Maximum Leaf Spanning Tree

Project Checkpoint Presentation

1905026 - Wasif Hamid 1905038 - Ajoy Dey
1905040 - Asif Al Shahriar 1905062 - Lara Khanom
1905104 - Kazi Istiak Uddin Toriqe

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

20 October 2024

Presentation Outline

Table of Contents

Maximum Leaf Spanning Tree

Spanning Tree

A spanning tree T, of a graph G, is a subgraph that includes all the vertices of the original graph and is a tree (i.e., a connected, acyclic graph).

Maximum Leaf Spanning Tree

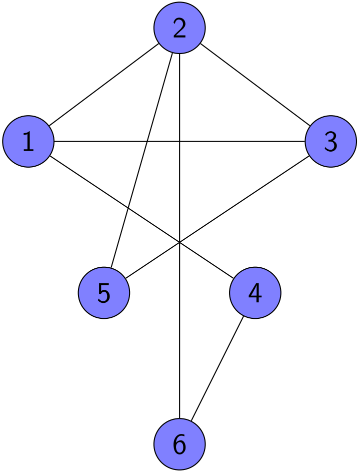
Spanning Tree

A spanning tree T, of a graph G, is a subgraph that includes all the vertices of the original graph and is a tree (i.e., a connected, acyclic graph).

Maximum Leaf Spanning Tree

A maximum leaf spanning tree is a spanning tree that maximizes the number of leaf nodes (vertices with degree 1) in the spanning tree.

Example



Example

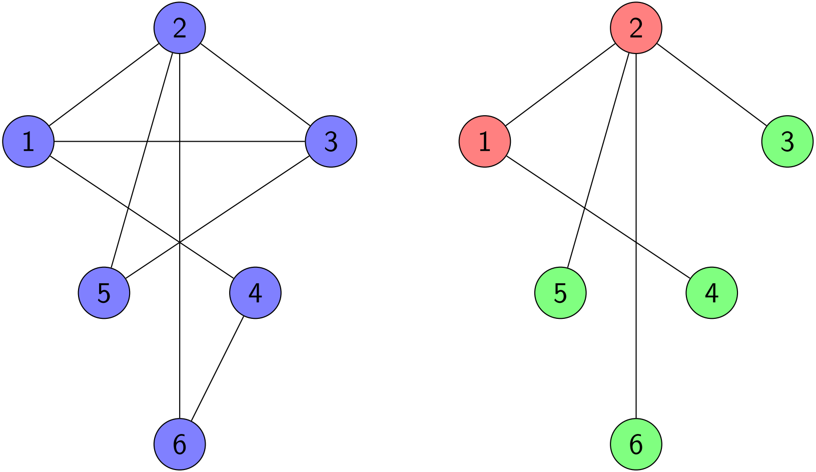


Figure: Maximum Leaf Spanning Tree

Problem Definition

The Maximum Leaf Spanning Tree (MaxLST) problem comes in two primary versions.

Problem Definition

The Maximum Leaf Spanning Tree (MaxLST) problem comes in two primary versions.

1. **Optimization Version:**
   In this version, the goal is to maximize the number of leaf nodes in the spanning tree of a given graph.

Problem Definition

The Maximum Leaf Spanning Tree (MaxLST) problem comes in two primary versions.

1. **Optimization Version:**
   In this version, the goal is to maximize the number of leaf nodes in the spanning tree of a given graph.
2. **Decision Version:**
   The decision version of the MLST problem asks whether it is possible to find a spanning tree with at least a given number of leaf nodes.

Optimization Version of MaxLST

Problem Statement:

- **Input:**
  A connected, undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges.

Optimization Version of MaxLST

Problem Statement:

- **Input:**
  A connected, undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges.
- **Objective:**
  Find a spanning tree of G that maximizes the number of leaf nodes

Decision Version of MaxLST

Problem Statement:

- **Input:**
  A connected, undirected graph $G = (V, E)$, and an integer k.

Decision Version of MaxLST

Problem Statement:

- **Input:**
  A connected, undirected graph $G = (V, E)$, and an integer k.
- **Question:**
  Does there exist a spanning tree of G with at least k leaf nodes?

Table of Contents

Complexity of MaxLST

The MaxLST problem , particularly the decision version of the problem is an NP-Complete problem. Which means that the problem is both in the NP and NP-Hard class of problems.

Complexity of MaxLST

The MaxLST problem, particularly the decision version of the problem is an NP-Complete problem. Which means that the problem is both in the NP and NP-Hard class of problems.

NP problems or Nondeterministic Polynomial time problem is the set of all decision problems solvable in polynomial time by a non-deterministic algorithm. A problem is in NP if it's solution can be verified in polynomial time.

Complexity of MaxLST

The MaxLST problem, particularly the decision version of the problem is an NP-Complete problem. Which means that the problem is both in the NP and NP-Hard class of problems.

NP problems or Nondeterministic Polynomial time problem is the set of all decision problems solvable in polynomial time by a non-deterministic algorithm. A problem is in NP if it's solution can be verified in polynomial time.

NP-Hard Problems are the set of all those problems that are as hard as the hardest problem in NP. We prove a problem is NP-Hard by reducing an instance of a well known NP problem such as 3-SAT, Vertex Cover, Independent Set to an instance of the relevant problem

Proof: MaxLST is in NP

- Given a solution T, for a Graph $G = (V, E)$ and a integer k is given, we can just check if the tree has k leaves or more.
- By applying a simple BFS or DFS we can both check
  - if the solution is indeed a spanning tree.
  - if it has k leaves.
- So, the MaxLST problem is in NP

Proof: MaxLST is in NP-Hard

We prove the NP-hardness of Max-Leaf Spanning Tree in two steps -

1. Reduce the Minimum Vertex Cover problem to Minimum Connected Dominating Set (MinCDS) problem.

Proof: MaxLST is in NP-Hard

We prove the NP-hardness of Max-Leaf Spanning Tree in two steps -

1. Reduce the Minimum Vertex Cover problem to Minimum Connected Dominating Set (MinCDS) problem.
2. Show that Minimum Connected Dominating Set is equivalent to Max-Leaf Spanning Tree problem.

Minimum Vertex Cover Problem

Vertex Cover: Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a vertex cover is a subset of vertices $C \subseteq V$ such that every edge in the graph has at least one of its endpoints in C.

Minimum Vertex Cover Problem

Vertex Cover: Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a vertex cover is a subset of vertices $C \subseteq V$ such that every edge in the graph has at least one of its endpoints in C.

Objective: We want a subset $C \subseteq V$ such that for every edge $(u, v) \in E$, at least one of the vertices u or v is in C, and the number of vertices in C is minimized.

Minimum Vertex Cover Problem

Vertex Cover: Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a vertex cover is a subset of vertices $C \subseteq V$ such that every edge in the graph has at least one of its endpoints in C.

Objective: We want a subset $C \subseteq V$ such that for every edge $(u, v) \in E$, at least one of the vertices u or v is in C, and the number of vertices in C is minimized.

Question: Given a connected graph $G = (V, E)$ and an integer k, does G have a vertex cover of size at most k?

Example

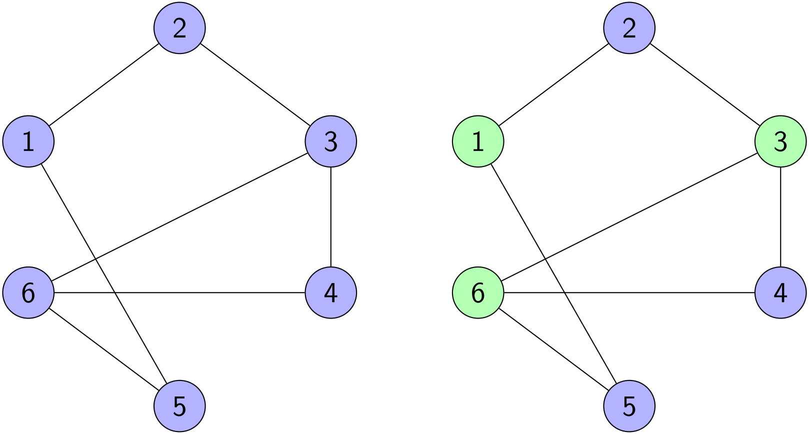


Figure: Minimum Vertex cover in a graph

Minimum Connected Dominating Set Problem

Dominating Set: Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a dominating set is a subset of vertices $D \subseteq V$ such that every vertex in the graph is either in D or adjacent to a vertex in D.

Minimum Connected Dominating Set Problem

Dominating Set: Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a dominating set is a subset of vertices $D \subseteq V$such that every vertex in the graph is either in D or adjacent to a vertex in D.

Objective: We want a subset $D \subseteq V$ such that for every node $u \in V$, u is either in D or is adjacent to a node $v \in D$, the nodes in D create a connected sub-graph and the number of vertices in D is minimized.

Minimum Connected Dominating Set Problem

Dominating Set: Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a dominating set is a subset of vertices $D \subseteq V$such that every vertex in the graph is either in D or adjacent to a vertex in D.

Objective: We want a subset $D \subseteq V$ such that for every node $u \in V$, u is either in D or is adjacent to a node $v \in D$, the nodes in D create a connected sub-graph and the number of vertices in D is minimized.

Question: Given a connected graph $G = (V, E)$ and an integer k, does G have a connected dominating set of size at most k?

Example

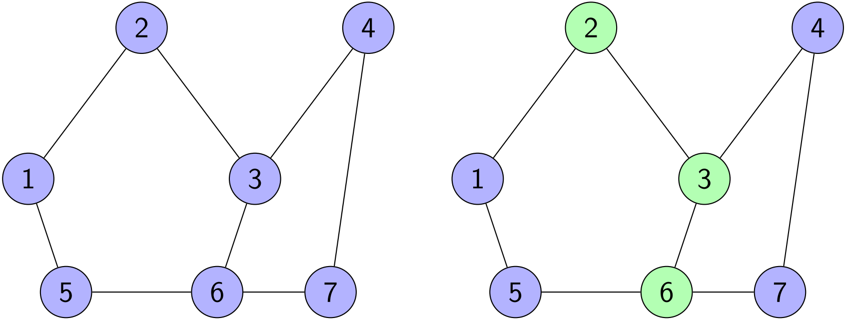


Figure: Minimum Connected Dominating Set in a graph

Reduction from VC to MinCDS

Given a vertex cover instance $G = (V, E)$, we reduce it to a minimum connected dominating set instance $G' = (V', E')$ as follows -

- G' has the complete graph induced by V.
- for each edge $(u, v) \in E$ we add a node x_{uv} in V' and add two edges (u, x_{uv}) and (v, x_{uv})

Formally-

$$V' = V \cup \{x_{uv} : (u, v) \in E\}$$

$$\text{E'} = \{\text{(u,v)} : \text{u} \in V, v \in V \textit{ and } u \neq v\} \cup \{(x_{uv}, u) : (u, v) \in E\}$$

This is clearly reducible in polynomial time.

Reduction from VC to MinCDS Example

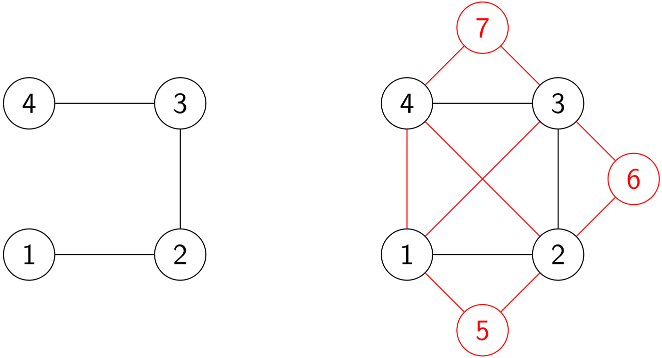


Figure: Reduction from VC to MinCDS

Reduction from VC to MinCDS (cont.)

Necessity

Let S be a vertex cover of G. For any $v \in V'$, there are two cases-

1. $v \in V$: By the definition of vertex cover we know that v is dominated by S
2. $v = v_{xy}$ is an edge vertex: Because (x, y) is covered by S, then $x \in S$ or $y \in S$, v_{xy} is dominated by S

Thus S is a connected dominating set of G'

Reduction from VC to MinCDS (cont.)

Sufficiency

Let S be a connected dominating set in G'.

- We first observe that for any edge vertex v_{xy} in, it can only be dominated by x or y
- For any edge vertex $v_{xy} \in S$. The replacement does not increase the cardinality of S and mantains S's property of being a dominating set.
- Now S contains no edge vertex, so every edge vertex is dominated by S, that is to say, every edge in G has at least one endpoint in S .

Thus S is a vertex cover for G

Equivalence of MinCDS to MaxLST

Proving Equivalence

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

Equivalence of MinCDS to MaxLST

Proving Equivalence

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

Necessity

Let T be a spanning tree of the sub-graph induced by S. For each vertex $v \in V \backslash S$, there exists a vertex $u \in S$ where $(u, v) \in E$. We add these vertices u and the edges (u, v) to T. All newly added nodes are new in T, has a degree of 1 and doesn't add a cycle. That is, T is still a spanning tree. Therefore, $|Leaves(T)| \geq n - k$

Equivalence of MinCDS to MaxLST

Proving Equivalence

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

Necessity

Let T be a spanning tree of the sub-graph induced by S. For each vertex $v \in V \backslash S$, there exists a vertex $u \in S$ where $(u, v) \in E$. We add these vertices u and the edges (u, v) to T. All newly added nodes are new in T, has a degree of 1 and doesn't add a cycle. That is, T is still a spanning tree. Therefore, $|Leaves(T)| \geq n - k$

Sufficiency

Let T be a spanning tree of V with at least $n - k$ leaves. Every leaf v is dominated by an inner vertex and all inner vertices are connected. So they form a dominating set S and $|S| \leq k$.

Table of Contents

Table: Exact Exponential Algorithms

Algorithm	Runtime	Year	Author	Comment
Naive brute-force	$\Omega(2^n)$	-	-	Considering all possible leafset
Simple Branching [9]	$O(4^k poly(n))$	2008	Kneis, Langer, and Rossmanith	Parameterized in the number of leaves k
Improved Branching [2]	$O(3.72^k poly(n))$	2010	Daligault et al.	Parameterized in the number of leaves k
Connected Dom Set [4]	$O(1.9407^n)$	2008	Fomin, Grandoni, and Kratsch	Equivalent to MaxLST
Branching Algo [3]	$O(1.8966^n)$	2011	Fernau et al.	Branching algo based on some properties
Worst Case Lower Bound [3]	$\Omega(1.4422^n)$	-	-	Lower Bound

Branching Algorithm

- Proposed by Fernau et. al at IWPEC 2009
- Improves upon the previous approach where leaves of a subtree of the graph were repeatedly branched in order to decide whether it can remain a leaf or must become an internal node of the spanning tree. (Kneis et.al and Dalgaut et. al)
- In that approach branching on nodes of small degree (with two possible successors) becomes the worst case resulting in a bad running time.
- Fernau et. al improved upon the technique by marking nodes as leaves as early as possible even when they are not yet attached to an internal node.
- The algorithm maintains 5 disjoint sets of vertices to mark the nodes and terminates when it has found a spanning tree

Node Sets Definition

Internal Nodes(IN): The nodes that were decided to be internal nodes of the spanning tree.

Node Sets Definition

Internal Nodes(IN): The nodes that were decided to be internal nodes of the spanning tree.

Leaf Nodes(LN): Nodes decided to be leaves in the spanning tree. All neighbors of these nodes are already processed

Node Sets Definition

Internal Nodes(IN): The nodes that were decided to be internal nodes of the spanning tree.

Leaf Nodes(LN): Nodes decided to be leaves in the spanning tree. All neighbors of these nodes are already processed

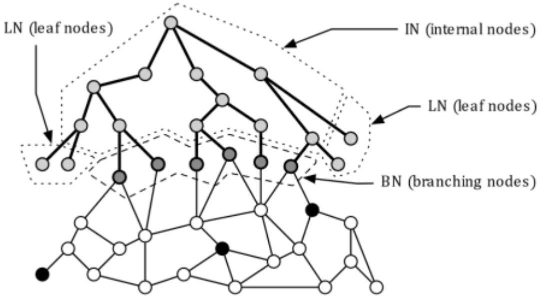
Branching Nodes(BN): Nodes that are right now treated as leaves but all their neighbors haven't been processed i.e they are candidates for branching.

Node Sets Definition

Internal Nodes(IN): The nodes that were decided to be internal nodes of the spanning tree.

Leaf Nodes(LN): Nodes decided to be leaves in the spanning tree. All neighbors of these nodes are already processed

Branching Nodes(BN): Nodes that are right now treated as leaves but all their neighbors haven't been processed i.e they are candidates for branching.



Node Sets Definition(cont.)

Floating Leaves(FL): The nodes that were decided to be leaves of the spanning tree but hasn't been attached to the current spanning tree yet.

Node Sets Definition(cont.)

Floating Leaves(FL): The nodes that were decided to be leaves of the spanning tree but hasn't been attached to the current spanning tree yet.

Free Nodes(FN): All the rest of the nodes that haven't been attached to the tree and hasn't been decided as leaves either.

Floating Leaves(FL): The nodes that were decided to be leaves of the spanning tree but hasn't been attached to the current spanning tree yet.

Free Nodes(FN): All the rest of the nodes that haven't been attached to the tree and hasn't been decided as leaves either.

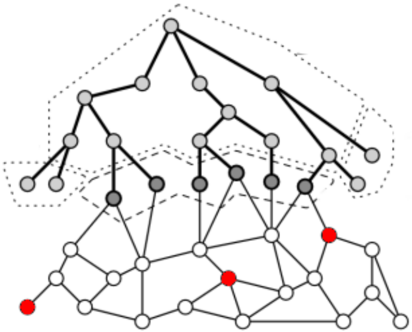


Figure: Free Nodes (in white) and Floating Leaves (in red)

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.
2. If there exists a node $v \in BN$ with $d(v) = 0$, then move v into LN

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.
2. If there exists a node $v \in BN$ with $d(v) = 0$, then move v into LN
3. If there exists a free vertex v with $d(v) = 1$, then move v into FL.

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.
2. If there exists a node $v \in BN$ with $d(v) = 0$, then move v into LN
3. If there exists a free vertex v with $d(v) = 1$, then move v into FL.
4. If there exists a free vertex v with no neighbors in $Free \cup FL$, then move v into FL

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.
2. If there exists a node $v \in BN$ with $d(v) = 0$, then move v into LN
3. If there exists a free vertex v with $d(v) = 1$, then move v into FL.
4. If there exists a free vertex v with no neighbors in $Free \cup FL$, then move v into FL
5. If there exists a triangle $\{x, y, z\}$ in G with x a free vertex and $d(x) = 2$, then move x into FL.

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.
2. If there exists a node $v \in BN$ with $d(v) = 0$, then move v into LN
3. If there exists a free vertex v with $d(v) = 1$, then move v into FL.
4. If there exists a free vertex v with no neighbors in $Free \cup FL$, then move v into FL
5. If there exists a triangle $\{x, y, z\}$ in G with x a free vertex and $d(x) = 2$, then move x into FL.
6. If there exists a node $u \in BN$ which is a cut vertex in G, then move u into IN.

Reduction Rules

The algorithm applies the following reduction rules recursively to reduce the search space-

1. 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.
2. If there exists a node $v \in BN$ with $d(v) = 0$, then move v into LN
3. If there exists a free vertex v with $d(v) = 1$, then move v into FL.
4. If there exists a free vertex v with no neighbors in $Free \cup FL$, then move v into FL
5. If there exists a triangle $\{x, y, z\}$ in G with x a free vertex and $d(x) = 2$, then move x into FL.
6. If there exists a node $u \in BN$ which is a cut vertex in G, then move u into IN.
7. If there exist two adjacent vertices $u, v \in V$ such that $u \in LN$ and $v \in V \backslash IN$, then remove the edge (u, v) from G.

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN

Branching and Halting Rules

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN
2. if $d(v) = 2$, branches based on the degrees of its neighbors $\{x_1, x_2\}$, deciding whether v and its neighbors should be leaves or internal nodes

Branching and Halting Rules

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN
2. if $d(v) = 2$, branches based on the degrees of its neighbors $\{x_1, x_2\}$, deciding whether v and its neighbors should be leaves or internal nodes
3. if $d(v) = 1$, the algorithm extends a path from v and branches based on its final neighbor, deciding whether vertices along the path become leaves or internal nodes.

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN
2. if $d(v) = 2$, branches based on the degrees of its neighbors $\{x_1, x_2\}$, deciding whether v and its neighbors should be leaves or internal nodes
3. if $d(v) = 1$, the algorithm extends a path from v and branches based on its final neighbor, deciding whether vertices along the path become leaves or internal nodes.
4. whenever a node v becomes an internal node, all it's neighbors are attached to the tree as well.

Branching and Halting Rules

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN
2. if $d(v) = 2$, branches based on the degrees of its neighbors $\{x_1, x_2\}$, deciding whether v and its neighbors should be leaves or internal nodes
3. if $d(v) = 1$, the algorithm extends a path from v and branches based on its final neighbor, deciding whether vertices along the path become leaves or internal nodes.
4. whenever a node v becomes an internal node, all it's neighbors are attached to the tree as well.
5. Over the course of the algorithm only nodes in BN or Free can change sets. Other nodes must remain in their sets.

Branching and Halting Rules

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN
2. if $d(v) = 2$, branches based on the degrees of its neighbors $\{x_1, x_2\}$, deciding whether v and its neighbors should be leaves or internal nodes
3. if $d(v) = 1$, the algorithm extends a path from v and branches based on its final neighbor, deciding whether vertices along the path become leaves or internal nodes.
4. whenever a node v becomes an internal node, all it's neighbors are attached to the tree as well.
5. Over the course of the algorithm only nodes in BN or Free can change sets. Other nodes must remain in their sets.

Halting Rules:

1. If at any point a node is found as unreachable after reduction, we return 0 denoting a failure

Branching and Halting Rules

Branching Rules:

1. if $d(v) \geq 3$ or if $d(v) = 2$ and none of v's neighbours are in FL then branch into two solutions where v is in IN or in LN
2. if $d(v) = 2$, branches based on the degrees of its neighbors $\{x_1, x_2\}$, deciding whether v and its neighbors should be leaves or internal nodes
3. if $d(v) = 1$, the algorithm extends a path from v and branches based on its final neighbor, deciding whether vertices along the path become leaves or internal nodes.
4. whenever a node v becomes an internal node, all it's neighbors are attached to the tree as well.
5. Over the course of the algorithm only nodes in BN or Free can change sets. Other nodes must remain in their sets.

Halting Rules:

1. If at any point a node is found as unreachable after reduction, we return 0 denoting a failure
2. if all nodes are either in IN or LN we return size of LN as answer.

Existing Approximation Algorithms

Table: Approximation Algorithms

Algorithm	Complexity	Approx ratio	Author	Comment
Local Optimization[11]	$O(n^4)$	Approx. 5	Lu and Ravi	Local Approximation Technique
Local Optimization [11]	$O(n^7)$	Approx. 3	Lu and Ravi	Local Approximation Technique
Greedy [10]	$O((m+n)\alpha(m,n))$	Approx. 3	Lu and Ravi	Introduced Leafy Forests
Incrementally building a subgraph [14]	$O(m)$	Approx. 2	Solis-Oba, Bonsma, and Lowski	Uses expansions rule
Expands the tree	$O(m)$	Approx. 2	Liao and Lu	Simple, Year2023

Incremental Sub-graph Construction

- Introduced by Roberto Solis-Oba at ESA 1998
- The first constant factor approximation algorithm was developed by Lu and Ravi with an approximation ratio of 3 and 5[11]. They later introduced the idea of leafy forests to create a more efficient algorithm[10].
- Solis-soba et. al introduced prioritized expansion rules to improve the approximation ratio to 2 for graphs with maximum degree of at least 3.
- The priority of a rule reflects the number of leaves that the rule adds to the forest. Hence it is desirable to use only high priority rules to build the forest.
- The low priority rules are needed though to ensure that the number of components of the forest can be kept small enough.

Notations

- $V(G)$ Set of all nodes of G.
- $\overline{V(G)}$ Set of all nodes not in G.
- $L(G)$ Set of leaves of G.
- $N_G(v)$ Neighborhood of node v in G.

Expanding a node v

Let F be a a (possibly empty) subgraph of a graph G. The operation of expanding a vertex $v \in V(G)$ consists of -

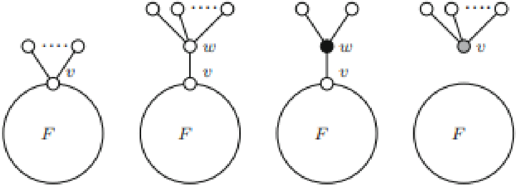
– Adding v to F, in case $v \notin V(F)$

– For every $w \in N_G(v) \backslash V(F)$: adding vertex w and edge (v, w) to F

This operation yields a new subgraph F' of G which is a forest if F was a forest and if $v \notin V(F)$ it increases the number of components of F by 1

We use the following four expansion rules, where the given order defines the priorities. That is, if for any node Rule 2 can be applied, then Rule 3 or 4 can't be applied.

1. If F contains a vertex v with at least two neighbors in $\overline{V(F)}$, then expand v.
2. If F contains a vertex v with only one neighbor w in $\overline{V(F)}$, which in turn has at least three neighbors in $\overline{V(F)}$, then first expand v, and next expand w.
3. If F contains a vertex v with only one neighbor w in $\overline{V(F)}$, which in turn has two neighbors in $\overline{V(F)}$, then first expand v, and next expand w.
4. If $\overline{V(F)}$ contains a vertex v with at least three neighbors in $\overline{V(F)}$, then expand v.



As we can see that, only rule 4 can increase the number of components. As this rule has the least priority, we limit the number of components. Also, once a new component has been introduced no other rule can change this as they don't operate on nodes already in the forest.

Since G has a maximum degree of at least 3, Rule 4 also ensures that the resulting forest is never empty.

Algorithm

1. We start with an empty forest F
2. Apply the expansion rules 1-4 as long as one of them can be applied on some node
3. We randomly choose a vertex in F and expanding it until we obtain a spanning subgraph.
4. Arbitrary edges are added to F without creating cycles, until a connected spanning tree is not obtained
5. This is the solution tree for MaxLST

Heuristic Algorithms:

- BFS Algorithm
- Priority BFS
- Max Priority BFS

Meta-Heuristic Algorithms:

- Simulated Annealing
- Genetic Algorithm
- Artificial Bee Colony Algorithm

Comparison between Heuristic And Meta-heuristic Algorithms

Graph		Opt.	BFS	PBFS	MaxPBFS	SA			ABC			GA		
n	m		solution	solution	solution	solution	iter.	time	solution	iter.	time	solution	iter.	time
4	4	9	8	8	**9**	**9**	14	0.16	**9**	0	0.07	**9**	0	0.68
	5	11	10	10	**11**	**11**	134	0.17	**11**	0	0.09	**11**	0	0.93
	6	14	12	13	13	**14**	198	0.23	**14**	0	0.12	**14**	1	1.12
	7	16	14	14	15	**16**	623	0.33	**16**	39	0.17	**16**	1	1.39
	8	18	16	17	**18**	**18**	177	0.39	**18**	0	0.21	**18**	0	1.72
	9	21	18	19	20	20	88	0.48	20	0	0.23	**21**	1	1.90
5	5	14	10	13	**14**	**14**	28	0.26	**14**	0	0.14	**14**	0	1.23
	6	18	12	17	17	**18**	1246	0.35	17	0	0.17	**18**	1	1.45
	7	20	14	19	**20**	**20**	3	0.45	**20**	0	0.21	**20**	0	1.72
	8	23	16	21	**23**	**23**	1737	0.58	**23**	0	0.31	**23**	0	2.11
	9	27	18	25	**26**	25	0	0.69	**26**	0	0.31	**26**	0	2.38
6	6	22	12	18	21	**22**	999	0.45	**22**	48	0.24	**22**	1	1.85
	7	26	14	**26**	**26**	**26**	0	0.59	**26**	0	0.29	**26**	0	2.19
	8	30	16	**30**	**30**	**30**	0	0.74	**30**	0	0.34	**30**	0	2.56
	9	34	18	32	33	**34**	700	0.90	**34**	1	0.51	**34**	1	3.13
7	7	29	14	27	**29**	**29**	143	0.76	**29**	0	0.37	**29**	0	2.89
	8	33	16	**33**	**33**	**33**	0	0.99	**33**	0	0.47	**33**	0	3.49
	9	39	18	37	37	**39**	3714	1.25	37	0	0.64	38	2	3.85
8	8	38	16	34	36	**38**	5019	1.29	37	11	0.76	**38**	1	4.36
	9	45	18	42	42	**45**	8693	1.62	43	145	0.97	**45**	41	6.13
9	9	51	18	47	48	**51**	4884	1.99	50	149	1.29	**51**	2	5.53

Figure: Experiments on small complete grid graphs[1]

Comparison between Heuristic And Meta-heuristic Algorithms

Graph		BFS	PBFS	MaxPBFS	Opt.	SA			ABC			GA		
n	p	sol.	sol.	sol.		sol.	iter.	time	sol.	iter.	time	sol.	iter.	time
30	0.1	17.10	17.55	18.60	18.95	18.85	720.9	0.34	18.85	9.8	0.19	18.85	0.3	1.44
	0.2	21.20	21.60	22.95	23.55	23.25	1480.2	0.34	23.20	17.2	0.22	23.50	1.1	1.41
	0.3	23.95	23.85	25.00	25.45	25.10	1369.8	0.34	25.15	4.2	0.28	**25.45**	1.0	1.46
	0.4	25.10	24.50	26.00	26.35	25.80	1188.6	0.34	26.10	7.7	0.36	26.30	0.4	1.50
	0.5	26.50	25.65	26.95	27.05	26.35	349.5	0.34	27.00	5.1	0.49	**27.05**	0.8	1.54
40	0.1	24.65	25.50	27.15	28.05	27.65	2042.4	0.58	27.65	34.0	0.34	27.75	1.2	2.05
	0.2	30.50	30.20	32.15	33.40	32.60	2277.6	0.56	32.60	31.9	0.39	33.15	1.6	2.15
	0.3	33.15	32.50	34.30	35.05	34.25	2212.5	0.55	34.65	28.2	0.52	**35.05**	1.9	2.27
	0.4	35.05	34.50	35.75	36.20	35.40	1105.6	0.55	35.95	11.9	0.73	36.15	0.5	2.28
	0.5	35.95	35.30	36.45	37.00	36.10	1434.2	0.54	36.45	2.1	0.87	36.85	0.4	2.36
50	0.1	33.00	34.50	36.90	38.10	37.70	2977.7	0.82	37.40	40.0	0.47	37.20	2.8	3.00
	0.2	39.90	39.75	41.65	43.25	42.25	3409.1	0.82	42.40	122.0	0.71	42.90	10.0	3.20
	0.3	42.90	42.10	44.05	44.95	43.70	2216.1	0.83	44.30	40.3	0.91	**44.95**	2.0	3.03
	0.4	44.60	43.75	45.10	46.00	44.65	1802.3	0.80	45.35	18.7	1.20	45.90	1.3	3.26
	0.5	45.90	45.00	46.35	47.00	45.75	1529.5	0.80	46.35	1.2	1.57	46.85	1.6	3.42
60	0.4	54.40	53.50	55.20	56.05	54.45	2277.1	1.18	55.30	7.0	2.07	56.00	2.1	4.88
	0.5	55.75	54.70	56.10	56.70	55.20	1271.7	1.22	56.10	0.2	2.80	56.45	0.6	5.27
	0.6	56.40	55.45	56.95	57.00	55.95	942.6	1.27	**57.00**	13.6	3.87	**57.00**	0.1	5.65

Figure: Experiments on small random graphs[1]

Table of Contents

Experiments

We will be comparing the performance of our chosen algorithms with the following algorithms -

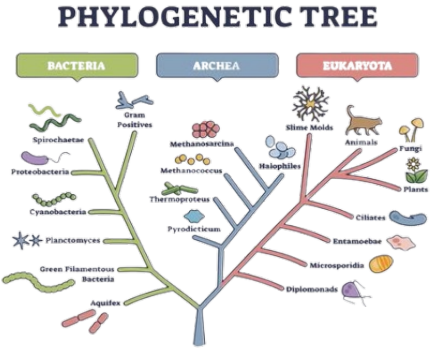
- BFS
- Max Priority BFS
- Artificial Bee Colony Algorithm
- Genetic Algorithm

The comparisons will be done in these types of dense and sparse graphs-

- Random Graph
- D-regular graph
- Complete and Incomplete Grid graphs

Table of Contents

Phylogenetic Tree



Phylogenetic trees are designed to increase species representation while minimizing internal nodes. MLST contributes to this by identifying spanning trees with the greatest number of leaves, reflecting genuine species, and minimizing inferred shared ancestors.

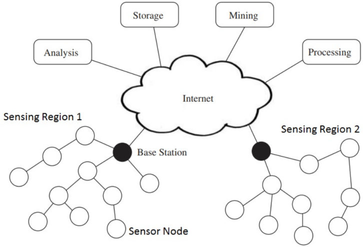
This is important for taxonomy and evolutionary studies because it allows researchers to focus on actual species rather than hypothetical ancestors.

More Theoretical Applications

- **Clustering:**
  MLST can be used as a clustering algorithm, where each cluster is represented by a node in the spanning tree. The minimum latency ensures that clusters are closely related.[6]

- **QoS Provisioning:**
  Ensuring that quality of service (QoS) requirements are met for different applications.[16]

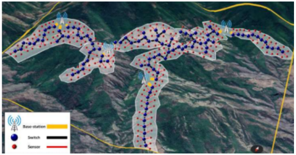
Ditributed Systems



By maximizing leaves, internal nodes are minimized. This is crucial in distributed systems where internal nodes represent servers or resources responsible for routing or control. Minimizing such nodes reduces complexity and resource consumption.

This leads to more efficient communication and control protocols.[12]

Forest Fire management



MLST (Maximum Leaves Spanning Tree) helps optimize forest fire management by maximizing monitoring coverage and communication efficiency with minimal resources. It can improve sensor network design, firefighting coordination, and resource allocation, ensuring better preparedness and response to forest fires.[7]

More Practical Applications

- **Routing Protocols for Wireless Networks:**
  In wireless networks, MLST can be used to determine the optimal topology for a mesh network. The network can achieve maximum throughput and minimize interference by selecting the links with the lowest latency.[8][5][13]
- **Circuit Layouts:**
  MLST can optimize circuit layout by minimizing wire length and delay. It helps identify the critical path for circuit optimization.[15]

Table of Contents

References I

- Christian Brggemann and Tomasz Radzik. "Development and Experimental Comparison of Exact and Heuristic Algorithms for the Maximum-Leaf Spanning Tree Problem Technical Report TR-09-08". In: ().
- Jean Daligault et al. "FPT algorithms and kernels for the directed k-leaf problem". In: *Journal of Computer and System Sciences* 76.2 (2010), pp. 144–152.
- Henning Fernau et al. "An exact algorithm for the maximum leaf spanning tree problem". In: *Theoretical Computer Science* 412.45 (2011), pp. 6290–6302.
- Fedor V Fomin, Fabrizio Grandoni, and Dieter Kratsch. "Solving connected dominating set faster than 2 n". In: *Algorithmica* 52 (2008), pp. 153–166.

- Sudipto Guha and Samir Khuller. "Approximation algorithms for connected dominating sets". In: *Algorithmica* 20 (1998), pp. 374–387.
- Anupam Gupta, Amit Kumar, and Tim Roughgarden. "Simpler and better approximation algorithms for network design". In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 2003, pp. 365–372.
- Amir Masoud Hosseinmardi, Hamid Farvaresh, and Isa Nakhai Kamalabadi. "A new formulation and algorithm for the maximum leaf spanning tree problem with an application in forest fire detection". In: *Engineering Optimization* 55.7 (2023), pp. 1226–1242.

References III

Sayaka Kamei et al. “A self-stabilizing 3-approximation for the maximum leaf spanning tree problem in arbitrary networks”. In: *Journal of Combinatorial Optimization* 25 (2013), pp. 430–459.

Joachim Kneis, Alexander Langer, and Peter Rossmanith. “A New Algorithm for Finding Trees with Many Leaves”. In: *Algorithmica* 61 (2008), pp. 882–897. URL: https://api.semanticscholar.org/CorpusID:2579003.

Hsueh-I Lu and R Ravi. “Approximating Maximum Leaf Spanning Trees in Almost Linear Time”. In: *Journal of Algorithms* 29.1 (1998), pp. 132–141. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1998.0944. URL: https://www.sciencedirect.com/science/article/pii/S0196677498909440.

References IV

Hsueh-I Lu and R Ravi. "The power of local optimization: Approximation algorithms for maximum-leaf spanning tree (DRAFT)". In: *Proc. 30th Annual Allerton Conference on Communication Control and Computing*. Citeseer. 1996, pp. 533–542.

C Payan, M Tchuente, and N.H Xuong. "Arbres avec un nombre maximum de sommets pendants". In: *Discrete Mathematics* 49.3 (1984), pp. 267–273. ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(84)90163-8. URL: https://www.sciencedirect.com/science/article/pii/0012365X84901638.

References V

Nadine Schwartges, Joachim Spoerhase, and Alexander Wolff. "Approximation algorithms for the maximum leaf spanning tree problem on acyclic digraphs". In: *International Workshop on Approximation and Online Algorithms*. Springer. 2011, pp. 77–88.

Roberto Solis-Oba, Paul Bonsma, and Stefanie Lowski. "A 2-approximation algorithm for finding a spanning tree with maximum number of leaves". In: *Algorithmica* 77 (2017), pp. 374–388.

James A Storer. "Constructing full spanning trees for cubic graphs". In: *Information Processing Letters* 13.1 (1981), pp. 8–11.

References VI

Chaitanya Swamy and Amit Kumar. "Primal–dual algorithms for connected facility location problems". In: *Algorithmica* 40 (2004), pp. 245–269.