Branch-and-Bound with Decision Diagrams

LSINF2266 - Advanced Algorithms for Optimization

Xavier Gillard and Pierre Schaus

Part 1: Reminders

Maximization with Branch-and-Bound (Reminder)

- Basically **Depth-First Search**
- Procedure to derive an **Upper Bound** on the objective (UB)
- Prune sub-problem space whenever

$$UB(\sigma) \leq v^*$$

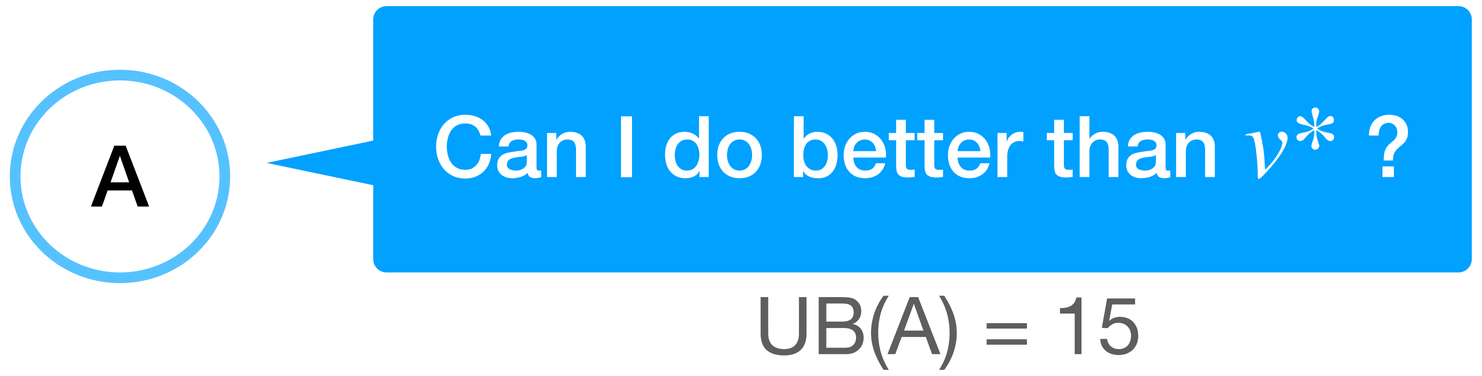
A node from
the search tree

Value of the best known solution
(Note: This is a Lower Bound on the true optimum)

Branch-and-Bound (Reminder)

Visually

$$\boxed{v^* = -\infty}$$



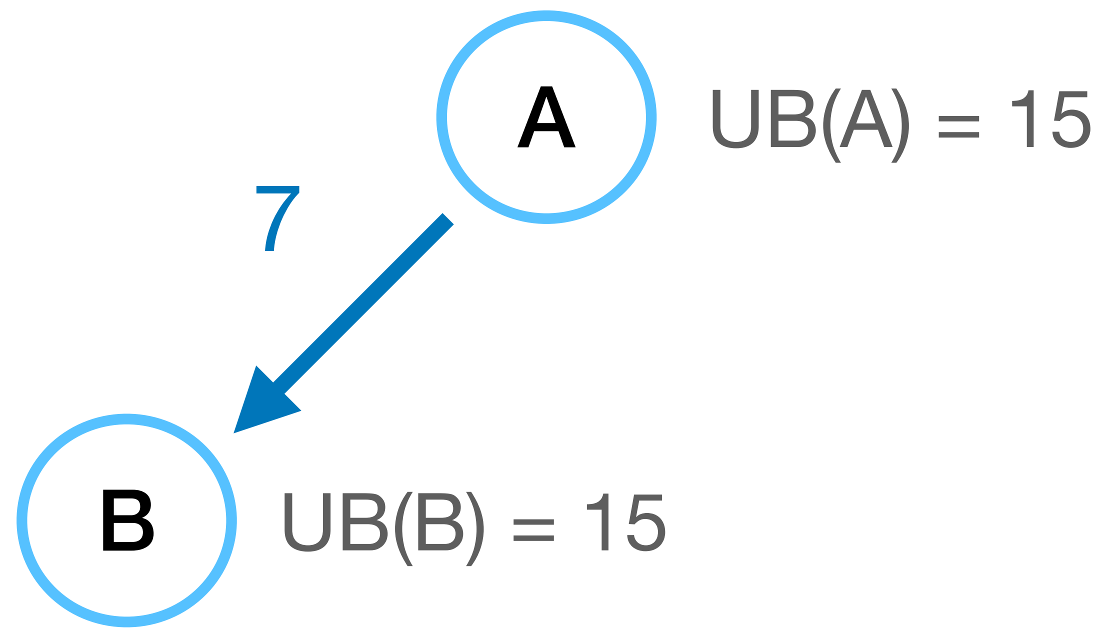
Can I do better than v^* ?

UB(A) = 15

Branch-and-Bound (Reminder)

Visually

$v^* = -\infty$

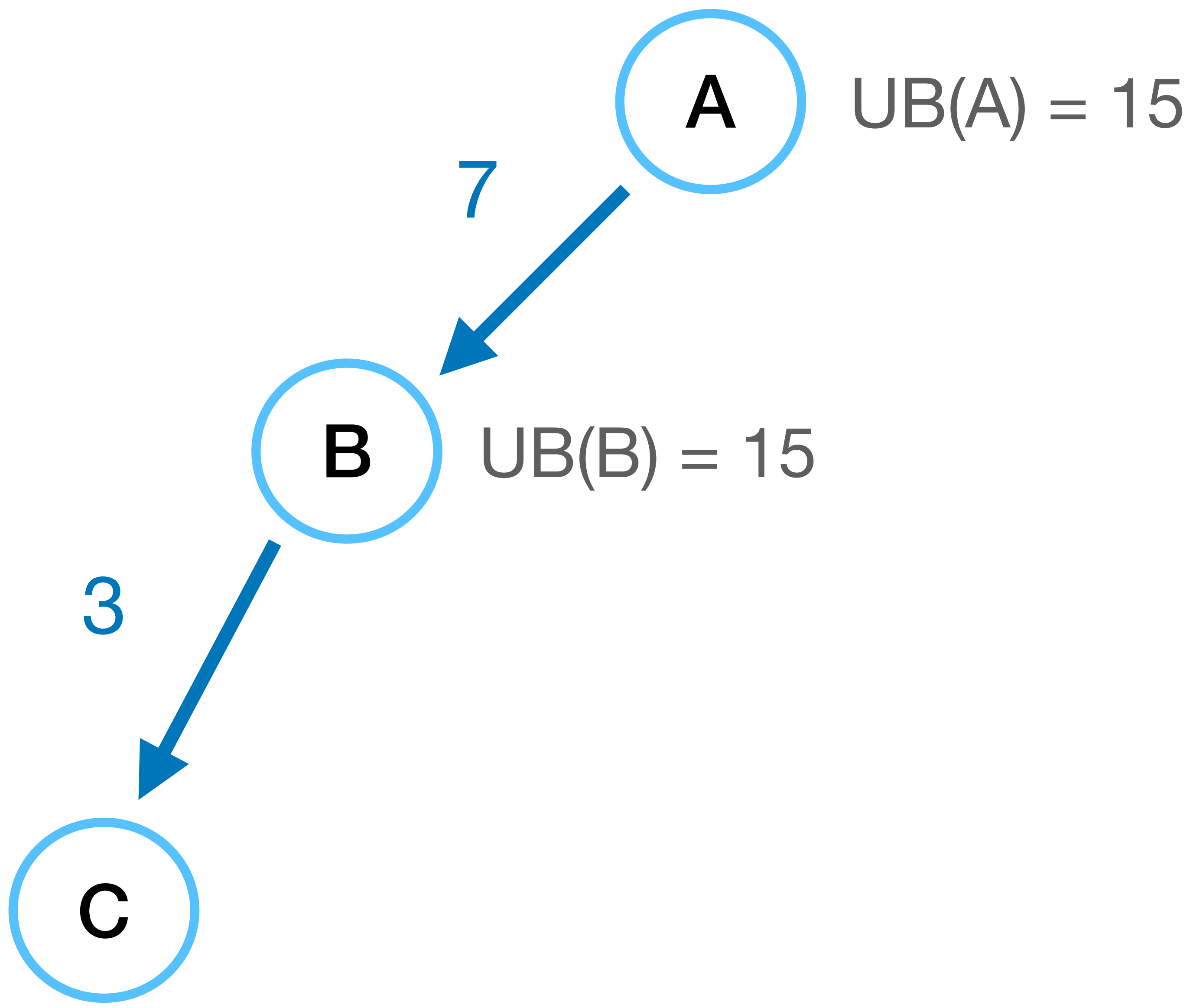


Can I do better than v^* ?

Branch-and-Bound (Reminder)

Visually

$v^* = -\infty$

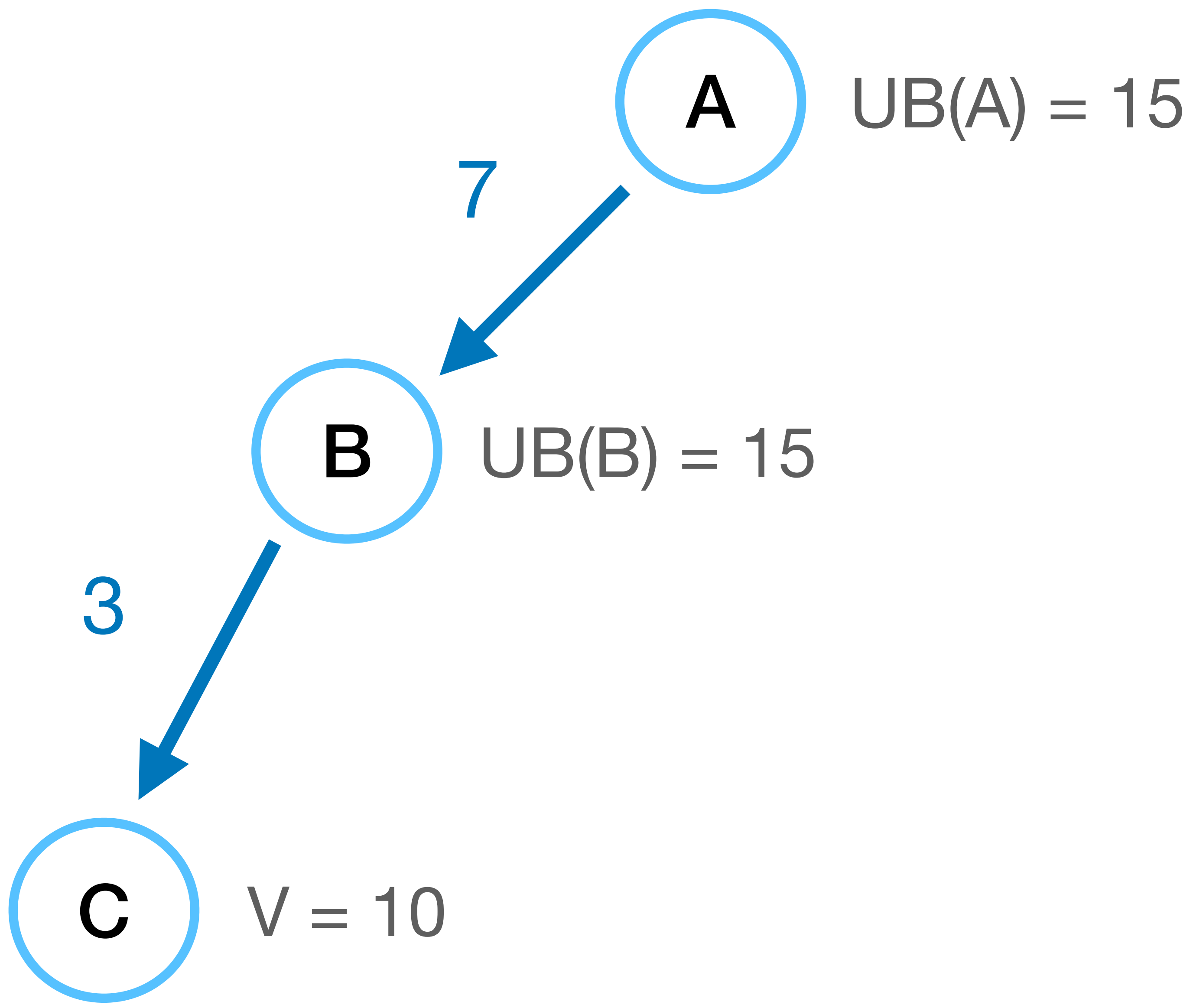


I have reached a new solution (10)
Is it better than v^* ?

Branch-and-Bound (Reminder)

Visually

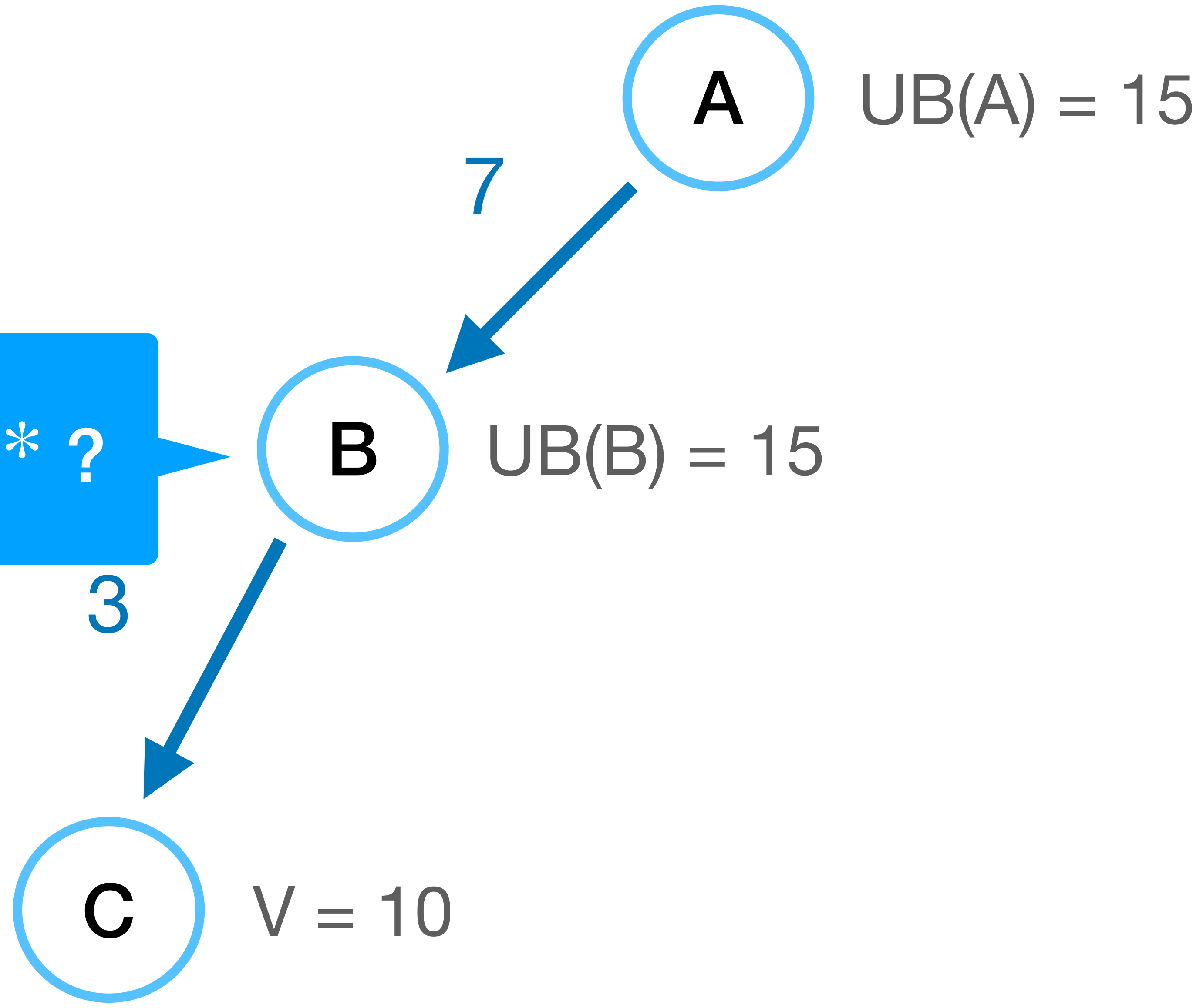
$$\boxed{v^* = 10}$$



Branch-and-Bound (Reminder)

Visually

$\boxed{v^* = 10}$

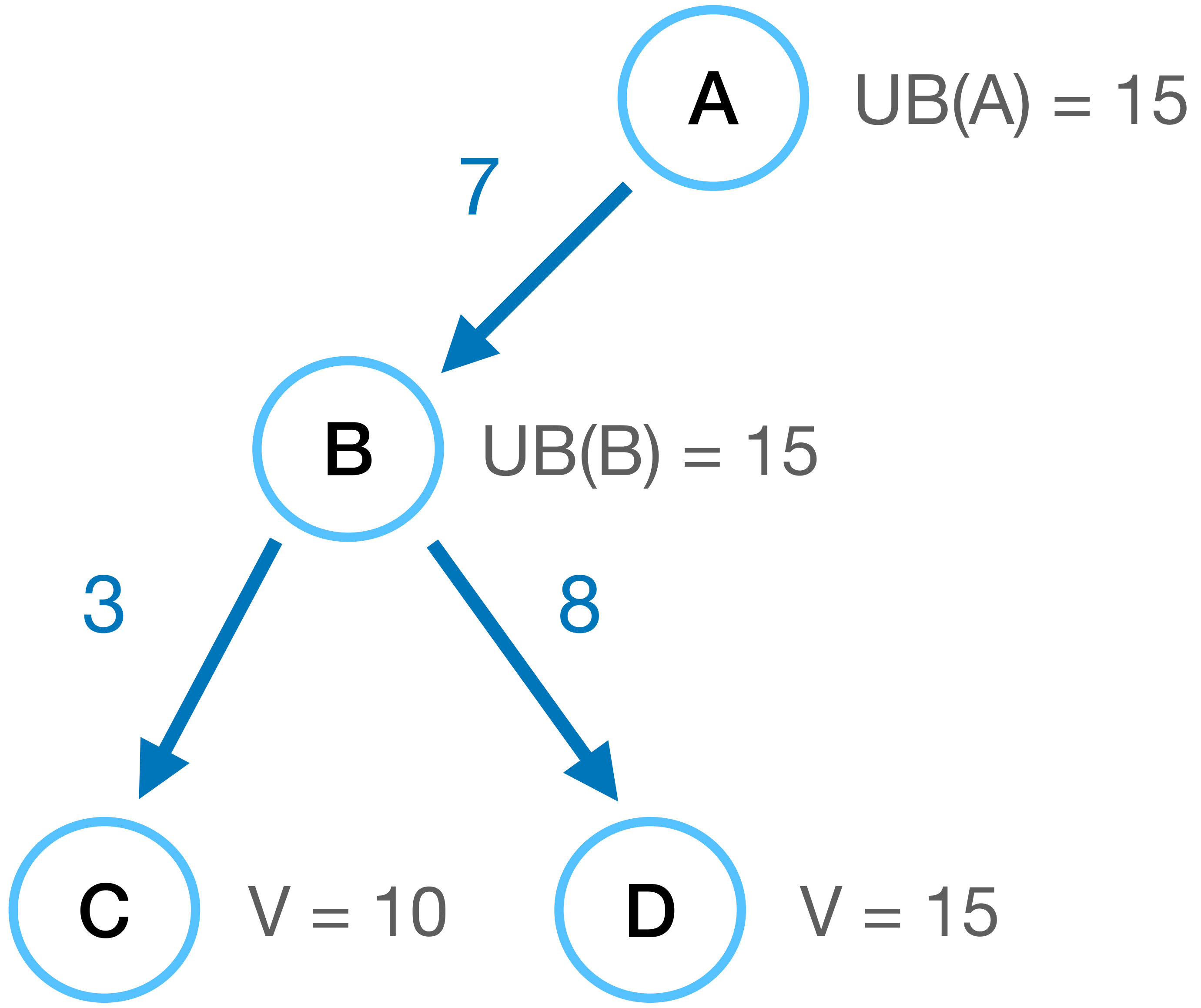


Can I do better than v^* ?

Branch-and-Bound (Reminder)

Visually

$\boxed{v^* = 10}$



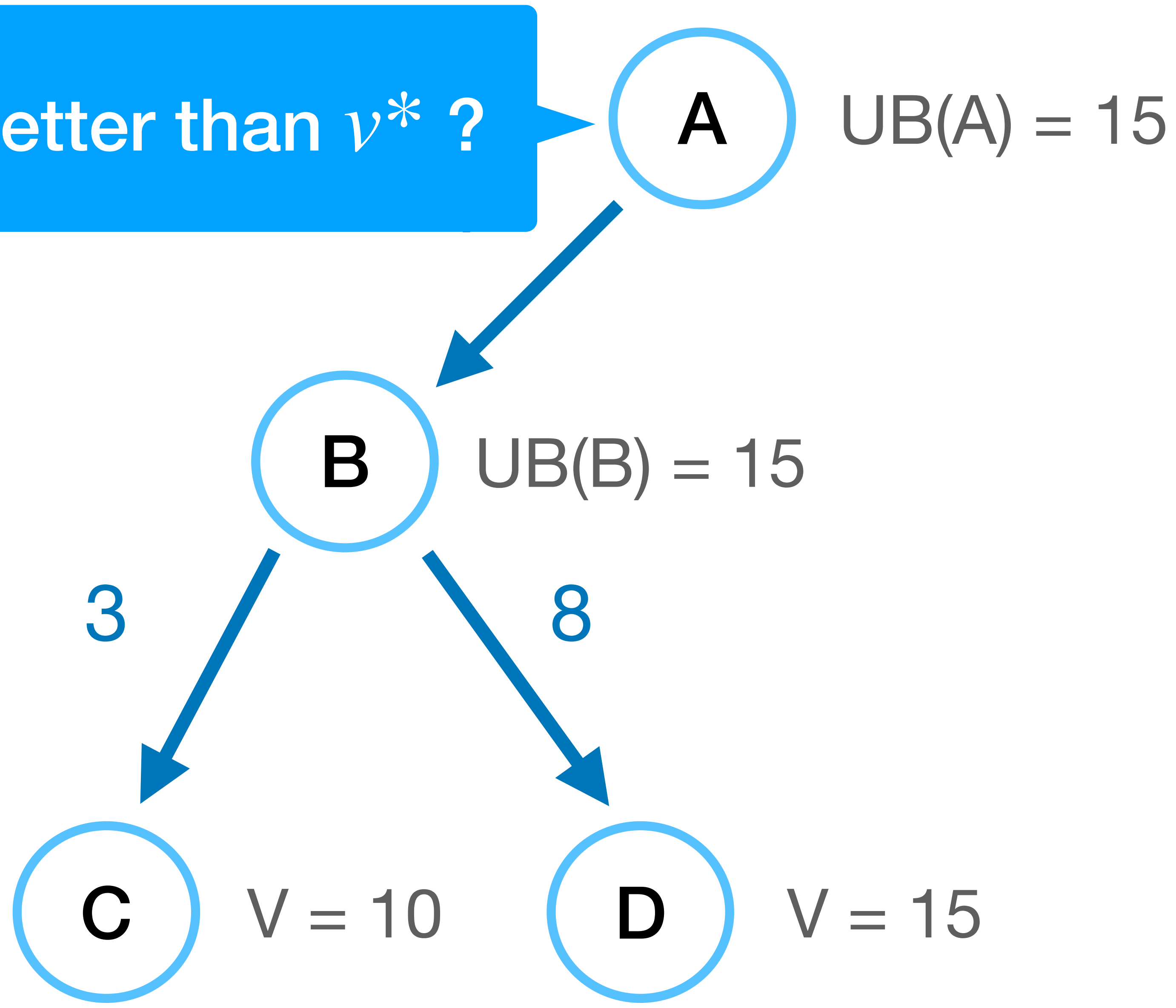
I have reached a new solution (15)
Is it better than v^* ?

Branch-and-Bound (Reminder)

Visually

$v^* = 15$

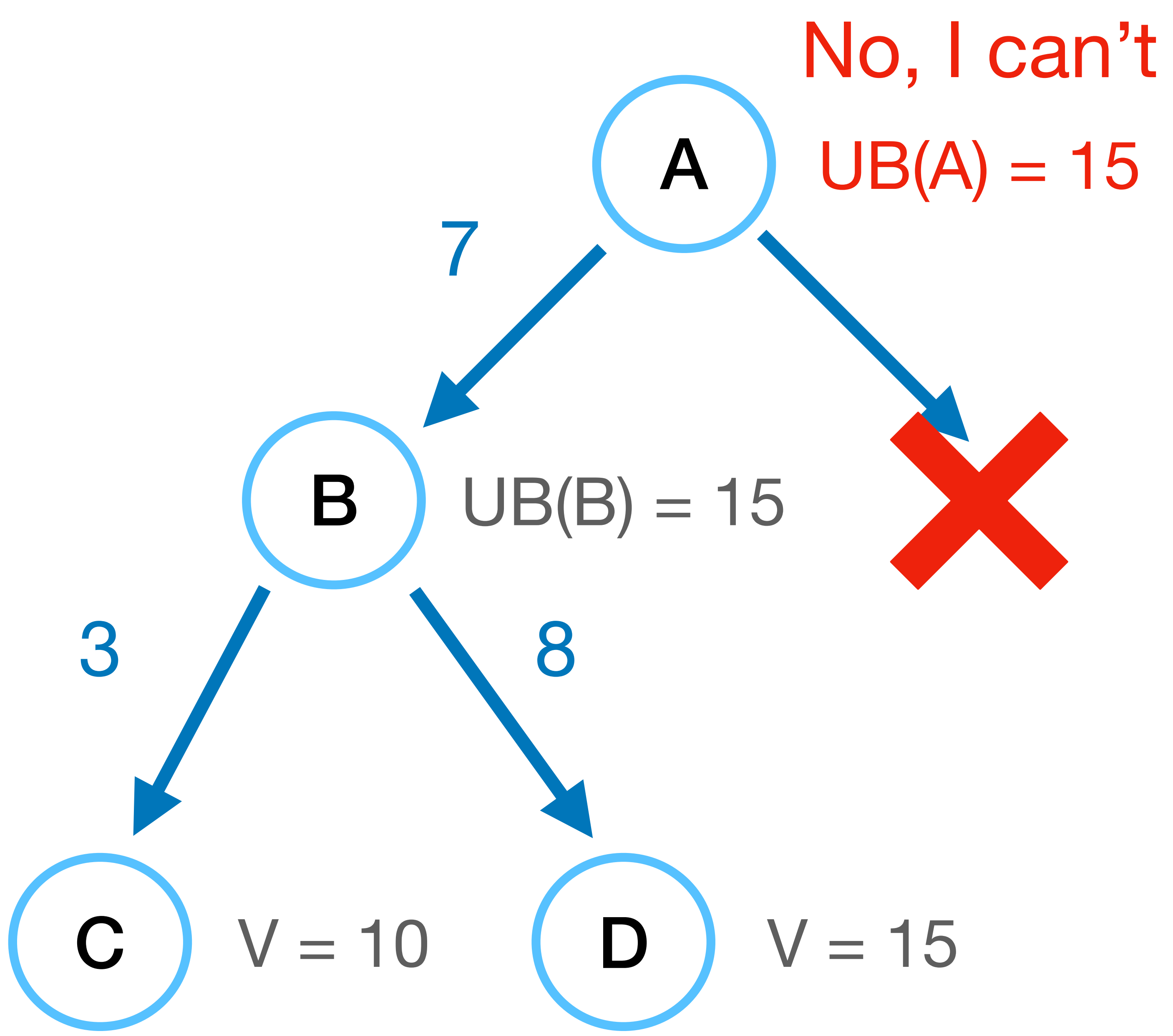
Can I do better than v^* ?



Branch-and-Bound (Reminder)

Visually

$v^* = 15$



Dynamic Programming (Reminder)

- Recursive Model (hence 2 steps)
  - Base Case
  - General Case

Embodied in

Bellman Recurrence Equations

Dynamic Programming (Reminder)

Knapsack Example

Base Case

$$h_0(c,0) = 0$$

$$h_0(c,1) = p_0 \qquad \text{if } \ w_0 \leq c$$

$$= \perp \qquad \text{otherwise}$$

Recurrence

$$h_i(c,0) = \max \left\{ h_{i-1}(c,0), h_{i-1}(c,1) \right\}$$

$$h_i(c,1) = p_i + \max \left\{ \ h_{i-1}(c,0), \ h_{i-1}(c - w_i,1) \ \right\} \qquad \text{if } \ w_i \leq c$$

$$= \perp \qquad \text{otherwise}$$

Objective

$$\max \left\{ \ h_N(C,0), \ \ h_N(C,1) \ \right\}$$

Dynamic Programming (Reminder)

Knapsack Example

Capacity

Base Case

$$h_0(c,0) = 0$$

$$h_0(c,1) = p_0 \qquad \text{if } \; w_0 \leq C$$

$$= \perp \qquad \text{otherwise}$$

Recurrence

$$h_i(c,0) = \max\left\{h_{i-1}(C,0), h_{i-1}(C,1)\right\}$$

$$h_i(c,1) = p_i + \max\left\{\; h_{i-1}(c,0),\; h_{i-1}(c-w_i,1) \;\right\} \qquad \text{if } \; w_0 \leq C$$

$$= \perp \qquad \text{otherwise}$$

Objective

$$\max\left\{\; h_N(C,0), \;\; h_N(C,1) \;\right\}$$

Dynamic Programming (Reminder)

Knapsack Example

Capacity

profit for item i

Base Case

$$
\begin{aligned}
h_0(c,0) &= 0 \\
h_0(c,1) &= p_0 && \text{if } w_0 \leq \boxed{C} \\
&= \perp && \text{otherwise}
\end{aligned}
$$

Recurrence

$$
\begin{aligned}
h_i(c,0) &= \max\left\{h_{i-1}(C,0), h_{i-1}(C,1)\right\} \\
h_i(c,1) &= \boxed{p_i} + \max\left\{\ h_{i-1}(c,0),\ h_{i-1}(c-w_i,1)\ \right\} && \text{if } w_0 \leq C \\
&= \perp && \text{otherwise}
\end{aligned}
$$

Objective

$$
\max\left\{\ h_N(C,0),\ \ h_N(C,1)\ \right\}
$$

Dynamic Programming (Reminder)

Knapsack Example

Capacity

profit for item i

Base Case

$h_0(c,0) = 0$

$h_0(c,1) = p_0$ if $w_0 \leq C$

$= \perp$ otherwise

Recurrence

$h_i(c,0) = \max\left\{h_{i-1}(C,0), h_{i-1}(C,1)\right\}$

$h_i(c,1) = p_i + \max\left\{\ h_{i-1}(c,0),\ h_{i-1}(c - w_i,1)\ \right\}$ if $w_0 \leq C$

$= \perp$ otherwise

Weight of item i

Objective

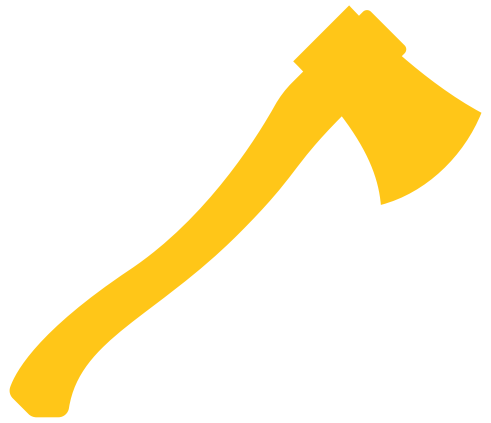
$$\max\left\{\ h_N(C,0),\ \ h_N(C,1)\ \right\}$$

Dynamic Programming (Reminder)

Numerical Knapsack Example

WEIGHT: 3

PROFIT: 15



WEIGHT: 3

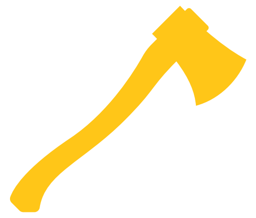
PROFIT: 12

WEIGHT: 12

PROFIT: 120

CAPACITY: 15

WEIGHT: 3
PROFIT: 15



WEIGHT: 3
PROFIT: 12

WEIGHT: 12
PROFIT: 120

Decision Variables

$x_0 =$ 🔨 $x_1 =$ 🪓 $x_2 =$ ⛺

🔨 **WEIGHT: 3**
PROFIT: 15

🪓 **WEIGHT: 3**
PROFIT: 12

⛺ **WEIGHT: 12**
PROFIT: 120

Decision Variables

$x_0 =$ 🔨 $x_1 =$ 🪓 $x_2 =$ ⛺

Domains

$$D_0 = D_1 = D_2 = \{\text{yes, no}\}$$

🔨 WEIGHT: 3
PROFIT: 15

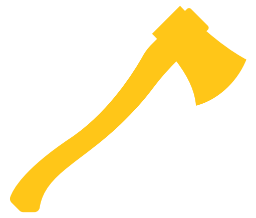
🪓 WEIGHT: 3
PROFIT: 12

⛺ WEIGHT: 12
PROFIT: 120

State Spaces

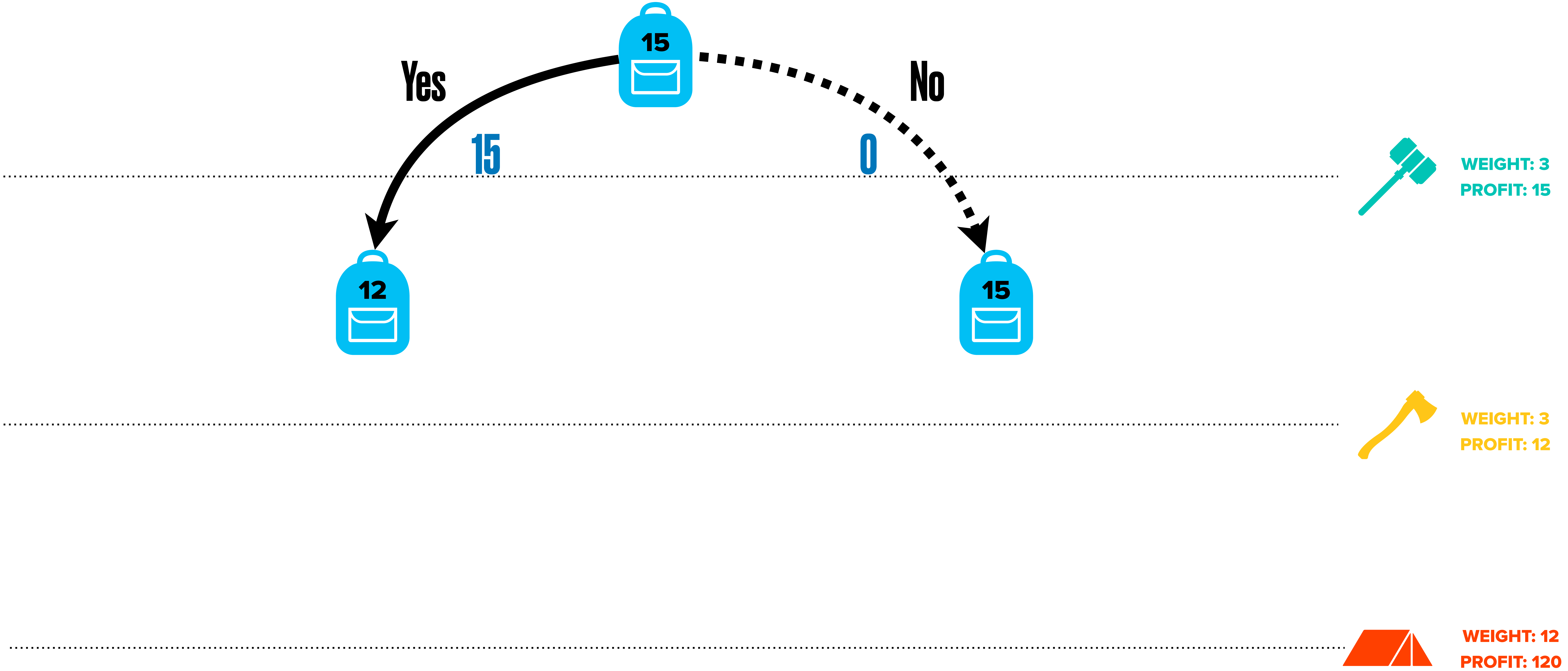
$$S_0 = S_1 = S_2 = \{0, 1, \ldots, 15\}$$

WEIGHT: 3
PROFIT: 15



WEIGHT: 3
PROFIT: 12

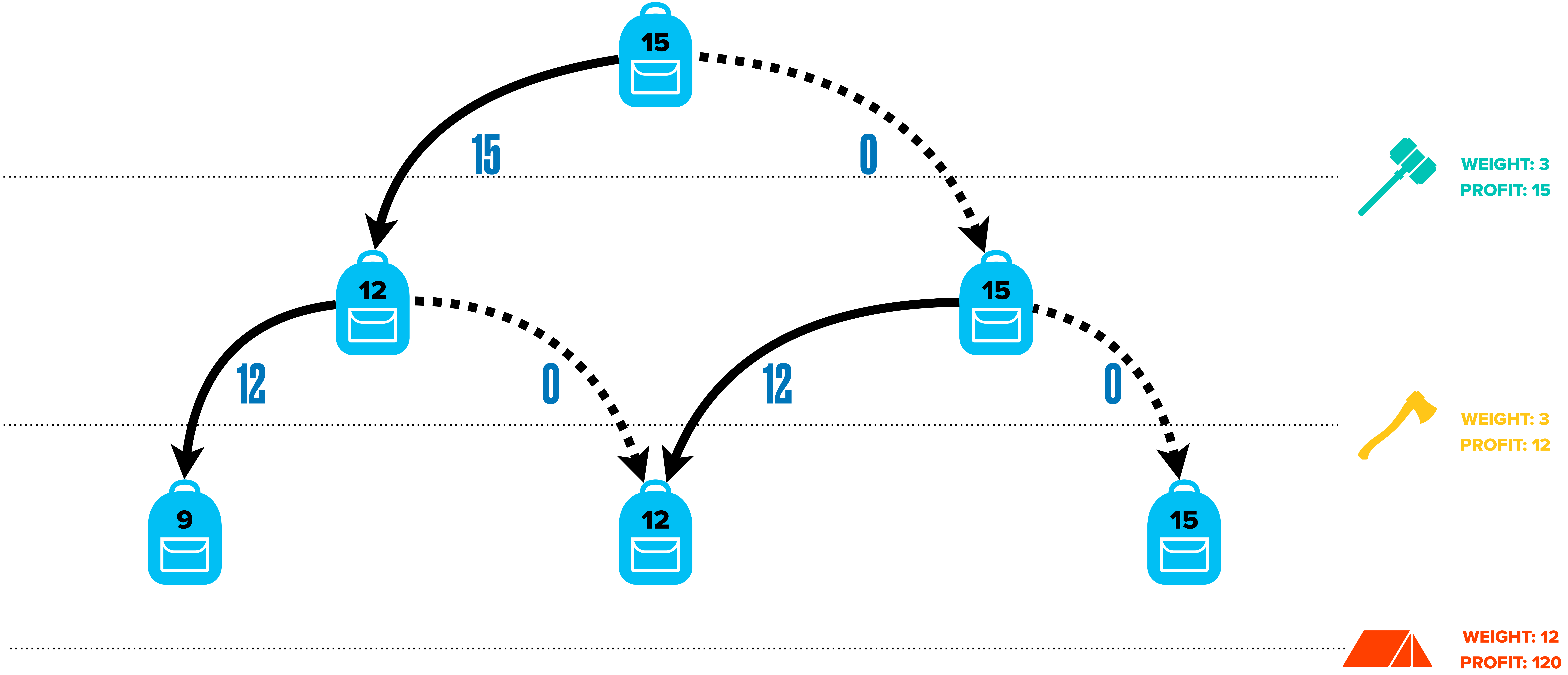
WEIGHT: 12
PROFIT: 120



WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

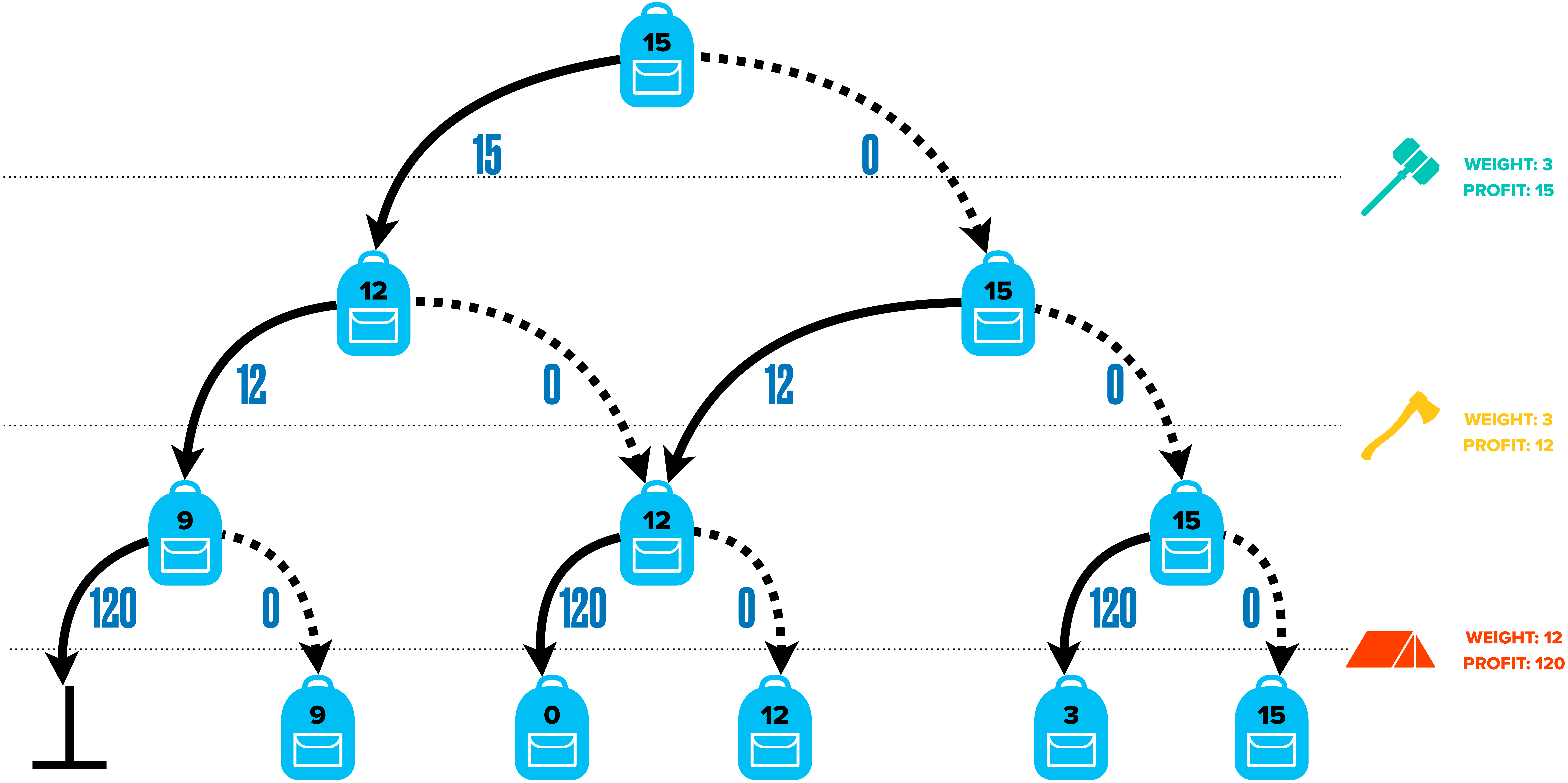
WEIGHT: 12
PROFIT: 120

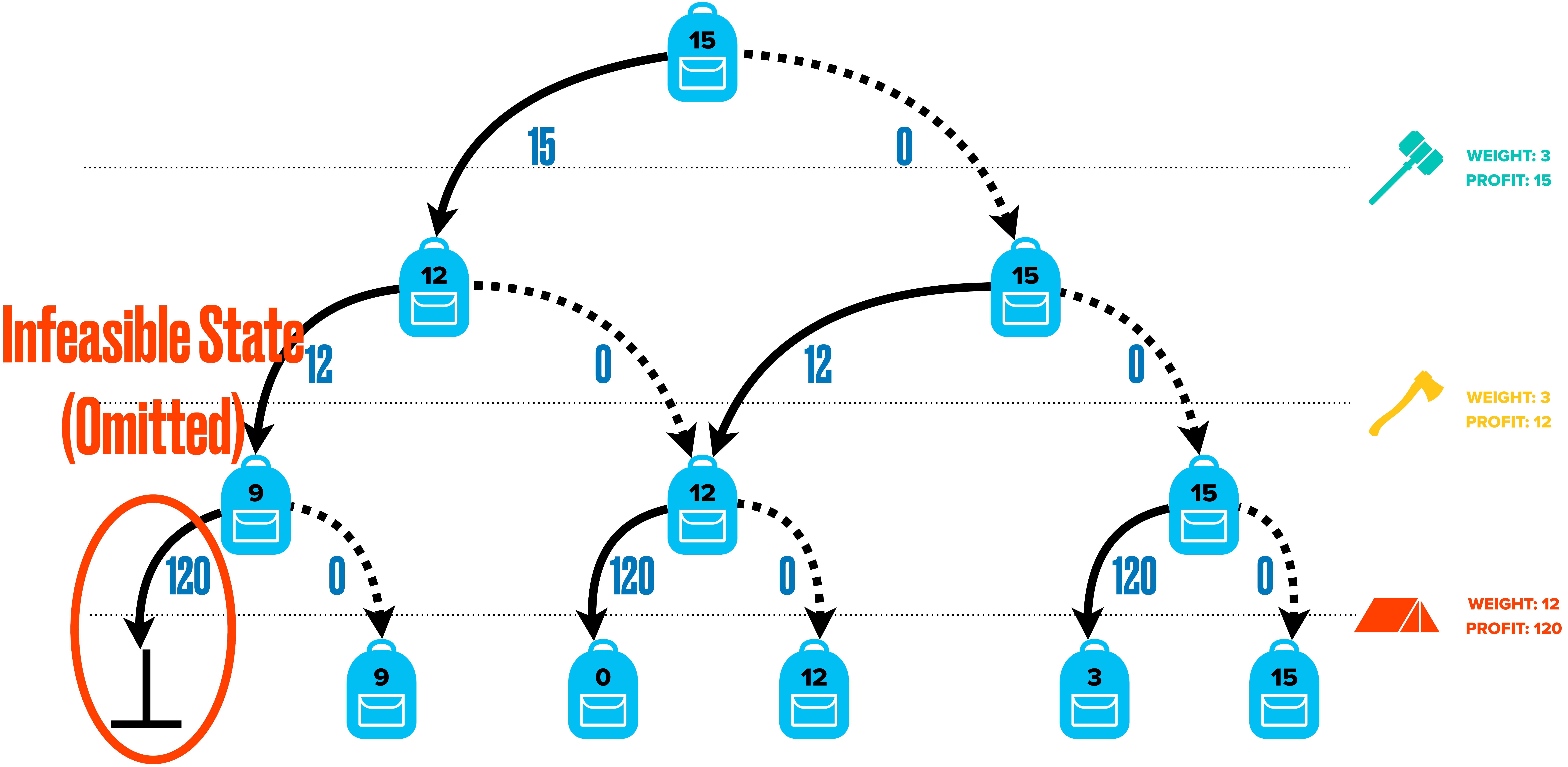


WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

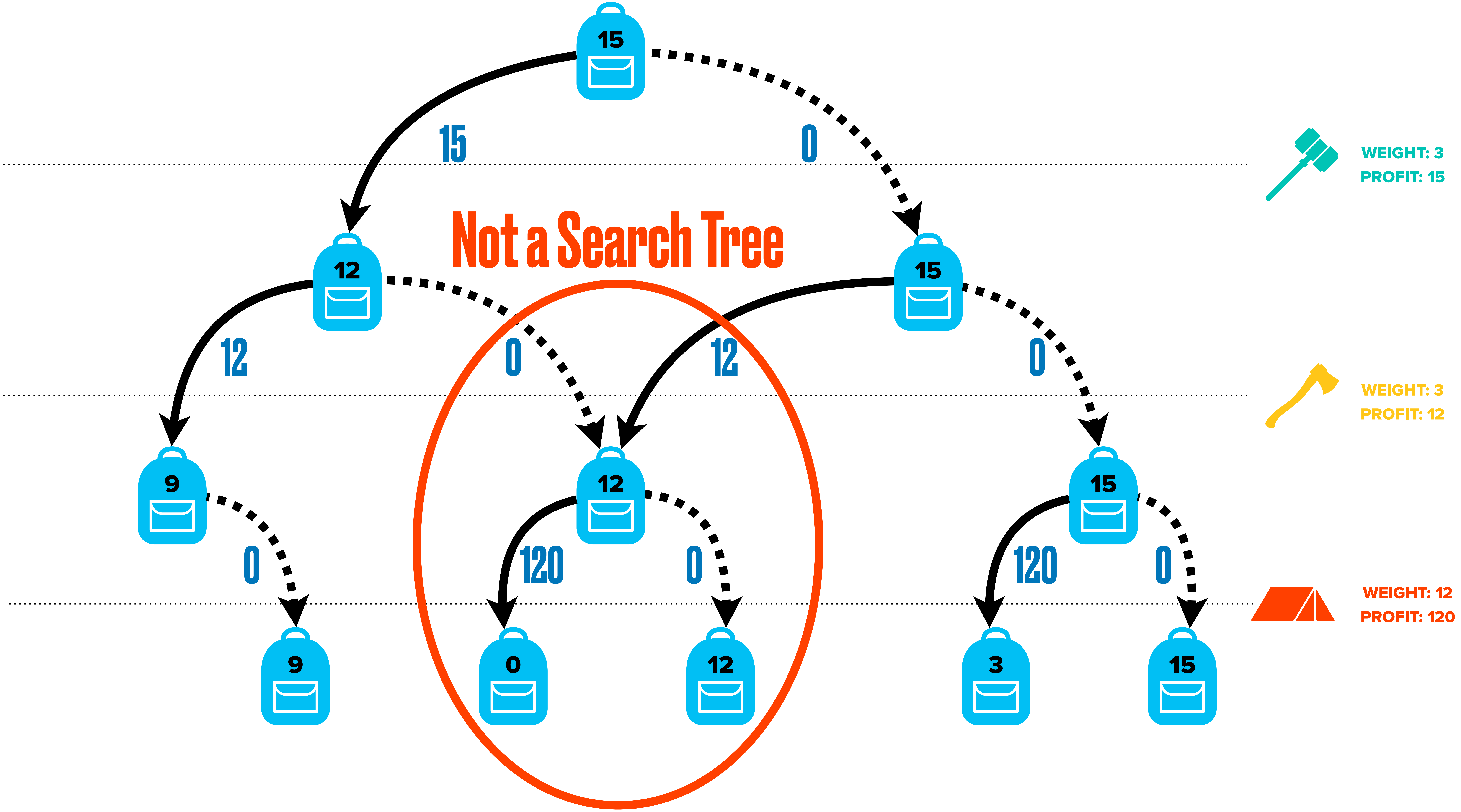
WEIGHT: 12
PROFIT: 120


15
15
0
12
15
12
0
12
0
9
12
15
120
0
120
0
120
0
9
0
12
3
15
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120



Infeasible State
(Omitted)

Not a Search Tree



WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

WEIGHT: 12
PROFIT: 120

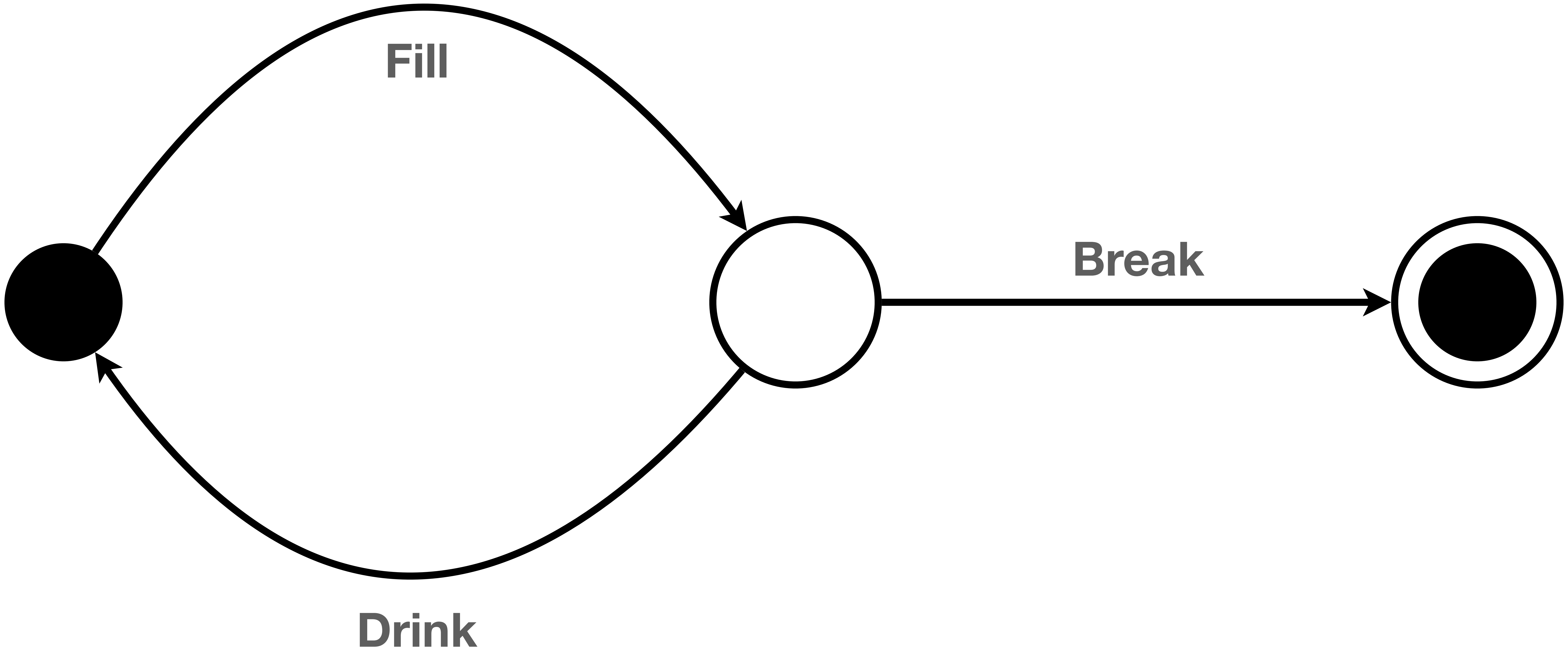
Observation

Dynamic Program can be Seen as a Labeled Transition System (LTS)

- State Spaces
- Initial State
- Initial Value
- Transition Function
- Transition Cost Function

Labeled Transition System

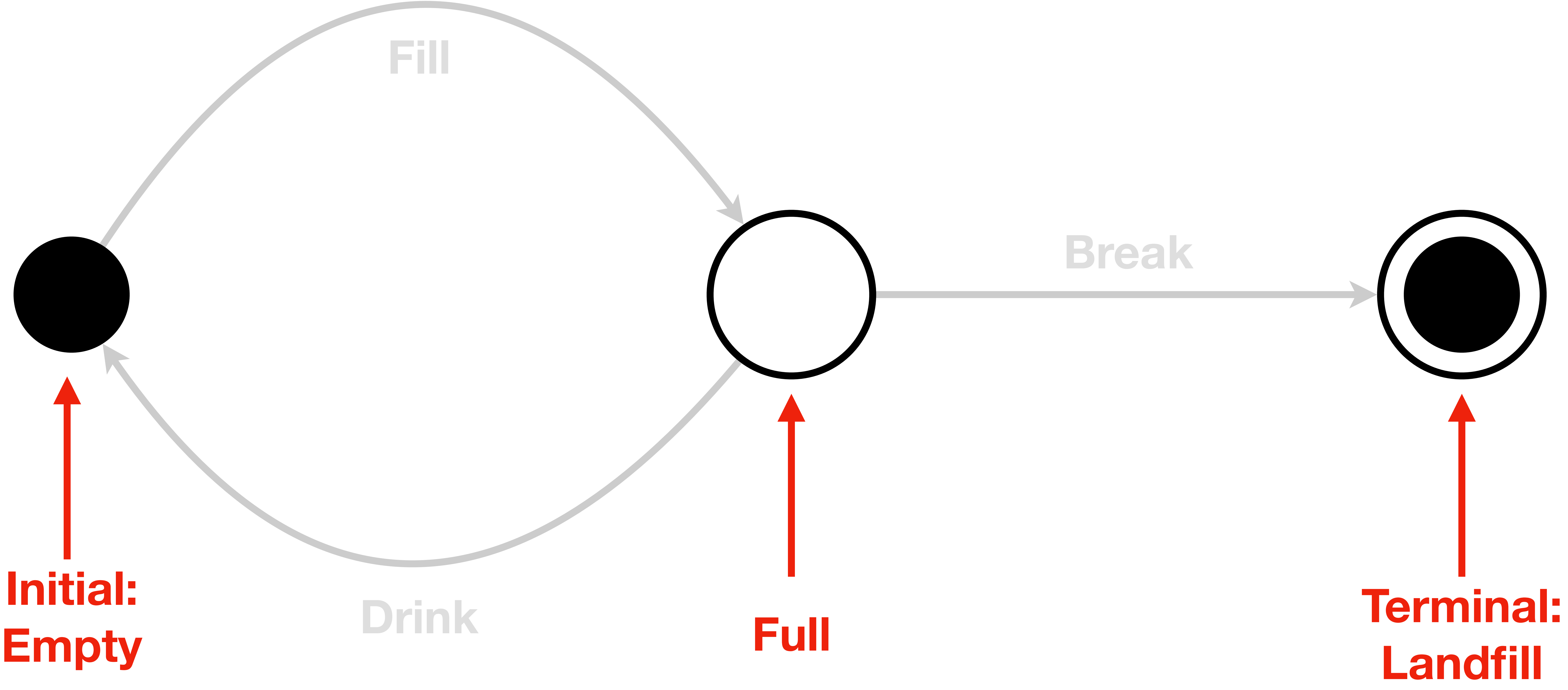
(Refresher/Example: ☕)



Labeled Transition System

(Refresher/Example: ☕)

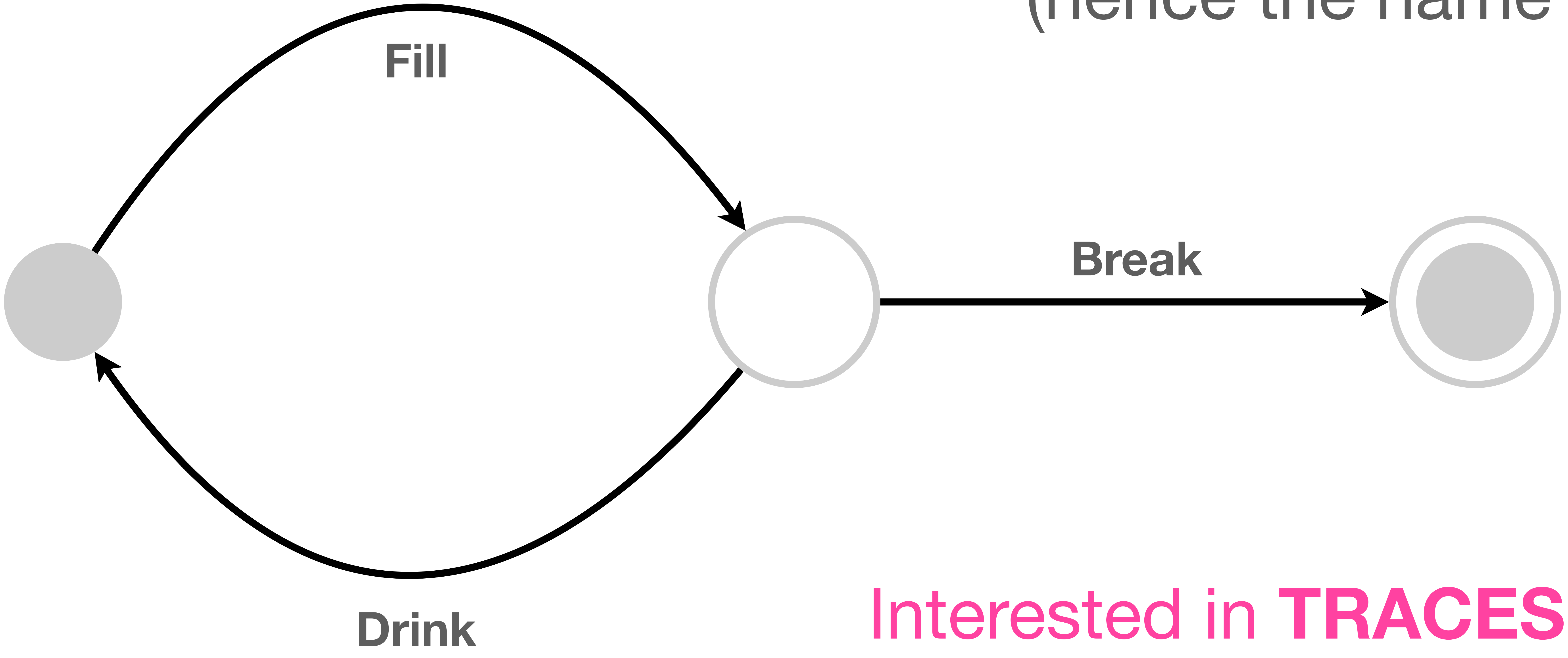
1st ingredient: set of states



Labeled Transition System

(Refresher/Example: )

2nd ingredient: Labeled Transitions
(hence the name !)



Interested in **TRACES** of the automaton

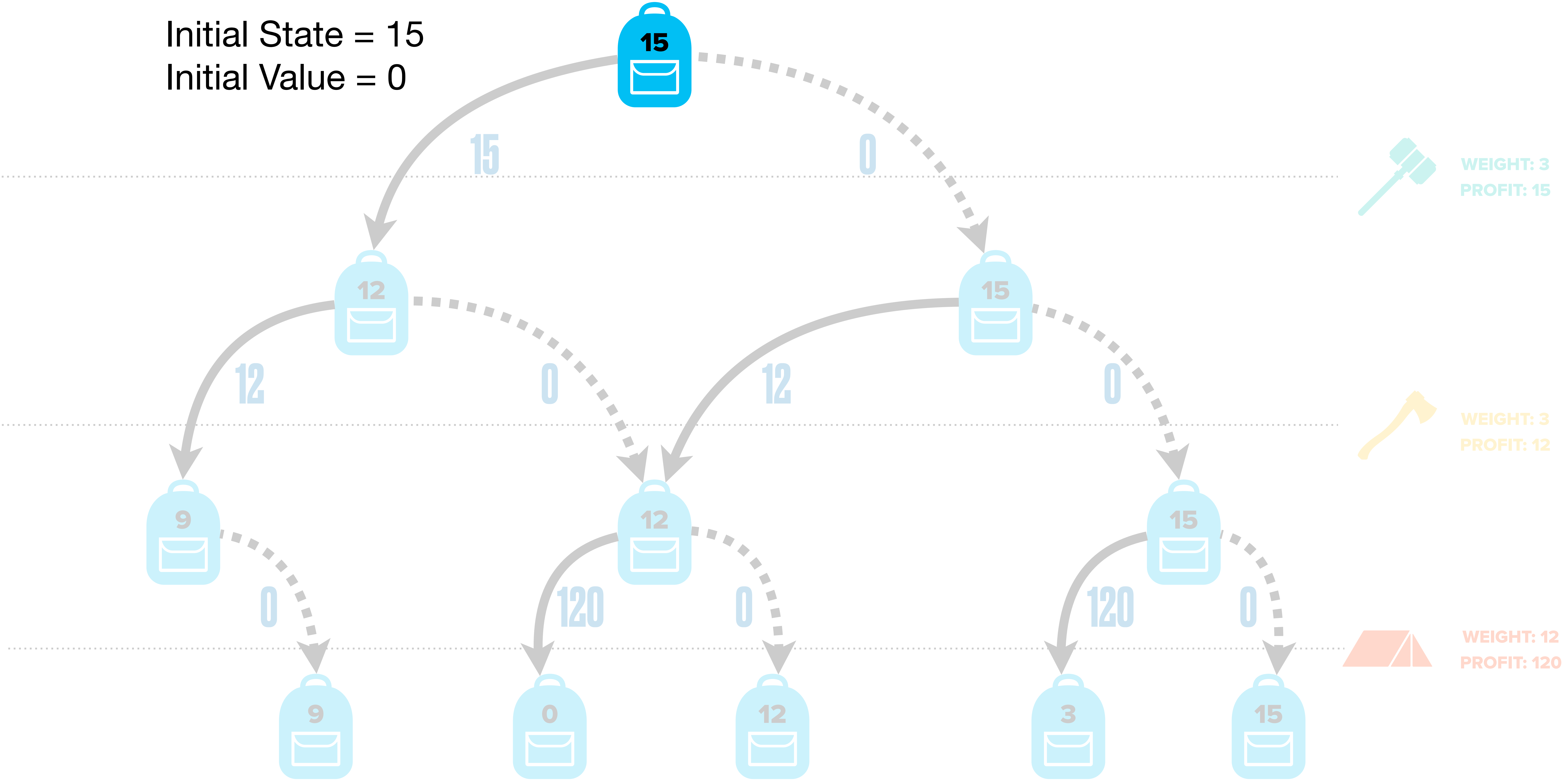
Observation

Dynamic Program can be Seen as a Labeled Transition System (LTS)

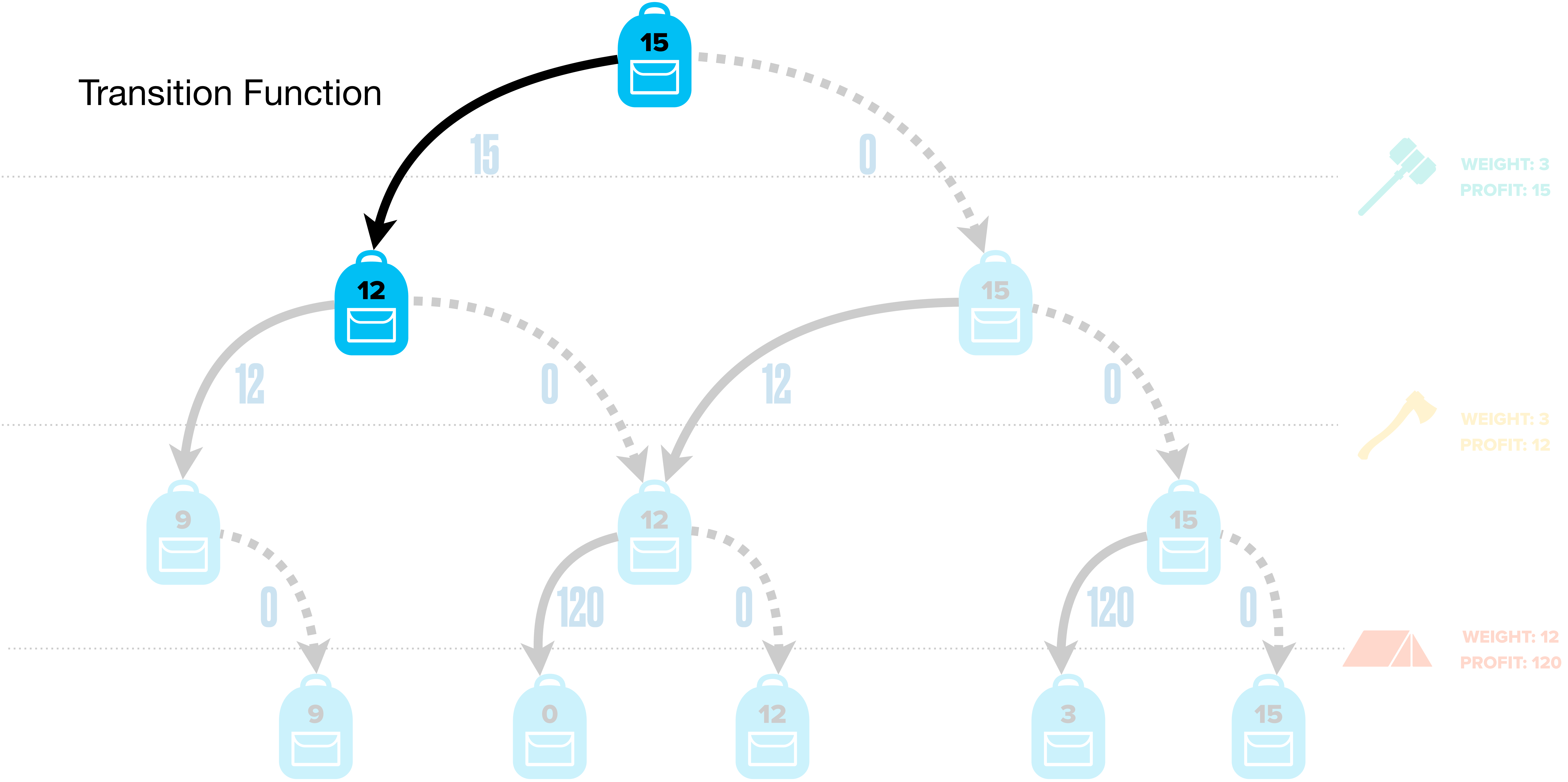
- State Spaces
- Initial State
- Initial Value
- Transition Function
- Transition Cost Function

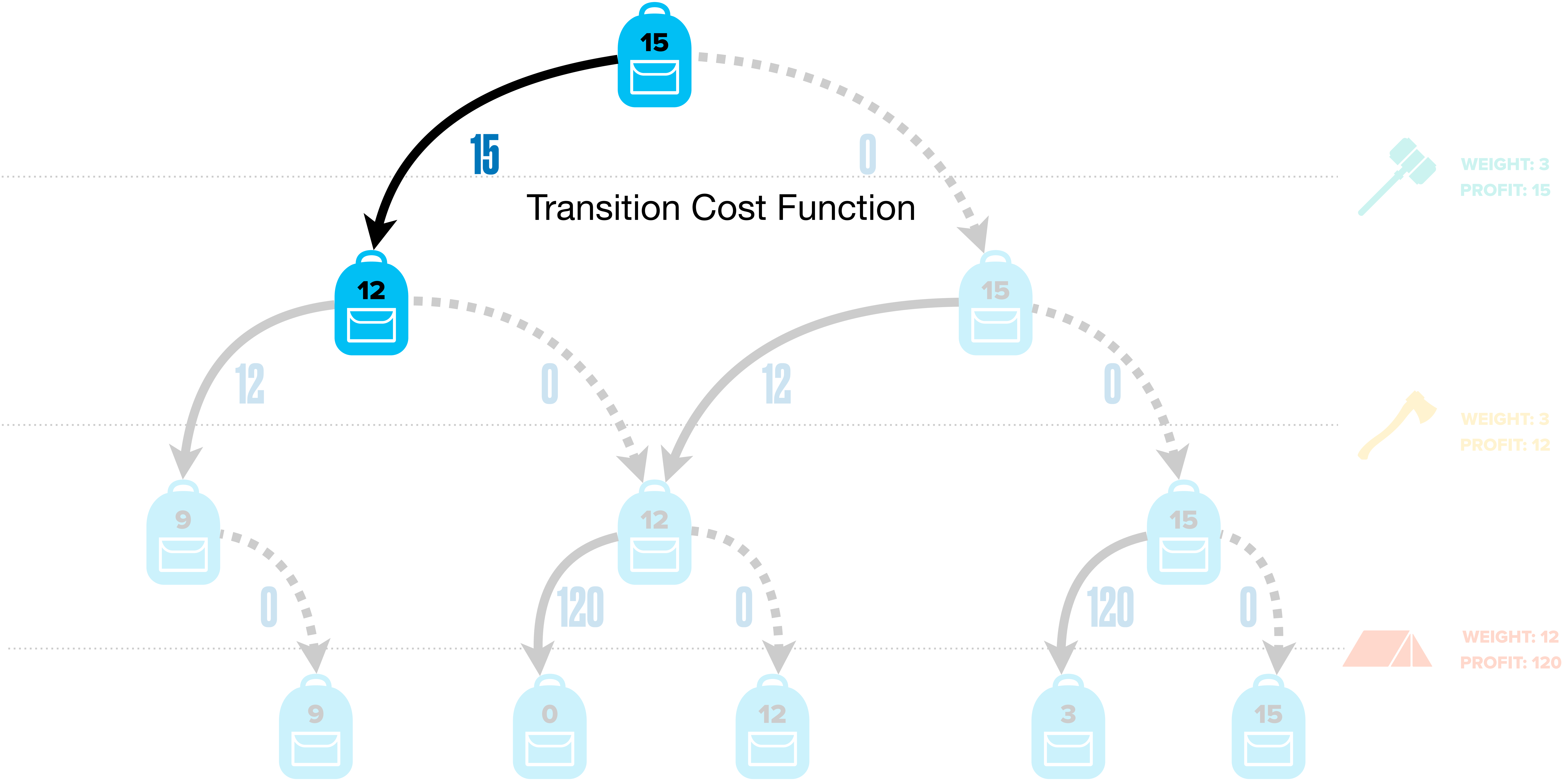
What is a state of the problem ?
Knapsack: Remaining Capacity

Initial State = 15
Initial Value = 0



Transition Function





Transition Cost Function

DP seen as a LTS — Formally

Objective maximize $f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i)$

Characterization

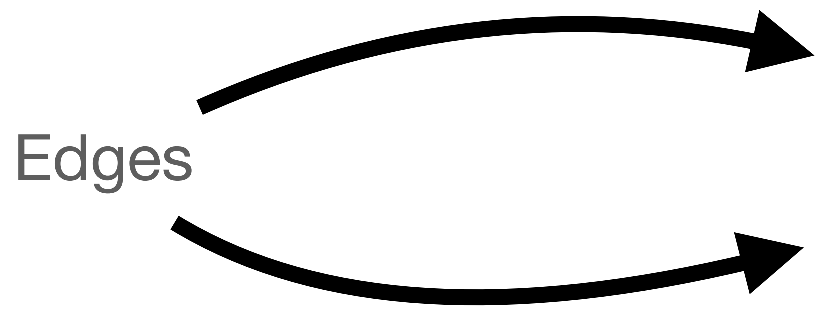
- Decision Variables: $x = \{x_0, x_1, \ldots, x_{n-1}\}$
- Domains: $D = \{D_0, D_1, \ldots, D_{n-1}\}$
- State Spaces : $S = \{S_0, S_1, \ldots, S_n\}$
- Initial State: $r \in S_0$ and $S_0 = \{r\}$
- Terminal State: $t \in S_n$
- Infeasible State: $\perp$ (irrecoverable !)
- Transition Functions: $\tau_i : S_i \times D_i \rightarrow S_{i+1}$
- Transition Cost Function: $h_i : S_i \times D_i \rightarrow \mathbb{R}$
- Initial Value: v_r

DP SEEN AS A LABELED TRANSITION SYSTEM —> DD

Objective maximize $f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i)$

Characterization

- **Decision Variables:** $x = \{x_0, x_1, \ldots, x_{n-1}\}$
- **Domains:** $D = \{D_0, D_1, \ldots, D_{n-1}\}$
- **State Spaces :** $S = \{S_0, S_1, \ldots, S_n\}$
- **Initial State:** $r \in S_0$ **and** $S_0 = \{r\}$
- **Terminal State:** $t \in S_n$
- **Infeasible State:** $\perp$ **(irrecoverable !)**



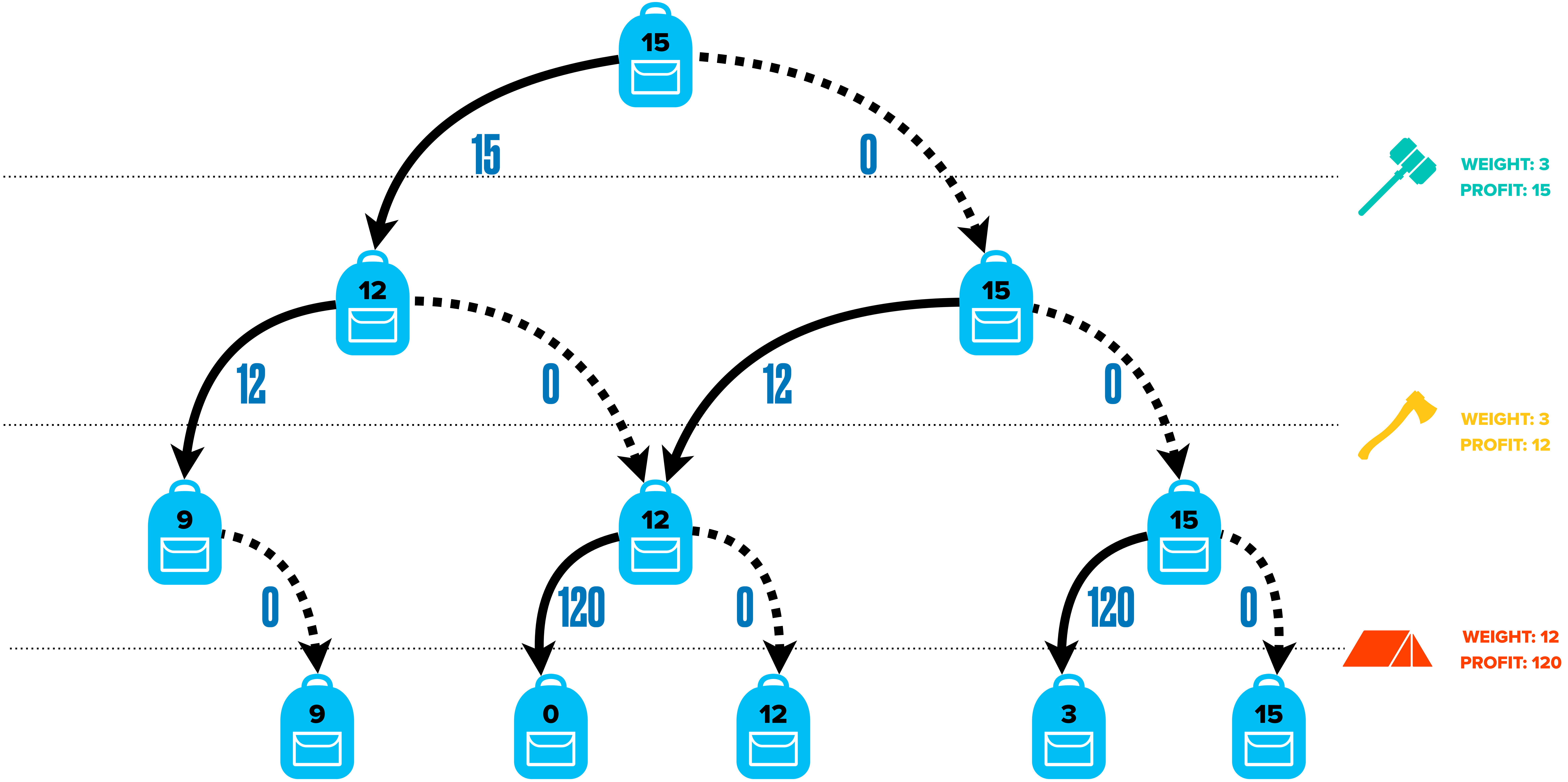
- **Transition Functions:** $\tau_i : S_i \times D_i \rightarrow S_{i+1}$
- **Transition Cost Function:** $h_i : S_i \times D_i \rightarrow \mathbb{R}$
- **Initial Value:** v_r

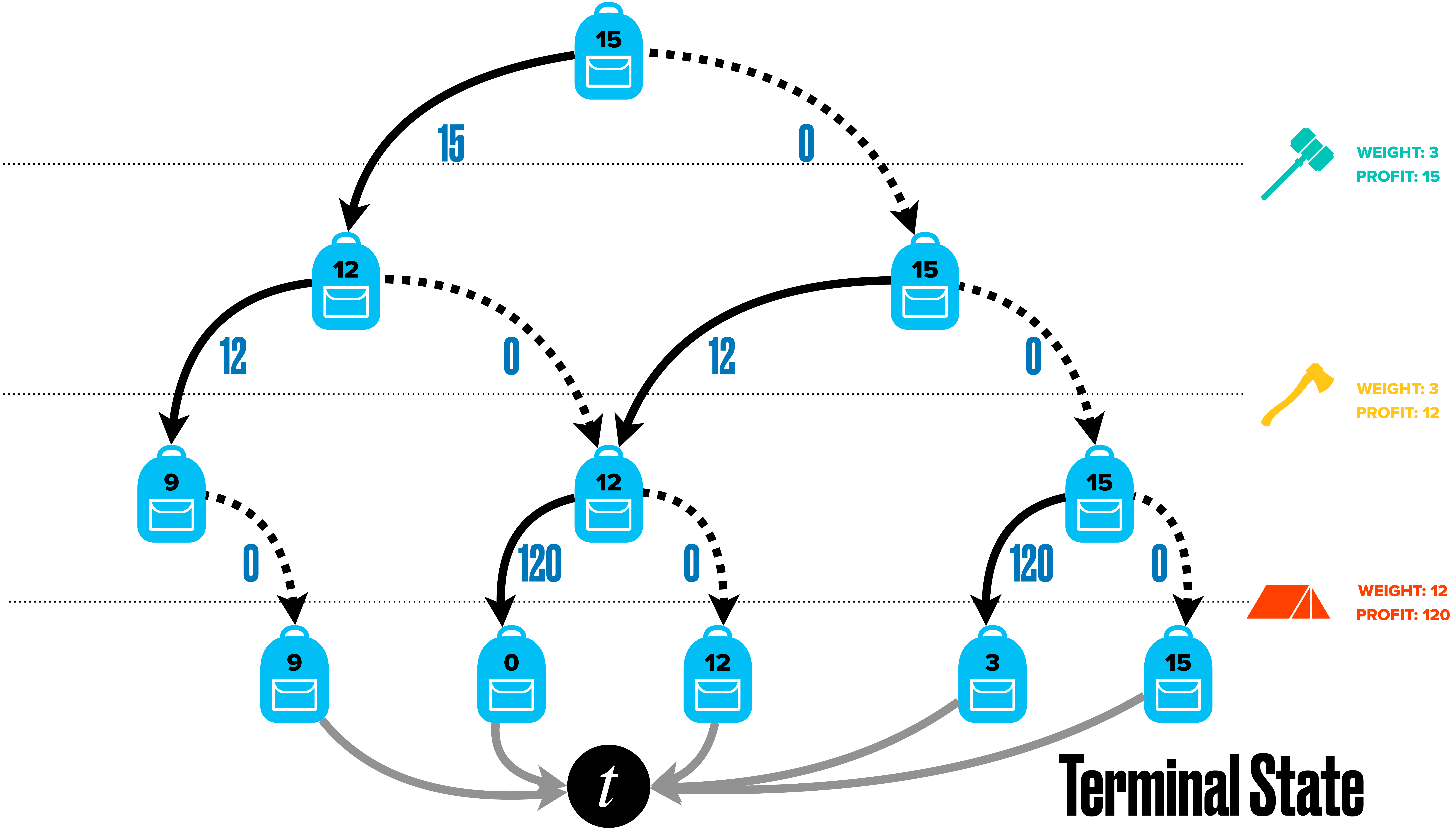
Part 2: Decision Diagrams

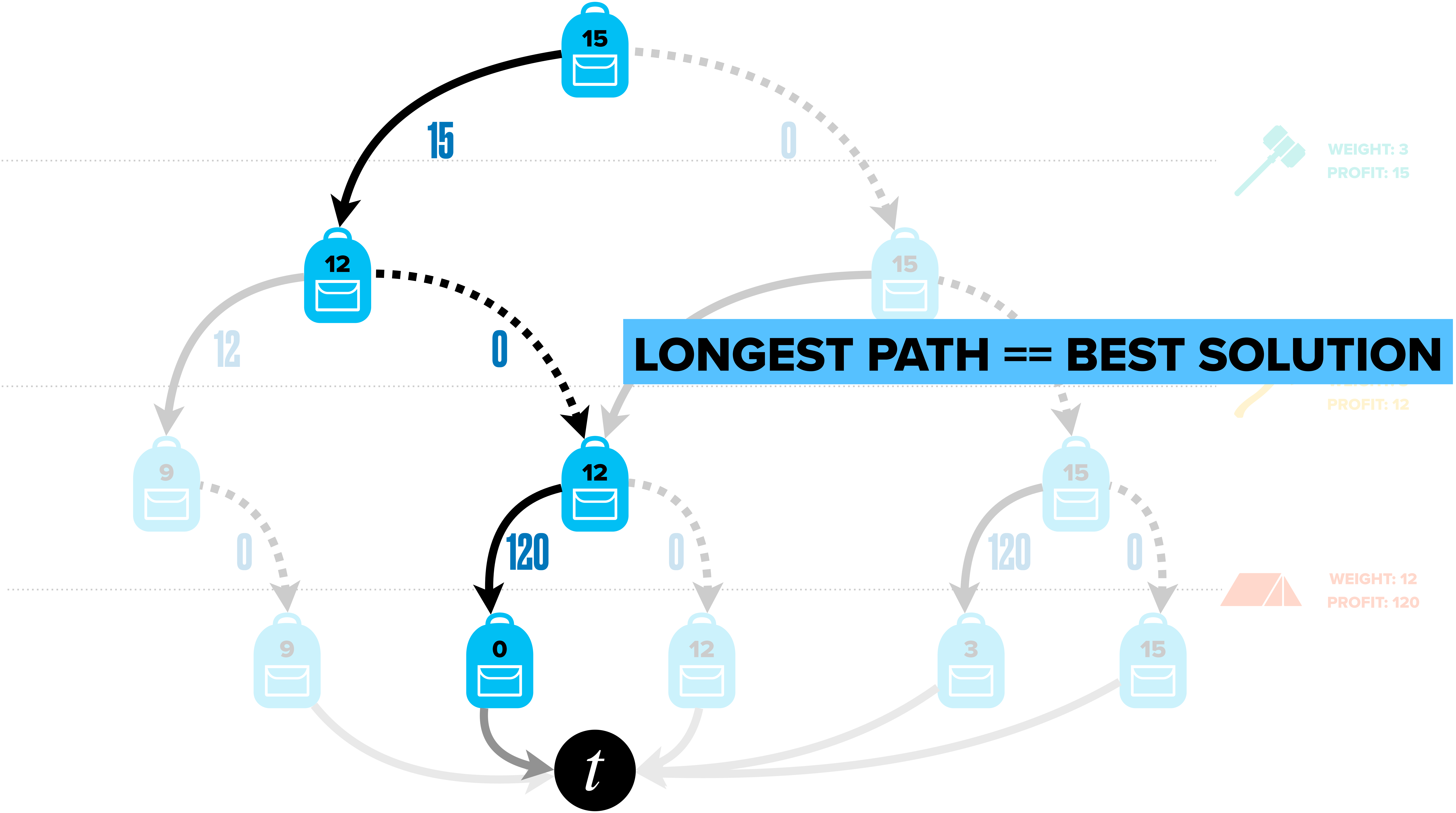
Decision Diagram

Formal Definition

Layered automaton encoding **sets of decision sequences**. In that graph, a path between the source and a terminal node traverses one node from each layer of the graph. In this structure, the **labels on the arcs** connecting two nodes are interpreted as the **assignment of a given value to a variable**: the value being the label of the arc and the variable, the one associated to the layer crossed by the arc.


15
15
0
12
15
12
0
12
0
9
12
15
0
120
0
120
0
9
0
12
3
15
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120


15
15
0
12
15
12
0
12
0
9
12
15
0
120
0
120
0
9
0
12
3
15
t
Terminal State
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120



LONGEST PATH == BEST SOLUTION

Problem

Some problems are just too hard to solve

DD is compact but it will not fit in a computer memory*

==> Solution: Control the size of the compiled DD

* Remember the TSP from the first lab on dynamic programming ?

Controlling the Size of the Compiled DD

Very important

Impose a maximum width *W* on the DD

- No layer can hold more than W nodes
- Prevents the exponential growth of the DD

Two approaches

- ***Delete*** the less promising nodes when there are too many
- ***Merge*** the less promising nodes when there are too many

We will use *both* approaches

… and use them in the context of a Branch-and-Bound

Very important

Purpose of each method

- ***Delete*** the less promising nodes → ***Derive lower bound***
- ***Merge*** the less promising nodes → ***Derive upper bound***

First method

WEIGHT: 3
PROFIT: 15

DELETE SOME NODES

Resulting DD is called

RESTRICTED DECISION DIAGRAM

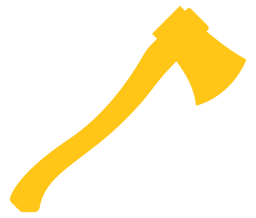
and provides a

LOWER BOUND

WEIGHT: 3
PROFIT: 12

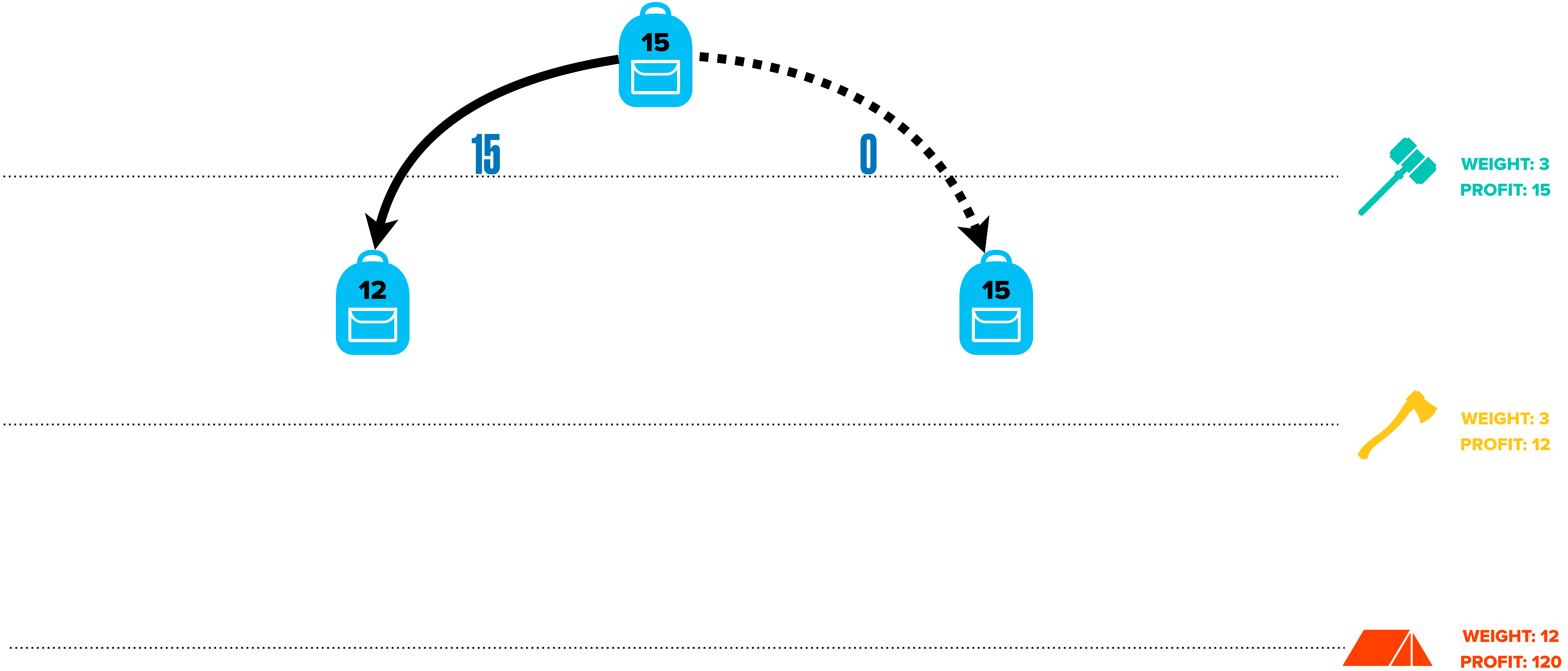
WEIGHT: 12
PROFIT: 120

WEIGHT: 3
PROFIT: 15



WEIGHT: 3
PROFIT: 12

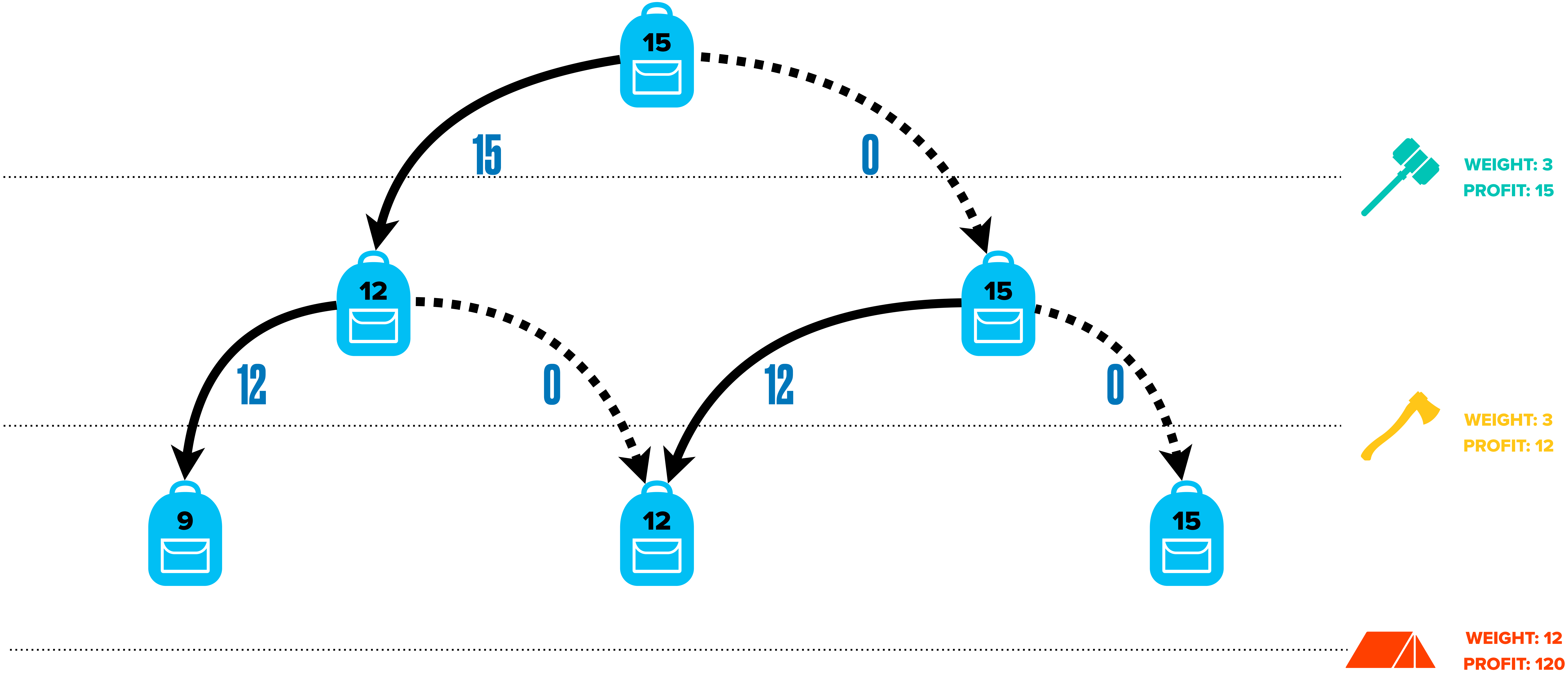
WEIGHT: 12
PROFIT: 120



WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

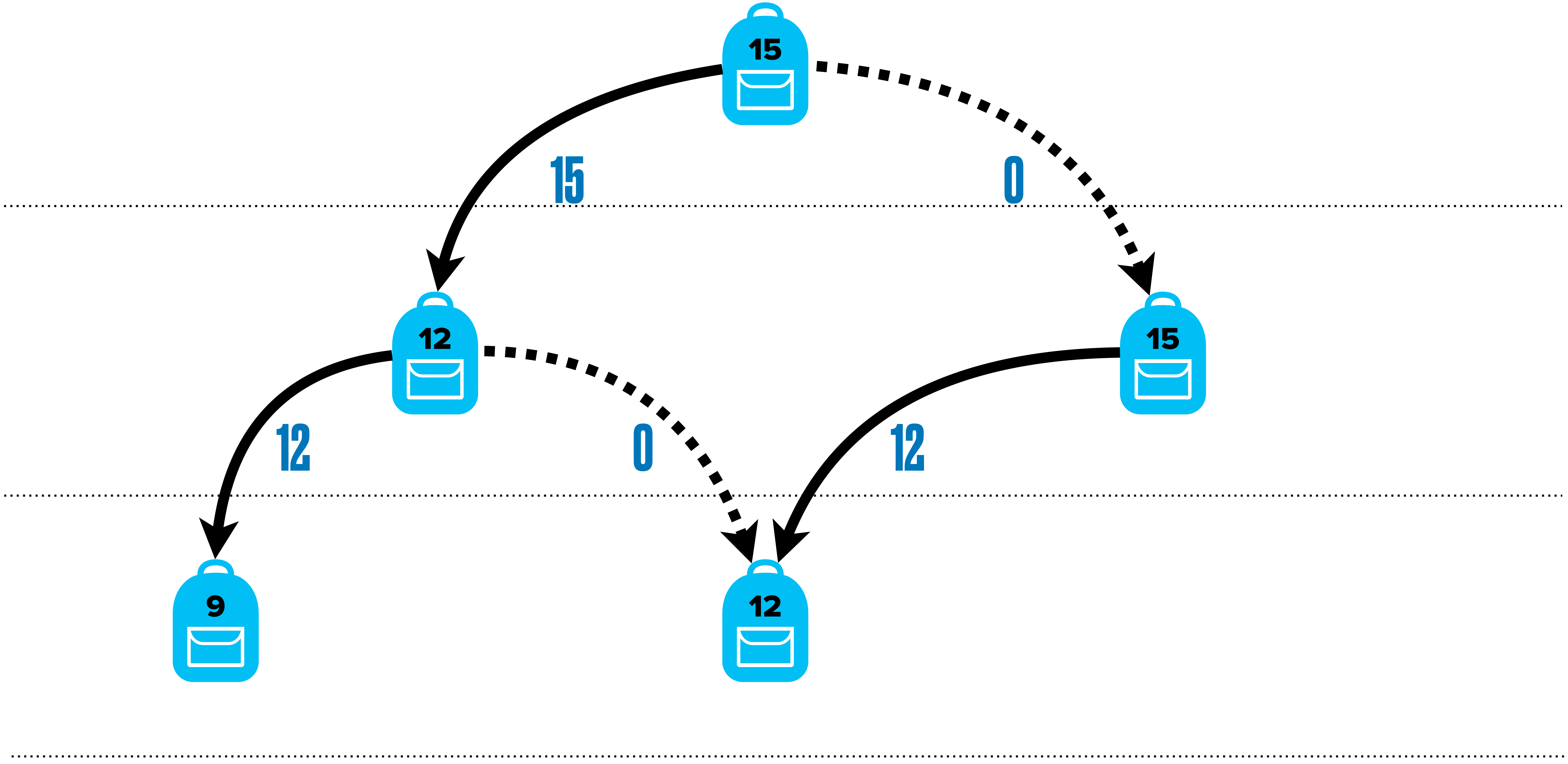
WEIGHT: 12
PROFIT: 120



WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

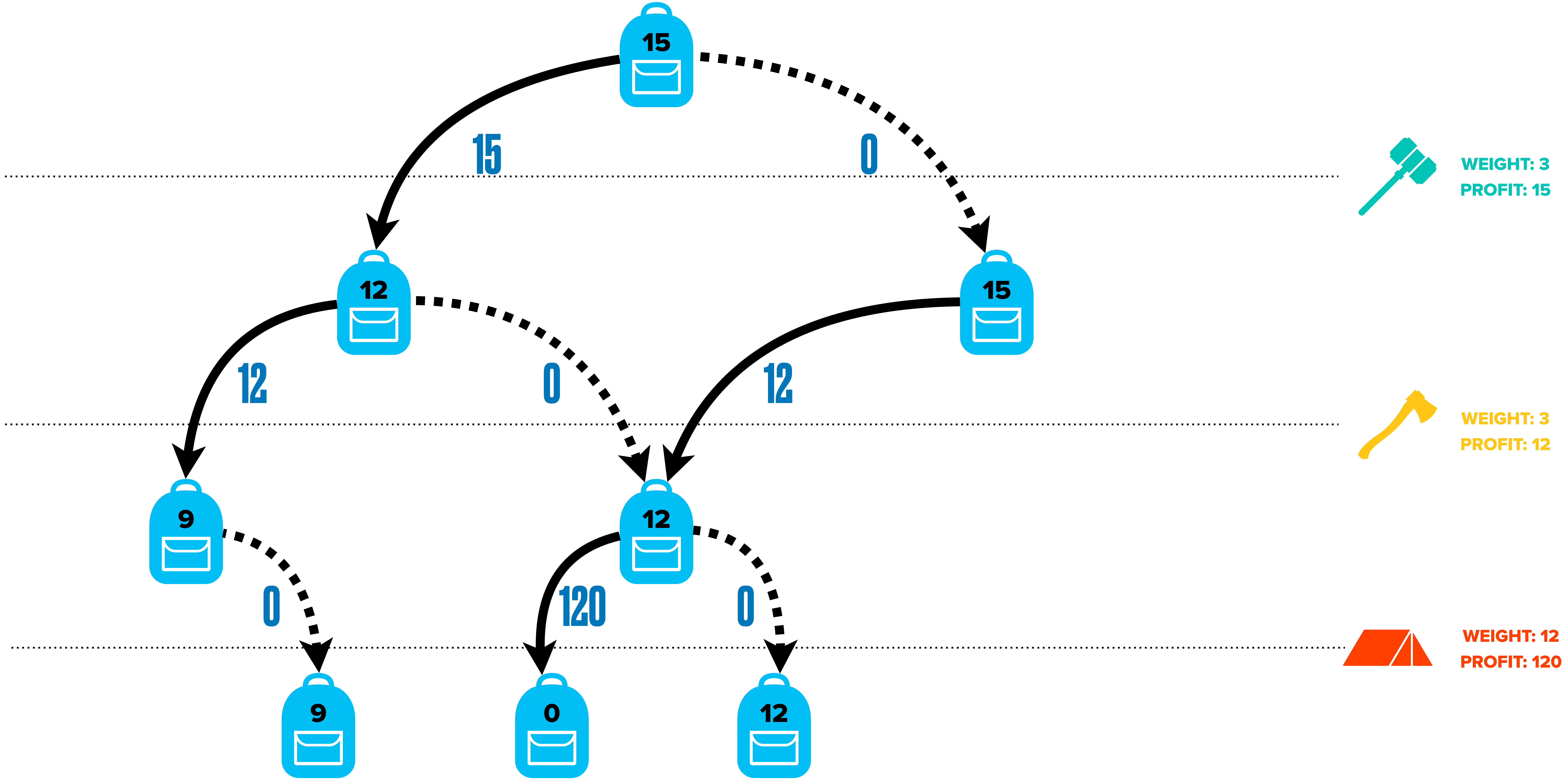
WEIGHT: 12
PROFIT: 120



WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

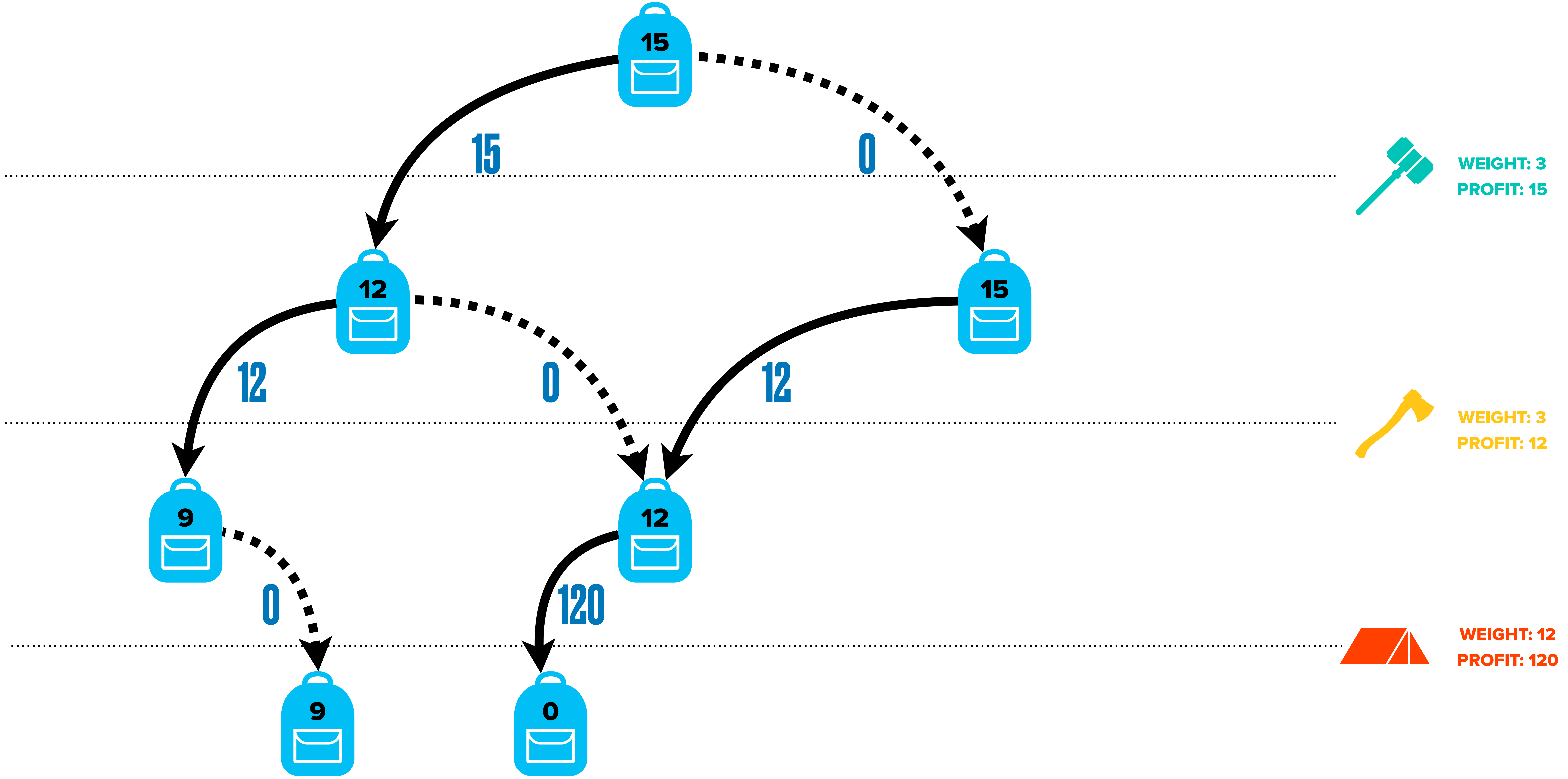
WEIGHT: 12
PROFIT: 120

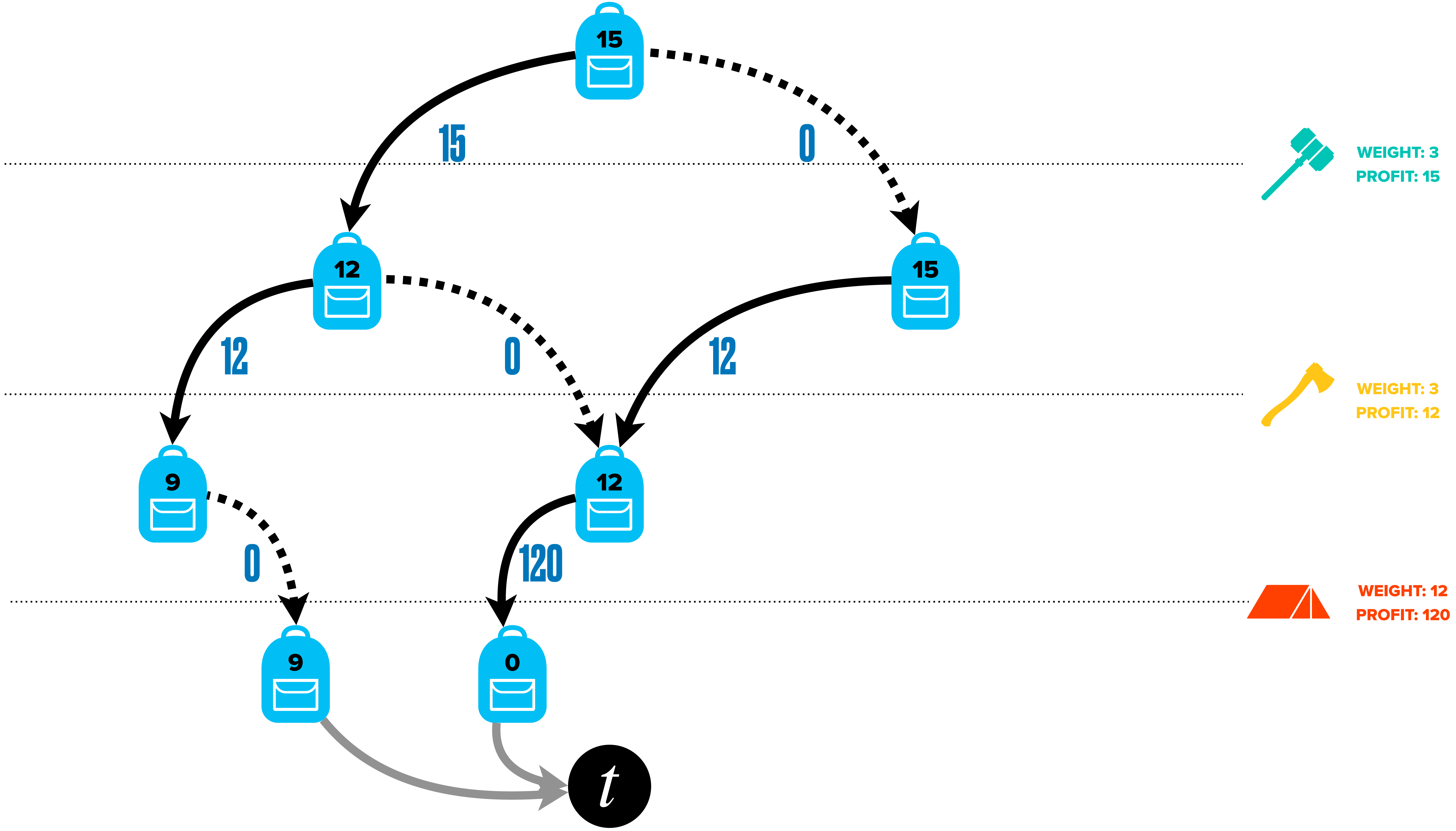


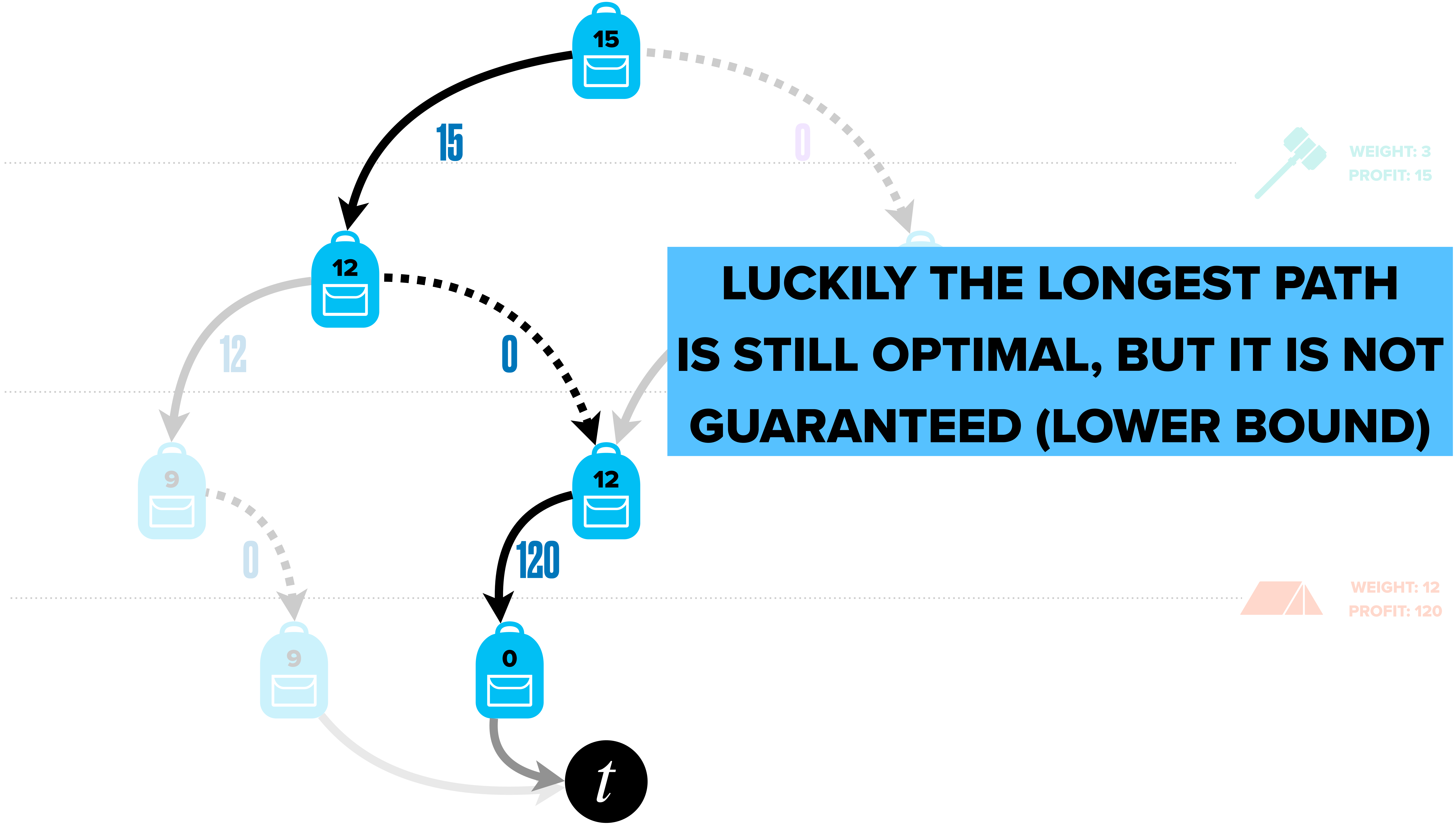
WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

WEIGHT: 12
PROFIT: 120


15
15
0
12
15
12
0
12
9
12
0
120
9
0
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120


15
15
0
12
15
12
0
12
9
12
0
120
9
0
t
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120



LUCKILY THE LONGEST PATH IS STILL OPTIMAL, BUT IT IS NOT GUARANTEED (LOWER BOUND)

First Method

Restricted Decision Diagrams

- Some paths are missing from the DD
- Longest path is guaranteed to be a valid solution
- Longest path is not guaranteed to be the optimal solution **(LOWER BOUND)**

Formally

$$Sol(\underline{B}) \subseteq Sol(P)$$

The set of all solutions encoded
in the restricted DD $\underline{B}$

The set of all solutions
to the problem

Second method

WEIGHT: 3
PROFIT: 15

MERGE SOME NODES

Resulting DD is called

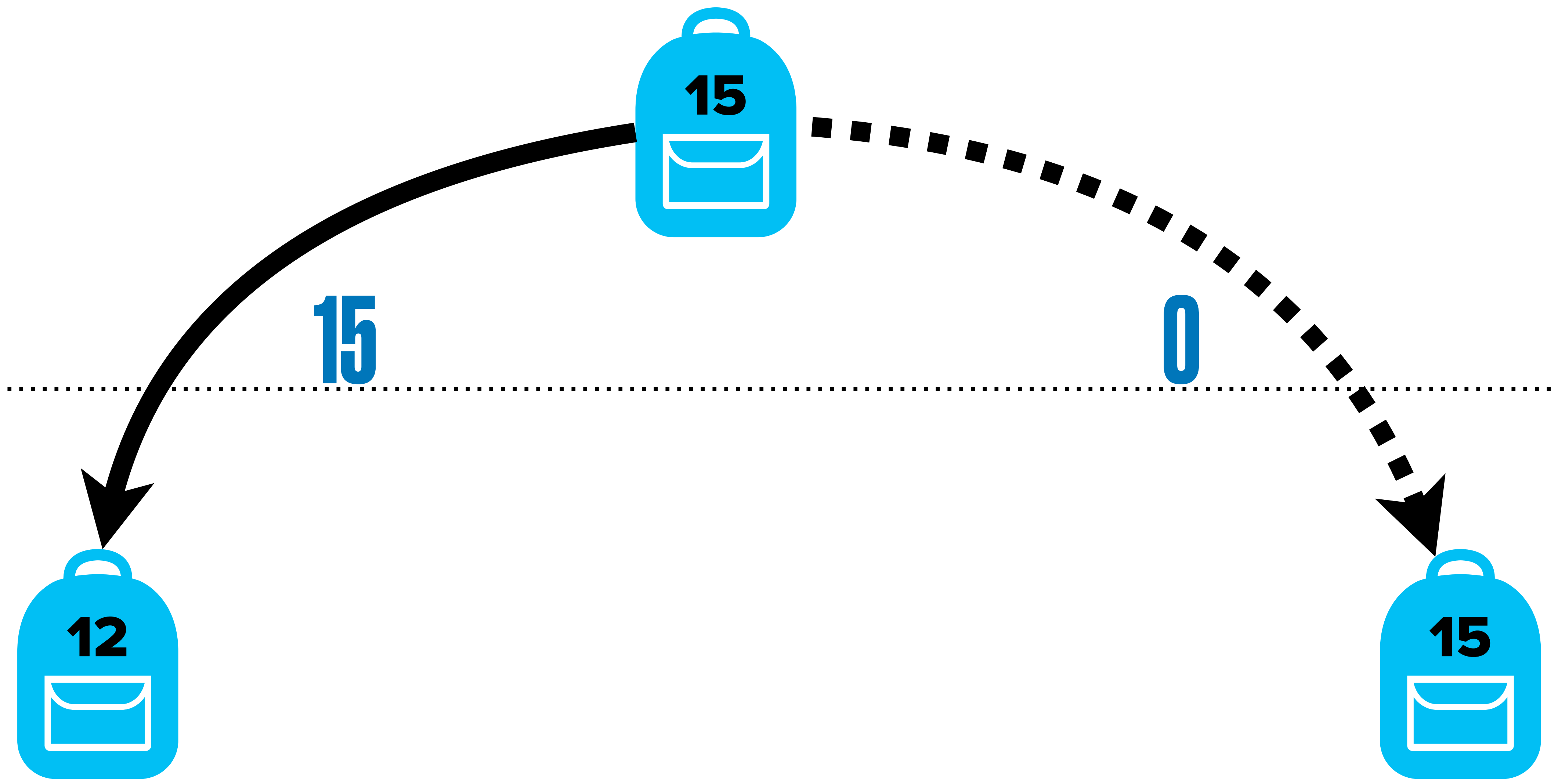
RELAXED DECISION DIAGRAM

and provides a

UPPER BOUND

WEIGHT: 3
PROFIT: 12

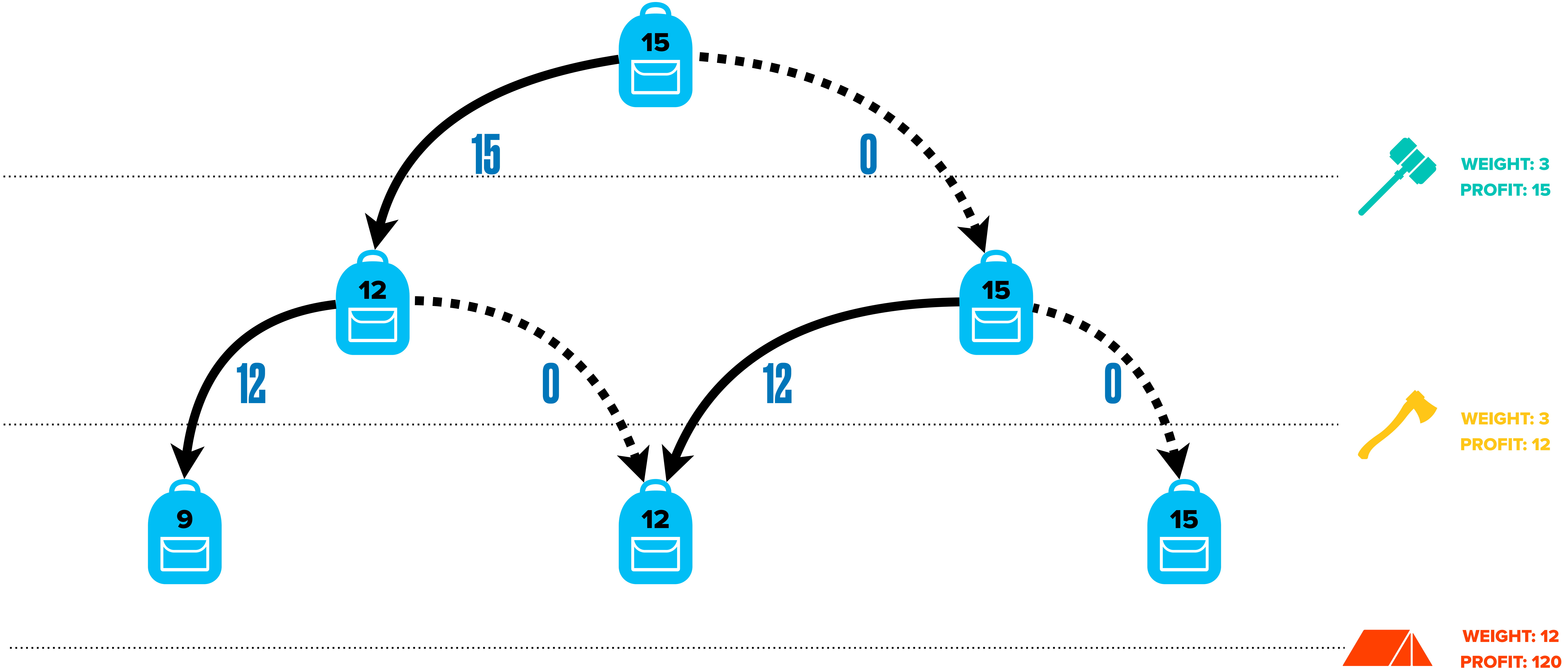
WEIGHT: 12
PROFIT: 120

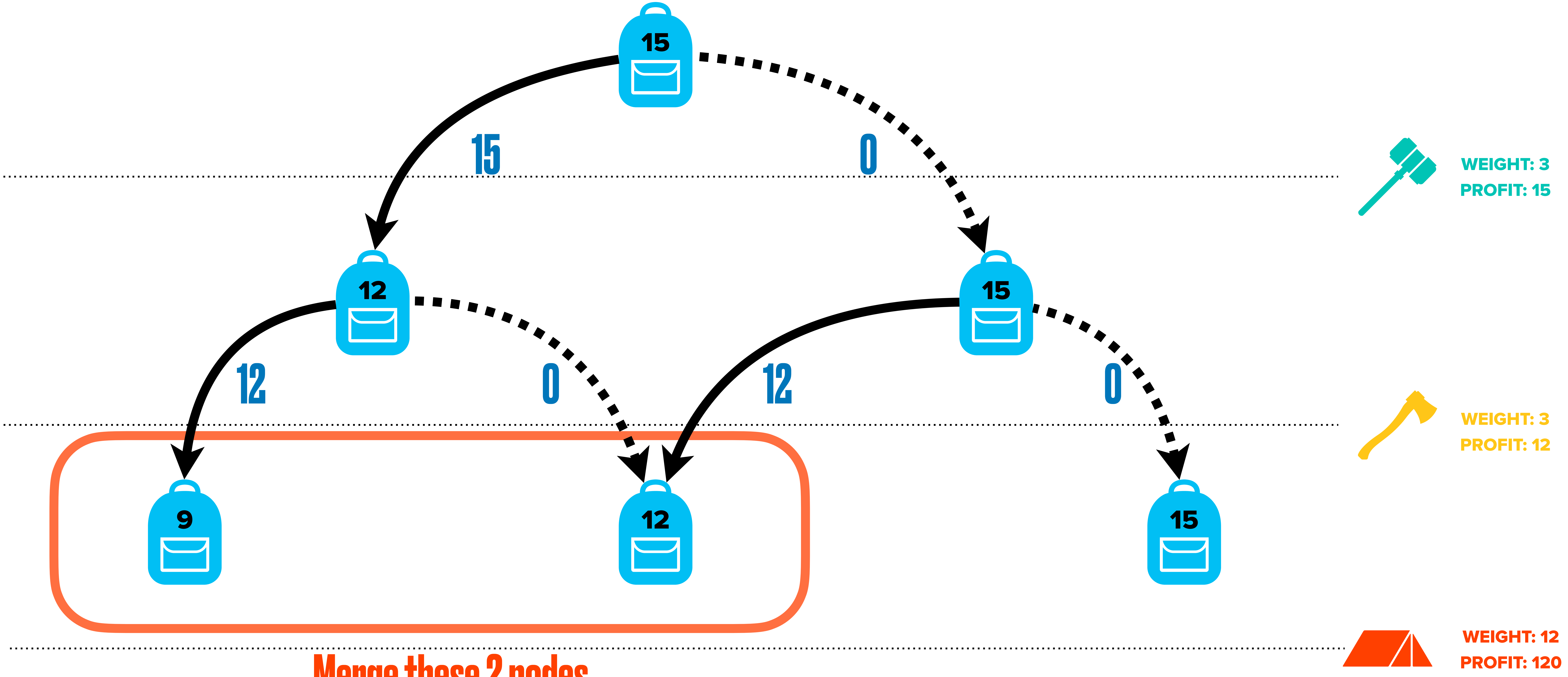


WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

WEIGHT: 12
PROFIT: 120

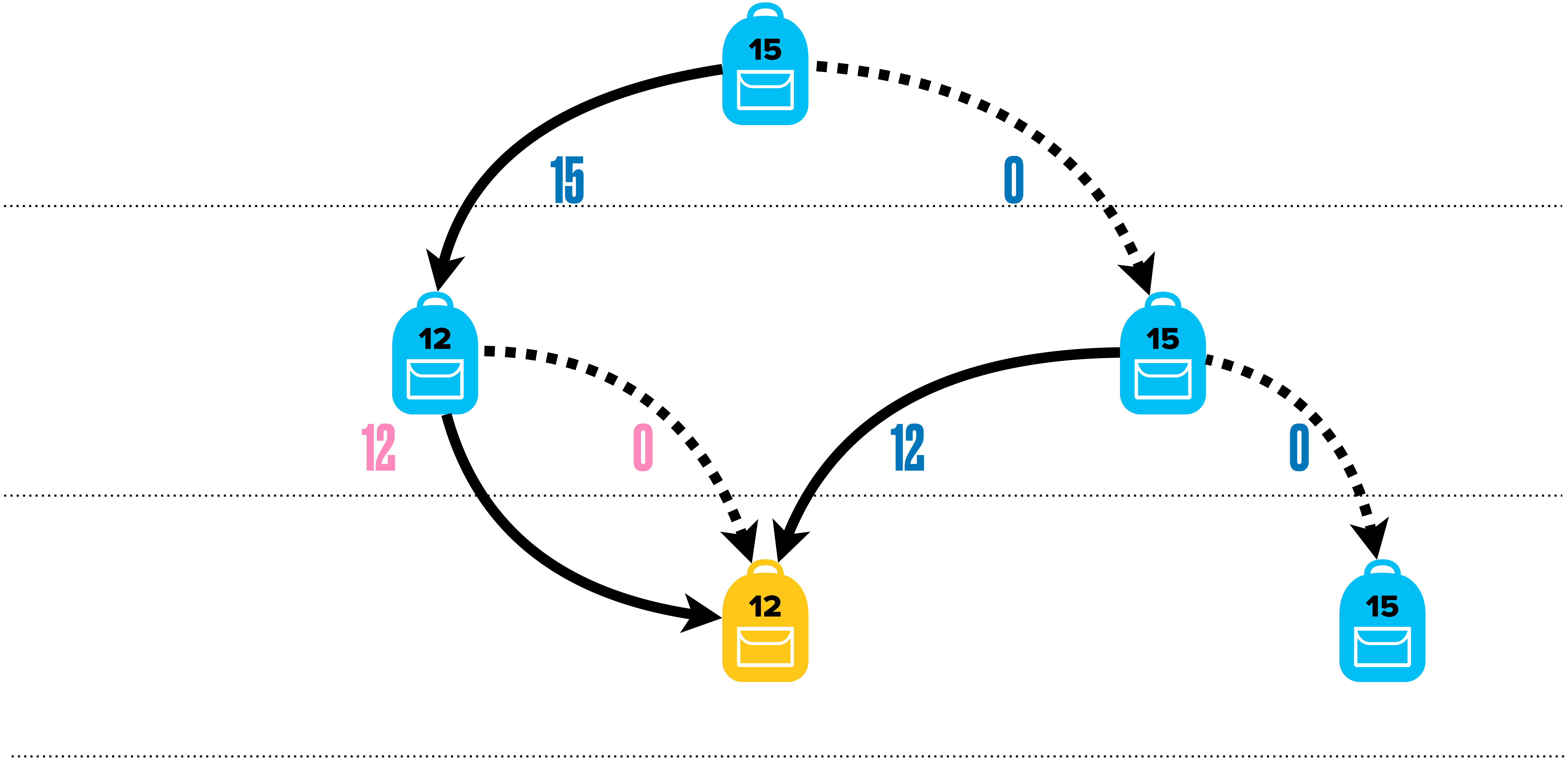

15
15
0
12
15
12
0
12
0
9
12
15
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120



Merge these 2 nodes

Requires 2 operators

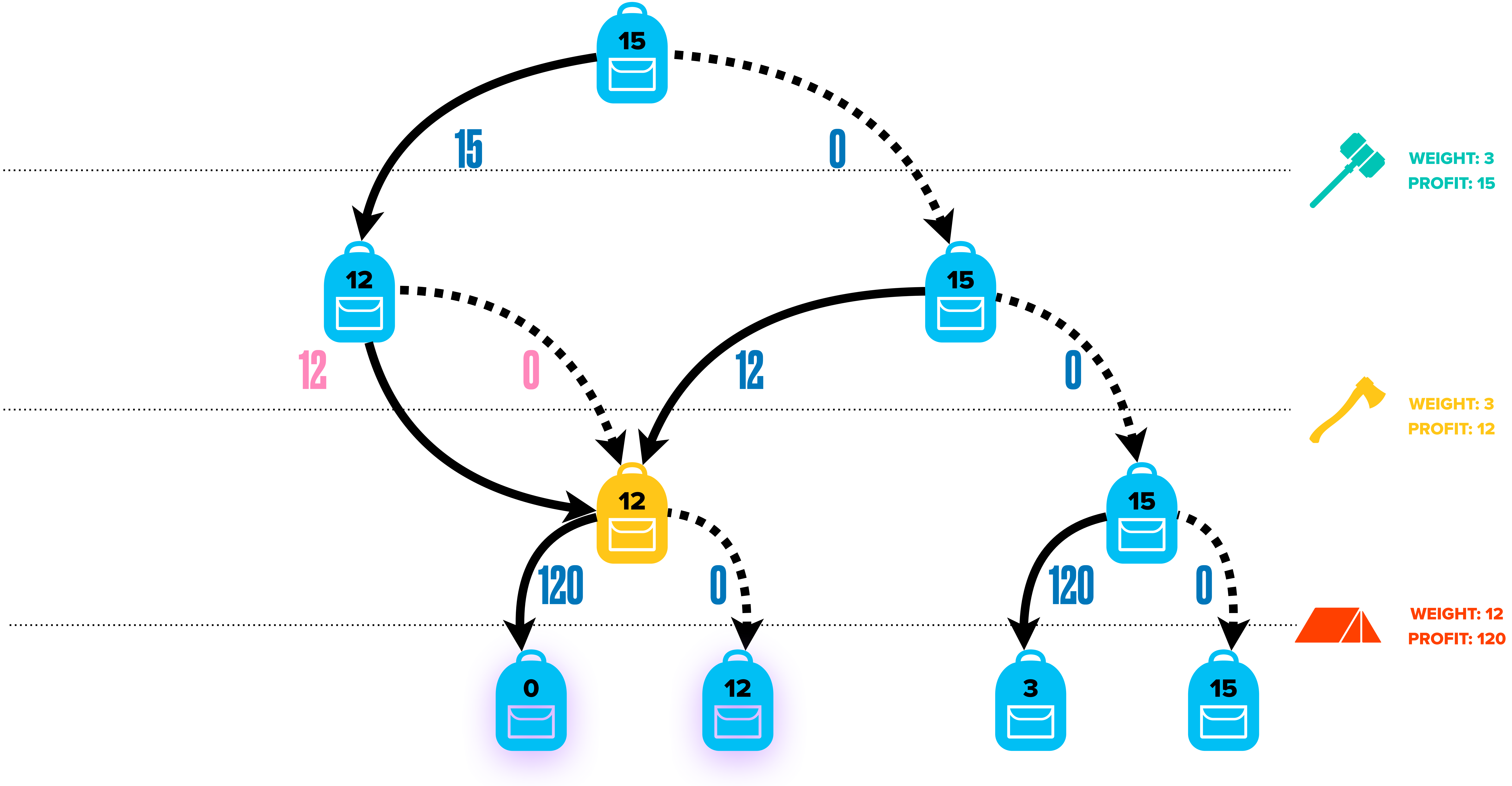
$\oplus\ (states)$ and $\Gamma(arc)$

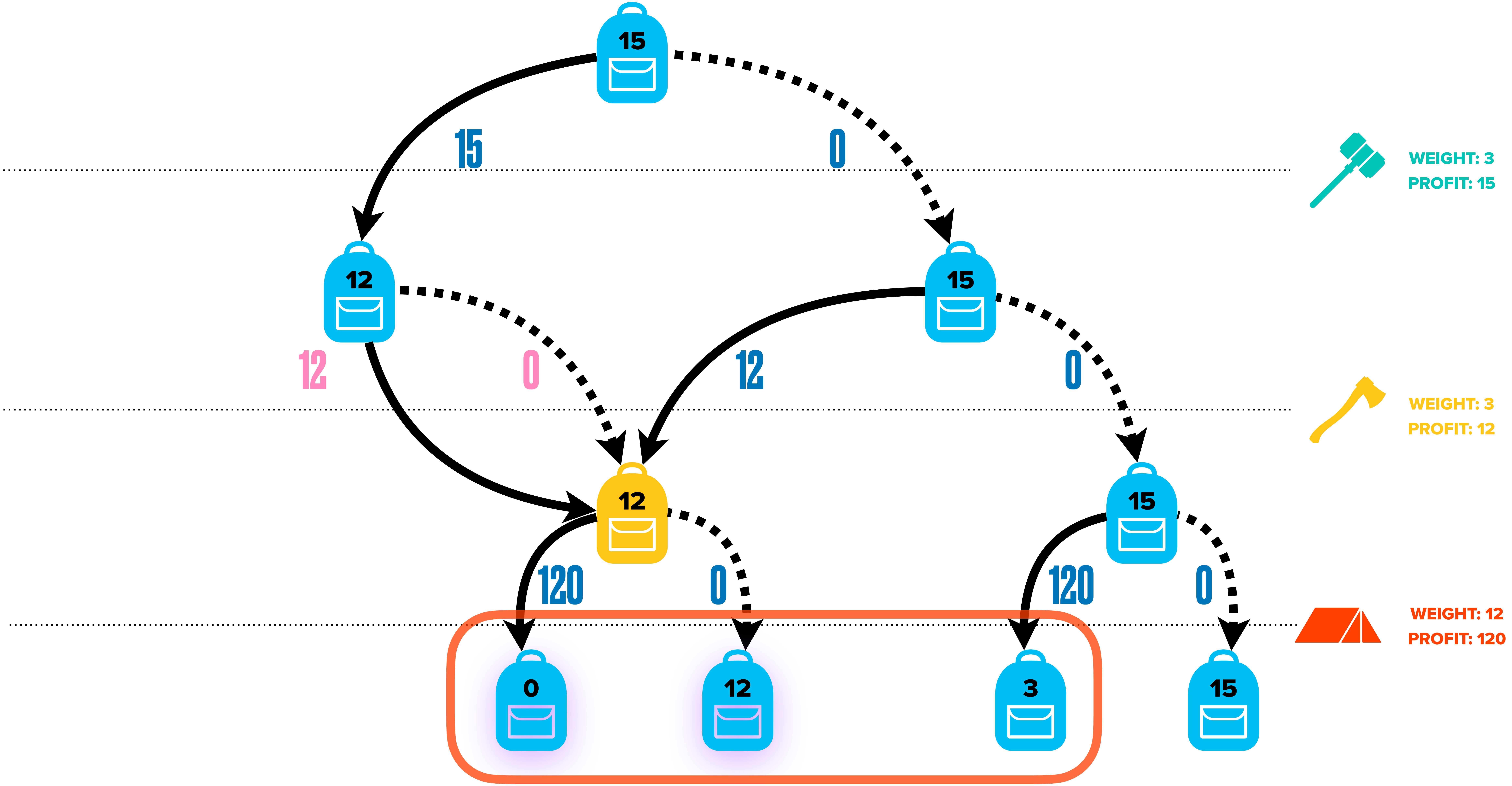


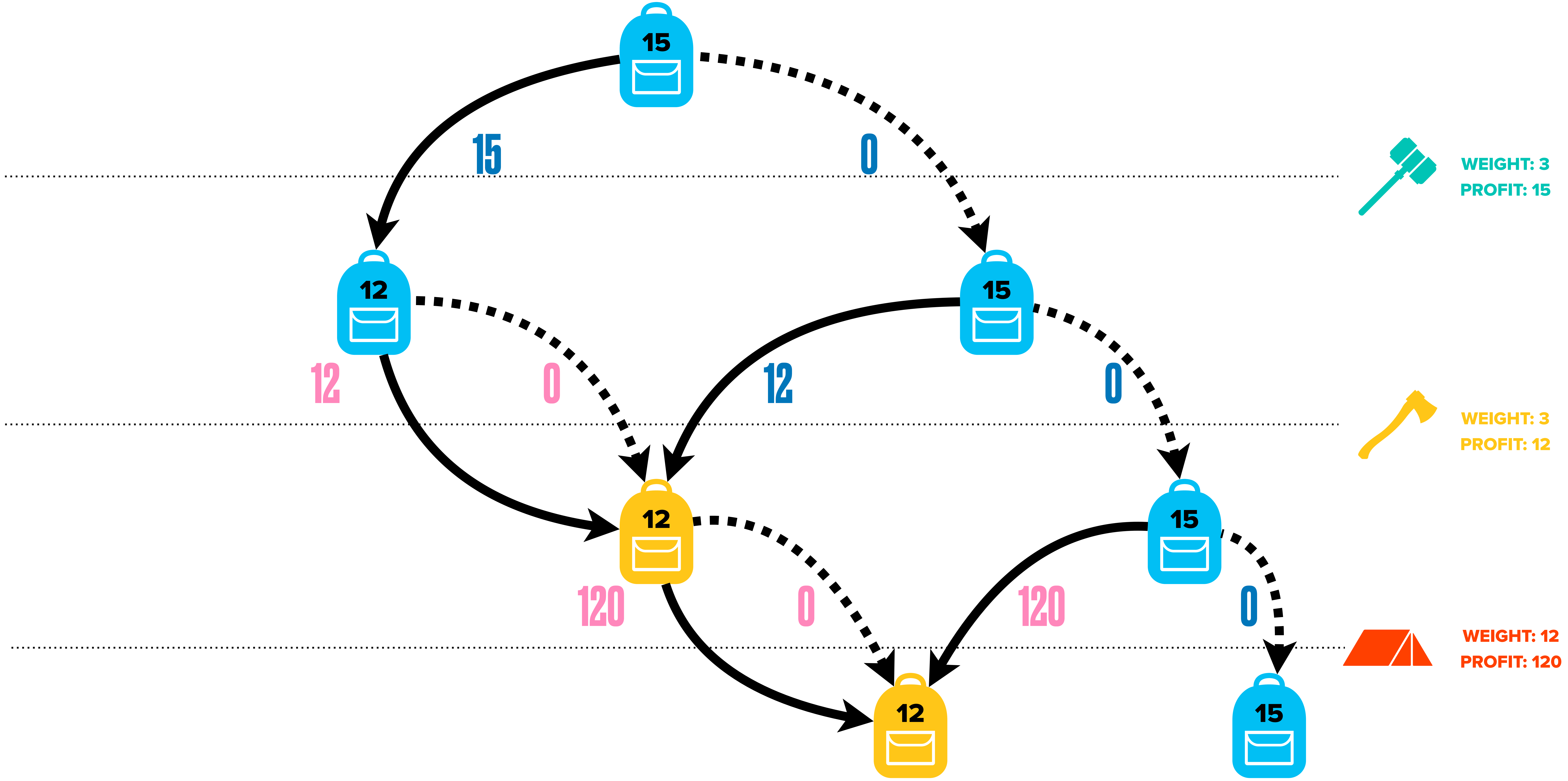
WEIGHT: 3
PROFIT: 15

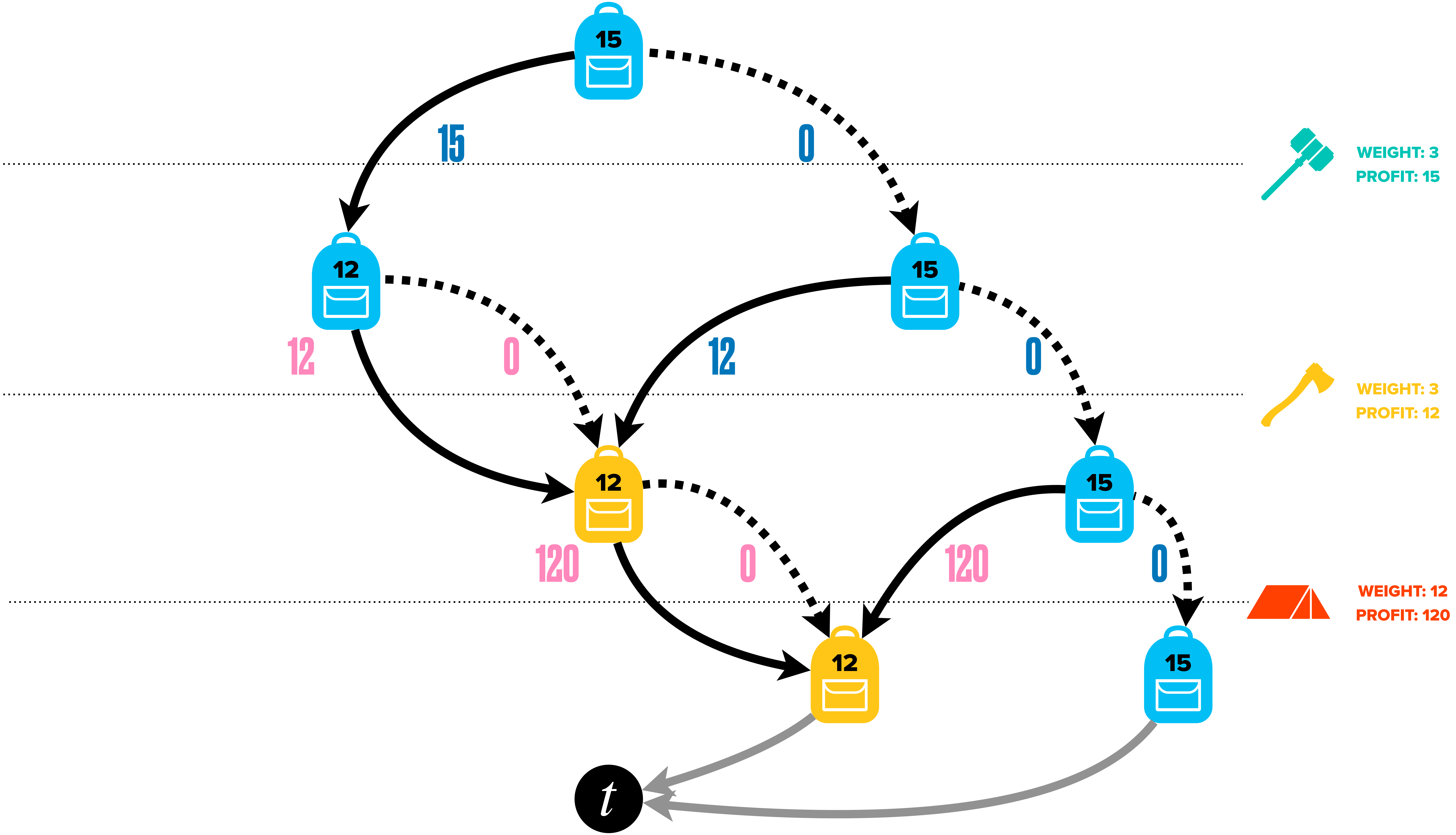
WEIGHT: 3
PROFIT: 12

WEIGHT: 12
PROFIT: 120


15
15
0
12
15
12
0
12
0
12
120
0
15
120
0
0
12
3
15
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120


15
15
0
WEIGHT: 3
PROFIT: 15
12
15
12
0
12
0
WEIGHT: 3
PROFIT: 12
12
15
120
0
120
0
WEIGHT: 12
PROFIT: 120
0
12
3
15


15
15
0
12
15
12
0
12
0
12
15
120
0
120
0
12
15
WEIGHT: 3
PROFIT: 15
WEIGHT: 3
PROFIT: 12
WEIGHT: 12
PROFIT: 120

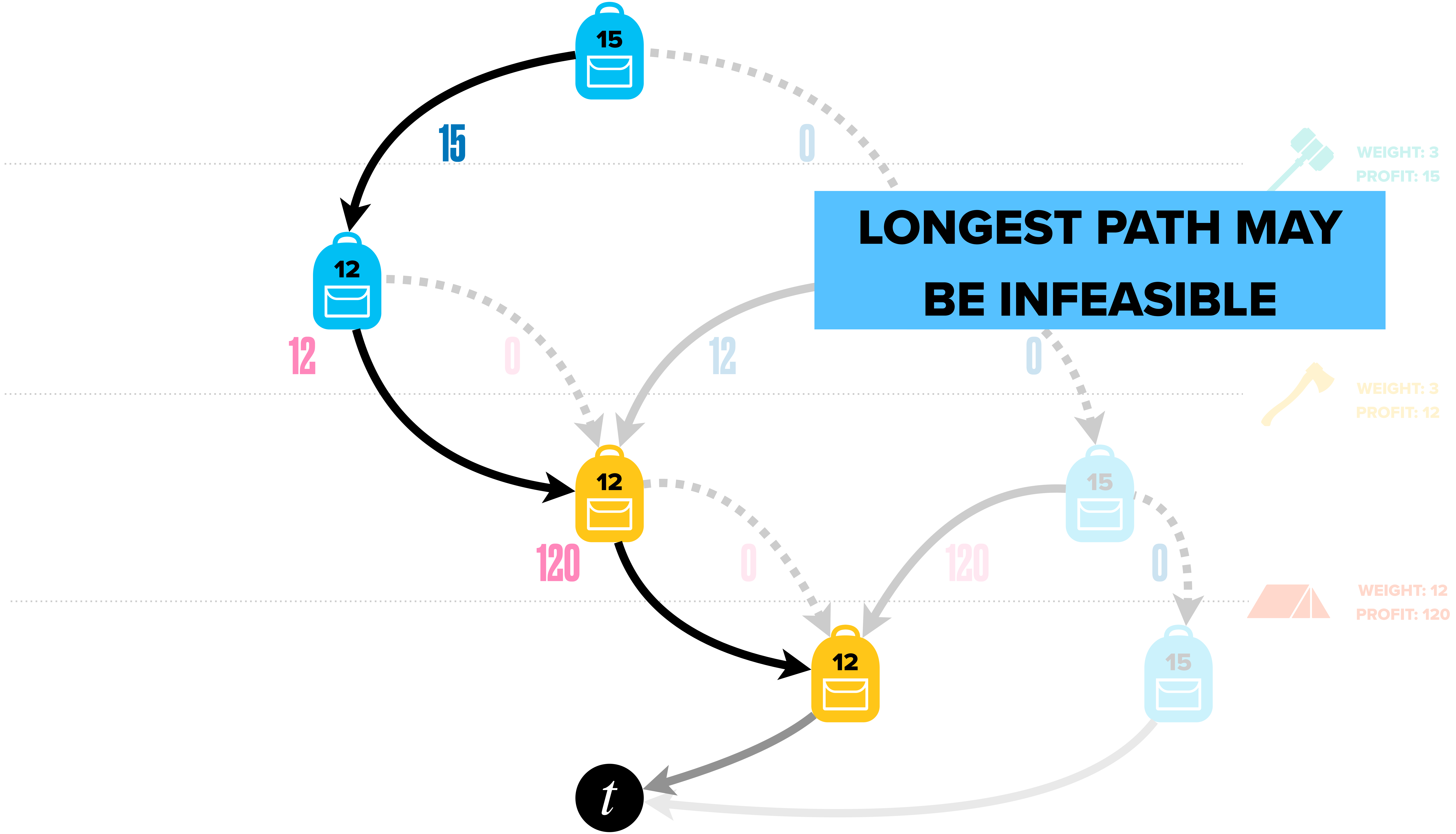


WEIGHT: 3
PROFIT: 15

WEIGHT: 3
PROFIT: 12

WEIGHT: 12
PROFIT: 120

LONGEST PATH MAY BE INFEASIBLE



Second Method

Relaxed Decision Diagrams

- Requires two additional operators to merge nodes $\oplus$ and relax arcs Γ
- Longest path is ***not*** guaranteed to be a valid solution
- Longest path is guaranteed to be at least as long as the optimal solution **(UPPER BOUND)**

Formally

$$Sol(\overline{B}) \supseteq Sol(P)$$

The set of all solutions encoded in the restricted DD $\overline{B}$

The set of all solutions to the problem

There may be more paths in the DD than exists actual solutions

Recap’

So far we have

- Restricted DD yield feasible solution (lower bound)

$$Sol(\underline{\mathscr{B}}) \subseteq Sol(\mathscr{P})$$

- Relaxed DD yield (possibly) non-feasible solution (upper bound)

$$Sol(\overline{\mathscr{B}}) \supseteq Sol(\mathscr{P})$$

where $Sol(\cdot)$ denotes the set of solutions, $\underline{\mathscr{B}}$ is a restricted DD, $\overline{\mathscr{B}}$ is a relaxed DD, and $\mathscr{P}$ is the original problem

Where do we go from there ?

Use these two ideas to derive a B-a-B framework that is able to find the longest path in the original MDD (that was to large to be build initially)

Compile bounded DD

Algorithm Top Down Compilation of a bounded-width DD

```
Input: a DP-model $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$
Input: a maximum layer width $W$
$L_0 \leftarrow \{r\}$
for $i \in \{0 \dots n-1\}$ do
    for $u \in L_i, d \in D_i$ do
        $u' \leftarrow$ a node associated with state $\tau_i(\sigma(u), d)$
        if $\sigma(u') \neq \bot$ then
            $U \leftarrow U \cup \{u'\}$
            $L_{i+1} \leftarrow L_{i+1} \cup \{u'\}$
            $a \leftarrow u \xrightarrow{d} u'$
            $v(a) \leftarrow h_i(\sigma(u), d)$
            $A \leftarrow A \cup \{a\}$
        end if
    end for
    if $|L_{i+1}| > W$ then
        Restrict or Relax the layer to get at most $W$ nodes
    end if
end for
```

Compile bounded DD

Algorithm Top Down Compilation of a bounded-width DD

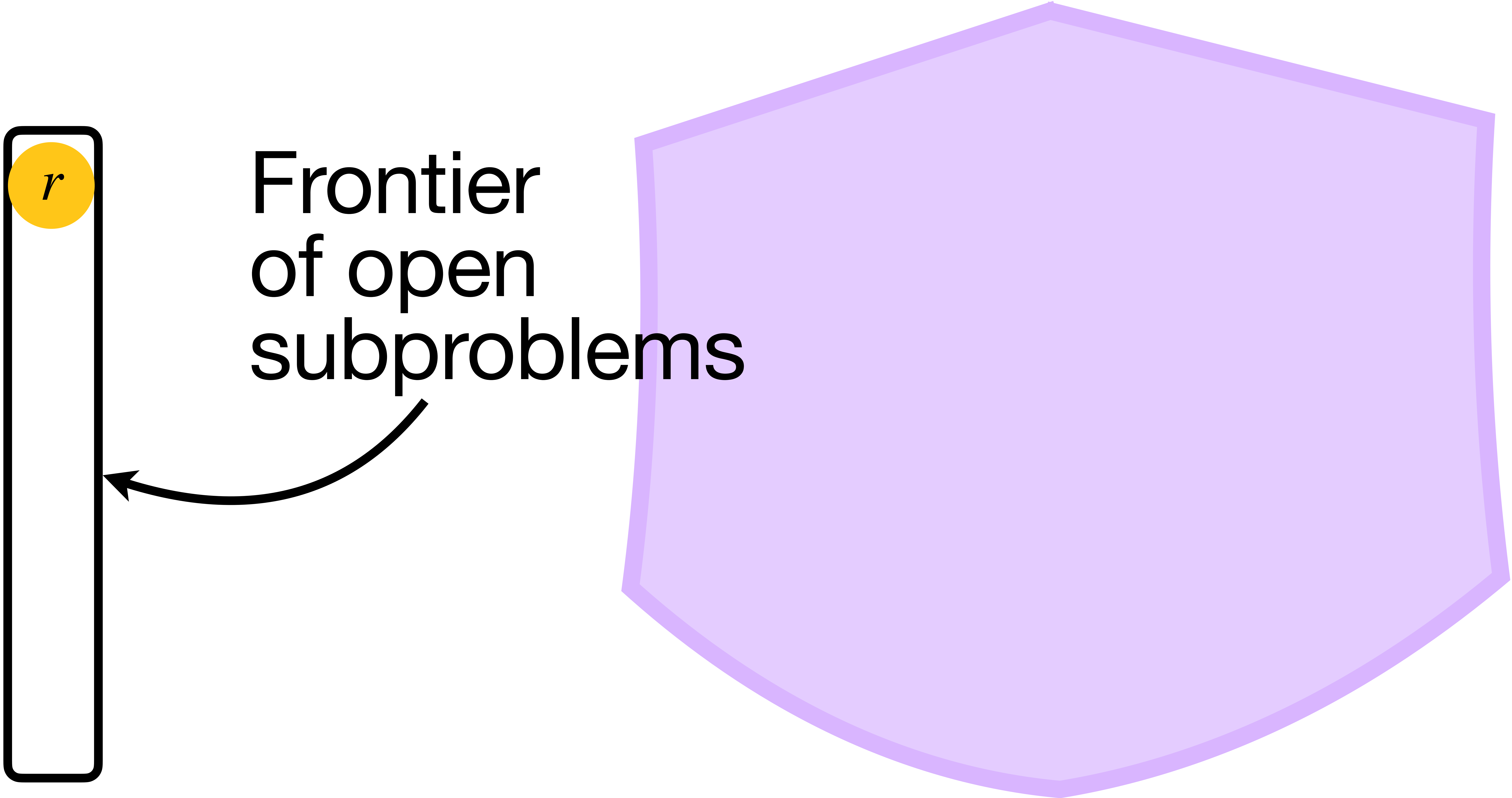
```
Input: a DP-model $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$
Input: a maximum layer width $W$
$L_0 \leftarrow \{r\}$
for $i \in \{0 \dots n-1\}$ do
    for $u \in L_i, d \in D_i$ do
        $u' \leftarrow$ a node associated with state $\tau_i(\sigma(u), d)$
        if $\sigma(u') \neq \bot$ then
            $U \leftarrow U \cup \{u'\}$
            $L_{i+1} \leftarrow L_{i+1} \cup \{u'\}$
            $a \leftarrow u \xrightarrow{d} u'$
            $v(a) \leftarrow h_i(\sigma(u), d)$
            $A \leftarrow A \cup \{a\}$
        end if
    end for
    if $|L_{i+1}| > W$ then
        Restrict or Relax the layer to get at most $W$ nodes
    end if
end for
```

Might be a subproblem

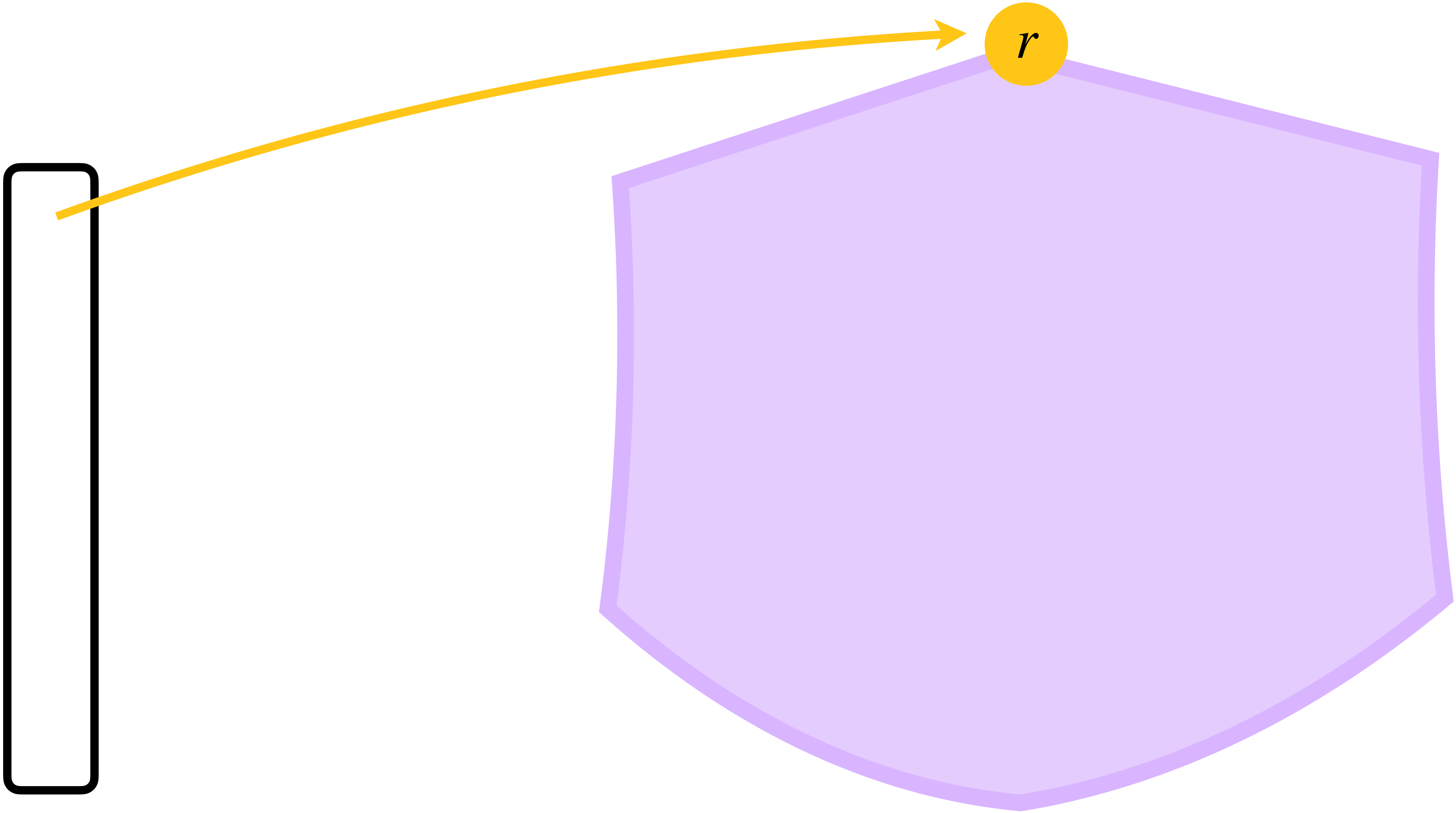
Branch-and-Bound

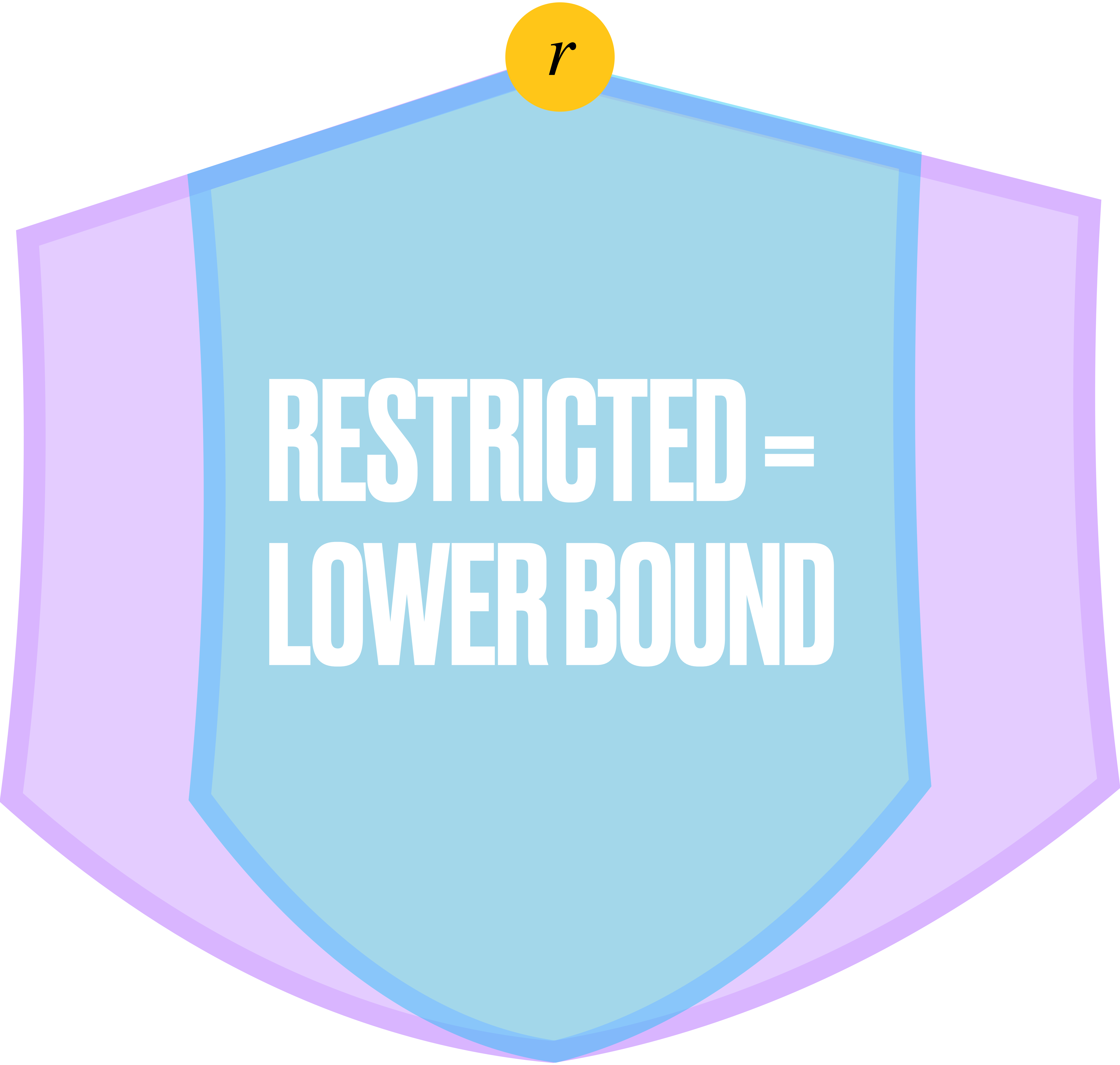
```
Algorithm Branch-And-Bound with DD
Input: a DP-model $\mathcal{P} = \langle S, r, t, \bot, v_r, \tau, h \rangle$
Input: a node merging operator $\oplus$
Input: an arc relaxation operator $\Gamma$
Create node $r$ and add it to $Fringe$
$\underline{x} \leftarrow \bot$
$\underline{v} \leftarrow -\infty$
while $Fringe$ is not empty do
    $u \leftarrow Fringe.pop()$
    $\underline{\mathcal{B}} \leftarrow Restricted(u)$
    if $v^*(\underline{\mathcal{B}}) > \underline{v}$ then
        $\underline{v} \leftarrow v^*(\underline{\mathcal{B}})$
        $\underline{x} \leftarrow x^*(\underline{\mathcal{B}})$
    end if
    if $\underline{\mathcal{B}}$ is not exact then
        $\overline{\mathcal{B}} \leftarrow Relaxed(u, \oplus, \Gamma)$
        if $v^*(\overline{\mathcal{B}}) > \underline{v}$ then
            for all $u' \in \overline{\mathcal{B}}.exact\_cutset()$ do
                $Fringe.add(u')$
            end for
        end if
    end if
end while
return $(\underline{x}, \underline{v})$
```

HUGE DD
EXACT REPRESENTATION



Frontier of open subproblems


r



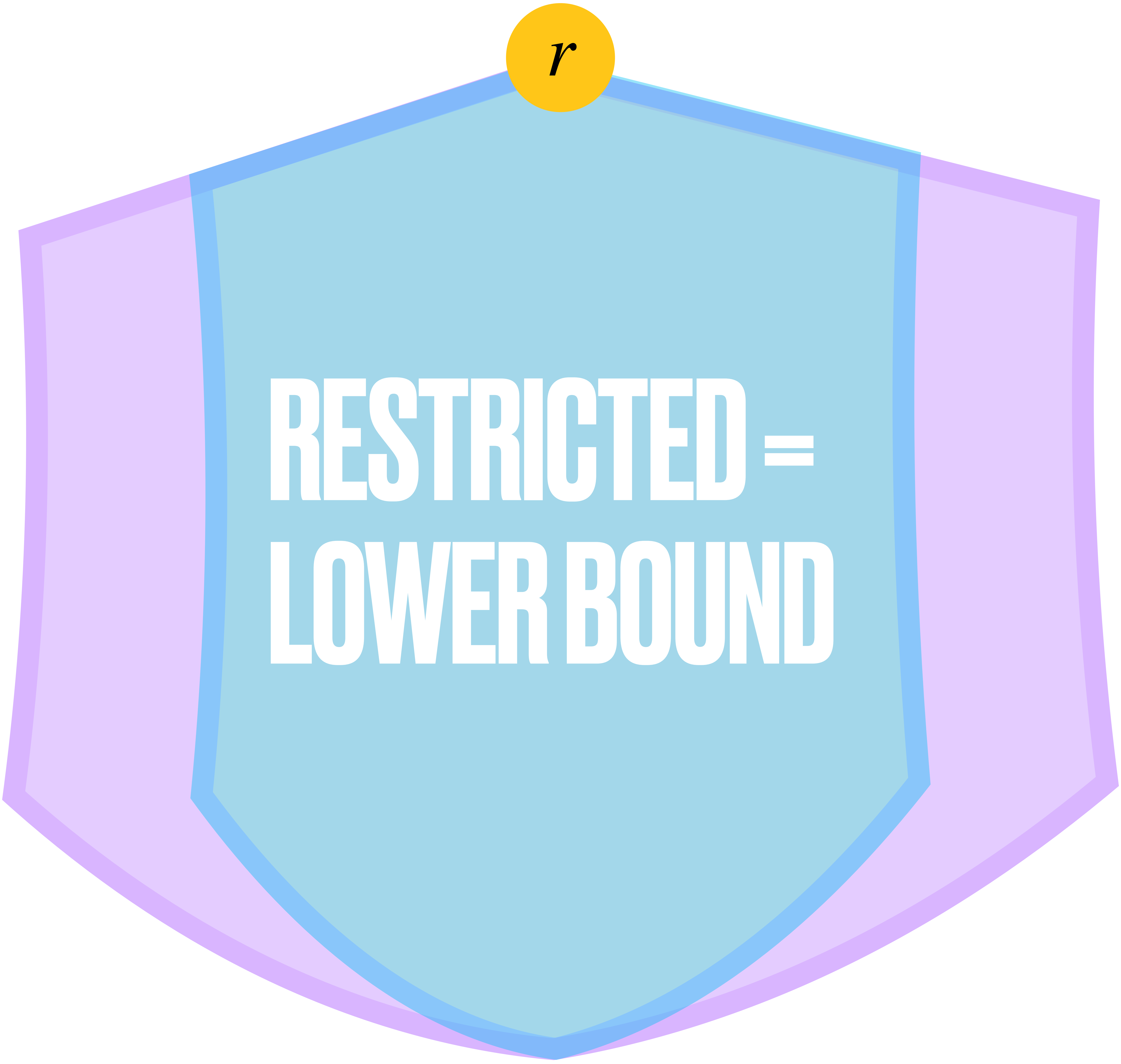
RESTRICTED = LOWER BOUND

1

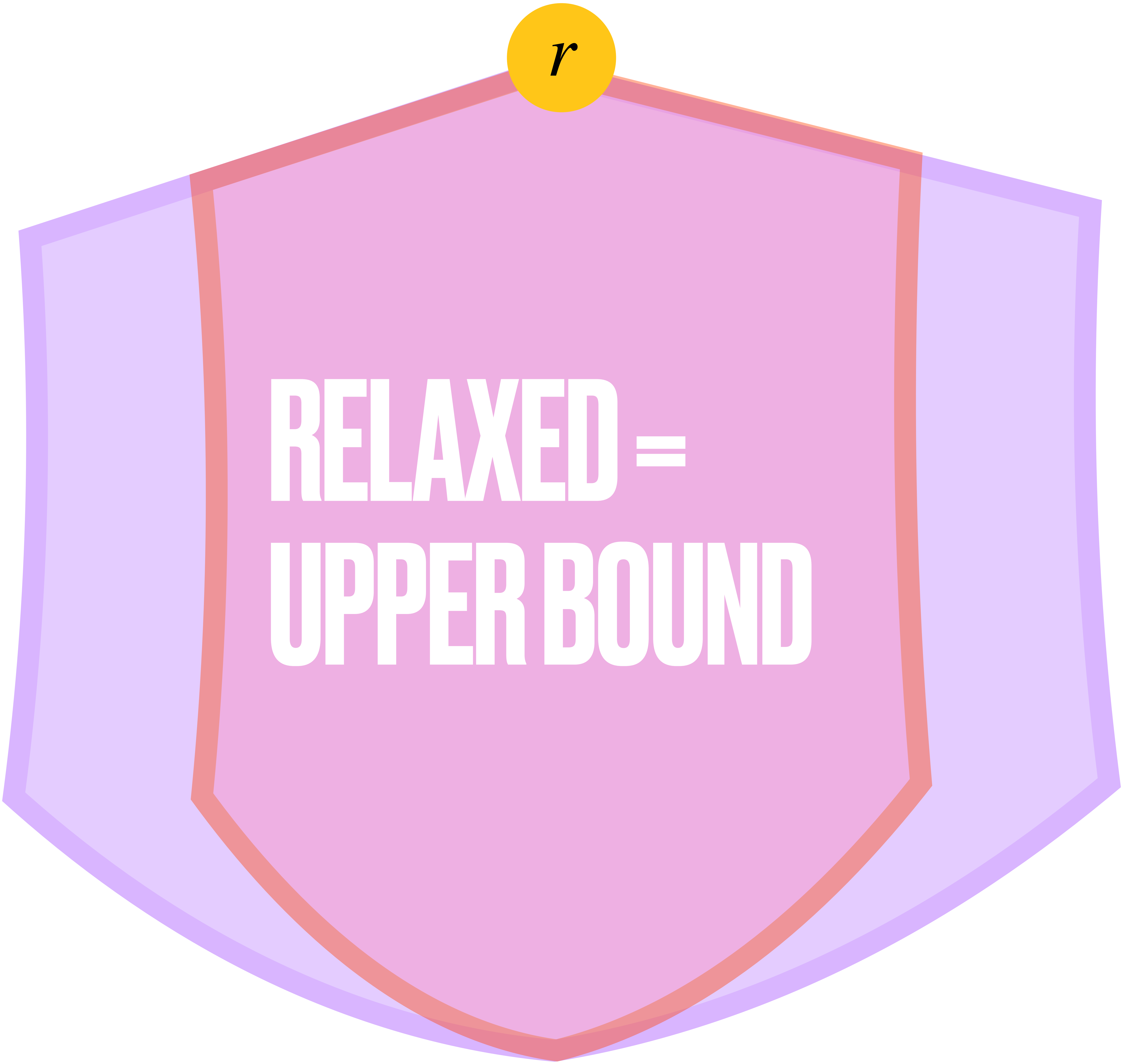
Does it improve the best known solution ?

2

Is it exact ?



RESTRICTED = LOWER BOUND



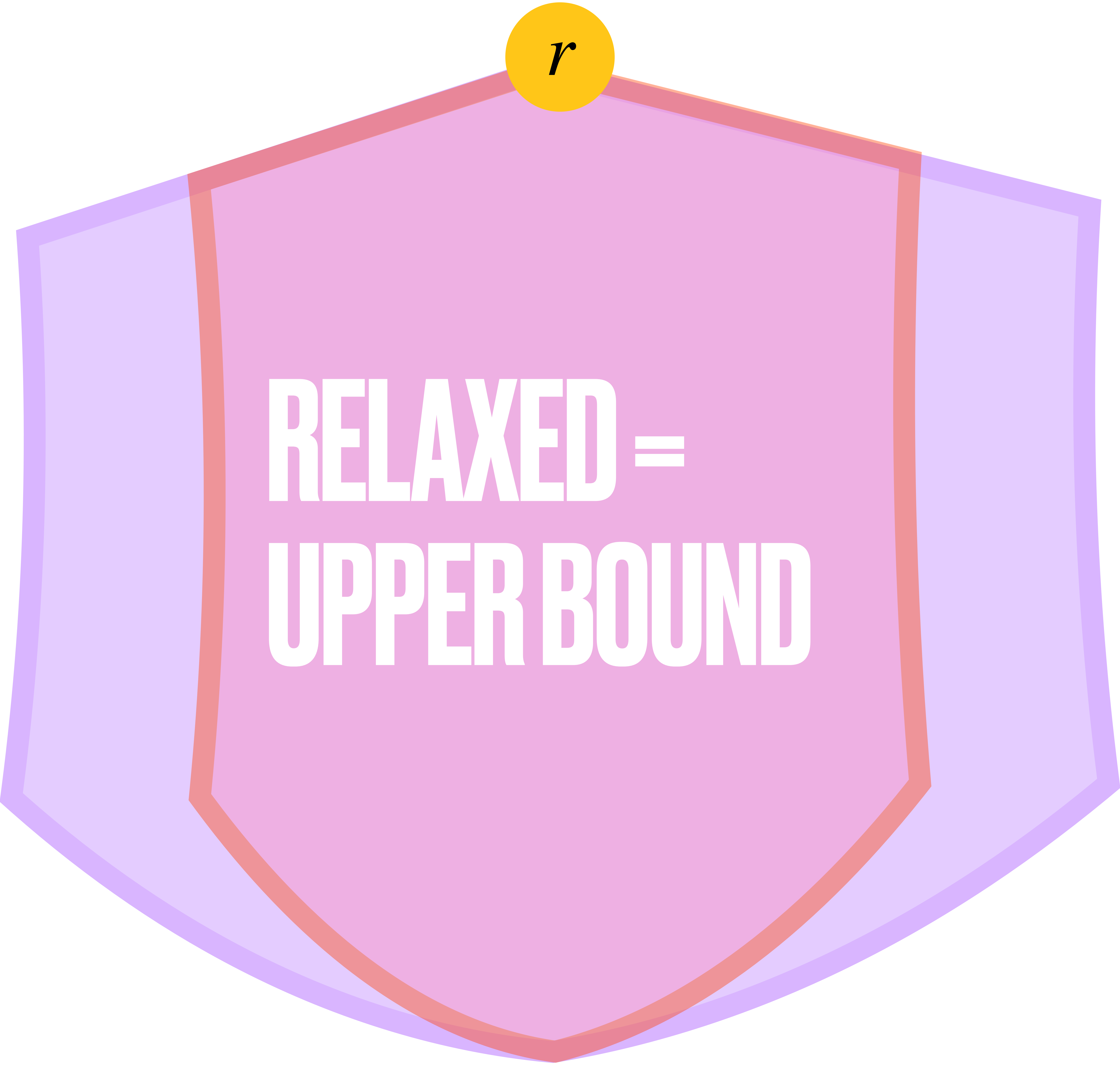
RELAXED =
UPPER BOUND

1

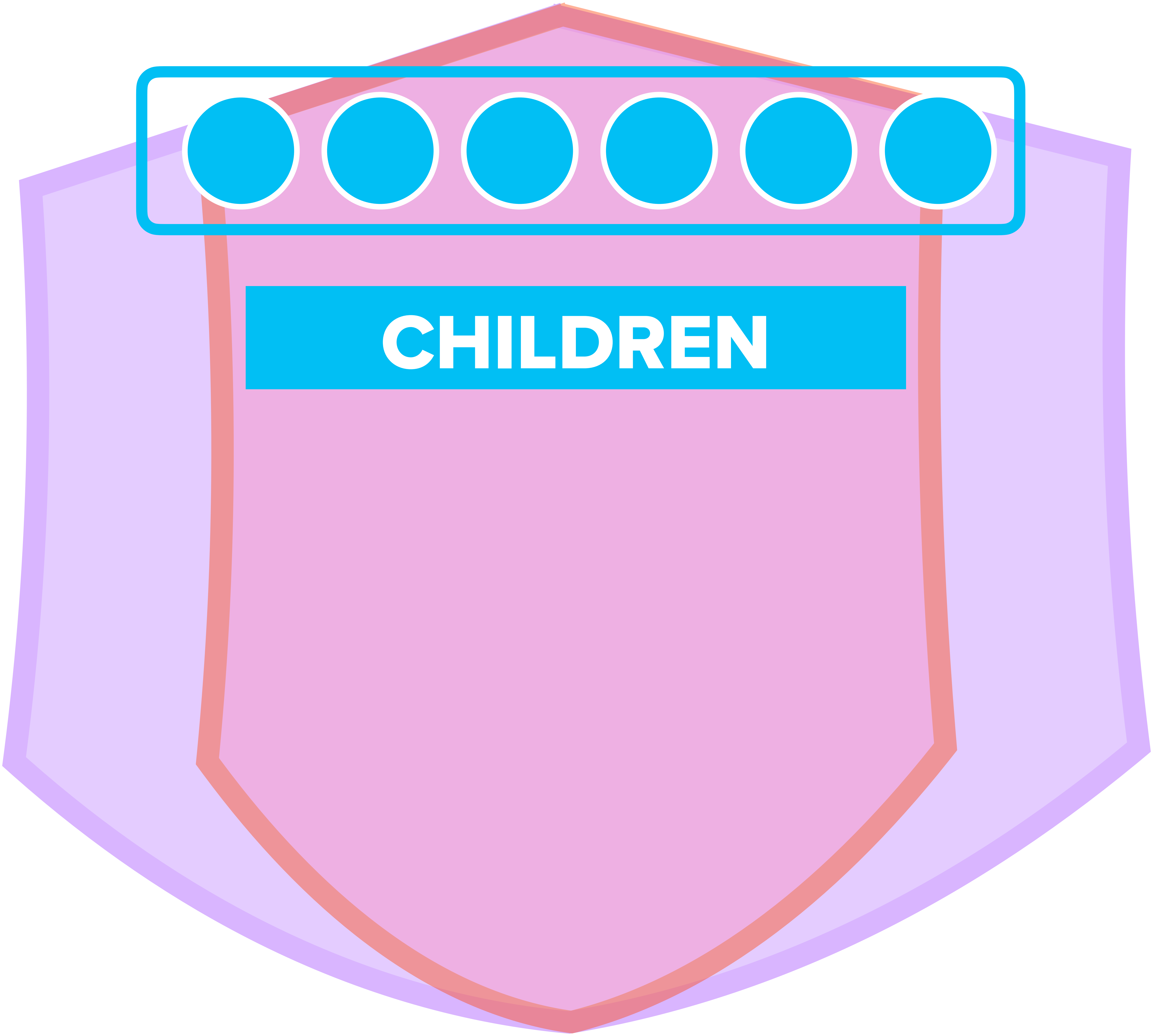
Is it equal to the lower bound ?

2

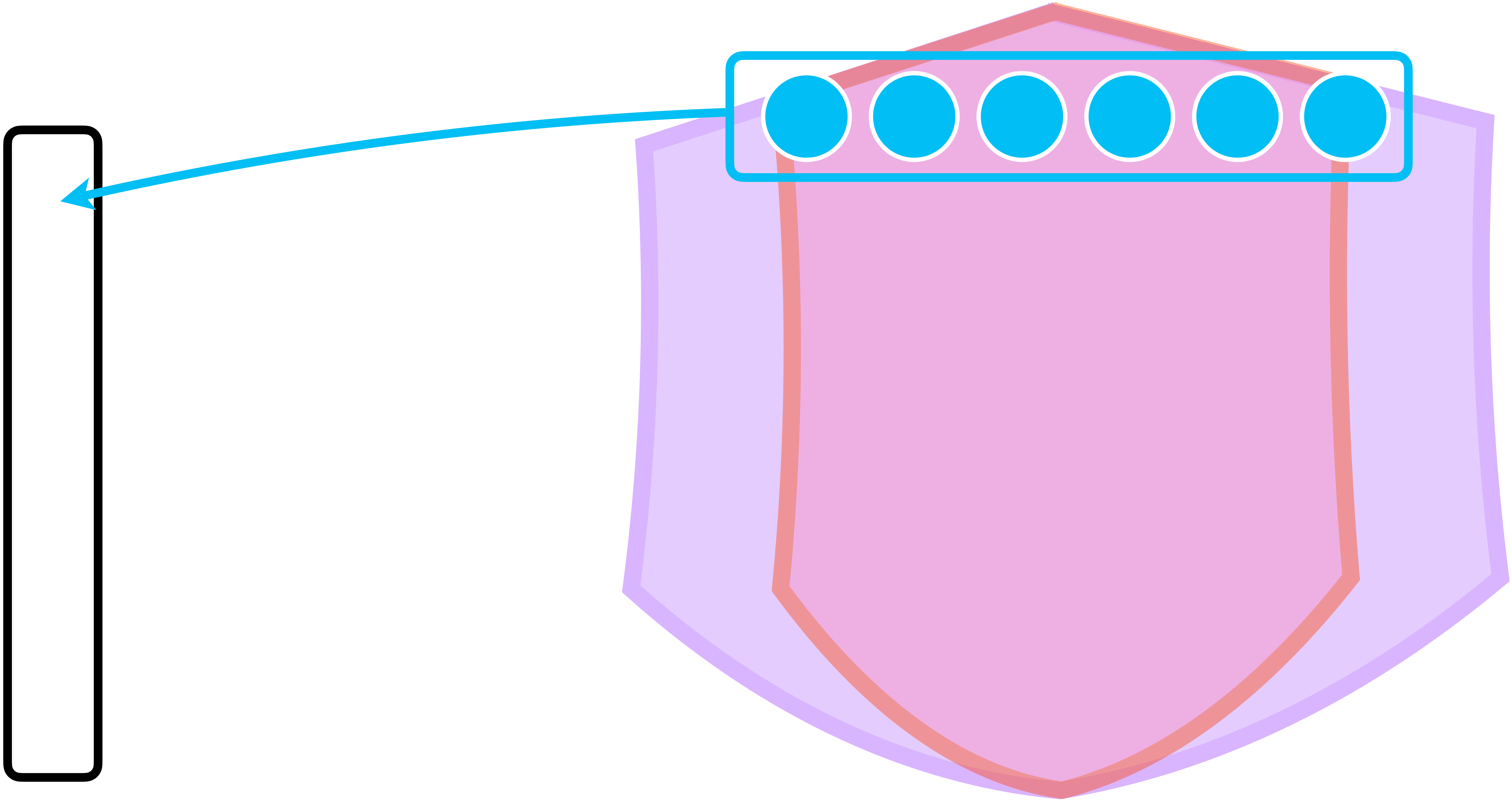
Is it possibly better than the best LB ?

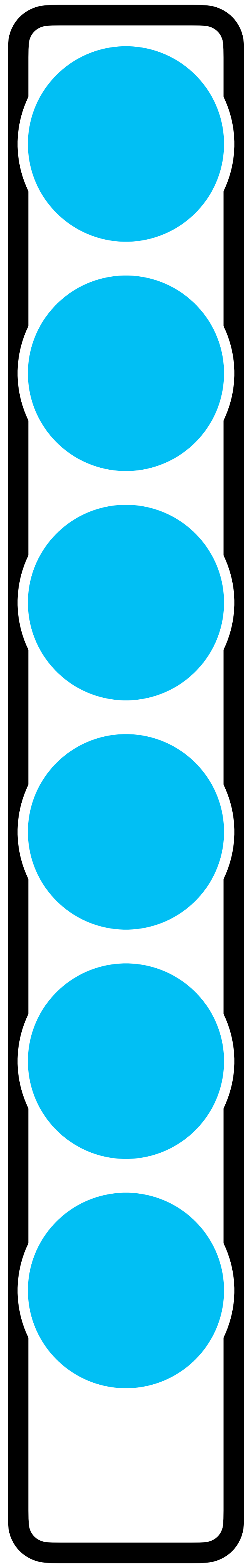


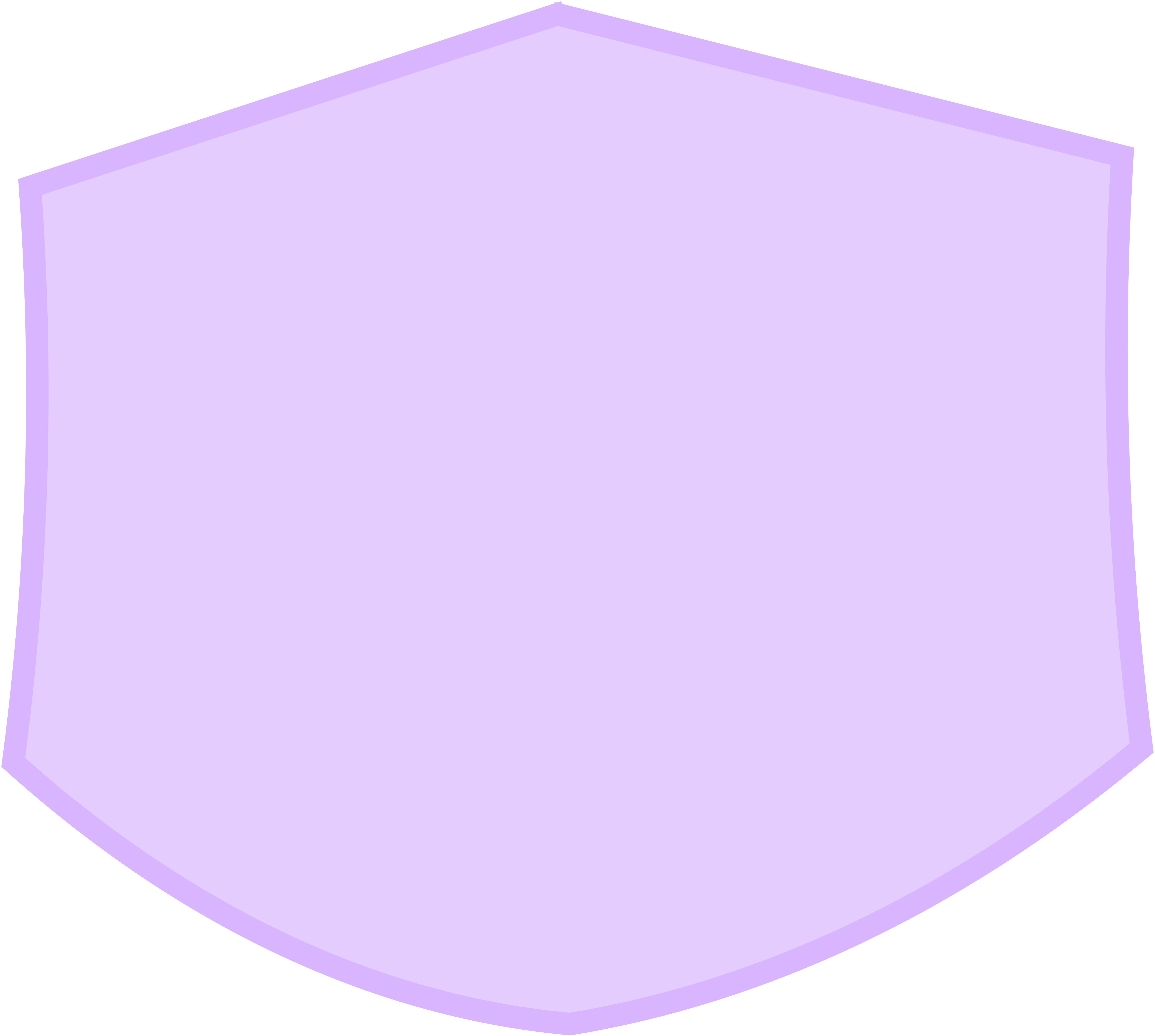
RELAXED = UPPER BOUND

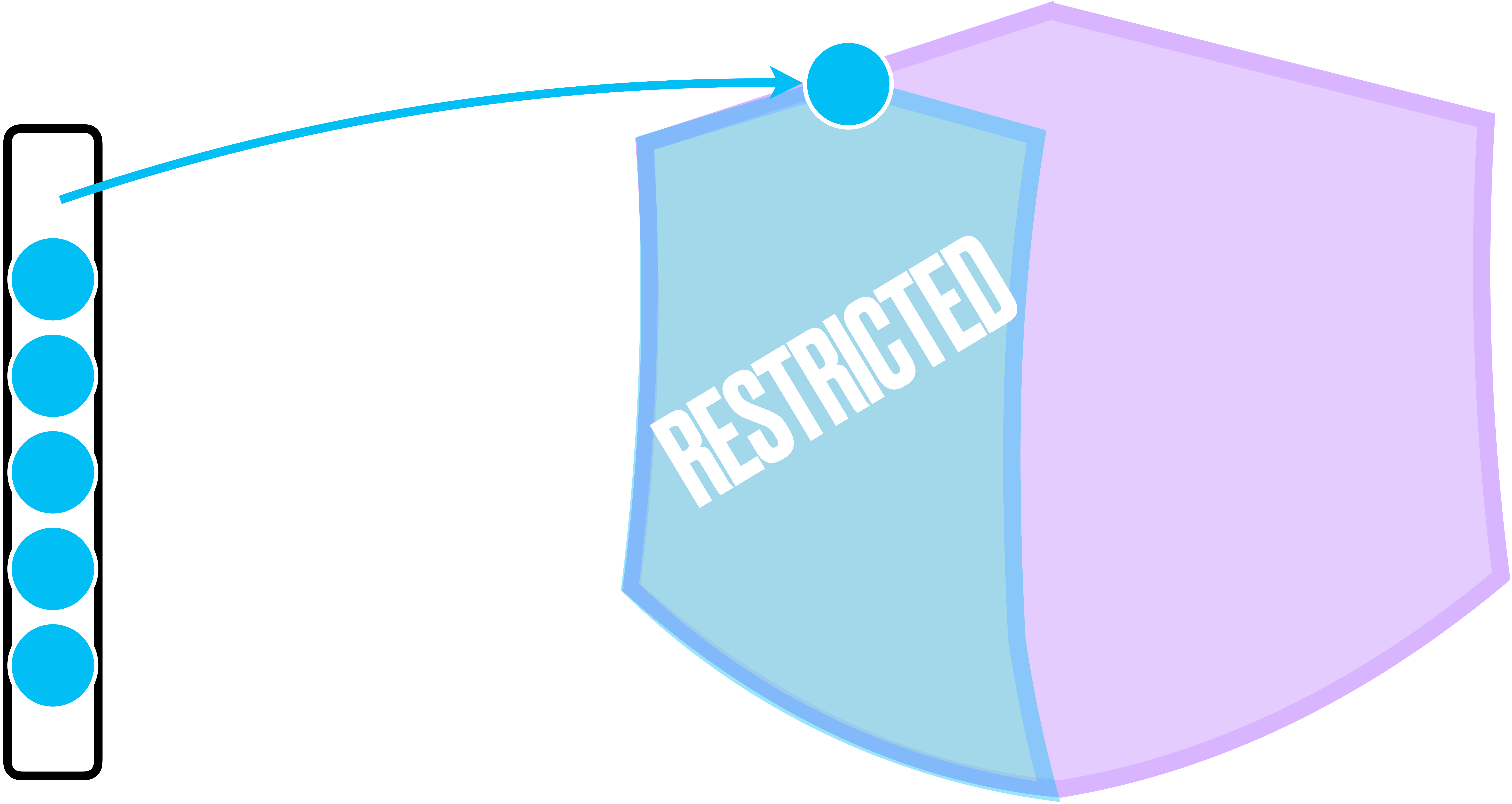


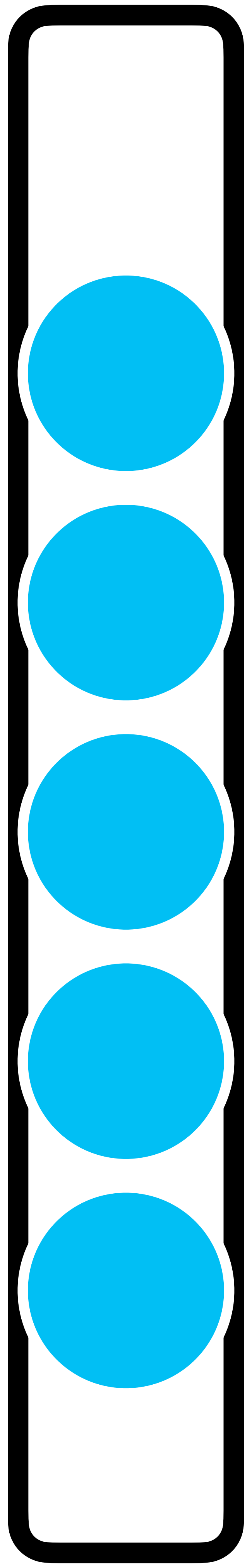
CHILDREN








RESTRICTED

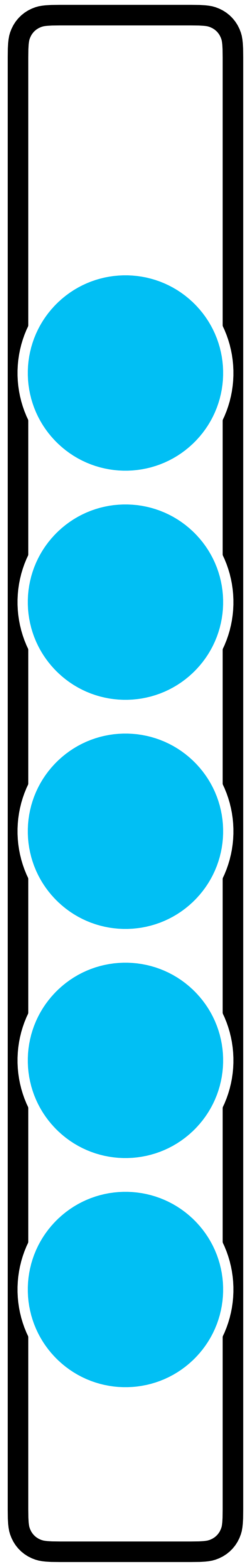


1

Does it improve the best known solution ?

2

Is it exact ?

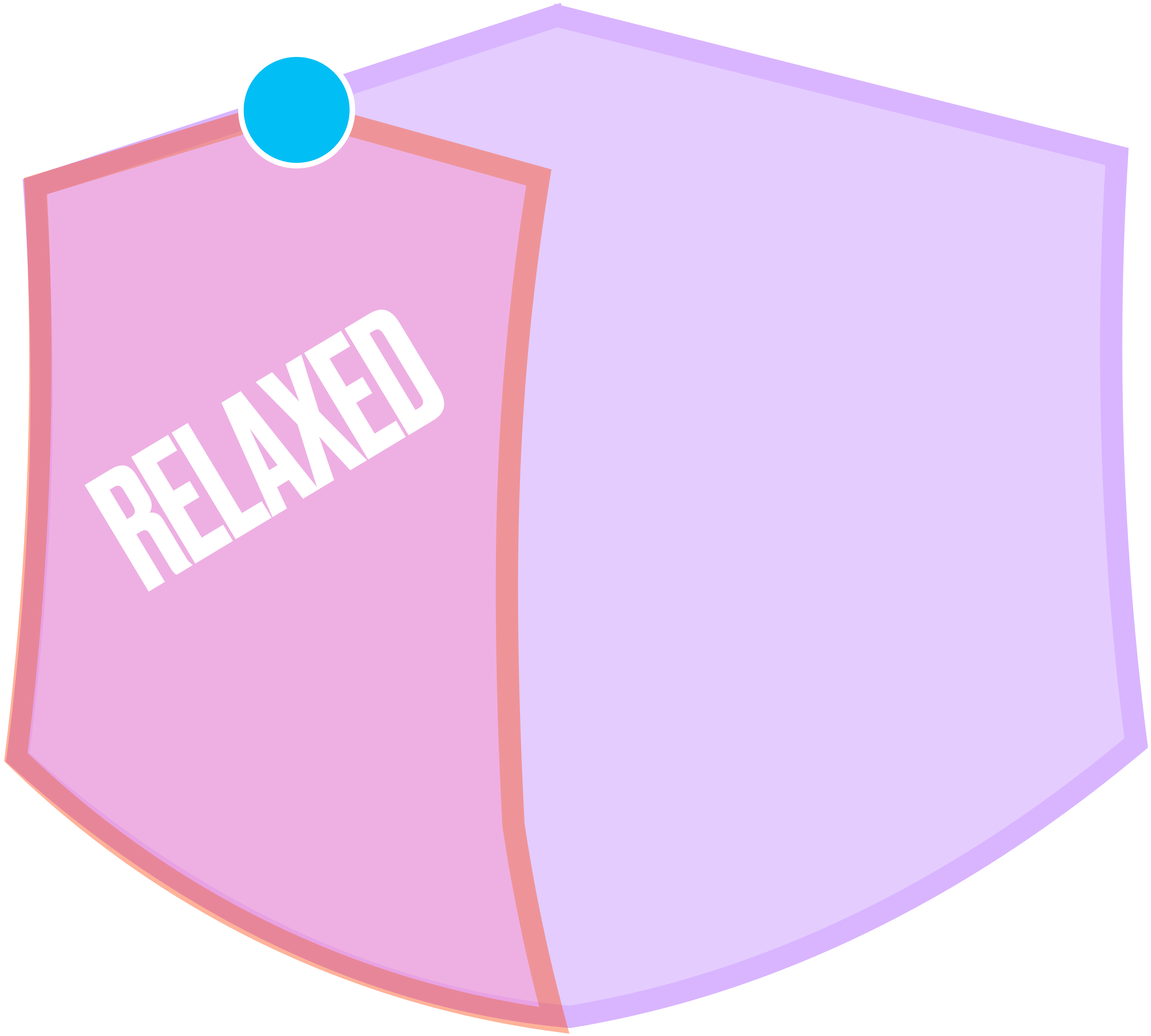


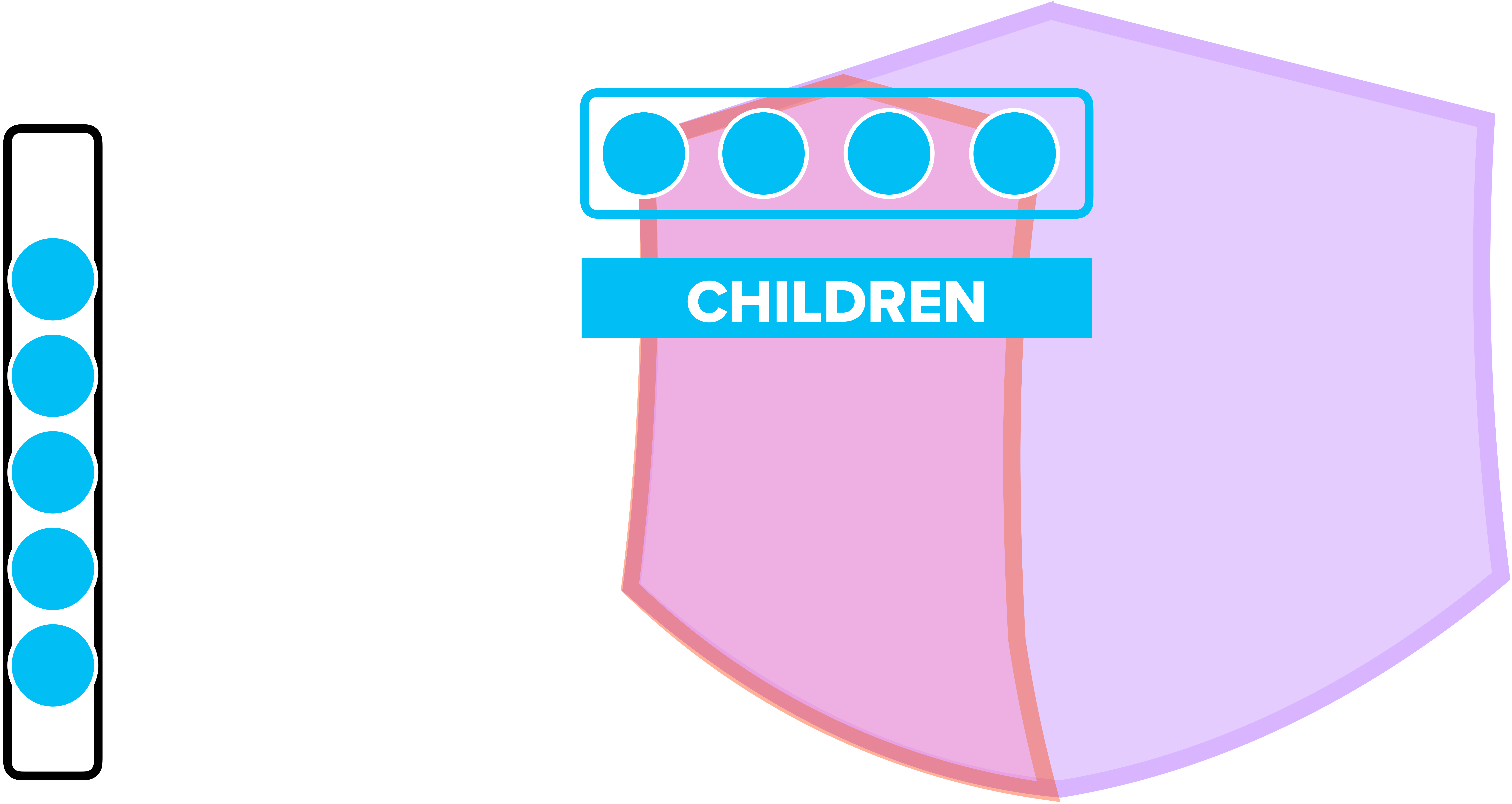
①

Is it equal to the lower bound ?

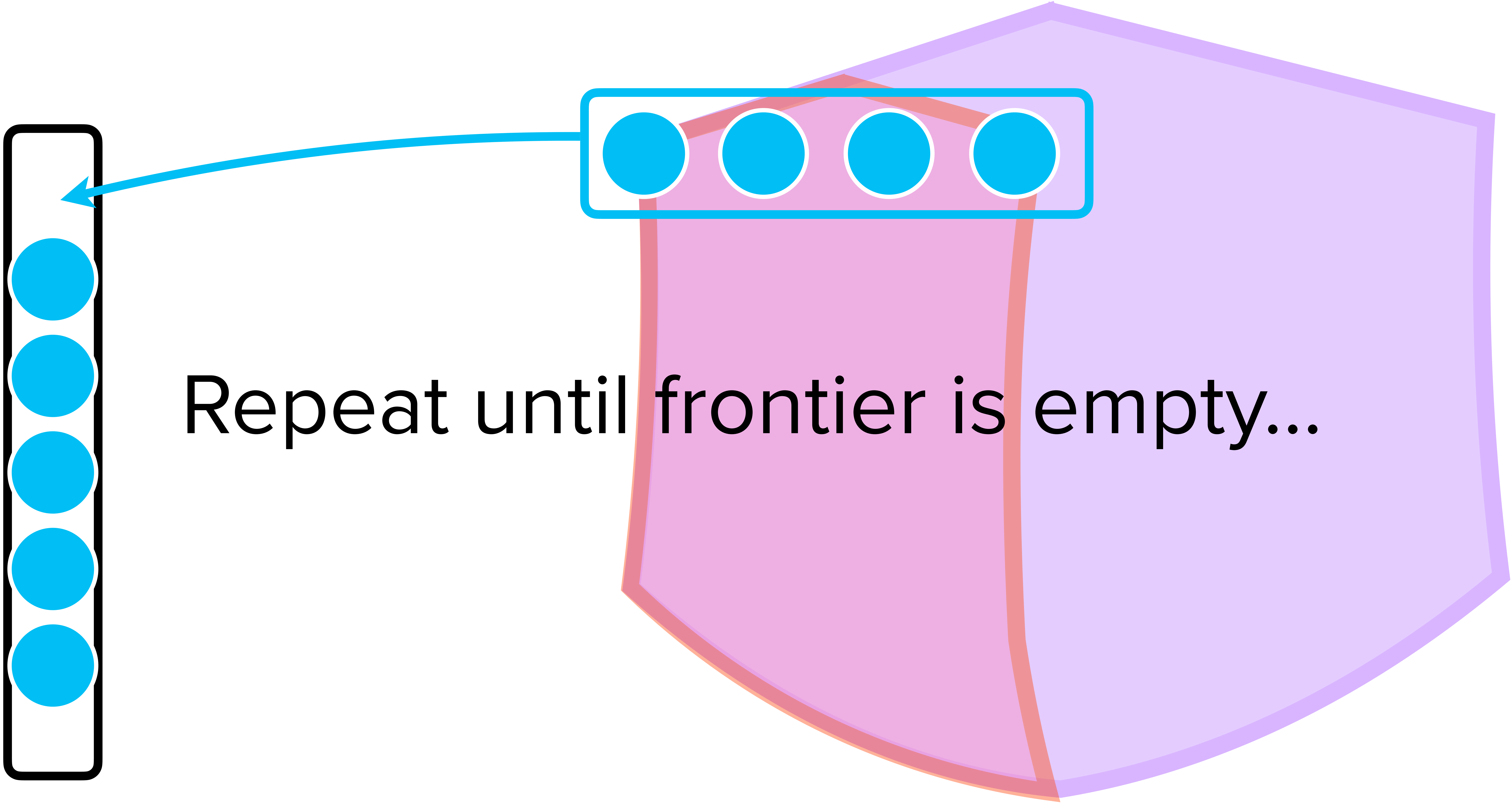
②

Is it possibly better than the best LB ?





CHILDREN

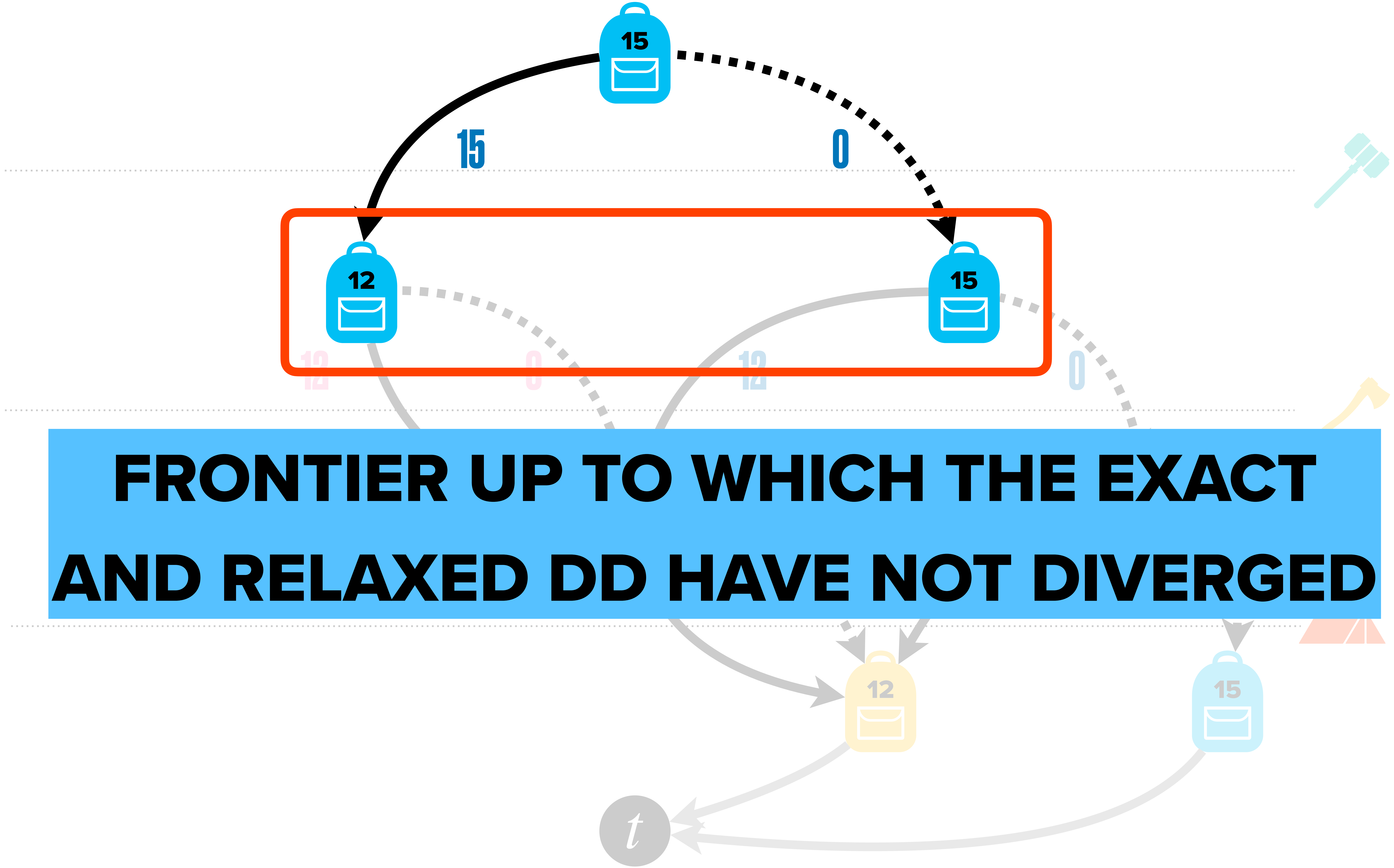


Repeat until frontier is empty...

How can we enumerate subproblems ?

Exact Cutset

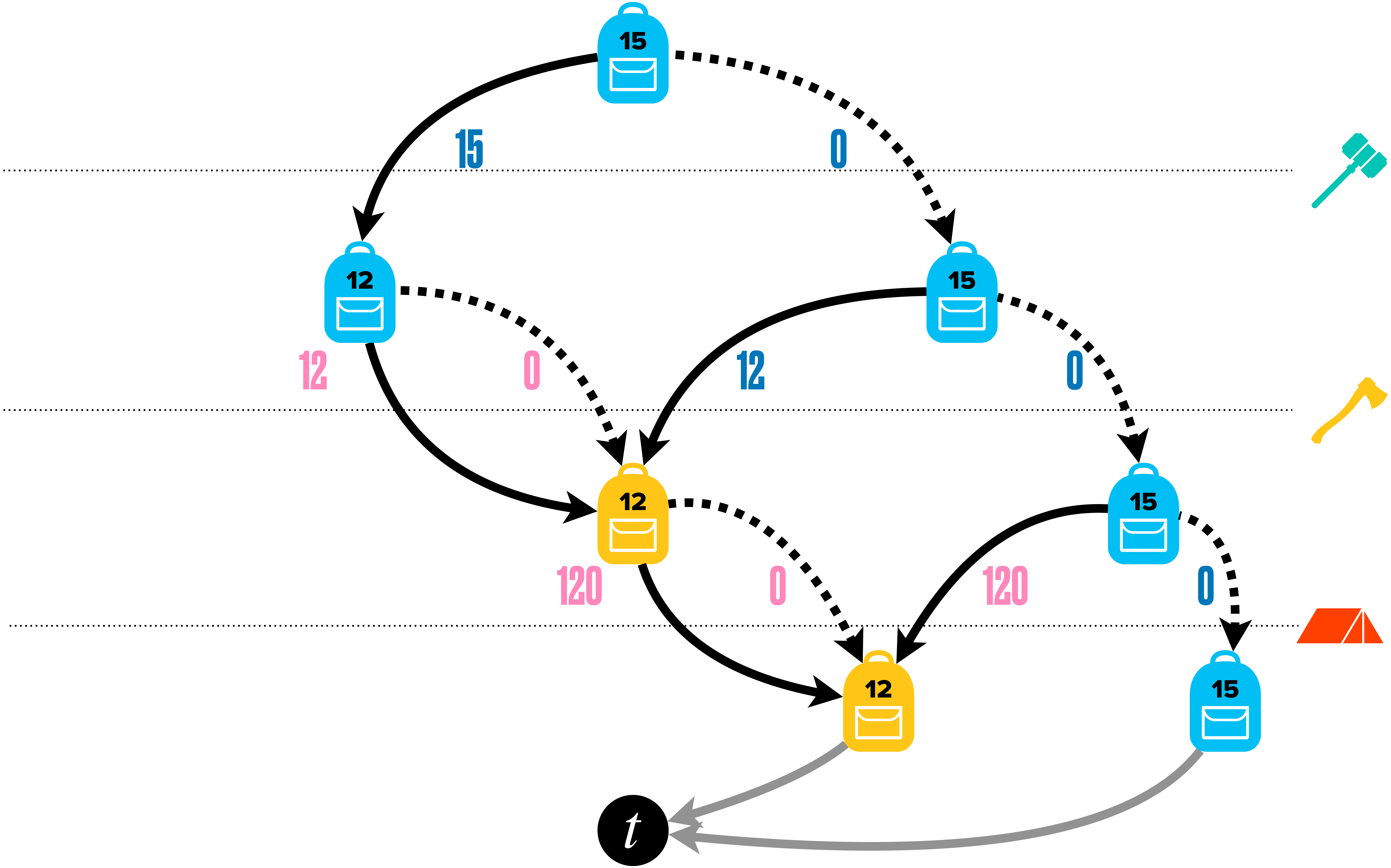
A subset $\mathscr{C}$ of the exact nodes s.t. any $r - t$ path must go through at least one node in $\mathscr{C}$



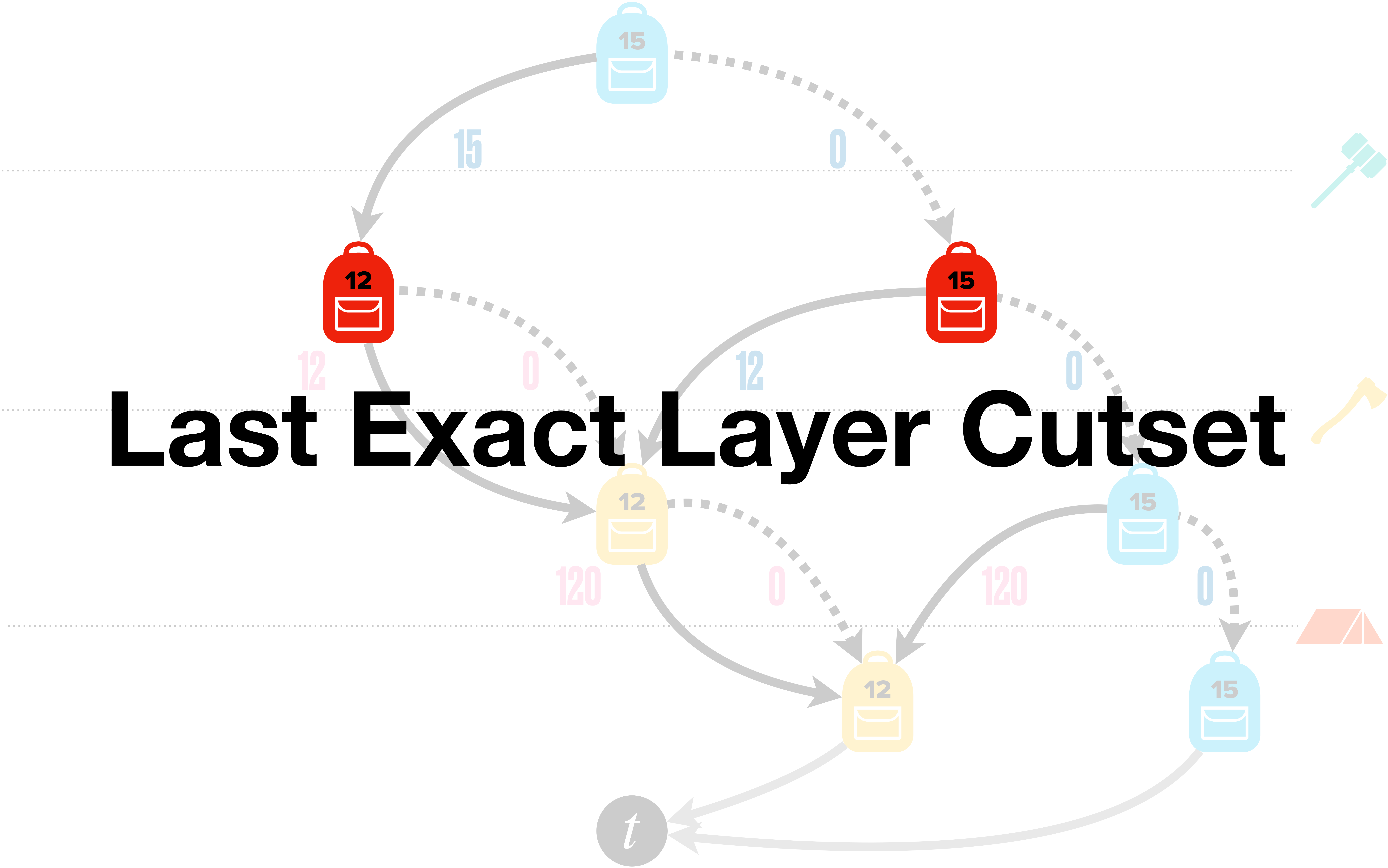
FRONTIER UP TO WHICH THE EXACT AND RELAXED DD HAVE NOT DIVERGED

Exact Cutset

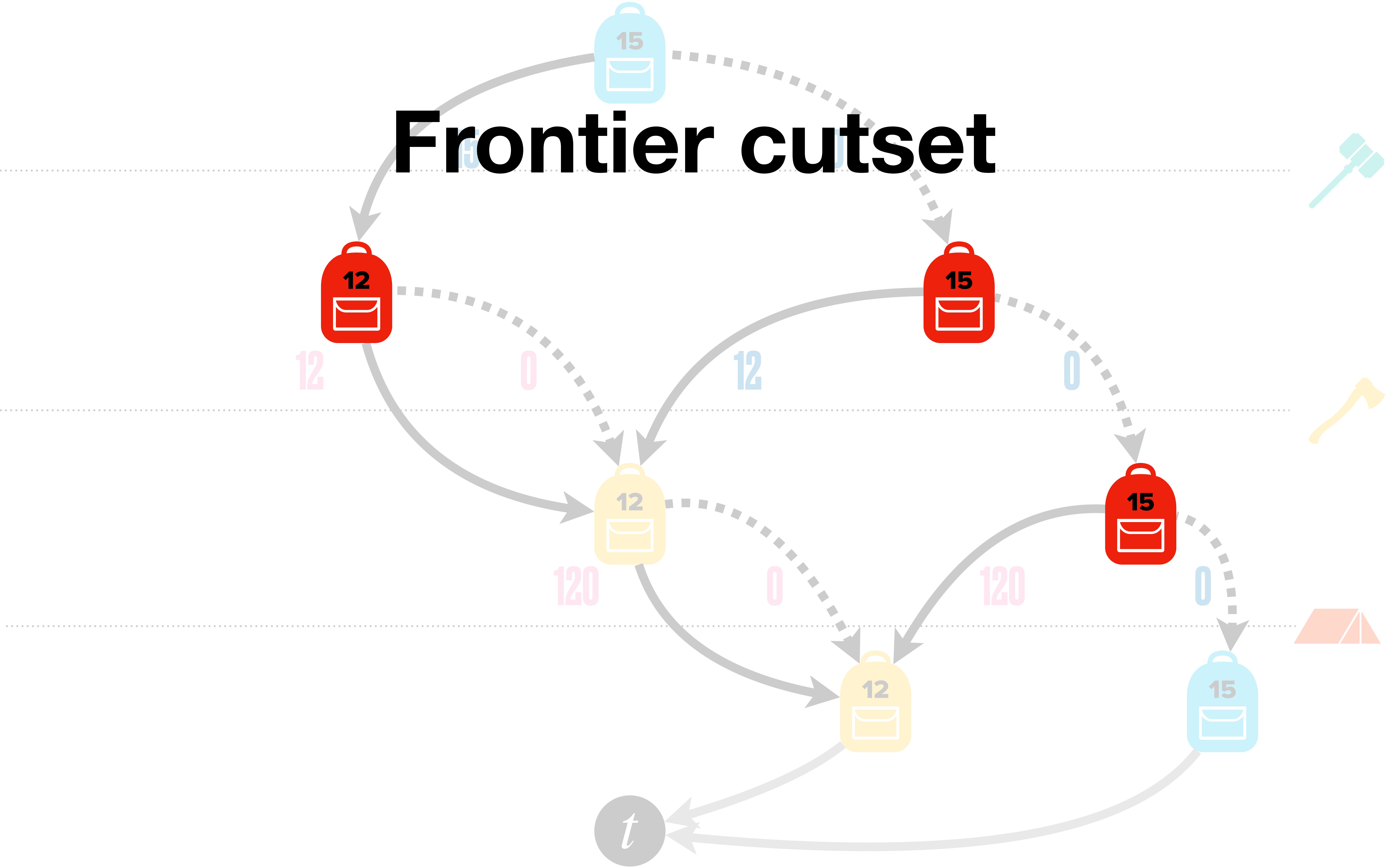
- There is always exists AT LEAST one exact cutset.
- The exact cutset is not guaranteed to be unique
    - First Exact Layer (Traditional branching)
    - Last Exact Layer (Deepest layer where all nodes are exact)
    - Frontier Cutset (Set of all the direct parents of inexact nodes)


15
15
0
12
15
12
0
12
0
12
15
120
0
120
0
12
15
t

Last Exact Layer Cutset



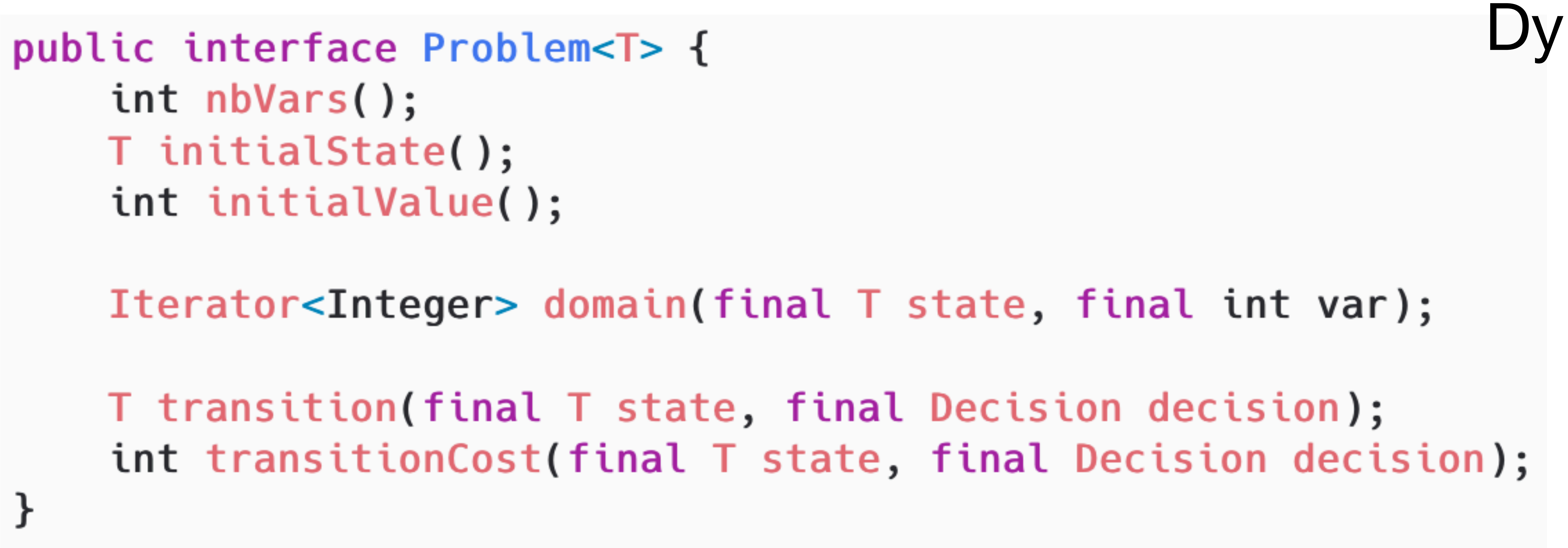
Frontier cutset



Part 3: Code

Interfaces (Core)

LTS Semantics of a
Dynamic Program



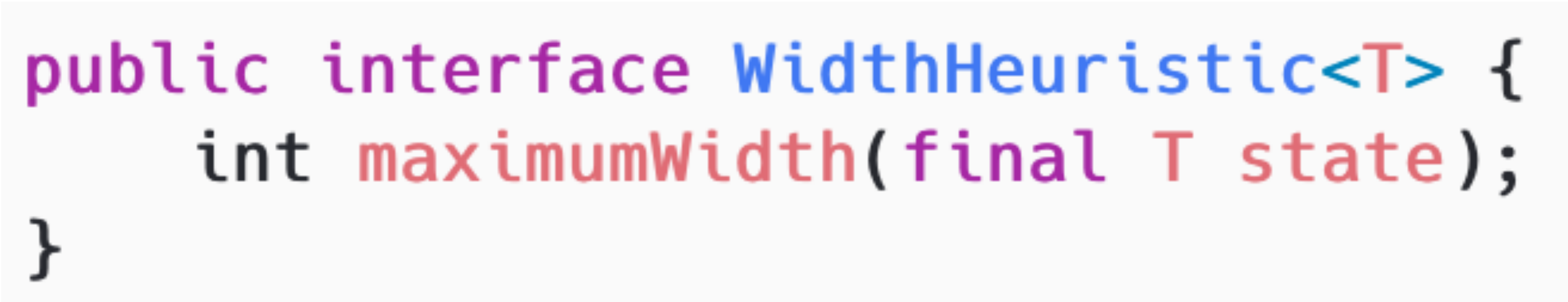
```
public interface Problem<T> {
    int nbVars();
    T initialState();
    int initialValue();

    Iterator<Integer> domain(final T state, final int var);

    T transition(final T state, final Decision decision);
    int transitionCost(final T state, final Decision decision);
}
```

$\oplus$ Operator

```
public interface Relaxation<T> {
    T mergeStates(final Iterator<T> states);
    int relaxEdge(final T from, final T to, final T merged, final Decision d, final int cost);
}
```

Γ Operator

Interfaces (Heuristics)



```
public interface WidthHeuristic<T> {
    int maximumWidth(final T state);
}
```

What is the maximum width
for an MDD rooted in the
subproblem with the given state ?

```
new FixedWidth<>(250);
```

```
public interface StateRanking<T> extends Comparator<T> {}
```

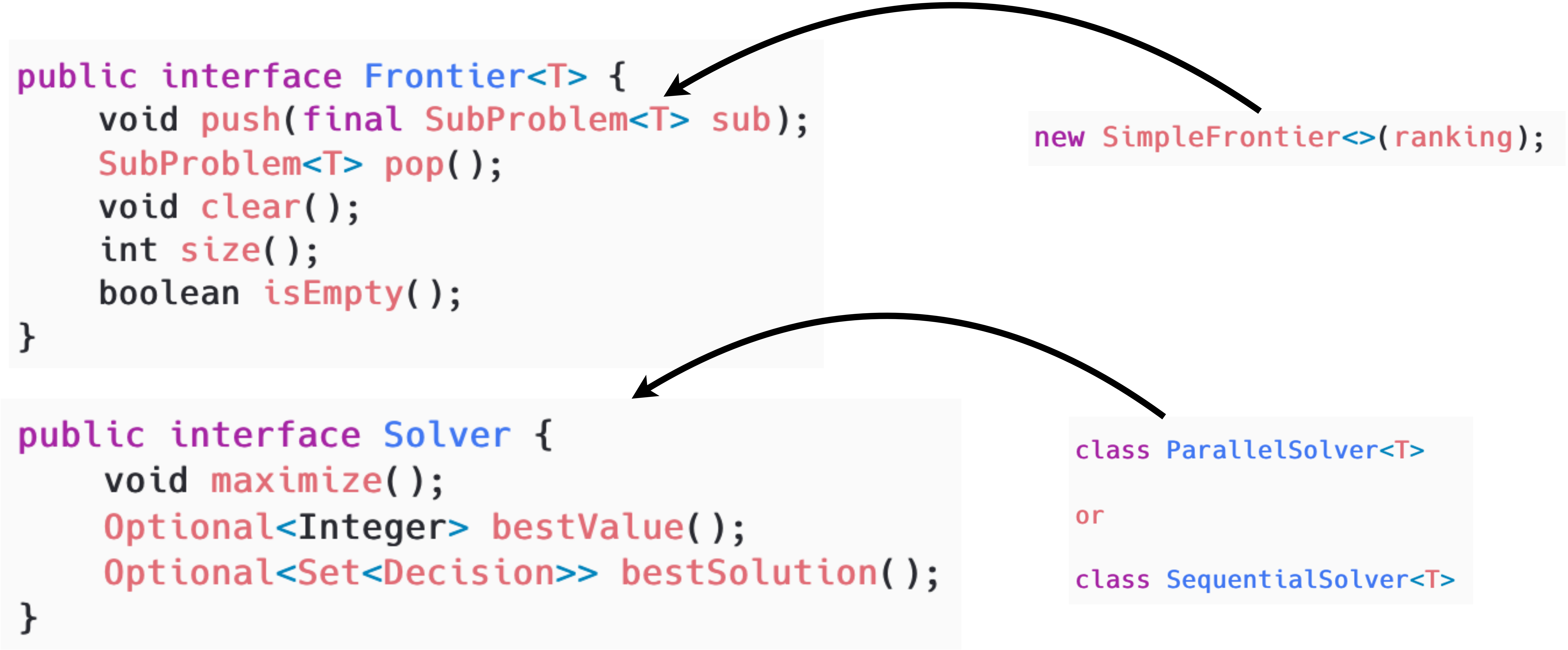
Discriminate promising from
other nodes

```
public interface VariableHeuristic<T> {
    Integer nextVariable(final Set<Integer> variables, final Iterator<T> states);
}
```

Order of the variables

```
new DefaultVariableHeuristic<>();
```

Interfaces (Utils)



```
public interface Frontier<T> {
    void push(final SubProblem<T> sub);
    SubProblem<T> pop();
    void clear();
    int size();
    boolean isEmpty();
}
```

```
new SimpleFrontier<>(ranking);
```

```
public interface Solver {
    void maximize();
    Optional<Integer> bestValue();
    Optional<Set<Decision>> bestSolution();
}
```

```
class ParallelSolver<T>

or

class SequentialSolver<T>
```

Interfaces (DD)

The same DecisionDiagram object
can be reused to compile different subproblems
(for performance reasons)

```
public interface DecisionDiagram<T> {
    void compile(final CompilationInput<T> input);
    boolean isExact();
    Optional<Integer> bestValue();
    Optional<Set<Decision>> bestSolution();
    Iterator<SubProblem<T>> exactCutset();
}
```

In practice, this interface is implemented for you:

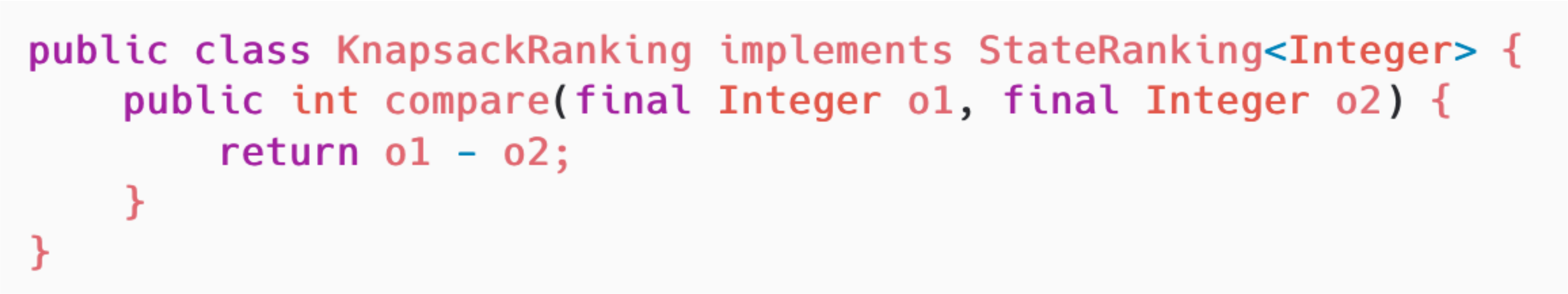
```
new LinkedDecisionDiagram<>();
```

Knapsack

Example

```java
public class KnapsackProblem implements Problem<Integer> {
        final int   capa;
        final int[] profit;
        final int[] weight;

        public KnapsackProblem(final int capa, final int[] profit, final int[] weight) {
            this.capa   = capa;
            this.profit = profit;
            this.weight = weight;
        }

        public int nbVars()           { return profit.length; }
        public Integer initialState() { return capa;          }
        public int initialValue()     { return 0;             }

        public Iterator<Integer> domain(Integer state, int var) {
            if (state >= weight[var]) {
                return Arrays.asList(1, 0).iterator();
            } else {
                return Arrays.asList(0).iterator();
            }
        }

        public Integer transition(Integer state, Decision decision) {
            return state - weight[decision.var()] * decision.val();
        }

        public int transitionCost(Integer state, Decision decision) {
            return profit[decision.var()] * decision.val();
        }
    }
```

```java
private static class KnapsackRelax implements Relaxation<Integer> {
    public Integer mergeStates(final Iterator<Integer> states) {
        int capa = 0;
        while (states.hasNext()) {
            final Integer state = states.next();
            capa  = Math.max(capa,  state);
        }
        return capa;
    }

    public int relaxEdge(Integer from, Integer to, Integer merged, Decision d, int cost) {
        return cost;
    }
}
```



```java
public class KnapsackRanking implements StateRanking<Integer> {
    public int compare(final Integer o1, final Integer o2) {
        return o1 - o2;
    }
}
```

```java
public static void main(final String[] args) throws IOException {
        final KnapsackProblem problem = readInstance("example_file.txt");
        final KnapsackRelax relax = new KnapsackRelax();
        final KnapsackRanking ranking = new KnapsackRanking();
        final FixedWidth<Integer> width = new FixedWidth<>(250);
        final VariableHeuristic<Integer> varh = new DefaultVariableHeuristic<>();
        final Frontier<Integer> frontier = new SimpleFrontier<>(ranking);
        final Solver solver = new ParallelSolver<Integer>(
                Runtime.getRuntime().availableProcessors(),
                problem,
                relax,
                varh,
                ranking,
                width,
                frontier);

        solver.maximize();
        int[] solution = solver.bestSolution();
        System.out.println(Arrays.toString(solution));
  }
```