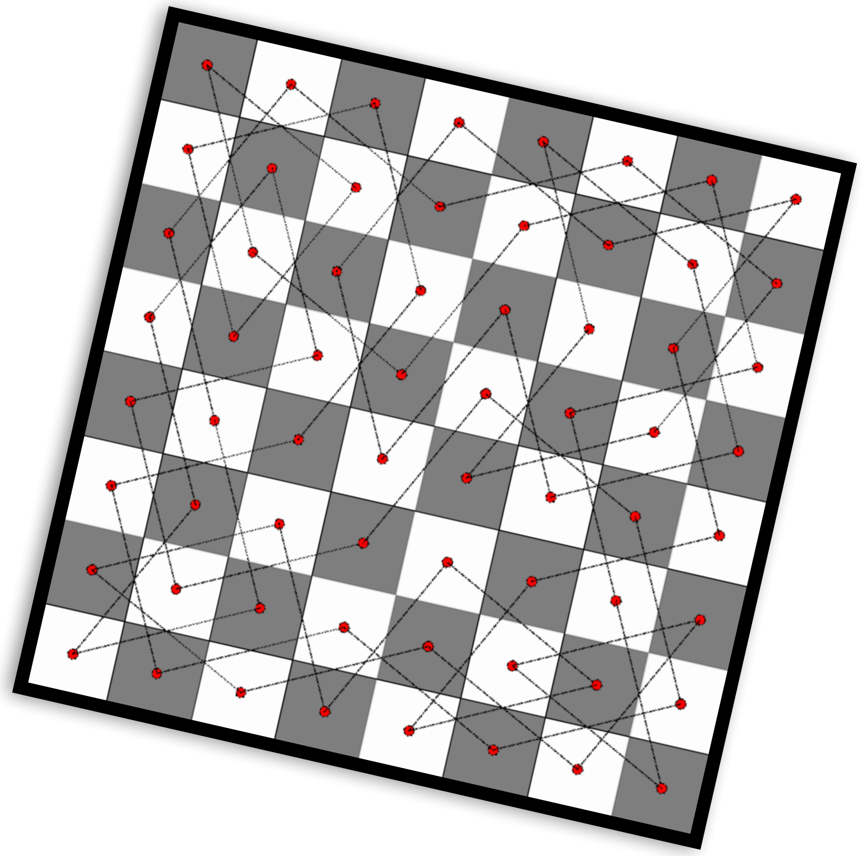
Advanced Algorithms for Optimization

LINFO2266

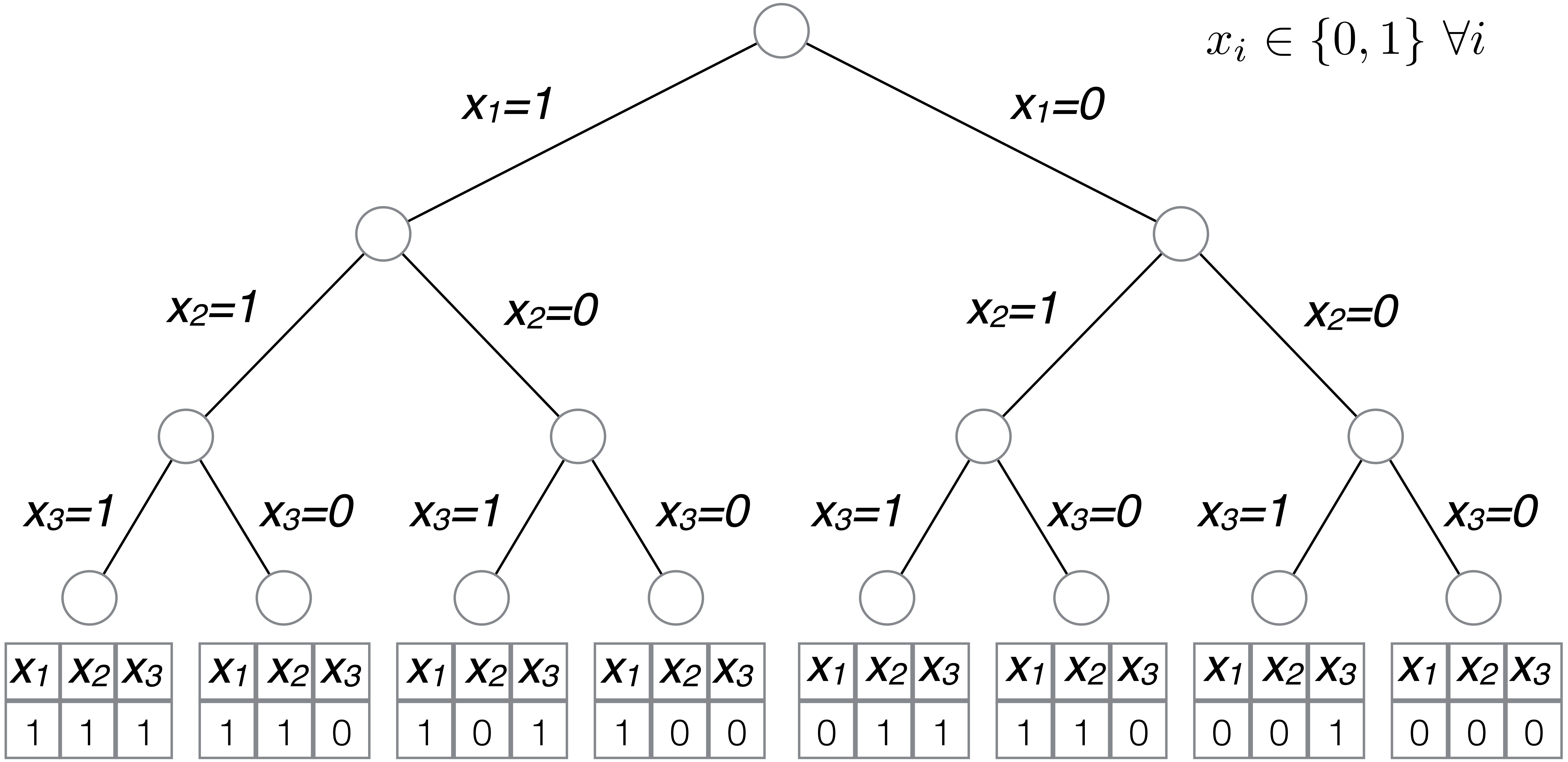
Branch & Bound

Pierre Schaus



Brute Force: Tree View (⚠️ for now, assume DFS)

$$\texttt{maximize } 28x_1 + 30x_2 + 20x_3$$
$$\texttt{subject to } 4x_1 + 6x_2 + 4x_3 \leq 9$$
$$x_i \in \{0, 1\} \; \forall i$$



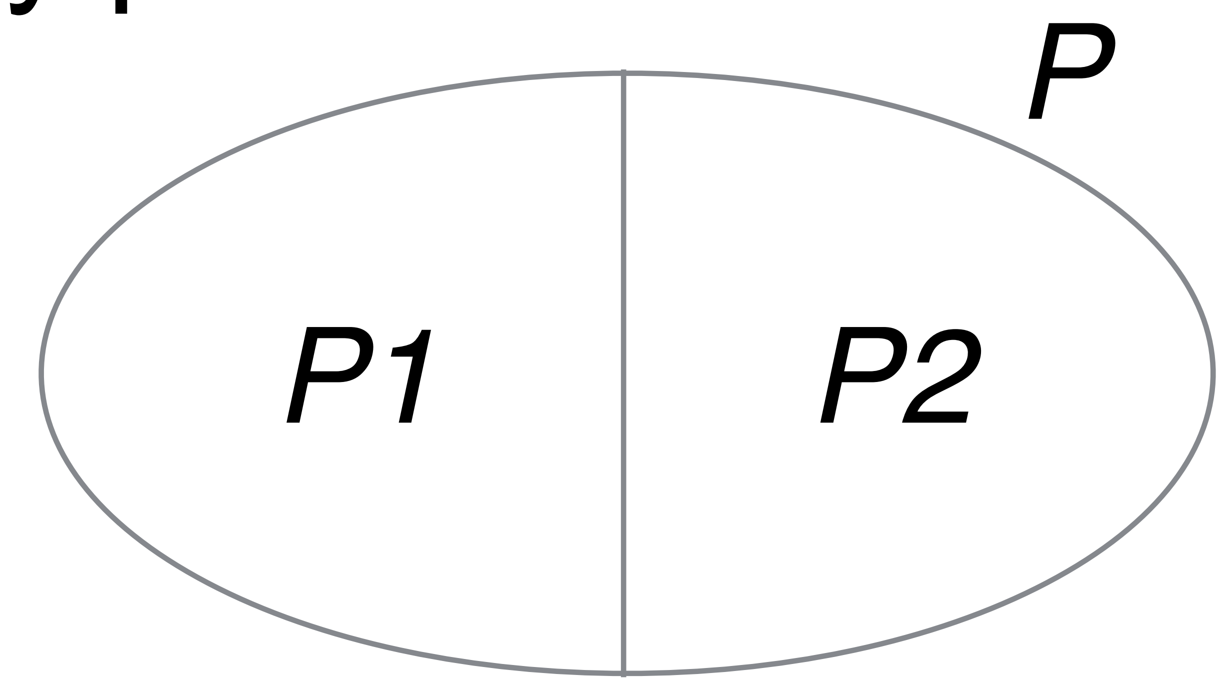
Can we reduce this search space
by cutting some branches?
What upper-bounding procedure do you suggest?

Branch & Bound: The Intuition

- Maximization Problem *P* : *maximize obj*

- Assume I have a feasible solution in hand (for instance obtained with a greedy algorithm) with objective *obj**

- Assume I can decompose my problem: $P = P1 \cup P2$



- Assume I have an upper-bound procedure *UB(P)*
  - Gives me an upper bound on obj for problem P

Do you prefer small or large upper-bound?

- If $UB(P1) \leq obj^*$, I can discard exploration of *P1* (idem for *P2*)

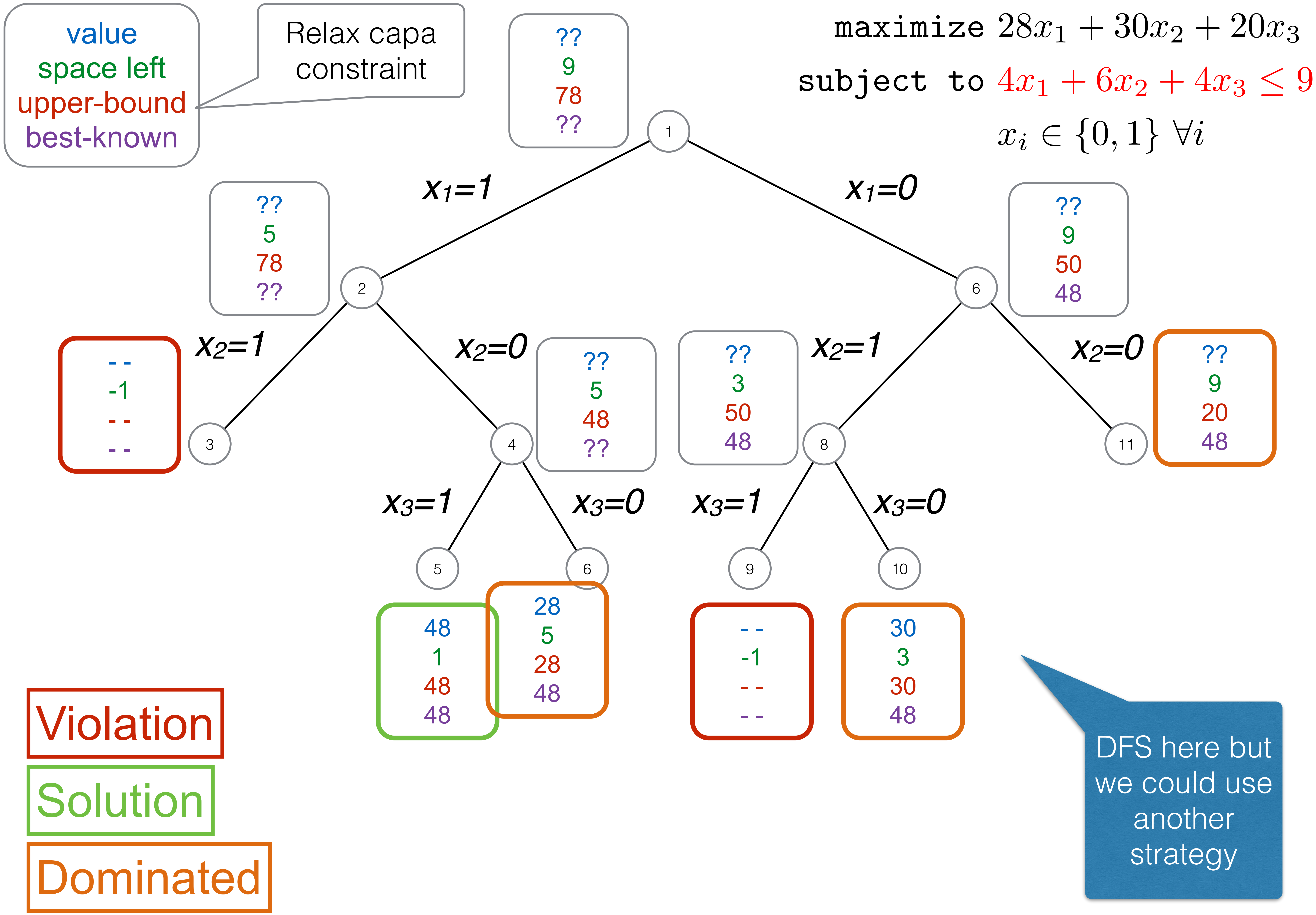
Knapsack: Upper-Bound

$$\begin{aligned} \texttt{maximize} \; & 28x_1 + 30x_2 + 20x_3 \\ \texttt{subject to} \; & 4x_1 + 6x_2 + 4x_3 \leq 9 \\ & x_i \in \{0, 1\} \; \forall i \end{aligned}$$

What upper-bounding procedure do you suggest?
(Think about relaxation)

Branch & Bound: Capa relaxation

$$\text{maximize } 28x_1 + 30x_2 + 20x_3$$
$$\text{subject to } 4x_1 + 6x_2 + 4x_3 \leq 9$$
$$x_i \in \{0,1\} \; \forall i$$



Knapsack - Linear Relaxation = Easy problem

- Relax the integrality constraint (called linear relaxation)

$$\begin{aligned} \texttt{maximize}\ & 28x_1 + 30x_2 + 20x_3 \\ \texttt{subject to}\ & 4x_1 + 6x_2 + 4x_3 \leq 9 \\ & x_i \in [0,1]\ \forall i \end{aligned}$$

Linear Relaxation

- Sort the items according to ratio v_i/w_i (we assume it is the case)

- Find the critical item

$$j = \min\{i \in I : \sum_{k \in 1..i} w_k > C\}$$

$$UB = \sum_{i<j} v_i + \frac{(C - \sum_{i \in 1..j-1} w_i)}{w_j} \cdot v_j$$

Exercise: Prove this solves the linear relaxation

Linear Relaxation

- Why is this the optimal linear relaxation ? Sketch of proof. Instance with C=14 (Capa)

index	0	1	2	3	4
v/w	4	3.66	3.6	3	1
v	28	22	18	6	1
w	7	6	5	2	1
x*	1	1	1/5	0	0

Bound = 28+22+1/5*18=53.6

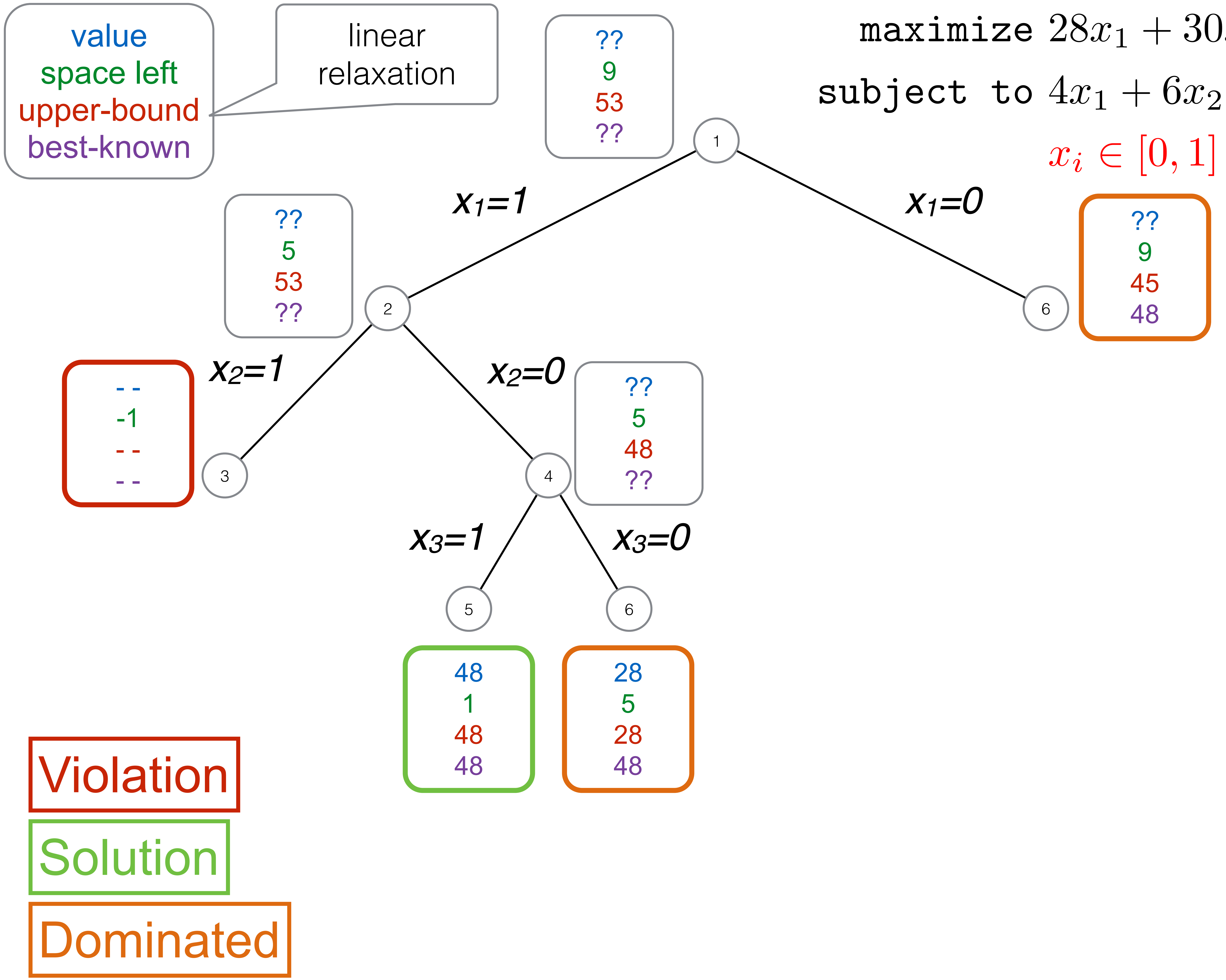
Let i* = “critical item index”
Assume x* is not optimal but instead x’ is.
Then there exist two variables with index i < j such that xi’<1 and xj’ >0. This solution can be improved by doing xi’+ε and xj’-ε since the items are sorted decreasingly by value/volume (contradiction)

Branch & Bound: Linear relaxation

$$\texttt{maximize } 28x_1 + 30x_2 + 20x_3$$

$$\texttt{subject to } 4x_1 + 6x_2 + 4x_3 \leq 9$$

$$x_i \in [0, 1] \ \forall i$$



Branch & Bound Implementation

https://github.com/pschaus/linfo2266

Implementation BranchAndBound.java

```
public static void minimize(OpenNodes openNode) {

    double upperBound = Double.MAX_VALUE;
    int iter = 0;

    while (!openNode.isEmpty()) {
        iter++;
        Node n = openNode.remove();
        if (n.isFeasible() && n.lowerBound() < upperBound) {
            upperBound = n.lowerBound();
        }
        else if (n.lowerBound() < upperBound) {
            for (Node child: n.children()) {
                openNode.add(child);
            }
        }
    }
    System.out.println("#iter:"+iter);
}
```

Collection with open nodes

Update best solution

Pruning by upper-bounding

Node Interfaces

```
interface Node {
    double lowerBound();
    boolean isFeasible();
    List<Node> children();
}
```

This implementation will be problem specific. A node contains the state of a problem modified according the “branching” decisions

```
interface OpenNodes<N extends Node> {
    void add(N n);
    N remove();
    boolean isEmpty();
    int size();
}
```

Collection with open nodes. Think about possible ways to implement DFS/BFS 🤔

Knapsack Node Implementation

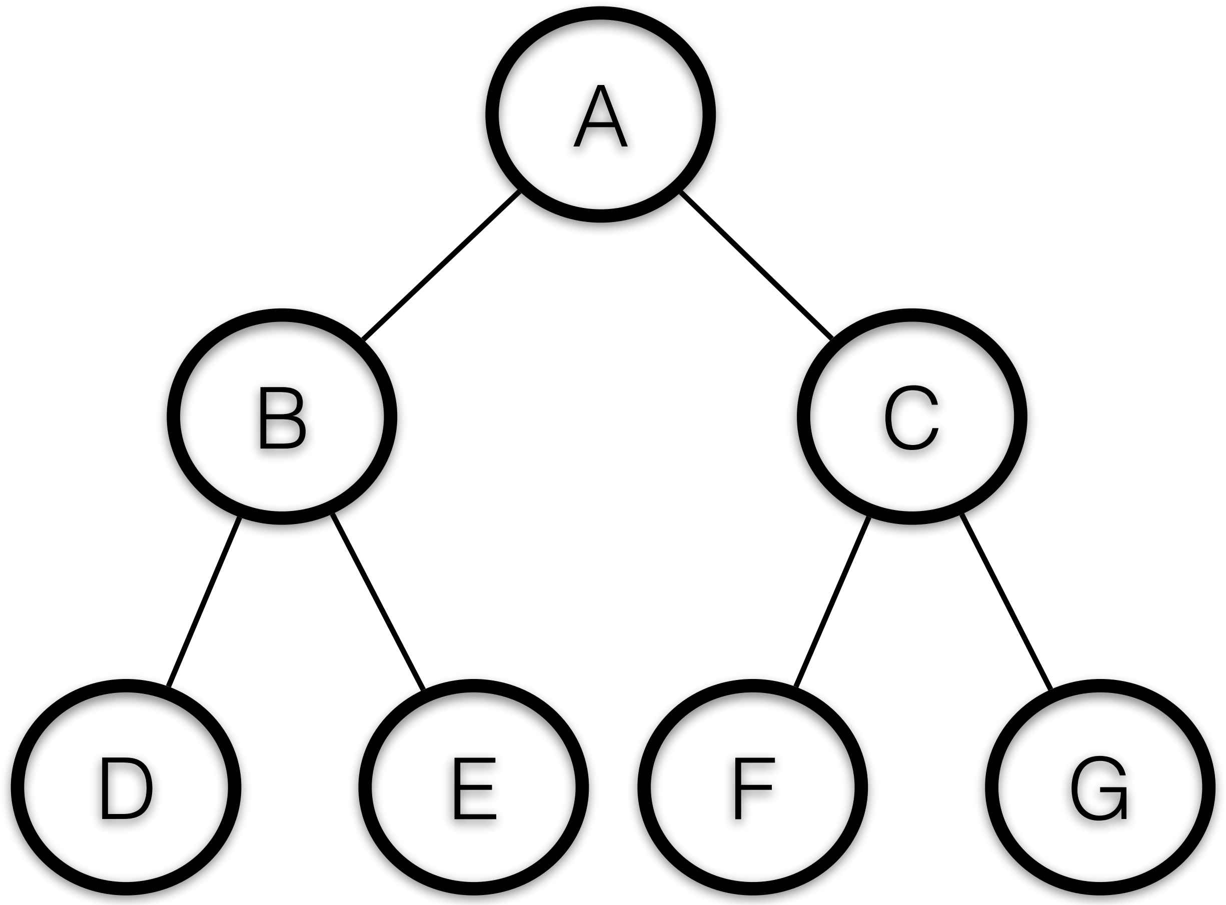
```
class NodeKnapsack implements Node {

    int[] value;
    int[] weight;
    int selectedValue;
    int capaLeft;
    int index;
    boolean selected;
    NodeKnapsack parent;
    double ub;

    public boolean isFeasible() {
        return index == value.length - 1;
    }

   @Override
   public List<Node> children() {

       List<Node> children = new ArrayList<>();
       // do not select item at index+1
       Node right = new NodeKnapsack(this, value, weight,
               selectedValue,
               capaLeft,
               index + 1, false);
       children.add(right);
       if (capaLeft >= weight[index+1]) {
           // select item at index+1
           Node left = new NodeKnapsack(this, value, weight,
                   selectedValue + value[index + 1],
                   capaLeft - weight[index + 1],
                   index + 1, true);
           children.add(left);
       }
       return children;
   }
}
```

index	0	1	2	3	4
v	28	22	18	6	1
w	7	6	5	2	1

Two different search strategies

- Best-First Search
  - Process first the promising nodes (i.e. with the best upper-bound)
  - This strategy is generally very good when you have a good upper-bounding procedure
  - Drawback: you don’t really have a control on the number of open-nodes (be careful with the memory you consume). In the worst-case you have a breadth first search
- Depth-First Search
  - Process first the deepest and left-most node.
  - Drawback: maybe-less good for proving optimality and to discover quickly a good first feasible solution
  - Advantage: Memory proportional to the height of the search tree (typically linear)
- Hybrid: Start with Depth-First to find a good feasible solution then continue with Best-First

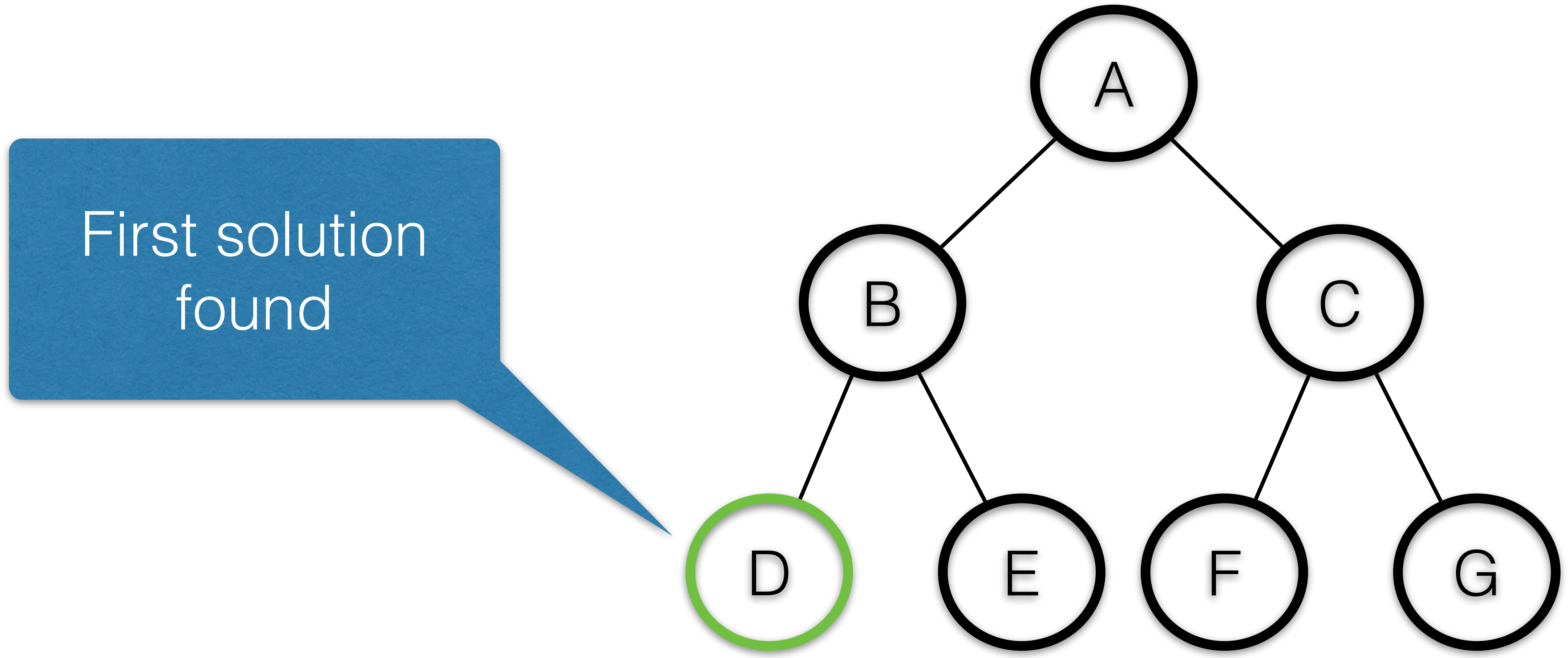
BFS vs DFS



- B(breadth) FS
- Current = A, Queue = [B,C]
- Current = B, Queue = [C,D,E]
- Current = C, Queue = [D,E,F,G]

- DFS
- Current = A, Stack = [C,B]
- Current = B, Stack = [C,E,D]
- Current = D, Stack = [D,E]

DFS and Heuristics



- The first solution should look good (have a reasonable quality).
- Why ? Because it will help pruning with the B&B
- How to make it look good ?

Open Node Interfaces

```
interface OpenNodes<N extends Node> {
    void add(N n);
    N remove();
    boolean isEmpty();
    int size();
}
```

Collection with open nodes. Think about possible ways to implement DFS/BFS 🤔

```
class DepthFirstOpenNodes<N extends  Node> implements OpenNodes<N> {

    Stack<N> stack;

    DepthFirstOpenNodes() {
        stack = new Stack<N>();
    }
    public void add(N n) {
        stack.push(n);
    }
    public N remove() {
        return stack.pop();
    }
    @Override
    public boolean isEmpty() {
        return stack.isEmpty();
    }
    @Override
    public int size() {
        return stack.size();
    }
}
```

```
class BestFirstOpenNodes<N extends  Node> implements OpenNodes<N> {

    PriorityQueue<N> queue;

    BestFirstOpenNodes() {
        queue = new PriorityQueue<N>(new Comparator<Node>() {
            @Override
            public int compare(Node o1, Node o2) {
                double lb1 = o1.lowerBound();
                double lb2 = o2.lowerBound();
                if (lb1 < lb2) {
                    return -1;
                } else if (lb1 == lb2) {
                    return 0;
                } else {
                    return 1;
                }
            }
        });
    }
    public void add(N n) {
        queue.add(n);
    }
    public N remove() {
        return queue.remove();
    }
    @Override
    public boolean isEmpty() {
        return queue.isEmpty();
    }
    @Override
    public int size() {
        return queue.size();
    }
}
```

Start the Knapasack

```
public static void main(String[] args) {

    int[] value = new int[]{1, 6, 18, 22, 28};
    int[] weight = new int[]{2, 3, 5, 6, 7};
    int capa = 11;
    int n = value.length;


    OpenNodes<NodeKnapsack> openNodes = new BestFirstOpenNodes<>();
    //OpenNodes<NodeKnapsack> openNodes = new DepthFirstOpenNodes<>();

    NodeKnapsack root = new NodeKnapsack(null,value,weight,0,capa,-1,false);
    openNodes.add(root);
    BranchAndBound.minimize(openNodes);
}
```

Heuristic

- Very important for depth first search

- It is better to branch and include first the next item with the largest ration v/w. Simple sort prior to the search

```
val items = Array((1,1),(6,2),(18,5),(22,6),(28,7)) // (value,weight)
// sort items in increasing value/weight ration
scala.util.Sorting.quickSort(items)(Ordering.by {case (v, w) => w / w })
val c = 11

val root = new KnapsackNode(
  items = items,
  obj = 0,
  selected = Nil,
  capa = c,
  selectable = (0 until items.size).toList)


val bestSol = BnB.solve(new PQueue(root))
```

Branch & Bound Experimentation:

- Importance of relaxation
- Importance of queue implementation
- Importance of heuristic

Optimality Gap

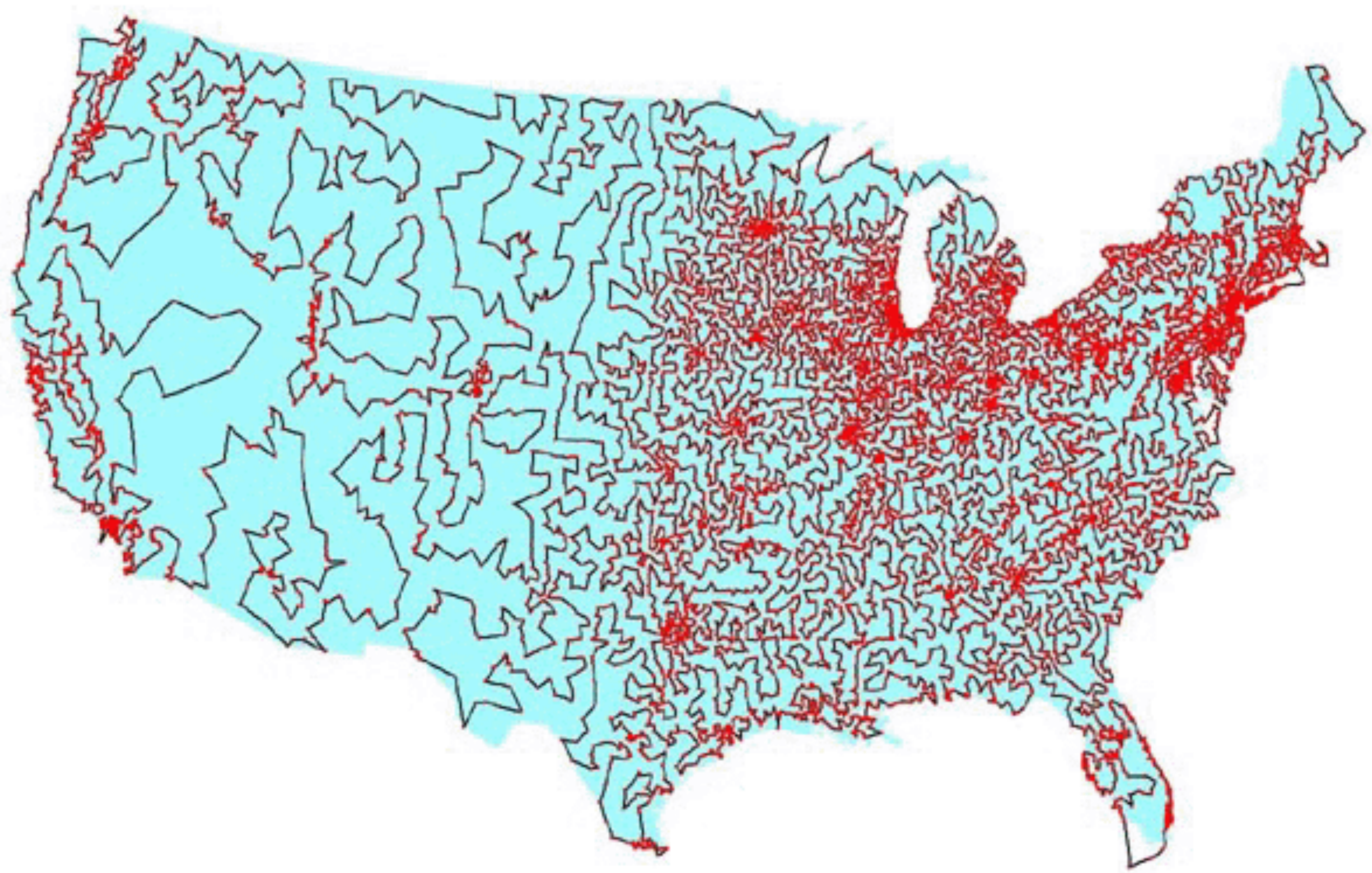
The Optimality Gap

- Can we provide some guarantee of how “sub-optimal” the best so far solution is ?
- Yes: This is the optimality gap.
- You should compute the most optimistic upper-bound (maximum upper-bound of all the open-nodes). Let us call it U.
- Gap = (U - bestObj) / bestObj
- Best-First-Search is better than Depth-First-Search to close the gap.

Other B&B Examples

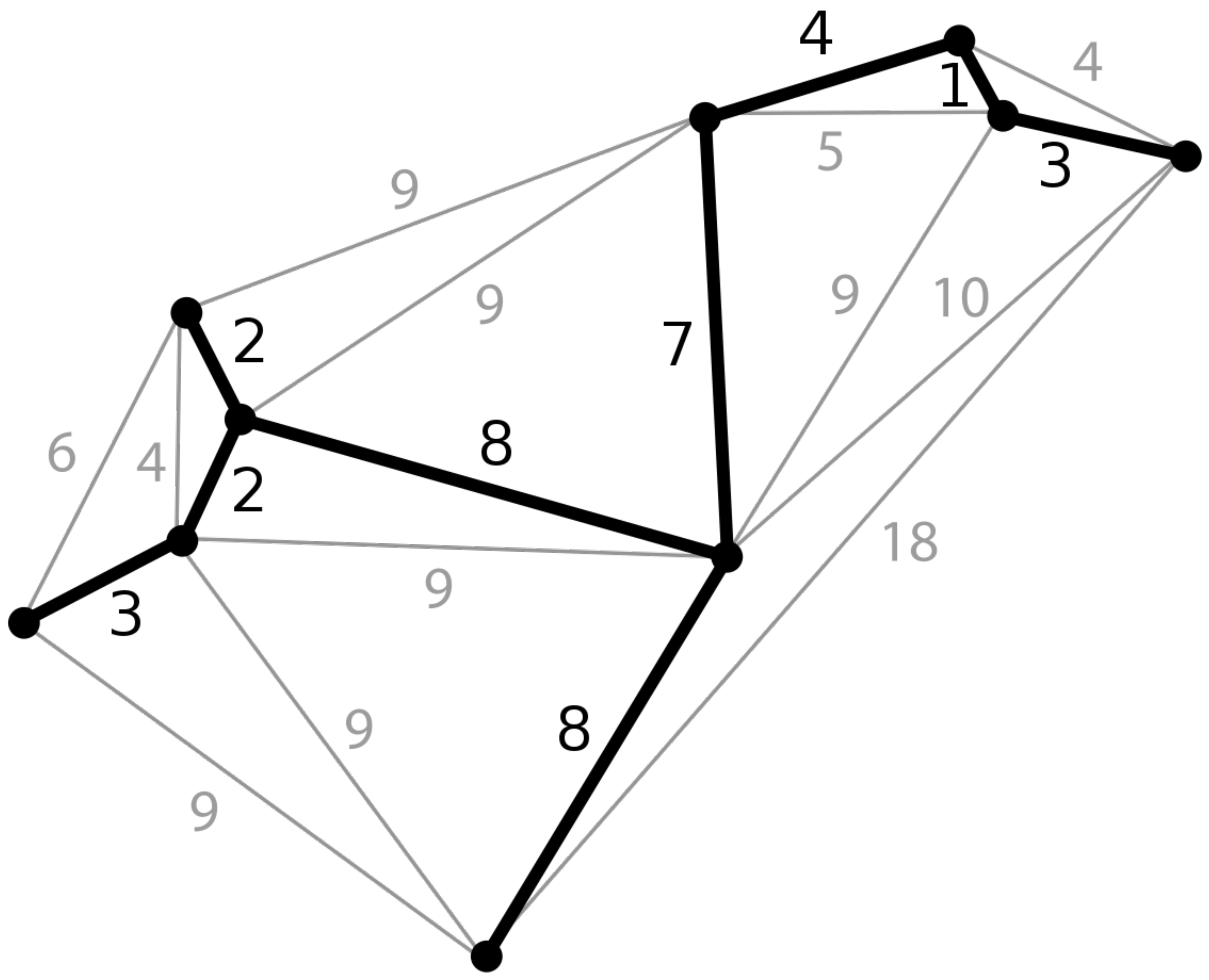
TSP (your project)

- Can you imagine a good lower-bound procedure for solving the TSP?
  - Hint: Think about relaxing « Hamiltonian tour » constraint.



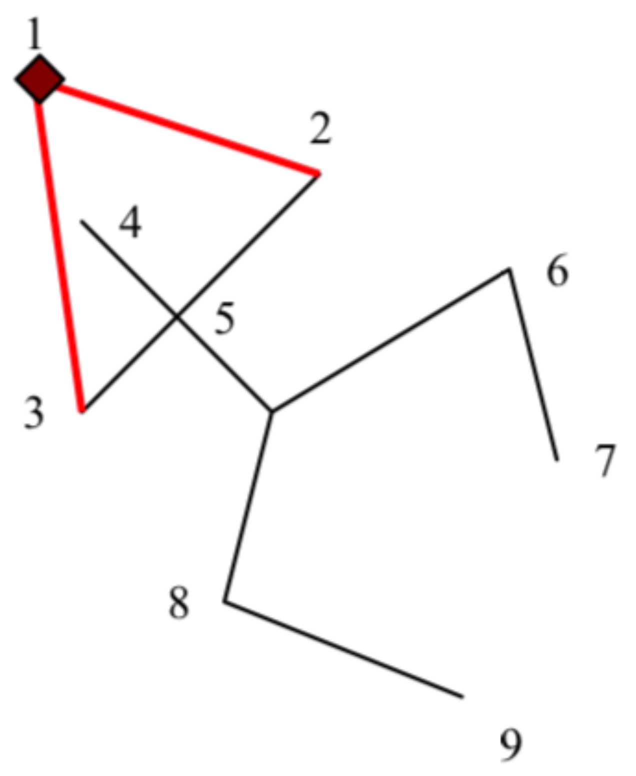
The Spanning Tree Relaxation

- Every tour is a tree (but not the opposite), hence the relaxation is to use the minimum spanning tree

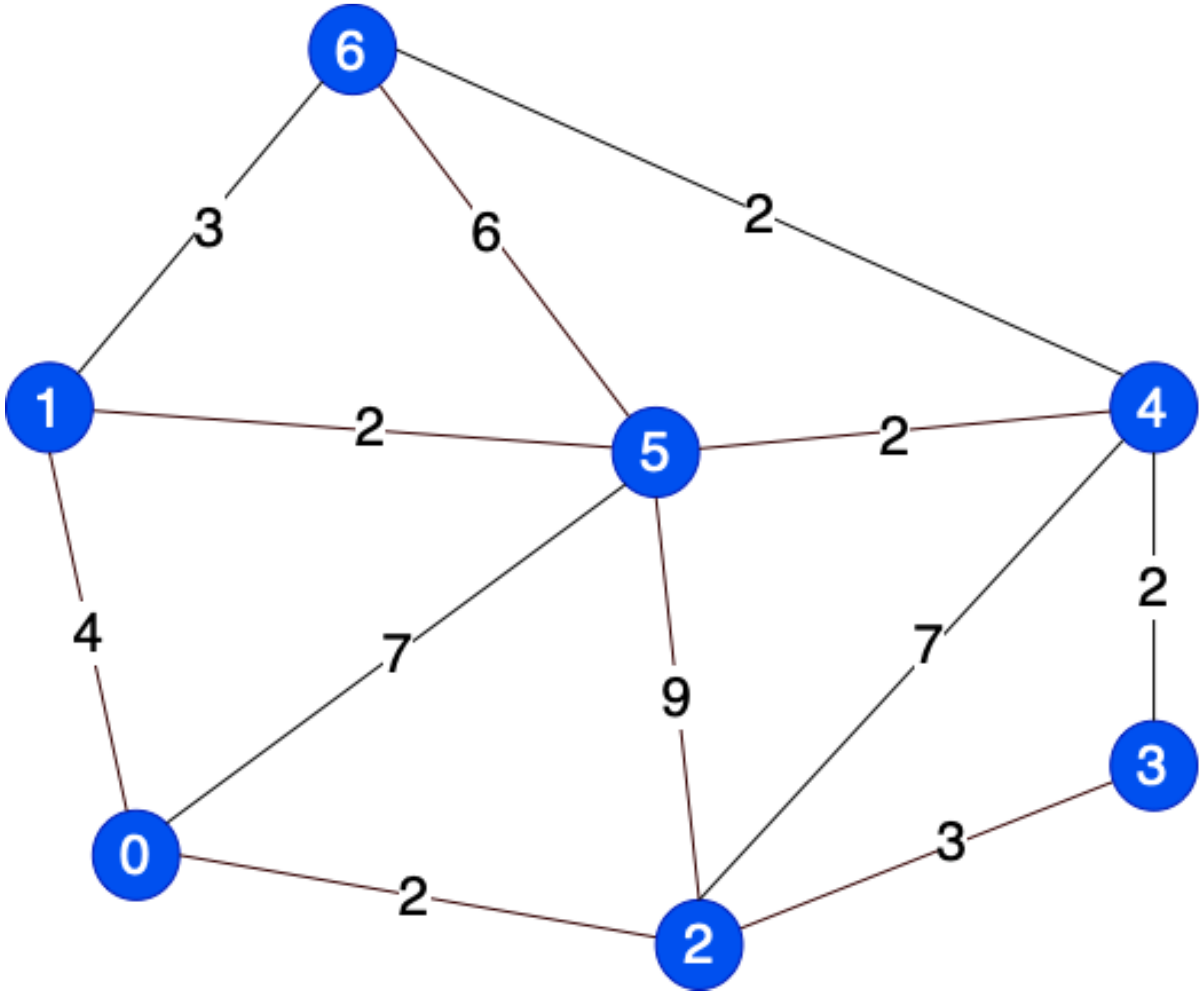


The One-Tree Relaxation

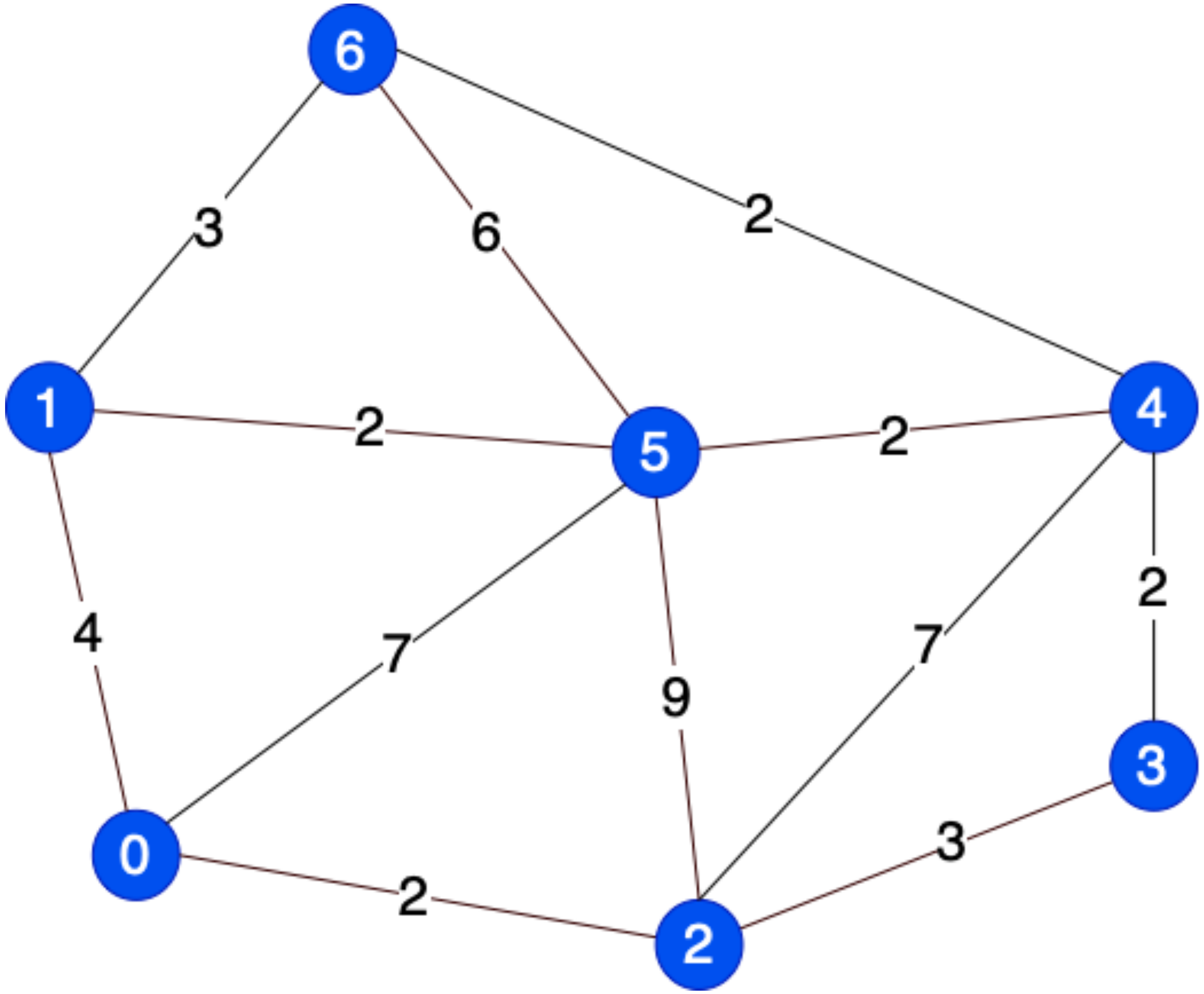
- For a given vertex, say vertex 1, a 1-Tree is a tree of {2,3,...n} +2 distinct edges connected to vertex 1.
- 1-Tree has precisely one cycle (stronger relaxation than the spanning tree since more constrained).
- Lower-Bound: To find minimum Weight 1- Tree, First Find minimum spanning tree of {2,3,...n} vertices, and add two lowest cost edges incident to vertex 1.



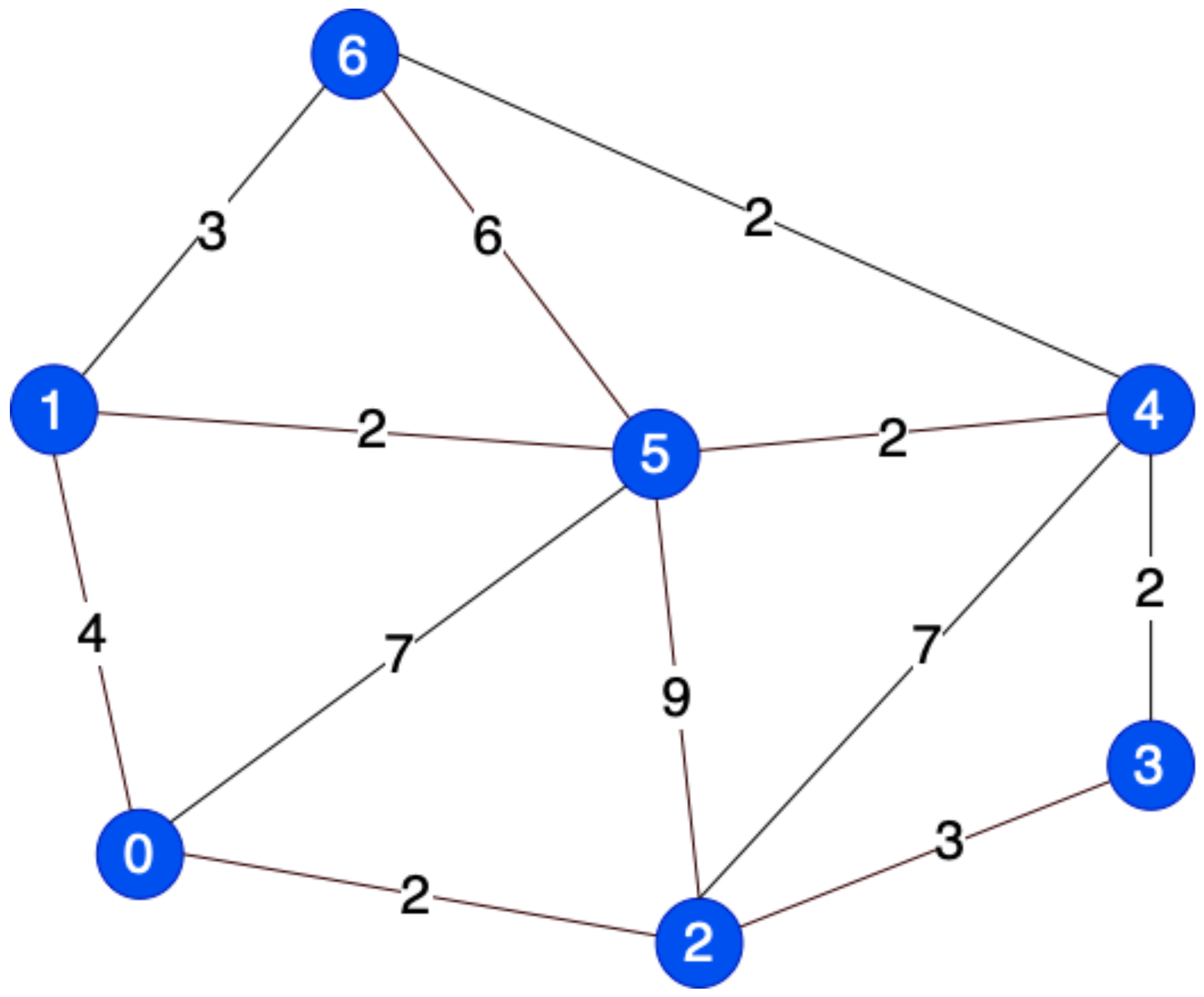
Spanning Tree Relaxation



One-Tree Relaxation



Branching for the TSP: Partial-Tour vs Excluded Edges



The Branch and Bound Project

Package "branchandbound"(don't forget to pull to get the latest update).

4 steps (impel) + Report:

1. Implement the simple one-tree based bound procedure in the `SimpleOneTree` class. You can test your result by executing `SimpleOneTreeTestFast`.

2. Implement the branch and bound for the TSP in the `BranchAndBoundTSP` class which will use the `SimpleOneTree` bound procedure you just implemented earlier. You can test your result by executing `BranchAndBoundTSPTestFast`.

3. (Next week) Implement an enhanced bound calculation for the one-tree based on Lagrangian relaxation in the `HeldKarpOneTree'' class. You can test your result by executing ``HeldKarpOneTreeFast`.

4. (Next week) Replace in your branch and bound for the TSP `BranchAndBoundTSP`, the bound calculation by your new reinforced bound. You can test your result by executing `BranchAndBoundTSPTest`.

Combinatorial Optimization is the art of relaxing

- Although we treat with NP-Hard problems:

- Solving them with Branch and Bound requires good (ie. tight) upper/lower bounds for maximization/minimization.

- Bound computation must be fast (most often using well known polynomial algorithms).

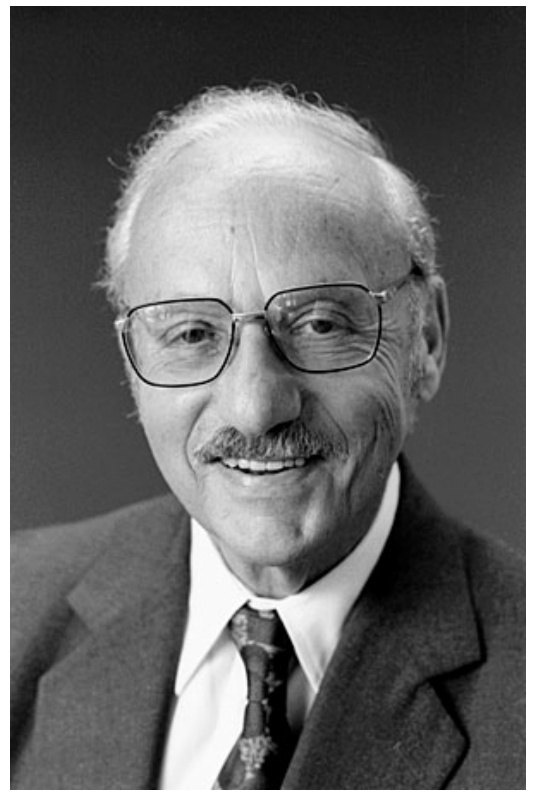
- Optimization is strongly related to algorithms and implementation.

History

Richard E. Bellman

1920-1984

Dynamic Programming 1950's



Georges Dantzig

1914-2005

Knapsack Relaxation 1957

Ailsa Land & Alison (Doig) Harcourt

Branch and Bound 1960