CSEP 521
Applied Algorithms
Spring 2005

Course Introduction

Graph Algorithms

Outline for the Evening

- Course administration
- Algorithm Design Process
- Spanning Tree
  - Depth-First Search
  - In-class exercise
  - Breath-First Search
- Minimum Spanning Tree
- Set Disjoint Union / Find

Instructors

- Instructor
  - Richard Ladner
  - ladner@cs.washington.edu
  - 206 543-9347
- TA
  - Neva Cherniavsky
  - (ncherna@cs.washington.edu)

Resources

- CSEP 521 Course Web Page
  - http://www.cs.washington.edu/csep521
- Papers and Sections from Books
- Recommended Algorithms Book
  - Introduction to Algorithms, 2nd Edition by Cormen, Leiserson, Rivest, and Stein
- E-mail list
  - For information from instructors
  - Check web page to sign up
- Message Board
  - For discussion

Engagement by Students

- Weekly Assignments
  - Algorithm design and evaluation
  - Algorithm animation
- In-class activities
- Project with a written report
  - Evaluate several alternative approaches to algorithmically solve a problem
  - Must include readings from literature
  - May include an implementation study
  - May be done in small teams

Final Exam and Grading

- There will be no Final Exam
- Percentages
  - Weekly Assignments 60%
  - Project 40%

Some Topics

- Graph Algorithms
- Maximum Flow
- Linear Programming
- Data Compression
- Computational Geometry
- Computational Biology

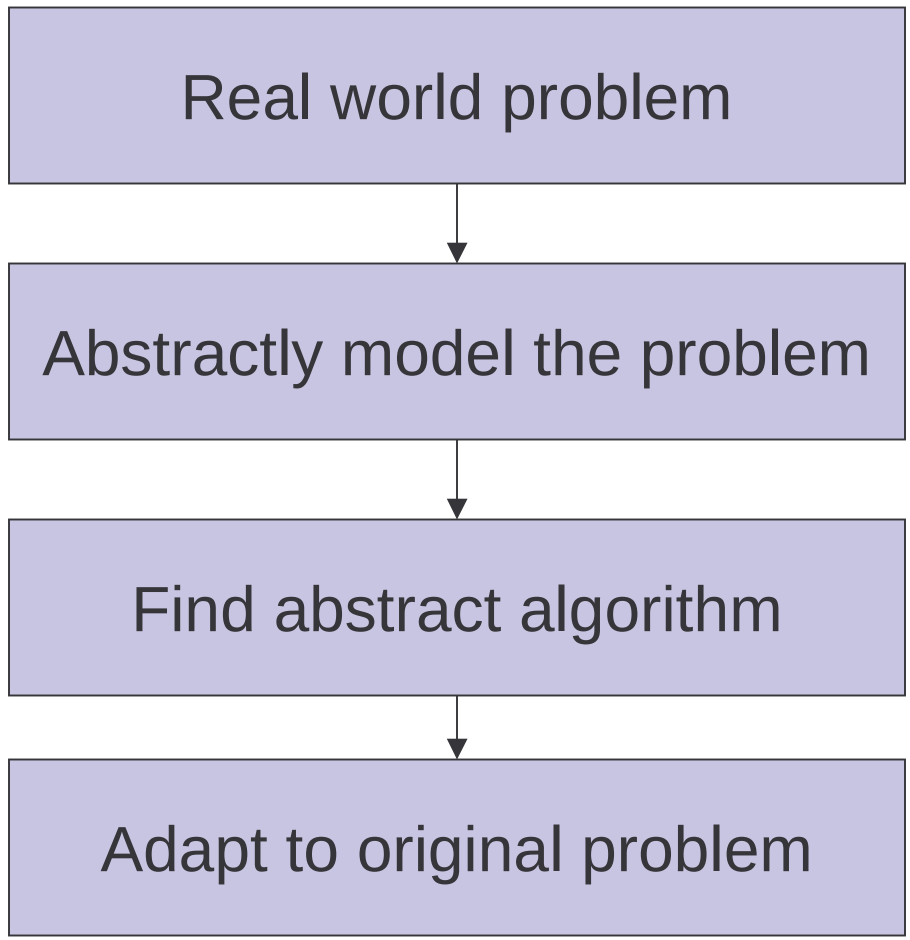
Along the Way

- Analysis of algorithms
- Data structures
- NP-completeness
- Dynamic programming
- Greedy algorithms
- Branch-and-bound algorithms
- Approximation algorithms
- Classics of algorithms

Reading

- Chapter 21 - Disjoint Union / Find
- Chapter 22 - Graph algorithms
- Chapter 23 - Minimum Spanning Tree
- Chapter 24 - Shortest Paths

Applied Algorithm Scenario



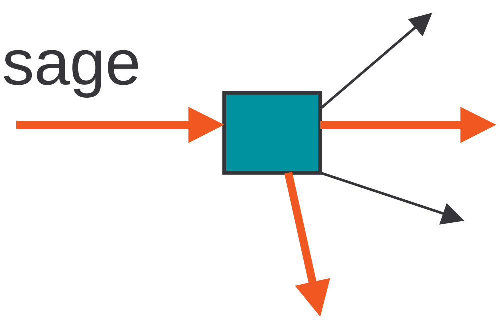
Modeling

- What kind of algorithm is needed
  - Sorting or Searching
  - Graph Problem
  - Linear Programming
  - Dynamic Programming
  - Clustering
  - Algebra
- Can I find an algorithm or do I have to invent one

Broadcasting in a Network

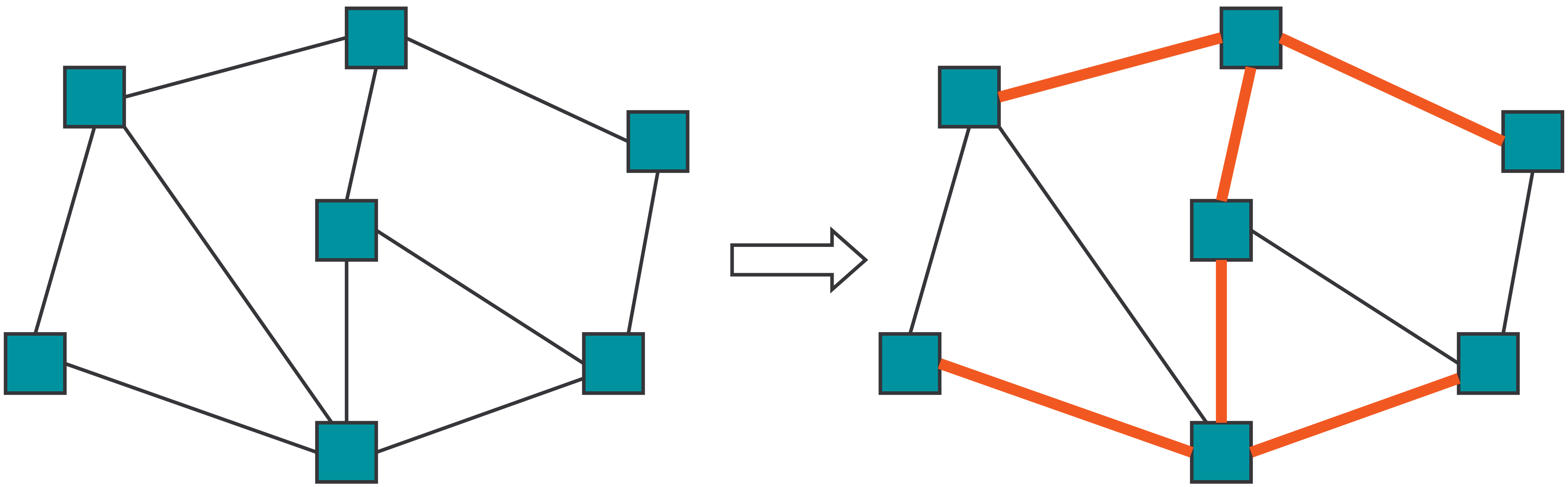
- Network of Routers
  - Organize the routers to efficiently broadcast messages to each other

Incoming message



• Duplicate and send to some neighbors.
• Eventually all routers get the message

Spanning Tree in a Graph

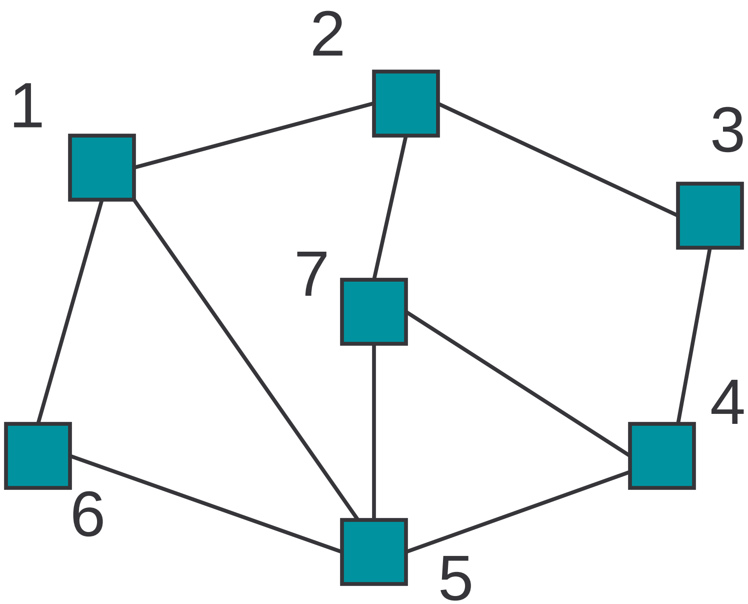


Vertex = router
Edge = link between routers

Spanning tree
- Connects all the vertices
- No cycles

Undirected Graph

- G = (V,E)
  - V is a set of vertices (or nodes)
  - E is a set of unordered pairs of vertices



V = {1,2,3,4,5,6,7}
E = {{1,2},{1,6},{1,5},{2,7},{2,3},
{3,4},{4,7},{4,5},{5,6}}

2 and 3 are adjacent
2 is incident to edge {2,3}

Spanning Tree Problem

- Input: An undirected graph G = (V,E). G is connected.
- Output: T contained in E such that
  - (V,T) is a connected graph
  - (V,T) has no cycles

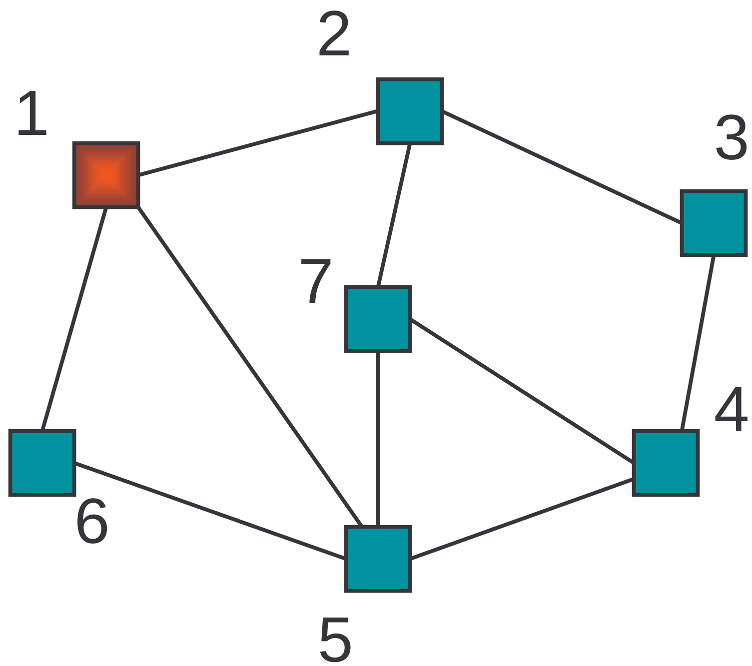
Depth First Search Algorithm

- Recursive marking algorithm
- Initially every vertex is unmarked

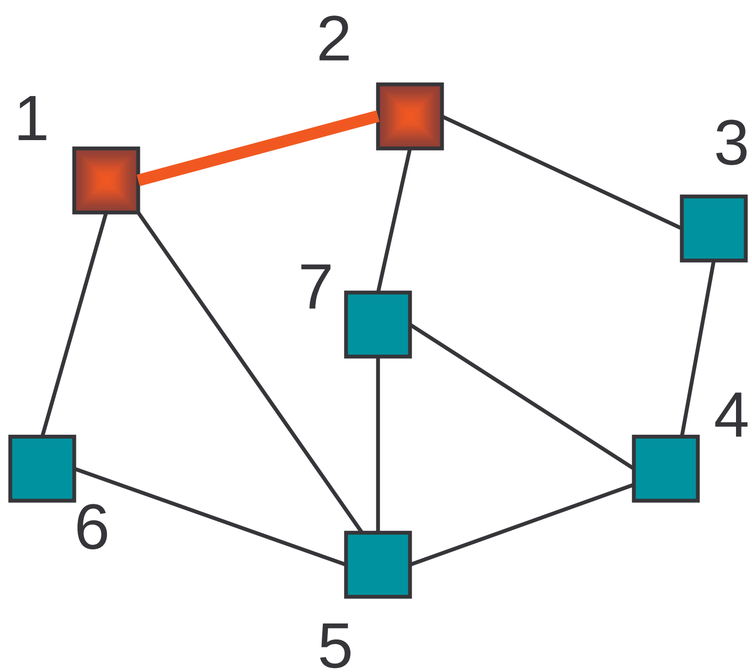
```
DFS(i: vertex)
   mark i;
   for each j adjacent to i do
      if j is unmarked then DFS(j)
end{DFS}
```

Example of Depth First Search

DFS(1)

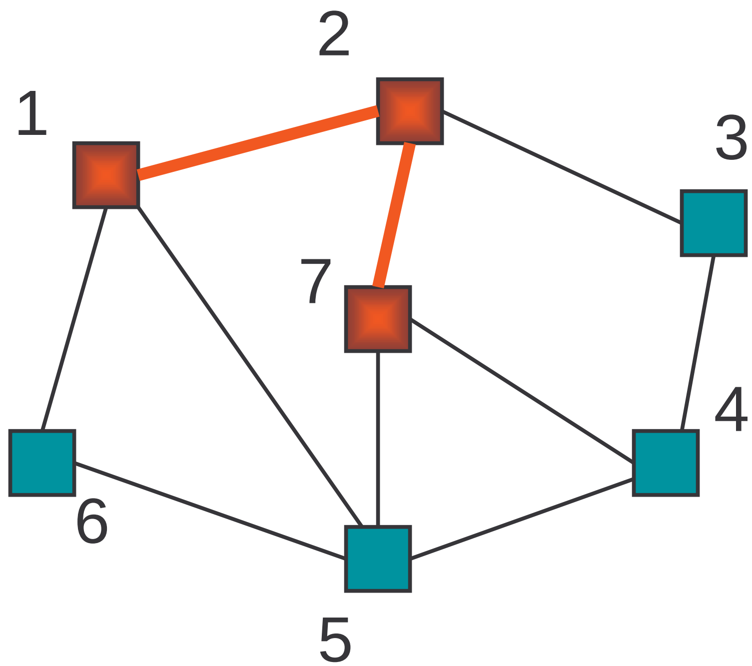


Example Step 2



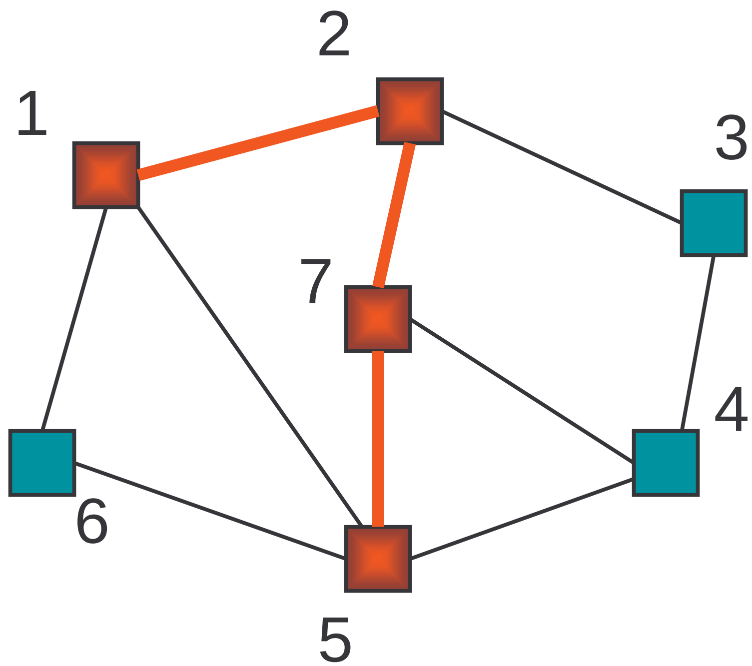
DFS(1)
DFS(2)

Example Step 3



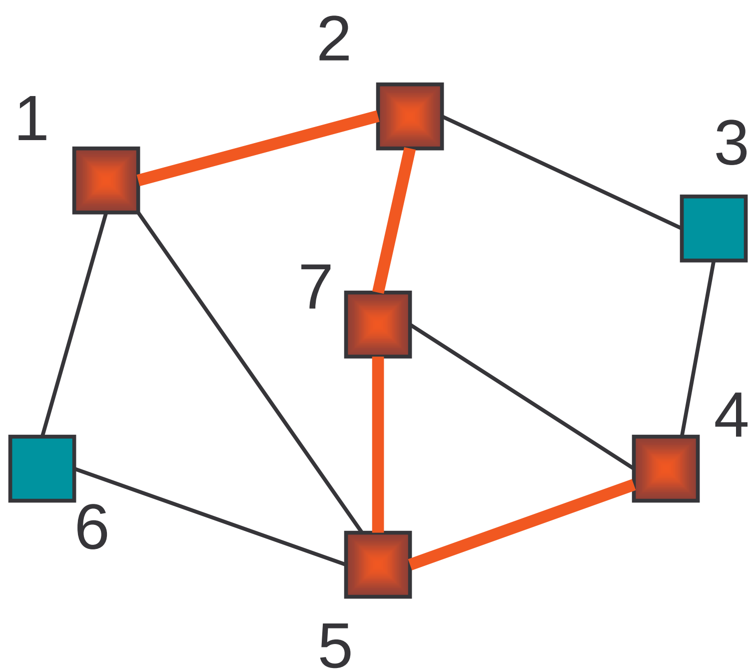
DFS(1)
DFS(2)
DFS(7)

Example Step 4



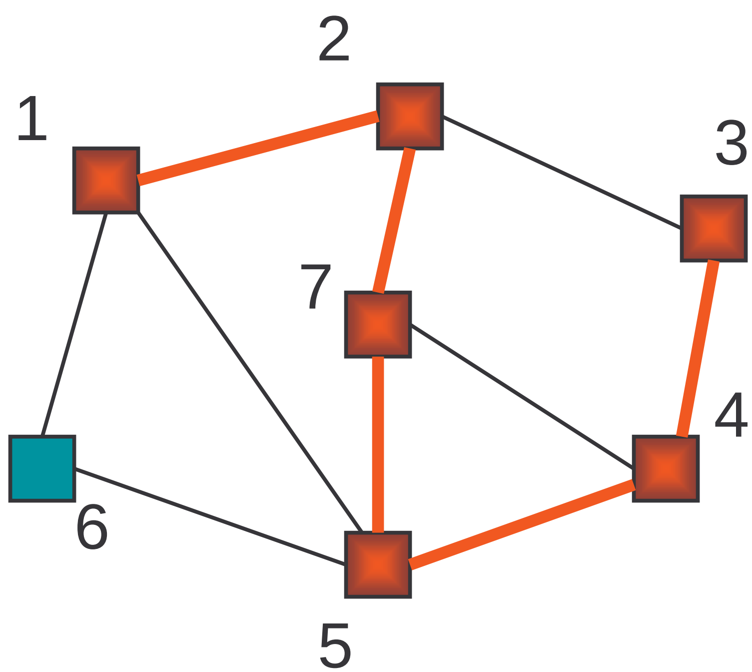
DFS(1)
DFS(2)
DFS(7)
DFS(5)

Example Step 5



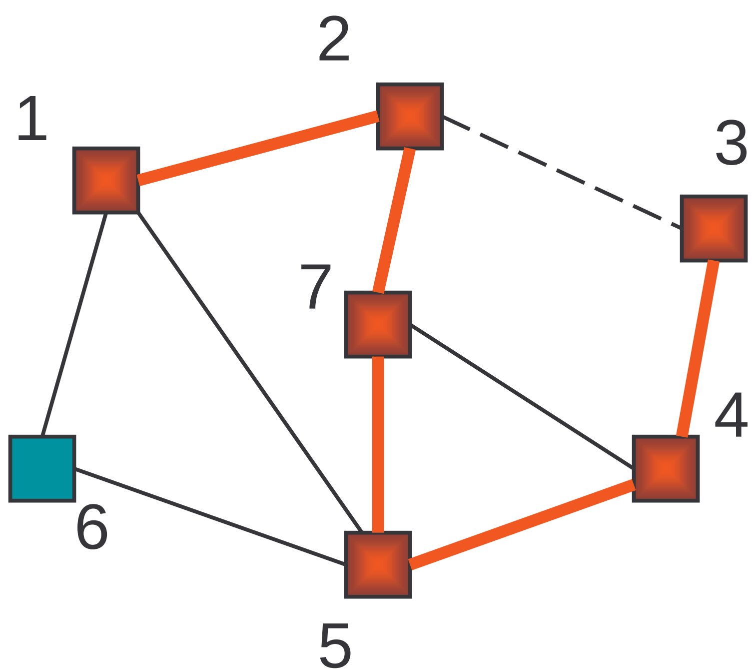
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(4)

Example Step 6



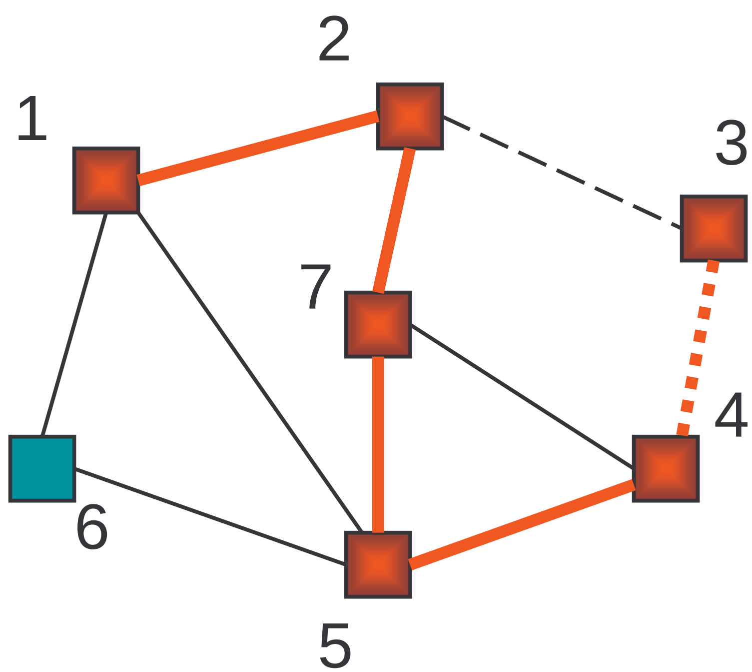
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(4)
DFS(3)

Example Step 7



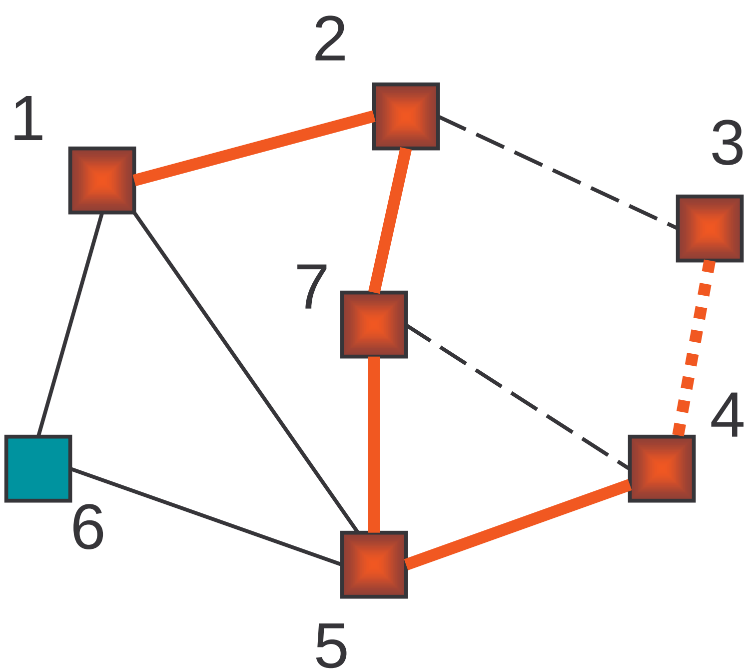
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(4)
DFS(3)

Example Step 8



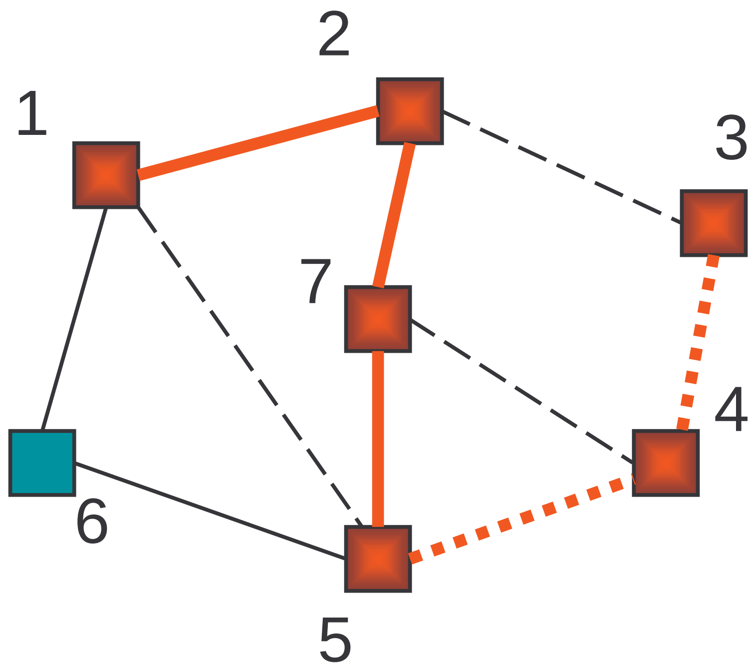
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(4)

Example Step 9



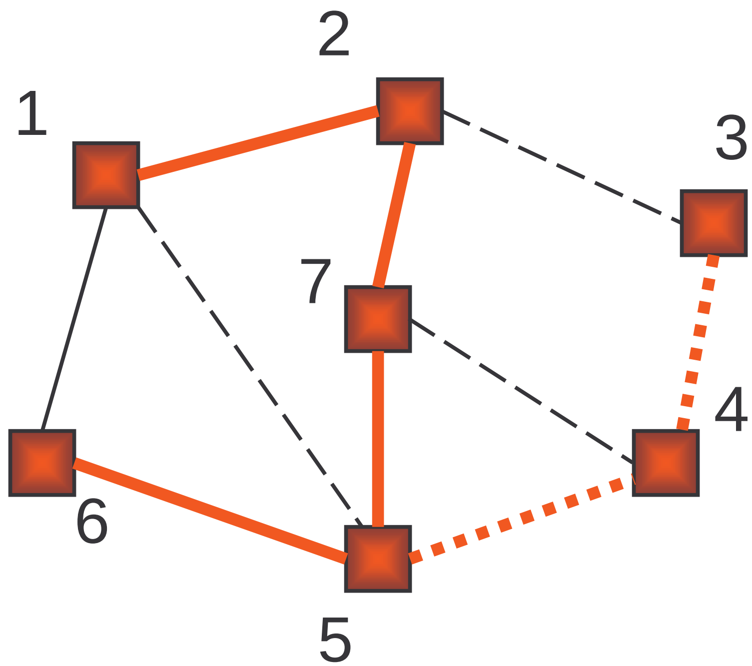
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(4)

Example Step 10



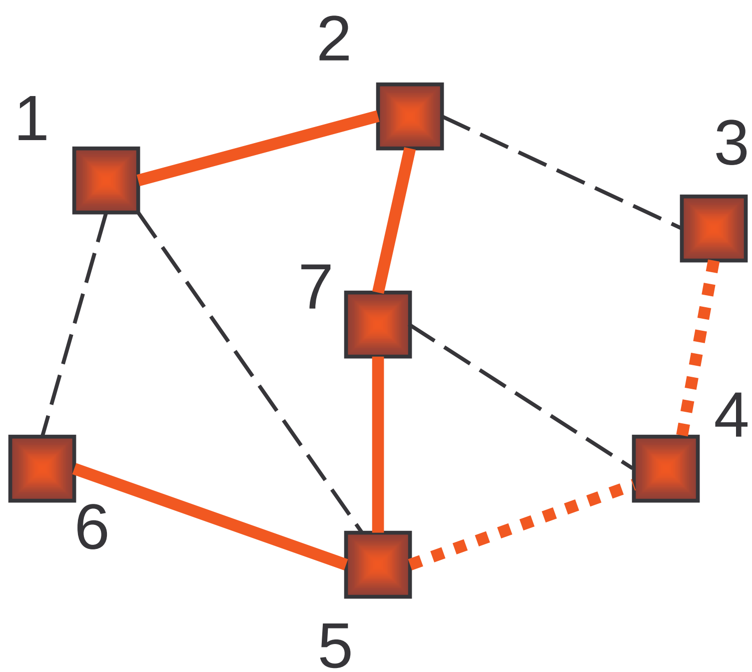
DFS(1)
DFS(2)
DFS(7)
DFS(5)

Example Step 11



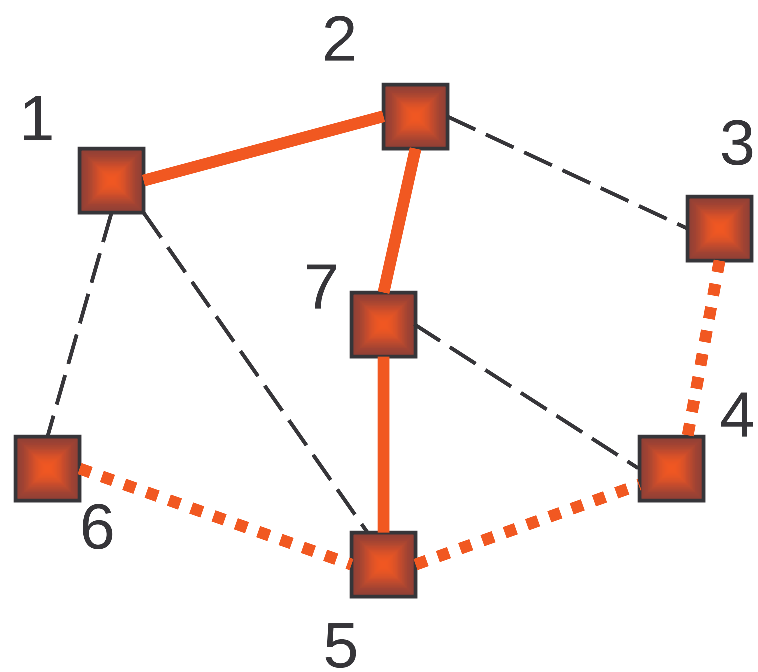
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(6)

Example Step 12



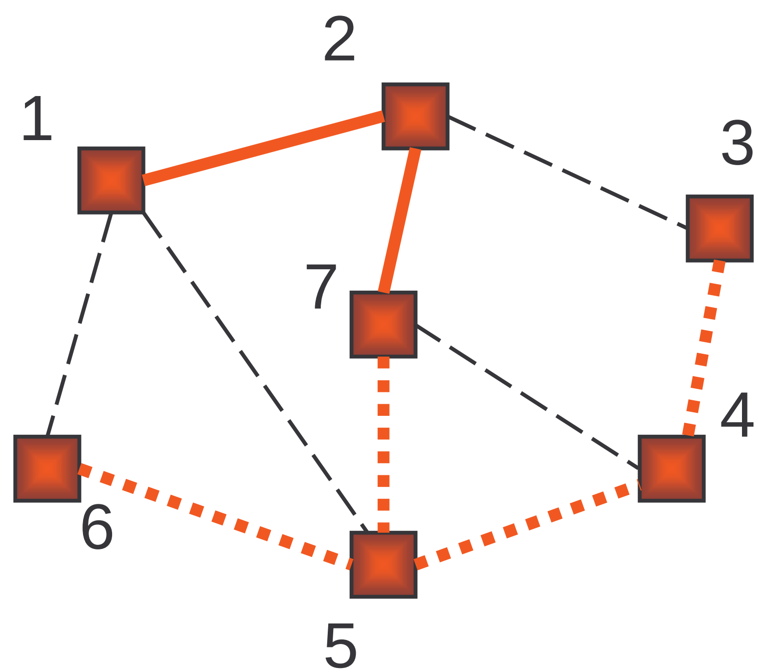
DFS(1)
DFS(2)
DFS(7)
DFS(5)
DFS(6)

Example Step 13



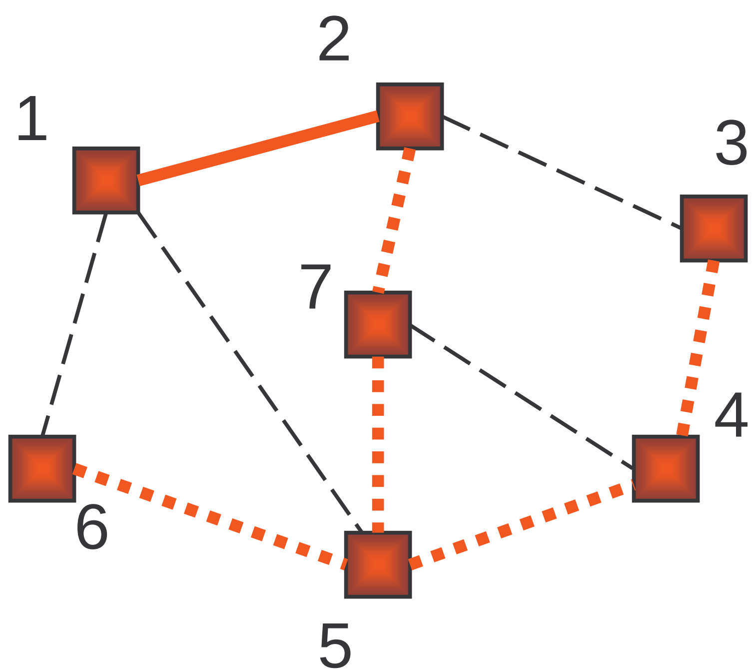
DFS(1)
DFS(2)
DFS(7)
DFS(5)

Example Step 14



DFS(1)
DFS(2)
DFS(7)

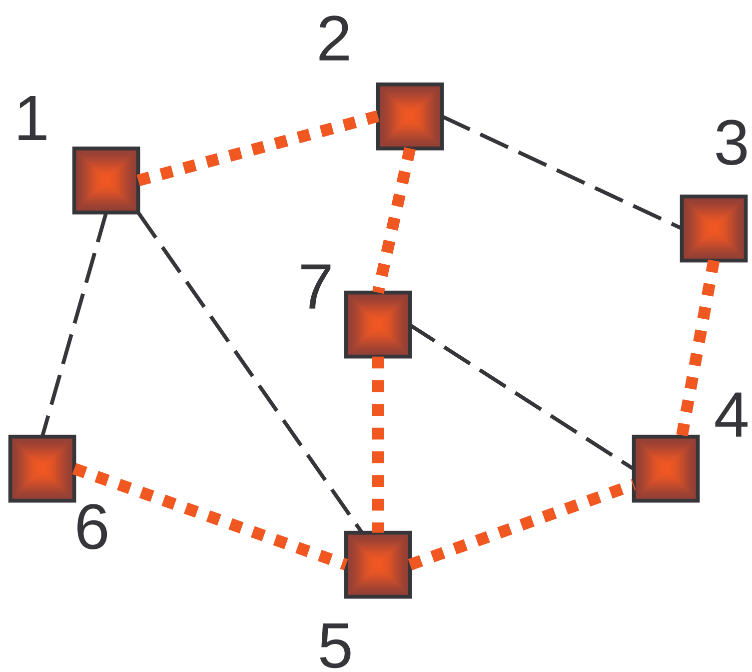
Example Step 15



DFS(1)
DFS(2)

Example Step 16

DFS(1)

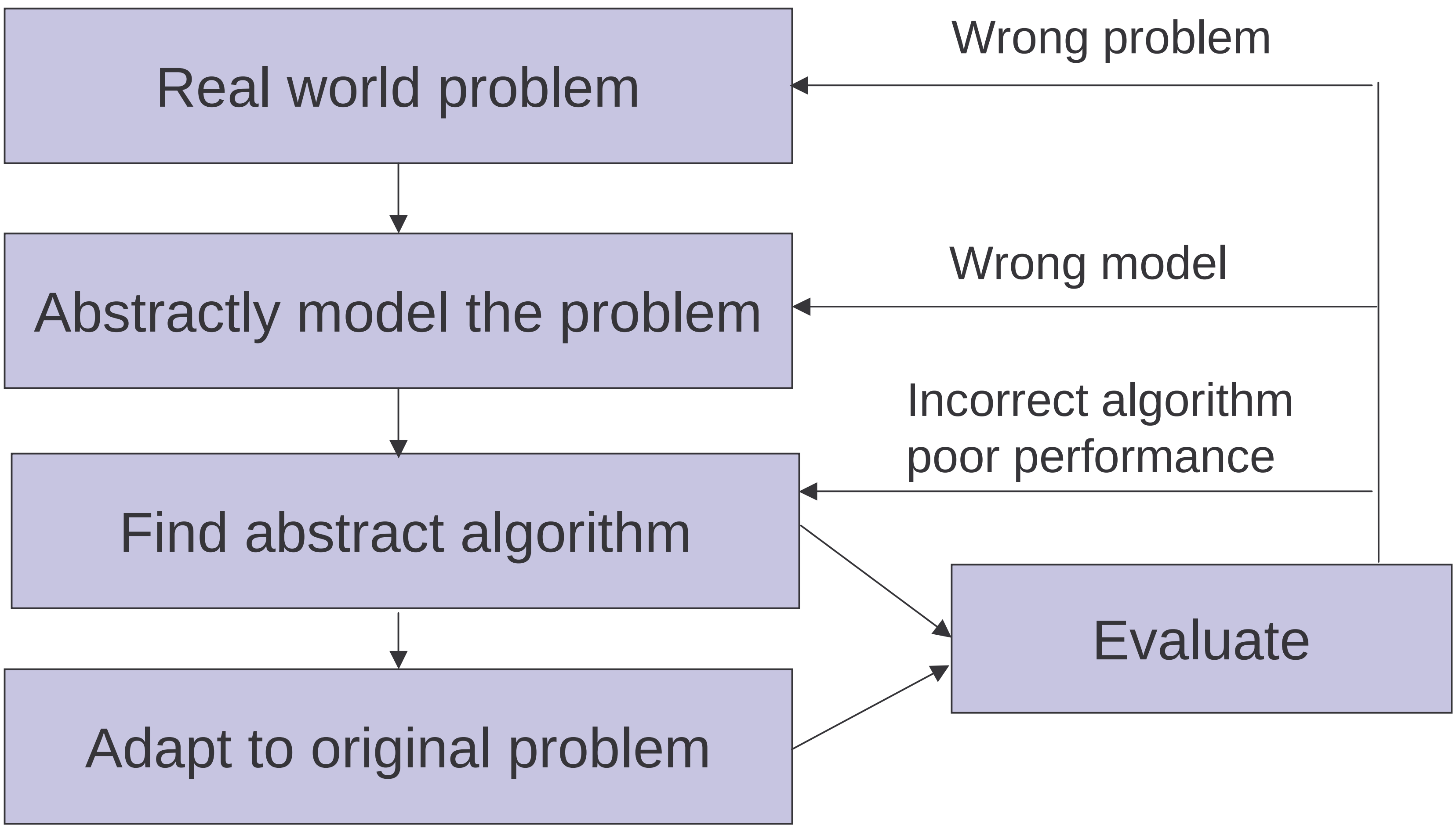


Spanning Tree Algorithm

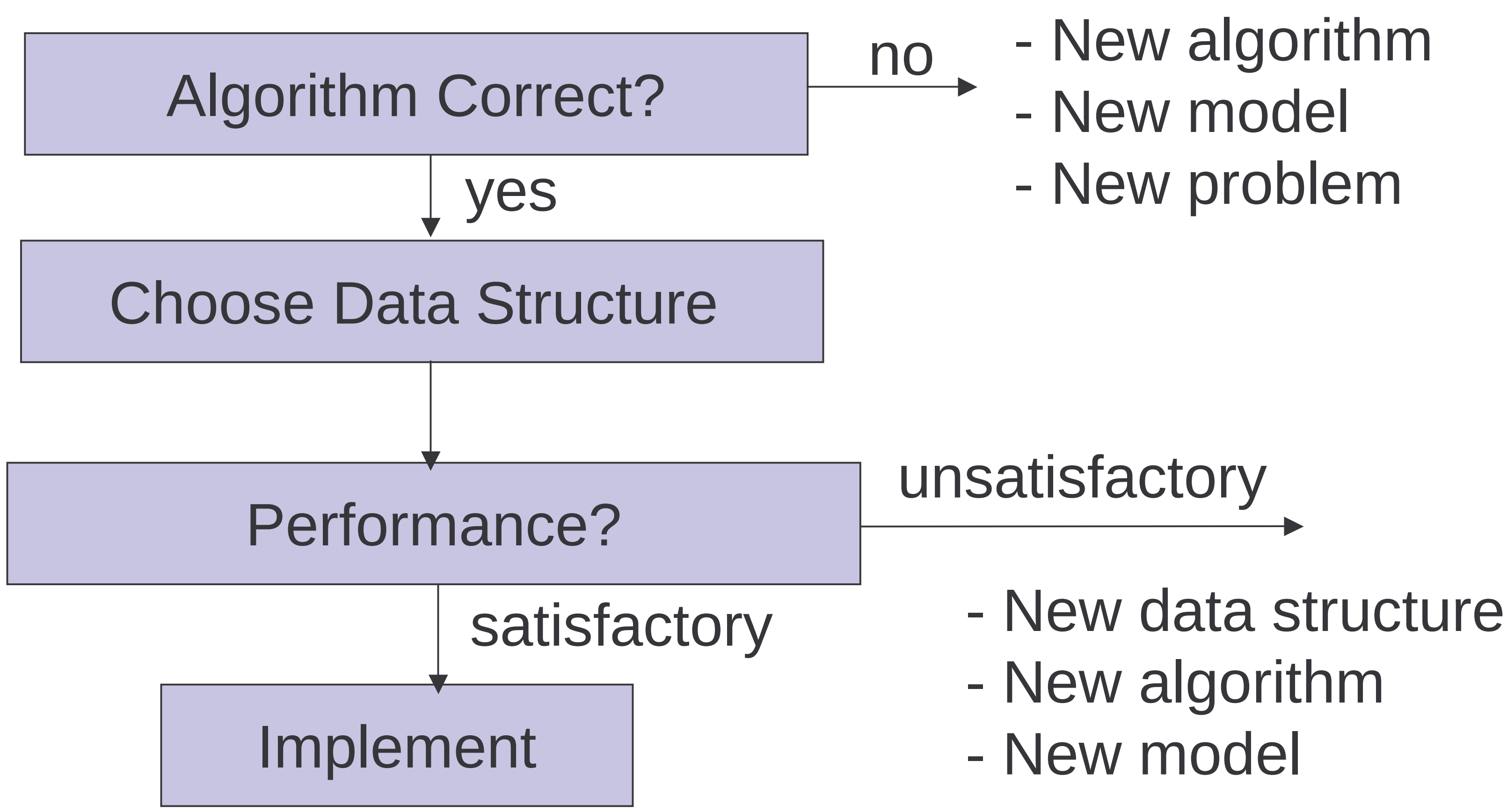
```
ST(i: vertex)
    mark i;
    for each j adjacent to i do
        if j is unmarked then
           Add {i,j} to T;
           ST(j);
end{ST}
```

```
Main
T := empty set;
ST(1);
end{Main}
```

Applied Algorithm Scenario

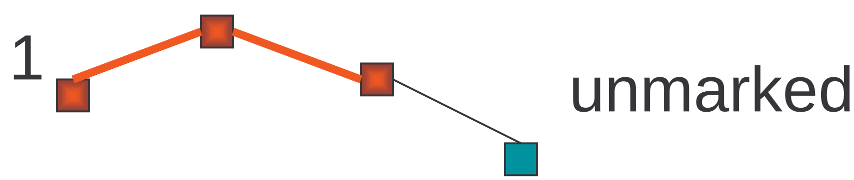


Evaluation Step Expanded



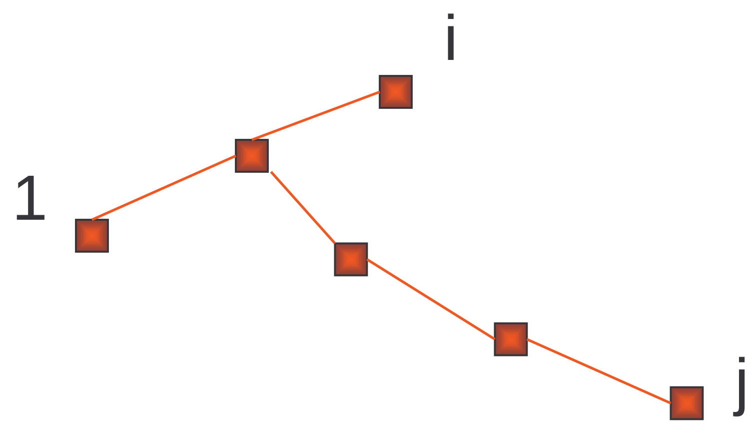
Correctness of ST Algorithm

- There are no cycles in T
  - This is an invariant of the algorithm.
  - Each edge added to T goes from a vertex in T to a vertex not in T.
- If G is connected then eventually every vertex is marked. (Proof by contradiction)

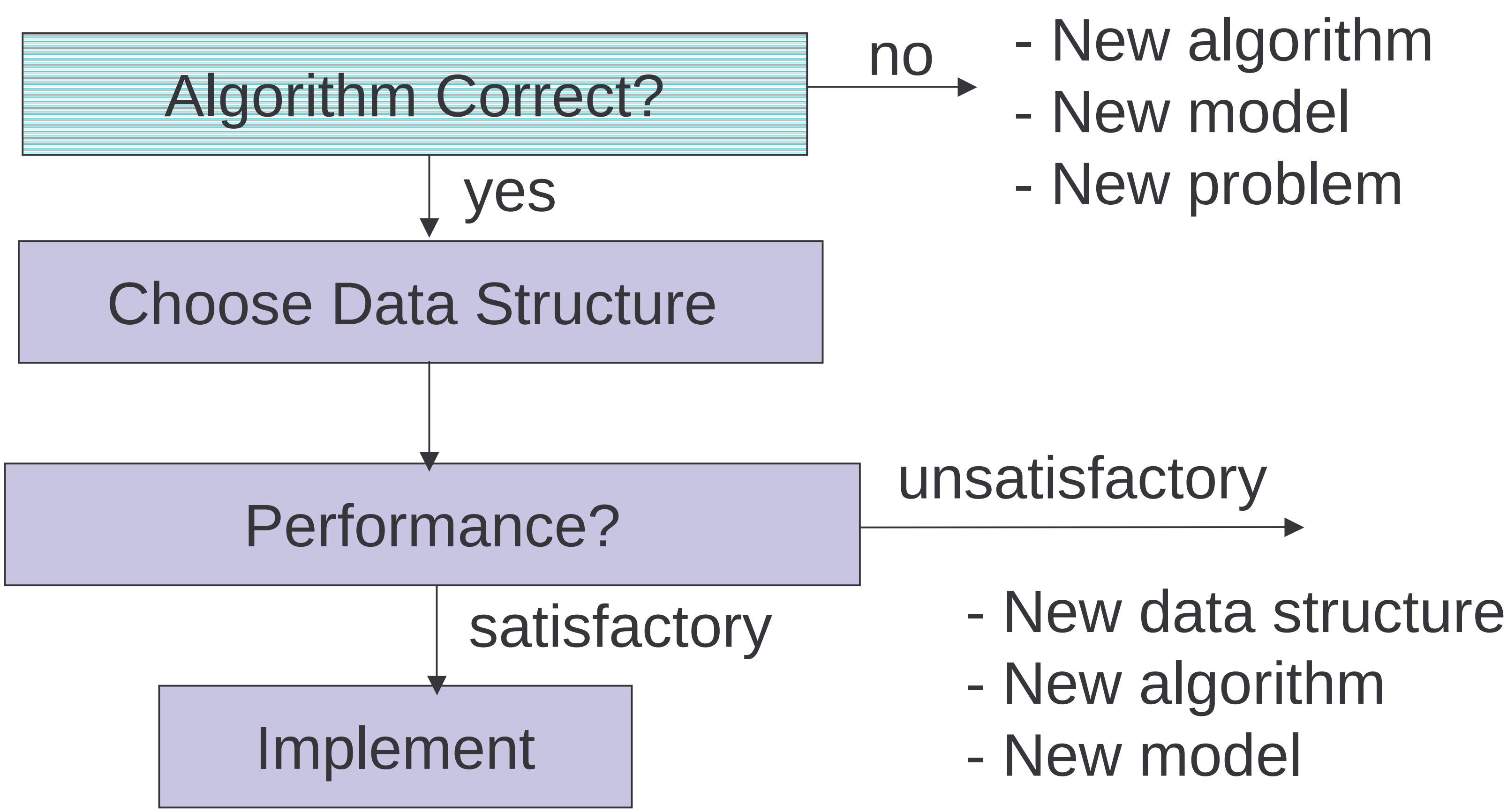


Correctness (cont.)

- If G is connected then so is (V,T)



Data Structure Step

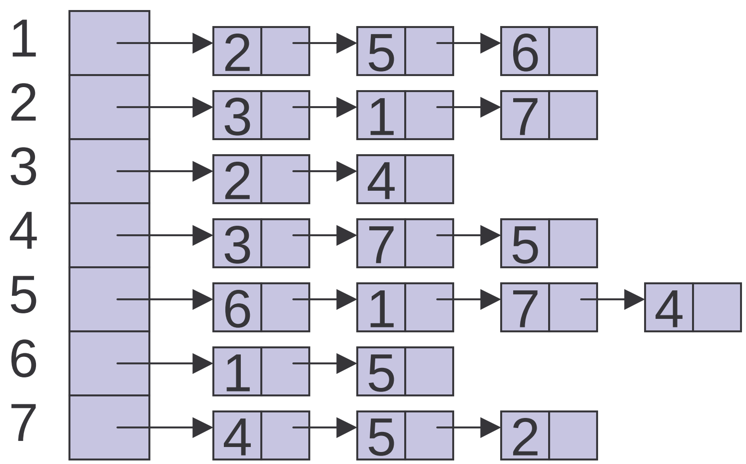


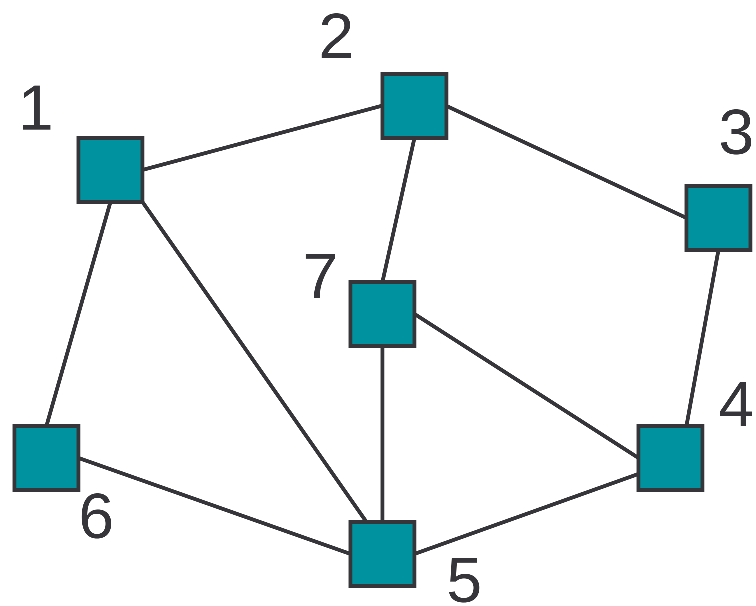
Edge List and Adjacency Lists

- List of edges

1	5	1	2	2	3	5	7	5	5
2	1	6	7	3	4	6	4	7	4

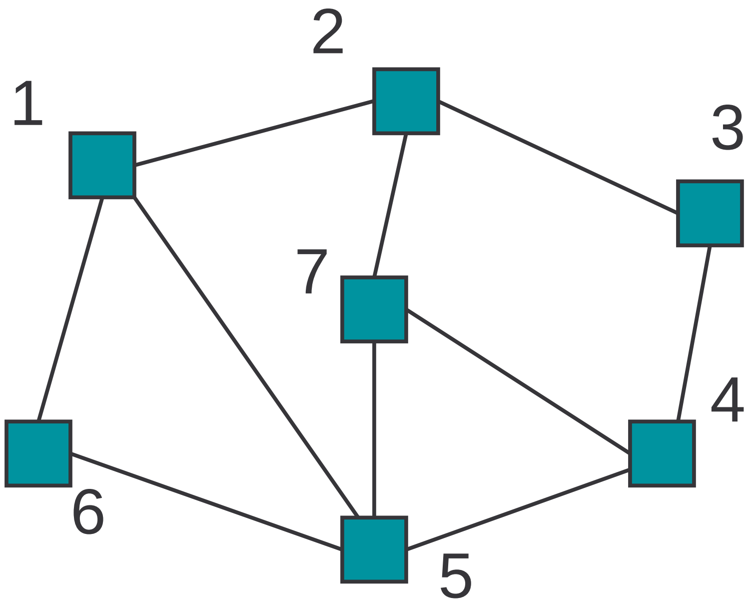
- Adjacency lists





Adjacency Matrix

	1	2	3	4	5	6	7
1	0	1	0	0	1	1	0
2	1	0	1	0	0	0	1
3	0	1	0	1	0	0	0
4	0	0	1	0	1	0	1
5	1	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	1	0	1	1	0	0



Data Structure Choice

- Edge list
  - Simple but does not support depth first search
- Adjacency lists
  - Good for sparse graphs
  - Supports depth first search
- Adjacency matrix
  - Good for dense graphs
  - Supports depth first search

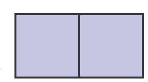
Spanning Tree with Adjacency Lists

```
ST(i: vertex)
    M[i] := 1;
    v := G[i];
    while  not(v = null)
        j := v.vertex;
        if M[j] = 0 then
            Add {i,j} to T;
            ST(j);
        v := v.next;
end{ST}
```

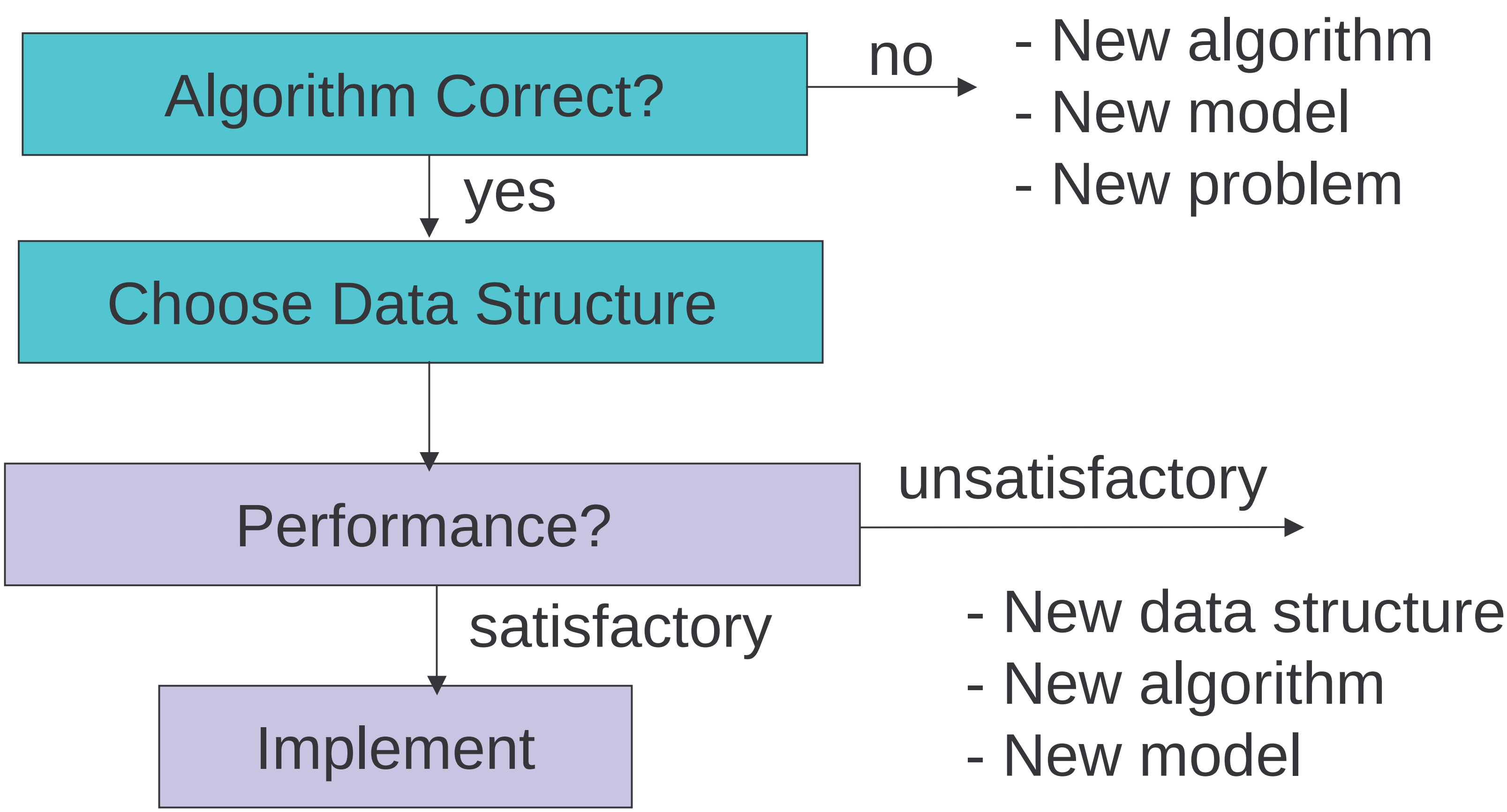
```
Main
    G is array of adjacency lists;
    M[i] := 0 for all i;
    T is empty;
    Spanning_Tree(1);
end{Main}
```

M is the marking array

Node of linked list



Performance Step



Performance of ST Algorithm

- n vertices and m edges
- Connected graph
- Storage complexity O(m)
- Time complexity O(m)

Other Uses of Depth First Search

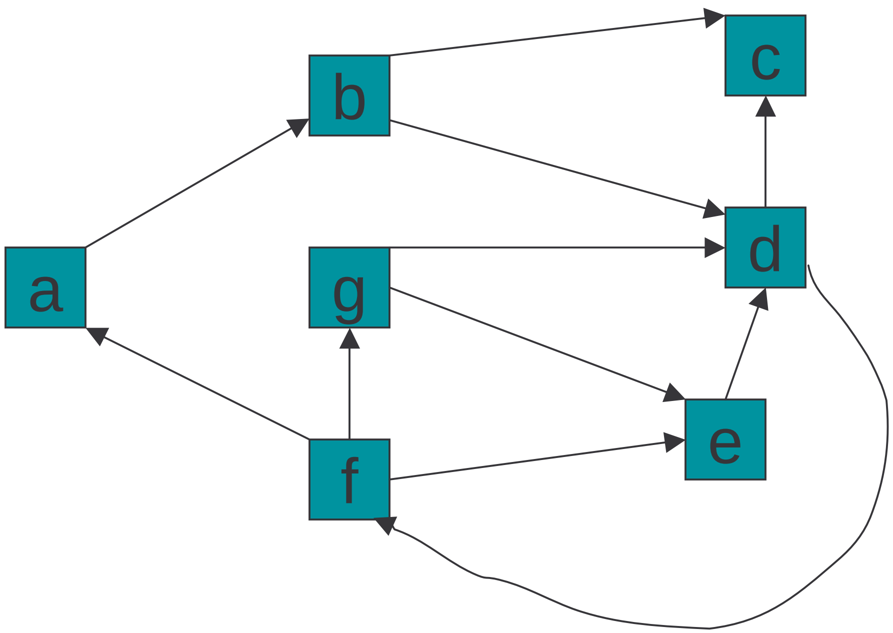
- Popularized by Hopcroft and Tarjan 1973
- Connected components
- Biconnected components
- Strongly connected components in directed graphs
- topological sorting of a acyclic directed graphs

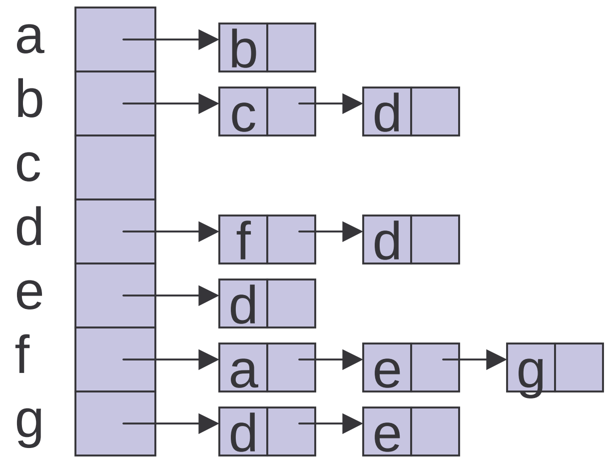
Depth-First Search in Directed Graphs

- Discovery and Finish Times
- Initially D[i] = F[i] = 0, time = 1

```
DFS(i: vertex)
    D[i] := time;
    time++;
    v := G[i];
    for each vertex j adjacent to i do
          if D[j] = 0 then DFS(j)
    F[i] := time;
    time++;
end{DFS}
```

Example





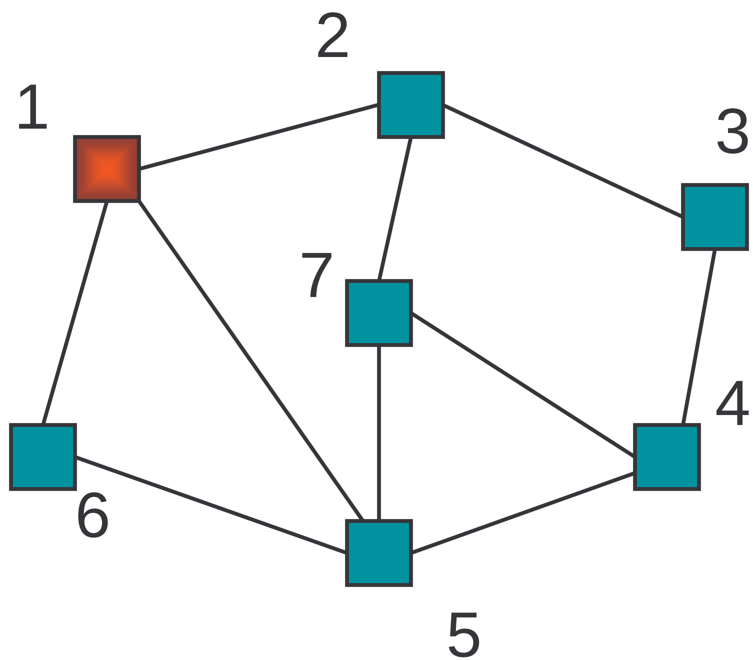
- Compute the discovery and finish times
- Classify the edges

Edge Classification

- Forward Edge (i,j)
  - D[i] < D[j] < F[j] < F[i]
- Backward Edge
  - D[j] < D[i] < F[i] < F[j]
- Cross Edge
  - D[j] < F[j] < D[i] < F[i]
- Note – A directed graph is acyclic if and only if it has no backward edges in a DFS.

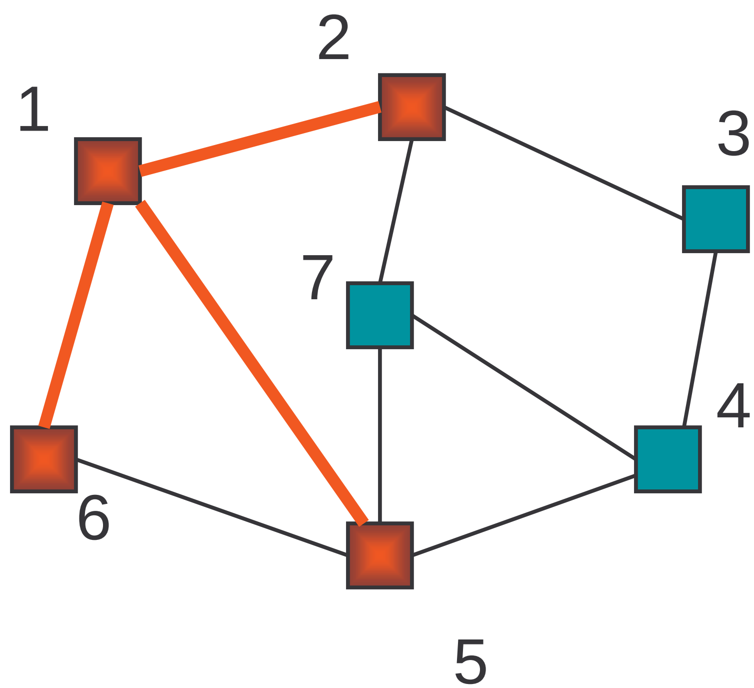
ST using Breadth First Search 1

- Uses a queue to order search



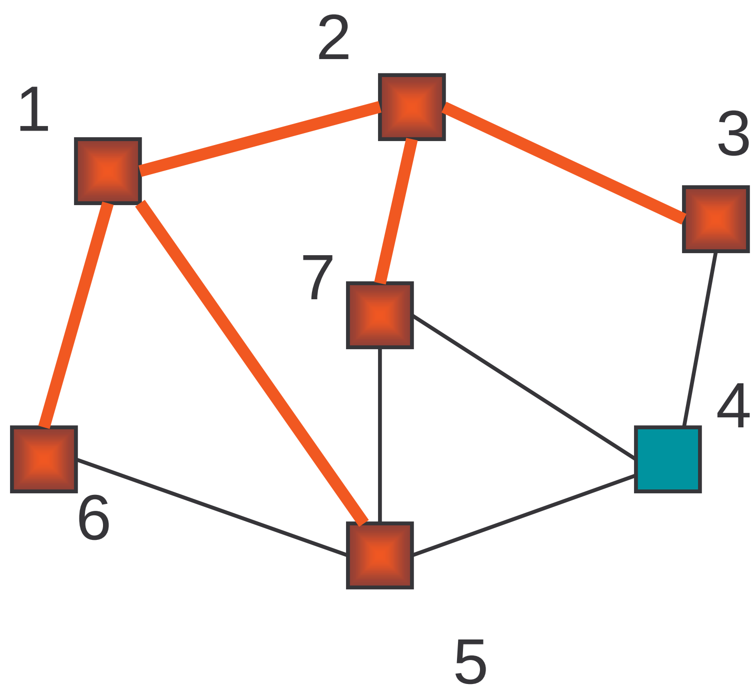
Queue = 1

Breadth First Search 2



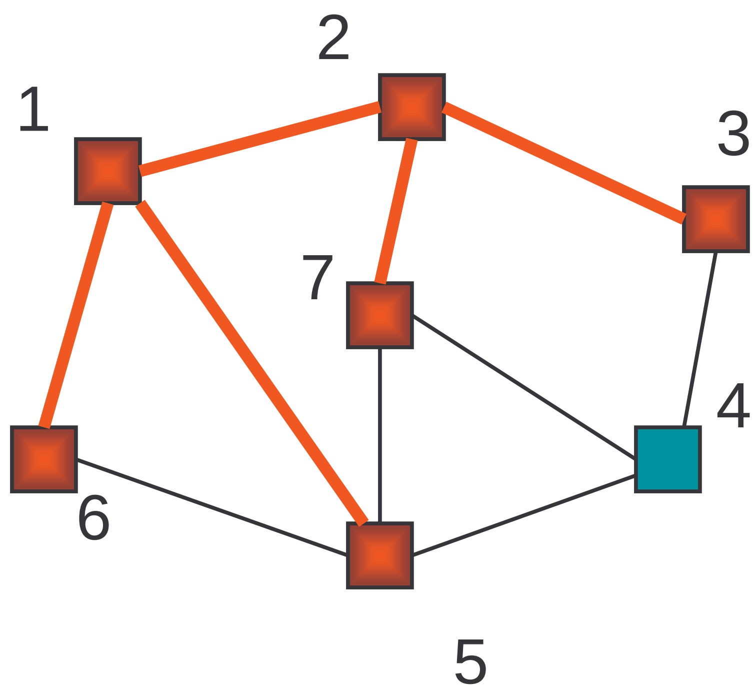
Queue = 2,6,5

Breadth First Search 3



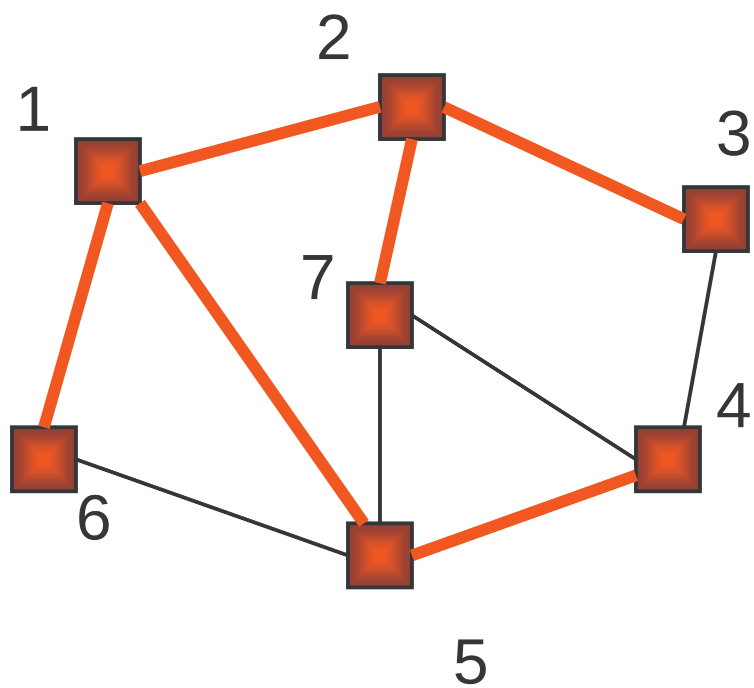
Queue = 6,5,7,3

Breadth First Search 4



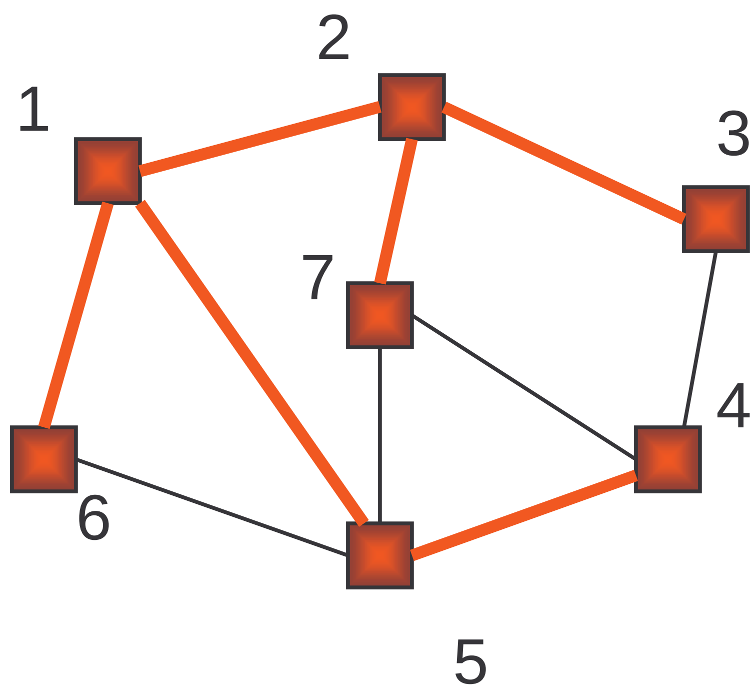
Queue = 5,7,3

Breadth First Search 5



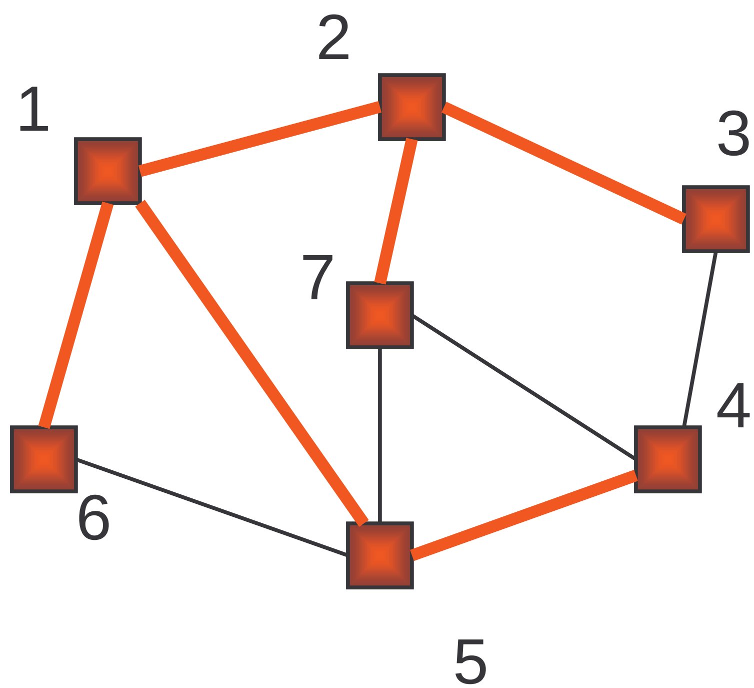
Queue = 7,3,4

Breadth First Search 6



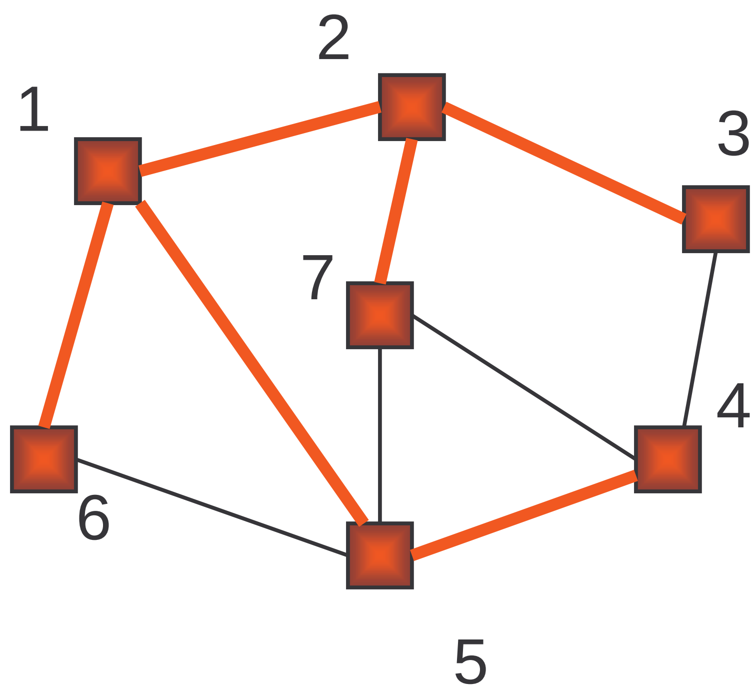
Queue = 3,4

Breadth First Search 7



Queue = 4

Breadth First Search 8



Queue =

Spanning Tree using Breadth First Search

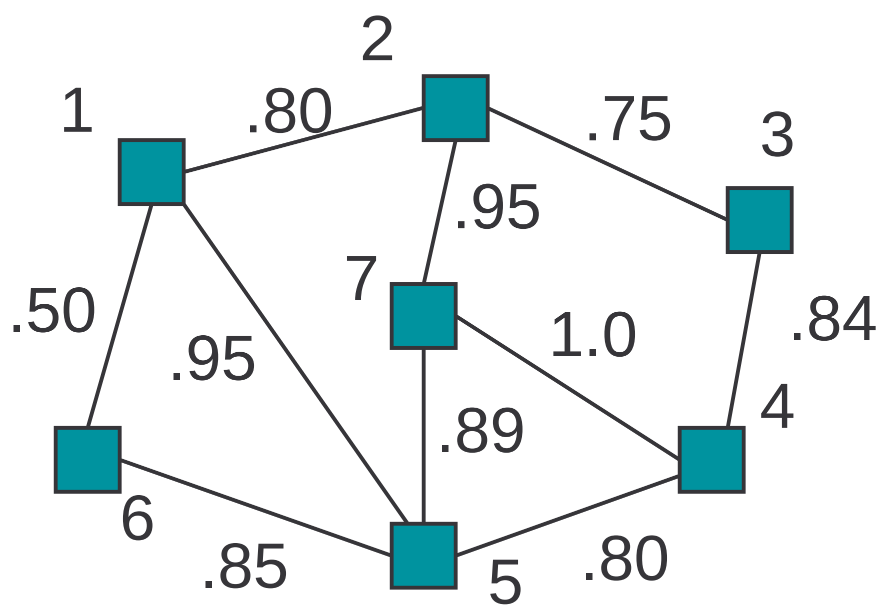
```
BFS
Initialize T to be empty;
Initialize Q to be empty;
Enqueue(1,Q) and mark 1;
while Q is not empty do
    i := Dequeue(Q);
    for each j adjacent to i do
        if j is not marked then
            add {i,j} to T;
            Enqueue(j,Q) and mark j;
end{BFS}
```

Depth First vs Breadth First

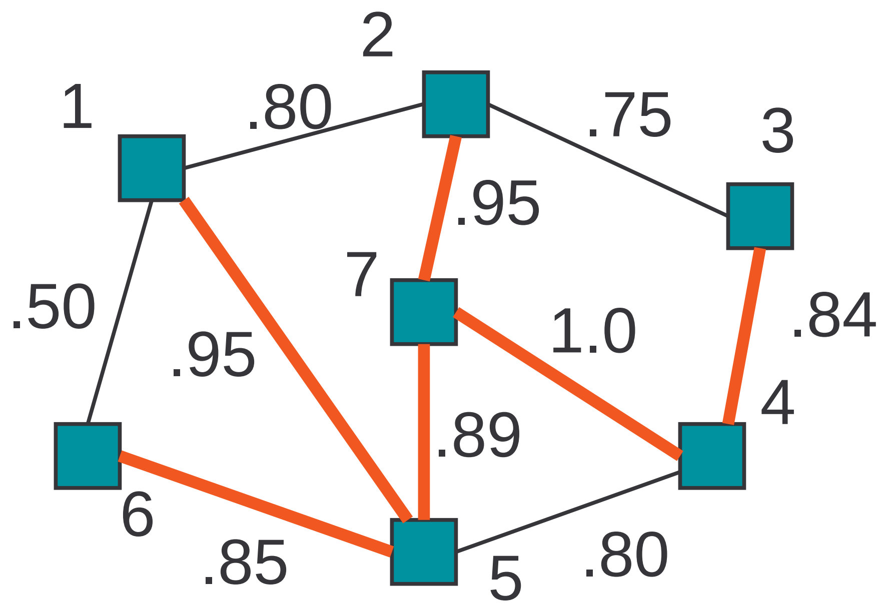
- Depth First
  - Stack or recursion
  - Many applications
- Breadth First
  - Queue (recursion no help)
  - Can be used to find shortest paths from the start vertex

Best Spanning Tree

- Each edge has the probability that it won't fail
- Find the spanning tree that is least likely to fail



Example of a Spanning Tree



Probability of success = .85 x .95 x .89 x .95 x 1.0 x .84
= .5735

Minimum Spanning Tree Problem

- Input: Undirected Graph G = (V,E) and a cost function C from E to the reals. C(e) is the cost of edge e.
- Output: A spanning tree T with minimum total cost. That is: T that minimizes

$$C(T) = \sum_{e \in T} C(e)$$

Reducing Best to Minimum

Let P(e) be the probability that an edge doesn't fail.
Define:

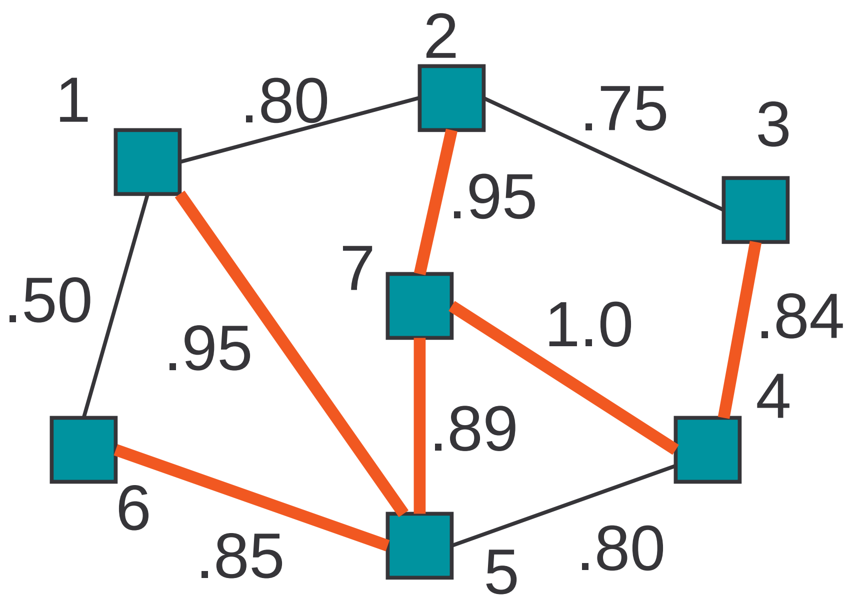
$$C(e) = -\log_{10}(P(e))$$

Minimizing $\sum_{e \in T} C(e)$

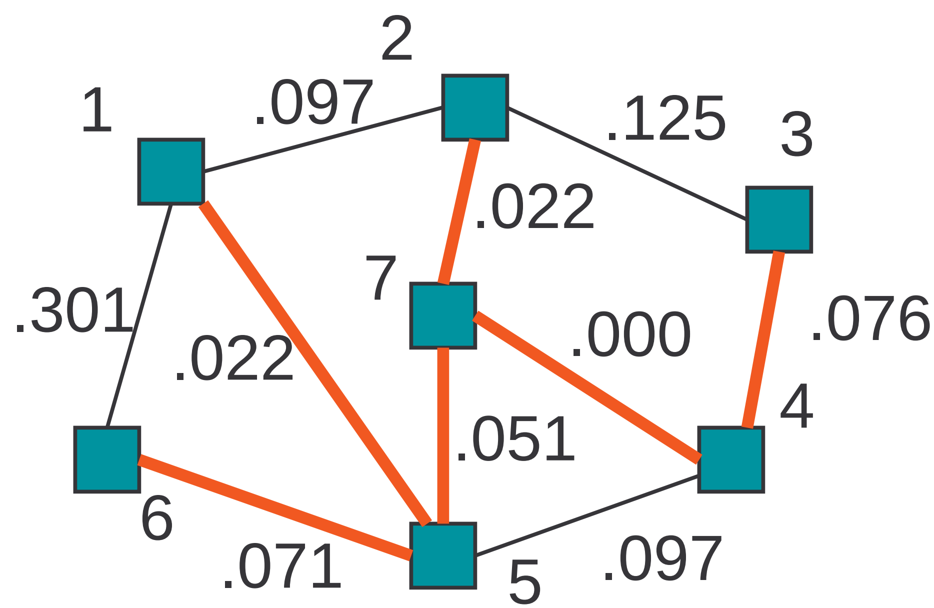
is equivalent to maximizing $\prod_{e \in T} P(e)$

because $\prod_{e \in T} P(e) = 10^{-\sum_{e \in T} C(e)}$

Example of Reduction



Best Spanning Tree Problem



Minimum Spanning Tree Problem

Minimum Spanning Tree

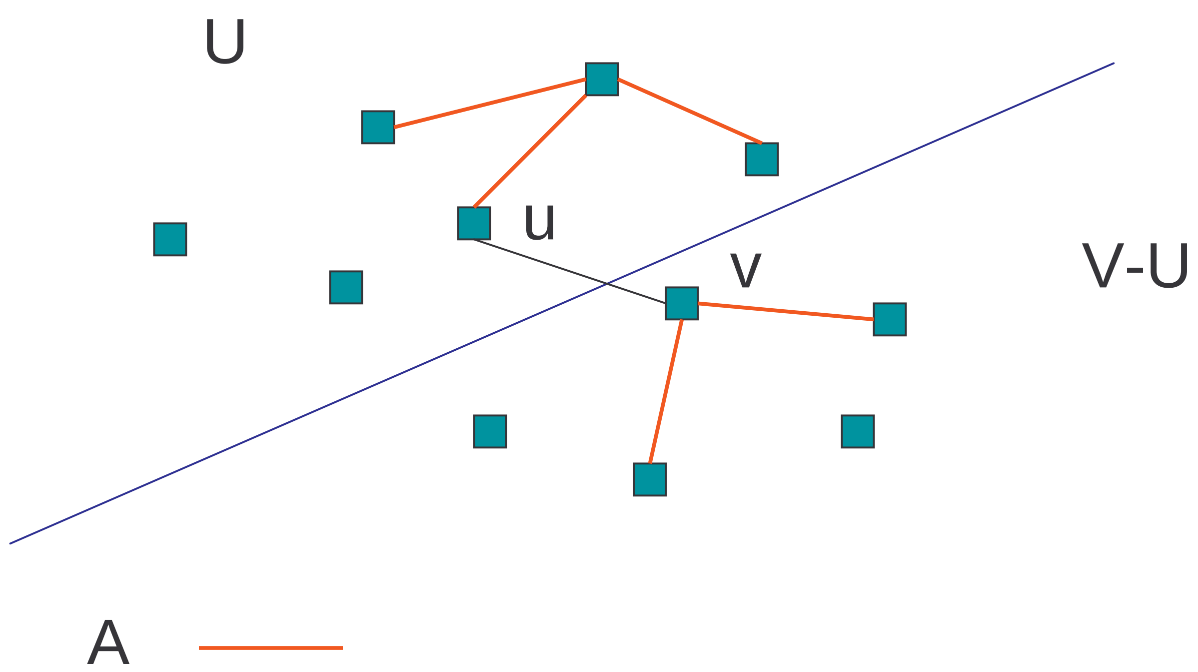
- Boruvka 1926
- Kruskal 1956
- Prim 1957 also by Jarnik 1930
- Karger, Klein, Tarjan 1995
  - Randomized linear time algorithm
  - Probably not practical, but very interesting

MST Optimality Principle

- G = (V,E) with costs C. G connected.
- Let (V,A) be a subgraph of G that is contained in a minimum spanning tree. Let U be a set such that no edge in A has one end in U and one end in V-U. Let C({u,v}) minimal and u in U and v in V-U. Let A’ be A with {u,v} added. Then (V,A’) is contained in a minimum spanning tree.

Proof of Optimality Principle

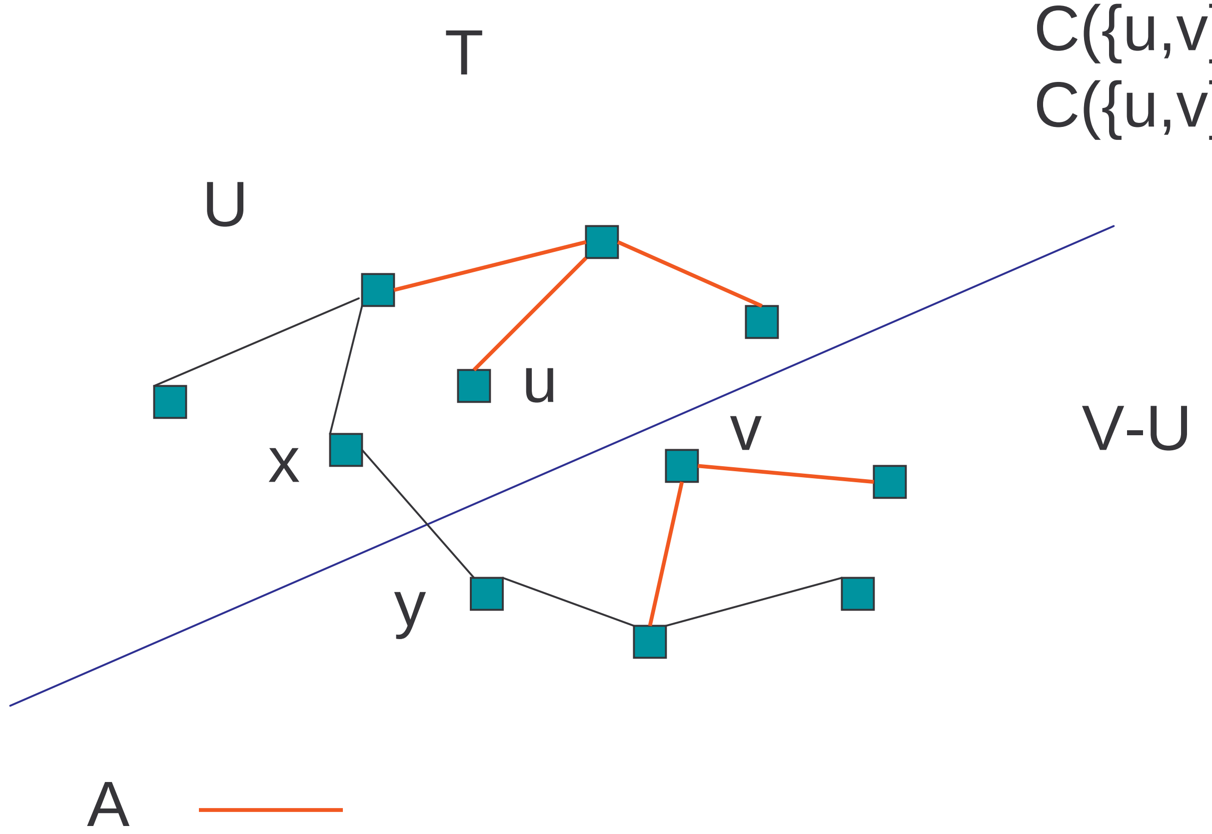
C({u,v}) is minimal



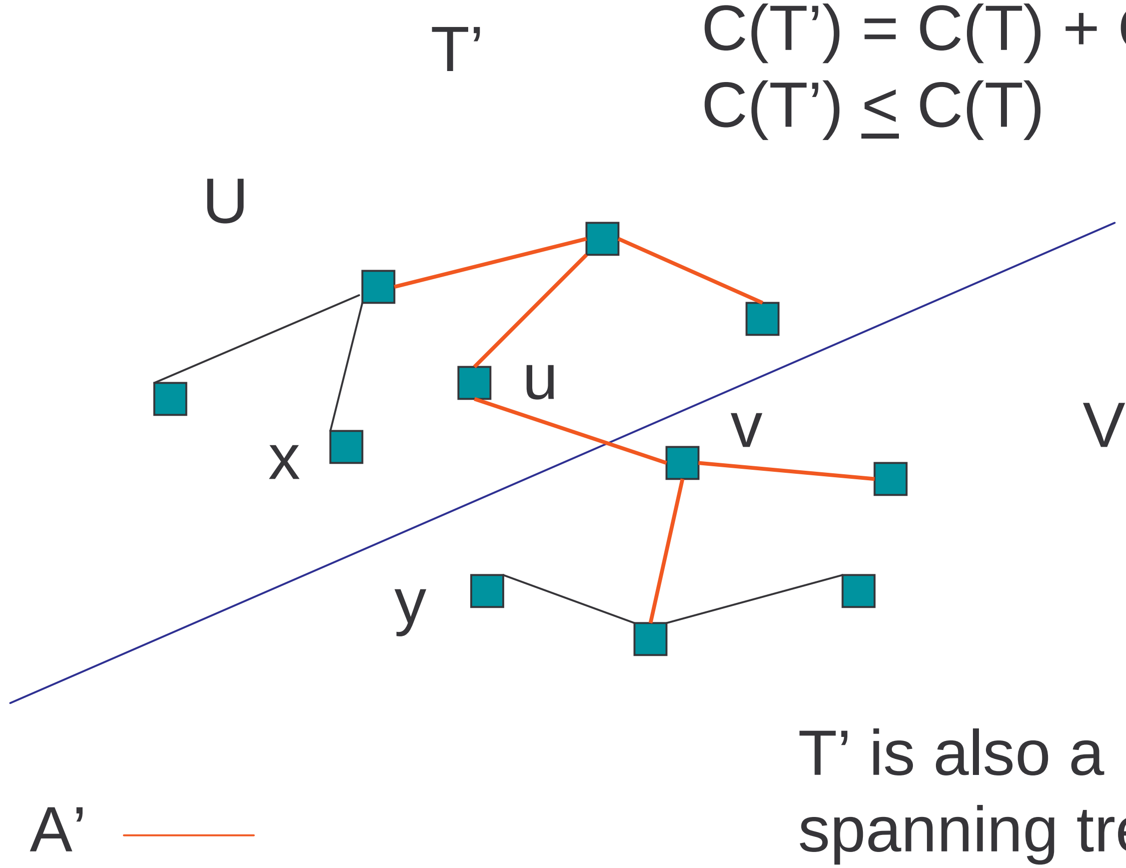
Proof of Optimality Principle

C({u,v}} is minimal

C({u,v}) $\leq$ C({x,y})



Proof of Optimality Principle



C(T') = C(T) + C({u,v}) - C({x,y})
C(T') $\leq$ C(T)

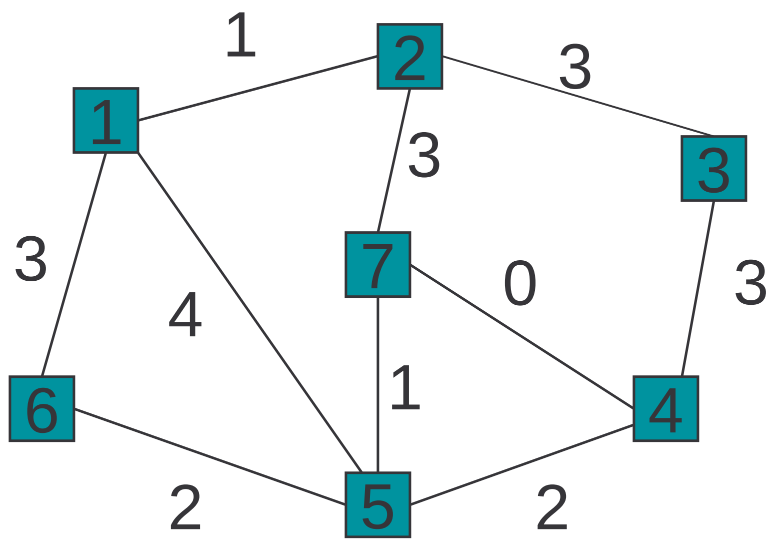
T' is also a minimum spanning tree

Kruskal’s Greedy Algorithm

```
Sort the edges by increasing cost;
Initialize A to be empty;
For each edge e chosen in increasing order do
    if adding e does not form a cycle then
        add e to A
```

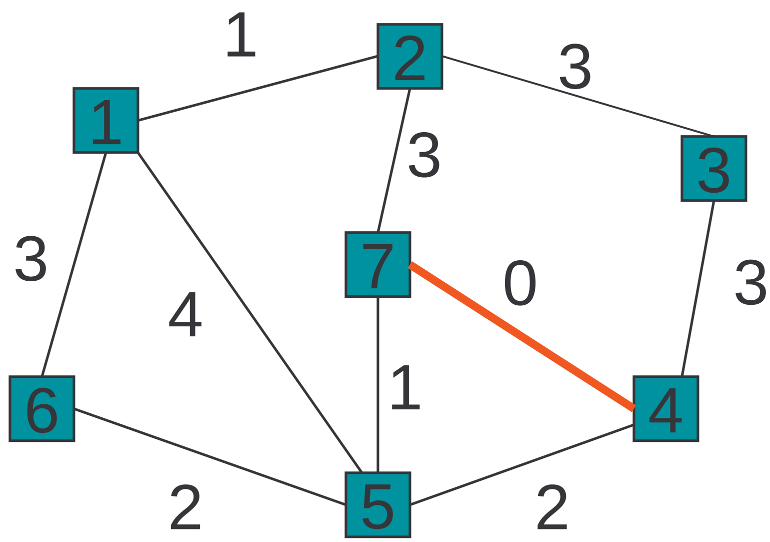
Invariant: A is always contained in some minimum spanning tree

Example of Kruskal 1



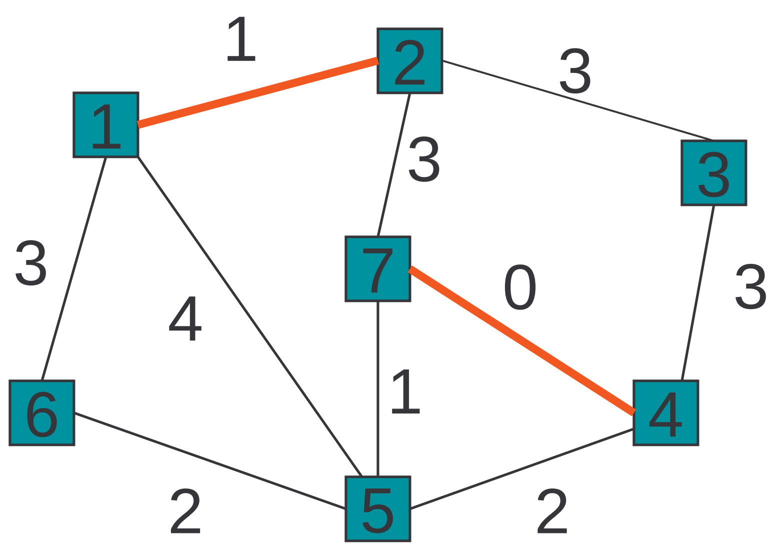
{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

Example of Kruskal 2



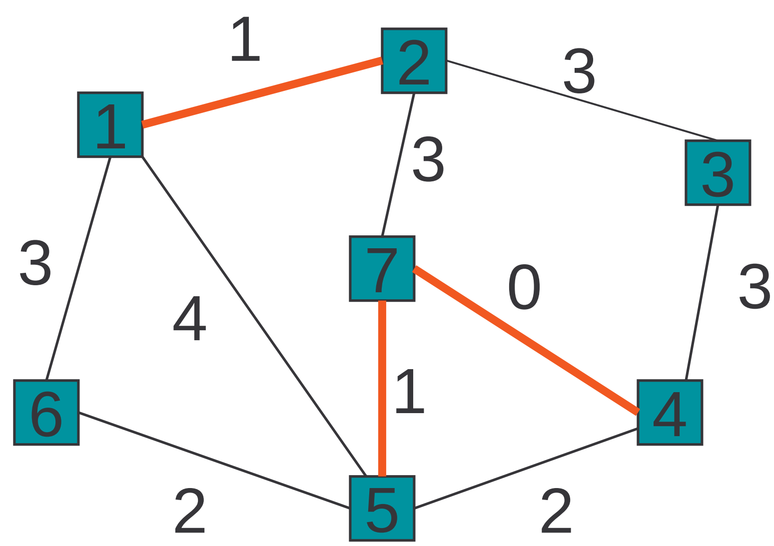
~~{7,4}~~	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	1	1	2	2	3	3	3	3	4

Example of Kruskal 2



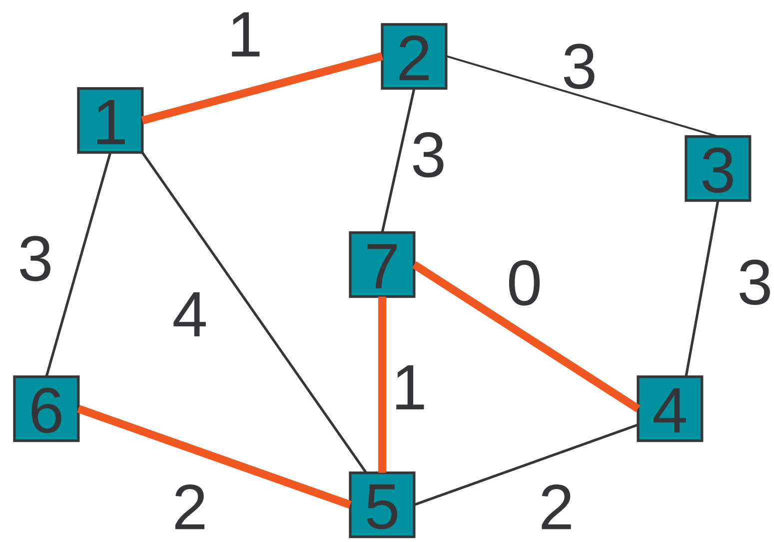
~~{7,4}~~	~~{2,1}~~	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	1	2	2	3	3	3	3	4

Example of Kruskal 3



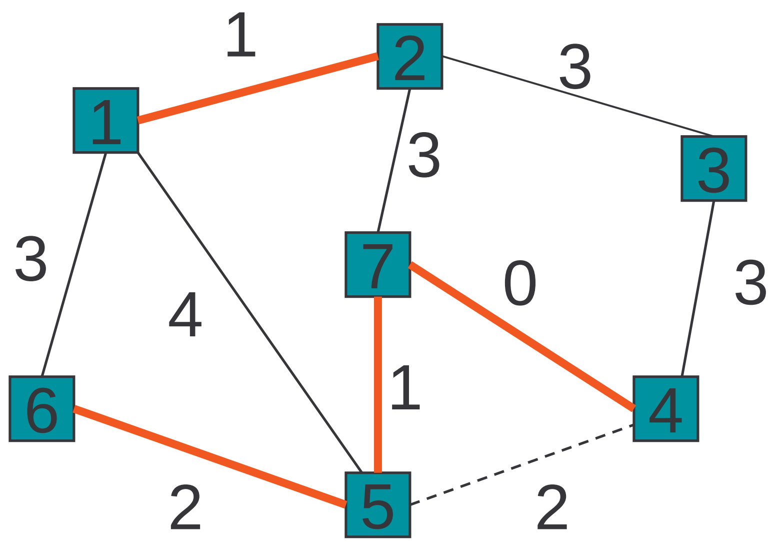
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	2	2	3	3	3	3	4

Example of Kruskal 4



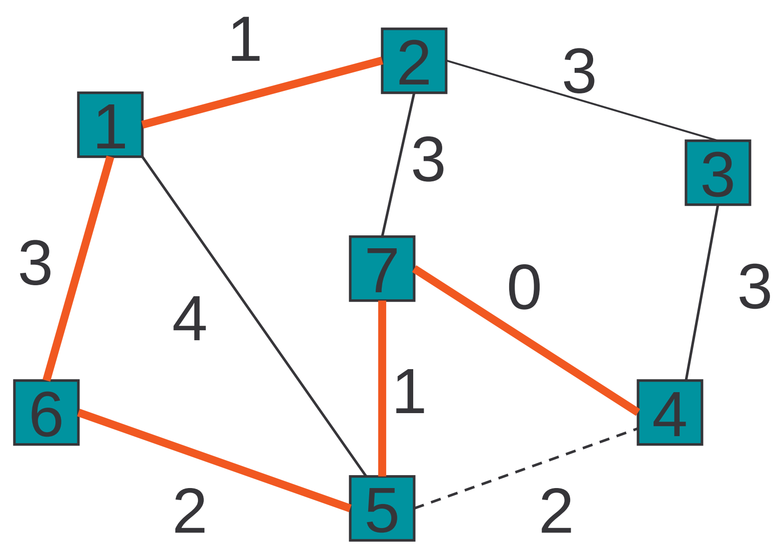
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	2	3	3	3	3	4

Example of Kruskal 5



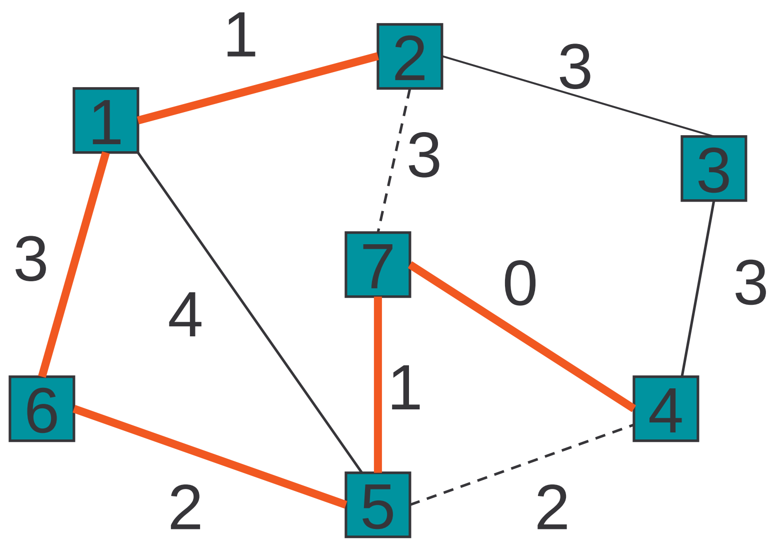
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	3	3	3	3	4

Example of Kruskal 6



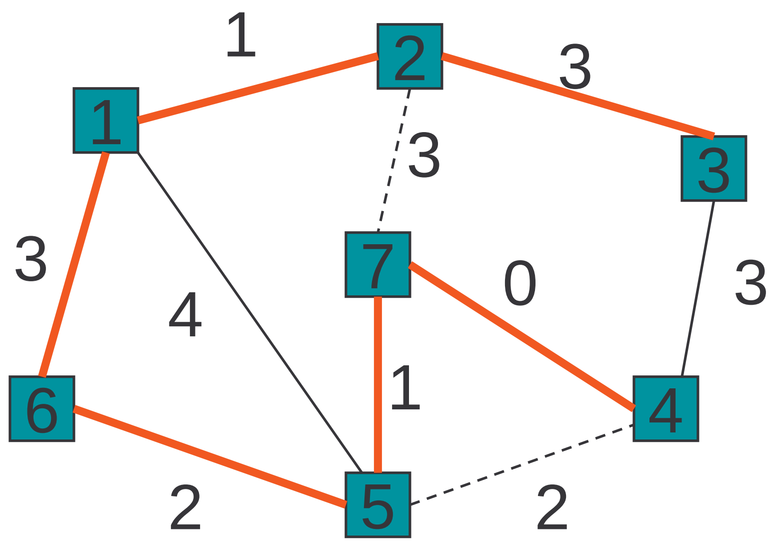
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	3	3	3	4

Example of Kruskal 7



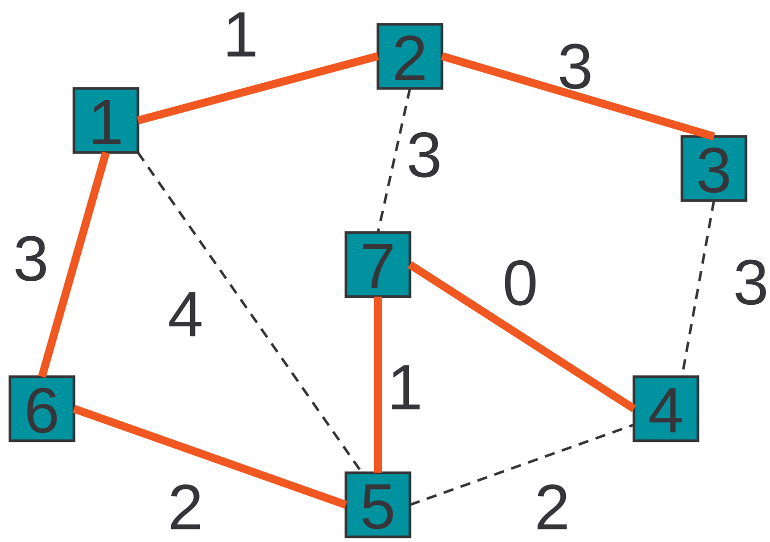
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	~~{2,7}~~	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	3	3	4

Example of Kruskal 7



~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	~~{2,7}~~	~~{2,3}~~	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	~~3~~	3	4

Example of Kruskal 8,9



~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	~~{2,7}~~	~~{2,3}~~	~~{3,4}~~	~~{1,5}~~
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	~~3~~	~~3~~	~~4~~

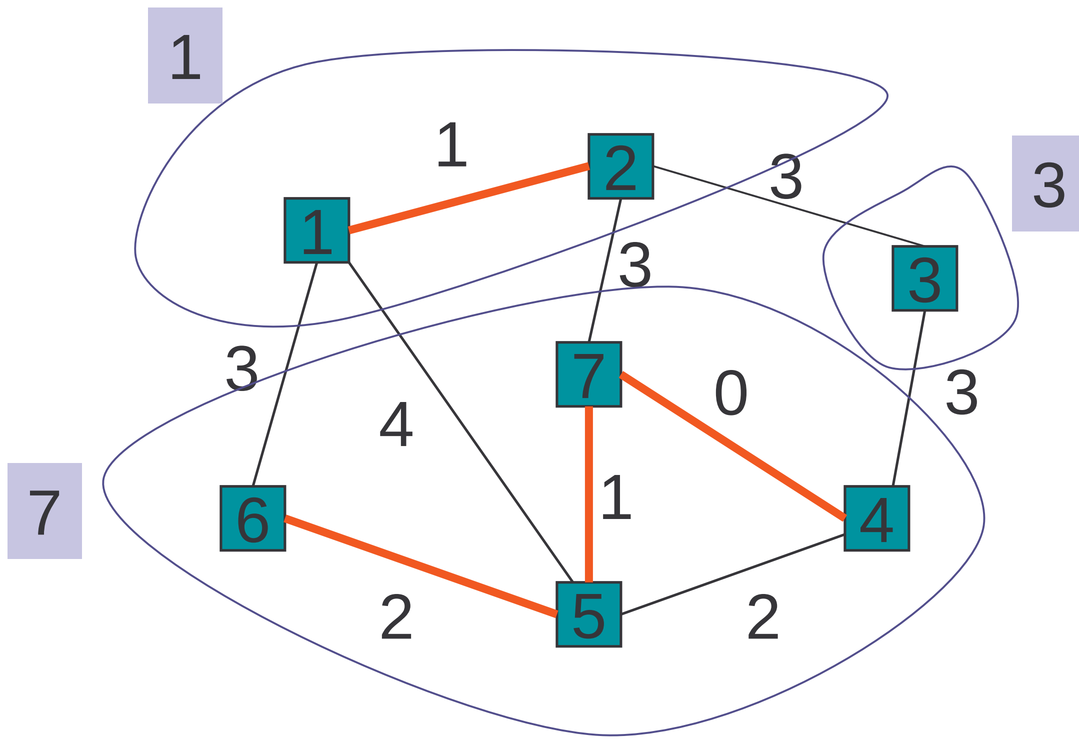
Data Structures for Kruskal

- Sorted edge list

{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

- Disjoint Union / Find
  - Union(a,b) - union the disjoint sets named by a and b
  - Find(a) returns the name of the set containing a

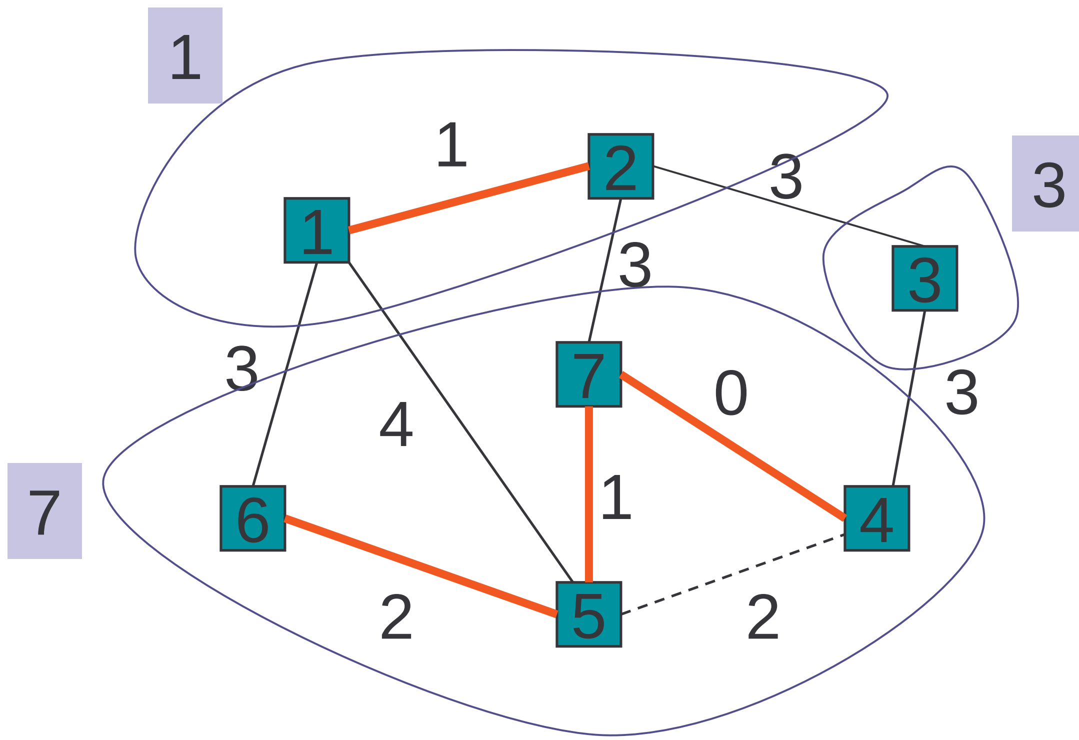
Example of DU/F 1



Find(5) = 7
Find(4) = 7

~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	2	3	3	3	3	4

Example of DU/F 2

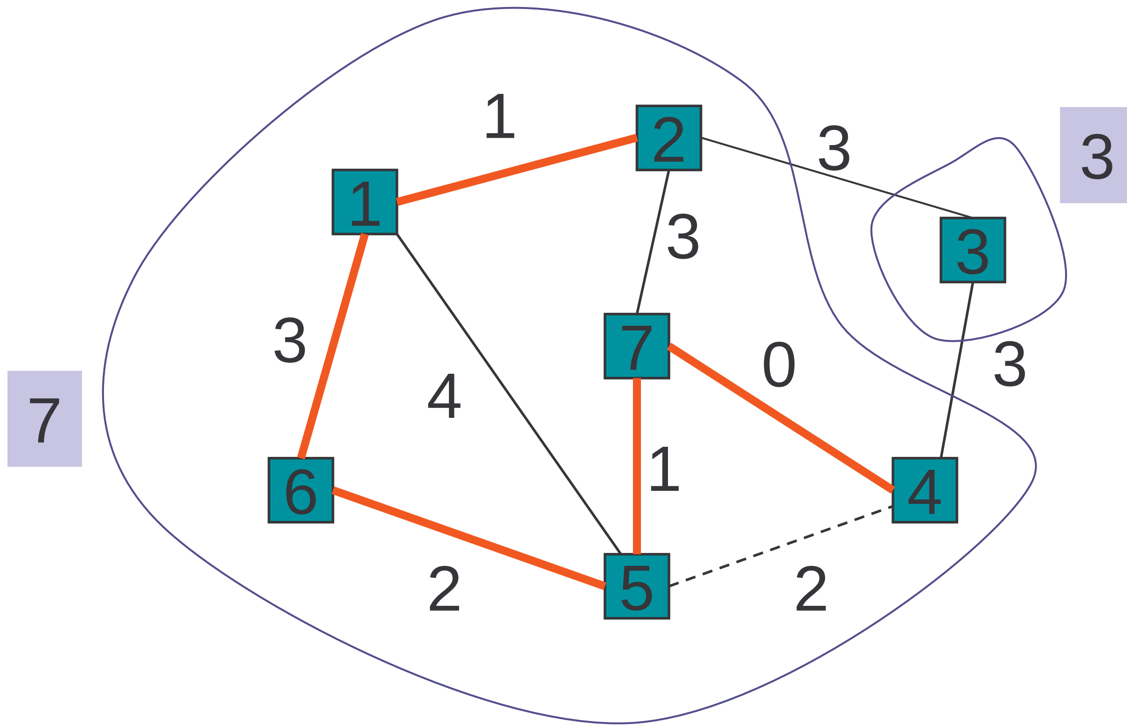


Find(1) = 1
Find(6) = 7

~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	3	3	3	3	4

Example of DU/F 3

Union(1,7)



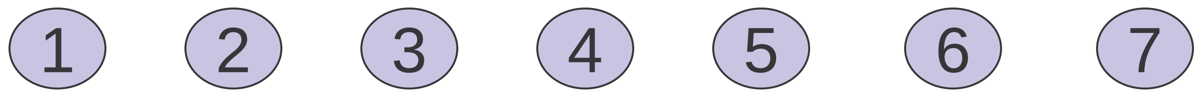
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	3	3	3	4

Kruskal’s Algorithm with DU / F

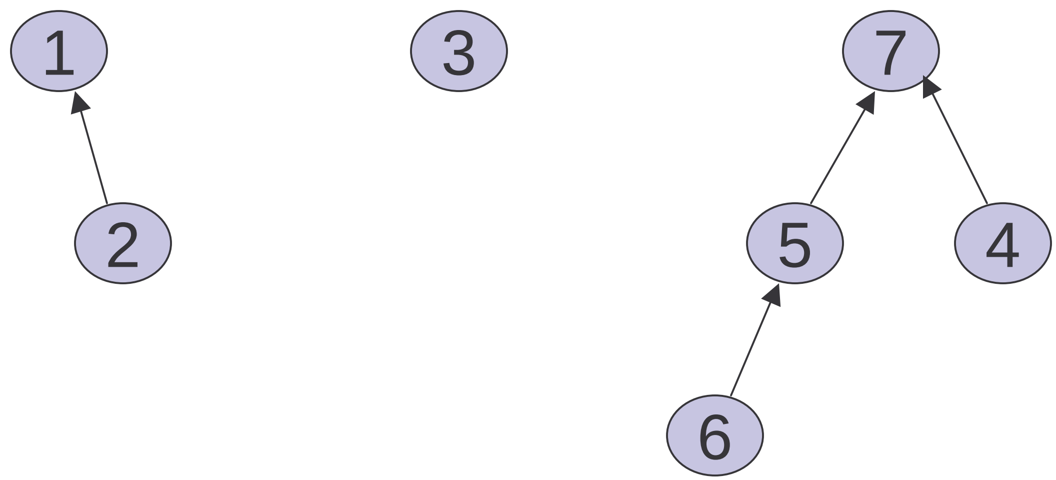
```
Sort the edges by increasing cost;
Initialize A to be empty;
for each edge {i,j} chosen in increasing order do
    u := Find(i);
    v := Find(j);
    if not(u = v) then
        add {i,j} to A;
        Union(u,v);
```

Up Tree for DU/F

Initial state

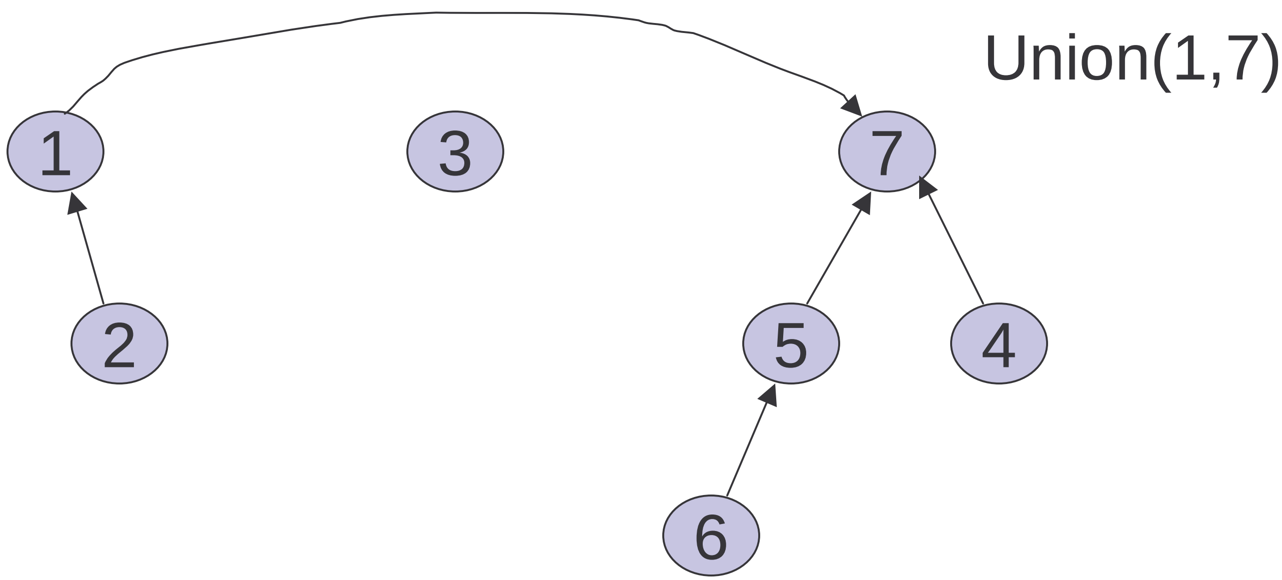


Intermediate state



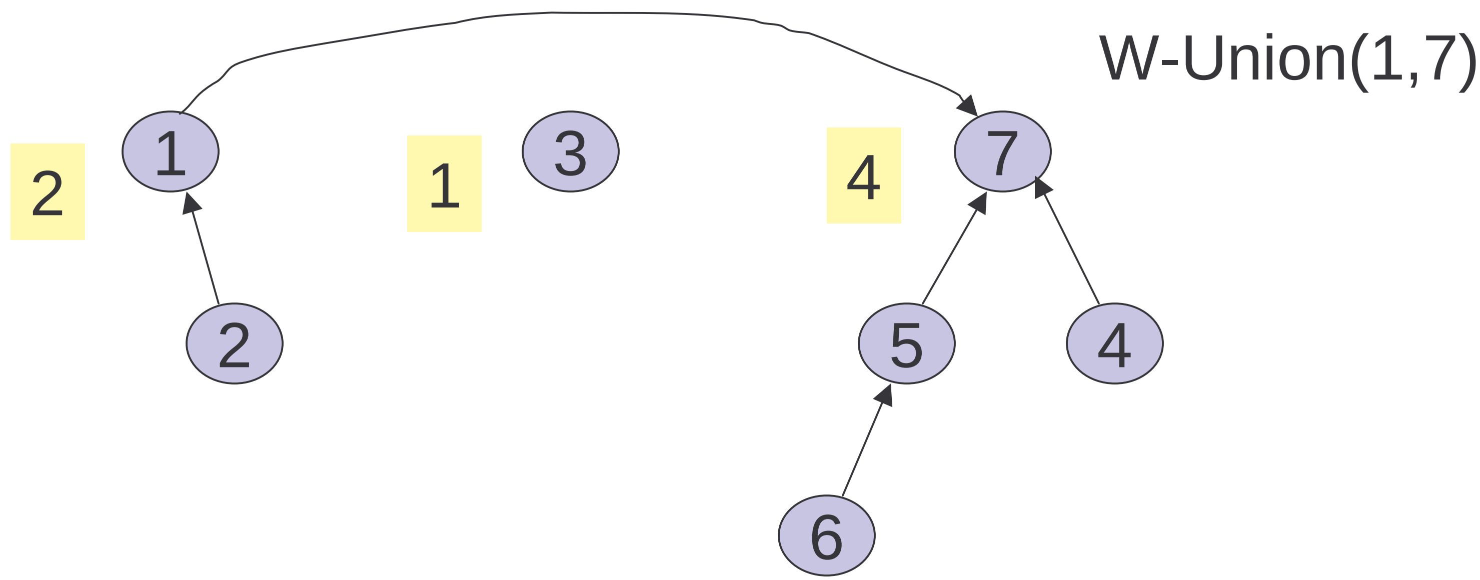
DU/F Operation

- Find(i) - follow pointer to root and return the root.
- Union(i,j) - assuming i and j roots, point i to j.



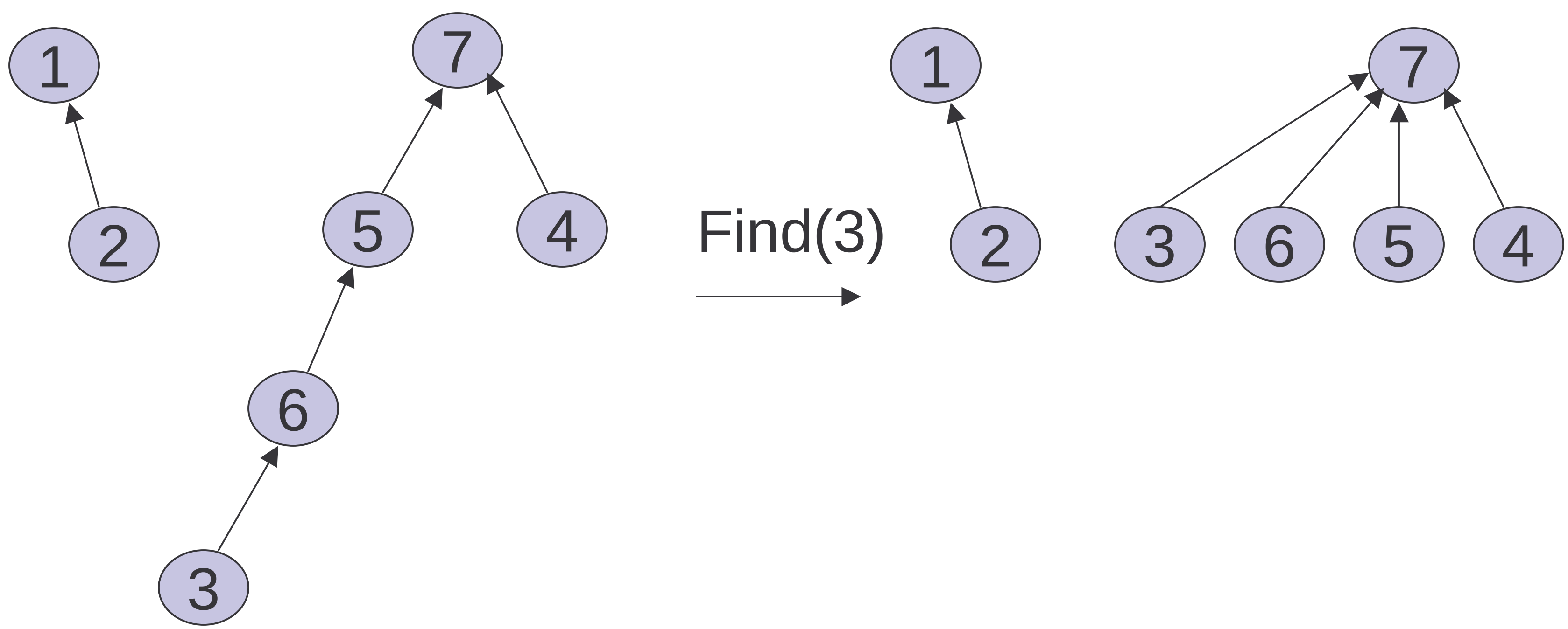
Weighted Union

- Weighted Union
  - Always point the smaller tree to the root of the larger tree

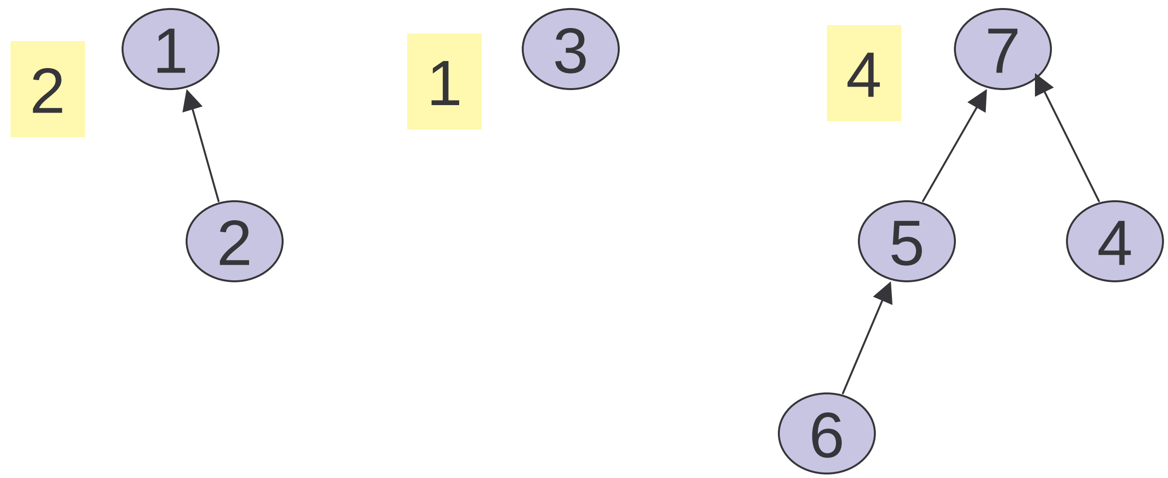


Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



Elegant Array Implementation



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Up Tree Pseudo-Code

```
PC-Find(i : index)
    r := i;
    while not(up[r] = 0) do
        r := up[r]
    k := up[i];
    while not(k = r) do
        up[i] := r;
        i := k;
        k := up[k]
    return(r)
end{Find}
```

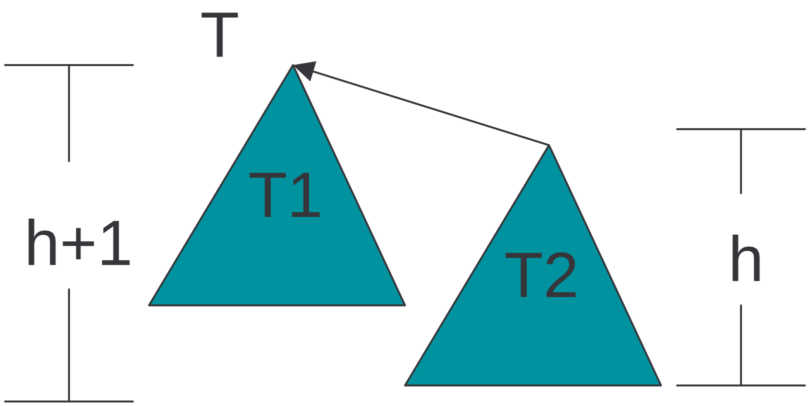
```
W-Union(i,j : index)
    // i and j are roots
    wi := weight[i];
    wj := weight[j];
    if wi < wj then
        up[i] := j;
        weight[j] := wi + wj;
    else
        up[j] :=i;
        weight[i] := wi +wj;
end{W-Union}
```

Disjoint Union / Find Notes

- Worst case time complexity for a W-Union is O(1) and for a PC-Find is O(log n).
- Time complexity for m operations on n elements is O(m log* n) where log* n is a very slow growing function. Essentially constant time per operation!
- Using "ranked union" gives an even better bound theoretically.

Performance of W-Union / PC-Find

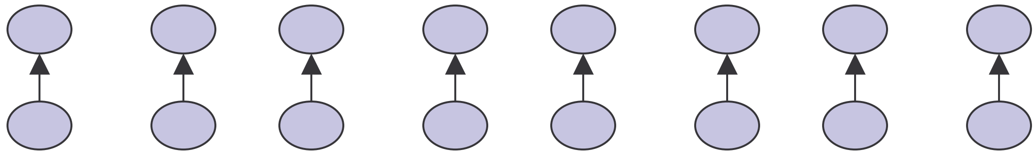
- The time complexity of PC-Find is O(log n).
- An up tree formed by W-Union of height h has at least 2^h nodes. Inductive Proof.



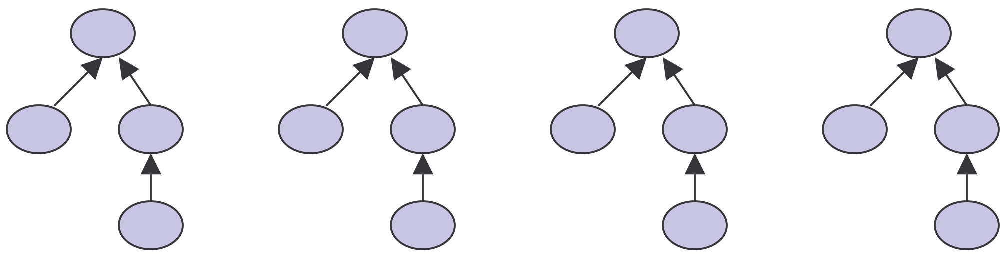
$$
\begin{aligned}
\text{Weight}(T2) &\geq 2^h \text{ (ind. hyp.)} \\
\text{Weight}(T1) &\geq \text{Weight}(T2) \\
&\geq 2^h \\
\text{Weight}(T) &\geq 2^h + 2^h = 2^{h+1}
\end{aligned}
$$

Worst Case for PC-Find

n/2 Weighted Unions

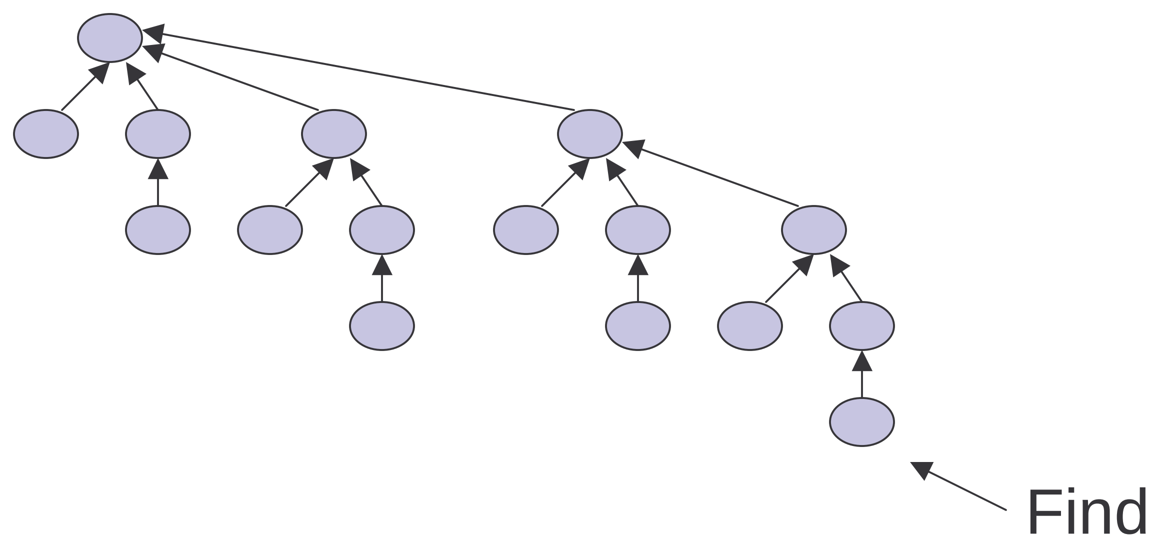


n/4 Weighted Unions



Example of Worst Cast (cont')

After n -1 = n/2 + n/4 + …+ 1 Weighted Unions



If there are $n = 2^k$ nodes then there are k pointers on the longest path to root.

Amortized Complexity

- For disjoint union / find with weighted union and path compression.
  - average time per operation is essentially a constant.
  - worst case time for a PC-Find is O(log n).
- An individual operation can be costly, but over time the average cost per operation is not.