Abstract
An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibn Musu Al-Khowarizmi, An algorithm is a set of rules for carrying out calculation either by hand or on a machine. Ever since man invented the idea of a machine which could perform basic mathematical operations, the study of what can be computed and how it can be done well was launched. This study inspired by the computer has led to the discovery of many important algorithms and design methods. Knowledge of design will certainly help one to create good algorithms, yet without the tools of analysis there is no way to determine the quality of the result. Design of an algorithm is an art which may never be fully automated. There are several design techniques for an algorithm the greedy method is perhaps the most straightforward design technique. This survey provides a comparative overview of design and analysis of certain greedy algorithms like minimum spanning trees, Huffman coding, clustering, job sequencing with deadlines, optimal storage on tapes. Furthermore we explicitly describe and analyze the time complexities of greedy algorithms and several applications of greedy algorithms.

1. Introduction
An algorithm is a finite step-by-step procedure to achieve a required result. The most famous algorithm in history dates well before the time of the ancient Greeks: this is the Euclid's algorithm for calculating the greatest common divisor of two integers. Algorithmic is a branch of computer science that consists of designing and analyzing computer algorithms. Two important ways to characterize the effectiveness of an
algorithm are its space complexity and time complexity. The performance evaluation of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm. Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Greedy algorithms are used to solve optimization problems.

2. Greedy Algorithm Overview

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time. In a greedy algorithm, the optimal solution is built up one piece at a time. At each stage the best feasible candidate is chosen as the next piece of the solution. There is no back-tracking

2.1 Specifics:

i. A candidate set, from which a solution is created
ii. A selection function, which chooses the best candidate to be added to the solution
iii. A feasibility function that is used to determine if a candidate can be used to contribute to a solution
iv. An objective function which assigns a value to a solution, or a partial solution
v. A solution function, which will indicate when we have discovered a complete solution

2.2 Greedy Method:

It is Applicable to optimization problems ONLY. It Constructs a solution through a sequence of steps. Each step expands a partially constructed solution so far, until a complete solution to the problem is reached. On each step, the choice made must be Feasible: It has to satisfy the problem’s constraints, or Locally optimal: it has to be the best local choice among all feasible choices available on that step or Irrevocable: Once made, it cannot be changed on subsequent steps of the problem

2.3 Characteristics:

I. Greedy Choice Property: We can make whatever choice seems best at the moment and then solve the sub problems that arise later. The choice made by greedy algorithm depends on choices made so far, but not on future choices or all the solutions to the sub problem. It iteratively makes one greedy choice after another reducing each given problem into a smaller one. In other words a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming which is exhaustive and is guaranteed to find the solution
II. Optimal Substructure: A problem is said to have optimal sub structure if an optimal solution can be constructed efficiently from optimal solutions of its sub problems. Typically a greedy algorithm is used to solve a problem with optimal substructure if it can be provided by induction that this is optimal at each step.

2.4 Greedy Analysis
i. *Greedy's first step leads to an optimum solution*. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.

ii. *Greedy algorithm stays ahead*. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.

iii. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.

iv. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example minimizing lateness.

2.5 Control Abstraction Of Greedy Method:
Algorithm Greedy (a, n)
//a[1:n] contains n inputs
{
Solution:=Φ//Initialize the solution
For i:=1 to n do
{
X:=Select(a);
If Feasible (solution, x) then
Solution:=Union(solution, x);
}
Return solution;
}

3. Greedy Method Design Techniques 
Several techniques include
1. Cashier's algorithm.
2. Minimum Spanning Tree (MST)
   2.1 PRIM’S Algorithm
   2.2 Kruskal’s Algorithm
2.3 Reverse Delete algorithm

2.4 Boruvka's algorithm

3. Shortest path problem—Dijsktra’s Algorithm
4. Job sequencing with dead lines
5. Single link clustering
6. Huffman coding
7. Min-cost arborescence:
8. An activity selection problem

3.1 Cashier’s Algorithm

Coin changing Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Cashier's algorithm: At each iteration, add coin of the largest value that does not take us past the amount to be paid.

3.1.1 CASHIERS-ALGORITHM (x, c1, c2, …, cn)
SORT n coin denominations so that c1 < c2 < … < cn
S ← Φ
WHILE x > 0
k ← largest coin denomination ck such that ck ≤ x
IF no such k, RETURN "no solution"
ELSE
x ← x – ck
S ← S ∪ {k}
RETURN S

3.1.2 Analysis of cashier's algorithm

- Consider optimal way to change ck ≤ x < ck+1: greedy takes coin k.
- We claim that any optimal solution must also take coin k. If not, it needs enough coins of type c1, …, ck–1 to add up to x. Table below indicates no optimal solution can do this.
- Problem reduces to coin-changing x – ck cents, which, by induction, is optimally solved by cashier's algorithm.

3.1.3 Time Complexity:

- The time complexity would be O(n^m) where m is the amount, and n is the number of coin denominations

3.2 Minimum Spanning Trees:

SPANNING TREE: Spanning tree is a connected acyclic sub-graph (tree) of the given graph (G) that includes all of G’s vertices.

Spanning tree properties: Let T = (V, F) be a subgraph of G = (V, E).
- T is a spanning tree of G.
- T is acyclic and connected.
- T is connected and has n – 1 edges.
$T$ is acyclic and has $n - 1$ edges.
$T$ is minimally connected: removal of any edge disconnects it.
$T$ is maximally acyclic: addition of any edge creates a cycle.
$T$ has a unique simple path between every pair of nodes.

Minimum Spanning Tree (MST): MST of a weighted, connected graph $G$ is defined as:
A spanning tree of $G$ with minimum total weight. Given a connected graph $G = (V, E)$
with edge costs $ce$, an MST is a subset of the edges $T \subseteq E$ such that $T$ is a spanning tree
whose sum of edge costs is minimized

3.2.1 PRIMS ALGORITHM:
Prim’s algorithm: start at a root node and grow a spanning tree by attaching successive
least cost edges directly to the partial tree being created.

Procedure:
STEP 1: Start with a tree, $T_0$, consisting of one vertex
STEP 2: “Grow” tree one vertex/edge at a time
STEP 3: Construct a series of expanding sub-trees $T_1, T_2, \ldots T_n$. At each stage
construct $T_{i+1}$ from $T_i$ by adding the minimum weight edge connecting a vertex in tree
($T_i$) to one vertex not yet in tree, choose from “fringe” edges (this is the “greedy” step!)
STEP 4: Algorithm stops when all vertices are included

3.2.1.1 Algorithm:
ALGORITHM Prims (G)
//Prim’s algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E } 
//Output: ET the set of edges composing a MST of G
// the set of tree vertices can be initialized with any vertex
VT $\leftarrow \{ v_0 \}$
ET $\leftarrow \emptyset$
for $i \leftarrow 1$ to $|V| - 1$ do
Find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$ such that $v$ is in
VT and u is in V - VT
VT $\leftarrow$ VT U { u*}
ET $\leftarrow$ ET U { e*}
return ET

3.2.1.2 Prim’s Algorithm: Implementation
PRIMS (V, E, c)
Create an empty priority queue.
s $\leftarrow$ any node in V.
FOR EACH $v \neq s : d(v) \leftarrow \infty$; $d(s) \leftarrow 0$.
FOR EACH $v :$ insert $v$ with key $d(v)$ into priority queue.
WHILE (the priority queue is not empty)
$u \leftarrow$ delete-min from priority queue.
FOR EACH edge $(u, v) \in E$ incident to $u$:
IF $d(v) > c(u, v)$
decrease-key of v to c(u, v) in priority queue.
d(v) ← c(u, v).

3.2.1.3 Time Complexity
Efficiency of Prim’s algorithm is based on data structure used to store priority queue.
- Unordered array: Efficiency: O(n^2)
- Binary heap: Efficiency: O(m log n)
- Min-heap: For graph with n nodes and m edges: O(n + m) log n

3.2.1.4 Conclusion:
- Prim’s algorithm is a “vertex based algorithm”
- Prim’s algorithm “Needs priority queue for locating the nearest vertex.” The choice of priority queue matters in Prim implementation.
- Array - optimal for dense graphs
- Binary heap - better for sparse graphs
- Fibonacci heap - best in theory, but not in practice

3.2.2 Kruskals Algorithm:
Kruskal’s algorithm: Build a spanning tree by successively inserting edges in order of increasing cost, as long as each new added edge does not create a cycle.
Consider edges in ascending order of weight Add to tree unless it would create a cycle.

PROCEDURE:
STEP 1: Sort the edges by increasing weight
STEP 2: Start with a forest having |V| number of trees.
STEP 3: Number of trees are reduced by ONE at every inclusion of an edge
At each stage:
- Among the edges which are not yet included, select the one with minimum weight AND which does not form a cycle.
- The edge will reduce the number of trees by one by combining two trees of the forest
STEP 4: Algorithm stops when |V| -1 edges are included in the MST i.e : when the number of trees in the forest is reduced to ONE

3.2.2.1 Algorithm
ALGORITHM Kruskal (G)
//Kruskal’s algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G Sort E in ascending order of the edge weights
// initialize the set of tree edges and its size
ET ← Ø
edge_counter ← 0
//initialize the number of processed edges
k ← 0
while edge_counter < |V| - 1
3.2.2.2 Kruskal's Algorithm: Implementation

Sort edges by weight.

Use union-find data structure to dynamically maintain connected components.

Kruskal (V, E, c)

SORT m edges by weight so that c(e1) ≤ c(e2) ≤ … ≤ c(em)

S ← φ

FOREACH v ∈ V: MAKESET(v).

FOR i = 1 TO m

(u, v) ← ei

IF FINDSET(u) ≠ FINDSET(v)

S ← S U { ei }

UNION(u, v).

RETURN S

3.2.2.3 Time Complexity

Efficiency of Kruskal’s algorithm is based on the time needed for sorting the edge weights of a given graph. With an efficient sorting algorithm: Efficiency: O(|E| log |E|)

3.2.2.4 Conclusion

- Kruskal’s algorithm is an “edge based algorithm”
- Prim’s algorithm with a heap is faster than Kruskal’s algorithm.

3.2.3 Reverse Delete Algorithm:

The reverse-delete algorithm is an algorithm in graph theory used to obtain a minimum spanning tree from a given connected, edge-weighted graph. It first appeared in Kruskal (1956), but it should not be confused with Kruskal's algorithm. If the graph is disconnected, this algorithm will find a minimum spanning tree for each disconnected part of the graph. The set of these minimum spanning trees is called a minimum spanning forest, which contains every vertex in the graph.

This algorithm is a greedy algorithm, choosing the best choice given any situation. It is the reverse of Kruskal's algorithm, which is another greedy algorithm to find a minimum spanning tree. Kruskal’s algorithm starts with an empty graph and adds edges while the Reverse-Delete algorithm starts with the original graph and deletes edges from it.

Procedure:

- Start with graph G, which contains a list of edges E.
- Go through E in decreasing order of edge weights.
- For each edge, check if deleting the edge will further disconnect the graph.
- Perform any deletion that does not lead to additional disconnection.
- In this algorithm Consider edges in descending order of weight
Remove edge unless it would disconnect the graph. Consider edges in descending order of weight

Remove edge unless it would disconnect the graph. Start with the full graph (V,E) and delete edges in order of decreasing cost, as long as doing so does not disconnect the graph.

3.2.3.1 Pseudo Code:
1. Function RecursiveDelete(edges[] E)
2. Sort E in decreasing order
3. Define an index i ← 0
4. While i < size(E)
5. Define edge ← E[i]
6. Delete E[i]
7. If edge V1 is not connected to V2
8. E[i] ← edge
9. i ← i + 1
10. Return edge[] E

3.2.3.2 Time Complexity:
The algorithm can be shown to run in O(E log V (log log V)^3) time, where E is the number of edges and V is the number of vertices.

This bound is achieved as follows:
- sorting the edges by weight using a comparison sort in O(E log E) time
- E iterations of loop
- Deleting in O(1) time
- Connectivity checked in O(log V (log log V)^3) time
- Equally, the running time can be considered O(E log E (log log E)^3) because the largest E can be is V^2. Remember that log V^2 = 2 * log V, so 2 is a multiplicative constant that will be ignored in big-O notation.

3.2.4 Borůvka's Algorithm
It is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct. It was first published in 1926 by Otakar Borůvka as a method of constructing an efficient electricity network for Moravia.

PROCEDURE: The algorithm begins by first examining each vertex and adding the cheapest edge from that vertex to another in the graph, without regard to already added edges, and continues joining these groupings in a like manner until a tree spanning all vertices is completed.

3.2.4.1 Pseudo Code: Initialize a forest T to be a set of one vertex trees, one for each vertex of the graph
1. While T has more than one component
2. For each component C of T
3. Begin with an empty set of edges S
4. For each vertex v in C
5. Find the cheapest edge from v to a vertex outside of C, and add it to S
6. Add the cheapest edge in S to T
7. Output: T is the minimum spanning tree of G

3.2.4.2 Time Complexity: Boruvka's algorithm can be shown to take $O(\log V)$ iterations of the outer loop until it terminates, and therefore to run in time $O(E \log V)$, where $E$ is the number of edges, and $V$ is the number of vertices in $G$. In planar graphs, and more generally in families of graphs closed under graph minor operations, it can be made to run in linear time, by removing all but the cheapest edge between each pair of components after each stage of the algorithm.

3.2.5 MST Applications:
1. Network design. Telephone, electrical, hydraulic, TV cable, computer, road
2. Cluster analysis.
3. Reducing data storage in sequencing amino acids in a protein
4. Learning salient features for real-time face verification
5. Auto config protocol for Ethernet bridging to avoid cycles in a network, etc
6. Dithering.
7. Max bottleneck paths.
8. LDPC codes for error correction.
10. Find road networks in satellite and aerial imagery.
11. Model locality of particle interactions in turbulent fluid flows.
12. Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).

3.3 Dijkstra’s Algorithm - to find Single Source Shortest Paths

□ Shortest Path Problem: Given a connected directed graph $G$ with non-negative weights on the edges and a root vertex $r$, find for each vertex $x$, a directed path $P(x)$ from $r$ to $x$ so that the sum of the weights on the edges in the path $P(x)$ is as small as possible.

E.g.: Dijkstra's algorithm is usually the working principle behind link-state routing protocols.

Procedure:
Dijkstra’s algorithm solves the single source shortest path problem in 2 stages.
Stage 1: A greedy algorithm computes the shortest distance from source to all other nodes in the graph and saves in a data structure.
Stage 2: Uses the data structure for finding a shortest path from source to any vertex $v$.

- At each step, and for each vertex $x$, keep track of a “distance” $D(x)$ and a directed path $P(x)$ from root to vertex $x$ of length $D(x)$.
- Scan first from the root and take initial paths $P(r, x) = (r, x)$ with
  $D(x) = w(rx)$ when $rx$ is an edge,
  $D(x) = \infty$ when $rx$ is not an edge.
  For each temporary vertex $y$ distinct from $x$, set
  $D(y) = \min\{ D(y), D(x) + w(xy) \}$

3.3.1 Algorithm Dijkstra(G, s) //Input: Weighted connected graph $G$ and source vertex $s$
//Output: The length $D_v$ of a shortest path from $s$ to $v$ and its penultimate vertex $P_v$ for every vertex $v$ in $V$
//initialize vertex priority in the priority queue
Initialize ($Q$)
for every vertex $v$ in $V$ do
$D_v \leftarrow \infty$; $P_v \leftarrow \text{null}$ // $P_v$, the parent of $v$
insert($Q$, $v$, $D_v$) //initialize vertex priority in priority queue
$ds \leftarrow 0$
//update priority of $s$ with $ds$, making $ds$, the minimum
Decrease($Q$, $s$, $ds$)
$VT \leftarrow \Phi$
for $i \leftarrow 0$ to $|V| - 1$ do
$u^* \leftarrow \text{DeleteMin}(Q)$
//expanding the tree, choosing the locally best vertex
$VT \leftarrow VT \cup \{u^*\}$
for every vertex $u$ in $V - VT$ that is adjacent to $u^*$ do
if $D_{u^*} + w(u^*, u) < D_u$
$D_u \leftarrow D_u + w(u^*, u); P_u \leftarrow u^*$
Decrease($Q$, $u$, $D_u$)

3.3.2 Time Complexity:
- Use unordered array to store the priority queue: Efficiency = $\mathcal{O}(n^2)$
- Use min-heap to store the priority queue: Efficiency = $O(m \log n)$

3.3.3 Conclusion:
- Doesn’t work with negative weights
- Applicable to both undirected and directed graphs

3.3.4 Applications:
- PERT/CPM.Map, routing, Seam, carving, Robot, navigation, Texturemapping, Type setting in LaTeX, Urban traffic planning, Telemarketer operator scheduling, Routing of telecommunications messages, Network routing protocols (OSPF, BGP, RIP), Optimal truck routing through given traffic congestion pattern.

3.4 Job Sequencing With Dead Lines:
Goal: Find a feasible schedule $S$ whose profit $P(S)$ is as large as possible; we call such a schedule optimal. The setting is that we have $n$ jobs, each of which takes unit time, and a processor on which we would like to schedule them in as profitable a manner as possible. Each job has a profit associated with it, as well as a deadline; if the job is not scheduled by the deadline, then we don't get the profit.

PROCEDURE: This algorithm begins by sorting the jobs in order of decreasing (actually non increasing) profits. Then, starting with the empty schedule, it considers the jobs one at a time; if a job can be (feasibly) added, then it is added to the schedule in the latest possible (feasible) slot.

STEP 1: Sort all the jobs in decreasing order of profit
STEP 2: Initialize the result sequence as first job in sorted jobs
STEP 3: Do following for remaining n-1 jobs
- If the current job can fit in the current result sequence without missing the
deadline, add current job to the result.
- Else ignore the current job

3.4.1 Algorithm
Sort the jobs so that: g1 >= g2 >= : : : >= gn
for t : 1….n
S(t) <- 0 {Initialize array S(1); S(2); :::; S(n)}
end for
for i : 1:::n
Schedule job i in the latest possible free slot meeting its deadline;
if there is no such slot, do not schedule i.
end for

3.4.2 Time Complexity:
- The worst case computing time of job scheduling with deadline is \(0(n^2)\)
- The computing time reduced from \(0(n^2)\) to \(0(n)\) by using disjoint set union and
find algorithms

3.4.3 Applications:
1. Algorithms for scheduling independent tasks
2. Resource allocation and sequencing problems

3.5 Single Link Clustering:
Single link clustering is one of the several methods of hierarchical clustering. It is based
on grouping clusters in bottom up fashion, at each step combining two clusters that
contain the closest pair of elements not yet belonging to the same cluster as each other.
Goal. Given a set \(U\) of \(n\) objects labeled \(p1, \ldots, pn\), partition into clusters so that objects
in different clusters are far apart.

3.5.1 Greedy Clustering Algorithm
“Well-known” algorithm in science literature for single-linkage k-clustering:
Form a graph on the node set \(U\), corresponding to \(n\) clusters.
Find the closest pair of objects such that each object is in a different cluster, and add
an edge between them.
Repeat \(n – k\) times until there are exactly \(k\) clusters.

3.5.2 Greedy Clustering Algorithm Analysis:
Theorem. Let \(C^*\) denote the clustering \(C^*1, \ldots, C^*k\) formed by deleting the \(k – 1\)
longest edges of an MST. Then, \(C^*\) is a \(k\)-clustering of max spacing.
Pf. Let \(C\) denote some other clustering \(C1, \ldots, Ck.\)
- The spacing of \(C^*\) is the length \(d^*\) of the \((k – 1)st\) longest edge in MST.
- Let \(pi\) and \(pj\) be in the same cluster in \(C^*\), say \(C^*r\), but different clusters in \(C,
say Cs and Ct.\)
- Some edge \((p, q)\) on \(pi – pj\) path in \(C^*r\) spans two different clusters in \(C.\)
- Edge \((p, q)\) has length \(\leq d^*\) since it wasn’t deleted.
Spacing of $C$ is $\leq d^k$ since $p$ and $q$ are in different clusters.

3.5.3: Time Complexity:
- The time complexity of single link clustering algorithm is $O(n^2)$, where $n$ is the number of nodes.

3.5.4: Applications
- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases.
- Skycat: cluster 109 sky objects into stars, quasars, galaxies.

3.6 Huffman Trees
The Huffman coding is a greedy method for obtaining an optimal prefix-free binary code, which uses the following idea: For each $i$, $1 \leq i \leq n$, create a leaf node $v_i$ corresponding to $a_i$ having frequency $f_i$. Let $D = \{v_1; \ldots; v_n\}$. Repeat the following until $\text{Mod } D = 1$.

- Select from $D$ the two nodes with the lowest frequencies. Call them $x$ and $y$.
- Create a node $z$ having $x$ as the left child and $y$ as the right child.
- Set $f[z] = f[x] + f[y]$.
- Remove $x$ and $y$ from $D$ and add $z$ to $D$.

Terminologies:
- Code word: Encoding a text that comprises $n$ characters from some alphabet by assigning to each of the text’s characters some sequence of bits. This bits sequence is called code word.
- Fixed length encoding: Assigns to each character a bit string of the same length.
- Variable length encoding: Assigns code words of different lengths to different characters.

Procedure:
Step 1: Initialize $n$ one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree’s root to indicate the tree’s weight. (More generally the weight of a tree will be equal to the sum of the frequencies in the tree’s leaves)
Step 2: Repeat the following operation until a single tree is obtained. “Find two trees with smallest weight. Make them the left and right sub-tree of a new tree and record the sum of their weights in the root of the new tree as its weight”

3.6.1 Huffman Algorithm:
Huffman algorithm: Constructs binary prefix code tree. Huffman’s algorithm achieves data compression by finding the best variable length binary encoding scheme for the symbols that occur in the file to be compressed. Huffman coding uses frequencies of the symbols in the string to build a variable rate prefix code:
- Each symbol is mapped to a binary string.
- More frequent symbols have shorter codes.
- No code is a prefix of another code.
- Huffman Codes for Data Compression achieves 20-90% Compression.
Algorithm Huffman(C)
1. n = |C|
2. Q = C
3. for i = 1 to n -1
4. Allocate a new node z
5. z.left = x = Extract-Min(Q)
6. z.right = y = Extract-Min(Q)
7. z.freq = x.freq + y.freq
8. Insert(Q, z)
9. return Extract-Min(Q) //return the root of the tree

3.6.2 TIME COMPLEXITY: O(n log n).
- Extract-Min(Q) needs O(log n) by a heap operation.

3.6.2 Applications:
1. Huffman coding today is often used as a back end to some other compression methods
2. It is used in multimedia codecs such as JPEG, and MP3

3.7 Min-Cost Arborescence’s:
- Given a digraph G = (V, E) and a root r ∈ V, an arborescence (rooted at r) is a subgraph T = (V, F) such that
  - T is a spanning tree of G if we ignore the direction of edges.
  - There is a directed path in T from r to each other node v ∈ V.

3.7.1 Algorithm:
Given a digraph G, find arborescence rooted at r (if one exists).
BFS or DFS from r is an arborescence (iff all nodes reachable).
A subgraph T = (V, F) of G is an arborescence rooted at r iff T has no directed cycles and each node v ≠ r has exactly one entering edge.
If T is an arborescence, then no (directed) cycles and every node v ≠ r has exactly one entering edge—the last edge on the unique r↝v path.
Suppose T has no cycles and each node v ≠ r has one entering edge.
  To construct an r↝v path, start at v and repeatedly follow edges in the backward direction.
  Since T has no directed cycles, the process must terminate.
  It must terminate at r since r is the only node with no entering edge

3.7.2 Time Complexity
- The time complexity of this algorithm is O(nm)

3.8 An Activity-Selection Problem:
It is a combinatorial optimization problem concerning the selection of non conflicting activities to perform within a given time frame,given a set of activities each marked by a start time (s_i) and finish time (f_i).
PROCEDURE: A greedy algorithm always makes the choice that looks best at the moment.
Given a set \( S = \{1, 2, \ldots, n\} \) of \( n \) proposed activities, with a start time \( s_i \) and a finish time \( f_i \) for each activity \( i \), select a maximum-size set of mutually compatible activities.

- If selected, activity \( i \) takes place during the half-open time interval \([s_i, f_i)\).
- Activities \( i \) and \( j \) are compatible if \([s_i, f_i) \) and \([s_j, f_j) \) do not overlap (i.e., \( s_i \geq f_j \) or \( s_j \geq f_i \)).

**ACTIVITY SELECTION:**

### 3.8.1 The Activity-Selection Algorithm

#### Greedy-Activity-Selector(s,f)

```plaintext
// Assume f1 <= f2 <= \ldots <= fn.
1. n = s.length
2. A = \{1\}
3. j = 1
4. for i = 2 to n
5. if si >= fj
6. A = A \cup \{i\}
7. j = i
8. return A
```

**3.8.2 Theorem:** Algorithm Greedy-Activity-Selector produces solutions of maximum size for the activity-selection problem.

1. (Greedy-choice property) Suppose \( A \subseteq S \) is an optimal solution. Show that if the first activity in \( A \) is not equal to 1, then \( B = A - \{k\} \cup \{1\} \) is an optimal solution.
2. (Optimal substructure) Show that if \( A \) is an optimal solution to \( S \), then \( A' = A - \{1\} \) is an optimal solution to \( S' = \{i \in S: si \geq f1\} \).

**Optimality Proofs:**

(Greedy-choice property) Suppose \( A \subseteq S \) is an optimal solution. Show that if the first activity in \( A \) is not equal to 1, then \( B = A - \{k\} \cup \{1\} \) is an optimal solution.

(Optimal substructure) Show that if \( A \) is an optimal solution to \( S \), then \( A' = A - \{1\} \) is an optimal solution to \( S' = \{i \in S: si \geq f1\} \).
3.8.3 Time Complexity:
- The time complexity of activity selection algorithm is $O(n^2)$

3.8.4: Applications:
1. A classic application of this problem is in scheduling a room for multiple competing events, each having its own time requirements
2. These are used in within the frame work of operations research.

4 Pros And Cons Of Greedy Algorithms

4.1 Pros:
(A) Usually (too) easy to design greedy algorithms
(B) Easy to implement and often run fast since they are simple
(C) Several important cases where they are effective/optimal
(D) Lead to a worst-cut heuristic when problem not well understood

4.2 Cons:
(A) Very often greedy algorithms don't work. Easy to lull oneself into believing they work.
(B) Many greedy algorithms possible for a problem and no structured way to find effectiveness.

5 Conclusions
Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum. If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, and the algorithm for finding optimum Huffman trees. The theory of matroids, and the more general theory of greedoids, provide whole classes of such algorithms. In the Macintosh computer game Crystal Quest the objective is to collect crystals, in a fashion similar to the travelling salesman problem. The game has a demo mode, where the game uses a greedy algorithm to go to every crystal. The artificial intelligence does not account for obstacles, so the demo mode often ends quickly. The matching pursuit is an example of greedy algorithm applied on signal approximation.

6. References:
[9] Introduction to Algorithms (Cormen, Leiserson, Rivest, and Stein) 2001, Chapter 16 "Greedy Algorithms".

Author’s Profile:
Ms AKELLA.SUBHADRA Born In Amalapuram, AP State, India. Received Master Of Computer Applications From Andhra University, Vizag. Received Master Of Technology In Computer Science From Nagarjuna University-Guntur And Received Master Of Philosophy In Computer Science From Periyar University, Salem. She Is Presently Associate Professor In Bonam Venkata Chalamayya Institute Of Technology & Science, Batlapalem, Amalapuram. She Is Having 10 Years Of Teaching Experience. Her Areas Of Interests Are Design And Analysis Of Algorithms, Information Security, Compiler Design, Data Mining, Computer Networks And. Formal Languages And Automata Theory, She Is Member Of ISTE And Computer Society Of India.