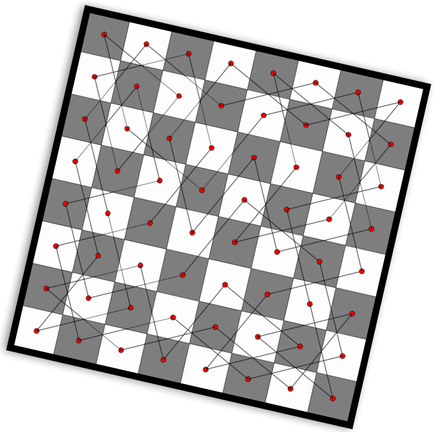
Advanced Algorithms for Optimization

LINFO2266

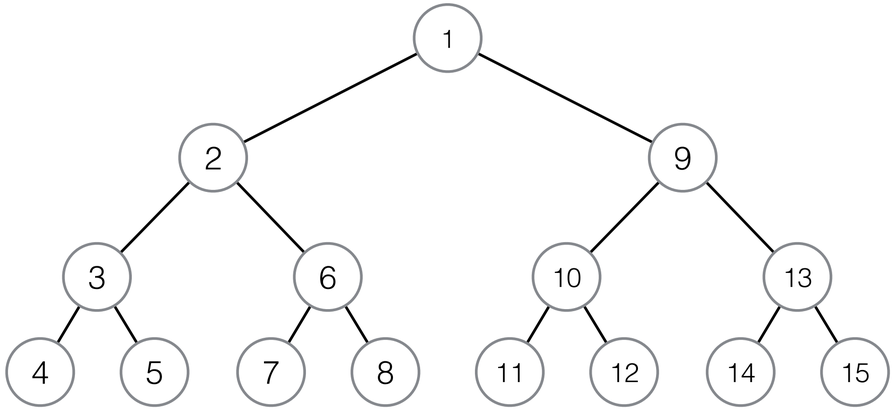
Branch & Bound

Pierre Schaus

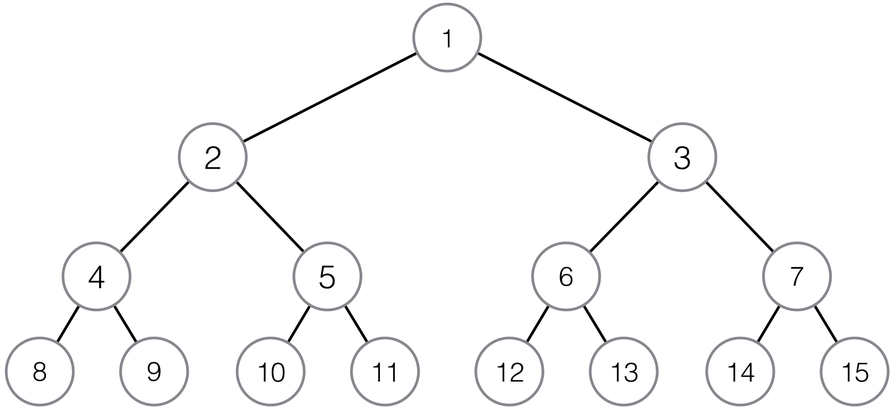


Reminder: DFS vs BFS for tree exploration

DSF

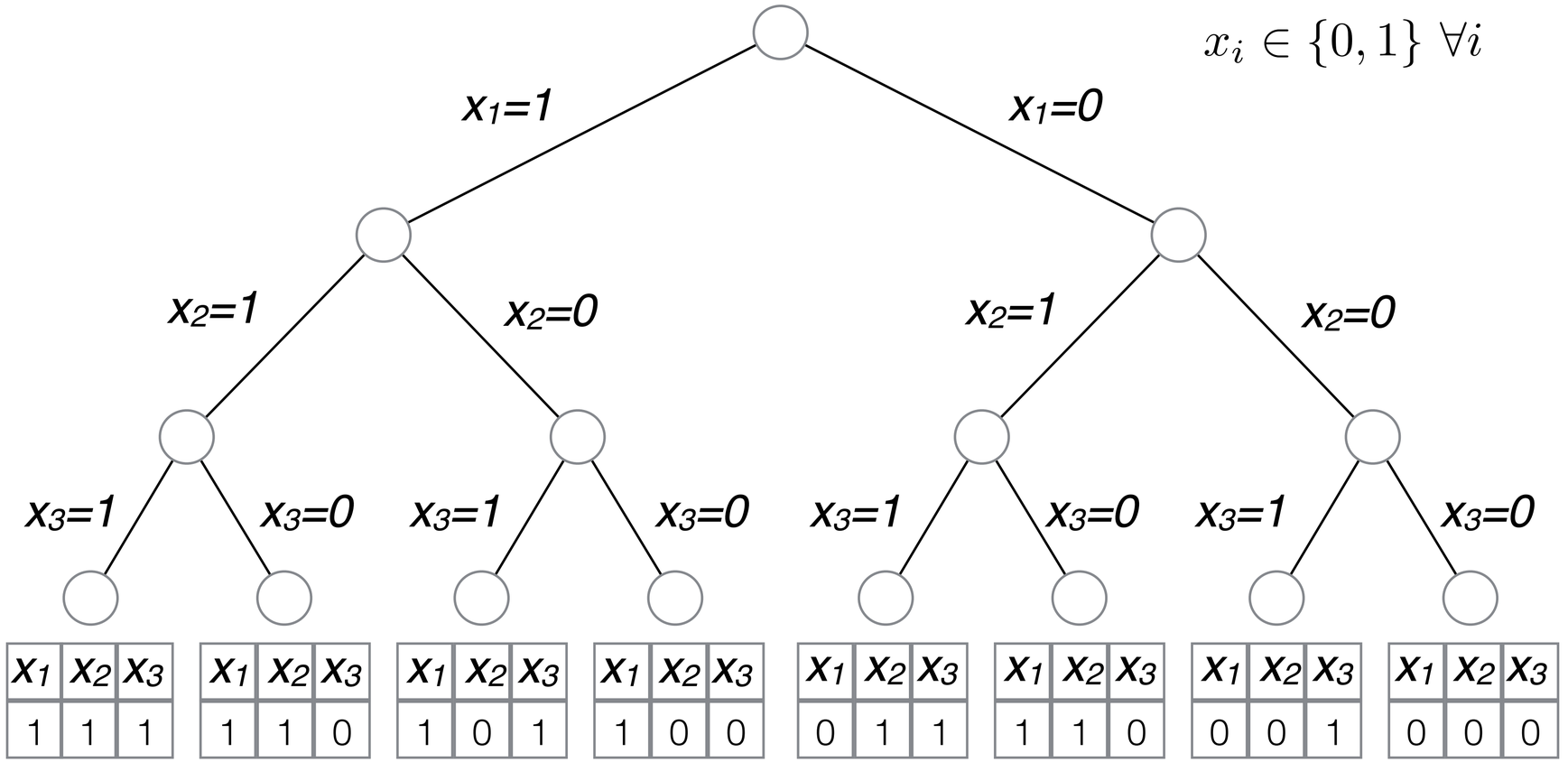


BFS



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

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



x_1	x_2	x_3
1	1	1

x_1	x_2	x_3
1	1	0

x_1	x_2	x_3
1	0	1

x_1	x_2	x_3
1	0	0

x_1	x_2	x_3
0	1	1

x_1	x_2	x_3
1	1	0

x_1	x_2	x_3
0	0	1

x_1	x_2	x_3
0	0	0

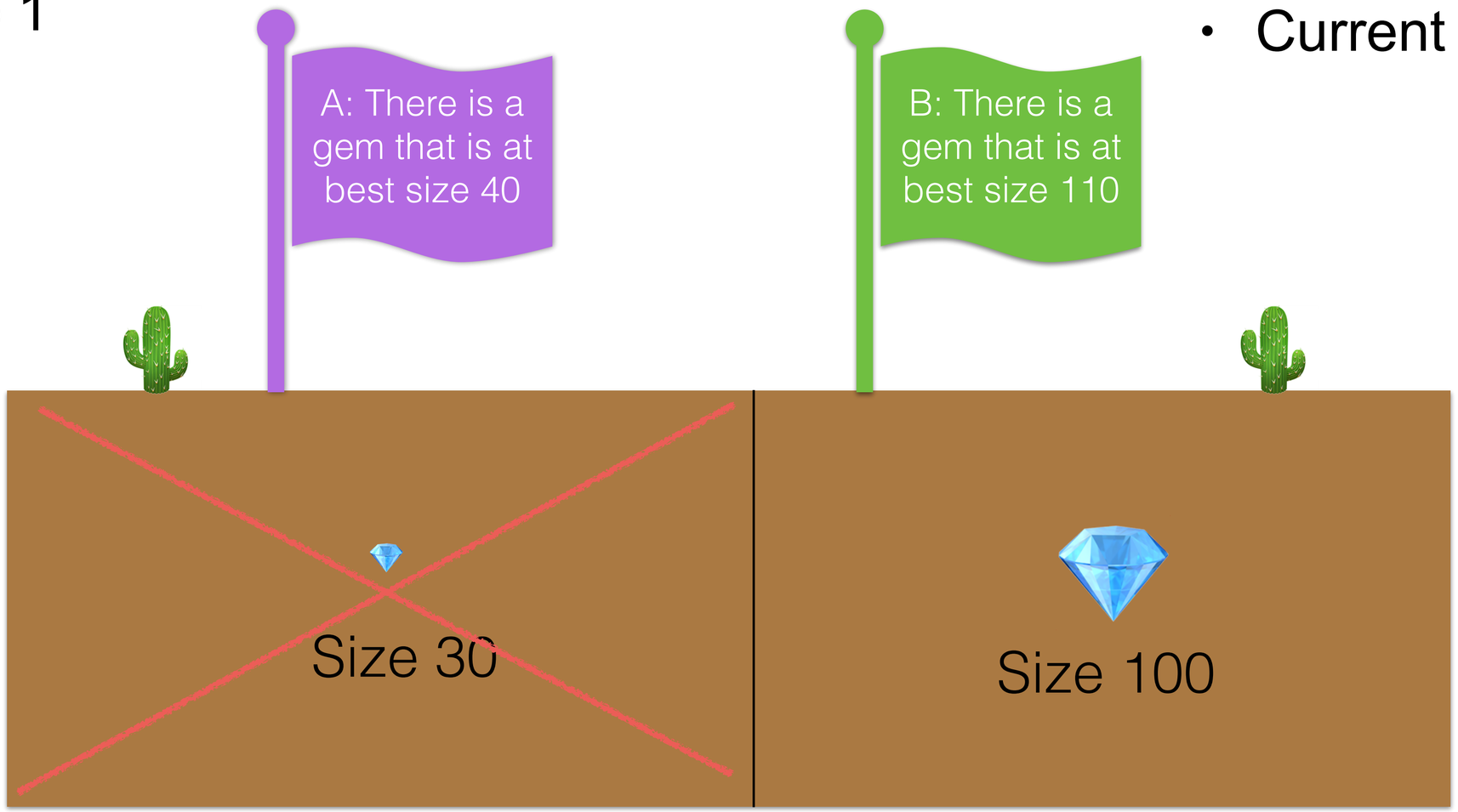
Can we reduce this search space by cutting some branches?

Branch and Bound: The general idea

- You are looking for the biggest possible Gem, you already found this one of size 50: 💎

Branch and Bound: Where to dig ?

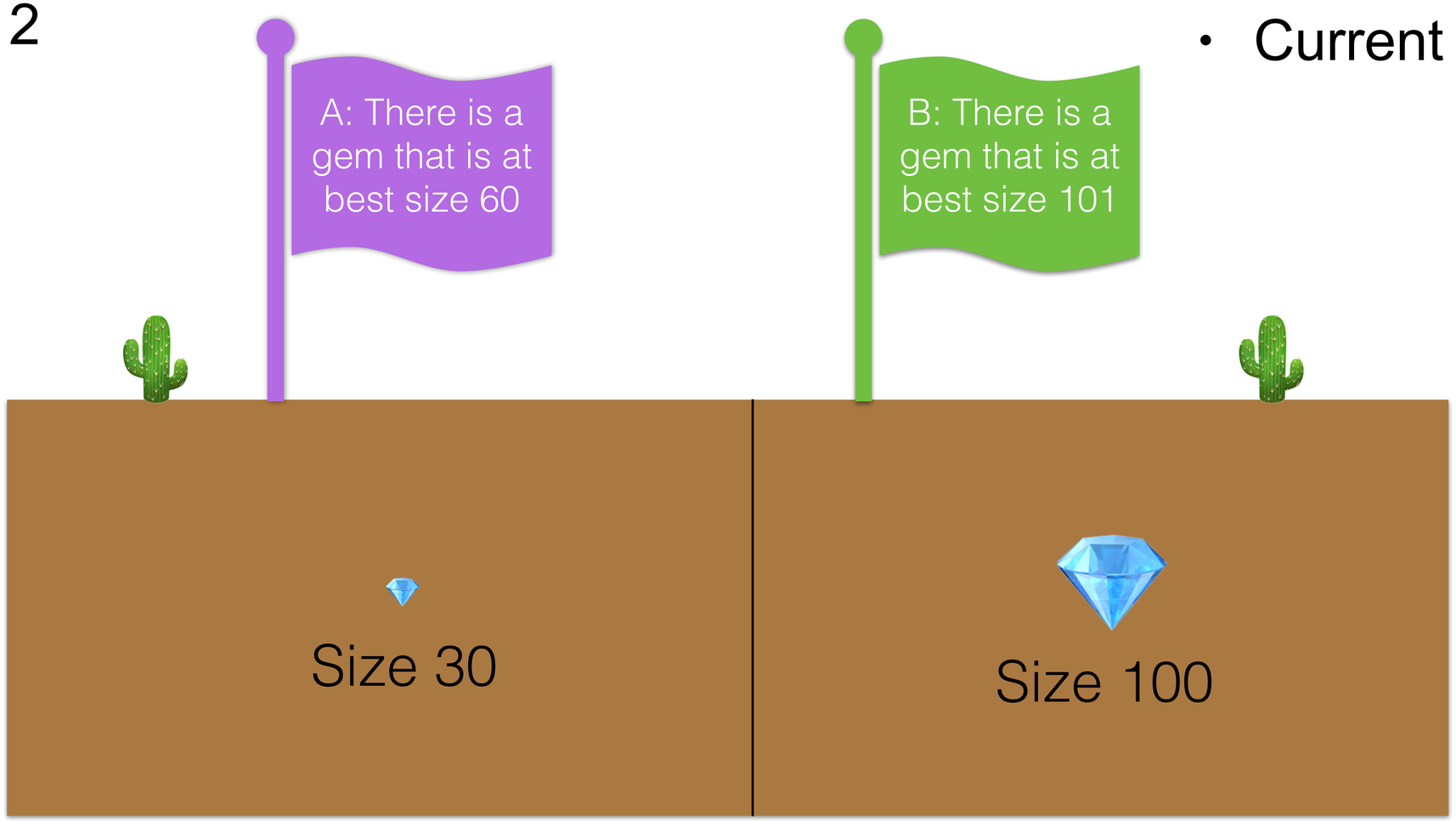
- Scenario 1



- Current gem size 50: 💎

Branch and Bound: Where to dig ?

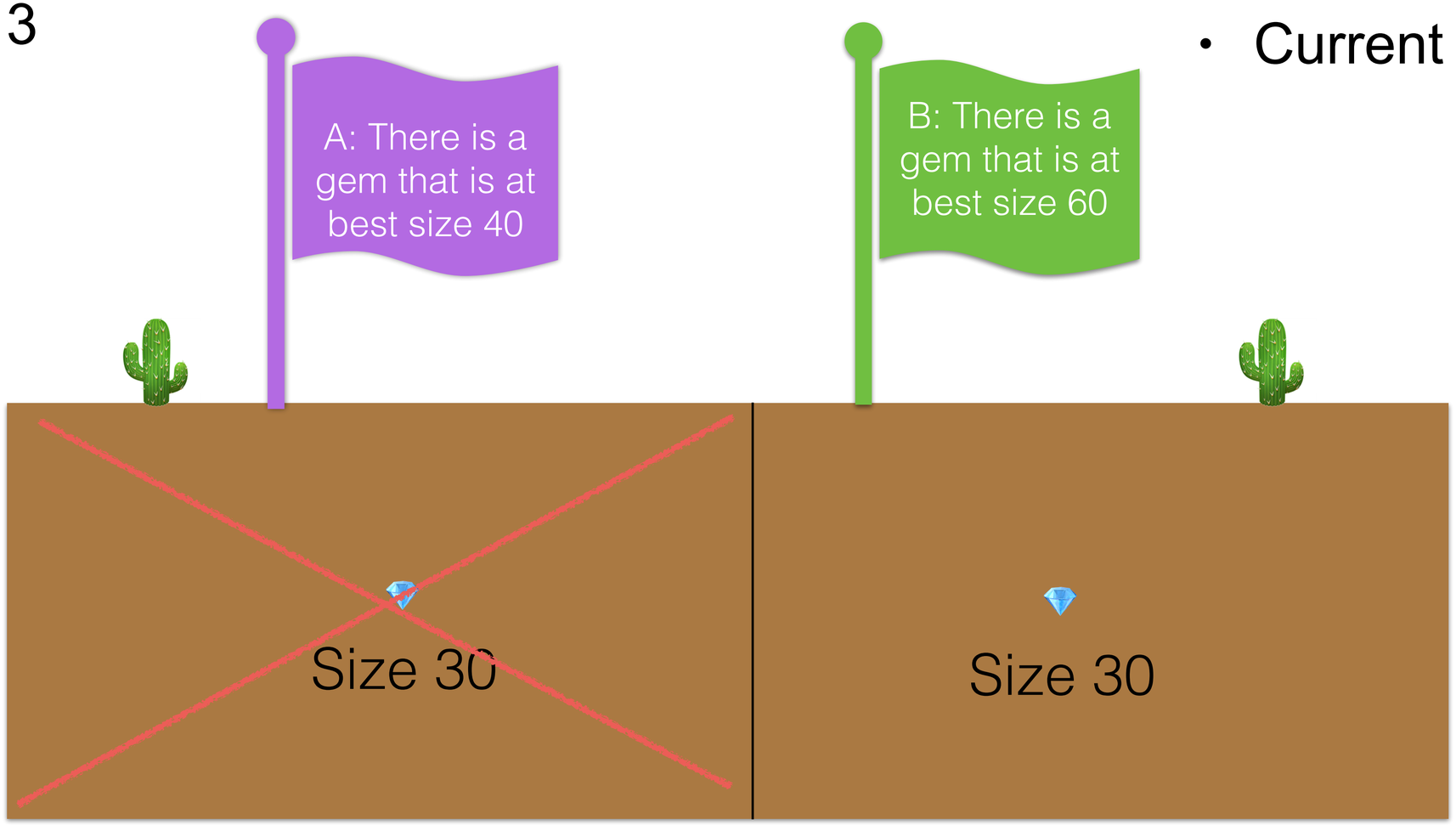
- Scenario 2



- Current gem size 50: 💎

Branch and Bound: Where to dig ?

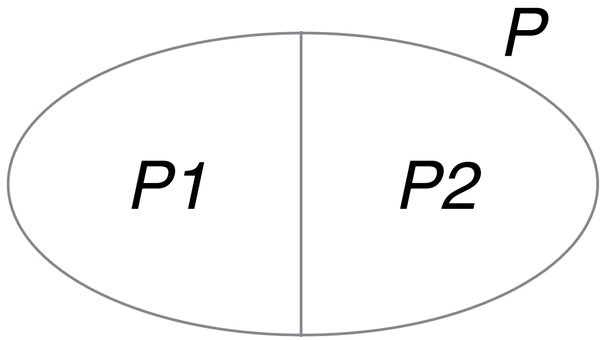
- Scenario 3



- Current gem size 50: 💎

Branch & Bound

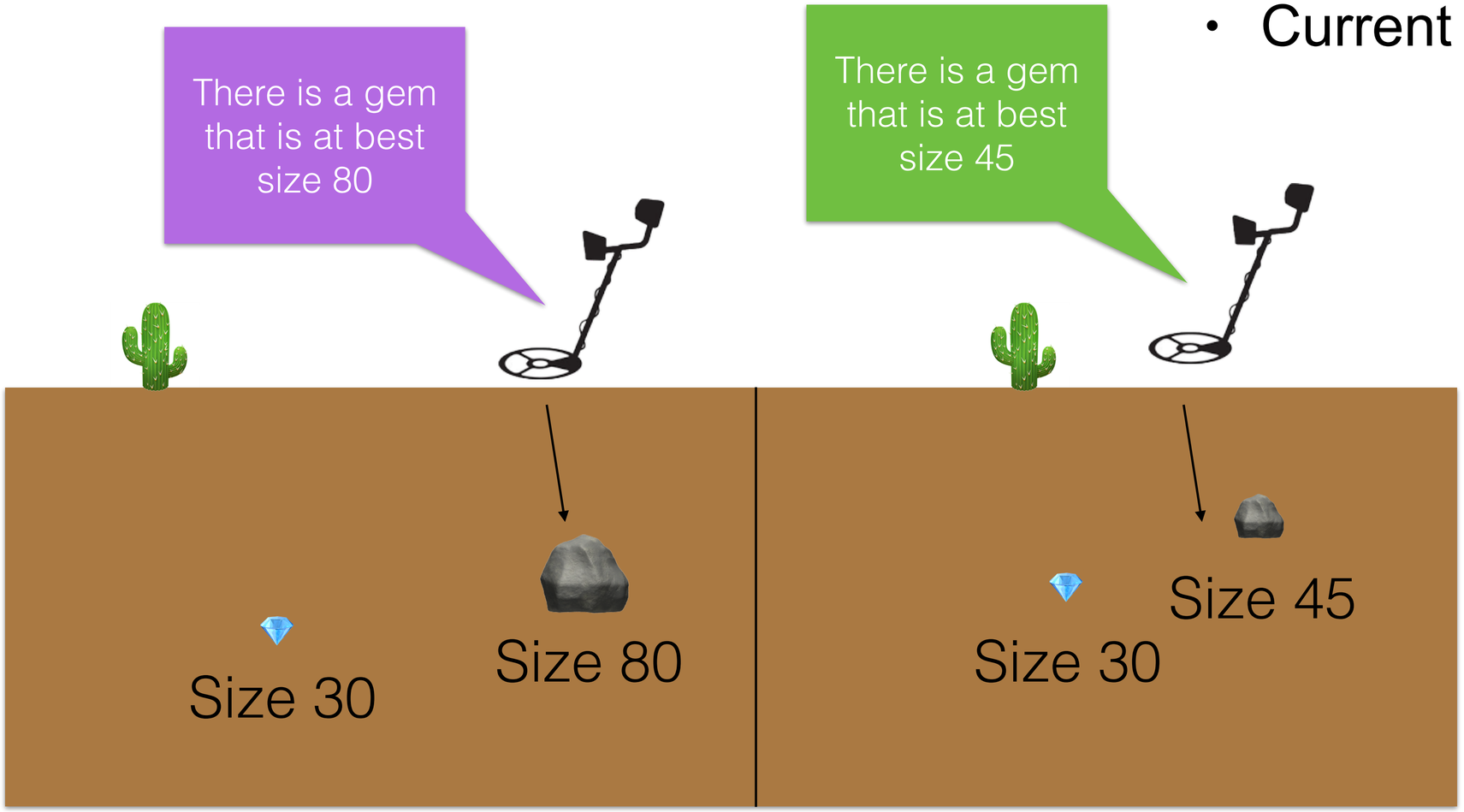
- Maximization Problem *P* : *maximize obj*
- You have a feasible solution in hand with objective *obj**
- You can decompose the problem: *P = P1* ∪ *P2*



- You have an upper-bound procedure *UB(P1) and UB(P2)*
- If *UB(P1)* ≤ *obj**, I can discard exploration of *P1* (idem for *P2*)

How to obtain an upper-bound ?

- With a rapid method that optimistically evaluate the possible value



- Current gem size 50: 💎

Knapsack: How to obtain an Upper-Bound

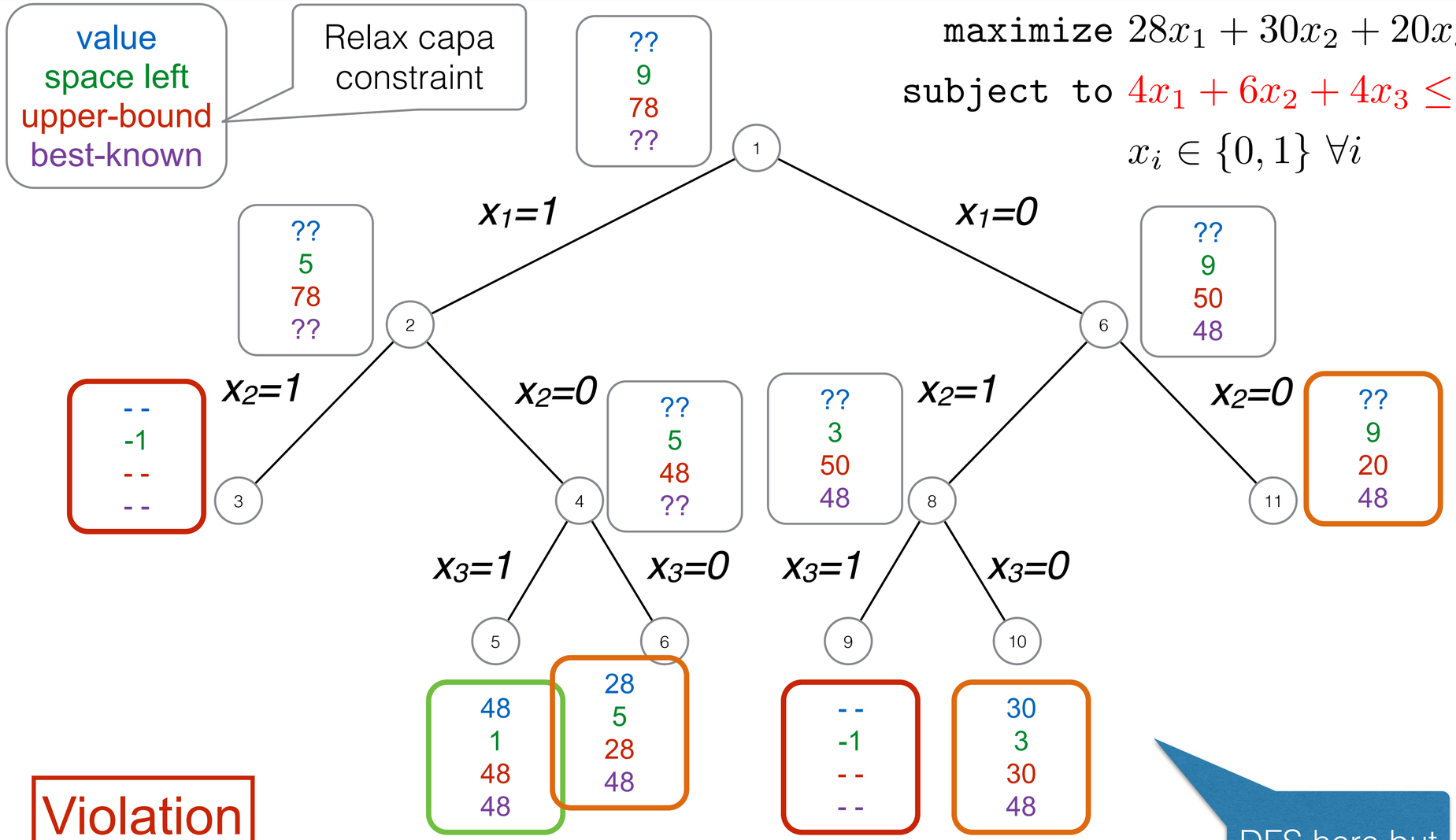
- For a constrained maximization problem, an upper-bound can be obtained by dropping some constraints. Let's relax the capacity

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

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

$$x_i \in \{0, 1\} \ \forall i$$

Branch & Bound: Capa relaxation (DFS Order)



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

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

$$x_i \in \{0, 1\}\ \forall i$$

Violation

Solution

Dominated

DFS here but we could use another strategy

Knapsack: How to obtain an Upper-Bound

- Another possibility is to relax the integrality constraint

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

- Rmq: when all the objective and the constraints are linear, this is called the *linear programming relaxation*

Knapsack - Linear Relaxation

- How to solve this problem efficiently ?

$$\text{maximize} \sum_i v_i \cdot x_i$$

$$\text{subject to} \sum_i w_i \cdot x_i \leq C$$

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

Knapsack - Linear Relaxation = Easy problem

- Sort the items according to ratio v_i/w_i (we assume it is the case)
- Find the critical item index $j = \min\{i \in I : \sum_{k \in 1..i} w_k > C\}$

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

Linear Relaxation

- C=14 (Capa)

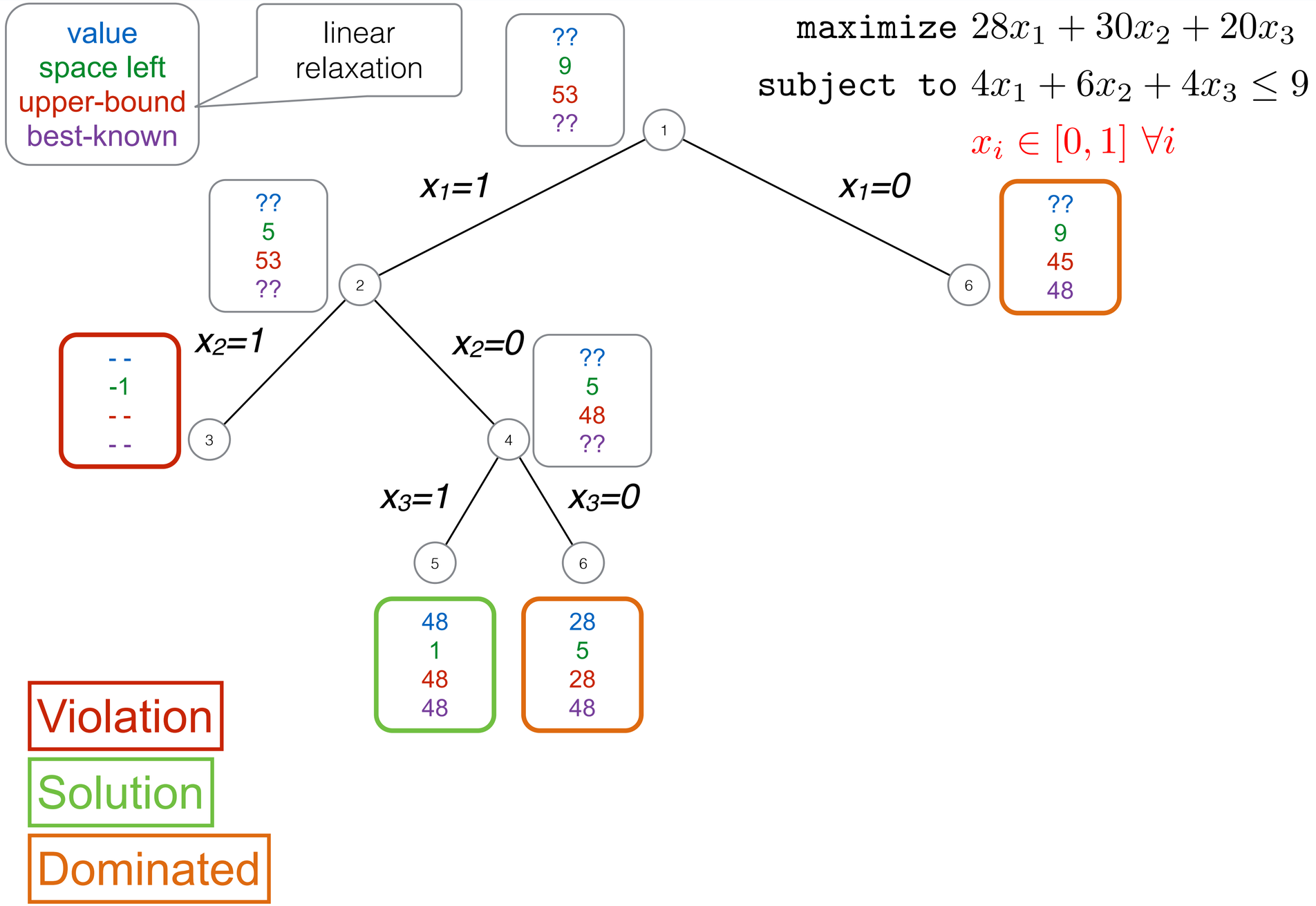
index	0	1	2	3	4
v/w	4	3.66	3.6	3	1
v	28	22	18	6	1
w	7	6	5	2	1
$\sum_{k\in 1..i} w_k$	7	13	18	20	21
x*	1	1	1/5	0	0

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

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

Why it works ?
Let i* = “critical item index”. Assume our solution x* with this method is not optimal but instead x’ is (hypothesis).
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/weight (this is contradiction, therefore x* is optimal)

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 <T> void minimize (OpenNodes<T> openNode, Consumer<Node<T>> onSolution) {
    double upperBound = Double.MAX_VALUE;
    int iter = 0;

    while (!openNode.isEmpty()) {
        iter++;
        Node<T> n = openNode.remove();
        if (n.isSolutionCandidate()) {
            double objective = n.objectiveFunction();
            if  (objective < upperBound) {
                upperBound = objective;
                onSolution.accept(n);
            }
        } else if (n.lowerBound() < upperBound) {
            for (Node<T> 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<T> {
    boolean isSolutionCandidate();
    double objectiveFunction();
    double lowerBound();
    int depth();
    List<Node<T>> children();
    T getState();
}
```

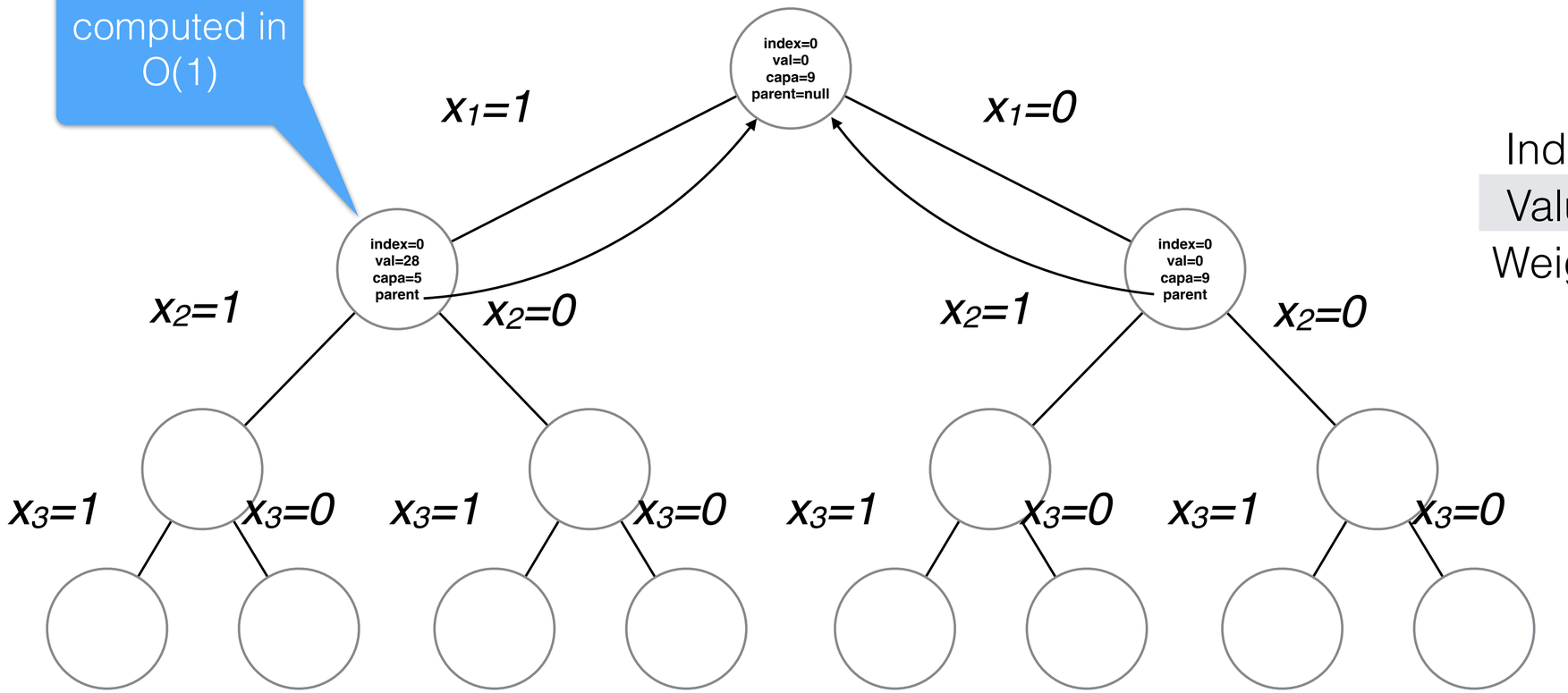
This implementation will be problem specific. A node contains the state of a problem modified according the “branching” decisions

```
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

State can be computed in O(1)



x1	x2	x3
1	1	1

x1	x2	x3
1	1	0

x1	x2	x3
1	0	1

x1	x2	x3
1	0	0

x1	x2	x3
0	1	1

x1	x2	x3
1	1	0

x1	x2	x3
0	0	1

x1	x2	x3
0	0	0

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

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

$$x_i \in \{0, 1\}\ \forall i$$

Index	0	1	2
Value	28	30	20
Weight	4	6	4

Knapsack Node Implementation

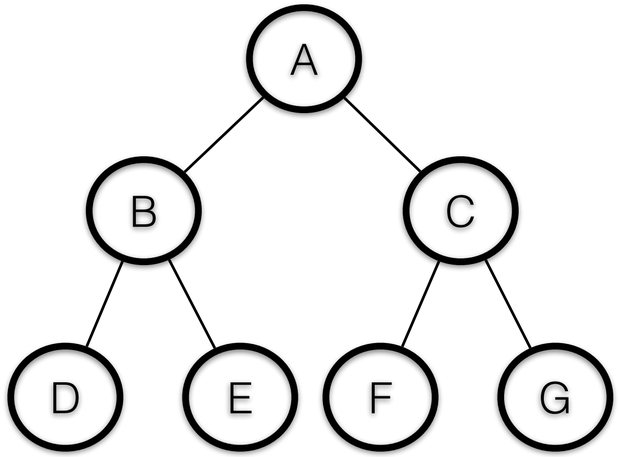
```
class NodeKnapsack implements Node {

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

    @Override
    public boolean isSolutionCandidate() {
        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
            int rightVal = selectedValue + value[index + 1];
            int rightCapa = capaLeft - weight[index + 1];
            Node left = new NodeKnapsack(this, value, weight, rightVal, rightCapa, 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

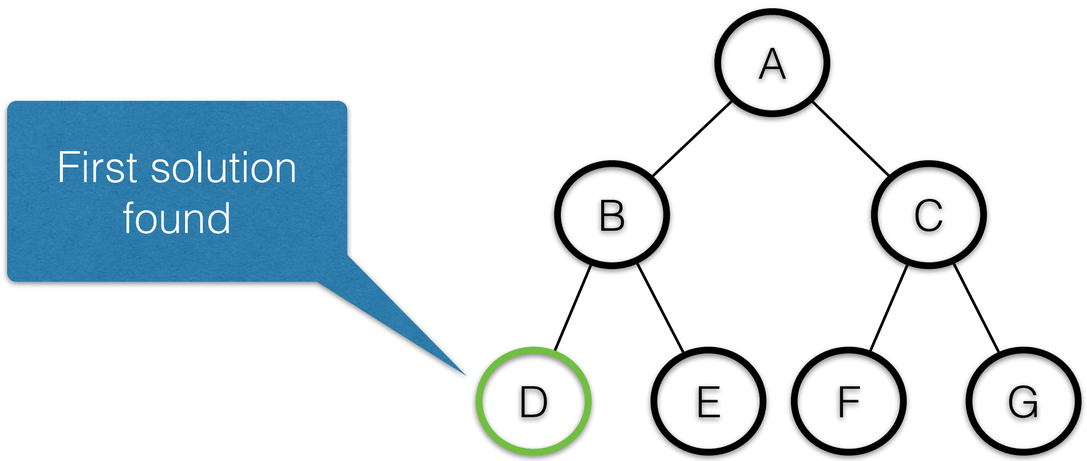
BFS vs DFS



- 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);
}
```

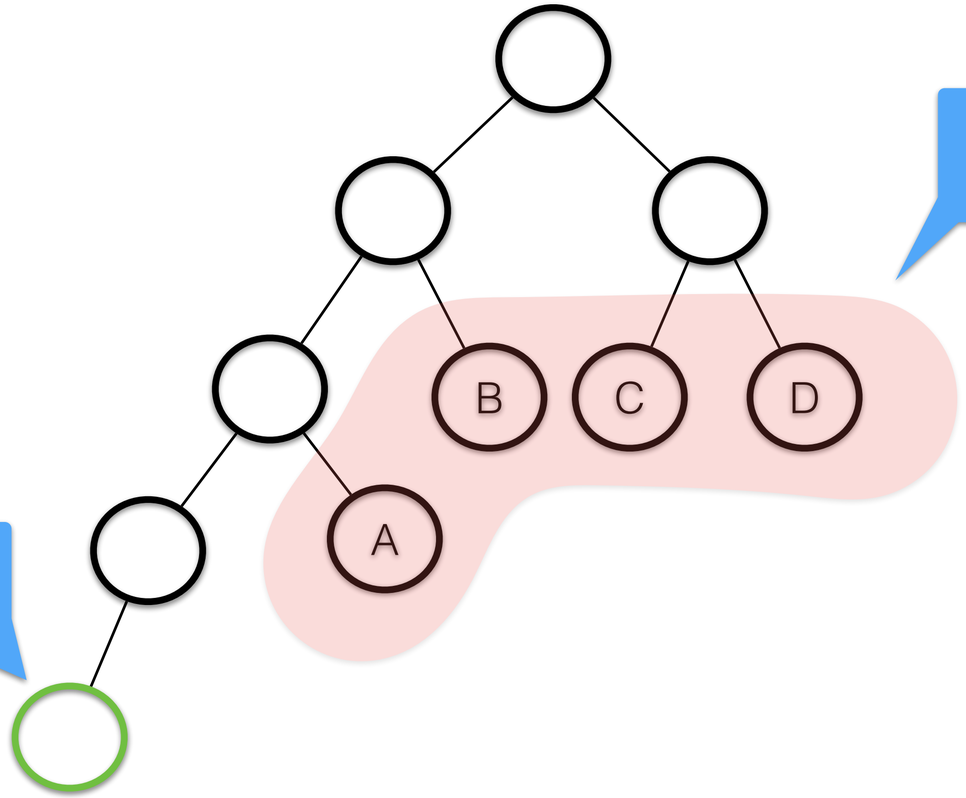
Branch & Bound Experimentation:

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

Optimality Gap

Let us assume this search tree explored so far

- Maximization Problem



A,B,C,D are the open nodes

Best solution = LB

Global Upper-Bound for the unexplored search space

$UB = \max(UB(A), UB(B), UB(C), UB(D))$

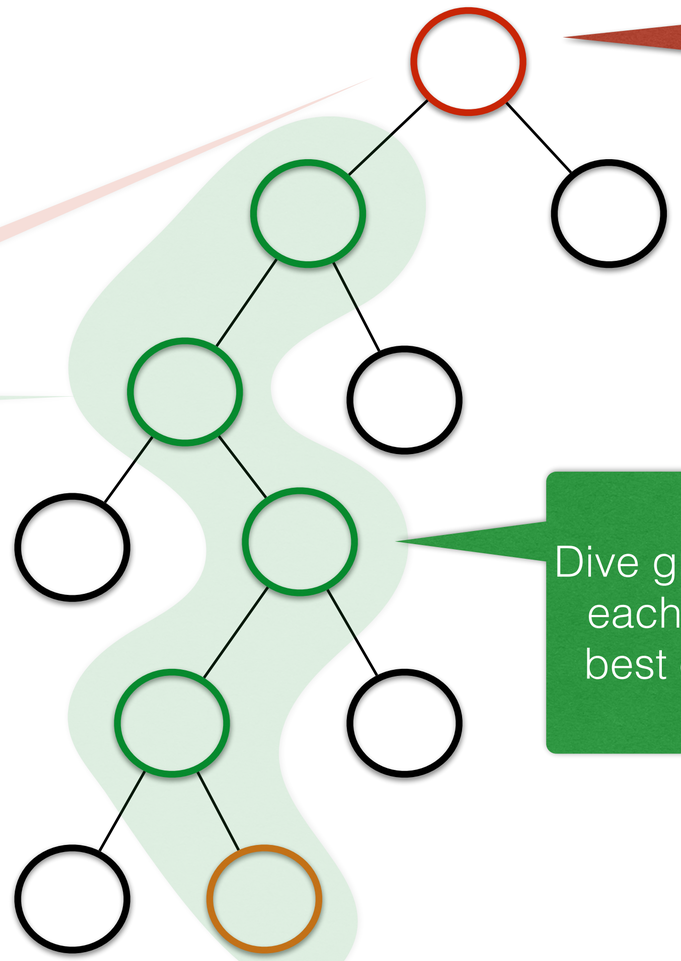
Optimality $gap = \dfrac{UB - LB}{LB}$

The Optimality Gap and Search

- DFS better than BFS (in general) to find rapidly a good LB (maximization)
- BFS is better than DFS (in general) to close the gap
- You do an hybridization of the two, any idea how to do it ?

Hybrid BFS, DFS

```
public static void minimizeHybridDFS-BFS(BestFirstOpenNodes bfsNodes, Consumer<Node> onSolution) {

    double upperBound = Double.MAX_VALUE;
    int iter = 0;
    PriorityQueue<Node> pqChildren = new PriorityQueue<>(new Comparator<Node>() {
        @Override
        public int compare(Node o1, Node o2) {
            return Double.compare(o1.lowerBound(), o2.lowerBound());
        }
    });

    while (!bfsNodes.isEmpty()) {
        iter++;
        Node n = bfsNodes.remove();
        // dive under node n until n becomes a leaf-node
        while (!n.isSolutionCandidate() && n.lowerBound() < upperBound) {
            pqChildren.clear();
            List<Node> children = n.children();
            pqChildren.addAll(children);
            n = pqChildren.poll();
            for (Node child: pqChildren){
                bfsNodes.add(child);
            }
        }

        if (n.isSolutionCandidate()) {
            double objective = n.objectiveFunction();
            if (objective < upperBound) {
                upperBound = objective;
                onSolution.accept(n);
            }
        } else if (n.lowerBound() < upperBound) {
            for (Node child : n.children()) {
                bfsNodes.add(child);
            }
        }
    }
    System.out.println("#iter:" + iter);
}
```



Node picked in bfsNodes

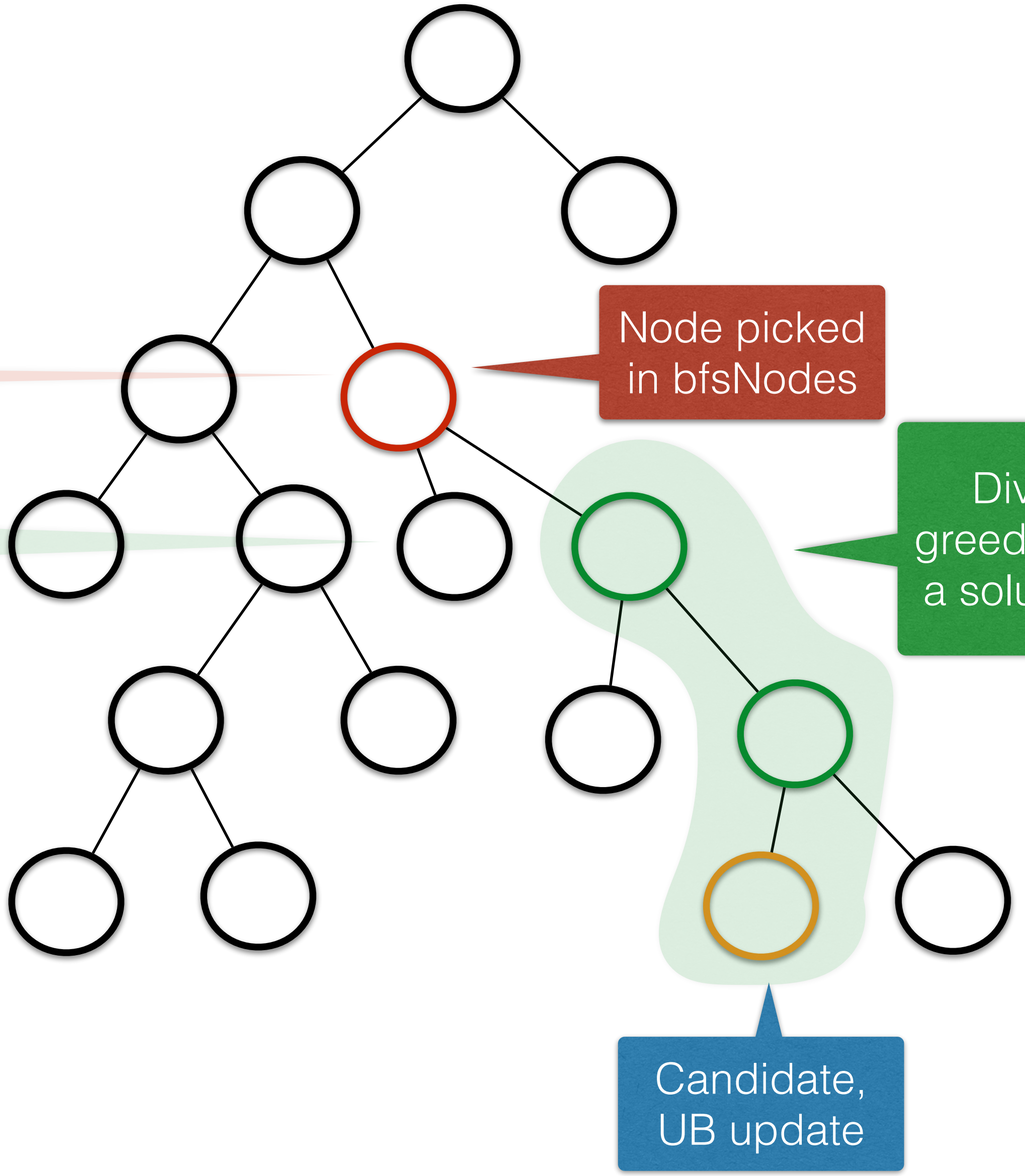
Those nodes are added to bfsNodes

Dive greedily to a solution, each time following the best of the two children

Candidate, UB update

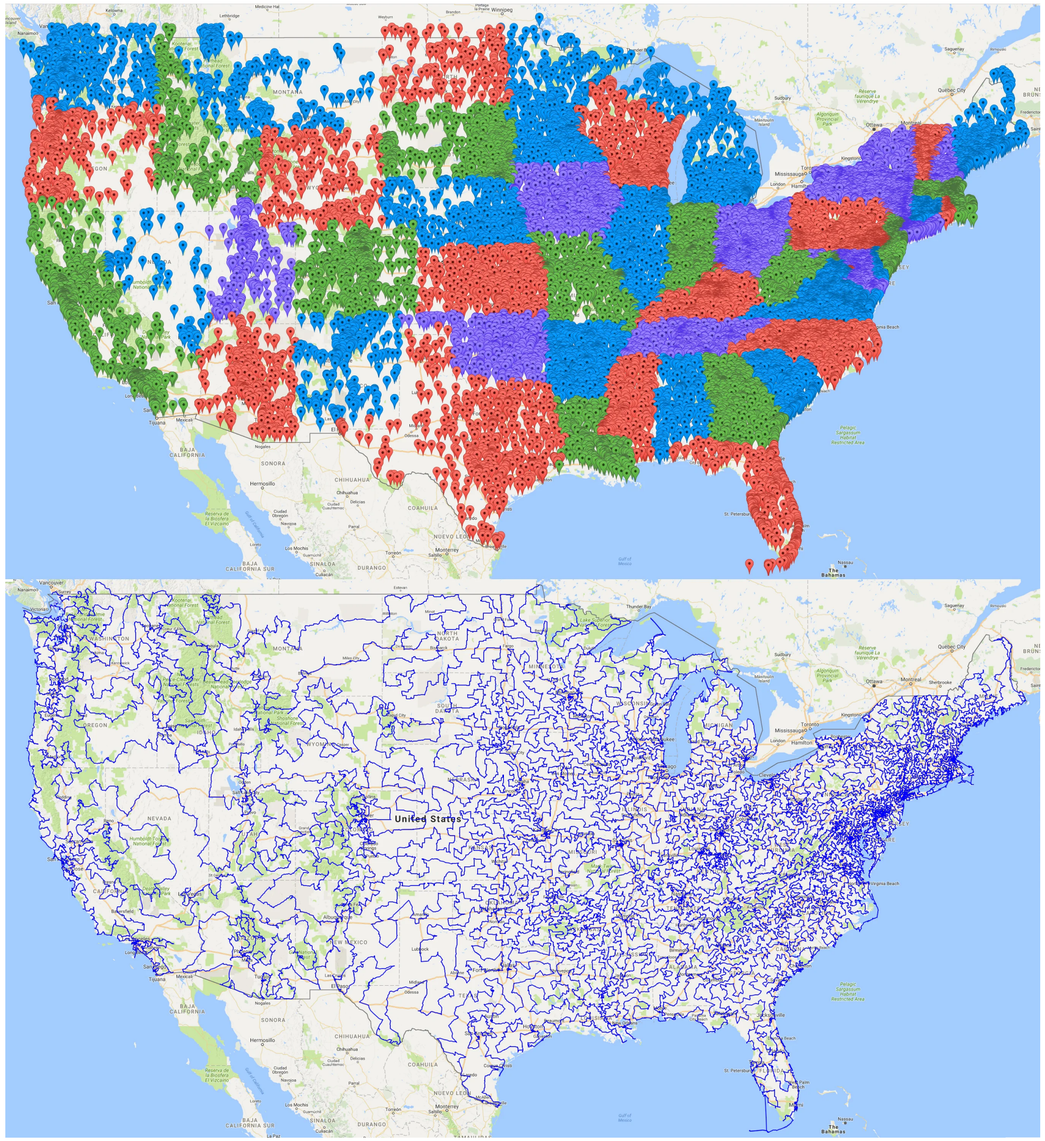
Hybrid BFS, DFS

```
public static void minimizeHybridDFS-BFS(BestFirstOpenNodes bfsNodes, Consumer<Node> onSolution) {

    double upperBound = Double.MAX_VALUE;
    int iter = 0;
    PriorityQueue<Node> pqChildren = new PriorityQueue<>(new Comparator<Node>() {
        @Override
        public int compare(Node o1, Node o2) {
            return Double.compare(o1.lowerBound(), o2.lowerBound());
        }
    });

    while (!bfsNodes.isEmpty()) {
        iter++;
        Node n = bfsNodes.remove();
        // dive under node n until n becomes a leaf-node
        while (!n.isSolutionCandidate() && n.lowerBound() < upperBound) {
            pqChildren.clear();
            List<Node> children = n.children();
            pqChildren.addAll(children);
            n = pqChildren.poll();
            for (Node child: pqChildren){
                bfsNodes.add(child);
            }
        }

        if (n.isSolutionCandidate()) {
            double objective = n.objectiveFunction();
            if (objective < upperBound) {
                upperBound = objective;
                onSolution.accept(n);
            }
        } else if (n.lowerBound() < upperBound) {
            for (Node child : n.children()) {
                bfsNodes.add(child);
            }
        }
    }
    System.out.println("#iter:" + iter);
}
```



B&B For the TSP

Traveling Salesman Problem



A TSP Model

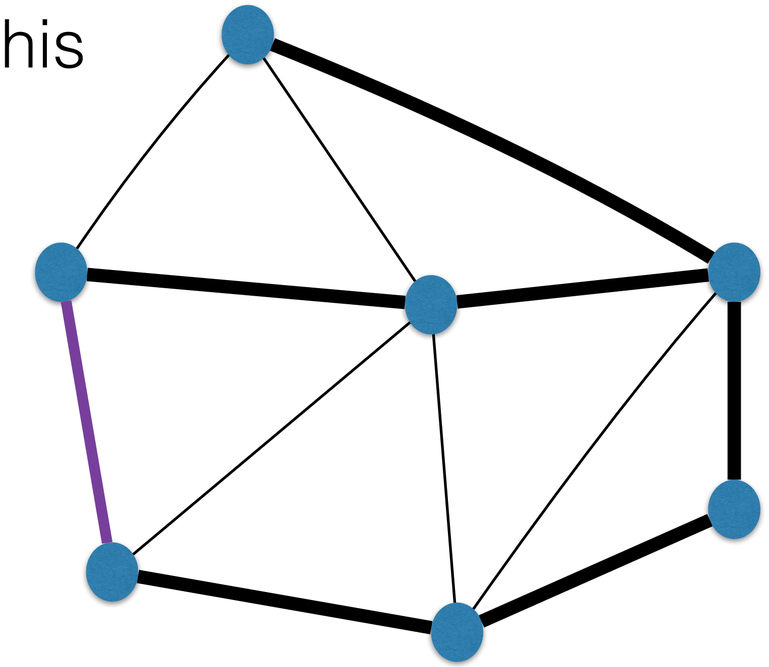
$$\min \sum_{e} x_e \cdot w_e$$

selected edges $\{e \mid x_e = 1\}$ form a spanning tree + 1 edge

$$\sum_{e \in \delta(i)} x_e = 2, \forall i$$

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

like this



TSP

- A TSP is a combination of two constraints:

$$\min \sum_{e} x_e \cdot w_e$$

selected edges $\{e \mid x_e = 1\}$ form a spanning tree + 1 edge

$$\sum_{e \in \delta(i)} x_e = 2, \forall i$$

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

- The two constraints can be relaxed, let us see how …

Min Spanning Tree Relaxation + Cheapest Edge

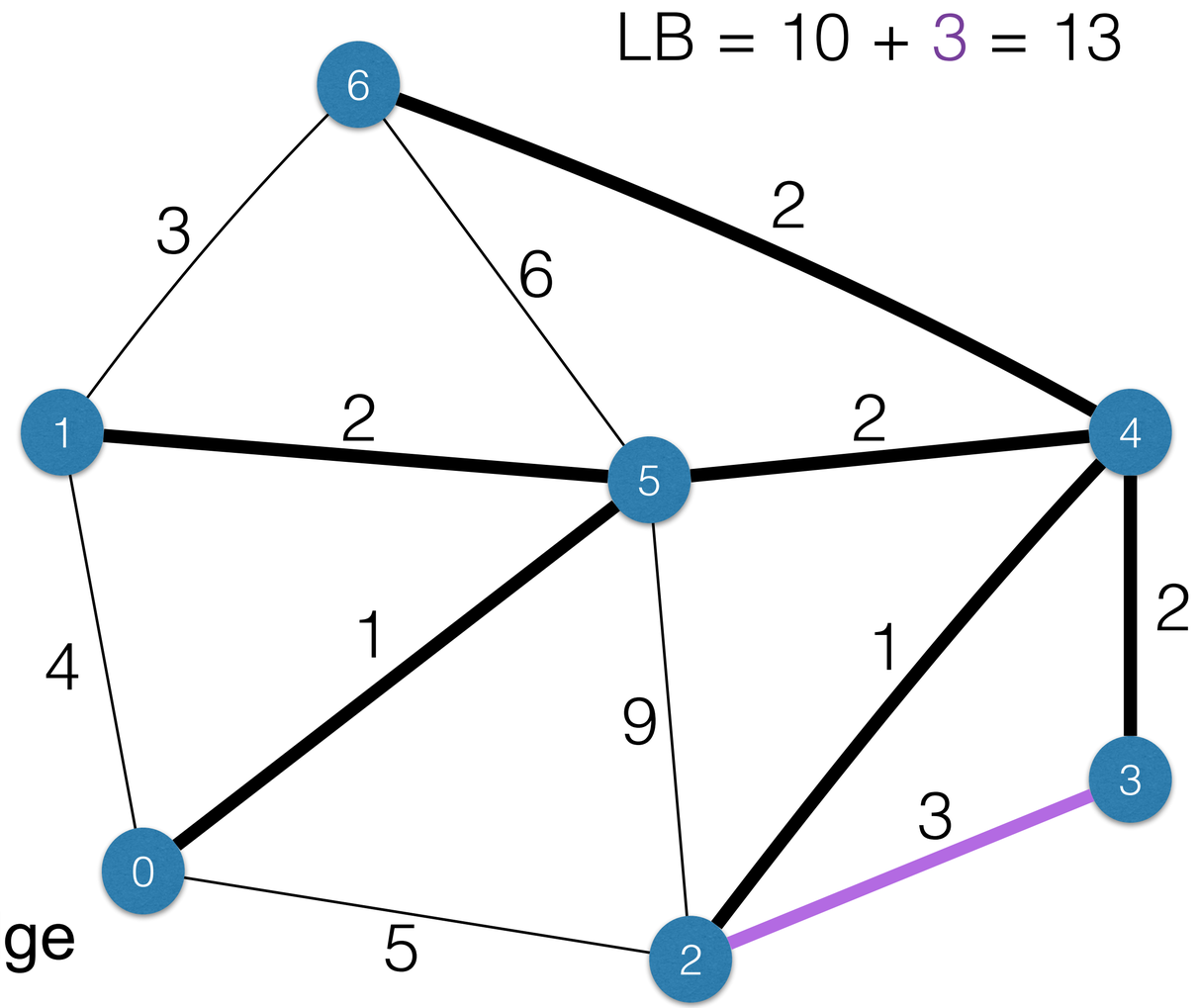
$$\min \sum_e x_e \cdot w_e$$

$\{e \mid x_e = 1\}$ = spanning tree + 1 edge

~~$\sum_{e \in \delta(i)} x_e = 2, \forall i$~~

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

= minimum spanning tree + cheapest edge



Slightly stronger relaxation

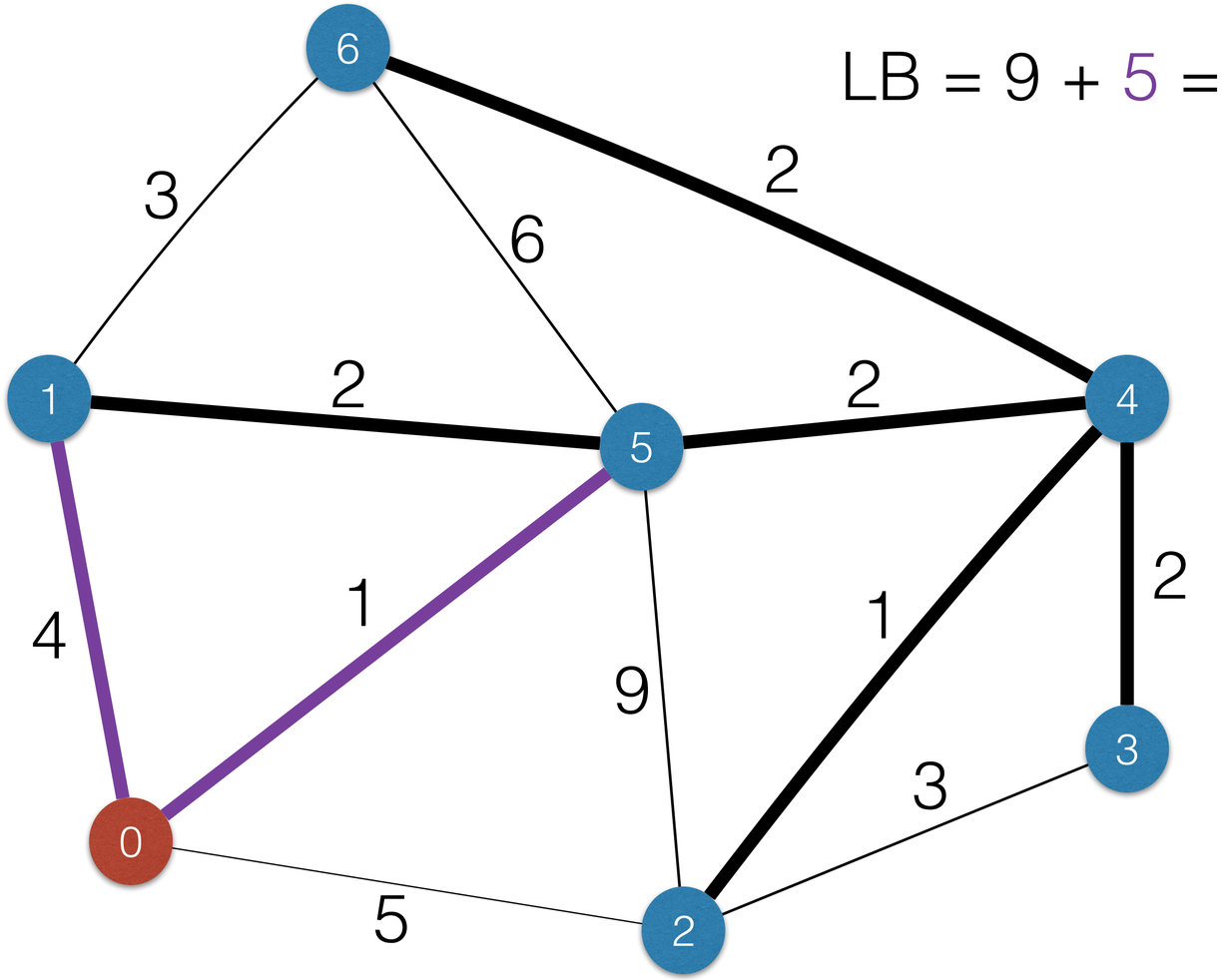
$$\min \sum_e x_e \cdot w_e$$

$\{e \mid x_e = 1\}$ = spanning tree + 1 edge

~~$\sum_{e \in \delta(i)} x_e = 2, \forall i > 0$~~

$$\sum_{e \in \delta(0)} x_e = 2$$

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



LB = 9 + 5 = 14

= minimum weight 1-tree

= min spanning tree in {1..n-1} + two cheapest edge connecting node 0

Time Complexity for Finding a min 1-Tree = Kruskal Complexity

- $\mathcal{O}(E \log V)$ = the time to sort the edges

```
Kruskal(graph):
    Initialize MST as an empty set
    Initialize a disjoint set (or union-find) structure to keep track of connected components

    Sort all the edges in the graph in non-decreasing order by their weight

    For each edge (u, v) in the sorted edge list:
        If u and v are in different components (i.e., they are not connected):
            Add the edge (u, v) to MST
            Union the sets of u and v (i.e., connect the components of u and v)

    Return MST
```

Another TSP Model

$$\min \sum_{e} x_e \cdot w_e$$

$\{e \mid x_e = 1\}$ form a unique connected component

$$\sum_{e \in \delta(i)} x_e = 2, \forall i$$

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

Another TSP Relaxation

$$\min \sum_e x_e \cdot w_e$$

~~$\{e \mid x_e = 1\}$ form a unique connected component~~

$$\sum_{e \in \delta(i)} x_e = 2, \forall i$$

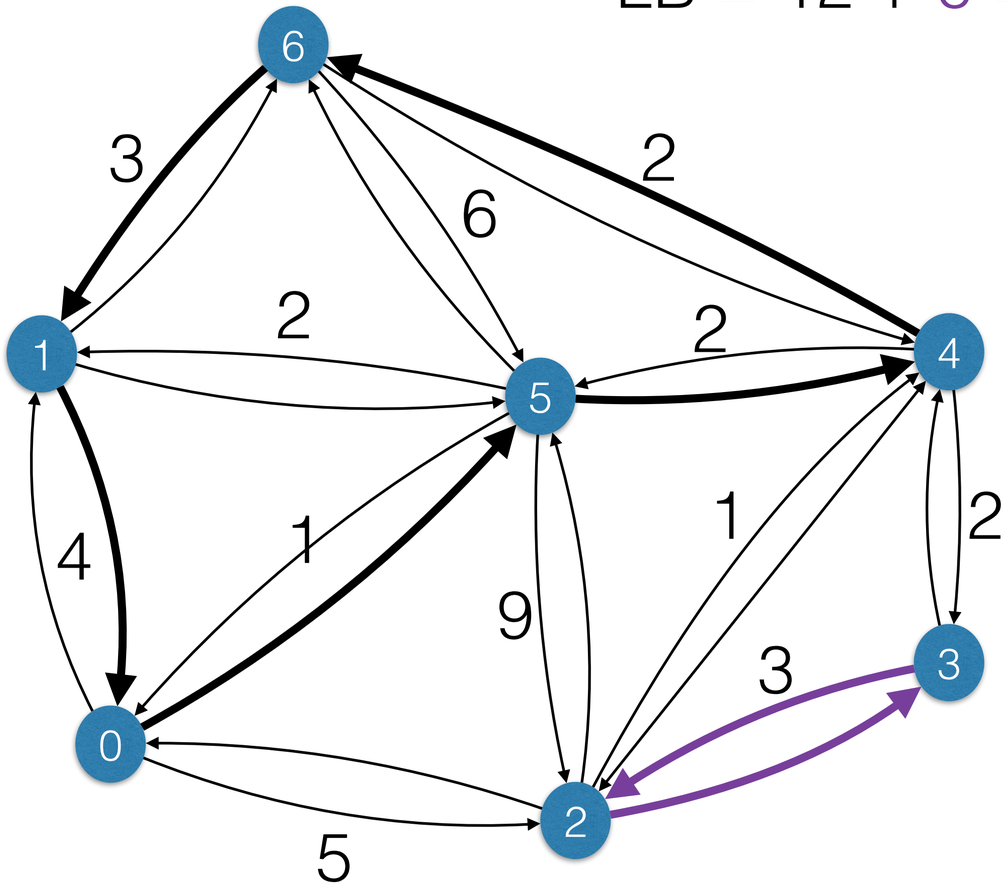
$$x_e \in \{0,1\}, \forall e$$

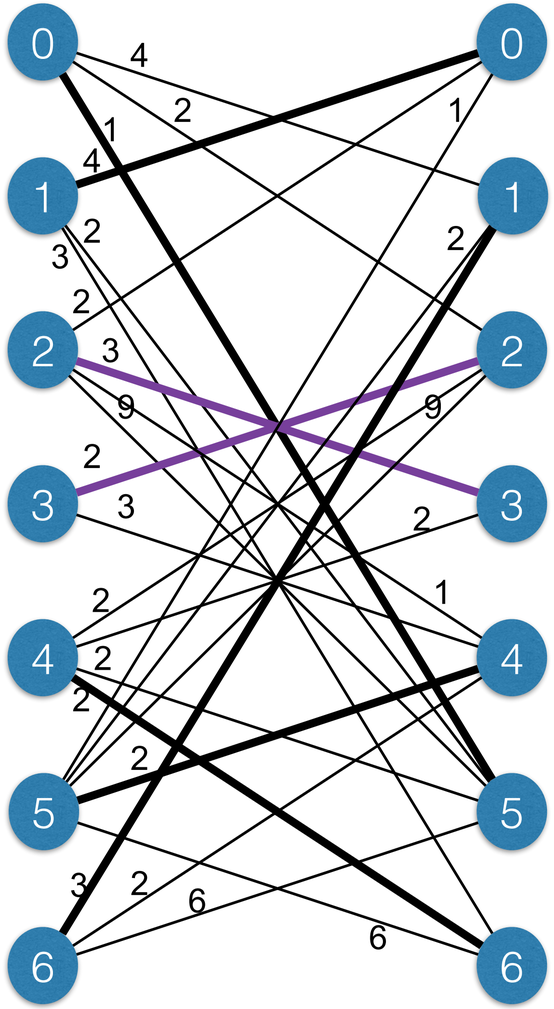
= minimum weight matching in a bipartite graph

= in this solution we may have several sub-tours

Minimum Weight Matching Relaxation

LB = 12 + 6 = 18





Another TSP Relaxation

$$\min \sum_e x_e \cdot w_e$$

~~$\{e \mid x_e = 1\}$ form a unique connected component~~

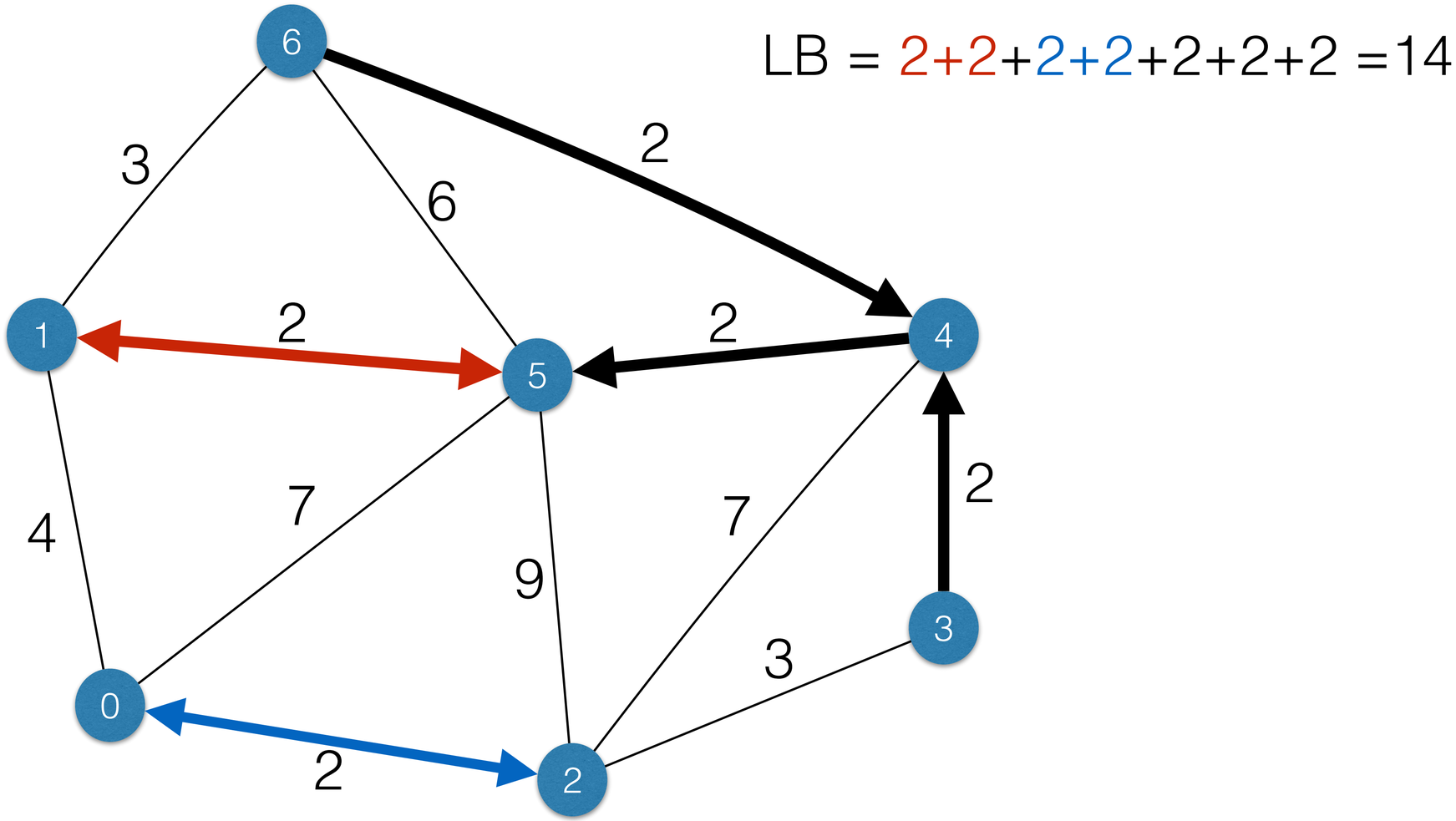
~~$\sum_{e \in \delta(i)} x_e = 2, \forall i$~~ → $\sum_{e \in \delta(i)} x_e \geq 1, \forall i \wedge \sum_e x_e = n$

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

= take the minimum cost edge adjacent to each node

Other possible Relaxation

- For each node, take it's lightest adjacent edge while keeping the in degree = 1(not necessarily connected). This is a minimum weight matching



The Branch and Bound Project

Package “branchandbound”(don’t forget to pull to get the latest update).

5 steps (implementation) + Report:

1. Implement the cheapest-incident relaxation `CheapestIncidentLowerBound` class. You can test your result by executing `CheapestIncidentLowerBoundTestFast`.
2. Implement the one-tree relaxation `OneTreeLowerBound` class. You can test your result by executing `OneTreeLowerBoundTestFast`.
3. Implement the branch and bound for the TSP in the `BranchAndBoundTSP` class which will use the `OneTreeLowerBound` bound procedure you just implemented earlier. You can test your result by executing `BranchAndBoundTSPTestFast`.
4. (Next week) Implement an enhanced bound calculation for the one-tree based on Lagrangian relaxation in the `HeldKarpLowerBound class. You can test your result by executing HeldKarpLowerBoundFast`.
5. (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

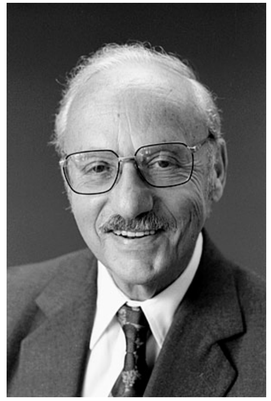
- Although we treat with NP-Hard problems:
- Solving them with Branch and Bound requires good (ie. tight) upper/lower bounds for maximization/minimization.
- Bound computation must be fast (most often using well known polynomial algorithms).
- Optimization is strongly related to algorithms and implementation.

History

Richard E. Bellman

1920-1984

Dynamic Programming 1950's



Georges Dantzig

1914-2005

Knapsack Relaxation 1957

MISS ALISON DOIG, of Melbourne University's Department of Statistics, prepares information for a computer.

Ailsa Land & Alison (Doig) Harcourt

Branch and Bound 1960