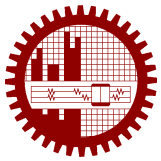


Bangladesh University of Science and Technology (BUET)
Computer Science and Engineering (CSE)

Algorithm Engineering Sessional

Course: CSE 462

A Comprehensive Study on the Maximum Leaf Spanning Tree Problem

Submitted by:

1805091 - Sk Ruhul Azgor, 1805032 - Mahdee Mushfique Kamal, 1805116 - Sanjida Islam Era, 1805045 - Md. Hasanul Islam, 1805016 - Ahmed Hossain

26.01.2024

Contents

1 Introduction

Spanning trees, fundamental to graph theory, connect all vertices of a graph without forming cycles. Among the various problems in this domain, the Maximum Leaf Spanning Tree (MaxLST) problem stands out due to its aim to maximize the number of leaf vertices in a spanning tree. This optimization is crucial for enhancing network coverage and connectivity, finding applications in areas ranging from communication networks to environmental monitoring.

1.1 Problem Definition

The MaxLST problem is formally defined as follows:

- **Given:** A connected graph G and an integer k.
- **Parameter:** An integer k.
- **Question:** Does G possess a spanning tree with at least k leaves?

This parameterized problem highlights the challenge of optimizing leaf vertices to achieve broader network coverage and efficiency, particularly valuable in wireless sensor networks and urban planning.

1.2 Computational Complexity and Significance

Despite its NP-hardness, which signifies the computational challenge in finding optimal solutions, the decision version of the MaxLST problem is in NP. This distinction emphasizes the balance between the complexity of the problem and the feasibility of solution verification.

- The pursuit of efficient algorithms and heuristics for solving the MaxLST problem represents a significant intersection of theoretical interest and practical application.
- The inherent challenges posed by its computational classification are countered with a dedicated examination to establish the NP-hardness of the MaxLST problem, further highlighting the complexities involved.

1.3 Research Objectives and Directions

Our primary aim is to address the MaxLST problem's intractability, exploring various strategies including exact solutions, approximation techniques, and heuristic/meta-heuristic algorithms,

as documented in existing literature. Moreover, the practical applications of the MaxLST problem will be showcased, underlining its relevance and utility in real-world scenarios.
In conclusion, this report proposes future research directions to further the understanding and solutions of the MaxLST problem, aiming for a balance between theoretical exploration and practical applicability.

2 Reductions and Polynomial Solvable Variants

2.1 NP-hardness of Max-Leaf Spanning Tree

A problem H is considered NP-hard if any problem in NP can be transformed to H in polynomial time [1]. Which means an NP-hard problem H is at least as hard as the hardest problems in NP. The Boolean satisfiability problem (SAT) was the first established NP-hard problem by Cook in 1971 [2]. Subsequently, Karp's work in 1972 identified 21 other fundamental NP-hard problems, including Maximum Clique, Minimum Vertex Cover, and Minimum Set Cover, etc [3].
To proof the NP-hardness of a problem H, we need to reduce a well known existing NP-hard problem to it. In this case to proof the NP-hardness of Max-Leaf Spanning Tree, we first reduce the Minimum Vertex Cover problem to Minimum Connected Dominating Set (MinCDS) problem and then reduce Minimum Connected Dominating Set to Max-Leaf Spanning Tree problem. [4]

2.1.1 Minimum Vertex Cover Problem

A vertex in a graph covers its incident edges. The vertex cover problem aims to identify a minimal subset of vertices that covers all the edges in the graph.
In a graph $G = (V, E)$, a set S is termed a vertex cover if and only if $S \subset V$ and for all edges $(u, v) \in E$, either $u \in S$ or $v \in S$. Formally, the Vertex Cover problem seeks to find a vertex cover S of minimum size, that is, to minimize $|S|$. The decision version of vertex cover problem is as followed:
Instance: A connected graph G = (V , E) and an integer k
Question: Does G have a vertex cover of size at most k?

2.1.2 Minimum Connected Dominating Set Problem

A vertex dominates itself and its neighbors in a graph. The minimum connected dominating set problem aims to find a minimal subset of vertices such that all vertices in the graph are connected and are dominated by at least one vertex in that subset. In a connected graph $G = (V, E)$, a connected dominating set $S \subseteq V$ satisfies the property that for every vertex $u \in V$, either $u \in S$ or u has a neighbor $v \in S$, and the induced subgraph of G by S is connected. Formally, the problem seeks to find a connected dominating set S of minimum size, i.e., to minimize $|S|$. The decision version of connected dominating set problem is as followed:
Instance: A connected graph G = (V , E) and an integer k
Question: Does G have a connected dominating set of size at most k?

2.1.3 Reducing Vertex Cover to Minimum Connected Dominating Set problem

Given an instance of Vertex Cover problem with graph G, we construct an instance of MinCDS problem with graph G' as follows:

- G' has all edges and vertices of G.
- For every edge $(u, v) \in G$, we add intermediate node on a parallel path in G', called w. Keeping (u, v) intact in G' , we add vertex w and edges (u, w) and (w, v) in G'.
- For every $(u, v) \notin G$, we add edge (u, v) in G'.

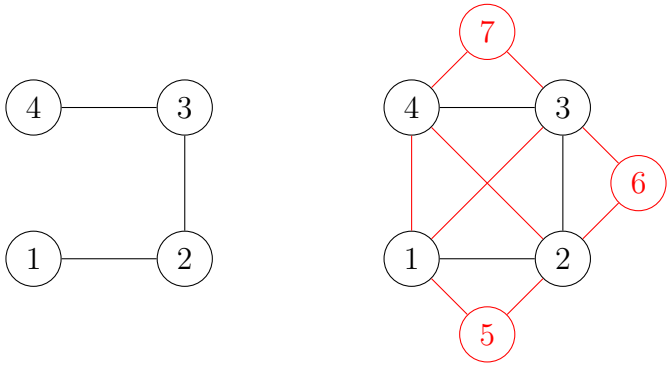


Figure 2.1: The black nodes and edges are from G and the red nodes and edges are newly added edges on G'

It can be proven that, G has a vertex cover of size at least k if and only if G' has a connected dominating set of size at least k.

Necessity: For each node s in vertex cover S, s dominated itself and all the nodes of $V - S$ of G. The newly created nodes on G' corresponds to an edge in G which is covered by a node s from the vertex cover set S. So, we will need at least k nodes to dominate G'. Also, Now the nodes of G forms a clique because of the newly added missing edges, it is connected. So, if G has a vertex cover of size at least k, G' also has a connected dominating set of size at least k.

Sufficiency: Each newly created node w in G' dominates itself and its two neighbours u and v. So, the set of newly created nodes in G' is a dominating set. If we closely observe the 3 nodes u, v and w, we can see that they form a clique. Here, u dominates w and v. Also, v dominates w and u. So, all the newly created nodes which forms the dominating set can be replaces by either of its two neighbours. Each newly created node w corresponds to an edge $(u, v) \in G$. For all w, its neighbour u or v can also correspond to an edge. So, for all w, either u or v is in vertex cover. So, if G' has a connected dominating set of size at least k, G also has a vertex

cover of size at least k.

So, we can reduce a vertex cover problem to a mininum connected dominating set in polynomial time. MinCDS problem is also an NP-Hard problem.

2.1.4 Interacttibility of MaxLSTP

To show *MaxLSTP* to be *NP-hard*, we will show *MinCDS* $\leq_P$ *MaxLST*. Proving the following lemma is sufficient.

Lemma 2.1.1. *A connected graph $G = (V, E)$ has a connected dominating set of size at most k if and only if it has a spanning tree of at least $n - k$ leaves.*

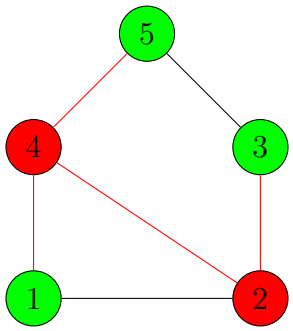


Figure 2.2: The MaxLST problem is equivalent to the MinCDS problem. In the given graph, the red vertices form a CDS. The red edges form a MaxLST, where the green vertices are leaves and the red vertices are internal nodes.

Proof. Necessity: Let $S \subset V$ be a connected dominating set of G and $|S| \leq k$. Let T be a connected subgraph of G with $V(T) = S$ of $|S| - 1$ edges. For each vertex $v \in V - S$, there exists a vertex $u \in S$. We add the vertex u and an edge (u, v) to T. Now T is a connected subgraph with $|V|$ vertices and $|V| - 1$ edges. That is, T is a spanning tree. Every vertex in $V - S$ is a leaf of T. Therefore, $|V - S| \geq n - k$.
Sufficiency: Let T be a spanning tree of V with at least $n - k$ leaves. We claim that the set $S = V - \text{Leaves}(T)$, which is connected and of size at most $n - (n - k) = k$ leaves, is a dominating set of G. Every vertex v is either a leaf of T and thus connected to a vertex in S, or it is not a leaf of T and thus belongs to S. The claim follows. □

2.2 Polynomially Solvable Variants of MaxLST

In certain cases, relaxing the constraints of a computationally hard problem may lead to a variant that is solvable in polynomial time. Considering the Maximum Leaf Spanning Tree problem, we turn our attention to a particular variant which we refer to as SetMaxLST.

2.2.1 SetMaxLST Problem Definition

- **Instance:** Given a connected graph $G = (V, E)$ and a subset $X \subseteq V$.
- **Question:** Is there a spanning tree T of G such that $X \subseteq \Pi_T$, where Π_T denotes the set of leaf vertices in T?

2.2.2 Characterization and Algorithm

We present a characterization offering a polynomial-time constructive algorithm for this problem.

Lemma: Let $G = (V, E)$ be a connected graph, $X \mid Y$ be a partition of V. Then there exists a spanning tree T such that $X \subseteq \Pi_T$, if and only if the subgraph G_Y is connected and every X-node has an adjacent node in Y.

We propose the following algorithm based on a depth-first search (DFS) strategy to construct the required spanning tree:

Algorithm 1 An algorithm to solve the set version of MaxLST

Data: A connected graph $G = (V, E)$ and $X \mid Y$ be a partition of V
Take a vertex $y \in Y$;
Call DFS from y to construct a tree T that spans Y;
for $x \in X$ **do**
  Find a neighbor $v \in Y$ of x;
  Append edge (x, v) to T;
end for
return T
=0

The provided algorithm efficaciously solves the SetMaxLST problem in polynomial time as demonstrated in the referenced literature [5].

3 Exact Algorithm

An exact algorithm is a computational method designed to find an optimal solution to a problem by exhaustively examining all possible solutions within a feasible set, ensuring no better solution exists. It guarantees finding the best solution but may be computationally intensive for large problem instances. Naive brute force $\Omega(2^n)$ algorithm exist that solves maxLST problem. Later many algorithm developed to improve the base of the exponential runtime. **Formin et el.** [6] gave an $\mathcal{O}^*(1.9407^n)$ algorithm to solve this problem. Later **Fernau et el.** [7] improved this to $\mathcal{O}^*(1.8966^n)$. We studied the exact algorithm developed by **Fernau et el.** [7] which has a worst case runtime complexity of $\mathcal{O}^*(1.8966^n)$.

3.1 The Algorithm

The basic idea of the algorithm is to build a subtree of the graph while repeatedly branching of the leaves until a spanning tree is found. The algorithm chooses the node with maximum degree and branch off depending on the number of unprocessed neighbour of the node. A branch of the algorithm terminates when a spanning tree is found or some node becomes unreachable. When all branches terminates the algorithm returns the maximum leaf spanning tree. The algorithm maintains 5 disjoint set of vertices.

IN(Internal Nodes): Internal Nodes are nodes that were decided to be an internal node of the spanning tree in the course of the algorithm.

LN (Leaf Nodes): Leaf nodes of the current spanning tree that have no more neighbors to process.

BN (Branching Nodes): Leaf nodes of the current spanning tree that still have some neighbors to process.

FL (Floating Leaves): Nodes in the original graphs designated to be leaves in the final spanning tree, but has not yet been attached to the current spanning tree.

Free (Free Nodes):All other nodes

The algorithm reduces the size of the graph and updates the partition it maintains before branching off each time using the following 7 rules.

R1:If there exist two adjacent vertices $u, v \in V$ such that $u, v \in FL$ or $u, v \in BN$, then remove the edge u, v from G.

R2:If there exists a node v in BN with $d(v) = 0$, then move v into LN.

R3:If there exists a free vertex v with $d(v) = 1$, then move v into FL.

R4:If there exists a free vertex v with no neighbors in $Free \cup$ FL, then move v into FL.

R5:If there exists a triangle $\{x, y, z\}$ in G with x a free vertex and $d(x) = 2$, then move x into FL.

R6:If there exists a node $u \in$ BN which is a cut vertex in G, then apply rule $u \to$ IN.
R7:If there exist two adjacent vertices $u, v \in V$ such that $u \in$ LN and $v \in V \setminus$ IN, then remove the edge $\{u, v\}$ from G.

The key idea of the algorithm is to recursively build a subtree $T = (V_T, E_T) \subseteq G$ with $V_T = IN \cup BN \cup LN$, internal$(T) = IN$ and leaves$(T) = BN \cup LN$, which might in some branch of the search tree eventually turn into a spanning tree T' of G that extends (IN, BN, LN, FL). Vertices in LN will always remain leaves in subsequent calls. The branching nodes in BN are leaves of the current subtree T , but might be promoted to internal nodes or leaves in recursive calls of the algorithm. Vertices in $FL \subseteq V \setminus VT$ are fixed to be leaves, but they are still "floating around". This means that they have not yet been attached to T, and therefore their parent node in the solution is unknown at the moment.

The recursive construction of the spanning tree exploits the following crucial observation. Roughly speaking, if there are $v \in BN$ and an optimal spanning tree T with $v \in internal(T)$, then there also is a solution T' where the $Free \cup FL$-neighbors of v in G are children of v in T'.
Lemma: Let $G = (V, E)$ be a graph and let $IN, BN, LN, FL \subseteq V$ be disjoint sets of vertices. Let $T \subseteq G$ be a spanning tree of G that extends (IN, BN, LN, FL) and has k leaves. If there

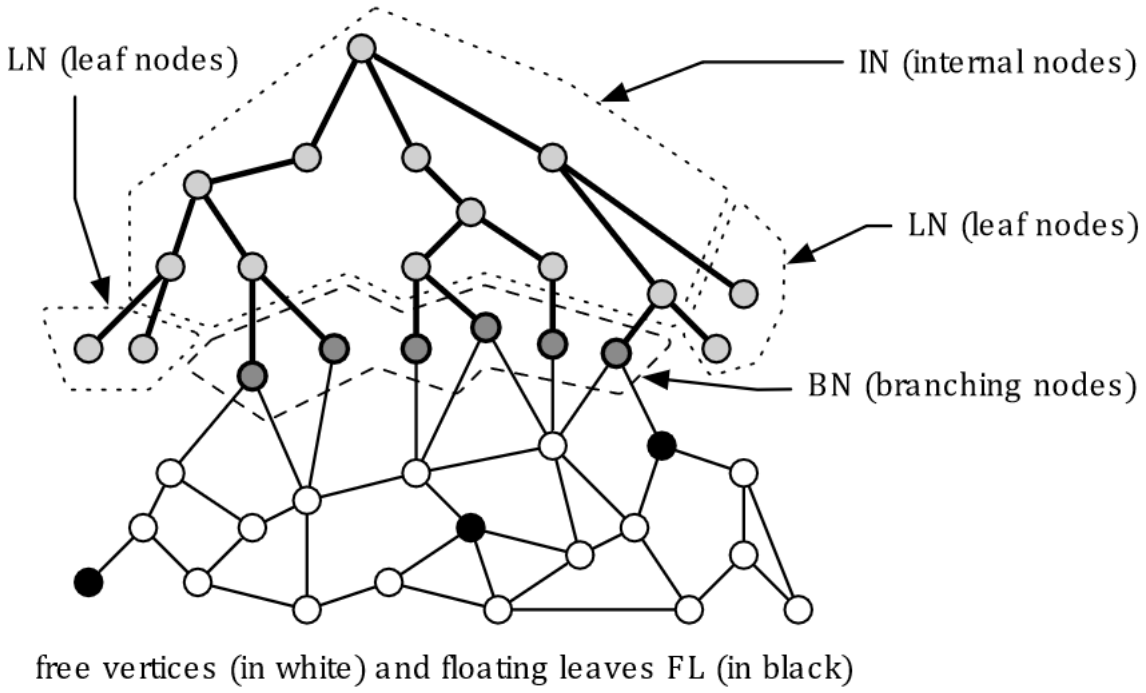


Figure 3.1: Internal Nodes

is $v \in BN$ with $v \in internal(T)$, then there also is a k-leaf spanning tree T' of G that extends $IN \cup v, (BN \setminus v) \cup N_{Free}(v) \cup N_{FL}(v), LN, FL \setminus N_{FL}(v)$.
The lemma can be proved by a simple exchange argument for the edges connecting the neighbors of v in T and T'.
The pseudo code for the algorithm is given in figure 3.2.

Algorithm $\mathcal{M}$
Input: A graph $G = (V, E)$, IN, BN, LN, FL $\subseteq V$

Reduce G according to the reduction rules.
if there is some unreachable $v \in$ Free $\cup$ FL **then** return 0
if $V =$ IN $\cup$ LN **then** return |LN|
Choose a vertex $v \in$ BN of maximum degree.
if $d(v) \geq 3$ **or** ($d(v) = 2$ and $N_{\text{FL}}(v) \neq \emptyset$) **then**
  $\langle v \to \text{LN} \mid\mid v \to \text{IN} \rangle$ (B1)
else if $d(v) = 2$ **then**
  Let $\{x_1, x_2\} = N_{\text{Free}}(v)$ such that $d(x_1) \leq d(x_2)$.
  if $d(x_1) = 2$ **then**
    Let $\{z\} = N(x_1) \setminus \{v\}$
    if $z \in$ Free **then**
      $\langle v \to \text{LN} \mid\mid v \to \text{IN}, x_1 \to \text{IN} \mid\mid v \to \text{IN}, x_1 \to \text{LN} \rangle$ (B2)
    else if $z \in$ FL **then** $\langle v \to \text{IN} \rangle$
  else if $(N(x_1) \cap N(x_2)) \setminus \text{FL} = \{v\}$ **and** $\forall z \in (N_{\text{FL}}(x_1) \cap N_{\text{FL}}(x_2))$,
    $d(z) \geq 3$ **then** (B3)
    $\langle v \to \text{LN} \mid\mid v \to \text{IN}, x_1 \to \text{IN} \mid\mid v \to \text{IN}, x_1 \to \text{LN}, x_2 \to \text{IN} \mid\mid$
     $v \to \text{IN}, x_1 \to \text{LN}, x_2 \to \text{LN}, \quad N_{\text{Free}}(\{x_1, x_2\}) \to \text{FL}, N_{\text{BN}}(\{x_1, x_2\}) \setminus \{v\} \to \text{LN} \rangle$
  else $\langle v \to \text{LN} \mid\mid v \to \text{IN}, x_1 \to \text{IN} \mid\mid v \to \text{IN}, x_1 \to \text{LN}, x_2 \to \text{IN} \rangle$ (B4)
else if $d(v) = 1$ **then**
  Let $P = (v = v_0, v_1, \ldots, v_k)$ be a maximum path such that
    $d(v_i) = 2, 1 \leq i \leq k, v_1, \ldots, v_k \in$ Free.
  Let $z \in N(v_k) \setminus V(P)$.
  if $z \in$ FL **and** $d(z) = 1$ **then** $\langle v_0, \ldots, v_k \to \text{IN}, z \to \text{LN} \rangle$
  else if $z \in$ FL **and** $d(z) > 1$ **then** $\langle v_0, \ldots, v_{k-1} \to \text{IN}, v_k \to \text{LN} \rangle$
  else if $z \in$ BN **then** $\langle v \to \text{LN} \rangle$
  else if $z \in$ Free **then** $\langle v_0, \ldots, v_k \to \text{IN}, z \to \text{IN} \mid\mid v \to \text{LN} \rangle$ (B5)

Fig. 2. An algorithm for Maximum Leaf Spanning Tree. The notation $\langle v \to \text{IN} \mid\mid v \to \text{LN} \rangle$ describes the corresponding branches, e.g., in this case v either becomes an internal node or a leaf (see Definition 2). To compute a solution for Maximum Leaf Spanning Tree of a given graph $G = (V, E)$, algorithm $\mathcal{M}$ must be called for each vertex $v \in V$ with IN $= \{v\}$, BN $= N(v)$ and LN $=$ FL $= \emptyset$ as stated in Lemma 4.

Figure 3.2: Internal Nodes

We have implemented and tested the algorithm in $C{+}{+}$. Our implementation can be found publicly at `https://github.com/skazgor/maxLST/blob/main/code/exact_exponential_algorithm.cpp`.

3.2 Runtime Analysis

The measure-and-conquer approach is used to determine algorithm M's runtime. We suppose that the computation performed in each node of a branching algorithm takes polynomial time.

Therefore, the overall runtime is equal to the product of a polynomial and the number of nodes in the search tree. A measure-and-conquer strategy involves introducing a complex measure and monitoring its decline at each branching step. The measure used by Fernau et al is:

$$\mu(I) = \sum_{i=1}^{n-1} \epsilon_i^{BN} |BN_i| + \sum_{i=2}^{n-1} \epsilon_i^{FL} |FL_i| + \sum_{i=2}^{n-1} \epsilon_i^{Free} |Free_i|$$

For $X \in \{BN, FL, Free\}$

- X_i denoted the vertices in X of degree i.
- $\epsilon_i^X \in [0, 1)$ are predefined constants.

We go over the branching rule (B1) to show how the bound is computed.

- When $v \to LN$
  - v becomes a leaf node
  - Each vertex in $N_{Free \cup FL}(v)$ loses degree by 1 by (R7).
  - A decrease in measure by at least

$$\Delta_1 = \epsilon^{BN}_{\deg(v)} + \sum_{x \in N_{Free}(v)} (\epsilon^{Free}_{\deg(x)} - \epsilon^{Free}_{\deg(x)-1}) + \sum_{x \in N_{Free}(v)} (\epsilon^{FL}_{\deg(x)} - \epsilon^{FL}_{\deg(x)-1})$$

- When $v \to IN$
  - v becomes an internal node
  - Each vertex in $N_{Free}(v)$ becomes branching nodes
  - A decrease in measure by at least

$$\Delta_2 = \epsilon^{BN}_{\deg(v)} + \sum_{x \in N_{Free}(v)} (\epsilon^{Free}_{\deg(x)} - \epsilon^{BN}_{\deg(x)-1}) + \sum_{x \in N_{Free}(v)} (\epsilon^{FL}_{\deg(x)})$$

We form a recurrence relation

$$T(x) \leq T(x - \Delta_1) + T(x - \Delta_2)$$

We determine the values of Δ_i through a thorough case-by-case analysis. Typically, computer simulations are employed to ascertain the specific values of ϵ_i and Δ_i.

Finally, the authors have demonstrated that algorithm M operates within a time complexity of $O^*(1.8966^n)$.

4 2-approximation Algorithm

Approximation algorithms are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with provable guarantees on the distance of the returned solution to the optimal one. Lu and Ravi [8] gave the first known polynomial time approximation algorithms with approximation factors 5 and 3. Later, they obtained a 3-approximation algorithm that runs in near-linear time [9]. In this project we studied the algorithm proposed by **Cheng et al** [10]. The algorithm finds a spanning tree having number of leaves no less the half of actual maximum number of leaves. The time complexity of the algorithm is linear in terms of the total number of vertices and edges of the graph. The specialty of the algorithm is its simplicity of implementation.

4.1 The algorithm

The algorithm maintains a spanning tree of the given graph. The spanning tree grows by branching out of its leaves and terminates when all vertices of the graph has been included. The algorithm maintains 3 classes of leaves of the spanning tree.
W2: The leaves of the spanning tree that can expand to more than 1 branch.
W1: The leaves of the spanning tree that can expand to exactly 1 branch but the leaf that has been expanded to can expand to more than 1 branch in the next iteration.
W0: Rest of the leaves.
The algorithm tries of expand leaves from these classes with **W2** given the highest priority and **W0** given the lowest Priority.

4.2 Implementation

The basic idea of the implementation is the following:

- We maintain 3 linked lists corresponding to $W2$, $W1$, $W0$.
- We will also maintain an array which will maintain the current state of each vertex(i.e. whether the vertex is an internal node, a leaf or is not added yet.
- Primarily we try to pick a vertex from $W2$. If not succeeded, we try out $W1$ and $W0$ one after another.
- When picking up a vertex from a linked list, we check whether the vertex really belongs to that linked list. If not, we remove that vertex from that linked list and add it to the

linked list immediate below in terms of priority i.e if the vertex previously belonged to $W2$, now it belongs to $W1$.

- When a vertex is expanded to, we add it to $W2$.

The pseudo code for the algorithm is given in figure 6.1. We have implemented and tested the algorithm in $C++$. Our implementation can be found publicly at `https://github.com/skazgor/maxLST/blob/main/code/2-approx.cpp`.

4.3 Runtime Analysis

- Each vertex u is push to $W2$ at most $deg(u)$ times. We also maintained an array which has the size of the total number of nodes. Thus we are using $O(n+m)$ memory.
- When choosing an vertex, we either pick it or downgrade it.
- Each pushed vertex is downgraded at most 3 times, thus incurring an $O(m)$ runtime where m is the total number of edges.

Algorithm tree(G):
  Let T be the initial tree of G consisting of a vertex a of G with $d_G(a) \geq 2$.
  Repeat the following steps until $V(T) = V(G)$:
    If $W_2(T) \neq \varnothing$, then
      let u be an arbitrary vertex in $W_2(T)$
    else if $W_1(T) \neq \varnothing$, then
      let u be an arbitrary vertex in $W_1(T)$
    else
      let u be the vertex in $W_0(T)$ that joins $V(T)$ most recently.
    Expand T at u by letting $T = T \cup E_T(u)$.
  Return T.

Figure 4.1: 2-approximation algorithm

5 Heuristics and Metaheuristics

Heuristics are problem-specific techniques that enhance the performance of an algorithm. Metaheuristics, on the other hand, are general problem-solving techniques that can be applied to a large class of hard optimization problems. Heuristics and metaheuristics often highly enhance the practical usability of a problem. But we cannot theoretically ensure the worst-case performance of such techniques. We briefly review a some heuristics and metaheuristics approach for our problem in this chapter. Each approaches were implemented by their respective authors and they established their methods' efficacy through rigorous empirical studies. We also recreate some of the results for a heuristic (Priority-BFS) and a metaheutistic (Simulated Anneling) in our own experiment which reflects the claim of the respective authors. The code and results can be found at https://github.com/skazgor/maxLST/tree/main/code.

5.1 Heuristics for MaxLSTP

Constructive heuristics, such as the Breadth-First-Search (BFS) traversal, provide efficient solutions for the Maximum Leaf Spanning Tree Problem (MLSTP) in random and regular graphs [11]. However, their performance diminishes significantly in grid graphs compared to approximation algorithms by Lu-Ravi and Solis-Oba [12] [11]. Priority-BFS (PBFS), a new heuristic, outperforms BFS, Lu-Ravi, and Solis-Oba on grid graphs. It offers fast and high-quality solutions, useful for initializing lower bounds and metaheuristic algorithms [11]. The key idea behind Priority-BFS is to prioritize nodes with the highest number of neighbors outside the current tree during the tree construction process. The algorithm is mentioned in algorithm-2.

5.2 Metaheristics for MaxLSTP

5.2.1 Simulated Annealing

We review the Simulated Annealing algorithm mentioned in [11]. Simulated Annealing (SA) algorithm leverages the tree obtained from the Priority-BFS heuristic as the starting configuration. In SA algorithm, a random neighboring tree T' is obtained from a given tree T using the following standard procedure:

1. Remove a random edge e_{ij} from the tree T, resulting in two disconnected components.
2. Find a random edge $(e_{i'j'}|i \neq i' \vee j \neq j')$ that connects the two components again.
3. Add $e_{i'j'}$ to the tree to obtain a new tree T'.

The cost function in the Maximum Leaf Spanning Tree Problem (MLSTP) solely counts the number of leaves in the tree. Utilizing neighborhood, it's only possible to find a neighbor with more leaves by deleting an edge that has at least one vertex of degree 2. Therefore, in fitness function, they assign a small weight to degree 2 vertices, which could potentially transform into leaves in subsequent iterations. This weight helps distinguish between the fitness of two trees with the same number of leaves.
They define fitness function as follows:

$$fit(T) = \sum_{i=0}^{n} \begin{cases} 1 & \text{if degree}(i) = 1, \\ \frac{1}{n} & \text{if degree}(i) = 2, \\ \frac{0.5}{n} & \text{if degree}(i) = 3, \\ 0 & \text{if degree}(i) > 3. \end{cases}$$

The above equation is tailored especially for grid graphs or low-degree graphs. For other types of graphs, positive weights should be considered for nodes with degrees greater than 3.
An essential aspect of the Simulated Annealing algorithm is the probability of transitioning to a worse configuration. In implementation, this probability p is defined as:

$$p = e^{\frac{a-x-1}{c}},$$

where a is the fitness of the new configuration, x is the fitness of the current configuration, and c is the current temperature.
They subtract 1 from $a - x$ in Equation (4) to prevent the algorithm from always moving to a configuration with the same fitness. Their preliminary experiments demonstrated that allowing the algorithm to transition to configurations with the same fitness led to worse results.
Initially, we set the temperature c to 1 and update $c = c \times 0.995$ at each iteration. We made c independent of the graph size because $a - x - 1$ is independent of the graph size. This decision was based on the observation that the difference between the number of leaves in the current tree and a neighboring tree falls within the range of -2 to 2. The values 1 and 0.995 were determined through experimentation.

5.2.2 A genetic Algorithm

We review the genetic algorithm formulation in [13]. At first, they constructed a mixed- integer programming model using edge-vertex-flow formulation. Each vertex i is assigned an indicator variable x_i. Our goal is to maximize $\sum x_i$ subject to the natural constraints of the problem as the total number of selected edges has to $n - 1$, the graph must be connected, etc. Then they introduced standard rules for mutation, selection and crossover. For example, in a crossover

operation, a new edge is added to the graph (between a leaf and a non-leaf node), and then an edge is chosen by a heuristic method to avoid a cycle.

5.2.3 Artificial Bee Colony Algorithm

The Artificial Bee Colony (ABC) algorithm, introduced by Karaboga and Basturk [14], is a meta-heuristic approach inspired by the behavior of honey bees, designed for numerical optimization of multivariate functions. We review the approach taken by Brüggemann, Christian and Radzik, Tomasz in here [11]. They adapt the ABC algorithm for discrete optimization problems, particularly for the task of finding maximum leaf spanning trees.
The ABC algorithm maintains a "colony" of artificial bees, each representing a computational process with a solution associated with it. There are three distinct groups of bees in the colony: Employed bees, onlooker bees, and scouts. Employed bees perform local searches around their assigned food sources, onlooker bees choose food sources based on the quality of solutions found by employed bees, and scouts generate new solutions if they cannot find better ones.
At the initialization of the algorithm, each employed bee is assigned a random "food source," which represents a solution to the optimization problem. Onlooker bees remain idle initially, while scouts are nonexistent. Onlooker bees primarily exploit promising food sources found by employed bees.
To adapt the ABC algorithm for discrete optimization, they use the same neighboring configuration as in the Simulated Annealing approach but with a different procedure for generating a neighbor. As the ABC algorithm never accepts a worse configuration, we design a neighboring function that only returns neighbors with higher fitness.
Consider a spanning tree T of a graph G. To obtain a new spanning tree T' with more leaves than T by deleting only one edge e and inserting a new edge e', we must satisfy certain conditions on e and e':

1. e must have at least one vertex of degree 2.
2. e' must have one vertex of degree ¿ 1 if e has two vertices of degree 2.
3. e' must have two vertices of degree ¿ 1 if e has precisely one vertex of degree 2.

During each iteration, an onlooker bee is assigned to a food source using a probability distribution based on the fitness of the food sources. Each bee, including onlooker and employed bees, generates a neighboring configuration. If a bee finds a better configuration, it moves to the new configuration. Otherwise, it either looks for another food source (if it's an onlooker bee) or becomes a scout (if it's an employed bee). A scout randomly generates a new food source using the Priority-BFS heuristic with a random start vertex and then becomes an employed bee again.

Since the algorithm generate only better neighboring configurations, we do not need to check certain conditions in the algorithm, leading to improved efficiency.

6 Application

The study of hard optimization problems such as the Maximum Leaf Spanning Tree (MaxLST) is often driven by their significant real-world applications, especially in various network designs. This section delves into specific applications where MaxLST plays a critical role in optimizing performance and efficiency.

6.1 Broadcasting Networks

In broadcasting networks, the objective is often to disseminate data across the network efficiently. This process involves selecting certain nodes as broadcasting points to relay the information.

- Minimizing the number of broadcasting nodes simplifies the network layout, reducing the number of connection points and leading to a more streamlined design.
- This reduction directly decreases network complexity, facilitating easier management and operation.

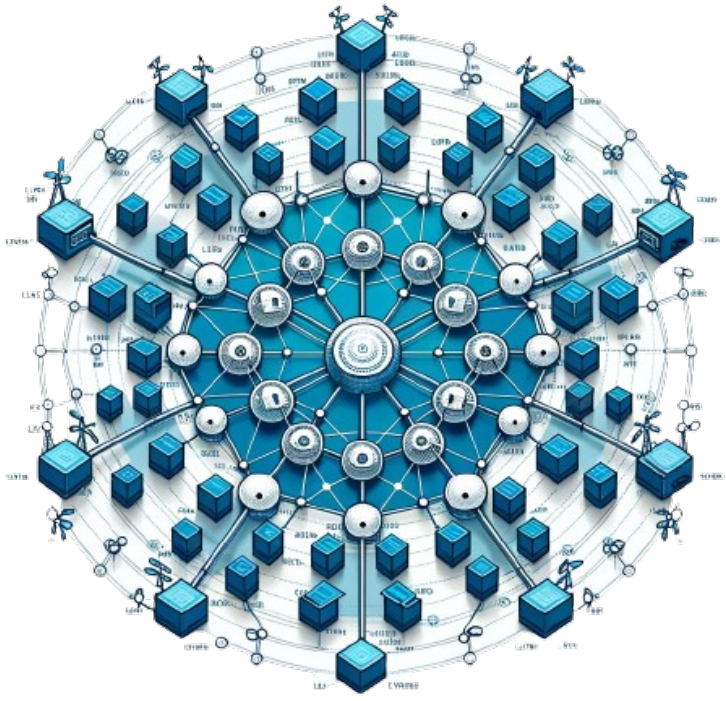


Figure 6.1: Illustration of a Broadcasting Network optimized using MaxLST.

This approach aligns with MaxLST principles, seeking configurations that maximize leaf nodes and minimize internal broadcaster nodes.

6.2 Forest Fire Detection using MaxLST

in 2022, Farvaresh et al. highlighted MaxLST's utility in optimizing a Wireless Sensor Network (WSN) for detecting forest fires in Iran's Arasbaran forest[15].

- The WSN was tailored to leverage MaxLST for optimal sensor placement, ensuring extensive coverage with minimal internal nodes.
- This approach enhances the fire detection system's efficiency, covering a vast area effectively.

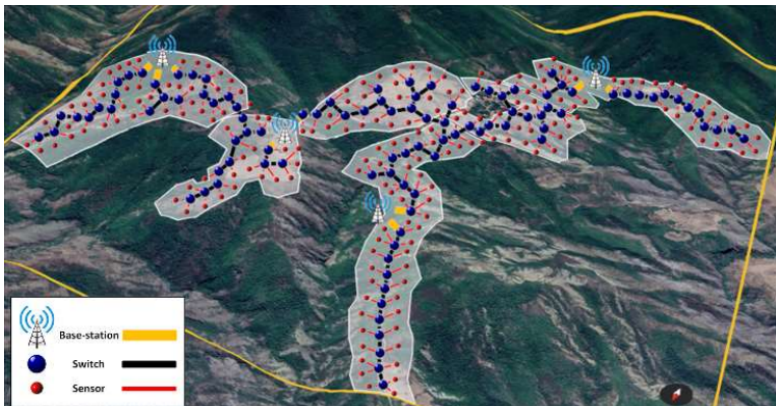


Figure 6.2: Forest Fire Detection using MaxLST in Arasbaran forest.

6.3 Circuit Layouts

MaxLST principles also apply in circuit design, aiming to minimize signal delay and optimize component layout.

- Ensuring minimal connection points and direct component connections streamlines the circuit layout.
- This optimization is crucial for reducing signal delays, enhancing circuit performance and reliability.

In each application, MaxLST emerges as a key tool for achieving efficient, streamlined designs, showcasing its broad applicability across various domains.

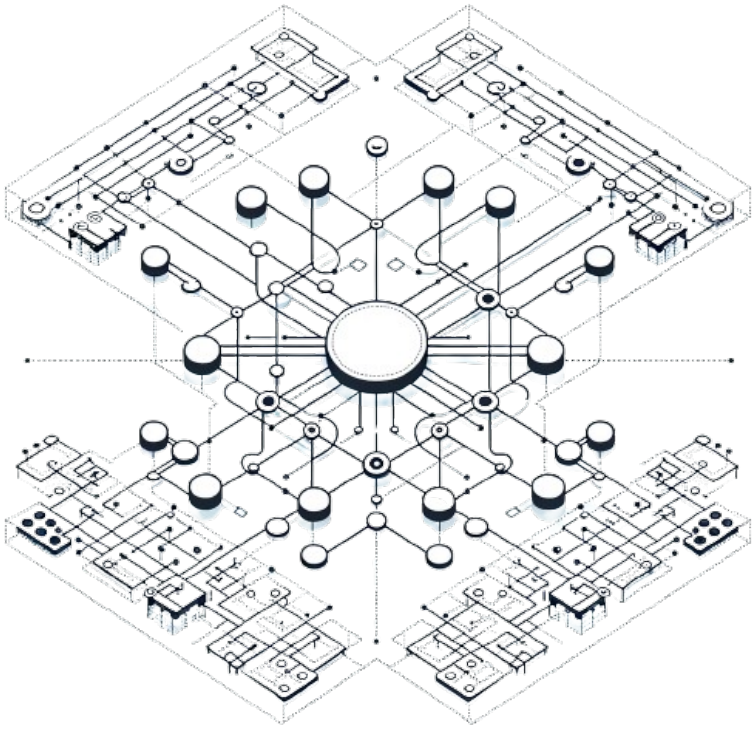


Figure 6.3: Streamlined Circuit Layout employing MaxLST principles.

7 Future Directions

We have presented a review of maximum leaf spanning tree problem. We observed throughout the report that MaxLST has been thoroughly studied in the current literature. We present a short list of open directions that can be suitable for future research.

1. The problem is MAX SNP-complete, so it is of interest to have nontrivial lower bounds on the approximation ratio.

2. As suggested by the experimental results, it is of interest to see if the approximation ratio can be improved of the 2-approximation algorithm [10].

3. We have not found any randomized algorithm in the literature for this problem. This avenue can be explored.

4. Measure-and-conquer approaches often do not give a tight upper bound on the algorithm concerned. Whether the analysis exact algorithm discussed for MaxLST can be improved is another future direction.

5. Polynomial time algorithm to find maximum leaf spanning tree for graph classes also can be studied.

Bibliography

[1] D. S. Hochba, "Approximation algorithms for np-hard problems," *ACM Sigact News*, vol. 28, no. 2, pp. 40–52, 1997.

[2] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pp. 143–152, 2023.

[3] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 2010.

[4] T. Fujie, "The maximum-leaf spanning tree problem: Formulations and facets," *Networks: An International Journal*, vol. 43, no. 4, pp. 212–223, 2004.

[5] M. S. Rahman and M. Kaykobad, "Complexities of some interesting problems on spanning trees," *Information Processing Letters*, vol. 94, pp. 93–97, April 2005.

[6] F. V. Fomin, F. Grandoni, and D. Kratsch, "Solving connected dominating set faster than 2 n," *Algorithmica*, vol. 52, pp. 153–166, 2008.

[7] H. Fernau, J. Kneis, D. Kratsch, A. Langer, M. Liedloff, D. Raible, and P. Rossmanith, "An exact algorithm for the maximum leaf spanning tree problem," *Theoretical Computer Science*, vol. 412, no. 45, pp. 6290–6302, 2011.

[8] H.-i. Lu and R. Ravi, "The power of local optimization: Approximation algorithms for maximum-leaf spanning tree," *Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control and Computing*, 02 2000.

[9] H.-I. Lu and R. Ravi, "Approximating maximum leaf spanning trees in almost linear time," *Journal of Algorithms*, vol. 29, no. 1, pp. 132–141, 1998.

[10] I.-C. Liao and H.-I. Lu, "A simple 2-approximation for maximum-leaf spanning tree," *International Journal of Foundations of Computer Science*, pp. 1–11, 2023.

[11] C. Brüggemann and T. Radzik, "Development and experimental comparison of exact and heuristic algorithms for the maximum-leaf spanning tree problem technical report tr-09-08,"

[12] R. Solis-Oba, P. Bonsma, and S. Lowski, "A 2-approximation algorithm for finding a spanning tree with maximum number of leaves," *Algorithmica*, vol. 77, no. 2, pp. 374–388, 2017.

[13] A. M. Hosseinmardi, H. Farvaresh, and I. Nakhai Kamalabadi, "A new formulation and algorithm for the maximum leaf spanning tree problem with an application in forest fire detection," *Engineering Optimization*, vol. 55, no. 7, pp. 1226–1242, 2023.

[14] D. Karaboga, "Artificial bee colony algorithm," *scholarpedia*, vol. 5, no. 3, p. 6915, 2010.

[15] A. M. Hosseinmardi, H. Farvaresh, and I. N. Kamalabadi, "A new formulation and algorithm for the maximum leaf spanning tree problem with an application in forest fire detection," *Engineering Optimization*, 2022.

8 Algorithms

8.1 Priority-BFS Algorithm

In Algorithm 2, we show the Priority-BFS algorithm proposed by Brüggemann, Christian and Radzik, Tomasz [11].

Algorithm 2 Priority-BFS

```
Choose a vertex start with maximum degree
Initialize an empty tree T
Initialize an empty priority queue Q with the leaves in T
Push (start, degree(start)) into Q
while Q is not empty do
  (i, degree) ← ExtractMax(Q)
  for each vertex v adjacent to i do
    Enqueue (v, degree_G(v) − degree_T(v)) into Q
    T ← T ∪ (i, v)
  end for
end while
return T =0
```

9 Results

9.1 Result

To evaluate the performance of 4 different algorithms (exact exponential algorithm, 2-approximation algorithm, priority BFS and simulated annealing) we generated 5 types of simple connected graph.

- Random Graph (Erdős-Rényi model)
- Scale-Free Graph (Albert-Barabási model)
- D-regular Graph
- Complete Grid Graph
- Incomplete Grid Graph

9.1.1 Run Time Analysis

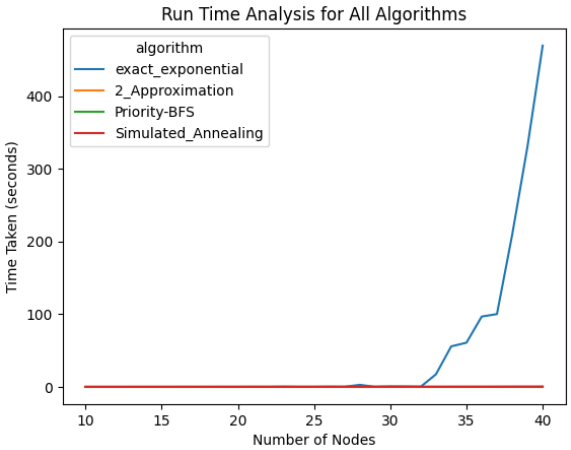
From the experimentation, we can observe that the exact exponential algorithm indeed grows exponentially. The other 3 algorithm such as 2-approximation, priority BFS and simmulated annealing takes linear time. Though the run time of simulated annealing is quite high with respect to ther 2 linear time algorithms.

9.1.2 Approximation Ratio

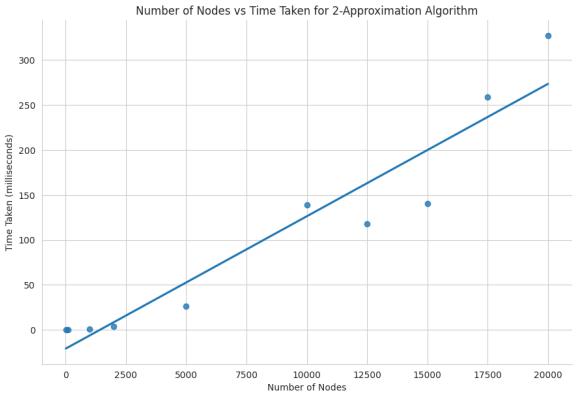
Since this is a maximization problem, the approximation ratio is calculated by dividing the number of leaf nodes from exact exponential algorithm by the number of leaf nodes from respected algorithm.
We can observe that the algorithms are near optimal having approximation ratio is close to 1.

9.1.3 Performance on different classes of graph

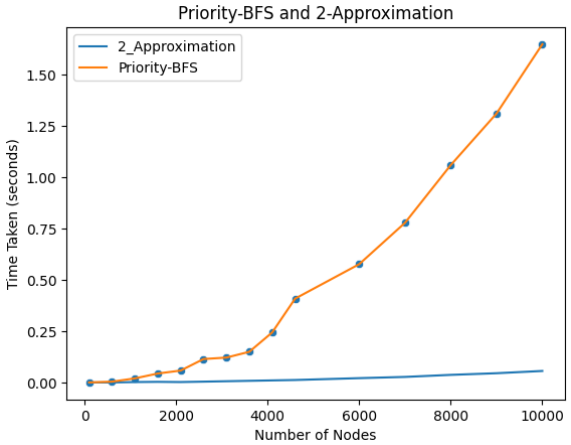
We can observe that 2 approximation and priority BFS works well on random graph. On d-regular graph and on incomplete grid graph, the 2 approximation and simulated annealing works well. On complete grid graph, 2-approximation algorithm outperforms others. On scale free network, simulated annealing works best.



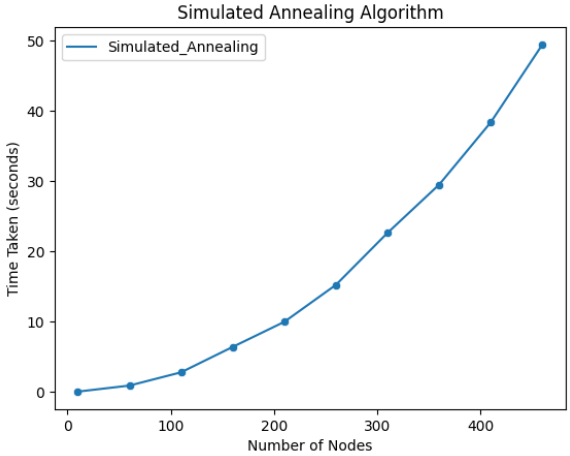
(a)



(b)



(c)



(d)

Figure 9.1: Run Time Analysis

Algorithm	**Average Apprx Ratio**
2-approximation	1.0733791412673865
Priority BFS	1.089891025939413
Simulated Annealing	1.107478296474102

Figure 9.2: Average Approximation Ratio for different algorithms

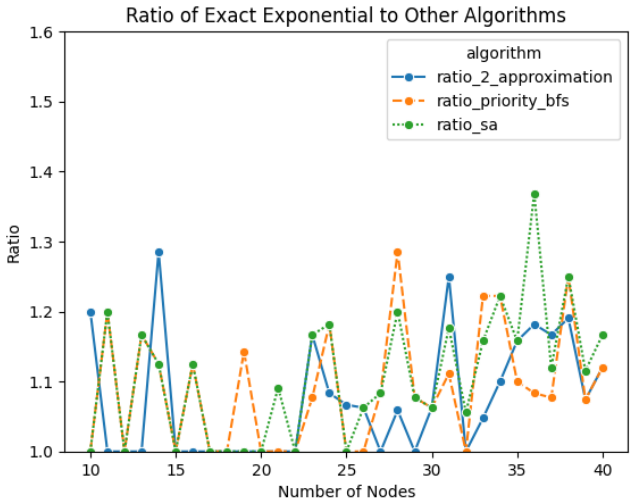
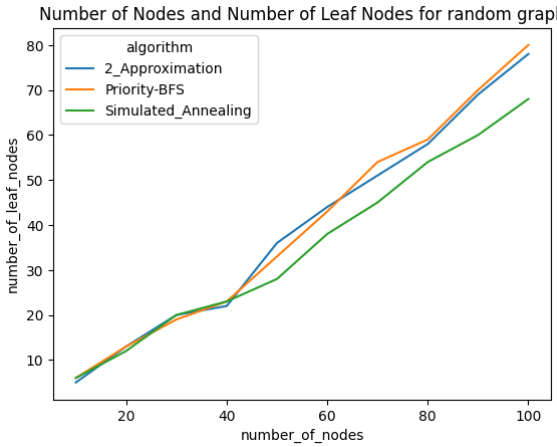
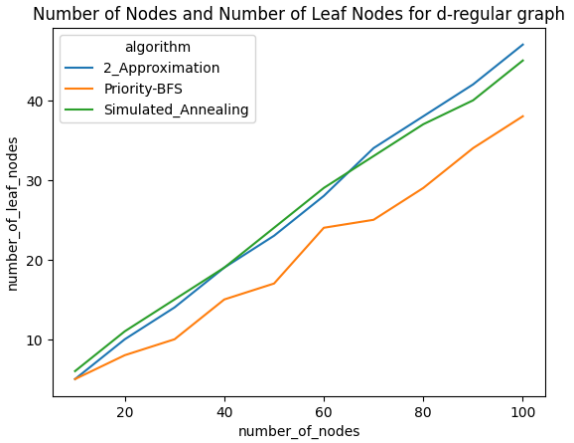


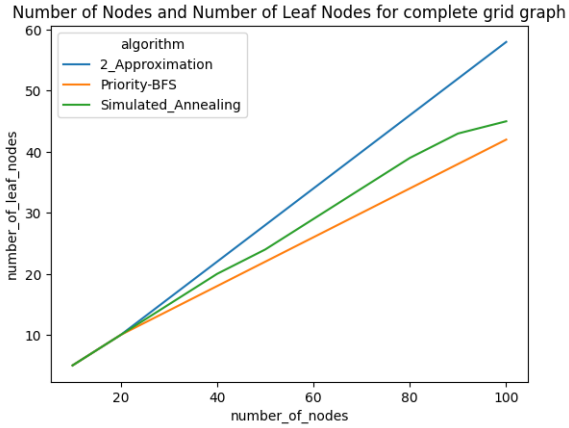
Figure 9.3: Approximation Ratio of 3 algorithms



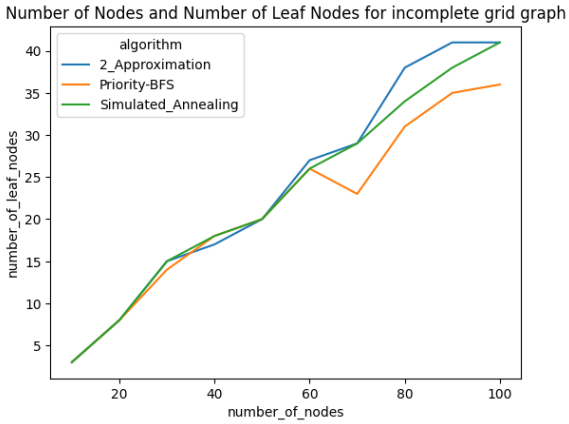
(a)



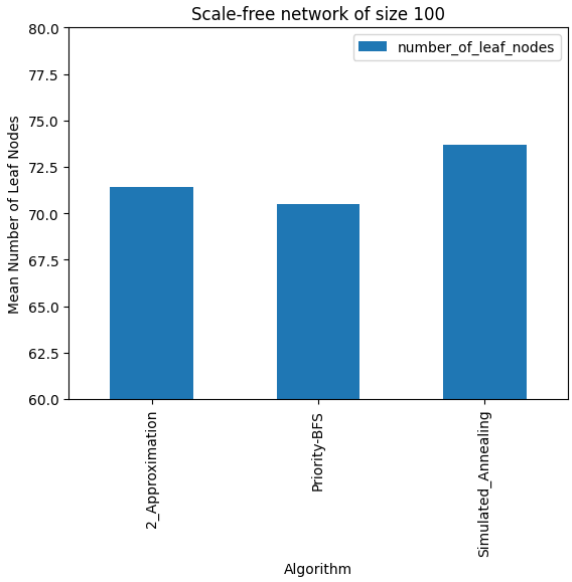
(b)



(c)



(d)



(e)

Figure 9.4: Performance on different classes of graph.