graph has loops in it. Note that this mechanism is stronger than just avoiding loops locally in every path; this is a global mechanism across all paths. In fact, it is more general than a loop check on each path, since by definition a loop will involve visiting a state more than once.

But, in addition to avoiding loops, the Visited list will mean that our search will never expand a state more than once. The basic idea is that we do not need to search for a path from any state to the goal more than once. If we did not find a path the first time we tried it, one is not going to materialize the second time. And, it saves work, possibly an enormous amount, not to look again. More on this later.

A word on implementation: Although we speak of a "Visited list", it is never a good idea to keep track of visited states using a list, since we will continually be checking to see if some particular state is on the list, which will require scanning the list. Instead, we want to use some mechanism that takes roughly constant time. If we have a data structure for the states, we can simply include a "flag" bit indicating whether the state has been visited. In general, one can use a hash table, a data structure that allows us to check if some state has been visited in roughly constant time, independent of the size of the table. Still, no matter how fast we make the access, this table will still require additional space to store. We will see later that this can make the cost of using a Visited list prohibitive for very large problems.

## **Terminology**

Another key concept to keep straight is that of a heuristic value for a state. The word heuristic generally refers to a "rule of thumb", something that's helpful but not guaranteed to work.

A heuristic function has similar connotations. It refers to a function (defined on a state - not on a path) that may be helpful in guiding search but which is not guaranteed

to produce the desired outcome. Heuristic searches generally make no guarantees on shortest paths or best anything (even when they are called best-first). Nevertheless, using heuristic functions may still provide help by speeding up, at least on average, the process of finding a goal.

If we can get some estimate of the "distance" to a goal from the current node and we introduce a preference for nodes closer to the goal, then there is a good chance that the search will terminate more quickly. This intuition is clear when thinking about "airline" (as-the-crow-flies) distance to guide a search in Euclidean space, but it generalizes to more abstract situations (as we will see).

## **Implementing the Search Strategies**

Best-first (also known as "greedy") search is a heuristic (informed) search that uses the value of a heuristic function defined on the states to guide the search. This will not guarantee finding a "best" path, for example, the shortest path to a goal. The heuristic is used in the hope that it will steer us to a quick completion of the search or to a relatively good goal state.

Best-first search can be implemented as follows: pick the "best" path (as measured by heuristic value of the node's state) from all of Q and add the extensions somewhere on Q. So, at any step, we are always examining the pending node with the best heuristic value.

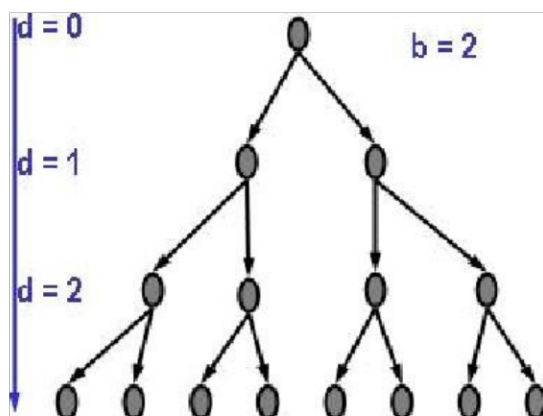
Note that, in the worst case, this search will examine all the same paths that depth or breadth first would examine, but the order of examination may be different and therefore the resulting path will generally be different. Best-first has a kind of breadth-first flavor and we expect that Q will tend to grow more than in depth-first search.

Note that best-first search requires finding the best node in Q. This is a classic problem in computer science and there are many different approaches that are appropriate in different circumstances. One simple method is simply to scan the Q completely, keeping track of the best element found. Surprisingly, this simple strategy turns out to be the right thing to do in some circumstances. A more sophisticated strategy, such as keeping a data structure called a "priority queue", is

more often the correct approach. We will pursue this issue further when we talk about optimal searches.

Let's think a bit about the worst case running time of the searches that we have been discussing. The actual running time, of course, will depend on details of the computer and of the software implementation. But, we can roughly compare the various algorithms by thinking of the number of nodes added to Q. The running time should be roughly proportional to this number.

In AI we usually think of a "typical" search space as being a tree with uniform branching factor  $b$  and depth  $d$ . The depth parameter may represent the number of steps in a plan of action or the number of moves in a game. The branching factor reflects the number of different choices that we have at each step. It is easy to see that the number of states in such a tree grows exponentially with the depth.



$d$  is depth

$b$  is branching factor

$$b^d < (b^{d+1} - 1) / (b - 1) < b^{d+1}$$

States in tree

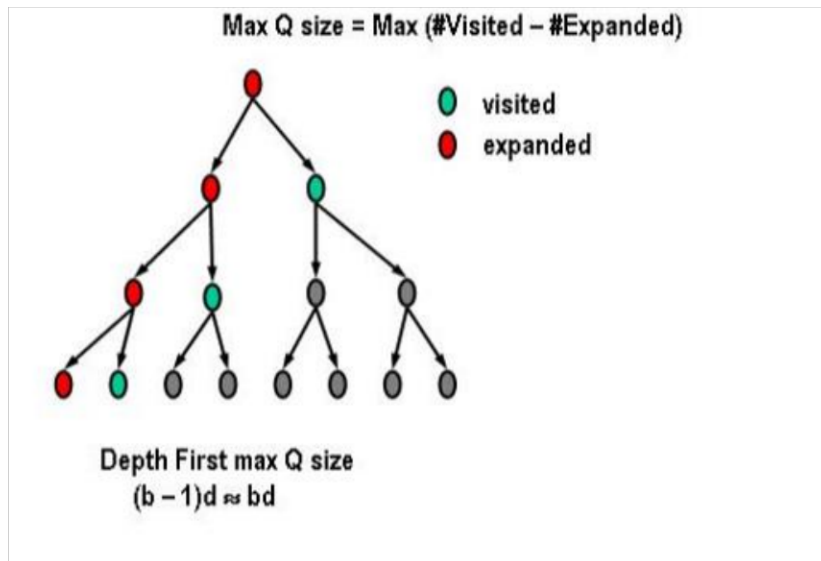
In a tree-structured search space, the nodes added to the search Q will simply correspond to the visited states. In the worst case, when the states are arranged in the worst possible way, all the search methods may end up having to visit or expand all of the states (up to some depth). In practice, we should be able to avoid this worst case but in many cases one comes pretty close to this worst case.

In addition to thinking about running time, we should also think about the memory space required for searches. The dominant factor in the space requirements for these searches is the maximum size of the search Q. The size of the search Q in a tree-structured search space is simply the number of visited states minus the number



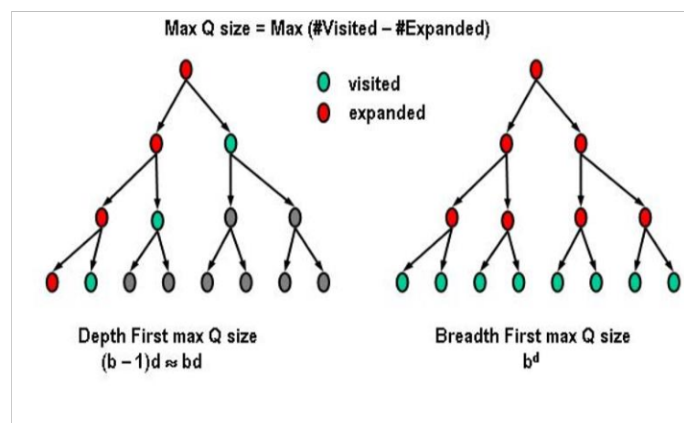
of expanded states.

For a depth-first search, we can see that Q holds the unexpanded "siblings" of the nodes along the path that we are currently considering. In a tree, the path length cannot be greater than  $d$  and the number of unexpanded siblings cannot be greater than  $b-1$ , so this tells us that the length of Q is always less than  $b*d$ , that is, the space requirements are linear in  $d$ .



The situation for breadth-first search is much different than that for depth-first search. Here the worst case happens after we've visited all the nodes at depth  $d-1$ . At that point, all the nodes at depth  $d$  have been visited and none expanded. So, the Q has size  $b^d$ , that is, a size exponential in  $d$ .

Note that, in the worst case, best-first behaves as breadth-first and has the same space requirements.



This table summarizes the key cost and performance properties of the different any-path search methods. We are assuming that our state space is a tree and so we cannot revisit states and a Visited list is useless.

**Searching a tree with branching factor  $b$  and depth  $d$   
(without using a Visited list)**

Search Method	Worst Time	Worst Space	Fewest states?	Guaranteed to find path?
Depth-First	$b^{d+1}$	$b^d$	No	Yes*
Breadth-first	$b^{d+1}$	$b^d$	Yes	Yes
Best-First	$b^{d+1}$ **	$b^d$	No	Yes*

\*If there are no infinitely long paths in the search space  
\*\* Best-First needs more time to locate the best node in  $Q$

**Worst case time is proportional to number of nodes added to  $Q$**   
**Worst case space is proportional to maximal length of  $Q$**

Recall that this analysis is done for searching a tree with uniform branching factor  $b$  and depth  $d$ . Therefore, the size of this search space grows exponentially with the depth. So, it should not be surprising that methods that guarantee finding a path will require exponential time in this situation. These estimates are not intended to be tight and precise; instead they are intended to convey a feeling for the tradeoffs.

Note that we could have phrased these results in terms of  $V$ , the number of vertices (nodes) in the tree, and then everything would have worst case behavior that is linear in  $V$ . We phrase it the way we do because in many applications, the number of nodes depends in an exponential way on some depth parameter, for example, the length of an action plan, and thinking of the cost as linear in the number of nodes is misleading. However, in the algorithms literature, many of these algorithms are described as requiring time linear in the number of nodes.

There are two points of interest in this table. One is the fact that depth-first search requires much less space than the other searches. This is important, since space tends to be the limiting factor in large problems (more on this later). The other is that the time cost of best-first search is higher than that of the others. This is due to the cost

of finding the best node in Q, not just the first one. We will also look at this in more detail later.

Remember that we are assuming in this slide that we are searching a tree, so states cannot be visited more than once - so the Visited list is completely superfluous when searching trees. However, if we were to use a Visited list (even implemented as a constant-time access hash table), the only thing that seems to change in this table is that the worst-case space requirements for all the searches go up (and way up for depth-first search). That does not seem to be very useful! Why would we ever use a Visited list?

**Searching a tree with branching factor  $b$  and depth  $d$   
(using a Visited list)**

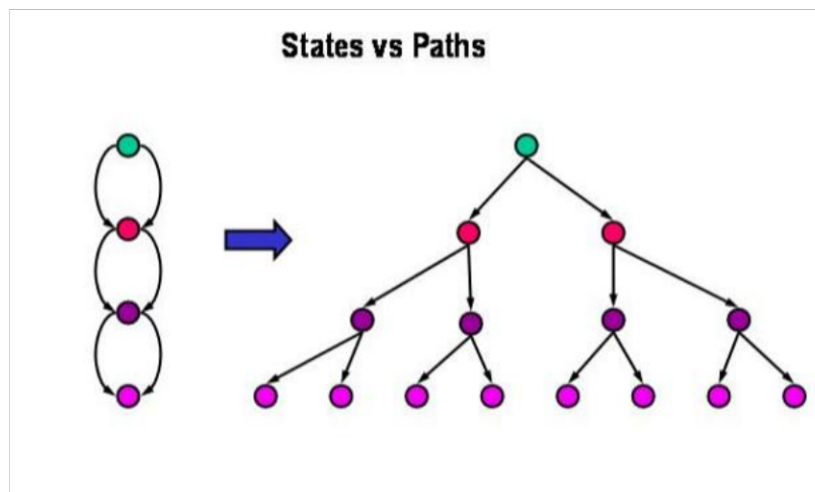
Search Method	Worst Time	Worst Space	Fewest states?	Guaranteed to find path?
Depth-First	$b^{d+1}$	<del><math>b^d</math></del> $b^{d+1}$	No	Yes*
Breadth-first	$b^{d+1}$	<del><math>b^d</math></del> $b^{d+1}$	Yes	Yes
Best-First	$b^{d+1}$ **	<del><math>b^d</math></del> $b^{d+1}$	No	Yes*

\*If there are no infinitely long paths in the search space  
\*\* Best-First needs more time to locate the best node in Q

**Worst case time is proportional to number of nodes added to Q**  
**Worst case space is proportional to maximal length of Q (and Visited list)**

As we mentioned earlier, the key observation is that with a Visited list, our worst-case time performance is limited by the number of states in the search space (since you visit each state at most once) rather than the number of paths through the nodes in the space, which may be exponentially larger than the number of states, as this classic example shows. Note that none of the paths in the tree have a loop in them, that is, no path visits a state more than once. The Visited list is a way of spending space to limit this time penalty. However, it may not be appropriate for very large search spaces where the space requirements would be prohibitive.





So far, we have been treating time and space in parallel for our algorithms. It is tempting to focus on time as the dominant cost of searching and, for real-time applications, it is. However, for large off-line applications, space may be the limiting factor.

If you do a back of the envelope calculation on the amount of space required to store a tree with branching factor 8 and depth 10, you get a very large number. Many real applications may want to explore bigger spaces.

$$(2^3)^{10} \times 2^3 = 2^{33} \text{ bytes} = 8,000 \text{ Mbytes} = 8\text{Gbytes}$$

One strategy for enabling such open-ended searches, which may run for a very long time, is Progressive Deepening Search (aka Iterative Deepening Search). The basic idea is to simulate searches with a breadth-like component by a succession of depth-limited depth-first searches. Since depth-first has negligible storage requirements, this is a clean tradeoff of time for space.

Interestingly, PDS is more than just a performance tradeoff. It actually represents a merger of two algorithms that combines the best of both. Let's look at that a little more carefully.

Depth-first search has one strong point - its limited space requirements, which are

linear in the depth of the search tree. Aside from that there's not much that can be said for it. In particular, it is susceptible to "going off the deep-end", that is, chasing very deep (possibly infinitely deep) paths. Because of this it does not guarantee, as breadth-first, does to find the shallowest goal states - those requiring the fewest actions to reach.

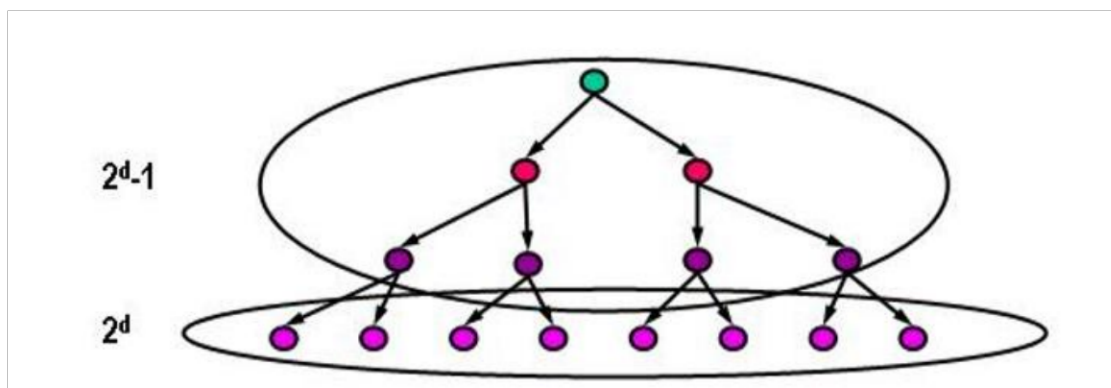
Breadth-first search on the other hand, does guarantee finding the shallowest goal, but at the expense of space requirements that are exponential in the depth of the search tree.

Progressive-deepening search, on the other other hand, has both limited space requirements of DFS and the strong optimality guarantee of BFS. Great! No?

At first sight, most people find PDS horrifying. Isn't progressive deepening really wasteful? It looks at the same nodes over and over again...

In small graphs, yes, it is wasteful. But, if we really are faced with an exponentially growing space (in the depth), then it turns out that the work at the deepest level dominates the total cost.

It is easy to see this for binary trees, where the number of nodes at level  $d$  is about equal to the number of nodes in the rest of the tree. The worst-case time for BFS at level  $d$  is proportional to the number of nodes at level  $d$ , while the worst case time for PDS at that level is proportional to the number of nodes in the whole tree which is almost exactly twice those at the deepest level. So, in the worst case, PDS (for binary trees) does no more than twice as much work as BFS, while using much less space.



This is a worst-case analysis, it turns out that if we try to look at the expected case, the situation is even better.

One can derive an estimate of the ratio of the work done by progressive deepening to that done by a single depth-first search:  $(b+1)/(b-1)$ . This estimate is for the average work (averaging over all possible searches in the tree). As you can see from the table, this ratio approaches one as the branching factor increases (and the resulting exponential explosion gets worse).

(Avg time for PDS)/(Avg time for DFS)  $\approx$   $(b+1)/(b-1)$

b	ratio
2	3
3	2
5	1.5
25	1.08
100	1.02

For many difficult searches, progressive deepening is in fact the only way to go. There are also progressive deepening versions of the optimal searches that we will see later, but that's beyond our scope.

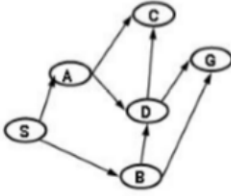
## Depth-First Search

We will now step through the any-path search methods looking at their implementation in terms of the simple algorithm. We start with depth-first search using a Visited list.

**Depth-First**

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1		
2		
3		
4		
5		

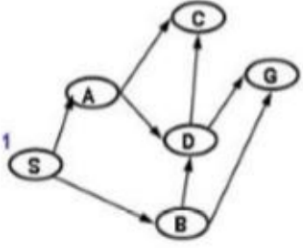


Added paths in blue  
 We show the paths in reversed order; the node's state is the first entry.

The table in the center shows the contents of Q and of the Visited list at each time through the loop of the search algorithm. The nodes in Q are indicated by reversed paths, blue is used to indicate newly added nodes (paths). On the right is the graph we are searching and we will label the state of the node that is being extended at each step.

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2		
3		
4		
5		



Added paths in blue  
 We show the paths in reversed order; the node's state is the first entry.

The first step is to initialize Q with a single node corresponding to the start state (S in this case) and the Visited list with the start state.

We pick the first element of Q, which is that initial node, remove it from Q, extend its path to its descendant states (if they have not been Visited) and add the resulting nodes to the front of Q. We also add the states corresponding to these new nodes to the Visited list. So, we get the situation on line 2.

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3		
4		
5		

Added paths in blue  
We show the paths in *reversed* order; the node's state is the first entry.

Note that the descendant nodes could have been added to Q in the other order. This would be absolutely valid. We will typically add nodes to Q in such a way that we end up visiting states in alphabetical order, when no other order is specified by the algorithm. This is purely an arbitrary decision.

We then pick the first node on Q, whose state is A, and repeat the process, extending to paths that end at C and D and placing them at the front of Q.

We pick the first node, whose state is C, and note that there are no descendants of C and so no new nodes to add.

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4		
5		

Added paths in blue  
We show the paths in *reversed* order; the node's state is the first entry.

We pick the first node of Q, whose state is D, and consider extending to states C and G, but C is on the Visited list so we do not add that extension. We do add the path to G to the front of Q.

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5		

Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

We pick the first node of Q, whose state is G, the intended goal state, so we stop and return the path.

Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S

Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

The final path returned goes from S to A, then to D and then to G.

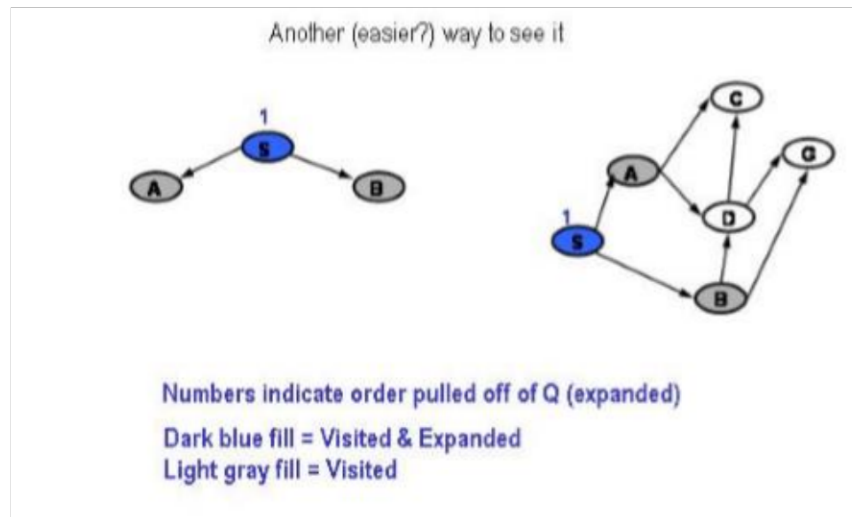
Pick first element of Q; Add path extensions to front of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C, D, B, A, S
4	(D A S) (B S)	C, D, B, A, S
5	(G D A S) (B S)	G, C, D, B, A, S

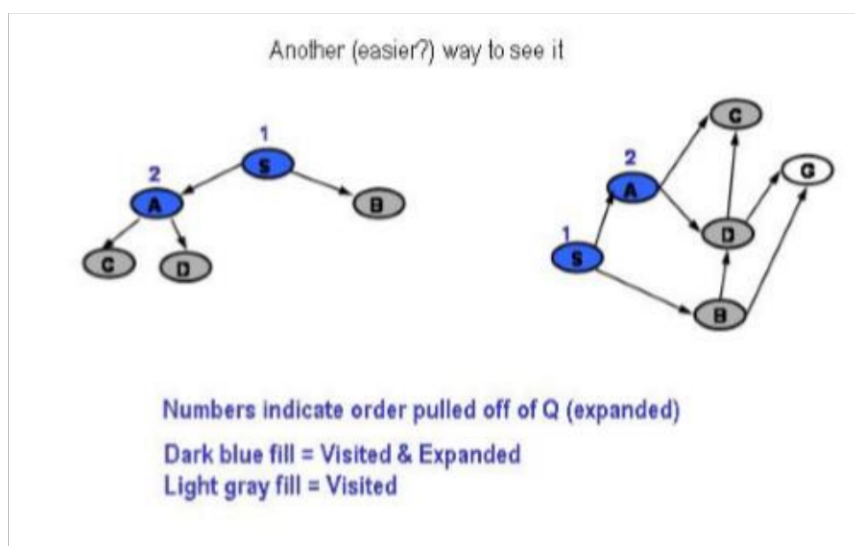
Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.



Tracing out the content of Q can get a little monotonous, although it allows one to trace the performance of the algorithms in detail. Another way to visualize simple searches is to draw out the search tree, as shown here, showing the result of the first expansion in the example we have been looking at.

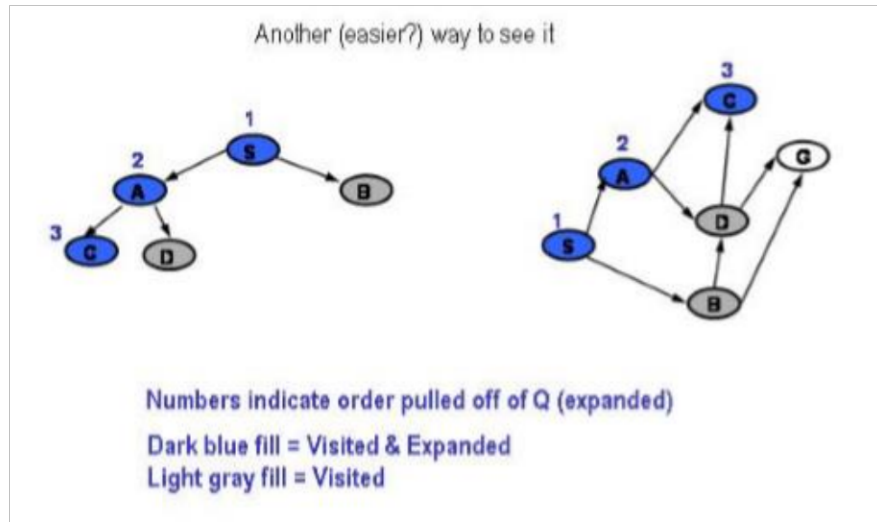


In this view, we introduce a left to right bias in deciding which nodes to expand—this is purely arbitrary. It corresponds exactly to the arbitrary decision of which nodes to add to Q first. Giving this bias, we decide to expand the node whose state is A, which ends up visiting C and D.

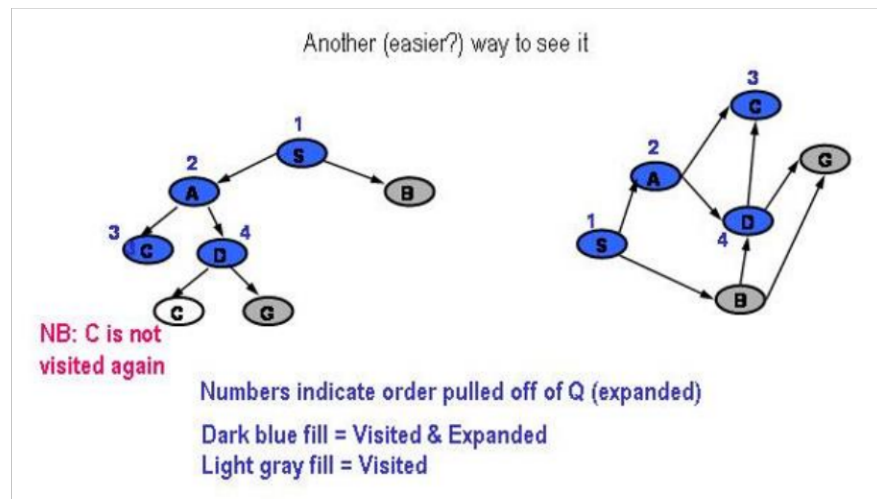


We now expand the node corresponding to C, which has no descendants, so we cannot continue to go deeper. At this point, one talks about having to **back up** or **backtrack** to the parent node and expanding any unexpanded descendant nodes of

the parent. If there were none at that level, we would continue to keep backing up to its parent and so on until an unexpanded node is found. We declare failure if we cannot find any remaining unexpanded nodes. In this case, we find an unexpanded descendant of A, namely D.

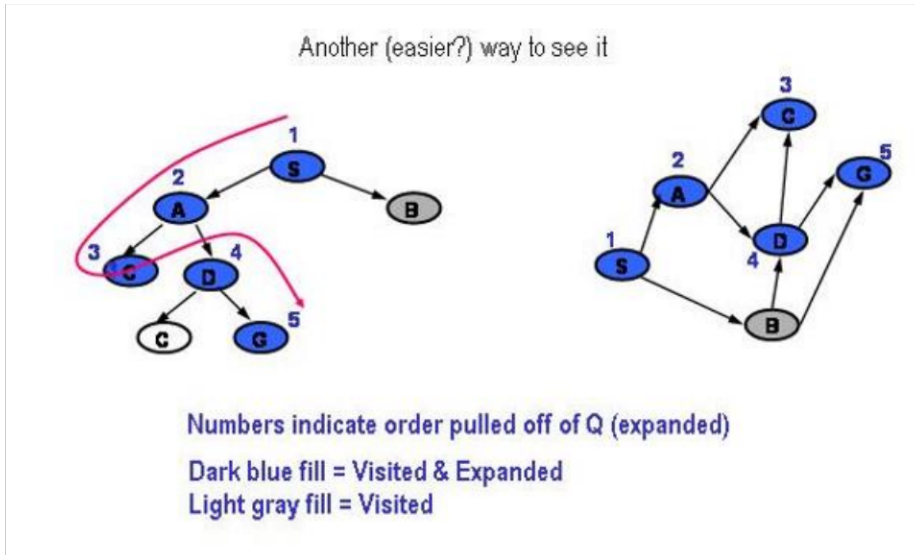


So, we expand D. Note that states C and G are both reachable from D. However, we have already visited C, so we do not add a node corresponding to that path. We add only the new node corresponding to the path to G.



We now expand G and stop.





This view of depth-first search is the more common one (rather than tracing Q). In fact, it is in this view that one can visualize why it is called depth-first search. The red arrow shows the sequence of expansions during the search and you can see that it is always going as deep in the search tree as possible. Also, we can understand another widely used name for depth-first search, namely backtracking search. However, you should convince yourself that this view is just a different way to visualize the behavior of the Q-based algorithm.

We can repeat the depth-first process without the Visited list and, as expected, one sees the second path to C added to Q, which was blocked by the use of the Visited list. I'll leave it as an exercise to go through the steps in detail.

Pick first element of Q; Add path extensions to front of Q

	Q
1	(S)
2	(A S) (B S)
3	(C A S) (D A S) (B S)
4	(D A S) (B S)
5	(C D A S) (G D A S) (B S)
6	(G D A S) (B S)

Added paths in blue  
 We show the paths in reversed order; the node's state is the first entry.  
 Do not extend a path to a state if the resulting path would have a loop.

Note that in the absence of a Visited list, we still require that we do not form any paths with loops, so if we have visited a state along a particular path, we do not re-visit that state again in any extensions of the path.

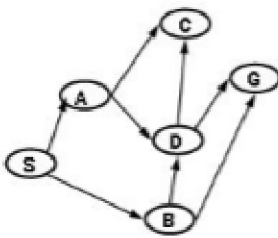
### Breadth-First Search

Let's look now at breadth-first search. The difference from depth-first search is that new paths are added to the back of Q. We start as with depth-first with the initial node corresponding to S.

**Breadth-First**

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2		
3		
4		
5		
6		

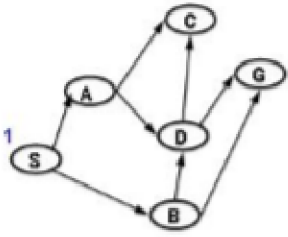


Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

We pick it and add paths to A and B, as before.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3		
4		
5		
6		

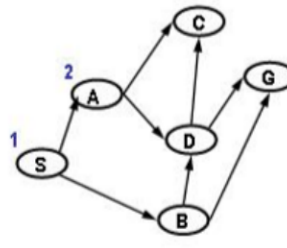


Added paths in blue  
We show the paths in reversed order; the node's state is the first entry.

We pick the first node, whose state is A, and extend the path to C and D and add them to Q (at the back) and here we see the difference from depth-first.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4		
5		
6		

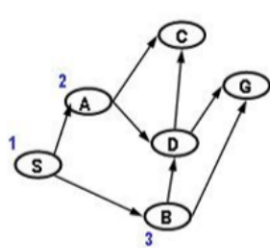


**Added paths in blue**  
We show the paths in **reversed** order; the node's state is the first entry.

Now, the first node in Q is the path to B so we pick that and consider its extensions to D and G. Since D is already Visited, we ignore that and add the path to G to the end of Q.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		



**Added paths in blue**  
We show the paths in **reversed** order; the node's state is the first entry.

At this point, having generated a path to G, we would be justified in stopping. But, as we mentioned earlier, we proceed until the path to the goal becomes the first path in Q.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		

Added paths in blue  
 We show the paths in *reversed* order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

We now pull out the node corresponding to C from Q but it does not generate any extensions since C has no descendants.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		

Added paths in blue  
 We show the paths in *reversed* order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

So we pull out the path to D. Its potential extensions are to previously visited states and so we get nothing added to Q.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		

**Added paths in blue**  
 We show the paths in **reversed** order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Finally, we get the path to G and we stop.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

**Added paths in blue**  
 We show the paths in **reversed** order; the node's state is the first entry.  
 \* We could have stopped here, when the first path to the goal was generated.

Note that we found a path with fewer states than we did with depth-first search, from S to B to G. In general, breadth-first search guarantees finding a path to the goal with the minimum number of states.

Pick first element of Q; Add path extensions to end of Q

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

Added paths in blue

We show the paths in **reversed** order; the node's state is the first entry.

\* We could have stopped here, when the first path to the goal was generated.

Here we see the behavior of breadth-first search in the search-tree view. In this view, you can see why it is called breadth-first -- it is exploring all the nodes at a single depth level of the search tree before proceeding to the next depth level.

Another (easier?) way to see it

NB: D is not visited again

Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

We can repeat the breadth-first process without the Visited list and, as expected, one sees multiple paths to C, D and G are added to Q, which were blocked by the Visited test earlier. I'll leave it as an exercise to go through the steps in detail.



Pick first element of Q; Add path extensions to end of Q

	Q
1	(S)
2	(A S) (B S)
3	(B S) (C A S) (D A S)
4	(C A S) (D A S) (D B S) (G B S)*
5	(D A S) (D B S) (G B S)
6	(D B S) (G B S) (C D A S) (G D A S)
7	(G B S) (C D A S) (G D A S) (C D B S) (G D B S)

Added paths in blue

We show the paths in **reversed** order; the node's state is the first entry.

\* We could have stopped here, when the first path to the goal was generated.

### Best-First Search

Finally, let's look at Best-First Search. The key difference from depth-first and breadth-first is that we look at the whole Q to find the best node (by heuristic value).

**Best-First**

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2		
3		
4		
5		

**Heuristic Values**

A=2	C=1	S=10
B=3	D=4	G=0

Added paths in blue; heuristic value of node's state is in front.

We show the paths in **reversed** order; the node's state is the first entry.

We start as before, but now we're showing the heuristic value of each path (which is the value of its state) in the Q, so we can easily see which one to extract next.

We pick the first node and extend to A and B.

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3		
4		
5		

Heuristic Values  
 A=2    C=1    S=10  
 B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in reversed order; the node's state is the first entry.

We pick the node corresponding to A, since it has the best value (= 2) and extend to C and D.

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4		
5		

Heuristic Values  
 A=2    C=1    S=10  
 B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in reversed order; the node's state is the first entry.

The node corresponding to C has the lowest value so we pick that one. That goes nowhere.



Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5		

Heuristic Values  
 A=2    C=1    S=10  
 B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in reversed order; the node's state is the first entry.

Then, we pick the node corresponding to B which has lower value than the path to D and extend to G (not C because of previous Visit).

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values  
 A=2    C=1    S=10  
 B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in reversed order; the node's state is the first entry.

We pick the node corresponding to G and rejoice.

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values  
 A=2    C=1    S=10  
 B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in reversed order; the node's state is the first entry.

We found the path to the goal from S to B to G.

Pick "best" (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S

Heuristic Values  
 A=2    C=1    S=10  
 B=3    D=4    G=0

Added paths in blue; heuristic value of node's state is in front.  
 We show the paths in reversed order; the node's state is the first entry.

## 1.2. Board Games & Search

In this section, we will look at some of the basic approaches for building programs that play two person games such as tic-tac-toe, checkers and chess.

Much of the work in this area has been motivated by playing chess, which has always been known as a "thinking person's game". The history of computer chess goes way back. Claude Shannon, the father of information theory, originated many of the ideas in a 1949 paper. Shortly after, Alan Turing did a hand simulation of a program to play checkers, based on some of these ideas. The first programs to play real chess didn't arrive until almost ten years later, and it wasn't until Greenblatt's

MacHack 6 that a computer chess program defeated a good player. Slow and steady progress eventually led to the defeat of reigning world champion Garry Kasparov against IBM's Deep Blue in May 1997.

### Game Tree Search

- Initial state: initial board position and player
- Operators: one for each legal move
- Goal states: winning board positions
- Scoring function: assigns numeric value to states
- Game tree: encodes all possible games

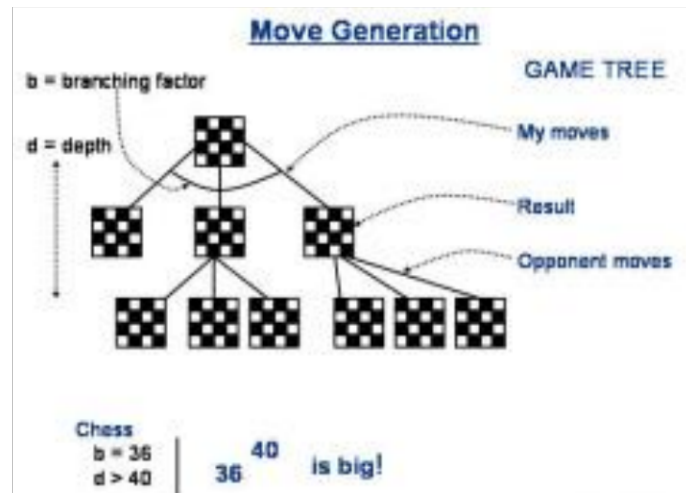
  

- We are not looking for a path, only the next move to make (that hopefully leads to a winning position)
- Our best move depends on what the other player does

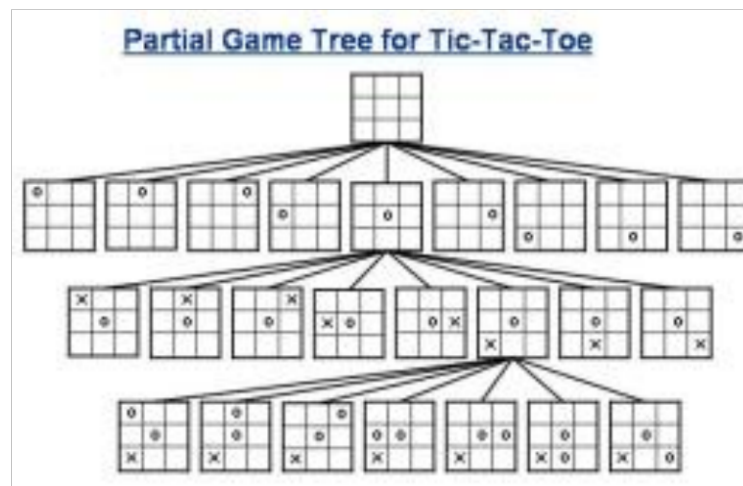
Game playing programs are another application of search. The states are the board positions (and the player whose turn it is to move). The operators are the legal moves. The goal states are the winning positions. A scoring function assigns values to states and also serves as a kind of heuristic function. The game tree (defined by the states and operators) is like the search tree in a typical search and it encodes all possible games.

There are a few key differences, however. For one thing, we are not looking for a path through the game tree, since that is going to depend on what moves the opponent makes. All we can do is choose the best move to make next.

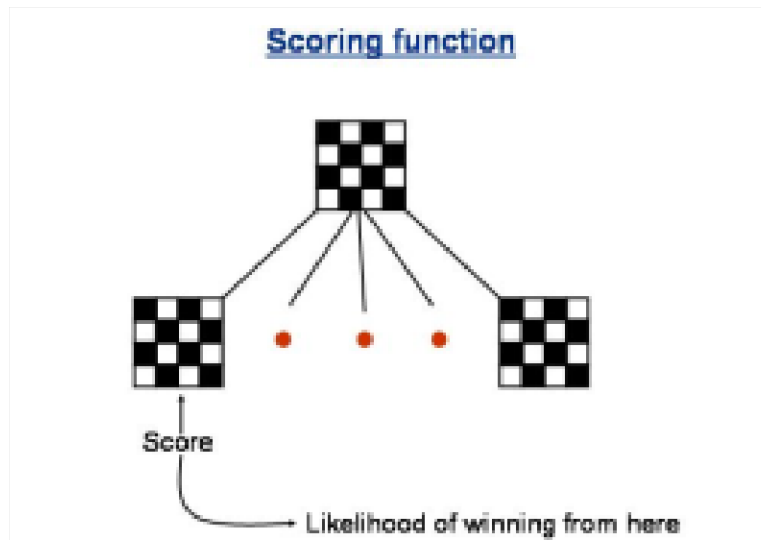
Let's look at the game tree in more detail. Some board position represents the initial state and it's now our turn. We generate the children of this position by making all of the legal moves available to us. Then, we consider the moves that our opponent can make to generate the descendants of each of these positions, etc. Note that these trees are enormous and cannot be explicitly represented in their entirety for any complex game.



Here's a little piece of the game tree for Tic-Tac-Toe, starting from an empty board. Note that even for this trivial game, the search tree is quite big.



A crucial component of any game playing program is the scoring function. This function assigns a numerical value to a board position. We can think of this value as capturing the likelihood of winning from that position. Since in these games one person's win is another's person loss, we will use the same scoring function for both players, simply negating the values to represent the opponent's scores.



A typical scoring function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position. Each feature is also a number that measures some characteristic of the position. One that is easy to see is "material", that is, some measure of which pieces one has on the board. A typical weighting for each type of chess piece is shown here. Other types of features try to encode something about the distribution of the pieces on the board.

In some sense, if we had a perfect evaluation function, we could simply play chess by evaluating the positions produced by each of our legal moves and picking the one with the highest score. In principle, such a function exists, but no one knows how to write it or compute it directly.

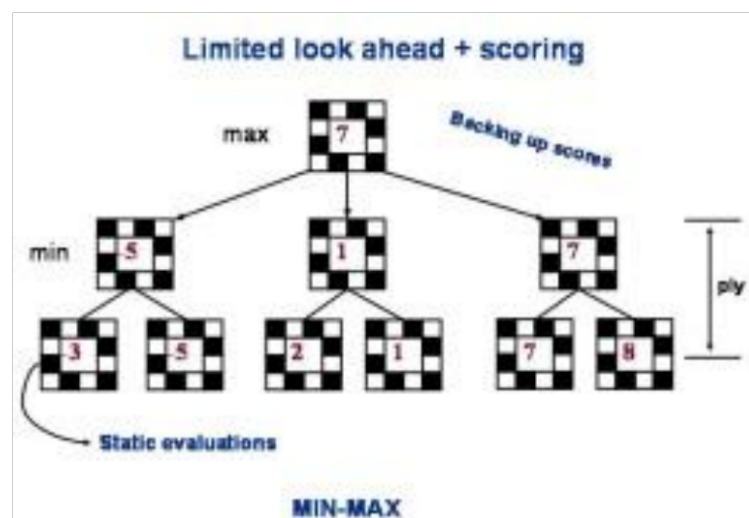
**Static Evaluation**

$S =$	$c_1$	$x$	material	<b>P 1</b> <b>K 3</b> <b>B 3.5</b> <b>R 5</b> <b>Q 9</b>
+	$c_2$	$x$	pawn structure	
+	$c_3$	$x$	mobility	
+	$c_4$	$x$	king safety	
+	$c_5$	$x$	center control	
+	...			

Too weak to predict ultimate success

Let's imagine that we are going to look ahead in the game-tree to a depth of 2 (or 2 ply as it is called in the literature on game playing). We can use our scoring function to see what the values are at the leaves of this tree. These are called the "static evaluations". What we want is to compute a value for each of the nodes above this one in the tree by "backing up" these static evaluations in the tree.

The player who is building the tree is trying to maximize their score. However, we assume that the opponent (who values board positions using the same static evaluation function) is trying to minimize the score (or think of this as maximizing the negative of the score). So, each layer of the tree can be classified into either a maximizing layer or a minimizing layer. In our example, the layer right above the leaves is a minimizing layer, so we assign to each node in that layer the minimum score of any of its children. At the next layer up, we're maximizing so we pick the maximum of the scores available to us, that is, 7. So, this analysis tells us that we should pick the move that gives us the best guaranteed score, independent of what our opponent does. This is the MIN-MAX algorithm.



Here is pseudo-code that implements Min-Max. As you can see, it is a simple recursive alternation of maximization and minimization at each layer. We assume that we count the depth value down from the max depth so that when we reach a depth of 0, we apply our static evaluation to the board.



## Min-Max

@ initial call is MAX-VALUE(state, MAX-DEPTH)

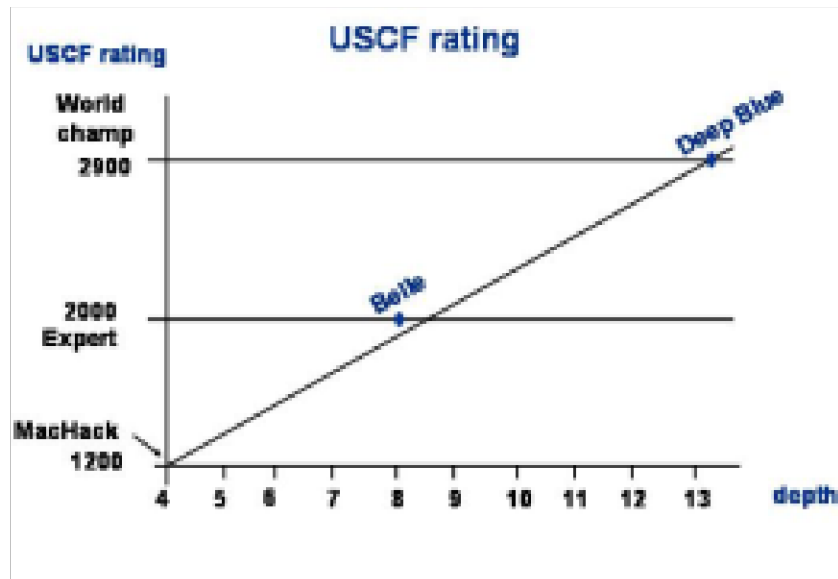
```
function MAX-VALUE (state, depth)
  if (depth == 0) then return EVAL (state)
  v = -∞
  for each s in SUCCESSORS (state) do
    v = MAX (v, MIN-VALUE (s, depth-1))
  end
  return v
```

```
function MIN-VALUE (state, depth)
  if (depth == 0) then return EVAL (state)
  v = ∞
  for each s in SUCCESSORS (state) do
    v = MIN (v, MAX-VALUE (s, depth-1))
  end
  return v
```

The key idea is that the more lookahead we can do, that is, the deeper in the tree we can look, the better our evaluation of a position will be, even with a simple evaluation function. In some sense, if we could look all the way to the end of the game, all we would need is an evaluation function that was 1 when we won and -1 when the opponent won.

The truly remarkable thing is how well this idea works. If you plot how deep computer programs can search chess game trees versus their ranking, we see a graph that looks something like this. The earliest serious chess program (MacHack6), which had a ranking of 1200, searched on average to a depth of 4. Belle, which was one of the first hardware-assisted chess programs doubled the depth and gained about 800 points in ranking. Deep Blue, which searched to an average depth of about 13 beat the world champion with a ranking of about 2900.

At some level, this is a depressing picture, since it seems to suggest that brute-force search is all that matters.



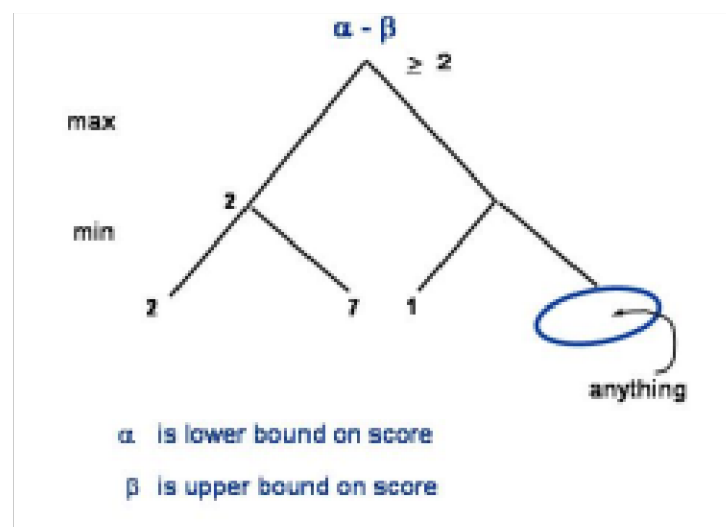
And Deep Blue is brute indeed... It had 256 specialized chess processors coupled into a 32-node supercomputer. It examined around 30 billion moves per minute. The typical search depth was 13 ply, but in some dynamic situations it could go as deep as 30.



There's one other idea that has played a crucial role in the development of computer game-playing programs. It is really only an optimization of Min-Max search, but it is such a powerful and important optimization that it deserves to be understood in detail. The technique is called alpha-beta pruning, from the Greek letters traditionally used to represent the lower and upper bound on the score.

Here's an example that illustrates the key idea. Suppose that we have evaluated the sub-tree on the left (whose leaves have values 2 and 7). Since this is a minimizing level, we choose the value 2. So, the maximizing player at the top of the tree knows at this point that he can guarantee a score of at least 2 by choosing the move on the left.

Now, we proceed to look at the subtree on the right. Once we look at the leftmost leaf of that subtree and see a 1, we know that if the maximizing player makes the move to the right then the minimizing player can force him into a position that is worth no more than 1. In fact, it might be much worse. The next leaf we look at might bring an even nastier surprise, but it doesn't matter what it is: we already know that this move is worse than the one to the left, so why bother looking any further? In fact, it may be that this unknown position is a great one for the maximizer, but then the minimizer would never choose it. So, no matter what happens at that leaf, the maximizer's choice will not be affected.



Here's some pseudo-code that captures this idea. We start out with the range of possible scores (as defined by alpha and beta) going from minus infinity to plus infinity. Alpha represents the lower bound and beta the upper bound. We call Max-Value with the current board state. If we are at a leaf, we return the static value. Otherwise, we look at each of the successors of this state (by applying the legal move function) and for each successor, we call the minimizer (Min-Value) and we keep track of the maximum value returned in alpha. If the value of alpha (the lower bound on the score) ever gets to be greater or equal to beta (the upper bound) then we know

that we don't need to keep looking - this is called a cutoff - and we return alpha immediately. Otherwise, we return alpha at the end of the loop. The minimizer is completely symmetric.

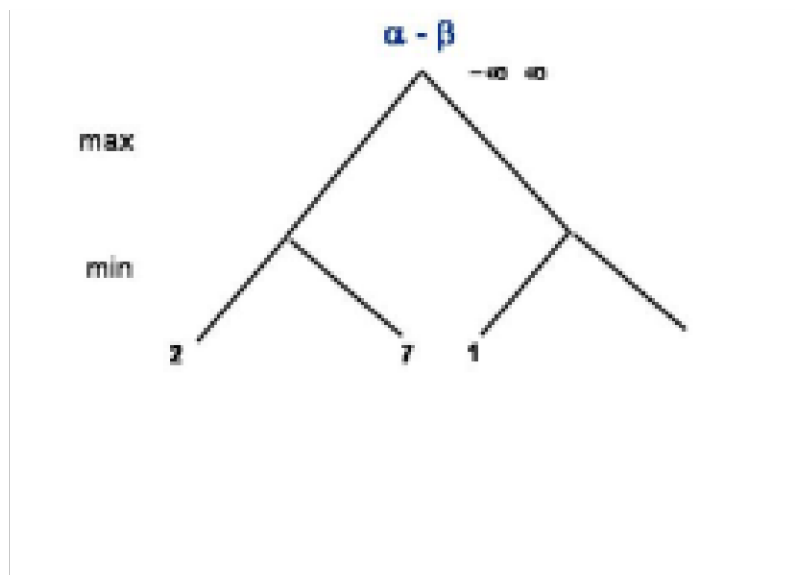
```
 $\alpha - \beta$ 
```

*//  $\alpha$  = best score for MAX,  $\beta$  = best score for MIN  
 // initial call is MAX-VALUE(state, - $\infty$ ,  $\infty$ , MAX-DEPTH)*

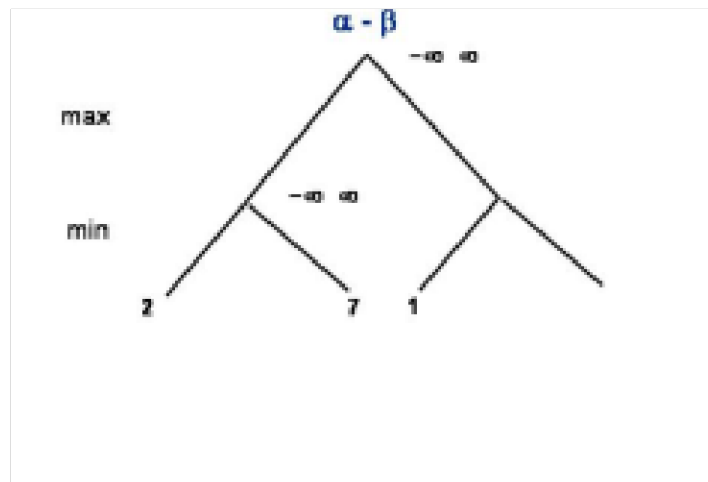
```
function MAX-VALUE (state,  $\alpha$ ,  $\beta$ , depth)
  if (depth == 0) then return EVAL (state)
  for each s in SUCCESSORS (state) do
     $\alpha$  = MAX ( $\alpha$ , MIN-VALUE (s,  $\alpha$ ,  $\beta$ , depth-1))
    if  $\alpha \geq \beta$  then return  $\alpha$  // cutoff
  end
  return  $\alpha$ 
```

```
function MIN-VALUE (state,  $\alpha$ ,  $\beta$ , depth)
  if (depth == 0) then return EVAL (state)
  for each s in SUCCESSORS (state) do
     $\beta$  = MIN ( $\beta$ , MAX-VALUE (s,  $\alpha$ ,  $\beta$ , depth-1))
    if  $\beta \leq \alpha$  then return  $\beta$  // cutoff
  end
  return  $\beta$ 
```

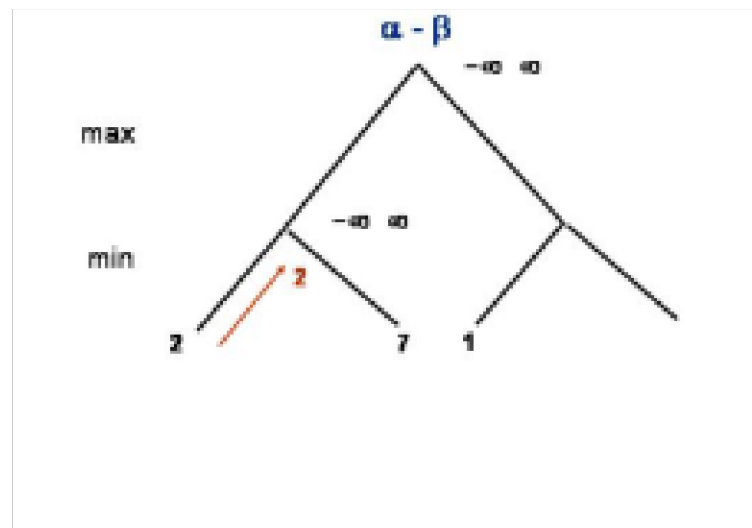
Let's look at this program in operation on our previous example. We start with an initial call to MaxValue with the initial infinite values of alpha and beta, meaning that we know nothing about what the score is going to be.



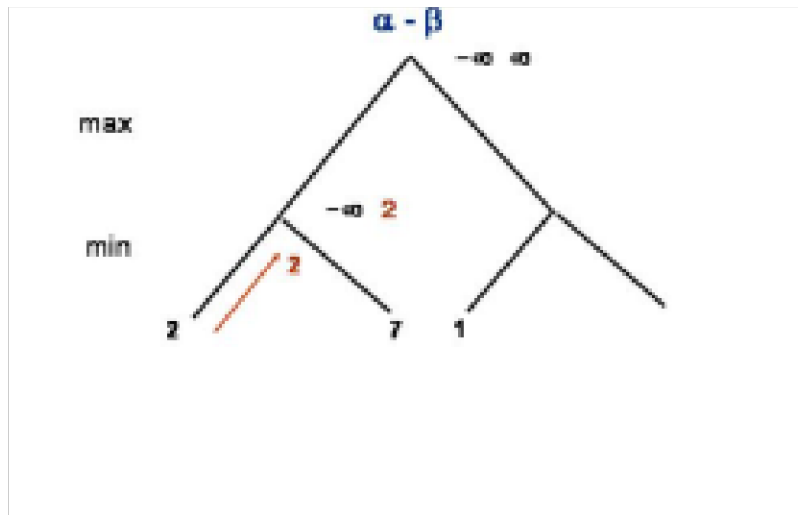
Max-Value now calls Min-Value on the left successor with the same values of alpha and beta. Min-Value now calls Max-Value on its leftmost successor.



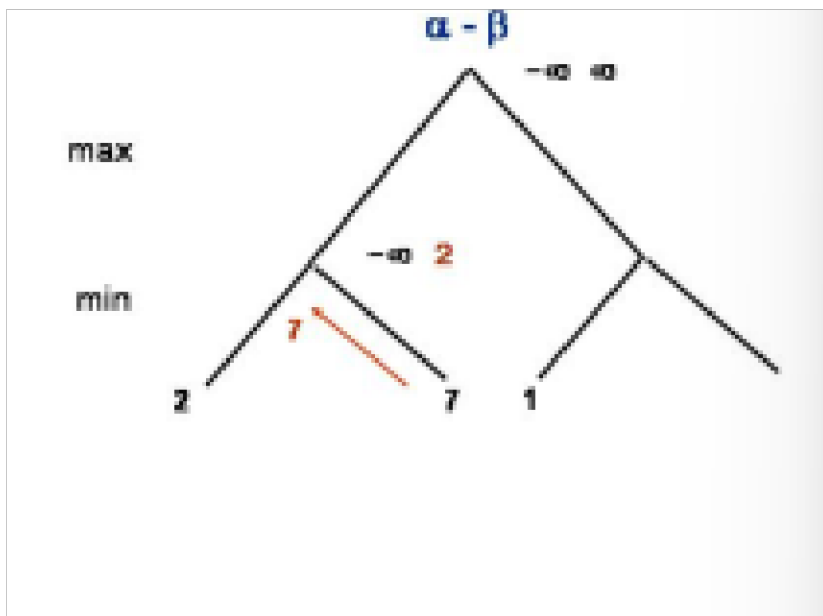
Max-Value is at the leftmost leaf, whose static value is 2 and so it returns that.



This first value, since it is less than infinity, becomes the new value of beta in Min-Value.

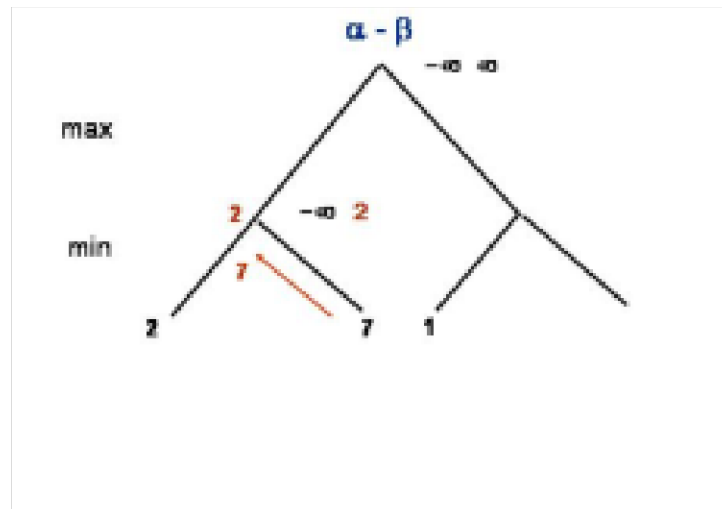


So, now we call Max-Value with the next successor, which is also a leaf whose value is 7.

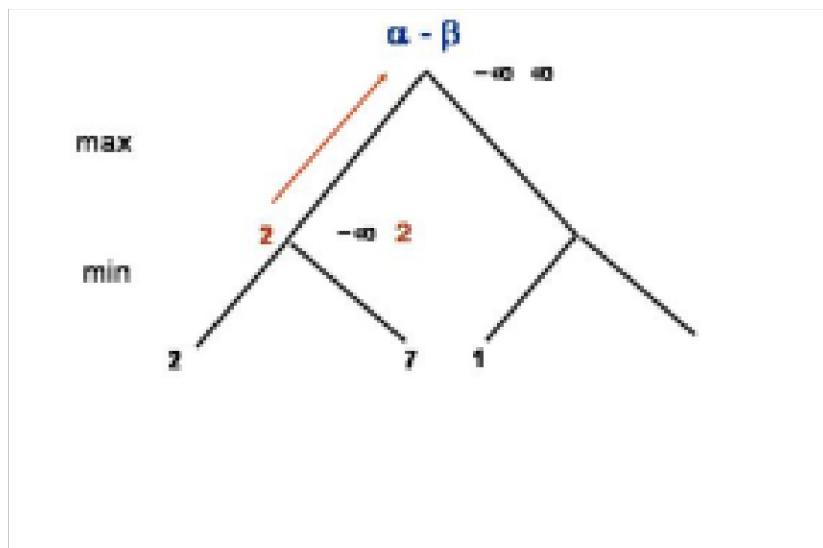


7 is not less than 2 and so the final value of beta is 2 for this node.

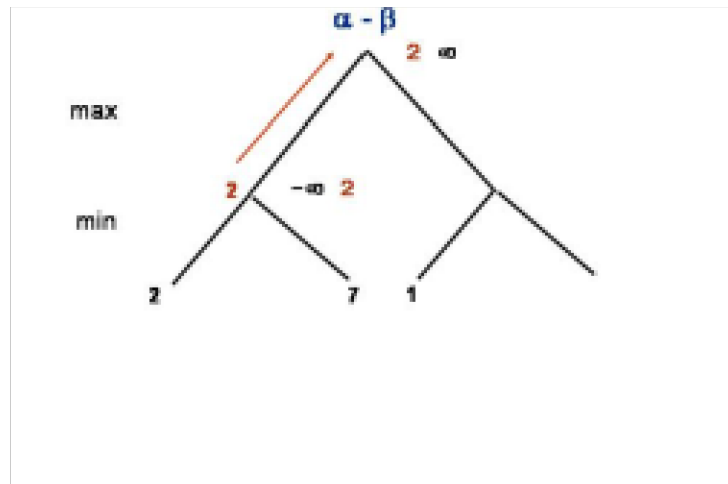




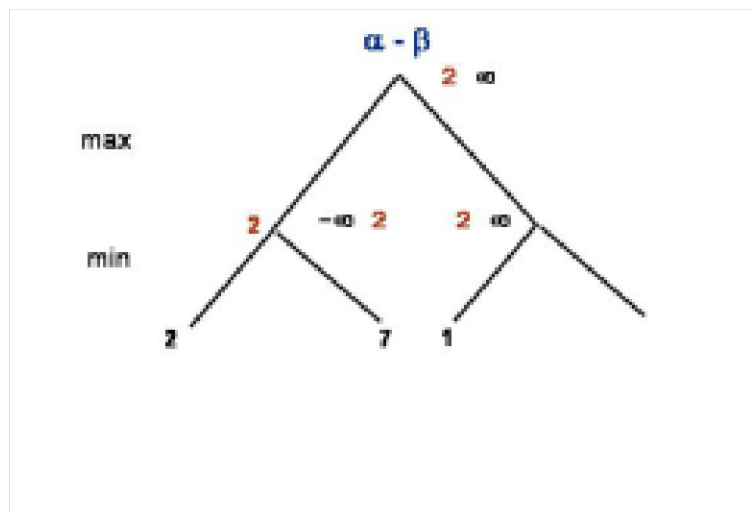
Min-Value now returns this value to its caller.



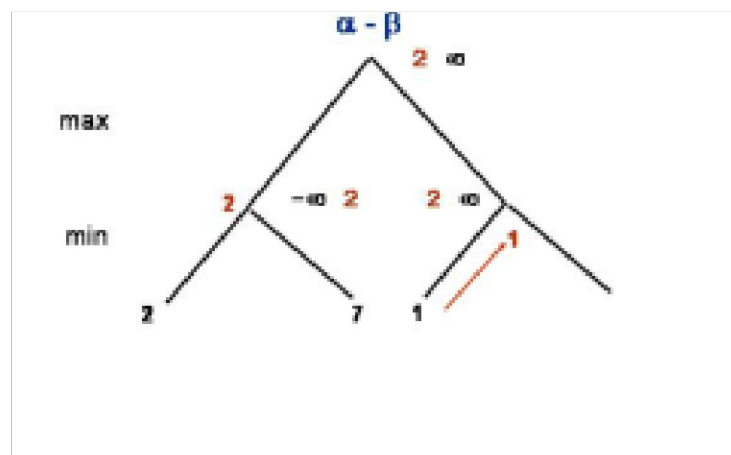
The calling Max-Value now sets alpha to this value, since it is bigger than minus infinity. Note that the range of  $[\alpha \beta]$  says that the score will be greater or equal to 2 (and less than infinity).



Max-Value now calls Min-Value with the updated range of [alpha beta].

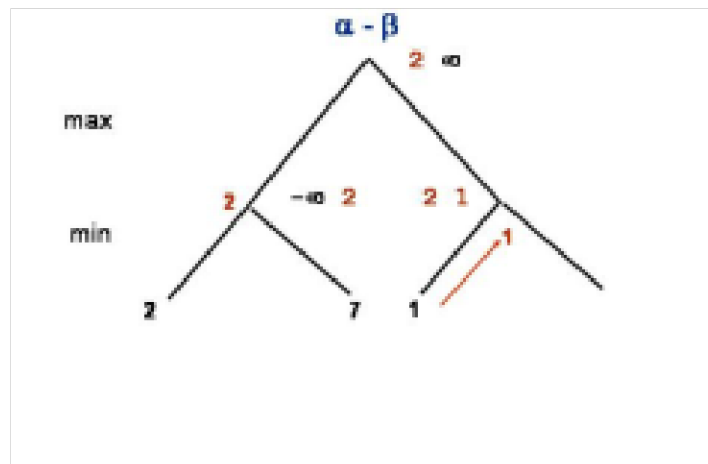


Min-Value calls Max-Value on the left leaf and it returns a value of 1.

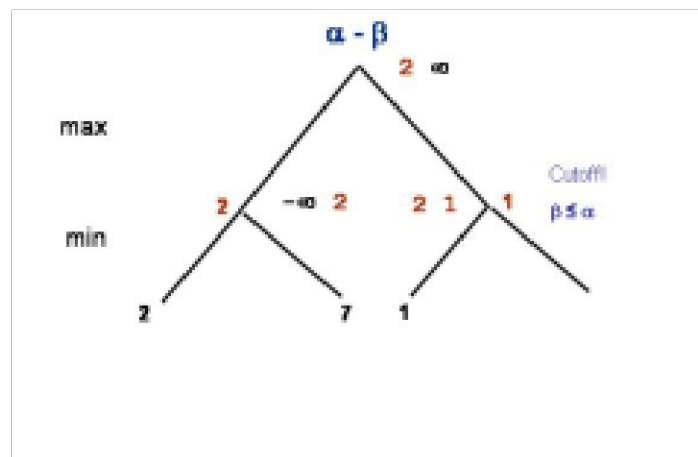


This is used to update beta in Min-Value, since it is less than infinity. Note that

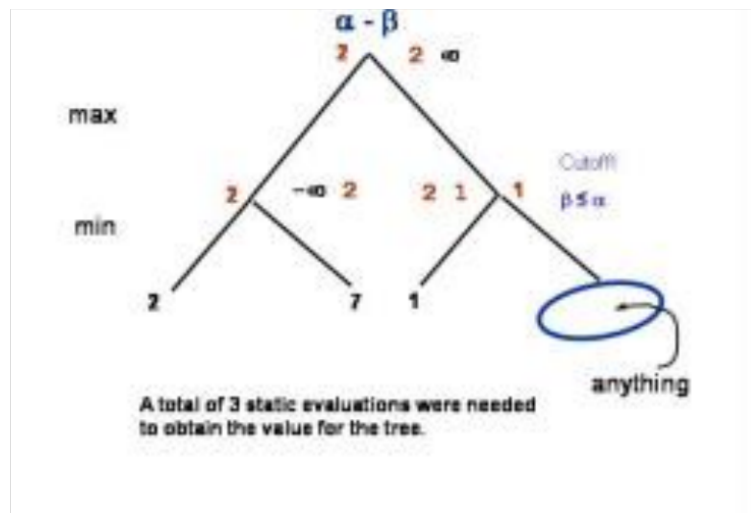
at this point we have a range where alpha (2) is greater than beta (1).



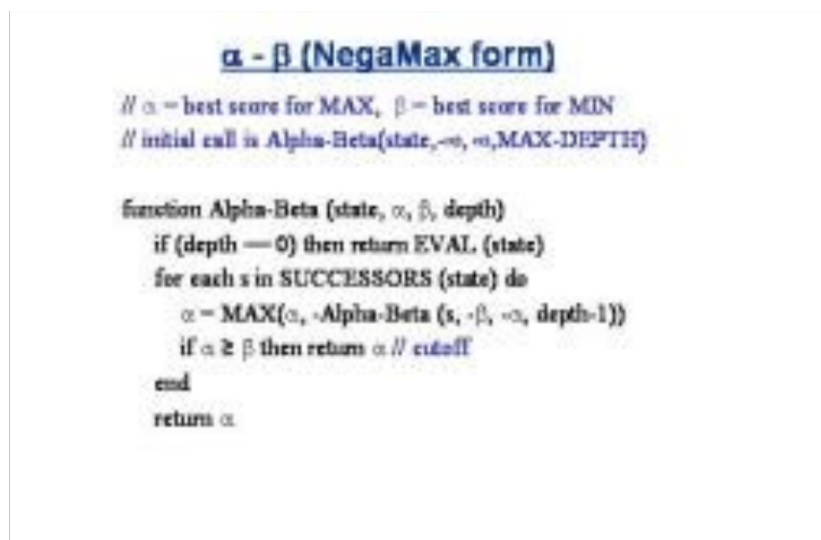
This situation signals a cutoff in Min-Value and it returns beta (1), without looking at the right leaf.



So, basically, we had already found a move that guaranteed us a score greater or equal to 2 so that when we got into a situation where the score was guaranteed to be less than or equal to 1, we could stop. So, a total of 3 static evaluations were needed instead of the four we would have needed under pure Min-Max.



We can write alpha-beta in a more compact form that captures the symmetry between the MaxValue and Min-Value procedures. This is sometimes called the NegaMax form (instead of the MinMax form). Basically, this exploits the idea that minimizing is the same as maximizing the negatives of the scores.



There are a couple of key points to remember about alpha-beta pruning. It is guaranteed to return exactly the same value as the Min-Max algorithm. It is a pure optimization without any approximations or tradeoffs.

In a perfectly ordered tree, with the best moves on the left, alpha beta reduces the cost of the search from order  $bd$  to order  $b^{(d/2)}$ , that is, we can search twice as deep! We already saw the enormous impact of deeper search on performance. So, this one simple algorithm can almost double the search depth.

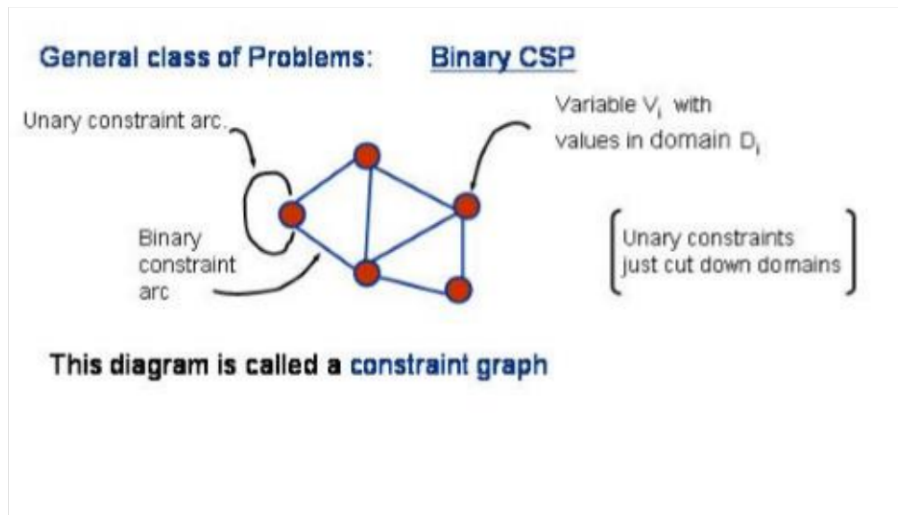
Now, this analysis is optimistic, since if we could order moves perfectly, we would not need alpha-beta. But, in practice, performance is close to the optimistic limit.

- $\alpha - \beta$
1. Guaranteed same value as Max-Min
  2. In a perfectly ordered tree, expected work is  $O(b^{d/2})$ , vs  $O(b^d)$  for Max-Min, so can search twice as deep with the same effort!
  3. With good move ordering, the actual running time is close to the optimistic estimate.

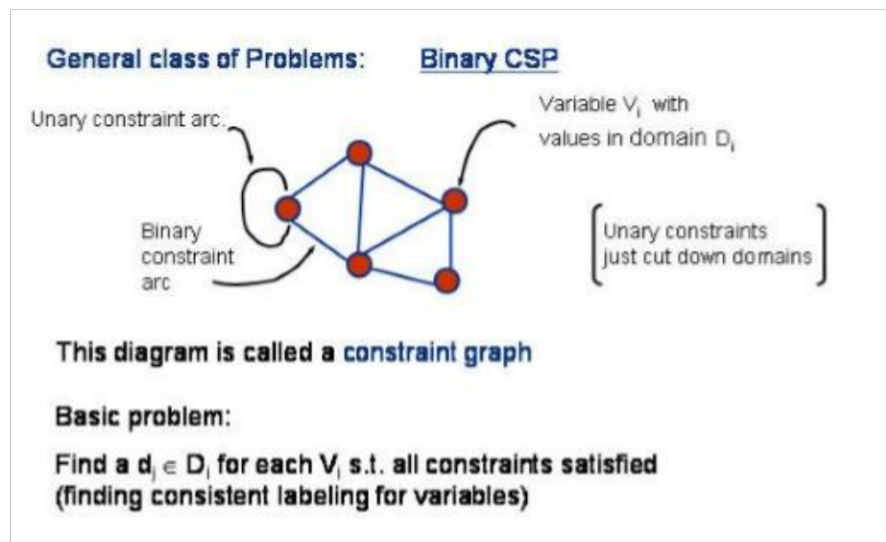
### 1.3. Constraint Satisfaction Problems

In this presentation, we'll take a look at the class of problems called Constraint Satisfaction Problems (CSPs). CSPs arise in many application areas: they can be used to formulate scheduling tasks, robot planning tasks, puzzles, molecular structures, sensory interpretation tasks, etc.

In particular, we'll look at the subclass of Binary CSPs. A binary CSP is described in term of a set of Variables (denoted  $V_i$ ), a domain of Values for each of the variables (denoted  $D_i$ ) and a set of constraints involving the combinations of values for two of the variables (hence the name "binary"). We'll also allow "unary" constraints (constraints on a single variable), but these can be seen simply as cutting down the domain of that variable. We can illustrate the structure of a CSP in a diagram, such as this one, that we call a **constraint graph** for the problem.



The solution of a CSP involves finding a value for each variable (drawn from its domain) such that all the constraints are satisfied. Before we look at how this can be done, let's look at some examples of CSP.



A CSP that has served as a sort of benchmark problem for the field is the so-called N-Queens problem, which is that of placing N queens on an  $N \times N$  chessboard so that no two queens can attack each other.

One possible formulation is that the variables are the chessboard positions and the values are either Queen or Blank. The constraints hold between any two variables representing positions that are on a line. The constraint is satisfied whenever the two values are not both Queen.



**N-Queens as CSP**  
Classic "benchmark" problem

Place N queens on an NxN chessboard so that none can attack the other.

**Variables** are board positions in NxN chessboard

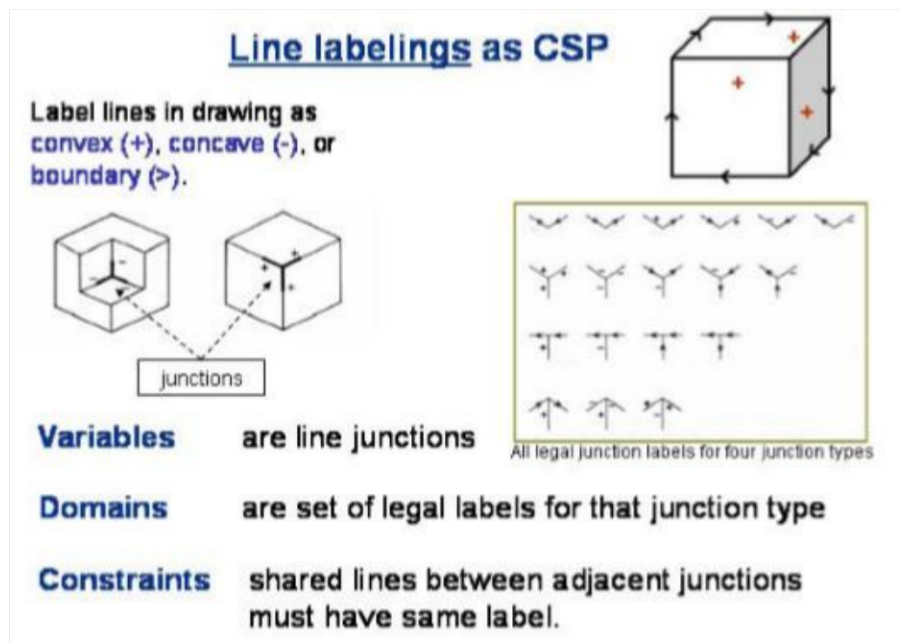
**Domains** Queen or blank

**Constraints** Two positions on a line (vertical, horizontal, diagonal) cannot both be Q

This formulation is actually very wasteful, since it has  $N^2$  variables. A better formulation is to have variables correspond to the columns of the board and values to the index of the row where the Queen for that column is to be placed. Note that no two queens can share a column and that every column must have a Queen on it. This choice requires only  $N$  variables and also fewer constraints to be checked.

In general, we'll find that there are important choices in the formulation of a CSP.

The problem of labeling the lines in a line-drawing of blocks as being either convex, concave or boundary, is the problem that originally brought the whole area of CSPs into prominence. Waltz's approach to solving this problem by propagation of constraints (which we will discuss later) motivated much of the later work in this area.



In this problem, the variables are the junctions (that is, the vertices) and the values are a combination of labels (+, -, >) attached to the lines that make up the junction. Some combinations of these labels are physically realizable and others are not. The basic constraint is that junctions that share a line must agree on the label for that line.

Note that the more natural formulation that uses lines as the variables is not a BINARY CSP, since all the lines coming into a junction must be simultaneously constrained.

Scheduling actions that share resources is also a classic case of a CSP. The variables are the activities, the values are chunks of time and the constraints enforce exclusion on shared resources as well as proper ordering of the tasks.

### Scheduling as CSP

Choose time for activities e.g. observations on Hubble telescope, or terms to take required classes.

**Variables** are activities

**Domains** sets of start times (or "chunks" of time)

**Constraints**

1. Activities that use same resource cannot overlap in time
2. Preconditions satisfied

Another classic CSP is that of coloring a graph given a small set of colors. Given a set of regions with defined neighbors, the problem is to assign a color to each region so that no two neighbors have the same color (so that you can tell where the boundary is). You might have heard of the famous Four-Color Theorem that shows that four colors are sufficient for any planar map. This theorem was a conjecture for more than a century and was not proven until 1976. The CSP is not proving the general theorem, just constructing a solution to a particular instance of the problem.

### Graph Coloring as CSP

Pick colors for map regions, avoiding coloring adjacent regions with the same color

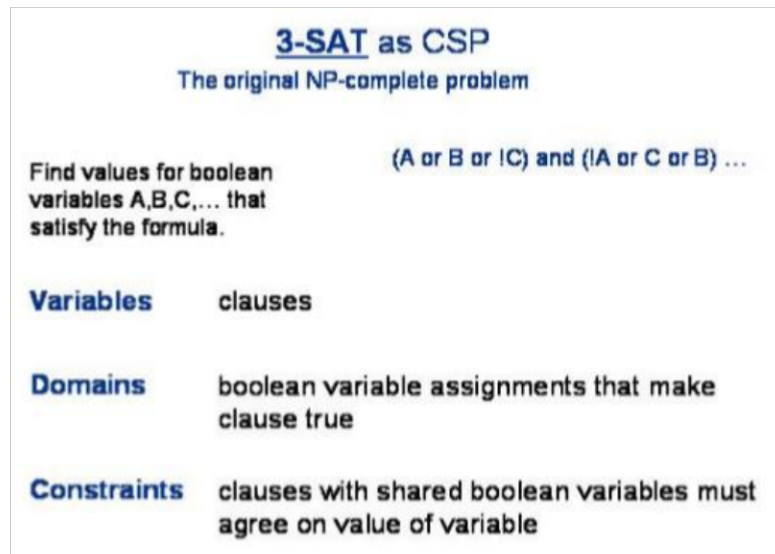
**Variables** regions

**Domains** colors allowed

**Constraints** adjacent regions must have different colors

A very important class of CSPs is the class of boolean satisfiability problems. One is given a formula over boolean variables in conjunctive normal form (a set of ORs connected with ANDs). The objective is to find an assignment that makes the

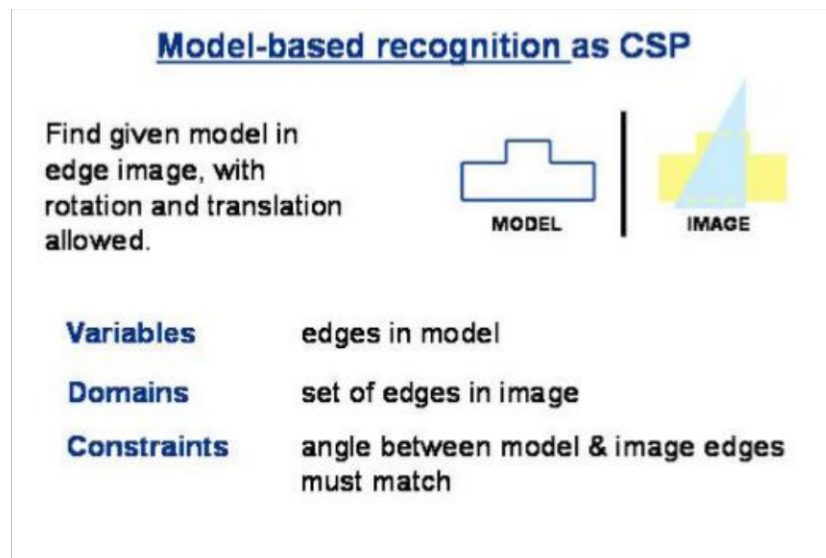
formula true, that is, a satisfying assignment.



SAT problems are easily transformed into the CSP framework. And, it turns out that many important problems (such as constructing a plan for a robot and many circuit design problems) can be turned into (huge) SAT problems. So, a way of solving SAT problems efficiently in practice would have great practical impact.

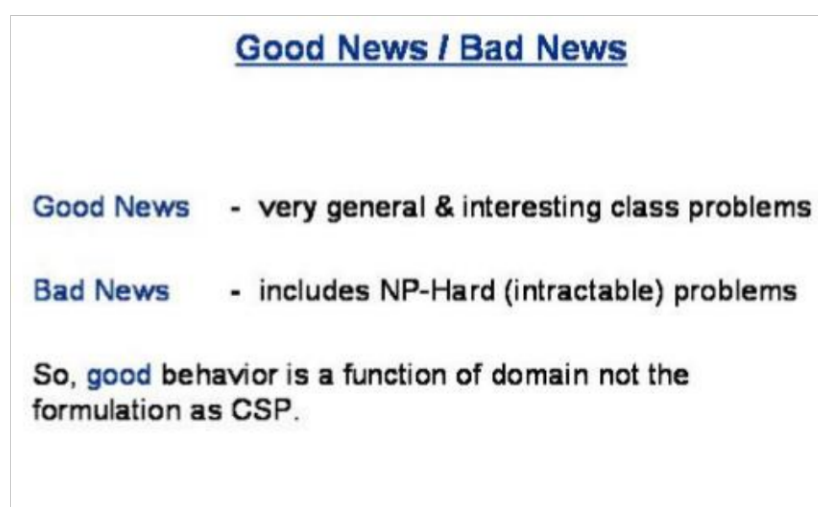
However, SAT is the problem that was originally used to show that some problems are NP-complete, that is, as hard as any problem whose solution can be checked in polynomial time. It is generally believed that there is no polynomial time algorithm for NP-complete problems. That is, that any guaranteed algorithm has a worst-case running time that grows exponentially with the size of the problem. So, at best, we can only hope to find a heuristic approach to SAT problems. More on this later.

Model-based recognition is the problem of finding an instance of a known geometric model, described, for example, as a line-boundary in an image which has been pre-processed to identify and fit lines to the boundaries. The position and orientation of the instance, if any, is not known.

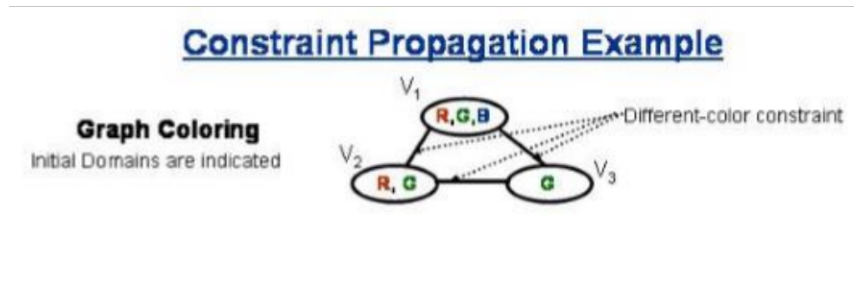


There are a number of constraints that need to be satisfied by edges in the image that correspond to edges in the model. Notably, the angles between pairs of edges must be preserved.

So, looking through these examples of CSPs we have some good news and bad news. The good news is that CSP is a very general class of problems containing many interesting practical problems. The bad news is that CSPs include many problem that are intractable in the worst case. So, we should not be surprised to find that we do not have efficient guaranteed solutions for CSP. At best, we can hope that our methods perform acceptably in the class of problems we are interested in. This will depend on the structure of the domain of applicability and will not follow directly from the algorithms.



Let's look at a trivial example of graph coloring. We have three variables with the domains indicated. Each variable is constrained to have values different from its neighbors.



We will now simulate the process of constraint propagation. In the interest of space, we will deal in this example with undirected arcs, which are just a shorthand for the two directed arcs between the variables. Each step in the simulation involves examining one of these undirected arcs, seeing if the arc is consistent and, if not, deleting values from the domain of the appropriate variable.

**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted

Each undirected constraint arc is really two directed constraint arcs, the effects shown above are from examining BOTH arcs.

We start with the V1-V2 arc. Note that for every value in the domain of V1 (R, G and B) there is some value in the domain of V2 that it is consistent with (that is, it is different from). So, for R in V1 there is a G in V2, for G in V1 there is an R in V2 and for B in V1 there is either R and G in V2. Similarly, for each entry in V2 there is a valid counterpart in V1. So, the arc is consistent and no changes are made.



**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none

We move to  $V_1-V_3$ . The situation here is different. While R and B in  $V_1$  can co-exist with the G in  $V_3$ , not so the G in  $V_1$ . And, so, we remove the G from  $V_1$ . Note that the arc in the other direction is consistent.

**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(G)$

Moving to  $V_2-V_3$ , we note similarly that the G in  $V_2$  has no valid counterpart in  $V_3$  and so we drop it from  $V_2$ 's domain. Although we have now looked at all the arcs once, we need to keep going since we have changed the domains for  $V_1$  and  $V_2$ .

**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(\mathbf{G})$
$V_2 - V_3$	$V_2(\mathbf{G})$

Looking at  $V_1 - V_2$  again we note that R in  $V_1$  no longer has a valid counterpart in  $V_2$  (since we have deleted G from  $V_2$ ) and so we need to drop R from  $V_1$ .

**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(\mathbf{G})$
$V_2 - V_3$	$V_2(\mathbf{G})$
$V_1 - V_2$	$V_1(\mathbf{R})$

We test  $V_1 - V_3$  and it is consistent.

**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(\mathbf{G})$
$V_2 - V_3$	$V_2(\mathbf{G})$
$V_1 - V_2$	$V_1(\mathbf{R})$
$V_1 - V_3$	none

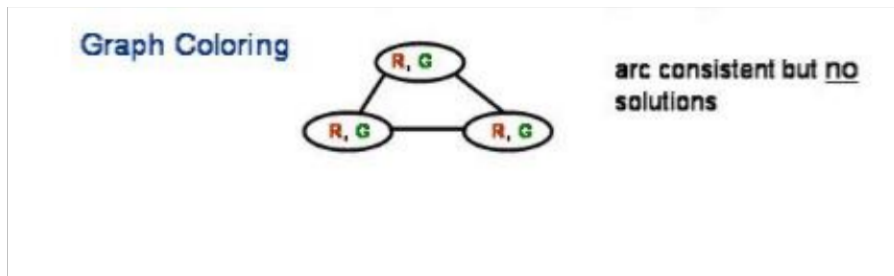
We test  $V_2 - V_3$  and it is consistent.

We are done; the graph is arc consistent. In general, we will need to make one pass through any arc whose head variable has changed until no further changes are observed before we can stop. If at any point some variable has an empty domain, the graph has no consistent solution.

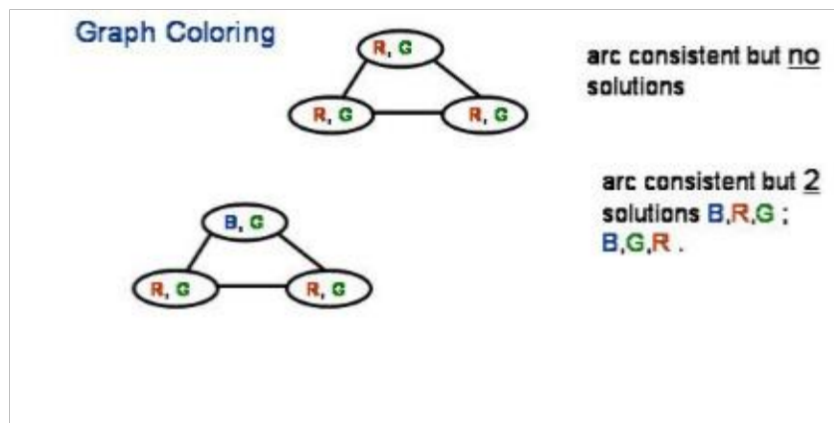
**Graph Coloring**  
Initial Domains are indicated

Arc examined	Value deleted
$V_1 - V_2$	none
$V_1 - V_3$	$V_1(\mathbf{G})$
$V_2 - V_3$	$V_2(\mathbf{G})$
$V_1 - V_2$	$V_1(\mathbf{R})$
$V_1 - V_3$	none
$V_2 - V_3$	none

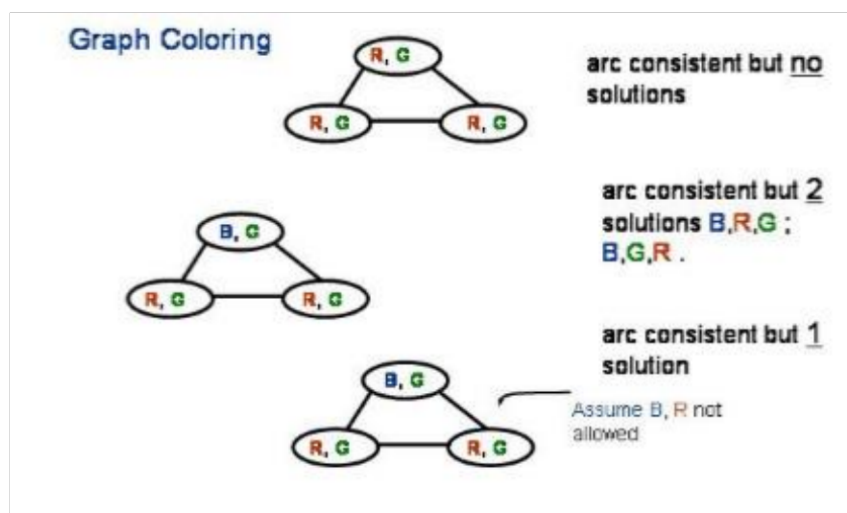
Note that whereas arc consistency is required for there to be a solution for a CSP, having an arc-consistent solution is not sufficient to guarantee a unique solution or even any solution at all. For example, this first graph is arc-consistent but there are NO solutions for it (we need at least three colors and have only two).



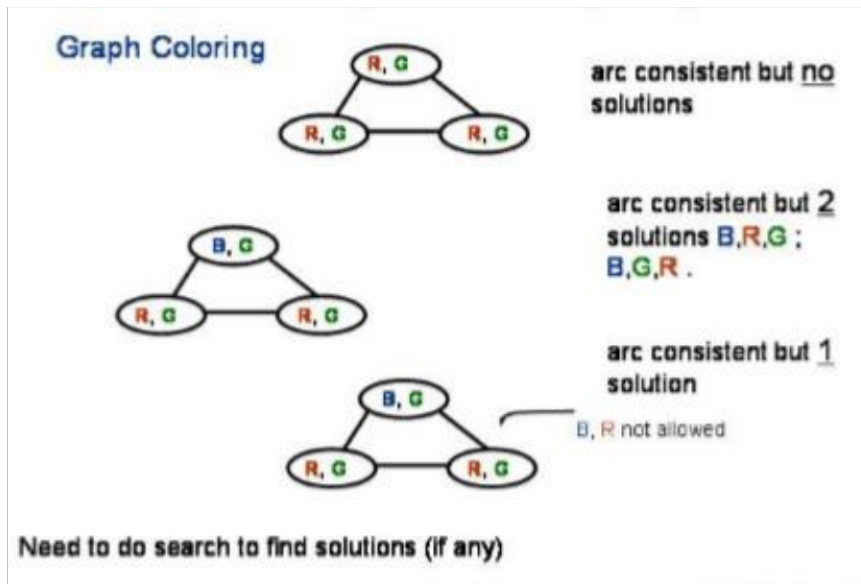
This next graph is also arc consistent but there are 2 distinct solutions: BRG and BGR.



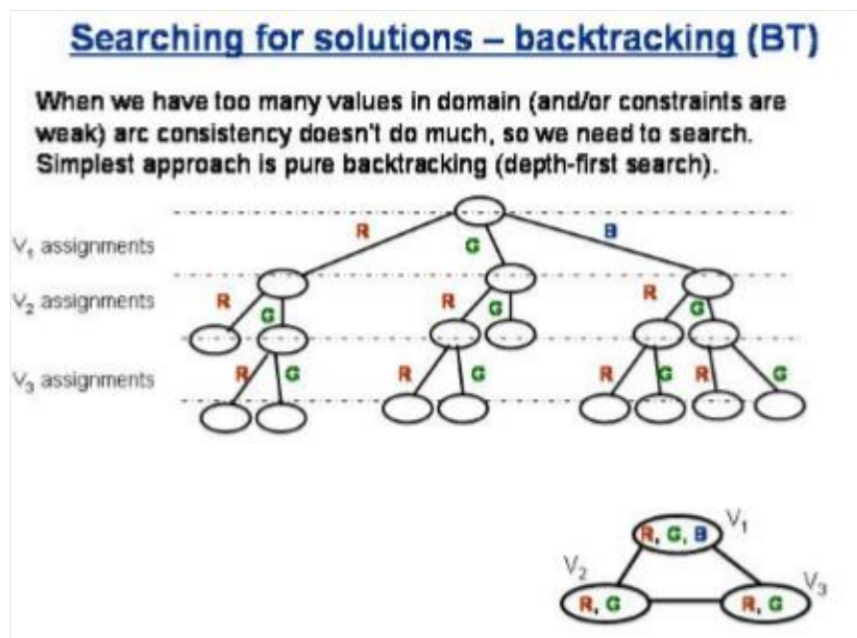
This next graph is also arc consistent but it has a unique solution, by virtue of the special constraint between two of the variables.



In general, if there is more than one value in the domain of any of the variables, we do not know whether there is zero, one, or more than one answer that is globally consistent. We have to search for an answer to actually know for sure.



How does one search for solutions to a CSP problem? Any of the search methods we have studied is applicable. All we need to realize is that the space of assignments of values to variables can be viewed as a tree in which all the assignments of values to the first variable are descendants of the first node and all the assignments of values to the second variable form the descendants of those nodes and so forth.

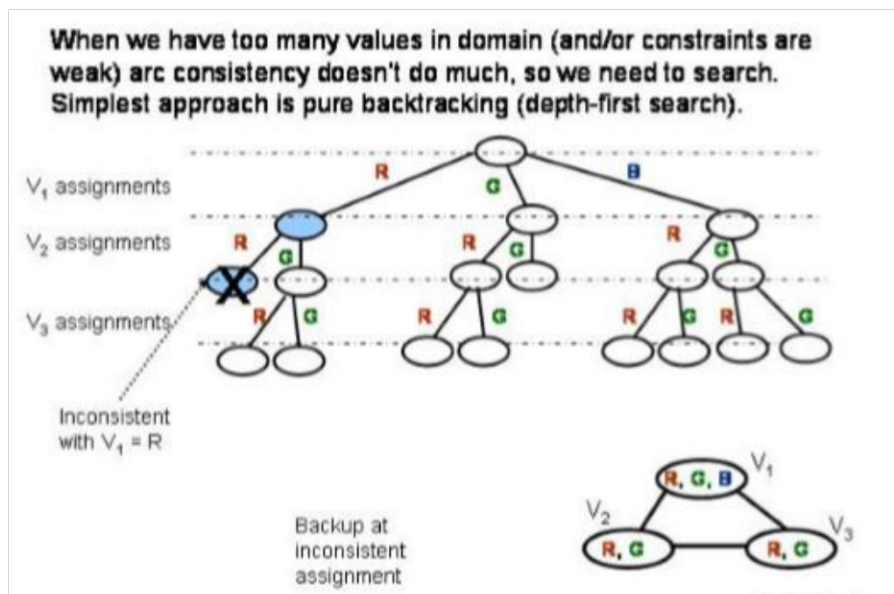


The classic approach to searching such a tree is called "backtracking", which is just another name for depth-first search in this tree. Note, however, that we could use breadth-first search or any of the heuristic searches on this problem. The heuristic value could be used to either guide the search to termination or bias it to a desired

solution based on preferences for certain assignments. UniformCost and A\* would make sense also if there were a non-uniform cost associated with a particular assignment of a value to a variable (note that this is another (better but more expensive) way of incorporating preferences).

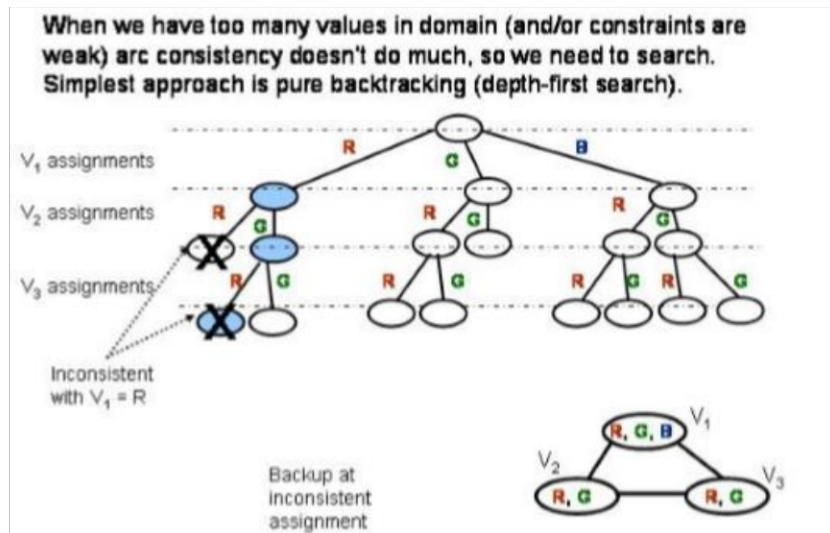
However, you should observe that these CSP problems are different from the graph search problems we looked at before, in that we don't really care about the path to some state but just the final state itself.

If we undertake a DFS in this tree, going left to right, we first explore assigning R to V1 and then move to V2 and consider assigning R to it. However, for any assignment, we need to check any constraints involving previous assignments in the tree. We note that  $V_2=R$  is inconsistent with  $V_1=R$  and so that assignment fails and we have to backup to find an alternative assignment for the most recently assigned variable.

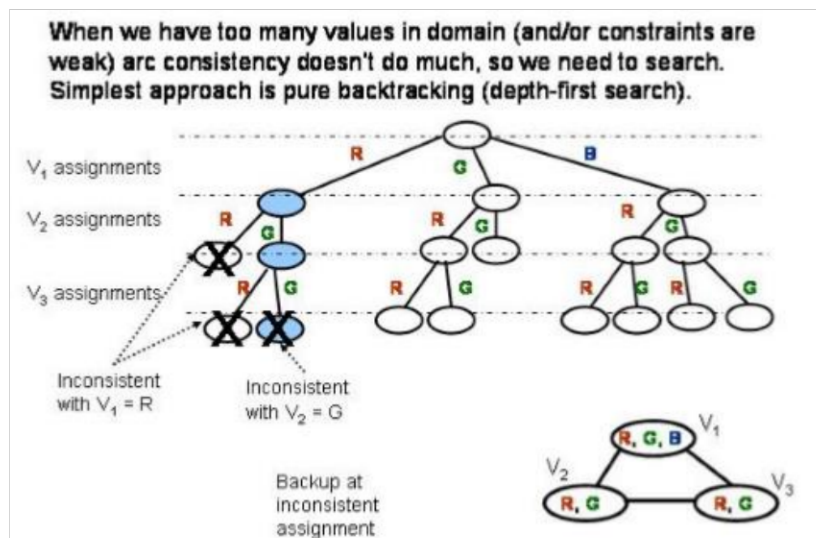


So, we consider assigning  $V_2=G$ , which is consistent with the value for  $V_1$ . We then move to  $V_3=R$ . Since we have a constraint between  $V_1$  and  $V_3$ , we have to check for consistency and find it is not consistent, and so we backup to consider another value for  $V_3$ .





But  $V_3=G$  is inconsistent with  $V_2=G$ , and so we have to backup. But there are no more pending values for  $V_3$  or for  $V_2$  and so we fail back to the  $V_1$  level.



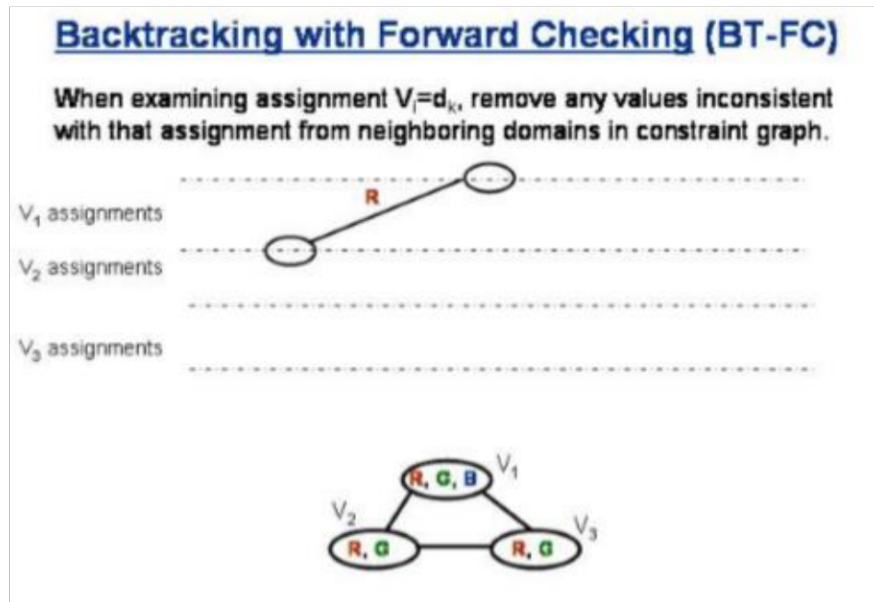
The process continues in that fashion until we find a solution. If we continue past the first success, we can find all the solutions for the problem (two in this case).



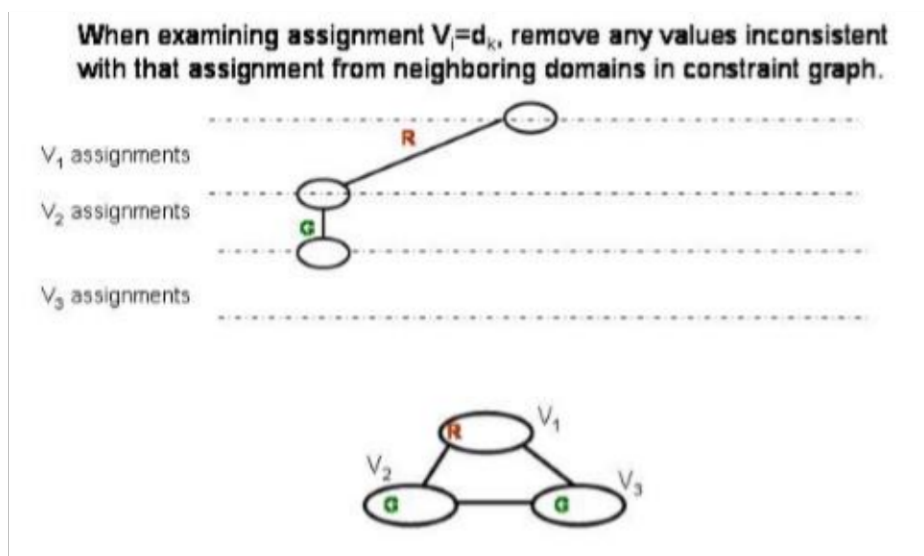


usually a waste of time. This conclusion suggests that forward checking is usually as much propagation as we want to do. This is, of course, only a rule of thumb.

Let's step through a search that uses a combination of backtracking with forward checking. We start by considering an assignment of  $V_1=R$ .

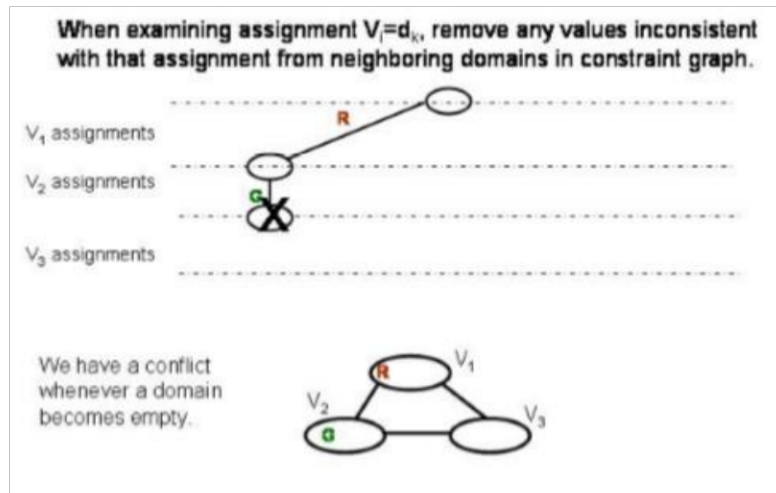


We then propagate to the neighbors of  $V_1$  in the constraint graph and eliminate any values that are inconsistent with that assignment, namely the value R. That leaves us with the value G in the domains of  $V_2$  and  $V_3$ . So, we make the assignment  $V_2=G$  and propagate.

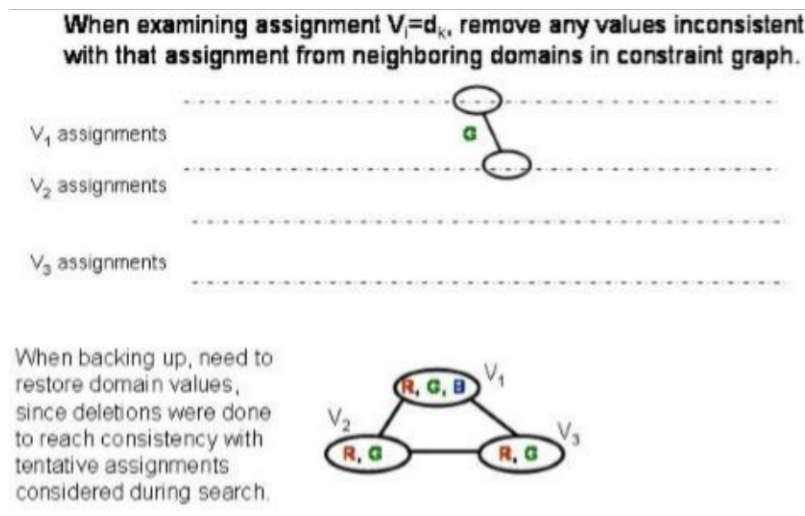


But, when we propagate to  $V_3$  we see that there are no remaining valid values

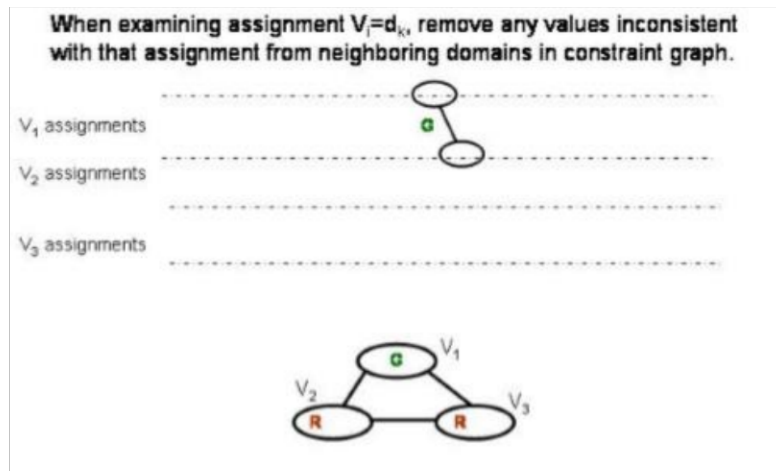
and so we have found an inconsistency. We fail and backup. Note that we have failed much earlier than with simple backtracking, thus saving a substantial amount of work.



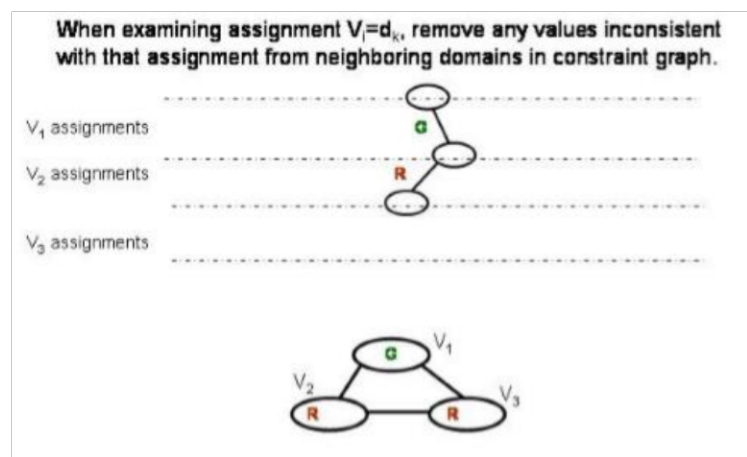
We now consider  $V_1=G$  and propagate.



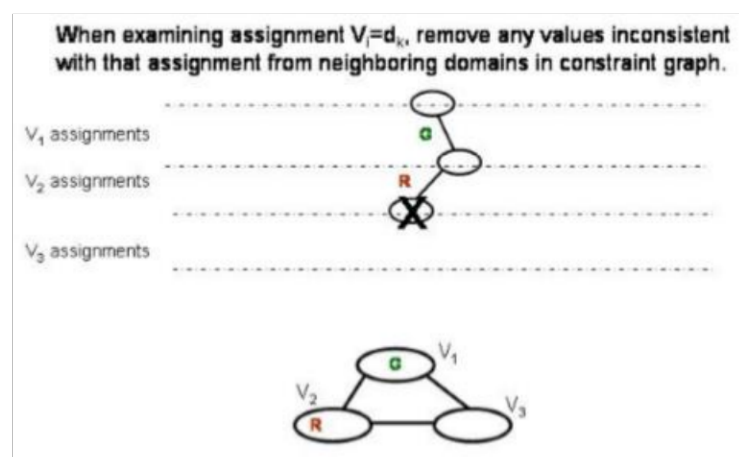
That eliminates G from  $V_2$  and  $V_3$ .



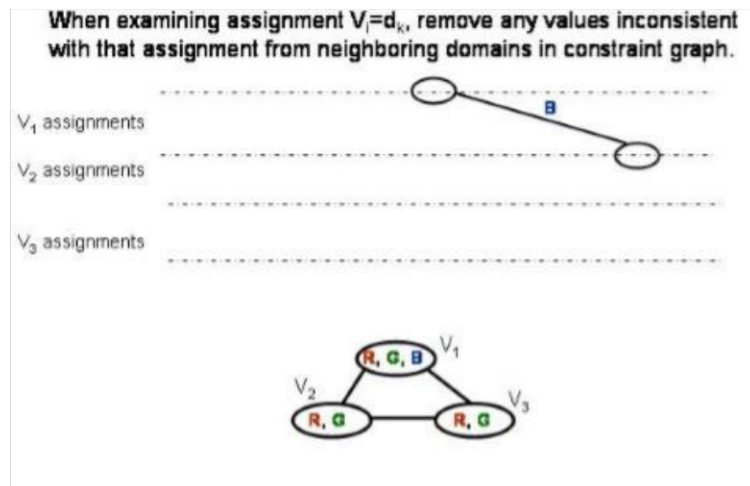
We now consider  $V_2=R$  and propagate.



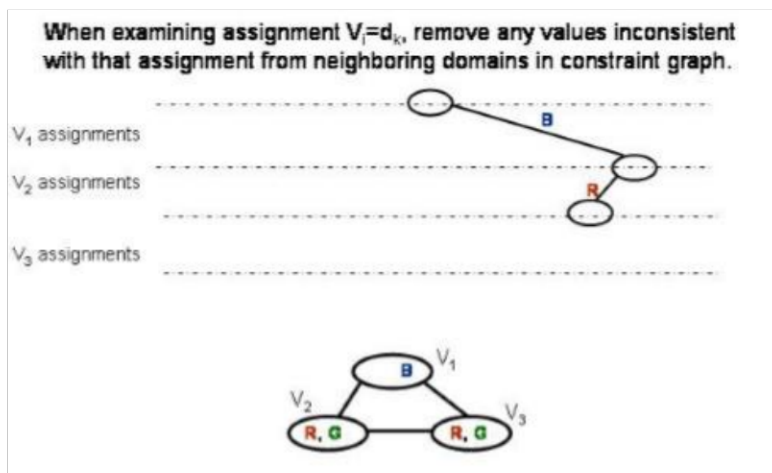
The domain of  $V_3$  is empty, so we fail and backup.



So, we move to consider  $V_1=B$  and propagate.

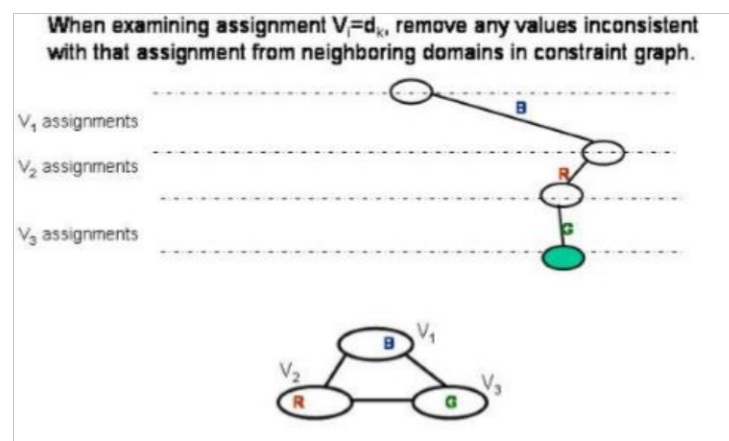


This propagation does not delete any values. We pick  $V_2=R$  and propagate.



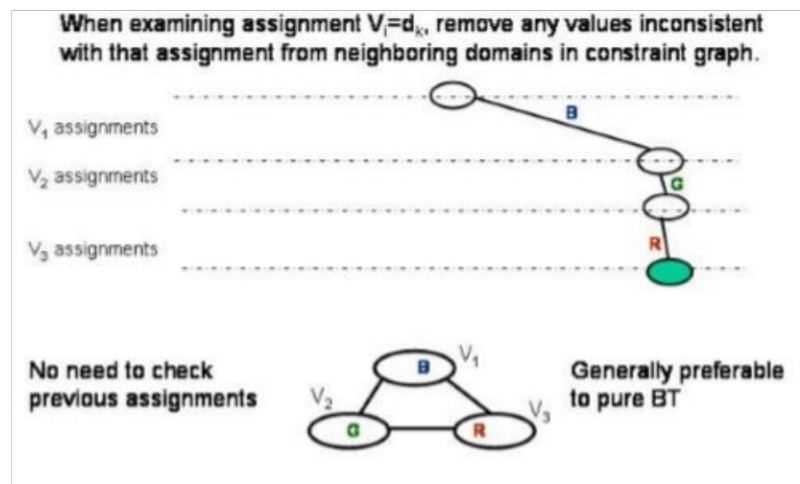
This removes the R values in the domains of V<sub>1</sub> and V<sub>3</sub>.

We pick  $V_3 = G$  and have a consistent assignment.



We can continue the process to find the other consistent solution.

Note that when doing forward checking there is no need to check new assignments against previous assignments. Any potential inconsistencies have been removed by the propagation. BT-FC is usually preferable to plain BT because it eliminates from consideration inconsistent assignments once and for all rather than discovering the inconsistency over and over again in different parts of the tree. For example, in pure BT, an assignment for  $V_3$  that is inconsistent with a value of  $V_1$  would be "discovered" independently for every value of  $V_2$ . Whereas FC would delete it from the domain of  $V_3$  right away.



We have been assuming that the order of the variables is given by some arbitrary ordering. However, the order of the variables (and values) can have a substantial effect on the cost of finding the answer. Consider, for example, the course scheduling problem using courses given in the order that they should ultimately be taken and assume that the term values are ordered as well. Then a depth first search will tend to find the answer very quickly.

Of course, we generally don't know the answer to start off with, but there are more rational ways of ordering the variables than alphabetical or numerical order. For example, we could order the variables before starting by how many constraints they have. But we can do even better by dynamically re-ordering variables based on information available during a search.



## CHAPTER 2

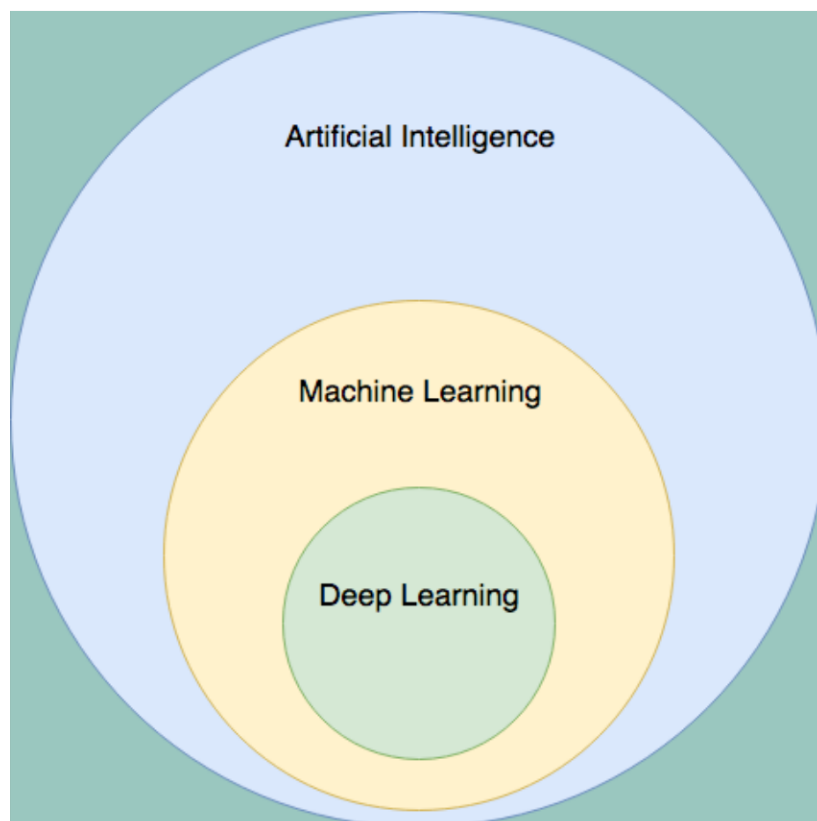
### Introduction to Machine Learning

#### 2.1 Machine Learning

Machine learning is a field of artificial intelligence. It uses statistical methods to give computer the ability to "learn" from data, without being explicitly programmed. If a computer program can improve how, it performs certain tasks based on past experiences, then it has learned. This differs from performing the task always the same way because it has been programmed to do so.

The learning process improves the so-called "model" over time by using different data points (training data). The involved model is used to make predictions.

Deep learning is a subset of machine learning. Machine learning is a subset of artificial intelligence. Said another way — all deep learning algorithms are machine learning algorithms, but many machine learning algorithms do not use deep learning. As a Venn diagram, it looks like this:



Deep learning refers specifically to a class of algorithm called a neural network,



and technically only to “deep” neural networks (more on that in a second). This first neural network was invented in 1949, but back then they weren’t very useful. In fact, from the 1970’s to the 2010’s traditional forms of AI would consistently outperform neural network-based models.

These non-learning types of AI include rule based algorithms (imagine an extremely complex series of if/else blocks); heuristic based AIs such as A\* search; constraint satisfaction algorithms like Arc Consistency; tree search algorithms such as minimax (used by the famous Deep Blue chess AI); and more.

There were two things preventing machine learning, and especially deep learning, from being successful. Lack of availability of large datasets and lack of availability of computational power. In 2018 we have exabytes of data, and anyone with an AWS account and a credit card has access to a distributed supercomputer. Because of the new availability of data and computing power, Machine learning — and especially deep learning — has taken the AI world by storm.

## **2.2 Machine Learning Algorithms**

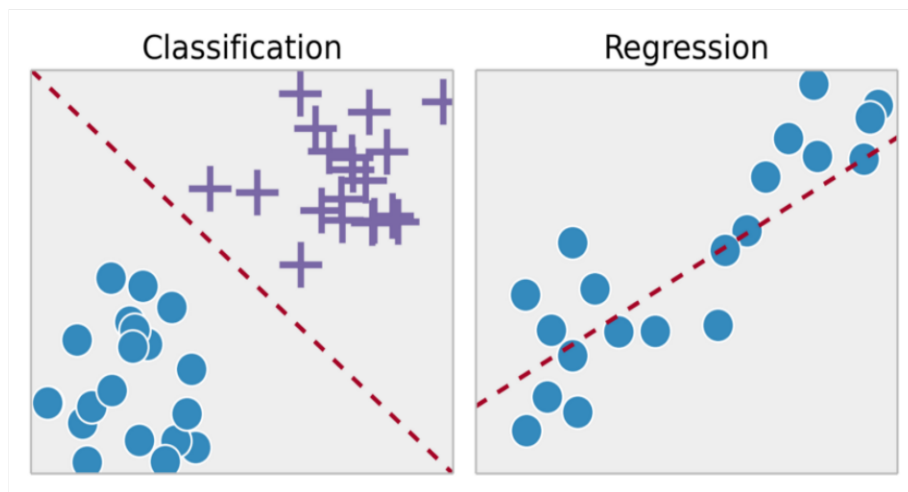
To estimate the function that represents the model, an appropriate learning algorithm must be used. In this context, the learning algorithm represents the technique through which the system extracts useful patterns from the input historical data. These patterns can be applied to new data in new situations. The objective is to have the system learn a specific input/output transformation and to make future predictions for a new data point. Finding the appropriate algorithms to solve complex problems in various domains and knowing how and when to apply them is an important skill that machine learning engineers should acquire. Because the machine learning algorithms depend on data, understanding and acquiring data with high quality is crucial for accurate results.

## **2.3 Machine Learning Approach (Supervised vs. Unsupervised Learning)**

Within the field of machine learning, there are two main types of tasks: supervised, and unsupervised. The main difference between the two types is that supervised learning is done using a **ground truth**, or in other words, we have prior

knowledge of what the output values for our samples should be. Therefore, the goal of supervised learning is to learn a function that, given a sample of data and desired outputs, best approximates the relationship between input and output observable in the data. Unsupervised learning, on the other hand, does not have labeled outputs, so its goal is to infer the natural structure present within a set of data points.

## Supervised Learning



Supervised learning is typically done in the context of classification, when we want to map input to output labels, or regression, when we want to map input to a continuous output. Common algorithms in supervised learning include logistic regression, naive bayes, support vector machines, artificial neural networks, and random forests. In both regression and classification, the goal is to find specific relationships or structure in the input data that allow us to effectively produce correct output data. Note that “correct” output is determined entirely from the training data, so while we do have a ground truth that our model will assume is true, it is not to say that data labels are always correct in real-world situations. Noisy, or incorrect, data labels will clearly reduce the effectiveness of your model.

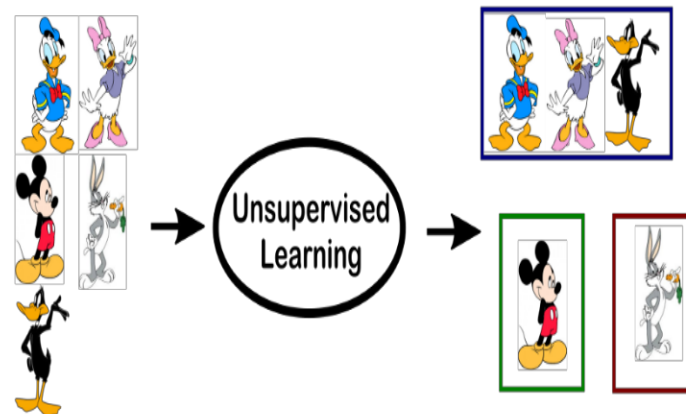
When conducting supervised learning, the main considerations are model complexity, and the bias-variance tradeoff. Note that both of these are interrelated.

Model complexity refers to the complexity of the function you are attempting to learn — similar to the degree of a polynomial. The proper level of model complexity is generally determined by the nature of your training data. If you have a small

amount of data, or if your data is not uniformly spread throughout different possible scenarios, you should opt for a low-complexity model. This is because a high-complexity model will **overfit** if used on a small number of data points. Overfitting refers to learning a function that fits your training data very well, but does not **generalize** to other data points — in other words, you are strictly learning to produce your training data without learning the actual trend or structure in the data that leads to this output. Imagine trying to fit a curve between 2 points. In theory, you can use a function of any degree, but in practice, you would parsimoniously add complexity, and go with a linear function.

The bias-variance tradeoff also relates to model generalization. In any model, there is a balance between bias, which is the constant error term, and variance, which is the amount by which the error may vary between different training sets. So, high bias and low variance would be a model that is consistently wrong 20% of the time, whereas a low bias and high variance model would be a model that can be wrong anywhere from 5%-50% of the time, depending on the data used to train it. Note that bias and variance typically move in opposite directions of each other; increasing bias will usually lead to lower variance, and vice versa. When making your model, your specific problem and the nature of your data should allow you to make an informed decision on where to fall on the bias-variance spectrum. Generally, increasing bias (and decreasing variance) results in models with relatively guaranteed baseline levels of performance, which may be critical in certain tasks. Additionally, in order to produce models that generalize well, the variance of your model should scale with the size and complexity of your training data — small, simple data-sets should usually be learned with low-variance models, and large, complex data-sets will often require higher-variance models to fully learn the structure of the data.

## Unsupervised Learning



The most common tasks within unsupervised learning are clustering, representation learning, and density estimation. In all of these cases, we wish to learn the inherent structure of our data without using explicitly-provided labels. Some common algorithms include k-means clustering, principal component analysis, and autoencoders. Since no labels are provided, there is no specific way to compare model performance in most unsupervised learning methods.

Two common use-cases for unsupervised learning are exploratory analysis and dimensionality reduction.

Unsupervised learning is very useful in exploratory analysis because it can automatically identify structure in data. For example, if an analyst were trying to segment consumers, unsupervised clustering methods would be a great starting point for their analysis. In situations where it is either impossible or impractical for a human to propose trends in the data, unsupervised learning can provide initial insights that can then be used to test individual hypotheses.

Dimensionality reduction, which refers to the methods used to represent data using less columns or features, can be accomplished through unsupervised methods. In representation learning, we wish to learn relationships between individual features, allowing us to represent our data using the latent features that interrelate our initial features. This sparse latent structure is often represented using far fewer features than we started with, so it can make further data processing much less intensive, and can eliminate redundant features.

	<i>Supervised Learning</i>	<i>Unsupervised Learning</i>
<i>Discrete</i>	classification or categorization	clustering
<i>Continuous</i>	regression	dimensionality reduction

## 2.4 Machine learning algorithms

In the following slides, we explore different machine learning algorithms. We describe the most prominent algorithms. Each algorithm belongs to a category of learning. We explore **supervised and unsupervised algorithms**, regression and classification algorithms, and linear and non-linear classification

Understanding your problem and the different types of ML algorithms helps in selecting the best algorithm.

Here are some machine learning algorithms:

- Naïve Bayes classification (supervised classification - probabilistic)
- Linear regression (supervised regression)
- Logistic regression (supervised classification)
- Support vector machine (SVM) (supervised linear or non-linear classification)
- Decision tree (supervised non-linear classification)
- K-means clustering (unsupervised learning)

## 2.4.1 Naive Bayes Classifier

### What is a classifier?

A classifier is a machine learning model that is used to discriminate different objects based on certain features.

### Principle of Naive Bayes Classifier:

A Naive Bayes classifier is a probabilistic machine learning model that's used for classification task. The crux of the classifier is based on the Bayes theorem.

### Bayes Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Using Bayes theorem, we can find the probability of **A** happening, given that **B** has occurred. Here, **B** is the evidence and **A** is the hypothesis. The assumption made here is that the predictors/features are independent. That is presence of one particular feature does not affect the other. Hence it is called naive.

### Example:

Let us take an example to get some better intuition. Consider the problem of playing golf. The dataset is represented as below.



	OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY GOLF
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny			True	No

We classify whether the day is suitable for playing golf, given the features of the day. The columns represent these features and the rows represent individual entries. If we take the first row of the dataset, we can observe that it is not suitable for playing golf if the outlook is rainy, temperature is hot, humidity is high and it is not windy. We make two assumptions here, one as stated above we consider that these predictors are independent. That is, if the temperature is hot, it does not necessarily mean that the humidity is high. Another assumption made here is that all the predictors have an equal effect on the outcome. That is, the day being windy does not have more importance in deciding to play golf or not.

According to this example, Bayes theorem can be rewritten as:

The variable  $y$  is the class variable (play golf), which represents if it is suitable to play golf or not given the conditions. Variable  $X$  represent the parameters/features.

$X$  is given as,



$$X = (x_1, x_2, x_3, \dots, x_n)$$

Here  $x_1, x_2, \dots, x_n$  represent the features, i.e they can be mapped to outlook, temperature, humidity and windy. By substituting for  $X$  and expanding using the chain rule we get,

$$P(y|x_1, \dots, x_n) = \frac{P(x_1|y)P(x_2|y)\dots P(x_n|y)P(y)}{P(x_1)P(x_2)\dots P(x_n)}$$

Now, you can obtain the values for each by looking at the dataset and substitute them into the equation. For all entries in the dataset, the denominator does not change, it remain static. Therefore, the denominator can be removed and a proportionality can be introduced.

$$P(y|x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i|y)$$

In our case, the class variable( $y$ ) has only two outcomes, yes or no. There could be cases where the classification could be multivariate. Therefore, we need to find the class  $y$  with maximum probability.

$$y = \operatorname{argmax}_y P(y) \prod_{i=1}^n P(x_i|y)$$

Using the above function, we can obtain the class, given the predictors.

### **Types of Naive Bayes Classifier:**

#### **Multinomial Naive Bayes:**

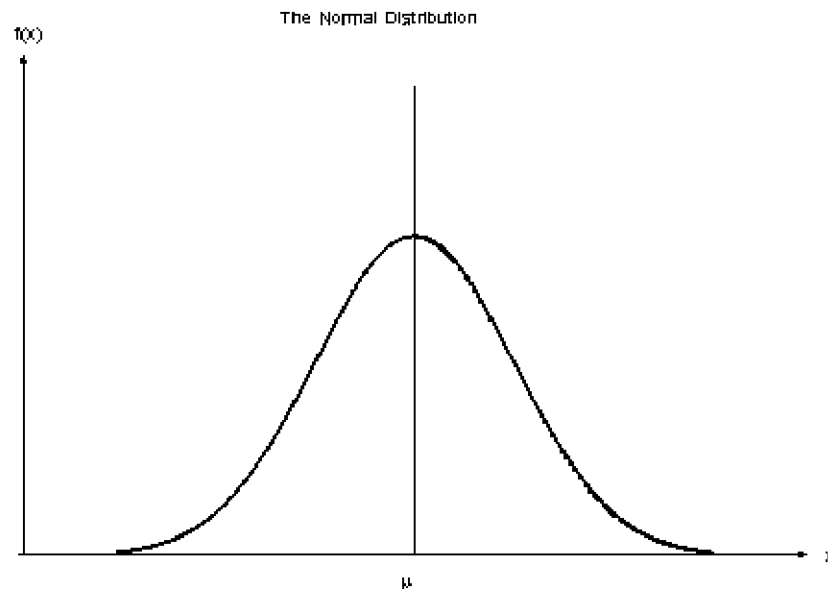
This is mostly used for document classification problem, i.e whether a document belongs to the category of sports, politics, technology etc. The features/predictors used by the classifier are the frequency of the words present in the document.

#### **Bernoulli Naive Bayes:**

This is similar to the multinomial naive bayes but the predictors are boolean variables. The parameters that we use to predict the class variable take up only values yes or no, for example if a word occurs in the text or not.

## Gaussian Naive Bayes:

When the predictors take up a continuous value and are not discrete, we assume that these values are sampled from a gaussian distribution.



Gaussian Distribution (Normal Distribution)

Since the way the values are present in the dataset changes, the formula for conditional probability changes to,

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

## 2.4.2 Linear Regression

Regression algorithms are one of the key algorithms that are used in machine learning. Regression algorithms help analysts to model relationships between input variables  $X$  and the output label  $Y$  for the training data points. This algorithm targets supervised regression problems, that is, the target variable is a continuous value.

In simple linear regression we establish a relationship between the target variable

and input variables by fitting a line.



The equation which defines the simplest form of the regression equation with one dependent and one independent variable:  $y = mx+c$ .

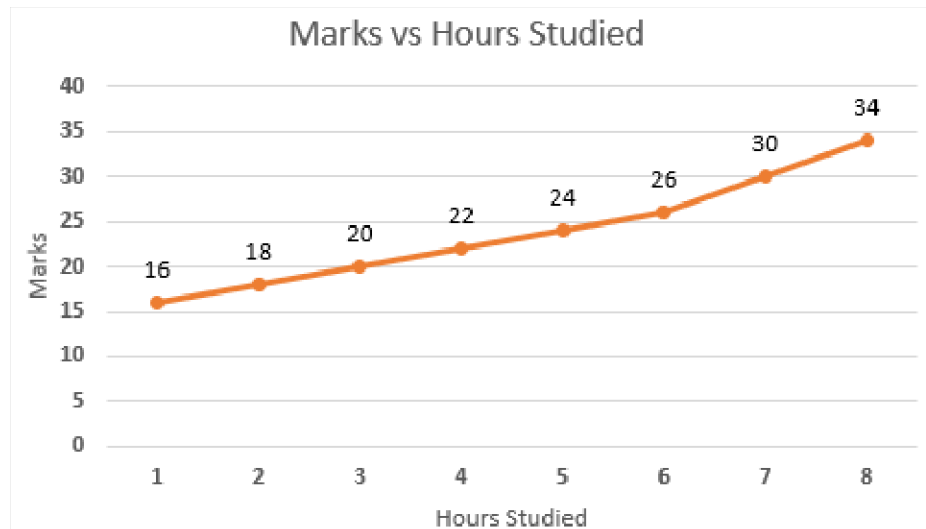
Where  $y$  = estimated dependent variable,  $c$  = constant,  $m$ = regression coefficient and  $x$  = independent variable.

Let's just understand with an example:

Hours Studied	Marks Scored
1	16
2	18
3	20
4	22
5	24
6	26
7	30
8	34

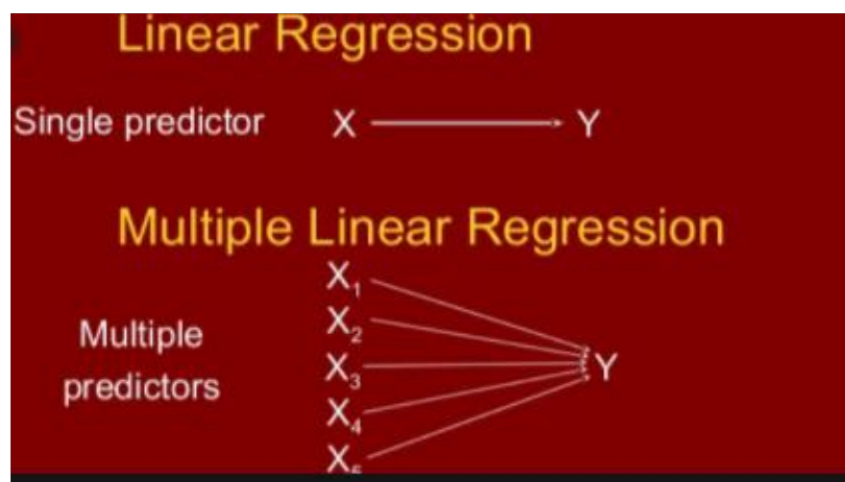
Hours vs Marks Scored by the Students

Say; There is a certain relationship between the marks scored by the students ( $y$ - Dependent variable) in an exam and hours they studied for the exam( $x$ - Independent Variable). Now we want to predict the marks scored by the students by input **the hours** studied. Here,  $x$  would be the input and  $y$  would be the output. Linear regression model would predict the marks of the students when the user will input the number of hours studied.



That is just a simple example of Linear regression, Linear Regression can be divided into two types:

1. **Simple Linear Regression:** If a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.
2. **Multiple Linear Regression:** If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

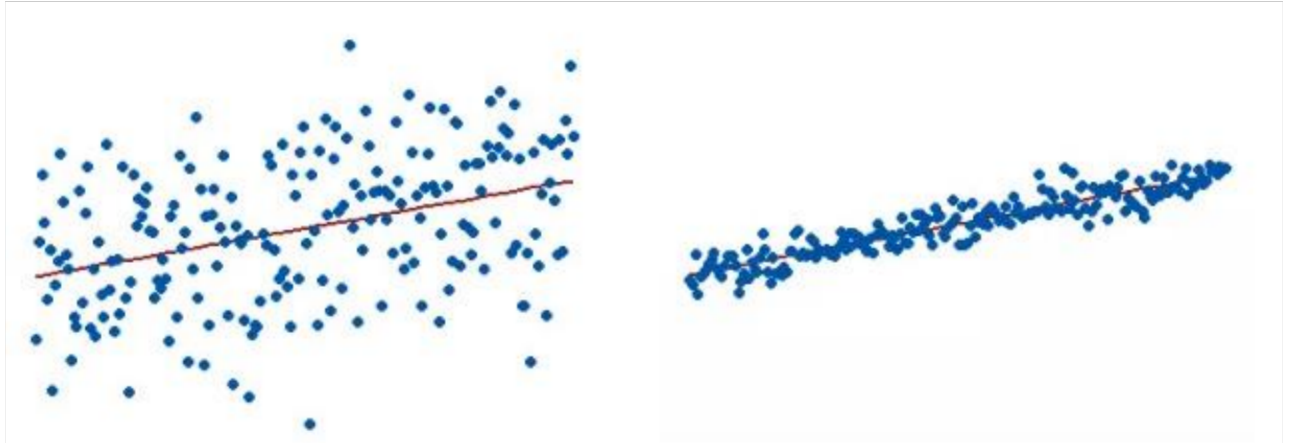


### Assessing Goodness-of-Fit in a Regression Model

**R-Squared:** It evaluates the scatter of the data points around the fitted regression line. We also call it the coefficient of determination, or the coefficient of multiple

determination for multiple regression. It measures the strength of the relationship between the dependent and independent variables on a scale of 0–100%.

To visually show how R-Squared values represent the scatter around the regression line, you can plot the fitted values by observed values.



The R-squared for the regression model on the left is 15%, and for the model on the right it is 85%. When a regression model accounts for more of the variance, the data points are closer to the regression line. You'll never see a regression model with an R<sup>2</sup> of 100%. In that case, the fitted values equal the data values and, all the observations fall exactly on the regression line.

### **R-Squared has Limitations**

You cannot use R-Squared to determine whether it biases the coefficient estimates and predictions, which is why you must assess the residual plots.

R-Squared does not show if a regression model provides an adequate fit to your data. A suitable model can have a low R<sup>2</sup> value. On the other hand, a biased model can have a high R<sup>2</sup> value.

### **Assumption of Linear Regression**

1. Linear relationship between the features and target.
2. Small or no multicollinearity between the features: Multicollinearity means high-correlation between the independent variables. Because of multicollinearity, it may difficult to find the true relationship between the

predictors and target variables.

3. Homoscedasticity: variance of the residual, or error term, in a **regression** model is constant.
4. No autocorrelations: Autocorrelation usually occurs if there is a dependency between residual errors.

### 2.4.3 Logistic Regression

It's a classification algorithm that is used where the response variable is *categorical*. The idea of Logistic Regression is to find a relationship between features and probability of particular outcome.

*E.g.* When we have to predict if a student passes or fails in an exam when the number of hours spent studying is given as a feature, the response variable has two values, pass and fail.

This type of a problem is referred to as Binomial Logistic Regression, where the response variable has two values 0 and 1 or pass and fail or true and false. Multinomial Logistic Regression deals with situations where the response variable can have three or more possible values.

#### Why Logistic, not Linear?

With binary classification, let 'x' be some feature and 'y' be the output which can be either 0 or 1. The probability that the output is 1 given its input can be represented as:

$$P(y = 1 | x)$$

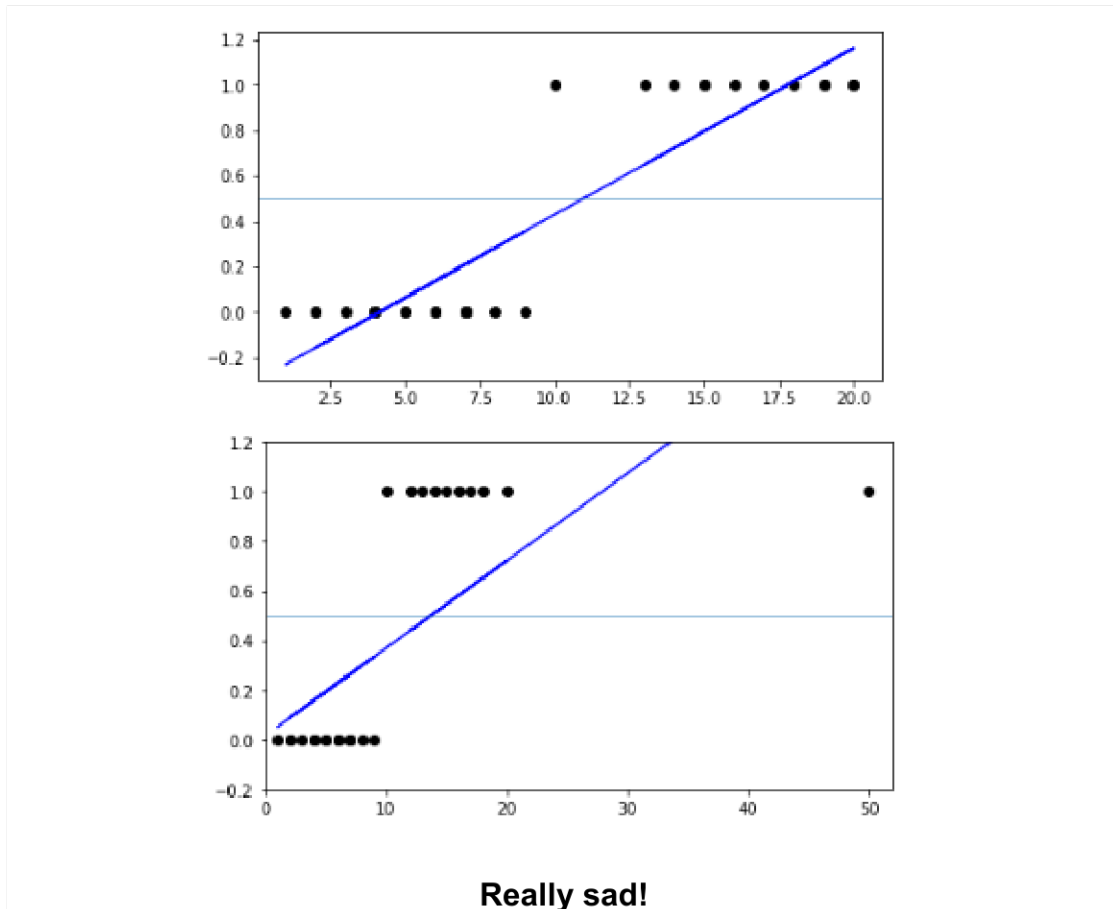
If we predict the probability via linear regression, we can state it as:

$$p(X) = \beta_0 + \beta_1 X.$$

**where,  $p(x) = p(y=1|x)$**

Linear regression model can generate the *predicted probability* as any number

ranging from negative to positive infinity, whereas probability of an outcome can only lie between  $0 < P(x) < 1$ .



Also, Linear regression has a considerable effect on outliers. To avoid this problem, *log-odds* function or *logit* function is used.

## Logit Function

Logistic regression can be expressed as:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X.$$

where, the left hand side is called the **logit** or log-odds function, and  $\frac{p(x)}{1-p(x)}$  is called odds.

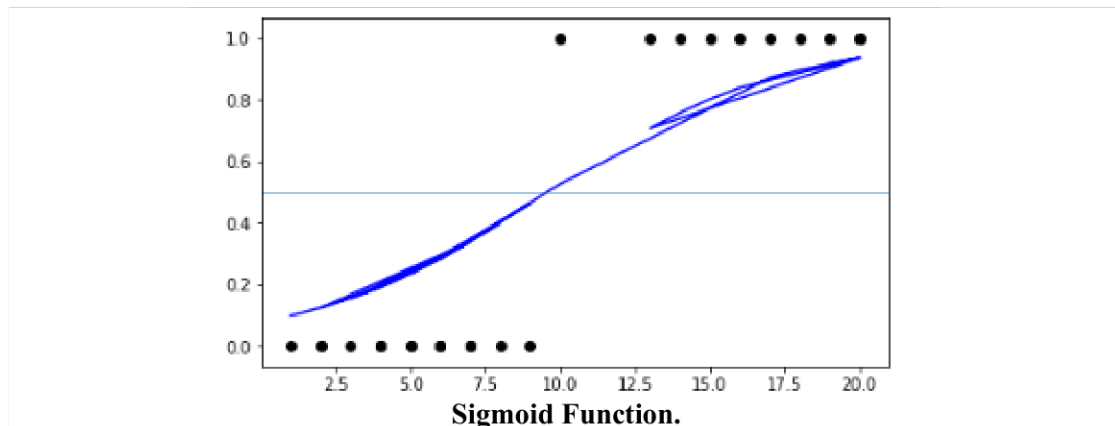
The *odds* signifies the ratio of probability of success to probability of failure. Therefore, in Logistic Regression, linear combination of inputs are mapped to the log (odds) - the output being equal to 1.



If we take an **inverse of the above function**, we get:

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

This is known as the *Sigmoid function* and it gives an S-shaped curve. It always gives a value of probability ranging from  $0 < p < 1$ .



### Estimation of Regression Coefficients

Unlike linear regression model, that uses Ordinary Least Square for parameter estimation, we use **Maximum Likelihood Estimation**.

There can be infinite sets of regression coefficients. The maximum likelihood estimate is that set of regression coefficients for which the probability of getting the data we have observed is maximum.

If we have binary data, the probability of each outcome is simply  $\pi$  if it was a success, and  $1 - \pi$  otherwise. Therefore we have the likelihood function:

$$\mathcal{L}(\beta; \mathbf{y}) = \prod_{i=1}^N \left( \frac{\pi_i}{1 - \pi_i} \right)^{y_i} (1 - \pi_i)$$

To determine the value of parameters, log of likelihood function is taken, since it does not change the properties of the function.

The log-likelihood is *differentiated* and using **iterative** techniques like Newton method, values of parameters that maximise the log-likelihood are determined.

### Some Familiar Example of Logistics Regression:

Some prominent examples like:

- Email Spam Filter: Spam /No Spam
- Fraud Detection : Transaction is fraudulent, Yes/No
- Tumor: Benign/Malignant

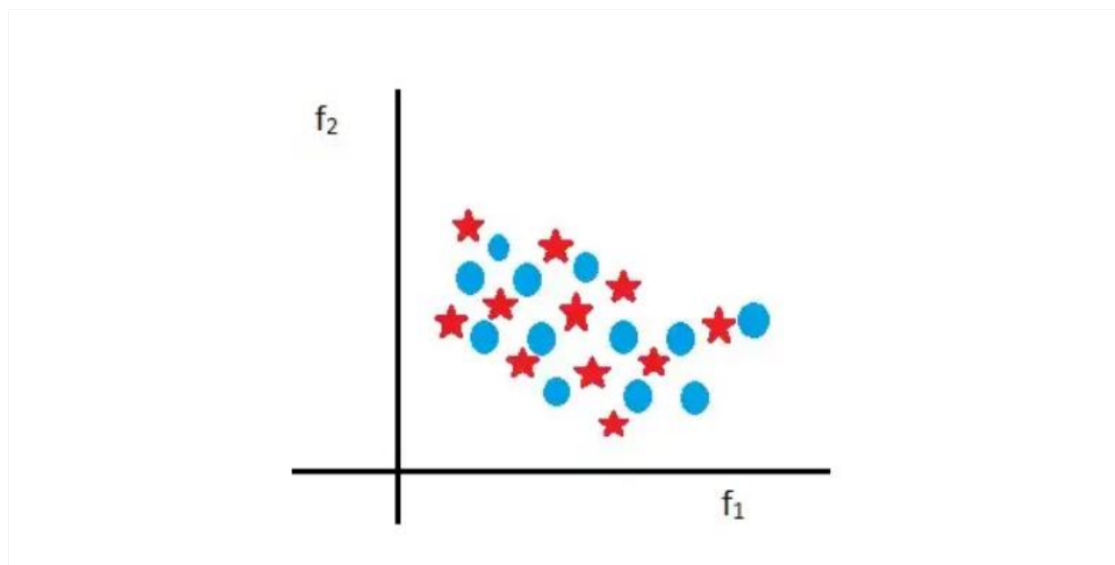
## 2.4.4 k-Nearest Neighbors

Let's start with the core idea of K-Nearest Neighbors (abbreviated as kNN).

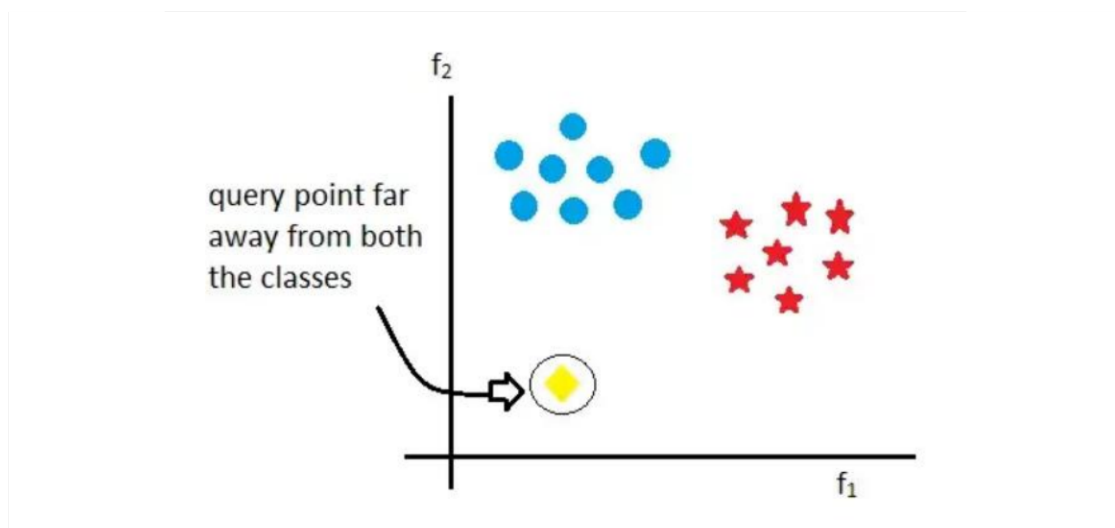
Given a query point  $x_i$ , we'll find k-nearest neighbors of that point in the given data set and predict a class  $y_i$  which is done by majority voting of classes of k-nearest neighbors. Same thing extends for Regression, but instead of majority voting, we take either mean or median of k-nearest neighbors.

### Failure cases of kNN:

Case 1: When positive and negative classes are randomly distributed



Case 2: When query point  $x_i$  is far away from either of the classes. In this case, we can't be sure which class does  $x_i$  belong to.



**How to decide whether a point is far away from either of the classes? What should be the threshold?**

Well, deciding the threshold is data specific but an acceptable way of deciding would be -

- Find 1st nearest neighbor distance for each point in our training data.
- Sort all the distances
- Find, let's say 90<sup>th</sup> or 95<sup>th</sup> percentile value ( $p'$ ) (or any other percentile value as per requirement).
- Now, if 1st nearest neighbor of query point  $x_i$  is greater than  $p'$ , then we can say that  $x_i$  is far from each of the classes.

**How do we find out the neighbors of a given point?**

It's same as finding the neighbors in real life. People who live close to your house are your neighbors. Similarly, data points which are close to the given point will be its neighbors. This closeness is defined by distance between the data points. There are lot of distance measures used in Machine Learning. Generally, Euclidean Distance is used in kNN.

### Time and Space Complexity for simple implementation of kNN

- Given a training dataset  $X$  with  $n$  datapoints and  $d$  dimensions.

For a query point  $x_t$ , we have to find distance with each of the points in training data.

For 1 distance calculation, we have to perform  $d$  operation ( $d =$  dimensions of the data)

$\therefore$  for  $n$  distance calculations , we will have to perform  $n*d$  operations or  $O(n*d)$  time

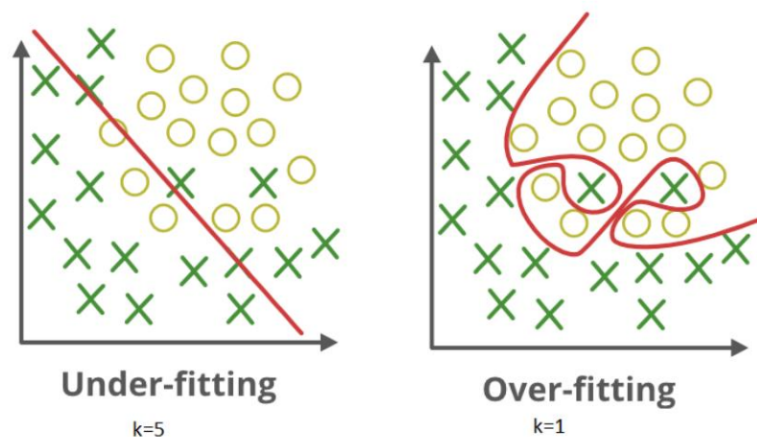
Then we have to pick nearest  $k$  points based on all the distances. Generally,  $k$  is very small as compared to size of dataset. Mostly,  $k=3, 5, 11, 15$  or so, and hence it would take  $\approx O(n)$  time

Then we have to decide using majority voting, which would take constant or  $O(1)$  time.

$\therefore$  Time Complexity of KNN will be:  $O(n*d) + O(n) + O(1) = O(n*d)$

When we say we have trained a kNN model , it simply means we have loaded the training data into the RAM. No operation is performed during the training phase and hence Space Complexity of kNN will be:  $O(n*d)$

### Decision Surface for kNN as $k$ changes



**Ask increases, smoothness of curve increases.**

**What if  $k=n$  (number of datapoints)?**

In that case, there won't be any decision surface and whole of the dataset would be taken into consideration. Given a query point, its class is decided by majority class in the training data. That would be a highly underfitting situation.

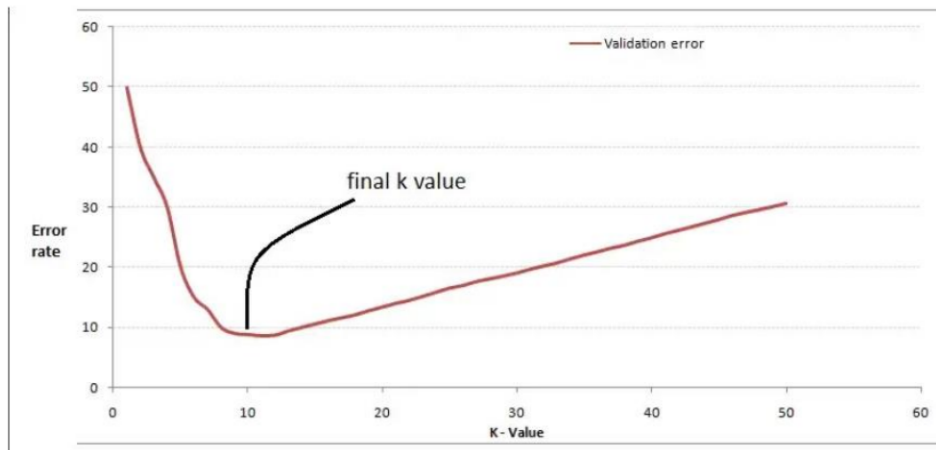
**Overfitting** is basically when we have found a function so well that it even takes every minute details into consideration including noisy outliers. The model will give approximately no error on training data, but will fail when it comes to unseen data. **It happens when  $k$  in kNN is small.**

**Underfitting** is basically when we have found a function which loosely fit to the data or in other words, is an imperfect way to describe the data. It will give error both on training and unseen data. **It happens when  $k$  in kNN is large.**

Finding the right value of  $k$

So we have seen how the value of  $k$  affects our model. But how should we determine the value of  $k$ ?

In practice, it is done using Cross-Validation, which could be a topic for my next article, but in a nutshell, we split our dataset into three parts: Train Data, CV (Cross-Validation) Data and Test Data. We train our model over the Train Data and for different values of  $k$ , accuracy is calculated over the CV Data. The  $k$  which gives maximum accuracy (or minimum error) is selected as final  $k$  value. After that, we test our model taking the final  $k$  value over the Test Data.



Source: Analytics Vidhya

## Weighted kNN

Sometimes we want that more weightage should be given to nearer points than to further away points from the query point. Hence we give weights to each of the k-nearest neighbors of the query point.

One way to assign weights is using inverse distance function. Let's understand this using an example:

	class	distance from $x_q$	weight = $1/d$
$x_1$	-ve	0.1	10
$x_2$	-ve	0.2	5
$x_3$	+ve	1.0	1
$x_4$	+ve	2.0	0.5
$x_5$	+ve	4.0	0.25

Now we calculate weight of each class.

$$\text{weight}(-\text{ve class}) = 10 + 5 = 15$$

$$\text{weight}(+\text{ve class}) = 1 + 0.5 + 0.25 = 1.75$$

Since, weight (-ve class) > weight (+ve class),  $\therefore$  We assign query point to negative class.

Well, that's all for now.

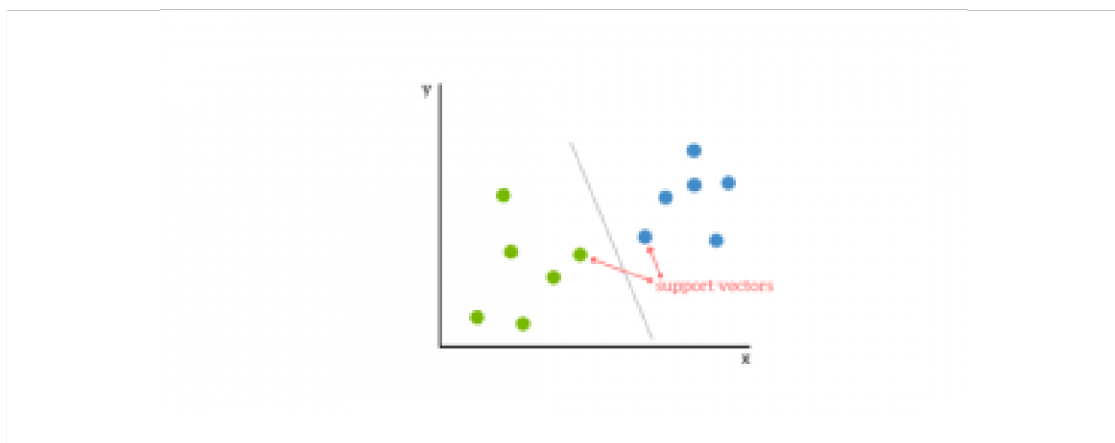
## 2.4.5 Support Vector Machines

### Introduction

In this post, we are going to introduce you to the Support Vector Machine (SVM) machine learning algorithm. We will follow a similar process to our recent post Naive Bayes for Dummies; A Simple Explanation by keeping it short and not overly-technical. The aim is to give those of you who are new to machine learning a basic understanding of the key concepts of this algorithm.

### Support Vector Machines - What are they?

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems and as such, this is what we will focus on in this post. SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown in the image below.



### Support Vectors

Support vectors are the data points nearest to the hyperplane, the points of a data



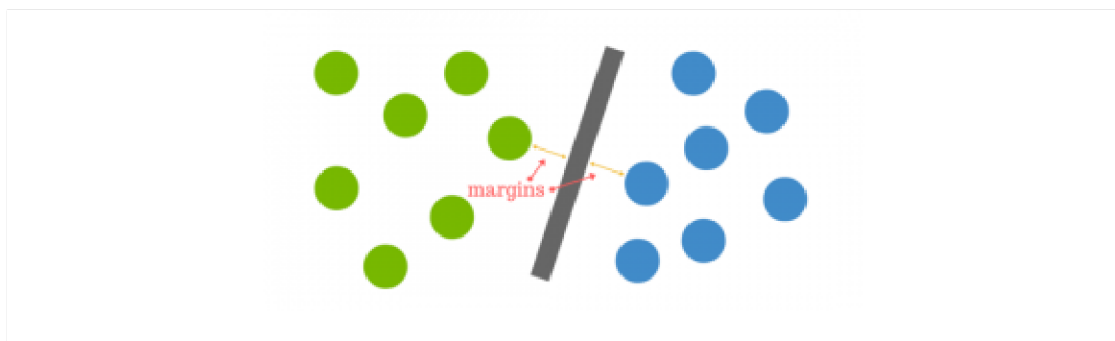
set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.

### **What is a hyperplane?**

As a simple example, for a classification task with only two features (like the image above), you can think of a hyperplane as a line that linearly separates and classifies a set of data. Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it. So when new testing data is added, whatever side of the hyperplane it lands will decide the class that we assign to it.

### **How do we find the right hyperplane?**

Or, in other words, how do we best segregate the two classes within the data? The distance between the hyperplane and the nearest data point from either set is known as the margin. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set, giving a greater chance of new data being classified correctly.

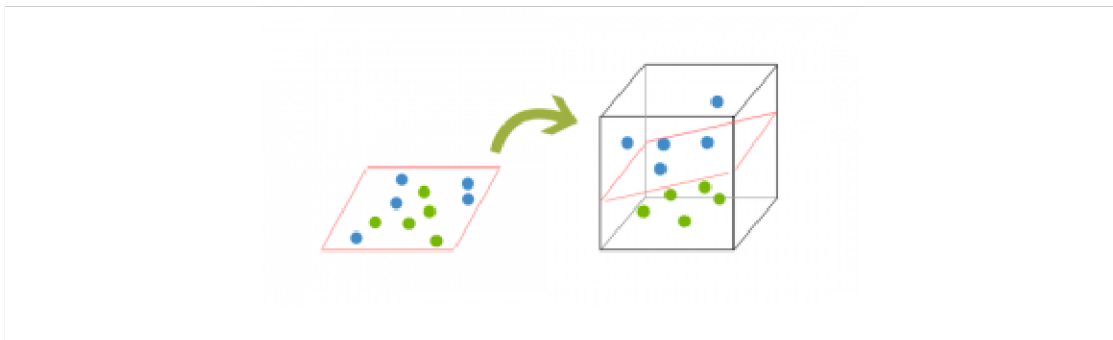


### **But what happens when there is no clear hyperplane?**

This is where it can get tricky. Data is rarely ever as clean as our simple example above. A dataset will often look more like the jumbled balls below which represent a linearly non separable dataset.



In order to classify a dataset like the one above it's necessary to move away from a 2d view of the data to a 3d view. Explaining this is easiest with another simplified example. Imagine that our two sets of colored balls above are sitting on a sheet and this sheet is lifted suddenly, launching the balls into the air. While the balls are up in the air, you use the sheet to separate them. This 'lifting' of the balls represents the mapping of data into a higher dimension. This is known as kernelling. You can read more on Kerneling [here](#).



Because we are now in three dimensions, our hyperplane can no longer be a line. It must now be a plane as shown in the example above. The idea is that the data will continue to be mapped into higher and higher dimensions until a hyperplane can be formed to segregate it.

## Pros & Cons of Support Vector Machines

### Pros

- Accuracy
- Works well on smaller cleaner datasets
- It can be more efficient because it uses a subset of training points

### Cons

- Isn't suited to larger datasets as the training time with SVMs can be high
- Less effective on noisier datasets with overlapping classes

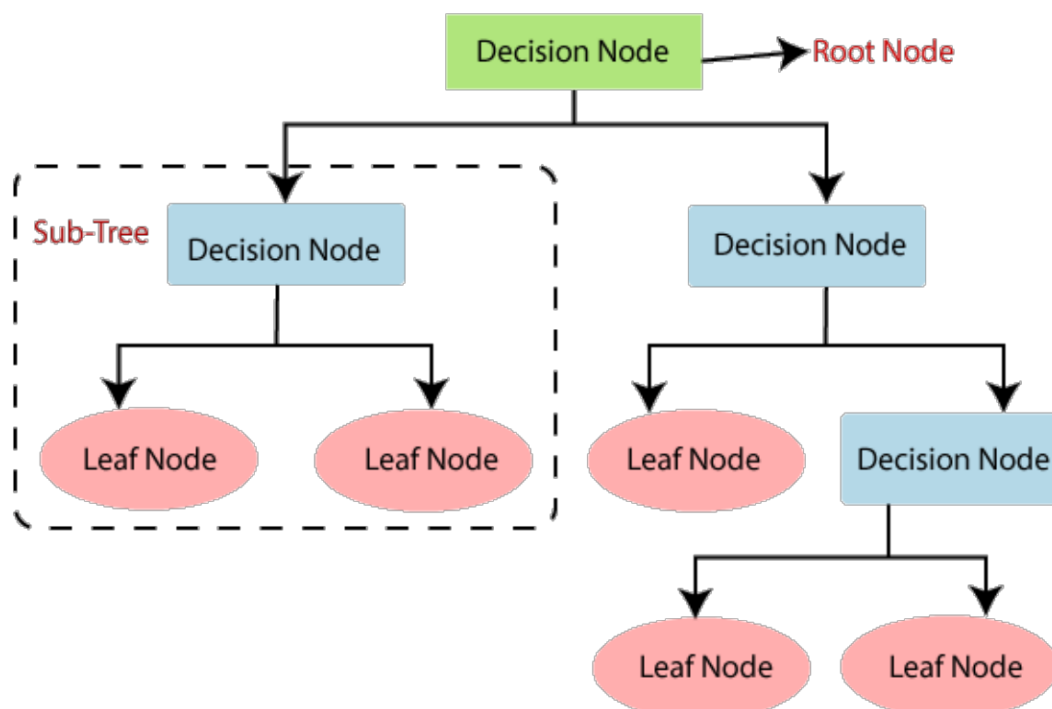
## SVM Uses

SVM is used for text classification tasks such as category assignment, detecting spam and sentiment analysis. It is also commonly used for image recognition challenges, performing particularly well in aspect-based recognition and color-based classification. SVM also plays a vital role in many areas of handwritten digit recognition, such as postal automation services. There you have it, a very high level introduction to Support Vector Machines.

## 2.4.6 Decision Trees

Decision trees are one of the most popular algorithms when it comes to data mining, decision analysis, and artificial intelligence. We will gently introduce you to decision trees and the reasons why they have gained so much popularity.

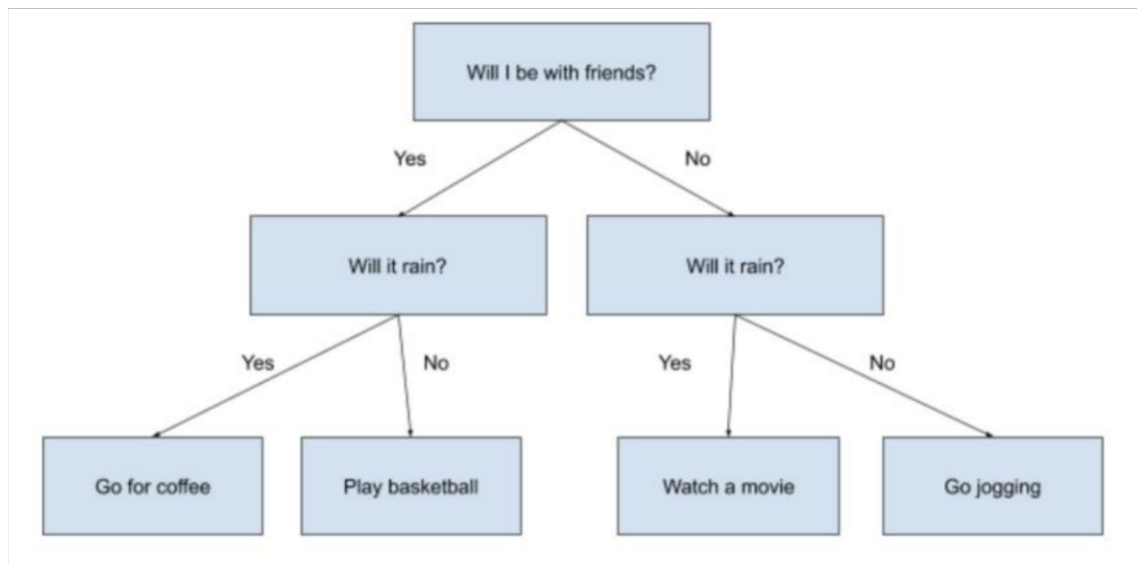
A **Decision Tree** is a tree-like graph with nodes representing the place where we pick an attribute and ask a question, edges represent the answers to the questions asked, and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface.



## Decision Tree in Layman's Terms

Imagine that you're planning next week's activities. The things that you'll get up to will pretty much depend on whether your friends have time and what the weather is like outside.

You come up with the following chart:



This chart sets out **simple decision rules**, which help you to decide what to do next week based on some other data. In this case, it's your friends' availability and weather conditions.

Decision trees do the same. They build up a set of decision rules in the form of a tree structure, which helps you to predict an outcome from the input data.

## Applications of Decision Trees

### Business Use-Cases

Decision trees mimic human decision-making and can therefore be used in a variety of business settings.

Companies often use them to predict future outcomes. For instance:

1. Which customer will stay loyal and which one will churn? (Classification decision tree)

2. By how much can we upsell a customer, given their product choices?  
(Regression decision tree)
3. Which article should I recommend to my blog readers next? (Classification decision trees)

### **Intuitive Advantages**

There are multiple reasons why decision trees are one of the go-to machine learning algorithms in real-life applications:

1. Intuitive
2. Informational
3. Scaling

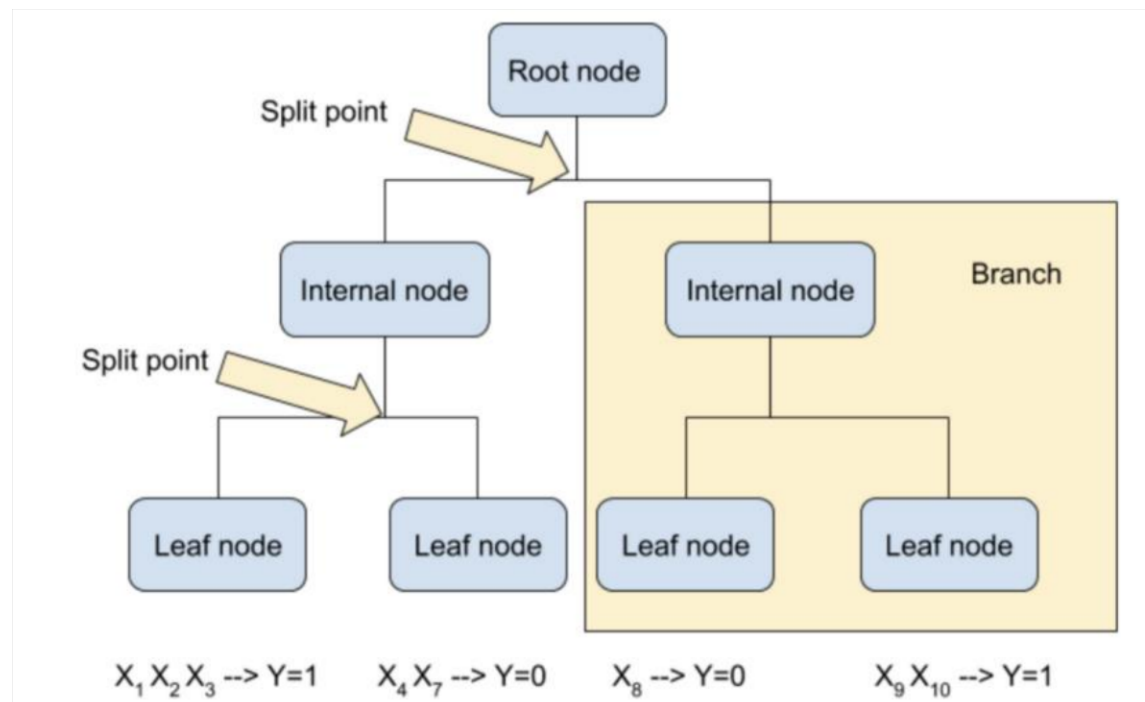
### **Machine Learning Approach to Decision Trees**

Decision trees belong to a class of supervised machine learning algorithms, which are used in both classification (predicts discrete outcome) and regression (predicts continuous numeric outcomes) predictive modeling.

The goal of the algorithm is to predict a target variable from a set of input variables and their attributes. The approach builds a tree structure through a series of binary splits (yes/no) from the root node via branches passing several decision nodes (internal nodes), until we come to leaf nodes.

It is here that the prediction is made.

Each split partitions the input variables into feature regions, which are used for lower splits. We can visualize the entire tree structure like this:



## Decision Tree Algorithms

There is no single decision tree algorithm. Instead, multiple algorithms have been proposed to build decision trees:

1. **ID3**: *Iterative Dichotomiser 3*
2. **C4.5**: *the successor of ID3*
3. **CART**: *Classification And Regression Tree*
4. **CHAID**: *Chi-square automatic interaction detection*
5. **MARS**: *multivariate adaptive regression splines*

Each new algorithm improves upon the previous ones, with the aim of developing approaches which achieve higher accuracy with noisier or messier data.

## Implementation

In general, we can break down the decision tree algorithm into a series of steps common across different implementations:

### Attribute selection:

1. Start with the entire dataset and look at every feature or attribute.
2. Look at all of the possible values of that attribute and pick a value which best splits the dataset into different regions.
3. What constitutes 'a best split' depends very much on whether we are building a regression tree algorithm or a classification tree algorithm.

- We'll expand upon the different methods for finding the **best split** below:

1. Split the dataset at the root node of the tree and move to the child nodes in each branch.
2. For each decision node, repeat the attribute selection and value for best split determination.
3. This is a greedy algorithm: It only looks at the best local split (not global optimum) given the attributes in its region to improve the efficiency of building a tree.
4. Continue iteratively until either:

a) We have grown terminal or leaf nodes so they reach each individual sample (there were no stopping criteria).

b) We reached some stopping criteria.

For example, we might have set a maximum depth, which only allows a certain number of splits from the root node to the terminal nodes. Or we might have set a minimum number of samples in each terminal node, in order to prevent terminal nodes from splitting beyond a certain point.

### Metrics for Decision Tree Classifiers

In classification problems, the two most popular metrics for determining the splitting point are Gini impurity and information gain:



- **Gini impurity:** As the name suggests, this measures how ‘pure’ our splits are. If a split results in one class being more predominant than another, e.g. 80% of class A and 20% of class B, this means that the split is 80% pure. The algorithm iteratively tries to find percentages like these of independent values, which produce homogenous classes.

$$\text{Gini}(K) = \sum_{i \in N} P_{i,K}(1 - P_{i,K}) = 1 - \sum_{i \in N} P_{i,K}^2$$

- $N$  is the list of classes (In this case  $N = \{\text{'Yes'}, \text{'No'}\}$ )
- $K$  is the category
- $P_{i,K}$  is the probability of category  $K$  having class  $i$

- **Information gain:** Information gain measures whether or not we lower the system’s entropy after splitting. Entropy, on the other hand, is defined as how chaotic our system is. This might sound abstract, but the concept is rather intuitive. If our decision tree were to split randomly without any structure, we would end up with splits of mixed classes (e.g. 50% class A and 50% class B). Chaos. But if the split results in sorting the classes into their own branches, we’re left with a more structured and less chaotic system.

$$\begin{aligned} IG(t) = & - \sum_{i=1}^m p(c_i) \log p(c_i) \\ & + p(t) \sum_{i=1}^m p(c_i | t) \log p(c_i | t) + p(\bar{t}) \sum_{i=1}^m p(c_i | \bar{t}) \log p(c_i | \bar{t}) \end{aligned}$$

$c_i$  represents the  $i$  th category,  $P(c_i)$  is the probability of the  $i$  th category.

$P(t)$  and  $P(\bar{t})$  are the probabilities that the term  $t$  appears or not in the documents.

$P(c_i | t)$  is the conditional probability of the  $i$  th category given that term  $t$  appeared, and  $P(c_i | \bar{t})$  is the conditional probability of the  $i$  th category given that term  $t$  does not appeared.

This is very similar to the Gini impurity logic, but information gain does not choose the split according to whether we get pure (structured, less chaotic, less entropic) segmentations after the split, but rather, by how much we improved on the entropy after the split.

When the algorithm traverses all possible values of an attribute, it calculates either

the Gini impurity at that point or information gain. The value for the attribute which best minimizes the cost function is used as a split.

## Metrics for Decision Tree Regressors

Introduced in the CART algorithm, decision tree regressors use *variance reduction* as a measure of the best split. Variance reduction indicates how homogenous our nodes are. If a node is extremely homogeneous, its variance (and the variance of its child nodes) will not be as big.

*The formula for variance is:*

$$\text{variance} = \frac{\sum (X-\mu)^2}{n}$$

The algorithm traverses different values of the independent variables, then picks such a variable and its value which generates the biggest variance reduction *after* the split.

## Advantages

**Decision trees offer several benefits:**

1. Interpretable
2. Little to no data preparation
3. Scale well
4. Handle numerical and categorical data
5. Robust to assumption violation

## Disadvantages

**Like most things, the machine learning approach also has a few disadvantages:**

1. Overfitting

2. Non-robust to input data changes
3. Biased towards the dominant class

## Improve the Model

There are several ways to improve decision trees, each one addressing a specific shortcoming of this machine learning algorithm by tuning hyperparameters:

- Minimum samples for leaf split
- Maximum depth
- Pruning
- Ensemble methods: Random forest
- Feature selection or dimensionality reduction
- Boosted trees

## 2.4.7 Random Forests

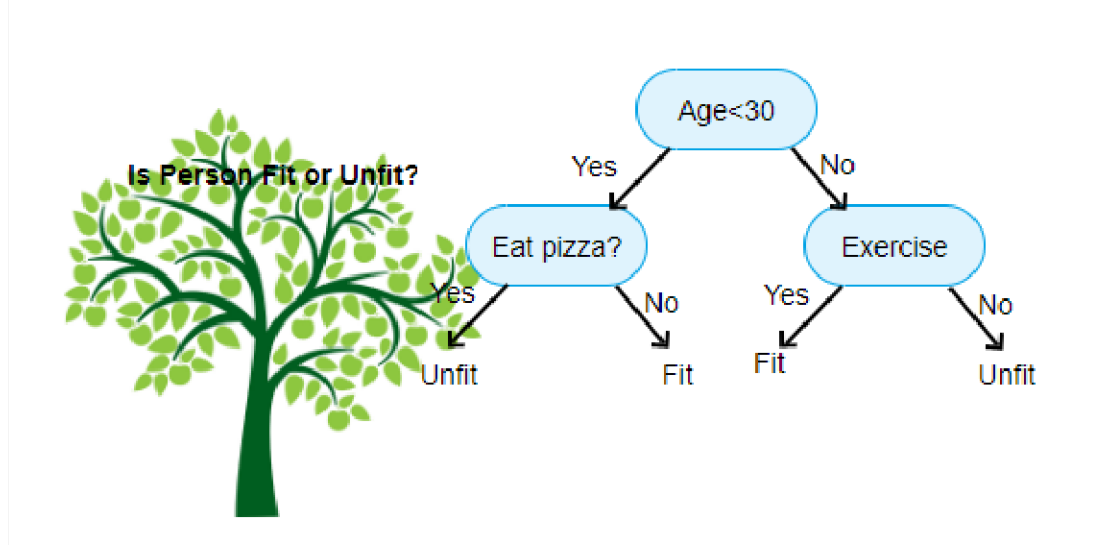
Random Forests is one of my favorite data mining algorithms. Invented by Leo Breiman and Adele Cutler back in the last century, it has retained its authenticity up to this day, no changes were added to it since its invention.

Without any exaggeration, it is one of the few universal algorithms. Random forests allow solving both the problems of regression and classification as well. It is good for searching for anomalies and selecting predictors. What is more, this algorithm is technically difficult to apply incorrectly. It is surprisingly simple in its essence. Unlike other algorithms, it has few configurable parameters. And at the same time, it is amazingly accurate.

Wow, so many advantages of using Random forests! It seems like a miracle for machine learning engineers ;) So, if you don't know yet how it works, it's the right time to fix this situation. Here is a learning adventure for beginners, where we see

things in terms of branches, leaves, and Random forests, of course.

## Decision Trees in a Nutshell



Let's first start with Decision Trees, because logically, there is no forest without trees.

Decision Trees is a non-parametric supervised learning algorithm that builds classification or regression models in the form of a tree structure. It breaks down a data set into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.

A Random Forest is actually just a bunch of Decision Trees. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from the root to the leaf represent classification rules.

A decision tree consists of three types of nodes:

- Decision nodes — typically represented by squares

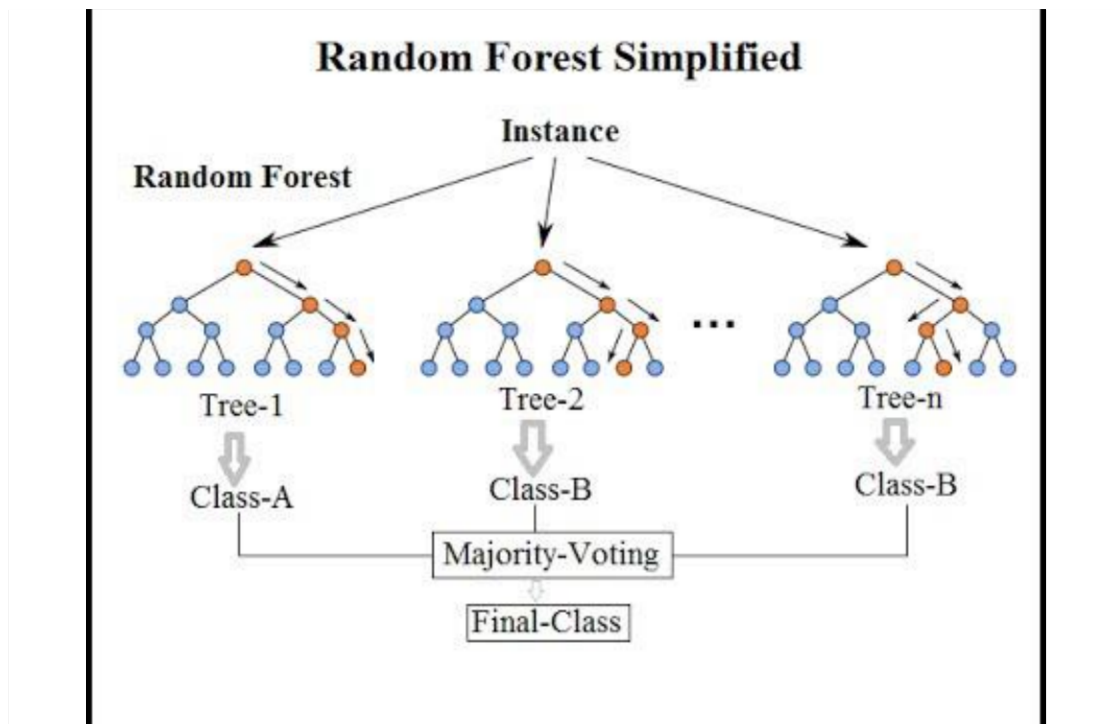
- Chance nodes — typically represented by circles
- End nodes — typically represented by triangles

So all-in-all, the learned tree can also be represented as a nested if-else rule to improve human readability. Trees have a high risk of overfitting the training data as well as becoming computationally complex if they are not constrained and regularized properly during the growing stage. This overfitting implies a low bias, high variance trade-off in the model. Therefore, in order to deal with this problem, we use Ensemble Learning, an approach that allows us to correct this overlearning habit and hopefully, arrives at better, stronger results.

## **What is an Ensemble Method?**

The method of ensembles is based on training algorithms that form many classifiers and then segment new data points, starting from voting or averaging. The original ensemble method is nothing but Bayesian averaging, but later algorithms include output coding error correction, bagging and boosting. Boosting is aimed at turning weak models into strong ones by building an ensemble of classifiers. Bagging also aggregates advanced classifiers, but it uses parallel training of basic classifiers. In the language of mathematical logic, bagging is an improving union, and boosting is an improving intersection.

In our case, a Random Forest (strong learner) is built as an ensemble of Decision Trees (weak learners) to perform different tasks such as regression and classification.



## What Is the Idea of Random Forest?

The idea is simple: let's say we have some very weak algorithm, say, a decision tree. If we make a lot of different models using this weak algorithm and average the result of their predictions, then the final result will be much better. This is the so-called Ensemble learning in action.

Well, here is a reason why Random forest is called this way, cause it creates many decision trees for the data and then averages the result of their predictions. A large number of decision trees are the parameters of the method, each of which is built according to a sample obtained from the original training select using bootstrap (sample with return).

An important point here is the element of randomness in the creation of each tree. After all, it is clear that if we create many identical trees, then the result of their averaging will have the accuracy of one tree.

## Simple explanation

A random forest is a collection of random decision trees (the number of

`n_estimators` in `sklearn`). You need to understand how to create one random decision tree.

Roughly speaking, to build a random decision tree that you start with a subset of your training samples. On each node, you arbitrarily draw a subset of functions (the number is determined by `max_features` in `sklearn`). For each of these functions, you will test different threshold values and see how they separate your samples according to a given criterion (usually entropy or gini, `criterion` in `sklearn`). Then you save the function and its threshold, which are the best way to separate your data and write it to node. When the construction of the tree finishes (this can be for various reasons: the maximum depth is reached (`max_depth` in `sklearn`), the minimum number of samples is reached (`min_samples_leaf` in `sklearn`), etc.), you look at the samples in each sheet and save the frequency of the marks. As a result, it looks like the tree gives you a section of your training samples according to meaningful functions.

Since each node is built from random functions, you understand that each tree constructed in this way will be different. This contributes to a good compromise between displacement and dispersion.

Then, in test mode, the test sample will go through each tree, giving you labels for each tree. The most represented label is usually the final result of the classification.

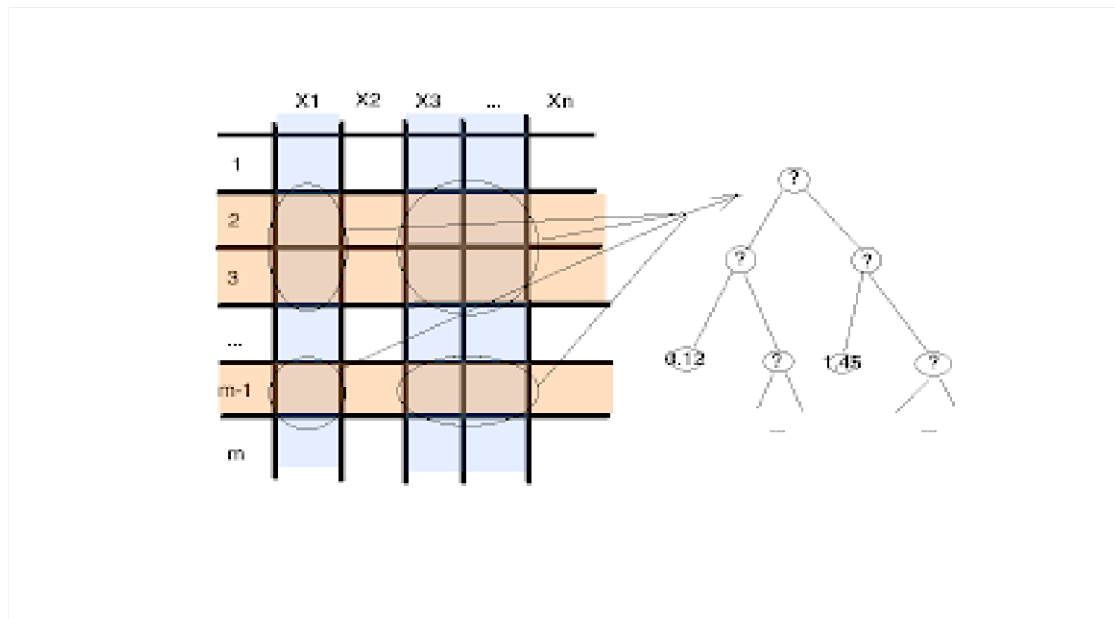
## How does it work?

Suppose we have some input data. Each column corresponds to a certain parameter, each row corresponds to a certain data element.

	$x_1$	$x_2$	$x_3$	...	$x_n$
1					
2					
3					
...					
$m-1$					
$m$					

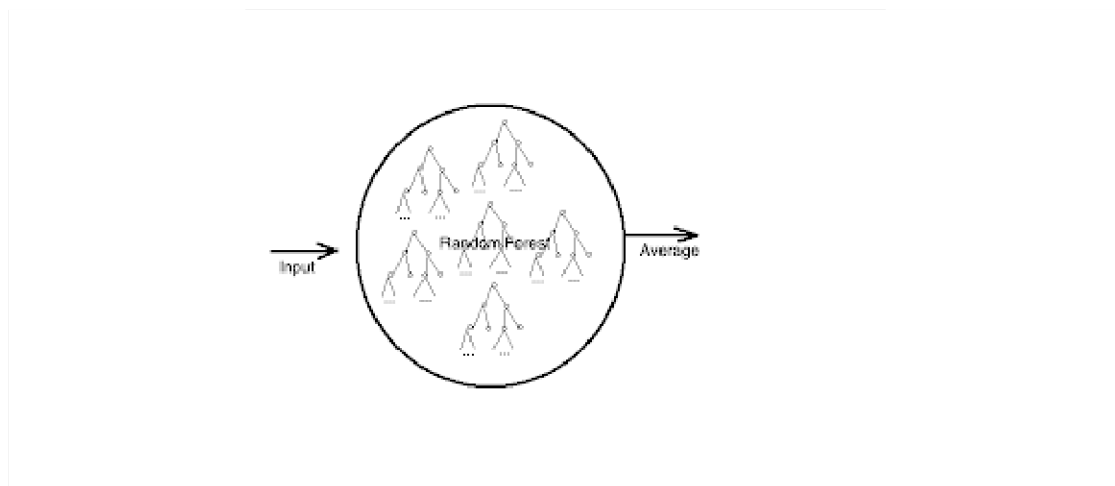


We can randomly select from the entire data set a certain number of columns and rows and build a decision tree from them.



Then we can repeat this process many times and get a lot of different trees. The process of the tree-building algorithm is very fast. And therefore, it will not be difficult for us to make as many trees as we need. At the same time, all of these trees are, in a sense, random, because we chose a random subset of data to create each of them.

The number of trees grown is often an important factor. This number may influence the achieved level of classification error. In addition, with sharply unbalanced classes (for example, a lot of 0 and only a small amount of 1), it is important to perform stratified sampling to even out the levels of classification error in each of these classes.



In the original version of the algorithm, a random subset is selected at each step of the tree construction. But this does not change the essence and the results are comparable.

This surprisingly simple algorithm, the most difficult step in its implementation — the construction of the tree decision tree. And despite its simplicity, it gives very good results in real tasks. From a practical point of view, it has one huge advantage: it requires almost no configuration. If we take any other machine learning algorithm, be it regression or a neural network, they all have a bunch of parameters and you need to know what algorithms are better to apply for a specific task.

The random forest algorithm has essentially only one parameter: the size of the random subset selected at each step of the tree construction. This parameter is important, but even the default values provide very acceptable results.

## Random Forests vs. Decision Trees

Both the random forest and decision trees are a type of classification algorithm, which are supervised in nature.

A decision tree is a graphical representation of all the possible solutions to a decision based on certain conditions. It's called a decision tree because it starts with a single box (or root), which then branches off into a number of solutions, just like a tree.

Random forests involve building several decision trees based on sampling

features and then making predictions based on majority voting among trees for classification problems or average for regression problems. This solves the problem of overfitting in Decision Trees.

When working with the forest, when constructing each tree at the stages of splitting the vertices, only a fixed number of randomly selected features of the training set is used (the second parameter of the method) and a complete tree is constructed (without truncation). In other words, each leaf of the tree contains observations of only one class.

## **2.4.8 Principal Components Analysis**

We will have an in-depth look at principal components analysis or **PCA**. We start with a simple explanation to build an intuitive understanding of PCA. In the second part, we will look at a more mathematical definition of Principal components analysis.

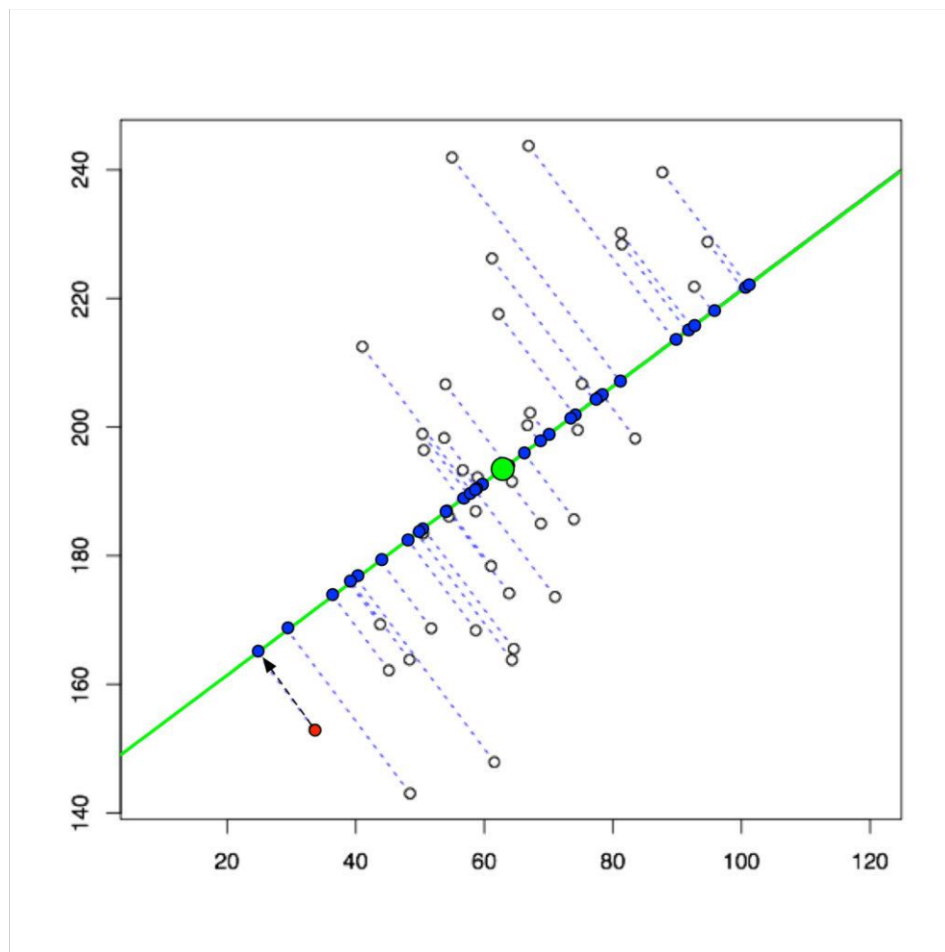
Lastly, we learn how to perform PCA in Python.

### **What is Principal Components Analysis (PCA)**

Principal Components Analysis, also known as PCA, is a technique commonly used for reducing the dimensionality of data while preserving as much as possible of the information contained in the original data. PCA achieves this goal by projecting data onto a lower-dimensional subspace that retains most of the variance among the data points.

What is dimensionality reduction, and what is a subspace? Let's illustrate this with an example.

If you have data in a 2-dimensional space, you could project all the data points onto a line using PCA.



You have essentially reduced the dimensionality of your data from 2D to 1D. The 1D space (your line) is a subspace of the 2D coordinate system.

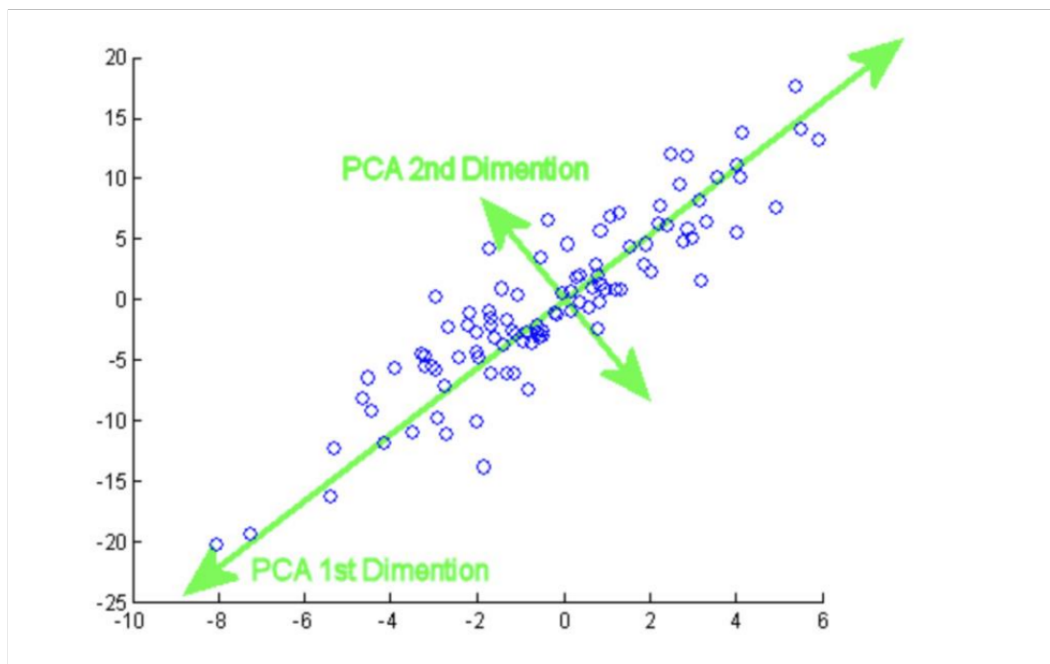
The green line has been constructed using mathematical optimization to maximize the variance between the data points as much as possible along that line. We call this line our first principal component. Naturally, the points on the line are still closer to each other than in the original 2D space because you are losing a dimension to distinguish them. But in many cases, the simplification achieved by dimensionality reduction outweighs the loss in information, and the loss can be partly or fully reconstructed.

Imagine you are taking a photo of some people. You are reducing the dimensionality from 3D (the real world) to a 2D representation. You are losing some explicit 3D information, such as the distance between a person in the front and another person further in the back. However, you will have a pretty decent idea of how far these two persons are apart in reality because the person in the back will

appear smaller than the person in the front. So the 3D information is not completely lost but sort of encoded in the 2D image.

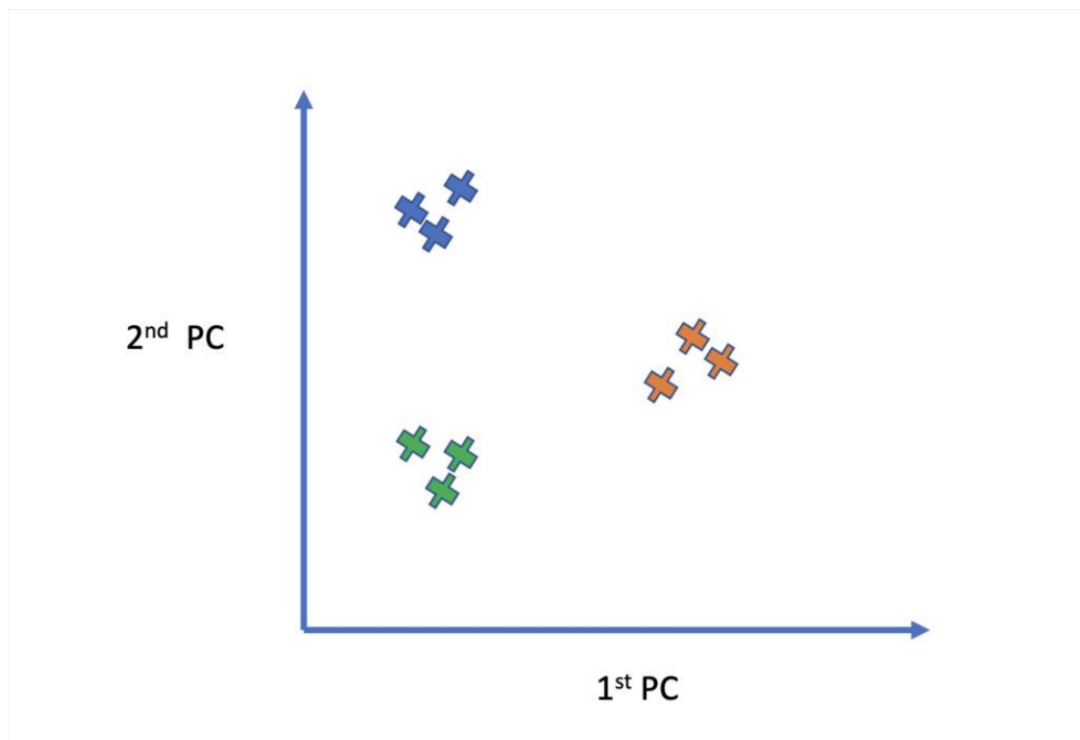
In our previous example, we only had one principal component. Once you move into higher dimensional spaces, you'll likely use several principal components because the variance explained by one principal component is often insufficient. Principal components are vectors that are orthogonal to each other. This means they form a 90-degree angle. Mathematically, orthogonal vectors are independent, meaning the variance explained by the second principal component does not overlap with the variance of the first. So they represent information as efficiently as possible. The first principal component will capture most of the variance; the second principal component will capture the second-largest part of the variance that has been left unexplained by the first one, etc.

Practically speaking, principal components are feature combinations that represent the data and its differences as efficiently as possible by ensuring that there is no information overlap between features. The original features often display significant redundancy, which is the main reason why principal components analysis works so well at reducing dimensionality.

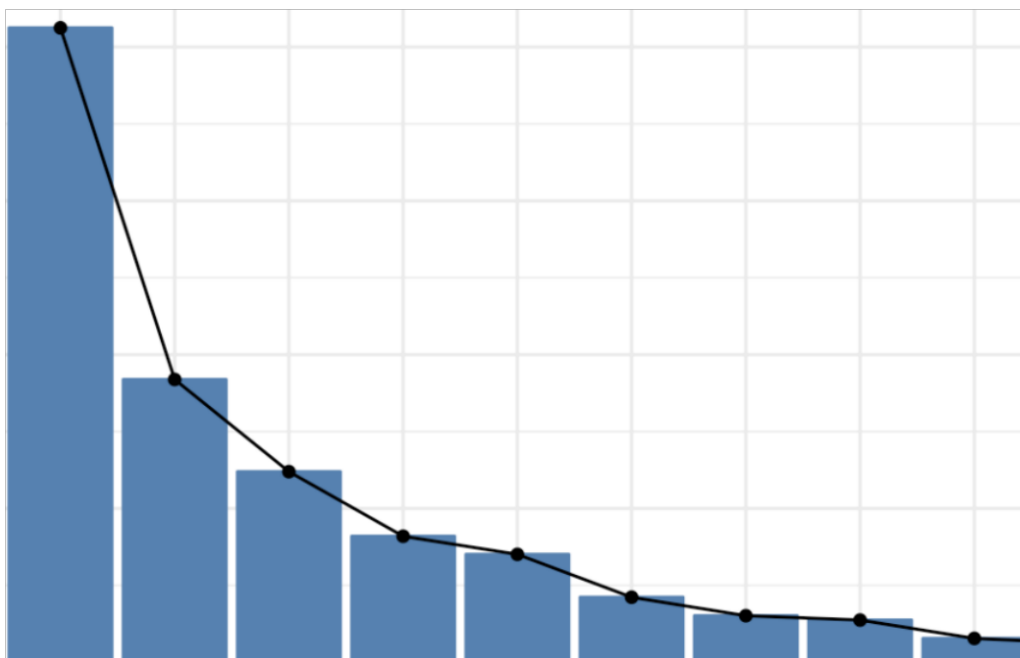


If you plot your data in your lower-dimensional subspace with the principal components as your axes, similar data points should cluster together. This happens

because you are explicitly focusing on axes that maximize the variance.



In highly dimensional datasets, the vast majority of the variance in the data is often captured by a small number of principal components. A plot of the distribution of the variance across principal components may look like this.



As you can see, the first principal component explains vastly more than the following ones. This allows us to project even highly dimensional data down to

relatively low-dimensional subspaces.

How does Principal Components Analysis Work?

Principal Components Analysis achieves dimensionality reduction through the following steps.

### 1. Standardize the data

The variables that make up your dataset will often have different units and different means. This can cause issues such as producing extremely large numbers during the calculation. To make the process more efficient, it is good practice to center the data at mean zero and make it unit-free. You achieve this by subtracting the current mean from the data and dividing by the standard deviation. This preserves the correlations but ensures that the total variance equals 1.

### 2. Calculate the Covariance Matrix

Principal components analysis attempts to capture most of the information in a dataset by identifying the principal components that maximize the variance between observations. The covariance matrix is a symmetric matrix with rows and columns equal to the number of dimensions in the data. It tells us how the features or variables diverge from each other by calculating the covariance between the pairwise means.

$$\begin{bmatrix} \text{COV}(x_1, x_1) & \dots & \text{COV}(x_1, x_n) \\ \dots & & \\ \text{COV}(x_n, x_1) & \dots & \text{COV}(x_n, x_n) \end{bmatrix}$$

Example covariance matrix

If you want to learn more about covariance matrices, I suggest you check out my post on them.



### 3. Calculate the Eigenvectors and Eigenvalues of the Covariance Matrix

Eigenvectors are linearly independent vectors that do not change direction when a matrix transformation is applied. Eigenvalues are scalars that indicate the magnitude of the Eigenvector. If you want to learn more, check out my post on Eigenvectors and Eigenvalues.

The Eigenvectors of the covariance matrix point in the direction of the largest variance. The larger the Eigenvalue, the more of the variance is explained. In other words, the Eigenvector with the largest Eigenvalue corresponds to the first principal component, which explains most of the variance, the Eigenvector with the second-largest Eigenvalue corresponds to the second principal component, etc.

The reason why Eigenvectors correspond to principal components is buried in an elaborate mathematical proof which we will tackle in the next section. But in this section, we focus on intuition rather than complicated proofs, so for now, we just take this relationship for granted.

### 4. Reduce Dimensionality

As stated previously, the principal components are efficient feature combinations that ensure that the information explained does not overlap between features. Eliminating information redundancy already helps in reducing dimensionality. But since the percentage of the overall variance explained declines with every new principal component, we can reduce dimensionality further by eliminating the least important principal components. At this stage, we have to decide how many principal components are sufficient and how much information loss we can tolerate.

Lastly, we need to project the data from our original feature space down to the reduced space spanned by our principal components.

Usually, you will perform principal components analysis using a software tool that will execute all the steps for you. In this case, a high-level understanding as presented up until here is usually enough. But if you are interested in the mathematical details that explain why PCA works and why Eigenvectors represent principal components, read on.

## 2.4.9 K-means Clustering

Machine learning is the science of making reliable predictions of outputs for inputs that the machine has never seen before. It consists of three main phases:

1. Composing a training set. Training sets contain samples (data points) such that each sample is **classified** into a group, or the set has to undergo a certain procedure to classify its samples before moving on to the next phase. This procedure is what distinguishes **supervised learning** from **unsupervised learning**.
2. Creating the model, and feeding it the classified training set.
3. Refining the model.

### Let's see where the above fits:

- Machine learning aims to create a model that can make decisions regarding new inputs.
- A training set is fed to the model. Usually represented by a set of samples (rows), each described by a set of features (columns).
- If samples of the training set are labeled (there exists a final feature dictating what class each sample belongs to), the learning procedure is supervised. Otherwise, it's unsupervised.
- In the case of unsupervised learning, we have to categorize each sample before feeding them to the model.
- This categorization process is called **clustering**. Put formally: Clustering is the process after which our samples are classified into groups, such that samples within one group are similar to each other than other samples in different groups.

- There are various types of clustering algorithms, each has its unique strengths and weaknesses.
- One particular clustering type is named partitional clustering.
- Partitional clustering divides samples into non-overlapping groups. That is, no sample can be a member of more than one cluster, and every cluster must have at least one sample.
- Our algorithm of interest (K-means clustering) is a widely known partitional clustering technique.

That was a lot to grasp. Feel free to re-read the above multiple times until you have a feel for it.

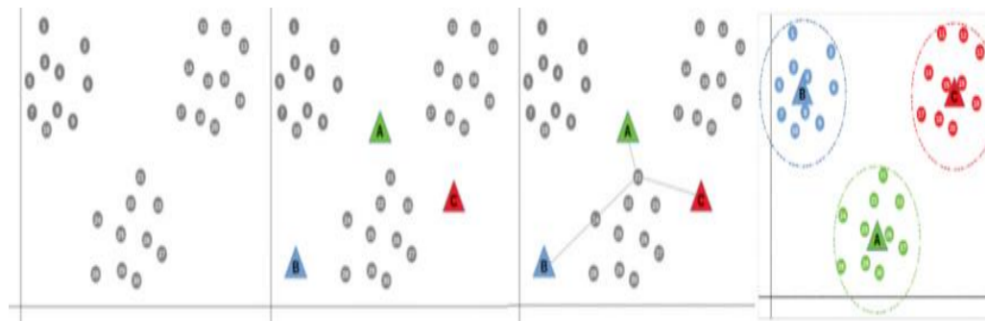
## **K-means: In Details**

### **How it Works**

The main idea behind k-means is to set  $k$  points (samples) to be our **centroids**, and each one of those centroids will be going around trying to center itself in the middle of one of the  $k$  clusters we have.

### **Pseudocode**

1. From the given dataset, choose  $k$  points at random to be our initial centroids of choice.
2. Set the closest centroid of each point to be its label.
3. Take the mean of the points belonging to each centroid, and set it as the new value of the centroid.
4. Stop when convergence. That is, the new centroids are equal to the old centroids. Otherwise, set the old centroids to the values of the new centroids, and repeat steps 2 through 4.



**The Lifecycle of K-means Clustering**

**Note:** The initial centroids of choice don't have to be actual samples.

### Complexity Analysis

K-means has a time complexity of  $(pkif)$ , and space complexity of:  $\mathcal{O}(p + kf)$ . Where:

- $p$ : is the number of points.
- $k$ : is the number of clusters.
- $i$ : is the number of iterations until convergence.
- $f$ : is the number of features.

### Implementation

A Python implementation of the above pseudocode:

```
1 def k_means(dataset, k):
2     # Choose k initial random centroids
3     centroids = dataset[np.random.choice(dataset.shape[0], k)]
4
5     while True:
6         """
7         Set the closest centroid of each point to be its label.
8         the pairwise_distances_argmin is a utility imported from sklearn.metrics package
9         and it finds the closest point from a set of points B for each point in A.
10        It assumes that each point is composed of X and Y. In other words,
11        that we are dealing with two features for each point.
12        """
13        labels = pairwise_distances_argmin(dataset, centroids)
14
15        # Set the new value of each centroid as the mean of all points belonging to that centroid
16        new_centroids = np.array([dataset[labels == label].mean(0) for label in range(k)])
17
18        if np.all(new_centroids == centroids):
19            break
20
21        centroids = new_centroids
22
23    return labels
```

K-means.py hosted with ❤ by GitHub

[view raw](#)

A Python Implementation of K-means Clustering

## K-means: In Action

### Generating Dummy Data

Let's put the above implementation to use. First off, we will use the `make_blobs` utility from the `sklearn.datasets` module to generate dummy samples and distribute them between clusters.

```
dataset, true_labels = make_blobs(
    n_samples=300,
    n_features=2,
    centers=4,
    cluster_std=.6,
    random_state=0
)
```

Here we are telling our function to generate 300 samples, each with 2 features, distributed along 4 clusters, and with a cluster standard deviation (spacing between

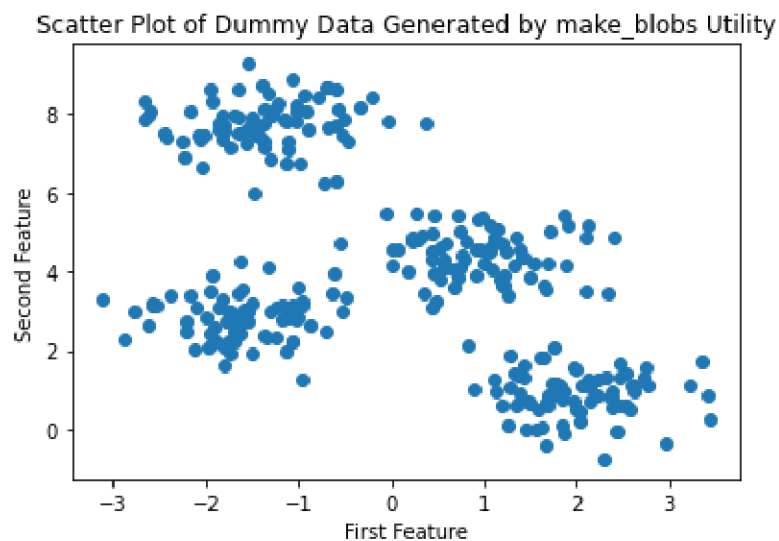
clusters) of 0.6. The `random_state` argument is the seed of randomness (useful for generating the same dataset from different runs). It returns a dataset, and the `true_label` of its samples (useful for validating our implementation later).

### Plotting the Dataset

```
second_feature_vals, first_feature_vals = np.rot90(dataset)

plt.scatter(first_feature_vals, second_feature_vals)
plt.title("Scatter Plot of Dummy Data Generated by
make_blobs Utility")
plt.xlabel("First Feature")
plt.ylabel("Second Feature")
plt.show()
```

We extract the values of the first and second features separately, then we draw a scatter plot of them using `matplotlib.pyplot`. The resulted plot:



Scatter Plot of the Dummy Data

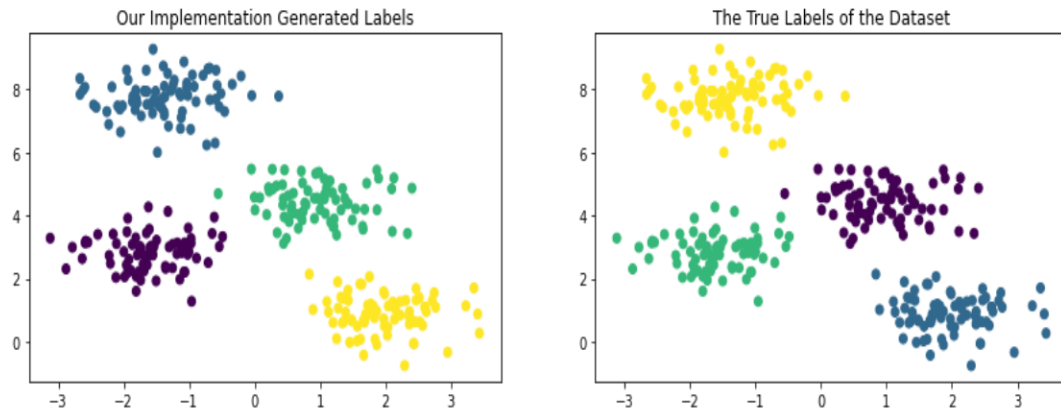
Looking at the generated scatter plot, we can conclude that setting  $k$  to 4 clusters sounds reasonable enough. We are also sure that this is the precise number of clusters since we set it to 4 when we generated our blobs with `make_blobs`.

## Running Our Implementation

```
labels = k_means(dataset, 4)
plt.subplot(1, 2, 1)
plt.scatter(first_feature_vals, second_feature_vals,
            c=labels)
plt.title("Our Implementation Generated Labels")
plt.subplot(1, 2, 2)
plt.scatter(first_feature_vals, second_feature_vals,
            c=true_labels)
plt.title("The True Labels of the Dataset")
plt.subplots_adjust(right=2, wspace=0.2)
plt.show()
```

We draw 2 scatter plots:

- One passing it the labels generated from our implementation,
- and the other passing it the true\_labels generated by the make\_blobs utility earlier.



Our Implementation Generated Labels vs The True Labels of the Dataset

Looking at both images side by side, we can conclude that our implementation has put its money where its mouth is and that the generated clusters from our implementation perfectly match the true clusters.



## Pros and Cons

### Pros

- Simplicity and popularity.
- Guarantees convergence.
- Serves as a good estimate of the centroids' initial positions.

### Cons

- Having to specify the number of clusters  $k$ .
- Depends on random initial values, which may result in inconsistent results between different runs.
- Some data has to be scaled before clustering.

## 2.5 Neural networks and other AI

Neural networks have become fashionable. But the technology is not as new as one might think. The first artificial neurons were actually created in the 1950s, so they are just as old as other forms of artificial intelligence and almost as old as digital computers in general.

The reason artificial neural networks did not catch on earlier is that they require relatively big amounts of computational power. Early computers did not have the ability to run neural network code at sufficient speeds to be practical.

Only since around 2010 has the development of hardware caught up with the requirements of deep neural networks. Suddenly, after decades of silence around neural networks, all the magical applications that we can see around us today became possible.

Before artificial neural networks were widely used, a programmer would tell the computer what to do by issuing a sequence of commands. The machine would then

execute these commands one by one. This way of programming a computer is called *imperative programming*. The problem with imperative programming is that a programmer can only solve problems for which she can provide a list of commands that solves the problem. Unfortunately, many real-world problems are so hard that programmers don't know how to solve them.

**Dealing with noisy input is precisely what neural networks are particularly good at.**

For example, assume that you want to recognise a handwritten letter. Let's say the letter *a*. You could try to describe to the computer what an *a* looks like. The problem is that different people have different handwriting. Even the same person might write the letter *a* differently from time to time. It would be very difficult to describe in abstract terms how an *a* should look and which variations should still count as an *a* while others do not.



Other kinds of real-world data that are not precise, but noisy, would cause similar problems. For example, voice recognition. Right now, I am dictating this paragraph, and the computer recognises my spoken words and types them into a document. It would be extremely hard, if not impossible, to describe in abstract terms how each one of these different words sounds. Particularly since no two utterances of the same word sound exactly the same. One might pronounce particular vowels differently from time to time, or one might have a cold, or other environmental noises might interfere with the recording. As we will see later, dealing with noisy input is precisely what neural networks are particularly good at.

Artificial neural networks, as opposed to conventional imperative programs, can be taught to recognise patterns *by example*. This means that we can create systems that are able to recognise patterns even if we are not able to clearly describe the pattern itself. A neural network that has been shown a great number of different

handwritten letters will be able to recognise these letters even if the programmer is not able to describe their differences in a precise way. A neural network that has been successfully trained to recognise a spoken word will be able to identify this word even if the programmer does not know how to describe the word in terms of sound frequencies.

Additionally, neural networks are not sensitive to small changes in their input patterns. If I have trained a neural network by showing it different versions of handwritten letters  $a$ , then the neural network will be able to recognise not only these letters that it has been trained upon but also similar but different letters. Neural networks can deal with noisy input.

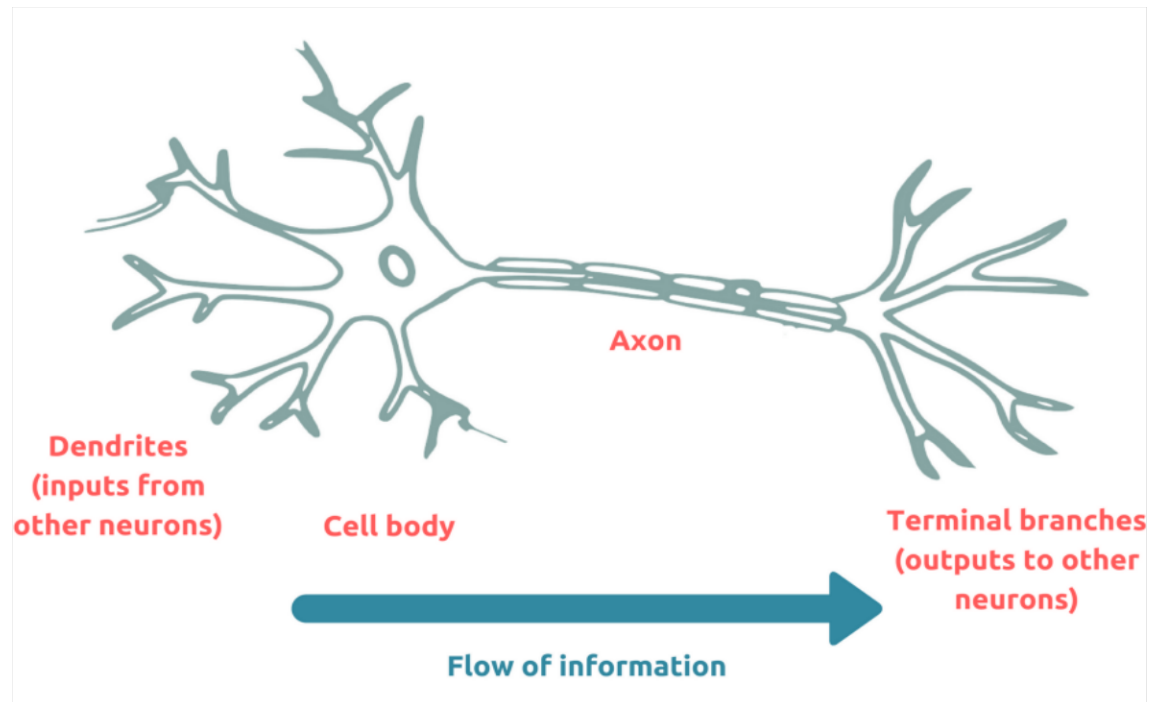
Then, we will see how neural networks achieve these results. Let us first begin with a look at the basic idea behind biological neurons.

## **Biological neurons**

Artificial neural networks are inspired by the neurons in living organisms. Although we don't know precisely and in every detail how biological neurons work, the basic principle behind them is easy to understand.

Whether artificial neural networks actually work like biological ones or not does not really matter much for AI. In the same way that an aeroplane or helicopter can fly without having feathers like a bird, an artificial neural network can process information successfully and perform some of the functions of a biological brain without needing to work technically in exactly the same way.

Here is a very basic image of the functional architecture of a biological neuron.



On the left side of this image, we can see the input side of the neuron. This is where the signals from other neurons come in. Every neuron is connected to many other neurons through long tendrils called dendrites. All the dendrites end up connecting to the cell body. The cell body processes the input signals and decides whether it should emit an output signal or not. If the cell decides to emit an output signal, this signal then travels down the axon. At the end of the axon, the signal splits into many tendrils again, which then connect to the dendrites of other neurons.

Each dendrite connects to its neuron at one point that is called a *synapse*. Every synapse has the ability to either strengthen or weaken the signal that comes through it. In an abstract sense, we can see the synapse as a kind of regulator that turns the input signal's "volume" up or down. We, therefore, speak of *synaptic weights*. A synaptic weight is just a factor by which the synapse multiplies the input signal before it reaches the neuron. Each synapse can have a different weight, and in this way, each synapse can process its signal differently, either strengthening or weakening it.

The human brain is made up of billions of such neurons. These neurons are arranged into bigger groups that specialize in particular kinds of information processing. Some neurons are responsible for the processing of images from our eyes, while other parts of our brain specialize in memory, hearing, and smell, the processing of speech or in controlling our muscles.

## The perceptron

In 1957, Rosenblatt developed the first artificial neuron. It is actually a very simple computational device.

The perceptron, as it is called, is inspired by biological neurons. On the left side, you can see the inputs to the perceptron, which correspond to the dendrites of a biological neuron. Each input has its own synaptic weight. A synaptic weight is just a factor that we multiply the input signal by. It is easiest to think of these weights as little “volume dials” that regulate the strength of the input signal. So Weight 1 would be a number between 0 and 1 that is multiplied by Input 1. In this way, the weight can regulate the strength of the input between a minimum of 0 and a maximum of the value of Input 1. The same applies to all other inputs and weights. We will see in a moment why these weights are important.

**If the sum of the weighted inputs exceeds a particular threshold, then the neuron will produce an output.**

All these weighted inputs are then fed into a function that adds them all up. This is the main processing unit of the neuron. If the sum of the weighted inputs exceeds a particular threshold, then the neuron will produce an output on its output side (which corresponds to the axon). If the sum of the inputs is not strong enough (that is, if it is lower than the threshold value), the neuron will not fire but stay quiet, effectively “swallowing” its input signals.

The threshold, therefore, is a cut-off value. If the sum of the inputs is lower than the threshold, the neuron stays quiet. If the sum of the inputs is bigger than the threshold, the neuron will produce an output signal.

Sometimes it is useful that neurons do not only work in this binary way, where the output is “silence” or “signal”, or in other words, “0” or “1”. Instead, we might want the neuron to produce a signal that is in some way proportional to the sum of its inputs. In the image you can see that a particular function is used. This is called the *sigmoid function*. There are many functions that one can use to calculate the output signal from the sum of the input signals of the neuron. All have different

properties and are suitable for particular kinds of applications.

But we don't need to go into this detail right now.

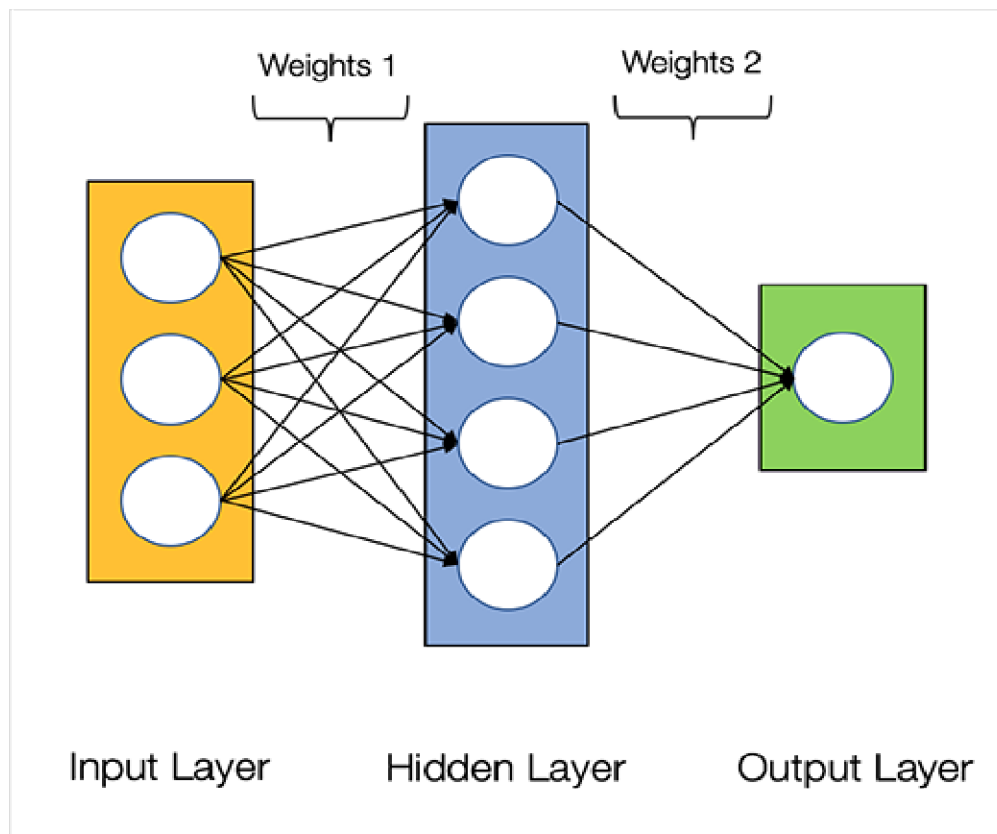
### What's a Neural Network?

Most introductory texts to Neural Networks brings up brain analogies when describing them. Without delving into brain analogies, I find it easier to simply describe Neural Networks as a mathematical function that maps a given input to a desired output.

Neural Networks consist of the following components

- An **input layer**,  $x$
- An arbitrary amount of **hidden layers**
- An **output layer**,  $\hat{y}$
- A set of **weights** and **biases** between each layer,  $W$  and  $b$
- A choice of **activation function** for each hidden layer,  $\sigma$ . In this tutorial, we'll use a Sigmoid activation function.

The diagram below shows the architecture of a 2-layer Neural Network (note that the input layer is typically excluded when counting the number of layers in a Neural Network)



Architecture of a 2-layer Neural Network

Creating a Neural Network class in Python is easy.

## Training the Neural Network

The output  $\hat{y}$  of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

You might notice that in the equation above, the weights  $W$  and the biases  $b$  are the only variables that affects the output  $\hat{y}$ .

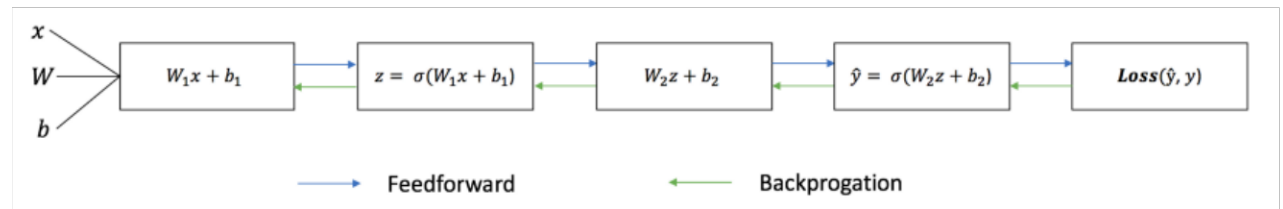
Naturally, the right values for the weights and biases determines the strength of the predictions. The process of fine-tuning the weights and biases from the input data is known as **training the Neural Network**.

**Each iteration of the training process consists of the following steps:**



- Calculating the predicted output  $\hat{y}$ , known as **feedforward**
- Updating the weights and biases, known as **backpropagation**

The sequential graph below illustrates the process.



## Feedforward

As we've seen in the sequential graph above, feedforward is just simple calculus and for a basic 2-layer neural network, the output of the Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

Let's add a feedforward function in our python code to do exactly that. Note that for simplicity, we have assumed the biases to be 0.

However, we still need a way to evaluate the “goodness” of our predictions (i.e. how far off are our predictions)? The **Loss Function** allows us to do exactly that.

## Loss Function

There are many available loss functions, and the nature of our problem should dictate our choice of loss function. In this tutorial, we'll use a simple **sum-of-squares error** as our loss function.

$$\text{Sum - of - Squares Error} = \sum_{i=1}^n (y - \hat{y})^2$$

That is, the sum-of-squares error is simply the sum of the difference between each predicted value and the actual value. The difference is squared so that we measure the absolute value of the difference.

**Our goal in training is to find the best set of weights and biases that**

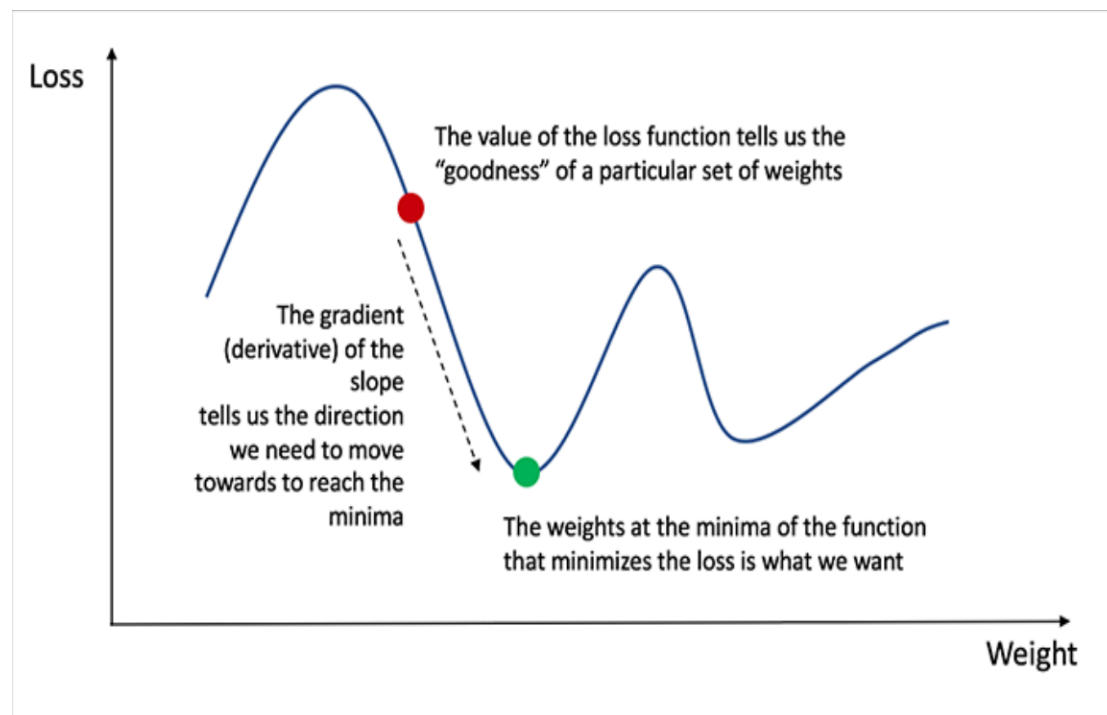
**minimizes the loss function.**

## Backpropagation

Now that we've measured the error of our prediction (loss), we need to find a way to propagate the error back, and to update our weights and biases.

In order to know the appropriate amount to adjust the weights and biases by, we need to know the **derivative of the loss function with respect to the weights and biases.**

Recall from calculus that the derivative of a function is simply the slope of the function.



Gradient descent algorithm

If we have the derivative, we can simply update the weights and biases by increasing/reducing with it (refer to the diagram above). This is known as **gradient descent**.

However, we can't directly calculate the derivative of the loss function with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. Therefore, we need the **chain rule** to help us calculate

it.

$$Loss(y, \hat{y}) = \sum_{i=1}^n (y - \hat{y})^2$$

$$\frac{\partial Loss(y, \hat{y})}{\partial W} = \frac{\partial Loss(y, \hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \quad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * z(1-z) * x$$

Chain rule for calculating derivative of the loss function with respect to the weights. Note that for simplicity, we have only displayed the partial derivative assuming a 1-layer Neural Network.

Phew! That was ugly but it allows us to get what we needed — the derivative (slope) of the loss function with respect to the weights, so that we can adjust the weights accordingly.

Now that we have that, let's add the backpropagation function into our python code.

```

class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        # application of the chain rule to find derivative of the loss function with respect to weights
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) * sigmoid_derivative(self.output)))
        d_weights1 = np.dot(self.input.T, (np.dot(2*(self.y - self.output) * sigmoid_derivative(self.output), self.weights2.T)))

        # update the weights with the derivative (slope) of the loss function
        self.weights1 += d_weights1
        self.weights2 += d_weights2

```

neural\_network\_backprop.py hosted with ❤ by GitHub [view raw](#)

## Multilayer networks

A single artificial neuron does not yet do anything very interesting. In order to get artificial neurons to do something useful, we must connect them with each other, just as biological neurons are connected to each other in our brains.

In the picture, we see an example of a small artificial neural network that has eight neurons.

You see that the neurons are now arranged in different layers. First, we have an *input layer* on the left. This is where the signals come into the neural network. Then we have one layer of neurons that are inside the network, which is where the biggest part of the information processing happens. This layer is called the *hidden layer* because it is not connected to the outside world and is, therefore, “hidden” from the viewpoint of an outside observer. And finally, we have an *output layer* of two neurons, which is how the neural network will communicate the results of its processing to us. The output units will usually be connected to some kind of output device, for example, a computer program that processes the network’s output further, or an LED display, a lamp, a screen, the steering wheel of a self-driving car, a

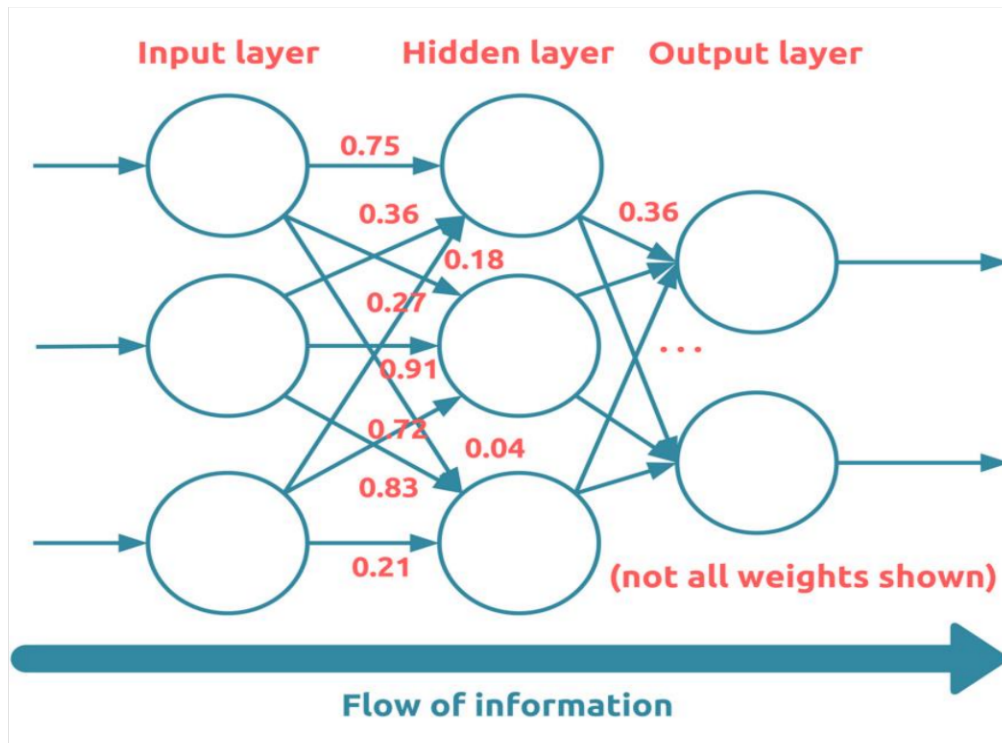
speaker, or any other device that we want the network to control.

**Multilayer networks can process much more information than a single neuron.**

Such a network is called a *multilayer network*. Multilayer networks can process much more information than a single neuron, and they can exhibit very complex information processing behaviors. The signals, in this case, travel from the input side, through the hidden layers, to the output.

Please observe that every neuron in each layer is connected to all the neurons in the next layer. The connections go only in one direction: from the input side to the output side. Neurons in this type of network are not connected backwards (but there are other types of artificial neural networks where the connections can form loops). So the signals travel always in one direction from the input to the output.

A network like this, where every neuron is connected to all neurons of the next layer, is called a fully connected feed-forward network. If only two layers in a bigger network are connected in this way, then we would speak of these as fully connected layers. Feed-forward means that the signals are fed in a forward direction through the network; and fully connected means that every neuron in one layer is connected to all the neurons in the next layer.



Please also observe that between the layers, we have the synaptic weights. The weights are not visible here as circles or dials, but they are represented with numbers that sit beside the arrows. These weights are crucial to the functioning of the network. Without them, every input signal would just walk through the whole network and come out unchanged on the output side, and no processing at all would be done.

We will explain this in detail later, but for the moment just observe that we can change the output signal in any way we want just by adjusting these synaptic weights. When the network learns, it adjusts these weights so that it produces the desired output signal for every input signal. So these synaptic weights are exactly where the knowledge of the network is stored.

## Importance of Neural Network

Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are *nonlinear* and *complex*. For instance, they can do the following tasks.

## Make generalizations and inferences

Neural networks can comprehend unstructured data and make general observations without explicit training. A neural network would know that both sentences mean the same thing.

## Where we can use Neural Networks?

Neural networks have several use cases across many industries, such as the following:

- Medical diagnosis by medical image classification
- Targeted marketing by social network filtering and behavioral data analysis
- Financial predictions by processing historical data of financial instruments
- Electrical load and energy demand forecasting
- Process and quality control
- Chemical compound identification

Important applications of Neural Networks are given below,

### 1. Computer vision

Computer vision is the ability of computers to extract information and insights from images and videos. With neural networks, computers can distinguish and recognize images similar to humans. Computer vision has several applications, such as the following:

- Visual recognition in self-driving cars so they can recognize road signs and other road users
- Content moderation to automatically remove unsafe or inappropriate content from image and video archives
- Facial recognition to identify faces and recognize attributes like open eyes, glasses, and facial hair



- Image labeling to identify brand logos, clothing, safety gear, and other image details

## 2. Speech recognition

Neural networks can analyze human speech despite varying speech patterns, pitch, tone, language, and accent. Virtual assistants like Amazon Alexa and automatic transcription software use speech recognition to do tasks like these:

- Assist call center agents and automatically classify calls
- Convert clinical conversations into documentation in real time
- Accurately subtitle videos and meeting recordings for wider content reach

## 3. Natural language processing

Natural language processing (NLP) is the ability to process natural, human-created text. Neural networks help computers gather insights and meaning from text data and documents. NLP has several use cases, including in these functions:

- Automated virtual agents and chatbots
- Automatic organization and classification of written data
- Business intelligence analysis of long-form documents like emails and forms
- Indexing of key phrases that indicate sentiment, like positive and negative comments on social media
- Document summarization and article generation for a given topic

## 4. Recommendation engines

Neural networks can track user activity to develop personalized recommendations. They can also analyze all user behavior and discover new products or services that interest a specific user. **Netflix, YouTube, Tinder, and Amazon** are

all examples of recommender systems in use. The systems entice users with relevant suggestions based on the choices they make.

How do neural networks work?

The human brain is the inspiration behind neural network architecture. Human brain cells, called neurons, form a complex, highly interconnected network and send electrical signals to each other to help humans process information. Similarly, an artificial neural network is made of artificial neurons that work together to solve a problem. Artificial neurons are software modules, called nodes, and artificial neural networks are software programs or algorithms that, at their core, use computing systems to solve mathematical calculations.

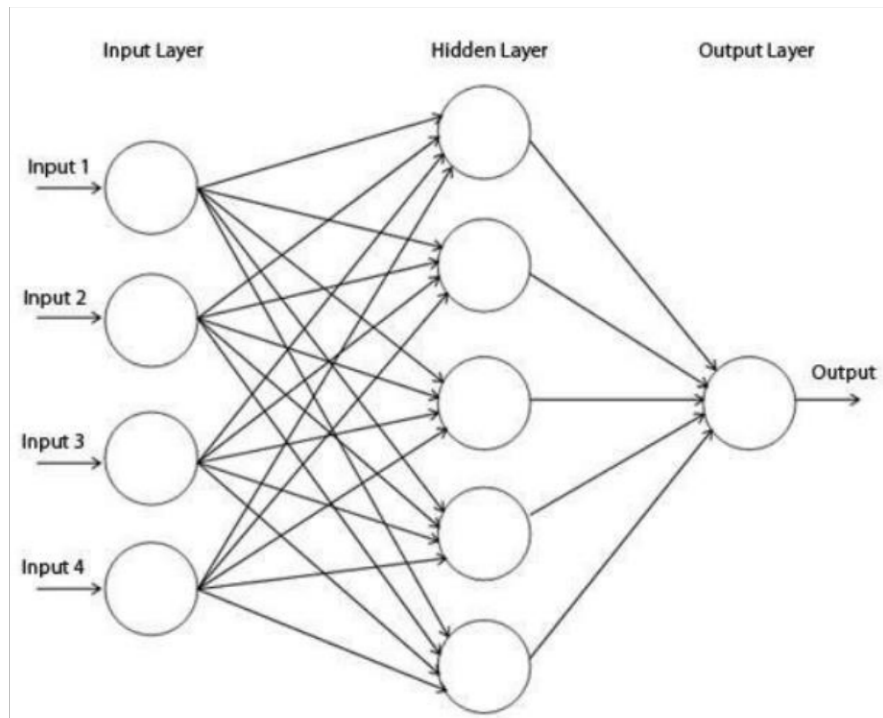
## Deep Learning

### The “Deep” in Deep Learning

In general terms, Deep learning is a system of successive layers to predict the output based on the provided input features. Each layer extracts some meaningful information out of the data and creates a meaningful representation of data. The deep in deep learning doesn't mean to have any kind of a deeper understanding; rather it stands for this idea of successive layers of representation. How many layers contribute to a model of the data is called the depth of the data.

In deep learning, these layered representations are learned via models called neural networks.

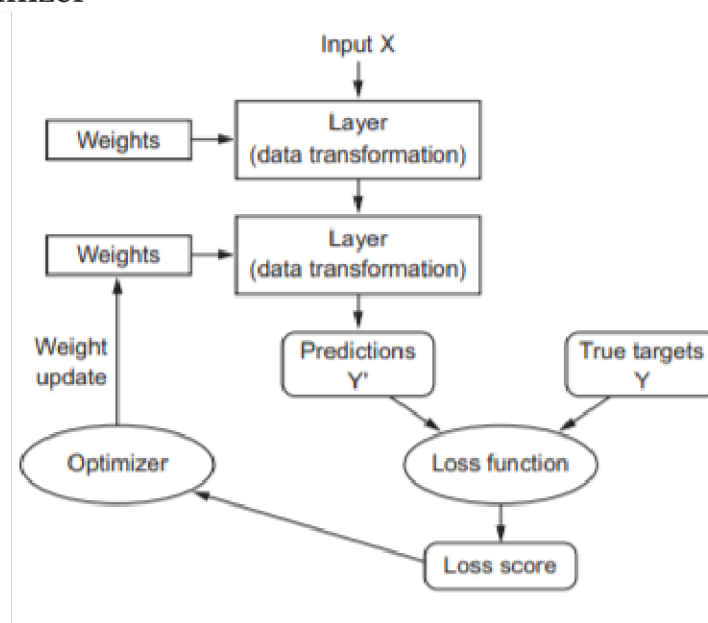
So neural network means building a network that has set of layers (as mentioned above) where each layer has certain number of nodes to hold data. Final goal is to understand data and give the desired output. The output of one layer becomes the input for the next layer.



## Anatomy of a neural network

Neural network mainly revolves around these four things:

1. Layers
2. Input data and corresponding targets
3. Loss function
4. Optimizer



Layers: Layer is a data-processing module that takes as input one or more tensors and that output one or more tensors {The word tensor comes from the Latin word tendere meaning “to stretch”. A tensor of order zero (zeroth-order tensor) is a scalar (simple number). A tensor of order one (first-order tensor) is a linear map that maps every vector into a scalar. A vector is a tensor of order one.} and that outputs one or more tensors.

For different tensor formats (Input data) we need different types of layers

2D tensor (sample, features) → Dense Layers

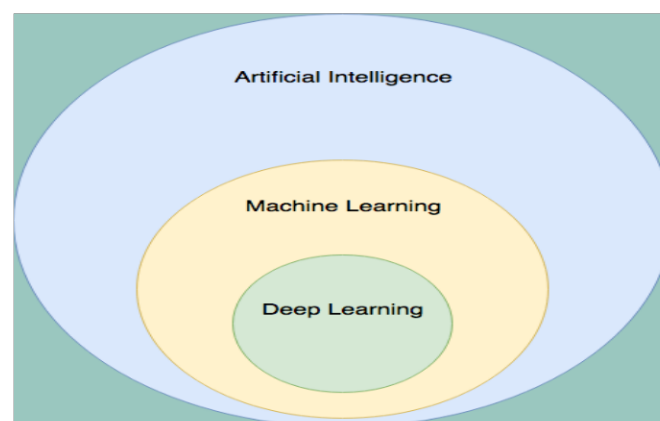
3D tensor (samples, timestamps, features) → LSTM layers

4D tensor (Image data) → 2D convolutional layers

Loss function and optimizers: Loss function represents a measure of success for the task at hand. This quantity would be minimized during training. Optimizers determines how the network will be updated based on loss function.

## Artificial Intelligence, Machine Learning, and Deep Learning

Deep learning is a subset of machine learning. Machine learning is a subset of artificial intelligence. Said another way — all deep learning algorithms are machine learning algorithms, but many machine learning algorithms do not use deep learning. As a Venn Diagram, it looks like this:



Deep learning refers specifically to a class of algorithm called a neural network, and technically only to “deep” neural networks (more on that in a second). This first neural network was invented in 1949, but back then they weren’t very useful. In fact, from the 1970’s to the 2010’s traditional forms of AI would consistently outperform neural network based models.

These non-learning types of AI include rule based algorithms (imagine an extremely complex series of if/else blocks); heuristic based AIs such as A\* search; constraint satisfaction algorithms like Arc Consistency; tree search algorithms such as minimax (used by the famous Deep Blue chess AI); and more.

There were two things preventing machine learning, and especially deep learning, from being successful. Lack of availability of large datasets and lack of availability of computational power. In 2018 we have exabytes of data, and anyone with an AWS account and a credit card has access to a distributed supercomputer. Because of the new availability of data and computing power, Machine learning — and especially deep learning — has taken the AI world by storm.

You should know that there are other categories of machine learning such as unsupervised learning and reinforcement learning but for the rest of this article, I will be talking about a subset of machine learning called supervised learning.

Supervised learning algorithms work by forcing the machine to repeatedly make predictions. Specifically, we ask it to make predictions about data that we (the humans) already know the correct answer for. This is called “labeled data” — the label is whatever we want the machine to predict.

Here’s an example: let’s say we wanted to build an algorithm to predict if someone will default on their mortgage. We would need a bunch of examples of people who did and did not default on their mortgages. We will take the relevant data about these people; feed them into the machine learning algorithm; ask it to make a prediction about each person; and after it guesses we tell the machine what the right answer actually was. Based on how right or wrong it was the machine learning algorithm *changes how it makes predictions*.

We repeat this process many many times, and through the miracle of

mathematics, our machine's predictions get better. The predictions get better relatively slowly though, which is why we need so much data to train these algorithms.

Machine learning algorithms such as linear regression, support vector machines, and decision trees all “learn” in different ways, but fundamentally they all apply this same process: make a prediction, receive a correction, and adjust the prediction mechanism based on the correction. At a high level, it's quite similar to how a human learns.

Recall that deep learning is a subset of machine learning which focuses on a specific category of machine learning algorithms called neural networks. Neural networks were originally inspired by the way human brains work — individual “neurons” receive “signals” from other neurons and in turn send “signals” to other “neurons”. Each neuron transforms the incoming “signals” in some way, and eventually an output signal is produced. If everything went well that signal represents a correct prediction!

This is a helpful mental model, but computers are not biological brains. They do not have neurons, or synapses, or any of the other biological mechanisms that make brains work. Because the biological model breaks down, researchers and scientists instead use graph theory to model neural networks — instead of describing neural networks as “artificial brains”, they describe them as complex graphs with powerful properties.

Viewed through the lens of graph theory a neural network is a series of layers of connected nodes; each node represents a “neuron” and each connection represents a “synapse”.

Different kinds of nets have different kinds of connections. The simplest form of deep learning is a deep neural network. A deep neural network is a graph with a series of fully connected layers. Every node in a particular layer has an edge to every node in the next layer; each of these edges is given a different weight. The whole series of layers is the “brain”. It turns out, if the weights on all these edges are set *just right* these graphs can do some incredible “thinking”.

