Improving the Held and Karp Approach with Constraint Programming

Jean-Charles RÉGIN
Université de Nice-Sophia Antipolis
jcregin@gmail.com

Joint work with L-M. Rousseau; M. Rueher; W-J van Hoeve

Outline

- Background
  - Traveling Salesman Problem
  - Constraint Programming
  - Minimum Spanning Tree
  - Union-find
- Held and Karp Bound for TSP
- Weighted Spanning Tree Constraint
  - Removing inconsistent edges
  - Handling mandatory edges
  - Maintaining mandatory edges
- Other filtering algorithms
- Implementation and experimental results
- Discussion and Conclusion

Traveling Salesman Problem

- Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

- Given a complete weighted graph, find a Hamiltonian cycle with the least weight

TSP

Each city is visited exactly once

Only one tour (no subtour)

TSP

Each city is visited exactly once

Only one tour (no subtour)

TSP

- Scheduling problem: find the order in which you have to build objects
- Pure TSP does not arise frequently in practice. More often:
  - Non euclidian
  - Asymetrical
  - Covering of a subset of nodes
- Common problems
  - Vehicle routing (time windows, pickup and delivery…)

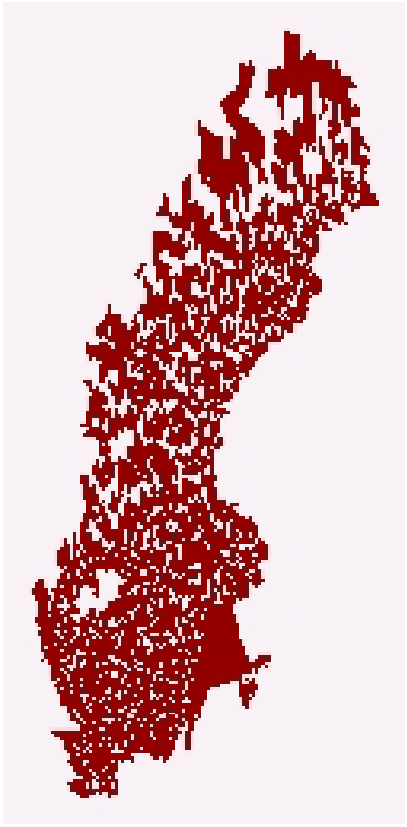
Procter and Gamble Contest 1962

USA 13,509 cities. Solved in 1998

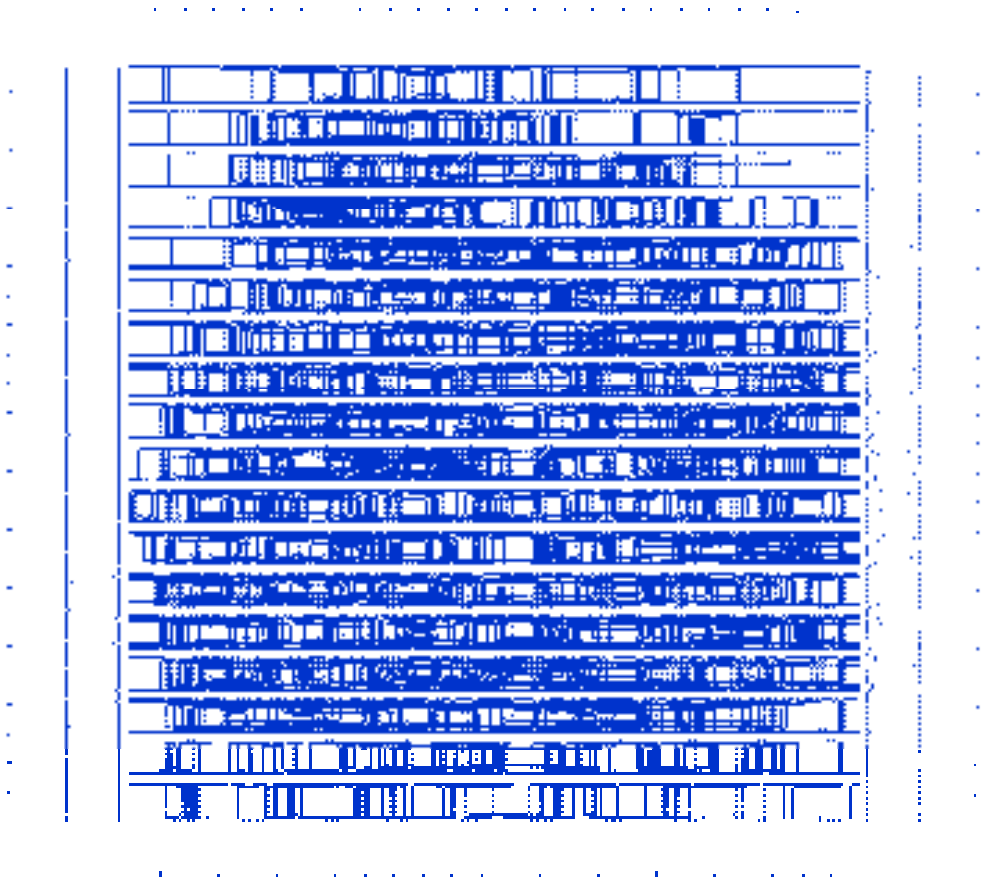
German Tour

- On April 20th, 2001, David Applegate, Robert Bixby, Vašek Chvátal, and William Cook announced the solution of a traveling salesman problem through 15,112 cities in Germany.
- Network of 110 processors located at Rice University and at Princeton University.
- Total computer time used in the computation was **22.6 years**

Germany 15,112 cities. Solved in 2001



Sweden 24,978 cities. Solved in 2004



VLSI 85,900. Solved in 2006

TSP

- There exist several solvers
- The most well known is Concorde TSP solver by William Cook (freely available)
- IBM ILOG CPLEX's reputation is partly based on the resolution of TSPs

TSP Solvers

- TSP Solvers are generally dedicated to the pure problem
- Almost impossible to use them when the problem is slightly change (asymetrical, side constraints …)

- **My goal : solve the problem with CP**
  - Promote CP in OR espacially for an optimization problem (not only combinatorial)
  - Instead of « 22,6 years with 110 processor (550Mhz) »
  - I would like to have « solve on my laptop »

Constraint Programming

Constraint Programming

- In CP a problem is defined from:
  - variables with possible values (domain)
  - constraints
- Domain can be discrete or continuous, symbolic values or numerical values
- Constraints express properties that have to be satisfied

Problem = conjunction of sub-problems

- In CP a problem can be viewed as a conjunction of sub-problems that we are able to solve
- A sub-problem can be trivial: $x < y$ or complex: search for a feasible flow
- A sub-problem = a constraint

Constraints

- Predefined constraints: arithmetic ($x < y$, $x = y + z$, $|x-y| > k$, alldiff, cardinality, sequence …
- Constraints given in extension by the list of allowed (or forbidden) combinations of values
- User-defined constraints: any algorithm can be encapsulated
- Logical combination of constraints using OR, AND, NOT, XOR operators. Sometimes called meta-constraints

Filtering

- We are able to solve a sub-problem: a method is available
- CP uses this method to remove values from domain that do not belong to a solution of this sub-problem: **filtering or domain-reduction**
- E.g: $x < y$ and $D(x)=[10,20]$, $D(y)=[5,15]$
  $\Rightarrow D(x)=[10,14]$, $D(y)=[11,15]$

Filtering

- A filtering algorithm is associated with each constraint (sub-problem).
- Can be simple ($x < y$) or complex (alldiff)
- Theoretical basics: **arc consistency**, remove all the values that do not belong to a solution of the underlined sub-problem.

Arc consistency

- **All the values which do not belong to any solution of the constraint are deleted.**
- Example: Alldiff({x,y,z}) with
  D(x)=D(y)={0,1}, D(z)={0,1,2}
  the two variables x and y take the values 0 and 1,
  thus z cannot take these values.
  FA by AC => 0 and 1 are removed from D(z)

Filtering

- Goal: answer to the question

  « how to remove as quickly as possible values that do not belong to a solution of a problem ? »

Propagation

- Domain Reduction due to one constraint can lead to new domain reduction of other variables
- When a domain is modified all the constraints involving this variable are studied and so on ...

Propagation

- D(x)=D(y)={1,3}, D(z)=D(t)=D(u){0,1,2,3,4}
- Alldiff({x,y,z,t}), z < t, u=z+t, y=u+1
- z < t $\Rightarrow$ z $\neq$ 4 and t $\neq$ 0
- Alldiff(x,y,z,t) $\Rightarrow$ z $\neq$ {1,3} and t $\neq$ {1,3}
- u=z+t $\Rightarrow$ {0,2} + {2,4} $\geq$2 so u $\geq$2
- y=u+1 $\Rightarrow$ u=y-1={1,3}-1 $\leq$ 2 so u=2 and y=3
- Alldiff(x,y,z,t) $\Rightarrow$ x=1
- u=z+t $\Rightarrow$ 2=z+t $\Rightarrow$ z=0; t=2

Why Propagation?

- A problem = conjunction of easy sub-problems.
- Sub-problems: local point of view. Propagation tries to obtain a global point of view from independent local point of view
- The conjunction is stronger that the union of independent resolution

Search

- Backtrack algorithm with strategies:
  try to successively assign variables with values. If a dead-end occurs then backtrack and try another value for the variable
- Strategy: define which variable and which value will be chosen.
- After each domain reduction (I.e assignement include) filtering and propagation are triggered

Constraint Programming

- 3 notions:
  - constraint network: variables, domains constraints + filtering (domain reduction)
  - propagation
  - search procedure (assignments + backtrack)

- Reformulation of the famous Kowalski's definition of Algorithm:
  - Algorithm = Logic + Control
  - **CP = Filtering + Propagation + Search**
    - Filtering and Propagation = Logic
    - Search Control.

TSP and CP

Motivation

- Held and Karp proposed (in the early 1970s) a relaxation for the Traveling Salesman Problem (TSP).

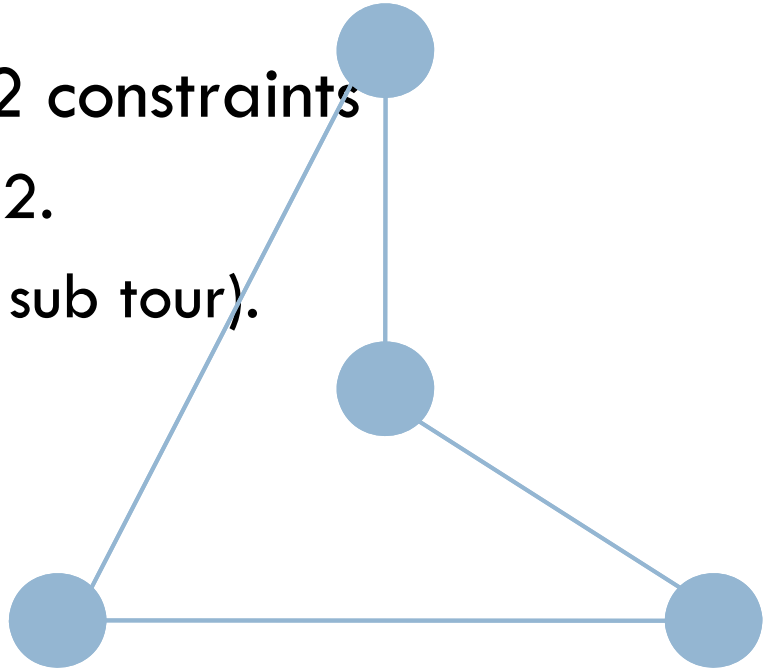
- It has been shown that this relaxation produces very tight bounds in practice and is therefore applied in the best TSP solvers such as Concorde or LKH.

We will show that the Held-Karp approach can benefit from CP techniques in such as domain filtering.

The Held and Karp bound for TSP

- The travelling salesman problem consist in finding the minimal cost Hamiltonian cycle on Graph G=(V,E) which contains n nodes and m edges.

- Basically a combination of 2 constraints
  - The degree of each node is 2.
  - The solution is connected (no sub tour).



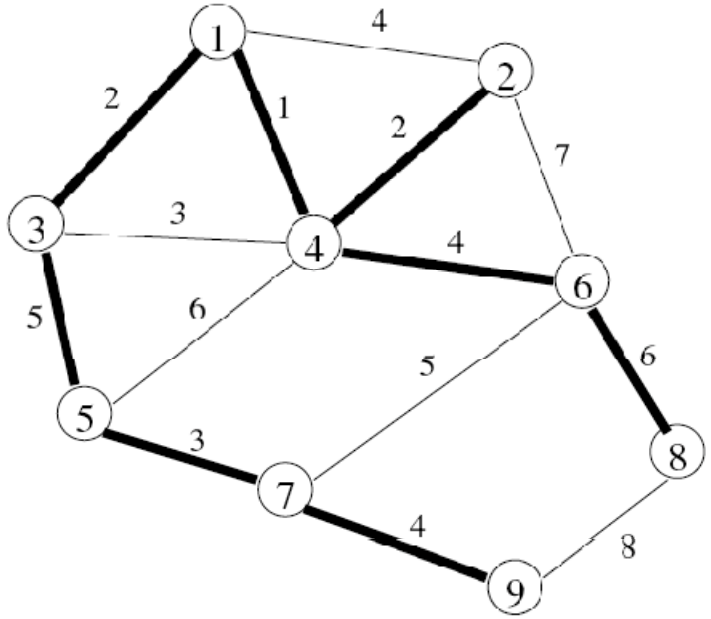
The Held and Karp bound for TSP

- Held and Karp proposed a lower bound based on a relaxation of the degree constraints.
- A minimum spanning tree gives such a relaxation.

Minimum Spanning Tree and Union-Find

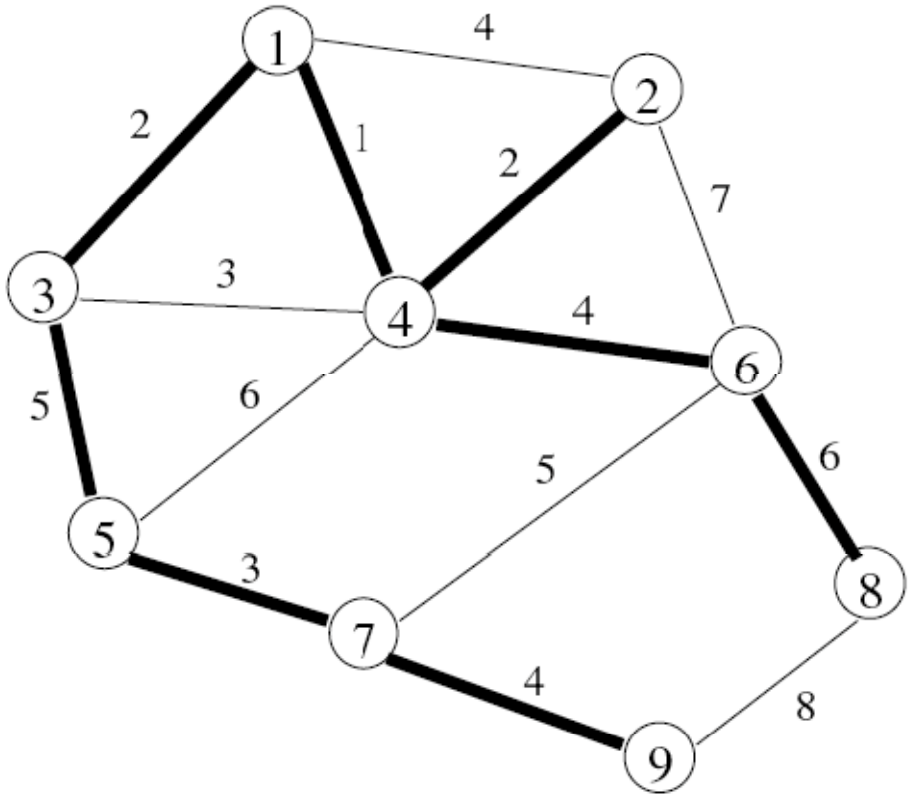
Minimum Spanning Tree

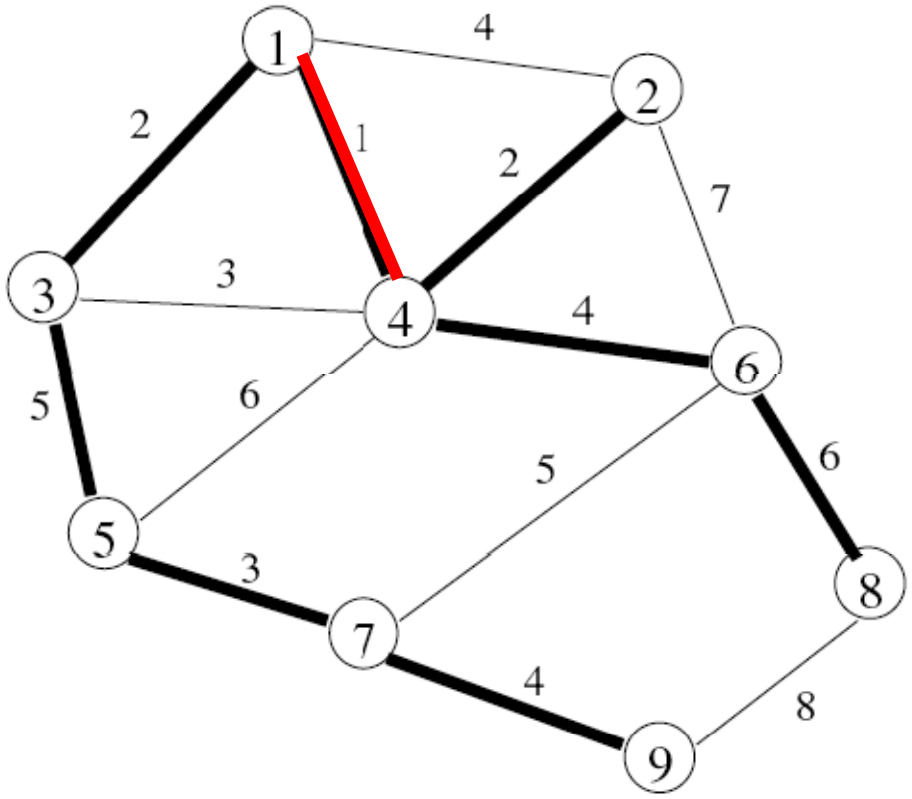
- Definition:
  - Tree: connected acyclic graph
  - T=(X',E') is a spanning tree of G=(X,E) if X'=X and E' $\subseteq$ E
  - Cost of a spanning tree = sum of the edge costs.
- Algorithms: Almost Linear
  if edges are sorted.
  2 are common
  - Prim
  - Kruskal

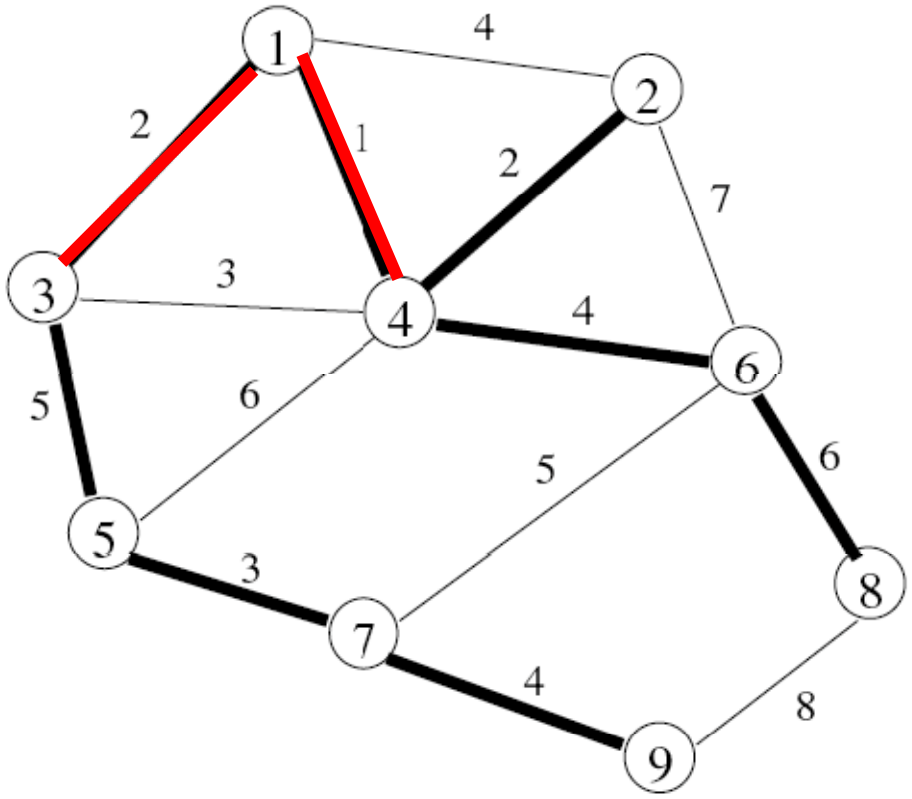


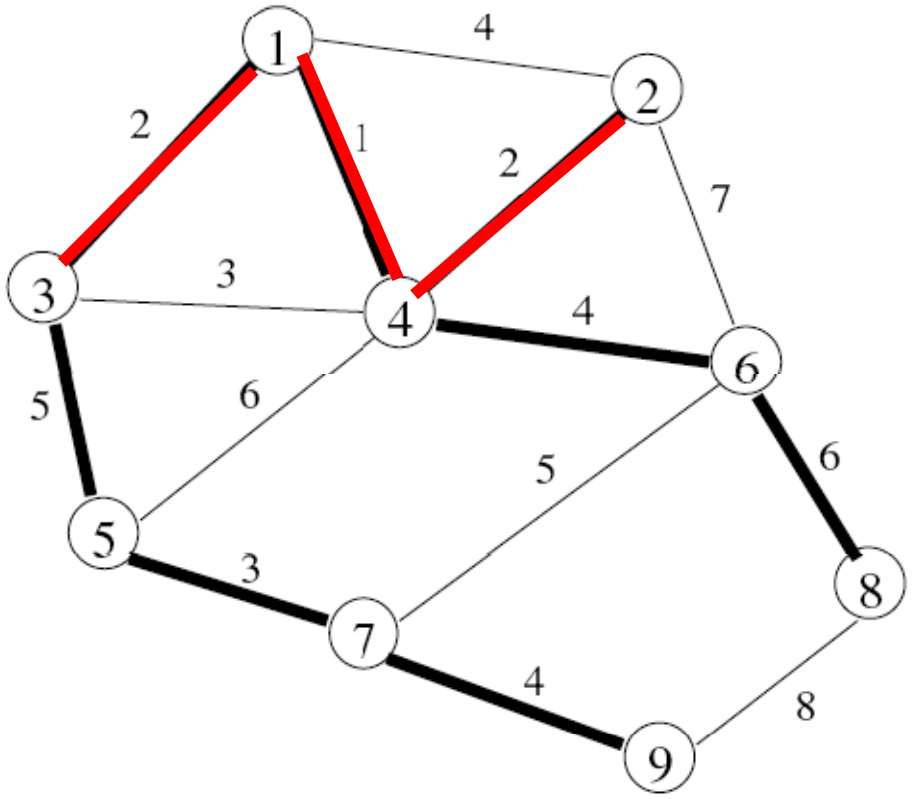
Kruskal's algorithm

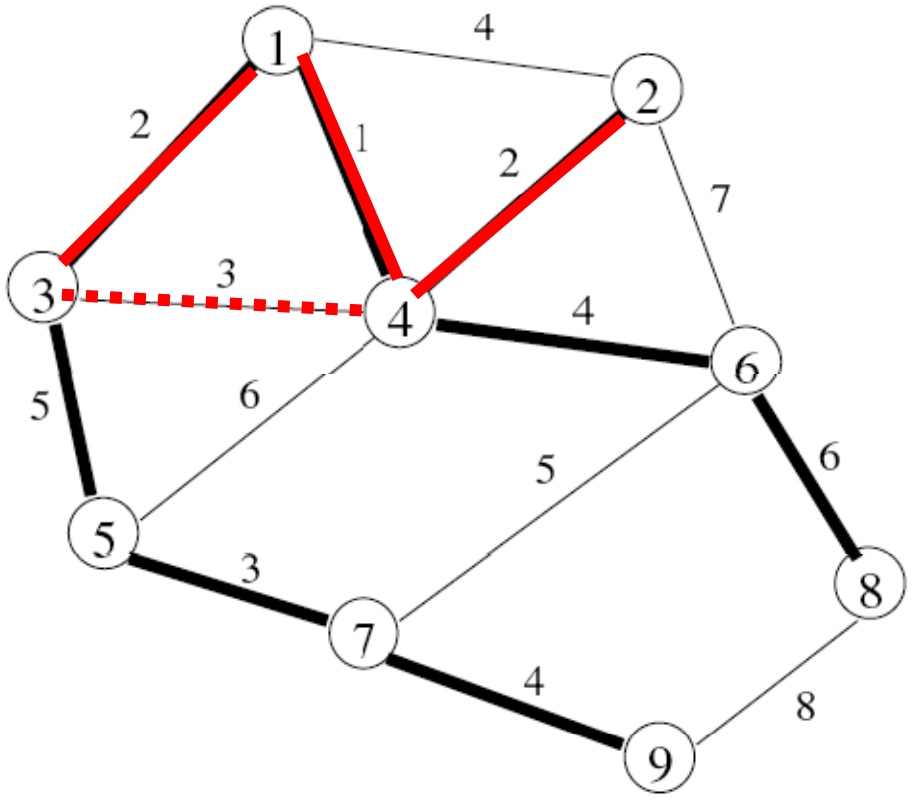
- M = set of edges ordered by non decreasing cost.
- Consider the graph without any edge
- For each {u,v} in M:
  - if u and v are not in the same connected component :
    - Add the edge {u,v}
    - Merge the connected components
  - Stop when a unique connected component is obtained

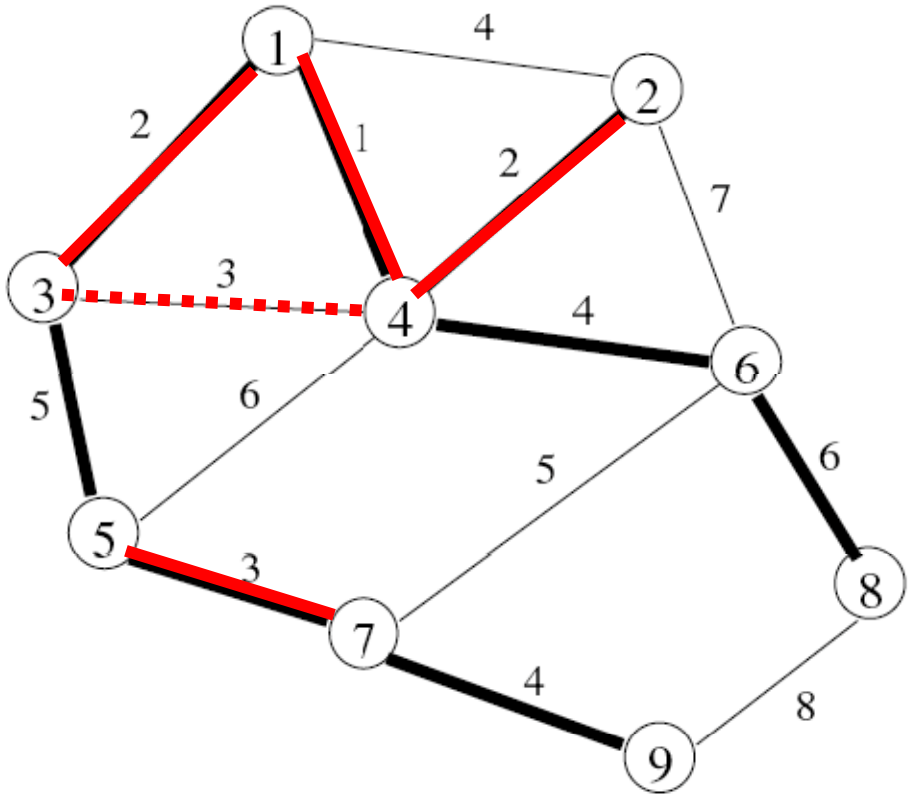

1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
3
4
5
6
6
5
3
4
8

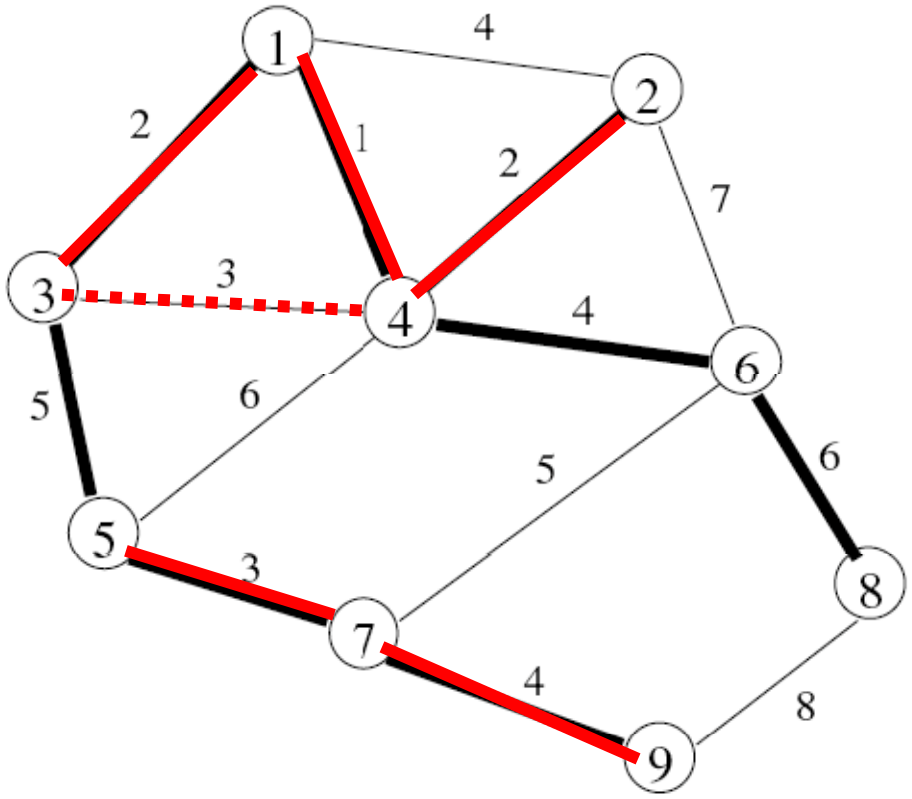

1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
5
6
3
4
8

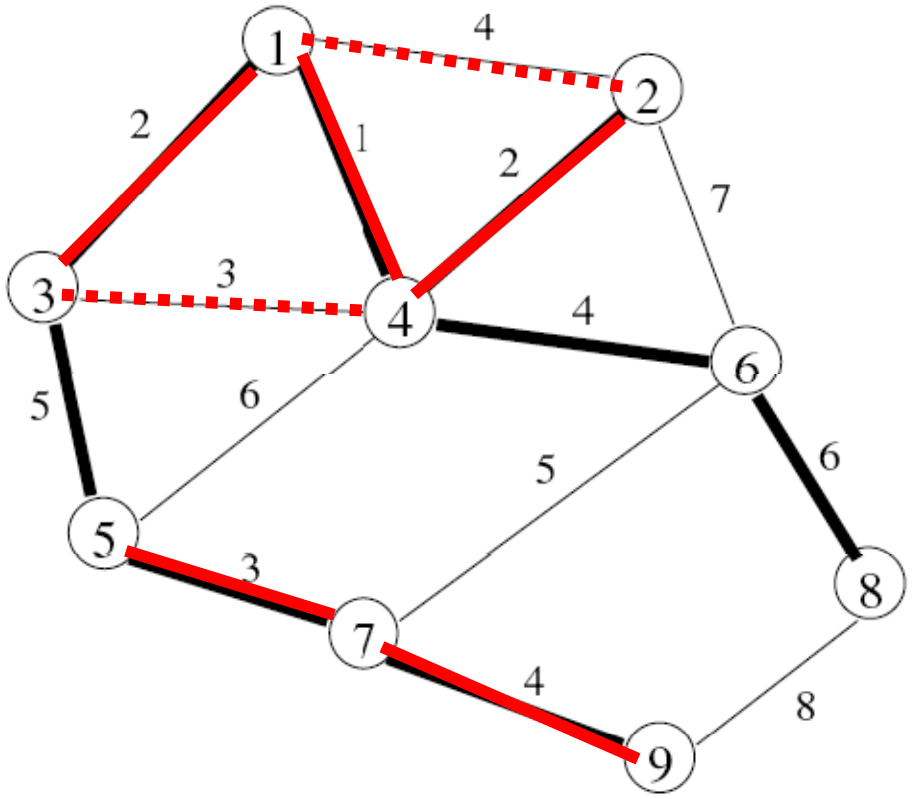

1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
6
5
3
4
8

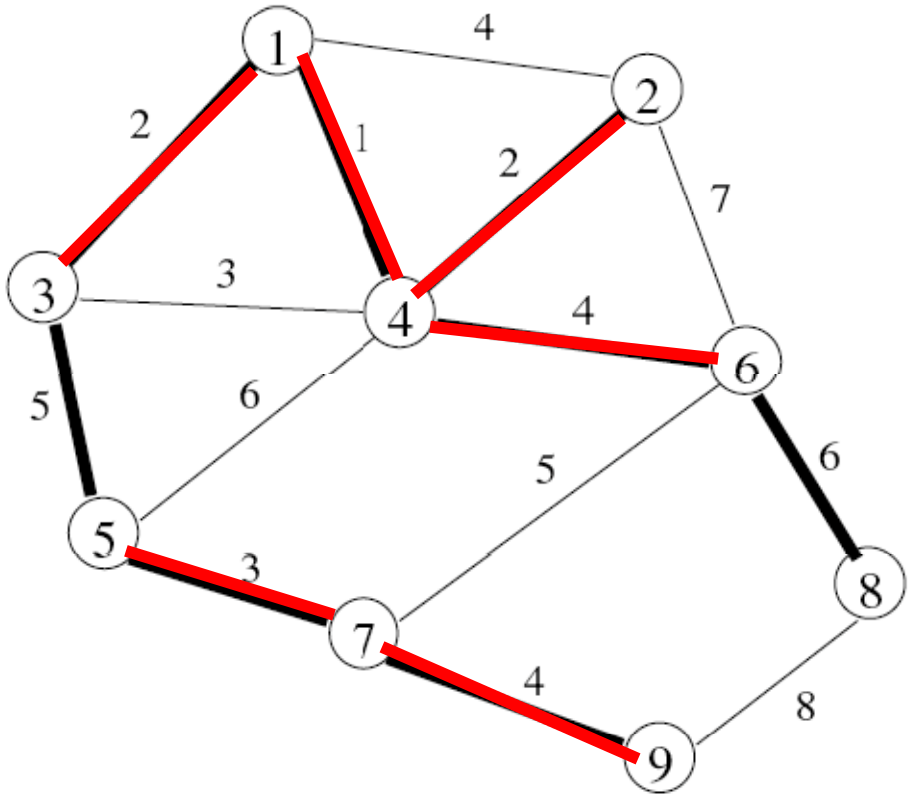

1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
6
5
3
4
8

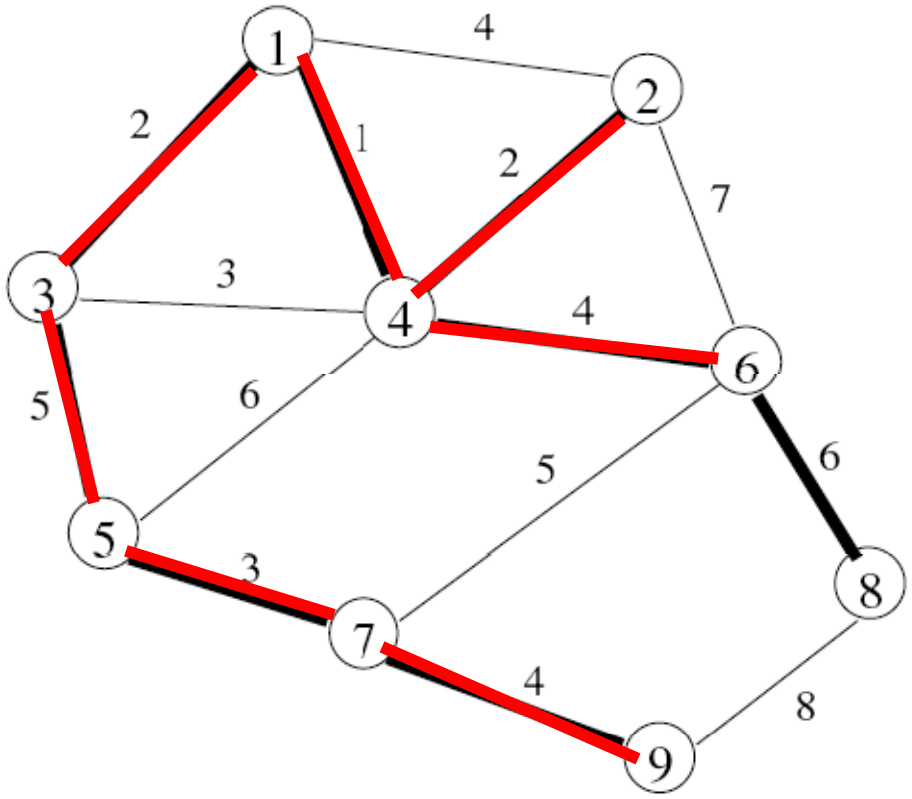

1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
5
6
3
4
8

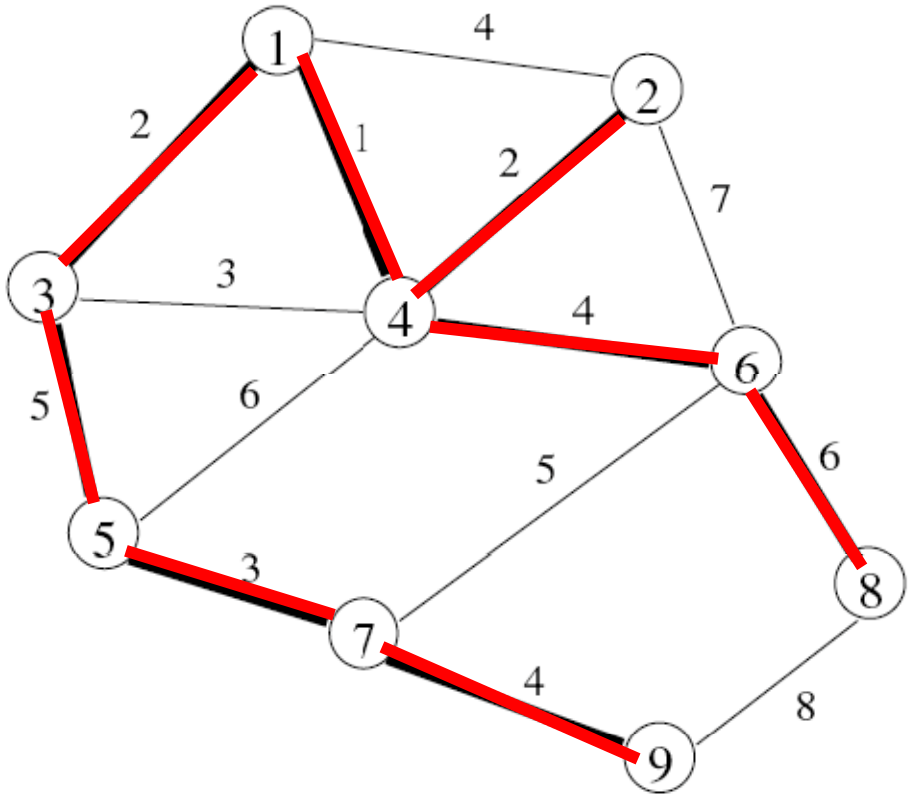

1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
5
6
3
4
8


1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
5
6
3
4
8


1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
4
5
6
5
6
3
4
8


1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
3
4
5
6
6
5
3
4
8


1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
3
4
5
6
6
5
3
4
8


1
2
3
4
5
6
7
8
9
4
2
1
2
7
3
3
4
5
6
6
5
3
4
8

Kruskal's algorithm

- Each time an edge {u,v} is considered such that u and v are not in the same connected component :
  - Add the edge {u,v}
  - Merge the connected components

- **Problem : how do we know that two nodes are not in the same connected component?**

Union-find

- How to
  - Know the connected component of a node
  - Merge 2 connected components

Union-find

- How to
  - Know the connected component of a node
  - Merge 2 connected components
- A **union-find algorithm** is an algorithm that performs two useful operations on a disjoint set data structure:
  - *Find*: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
  - *Union*: Combine or merge two sets into a single set.

Union-find

- **function** MakeSet(x)

  x.parent := x

- **function** Find(x)

  if x.parent == x **return** x

  else return Find(x.parent)

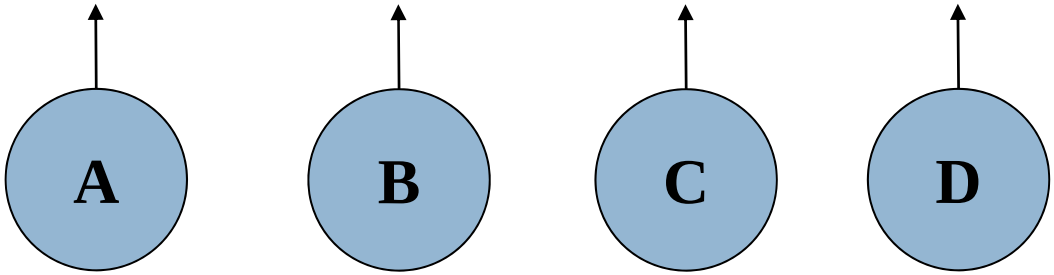
- **function** Union(x, y)

  xRoot := Find(x)

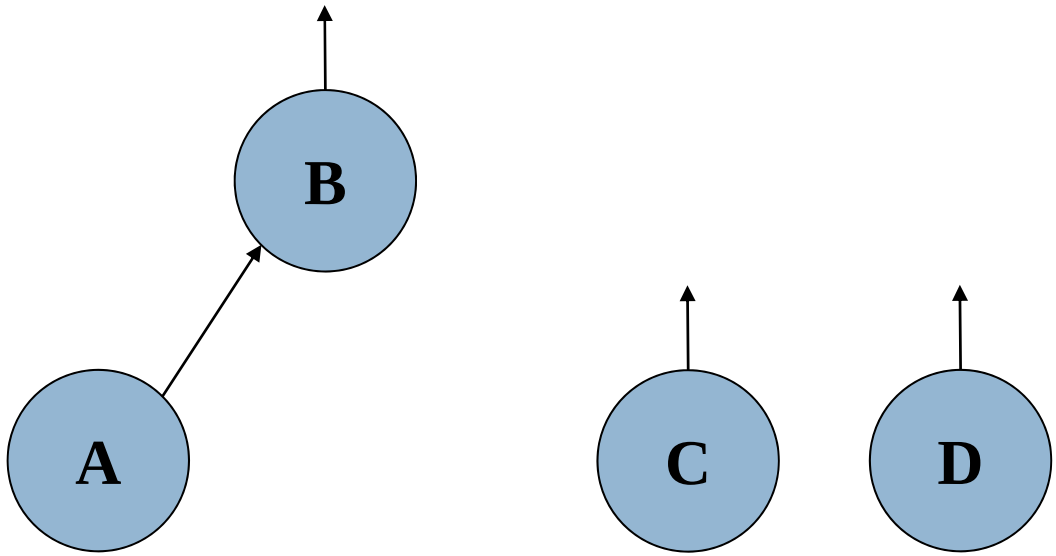
  yRoot := Find(y)

  xRoot.parent := yRoot

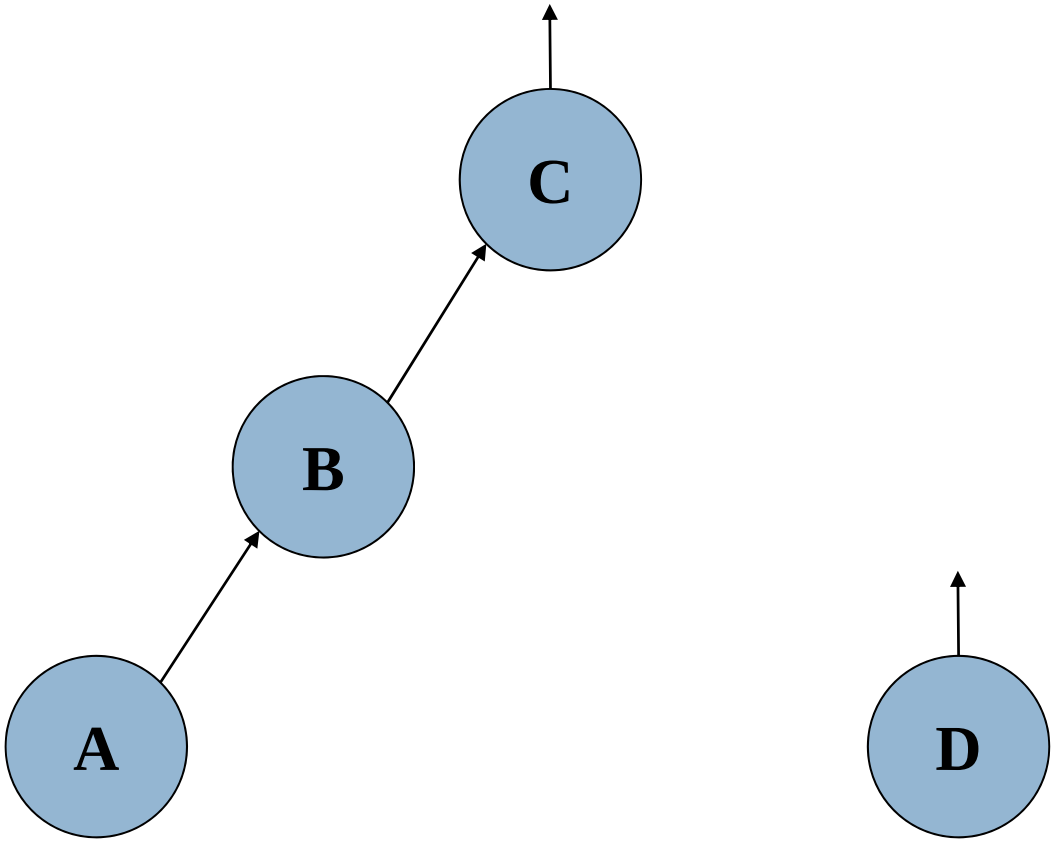
How to implement?



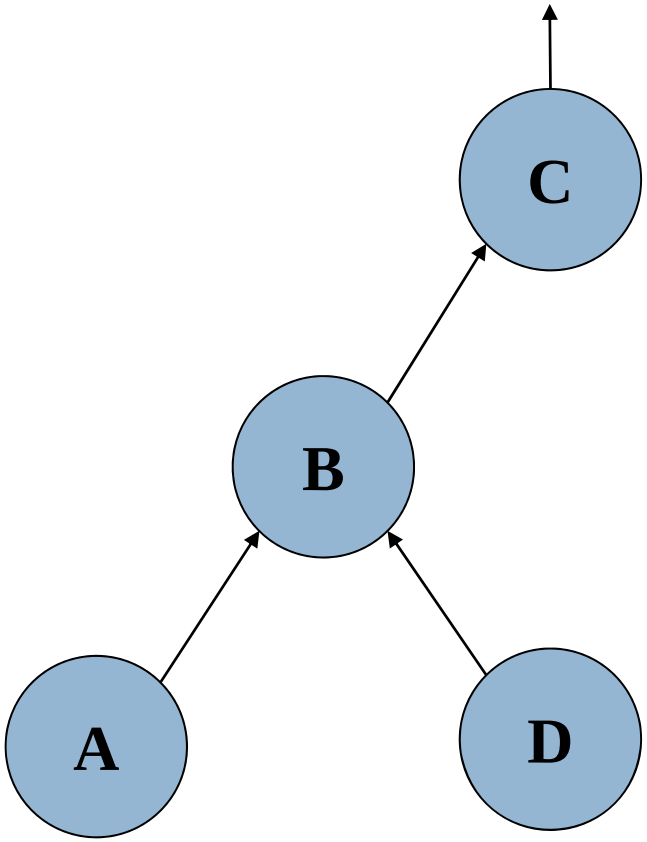
Union(A, B)



Union(A, C)



Union(D, B)



Union-find: heuristics

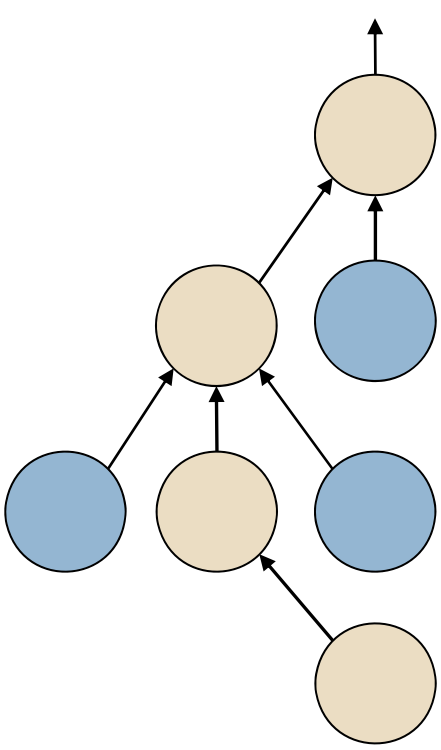
- **Path compression**

```
function Find(x)
        if x.parent == x return x
        else    x.parent := Find(x.parent)
                return x.parent
```

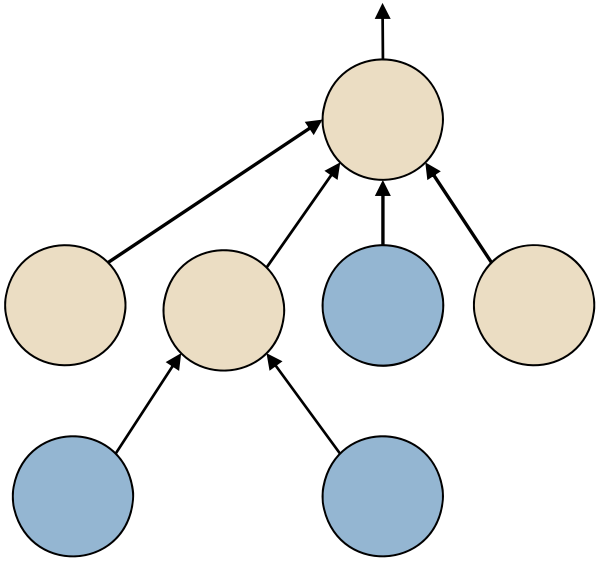
- **Union by rank (or height)**

```
function Union
```

Path Compression



Before



After

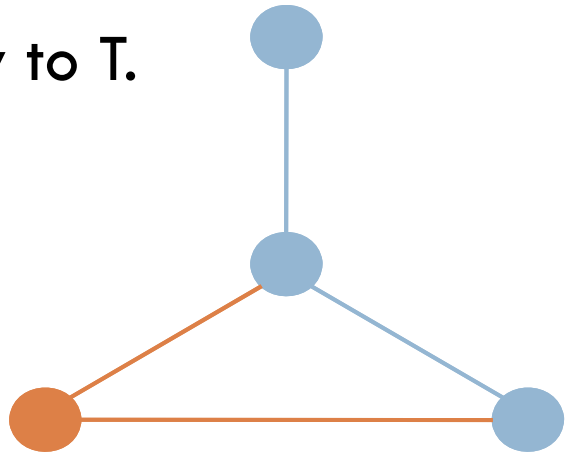
Union-find

- Tarjan proved that
  - The amortized time per operation is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse of A(n,n) the Ackerman. Max of $\alpha(n)$ is 5
  - Optimal bound

Held and Karp bound for TSP

The Held and Karp bound for TSP

- Held and Karp proposed a lower bound based on a relaxation of the degree constraints.
- A 1-tree is a stronger relaxation, which can be obtained by:
  - Choosing any node v (which is called the 1-node)
  - Building a minimum spanning three T on G = (E,V\{v})
  - Adding the smallest edges linking to v to T.



The Held and Karp bound for TSP

- The 1-tree can be tightened through the use of Lagrangian relaxation by relaxing the degree constraint in the well known TSP model:

$$\begin{aligned}
\min \quad & \sum_{e \in E) c_e} x_e \\
s.t. \sum_{e \in \delta(i)} x_e \quad &= \quad 2 \; \forall i \in V \\
\sum_{(i,j) \in (S, N \setminus S)} x_{(i,j)} \quad &\geq \quad 1 \; \forall S \subset N \\
x_e \quad &\in \quad \{0,1\} \; \forall e \in E
\end{aligned}$$

The Held and Karp bound for TSP

- The vector π thus penalizes node degree violation

$$\min \quad \sum_{e \in E) c_e} x_e + \pi_i (2 - \sum_{e \in \delta(i)} x_e)$$

$$s.t. \sum_{(i,j) \in (S, N \setminus S)} x_{(i,j)} \geq 1 \; \forall S \subset N$$

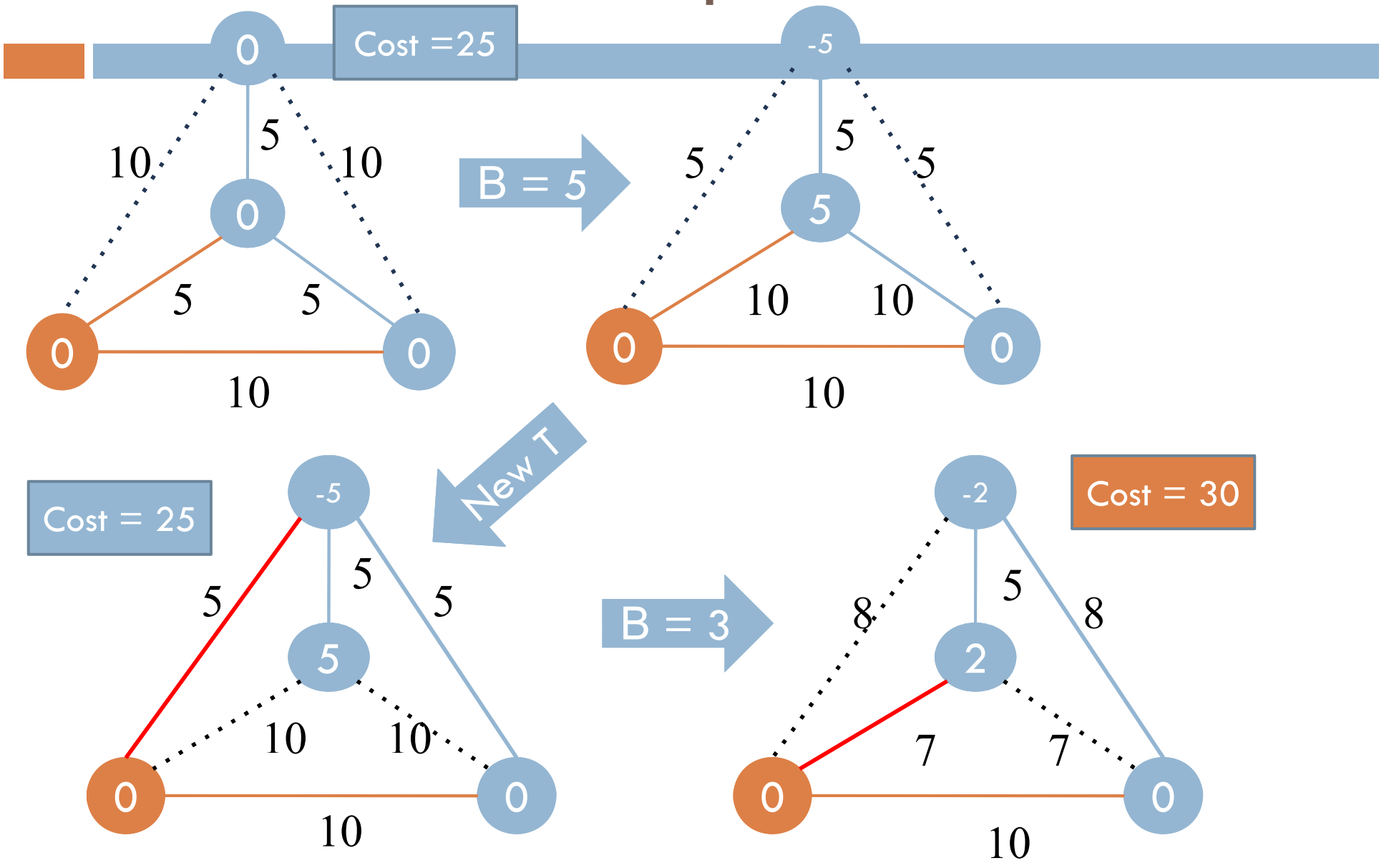
$$x_e \in \{0,1\} \; \forall e \in E$$

- It has been shown that sub-gradient optimization of the π vector allows to increase the value of the HK bound (the cost of T + penalty) to a few % of the optimal TSP solution

The Held and Karp bound for TSP

- The cost of each edge (i,j) is defined as $c'ij = cij + \pi i + \pi j$
- If T is a tour (a TSP solution), the $cost(TSP) = cost(T) - 2 \sum \pi$
  - As every penalty as been counted exactly twice…
- If T is not a tour, then we iteratively
  - increase π for star nodes (degree is > 2) by a factor β * (degree-2)
  - decrease π for leafs (degree = 1) by a factor β
- There are many procedure to adjust step size (β), they are essentially based on sub gradient optimization.

The Held and Karp bound for TSP



Improving HK through CP

- We propose several techniques to filter the graph, such as:
  - Removing edges based on reduced costs
  - Forcing edges based on replacement costs
  - Forcing edges based on the MST computation
  - Forcing edges based on degree
- We will investigate two propagation level
  - One round of filtering
  - Fix point computation (keep filtering until nothing happens)
- We will test this approach on small to medium size TSP and compare with the original HK approach.

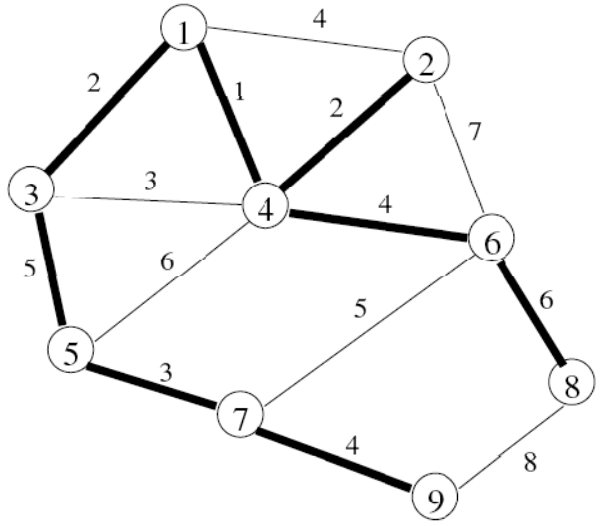
Filtering the graph

- We can do that cleanly
- This is equivalent to define the constraint:
  - **Weighted spanning tree constraint**

Weighted Spanning Tree Constraint

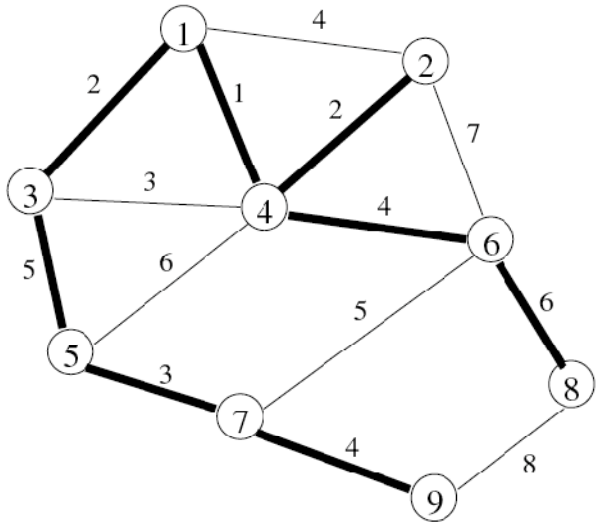
Weighted Spanning Tree Constraint (wst)

- Wst constraint:
  - States that there exists in G a spanning tree whose cost is at most K



Representation

- We choose to represent the constraint using a **set variable**
  - *X* is a set variable representing the set of edges that will form the spanning tree
  - WST(*X*, *K*, *G*) states that *X* forms a spanning tree in *G* with weight at most *K*



- Example: *X* = { (1,3), (1,4), (2,4), (3,5), (4,6), (5,7), (6,8), (7,9) } is a solution to WST(*X*, 30, *G*)

Representation

- In our case, the domain of X, or $D(X)$, is represented by
  - a lower bound of mandatory edges
  - an upper bound of possible edges
- Initially, $D(X) = [\ \emptyset, E\ ]$
- Propagating the WST constraint now amounts to
  1. verifying that a solution exists (consistency check)
  2. removing inconsistent edges from the upper bound
  3. adding mandatory edges to the lower bound

  Bounds consistency: all possible inferences have been made

Consistency Algorithm

- Consistency: check whether there is a solution.
- Easy: Compute an mst T and check if cost(T) $\leq$ K

Filtering Algorithms for wst constraint: **edge removal**

- G. Dooms and I. Katriel (CPAIOR'07)
  - proposed a more general constraint (edge weight are variables instead of scalar, G is not fixed…)
  - gave an arc consistency algorithm for the wmst constraint based on B. Dixon, M. Rauch, R. Tarjan (SIAM J. Computing 1992)

Filtering Algorithms for wst constraint: **edge removal**

- Several Drawbacks:
  - Paper of B. Dixon, M. Rauch, R. Tarjan (SIAM J. Computing 1992) is
    - very complex, use several passes based on different algorithms.
    - Very difficult to follow
    - Very difficult to implement.
  - It is purely theoretical. Some claims in papers about the impracticability of the algorithm.
  - Computation only from scratch

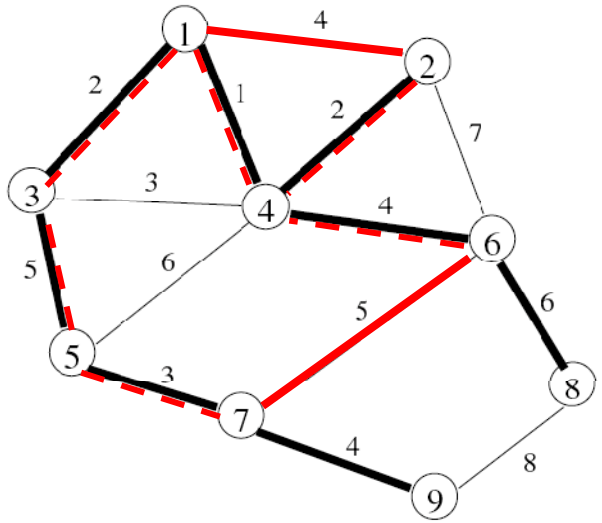
Removing Inconsistent Edges

Dooms and Katriel [*CPAIOR* 2007]

Régin [*CPAIOR* 2008]

Replacement costs

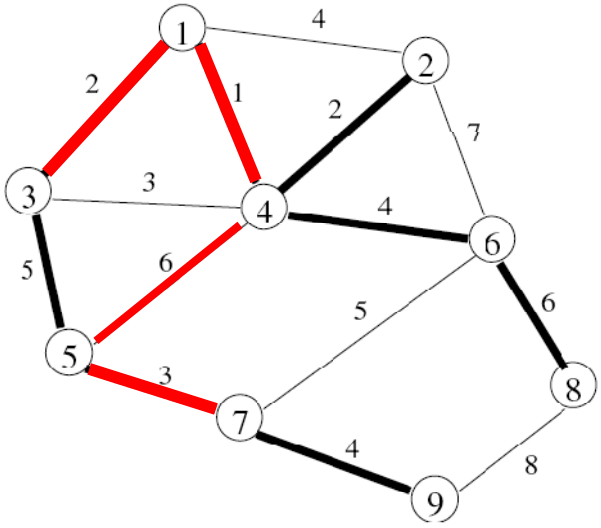
- An edge e is inconsistent iff every spanning tree that contains e has weight > *K*
- Replacement edge
  - Replacement edge minimizes the increase of cost
  - Replacement edge = maximum edge on the *i*-*j* path in *T*

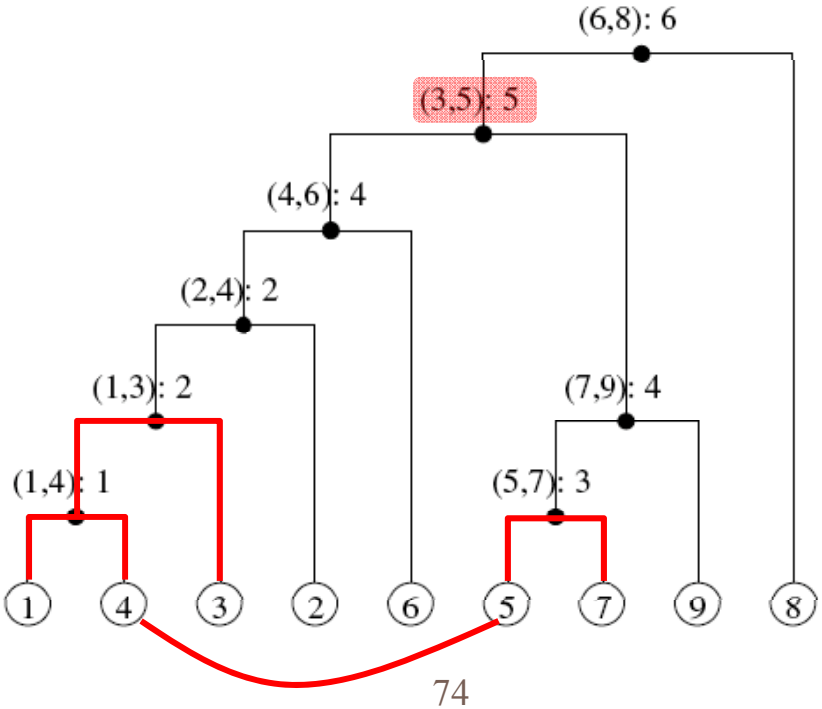


Replacement cost of
- (1,2) is 4 - 2 = 2
- (6,7) is 5 - 5 = 0

Computing replacement costs

- We can now remove an edge e if
  weight(MST) + replacement cost of e > *K*
- The replacement edge = *Lowest Common Ancestor* in the 'connected component tree', *ccTree*





Computing replacement costs (cont'd)

- LCAs can be computed in linear time, but the algorithms are not practical when the tree is unbalanced (as in our case)
- However, the LCAs can be computed by solving a 'Range Minimum Query' problem instead
  - time complexity $O(n \log n)$ [Bender et al., 2005]

Incremental Algorithm

- Non incremental is O(n + m + n log(n))
- Problem also minimum complexity
- Goal: avoid considering all the edges.
- Maintain of the ordered list of edges is simple

Incremental AC Algorithm

- Some edges have been deleted and the mst changes and so its cost. Therefore, some arcs must be reconsidered because
  - We need to re-check the consistency with the new cost of the mst
  - Their support has been deleted

Incremental AC Algorithm

- For an edge (i,j), if its support (u,v) is valid then we just need to check if
  $cost(i,j) - cost(u,v) + cost(T') \leq K$
- We can do that quickly for all valid supports. If the supported edges are ordered, then we can stop at a time and removes all the remaining edges.
- Question: how to compute the valid support?
  - Some edges having no support may need one
  - We can use the ccTree to determine the unvalid support

Incremental AC Filtering

Formula: $cost(i,j) - cost(u,v) + cost(T) \leq K$

Edges such that:
$Cost(i,j) - minEdgeCost(T) + cost(T) \leq K$

Edges such that:
$Cost(i,j) - maxEdgeCost(T) + cost(T)$

Now we have T' instead of T with cost(T') ≥ cost(T)

Edges are ordered by nondecreasing cost

Incremental AC Filtering

Formula: $cost(i,j) - cost(u,v) + cost(T) \leq K$

Edges such that:
$Cost(i,j)\text{-}minEdgeCost(T)+cost(T) \leq K$

Edges such that:
$Cost(i,j)\text{-}maxEdgeCost(T)+cost(T)$

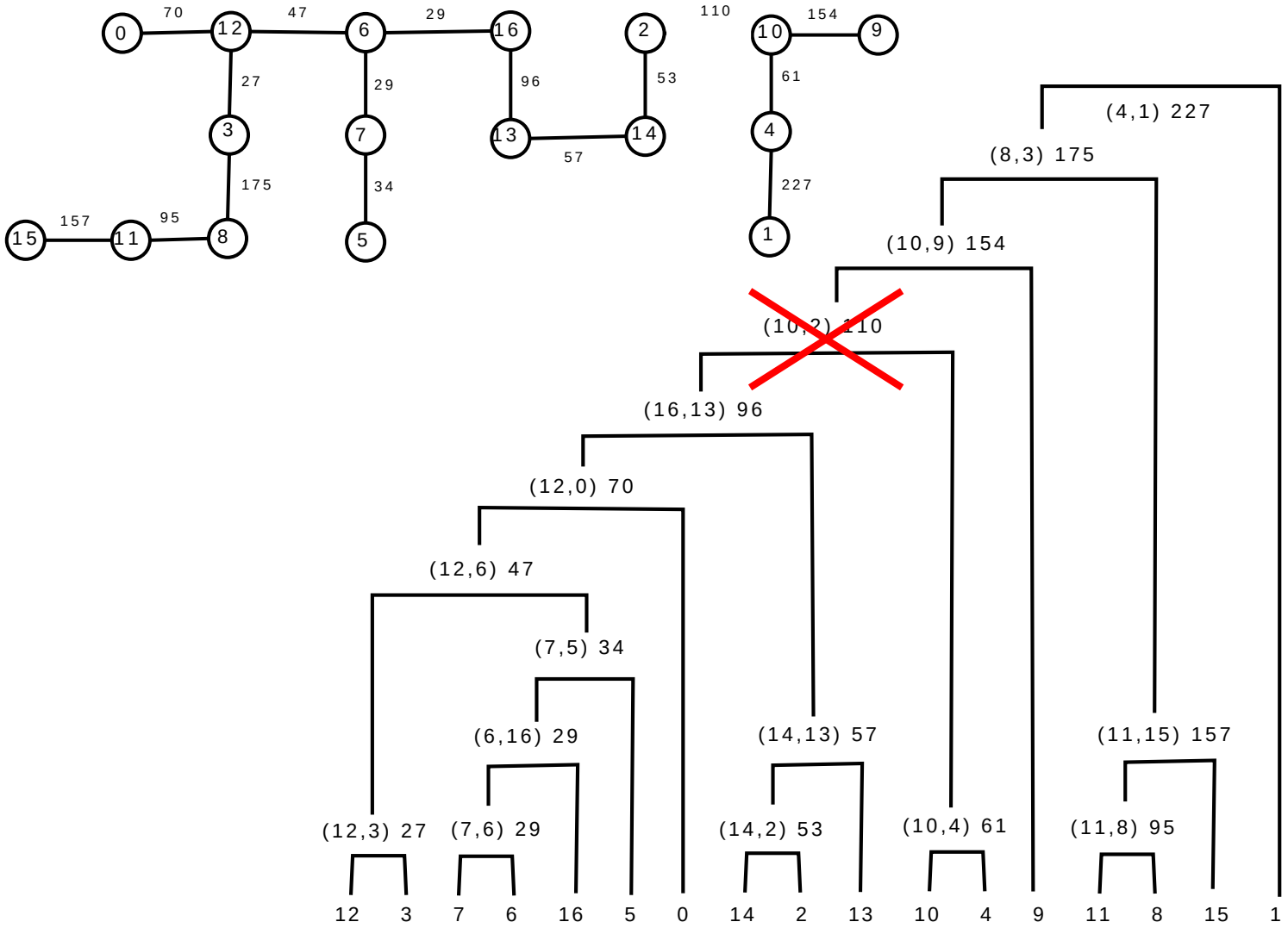
Now we have T' instead of T with cost(T') ≥ cost(T)

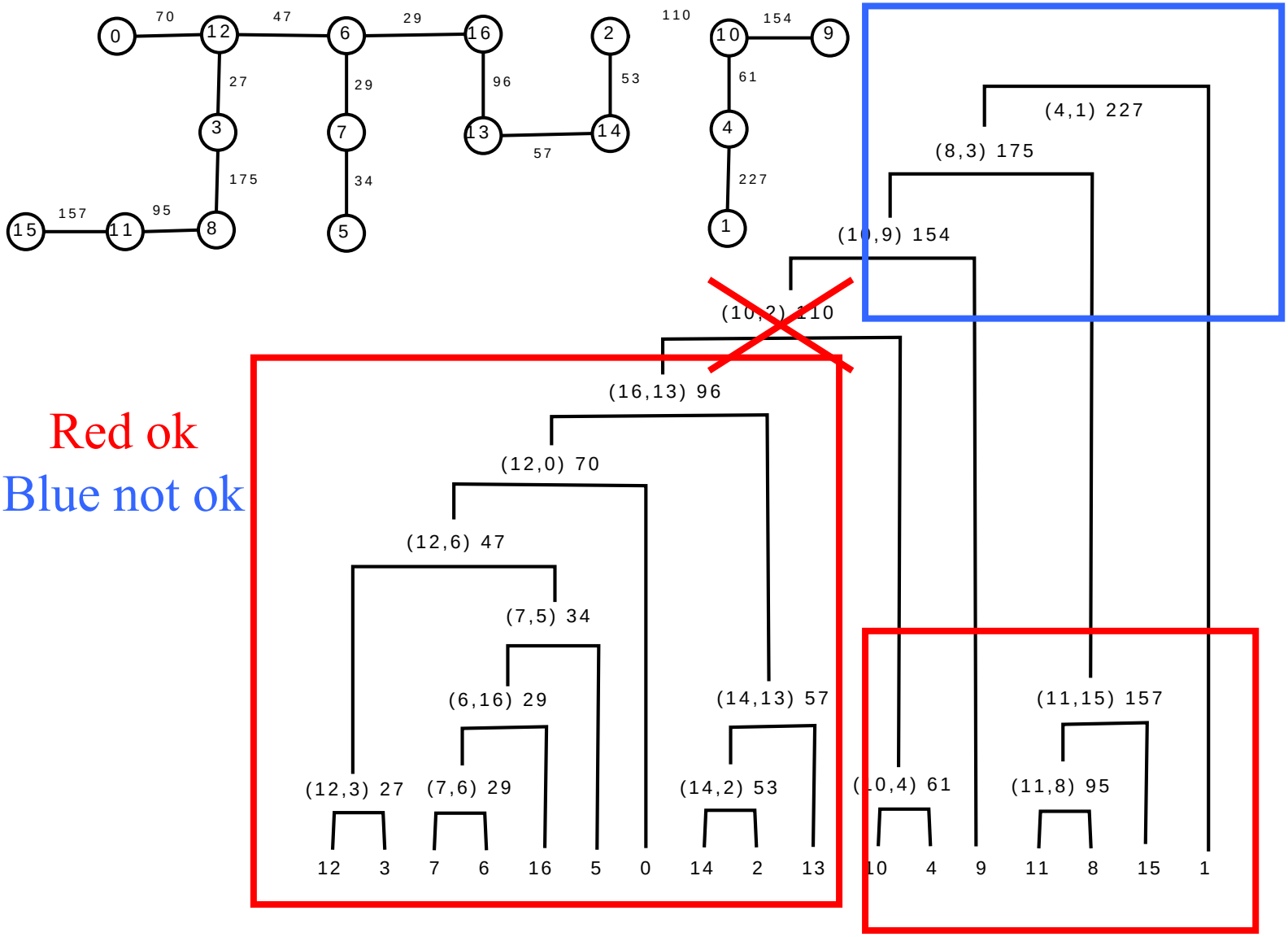
Entering edges

Edges are ordered by nondecreasing cost

Non Valid Support

- Any edge (i,j) whose node in ccTree has been removed or whose a descendant node in ccTree has been removed is no longer valid as a support.
- All the edges supported by these nodes are named **pending edges**
- We need to find a support for all entering + pending edges.


0
12
6
16
2
10
9
3
7
13
14
4
15
11
8
5
1
70
47
29
110
154
27
29
96
53
61
175
34
57
227
157
95
(4,1) 227
(8,3) 175
(10,9) 154
(10,2) 110
(16,13) 96
(12,0) 70
(12,6) 47
(7,5) 34
(6,16) 29
(14,13) 57
(11,15) 157
(12,3) 27
(7,6) 29
(14,2) 53
(10,4) 61
(11,8) 95
12 3 7 6 16 5 0 14 2 13 10 4 9 11 8 15 1


0
12
6
16
2
10
9
3
7
13
14
4
15
11
8
5
1
70
47
29
110
154
27
29
96
53
61
157
95
175
34
57
227
(4,1) 227
(8,3) 175
(10,9) 154
(10,2) 110
(16,13) 96
(12,0) 70
(12,6) 47
(7,5) 34
(6,16) 29
(14,13) 57
(12,3) 27
(7,6) 29
(14,2) 53
(10,4) 61
(11,8) 95
(11,15) 157
12 3 7 6 16 5 0 14 2 13
10 4 9 11 8 15 1
Red ok
Blue not ok

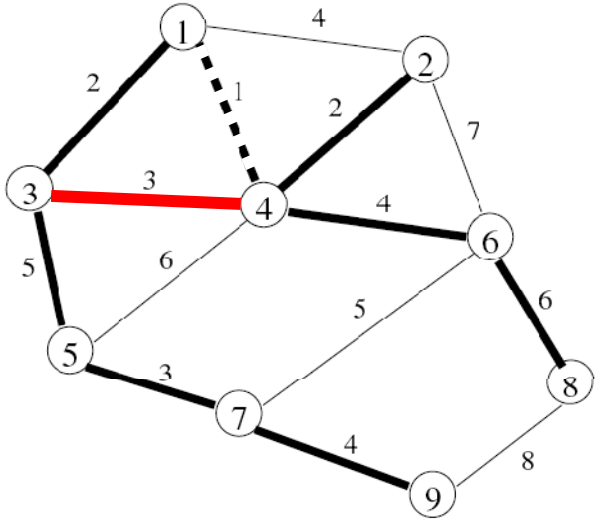
Restoration

- Non monotonic value of the mst: we need to restore the previous mst.
- The restored edges are added to the pending edges and we recompute support for them

Replacement cost for tree edges

- The replacement cost of a *tree* edge e is
  w(*T'*) - w(*T*), where

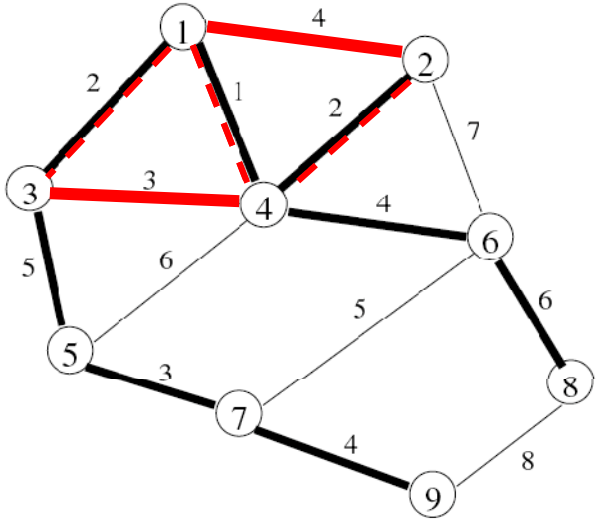
  T is a minimum spanning tree of *G*, and *T'* is a minimum spanning tree of *G* \ e
- In other words, it represents the minimum marginal increase if we replace e by another edge
- An edge e is mandatory iff its replacement cost + w(*T*) > *K*



Replacement cost of (1,4)?
we need to find the cheapest edge to reconnect: 3 - 1 = 2

Computing replacement costs for tree edges

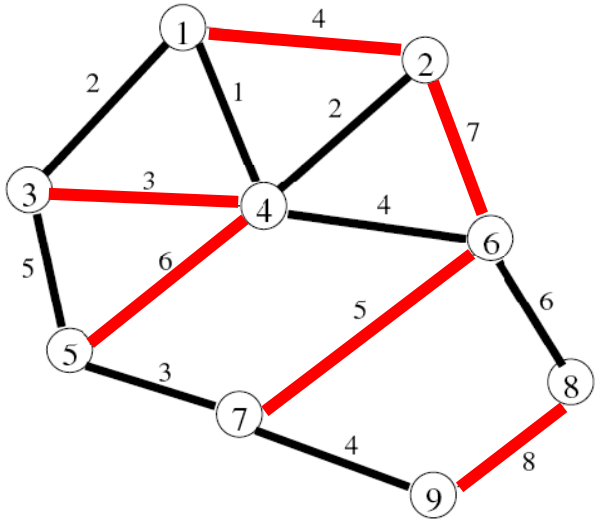
1. Compute minimum spanning tree *T* in *G* and Mark all edges in *T* 'unmarked'
2. Consider non-tree edges, ordered by non-decreasing weight:
   - For non-tree edge (*i*,*j*), traverse the *i*-*j* path in T
   - Mark all unmarked edges e on this path, and assign replacement cost w(*i*,*j*) - w(e)
3. Basic time complexity O(*mn*)

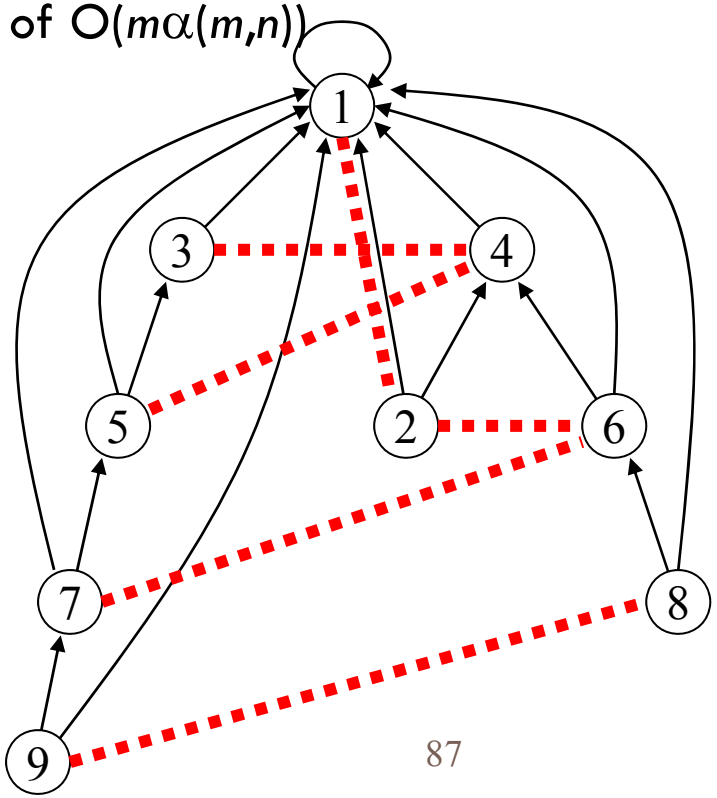


non-tree edge	mark edge	replacement cost
(3,4)	(1,4)	3 - 1 = 2
	(1,3)	3 - 2 = 1
(1,2)	(2,4)	4 - 2 = 2
	(edge (1,4) already marked)	
...		

Improving the time complexity

- '**Contracting**' the marked edges (that is, we merge the extremities of the edge)
  - First, root the minimum spanning tree
  - Apply Tarjan's union-find technique during the algorithm
  - This leads to a time complexity of $O(m\alpha(m,n))$





Maintaining Mandatory Edges

Updating the ccTree

- Not consider in the previous publication
- Consequences of mandatory edges
  - if non-tree edge becomes mandatory (e.g., because of other constraints) we need to force it into the MST, and update the ccTree
  - if a tree edge becomes mandatory, it can remain in the tree, but we must forbid it to be used as a replacement edge in the ccTree

Updating the ccTree

- Our goal is therefore to update the ccTree when mandatory edges are taken into account

  Version 1: re-compute the ccTree by modifying edge weights

  (Efficient when several edges have become mandatory)

  Version 2: repair the ccTree

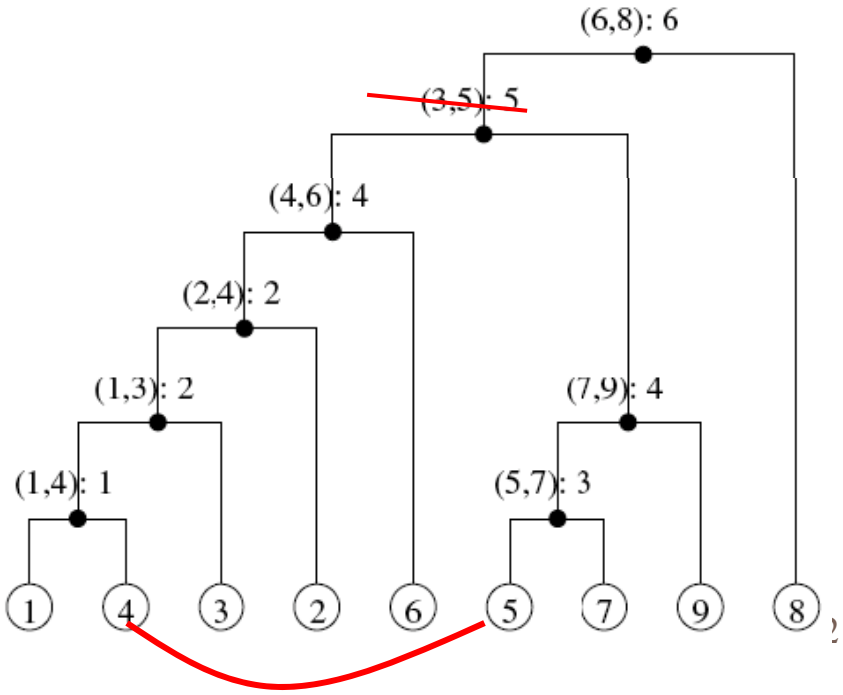
  (Efficient when a few edges have become mandatory)

Version 1: Re-computing the ccTree

- The ccTree is constructed using Kruskal's algorithm, which adds the edges by non-decreasing weight
- Mandatory edges cost = low weight, so that they are added first when re-building the ccTree
  - As such, they can never appear as replacement edge
  - (Note that non-tree edges in a component formed by mandatory edges can be removed)
- Time complexity as before: O(n)

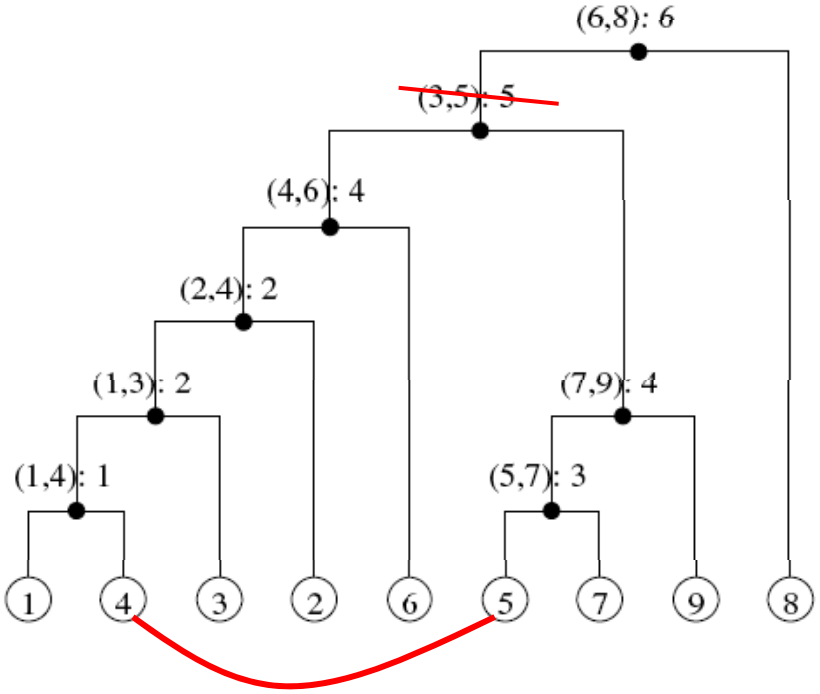
Version 2: Repairing the ccTree

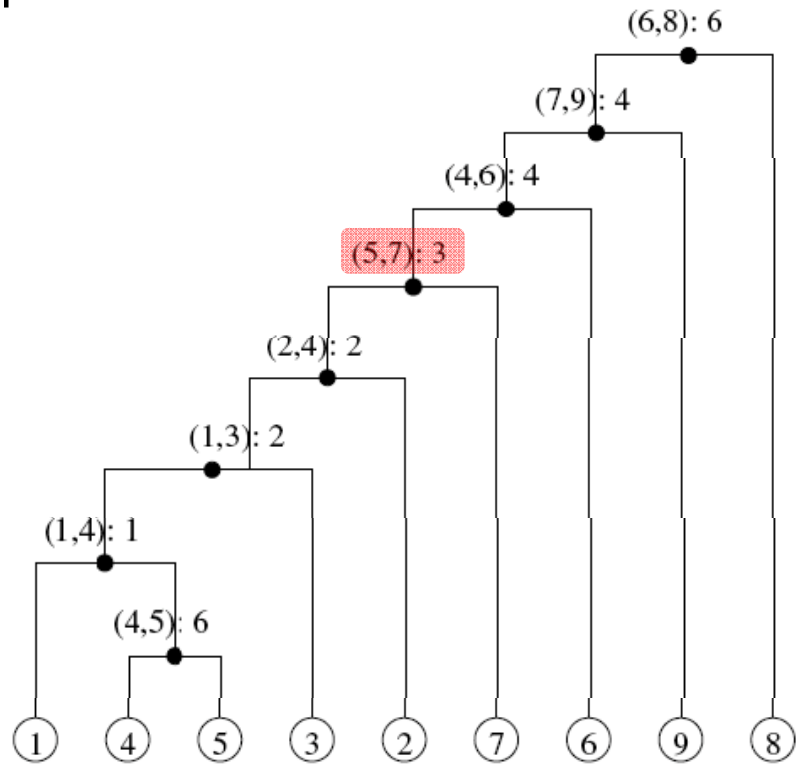
- Consider a non-tree edge (*i*,*j*) that becomes mandatory
- It will replace the LCA of *i* and *j* in the ccTree, while the edge represented by this LCA will disappear
- We thus need to rebuild the ccTree up to the LCA



Repairing the ccTree (cont'd)

- First, we merge i and j (we assume that i < j in ccTree)
- Second, we need to re-sort the two parts again
- Proceed upwards from i and j until the parent of the i-path has higher weight than parent of j-path, and insert at that point
- Only *local* repair necessary (up to LCA)





Other filtering algorithms

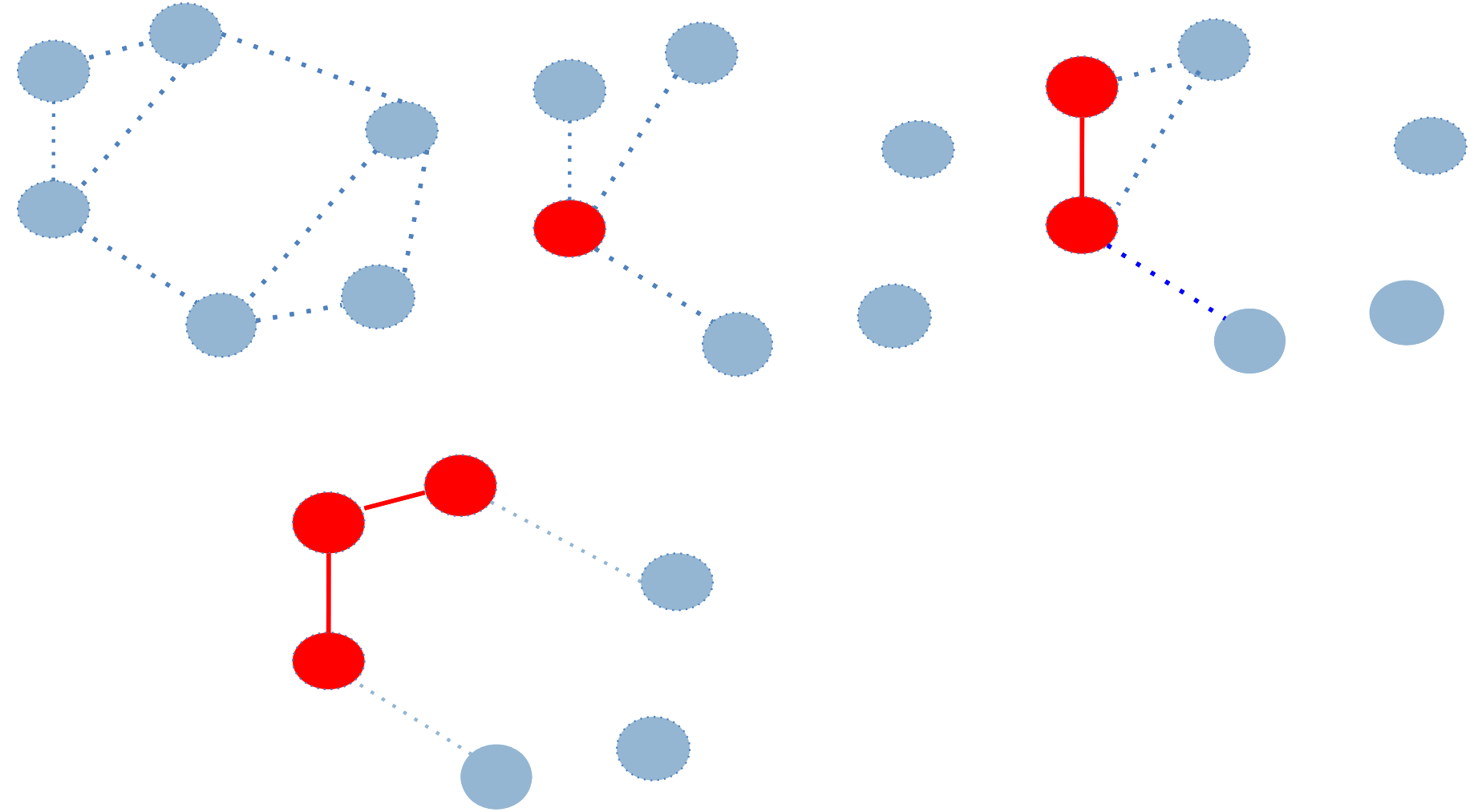
Other filtering

- We reconsider a circuit problem (TSP) and not only a weighted spanning tree

Forcing Edges when builging MST

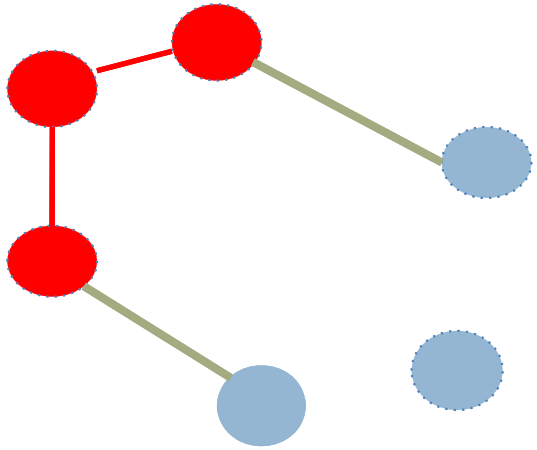
- Prim's algorithm
- Starting from any node i,
  - partitions the graph into disjoints subsets $S = \{i\}$ and $S^* = V \setminus i$
  - creates an empty tree T.
- Then it iteratively adds to T **the minimum edge (i,j) $\in$ (S,S*)**, defined as the set of edges where $i \in S$ and $j \in S^*$, and moves j from S* to S.

Forcing Edges when builging MST



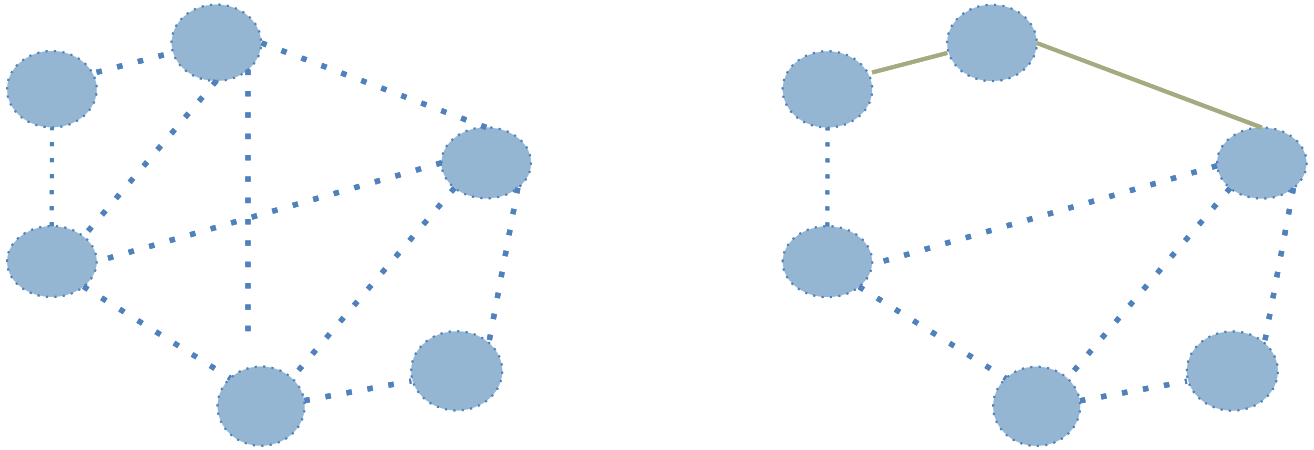
Forcing Edges when builging MST

- Whenever (S,S*) contains only 2 edges, then it is possible to force these 2 edges in the solution…



Forcing edges based on degree

- When a node has 2 adjacent forced edges, we can remove all its other adjacent edges…



Implementation and experimental results

Propagation level

- We consider two propagation setting
- **One round:** that is executing each filtering once
- **Fix point computation :** executing each filtering, in turns, until no edge can be forced nor filtered
- More fine grain propagation is a bit tricky…

Branching strategy

- We have considered three branching scheme, all based on the current solution of the 1-tree computed in HK.
- **Star**: find the largest star and force out an adjacent edge with the largest cost.
- **In**: forcing in the arc with the maximum reduced cost.
- **Out**: forcing out the edge with the maximum replacement cost.

Experimental results

- **Preliminary results:**
  - **goal validate our approach**
- To evaluate the benefits of using CP within the Held-Karp algorithm, we ran experiments on several instances of the TSPlib.

- To eliminate the impact of the upper bound can have on search tree, we ran these experiments using the UB computed by the LKH heuristic.

Experimental Results

	Original HK		1 round		fix point	
	Time	Nodes	Time	Nodes	Time	Nodes
dantzig42	0,65	92	**0,09**	**4**	0,17	4
swiss42	0,79	112	**0,09**	**8**	**0,09**	**8**
att48	1,7	140	**0,21**	18	0,23	**15**
gr48	94	13554	**5,18**	**2481**	7,38	3661
hk48	1,37	94	0,17	4	**0,16**	**4**
eil51	15,9	2440	**0,39**	**131**	0,84	426
berlin52	0,63	80	**0,02**	**0**	**0,02**	**0**
brazil58	13	878	1,09	319	**1,02**	**296**
st70	236	13418	1,21	183	**1,1**	**152**
eil76	15	596	1,03	125	**0,88**	**99**
rat99	134	2510	5,44	592	**4,88**	**502**
kroD100	16500	206416	**11**	7236	50,83	**4842**
rd100	67	782	0,76	0	**0,73**	**0**
eil101	187	3692	**8,17**	**1039**	9,59	1236
lin105	31	204	**1,81**	**4**	**1,85**	**4**
pr107	41	442	**4,65**	**45**	**4,49**	**48**

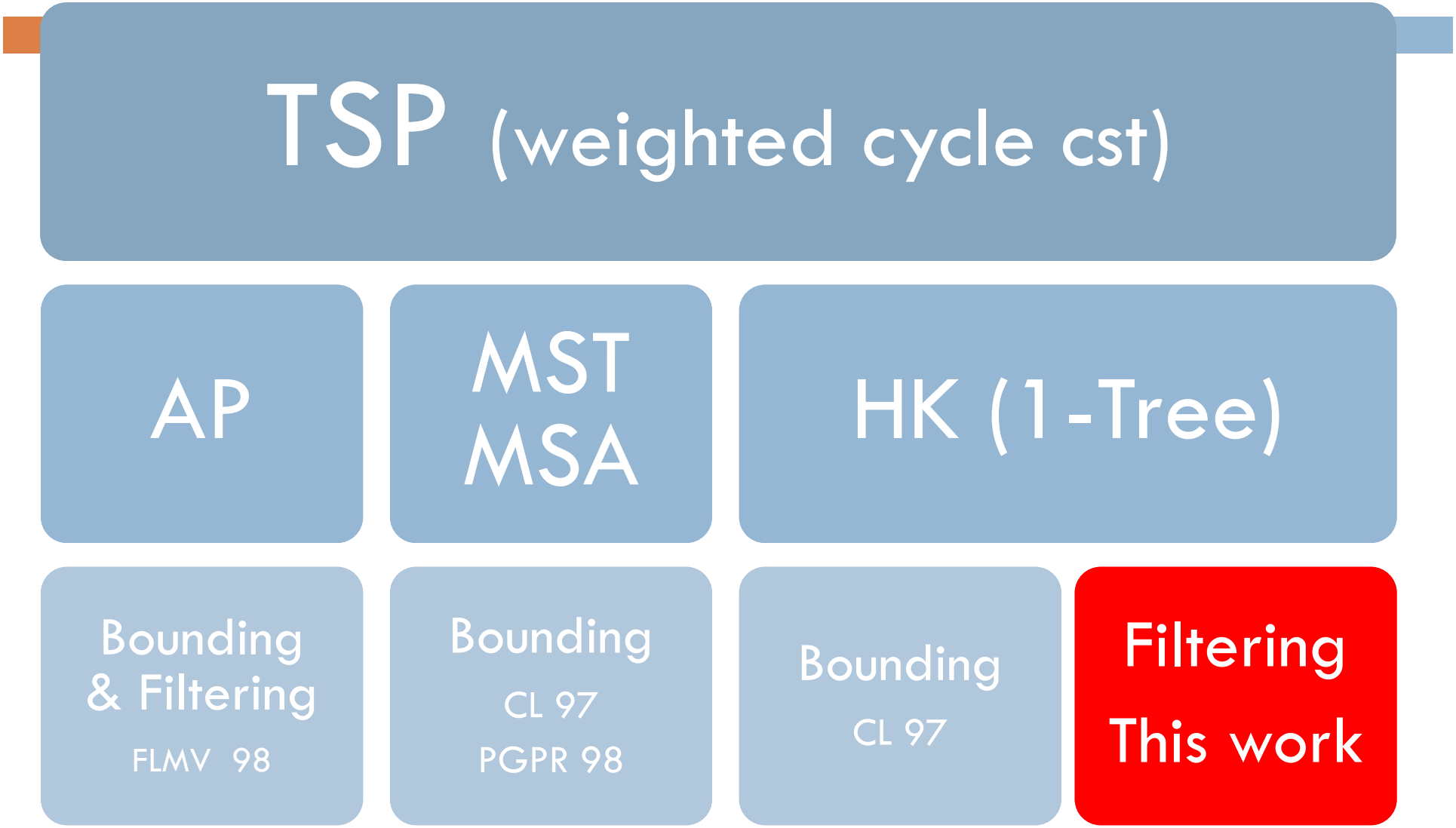
Speedups (a rough analysis)

	Time	Nodes
dantzig42	6	23
swiss42	9	14
att48	8	9
gr48	15	5
hk48	8	24
eil51	30	12
brazil58	12	3
st70	205	81
eil76	16	5
rat99	26	5
kroD100	912	36
eil101	21	3
lin105	17	51
pr107	9	10

> 500 nodes

< 500 nodes

From a CP viewpoint



Discussion and Conclusion

Discussion : large scale problems

- A lot of work must be done
- Large scale problems:
  - 100,000 nodes
  - Potentially 10,000,000,000 edges
  - n.O(n log(n)) is fine but n .O(m) may be complex!

Discussion: new research areas

- New point of view: CP is based on filtering algorithm, i.e. : Given a property P defining a necessary condition for an element to be in a solution
  Find as quickly as possible ALL elements that do not satisfy P
- Close to sensitivity analysis, but also different (for instance we only have monotonic modifications).

Discussion: new research areas

- The very same algorithm is called thousand times (million sometimes)
- The incremental aspect of the algorithm becomes really important.

Conclusion

- We have presented filtering algorithms for the weighted spanning tree constraint
  - practical and incremental algorithms to eliminate edges with unacceptable replacement cost
  - practical and incremental algorithm to determine mandatory edges
  - incremental algorithms for maintaining the ccTree in the presence of mandatory edges
- Incorporation of this algorithm into Held and Karp procedure
- Encouraging experimental results