
WORLD
OF
ALGORITHMS
BY ABHIJNAN VEGI

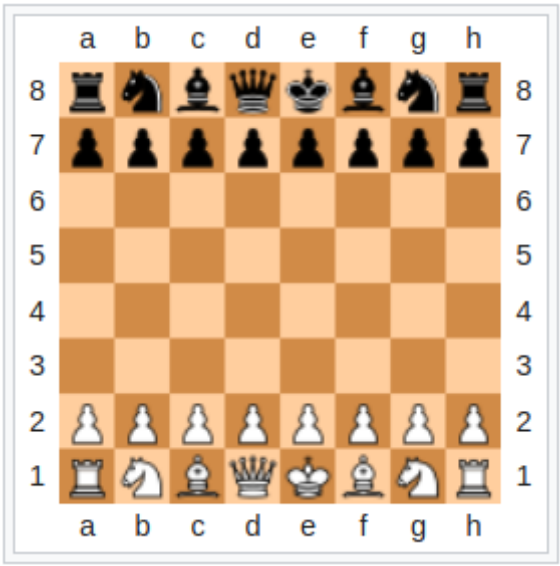
Preface

One of the important questions one asks before learning something is whether the new the knowledge he's going to acquire have any practical applications in day to day life. This books answers that question for anyone who wishes to learn about algorithms. In this book I explore the various applications of algorithms in day to day life and explain them in detail with good visualisations. Some of the topics have been linked to pages that cover the topic with more detail for anyone who is interested.

Table of Contents

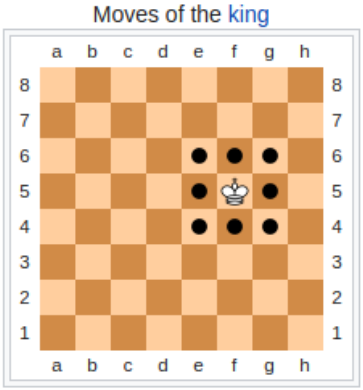
How does a computer play chess ?

Chess is an abstract strategy game. It is played on an 8×8 grid, with each player controlling sixteen pieces: one king, one queen, two rooks, two knights, two bishops and eight pawns. The object of the game is to checkmate the opponent's king, whereby the king is under immediate attack and there is no way for it to escape.

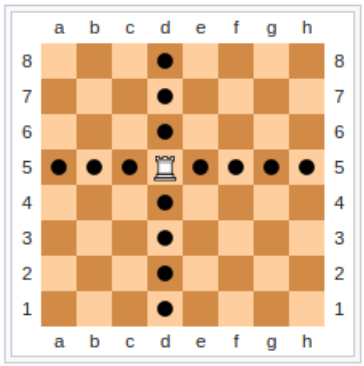


The moves of each of the pieces is shown below

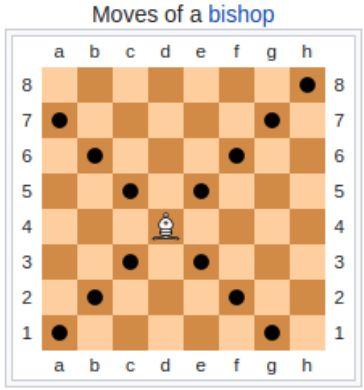
Moves of the king



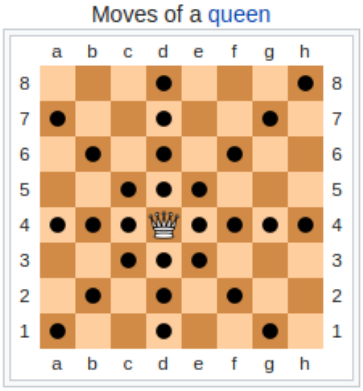
Moves of a rook



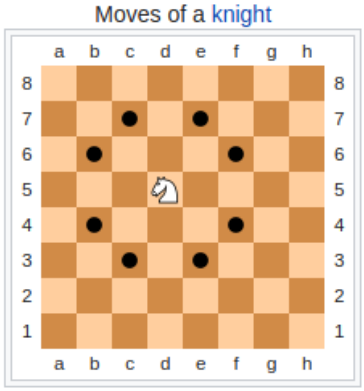
Moves of a bishop



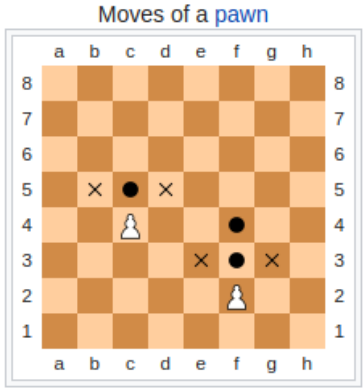
Moves of a queen



Moves of a knight



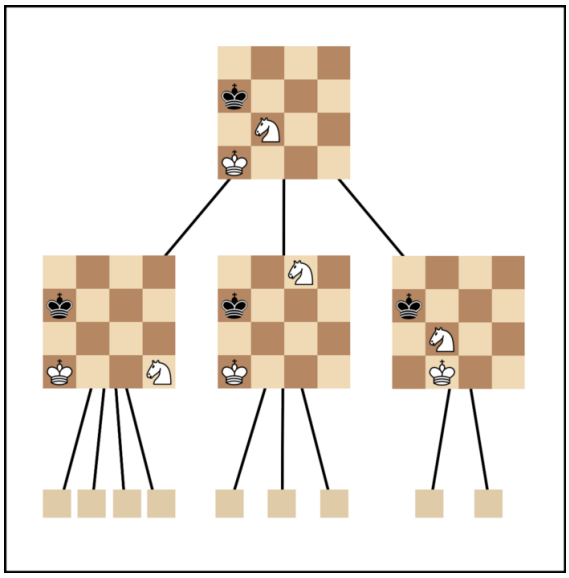
Moves of a pawn



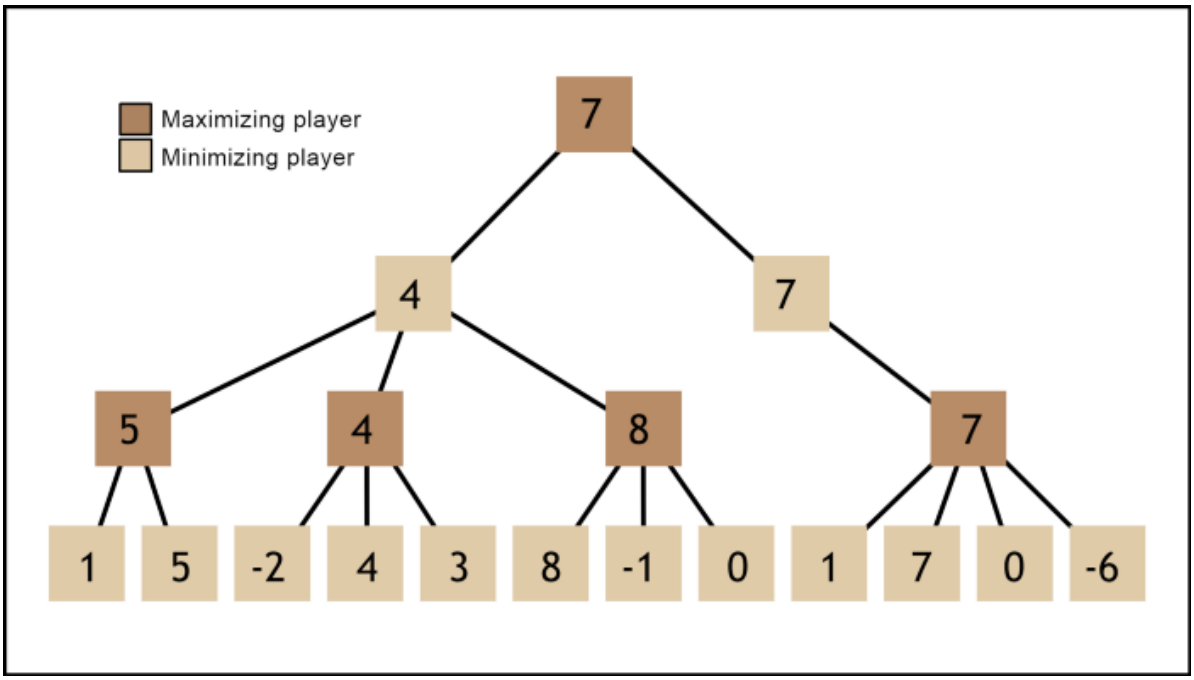
We will now explore an algorithm that's responsible for your computer beating you at chess.

Minimax

Lets start of by looking at the moves a computer can make if its starting the game. Each of the 8 pawns have 2 possible moves and the knights have 2 possible moves each giving a total of 20 moves. Similarly the opponent also has 20 possible moves. Thus the computer has to look at 20×20 scenarios in just two moves. All the possile configurations of the chessboard are approximately 10^{120}, which is greater than the number of atoms in the visible universe! We model these scenarios in the form of tree and the computer generates this tree to best of its capabilities.



Chess is zero-sum game, thus maximizing your chances is same as minimizing the opponent's chances of winning. We now decide upon an evaluation function, this varies from author to author and depends on the many complicated factors such as individual pieces, board position, control of the center, vulnerability of the king to check, vulnerability of the opponent's queen, etc. We will assume a simple evaluation function that is number of white pieces - number of black pieces. Thus in terms of search tree, the computer choses the children nodes with either the best or worst scores based on the colour played by it.



The above is an example of a minimax search tree. The leaf nodes are assigned scores based on some evaluation function and the higher nodes then decide their scores based on the min or max of their children nodes.

This pseudocode for algorithm is given below

```
def minimax(node, depth, maximizing player):
    # depth is the maximum depth to which algorithm is applied
    # maximizing player is a boolean of whether the player is maximizing or
not
    if depth = 0 or node is terminal node:
        return evaluation(node)
    if maximizing player:
        value = -∞
        for child of node:
            value = max(value, minimax(child, depth - 1, False))
        return value
       else:
        value = +∞
        for child of node:
            value = min(value, minimax(child, depth-1, True))
        return value
```

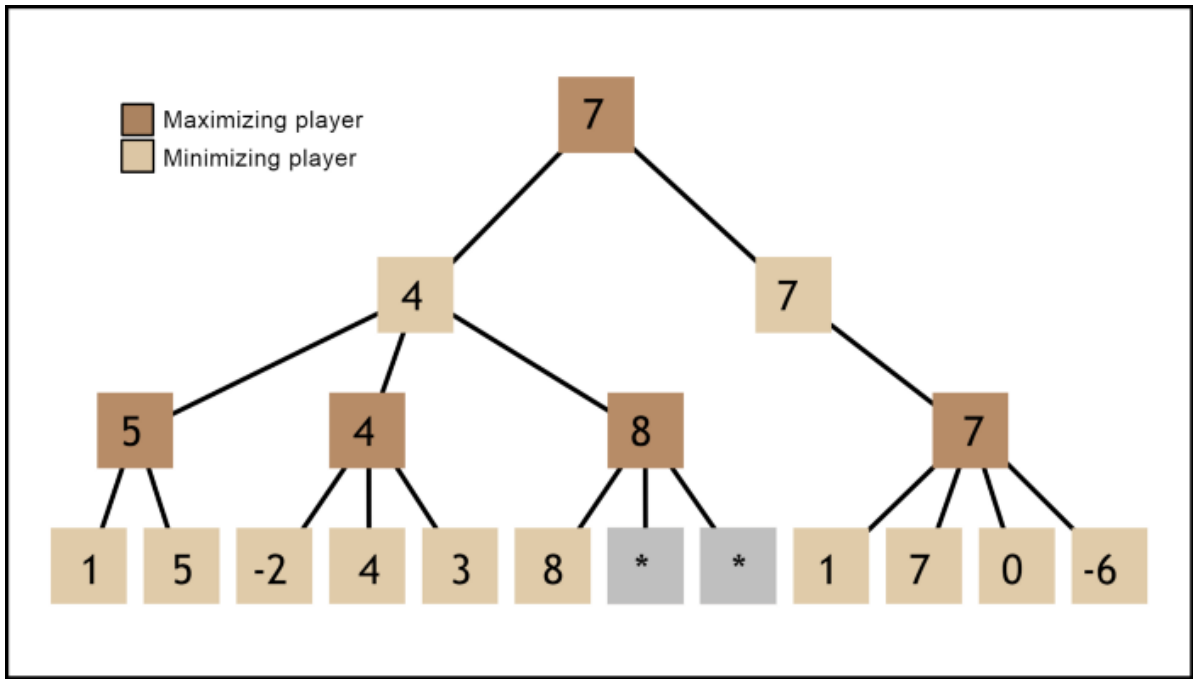
The number of scenarios to be evaluated is still high for any computer to check in a reasonable amount of time. We use technique called Alpha Beta pruning to reduce this number.

Alpha-Beta pruning

In chess, there are moves that allow the other player to get a clear upper hand in the next few turns. A standard minimax evaluates these moves just as much as the others, thus slowing it down.

Alpha-Beta pruning speeds up minimax by skipping the irrelevant nodes of the search tree. We accomplish this by adding an alpha and beta value, which represent the worst outcome for each player from that node.

Since the maximizing player knows that the minimizing player will pick a response that minimizes the evaluation, they also know that they can avoid thinking about moves that allow the minimizing player to make things worse than they already are. These moves are pruned from the search tree and skipped.

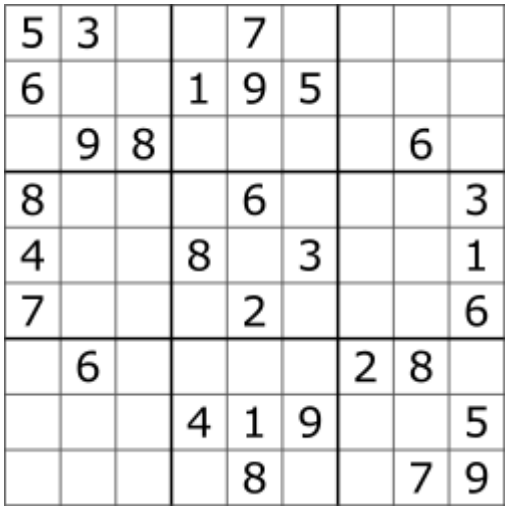


Alpha-beta search skips nodes once it knows for certain that they won't be played. In this example, the minimizing player comes across a position with a score of 8. The maximizing player will either play that position or one with a higher score. Since the minimizing player already found a better move to play (the one with a score of 4), there's no need to further explore this node.

- The algorithm can be further optimised by using move ordering, transposition tables and quiescence search. You can read about these here.
- Another approach to the same problem is AlphaZero, which utilises neural networks along with monte-carlo tree search.
- This link includes a good board evaluation function along with other ways to improve our algorithm.

How does a computer solve Sudoku ?

Sudoku is a logic based, combinatorial number placement puzzle. Given a partially completed 9×9 grid of numbers from 1 to 9 we fill all the remaining boxes with digits from 1 to 9 such that every row, column, diagonal and each of the nine 3×3 grids contain all the digits from 1 to 9.

Sudoku puzzle	Solution
	

A naive approach to solving this problem is to generate all possible configurations by placing numbers from 1 to 9 in the empty cells. This would result in a total of 9^e configurations, where e is the number of empty cells in the puzzle, whose validity can checked in constant time. This gives us the time complexity of this algorithm as $O(9^e)$.

We will now explore an algorithm that can solve the same puzzle faster than our naive approach.

Backtracking

Backtracking is an algorithm that incrementally builds candidates to the solutions and abandons a candidate as soon as it determines that the candidate cannot possibly lead to a valid solution. When the algorithm abandons a candidate, it typically returns to a previous stage in the problem solving process. *Backtracking* is often much faster than the brute force solutions for a problem since it can eliminate multiple candidates with a single test.

The algorithm to solve Sudoku with backtracking is as follows

1. Assign a number to an empty cell such that it can be part of the solution, that is it doesn't violate any constraints of the puzzle.
2. Recursively check whether the above assignment leads to valid solution or not. If the above assignment fails we try the next number. If none of the numbers 1 to 9 lead to solution, we can conclude that puzzle has no solution.

The above algorithm is applied as follows for the puzzle above

1. Assign a number to any cell without violating constraints

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

2. Apply the same algorithm to see find solution for the puzzle above. The puzzle doesn't have a valid solution when there is a cell in which inserting any digit violates the constraints, In which case change the number in step 1 repeat the steps.

A visualization of this code can be found here.

The generalized Sudoku problem

The generalized Sudoku problem is the generalized version of the problem discussed above where the total number of symbols are N instead of 9 in the above version. The square is also scaled to the size $N \times N$ and to retain the blocks we choose N such that it is a perfect square. We find that this problem is NP complete and is equivalent to the Hamiltonian cycle problem.

The Hamiltonian cycle problem is defined as, For a simple graph G containing vertex set V and the edge set E determine whether there exists a path such that each vertex is visited exactly once. This problem is shown to NP complete thus by proving the equivalence of the generalized Sudoku problem and Hamiltonian cycle problem we prove that the generalized cycle problem is NP complete.

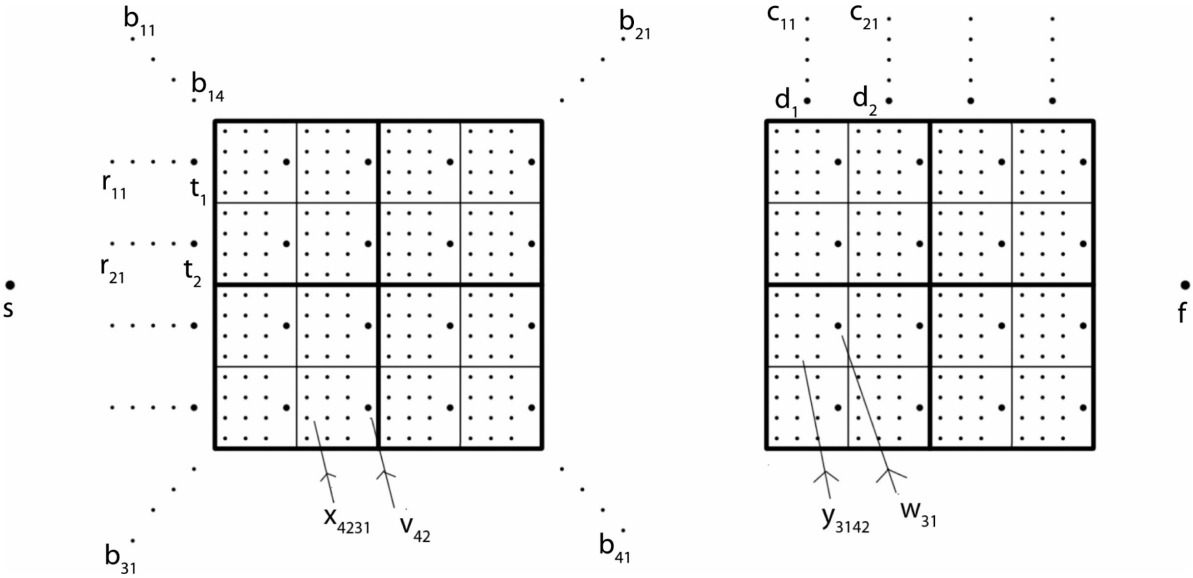
Conversion of generalized Sudoku to Hamiltonian cycle

Idea : We encode every possible variable choice in Sudoku to a subgraph. Traversing the subgraph in a certain way will correspond to particular choice for that variable. Then we link the subgraphs such that they can only be consecutively traversed in none of the constraints are violated.

In the final instance of Hamiltonian cycle problem that is produced, the vertex set V will comprise of the following, where a, i, j and k all take values from 1 to N:

- A single starting vertex s and finishing vertex f.
- Block vertices: N^2 vertices b_{ak}, corresponding to number k in block a.
- Row vertices: N^2 vertices r_{ik}, corresponding to number k in row i.
- End Row vertices: N vertices t_i corresponding to row i.
- Column vertices: N^2 vertices c_{jk} corresponding to number k in column j.
- End Column vertices: N vertices d_j corresponding to column j.
- Puzzle vertices: $3N^3$ vertices x_{ijkl} corresponding to number k in position (i, j), for $l = 1, 2, 3$.
- End Puzzle vertices: N^2 vertices v_{ij} corresponding to position (i, j).
- Duplicate Puzzle vertices: $3N^3$ vertices y_{ijkl} corresponding to number k in position (i, j), for $l = 1, 2, 3$.
- End Duplicate Puzzle vertices: N_2 vertices w_{ij} corresponding to position (i, j).

Let us see this conversion for $N = 4$.



Only a few vertices have been labeled to aid in understanding where the vertices lie. The individual vertices on the left and right s and f respectively. The Sudoku puzzle is duplicated as it is only convenient to satisfy two constraints at a time. Hence the vertices in the first puzzle will be linked together in such a way as to ensure the block constraints, and the row constraints, are simultaneously satisfied. Then, the vertices in the second puzzle will be linked together in such a way to ensure the solution in the second puzzle is a duplicate of the first, and the column constraints are also satisfied.

The graph will be linked together such that any valid situation to the Sudoku puzzle will correspond to a Hamiltonian cycle in the following manner:

1. The starting vertex s is visited first.
2. For each a and k, suppose number k is placed in position (i, j) in block a. Then, vertex b_{ak} is visited, followed by all x_{ijml} for $m \neq k$, followed by all y_{ijml} for $m \neq k$. This process will ensure that each of the N blocks contain each number from 1 to N.
3. For each i and k, suppose number k is placed in position (i, j) in row i. Then, vertex r_{ik} is visited, followed by $x_{ijk3}, x_{ijk2}, x_{ijk1}$ and then v_{ij}. If $k = N$ then this is followed by t_i. This process will ensure that each of the N rows contain each number from 1 to N.
4. For each j and k, suppose number k is placed in position (i, j) in column j. Then, vertex c_{jk} is visited, followed by $y_{ijk3}, y_{ijk2}, y_{ijk1}$ and then w_{ij}. If $k = N$ then this is followed by d_j. This process will ensure that each of the N columns contain each number from 1 to N.
5. The finishing vertex f is visited last and the Hamiltonian cycle returns to s.

The idea was to create two identical copies of the puzzle, in step 2 we place numbers in the puzzle to ensure numbers are placed identically in both the copies. Placing a number k into position (i, j), contained in block a, is achieved by first visiting b_{ak}, and then proceeding to visit every puzzle vertex x_{ijml} except for when $m = k$, effectively leaving the assigned number open, or unvisited.

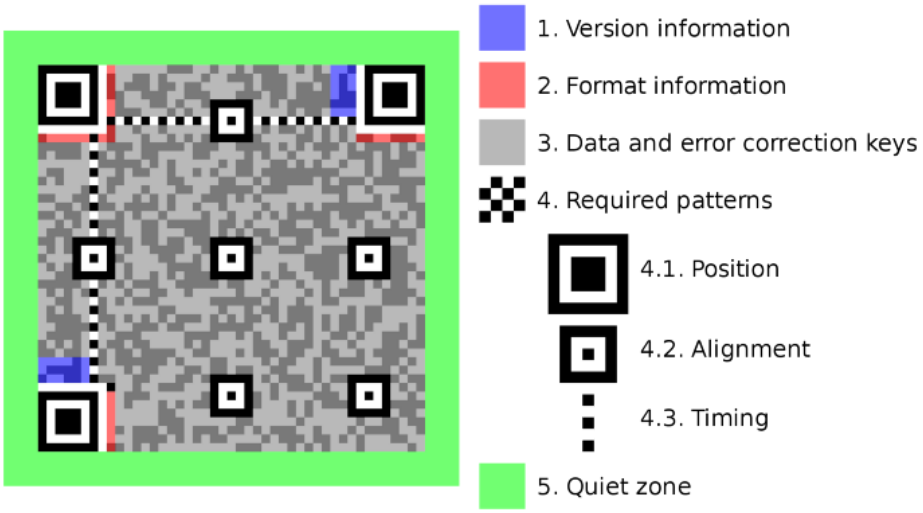
The complete proof of this equivalence can be found here

How do QR Codes work ?

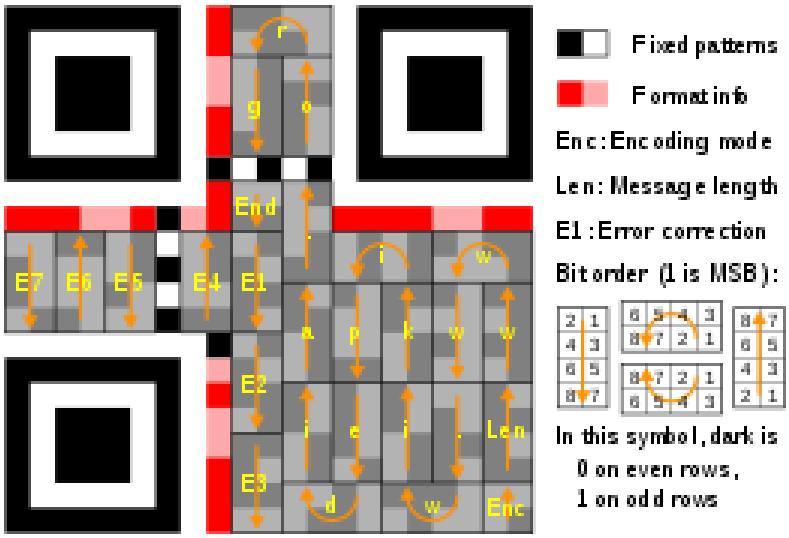
QR code is short for quick response code is type of two-dimensional barcode. The QR code became popular due to its fast readability and greater storage capacity compared to standard barcodes. A QR code consists of black squares arranged in a square grid on a white background, which can be read by an imaging device. Below is the QR code for a Wikipedia page.

We will now explore how a QR code is generated and also how its interpreted.

Encoding of a QR Code



The above image consists of the breakdown of a QR Code. Position and alignment markings help the scanner identify the QR code. Timing pattern tells the scanner the size of the data matrix. Version specifies the version since multiple of them are in use. Format provides information regarding error tolerance and data mask pattern.



The above image shows how bytes are encoded in qr code. It starts from bottom right where the encoding is mentioned moving up specifying the length of data encoded and the data following it in a zig zag pattern. The other blocks contain the error correction codes which will be discussed in further section.

Error correction codes

QR Codes use **Reed-Solomon error correction**, so that they can withstand a higher margin of error during the decoding process. They were introduced by Irving S. Reed and Gustave Solomon.

Encoding

Reed-Solomon codes operate on a block of data treated as a set of finite field elements called symbols (In this case its the finite field $\mathbb{F}_{256}$). If the message if size k, the $t = n - k$ symbols are used as check symbols. reed Solomon code can detect but not correct any combination of up to t erroneous symbols, or locate and correct up to $\lfloor t \rfloor$ erroneous symbols.

The message is interpreted as the coefficients of a polynomial $p(x)$. The encoder computes a related polynomial $s(x)$ of degree $n - 1$ where $n \leq q - 1$ and sends the polynomial $s(x)$. The polynomial $s(x)$ is generated by multiplying the message polynomial of degree $k - 1$ with a generator polynomial of degree $n - k$ that is known to both sender and the receiver. The generator polynomial is defined as the the polynomial whose roots are sequential powers of the field primitive α (In this case $\alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + 1 = 0$). The Reed-Solomon code uses one of 37 different polynomials over $\mathbb{F}_{256}$. The rules for selecting the degree are specific to the QR standard.

Decoding

The encoded message is viewed as the coefficients of a polynomial $s(x)$. As a result of the Reed-Solomon encoding procedure, $s(x)$ is divisible by the generator polynomial $g(x)$. Since $s(x)$ is a multiple of $g(x)$, it inherits all its roots.

Let the transmitted polynomial b corrupted in transit by an error polynomial $e(x)$. Thus giving the received polynomial $r(x) = s(x) + e(x)$. The coefficient e_i of x in $e(x)$ is 0 if no error in that power of x. The goal of the decoder is to find the number of errors, positions of errors and error values.

This involves multiple steps

1. **Syndrome decoding**
   The decoder starts by evaluating the polynomial as received at points $\alpha^1 \dots \alpha^{n-1}$. We call the results of that evaluation the syndromes, S_j. They are defined as

$$S_j = r(\alpha^j) = e(\alpha^j),\ j = 1, 2, \dots, n-k$$
$$= \sum_{k=1}^{v} e_{i_k} (\alpha^j)^{i_k}$$

   Thus if all syndromes are zero, the message wasn't corrupted and the decoder exits.

2. **Error locators and error values**
   Let us define error locators X_k and error values Y_k as

$$X_k = \alpha^{i_k}$$
$$Y_k = e_{i_k}$$

   The syndromes can be written as

$$S_j = \sum_{k=1}^{v} Y_k X_k^j$$

   The syndromes give a system of $n - k \geq 2\nu$ equations in 2ν unknowns, but that system of equations is nonlinear in the X_k and does not have an obvious solution. However, if the X_k were known, then the syndrome equations provide a linear system of equations that can easily be solved for the Y_k error values.

   The rest of the algorithm serves to locate the errors, and will require syndrome values up to $2v$, instead of just the v used thus far. This is why twice as many error correcting symbols need to be added as can be corrected without knowing their locations.

3. **Error locator polynomial**
   Define the error locator polynomial $A(x)$ as

$$A(X) = \prod_{k=1}^{v} (1 - xX_k)$$

   The zeros of $A(x)$ are the reciprocals X_k^{-1}. That is $A(X_k^{-1}) = 0$. Let j be any integer such that $1 \leq j \leq v$. Multiply both sides by $Y_k X_k^{j+v}$. The equation then reduces to

$$S_j A_v + S_{j+1} A_{v-1} + \dots + S_{j+v-1} A_1 = -S_{j+v}$$

   Since j can be any integer between 1 and v, we have v linear equations, which can be solved to give the coefficients of the error locator polynomial, the roots of which give us the error locations. The above calculation above assumes that the decoder knows the number number of errors v. The decoder does not determine this value directly instead searches for it by trying successive values.

4. **Roots of error locator polynomial**
   We have now fully constructed the error locator polynomial from the coefficients A_i found in the previous step. The roots are calculated using an exhaustive search. The error locators are the reciprocals of these roots. An example of the exhaustive search is Chien search. You can read more about it here

5. **Calculation of error values and locations**
   Once the values of X_k are known, the syndrome equations obtained above can be solved to obtain the values of Y_k. i_k can be calculated by $\log_\alpha X_k$. This is generally done using a precomputed lookup table.

6. **Fix errors**
   Finally, $e(x)$ is generated from i_k and e_{i_k} and then is subtracted from $r(x)$ to get the originally sent message $s(x)$, with errors corrected

With that we have successfully decoded the QR code and fixed any errors that might have occurred during its scanning

---

- An example of decoding Reed-Solomon code can be found here
- There are other algorithms for decoding Reed-Solomon code which can be found here
- A code oriented working of the QR code can be found here

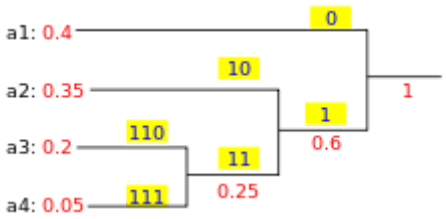
How do Zip files work ?

ZIP is an archive file format that supports lossless data compression. A zip file may contain one or more directories that may have been compressed. The ZIP format permits a number of compression algorithms.

We will look into the **DEFLATE** algorithm, the most common algorithm for compressing a zip file. It is in fact a combination of two algorithms **Huffman encoding** and **LZ77**.

Huffman encoding

We create a binary tree with each of the leaf nodes containing a symbol and the weight of the symbol. The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.



This way we get the shortest possible encoding for given input. The data used to generate the tree must also be encoded for the decoder to decode the transmission.

LZ77

LZ77 is sometimes also known as *sliding window compression*. It achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a length-distance pair, which is equivalent to the statement “each of the next length characters is equal to the character exactly distance character behind it in the uncompressed stream.”. The representation used in deflate format limits distances to 32K bytes and lengths to 258 bytes.

The pseudocode for this algorithm is given by

```
while input is not empty do
    prefix := longest prefix of input that begins in window
    if prefix exists then
        d := distance to start of prefix
        l := length of prefix
        c := char following prefix in input
    else
        d := 0
        l := 0
        c := first char of input
    end if
    output (d, l, c)
    discard l + 1 chars from front of window
    s := pop l + 1 chars from front of input
```

```
    append s to back of window
repeat
```

This is the most computationally expensive part of the DEFLATE algorithm. Each block of data is compressed using LZ77 and then compressed using Huffman encoding. Each block of the output stream is then preceded by a 3-bit header providing necessary information for decompression.

- First bit: Last-block-in-stream marker:
  - `1`: This is the last block in the stream.
  - `0`: There are more blocks to process after this one.
- Second and third bits: Encoding method used for this block type:
  - `00`: A stored section, between 0 and 65,535 bytes in length
  - `01`: A *static Huffman* compressed block, using a pre-agreed Huffman tree defined in the RFC
  - `10`: A compressed block complete with the Huffman table supplied
  - `11`: Reserved—don't use.

Most of the compressible data ends up being encoded using method `01`, which produces an optimized Huffman tree. Instructions to generate the necessary Huffman tree immediately follow the block header. The static Huffman option is used for short messages, where the fixed saving gained by omitting the tree outweighs the percentage compression loss due to using a non-optimal code.

Decoding

Knowing the encoding and the output format of the DEFLATE algorithm, decoding is pretty straight forward. The pseudocode for decoding is given by

```
do
   read block header from input stream.
   if stored with no compression
      skip any remaining bits in current partially
         processed byte
      read LEN and NLEN (see next section)
      copy LEN bytes of data to output
   otherwise
      if compressed with dynamic Huffman codes
         read representation of code trees (see
            subsection below)
      loop (until end of block code recognized)
         decode literal/length value from input stream
         if value < 256
            copy value (literal byte) to output stream
         otherwise
            if value = end of block (256)
               break from loop
            otherwise (value = 257..285)
               decode distance from input stream
               move backwards distance bytes in the output
               stream, and copy length bytes from this
               position to the output stream.
      end loop
while not last block
```

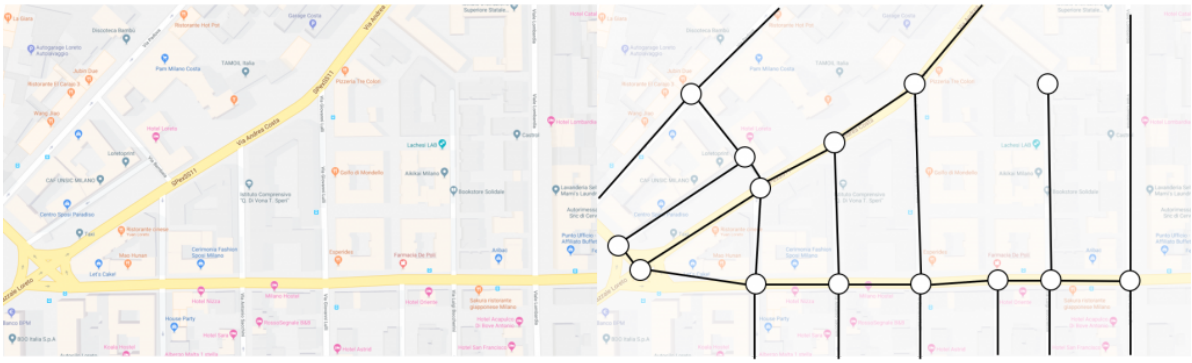
How does Google Maps work ?

A **Web mapping** or an online mapping is the process of using the maps delivered by geographic information system on the Internet. With the rise of web mapping, software which provide directions from one place to another have come into existence, such as Google Maps.

The algorithm behind the fastest route comes is Dijkstra's algorithm. We will now explore this algorithm.

Dijkstra's algorithm

We first model our map as a weighted graph, where each turn is node and the roads between them as edges. The weight of the edge depends on various conditions such as length, traffic and other conditions.
This data is updated in real time by Google using the data collected from other users using the software.



Lets assume weight of an edge is directly proportional to the time taken taken to move along that edge/road. In terms of this graph we will now restate our original problem as

> Given a directed graph with unbounded non-negative weights, find the path from source to destination such that the sum of weights of all edges in the path is minimized.

The algorithm tracks a cost associated with each node, that is the shortest time taken to reach a particular node. The algorithm is as follows

1. Initialize the cost of all nodes to infinity, except the starting node, which is set to 0.
2. Initialize a list of nodes to be visited.
3. For the current node, consider all of its unvisited neighbors and calculate their cost from the current node. That is cost of neighbor node is equal to the sum of cost of current node and weight of the edge connecting them. If the value obtained is lower than the cost of neighbor node, it is updated, otherwise its left unchanged.
4. Once all the neighbors of the current node are visited, it is removed from the list of nodes to be visited.
5. If the destination node has been marked visited or if the weight of all nodes in the list of nodes to be visited is infinity, there is no path from source to destination, then the algorithm has completed.
6. Otherwise, select the node from the list of nodes to be visited with the least cost, set it as the current node and repeat from step 3.

The pseudocode for this algorithm is given by

```
function Dijkstra(Graph, source):
      create vertex set Q

       for each vertex v in Graph:
           dist[v] ← INFINITY
           prev[v] ← UNDEFINED
           add v to Q
       dist[source] ← 0

       while Q is not empty:
           u ← vertex in Q with min dist[u]

           remove u from Q

           for each neighbor v of u still in Q:
               alt ← dist[u] + length(u, v)
               if alt < dist[v]:
                   dist[v] ← alt
                   prev[v] ← u

       return dist[], prev[]
```

Another faster algorithm for the same purpose is the **A* algorithm**

A* Algorithm

A* Algorithm is an informed search algorithm, or a best-first search. Informed search algorithm use additional information that directs the search towards the goal. This additional information is usually just an approximation.

Starting from source, the algorithm aims to find a path to the given goal node having the smallest cost. It does this by maintaining a tree of paths originating from the start node and extending those paths one edge at a time until its termination criterion is satisfied. In each iteration the algorithm determines which of its paths to extend. The algorithm selects the path that minimizes $f(n) = g(n) + h(n)$, where n is the next node, $g(n)$ is cost of the path and $h(n)$ is a heuristic function.

A few examples of heuristic functions are:

1. Manhattan Distance
   It is the sum of absolute values of differences in the destination's x and y coordinates and current cell's x and y coordinates. $h(x) = abs(x_{dest} - x_n) + abx(y_{dest} - y_n)$
2. Euclidean Distance
   It is the distance between the current node and the destination in the 2D plane.
   $h(x) = \sqrt{(x_{dest} - x_n)^2 + (y_{dest} - y_n)^2}$

If the heuristic $h(n)$ satisfies the additional condition that $h(x) \leq d(x, y) + h(y)$ for every edge (x, y) of graph, then h is called monotone or consistent. With a consistent heuristic, A* is guaranteed to find an optimal path without processing any node more than once and A* is equivalent to running Dijkstra's algorithm with $d'(x, y) = d(x, y) + h(y) - h(x)$. The pseudocode for this algorithm is given by

```
// Initialize both open and closed list
let the openList equal empty list of nodes
let the closedList equal empty list of nodes

```

```
// Add the start node
put the startNode on the openList (leave it's f at zero)

// Loop until you find the end
while the openList is not empty
    // Get the current node
    let the currentNode equal the node with the least f value

    remove the currentNode from the openList
    add the currentNode to the closedList
    // Found the goal
    if currentNode is the goal
        Congratz! You've found the end! Backtrack to get path

    // Generate children
    let the children of the currentNode equal the adjacent nodes

    for each child in the children
        // Child is on the closedList
        if child is in the closedList
            continue to beginning of for loop
        // Create the f, g, and h values
        child.g = currentNode.g + distance between child and current
        child.h = distance from child to end
        child.f = child.g + child.h
        // Child is already in openList
        if child.position is in the openList's nodes positions
            if the child.g is higher than the openList node's g
                continue to beginning of for loop
        // Add the child to the openList
        add the child to the openList
```

You can compare both the algorithms by looking at the visualizations links given below

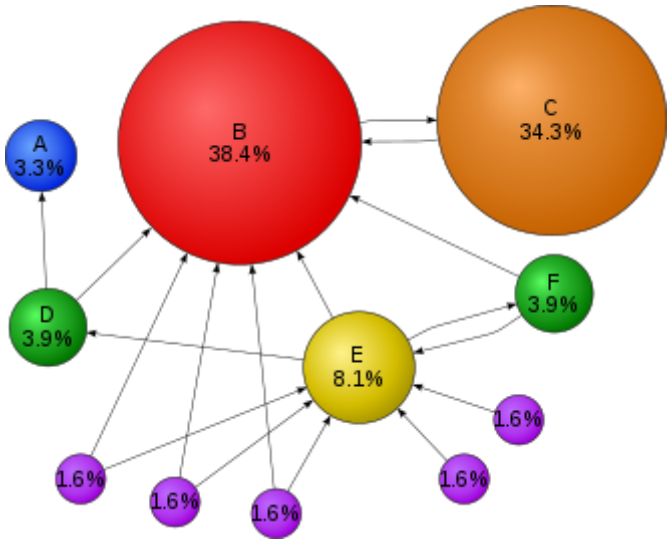
- Dijkstra's Algorithm
- A* Algorithm

How does a Search Engine work ?

A **Search engine** is a software system that is designed to search the World wide web in a systematic way for a particular query. Search engines also maintain real-time information by running an algorithm on a web crawler. A crawler is program that browses the World Wide web for web indexing.

We will look at the algorithm behind Google, the most popular search engine.

PageRank



Google's keyword search function is similar to other search engines. Automated programs called **spiders** or **crawlers** travel the Web, moving from link to link and building up an index page that includes certain keywords. Google references this index when a user enters a search query. The search engine lists the pages that contain the same keywords that were in the user's search terms. Google's spiders may also have some more advanced functions, such as being able to determine the difference between Web pages with actual content and redirect sites -- pages that exist only to redirect traffic to a different Web page.

PageRank is the algorithm used by Google search to rank web pages in their search engine. It isn't the only algorithm used by Google, but it was the first algorithm used by it and is the best known one. PageRank is a link analysis algorithm where it assigns a numerical weight to each element of a hyperlinked set of documents on the World Wide Web. It results from a mathematical algorithm based on the graph created with all the web pages as nodes and the hyperlinks as edges.

PageRank represents the probability of a surfer to reach web page. Thus a PageRank of 0.5 implies that a person has a 0.5 probability of reaching that web page.

Initially PageRank is initialized to the same value for all pages, such that the sum of PageRanks is equal to 1. In each iteration of the algorithm, the web page gains PageRank from the web pages linking to it. Let r_i denote the page rank of ith web page. The PageRank of the jth web page in the next iteration is given by

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

where, d_i is the out-degree of the ith node.

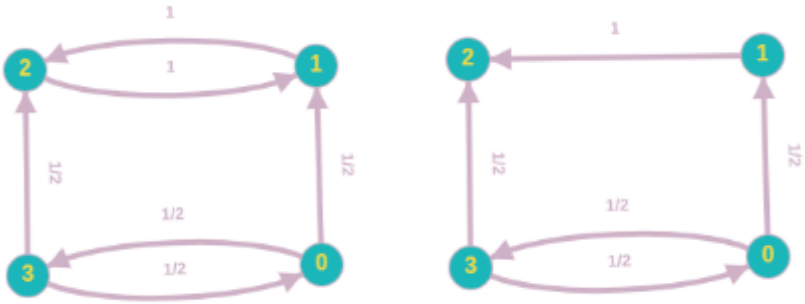
Damping factor

A person who is randomly clicking on links will eventually stop clicking, to account for this, a damping factor d is introduced. It is generally assumed that the damping factor will be set around 0.85. The formula for the PageRank of the jth web page is given by

$$r_j = \frac{1-d}{N} + d(\sum_{i \to j} \frac{r_i}{d_i})$$

Google recalculates the PageRank scores each time it crawls the Web and rebuilds its index. As Google increases the number of documents in its collection, the initial approximation of PageRank decreases for all documents.

If a page has no links to other pages, it becomes a sink and therefore terminates the random surfing process. If the random surfer arrives at a sink page, it picks another web page at random and continues surfing again. The same is done in case the surfer is stuck between two nodes.



In the first figure, if the surfer reaches node 1 or 2, he can only move between those two. This is known as a spider trap. In the second figure if the surfer reaches 2, it has no where to go.

Computation of PageRank

The PageRank values are the entries of the dominant right eigenvector of the modified adjacency matrix rescaled so that each column adds up to one. Let R denote the eigenvector the R is given by

$$R = \begin{bmatrix} r_1 \\ r_2 \\ . \\ . \\ r_n \end{bmatrix}$$

We do this computation iteratively as follows

1. Initialize $r_i = \frac{1}{N}$

   where, N is the total number of pages.

2. In each iteration compute

$$R_{t+1} = dMR_t + \frac{1-d}{N}1$$

   where R_t is the value of the eigenvector in the t^{th} iteration, 1 is the column vector of length N consisting of only ones.
   The matrix M is defined as

$$M_{ij} = \begin{cases} \frac{1}{L(p_j)}, & \text{if j links to i} \\ 0, & \text{otherwise} \end{cases}$$

3. The probability calculation is concluded if $|R_{t+1} - R_t| < \epsilon$ for some small ϵ, that is when convergence is assumed.

How does Find work ?

One of the most commonly used shortcut is Ctrl + F, which opens a find menu in most of the application to let you search for particular text. This algorithms used for such an operation are called string search algorithms. We will now explore the Boyer-Moore string-search algorithm.

Boyer-Moore string-search algorithm

The Boyer-Moore string-search algoritm is an efficient string-search algorithm and has been the standard benchmark for the practical string search literature.

We will start by defining a few terms

- **T** denotes the input text to be searched. Its length is **n**
- ***P*** denotes the string to be searched for, called the *pattern*. Its length is **m**.
- **S[i]** denotes the character at index *i* of string *S*, counting from 1.
- **S[i..j]** denotes the sub of string *S* starting at index *i* and ending at *j*, inclusive.
- An alignment of P to T is an index *k* in T such that the last character of P is aligned with index k of T.
- A match or occurrence of P occurs at an alignment k if P is equivalent to T[(k-m+1)..k].

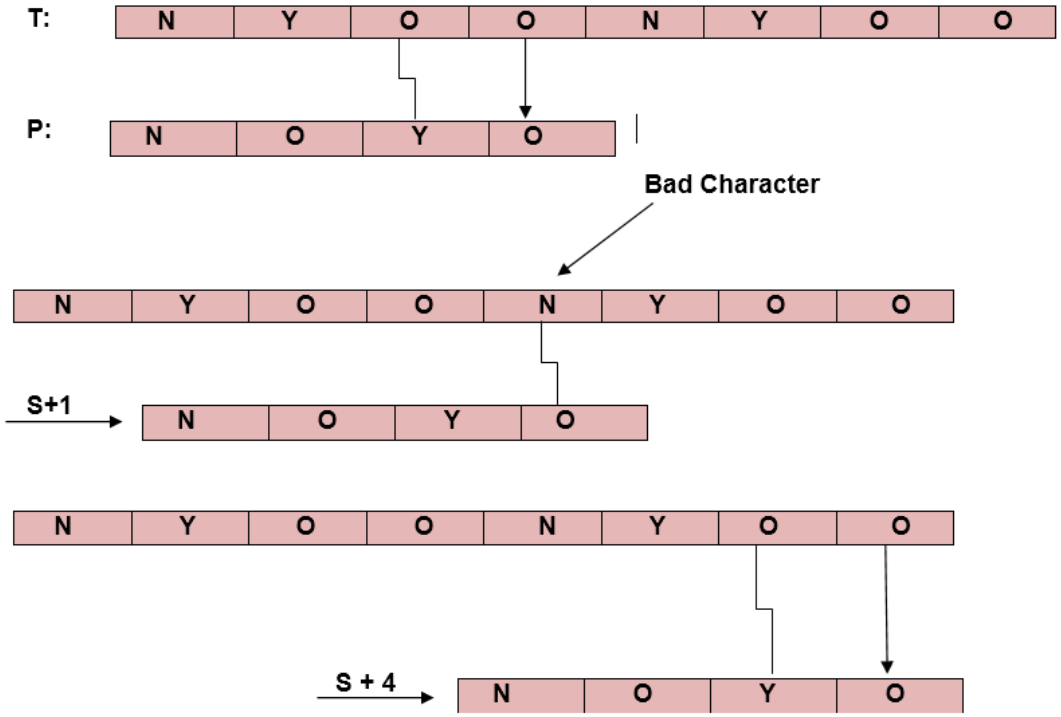
The Boyer-Moore algorithm searches for occurrences of P in T by performing explicit character comparisions at different alignments. The algorithm uses the imformation gained by preprocession P to skip as many alignments as possible.

The usual way for string search is examine each character for the first character of the patter and match the subsequent characters. Thus every character in the input text T needs to be checked. The idea in the Byer-Moore string search algorithm is that if the end of the pattern is compared to the text, then jumps along the text can be made rather than checking every character of the text.

The pattern P is aligned with the start of text string and then compares the character of a pattern from right to left, beginning with the righmost character. If a character is compared that is not within the pattern, no match can be found by analyzing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, the algorithm uses two preprocession strategies simultaneously. Whenever a mismatch occurs the algorithm calculates a variation using both approaches and selects the more significant shift.

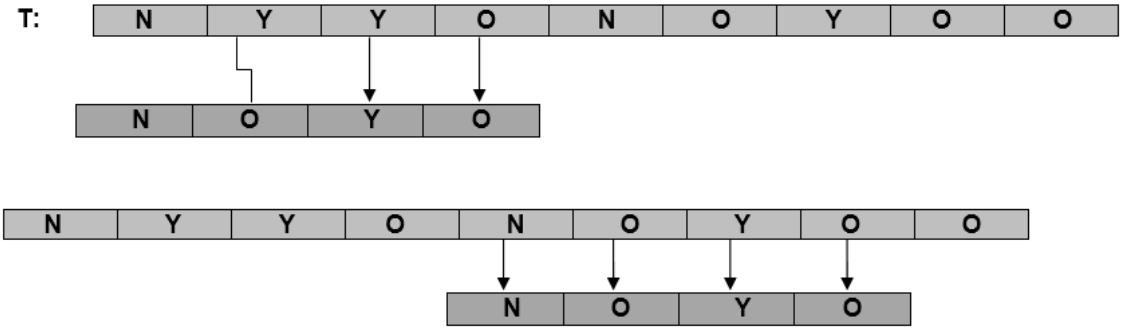
1. **Bad characyer heuristics**
   The hueristics has two implications
   1. Suppose there is a character ina text in which does not occue in patetrn at all. When a mismatch happens at this character, called a bad character, the whole pattern can be changed, begin matchin from substring next to this bad character.
   2. If the bad character is a part of the pattern, align the pattern with the bad character in text.



In some cases, Bad character Heuristics can produce negative shifts. Thus we need extra information to produce a shift on encountering a bad character.

2. **Good suffix heuristics**

   A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.



The pseudocode is given by

```
COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑ )
    for each character a ∈ ∑
        do λ [a] = 0
    for j ← 1 to m
        do λ [P [j]] ← j
    Return λ

COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
    Π ← COMPUTE-PREFIX-FUNCTION (P)
    P'← reverse (P)
    Π'← COMPUTE-PREFIX-FUNCTION (P')
    for j ← 0 to m
        do ɣ [j] ← m - Π [m]
    for l ← 1 to m
        do j ← m - Π' [L]
    If ɣ [j] > l - Π' [L]
        then ɣ [j] ← 1 - Π'[L]
    Return ɣ
```

```
19
20  BOYER-MOORE-MATCHER (T, P, ∑)
21      n ←length [T]
22      m ←length [P]
23      λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑ )
24      ɣ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
25      s ←0
26      While s ≤ n - m
27          do j ← m
28      While j > 0 and P [j] = T [s + j]
29          do j ←j-1
30      If j = 0
31          then print "Pattern occurs at shift" s
32          s ← s + ɣ[0]
33      else s ← s + max (ɣ [j], j -  λ[T[s+j]])
```

How do passwords work ?

A **password**, sometime also called passcode is secret data, typically a string of characters usually used to confirm a user's identity. Passwords are generally not stored in plain text, as they can be easily grabbed by malicious eyes. Instead we hash the password using a hash function and store its output. Every time the user inputs the password, we calculate its hash using the same function and compare the output with whatever was stored.

A cryptographic hash function must have the following properties

1. It is quick to compute the hash value of any given message
2. It is infeasible to generate a message that yields a given hash value
3. It is infeasible to find two different messages with the same hash value
4. A small change to a message should change the hash value so extensively that a new hash value appears uncorrelated with the old hash value.

Some common examples of cryptographic hash functions are MD5 and SHA-256.

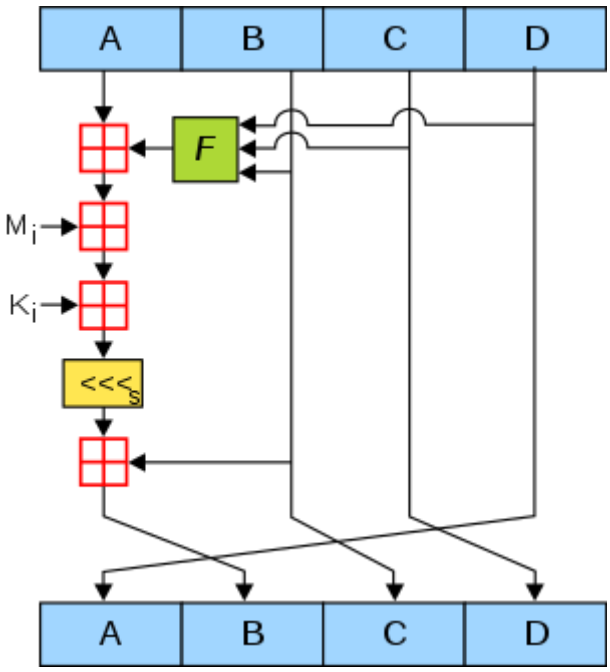
We will now explore the algorithm behind the MD5 hash function.

MD5

MD5 processes a variable length message into a fixed length output of 128 bits.

1. The input message in broken up into chunks of 512 bit blocks.
2. The message is then padded so its length is divisible by 512.
   The message is padded as follows
    1. Append a single bit, 1 to the end of the message
    2. Add as many zeros a required to bring the length of the message up to 64 bits fewer than a multiple of 512.
    3. The remaining bits are filled up with 64 bits representing the length of the original message modulo 2^{64}

The algorithm operates in a 128 bit state, divided into four 32 bit words, denoted A, B, C and D, which are initialized to certain fixed constants. It then uses each of the 512 bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, modular addition, and left rotation. There are four possible functions; a different one is used in each round.



Given below are the four possible functions, where $\oplus$, $\wedge$, $\vee$, $\neg$ represent XOR, AND, OR and NOT respectively.

$$F(B,C,D) = (B \wedge C) \vee (\neg B \wedge D)$$
$$G(B,C,D) = (B \wedge D) \vee (C \wedge \not{D})$$
$$H(B,C,D) = B \oplus C \oplus D$$
$$I(B,C,D) = C \oplus (B \wedge \neg D)$$

The psuedocode for MD5 algorithm is given below

```
# : All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int s[64], K[64]
var int i
# s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22
}
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20
}
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23
}
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21
}

# Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63 do
    K[i] := floor(232 × abs (sin(i + 1)))
end for

# Initialize variables:
var int a0 := 0x67452301   // A
var int b0 := 0xefcdab89   // B
var int c0 := 0x98badcfe   // C
var int d0 := 0x10325476   // D

# Pre-processing: adding a single 1 bit
append "1" bit to message
 # Notice: the input bytes are considered as bits strings,
 #  where the first bit is the most significant bit of the byte.[51]
```

```
# Pre-processing: padding with zeros
append "0" bit until message length in bits ≡ 448 (mod 512)

# Notice: the two padding steps above are implemented in a simpler way
#  in implementations that only work with complete bytes: append 0x80
#  and pad with 0x00 bytes so that the message length in bytes ≡ 56 (mod 64).
append original length in bits mod 264 to message

# Process the message in successive 512-bit chunks:
for each 512-bit chunk of padded message do
    break chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15
    # Initialize hash value for this chunk:
    var int A := a0
    var int B := b0
    var int C := c0
    var int D := d0
    # Main loop:
    for i from 0 to 63 do
        var int F, g
        if 0 ≤ i ≤ 15 then
            F := (B and C) or ((not B) and D)
            g := i
        else if 16 ≤ i ≤ 31 then
            F := (D and B) or ((not D) and C)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
            F := B xor C xor D
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
            F := C xor (B or (not D))
            g := (7×i) mod 16
        // Be wary of the below definitions of a,b,c,d
        F := F + A + K[i] + M[g]  // M[g] must be a 32-bits block
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
    end for
    # Add this chunk's hash to result so far:
    a0 := a0 + A
    b0 := b0 + B
    c0 := c0 + C
    d0 := d0 + D
end for
var char digest[16] := a0 append b0 append c0 append d0 # (Output is in little-endian)
```

MD5 is no longer a suggested to use in security applications since it has been proven to generate the same hash for different inputs.

How does encryption work ?

Encryption is the processes of encoding information. This process converts the original representation of information into an alternative form caller cipher text. Encryption protects that data you send via the internet like messages, mails, etc. Encryption allows an authorized party to decrypt the data and prevents the data from being visible to any prying eyes. It also prevents reading your data from being read without authorization.

An encryption scheme usually uses a pseudo-random encryption key generated by an algorithm. This key will be used to decrypt the data encrypted by the algorithm. It is possible to decrypt the message without possessing the key but, for a good algorithm considerable amount of computational resources and skills are required. A user is authorized by sharing this key with them.

There a two types of cryptographic systems

1. Symmetric key, where both encryption and decryption keys are the same
2. Asymmetric or Public Key, where encryption key is published for everyone to use, however only the party with decryption key can read it.

We will now describe the **RSA** algorithm, a very popular public key cryptosystem.

RSA

RSA is short for Rivest-Shamir-Adleman, the surnames of the people who publicly described the algorithm. In a public-key cryptosystem, the encryption key used to encrypt the data is public and is different from the decryption key used to decrypt the data which is kept a secret.

The RSA algorithm creates a public and private key pair based on two large prime numbers, which are kept secret. Messages can be encrypted by anyone by anyone with the public key but can only be decrypted using the private key. Authorizing a party is as simple as sharing the private key with it.

The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers. There are no published methods to decrypt the message in reasonable time if a large key is used. RSA is a relatively slow algorithm, because of this it is not commonly used to directly encrypt user data.

RSA involves four steps:

1. Key generation
2. Key distribution
3. Encryption
4. Decryption

Principle : The basic principle behind RSA come from the observation that it is practical to find three very large positive integers e, d and n, such that with modular exponentiation for all integers m, the below equation is satisfied

$$(m^e)^d \equiv m \ (mod \ n)$$

and the fact that even after knowing e, n and m, it can be extremely difficult to find d. The public key is represented by integer n and e, and the private key by the integer d. m represents the message and the method for generating it is described below.

1. **Key generation**
    1. Choose two distinct prime numbers p and q
        - For security purposes they are chosen at random and should be similar in magnitude but differ in length by a few digits to make factoring harder.
        - Prime numbers can be efficiently found using the primality test, where you check whether the random number generated is prime or not in polynomial time. A simple primality test is trial division where you check the divisibility of n with prime numbers between 2 and $\sqrt{n}$. There are other algorithms for primality testing as well which can be found here
    2. Compute $n = pq$
        - n is used as the modulus for both the public and private keys. Its length, usually expressed bits is the length of the key.
        - n is released as a part of the public key.
    3. Compute $\lambda(n)$

        λ is the Carmichael's totient function. It is defined as the smalles positive integer m such that $a^m \equiv 1 \ (mod \ n)$ for every integer a between 1 and n that is co-prime to n. In algebraic terms, $\lambda(n)$ is the exponent of the multiplicative group of integers modulo n. If n can be written in a unique way as $n = p_1^{r_1}p_2^{r_2}..$, then $\lambda(n) = lcm(\lambda(p_1^{r_1}), \lambda(p_2^{r_2}), \dots)$, this can be proven using Chinese remainder theorem. Carmichaels' theorem states that $\lambda(p^r)$ is equal to euler totient $\phi(p^r)$ if r is power of an odd prime.

        - Since $n = pq$, $\lambda(n) = lcm(\lambda(p), \lambda(q))$ and since p and q are prime $\lambda(p) = \phi(p) = p - 1$ and $\lambda(q) = \phi(q) = q - 1$. Thus $\lambda(n) = lcm(p - 1, q - 1)$. lcm can be calculated using the euclidean algorithm.
    4. Choose an integer e such that $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$
        - e having a short bit length and small hamming weight (that is small sum of digits) results in more efficient encryption. The most commonly chosen value for e is $2^16 + 1 = 65,537$. The smallest and fastest possible value for e is 3, but it has been shown to be less secure.
        - e is released as a part of public key
    5. Determine d as $d \equiv e^{-1}(mod \ \lambda(n))$
        - Solve for d in the equation $de \equiv 1(mod \ \lambda(n))$, d can be computed efficiently by using the extended euclidean algorithm, since e and $\lambda(n)$ are co prime, the equation is a form Bézout's identity, where d is one of the coefficients.
        - d is kept secret as the private key exponent.

    The public key now consists of the modulus n and the public exponent e. The private key consists of the exponent d. The values used in generating keys p, q and $\lambda(n)$ must also be kept secret, and are discarded.

2. **Key distribution**
    The receiver sends the public key to encrypt the message to the sender via a reliable but not neccesarily a secure route.

3. **Encryption**
    After the sender receives the receiver's public key, he now encrypts the message as follows. The sender first converts the message into an integer m, such that $0 \leq m < n$ by using an agreed upon reversible protocol known as a padding scheme. He then computes the ciphertext c using receiver's public key e corresponding to $m^e \equiv c \ (mod \ n)$. This is efficient even for very large numbers. This ciphertext is now transmitted to the receiver.

4. **Decryption**
   The receiver can recover m from c by using the private key which consists of the exponent d by computing $c^d \equiv (m^e)^d \equiv m \ (mod \ n)$. Once m is recovered, the original message M is obtained by reversing the padding scheme.

Padding schemes

RSA is prone to multiple attacks such as

- When encrypting with low encryption exponents small values of m then the result of m^e is lessthan the modulus n. In this case ciphertexts can be decrypted easily by taking eth root.
- An attacker, try out multiple likely plain text messages to check which one is equal to encrypted message after encryption. This is possible since RSA is deterministic and has no random component.

The above are only a few examples of attacks that are possible against RSA. To avoid these problems, RSA typically embed some form of structured randomized padding into value m before encrypting it. There are various standards for padding. An example of which is Optimal Asymmetric Encryption Padding.

Digital Signature

Another application of **RSA** is digital signature, used for verifying the authenticity of digital messages or documents. Here the sender's public and private key are used instead of the receiver's.

1. The sender converts the message M into an integer m using a message digest function like MD5, which is then encrypted with the sender's public key and sent along with the original message. The encrypted part is known as the **Digital signature**
2. After the receiver receives the original message, sender's digital signature, It converts the received message into an integer using the same message digest function used above. The digital signature is then decrypted using sender's public key and compared with the value obtained from the message digest function. If it matches the message is accepted else the message is rejected.


Thank You