EFFICIENT ANYTIME SEARCH ALGORITHM FOR SIMPLIFIED INFLUENCE MAXIMIZATION (VERTEX COVER) PROBLEM IN SOCIAL NETWORKS

Ali Vâlâ Barbaros

SPRING-13, CS261A PROJECT

OUTLINE

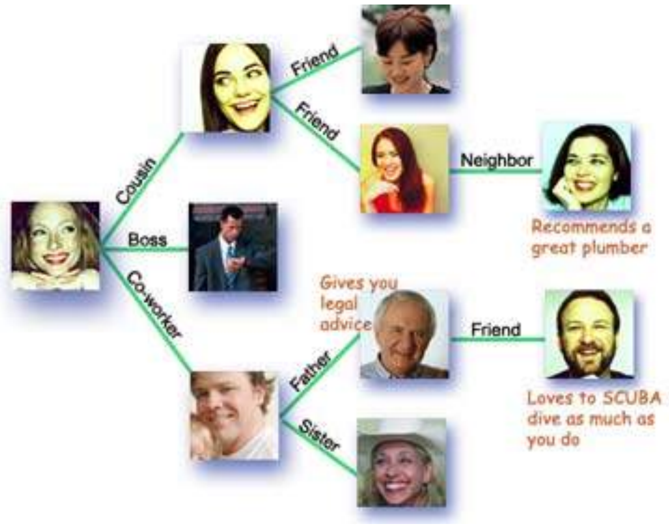
- Original Problem
- Simplified Problem
- Detailed Problem Space
- Pruning of Search Tree
- Node Ordering Heuristic
- Selection of Search Algorithm
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

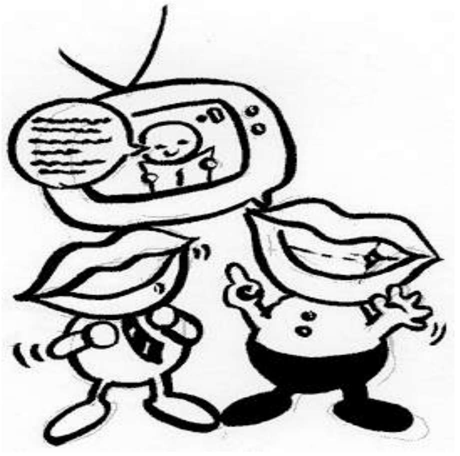
OUTLINE

- ✓ Original Problem
- Simplified Problem
- Detailed Problem Space
- Pruning of Search Tree
- Node Ordering Heuristic
- Selection of Search Algorithm
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

Social Network and Spread of Influence

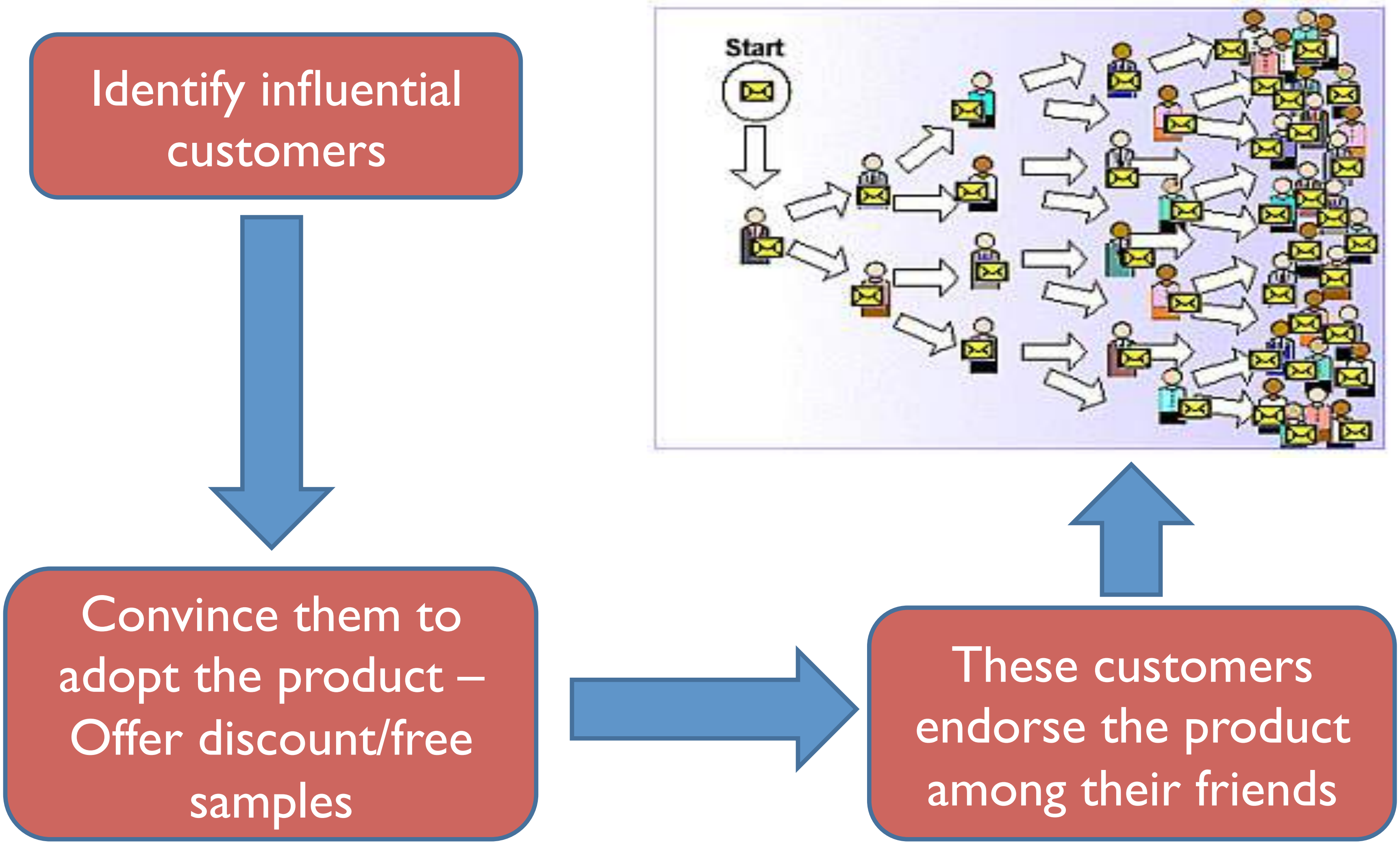
- Social network plays a fundamental role as a medium for the spread of INFLUENCE among its members
  - Opinions, ideas, information, innovation…





- Direct Marketing takes the “word-of-mouth” effects to significantly increase profits (Gmail, Tupperware popularization, Microsoft Origami …)

Spread of Influence / Viral Marketing



Identify influential customers

Convince them to adopt the product – Offer discount/free samples

These customers endorse the product among their friends

Original Problem Statement

– **Given:**

✓ A directed graph **G(V, E)**:

-- **Vertices:** individuals in social network

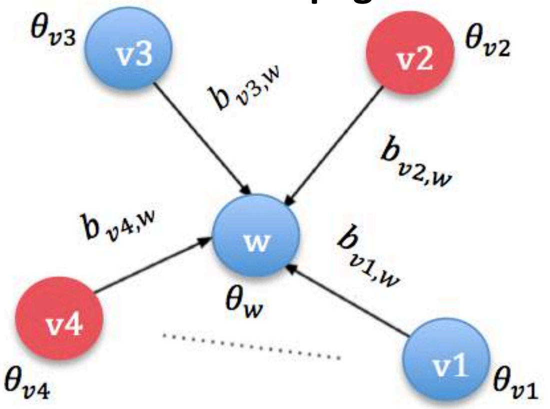
-- **Edges:** connection or relationship between individuals

-- **Weight of Edges(Influence weight):** Each node *w* is influenced by each of its neighbor *v*, according to a edge weight $b_{v,w}$ that is given between interval [0,1]

-- **Node Thresholds:** Each node w has a threshold value θ_w that is given between the interval [0,1]

✓ **k**, size of output seed nodes

✓ **Linear Threshold Propagation Model**



● Active Node

● Inactive Node

The total influence over inactive node w :

$$\sum_{v \in Active\ Neighbors(w)} b_{v,w} = b_{v2,w} + b_{v4,w}$$

– **Output:** **S**, a set of seed nodes

– **GOAL:** Find a small subset of nodes in a social network that could maximize the spread of influences.

Original Problem Statement

NP-hard

- **Given:**

✓ A directed graph **G(V, E)**:

-- **Vertices:** individuals in social network

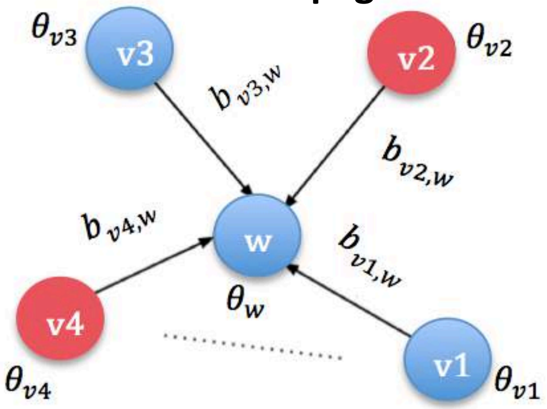
-- **Edges:** connection or relationship between individuals

-- **Weight of Edges(Influence weight):** Each node *w* is influenced by each of its neighbor *v*, according to a edge weight $b_{v,w}$ that is given between interval [0,1]

-- **Node Thresholds:** Each node w has a threshold value θ_w that is given between the interval [0,1]

✓ **k**, size of output seed nodes

✓ **Linear Threshold Propagation Model**



● Active Node

● Inactive Node

The total influence over inactive node w :

$$\sum_{v \in Active\ Neighbors(w)} b_{v,w} = b_{v2,w} + b_{v4,w}$$

- **Output:** **S**, a set of seed nodes
- **GOAL:** Find a small subset of nodes in a social network that could maximize the spread of influences.

Previous Work

- Simple Greedy Algorithm (Kempe et al.,2003)
- Cost-effective Lazy Forward (CELF) Algorithm(Leskovec et al., 2007)
- Linear-time Directed Acyclic Graph (LDAG) Algorithm (Chen et al.,2009)
- SIMPATH Algorithm (Goyal et al.,2012)

ALL OF THEM ARE APPROXIMATION ALGORITHMS

Not guarantee optimal solutions…

OUTLINE

✓ Original Problem
✓ Simplified Problem
- Detailed Problem Space
- Pruning of Search Tree
- Node Ordering Heuristic
- Selection of Search Algorithm
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

Simplified Problem Statement

– **Given:**

✓ An **undirected** graph **G(V, E)**:

-- **Vertices:** individuals in social network

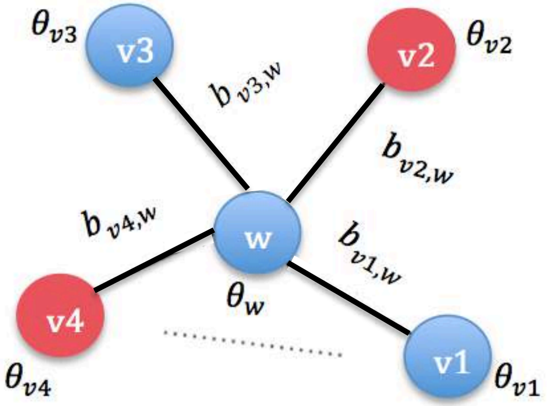
-- **Edges:** connection or relationship between individuals

-- **Weight of Edges(Influence weight):** Each edge has weight ***$b_{v,w}$* = 1**

-- **Node Thresholds:** Each node w has a threshold value θ_w that **equals to its edge degree**

✓ **k**, size of output seed nodes

✓ **Linear Threshold Propagation Model**



● Active Node

● Inactive Node

The total influence over inactive node w :

$$\sum_{v \in Active\ Neighbors(w)} b_{v,w} = b_{v2,w} + b_{v4,w}$$

– **Output:** **S**, a set of seed nodes

– **GOAL:** Find **seed set of size |S| ≤ k** guarantees that **all nodes are eventually active**.

Simplified Problem Statement

Still NP-hard

- **Given:**

✓ An **undirected** graph **G(V, E)**:

-- **Vertices:** individuals in social network

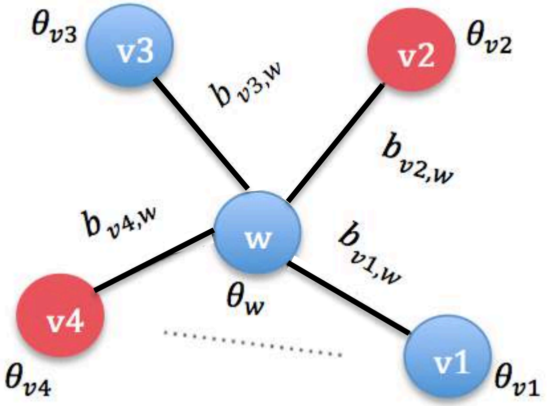
-- **Edges:** connection or relationship between individuals

-- **Weight of Edges(Influence weight):** Each edge has weight $\boldsymbol{b_{v,w} = 1}$

-- **Node Thresholds:** Each node w has a threshold value θ_w that **equals to its edge degree**

✓ **k**, size of output seed nodes

✓ **Linear Threshold Propagation Model**



● Active Node

● Inactive Node

The total influence over inactive node w :

$$\sum_{v \in Active\ Neighbors(w)} b_{v,w} = b_{v2,w} + b_{v4,w}$$

- **Output:** **S**, a set of seed nodes
- **GOAL:** Find **seed set of size |S| ≤ k** guarantees that **all nodes are eventually active**.

Simplified Problem Statement

SIMPLIFIED INFLUENCE MAXIMIZATION PROBLEM

- In Simplified Influence Maximization Problem, we have to cover all the incident edges of each node to activate all nodes.

Simplified Problem Statement

Perspective-I

SIMPLIFIED INFLUENCE MAXIMIZATION PROBLEM

=

Perspective-II

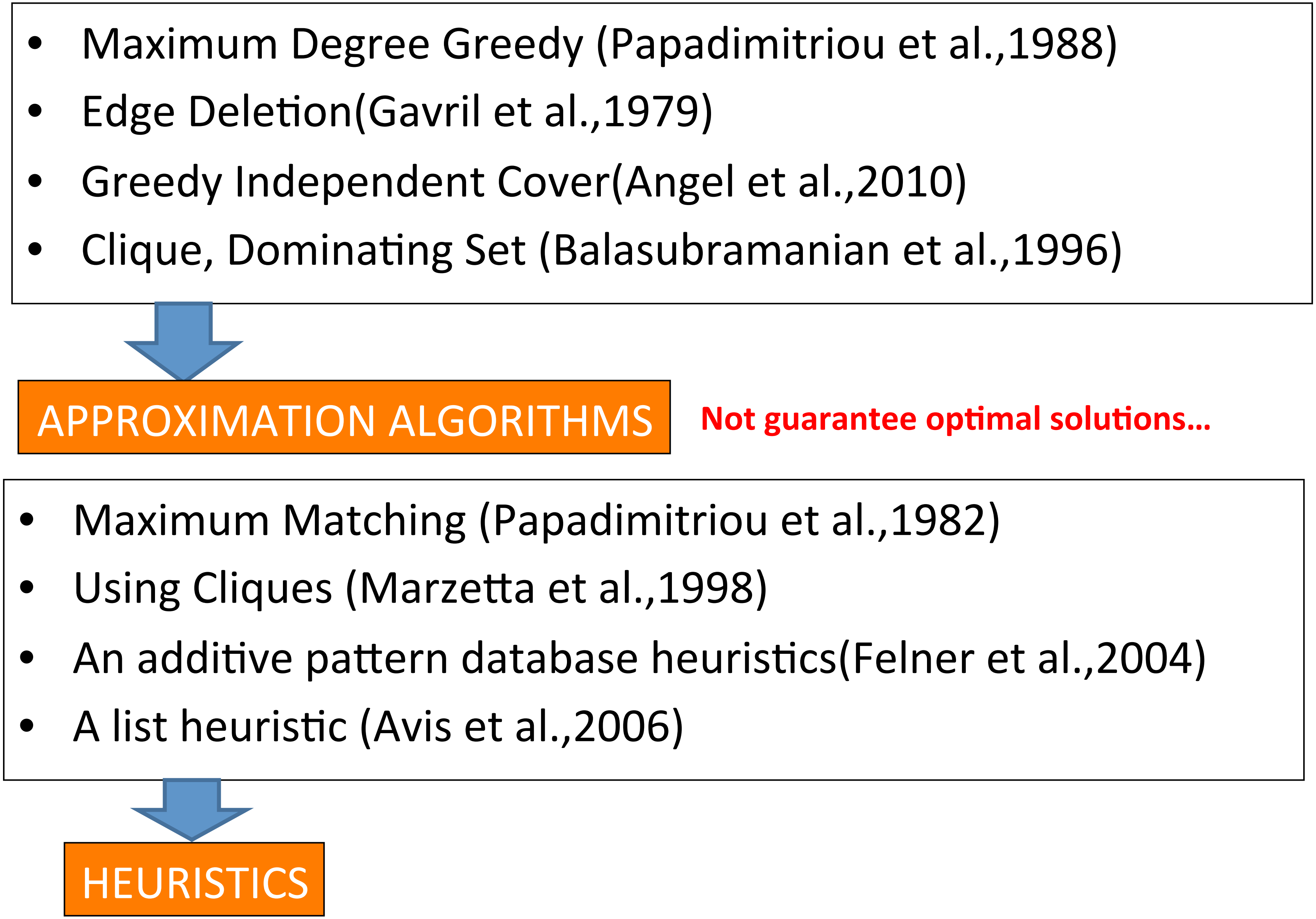
VERTEX COVER PROBLEM

✧ In Simplified Influence Maximization Problem, we have to cover all the incident edges of each node to activate all nodes.

VERTEX COVER:

- A subset of vertices C such that every edge has one of its end points in C.
- Minimum (Optimal) vertex cover is finding a minimum cardinality vertex cover.

Previous Work about Vertex Cover



- Maximum Degree Greedy (Papadimitriou et al.,1988)
- Edge Deletion(Gavril et al.,1979)
- Greedy Independent Cover(Angel et al.,2010)
- Clique, Dominating Set (Balasubramanian et al.,1996)

APPROXIMATION ALGORITHMS

Not guarantee optimal solutions…

- Maximum Matching (Papadimitriou et al.,1982)
- Using Cliques (Marzetta et al.,1998)
- An additive pattern database heuristics(Felner et al.,2004)
- A list heuristic (Avis et al.,2006)

HEURISTICS

OUTLINE

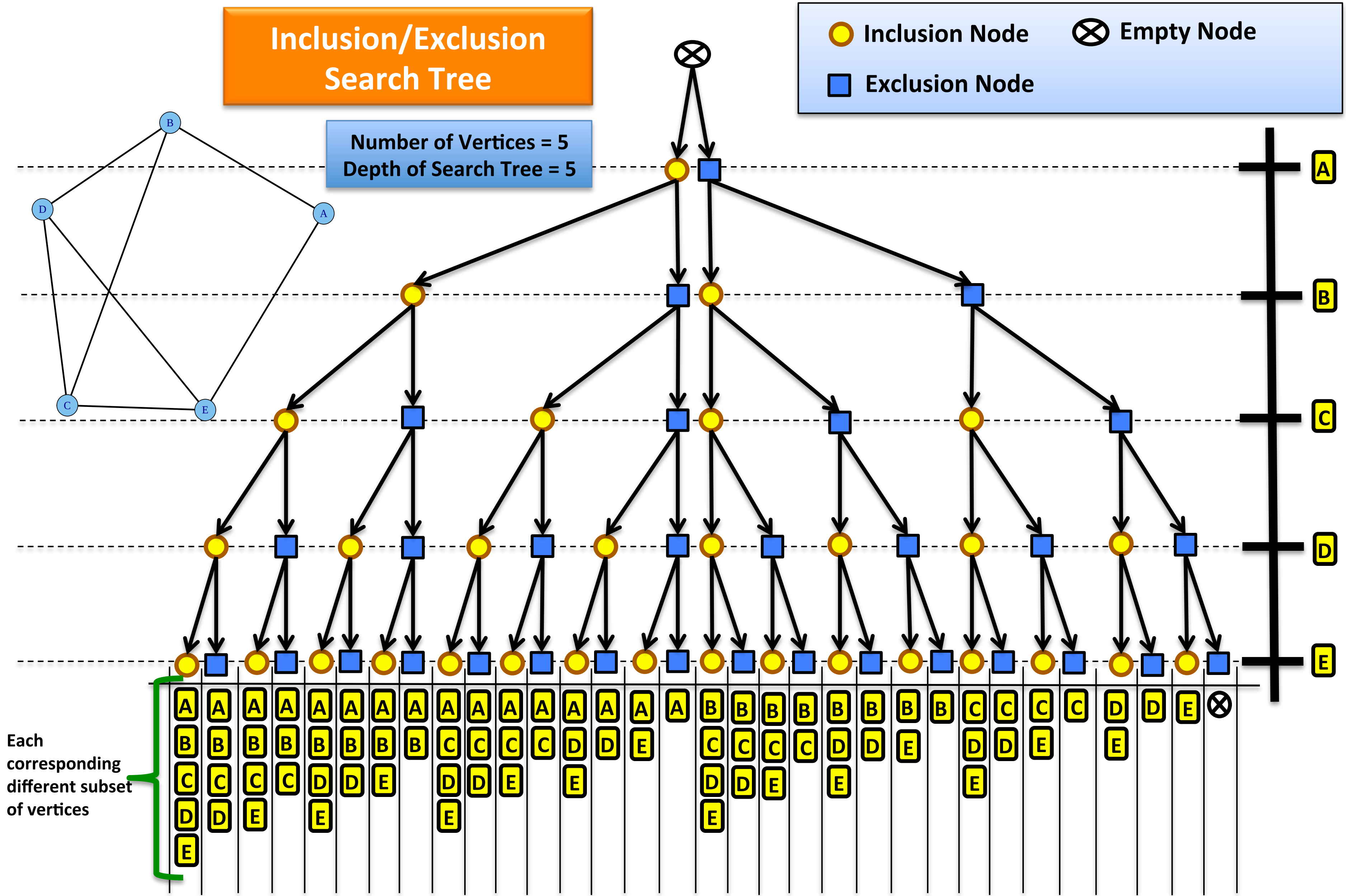
✓ Original Problem

✓ Simplified Problem

✓ Detailed Problem Space

- Pruning of Search Tree
- Node Ordering Heuristic
- Selection of Search Algorithm
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

Detailed Problem Space



Inclusion/Exclusion Search Tree

Number of Vertices = 5
Depth of Search Tree = 5

● Inclusion Node
⊗ Empty Node
■ Exclusion Node

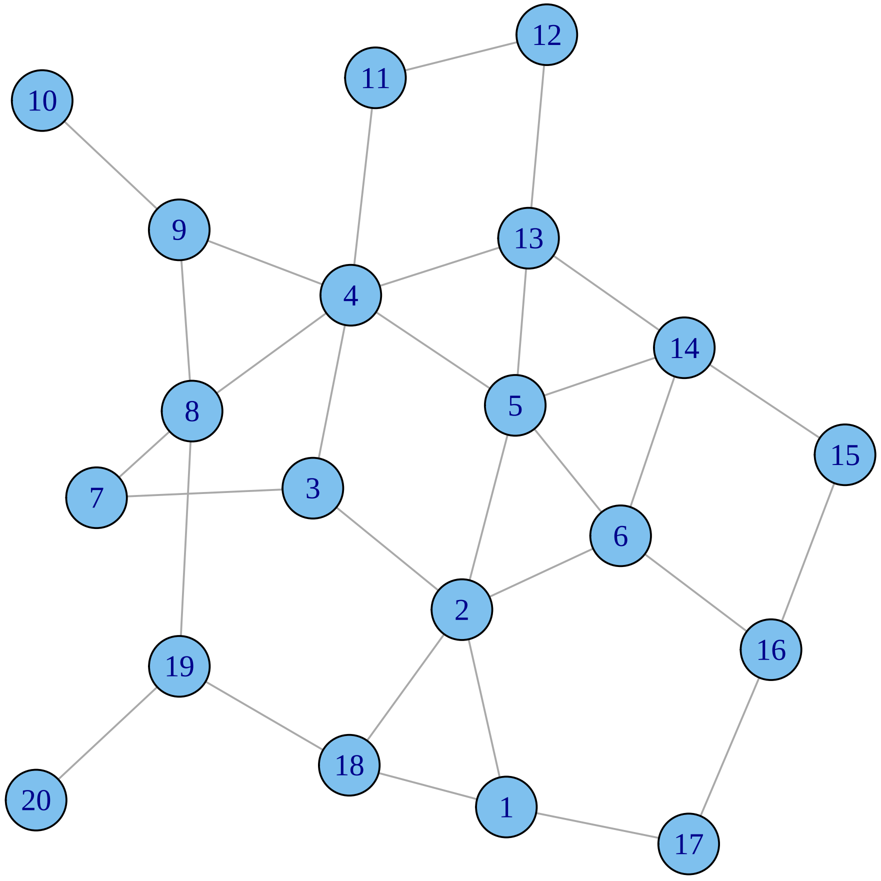
Each corresponding different subset of vertices

OUTLINE

- ✓ Original Problem
- ✓ Simplified Problem
- ✓ Detailed Problem Space
- ✓ Pruning of Search Tree
- Node Ordering Heuristic
- Selection of Search Algorithm
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

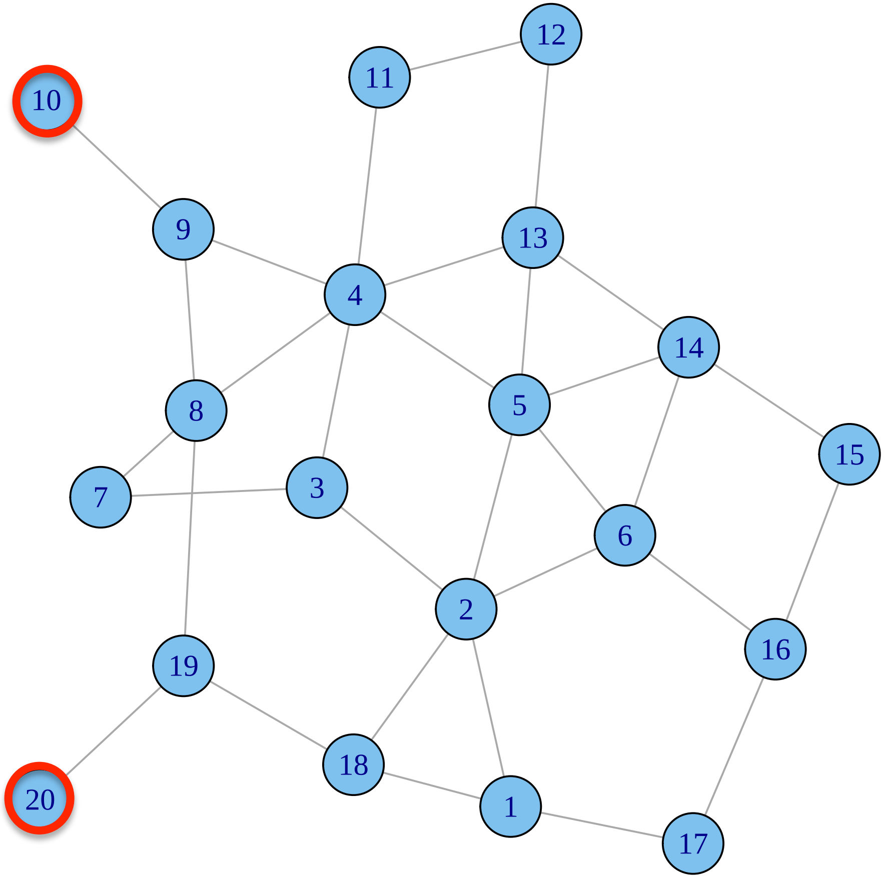
Pre-evaluation of Graph

- If a vertex has only one neighbor, we can exclude that vertex and include its neighbor in our vertex cover set.



Pre-evaluation of Graph

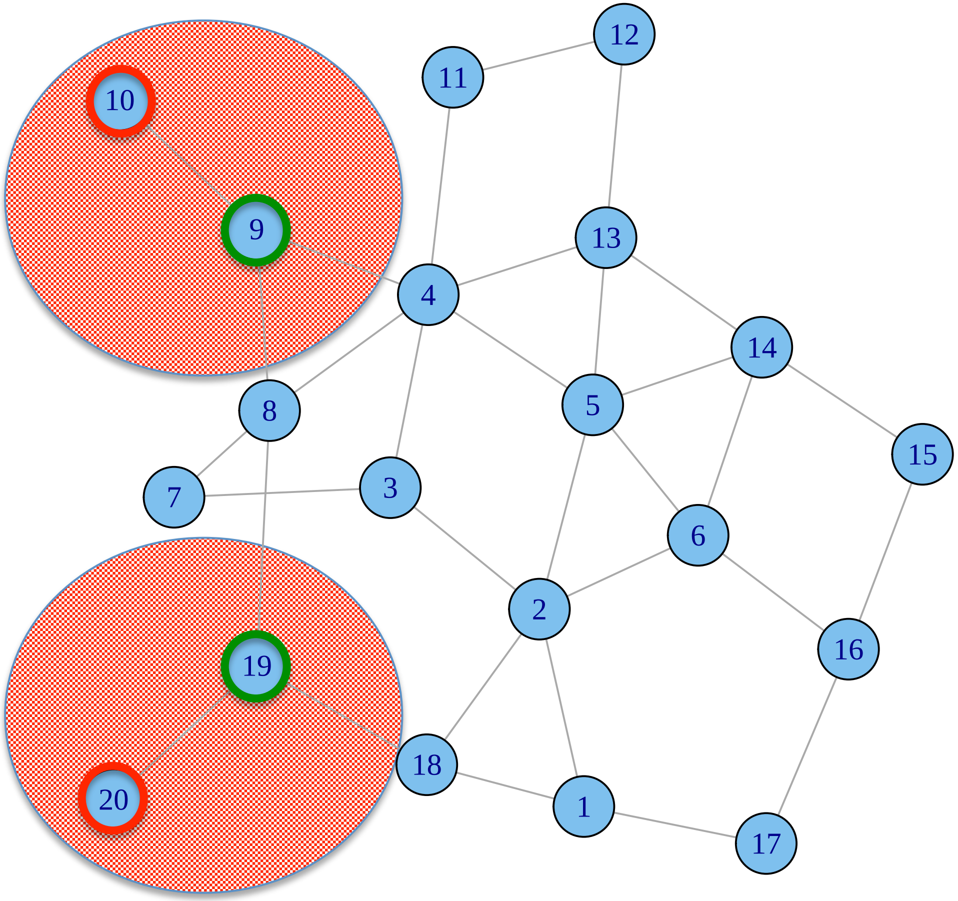
- If a vertex has only one neighbor, we can exclude that vertex and include its neighbor in our vertex cover set.



Pre-evaluation of Graph

- If a vertex has only one neighbor, we can exclude that vertex and include its neighbor in our vertex cover set.

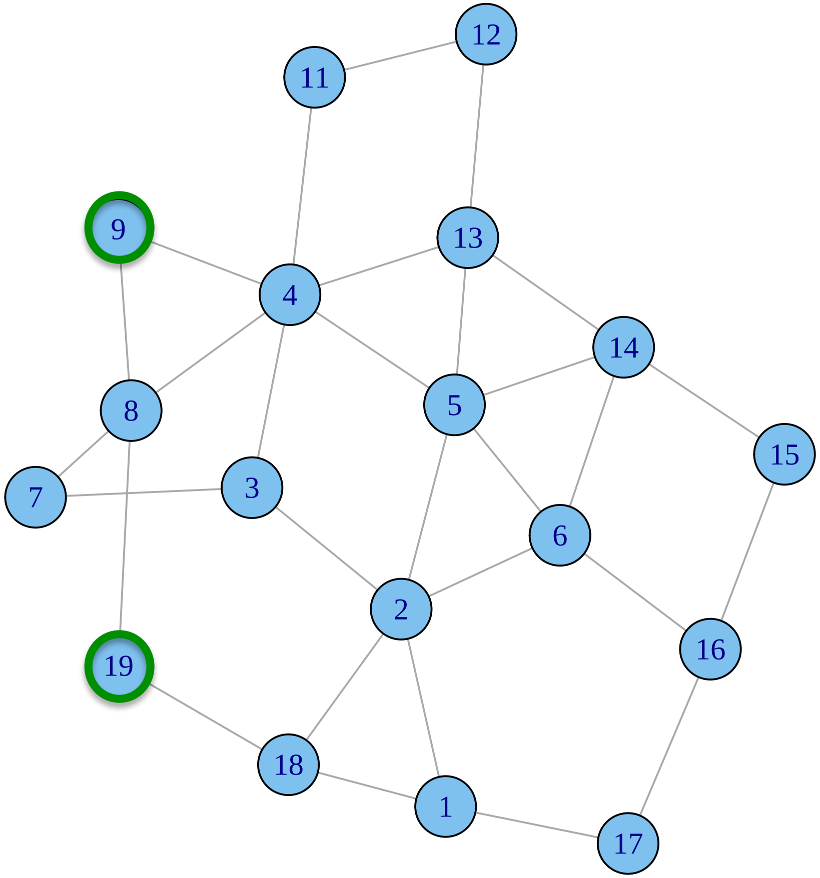
VC: {9,19,….}



Pre-evaluation of Graph

- If a vertex has only one neighbor, we can exclude that vertex and include its neighbor in our vertex cover set.

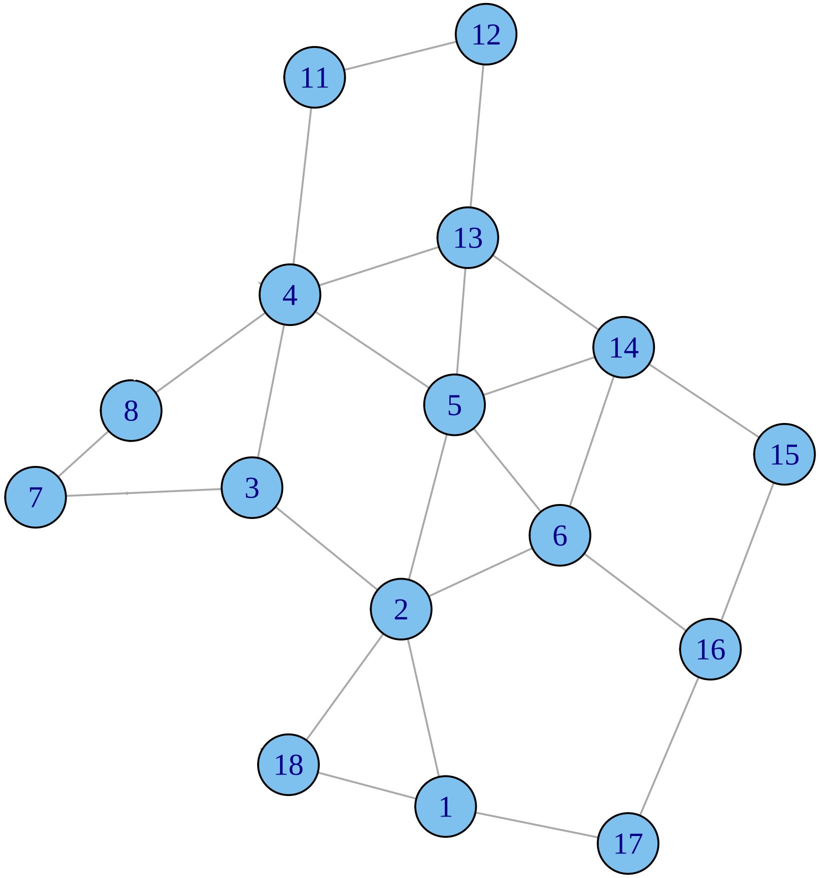
VC: {9,19,….}



Pre-evaluation of Graph

- If a vertex has only one neighbor, we can exclude that vertex and include its neighbor in our vertex cover set.

VC: {9,19,….}



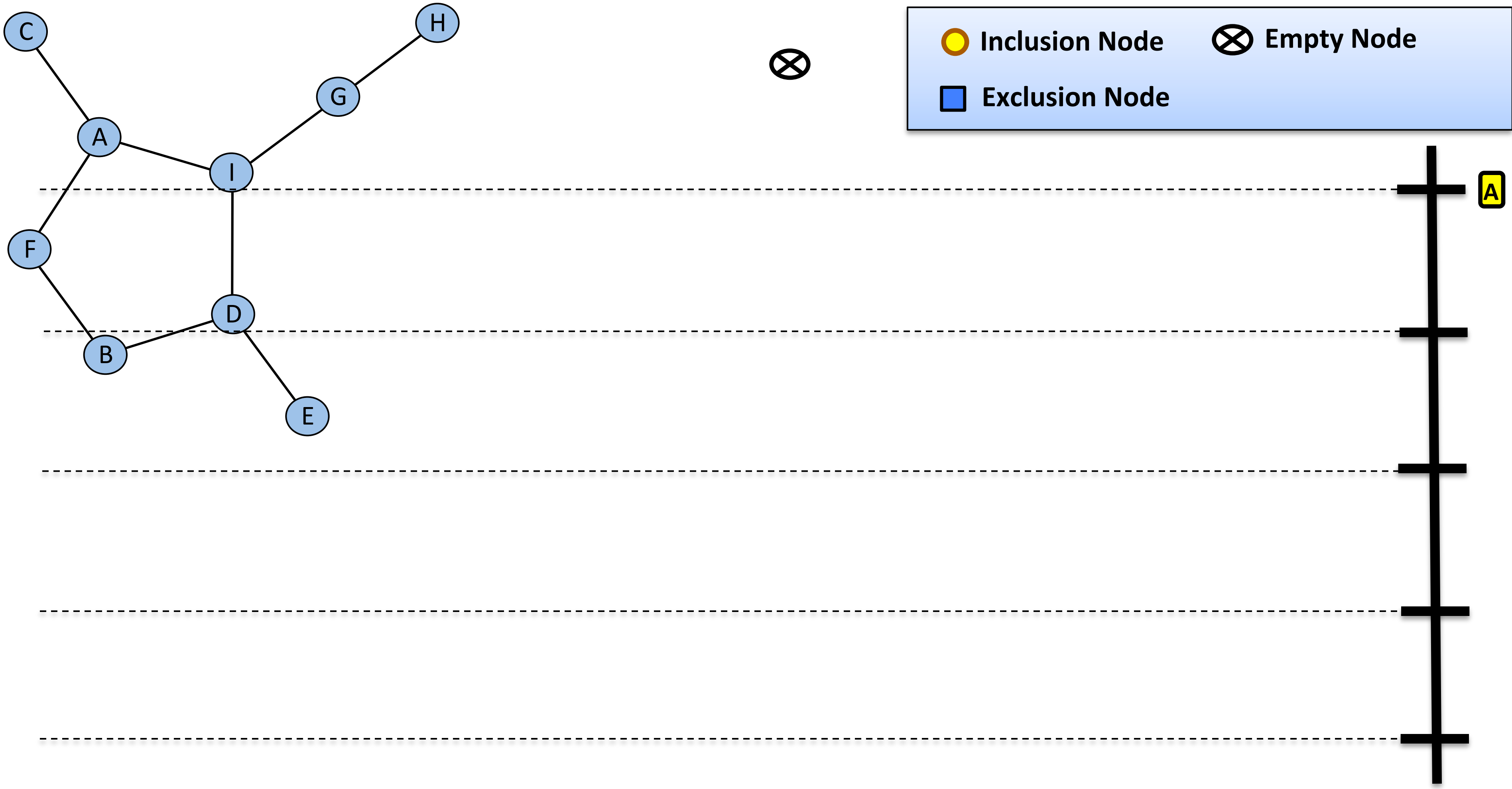
Pruning Search Tree

✓ Exclusion Pruning

– Inclusion Pruning

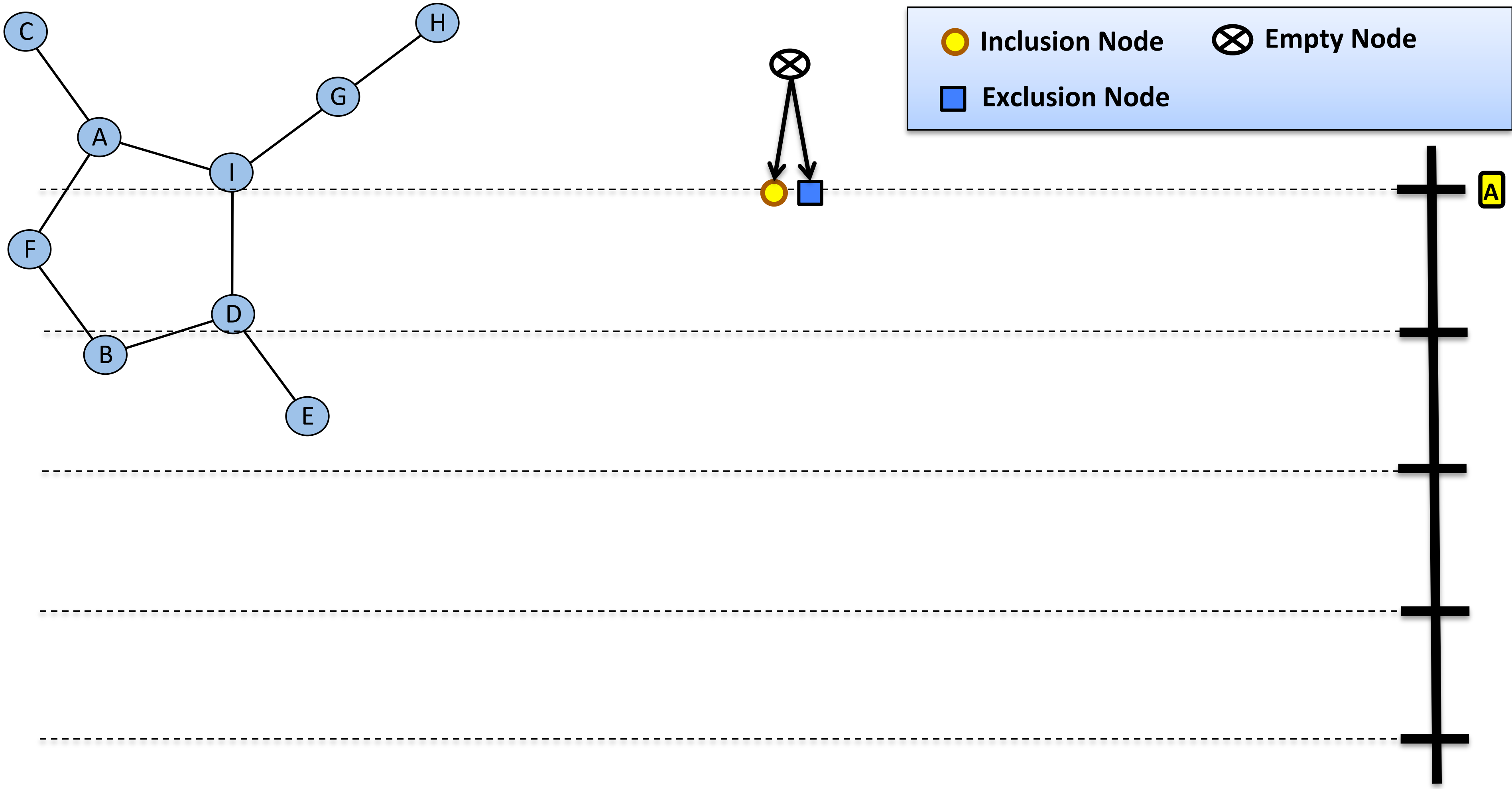
– k-Bound Pruning

Exclusion Pruning



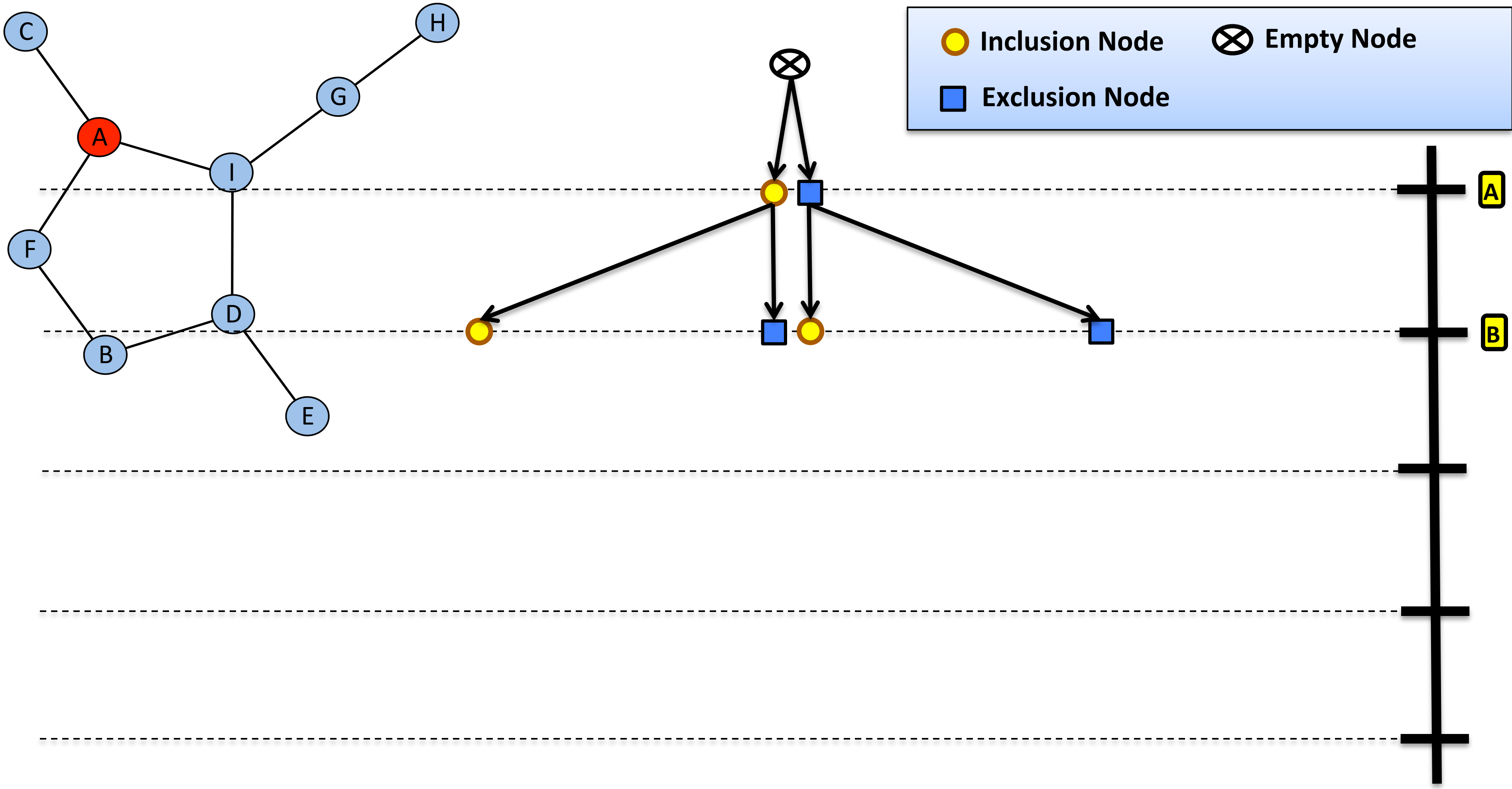
- If a node is excluded from the vertex cover set, for covering the edges between them all incident nodes have to be included. In these situations, branches which don’t include these incident vertices are pruned.

Exclusion Pruning



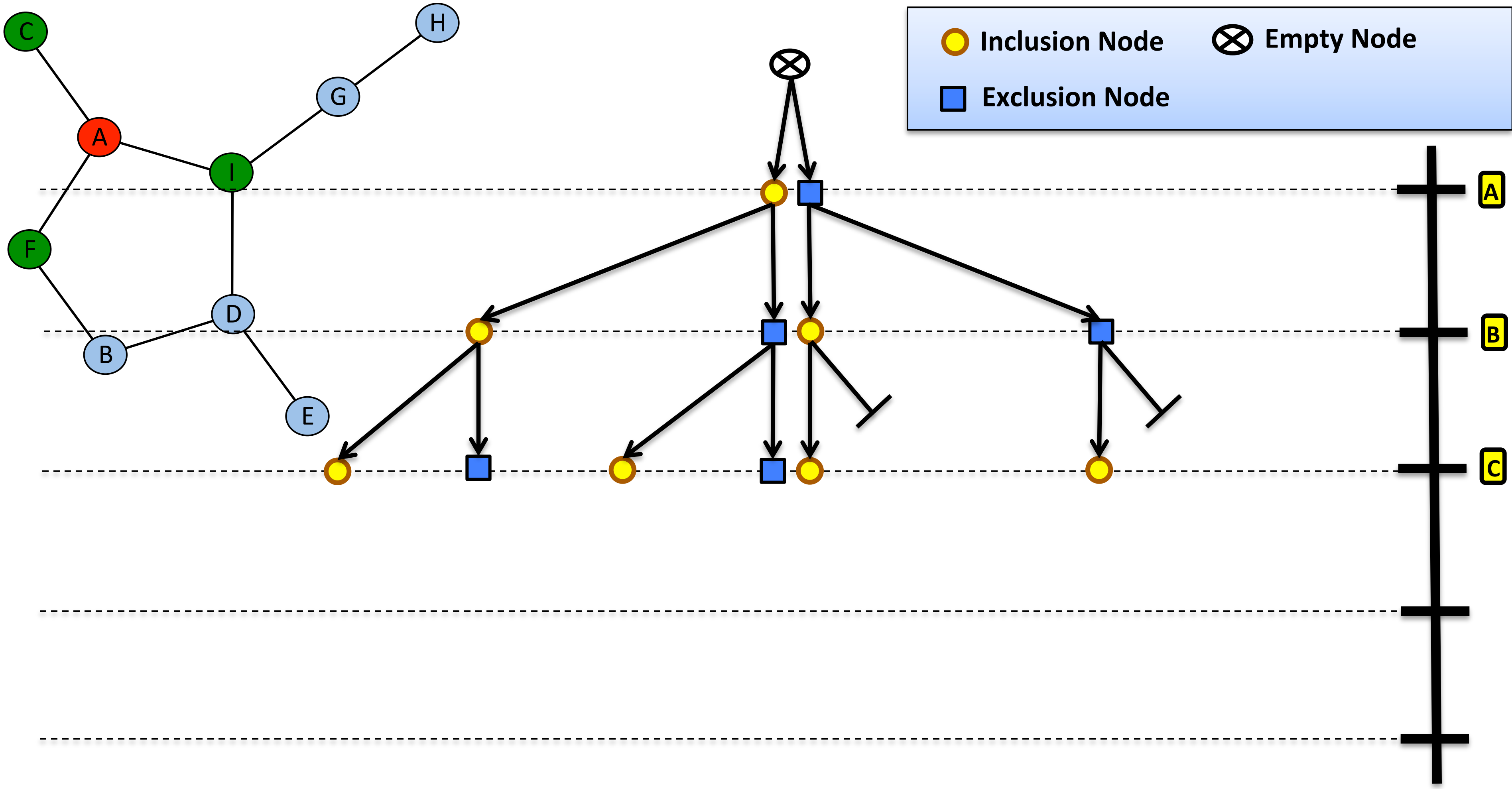
- If a node is excluded from the vertex cover set, for covering the edges between them all incident nodes have to be included. In these situations, branches which don’t include these incident vertices are pruned.

Exclusion Pruning



- If a node is excluded from the vertex cover set, for covering the edges between them all incident nodes have to be included. In these situations, branches which don’t include these incident vertices are pruned.

Exclusion Pruning

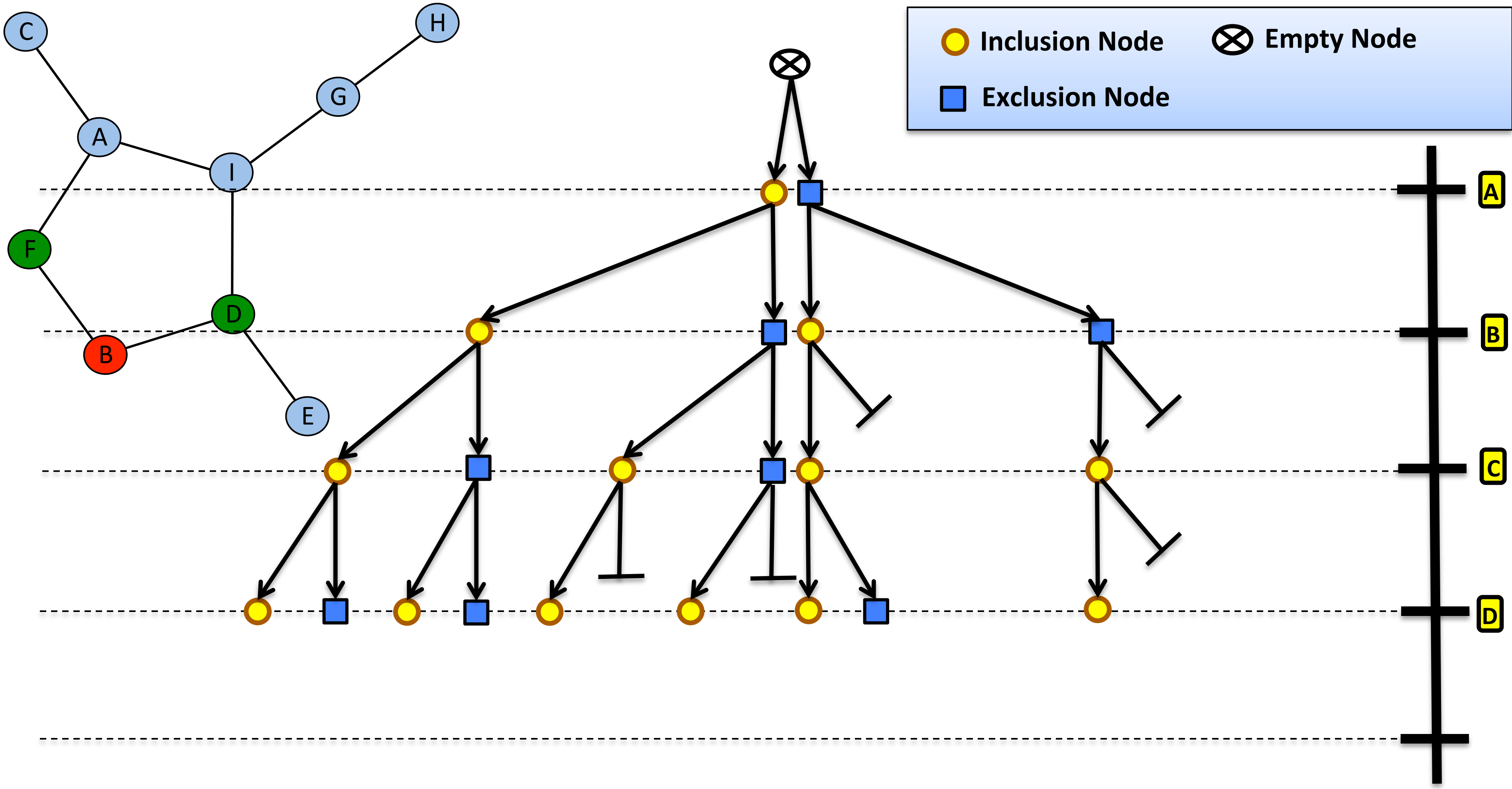


● Inclusion Node ⊗ Empty Node

■ Exclusion Node

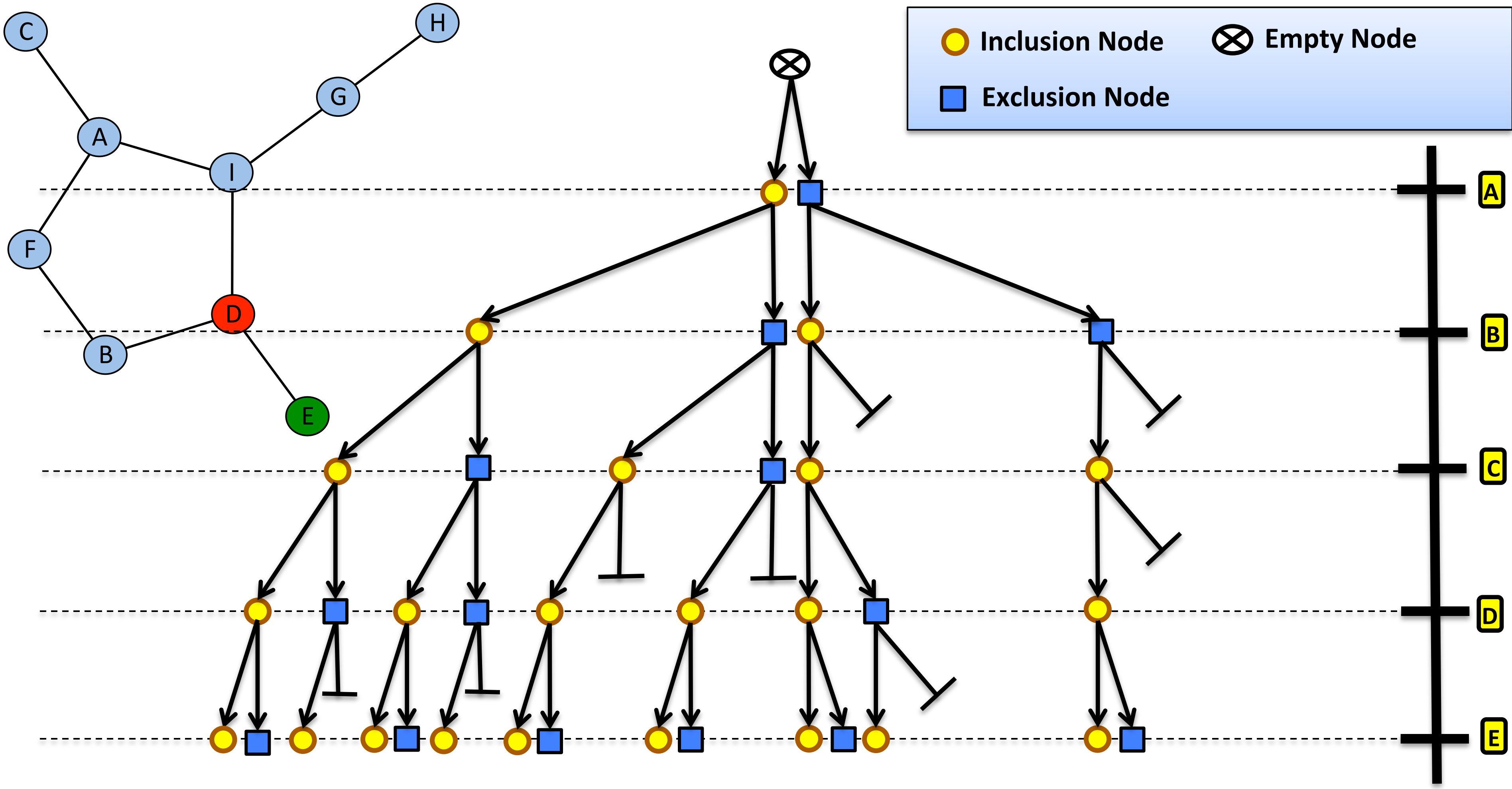
- If a node is excluded from the vertex cover set, for covering the edges between them all incident nodes have to be included. In these situations, branches which don’t include these incident vertices are pruned.

Exclusion Pruning



- If a node is excluded from the vertex cover set, for covering the edges between them all incident nodes have to be included. In these situations, branches which don’t include these incident vertices are pruned.

Exclusion Pruning



- If a node is excluded from the vertex cover set, for covering the edges between them all incident nodes have to be included. In these situations, branches which don't include these incident vertices are pruned.

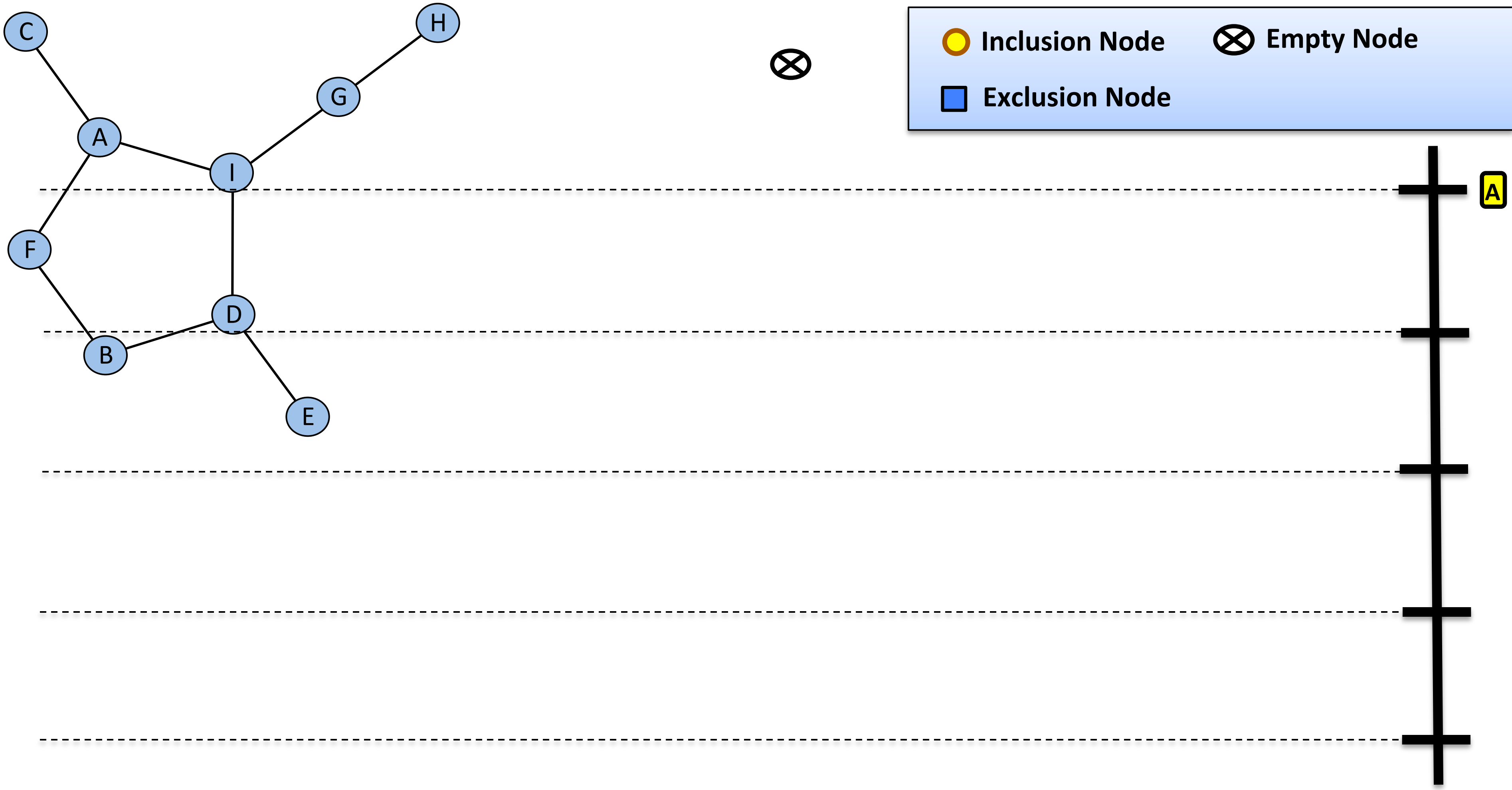
Pruning Search Tree

✓Exclusion Pruning

✓Inclusion Pruning

– k-Bound Pruning

Inclusion Pruning

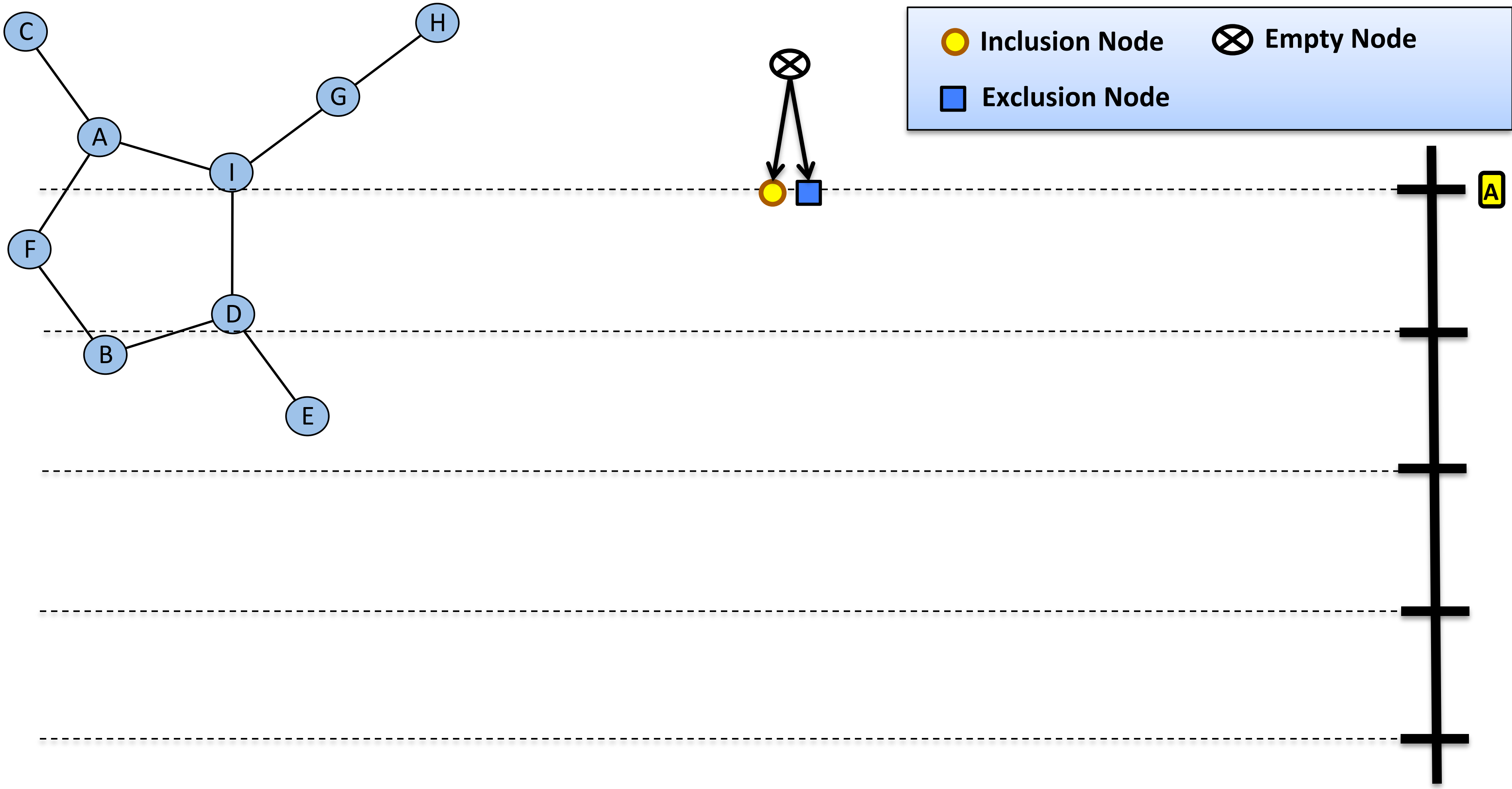


● Inclusion Node ⊗ Empty Node

■ Exclusion Node

- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning

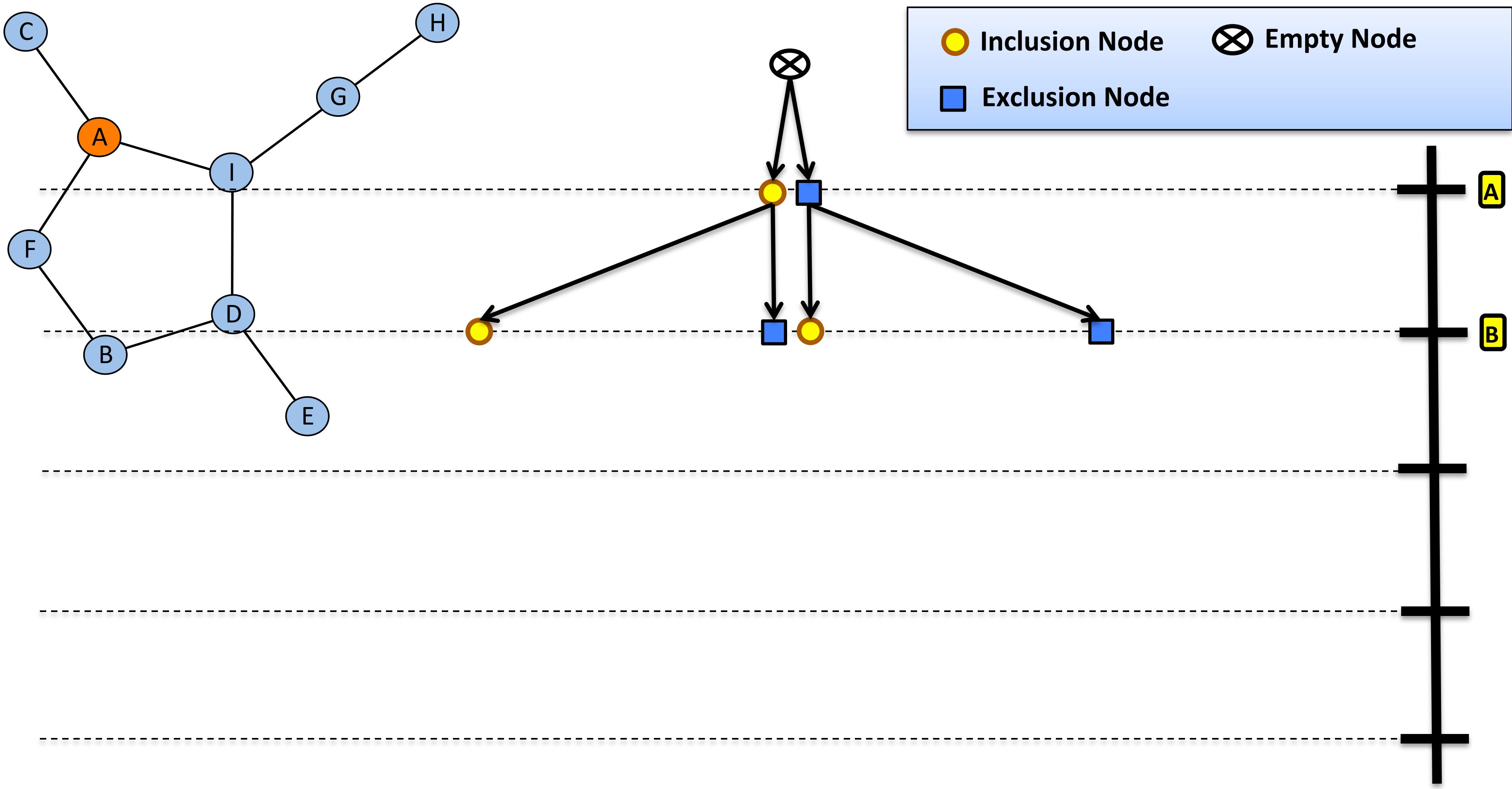


● Inclusion Node ⊗ Empty Node

■ Exclusion Node

- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning

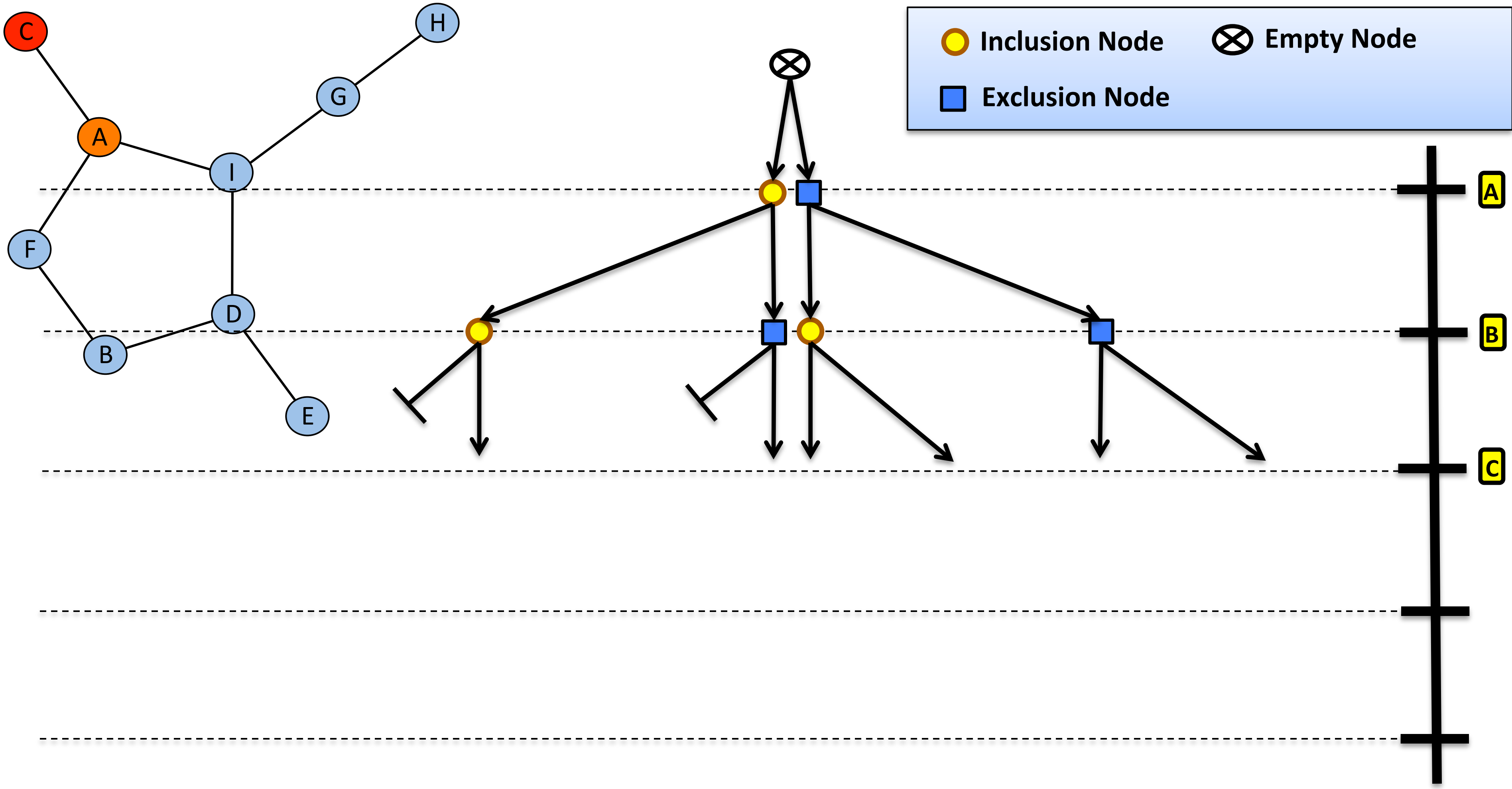


● Inclusion Node ⊗ Empty Node

■ Exclusion Node

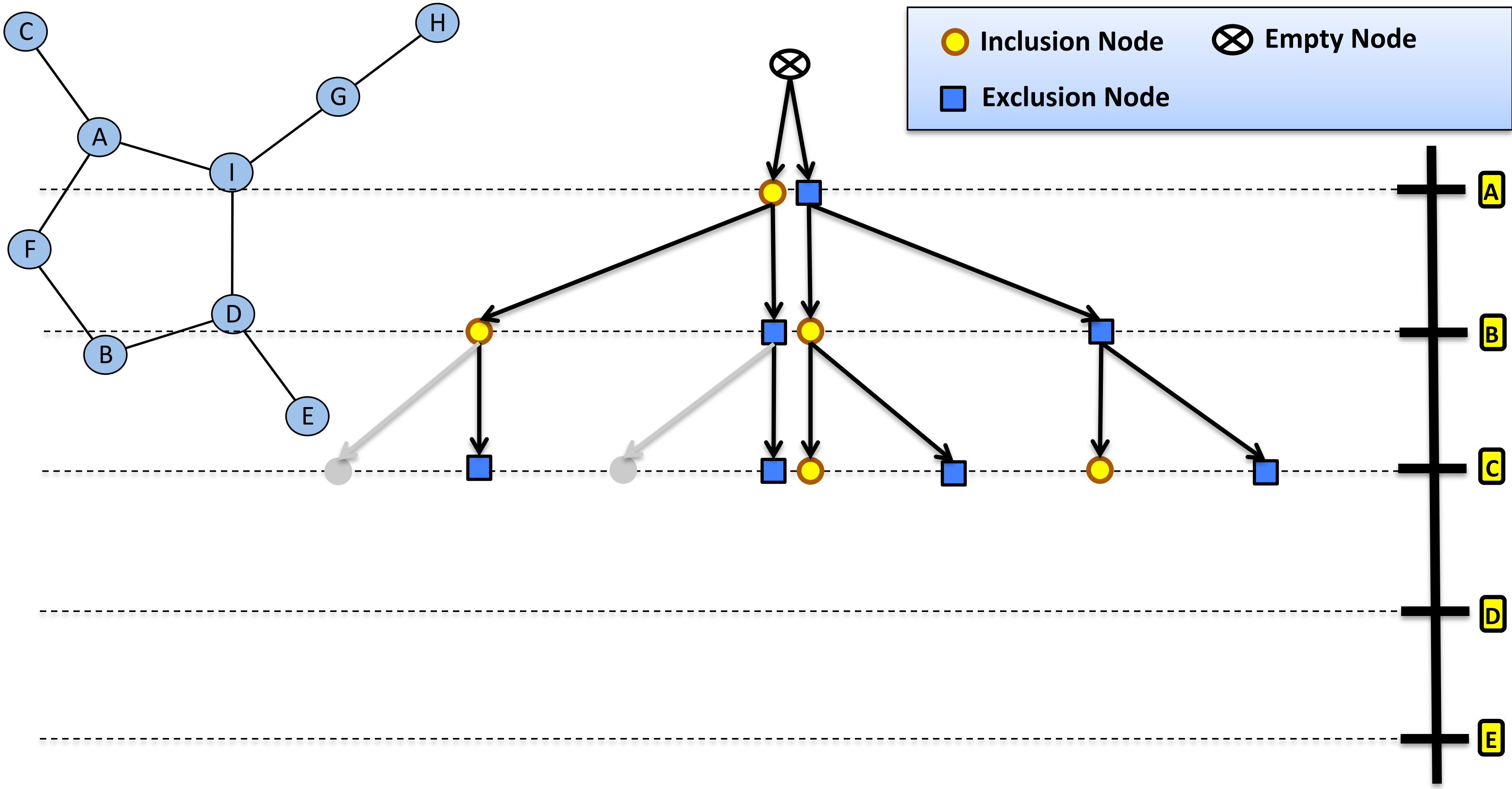
- If a node is included, all of its incident edges are covered. So we don’t need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



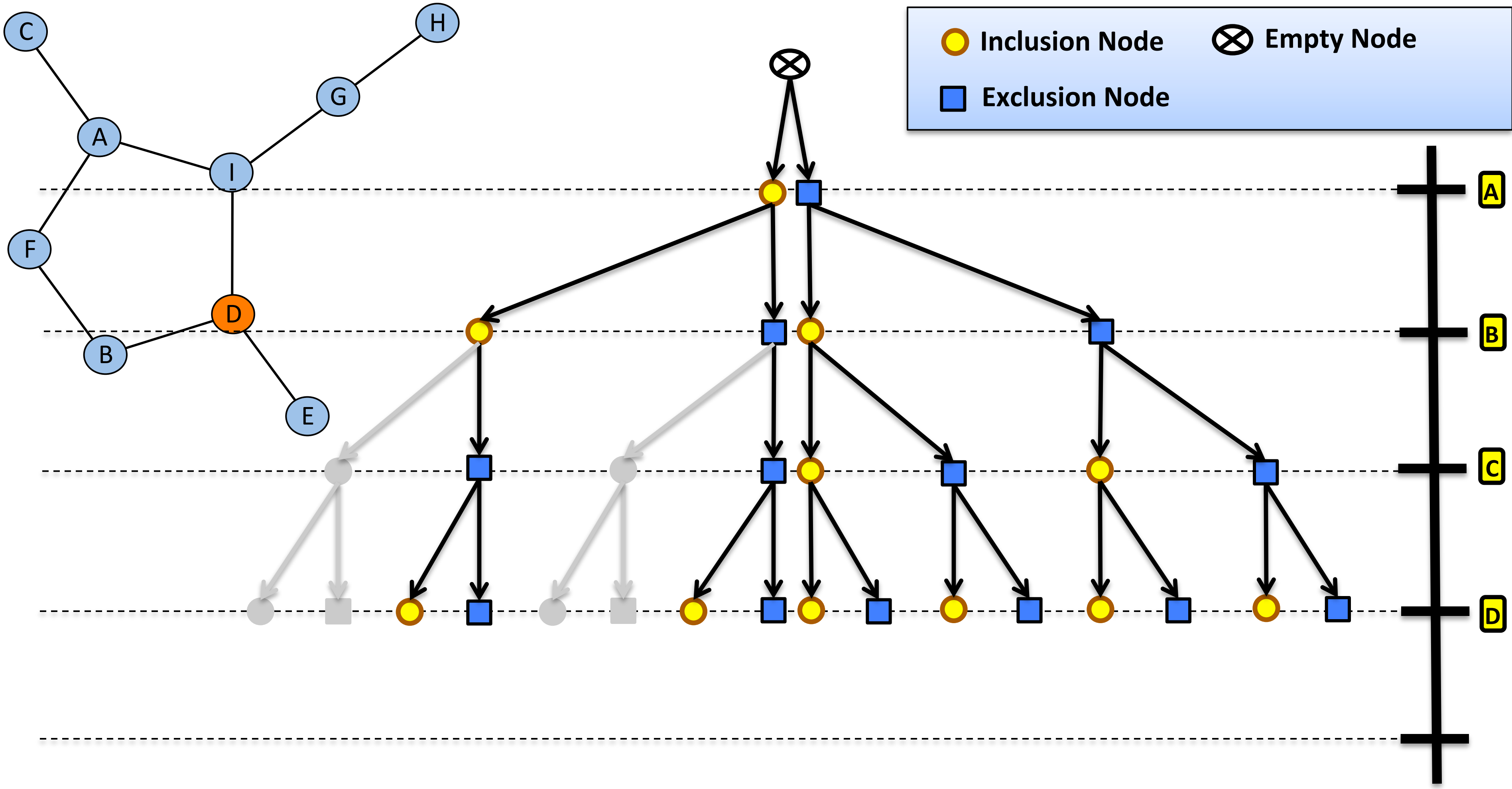
- If a node is included, all of its incident edges are covered. So we don’t need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



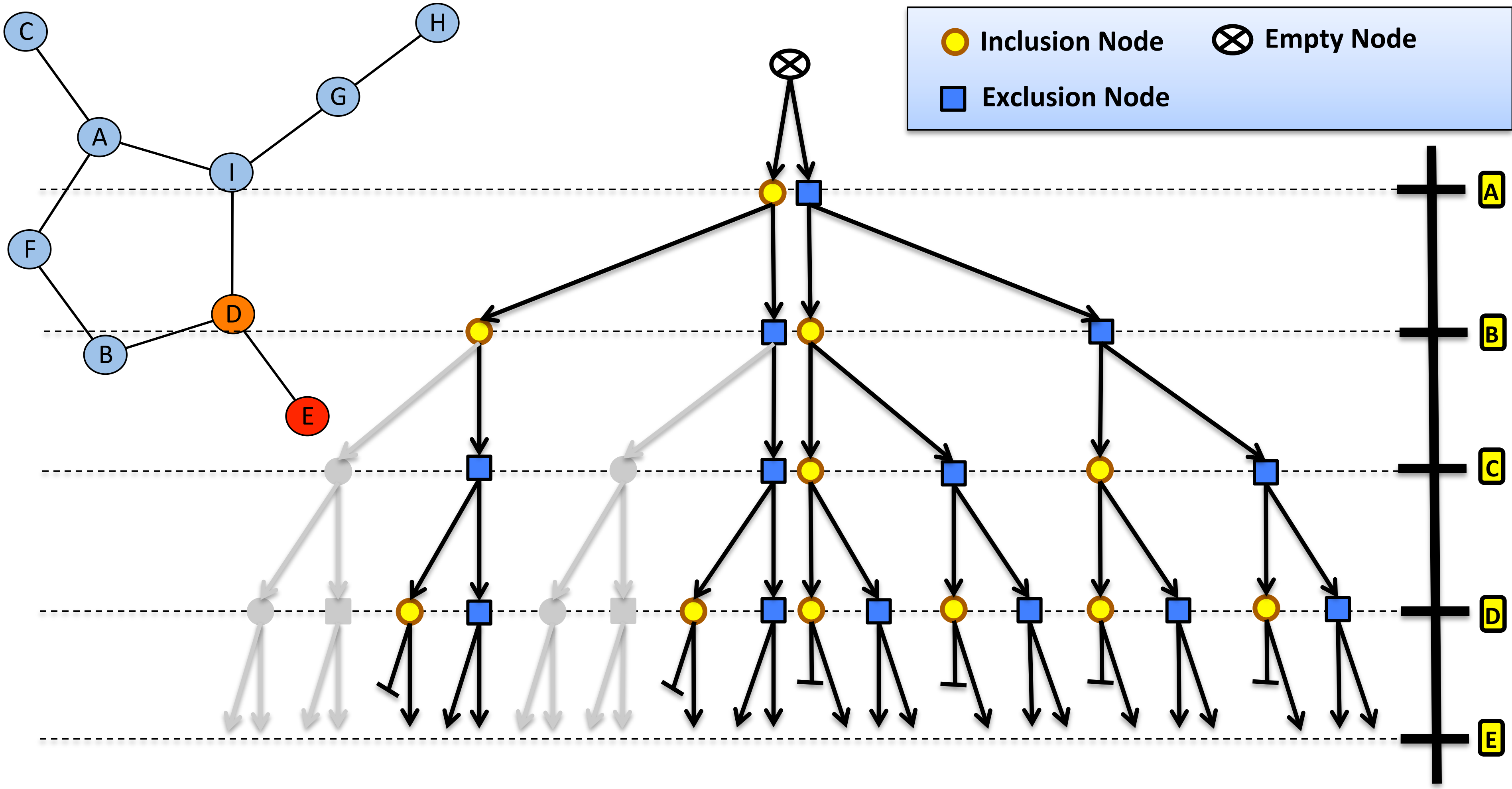
- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



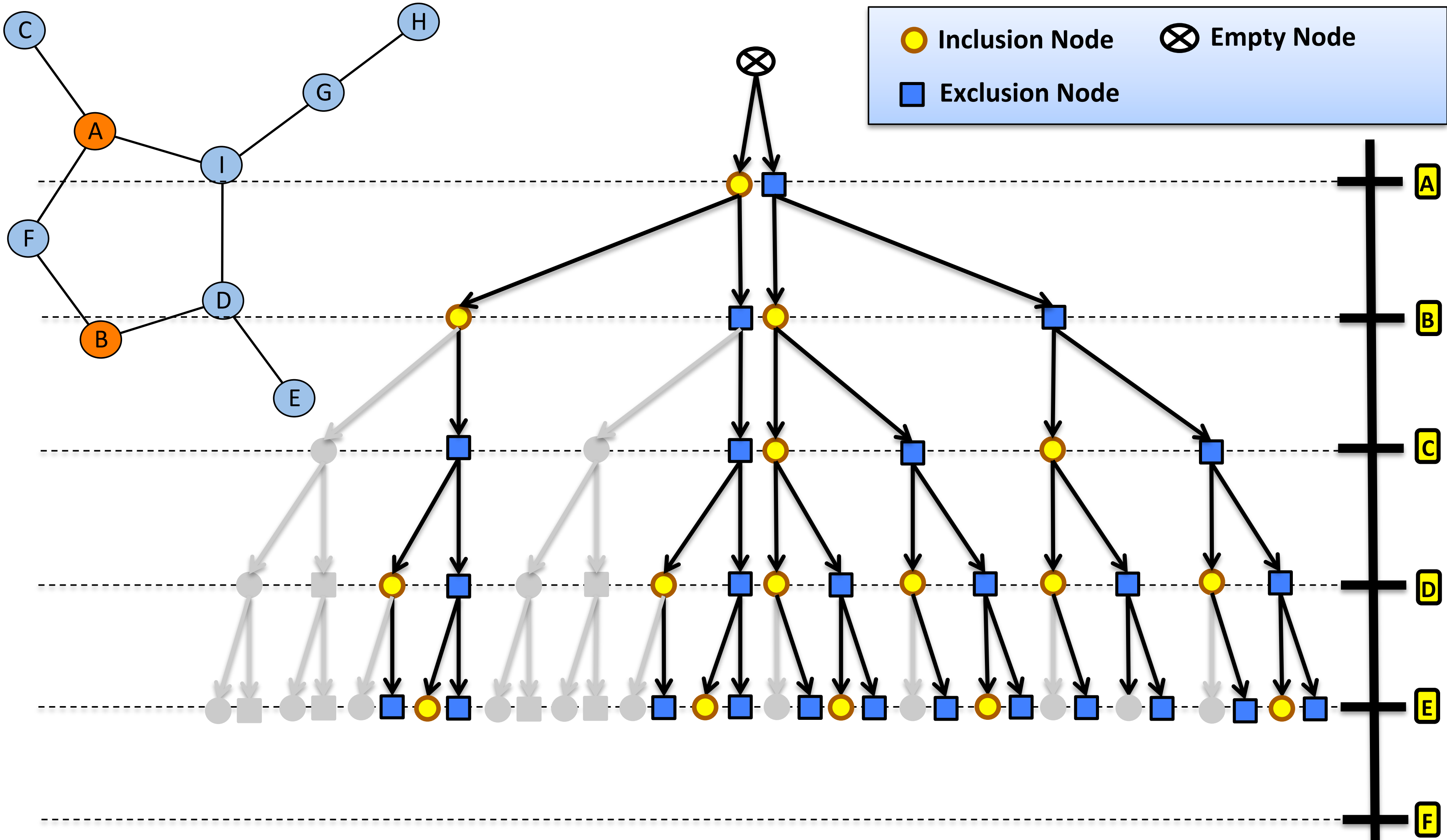
- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



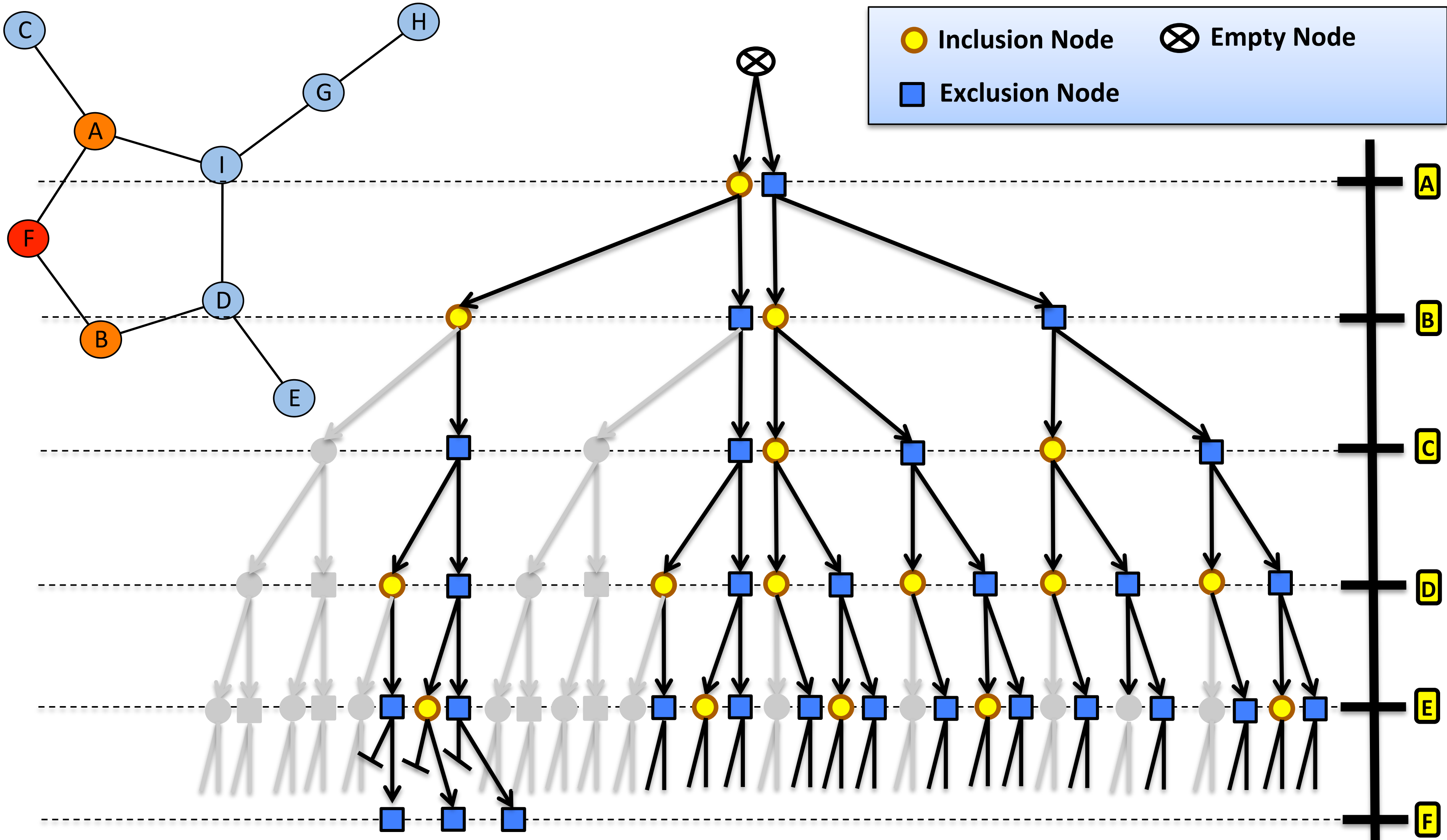
- If a node is included, all of its incident edges are covered. So we don’t need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



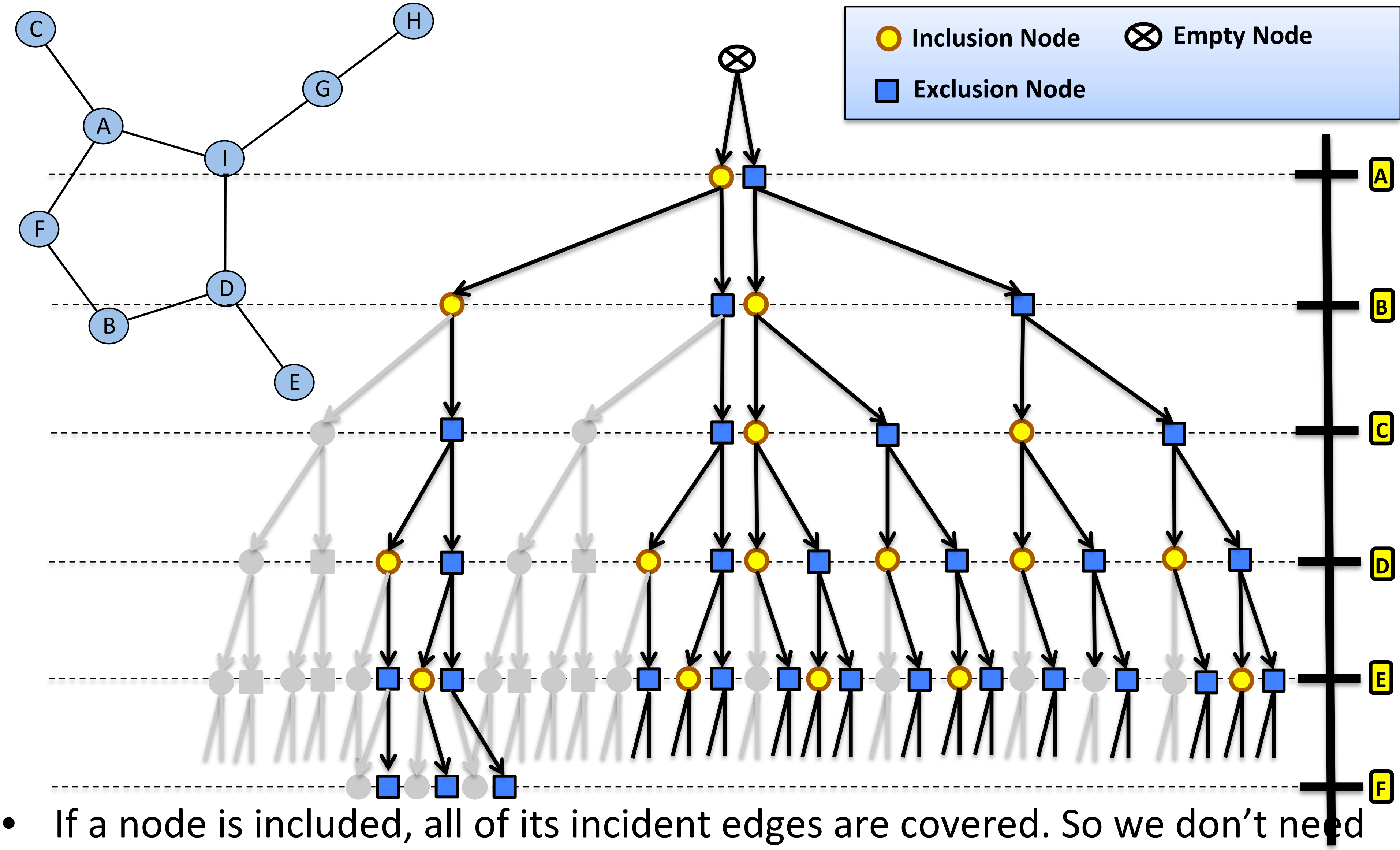
- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

Inclusion Pruning



- If a node is included, all of its incident edges are covered. So we don't need to include nodes whose incident edges covered by other included nodes.

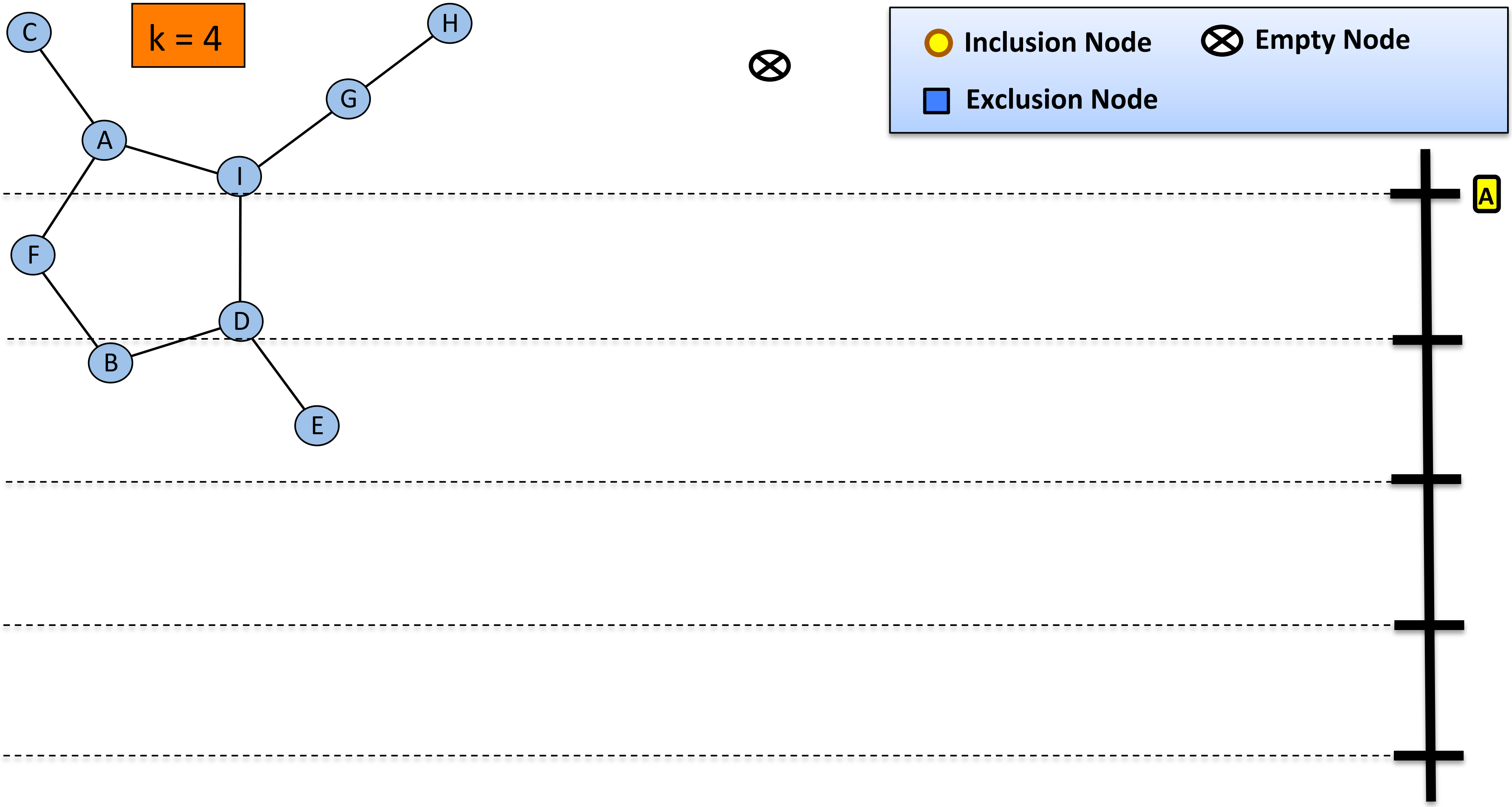
Pruning Search Tree

✓ Exclusion Pruning

✓ Inclusion Pruning

✓ k-Bound Pruning

k-Bound Pruning



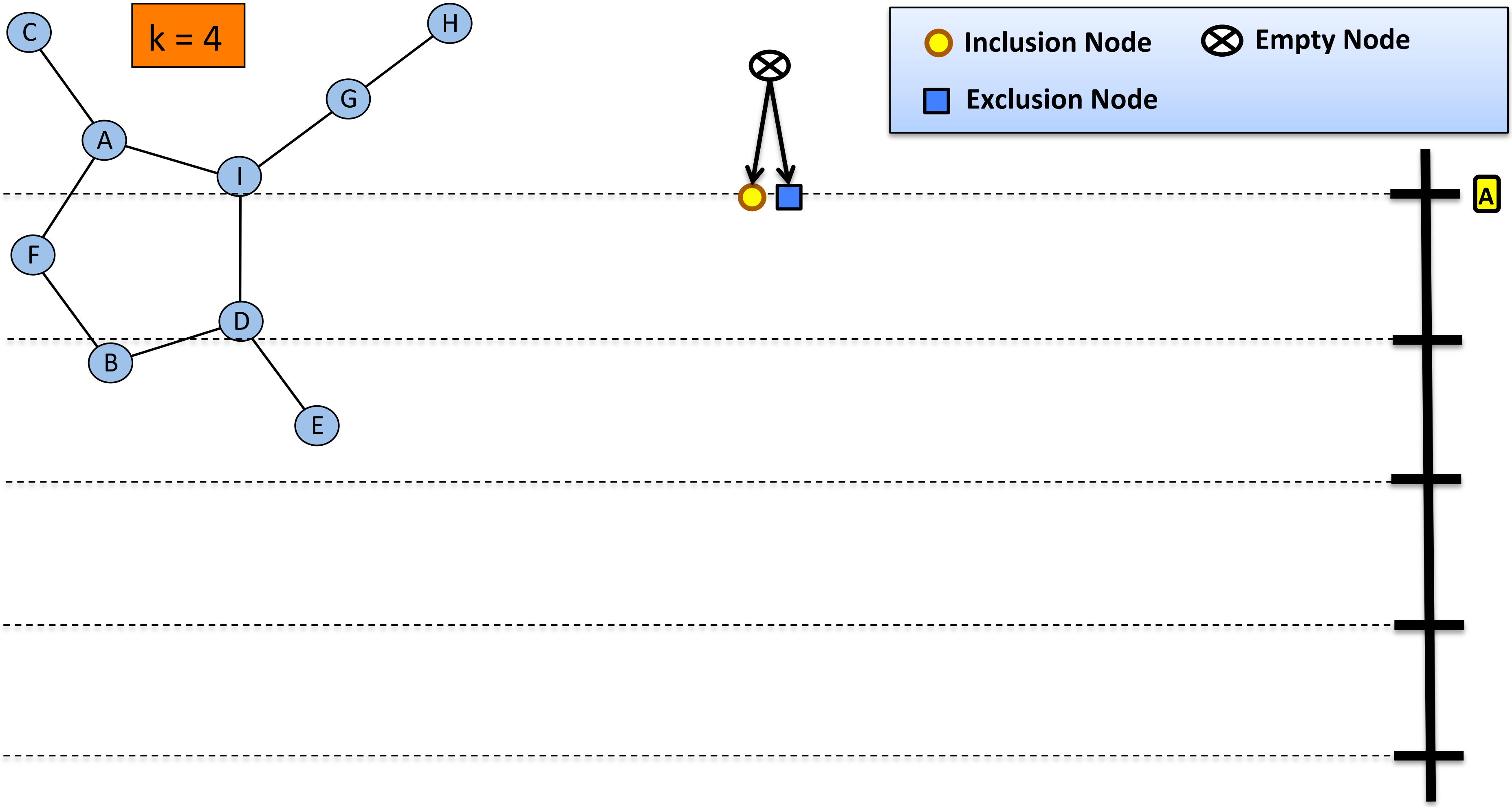
k = 4

● Inclusion Node

⊗ Empty Node

■ Exclusion Node

k-Bound Pruning



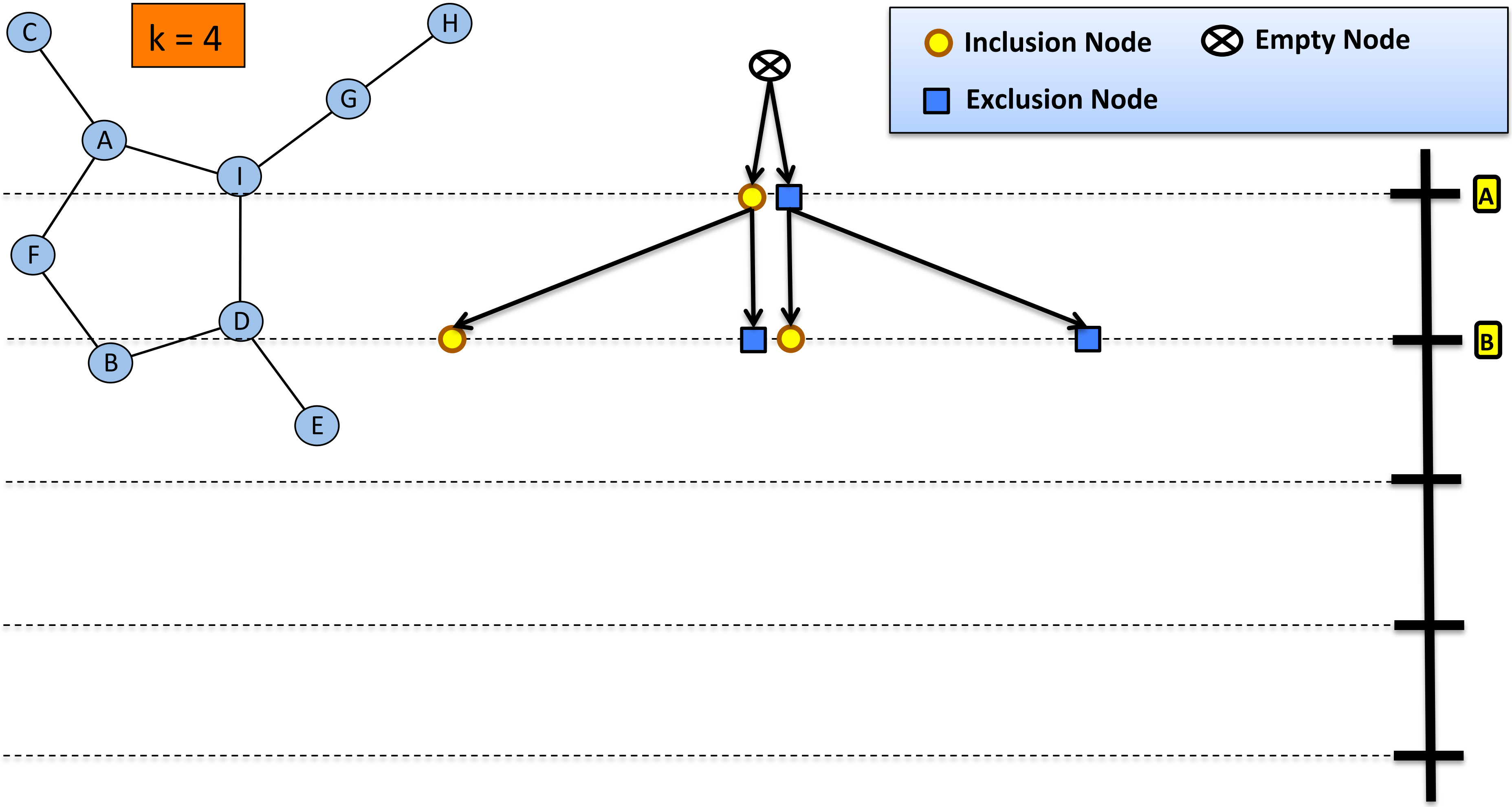
k = 4

● Inclusion Node

⊗ Empty Node

■ Exclusion Node

k-Bound Pruning

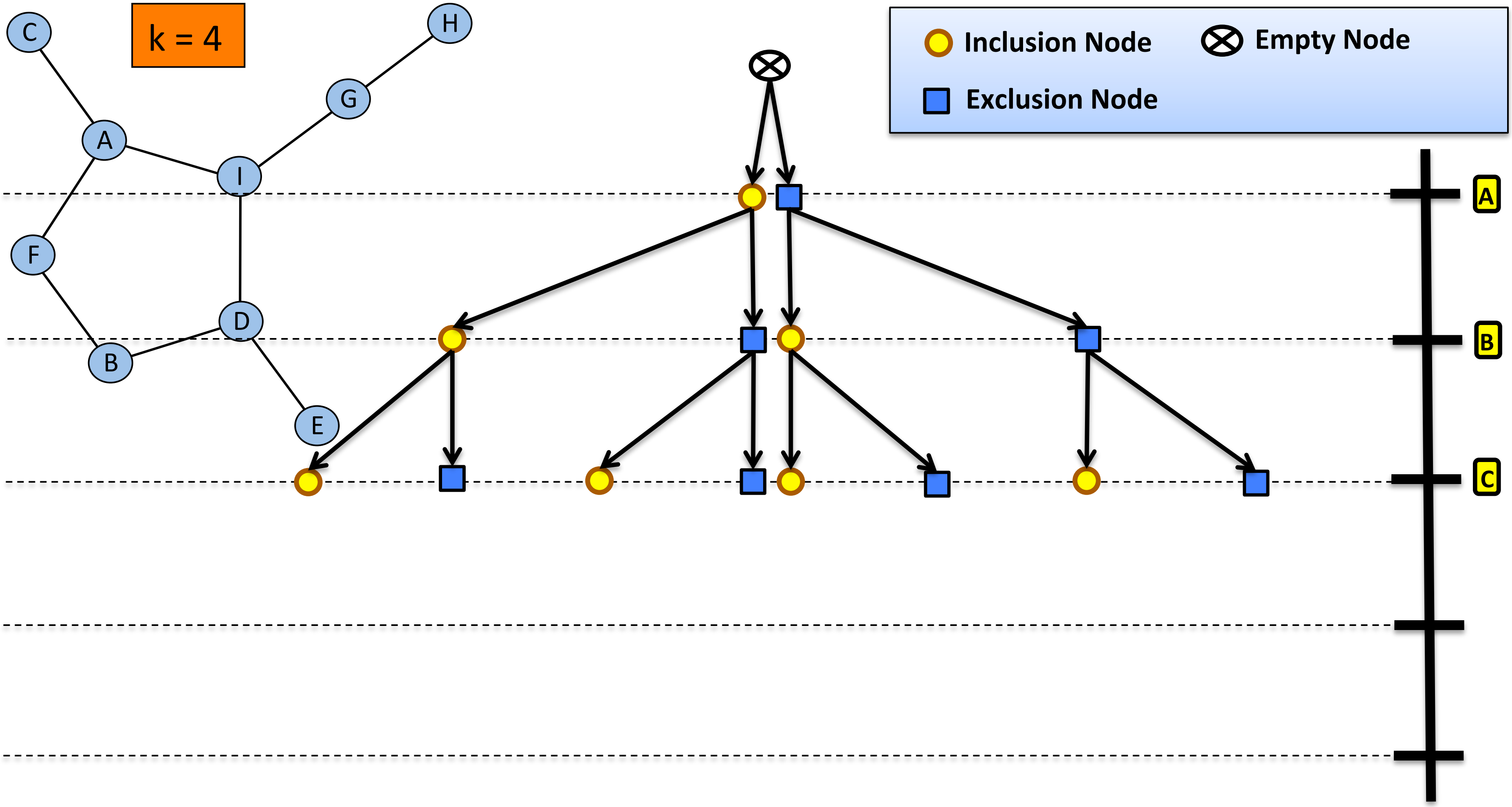


● Inclusion Node

⊗ Empty Node

■ Exclusion Node

k-Bound Pruning



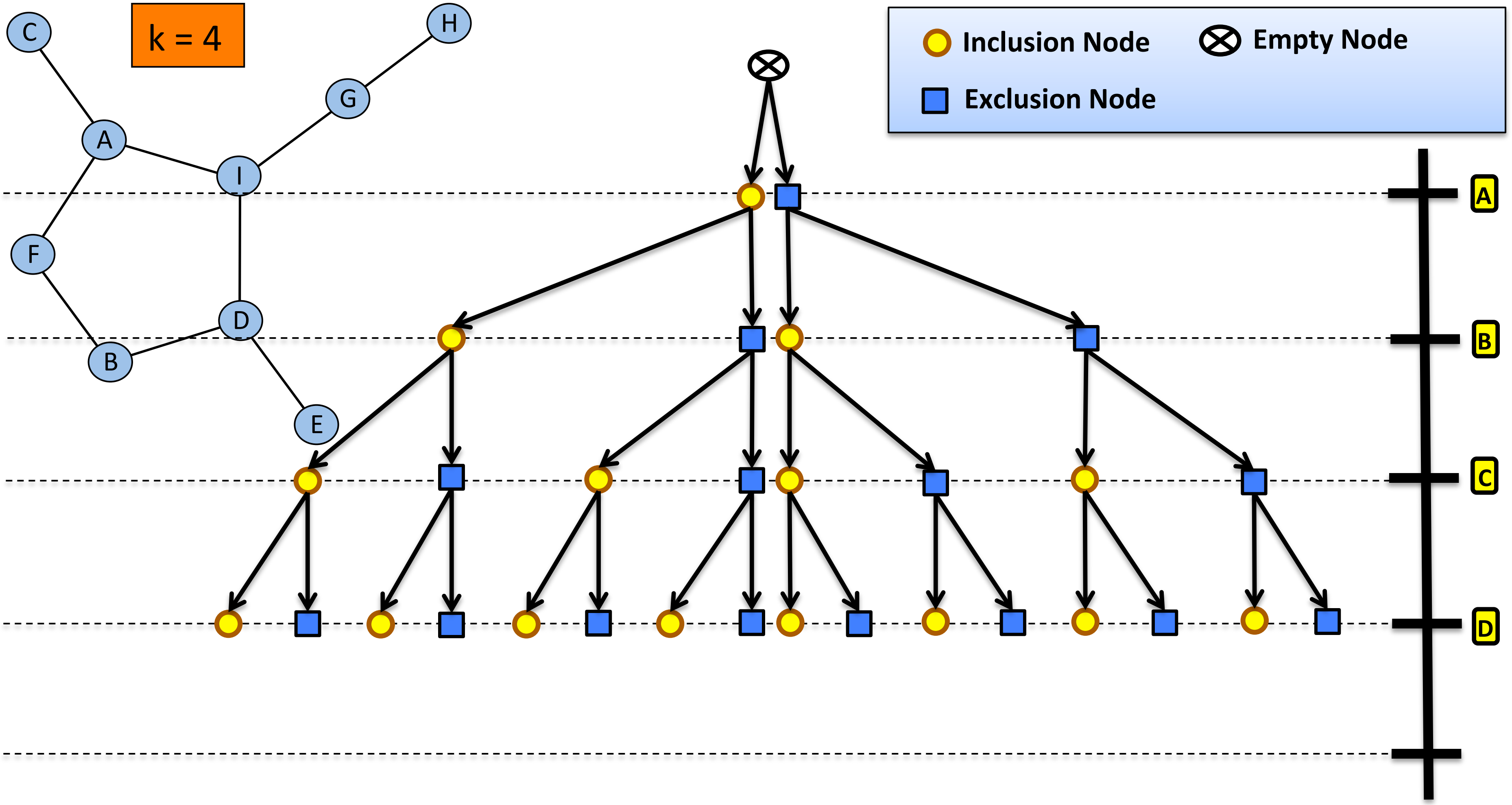
k = 4

● Inclusion Node

⊗ Empty Node

■ Exclusion Node

k-Bound Pruning



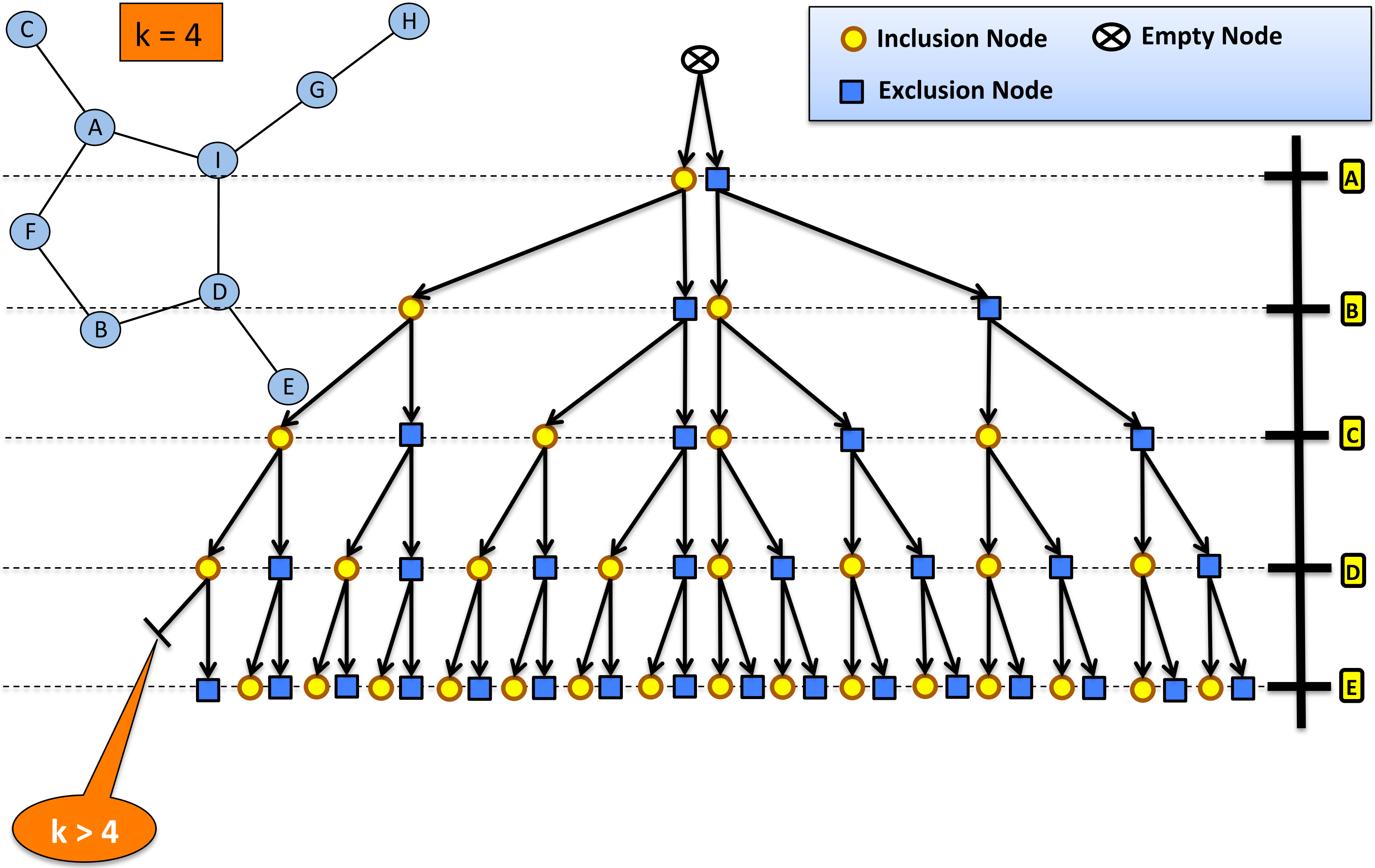
k = 4

● Inclusion Node

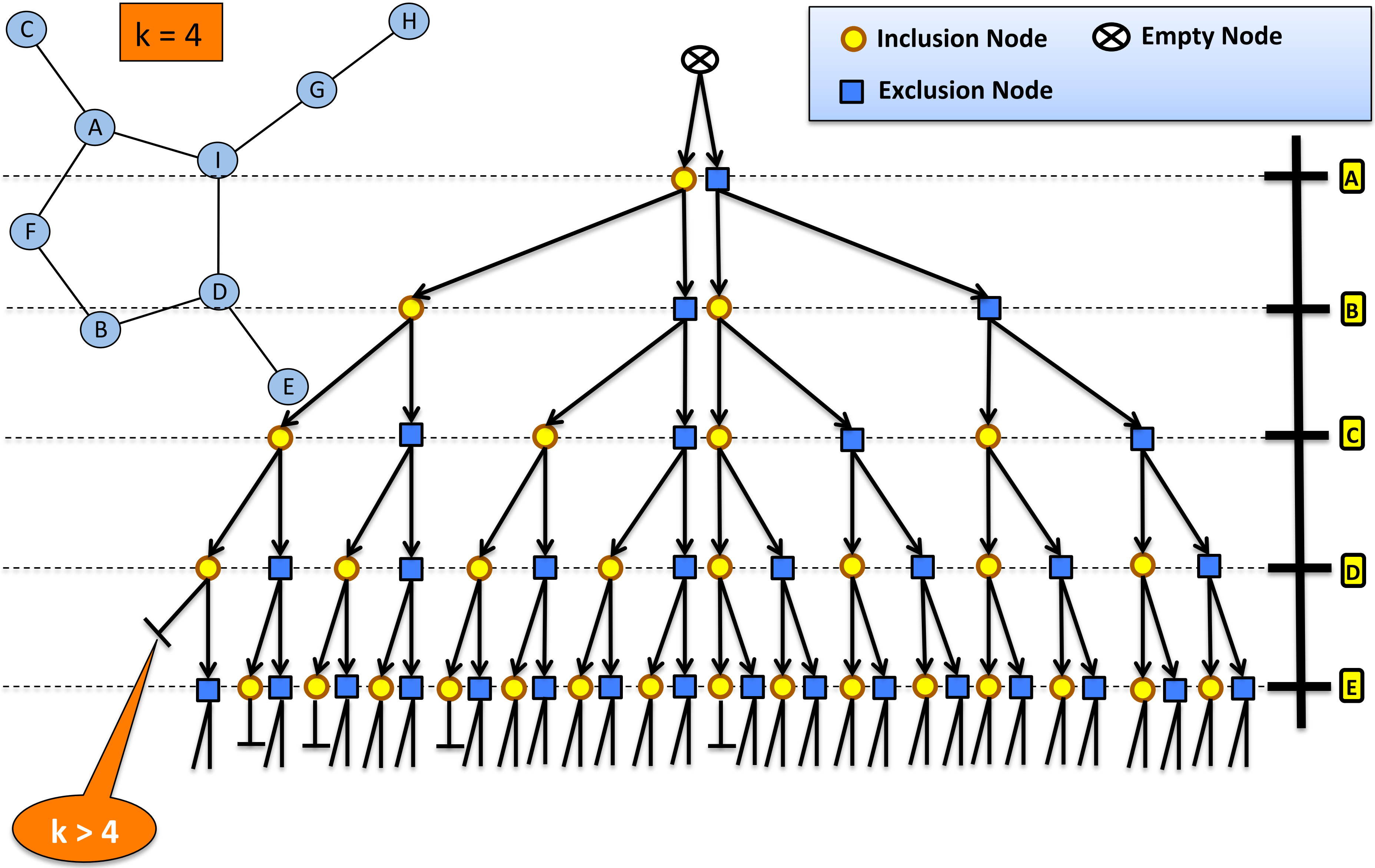
⊗ Empty Node

■ Exclusion Node

k-Bound Pruning



k-Bound Pruning



OUTLINE

✓ Original Problem
✓ Simplified Problem
✓ Detailed Problem Space
✓ Pruning of Search Tree
✓ Node Ordering Heuristic

- Selection of Search Algorithm
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

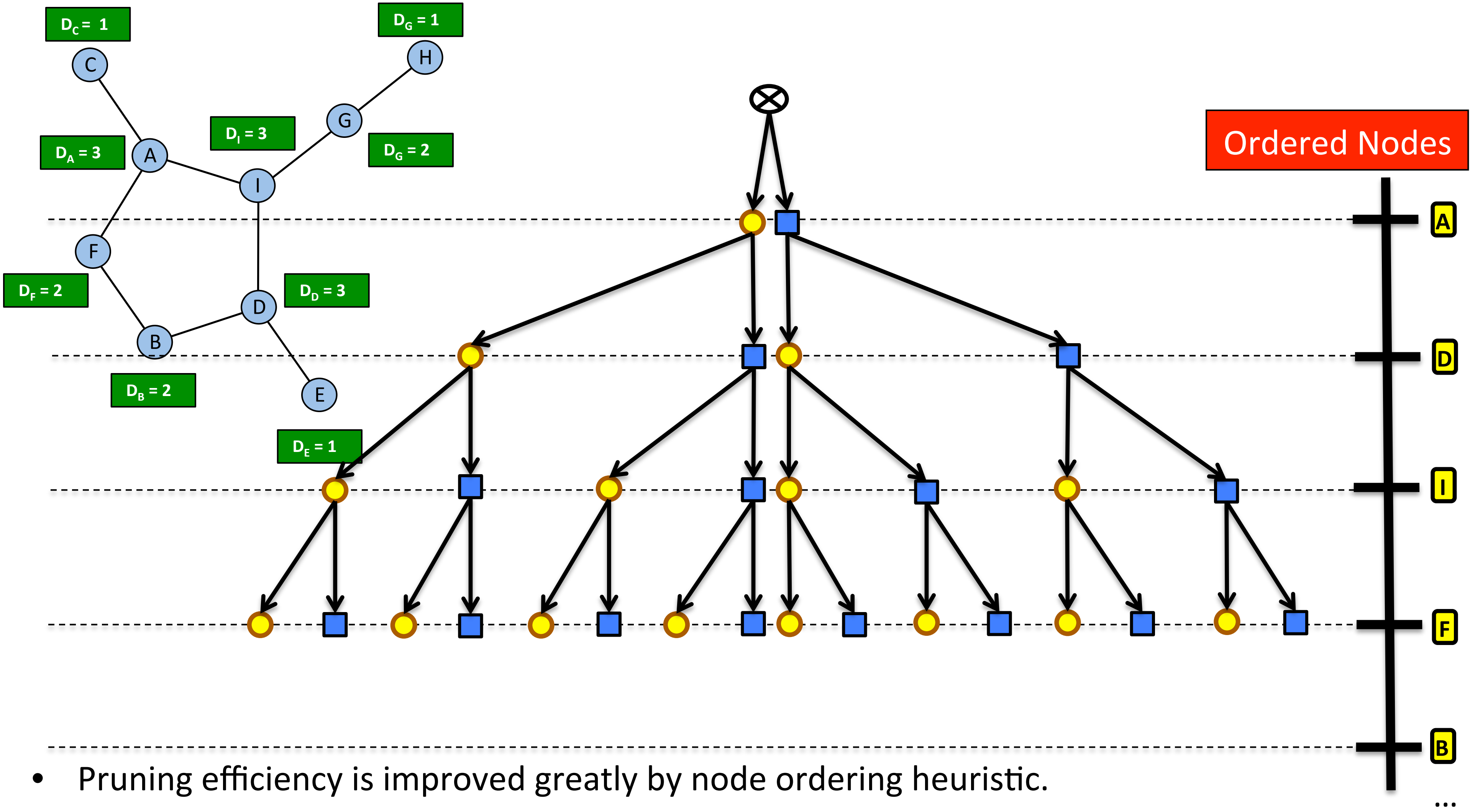
Node Ordering Heuristic

- In sociology literature, degree and other centrality-based heuristics are commonly used to estimate the influence of nodes in social networks

- Degree is frequently used for selecting seeds in influence maximization. Experimental results show that selecting vertices with maximum degrees as seed results in larger influence spread than other heuristics.

Node Ordering Heuristic

- Ordering the nodes of the graph from maximum edge degree to minimum edge degree in inclusion/exclusion search tree.



- Pruning efficiency is improved greatly by node ordering heuristic.

OUTLINE

✓ Original Problem

✓ Simplified Problem

✓ Detailed Problem Space

✓ Pruning of Search Tree

✓ Node Ordering Heuristic

✓ Selection of Search Algorithm

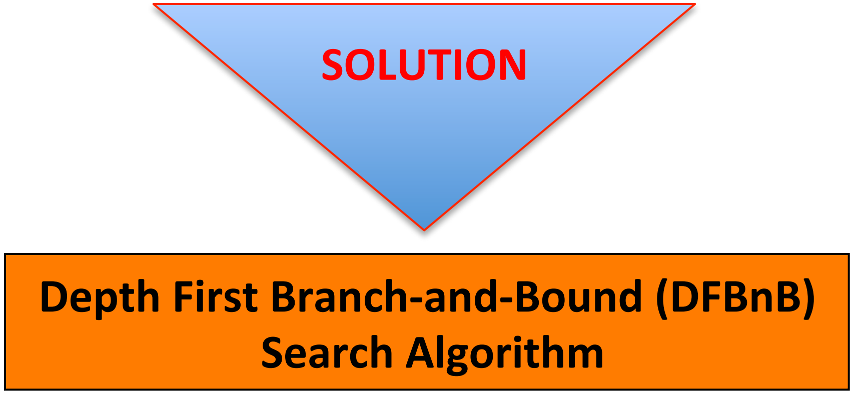
- Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

Selection of Search Algorithm

- The search tree is finite
- The solution depth is known a priori.
- In a simplified influence maximization(vertex cover) problem with N nodes, this search tree is only N levels deep and all solutions are at depth N.
- Inclusion/Exclusion Search Tree guarantees that there is a unique path to each vertex cover so no duplicate nodes , eliminating the need for storing previously generated nodes and allowing a linear space depth-first search.

Selection of Search Algorithm

- The search tree is finite
- The solution depth is known a priori.
- In a simplified influence maximization(vertex cover) problem with N nodes, this search tree is only N levels deep and all solutions are at depth N.
- Inclusion/Exclusion Search Tree guarantees that there is a unique path to each vertex cover so no duplicate nodes , eliminating the need for storing previously generated nodes and allowing a linear space depth-first search.

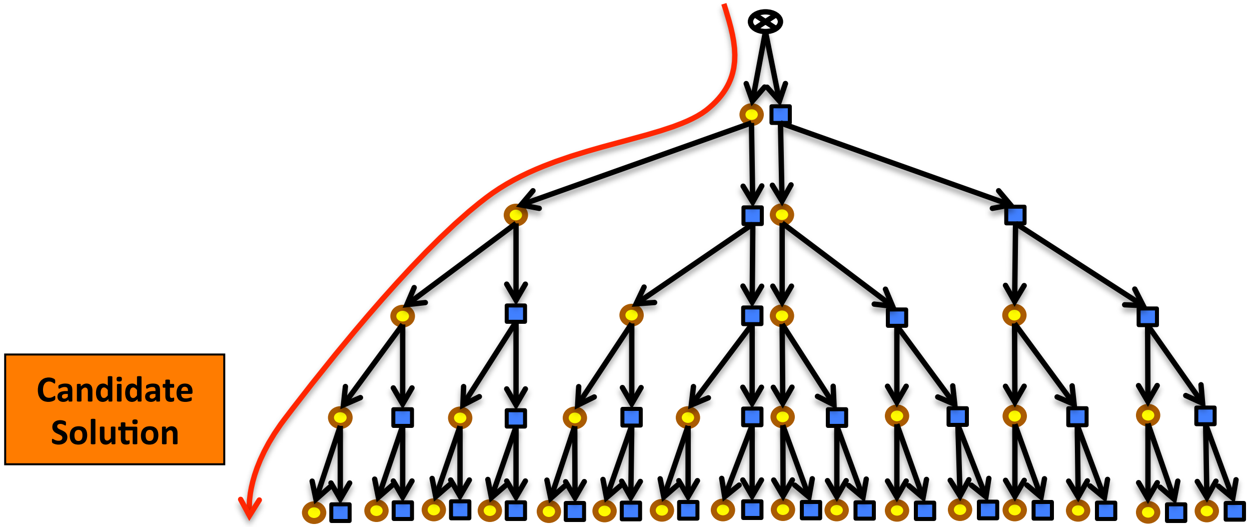


SOLUTION

Depth First Branch-and-Bound (DFBnB) Search Algorithm

Depth First Branch-and-Bound (DFBnB)

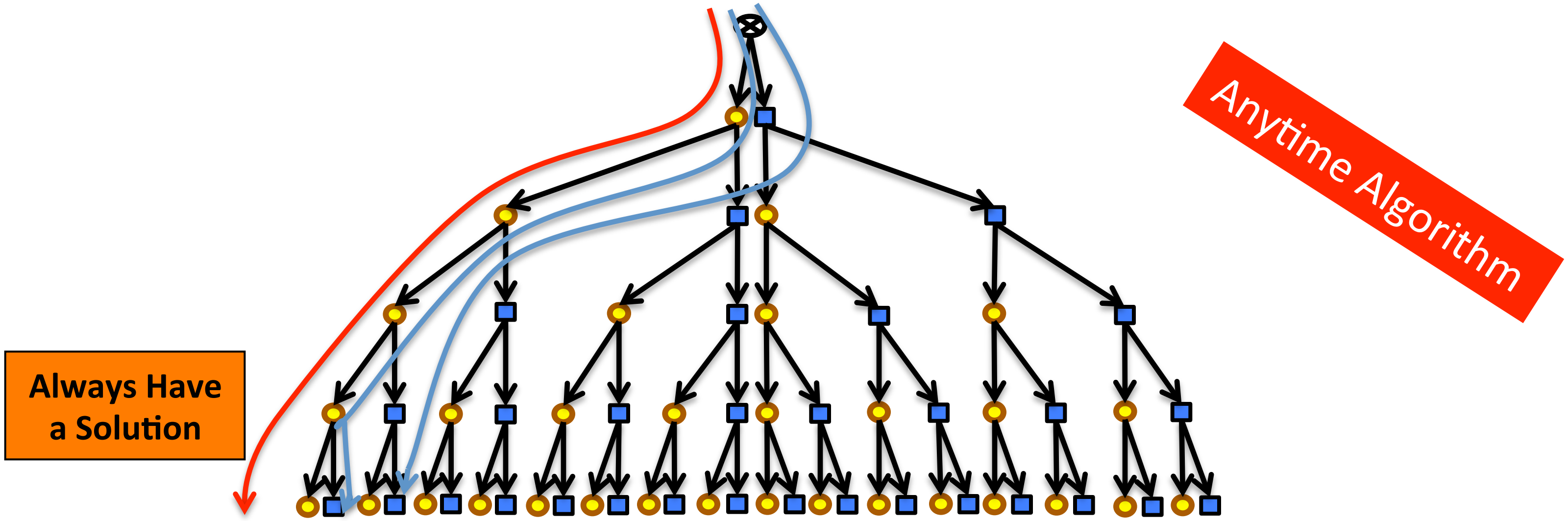
- DFBnB performs a single depth-first search of the inclusion/ exclusion problem-space tree from left to right.



- As soon as the left-most branch terminates, we have a candidate solution which is a seed set can activate all the nodes. However this solution is not the optimal one. Let the number of seed nodes in this seed set solution be “a”, which is the number of seed nodes in the best seed set found so far.

Depth First Branch-and-Bound (DFBnB)

- As the search continues, if the number of seed nodes in seed set equals or exceeds "a", the branch below that node is pruned, because it can only lead to seed sets which has a number of seed nodes is equal or greater than the best seed set so far.



- If a complete solution is found which has a number of seed nodes is less than "a", "a" is updated to this lower value and the best solution so far is replaced this new solution. When the search terminates by exhausting the entire tree, the best solution found is returned as the optimal solution.

OUTLINE

- ✓ Original Problem
- ✓ Simplified Problem
- ✓ Detailed Problem Space
- ✓ Pruning of Search Tree
- ✓ Node Ordering Heuristic
- ✓ Selection of Search Algorithm
- ✓ Cost Function
- Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

Cost Function

f(n) **:** A* Cost Function

g(n) **:** The number of nodes that have been included so far at node n of the search tree. This is the cost of the partial vertex cover computed so far and can never decrease as we go deep the search tree.

h(n) **:** An estimation of the number of additional vertices that must be included to cover the edges of the remaining graph.

$$\boxed{f(n) = g(n) + h(n)}$$

- In our DFBnB algorithm at each interior node of n of the search we apply the cost function f(n). If this cost equals or exceeds the size of the smallest vertex cover found so far, we can prune this node. We are still guaranteed an optimal solution as long as h(n) never overestimates the number of additional vertices we have to add to the current vertex cover.

OUTLINE

- ✓ Original Problem
- ✓ Simplified Problem
- ✓ Detailed Problem Space
- ✓ Pruning of Search Tree
- ✓ Node Ordering Heuristic
- ✓ Selection of Search Algorithm
- ✓ Cost Function
- ✓ Design of Heuristic Evaluation Function
- Empirical Results
- Conclusion
- References

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Why removing the constraints???

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Why removing the constraints??? Because constraints increase the cost

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Why removing the constraints??? Because constraints increase the cost

- Original vertex cover problem:
  - **Constraints**:
    - minimum cardinality of vertex cover nodes
    - covering all the edges

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Why removing the constraints??? Because constraints increase the cost

- Original vertex cover problem:
  - **Constraints**:
    - o minimum cardinality of vertex cover nodes
    - o covering all the edges

We can relax this constraint

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Why removing the constraints??? Because constraints increase the cost

- Original vertex cover problem:
  - **Constraints**:
    - minimum cardinality of vertex cover nodes
    - covering all the edges

We can relax this constraint

➢ Can we solve the exact cost of vertex cover problem in a graph which has less edges than original graph optimally???

Design of Admissible Heuristic Evaluation Function

How to design admissible and consistent heuristic function?

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem.

Why removing the constraints??? Because constraints increase the cost

- Original vertex cover problem:
  - **Constraints**:
    - minimum cardinality of vertex cover nodes
    - covering all the edges

We can relax this constraint

➢ Can we solve the exact cost of vertex cover problem in a graph which has less edges than original graph optimally???

➢ How can we produce this kind of graphs which has less edges than original graphs???

Spanning Tree Heuristic Evaluation Function

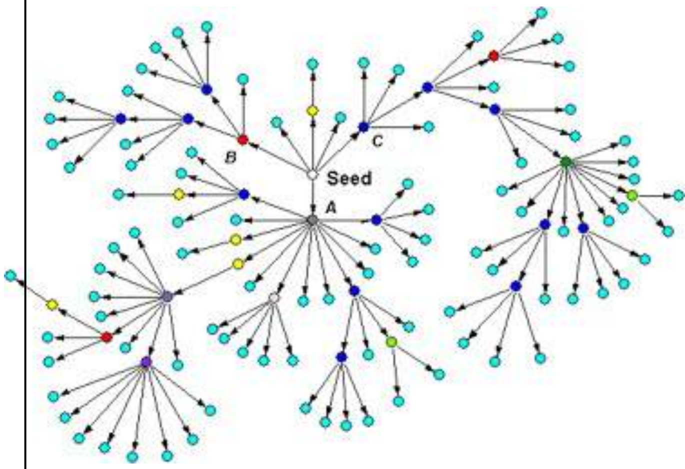
What if underlying graph is a tree???

Spanning Tree Heuristic Evaluation Function

What if underlying graph is a tree???

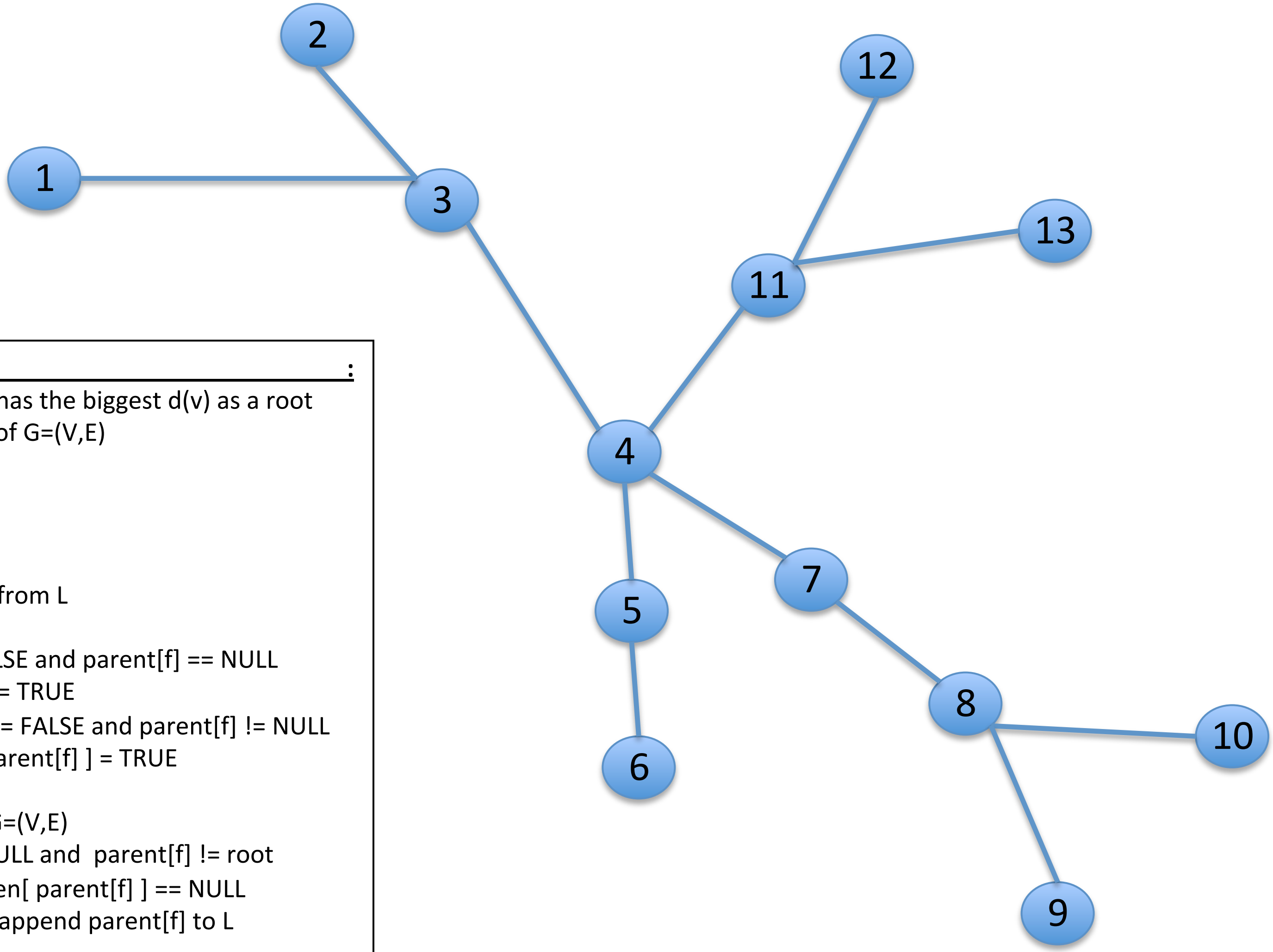
```
Algorithm(input tree G=(V,E))                                :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v∈V | mark[v]=TRUE }
```

Then we can solve the vertex cover problem optimally in polynomial time



Time Complexity = O(n)

Spanning Tree Heuristic Evaluation Function

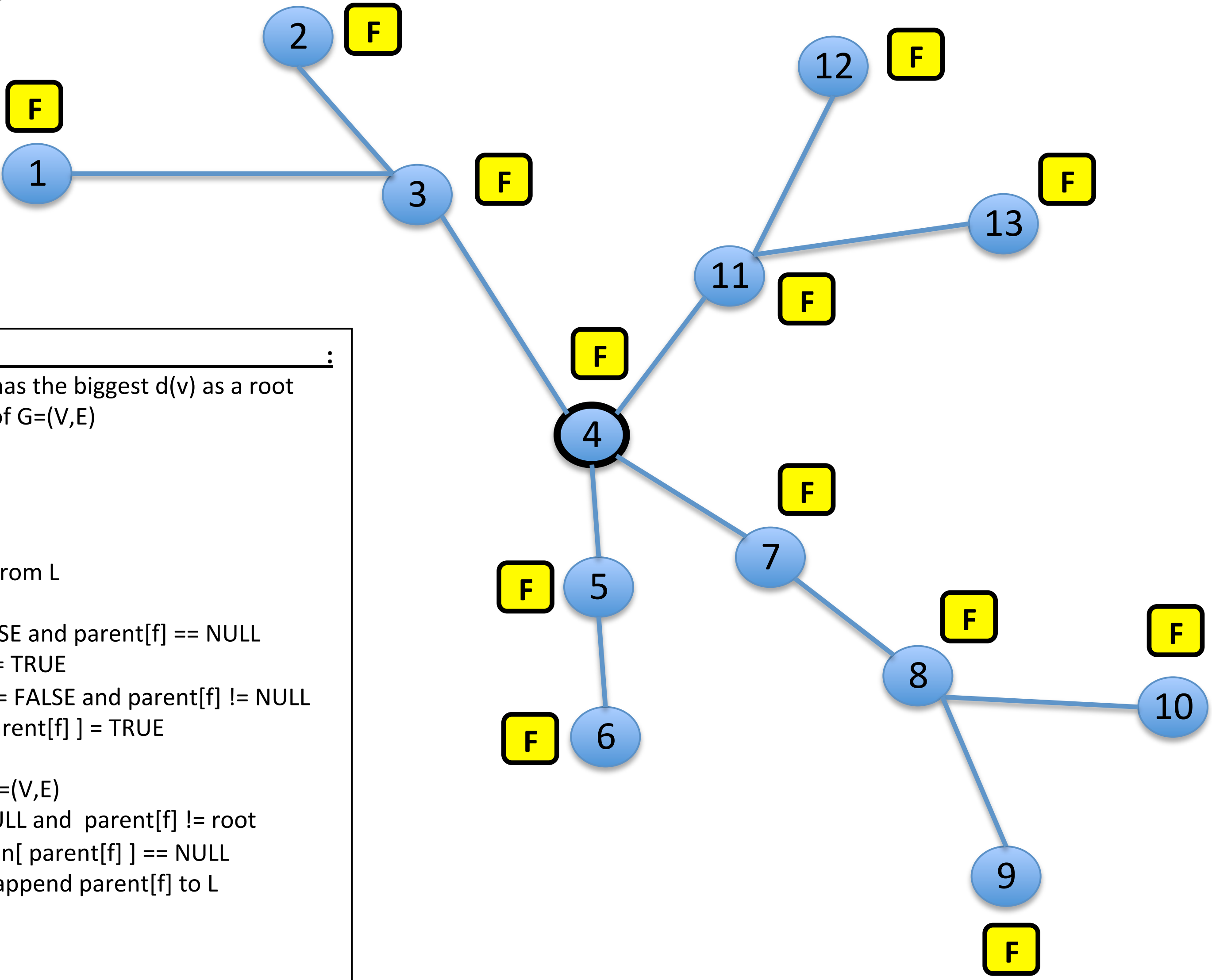


```
Algorithm(input tree G=(V,E))                                   :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {1,2,6,9,10,12,13}

S = {}



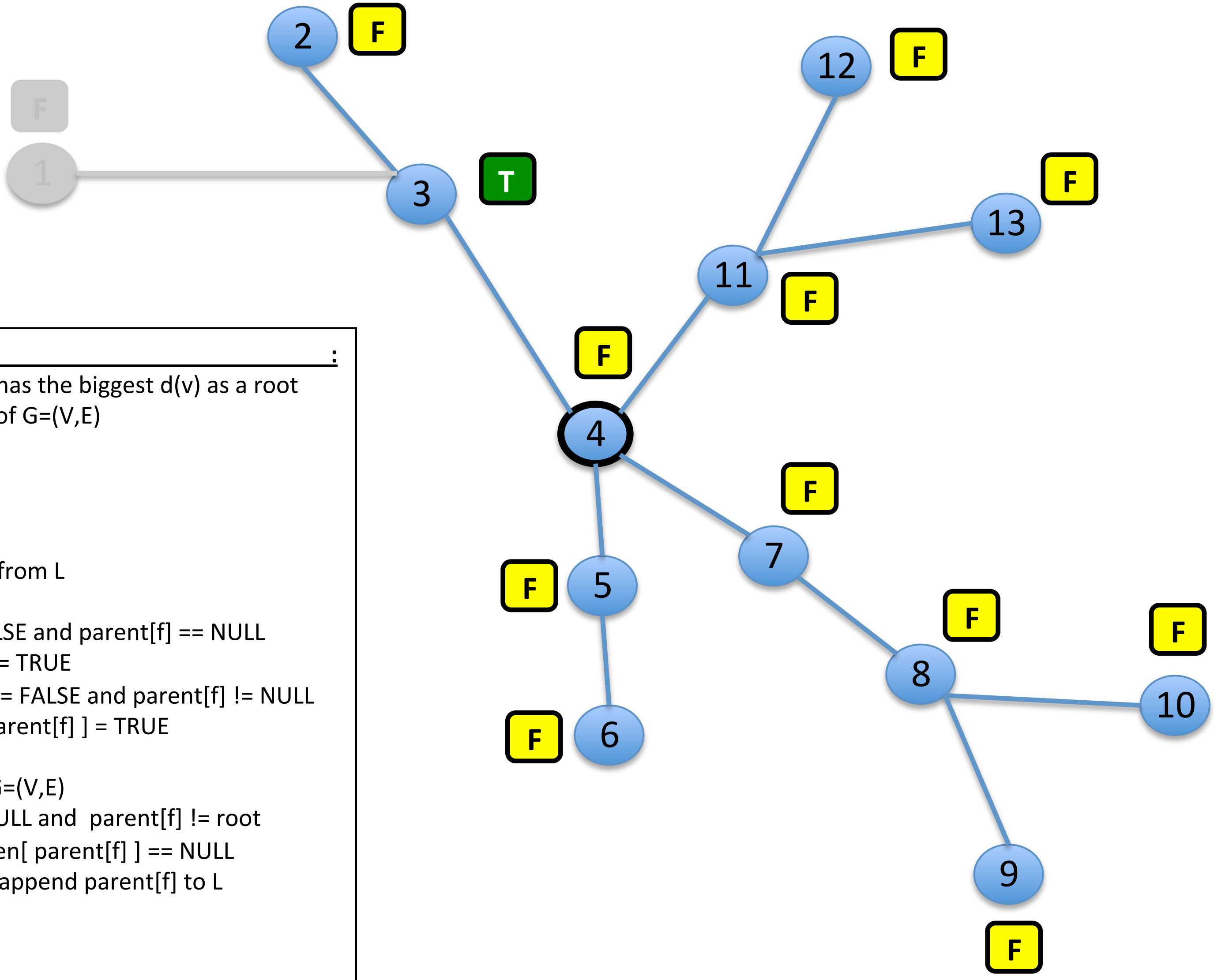
```
Algorithm(input tree G=(V,E))                                          :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {2,6,9,10,12,13}

f = 1

S = {3}



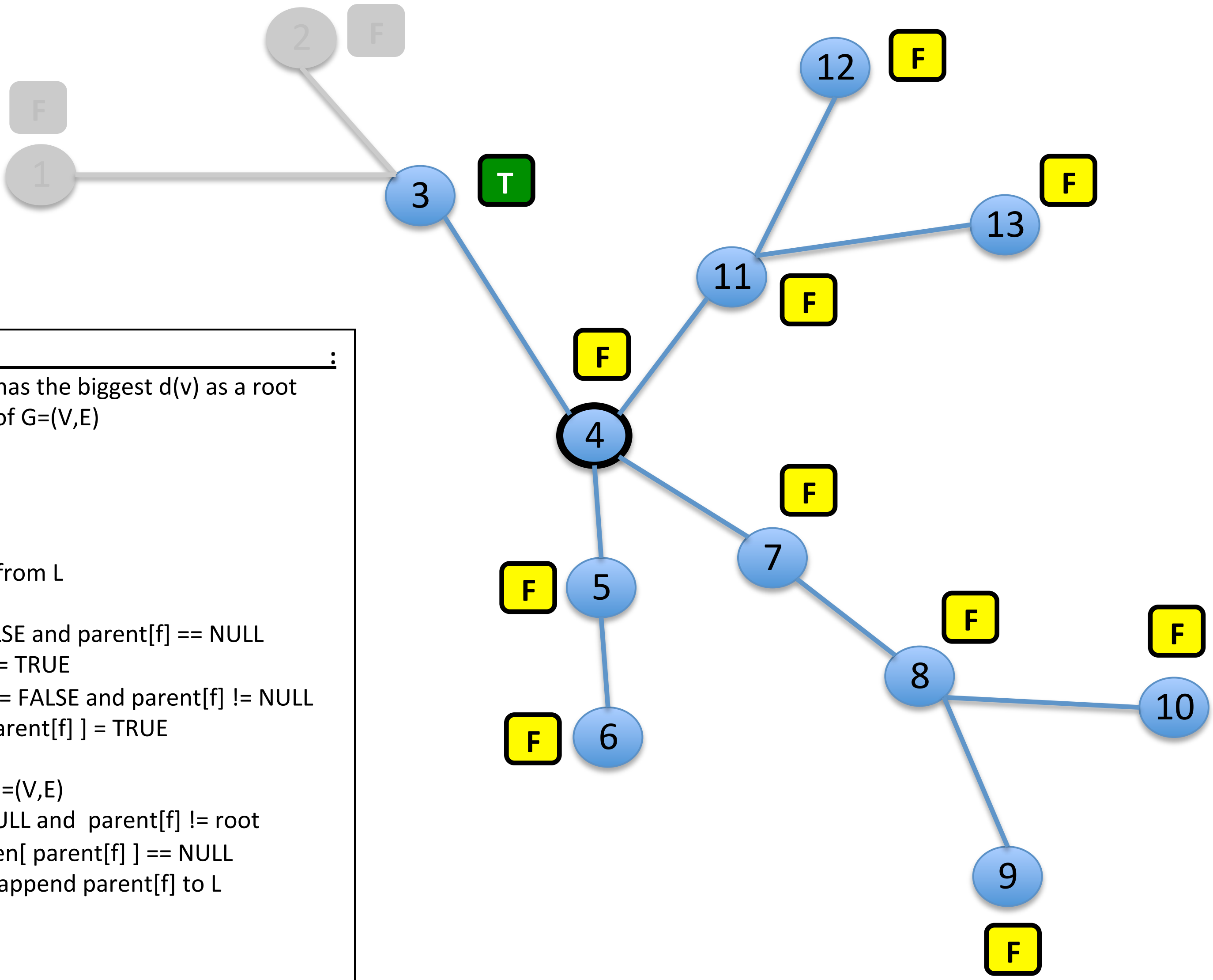
```
Algorithm(input tree G=(V,E))                                    :
    Choose one of the nodes which has the biggest d(v) as a root
    Let L , denotes the list of leaves of G=(V,E)
    for each node v∈ V
        mark[v] = FALSE
    endfor
    While L is not empty
        f = remove the first leaf from L
        If f is in G=(V,E)
            If mark[f] == FALSE and parent[f] == NULL
                mark[f] = TRUE
            else if  mark[f] == FALSE and parent[f] != NULL
                mark[ parent[f] ] = TRUE
            endif
            remove f from G=(V,E)
            If parent[f] != NULL and  parent[f] != root
                If children[ parent[f] ] == NULL
                    append parent[f] to L
                endif
            endif
        endif
    endwhile
    Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {6,9,10,12,13,3}

f = 2

S = {3}



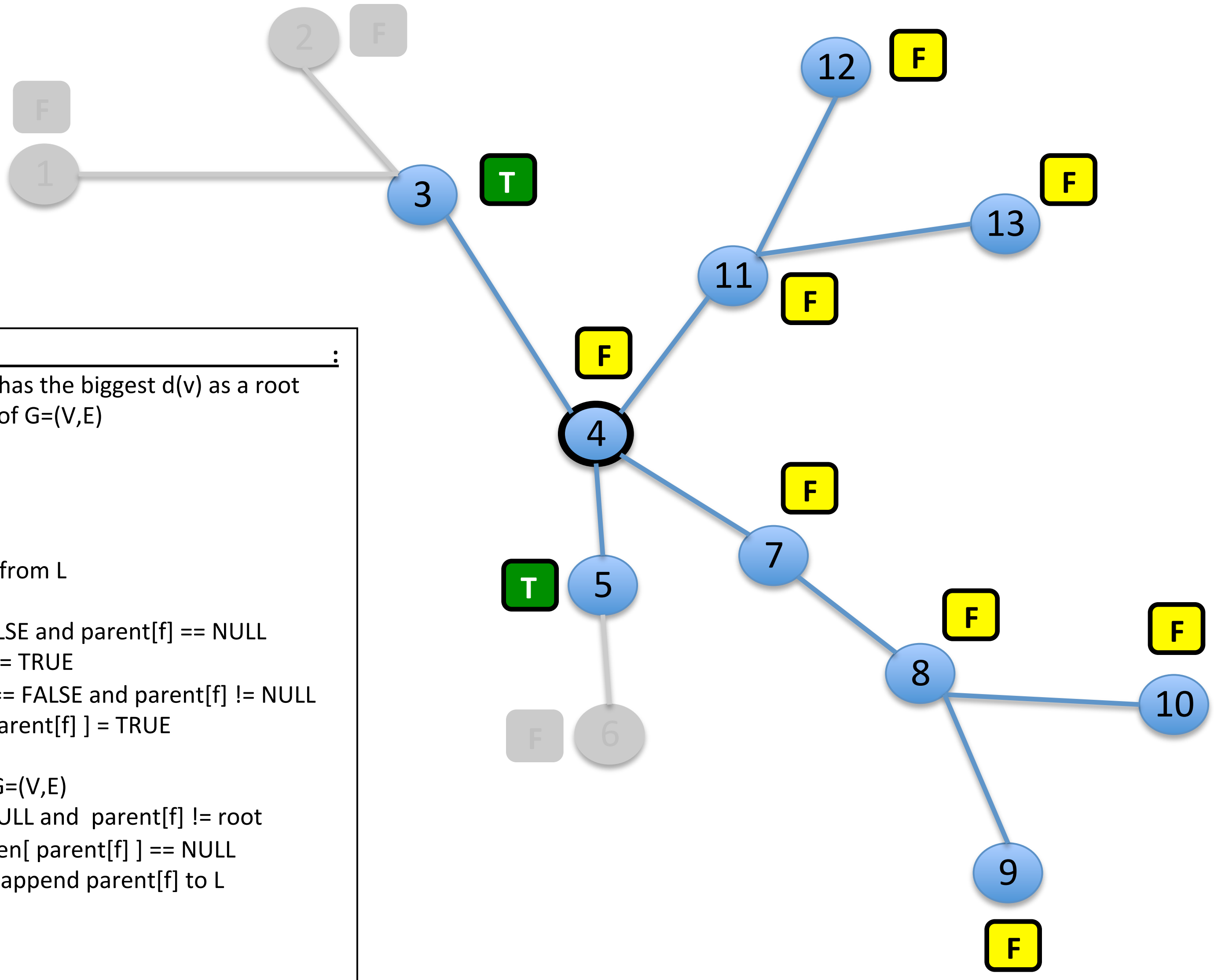
```
Algorithm(input tree G=(V,E))                                          :
    Choose one of the nodes which has the biggest d(v) as a root
    Let L , denotes the list of leaves of G=(V,E)
    for each node v∈ V
        mark[v] = FALSE
    endfor
    While L is not empty
        f = remove the first leaf from L
        If f is in G=(V,E)
            If mark[f] == FALSE and parent[f] == NULL
                mark[f] = TRUE
            else if  mark[f] == FALSE and parent[f] != NULL
                mark[ parent[f] ] = TRUE
            endif
            remove f from G=(V,E)
            If parent[f] != NULL and  parent[f] != root
                If children[ parent[f] ] == NULL
                    append parent[f] to L
                endif
            endif
        endif
    endwhile
    Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {9,10,12,13,3,5}

f = 6

S = {3,5}



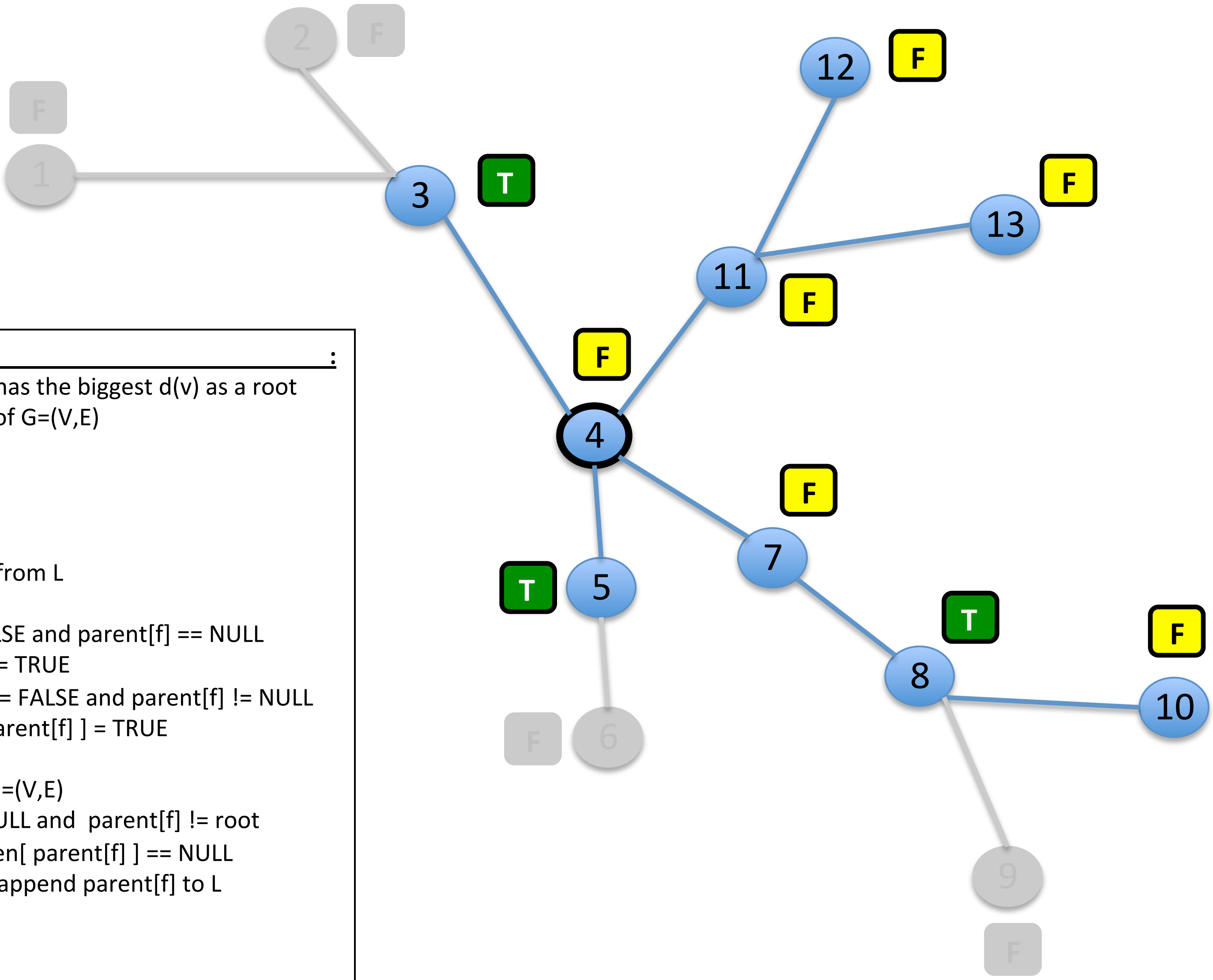
```
Algorithm(input tree G=(V,E))                                    :
    Choose one of the nodes which has the biggest d(v) as a root
    Let L , denotes the list of leaves of G=(V,E)
    for each node v∈ V
        mark[v] = FALSE
    endfor
    While L is not empty
        f = remove the first leaf from L
        If f is in G=(V,E)
            If mark[f] == FALSE and parent[f] == NULL
                mark[f] = TRUE
            else if  mark[f] == FALSE and parent[f] != NULL
                mark[ parent[f] ] = TRUE
            endif
            remove f from G=(V,E)
            If parent[f] != NULL and  parent[f] != root
                If children[ parent[f] ] == NULL
                    append parent[f] to L
                endif
            endif
        endif
    endwhile
    Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {10,12,13,3,5}

f = 9

S = {3,5,8}



Algorithm(input tree G=(V,E)) :

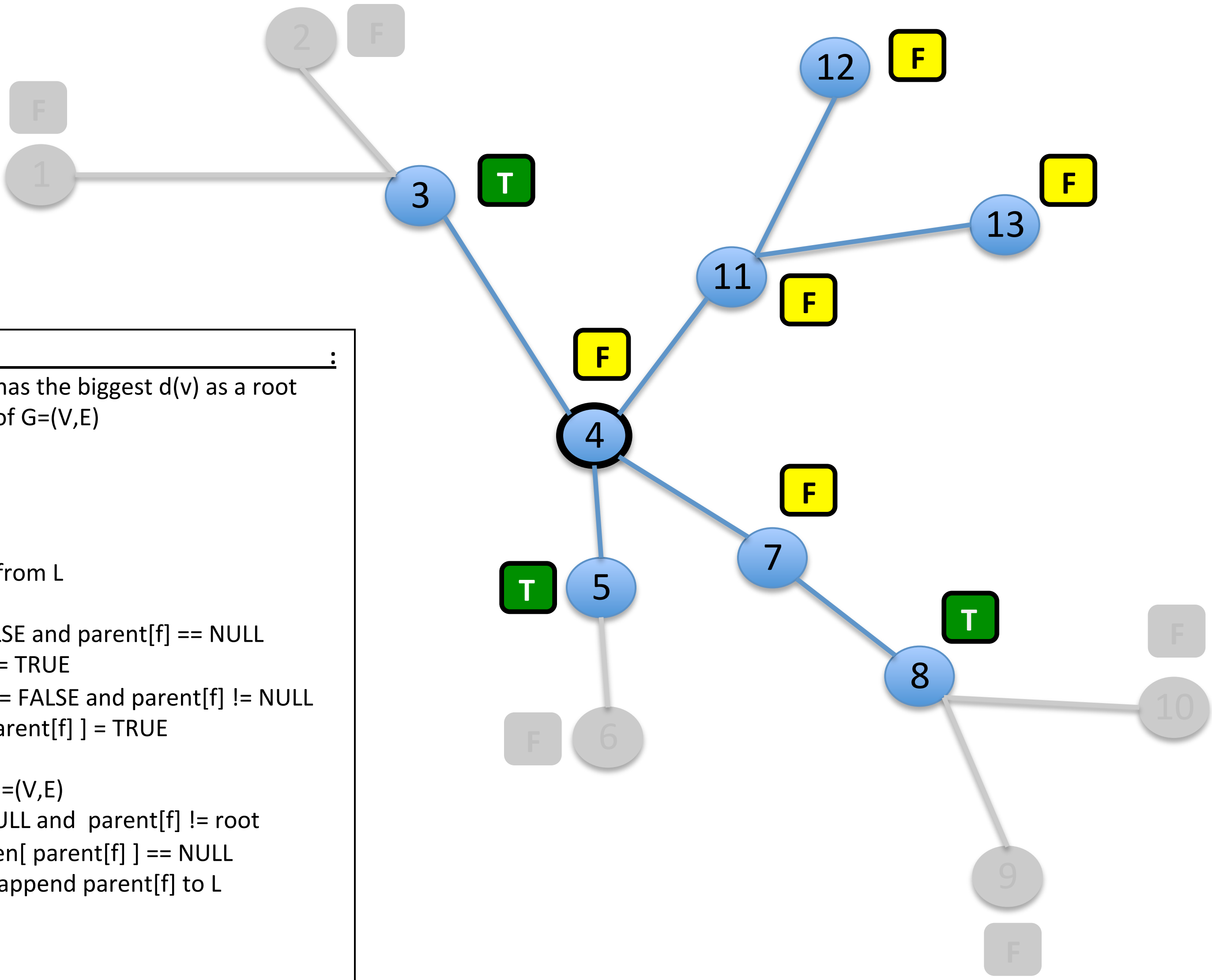
```
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
          mark[v] = FALSE
endfor
While L is not empty
          f = remove the first leaf from L
          If f is in G=(V,E)
                    If mark[f] == FALSE and parent[f] == NULL
                              mark[f] = TRUE
                    else if  mark[f] == FALSE and parent[f] != NULL
                              mark[ parent[f] ] = TRUE
                    endif
                    remove f from G=(V,E)
                    If parent[f] != NULL and  parent[f] != root
                              If children[ parent[f] ] == NULL
                                        append parent[f] to L
                              endif
                    endif
          endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {12,13,3,5,8}

f = 10

S = {3,5,8}



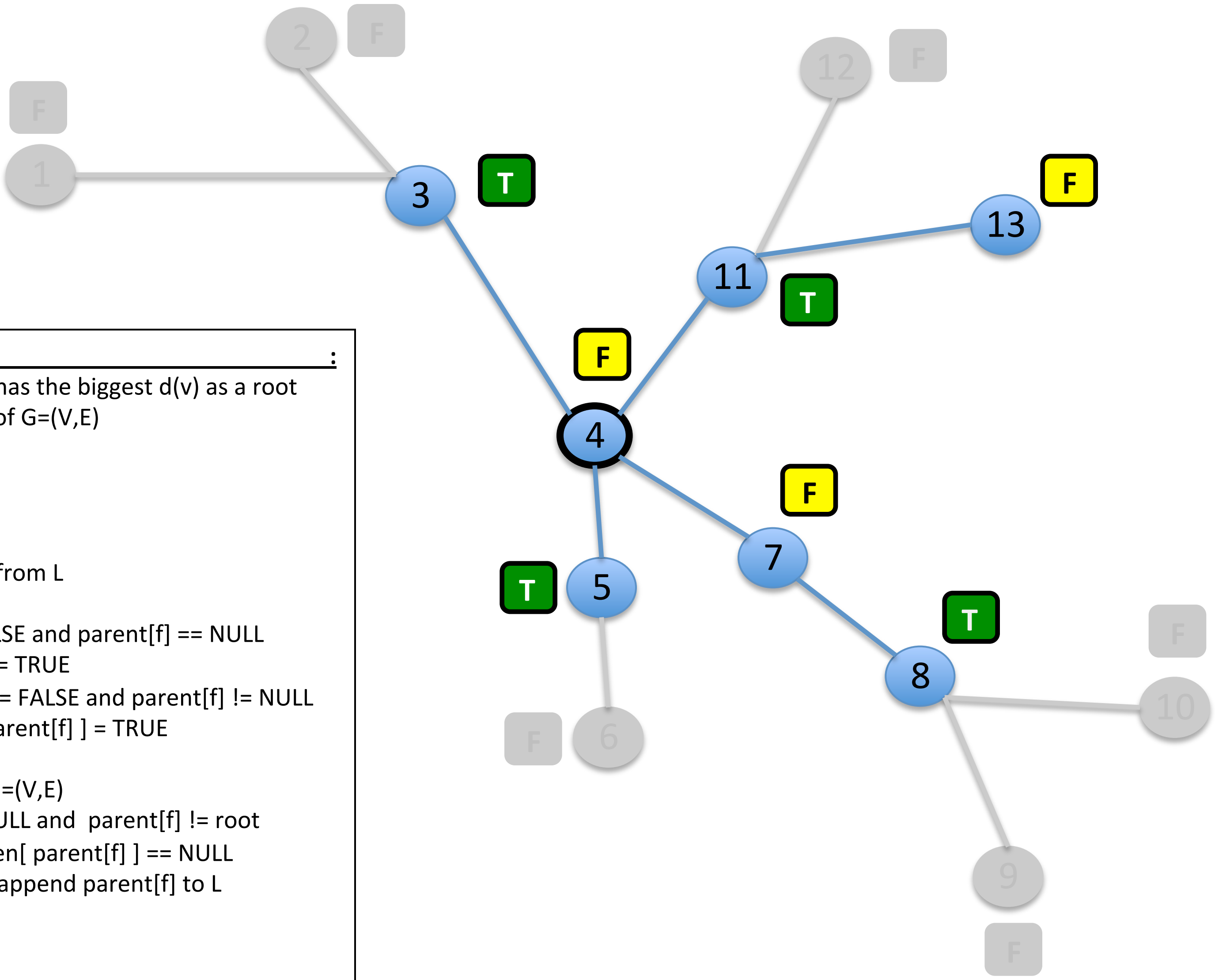
```
Algorithm(input tree G=(V,E))                                        :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {13,3,5,8}

f = 12

S = {3,5,8,11}



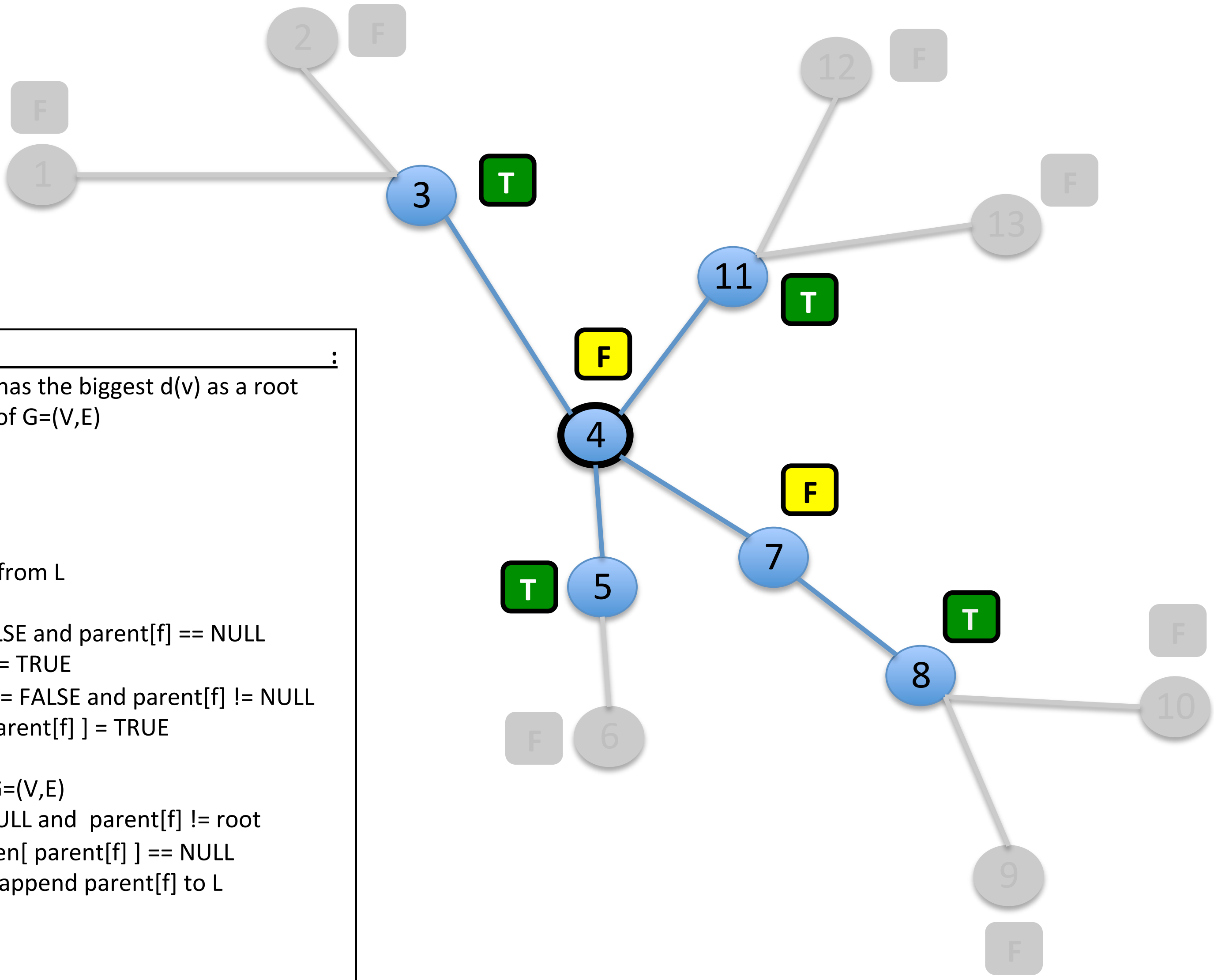
```
Algorithm(input tree G=(V,E))                                    :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {3,5,8,11}

f = 13

S = {3,5,8,11}



```
Algorithm(input tree G=(V,E))                                    :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

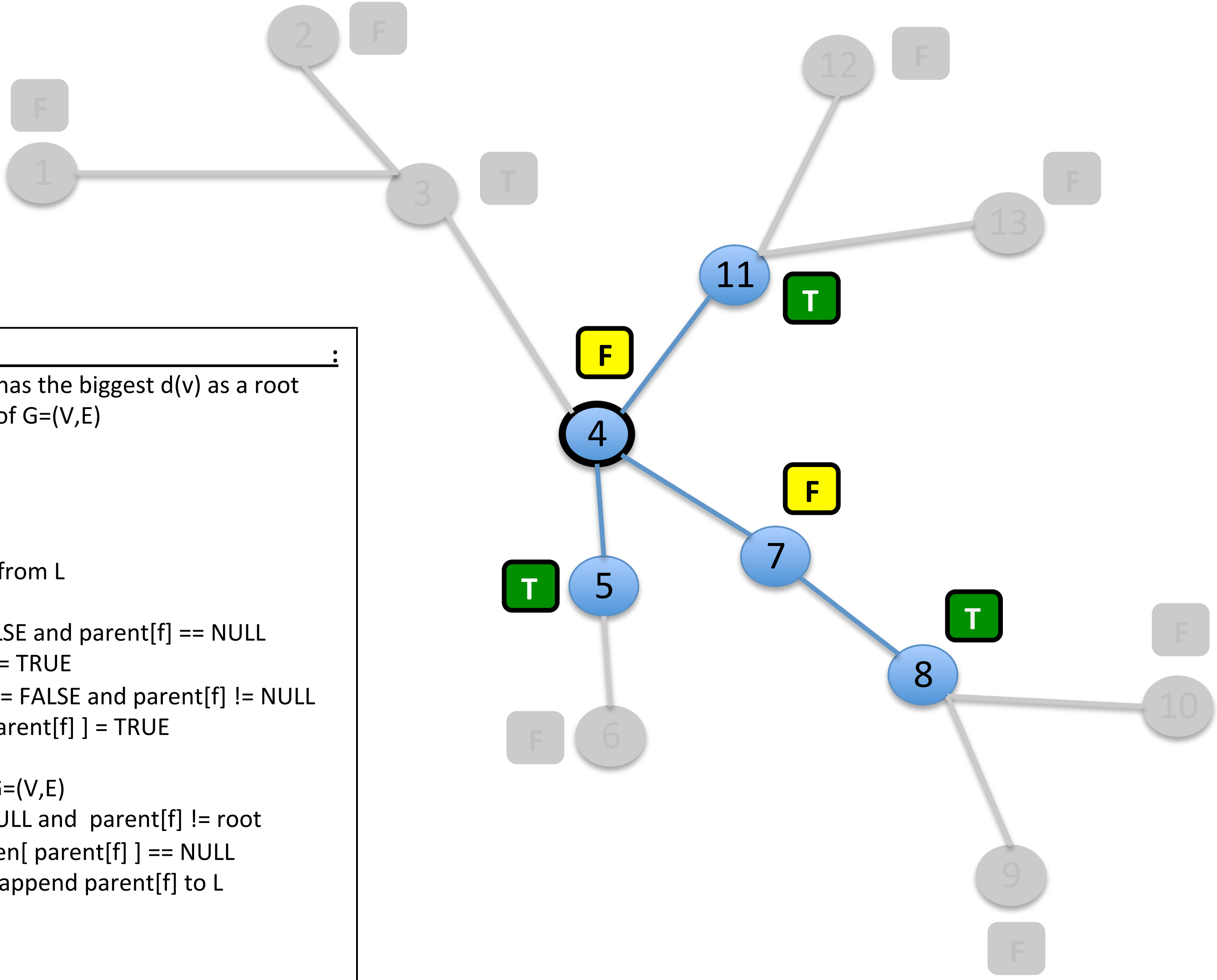
L = {5,8,11}

f = 3

S = {3,5,8,11}

Algorithm(input tree G=(V,E)) :

```
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```



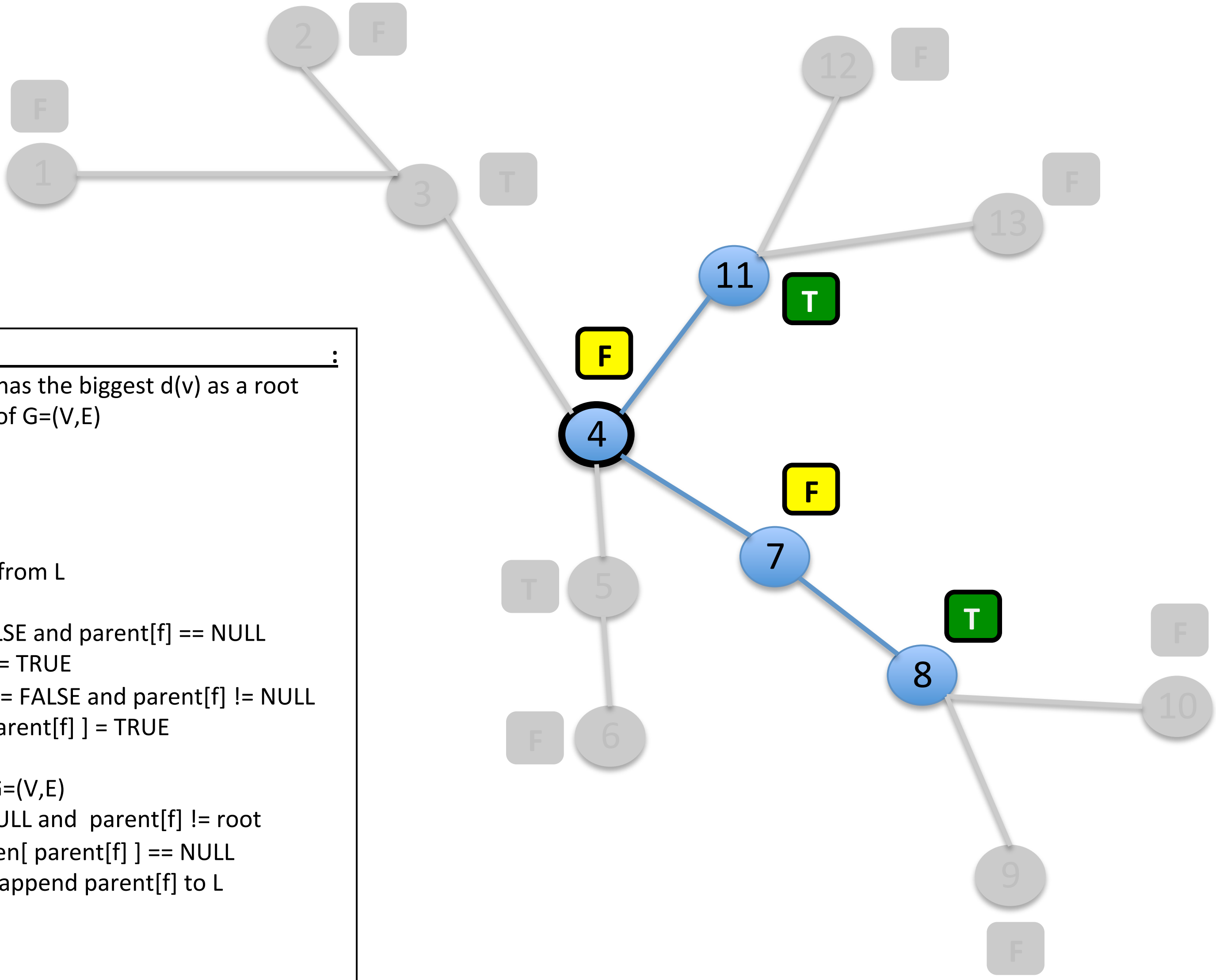
Spanning Tree Heuristic Evaluation Function

L = {8,11}

f = 5

S = {3,5,8,11}

```
Algorithm(input tree G=(V,E))                                    :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

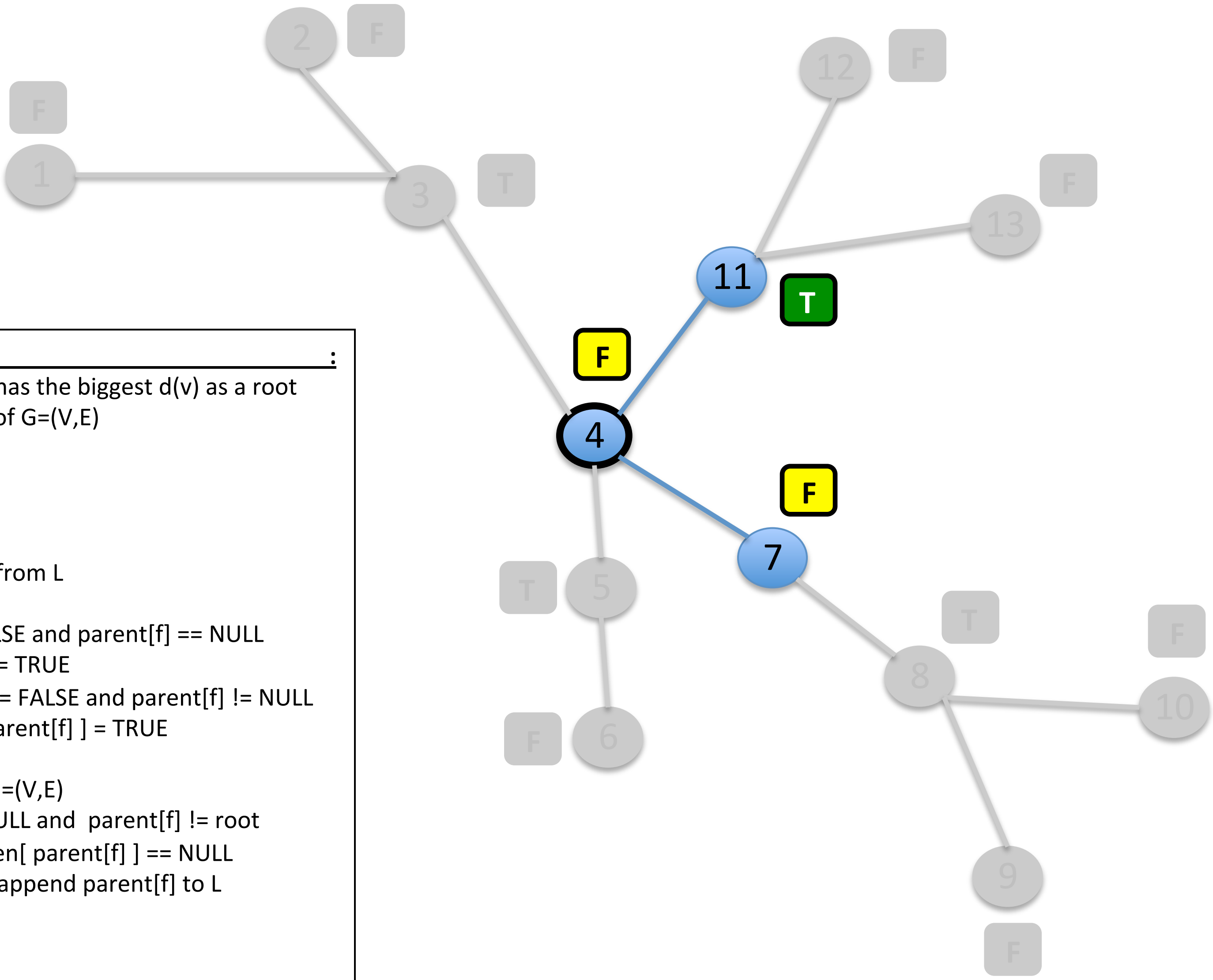


Spanning Tree Heuristic Evaluation Function

L = {11,7}

f = 8

S = {3,5,8,11}



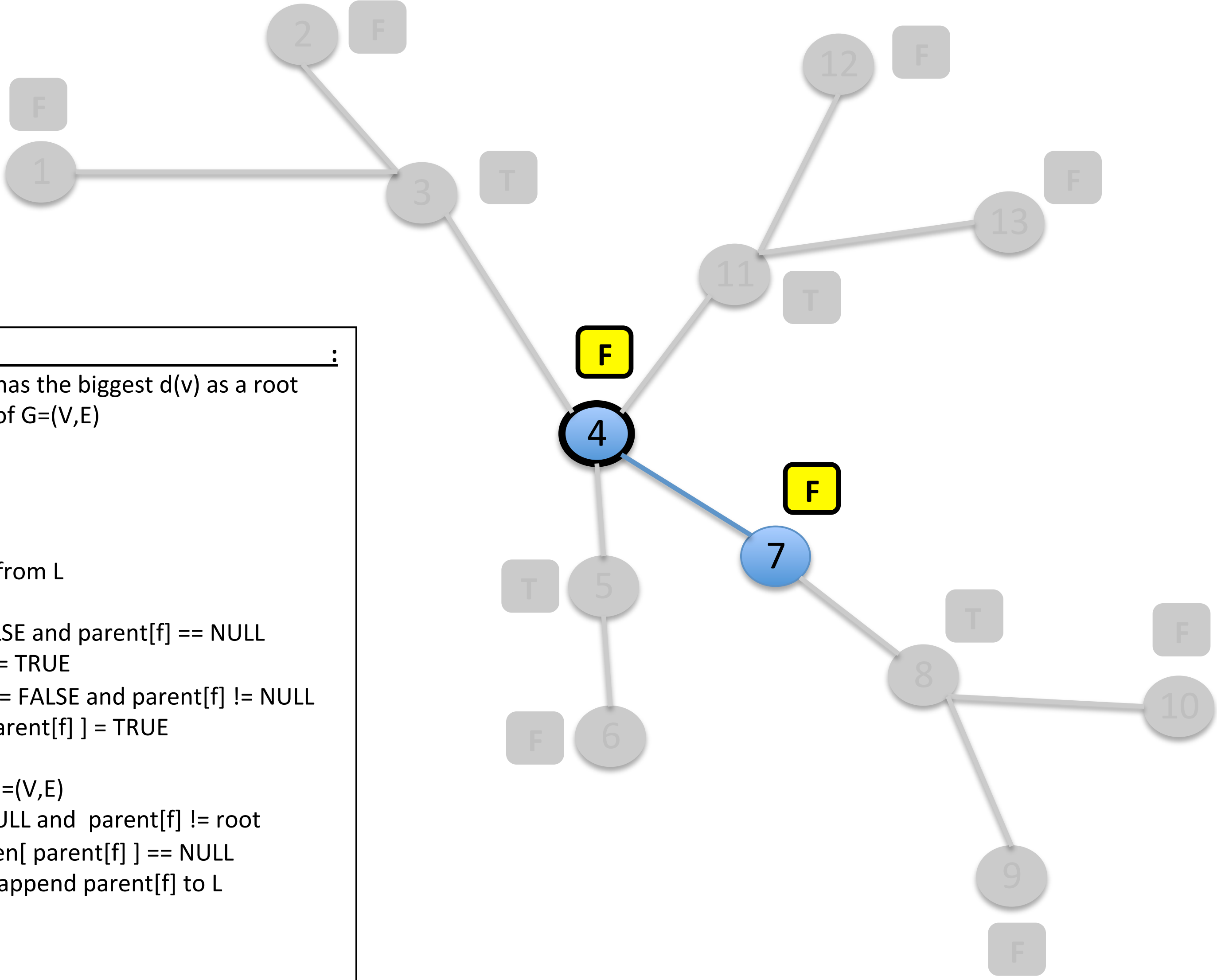
```
Algorithm(input tree G=(V,E))                                   :
    Choose one of the nodes which has the biggest d(v) as a root
    Let L , denotes the list of leaves of G=(V,E)
    for each node v∈ V
        mark[v] = FALSE
    endfor
    While L is not empty
        f = remove the first leaf from L
        If f is in G=(V,E)
            If mark[f] == FALSE and parent[f] == NULL
                mark[f] = TRUE
            else if  mark[f] == FALSE and parent[f] != NULL
                mark[ parent[f] ] = TRUE
            endif
            remove f from G=(V,E)
            If parent[f] != NULL and  parent[f] != root
                If children[ parent[f] ] == NULL
                    append parent[f] to L
                endif
            endif
        endif
    endwhile
    Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {7}

f = 11

S = {3,5,8,11}



Algorithm(input tree G=(V,E)) :

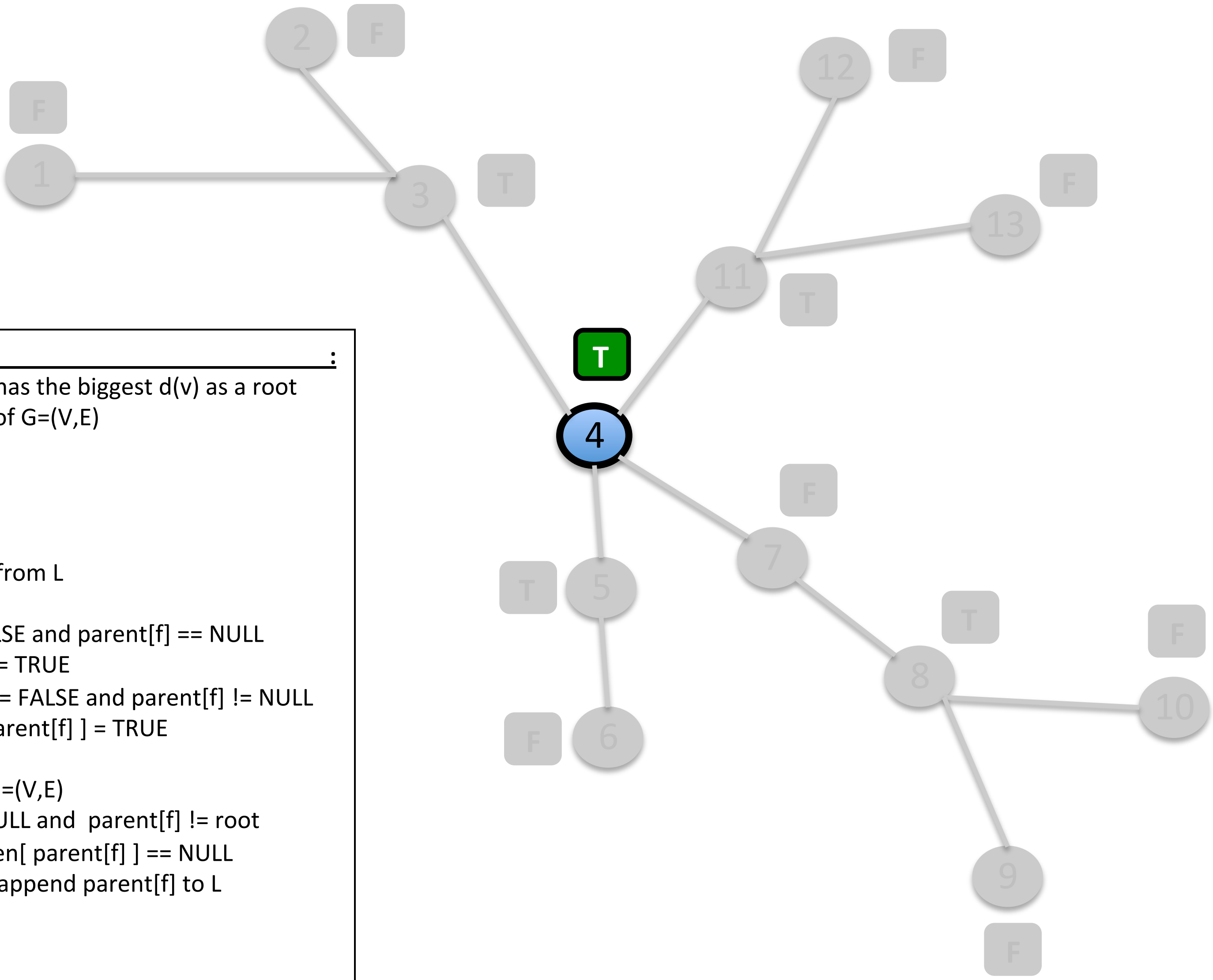
```
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

L = {}

f = 7

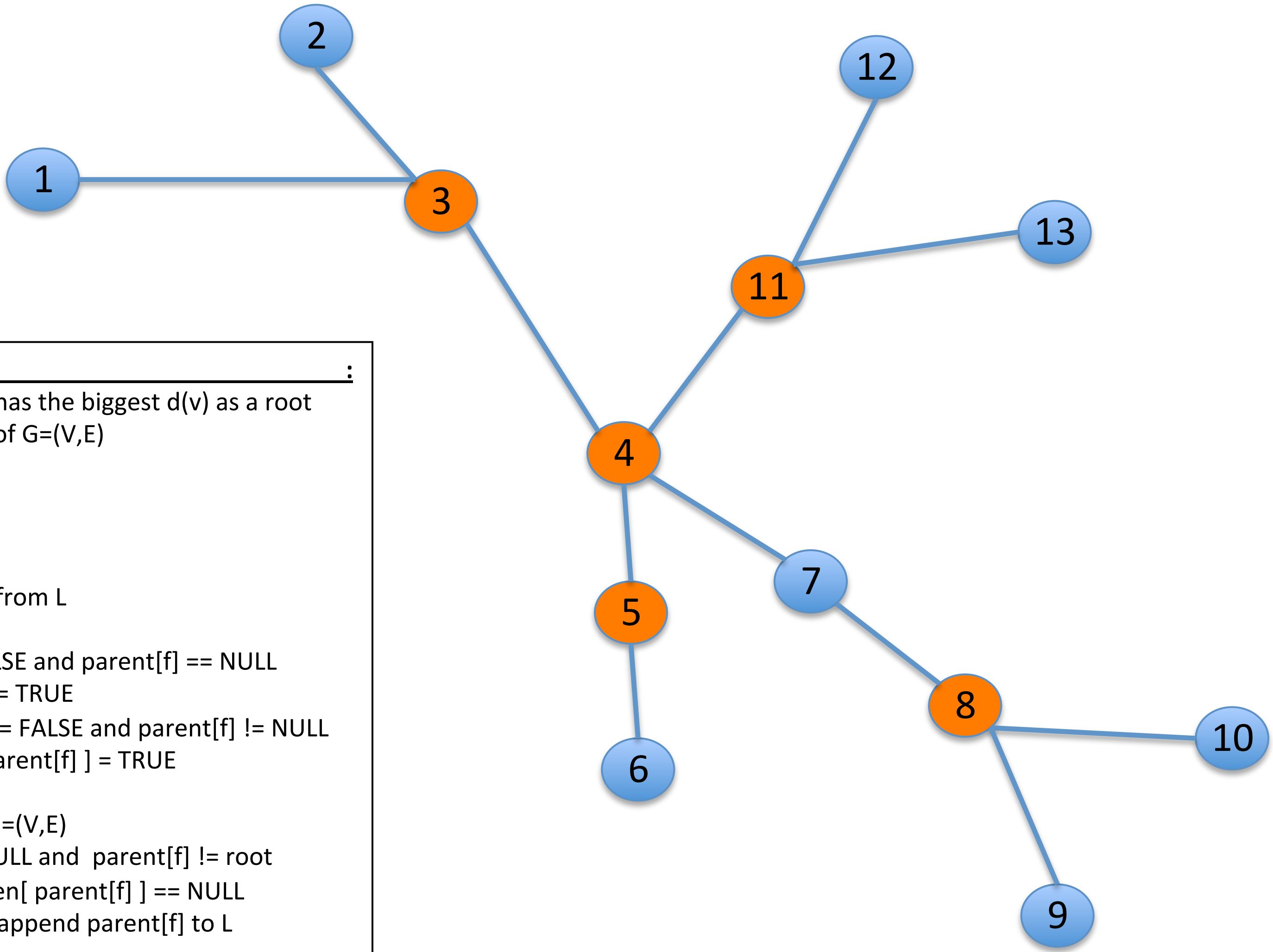
S = {3,5,8,11,4}



```
Algorithm(input tree G=(V,E))                                        :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

S = {3,5,8,11,4}



```
Algorithm(input tree G=(V,E))                                        :
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v    V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

Outlines of the prove of correctness and optimality of algorithm:

- Never need to include original leaf of G in vertex cover unless leaf is root. This is because (unless leaf is root) degree of leaf < degree of its parent

- Must cover edge from leaf to parent of leaf. Since G is a tree, only one edge comes from a parent. Thus, we must include parent of each leaf in cover if we choose to not include leaf in cover

- If parent x of a node is in cover, parent x also covers edges to all children of x and the edge to parent[x].

Algorithm(input tree G=(V,E)) :

```
Choose one of the nodes which has the biggest d(v) as a root
Let L , denotes the list of leaves of G=(V,E)
for each node v∈ V
    mark[v] = FALSE
endfor
While L is not empty
    f = remove the first leaf from L
    If f is in G=(V,E)
        If mark[f] == FALSE and parent[f] == NULL
            mark[f] = TRUE
        else if  mark[f] == FALSE and parent[f] != NULL
            mark[ parent[f] ] = TRUE
        endif
        remove f from G=(V,E)
        If parent[f] != NULL and  parent[f] != root
            If children[ parent[f] ] == NULL
                append parent[f] to L
            endif
        endif
    endif
endwhile
Return min vertex cover set{ v   V | mark[v]=TRUE }
```

Spanning Tree Heuristic Evaluation Function

How can we produce a tree from given graph???

- ✓ We can find **Spanning Tree** of the graph in polynomial time

Spanning Tree Heuristic Evaluation Function

How can we produce a tree from given graph???

- ✓ We can find **Spanning Tree** of the graph in polynomial time

Hence, our heuristic evaluation function h(n),

- Firstly, produce the Depth-First Spanning (DFS) Tree of the remaining graph,
- Then apply our polynomial algorithm to DFS tree and eventually returns the perfect estimation of the number of additional vertices that must be included to cover the edges of the remaining graph.

Spanning Tree Heuristic Evaluation Function

How can we produce a tree from given graph???

✓ We can find **Spanning Tree** of the graph in polynomial time

Hence, our heuristic evaluation function h(n),

- Firstly, produce the Depth-First Spanning (DFS) Tree of the remaining graph,
- Then apply our polynomial algorithm to DFS tree and eventually returns the perfect estimation of the number of additional vertices that must be included to cover the edges of the remaining graph.

Admissible

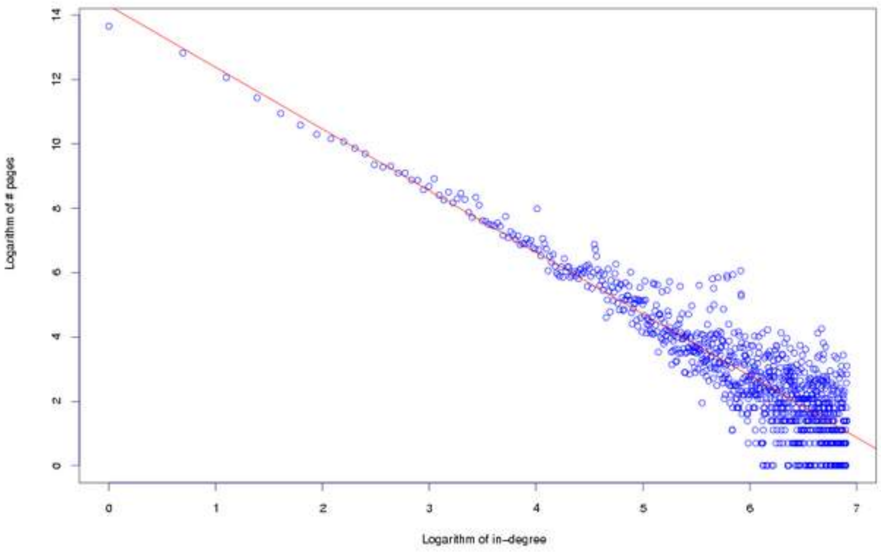
Given a remaining graph any vertex cover of it must be at least as large as a heuristic value returned by our heuristic evaluation function.

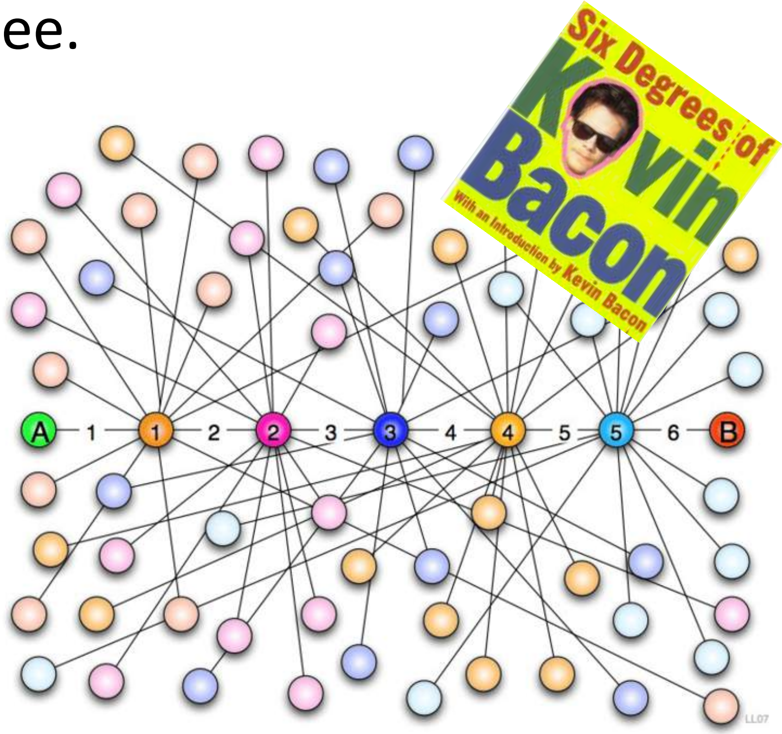
OUTLINE

- ✓ Original Problem
- ✓ Simplified Problem
- ✓ Detailed Problem Space
- ✓ Pruning of Search Tree
- ✓ Node Ordering Heuristic
- ✓ Selection of Search Algorithm
- ✓ Cost Function
- ✓ Design of Heuristic Evaluation Function
- ✓ Empirical Results
- Conclusion
- References

Small World Phenomenon and Power-Law Edge Degree Distribution

- Most of the communication and social networks have power-law link distributions, containing a few nodes that have a very high degree and many with low degree.





Six Degrees of Kevin Bacon
With an Introduction by Kevin Bacon
A
1
1
2
2
3
3
4
4
5
5
6
B
LL07

- The high connectivity nodes play the important role of hubs in communication and networking, a fact that can be exploited when designing efficient search algorithms.

Implementation Setup

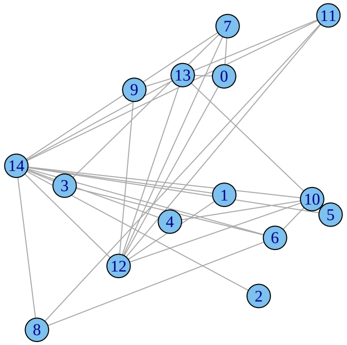
MacBook Pro(Mac OS X 10.7.4):
Memory :   4GB
Processor: 2.5 GHz Intel Core i5

+ **PYTHON** + **igraph library** + **CPROFILE**

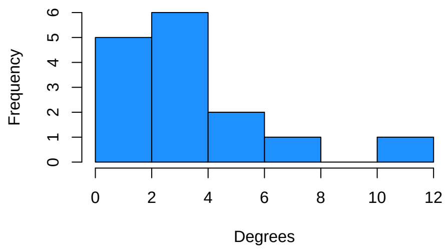
- All search and pruning algorithms were implemented in Python2.7 with igraph library.
- Cprofile is used for profiling algorithms

Test Graphs-I

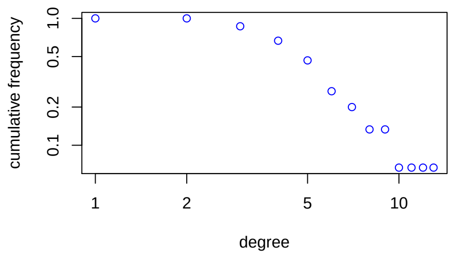
Small World Graph–N(15)
Edge Count: 30



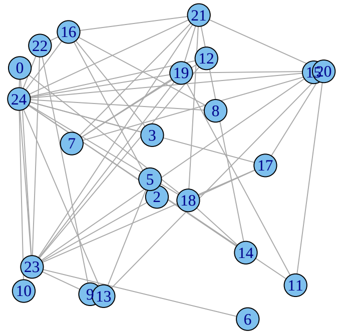
Histogram of Edge Degrees
(Graph–N(15))



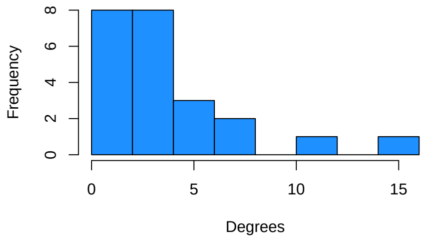
Power–Law Edge Degree Distribution



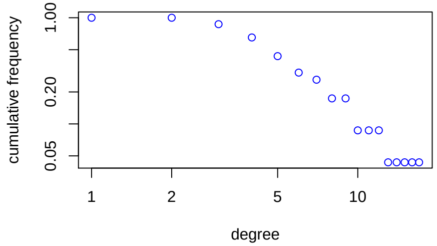
Small World Graph–N(25)
Edge Count: 50



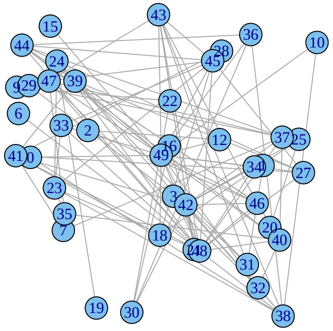
Histogram of Edge Degrees
(Graph–N(25))



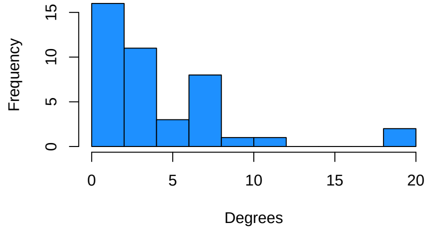
Power–Law Edge Degree Distribution



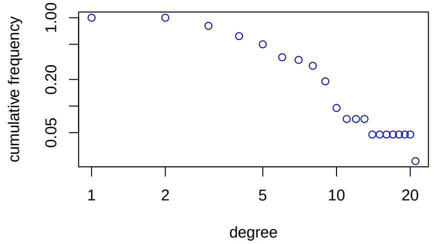
Small World Graph–N(50)
Edge Count: 100



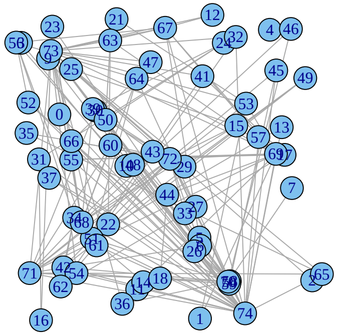
Histogram of Edge Degrees
(Graph–N(50))



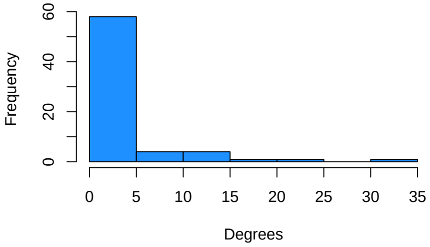
Power–Law Edge Degree Distribution



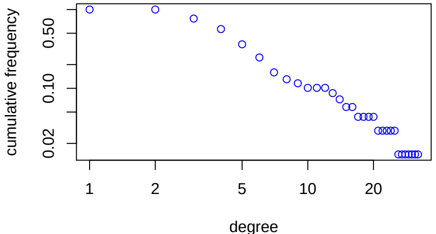
Small World Graph–N(75)
Edge Count: 150



Histogram of Edge Degrees
(Graph–N(75))

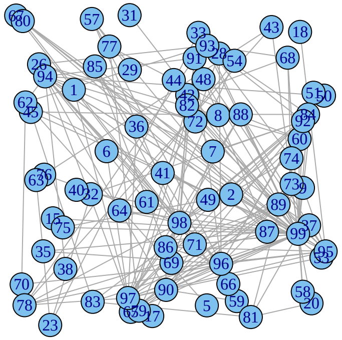


Power–Law Edge Degree Distribution

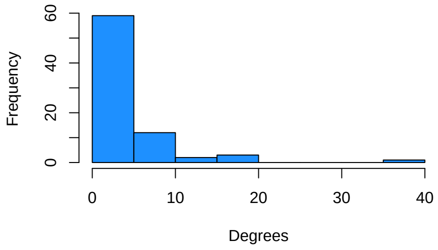


Test Graphs-II

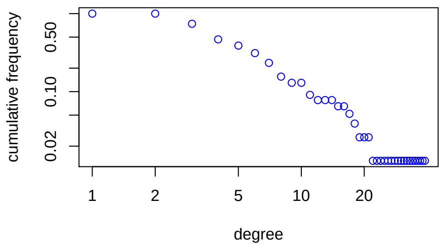
Small World Graph–N(100)
Edge Count: 170



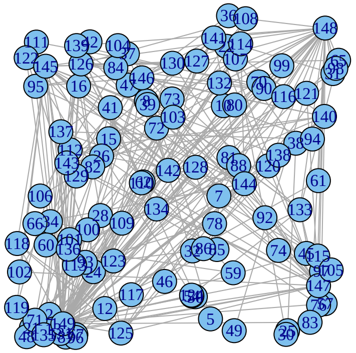
Histogram of Edge Degrees
(Graph–N(100))



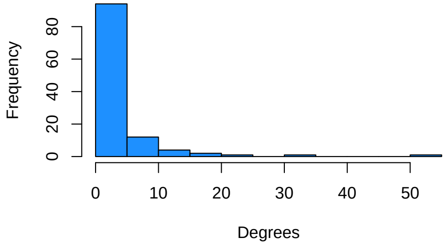
Power–Law Edge Degree Distribution



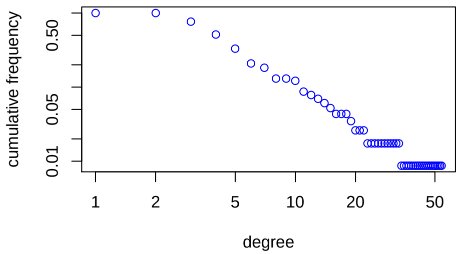
Small World Graph–N(150)
Edge Count: 250



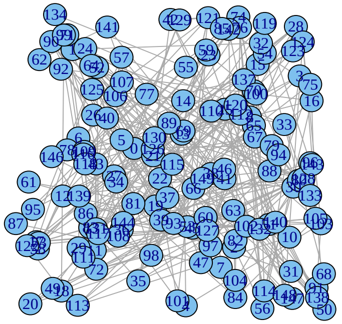
Histogram of Edge Degrees
(Graph–N(150))



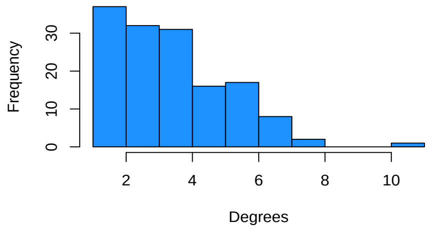
Power–Law Edge Degree Distribution



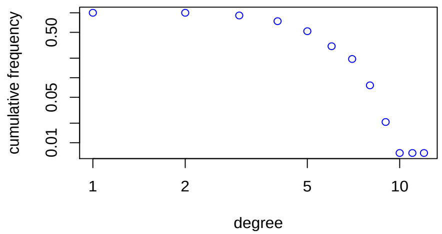
Graph–N(150)
DENSITY:8



Histogram of Edge Degrees
(Graph–N(150))



Edge Degree Distribution



Nodes(N) =150

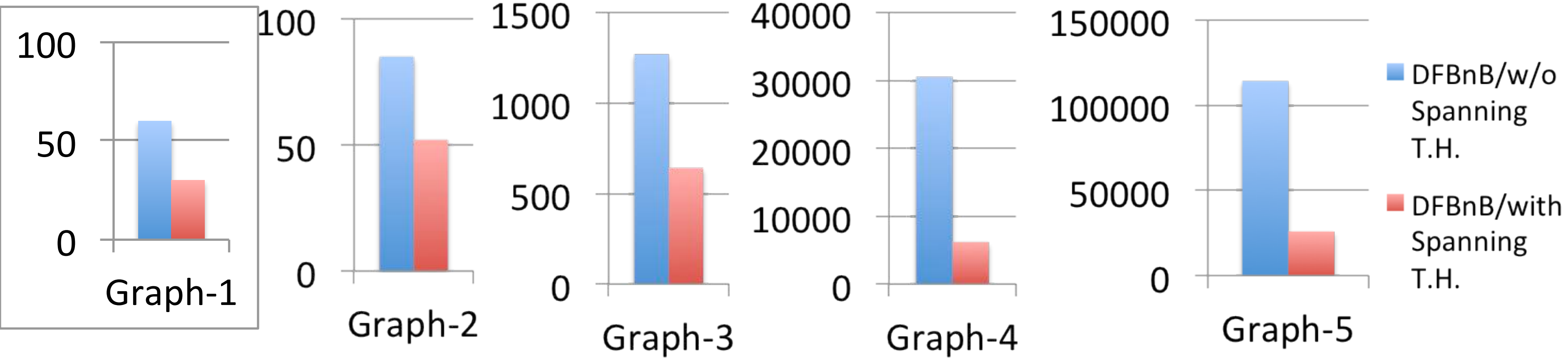
Density(D) = 8

- Each edge was added to graph with probability D/N

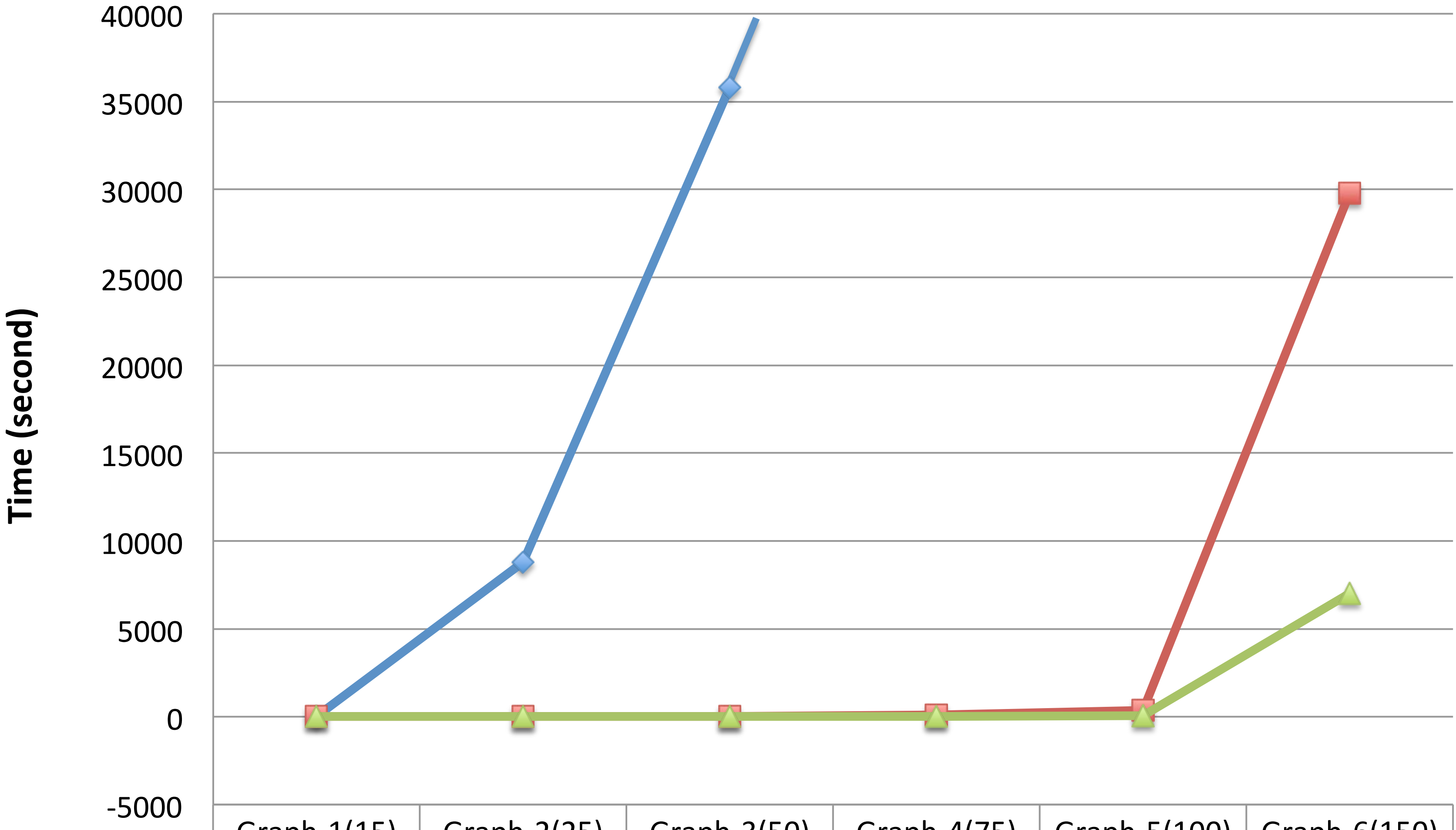
Comparison of Total Tested Seed Sets
Pruning Performance

Total Tested Seed Sets

Algorithm	Graph-1 (15)	Graph-2 (25)	Graph-3 (50)	Graph-4 (75)	Graph-5 (100)
Brute Force Search	32768	33.55×10^{6}	11.26×10^{14}	12.68×10^{29}	14.27×10^{44}
DFBnB / Without Spanning Tree Heuristic	0	85	449	25148	72713
DFBnB/ With Spanning Tree Heuristic	0	33	214	3361	5896

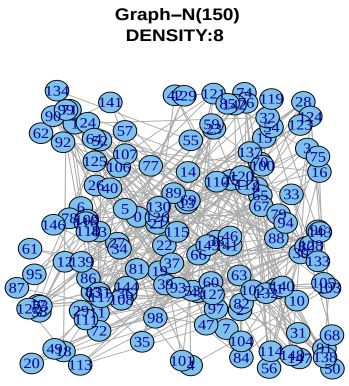


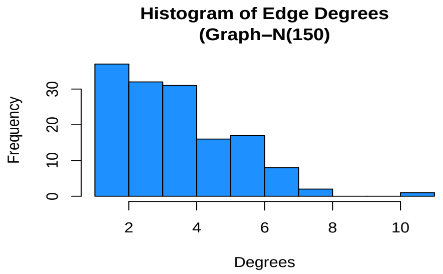
Running Time Performance

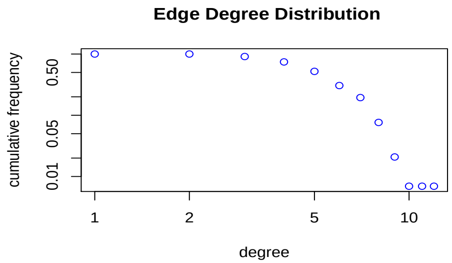


	Graph-1(15)	Graph-2(25)	Graph-3(50)	Graph-4(75)	Graph-5(100)	Graph-6(150)
◆ Brute Force Search	11.003	8794.103	35789.745	—	—	—
■ DFBnB/ w/o Spanning T.H	0.001	0.187	1.378	103.851	352.469	29783.126
▲ DFBnB/with Spanning T.H	0.001	0.077	0.626	15.06	36.337	6998.579

Comparison with Additive Pattern Database 4-clique Heuristic







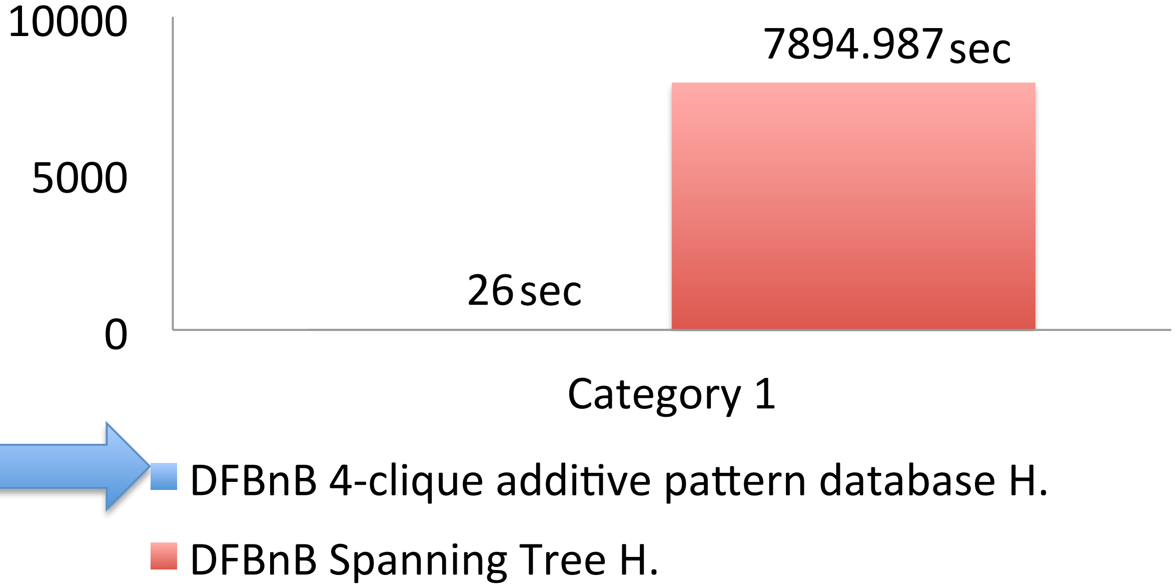
Nodes(N) =150

Density(D) = 8

- Each edge was added to graph with probability D/N

A.Felner,
R.Korf,
S.Hanan

“Additive Pattern Database Heuristic”



OUTLINE

- ✓ Original Problem
- ✓ Simplified Problem
- ✓ Detailed Problem Space
- ✓ Pruning of Search Tree
- ✓ Node Ordering Heuristic
- ✓ Selection of Search Algorithm
- ✓ Cost Function
- ✓ Design of Heuristic Evaluation Function
- ✓ Empirical Results
- ✓ Conclusion
- References

Conclusion

- ✓ The current influence maximization problem is simplified and reduced to minimum vertex cover problem.

- ✓ New polynomial time and admissible Spanning Tree Heuristic Evaluation function that gives always lower bound of optimal solutions is purposed.

- ✓ Pre-evaluation of graph and three pruning techniques that are specially designed for this problem are used in Depth-First Branch-and-Bound (DFBnB) based anytime search algorithm.

- ✓ Empirical results show that newly designed spanning tree heuristic evaluation function reduce the running time of DFBnB algorithm especially in social networks graphs that have large node count and power-law edge degree distribution.

References

- R.E.Korf. Heuristic Search, CS 261A UCLA Course Reader
- A.Felner, R.E.Korf, S.Hanan. An Additive Pattern Database Heuristic
- D.Kempe, J.Kleinberg, E.Tardos. Maximizing the Spread of Influence through a Social Network
- W.Chen, Y.Wang, S.Yang. Efficient Influence Maximization in Social Networks
- W.Chen,Y.Yuan, L.Zhang. Scalable Influence Maximization in Social Networks under the Linear Threshold Model
- A.Goyal, W.Lu, L.V.S. Lakshmanan. SIMPATH: An Efficient Algorithm for Influence Maximization under the Linear Threshold Model
- D.Easley, J.Kleinberg. Networks, Crowds, and Markets: Reasoning about Highly Connected World, Cambridge University Press
- L.Adamic, R.Lukose, A.Puniyani, B.Huberman. Search in Power-Law Networks, Physical Review E, Volume-64, 046134