Let me now make an analysis of the probability of false-positive errors. Let s be the size of the bit table, k the number of hash functions, and n the number of elements inserted in the filter. During each insertion, each bit in A is set by each hash function with probability $1/s$. Therefore, the probability that a bit in A is set (at least once) during n insertions is

$$1-\left(1-\frac{1}{s}\right)^{kn}.$$

Now, consider an element u not inserted in the filter is searched. The k hash functions produce k uniformly random locations in A, each containing a 1-bit with the above probability. Therefore, the probability that u is a false positive is

$$\left(1-\left(1-\frac{1}{s}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/s}\right)^{k}.$$

This probability is minimized with respect to k for

$$k \approx \frac{s}{n}\ln 2,$$

and the corresponding false positive probability is about

$$2^{-k} \approx (2^{-\ln 2})^{s/n} = p,$$

where p is the allowed error probability suitable for an application. This lets us choose the size s of the bit table A as

$$s \approx -\frac{n\ln p}{\ln^2 2}.$$

25.3 Backtracking and branch-and-bound algorithms

A non-deterministic algorithm is associated with a computation tree for every instance of input. Branching in the tree represents non-deterministic choices. Each leaf in the tree is marked by *Yes* or *No* in order to indicate the decision of the unique computation path from the root to that leaf. A systematic traversal of this computation tree visits all the leaves of the tree and lets us conclude about the decision on the given input instance. Indeed, this is precisely how we emulated a non-deterministic algorithm by a deterministic one (See Chapter 24). This emulation may lead to exponential running time (and even exponential space), but may be the only known way to solve certain problems in NP. Use of suitable heuristic strategies during the traversal may often curtail the search considerably, achieving significant practical speedup.

25.3.1 Backtracking

I now elaborate this idea formally. Each node in the computation tree is uniquely specified by the path from the root to that node, that is, by the non-deterministic choices made so far. We represent this information by a string C. In addition to C, we may need to maintain some other information D associated with the current sequence of choices. Against every node in the computation tree, we maintain the pair (C,D). Initially, no choices are made, so C is the empty string ε, and D_{init} represents the initial bookkeeping information.

While processing a node corresponding to the pair (C,D), we find out which nodes down the tree are reachable in one step from this node. Let these nodes be $(C_1,D_1),(C_2,D_2),\ldots,(C_k,D_k)$ (where k is the number of branches at the node (C,D)). We investigate the nodes (C_i,D_i) for all $i=1,2,\ldots,k$ in due course of time. We maintain a list Q of all nodes from which paths in the tree need to be explored in the future.

When a leaf node is generated (as some (C_i,D_i)), we check whether it is marked *Yes*. If so, we output *Yes*, and abort the search. If not, this particular leaf alone does not lead to a conclusive decision about the input, so the search continues. When the entire tree is traversed without ever visiting a leaf marked *Yes*, the algorithm outputs *No*. This traversal of the computation tree is often referred to as *backtracking*.

It is often possible to determine that the current path cannot lead to the answer *Yes* in any of the leaves present in the subtree rooted at the current node. If (C_i,D_i) is such a node, there is no point investigating further branching from this node. In that case, we say that the search has reached a *dead end* at (C_i,D_i). We do not need to add such nodes (C_i,D_i) to Q. Eliminating dead ends from the search process is often called *pruning*.

A high-level description of the backtracking algorithm is provided now.

```
Initialize $Q=\{(\varepsilon,D_{\text{init}})\}$.
while $Q$ is not empty {
    Take a node $(C,D)$ from $Q$, and delete that node from $Q$.
    Determine all the children $(C_1,D_1),(C_2,D_2),\ldots,(C_k,D_k)$ of $(C,D)$.
    for $i=1,2,\ldots,k$ {
        If $(C_i,D_i)$ is a leaf node marked Yes, return Yes.
        If $(C_i,D_i)$ is a non-leaf node {
            If the search has not reached a dead end at $(C_i,D_i)$, add $(C_i,D_i)$ to $Q$.
        }
    }
}
Return No.
```

Depending upon how Q is organized, we make different types of traversals in the computation tree. If Q is organized as a queue, we make a breadth-first traversal of the tree. If Q is organized as a stack, we make a depth-first traversal of the tree. Another interesting case is that Q is organized as a priority queue (heap). But then we need a way to assign relative priorities to the nodes. Suitable choices of node priorities may make the search efficient.

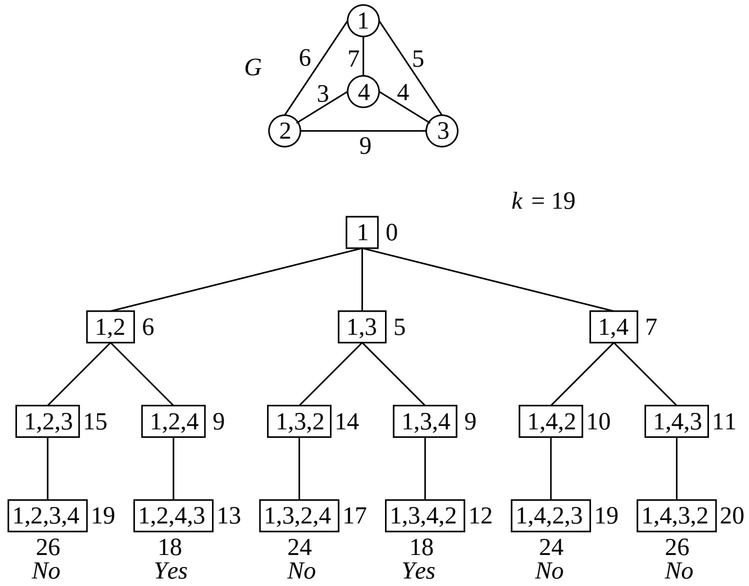
25.3.2 A backtracking algorithm for the traveling salesperson problem

Let us consider the decision version of the traveling salesperson problem. We are given a complete undirected graph G on n vertices with each edge e associated with a positive cost $c(e)$. We are also given a positive real number k. Our task is to decide whether G contains an undirected Hamiltonian cycle of total cost $\leqslant k$.

A Hamiltonian cycle in G is specified by a permutation $v_1,v_2,\ldots,v_n$ of the n vertices of G (with the implication that we follow the edge from (v_i,v_{i+1}) for $i=1,2,\ldots,n-1$, and finally come back to v_1 using the edge (v_n,v_1)). Cyclic permutations of $v_1,v_2,\ldots,v_n$ generate the same Hamiltonian cycle, so we start with a fixed vertex v_1. Subsequently, we non-deterministically choose the next vertices $v_2, v_3, \ldots$ so that there is no repetition in the sequence $v_1, v_2, \ldots, v_i$ for every $i \leqslant n$. Against every chosen path $v_1, v_2, \ldots, v_i$, we maintain the total cost incurred along that path.

When n (distinct) vertices are non-deterministically generated, we add the cost of the final edge (v_n, v_1) to the cost of the path $v_1, v_2, \ldots, v_n$, and check whether this cost is $\leqslant k$. A weighted complete undirected graph G on 4 vertices and the associated non-deterministic computation tree are shown in Figure 128. We take $k = 19$ as the second input. The current cost of the partially constructed path is shown beside every node in the tree. Below every leaf in the tree, the cost of the corresponding Hamiltonian cycle and the decision attached to that leaf are shown.

Figure 128: A non-deterministic computation tree for the TSP



A BFS traversal of this non-deterministic computation tree is shown in Figure 129. The shaded nodes are all the nodes in the tree, from which further exploration possibilities are investigated. The unshaded nodes in dark color are generated, but not explored. Either these are leaf nodes (from which no further explorations are possible), or the search terminates before their turns come in the exploration process. The nodes in light color are not even generated by the search. The search stops as soon as the leaf node labeled $1, 2, 4, 3$ is discovered.

Figure 130 shows a DFS traversal of the non-deterministic computation tree of Figure 128. Notice that a DFS traversal exhibits the potential of reaching (some) leaf nodes quickly without generating all the non-leaf nodes in the tree. In view of this, a DFS traversal is often preferred to a BFS traversal.

Figure 131 describes a traversal of the non-deterministic computation tree of Figure 128 with Q organized as a min-heap. The heap ordering is with respect to the current cost incurred in the search path. The shaded nodes are explored in the increasing sequence of these cost values $(0, 5, 6, 7, 9)$.

Figure 129: BFS traversal of the tree of Figure 128

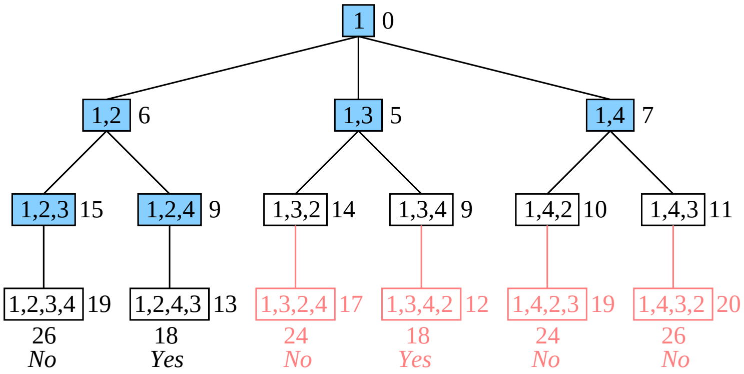


Figure 130: DFS traversal of the tree of Figure 128

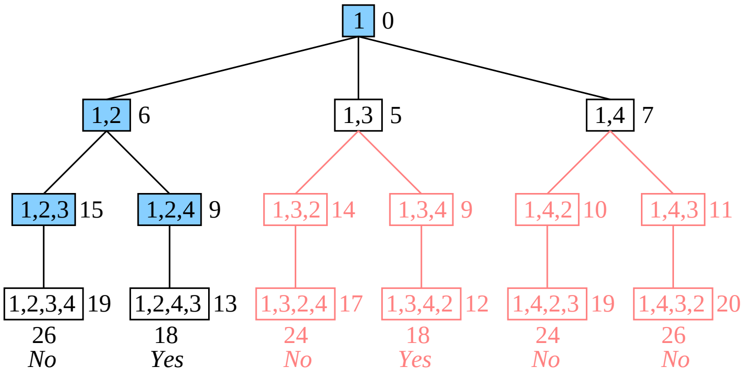
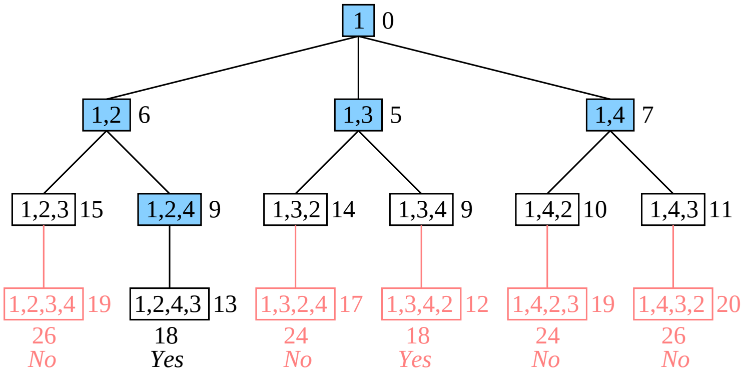


Figure 131: Heap traversal of the tree of Figure 128



A pruning heuristic is now explained in connection with the heap traversal. Such heuristics apply equally well to other types of traversal (like BFS and DFS). Instead of the cost of the current path, we maintain, against every node, a quantity that gives a reasonable *lower bound* on the set of all Hamiltonian cycles that start with the current search path. Suppose that an intermediate node is marked $v_1, v_2, \ldots, v_i$. Earlier, we have been maintaining the cost $c(v_1, v_2) + c(v_2, v_3) + \cdots + c(v_{i-1}, v_i)$ of the path $v_1, v_2, \ldots, v_i$, against this node. While this supplies a lower bound on the costs of all Hamiltonian cycles starting with $v_1, v_2, \ldots, v_i$, this estimate can be improved upon.

Let γ denote the minimum among all edge costs. A Hamiltonian cycle starting as $v_1, v_2, \ldots, v_i$ must encounter $n - i + 1$ additional edges before going back to v_1. Therefore, the cost on any Hamiltonian cycle of the form $v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_n, v_1$ cannot be less than $c(v_1, v_2) + c(v_2, v_3) + \cdots + c(v_{i-1}, v_i) + (n - i + 1)\gamma$. If this cost is already larger than k, there is no need to explore further down the tree under the node $v_1, v_2, \ldots, v_i$.

Figure 132: Heap traversal of the tree of Figure 128 with pruning

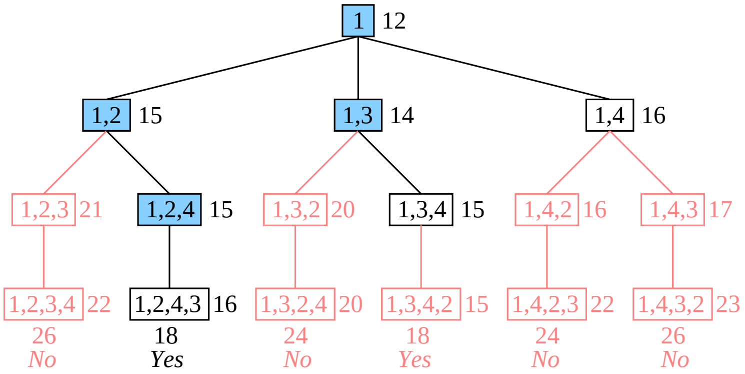


Figure 132 illustrates this pruning strategy. The heap is now kept ordered with respect to the revised cost estimate maintained against every node. For the graph shown in Figure 128, we have $\gamma = 3$. The nodes 1 (with cost $0 + 4 \times 3 = 12$) and $1, 3$ (with cost $5 + 3 \times 3 = 14$) are initially explored. The child $1, 3, 2$ of $1, 3$ is associated with a lower bound of $5 + 9 + 2 \times 3 = 20$ which is larger than $k = 19$, so this node is not inserted in Q. The other child $1, 3, 4$ corresponds to a lower bound of $5 + 4 + 2 \times 3 = 15 \leqslant k$, and is inserted in Q.

Now, there is a tie between the nodes $1, 2$ (with cost $6 + 3 \times 3 = 15$) and $1, 3, 4$ (with cost $5 + 4 + 2 \times 3 = 15$). Let us first explore $1, 2$. We have two possibilities: $1, 2, 3$ and $1, 2, 4$. A lower bound on the cost associated with $1, 2, 3$ is $6 + 9 + 2 \times 3 = 21$ which is larger than $k = 19$. Consequently, the node $1, 2, 3$ is discarded (that is, not inserted in Q). The node $1, 2, 4$, on the other hand, has a cost bounded from below by $6 + 3 + 2 \times 3 = 15 \leqslant k$ and is, therefore, inserted in Q.

Now again, there is a tie between the nodes $1, 2, 4$ and $1, 3, 4$. We arbitrarily decide to explore the possibilities of extending the path $1, 2, 4$. This gives us the leaf node $1, 2, 4, 3$ corresponding to a Hamiltonian cycle of cost 18, and the search terminates.

A pruning strategy may significantly curtail the time taken by the traversal of a non-deterministic computation tree. However, unlike approximation or randomized algorithms, there is no guarantee of achieving polynomial running time (worst-case or expected). Indeed, a backtracking algorithm may take fully exponential time in the input size.

25.3.3 Branch-and-bound algorithms

Branch-and-bound algorithms are counterparts of backtracking algorithms for solving optimization problems. In a backtracking algorithm, reaching a single leaf node marked *Yes* suffices. On the contrary, reaching a single solution does not suffice for a branch-and-bound algorithm. We need to generate all the solutions for any given input, and minimize/maximize the objective function over this entire solution space.

For simplicity, let us restrict our attention to minimization problems only. We again consider a non-deterministic computation tree for a problem. Each node maintains a pair (C,D), where C is the sequence of choices needed to reach the node from the root, and D is some bookkeeping information associated with the current configuration. In addition to these values, we maintain a global value B to stand for the best (minimum) solution obtained so far. B is initialized to $+\infty$. Later, as the traversal of the tree proceeds, and new solution nodes (leaves) are discovered, B is updated if a new solution is found to be better than the best of the previous solutions. When the entire tree is traversed, B holds the desired minimum value of the objective function over all solutions.

Pruning heuristics prove to be vital for branch-and-bound algorithms too. The bookkeeping information D at any node includes (perhaps among other things) a lower bound on the objective function values over all solutions (leaf nodes) under the current node. If this lower bound is already larger than the current best solution B, then there is no need to explore further down the current node. Like backtracking algorithms, suitable pruning strategies may curtail the search time significantly.

A high-level description of a branch-and-bound algorithm follows. Depending upon the organization of the data structure Q holding discovered (but unexplored) nodes, we have different forms of tree traversal. Pruning heuristics apply equally well to any kind of traversal.

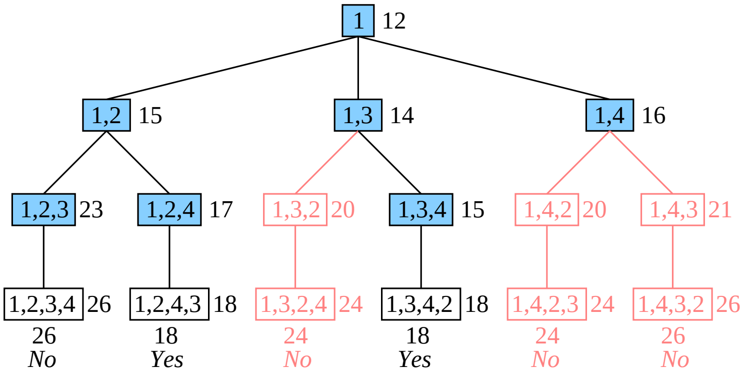
```
Initialize $B = +\infty$, and $Q = \{(\varepsilon, D_{\text{init}})\}$.
while $Q$ is not empty {
    Take a node $(C,D)$ from $Q$, and delete that node from $Q$.
    Determine all the children $(C_1,D_1),(C_2,D_2),\ldots,(C_k,D_k)$ of $(C,D)$.
    for $i = 1,2,\ldots,k$ {
        If $(C_i,D_i)$ is a leaf node {
            Compute the value $F$ of the objective function at $(C_i,D_i)$.
            If ($F < B$), replace $B$ by $F$, and remember the solution $(C_i,D_i)$.
        } else { /* $(C_i,D_i)$ is a non-leaf node */
            Compute the lower bound $L$ for the node $(C_i,D_i)$.
            If ($L < B$), add $(C_i,D_i)$ to $Q$.
        }
    }
}
Return $B$ along with the stored best solution.
```

25.3.4 A branch-and-bound algorithm for the traveling salesperson problem

Let us now turn to the optimization version of the traveling salesperson problem (TSP). As earlier, each node maintains a list $v_1, v_2, \ldots, v_i$ of some $i \leqslant n$ distinct vertices of the input graph $G = (V, E)$ (with n nodes). The pruning value used for the backtracking algorithm of TSP may be used as the lower bound L, that is, at the node $v_1, v_2, \ldots, v_i$ in the computation tree, one computes $L = c(v_1, v_2) + c(v_2, v_3) + \cdots + c(v_{i-1}, v_i) + (n - i + 1)\gamma$, where γ can be taken as the smallest edge cost in G. A better choice for γ could be the smallest edge cost among the vertices $V \setminus \{v_2, v_3, \ldots, v_{i-1}\}$, since the remaining part of any Hamiltonian cycle starting as $v_1, v_2, \ldots, v_i$ cannot involve the vertices $v_2, v_3, \ldots, v_{i-1}$. In this case, γ is dependent upon the node in the computation tree, and may provide significantly improved lower bounds.

As a specific example, we use this refined pruning heuristic on the non-deterministic computation tree of Figure 128. Assume that Q is maintained as a stack, so we are making a depth-first traversal of the tree. Suppose that nodes at the same level are explored from left to right.

Figure 133: Branch-and-bound algorithm for TSP on the graph of Figure 128



The first solution discovered in the DFS traversal is $1, 2, 3, 4$ having a cost of 26. So the current best cost B is changed from $+\infty$ to 26. Then, the better solution $1, 2, 4, 3$ is discovered, reducing the value of B to 18. When the node $1, 3$ is explored, two possibilities arise: $1, 3, 2$ and $1, 3, 4$. The former of these two has a lower bound of 20 which is already poorer than the current value of B. So the node $1, 3, 2$ is not pushed into Q. Following the second child $1, 3, 4$ leads to the solution $1, 3, 4, 2$ having the same cost as the previous best, so B continues to remain 18. Finally, the node $1, 4$ is explored. Both its children have lower bounds larger than the current B, so neither of these nodes is pushed to Q, and the search terminates.

Suitable pruning heuristics may result in significant practical speedup during the traversal of the computation tree. Nonetheless, the running time of the search may continue to remain exponential.