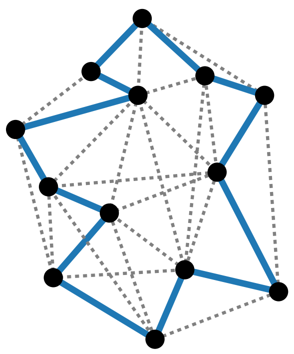
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

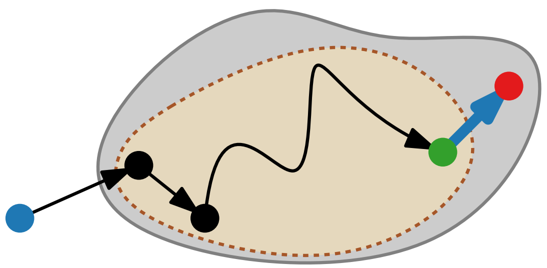
Advanced Algorithms

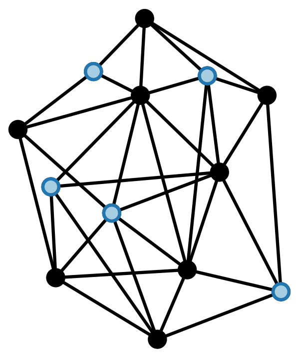
Exact algorithms for NP-hard problems

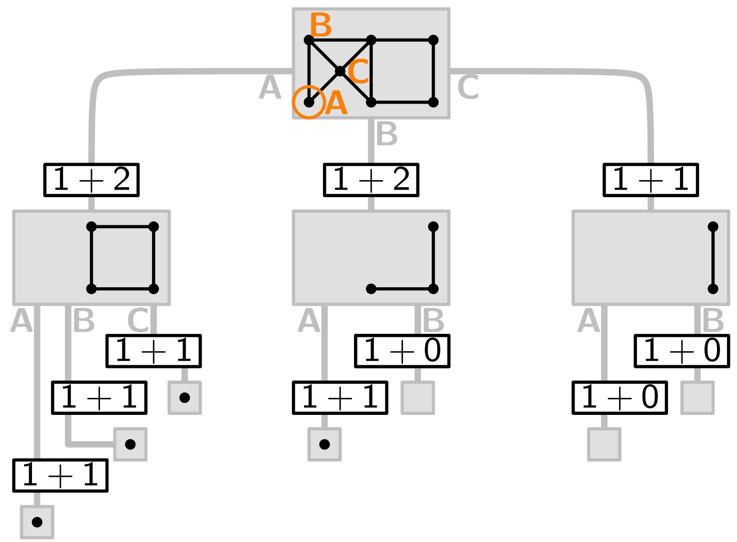
TSP and MIS

Jonathan Klawitter · WS20







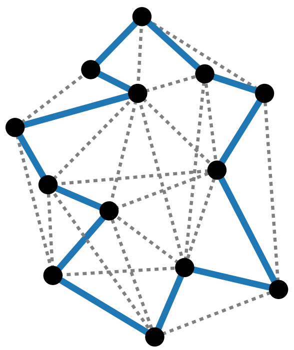


Examples of NP-hard problems

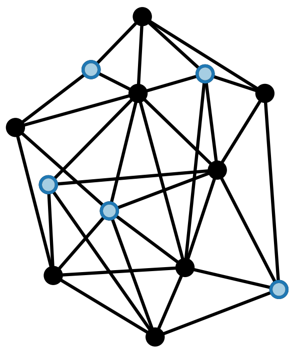
Many important (practical) problems are NP-hard, for example . . .

Examples of NP-hard problems

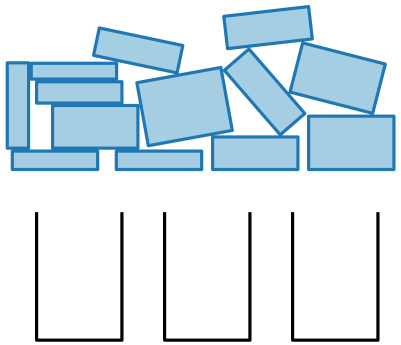
Many important (practical) problems are NP-hard, for example . . .



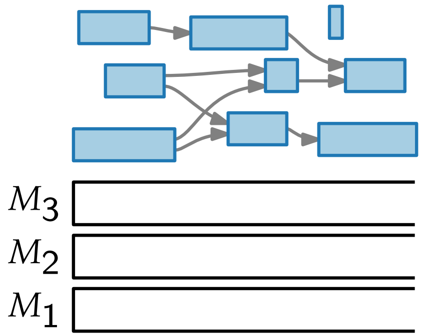
TSP



MIS



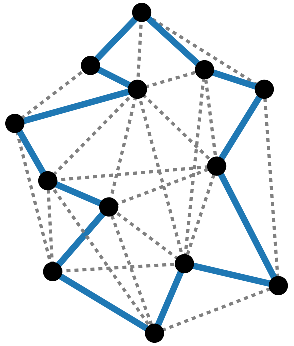
Bin Packing



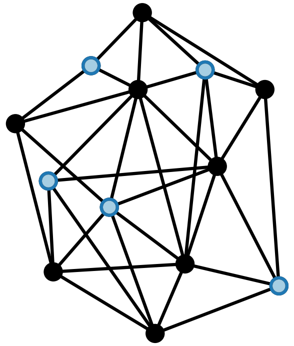
Scheduling

Examples of NP-hard problems

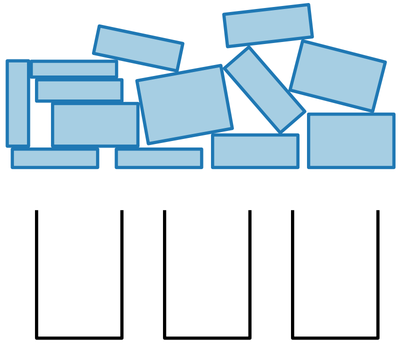
Many important (practical) problems are NP-hard, for example . . .



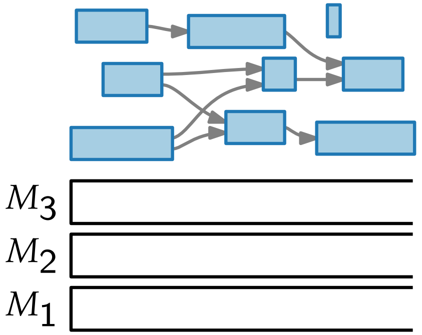
TSP



MIS



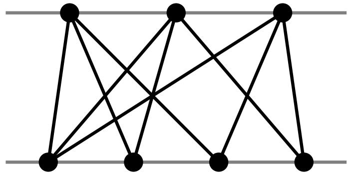
Bin Packing



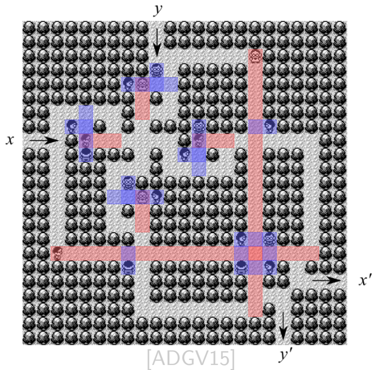
Scheduling

$(x_1 \vee x_2 \vee \neg x_4) \wedge$
$(\neg x_2 \vee x_3 \vee \neg x_4) \wedge$
$(x_3 \vee x_7 \vee \neg x_8) \wedge$
$\dots$

SAT



Graph Drawing



Games

...

Formal view on NP-hardness

But what does NP-hard/-complete actually mean?

Formal view on NP-hardness

But what does NP-hard/-complete actually mean?

- NP-hard = non-deterministic polynomial-time hard

Formal view on NP-hardness

But what does NP-hard/-complete actually mean?

- NP-hard $=$ non-deterministic polynomial-time hard
- A decision problem H is NP-hard when it is "at least as hard as the hardest problems in P".
- or: There is a polynomial-time many-one reduction from an NP-hard problem L to H.

Formal view on NP-hardness

But what does NP-hard/-complete actually mean?

- NP-hard = non-deterministic polynomial-time hard
- A decision problem H is NP-hard when it is "at least as hard as the hardest problems in P".
- or: There is a polynomial-time many-one reduction from an NP-hard problem L to H.
- If P $\neq$ NP, then NP-hard problems cannot be solved in polynomial time.

Misconceptions about NP-hardness

Common misconceptions [Mann '17]

- If similar problems are NP-hard, then the problem at hand is also NP-hard.

Misconceptions about NP-hardness

Common misconceptions [Mann '17]

- If similar problems are NP-hard, then the problem at hand is also NP-hard.
- Problems that are hard to solve in practice by an engineer are NP-hard.
- NP-hard problems cannot be solved optimally.

Misconceptions about NP-hardness

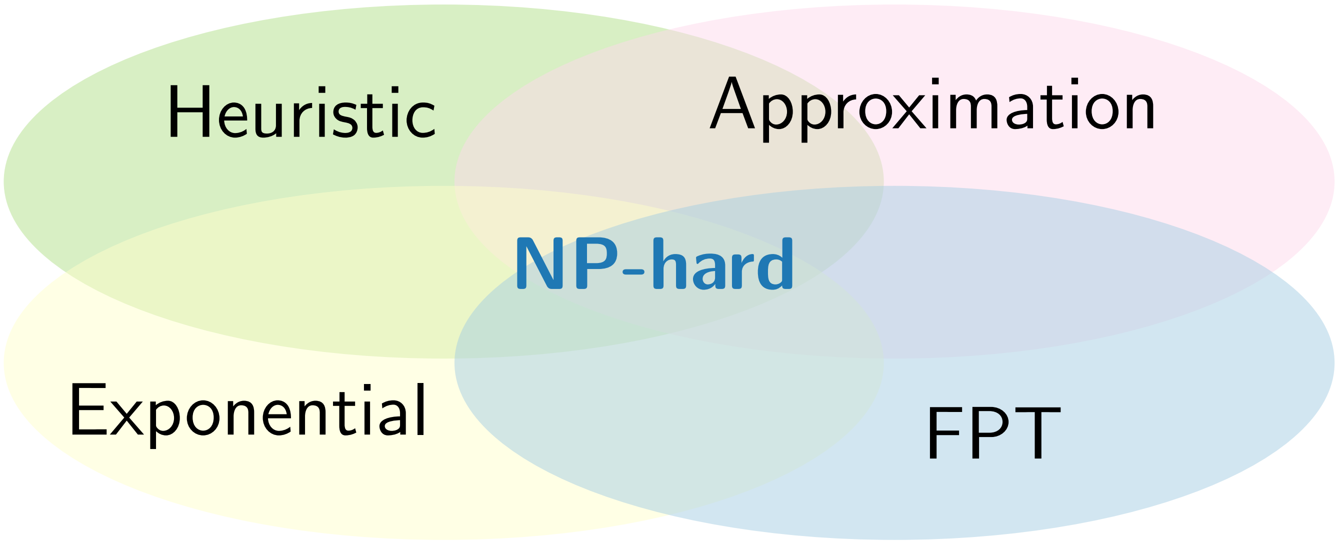
Common misconceptions [Mann '17]

- If similar problems are NP-hard, then the problem at hand is also NP-hard.
- Problems that are hard to solve in practice by an engineer are NP-hard.
- NP-hard problems cannot be solved optimally.
- NP-hard problems cannot be solved more efficiently than by exhaustive search.
- For solving NP-hard problems, the only practical possibility is the use of heuristics.

Dealing with NP-hard problems

What should we do?

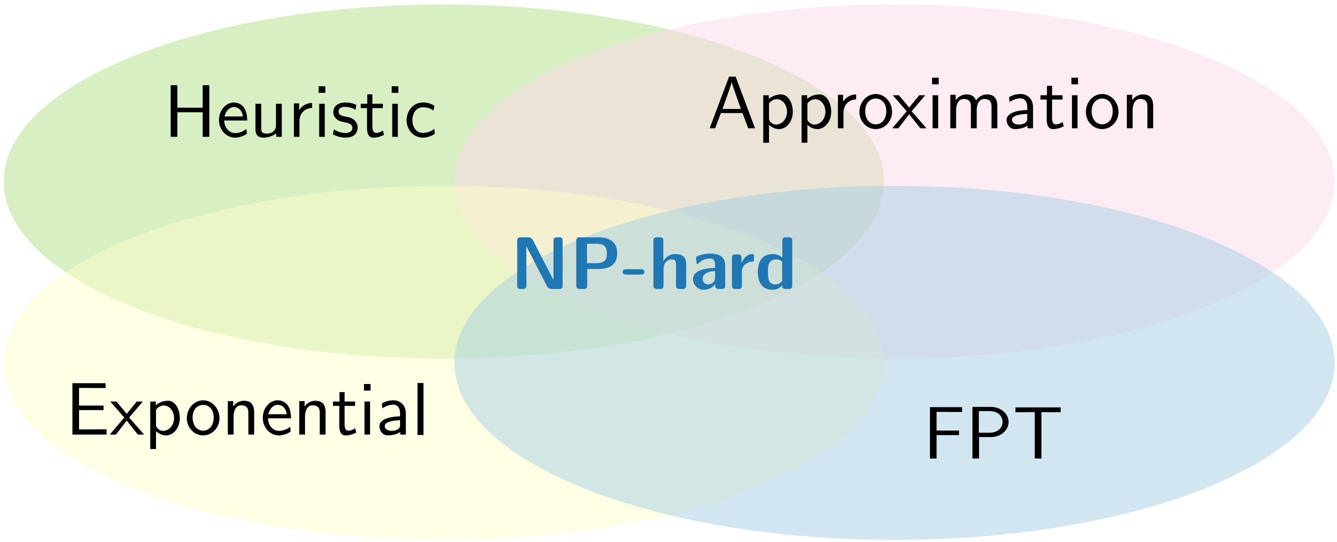
- Sacrifice optimality for speed
  - Heuristics (Simulated Annealing, Tabu-Search)
  - Approximation Algorithms (Christofides-Algorithm)
- Optimal Solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized algorithms



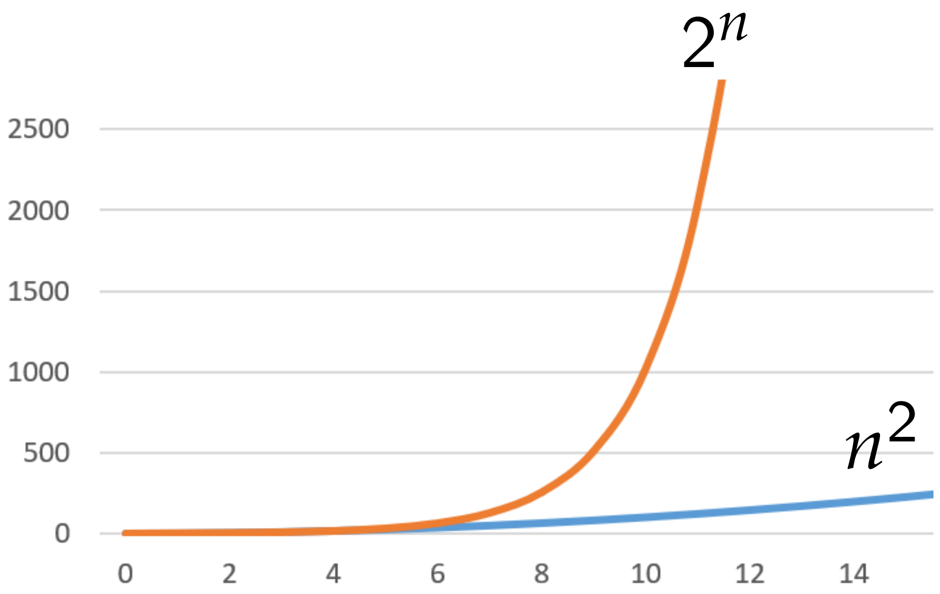
Dealing with NP-hard problems

What should we do?

- Sacrifice optimality for speed
  - Heuristics (Simulated Annealing, Tabu-Search)
  - Approximation Algorithms (Christofides-Algorithm)
- Optimal Solutions
  - Exact exponential-time algorithms ← this lecture
  - Fine-grained analysis – parameterized algorithms



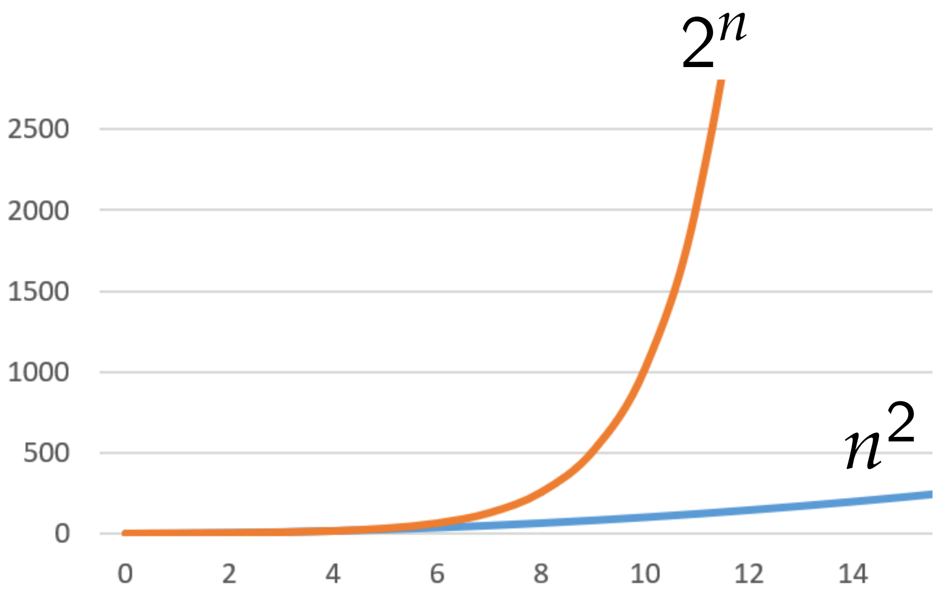
Motivation



- efficient vs.
  inefficient algorithms
- polynomial-time vs.
  super-polynomial-time

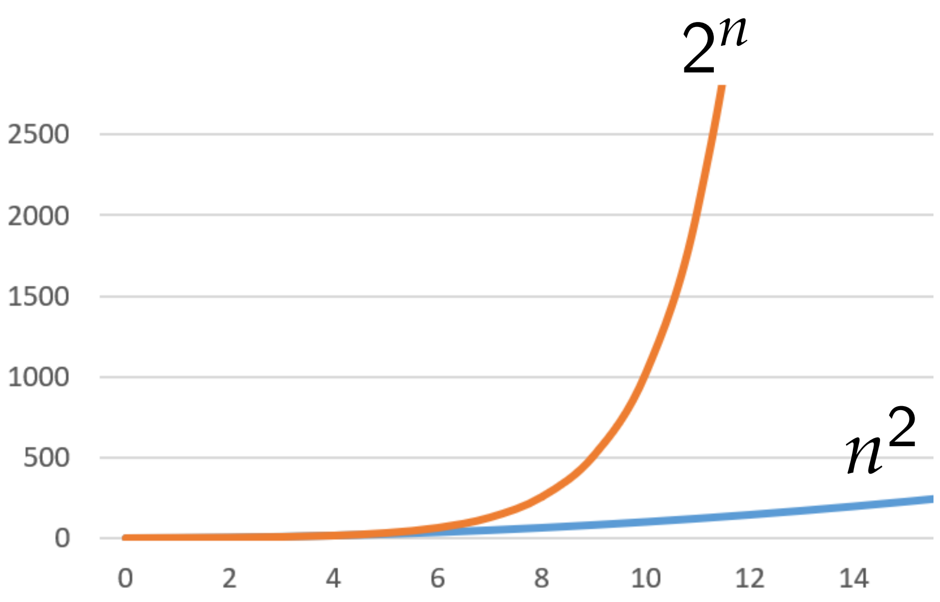
Motivation

Exponential runningtime . . . should we just give up?



- efficient vs.
  inefficient algorithms
- polynomial-time vs.
  super-polynomial-time

Motivation



- efficient vs. inefficient algorithms
- polynomial-time vs. super-polynomial-time

Exponential runningtime . . . should we just give up?

- . . . can be *"fast"* for medium-sized instances:
  - $n^4 > 1.2^n$ for $n \leq 100$
  - TSP solvable exactly for $n \leq 2000$ and specialized instances with $n \leq 85900$
  - "hidden" constants in polynomial-time algorithms:
    $2^{100}n > 2^n$ for $n \leq 100$

Motivation

Exponential runningtime . . . maybe we need better hardware?

Motivation

Exponential runningtime . . . maybe we need better hardware?

- Suppose an algorithm uses a^n steps & can solve for a fixed amount of time t instances up to size n_0.
- Improving hardware by a constant factor c only *adds a constant* (relative to c) to n_0:

$$a^{n_0'} = c \cdot a^{n_0} \rightsquigarrow n_0' = \log_a c + n_0$$

Motivation

Exponential runningtime ... maybe we need better hardware?

- Suppose an algorithm uses a^n steps & can solve for a fixed amount of time t instances up to size n_0.
- Improving hardware by a constant factor c only *adds a constant* (relative to c) to n_0:

$$a^{n_0'} = c \cdot a^{n_0} \rightsquigarrow n_0' = \log_a c + n_0$$

- Reducing the base of the runtime to $b < a$ results in a *multiplicative* increase:

$$b^{n_0'} = a^{n_0} \rightsquigarrow n_0' = n_0 \cdot \log_b a$$

Motivation

Exponential runningtime . . . but can we at least find exact algorithms that are faster than **brute-force** (trivial) approaches?

Motivation

Exponential runningtime ... but can we at least find exact algorithms that are faster than **brute-force** (trivial) approaches?

- TSP: Bellman-Held-Karp algorithm has running time $\mathcal{O}(2^n n^2)$ compared to a $\mathcal{O}(n!n)$-time brute-force search.
- MIS: algorithm by Tarjan & Trojanowski runs in $\mathcal{O}(2^{n/3})$ time compared to a trivial $\mathcal{O}(n2^n)$-time approach.
- $\textsc{Coloring}$: Lawler gaven an $\mathcal{O}(n(1+\sqrt[3]{3})^n)$ algorithm compared to $\mathcal{O}(n^{n+1})$-time brute-force.

Motivation

Exponential runningtime ... but can we at least find exact algorithms that are faster than **brute-force** (trivial) approaches?

- TSP: Bellman-Held-Karp algorithm has running time $\mathcal{O}(2^n n^2)$ compared to a $\mathcal{O}(n!n)$-time brute-force search.
- MIS: algorithm by Tarjan & Trojanowski runs in $\mathcal{O}(2^{n/3})$ time compared to a trivial $\mathcal{O}(n2^n)$-time approach.
- Coloring: Lawler gaven an $\mathcal{O}(n(1+\sqrt[3]{3})^n)$ algorithm compared to $\mathcal{O}(n^{n+1})$-time brute-force.
- SAT: No better algorithm than trivial brute-force search known.

$\mathcal{O}^*$-notation

$$\mathcal{O}(1.4^n \cdot n^2) \subsetneq \mathcal{O}(1.5^n \cdot n) \subsetneq \mathcal{O}(2^n)$$

- negligible polynomial factors
- base of exponential part dominates

$\mathcal{O}^*$-notation

$$\mathcal{O}(1.4^n \cdot n^2) \subsetneq \mathcal{O}(1.5^n \cdot n) \subsetneq \mathcal{O}(2^n)$$

- negligible polynomial factors
- base of exponential part dominates

$$f(n) \in \mathcal{O}^*(g(n)) \Leftrightarrow \exists \text{ polynomial } p(n) \text{ with } f(n) \in \mathcal{O}(g(n)p(n))$$

$\mathcal{O}^*$-notation

$$\mathcal{O}(1.4^n \cdot n^2) \subsetneq \mathcal{O}(1.5^n \cdot n) \subsetneq \mathcal{O}(2^n)$$

- negligible polynomial factors
- base of exponential part dominates

$$f(n) \in \mathcal{O}^*(g(n)) \Leftrightarrow \exists \text{ polynomial } p(n) \text{ with } f(n) \in \mathcal{O}(g(n)p(n))$$

- typical result

Approach	Runtime in $\mathcal{O}$-Notation	$\mathcal{O}^*$-Notation
Brute-Force	$\mathcal{O}(2^n)$	$\mathcal{O}^*(2^n)$
Algorithm A	$\mathcal{O}(1.5^n \cdot n)$	$\mathcal{O}^*(1.5^n)$
Algorithm B	$\mathcal{O}(1.4^n \cdot n^2)$	$\mathcal{O}^*(1.4^n)$

Traveling Salesperson Problem (TSP)

Input. Distinct cities $\{v_1, v_2, \ldots, v_n\}$ with distances $d(c_i, c_j) \in Q_{\geq 0}$;
directed, complete graph G with edge weights d

Traveling Salesperson Problem (TSP)

Input. Distinct cities $\{v_1, v_2, \ldots, v_n\}$ with distances $d(c_i, c_j) \in Q_{\geq 0}$;
directed, complete graph G with edge weights d

Output. Tour of the traveling salesperson of minimal total length that visits all the cities and returns to the starting point;

i.e. a Hamiltonian cycle $(v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)})$ of G of minimum weight

$$\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)})$$

Traveling Salesperson Problem (TSP)

Input. Distinct cities $\{v_1, v_2, \dots, v_n\}$ with distances $d(c_i, c_j) \in Q_{\geq 0}$;
directed, complete graph G with edge weights d

Output. Tour of the traveling salesperson of minimal total length that visits all the cities and returns to the starting point;

i.e. a Hamiltonian cycle $(v_{\pi(1)}, \dots, v_{\pi(n)}, v_{\pi(1)})$ of G of minimum weight

$$\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)})$$

Brute-force.

- Try all permutations and pick the one with smallest weight.
- Runtime: $\Theta(n! \cdot n) = n \cdot 2^{\Theta(n \log n)}$

TSP – Dynamic programming

Bellman-Held-Karp algorithm

Idea.

- Reuse optimal substructures with dynamic programming.

Richard M. Karp

Richard E. Bellman

TSP – Dynamic programming

Bellman-Held-Karp algorithm

Idea.

- Reuse optimal substructures with dynamic programming.
- Select a starting vertex $s \in V$.

s

Richard M. Karp

Richard E. Bellman

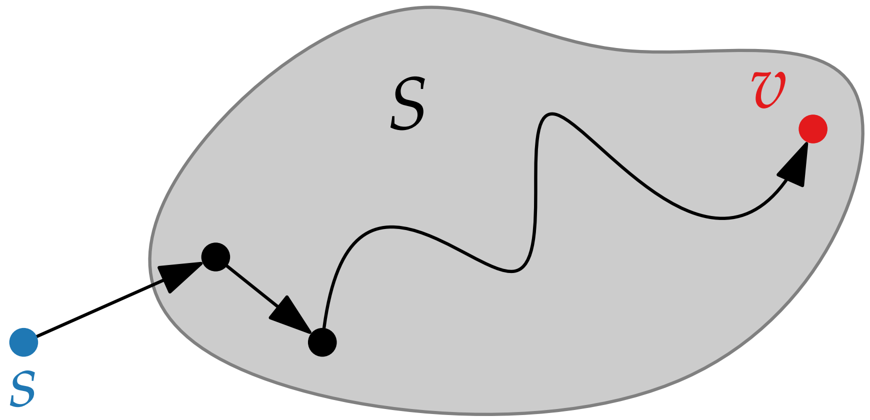
TSP – Dynamic programming

Bellman-Held-Karp algorithm

Idea.

- Reuse optimal substructures with dynamic programming.
- Select a starting vertex $s \in V$.
- For each $S \subseteq V - s$ and $v \in S$, let:

  $\mathrm{OPT}[S, v] =$ length of a shortest s-v-path
  that visits precisely the vertices of $S \cup \{s\}$.



Richard M. Karp

Richard E. Bellman

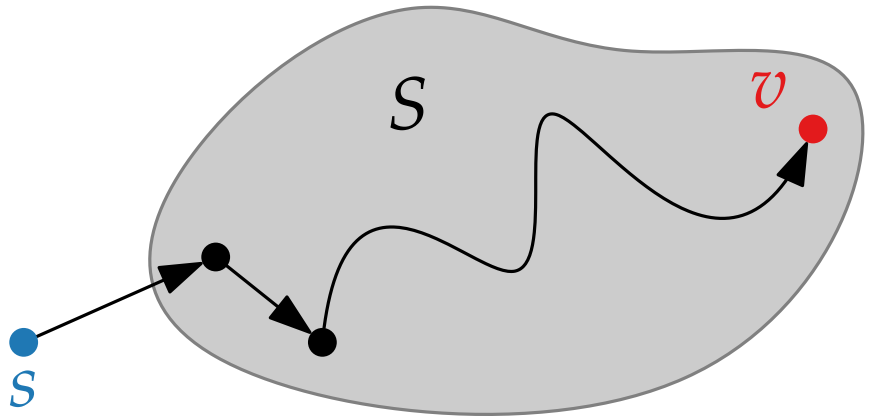
TSP – Dynamic programming

Bellman-Held-Karp algorithm

Idea.

- Reuse optimal substructures with dynamic programming.
- Select a starting vertex $s \in V$.
- For each $S \subseteq V - s$ and $v \in S$, let:

  $\mathrm{OPT}[S, v]$ = length of a shortest s-v-path
  that visits precisely the vertices of $S \cup \{s\}$.



- Use $\mathrm{OPT}[S - v, u]$ to compute $\mathrm{OPT}[S, v]$.

Richard M. Karp

Richard E. Bellman

TSP – Dynamic programming

Details.

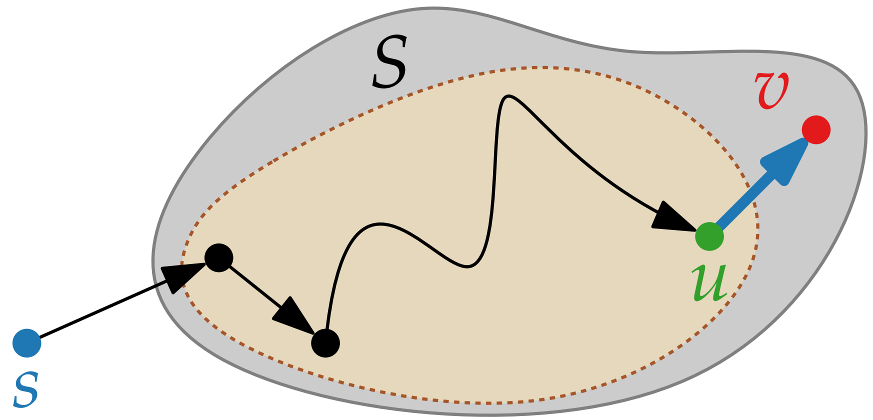
- The base case $S = \{v\}$ is easy: $\mathrm{OPT}[\{v\}, v] = d(s, v)$.

TSP – Dynamic programming

Details.

- The base case $S = \{v\}$ is easy: $\text{OPT}[\{v\}, v] = d(s, v)$.
- When $|S| \geq 2$, compute $\text{OPT}[S, v]$ recursively:

$$\text{OPT}[S, v] = \min\{\text{OPT}[S - v, u] + d(u, v) \mid u \in S - v\}$$

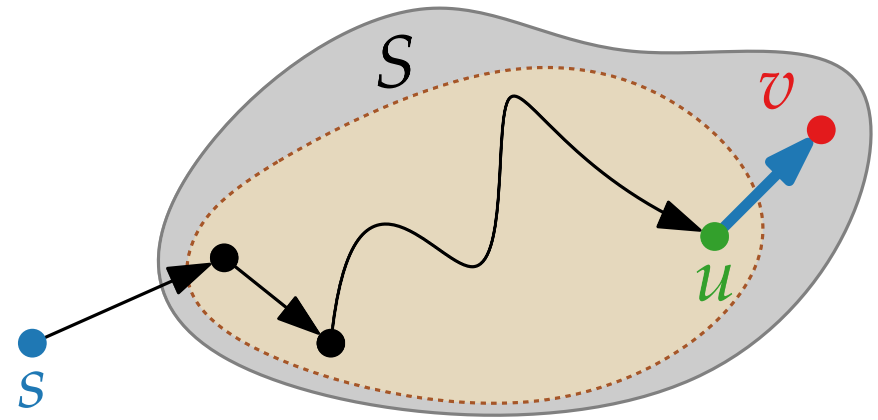


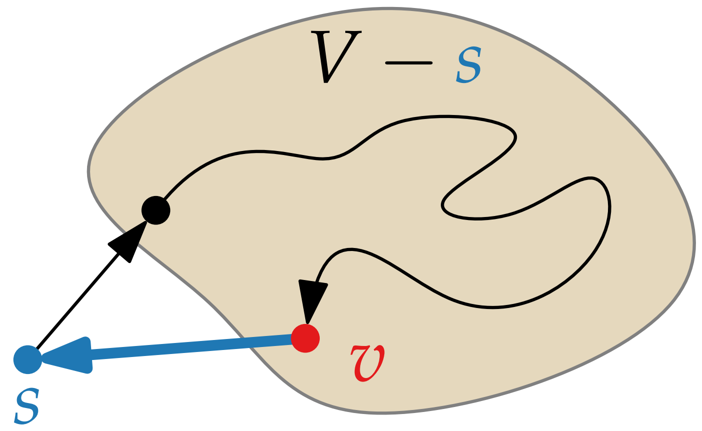
TSP – Dynamic programming

Details.

- The base case $S = \{v\}$ is easy: $\mathrm{OPT}[\{v\}, v] = d(s, v)$.
- When $|S| \geq 2$, compute $\mathrm{OPT}[S, v]$ recursively:

$$\mathrm{OPT}[S, v] = \min\{\mathrm{OPT}[S - v, u] + d(u, v) \mid u \in S - v\}$$





- After computing $\mathrm{OPT}[S, v]$ for each $S \subseteq V - s$ and each $v \in V - s$, the optimal solution is easily obtained as follows:

$$\mathrm{OPT} = \min\{\mathrm{OPT}[V - s, v]\} + d(v, s) \mid v \in V - s\}$$

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
- $\mathrm{OPT}[\{v\}, v] = c(s, v)$

for $j \leftarrow 2$ **to** $n - 1$ **do**
- **foreach** $S \subseteq V - s$ with $|S| = j$ **do**
  - **foreach** $v \in S$ **do**
    - $\mathrm{OPT}[S, v] \leftarrow \min\{ \mathrm{OPT}[S - v, u]$
      $+c(u, v) \mid u \in S - v \}$

return $\min\{ \mathrm{OPT}[V - s, v] + c(v, s) \mid v \in V - s \}$

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
- $\mathrm{OPT}[\{v\}, v] = c(s, v)$

for $j \leftarrow 2$ **to** $n - 1$ **do**
- **foreach** $S \subseteq V - s$ with $|S| = j$ **do**
  - **foreach** $v \in S$ **do**
    - $\mathrm{OPT}[S, v] \leftarrow \min\{\, \mathrm{OPT}[S - v, u] + c(u, v) \mid u \in S - v \,\}$

return $\min\{\, \mathrm{OPT}[V - s, v] + c(v, s) \mid v \in V - s \,\}$

- A shortest tour can be produced by backtracking the DP table (as usual).

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
  $\mathsf{OPT}[\{v\}, v] = c(s, v)$

for $j \leftarrow 2$ **to** $n - 1$ **do**
  foreach $S \subseteq V - s$ with $|S| = j$ **do**
    foreach $v \in S$ **do**
      $\mathsf{OPT}[S, v] \leftarrow \min\{\, \mathsf{OPT}[S - v, u]$
      $+c(u, v) \mid u \in S - v \,\}$

return $\min\{\, \mathsf{OPT}[V - s, v] + c(v, s) \mid v \in V - s \,\}$

- A shortest tour can be produced by back-tracking the DP table (as usual).

Analysis.

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
  OPT$[\{v\}, v] = c(s, v)$

for $j \leftarrow 2$ **to** $n - 1$ **do**
  foreach $S \subseteq V - s$ with $|S| = j$ **do**
    foreach $v \in S$ **do**
      OPT$[S, v] \leftarrow \min\{$ OPT$[S - v, u]$
      $+c(u, v) \mid u \in S - v \}$ $\}\mathcal{O}(n)$

return $\min\{$ OPT$[V - s, v] + c(v, s) \mid v \in V - s \}$

- A shortest tour can be produced by back-tracking the DP table (as usual).

Analysis.

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
    $\text{OPT}[\{v\}, v] = c(s, v)$
for $j \leftarrow 2$ **to** $n - 1$ **do**
    foreach $S \subseteq V - s$ with $|S| = j$ **do**
        foreach $v \in S$ **do**
            $\text{OPT}[S, v] \leftarrow \min\{\, \text{OPT}[S - v, u] + c(u, v) \mid u \in S - v \,\}$
return $\min\{\, \text{OPT}[V - s, v] + c(v, s) \mid v \in V - s \,\}$

$\}\mathcal{O}(2^n)$

$\}\mathcal{O}(n)$

- A shortest tour can be produced by back-tracking the DP table (as usual).

Analysis.

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
  $\text{OPT}[\{v\}, v] = c(s, v)$
for $j \leftarrow 2$ **to** $n - 1$ **do**
  foreach $S \subseteq V - s$ with $|S| = j$ **do**
    foreach $v \in S$ **do**
      $\text{OPT}[S, v] \leftarrow \min\{\, \text{OPT}[S - v, u] + c(u, v) \mid u \in S - v \,\}$
return $\min\{\, \text{OPT}[V - s, v] + c(v, s) \mid v \in V - s \,\}$

$\}\mathcal{O}(2^n)$

$\}\mathcal{O}(n)$

- A shortest tour can be produced by back-tracking the DP table (as usual).

Analysis.

- innermost loop executes $\mathcal{O}(2^n \cdot n)$ iterations
- each takes $\mathcal{O}(n)$ time
- total of $\mathcal{O}(2^n n^2) = \mathcal{O}^*(2^n)$

TSP – Dynamic programming

Pseudocode.

Algorithm Bellmann-Held-Karp(G, c)

foreach $v \in V - s$ **do**
  $\mathrm{OPT}[\{v\}, v] = c(s, v)$

for $j \leftarrow 2$ **to** $n - 1$ **do** $\Big\}\mathcal{O}(2^n)$
  foreach $S \subseteq V - s$ with $|S| = j$ **do**
    foreach $v \in S$ **do** $\Big\}\mathcal{O}(n)$
      $\mathrm{OPT}[S, v] \leftarrow \min\{\, \mathrm{OPT}[S - v, u] + c(u, v) \mid u \in S - v \,\}$

return $\min\{\, \mathrm{OPT}[V - s, v] + c(v, s) \mid v \in V - s \,\}$

- A shortest tour can be produced by backtracking the DP table (as usual).

Analysis.

- innermost loop executes $\mathcal{O}(2^n \cdot n)$ iterations
- each takes $\mathcal{O}(n)$ time
- total of $\mathcal{O}(2^n n^2) = \mathcal{O}^*(2^n)$

- Space usage in $\Theta(2^n \cdot n)$
- or actually better? What table values do we need to store?

TSP – Discussion

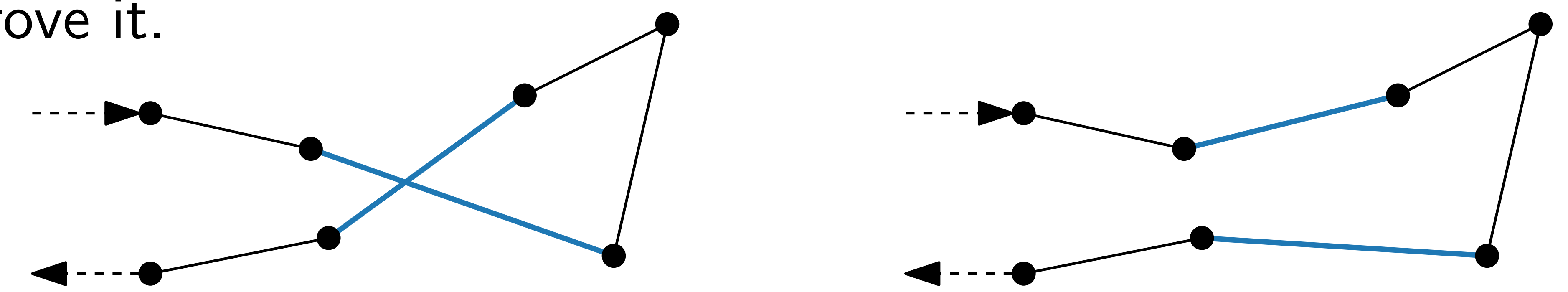
- DP algorithm that runs in $\mathcal{O}^*(2^n)$ time and $\mathcal{O}(2^n \cdot \overset{?}{n})$ space
- Brute-force runs in $2^{\mathcal{O}(n \log n)}$ time
  $\Rightarrow$ Sacrifice space for speedup

TSP – Discussion

- DP algorithm that runs in $\mathcal{O}^*(2^n)$ time and $\mathcal{O}(2^n \cdot \overset{?}{n})$ space
- Brute-force runs in $2^{\mathcal{O}(n \log n)}$ time
  $\Rightarrow$ Sacrifice space for speedup
- Many variants of TSP: symmetric, assymetric, metric, vehicle routing problem, ...
- Metric TSP can easily be 2-approximated. (Do you remember how?)
- Eucledian TSP considered in course Approxiomation Algorithms.

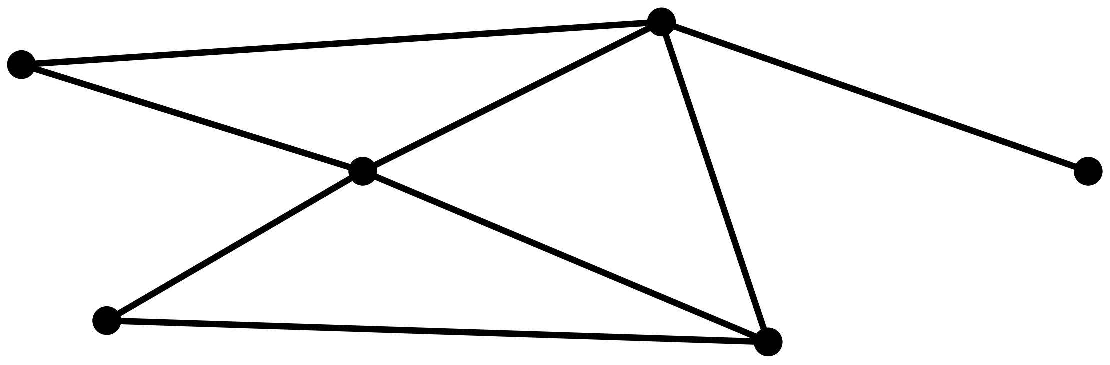
TSP – Discussion

- DP algorithm that runs in $\mathcal{O}^*(2^n)$ time and $\mathcal{O}(2^n \cdot \overset{?}{n})$ space
- Brute-force runs in $2^{\mathcal{O}(n \log n)}$ time
  $\Rightarrow$ Sacrifice space for speedup
- Many variants of TSP: symmetric, assymetric, metric, vehicle routing problem, ...
- Metric TSP can easily be 2-approximated. (Do you remember how?)
- Eucledian TSP considered in course Approximation Algorithms.
- In practice, one successful approach is to start with a greedily computed Hamiltonian cycle and then use 2-OPT and 3-OPT swaps to improve it.



Maximum Independent Set (MIS)

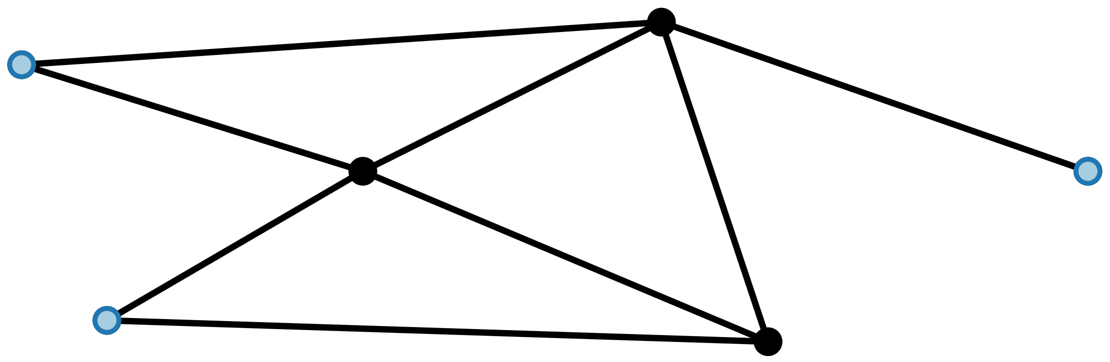
Input. Graph $G = (V, E)$ with n vertices.



Maximum Independent Set (MIS)

Input. Graph $G = (V, E)$ with n vertices.

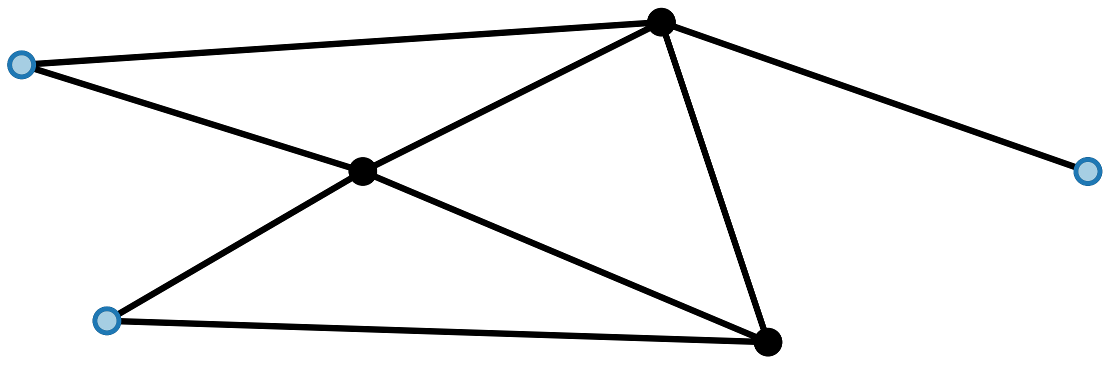
Output. Maximum size **independent** set, i.e., a largest set $U \subseteq V$, such that no pair of vertices in U are adjacent in G.



Maximum Independent Set (MIS)

Input. Graph $G = (V, E)$ with n vertices.

Output. Maximum size **independent** set, i.e., a largest set $U \subseteq V$, such that no pair of vertices in U are adjacent in G.



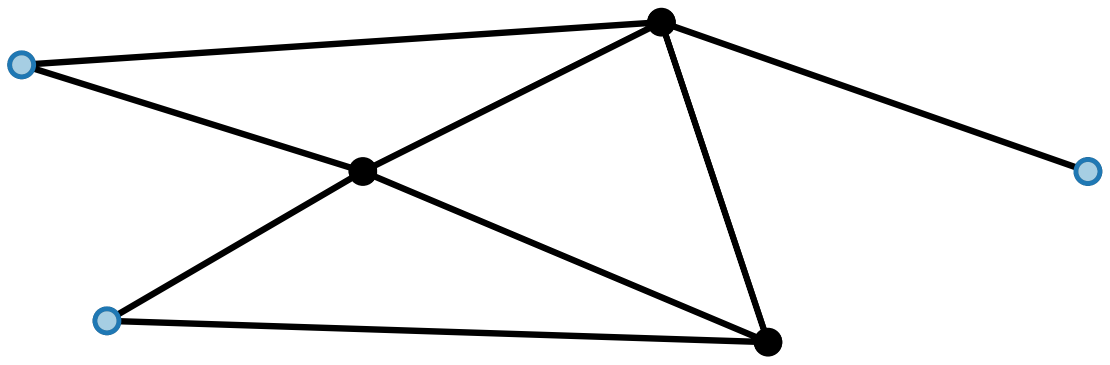
Brute-force.

- Try all subets of V.
- Runtime: $\mathcal{O}(2^n \cdot n)$

Maximum Independent Set (MIS)

Input. Graph $G = (V, E)$ with n vertices.

Output. Maximum size **independent** set, i.e., a largest set $U \subseteq V$, such that no pair of vertices in U are adjacent in G.



Brute-force.

- Try all subets of V.
- Runtime: $\mathcal{O}(2^n \cdot n)$

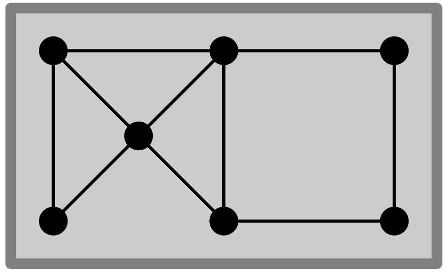
Naive MIS branching.

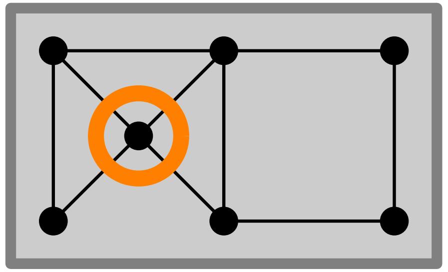
- Take a vertex v or don't take it.

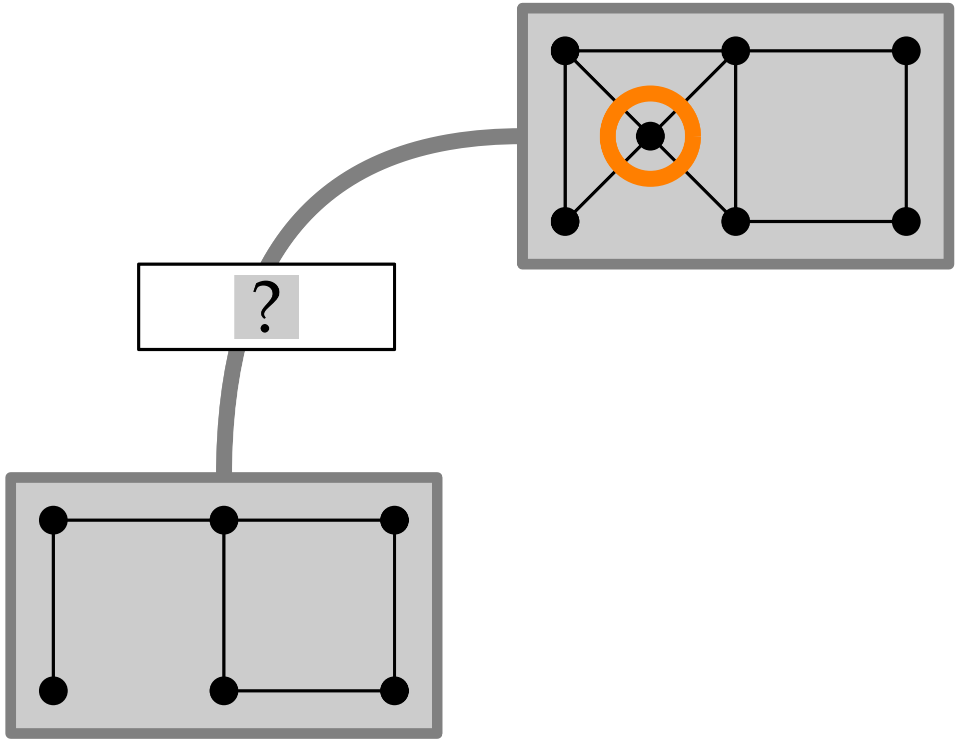
Algorithm NaiveMIS(G)

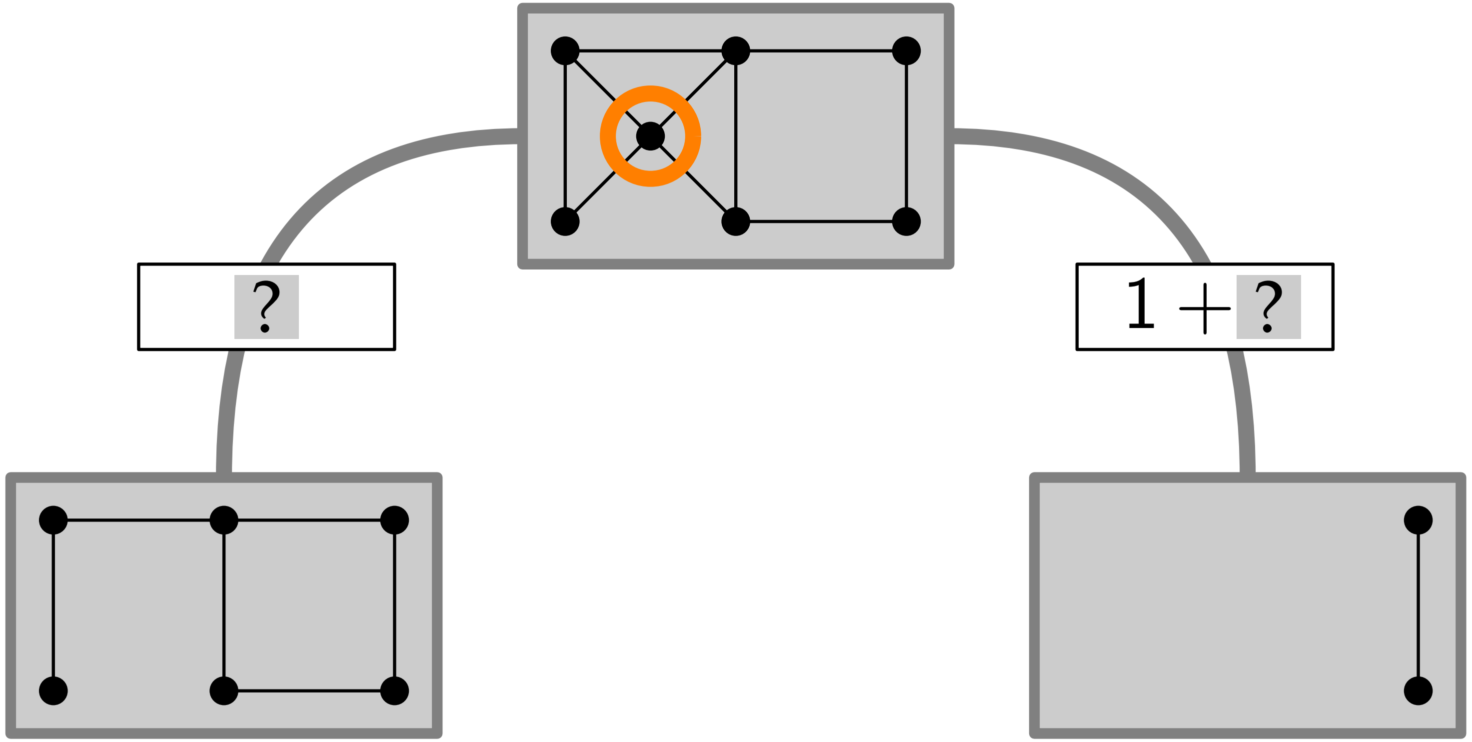
if $V = \emptyset$ **then**
  return 0

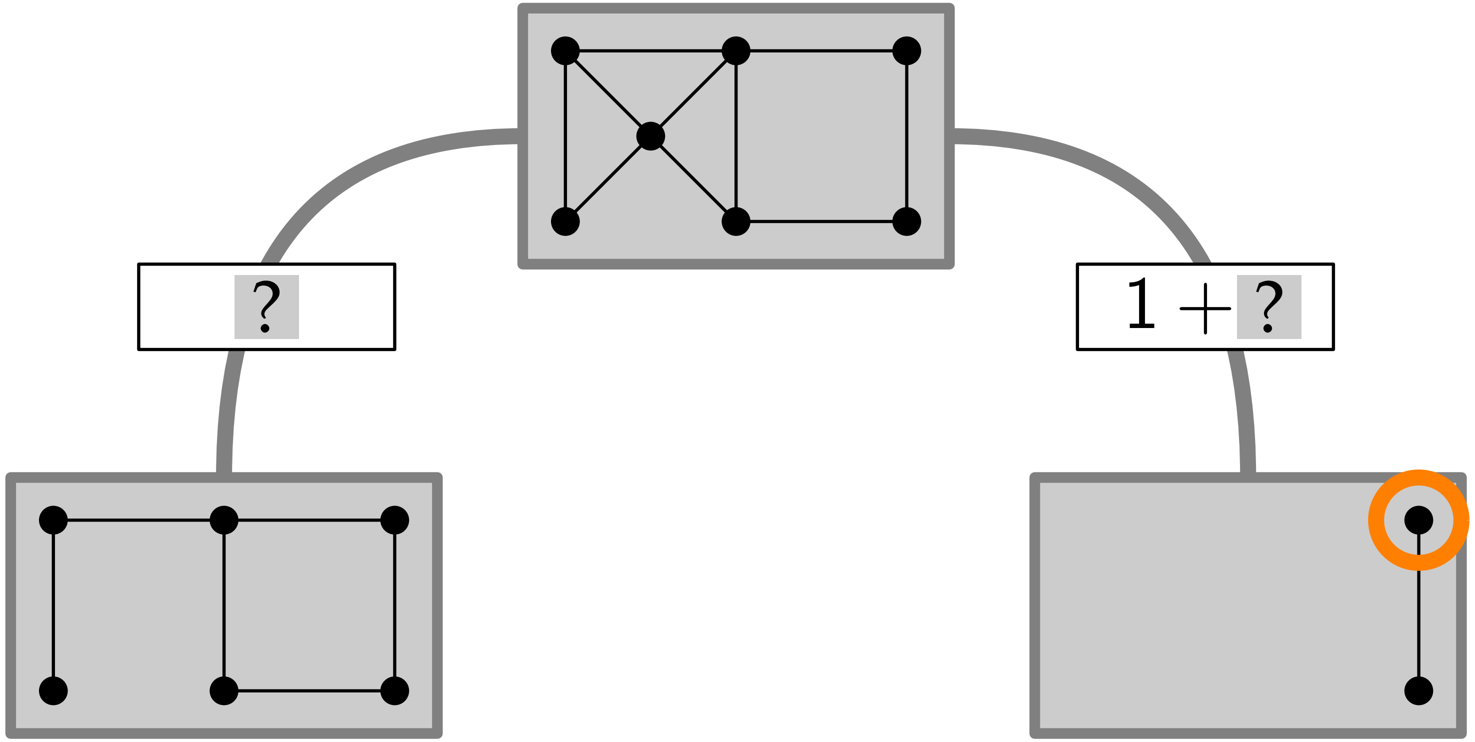
$v \leftarrow$ arbitrary vertex in $V(G)$
return max$\{1+$ NaiveMIS$(G - N(v) - \{v\})$,
 NaiveMIS$(G - \{v\})\}$

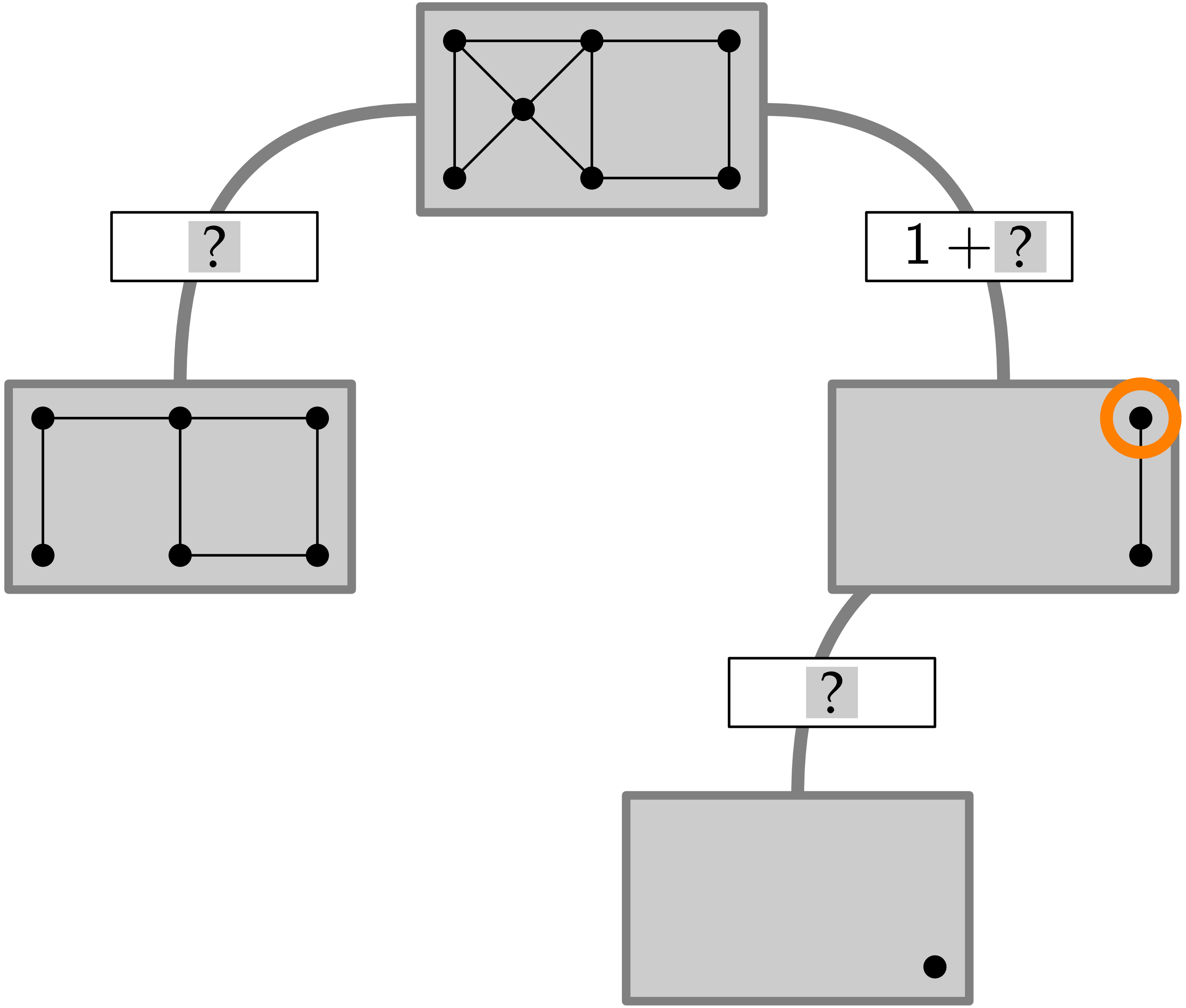


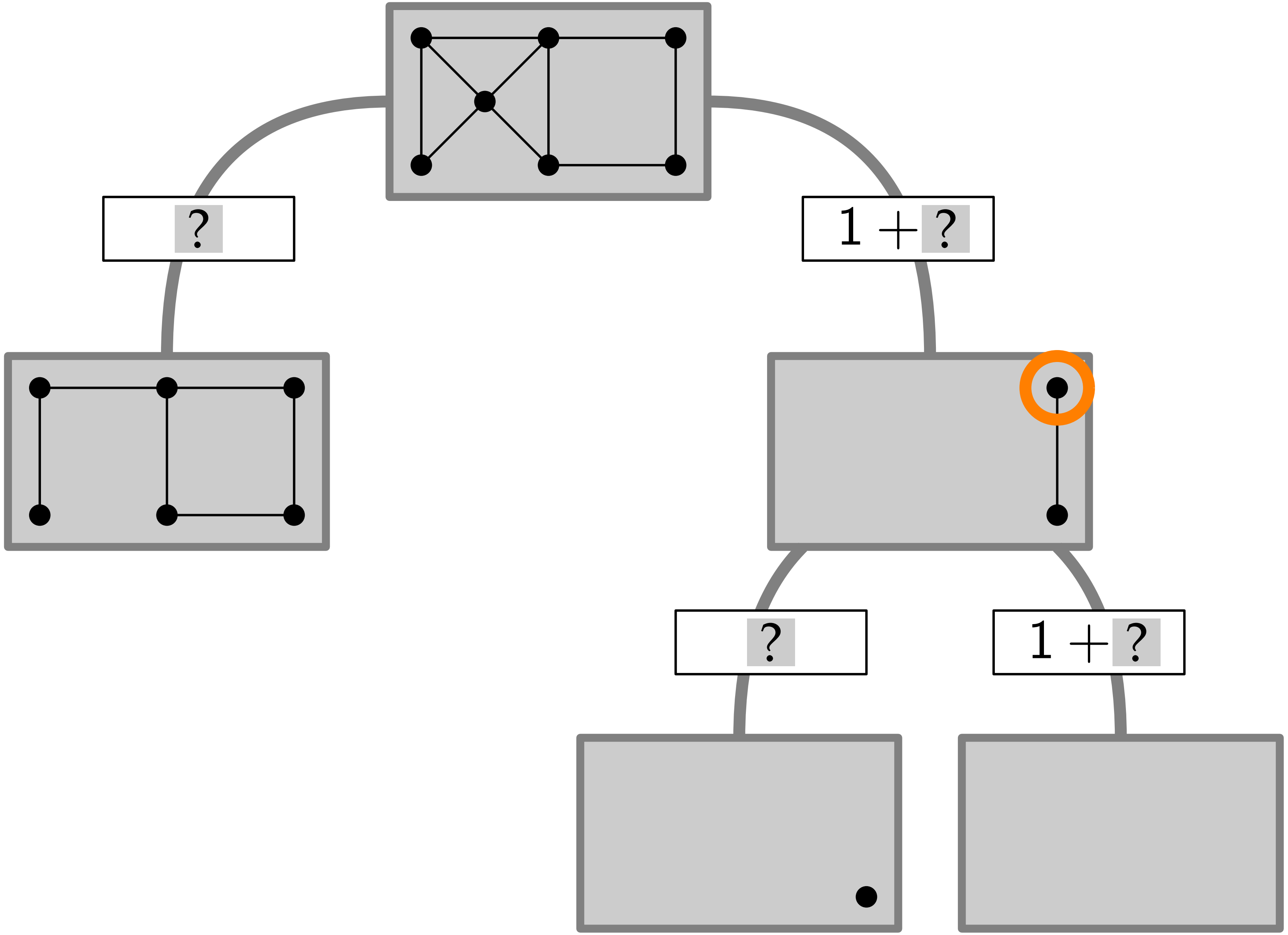


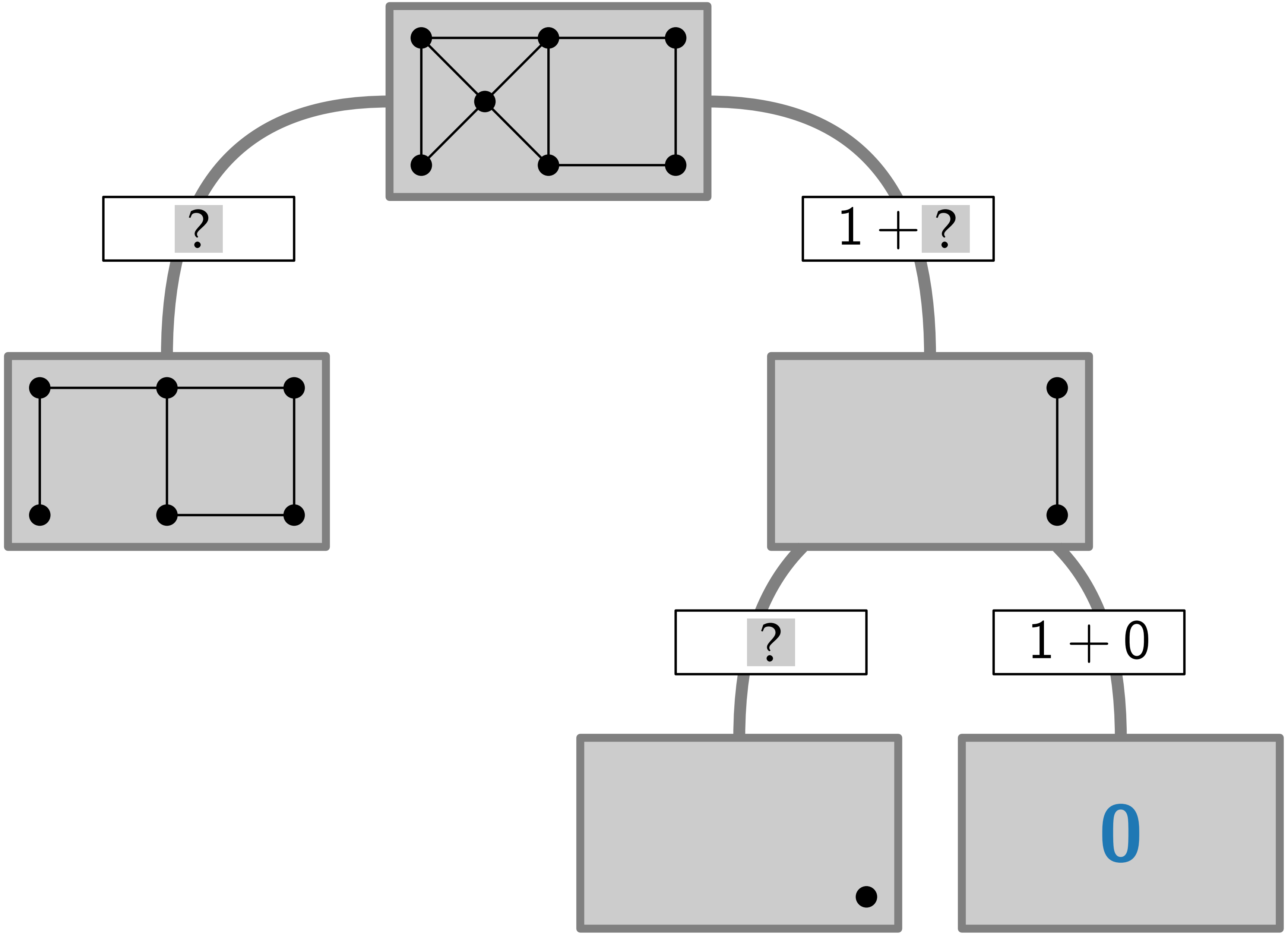


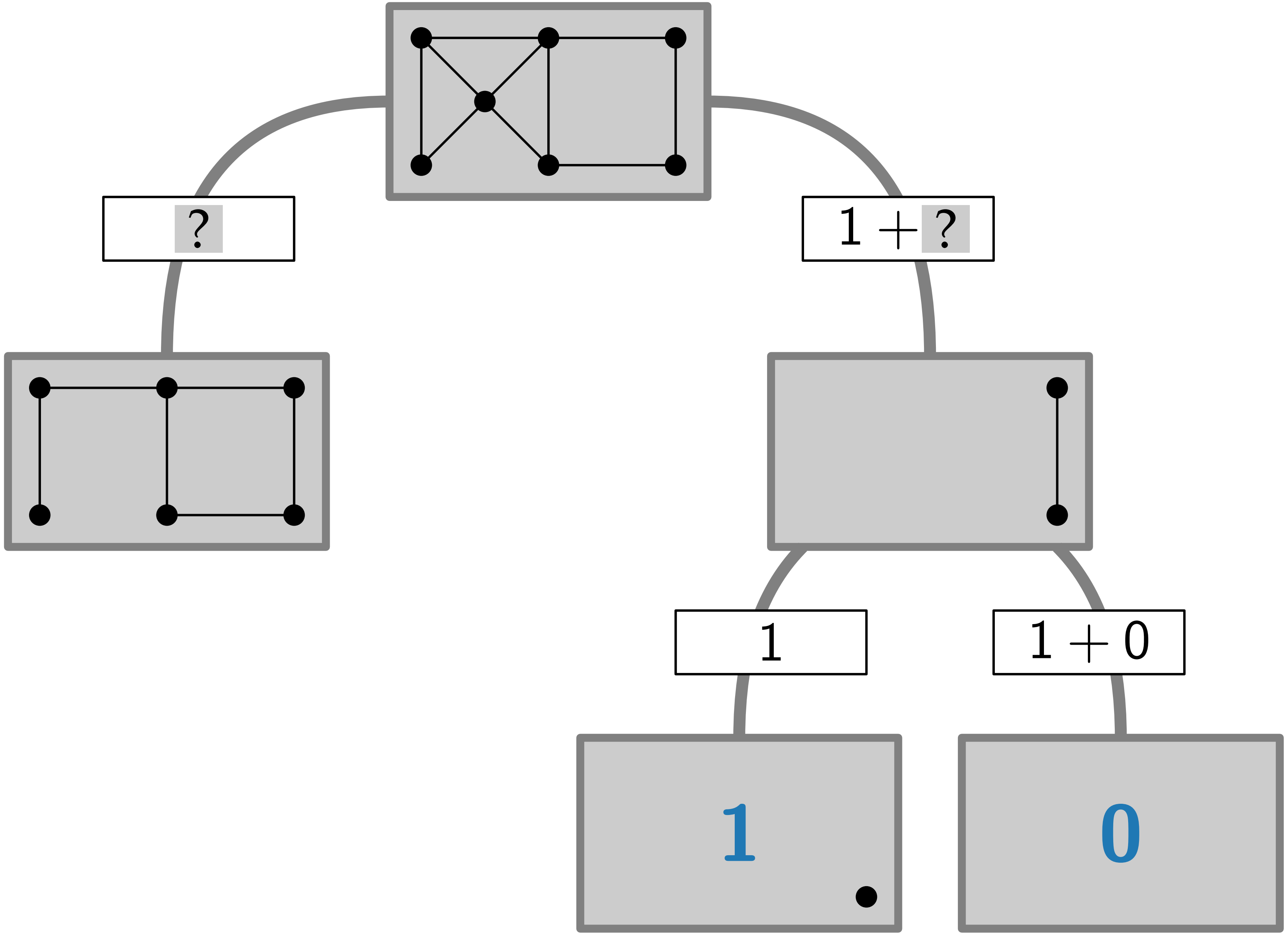

?
1 + ?

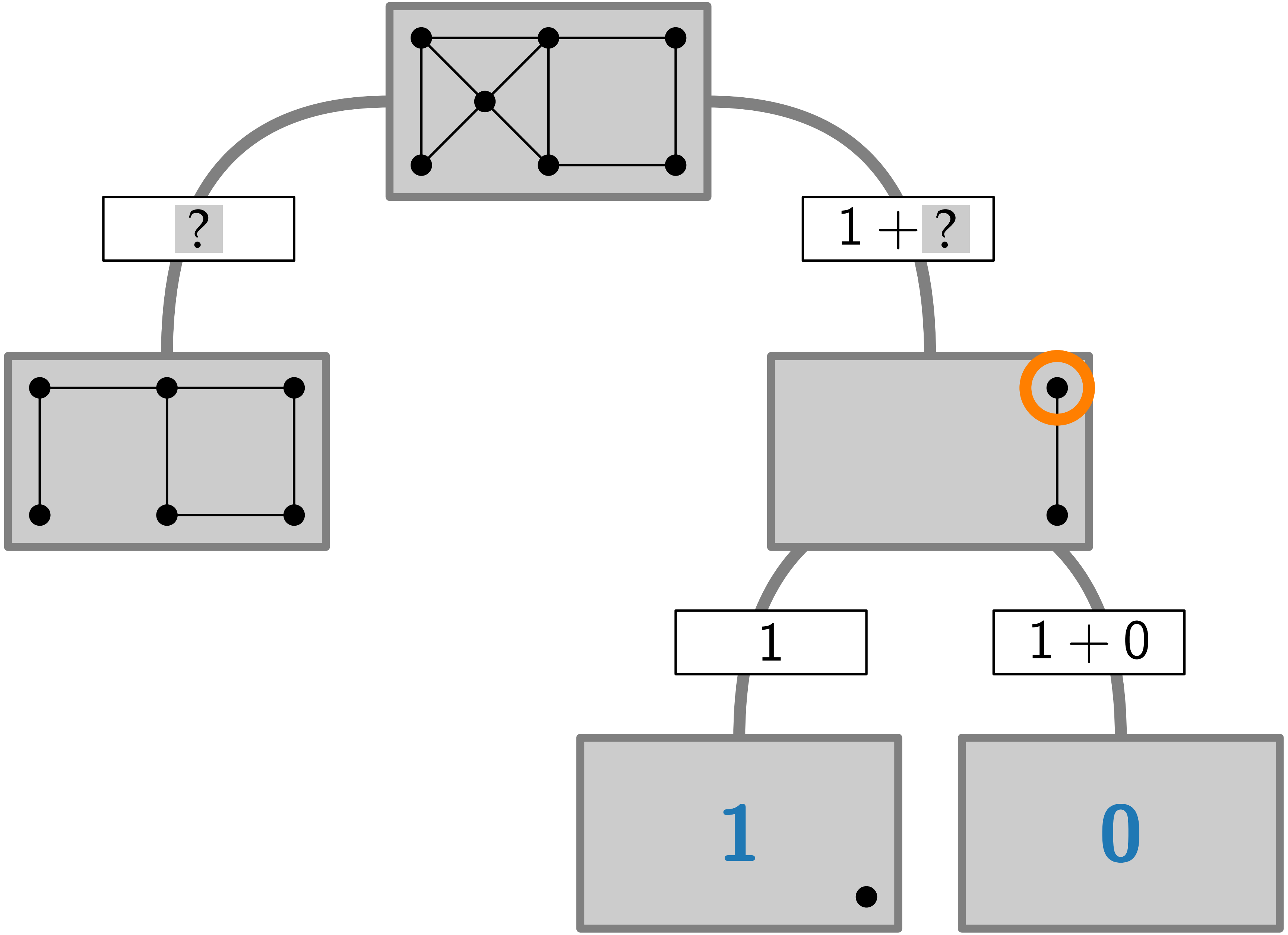

?
1 + ?

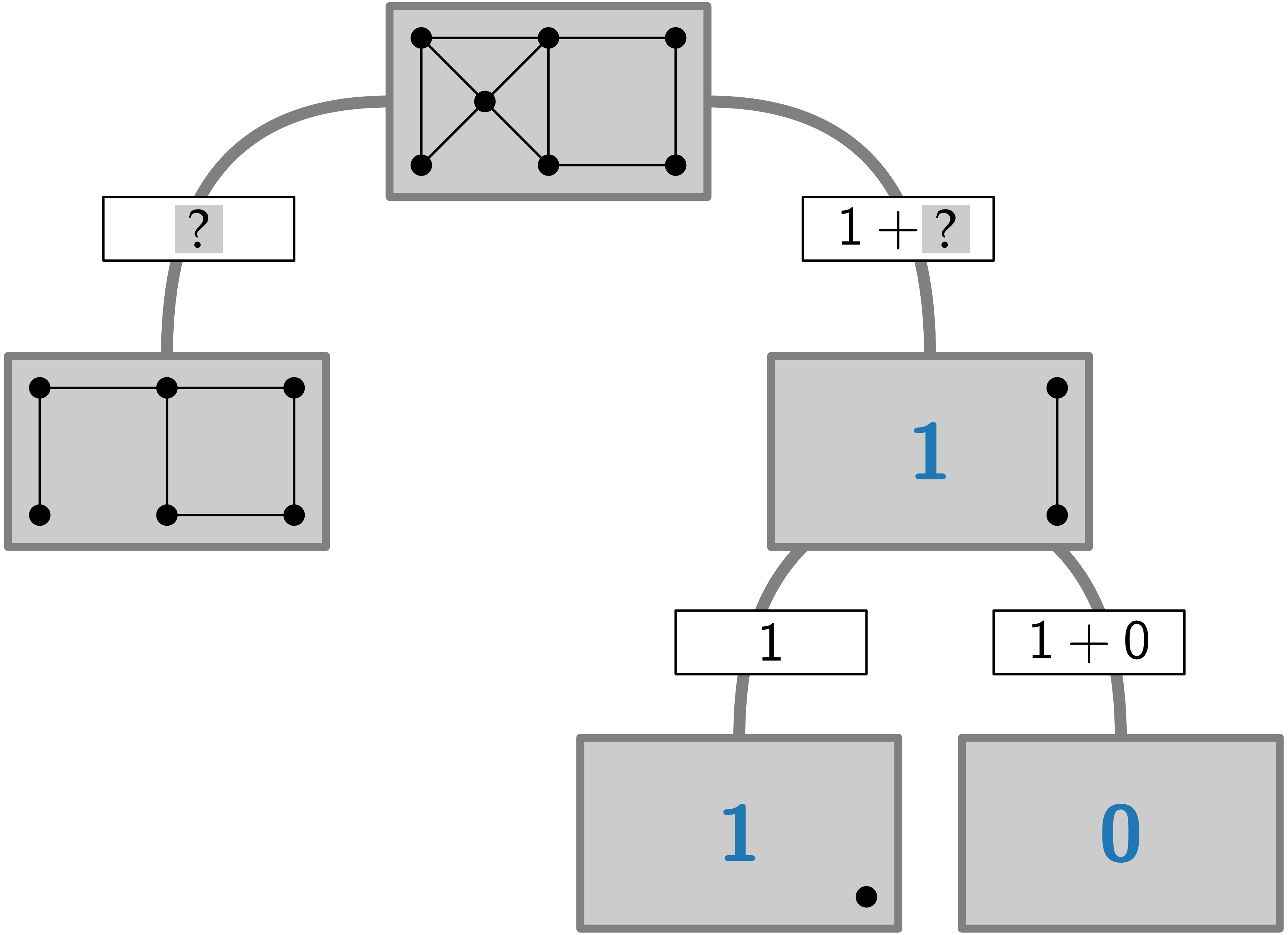

?
1 + ?
?


?
1 + ?
?
1 + ?

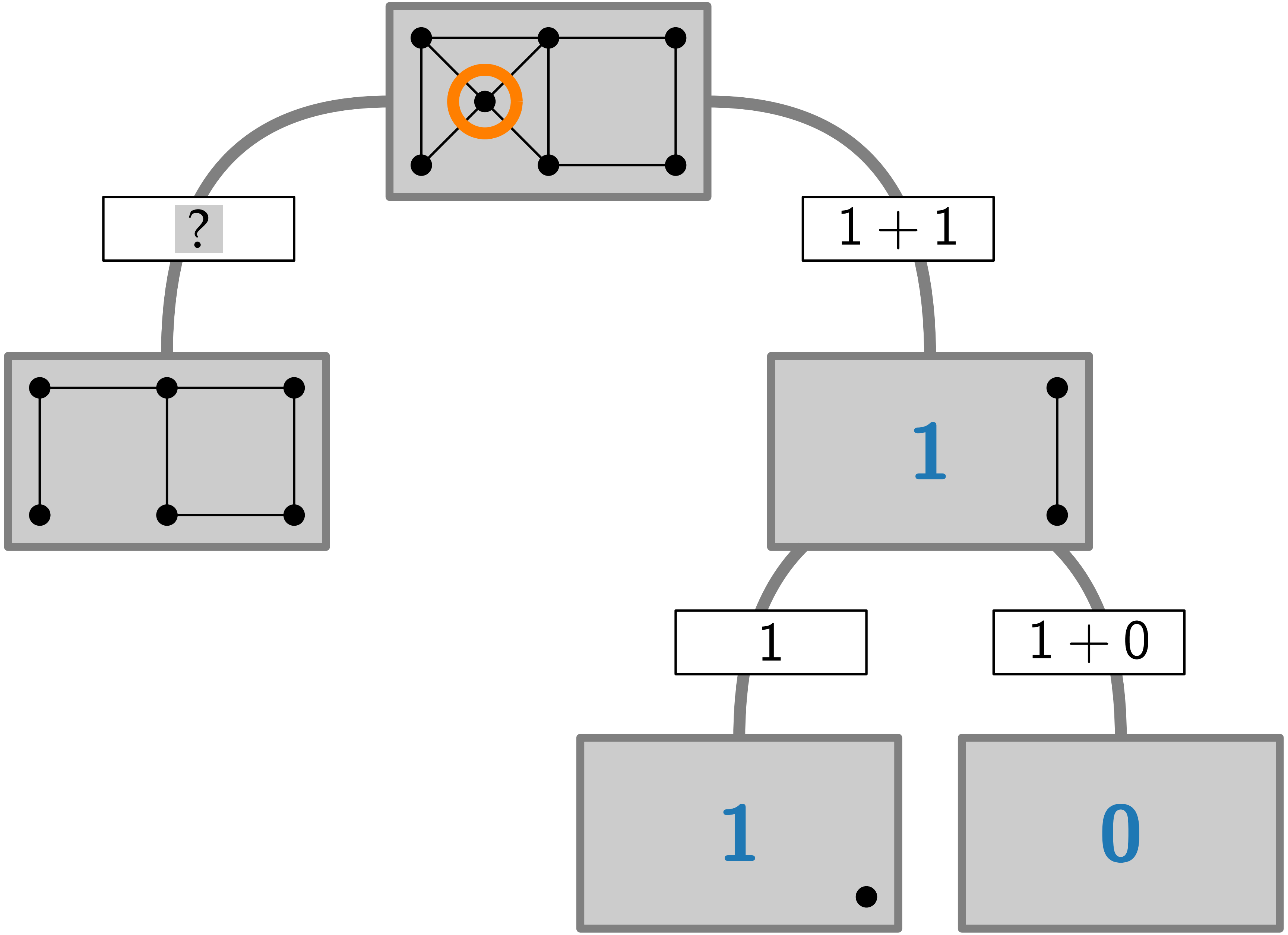

?
1 + ?
?
1 + 0
0

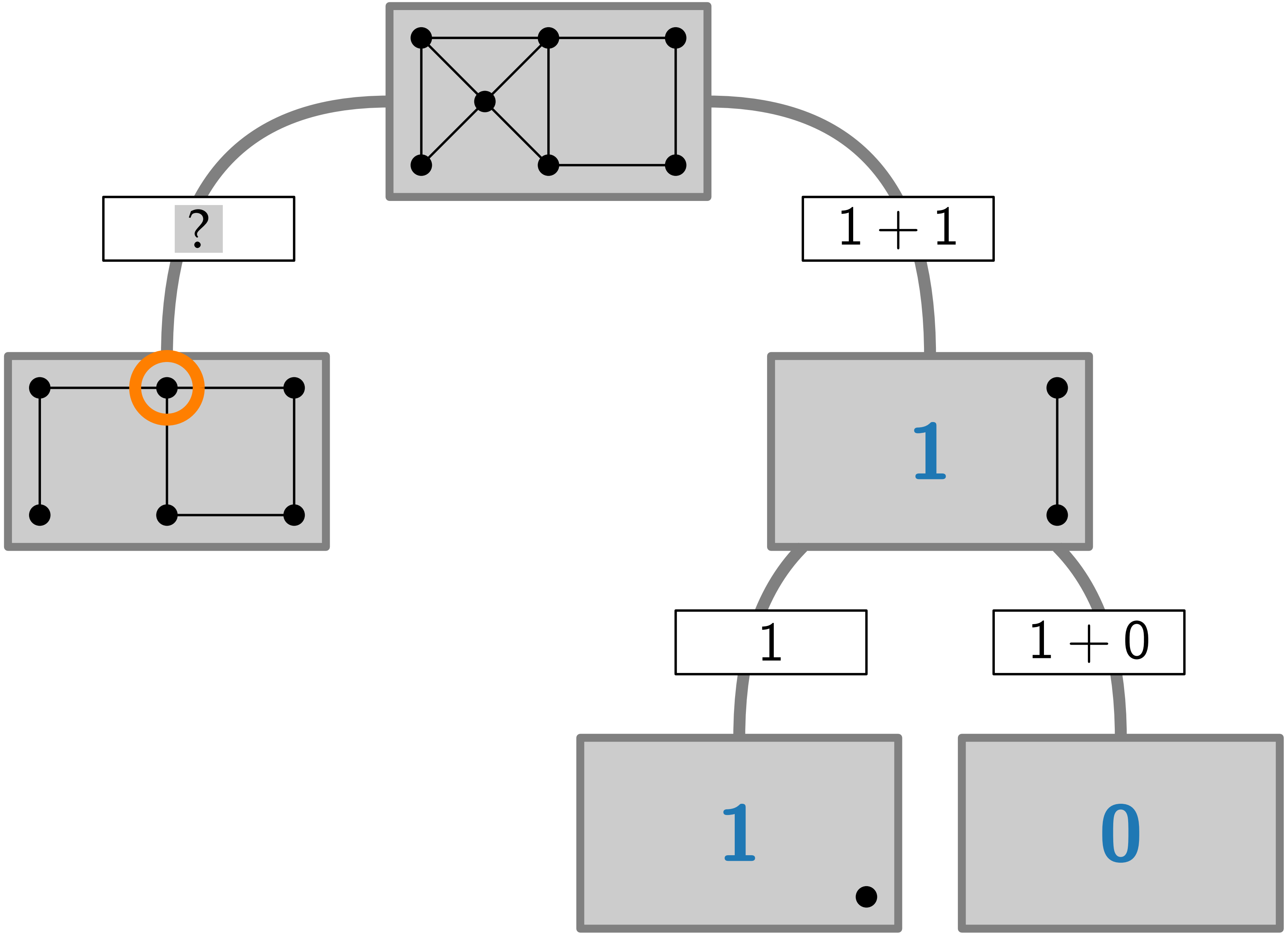

?
1 + ?
1
1 + 0
1
0

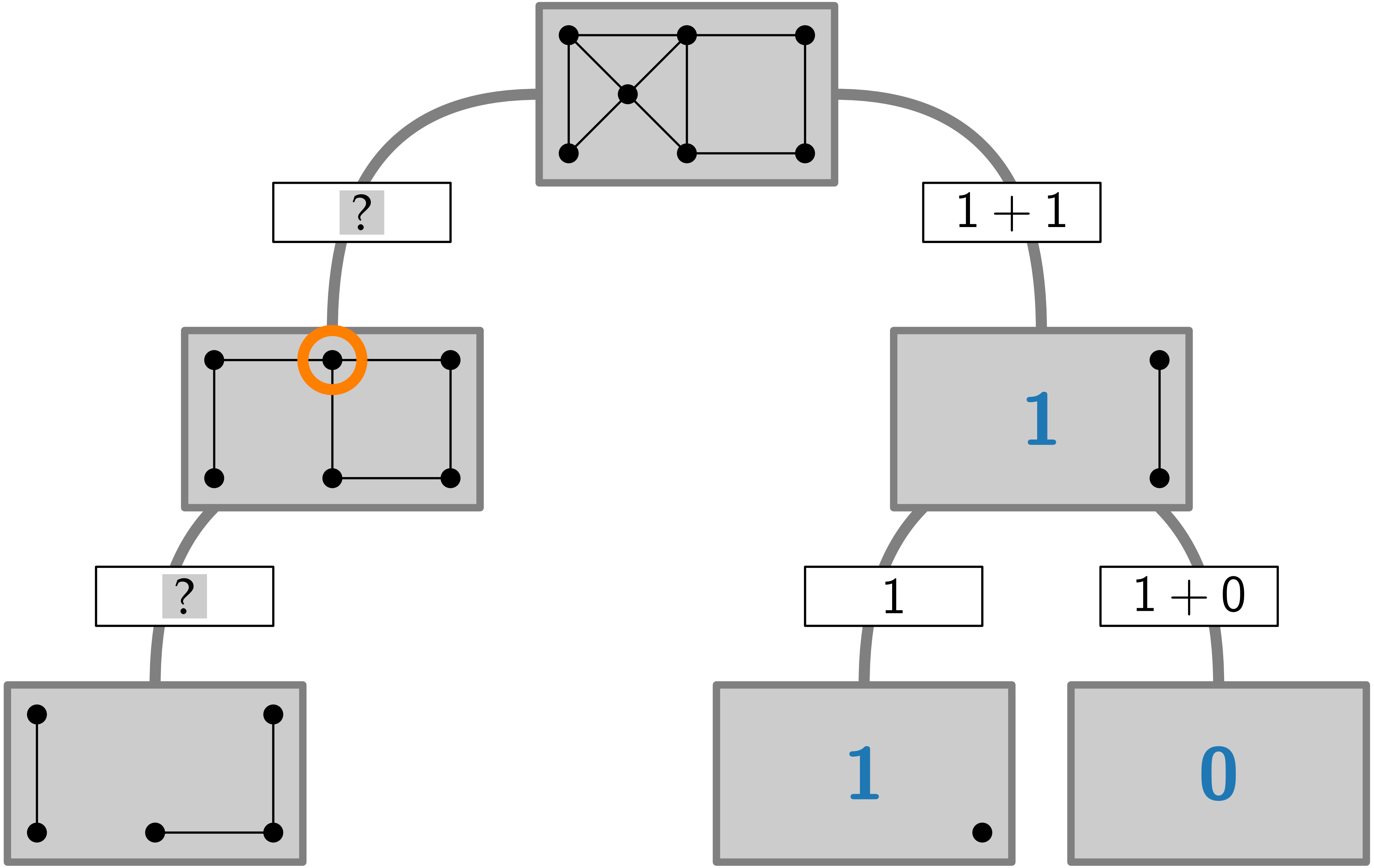

?
1 + ?
1
1 + 0
1
0

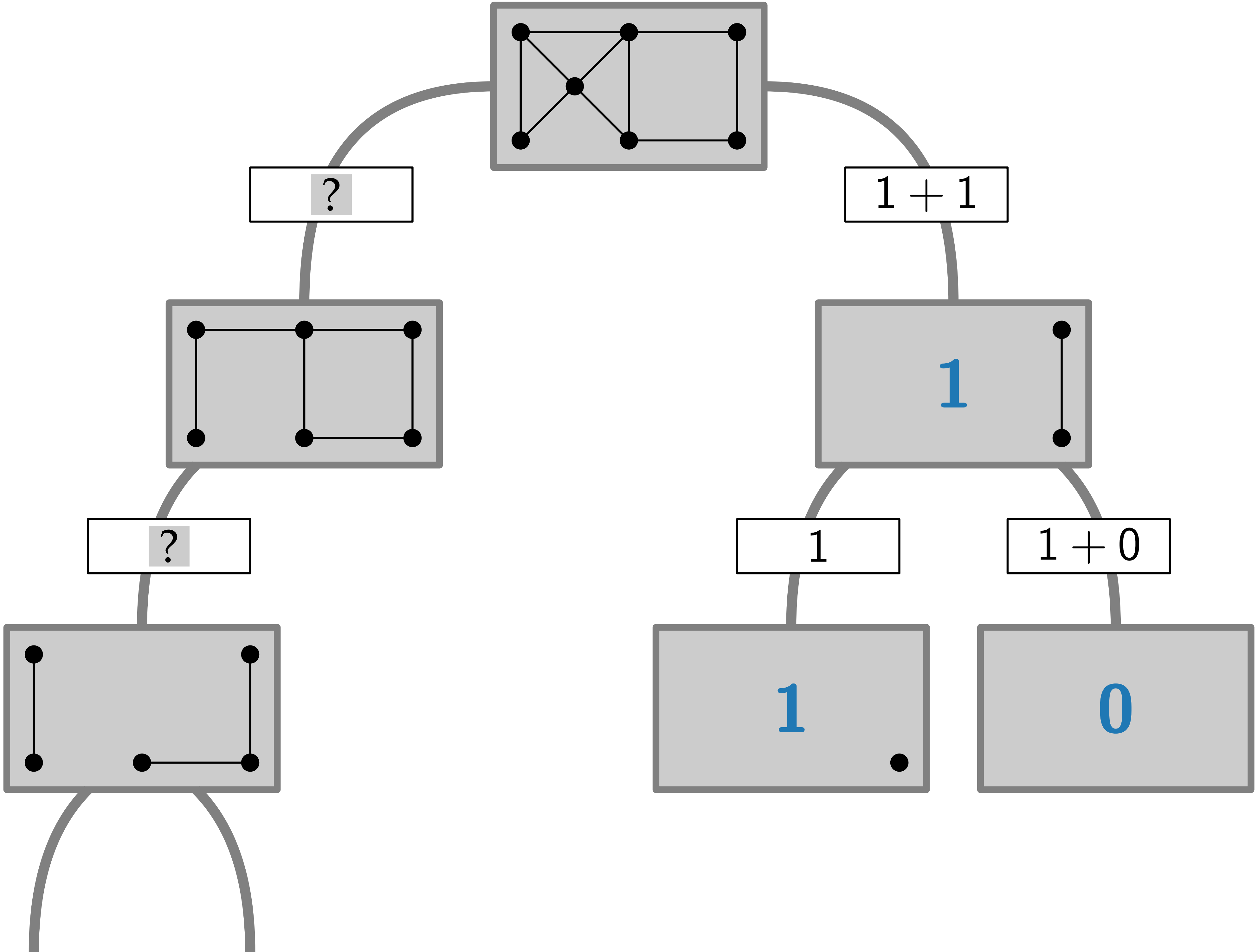


?
1 + ?
1
1
1 + 0
1
0

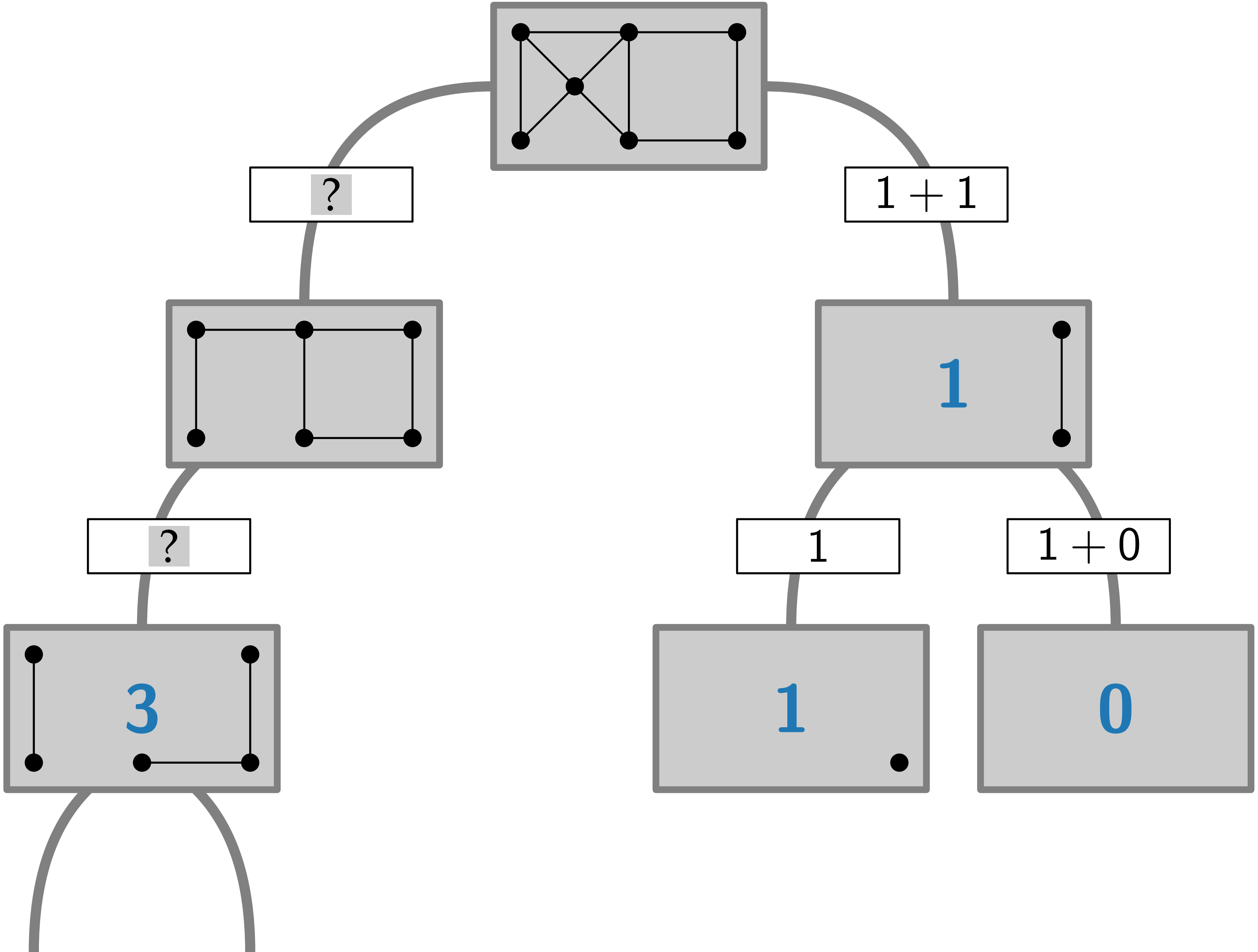

?
1 + 1
1
1
1 + 0
1
0

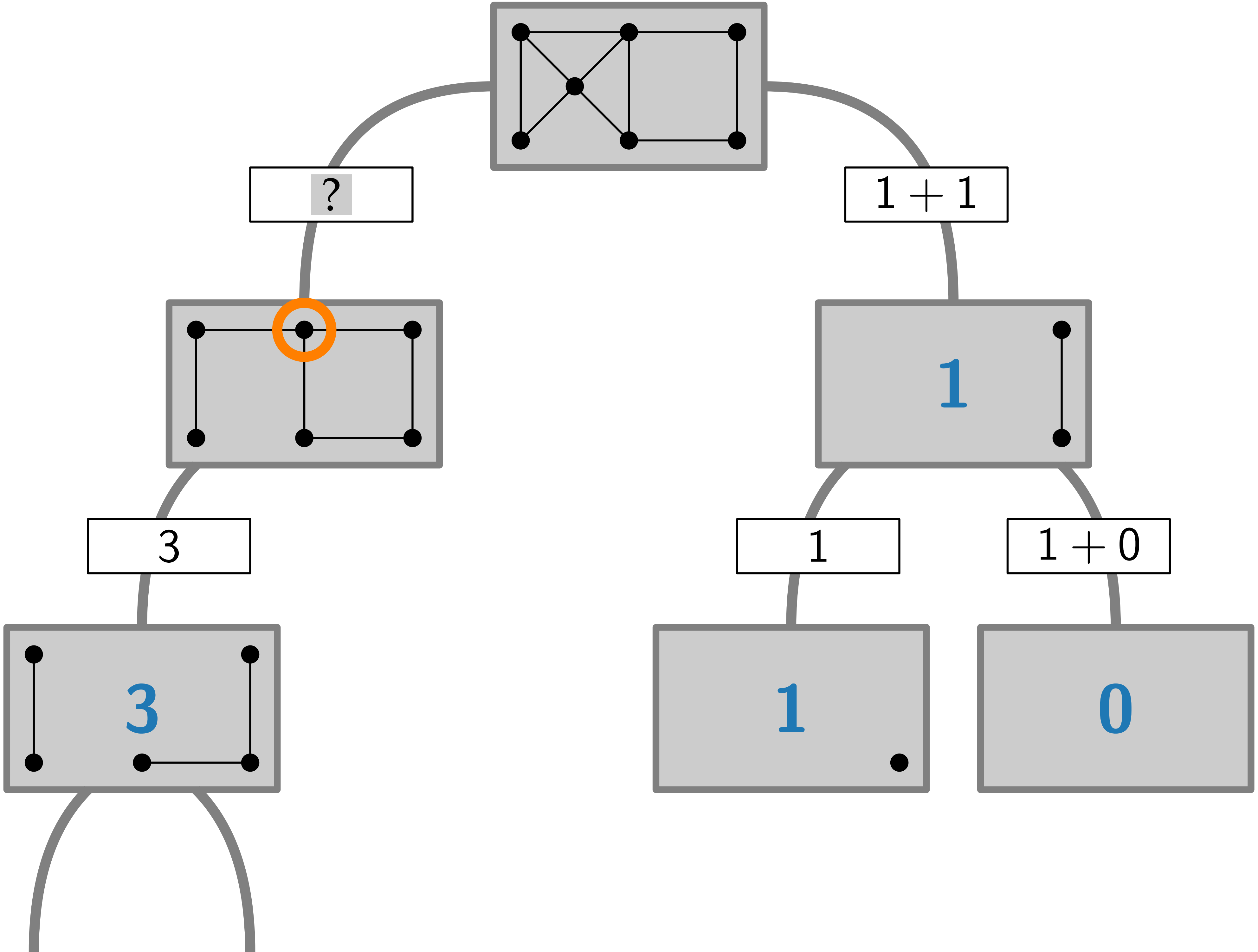

?
1 + 1
1
1
1 + 0
1
0


?
1 + 1
1
?
1
1 + 0
1
0

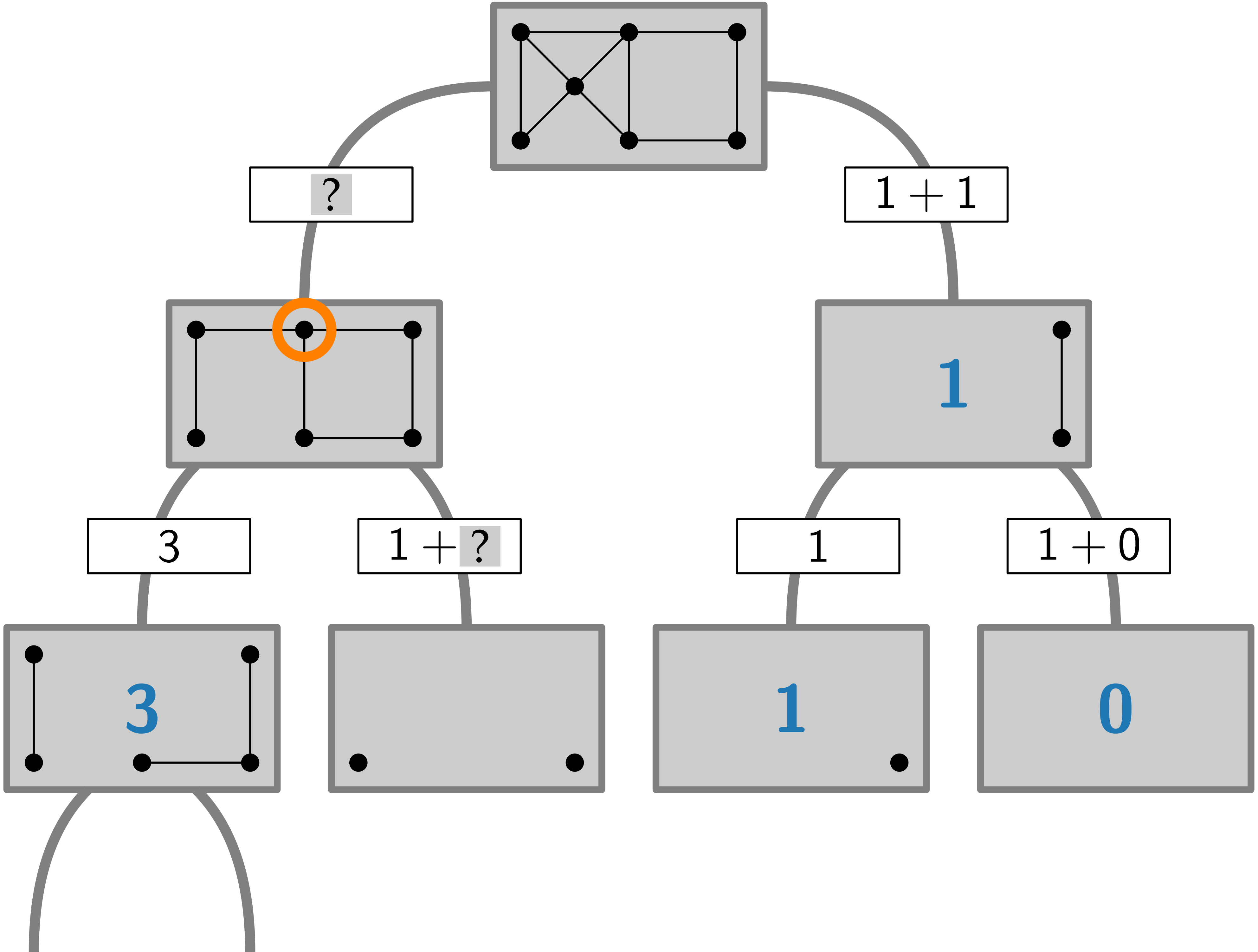


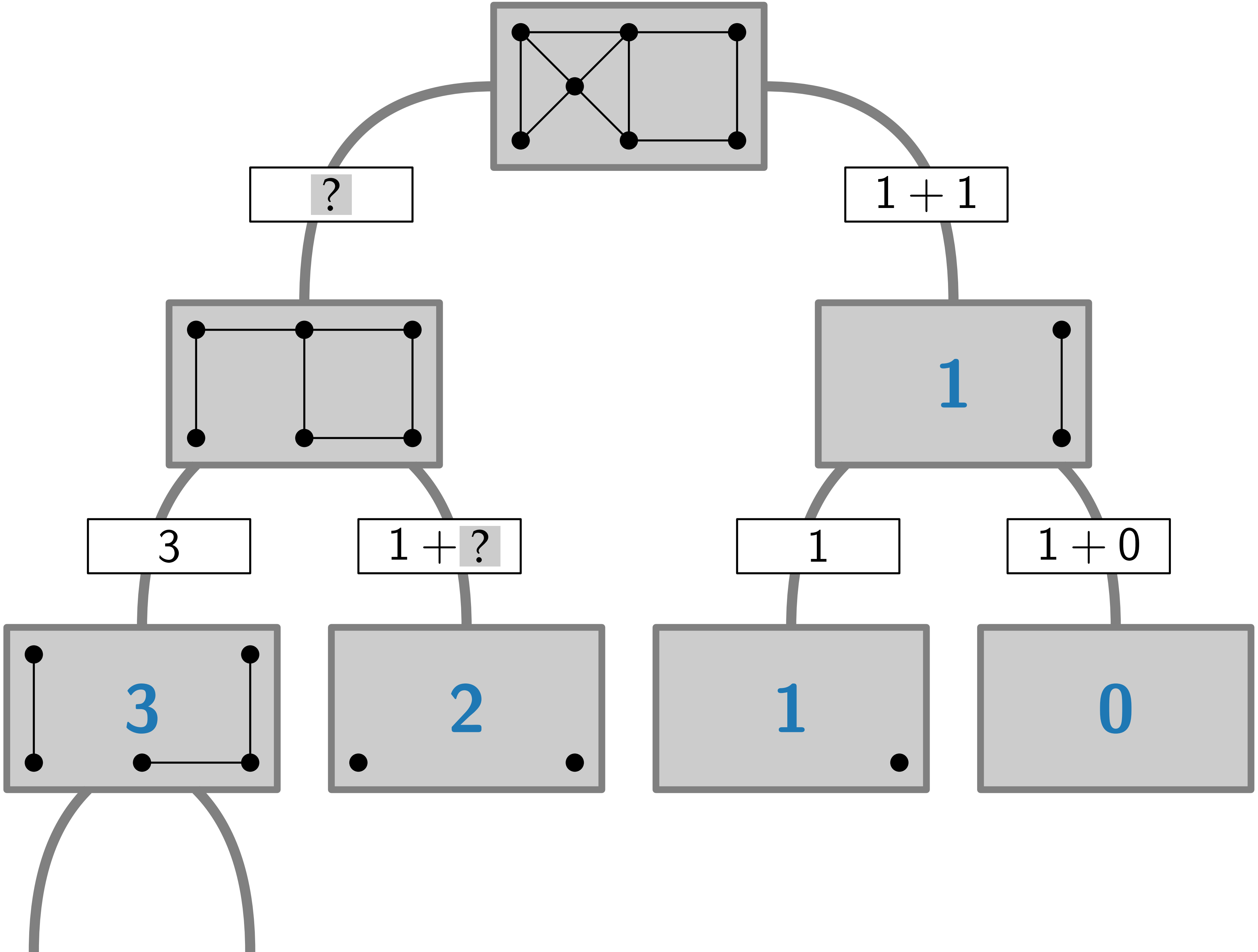
?
1 + 1
1
?
1
1 + 0
1
0

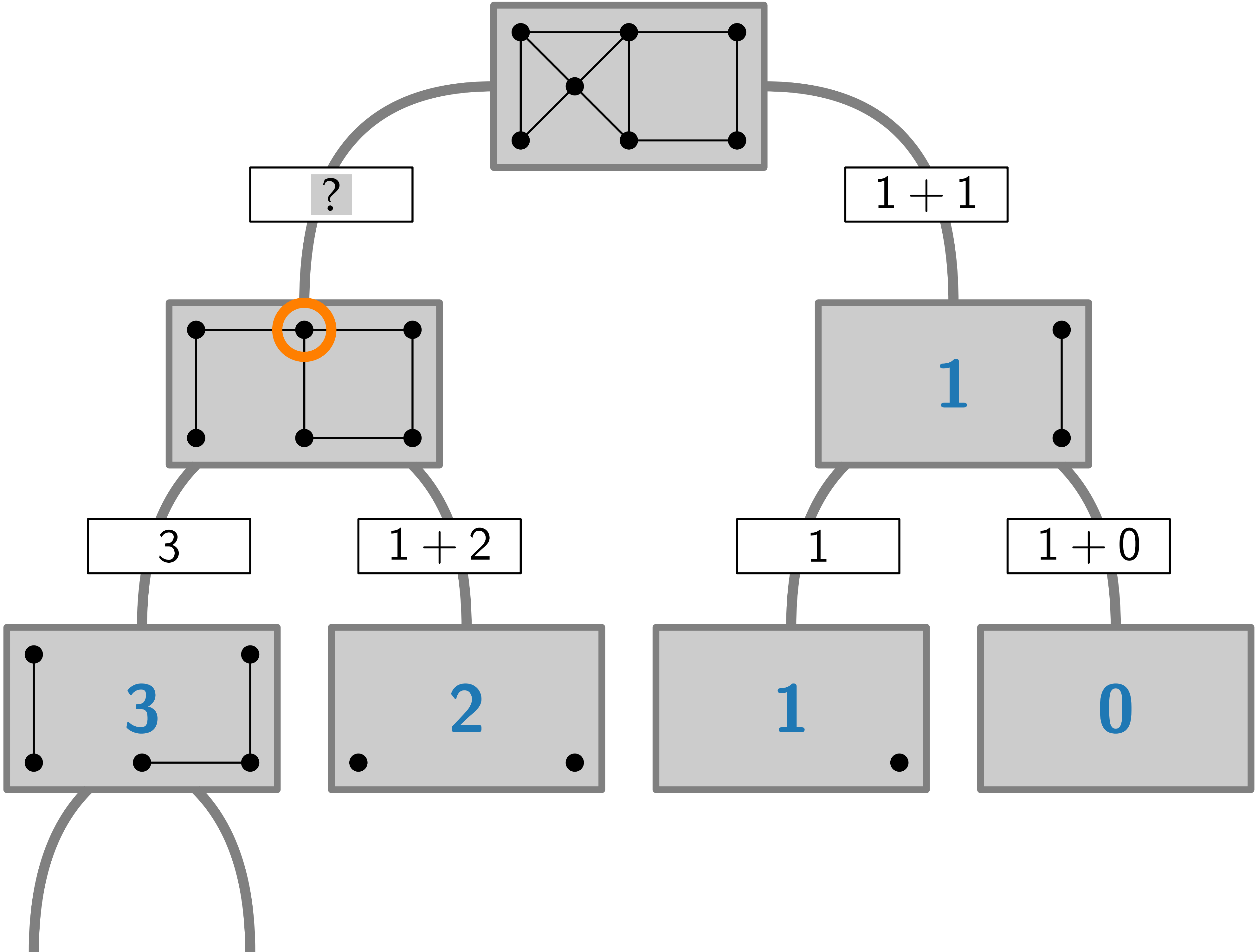

?
1 + 1
1
?
1
1 + 0
3
1
0

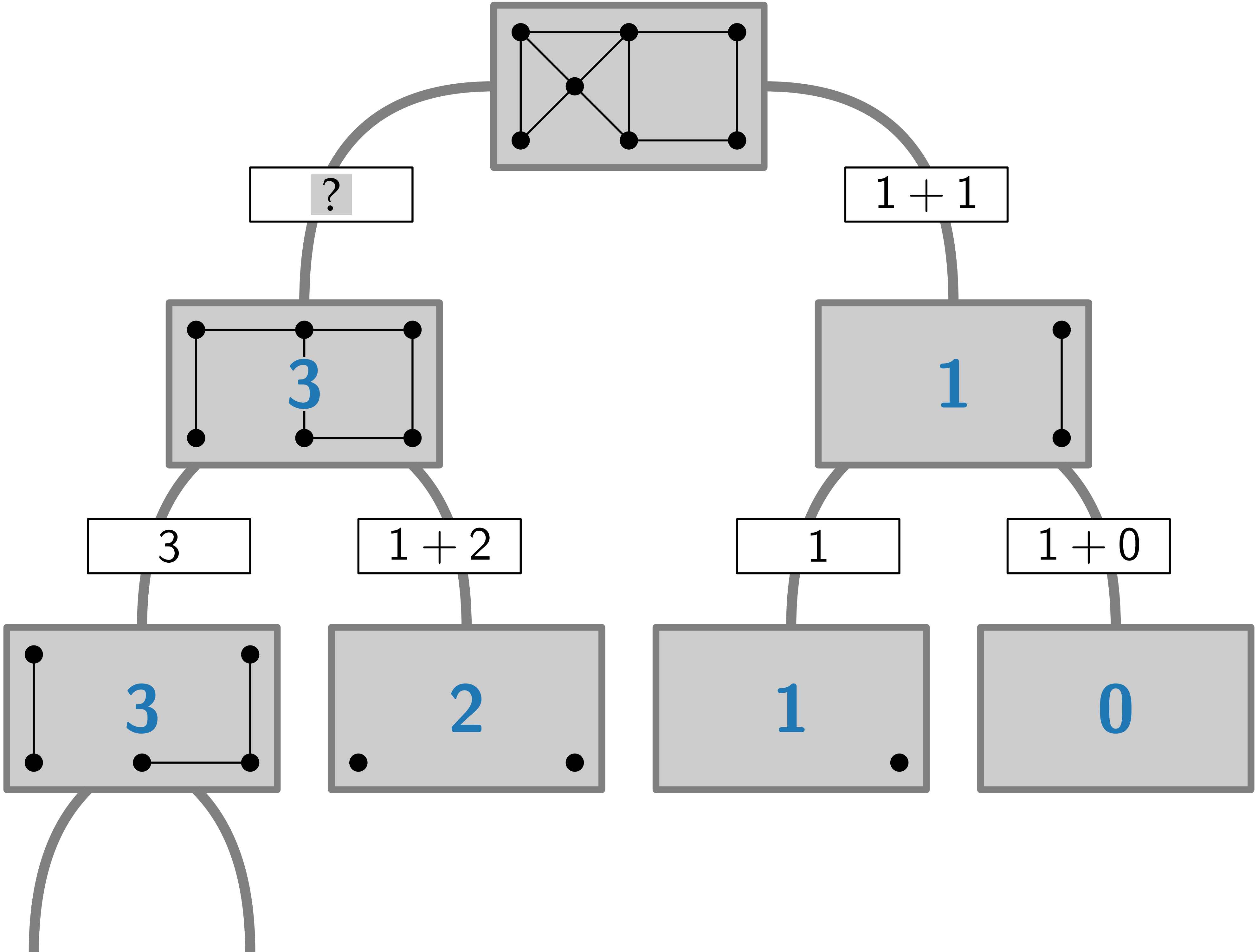


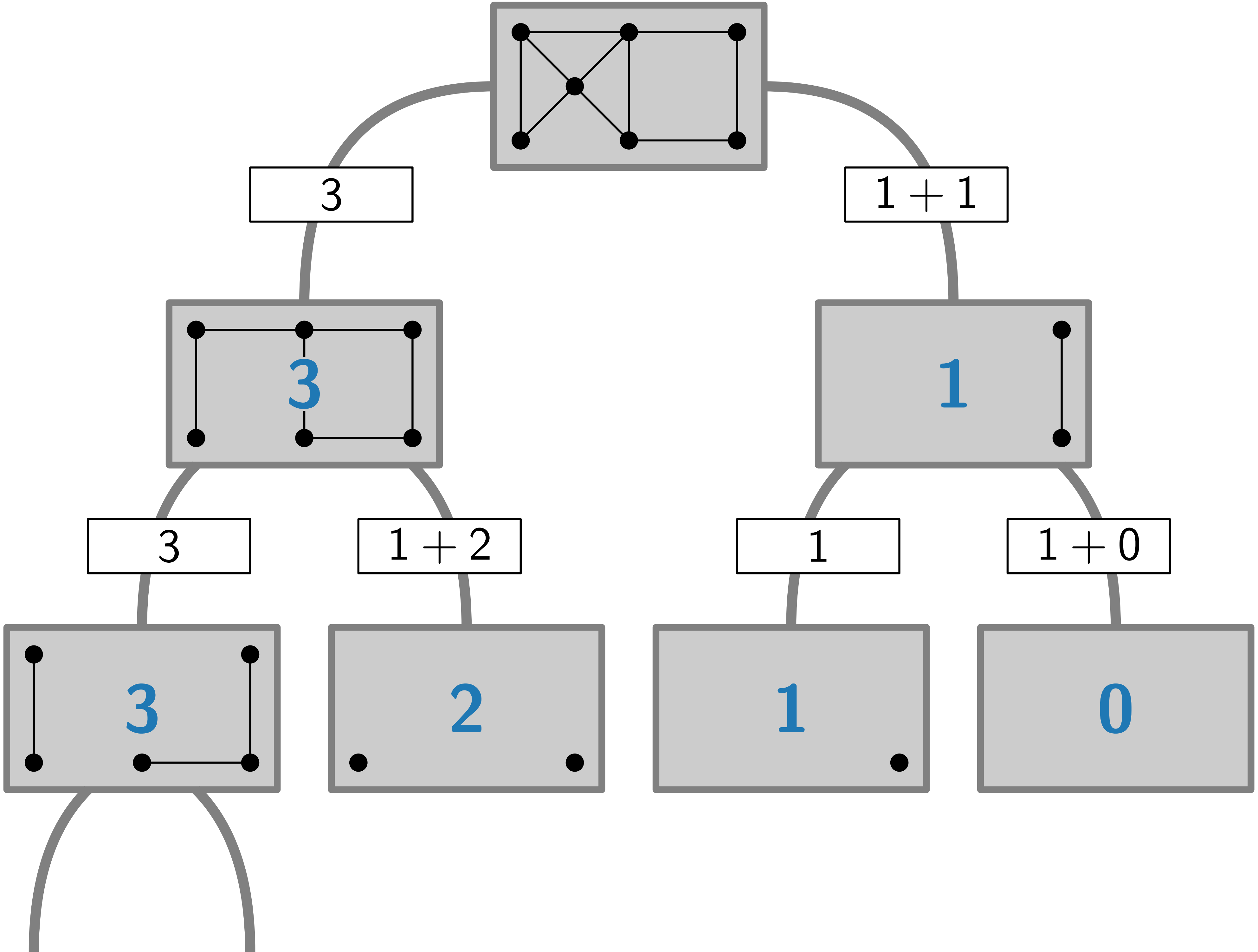
?
1 + 1
1
3
1
1 + 0
3
1
0

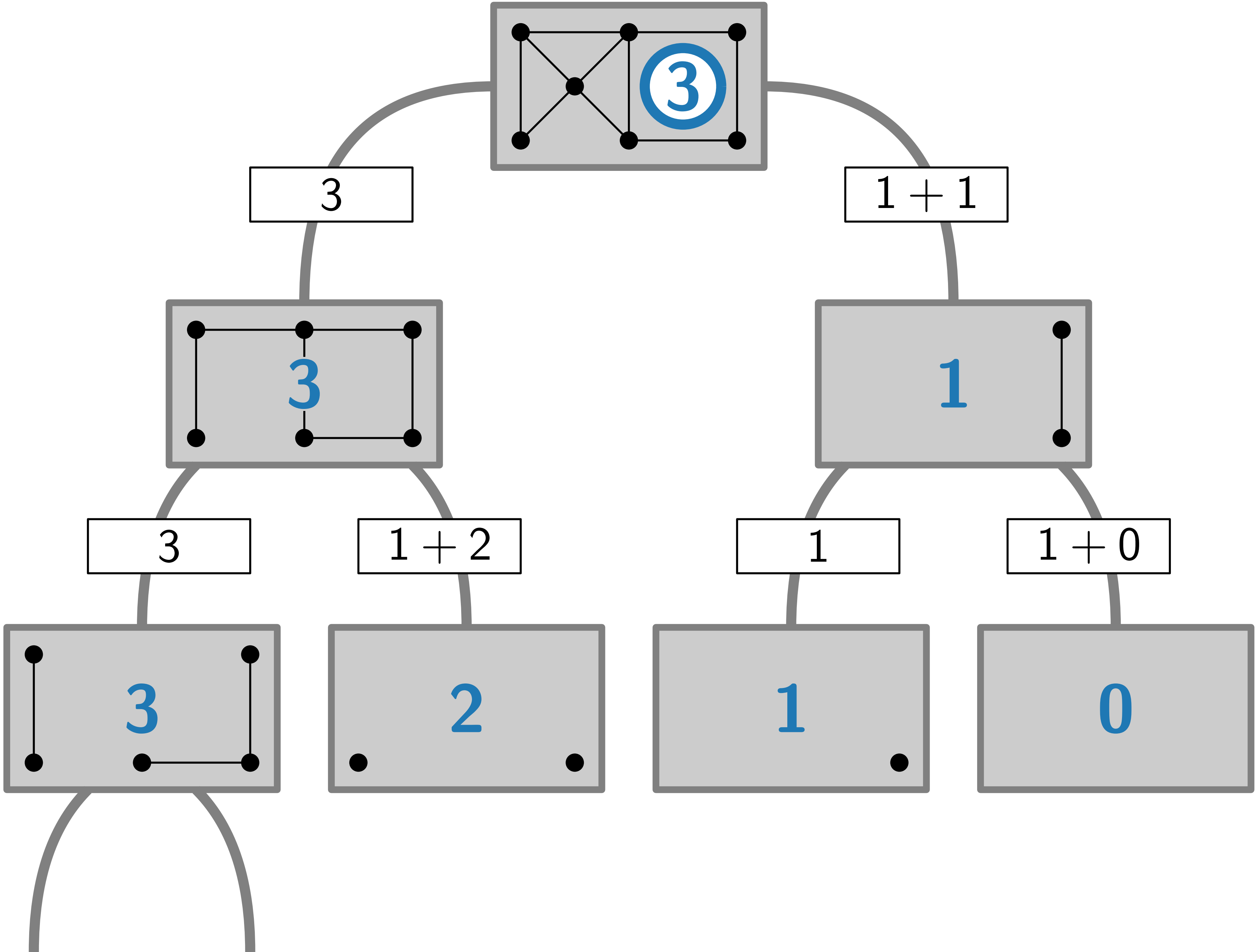

?
1 + 1
1
3
1 + ?
1
1 + 0
3
1
0


?
1 + 1
1
3
1 + ?
1
1 + 0
3
2
1
0


?
1 + 1
1
3
1 + 2
1
1 + 0
3
2
1
0


?
1 + 1
3
1
3
1 + 2
1
1 + 0
3
2
1
0


3
1 + 1
3
1
3
1 + 2
1
1 + 0
3
2
1
0

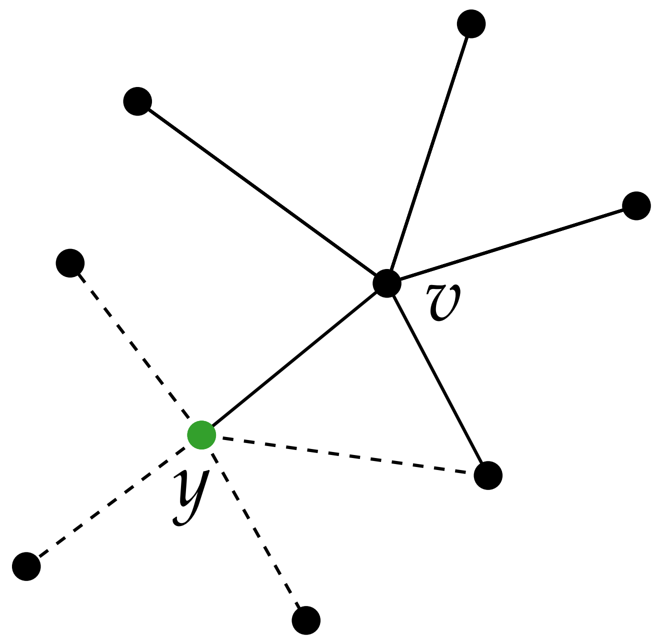

3
3
1 + 1
3
1
3
1 + 2
1
1 + 0
3
2
1
0

MIS – Smarter branching

Lemma.
Let U be a maximum independent set in G. Then for each $v \in V$:

1. $v \in U \Rightarrow N(v) \cap U = \emptyset$
2. $v \notin U \Rightarrow |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$ and no other vertex of $N[y]$ is in U.

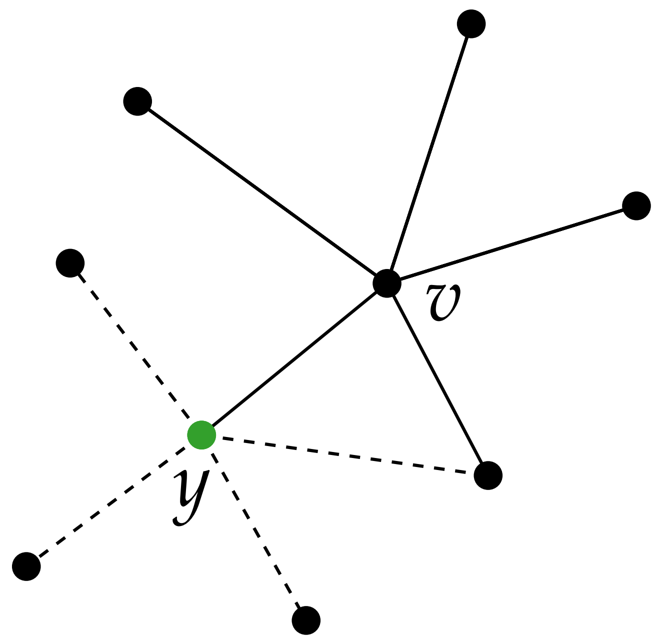


MIS – Smarter branching

Lemma.
Let U be a maximum independent set in G. Then for each $v \in V$:

1. $v \in U \Rightarrow N(v) \cap U = \emptyset$
2. $v \notin U \Rightarrow |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$ and no other vertex of $N[y]$ is in U.



Smarter MIS branching.

- For some vertex v, branch on vertices in $N[v]$.

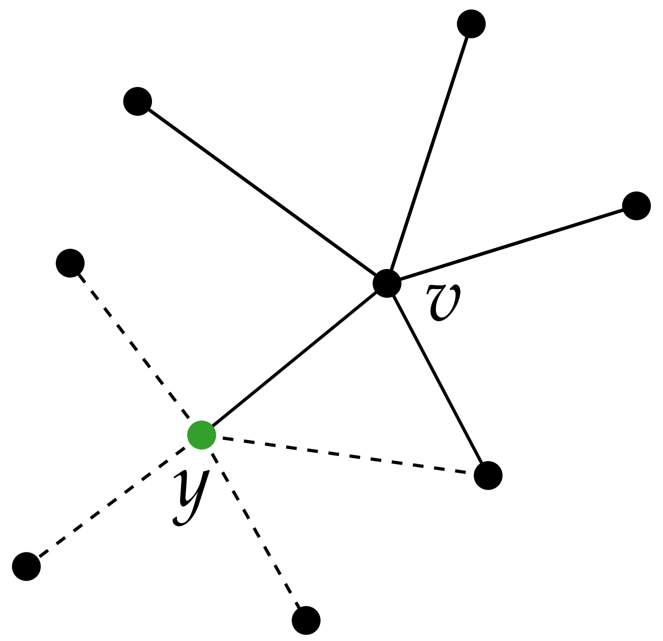
MIS – Smarter branching

Lemma.

Let U be a maximum independent set in G. Then for each $v \in V$:

1. $v \in U \Rightarrow N(v) \cap U = \emptyset$
2. $v \notin U \Rightarrow |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$ and no other vertex of $N[y]$ is in U.



Smarter MIS branching.

- For some vertex v, branch on vertices in $N[v]$.

Algorithm MIS(G)

  if $V = \emptyset$ **then**
    return 0

  $v \leftarrow$ vertex of minimum degree in $V(G)$
  return $1 + \max\{\text{MIS}(G - N[y]) \mid y \in N[v]\}$

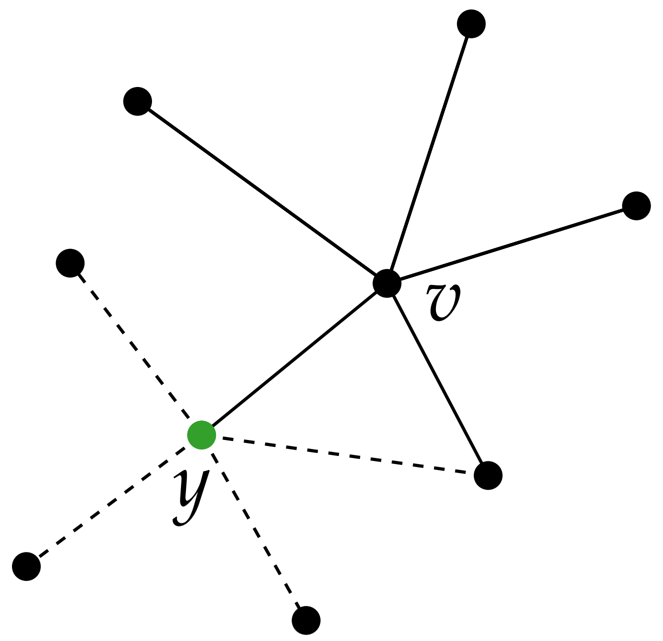
MIS – Smarter branching

Lemma.

Let U be a maximum independent set in G. Then for each $v \in V$:

1. $v \in U \Rightarrow N(v) \cap U = \emptyset$
2. $v \notin U \Rightarrow |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$ and no other vertex of $N[y]$ is in U.



Smarter MIS branching.

- For some vertex v, branch on vertices in $N[v]$.

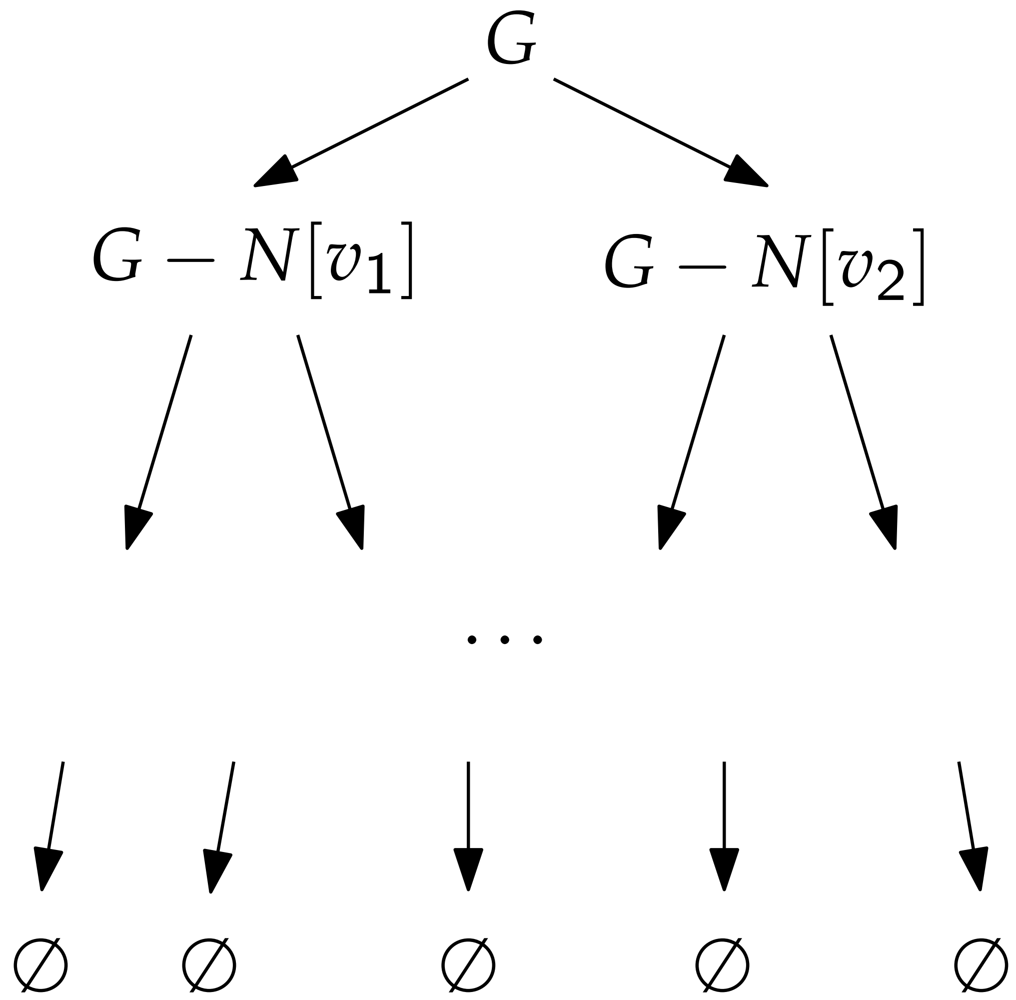
Algorithm MIS(G)

  if $V = \emptyset$ **then**
    return 0
  $v \leftarrow$ vertex of minimum degree in $V(G)$
  return $1 + \max\{\text{MIS}(G - N[y]) \mid y \in N[v]\}$

- Correctness follows from Lemma.
- We prove a runtime of $\mathcal{O}^*(3^{n/3}) = \mathcal{O}^*(1.4423^n)$.

MIS – Branching analysis

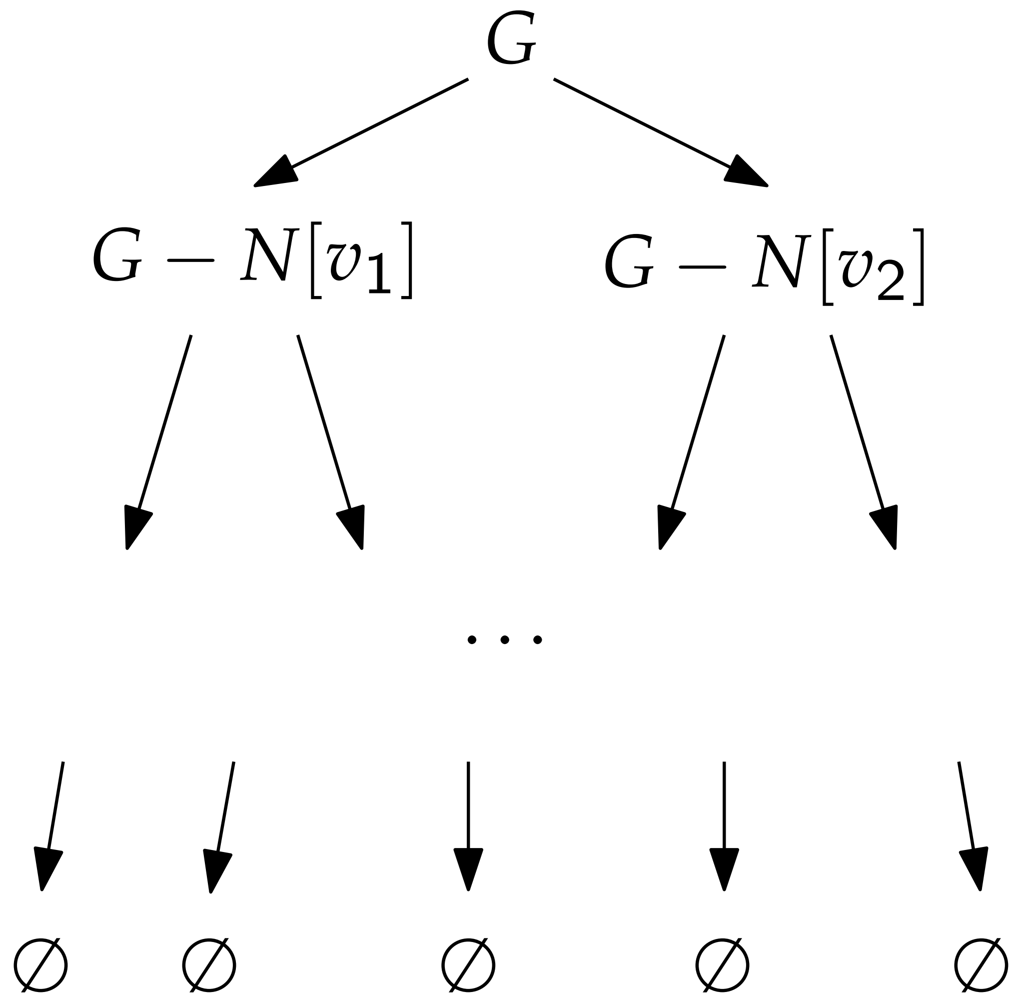
Execution corresponds to a **search tree** whose vertices are labeled with the input of the respective recursive call.



MIS – Branching analysis

Execution corresponds to a **search tree** whose vertices are labeled with the input of the respective recursive call.

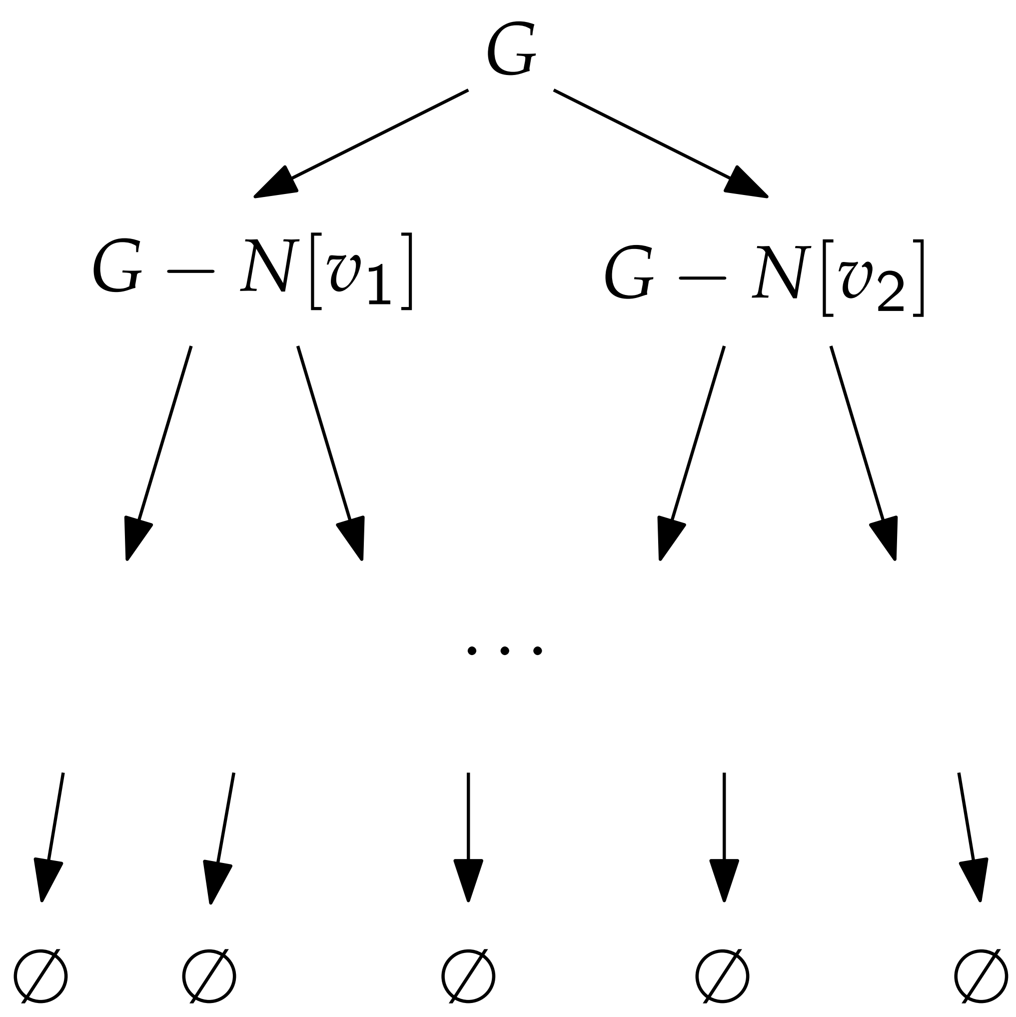
- Let $B(n)$ be the maximum number of leaves of a search tree for a graph with n vertices.



MIS – Branching analysis

Execution corresponds to a **search tree** whose vertices are labeled with the input of the respective recursive call.

- Let $B(n)$ be the maximum number of leaves of a search tree for a graph with n vertices.
- Search-tree has height $\leq n$.



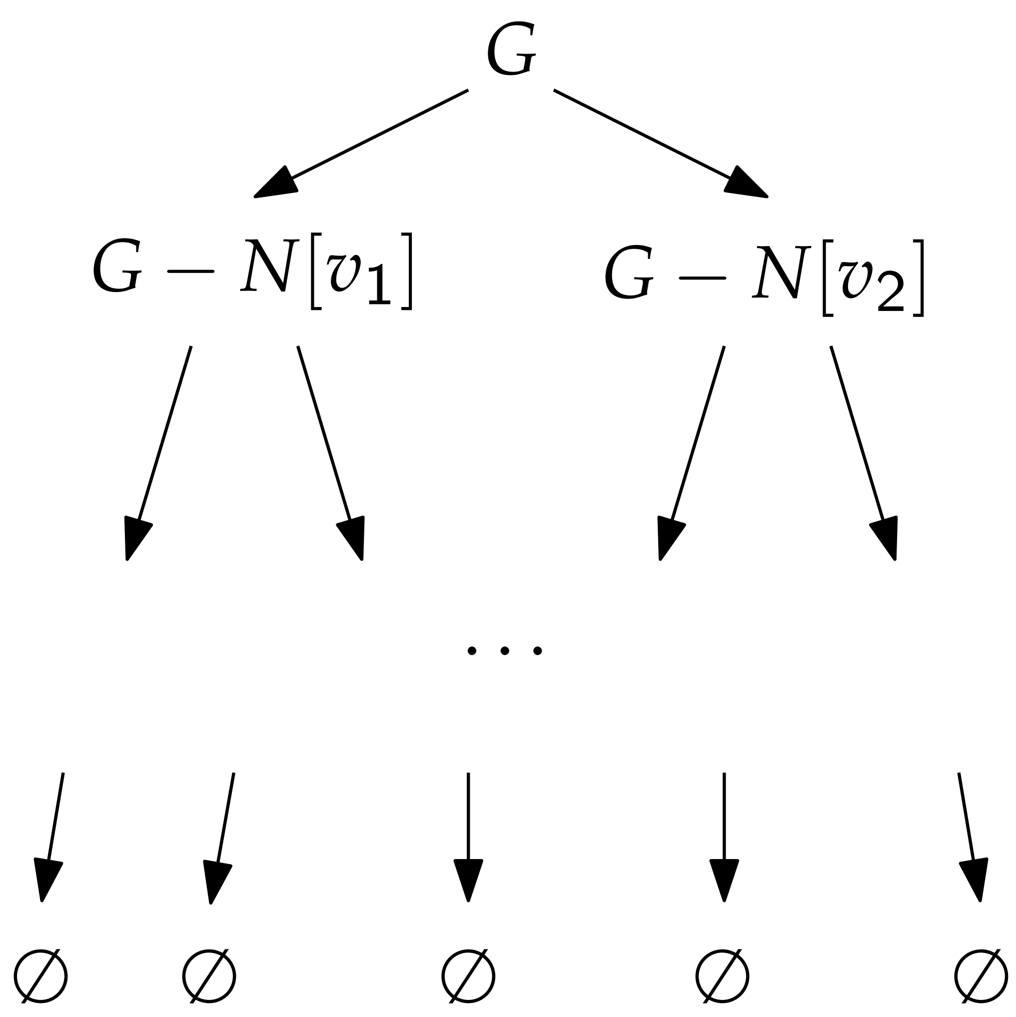
MIS – Branching analysis

Execution corresponds to a **search tree** whose vertices are labeled with the input of the respective recursive call.

- Let $B(n)$ be the maximum number of leaves of a search tree for a graph with n vertices.
- Search-tree has height $\leq n$.

$\rightsquigarrow$ The algorithm's runtime is

$$T(n) \in O^*(nB(n)) = O^*(B(n)).$$



MIS – Branching analysis

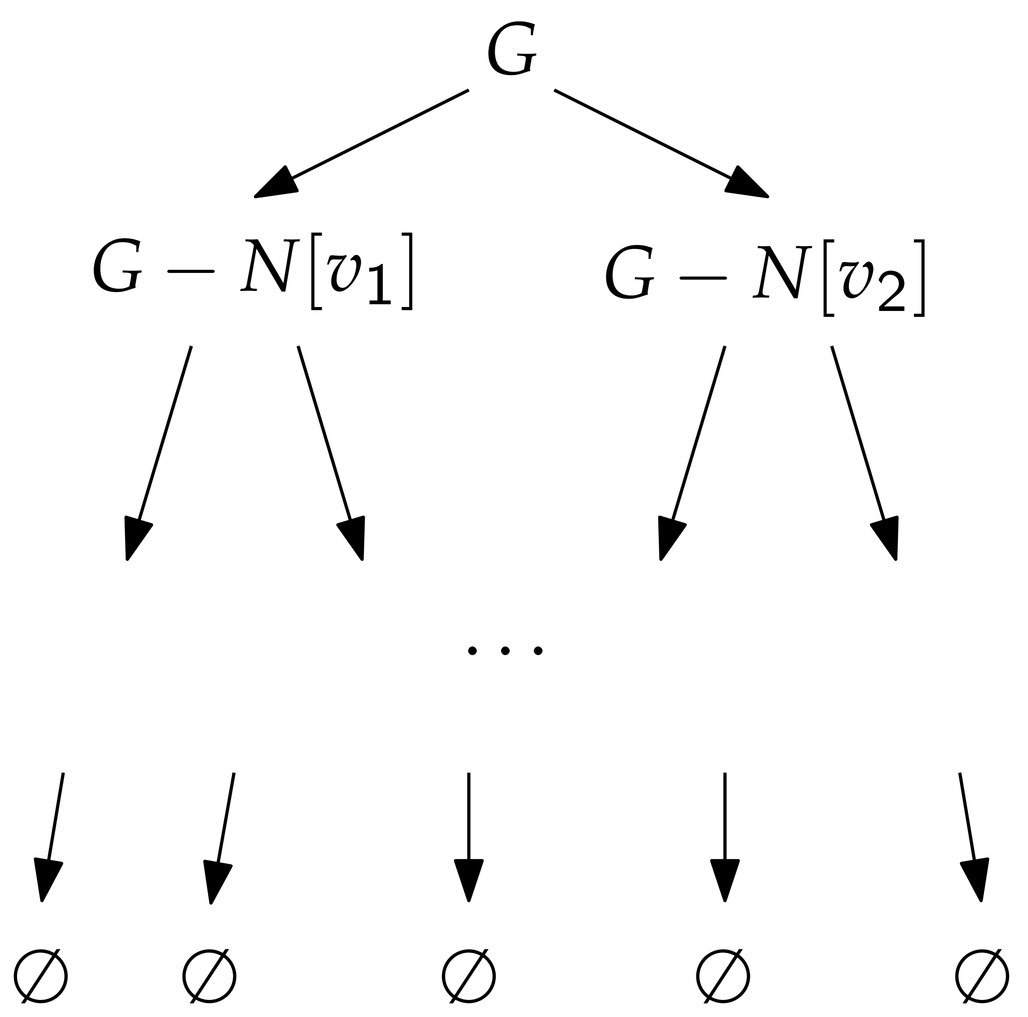
Execution corresponds to a **search tree** whose vertices are labeled with the input of the respective recursive call.

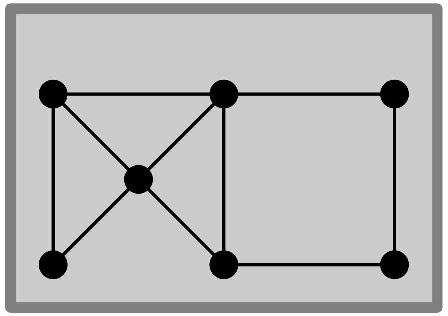
- Let $B(n)$ be the maximum number of leaves of a search tree for a graph with n vertices.
- Search-tree has height $\leq n$.

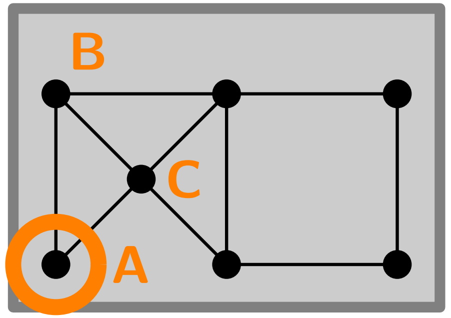
$\rightsquigarrow$ The algorithm's runtime is

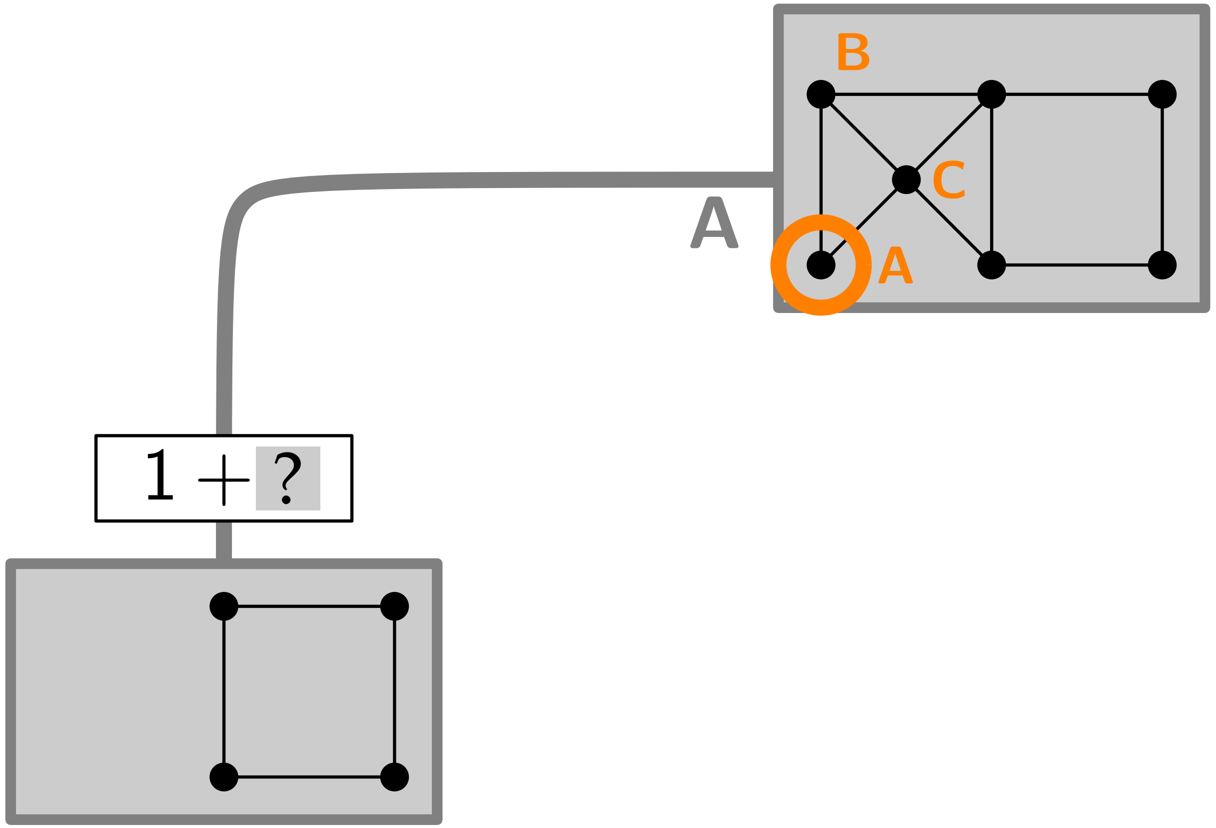
$$T(n) \in O^*(nB(n)) = O^*(B(n)).$$

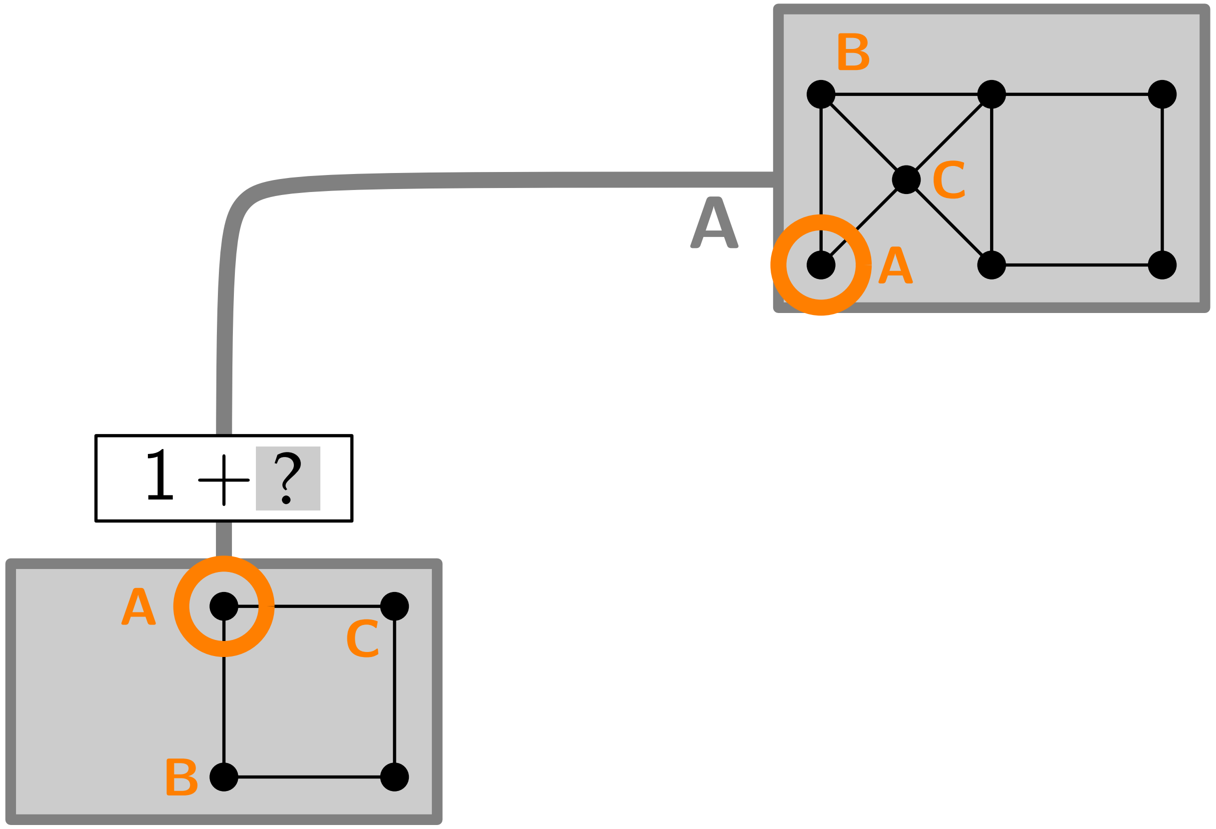
- Let's consider an example run.

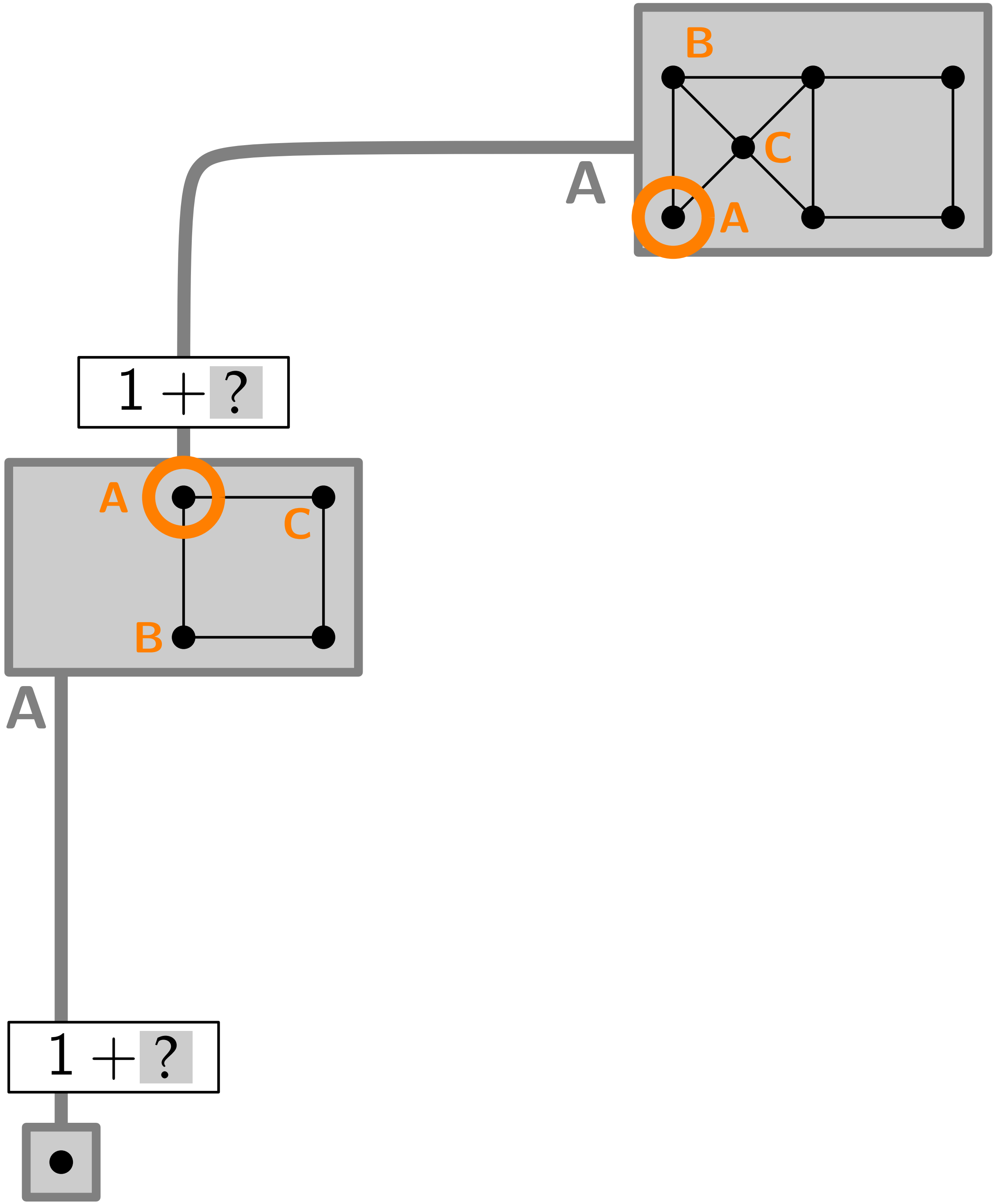


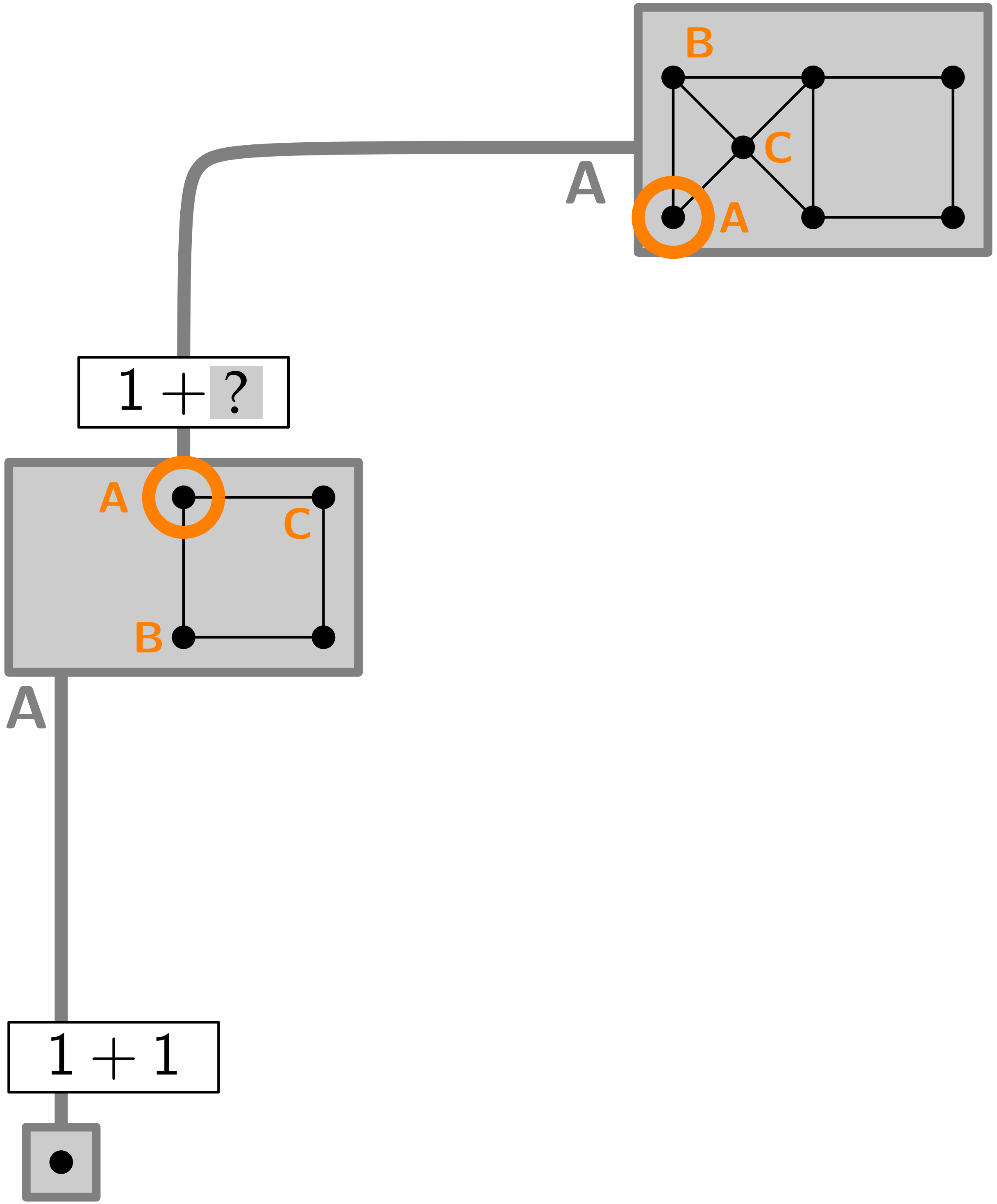


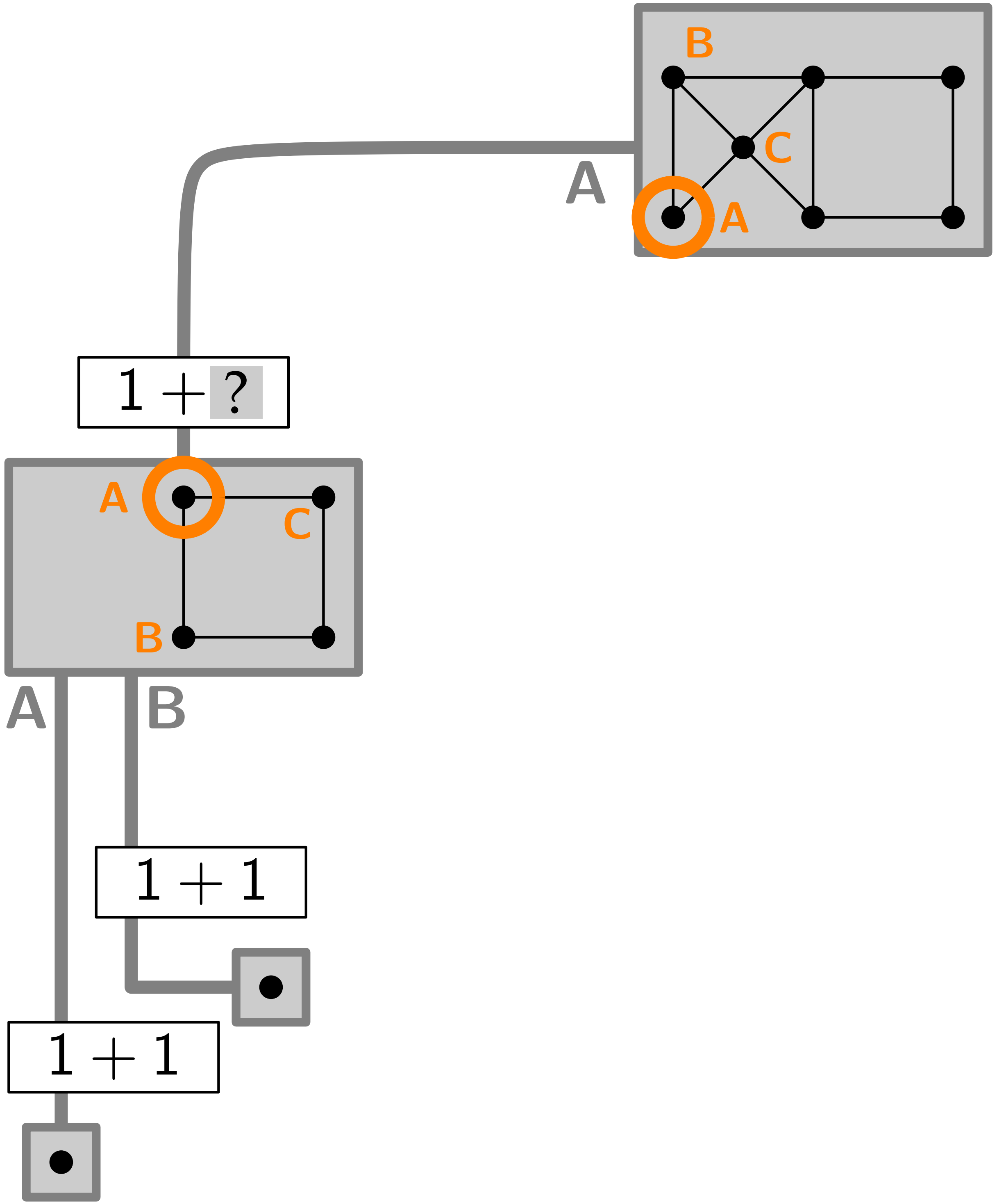

B
C
A

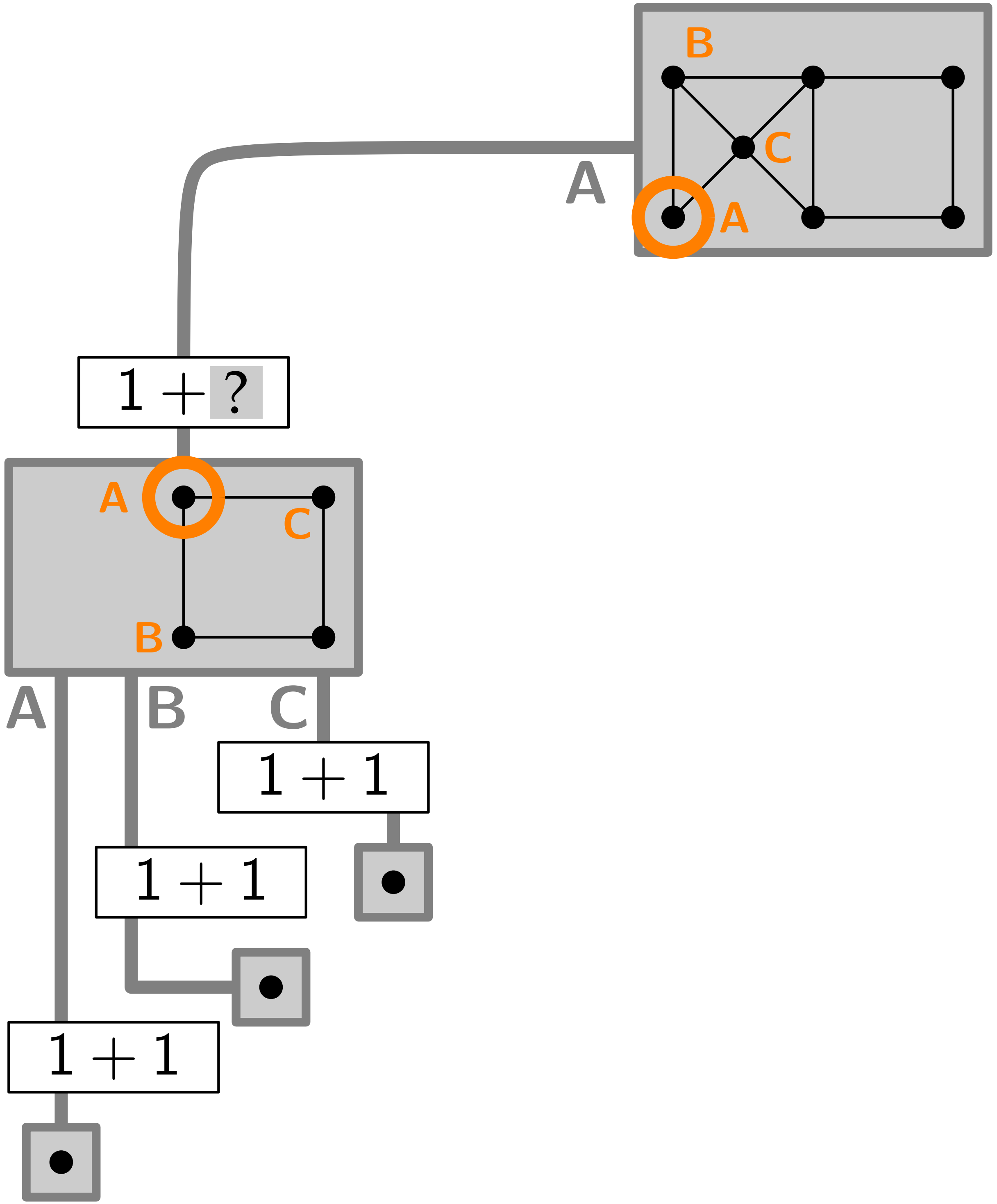

B
C
A
A
1 + ?


B
C
A
A
$1 + ?$
A
C
B

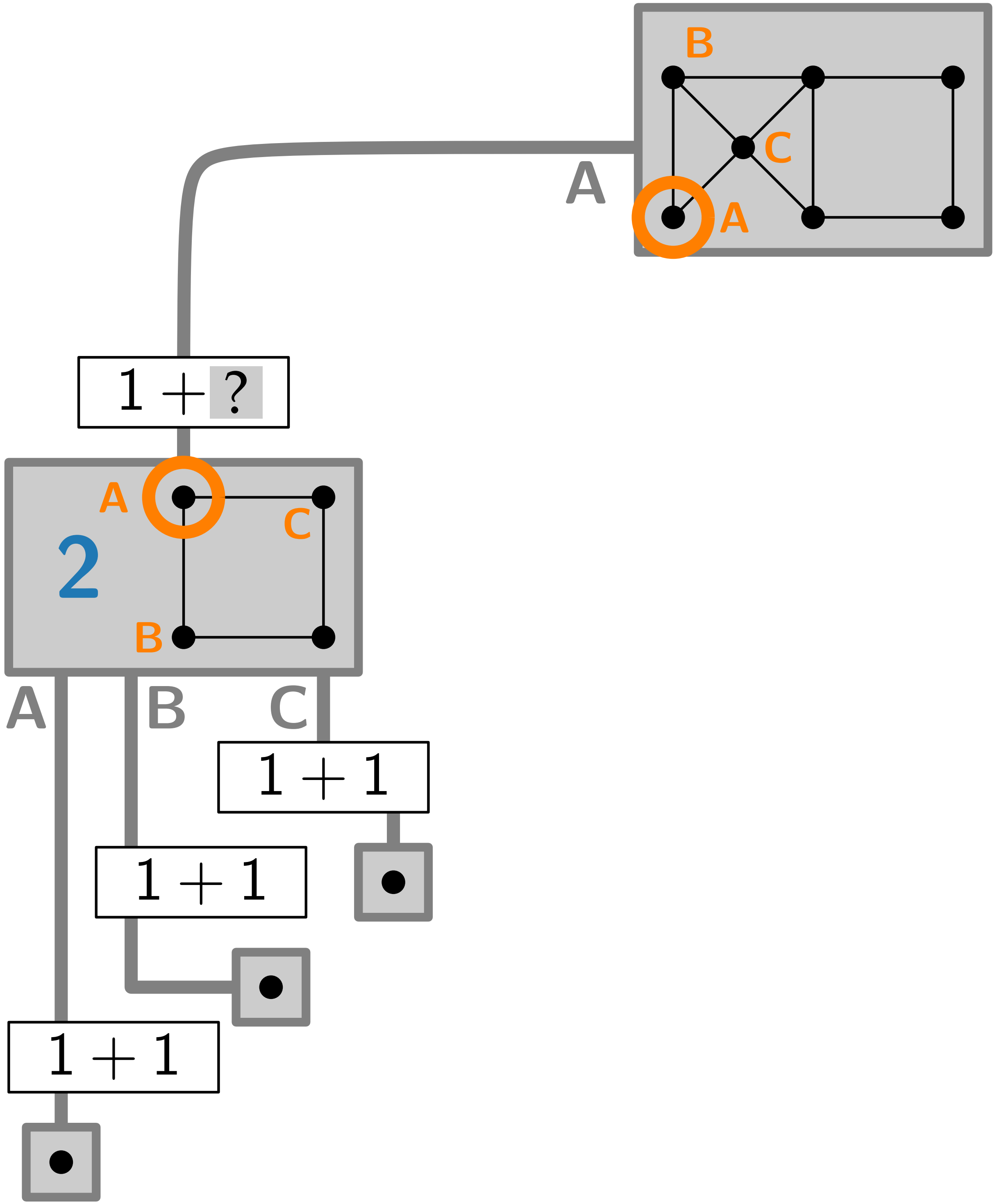

B
C
A
A
1 + ?
A
C
B
A
1 + ?

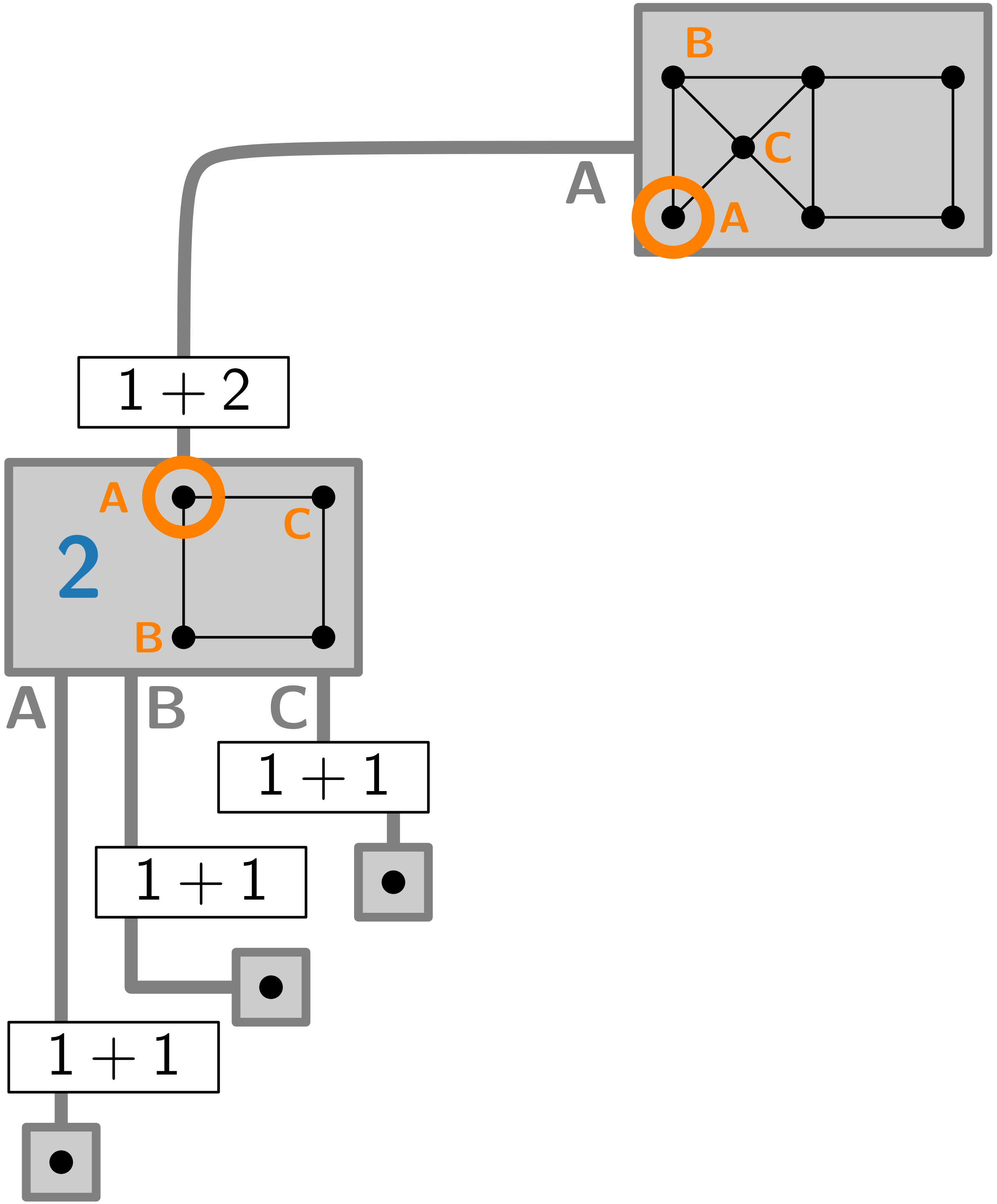

B
C
A
A
1 + ?
A
C
B
A
1 + 1

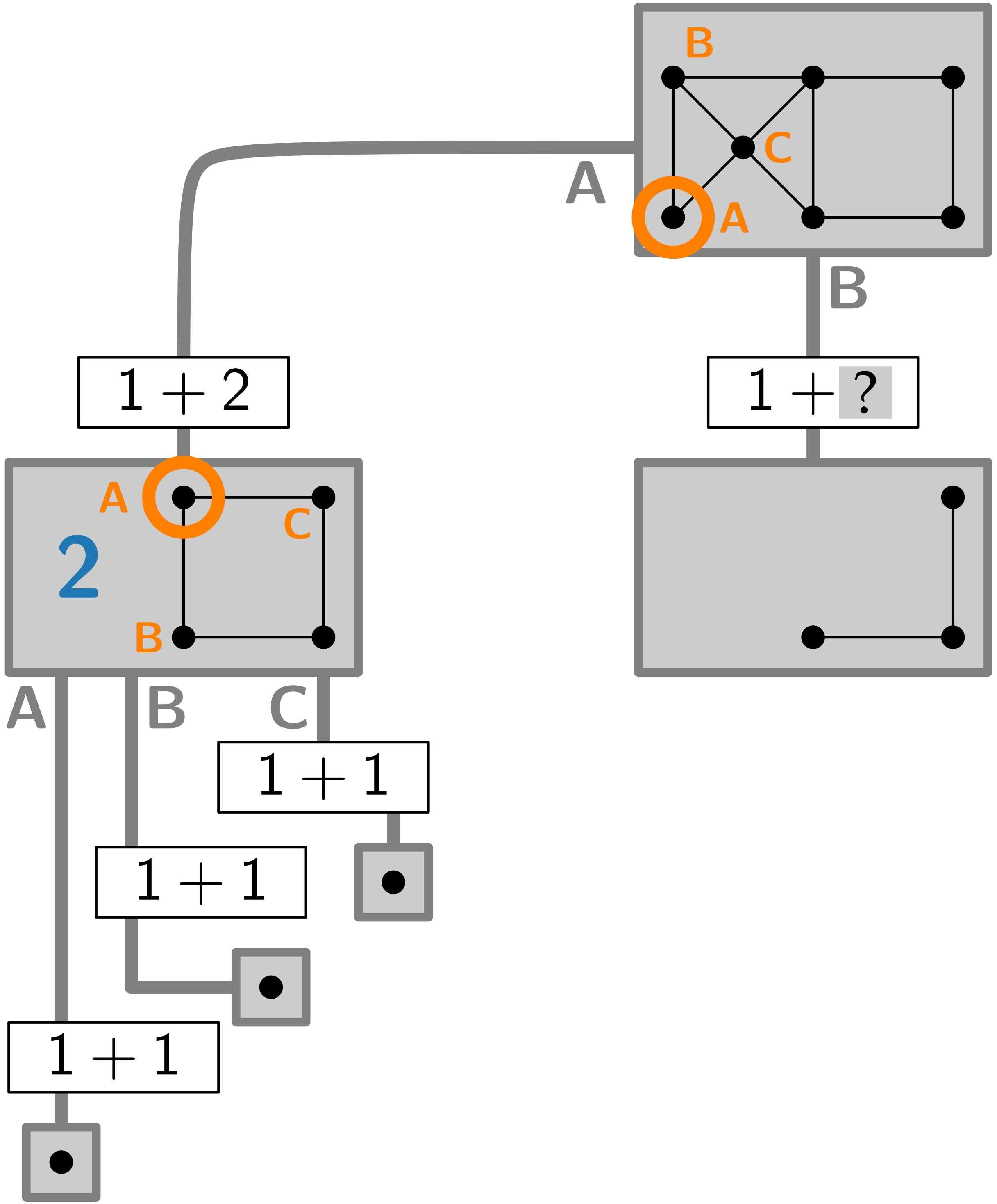

B
C
A
A
1 + ?
A
C
B
A
B
1 + 1
1 + 1

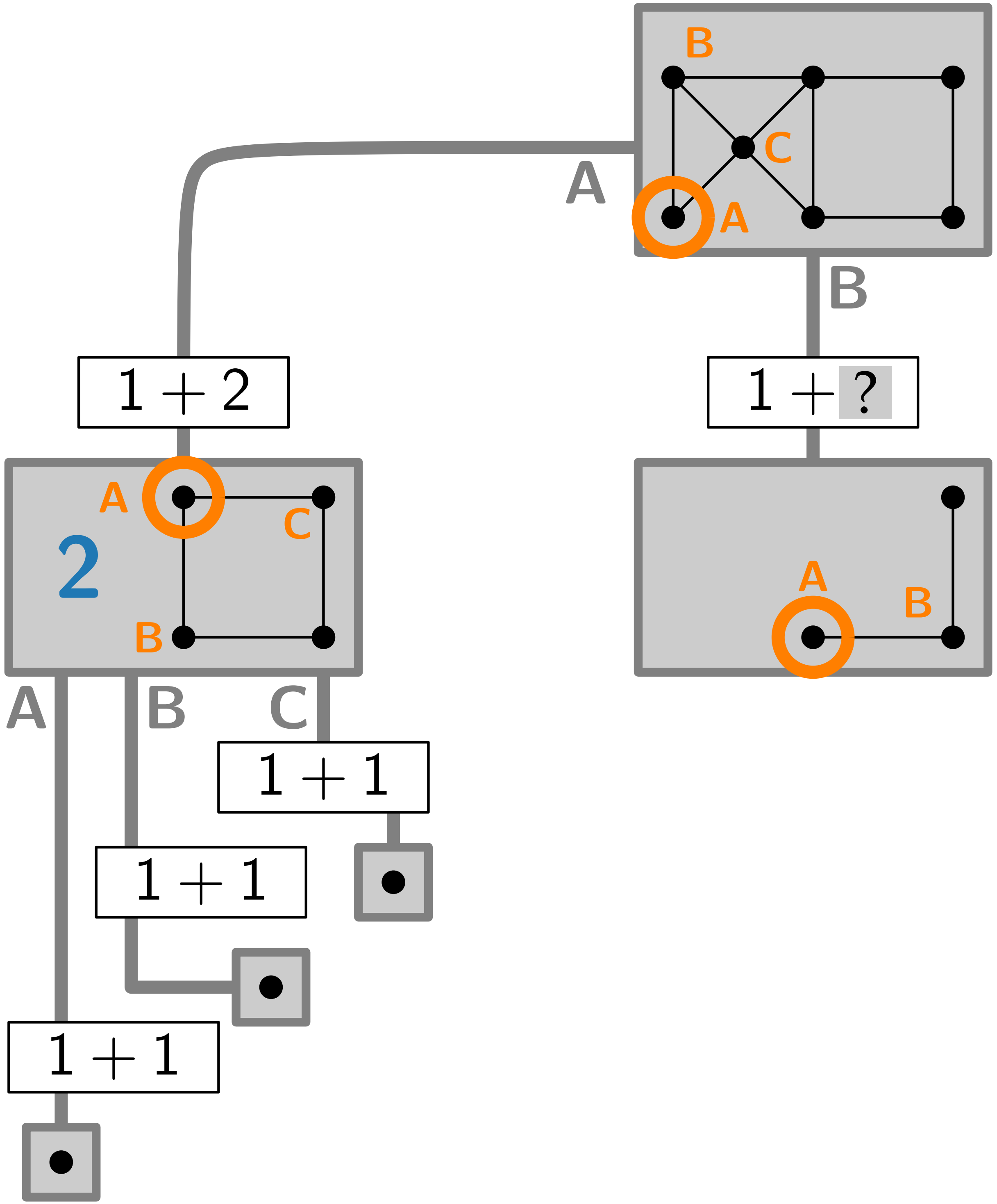


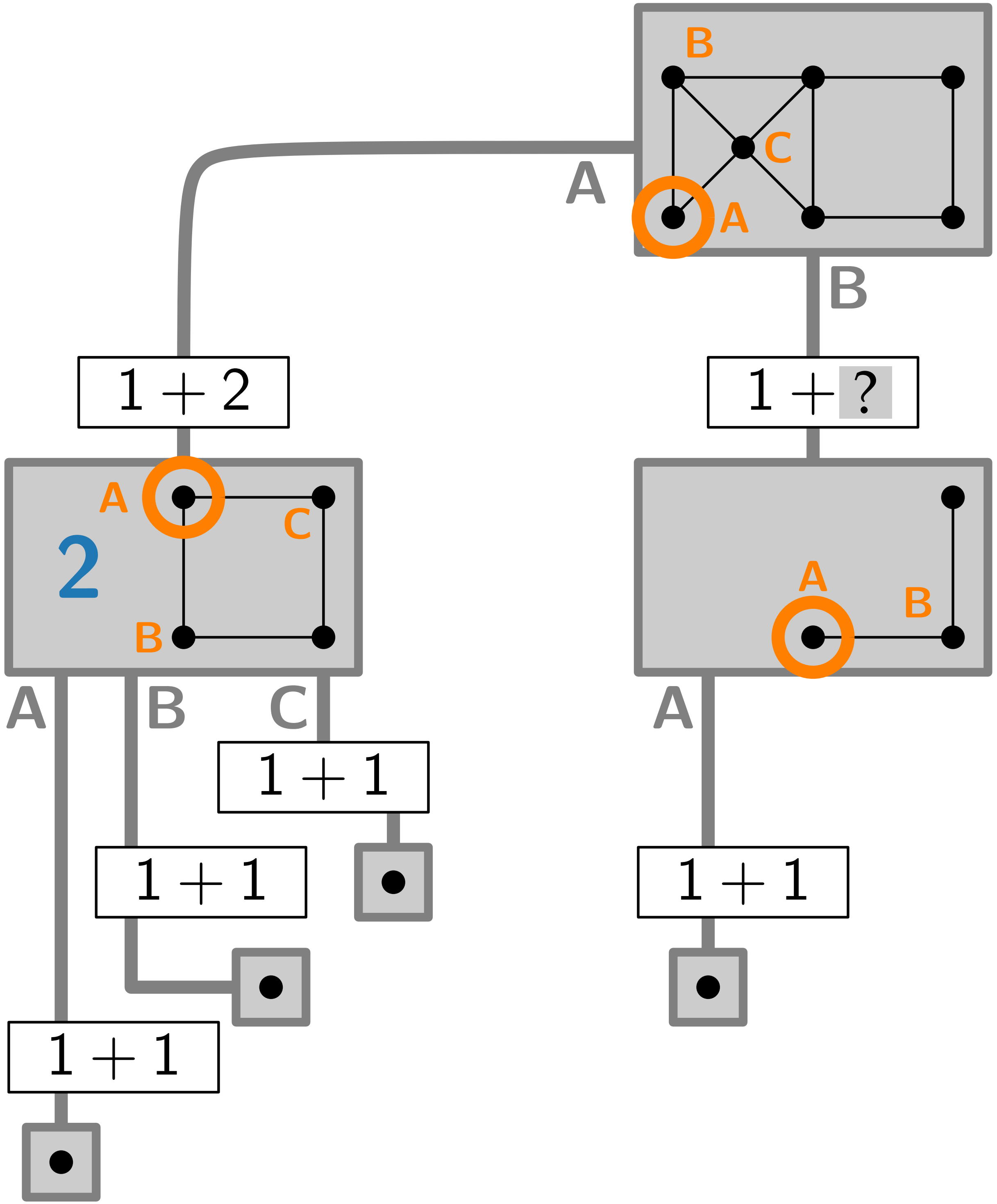
B
C
A
A
1 + ?
A
C
B
A
B
C
1 + 1
1 + 1
1 + 1

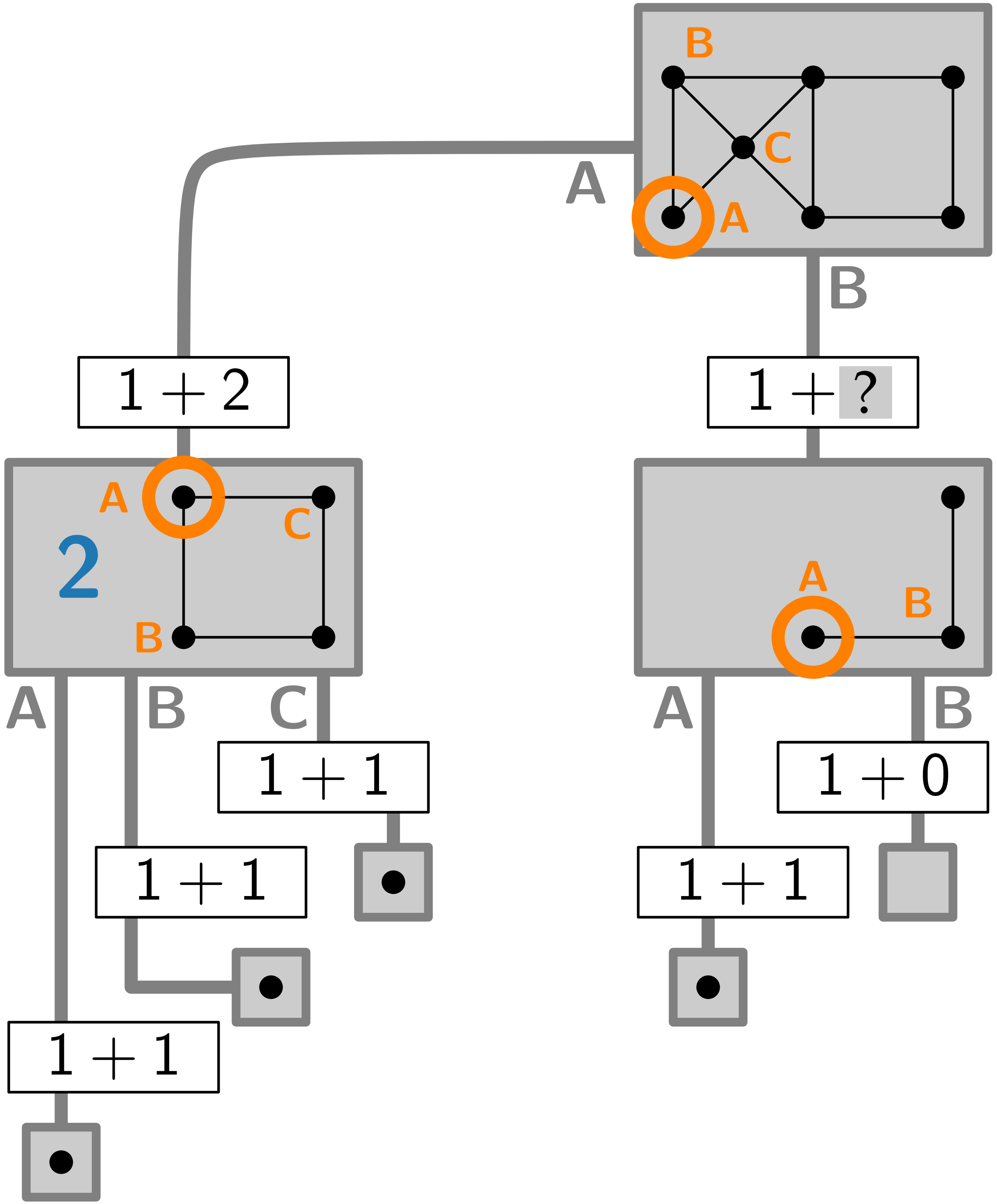

B
C
A
A
1 + ?
A
C
2
B
A
B
C
1 + 1
1 + 1
1 + 1

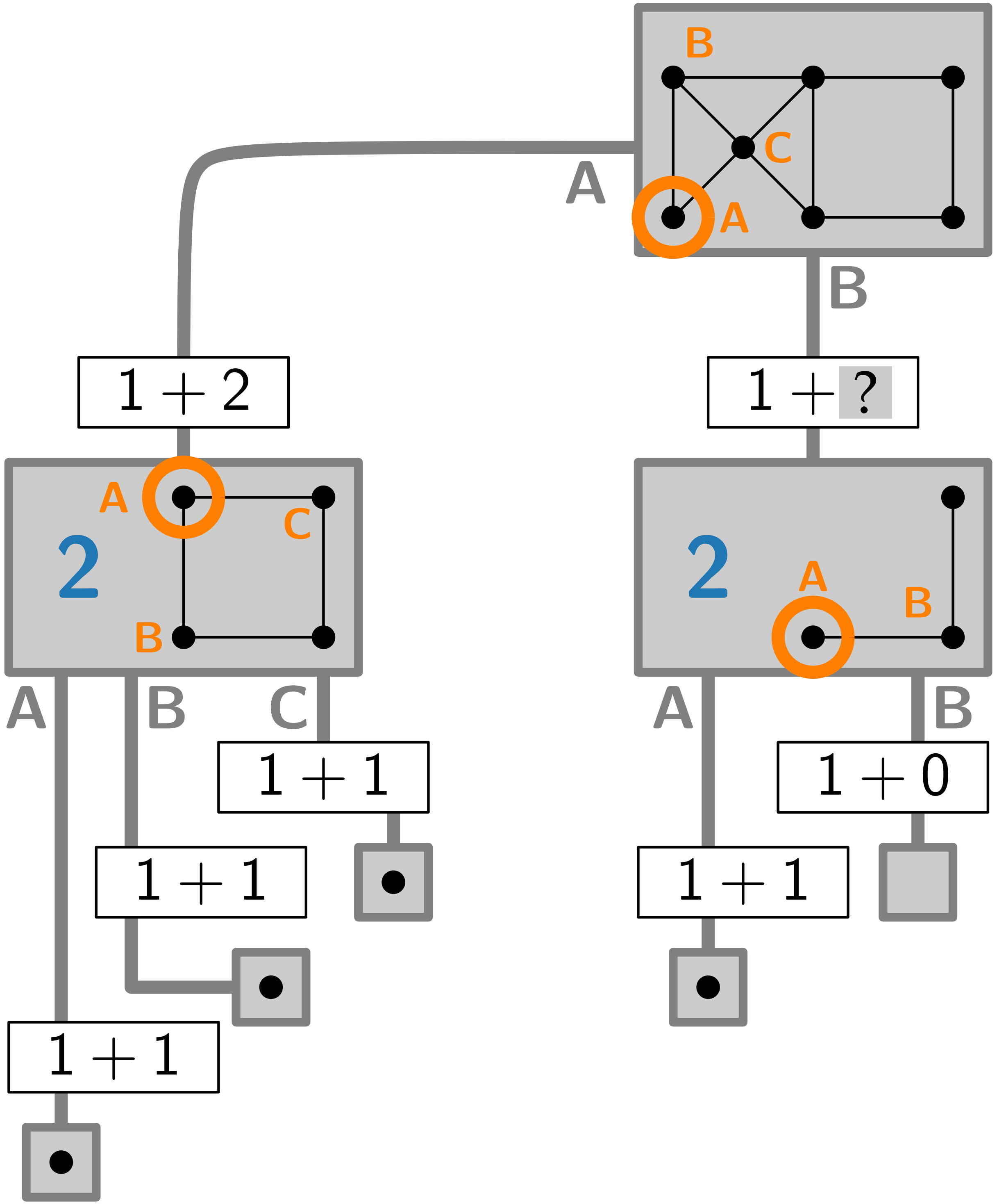

B
C
A
A
1 + 2
A
C
2
B
A
B
C
1 + 1
1 + 1
1 + 1

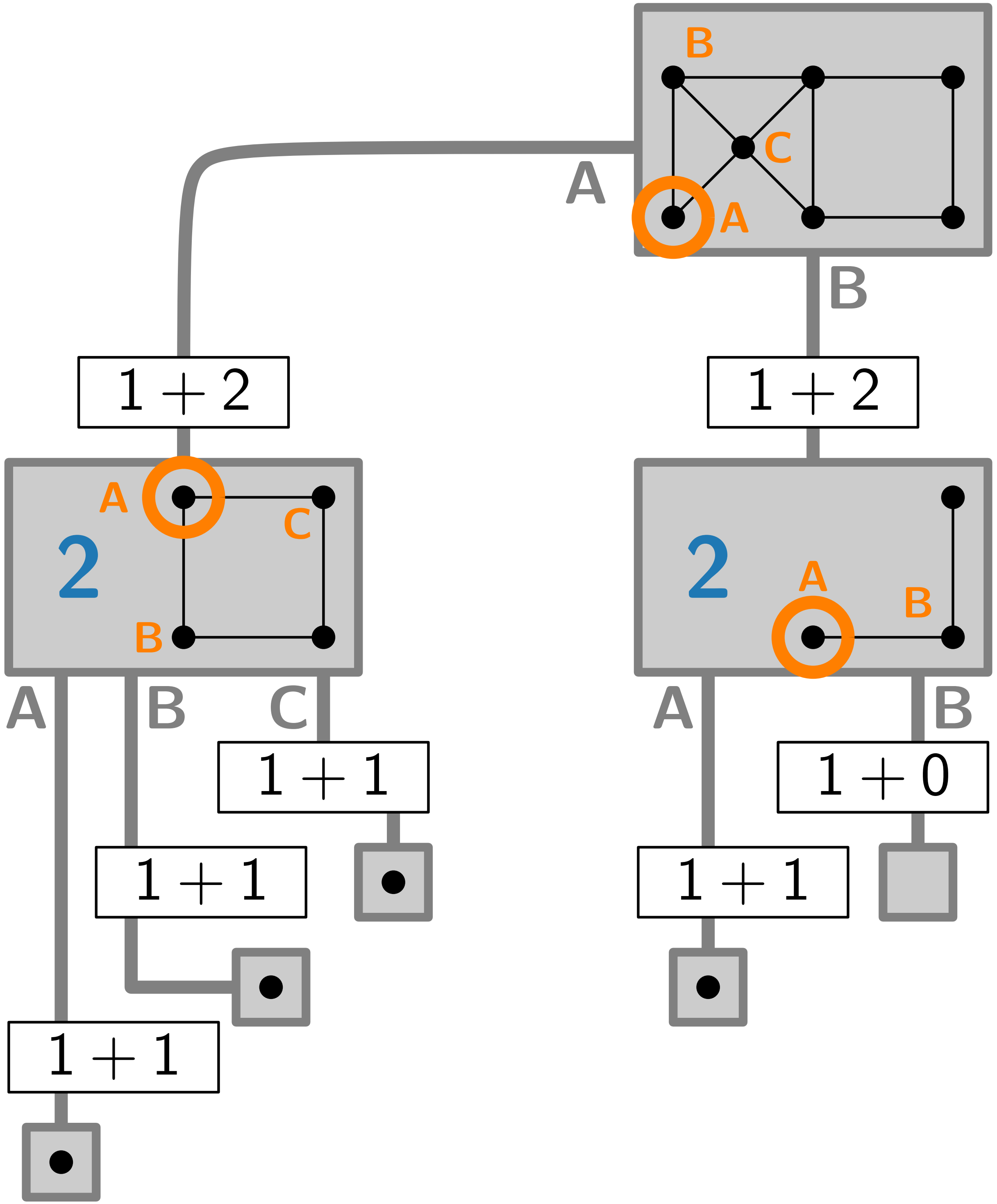

A
B
C
A
B
1 + 2
1 + ?
A
C
2
B
A
B
C
1 + 1
1 + 1
1 + 1


B
C
A
A
B
1 + 2
1 + ?
A
C
2
B
A
B
A
B
C
1 + 1
1 + 1
1 + 1

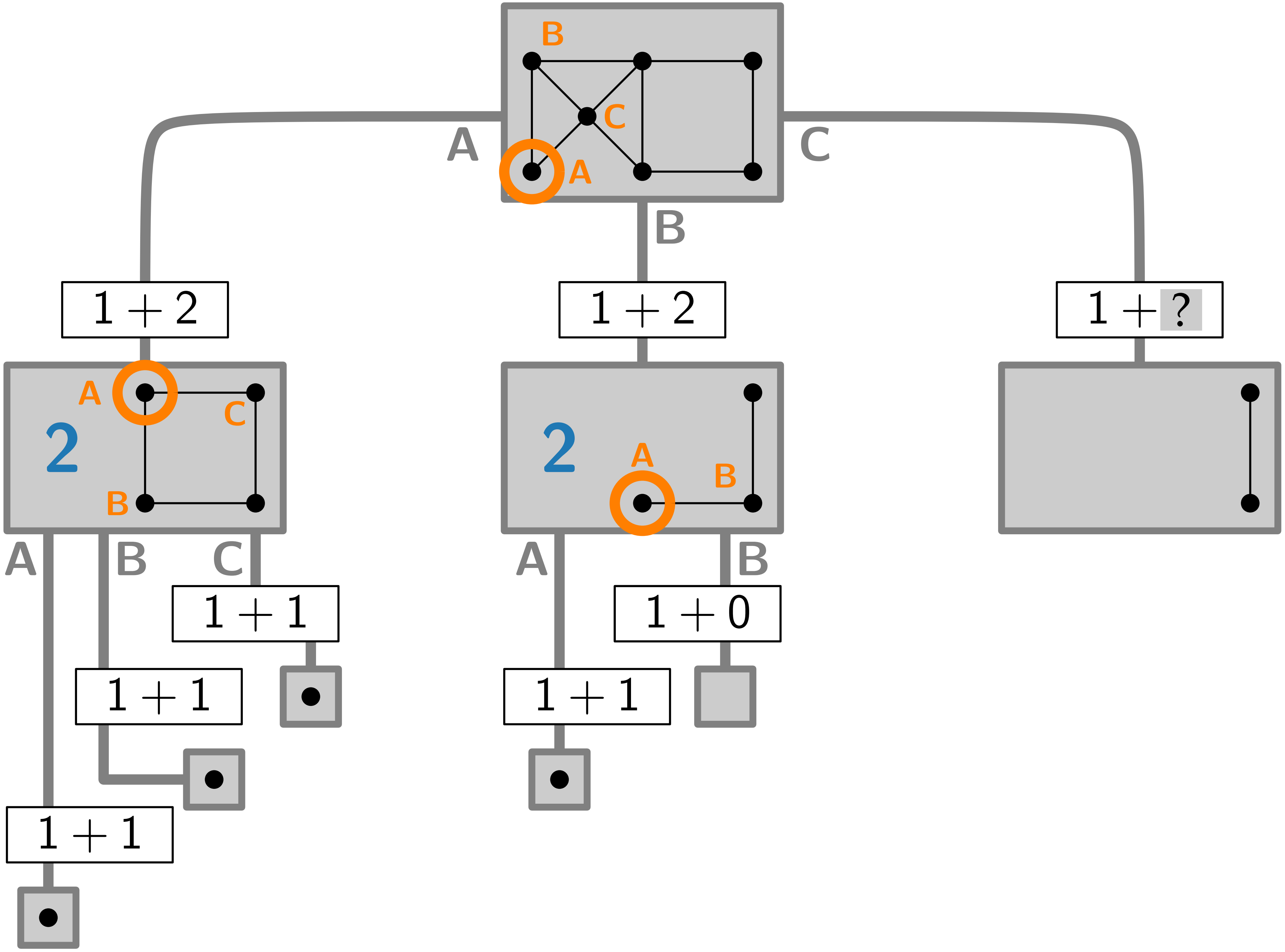

B
C
A
A
B
1 + 2
1 + ?
A
C
2
B
A
B
A
B
C
A
1 + 1
1 + 1
1 + 1
1 + 1

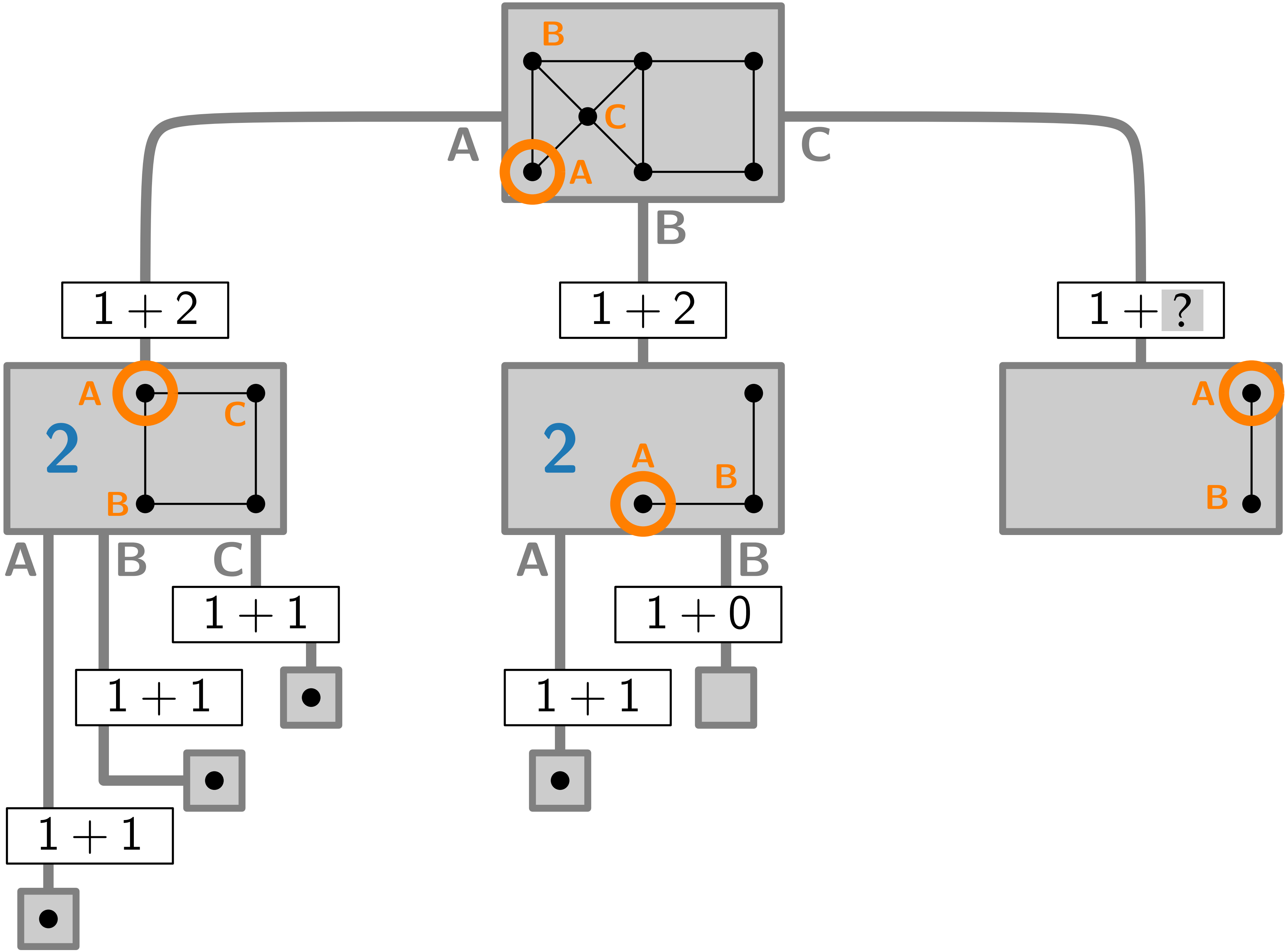

B
C
A
A
B
1 + 2
1 + ?
A
C
2
B
A
B
A
B
C
A
B
1 + 1
1 + 0
1 + 1
1 + 1
1 + 1


B
C
A
A
B
1 + 2
1 + ?
A
C
2
B
2
A
B
A
B
C
A
B
1 + 1
1 + 0
1 + 1
1 + 1
1 + 1

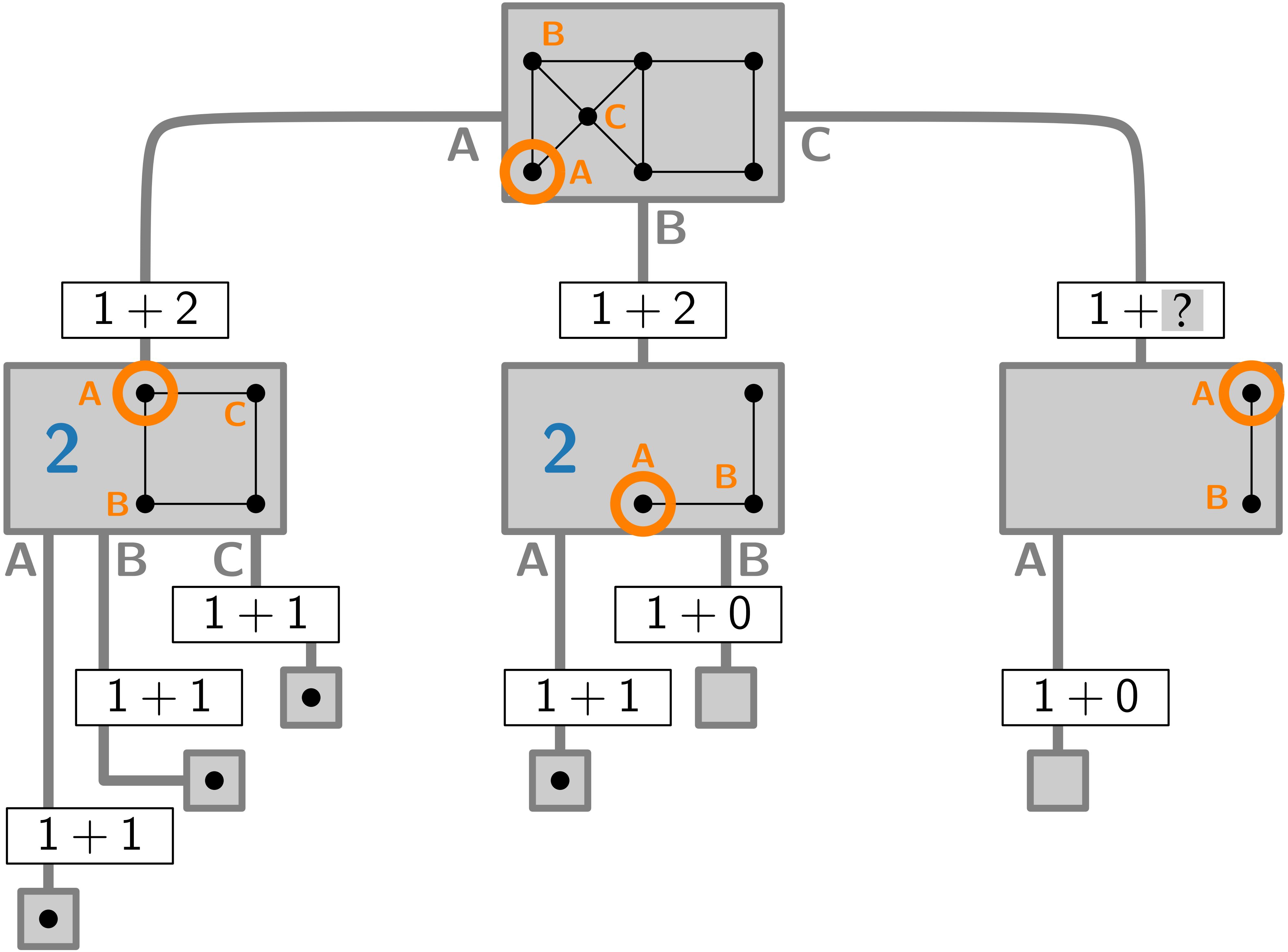


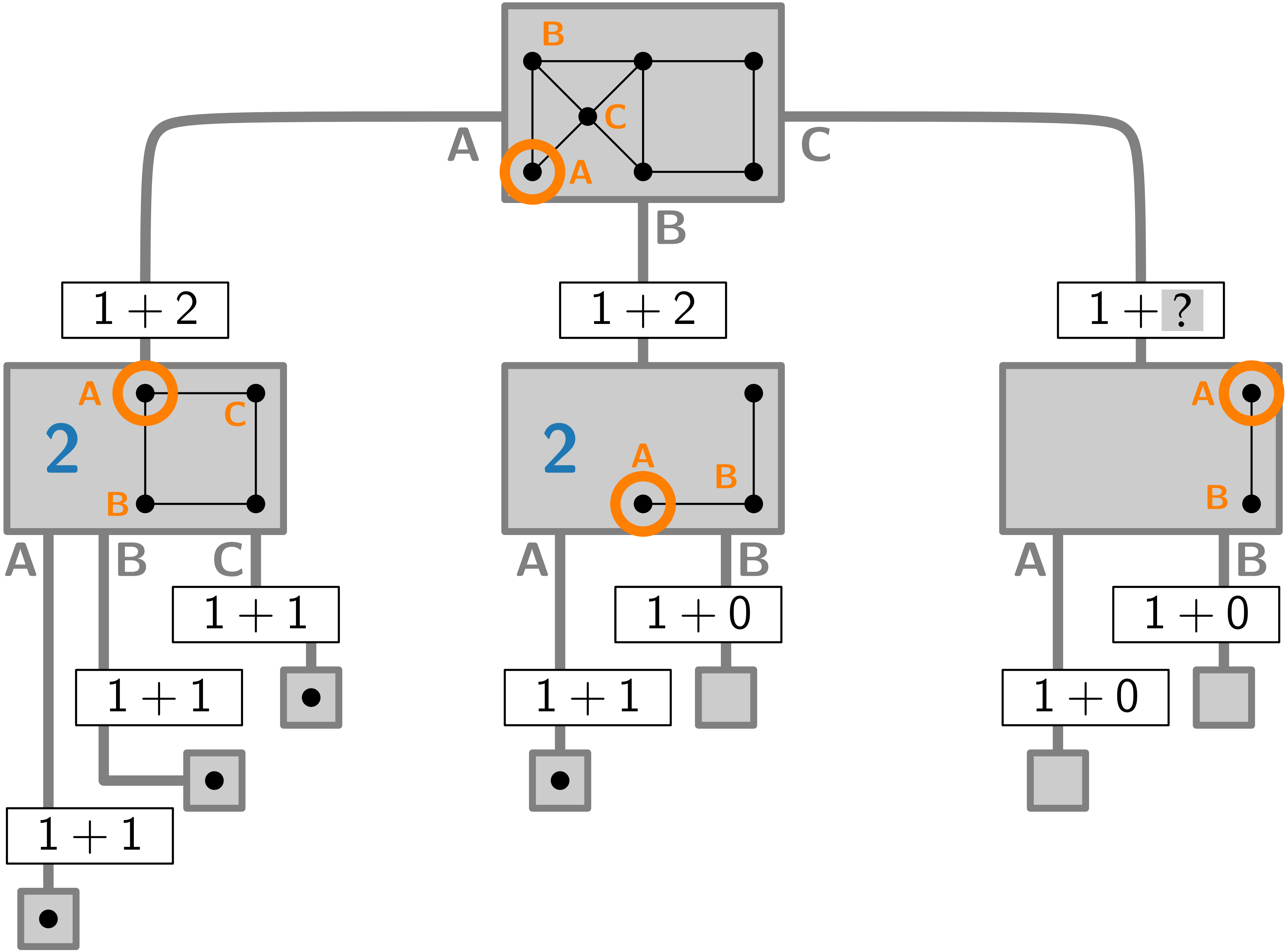
B
C
A
A
B
1 + 2
1 + 2
A
2
C
B
2
A
B
A
B
C
A
B
1 + 1
1 + 0
1 + 1
1 + 1
1 + 1

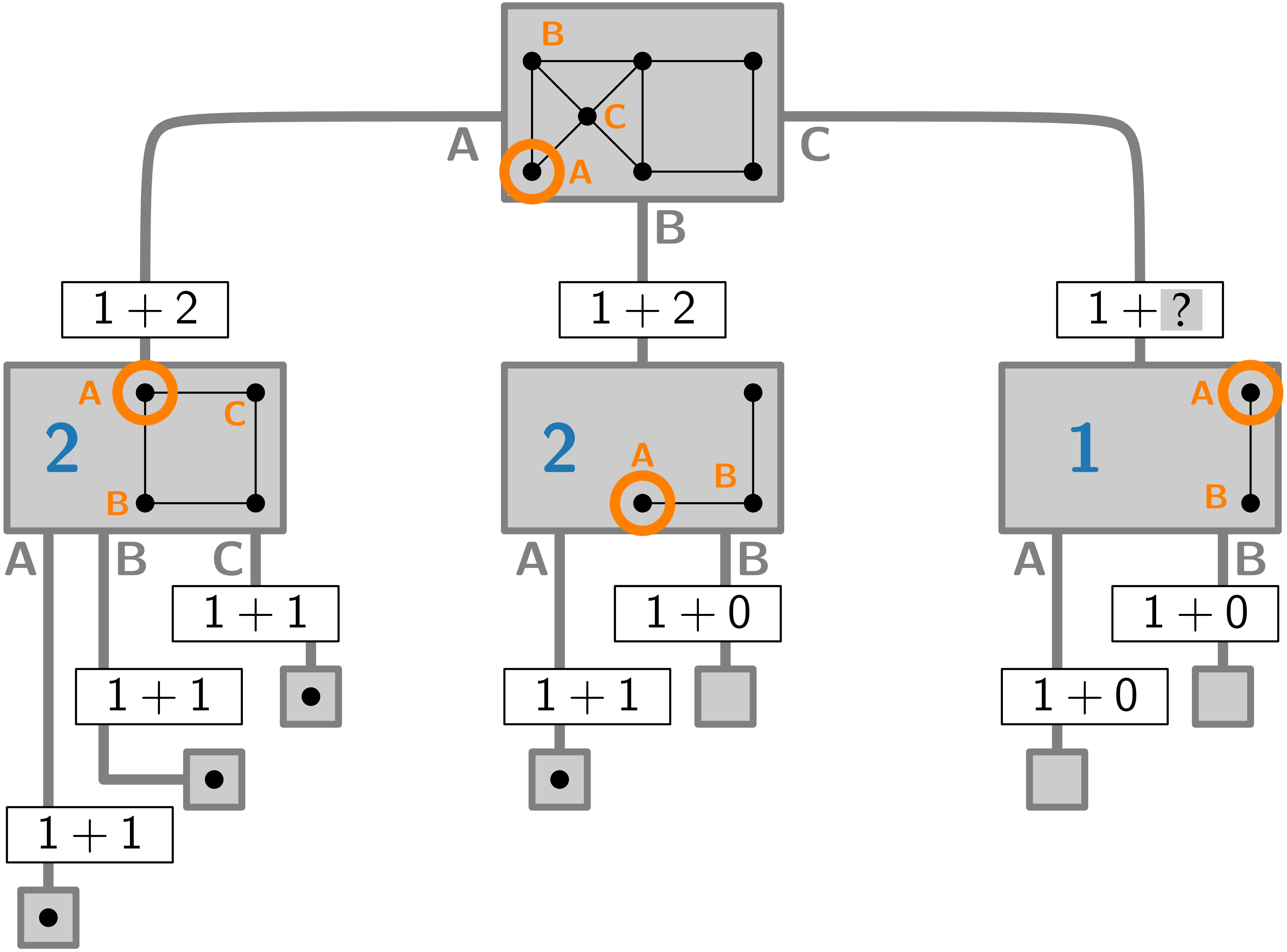

B
C
A
A
B
C
1 + 2
1 + 2
1 + ?
A
C
2
B
2
A
B
A
B
C
A
B
1 + 1
1 + 0
1 + 1
1 + 1
1 + 1

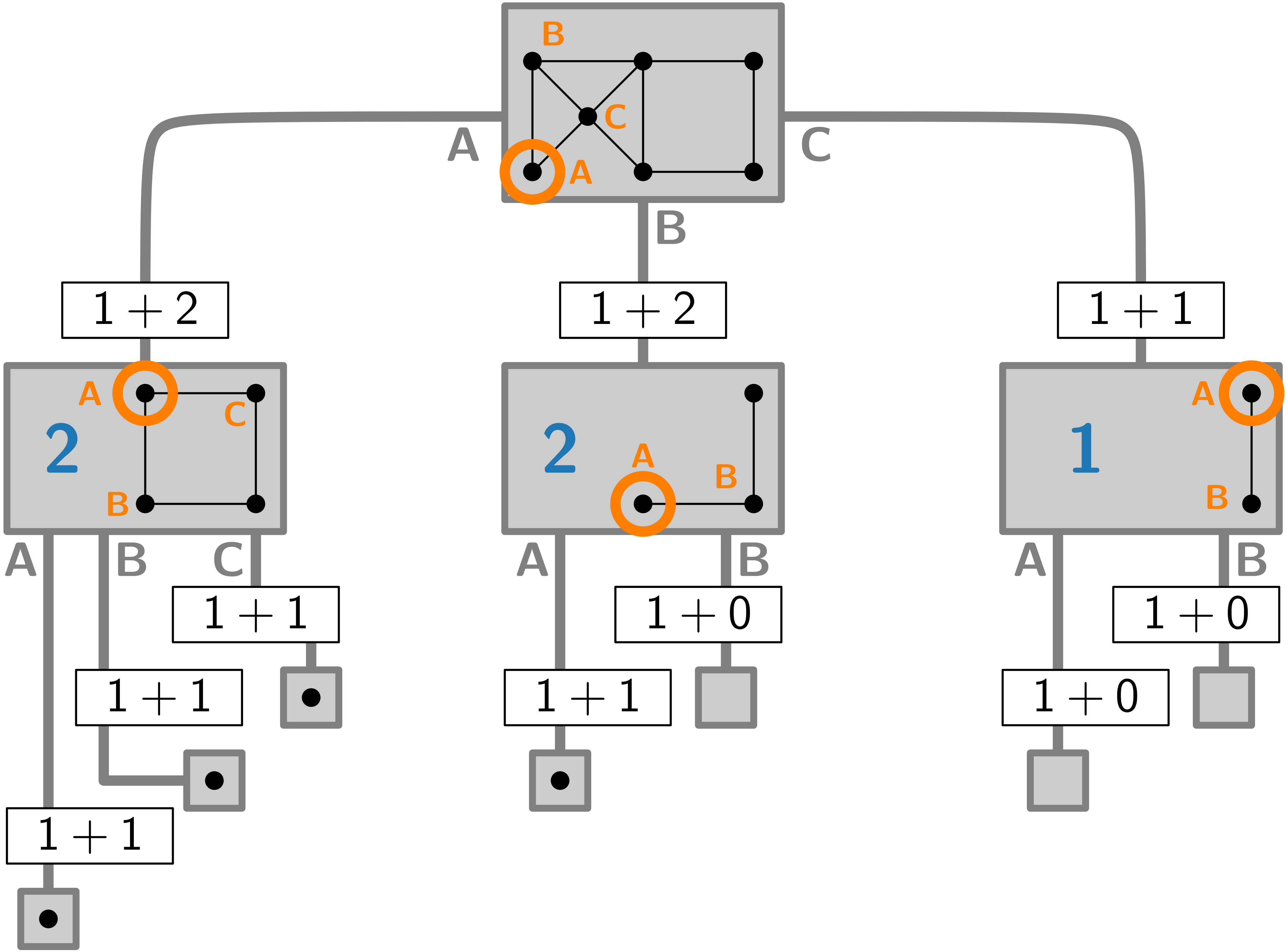


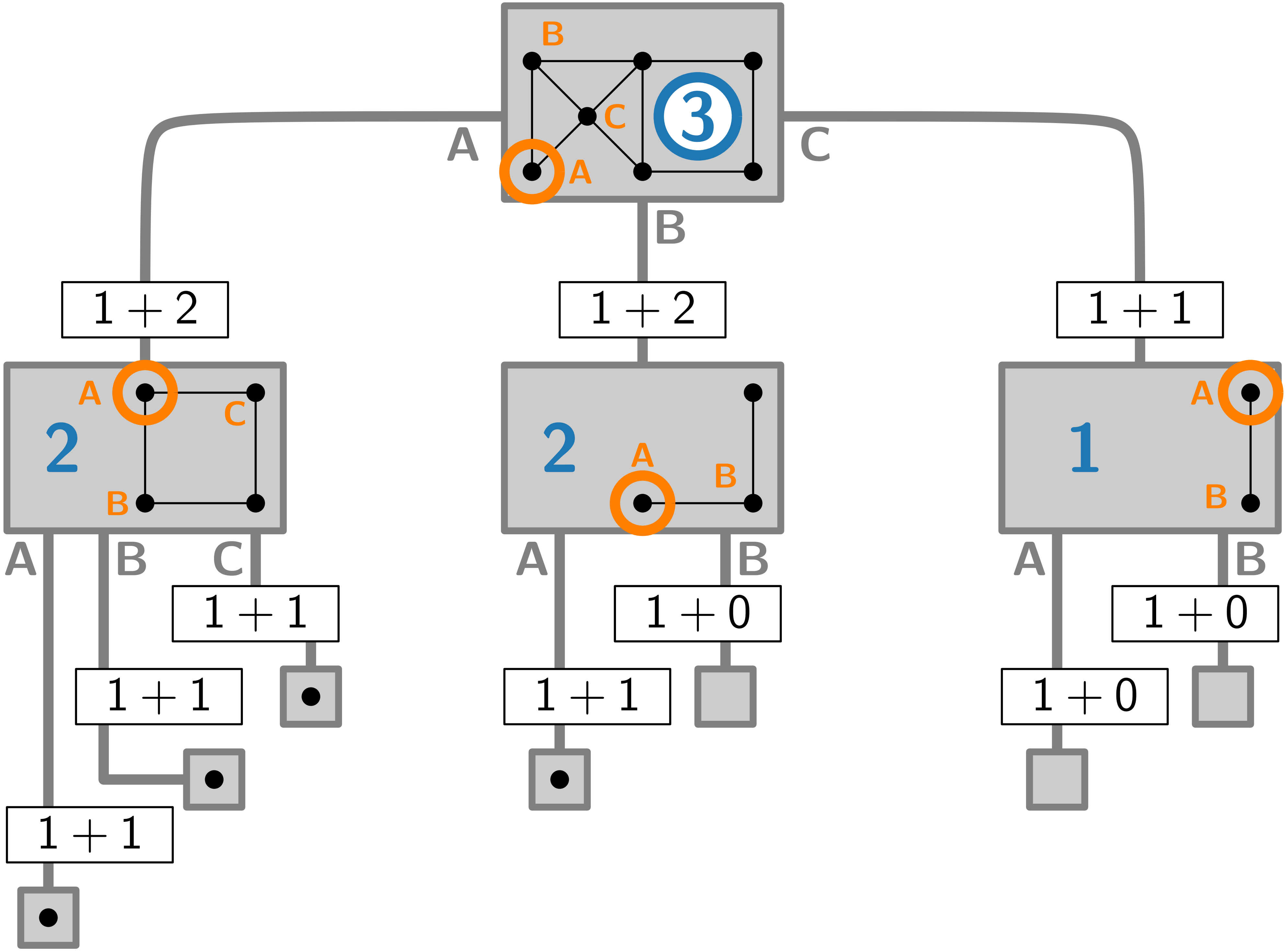
B
C
A
A
B
C
1 + 2
1 + 2
1 + ?
A
C
B
2
2
A
B
A
B
A
B
C
A
B
1 + 1
1 + 0
1 + 1
1 + 1
1 + 1


B
C
A
A
B
C
1 + 2
1 + 2
1 + ?
A
C
B
2
2
A
B
A
A
B
A
B
C
A
B
A
1 + 1
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1


B
C
A
A
B
C
1 + 2
1 + 2
1 + ?
A
C
2
B
2
A
B
A
B
A
B
C
A
B
A
B
1 + 1
1 + 0
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1


B
C
A
A
B
C
1 + 2
1 + 2
1 + ?
A
C
B
2
2
A
B
1
A
B
A
B
C
A
B
A
B
1 + 1
1 + 0
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1


B
C
A
A
B
C
1 + 2
1 + 2
1 + 1
A
C
B
2
2
A
B
1
A
B
A
B
C
A
B
A
B
1 + 1
1 + 0
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1


B
C
A
3
A
B
C
1 + 2
1 + 2
1 + 1
A
C
B
2
B
2
A
B
1
A
B
A
B
C
A
B
A
B
1 + 1
1 + 0
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1

MIS – Runtime analysis

For a worst-case n-vertex graph G ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1))$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

MIS – Runtime analysis

For a worst-case n-vertex graph G ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1))$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

MIS – Runtime analysis

For a worst-case n-vertex graph G ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\mathrm{deg}(y) + 1)) \leq (\mathrm{deg}(v) + 1) \cdot B(\ n - (\mathrm{deg}(v) + 1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

MIS – Runtime analysis

For a worst-case n-vertex graph G $(n \geq 1)$:

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

MIS – Runtime analysis

For a worst-case n-vertex graph G ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$

MIS – Runtime analysis

For a worst-case n-vertex graph G ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$
- Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n - s)$$

MIS – Runtime analysis

For a worst-case n-vertex graph G $(n \geq 1)$:

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y)+1)) \leq (\deg(v)+1) \cdot B(\ n - (\deg(v)+1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$
- Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n-s) \leq s \cdot 3^{(n-s)/3}$$

MIS – Runtime analysis

For a worst-case n-vertex graph G $(n \geq 1)$:

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y)+1)) \leq (\deg(v)+1) \cdot B(\ n - (\deg(v)+1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$
- Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n-s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3}$$

MIS – Runtime analysis

For a worst-case n-vertex graph G $(n \geq 1)$:

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$
- Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n - s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \overset{?}{\leq} 3^{n/3}$$

MIS – Runtime analysis

For a worst-case n-vertex graph G $(n \geq 1)$:

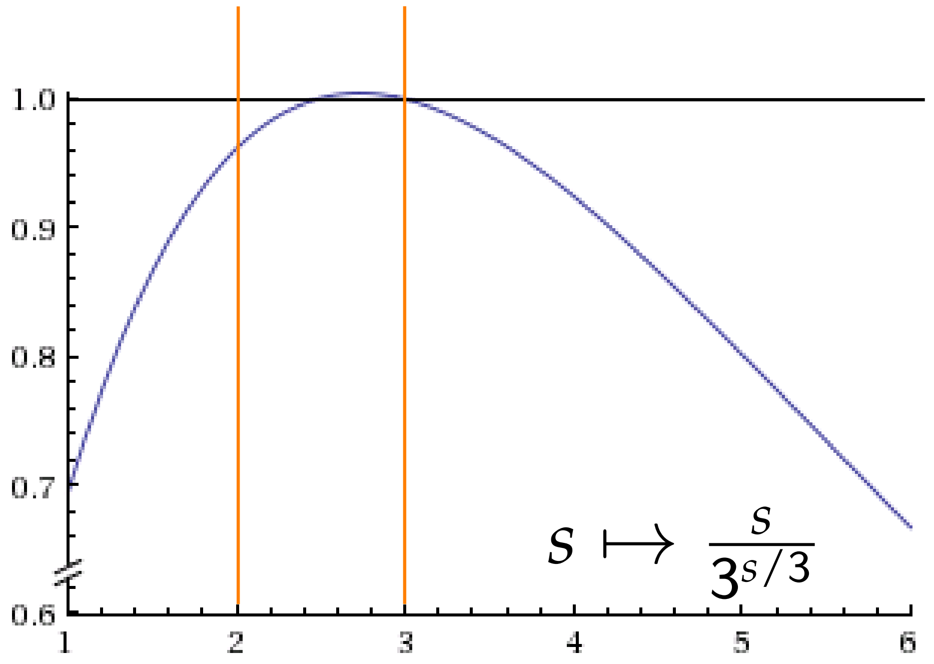
$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1)\ )$$

where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$
- Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n - s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \overset{?}{\leq} 3^{n/3}$$



MIS – Runtime analysis

For a worst-case n-vertex graph G ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y)+1)) \leq (\deg(v)+1) \cdot B(\ n - (\deg(v)+1)\ )$$

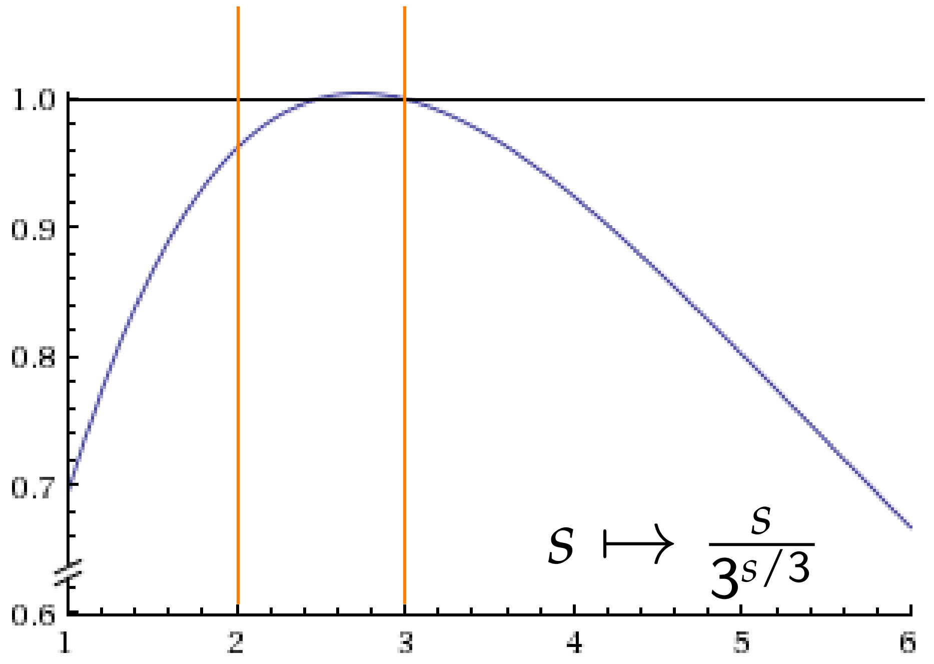
where v is a minimum degree vertex of G, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- Base case: $B(0) = 1 \leq 3^{0/3}$
- Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n-s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \overset{?}{\leq} 3^{n/3}$$

$$B(n) \in O^*(\sqrt[3]{3}^n) \subset O^*(1.44225^n)$$



MIS – Discussion

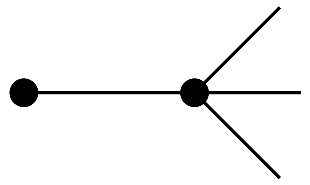
- Smarter branching leads to $\mathcal{O}^*(1.44225^n)$-time algorithme,
- compared to brute-force, which runs in $\mathcal{O}(2^n \cdot n)$ time.

MIS – Discussion

- Smarter branching leads to $\mathcal{O}^*(1.44225^n)$-time algorithme,
- compared to brute-force, which runs in $\mathcal{O}(2^n \cdot n)$ time.
- Algorithms for MIS known that run in $\mathcal{O}^*(1.2202^n)$ time and polynomial space,
- and in $\mathcal{O}^*(1.2109^n)$ time and exponential space.

MIS – Discussion

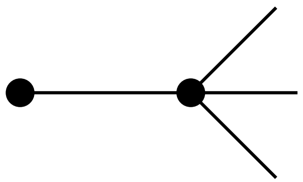
- Smarter branching leads to $\mathcal{O}^*(1.44225^n)$-time algorithme,
- compared to brute-force, which runs in $\mathcal{O}(2^n \cdot n)$ time.
- Algorithms for MIS known that run in $\mathcal{O}^*(1.2202^n)$ time and polynomial space,
- and in $\mathcal{O}^*(1.2109^n)$ time and exponential space.
- What vertices are always in a MIS?
- What vertices can we savely assume are in a MIS?



- Advanced case analysis in [Fomin, Kratsch Ch 2.3] leading to a $\mathcal{O}^*(1.2786^n)$-time algorithm.

MIS – Discussion

- Smarter branching leads to $\mathcal{O}^*(1.44225^n)$-time algorithme,
- compared to brute-force, which runs in $\mathcal{O}(2^n \cdot n)$ time.
- Algorithms for MIS known that run in $\mathcal{O}^*(1.2202^n)$ time and polynomial space,
- and in $\mathcal{O}^*(1.2109^n)$ time and exponential space.
- What vertices are always in a MIS?
- What vertices can we savely assume are in a MIS?



- Advanced case analysis in [Fomin, Kratsch Ch 2.3] leading to a $\mathcal{O}^*(1.2786^n)$-time algorithm.
- **Exercise**: Enumerating MISs
- **Exercise**: Edge-branching for MIS

Literature

Main source:

- [Fomin, Kratsch Ch1] “Exact Exponential Algorithms”

Referenced papers:

- [ADMV ’15] Classic Nintendo Games are (Computationally) Hard
- [Mann ’17] The Top Eight Misconceptions about NP-Hardness