Longest Prefix Matching (Maximum prefix length match): It refers to an algorithm used by routers in Internet Protocol (IP) networking to select an entry from a forwarding table. Its lookup the IP prefix which will be the destination of the next hop from the router. The routing tables each router stores IP prefix and the corresponding router. This algorithm is used to find the prefix matching the given IP address and returns the corresponding router node

IP prefix is a prefix of IP address. All computers on one network have same IP prefix. For example, in 192.24.0.0/18, 18 is length of prefix and prefix is first 18 bits of the address. Routers basically look at destination address's IP prefix, search the forwarding table for a match and forward the packet to corresponding next hop in forwarding table.

Because each entry in a forwarding table may specify a sub-network, one destination address may match more than one forwarding table entry. The most specific of the matching table entries — the one with the longest subnet mask, is called the longest prefix match. It is called this because it is also the entry where the largest numbers of leading address bits of the destination address match those in the table entry.

The router implements the Longest Match Routing Rule as follows:

1. The router receives a packet.
2. While processing the header, the router compares the destination IP address, bit-by-bit, with the entries in the routing table.
3. The entry that has the longest number of network bits that match the IP destination address is always the best match (or best path) as shown in the following example:

Longest Match Example:

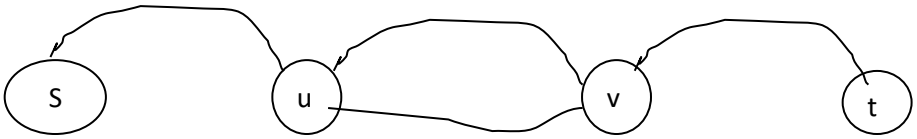
- The router receives a packet with a destination IP address of 192.168.1.33.
- The routing table contains the following possible matches:
  - 192.168.1.32/28
  - 192.168.1.0/24
  - 192.168.0.0/16
- To determine the longest match, it's easiest to convert the IP addresses to binary and compare them.

Address	Converted Binary Address
192.168.1.33	11000000.10101000.00000001.00100001 (destination IP address)
192.168.1.32/28	11000000.10101000.00000001.00100000 (<–Best match)
192.168.1.0/24	11000000.10101000.00000001.00000000
192.168.0.0/16	11000000.10101000.00000000.00000000

Naïve Algorithm:

Observation: Any subpath of an optimal path is also optimal

Proof: Consider an optimal path from some origin S to t and two vertices u and v on this path. They can coincide with some S or t or some node in the middle. Suppose there are some shorter path from u to v , then would be able to go from shortest path from s to t. If there were a shorter path from u to v we would get a shorter path from S to t.



Corollary: If S->……..->U->t is the shortest path from S to t,

Then

$$d(S, t) = d(S, u) + w(u, t)$$

Distance Weight of edge between u to t

And we will use this property to improve our estimates of distances from some node or origin

dist[v] will be an upper bound on the actual distance from S to v

The edge relaxation procedure for an edge (u,v) just checks whether going from S to v through u improve the current value of dist[v]

Relax(u,v)

If dist[v] > dist[u] + w(u,v)

dist[v] <- dist[u] + w(u,v)

prev[v] <- u

Naïve(G,S)

For all u,

prev[u]<-nil

dist[S]<-0

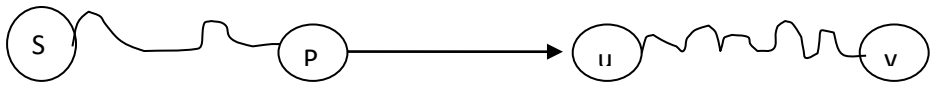
do: relax all the edges

while at least one dist changes

Lemma:- After the call to Naïve Algorithm all the distances are set correctly.

Proof:- Assume , for the sake of contradiction, that no edge can be relaxed and there is a vertex v such that dist[v] > d(S,v)

Consider a shortest path from S to v and let u be the first vertex on this path with the same property. Let P be the vertex right before



Then d(S,P)= dist[P] and hence

d(S,u)= d(S,p)+ w(P,u)

= dist[p] + w(P,u)

dist[u] > d(S,u) = dist[P] + w(P,u) ➔ edge(P,u) can be relaned-

A contradiction

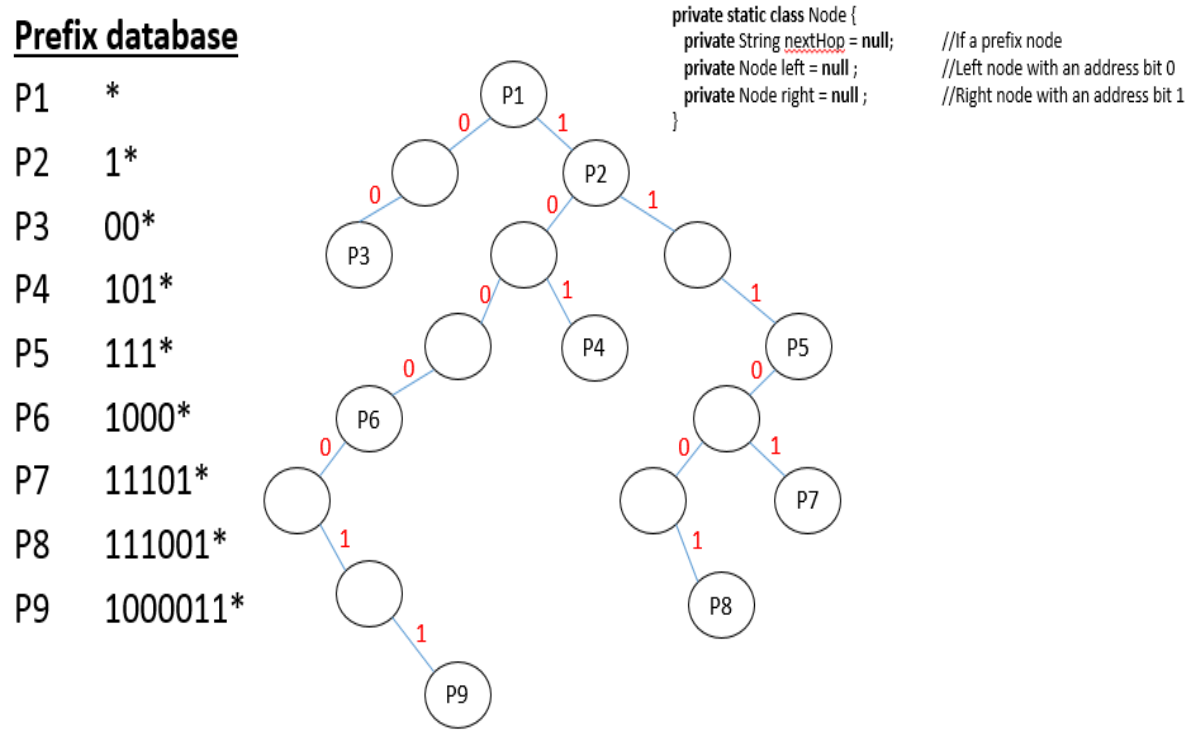
Binary Tries

A **trie**, also called **digital tree** or **prefix tree**, is a type of search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In order to access a key (to recover its value, change it, or remove it), the trie is traversed depth-first, following the links between nodes, which represent each character in the key.

Unlike a binary search tree, nodes in the trie do not store their associated key. Instead, a node's position in the trie defines the key with which it is associated. This distributes the value of each key across the data structure, and means that not every node necessarily has an associated key. All the children of a node have a common prefix of the string associated with that parent node, and the root is associated with the empty string. This task of storing data accessible by its prefix can be accomplished in a memory-optimized way by employing a radix tree.

Though tries can be keyed by character strings, they need not be. The same algorithms can be adapted for ordered lists of any underlying type, e.g. permutations of digits or shapes. Tries allow finding the longest prefix that matches a given destination address and the search is guided by the bits of the destination address. While traversing the tries and visiting a node marked as a prefix, this prefix is marked as the longest match found so far. The search ends when no more branches to take exist and the longest match is the prefix of the latest visited prefix node.

In particular, a **binary tries** is keyed on the individual bits making up a piece of fixed-length binary data, such as an integer or memory address. The idea behind it is simple: Each string is represented by a leaf in a tree structure, and the value of the string corresponds to the path from the root of the tree to the leaf.



For a binary tries the worst case scenario for IPv4 lookup is 32 random reads and for IPv4 insertion the worst case would be 32 random writes (to add 32 new nodes to the tries). The upper bound lookup complexity is O(n) where n is the length of the prefix being search for; a /8 requires fewer steps through the node tries than a /32. The upper bound is roughly O(n^w) where n is the number of prefixes and w is their length (this is only roughly because some prefixes in the tries will share branch nodes).

The search algorithm can be implemented very efficiently as demonstrated by the pseudo code. Let s be the string searched for and let EXTRACT(p,b,s) be a function that returns the number given by the b bits starting at position p in the string s. We denote the array representing the tries by T. The root of the tries is stored in T[0]. Note that the address returned only indicates a possible hit; the bits that have been skipped during the search may not match. Therefore, we need to store the values of the strings separately and perform one additional comparison to check whether the search actually was successful.

```
node = T[0];
pos = node.skip;
branch = node.branch;
adr = node.adr;
while (branch != 0) {
  node = T[adr + EXTRACT(pos, branch, s)];
  pos = pos + branch + node.skip;
  branch = node.branch;
  adr = node.adr;
}
return adr;
```

This simple structure is not very efficient. The number of nodes may be large and the average depth may be long. The traditional technique to overcome this problem is to use **path compression.** Each internal node with only one child is removed. We store a number, the skip value, in each node that indicates how many bits have been skipped on the path. The total number of nodes in a path-compressed binary tries is exactly 2n-1, where n is the number of leaves in the tries. For a large class of distributions the average depth of a path-compressed tries is proportional to log n.

Path compression is a way to compress parts of the tries that are sparsely populated. The idea is to replace the i highest complete levels of the binary tries with a single node with 2^i descendants. This replacement is performed recursively on each subtries. The standard implementation of a trie, where a set of children pointers are stored at each internal node, is not a good solution, since it has a large space overhead. A space-efficient alternative is to store the children of a node in consecutive memory locations. In this way, only a pointer to the left-most child is needed. In fact, the nodes may be stored in an array and each node can be represented by a single word containing three numbers:

- The branching factor.
- The skip value.
- A pointer to the left-most child.

Compressed unibit or Patricia tries compress the height (the distance from the root node to a leaf node); "[for any] internal node of the unibit trie that does not contain a next-hop and has only one child [it] can be removed in order to shorten the path from the root node. By removing these nodes, we need a mechanism to record which nodes are missing" so some extra information must be stored at each node in order to reduce trie height. "Each IP lookup starts at the root node of the trie. Based on the value of skipped bits of the destination address of the packet [the "skipped bits" value in the current node], the lookup algorithm determines whether the left or the right node is to be visited. The next hop of the longer matching prefix found along the path is maintained as Best Matching Prefix (BMP) while the tries is traversed".

"Compression can reduce the height of a sparse unitbit trie. However, when the unibit trie is full (every node has two children) then there is no compression possible and the two versions will look identical. Thus, a lookup operation requires 32 random reads in the worst case for IPv4, O(n) [where n is the length of the prefix to lookup]. Considering a unibit trie with N leaves, there can be N-1 internal nodes between the leaves and the root including the root.The total amount of memory required will be at most $2n - 1$ [where n is the number of prefixes]. Since path compression is possible and some internal nodes can be removed, the space complexity becomes O(n) [number of prefixes], independent of the length of a prefix. Thus, path compressed tries reduce space requirements, but not the search complexity."

MULTIBIT TRIES

Drawback of binary (1-bit) trie is that one bit at a time is inspected and the number of memory accesses (in the worst case) can be 32 for IPv4. If the memory access time is 10 ns, the lookup operation will consume 320 ns. This implies a maximum forwarding speed of 3.125 million packets per second (mpps) (1/320ns). If we assume minimum sized 40-byte IP packets which is a TCP-acknowledgment packets, this can support links speed of at most 1 Gbps. However, networks these days require supporting link speeds as high as 10 Gbps and more. The main principle of the multibit trie to examine several bits at a time (called a stride) in order to improve the performance. For example, if we examine 4 bits at a time, the lookup will finish in 8 memory accesses as compared to 32 memory accesses in a unibit trie.

Number of bits (K) to be inspected is called a stride and the stride can be constant or variable. In the multibit trie structure, each node has a record of 2^{stride} entries and each has two fields: one for the stored prefix and one for a pointer that points to the next child node. If all the nodes at the same level have the same stride size, we call it a fixed stride; otherwise, it is a variable stride.

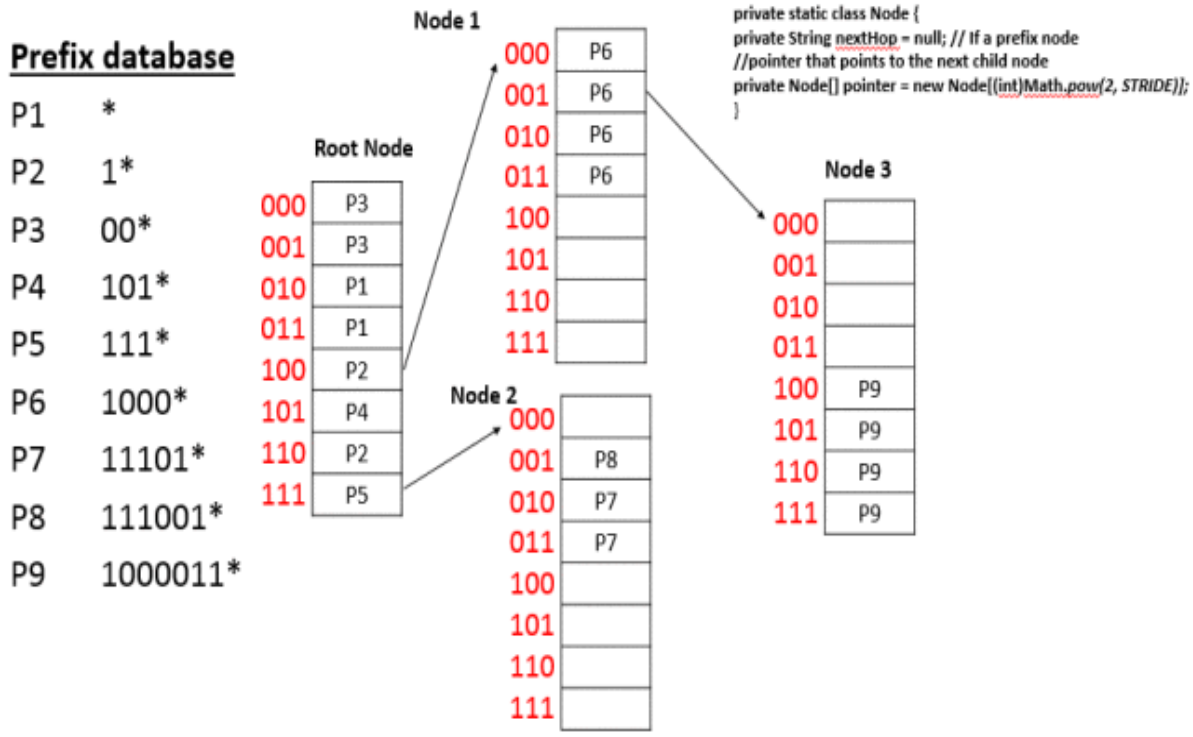
With multi-bit tries the "stride" length of the key is examined at each level in the tries. A unit-bit trie would in the worst case require 32 memory accesses for an IPv4 prefix lookup because it has a stride length of 1 bit. If the memory access time is 10ns this would equate to 320ns and 67ns it required for 10Gbps so the throughput would drop to circa 2.1Gbps. With a multi-bit tree k lookups are required (where k is the stride size). Given a prefix of length n the upper bound lookup complexity is O(n/k). If k is 8, then at worst for an IPv4 address lookup 4 memory accesses would be required (32/8 == 4). This is also the trie height calculation for the multi-bit trie; key size / stride size (32 / 8 == 4 for example). For a unit-bit trie for IPv4 forwarding the trie height would be 32 / 1 == 32.

"An update [of a multi-bit trie] requires a search through [prefix length]/k lookup iterations plus access to each child node (2k). The update complexity is O([prefix length]/k + 2k). In the worst case, each prefix would need an entire path of length ([prefix length]/k) and each node would have 2k entries. The space complexity would then be O((2k ∗ [number of prefixes] ∗ [prefix length])/k). As we increase the stride size the number of levels will decrease and the number of memory access will also decrease. However, memory space consumed will be larger. Therefore, when choosing stride size a tradeoff

happens between the memory accesses and space requirements. The designer needs to determine the proper number of levels based on the desired lookup speed.

In a K-bit trie, each node has 2K pointers (children). If a route prefix is not a multiple of K, it needs to be expanded to K or its multiples. Lookup time is O(W/K) and storage requirement O(2 (K-1) NW/K)
For example, if the maximum lookup speed should be 30ns and the designer is presented with memory chips at a cost of 10ns for access time, then the number of levels needs to be at max 3. Obviously, choosing the optimal stride size of each level in order to minimize the memory space varies and is highly dependent on the prefixes database." - this is where variable stride size multi-bit tries come in.

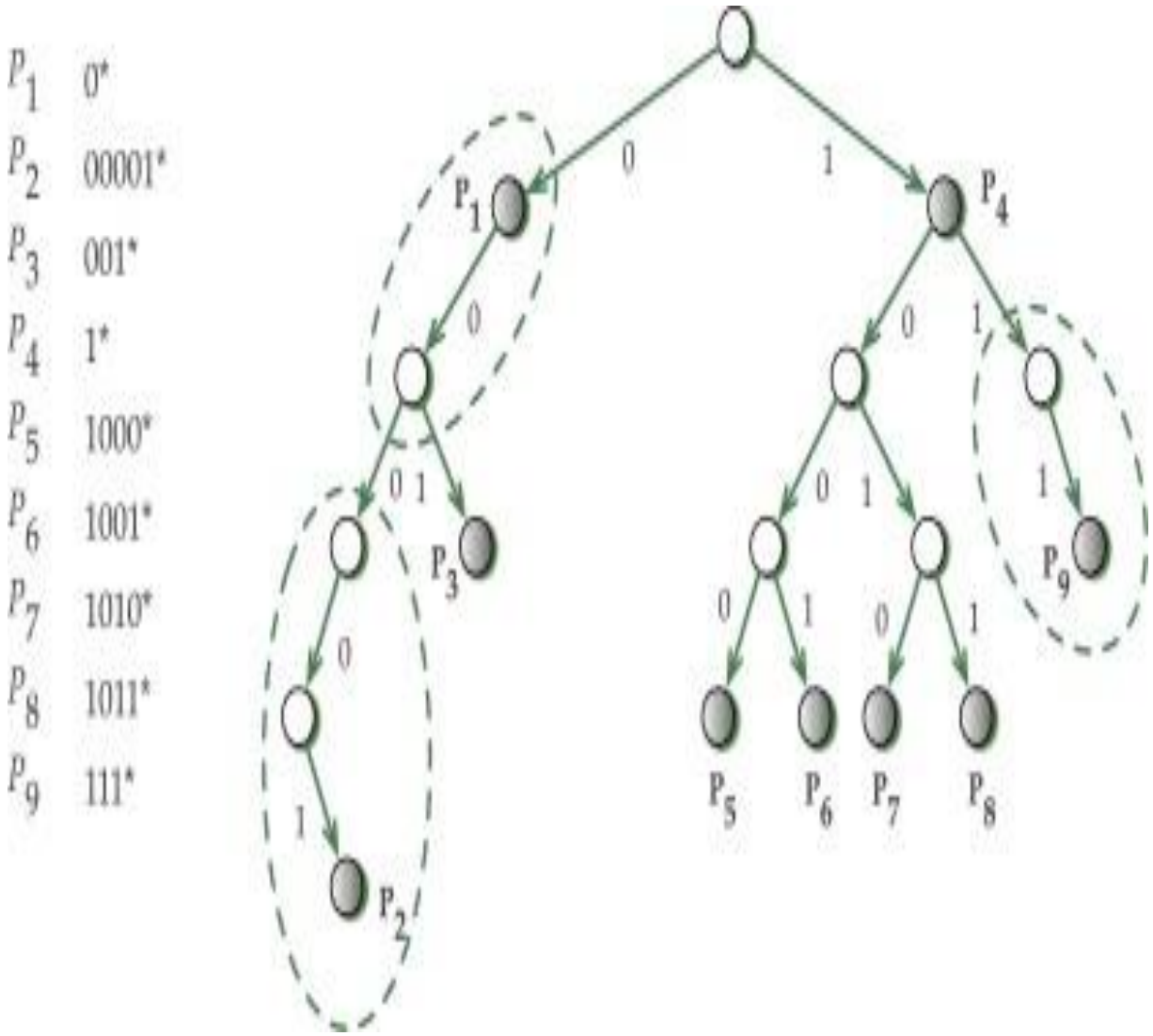
Figure shows an expanded tries with a fixed stride length of 3. Thus each trie node uses 3 bits. The replicated entries correspond exactly to the expanded prefixes. For example, P7 has two expansions (111010 and 111011). These two expanded prefixes are pointed by the 111 pointer in the root node (because both expanded prefixes start with 111) and are stored in 010 and 011 entries of node 2. Notice also that the entry 111 in the root node has a stored prefix P5 (besides the pointer pointing to P7 expansions), because P5 = 111 is itself an expanded prefix.



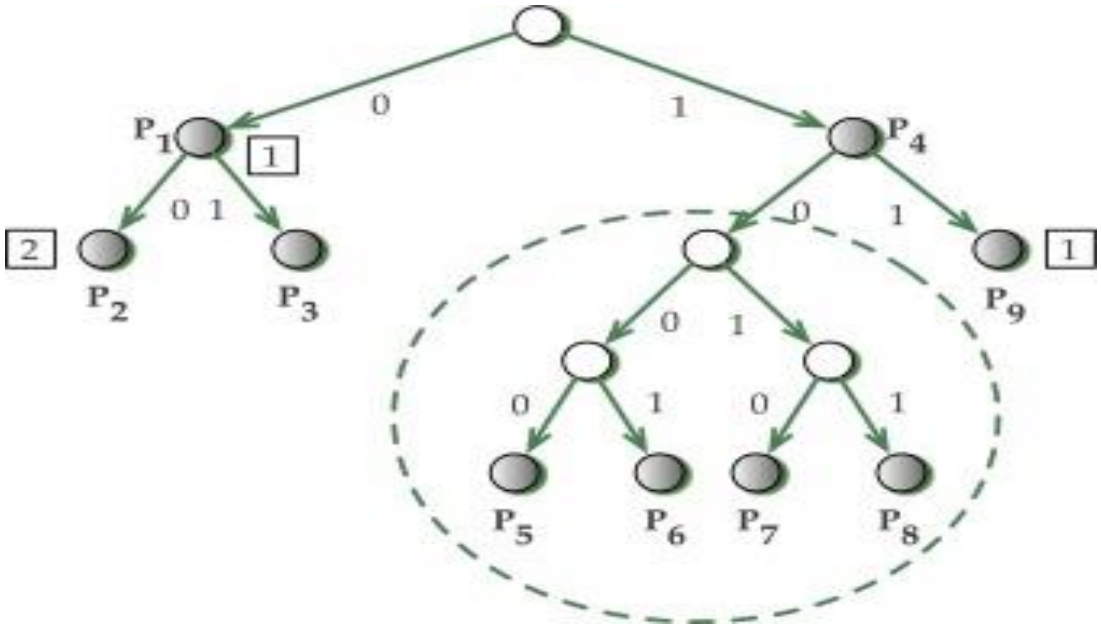
Compressing Multi-bit Strides:

Compression algorithms exist for multi-bit tries, namely an LC tries (Level Compressed). Multibit tries use prefix expansion to reduce the number of levels in a tries; however, this is at the expense of increased storage space. This can be viewed alternatively as compressing the levels of the tries, which sometimes is referred to as level compression. A closer examination of multibit tries shows that space is especially wasted in the sparsely populated regions of the tries. The trie region containing the prefix P2 is sparse as no other prefixes are nearby. Now examine the fixed stride multibit tries variant of the binary tries in Figure. The leftmost subtrie contains only one prefix P2 and the rest of the three locations are not used. Such sparse regions of a binary tries that contain long sequences of one-child nodes can be compressed by the technique called path compression. There is another trie-based scheme called level compressed tries (LC-tries) that combines both path and level compression . The main motivation behind this scheme is to reduce storage by ensuring that the resulting tries nodes do not contain empty locations. The scheme starts with a binary tries and transforms it into a level compressed tries in multiple steps. We illustrate this transformation using the binary tries shown in Figure. First, path compression is applied. This removes the sequences of internal nodes having one child. However, we need to keep track of the missing nodes somehow. A simple solution is to store a number called the *skip value* (*s*), in each node that indicates how many bits need to be skipped on the path. We show that the sequences of nodes leading to P2 and P9 have only one child, and hence, they are candidates for path compression.

Identifying the paths to be compressed in the binary tries shown in Figure

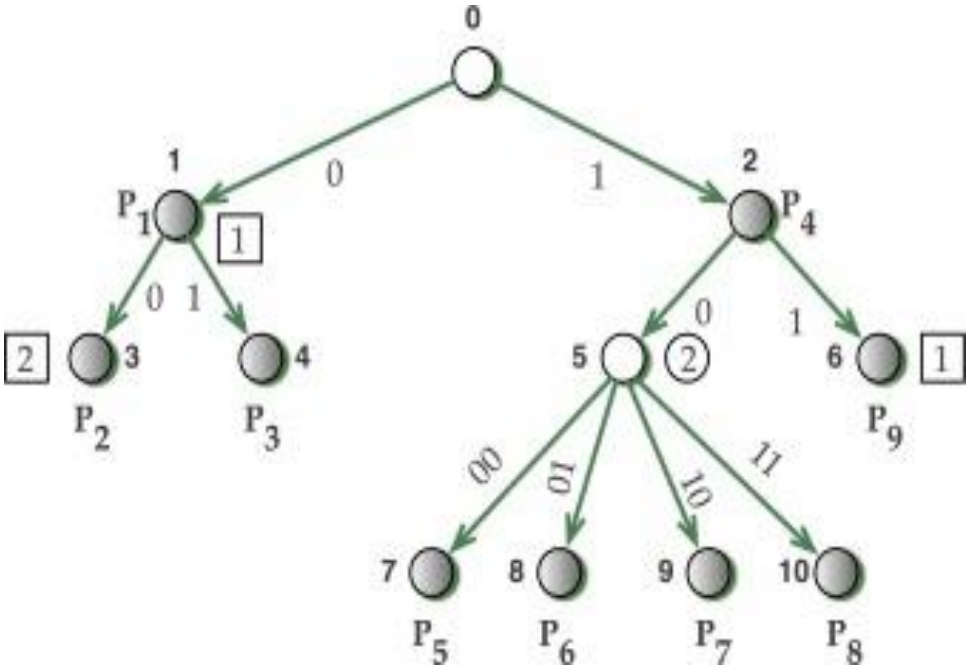


Identifying the levels to be compressed in the tries



After path compression, level compression is used for compressing the parts of the binary tries that are densely populated. Instead of a node having two children, as in a binary tries, each internal node in a multibit tries is allowed to have 2k children, where *k* is called the *branching factor*. The level compression is applied as follows. Every node *n* that is the root of a complete subtries of maximum depth *d* is replaced by a corresponding one-level multibit subtries. This process is repeated recursively on the children of the multibit tries thus obtained. Again, referring to Figure, we find that the tries rooted at the left child of node P4 is a complete subtries of depth 2. This tries can be replaced by a single-level multibit tries with four nodes. The branching factor for the left child of node P4 is 2, indicated by the number enclosed in a circle adjacent to the node. The rest of the internal nodes have a default branching factor of 1 and are not shown in Figure. The leaf nodes have a branching factor of 0.

Level compressed tries



Since at each step we replace several levels with a single-level multibit tries, this process can be viewed as the compression of levels of the original binary tries. Hence, the resultant tries is termed a level compressed tries or simply an LC-tries.

The main drawback of the scheme is that the structure of the binary trie determines the choice of strides at any given level without any regard for the height of the resulting multibit tries. This is because a subtries of depth *d* can be substituted only if it contains all the 2d leaves (a full binary subtries). Hence, a few missing nodes might have a considerable negative influence on the efficacy of the level compression.

A simple optimization is to use a relaxed criterion where nearly complete binary tries are replaced with a multibit subtries. In other words, if the nearly complete binary subtries has a sufficient fraction of the 2d tries at level *d*, then it is replaced with a multibit subtries of stride *d* pointing to 2d nodes. The parameter that controls the fraction is referred to as the *fill factor* x,$0<x\leqslant 1$. Such a relaxed criteria will decrease the depth of the tries but will introduce empty nodes into the tries. However, in practice, this scheme results in substantial time improvements with only a moderate increase in space.

For optimizing storage, LC-tries do not use the standard implementation technique that uses a set of child pointers at each internal node. Instead, an LC-tries is represented using a single array and each tries node is an element in the array. The LC-tries can be considered a special case of a variable-stride tries.

Hardware Algorithms:

A hardware algorithm is a procedure suitable for hardware implementation and the target hardware model. There are several hardware algorithms used for processing implementation in hardware, with specific emphasis on parallelism, control, and data-flow of processing.

Over the past 10-15 years the area of hardware algorithms has grown from a collection of isolated algorithms and analysis methods into a cohesive body of research and development. The problems in this area are characterized by several requirements, of which speed, scalability and simplicity are the most important. For algorithms designed to operate in high-speed router hardware, there is the additional stringent constraint of low heat dissipation.
While deployment of embedded and distributed network services imposes new demands for flexibility and programmability, IP address lookup has become significant performance bottleneck for the highest performance routers.

Hardware algorithms significantly influence design cost and system performance of NoC. Networks on Chip (NoC) have been widely used for its smart structure and high performance. We evaluate NoC in terms of circuit resource, average latency, max latency, average throughput and power consumption.

Networking hardware on a board falls under a type of I/O (input/output) hardware, and is responsible for transmitting data into and out of the device. At the highest level, I/O networking hardware can be classified according to how the hardware manages the transmission and reception of data, specifically whether the physical layer manages data in *serial*, in *parallel*, or some hybrid combination of *both*. Networking hardware that is classified as serial, such as EIA/RS-232, manages incoming and outgoing data one bit at a time. Hardware that can manage data in parallel is a physical layer which has the ability to manage multiple bits simultaneously. Hardware such as that based on IEEE

802.3 Ethernet has the capability of supporting both serial and parallel communication and can be configured to manage data either way.
An I/O networking hardware subsystem on an embedded systems board is typically made up of some combination of the following six logical units:

a. **Transmission Medium**: wireless or wired medium that connect the embedded system to a network

b. C**ommunication (COM) port**: the components on the embedded board in which a wired medium connects to or that receives the signal of a wireless transmission medium

c. **NetworkController:** a slave processor that manages the networking communication from the other logical units on the board

d. **Master processor's integrated networking I/O**, master processor-specific networking components

e. **Communication interface**: which manages data communication and the encoding/decoding of data. It can be integrated into the master processor or another IC (integrated circuit) on the board

f. **I/O Bus**: connects master processor to other networking I/O logical units on the board.

Given a serial networking subsystem, for example, that hardware would be made up of some combination of the above logical units, including a ‘serial’ interface and ‘serial’ port. A parallel networking subsystem would, instead, have a ‘parallel’ interface and a ‘parallel’ port.

How that networking hardware fits into the technical architecture for our enterprise application. From the enterprise applications administrator perspective, we’ll focus on how to apply those designs to our technical architecture and reflect them on the technical architecture diagram.

The networking industry is facing enormous challenges of scaling devices to support the exponential growth of internet traffic as well as increasing number of features being implemented inside the network. Algorithmic hardware improvements to networking components have largely been neglected due to the ease of leveraging increased clock frequency and compute power and the risks of implementing complex hardware designs. As clock frequency slows its growth, algorithmic solutions become important to fill the gap between current generation capability and next generation requirements.

Comparing Different Approaches:

Comparison of Pre-processing Algorithms:

As a result of the differences between wired and wireless networks, the kind of graph that is more appropriate to each environment varies. For instance, in wireless environments it is crucial to use few messages and to use localized pre-processing algorithms. Unlike this, in wired environments there is no broadcasting facility and a higher communication overhead is admissible. In Table 1, we evaluate the difficulty of creating each of the graphs for wired and wireless environments. We use the following abbreviations: “S” — possible and simple; “P” — possible but may take several rounds of messages or have a high communication cost; “I” — impossible; “N” — makes little sense. Some of the entries of the table are to some extent subjective. Hence, according to this logic, all algorithms take several rounds in wired networks (when compared to wireless networks). We also classify the LDel (k) (V), $k \geq 2$ algorithm for wireless networks with a “P”, because good algorithms to do this graph tend to be more complex. Naturally, new, simpler algorithms could make this classification change.

Table 1: Comparison of pre-processing algorithms for wireless and wired networks

	RNG	GG	[28]	P LDel(V)	LDel (k) (V), $k \geq 2$	DT
Wireless	S	S	S	S	P	I
Wired	P	P	N	N	N	P

Comparison of Routing Algorithms: Since the main goal of any routing algorithm is to deliver packets, the most important evaluation criterion is unquestionably the delivery success rate. However, guaranteed delivery cannot be achieved at any price, for instance, by flooding each packet throughout the entire network. Hence, another relevant criterion is the communication effort of the algorithm. In particular, one important distinction is whether the algorithm uses flooding to deliver packets or not. Memory requirement of the algorithm is also important, because some algorithms require information about Comparison of Routing Algorithms Since the main goal of any routing algorithm is to deliver packets, the most important evaluation criterion is unquestionably the delivery success rate. However, guaranteed delivery cannot be achieved at any price, for instance, by flooding each packet throughout the entire network. Hence, another relevant criterion is the communication effort of the algorithm. In particular, one important distinction is whether the algorithm uses flooding to deliver packets or not. Memory requirement of the algorithm is also important, because some algorithms require information about

Algorithm	Guarant. del.	Local.	Memory	c-comp.
Greedy, Compass	No/DT	Yes/No	Memoryless	No
MFR	No	Yes	Memoryless	No
Rand. compass	Yes	Yes	Memoryless	No
GEDIR	No/DT	Yes/No	O(1)	No
Voronoi	DT	No	O(1)	No
Parallel Voronoi	DT	No	O(1)	Yes
FACE-1, FACE-2, GPSR, AFR, GOAFR+	Planar G	Yes	O(1)	No
SP Algorithms	Yes	No	O(n log n), O(1	Yes

Table 2 resumes a comparison between routing algorithms. We include “Shortest Path Algorithms” (SP algorithms), i.e., algorithms that are not based on position. These algorithms inherit from the techniques of the wired IP networks. The values of the table are valid for the Destination Sequenced Distance-Vector (DSDV), Ad hoc On-Demand Distance Vector Routing (AODV) and Dynamic Source Routing (DSR) . We assumed a worst-case sscenario where all nodes have routing entries to all other nodes, thus requiring O(n log n) memory, per node, where n is the number of nodes.

The memory entry of the table for SP algorithms includes this value, followed by the space required by the algorithm in each packet. Position-based algorithms only include the latter, because space needed to store the graph strongly depends on the pre-processing algorithm used (but this is typically O(log n)). Regarding the space used in each packet, all the algorithms need at least O(log n) bits to identify the destination. However, to make a clearer distinction between them, in Table 2, we refer to the number of other nodes in the packet, besides the destination (O(1) means that the algorithm needs the source, while memory less means that the source is not needed).

Of the routing algorithms included in the table, Greedy, MFR, GEDIR, Randomized Compass, Voronoi as well as shortest path algorithms are loop-free. The face and hybrid algorithms are also loop free in the sense that partial loops occur in a controlled way, in planar graphs. Of the localized algorithms included, only randomized compass guarantees delivery in arbitrary graphs. We assume that SP algorithms can find the optimal path, under favorable circumstances. Algorithms may use hop count, energy or distance as their metric. Finally, of the algorithms that we include in the comparison, only shortest path algorithms use flooding. Depending on the particular case, these algorithms use flooding to propagate route requests or to propagate routing table.