Exploring Sparse Graphs with Advice

H.-J. Böckenhauer[1] **J. Fuchs**[2] W. Unger[2]

[1]Department of Computer Science
ETH Zürich

[2]Computer Science 1
RWTH Aachen University

April 17, 2019

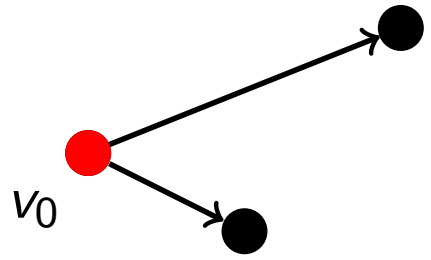
Outline

Online Graph Exploration

Task: find a shortest tour visiting every vertex of an unknown graph G at least once.

Online Graph Exploration

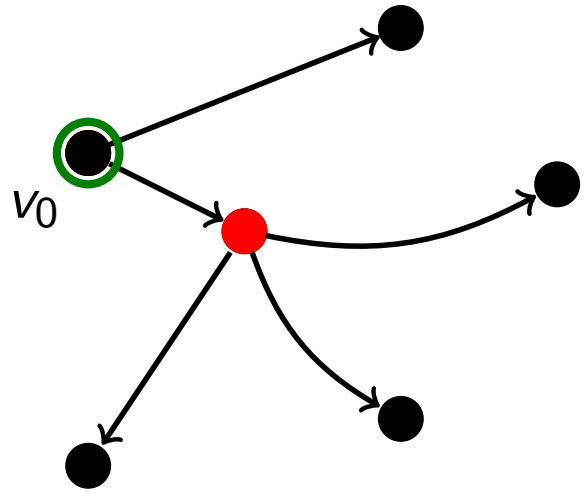
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.

Online Graph Exploration

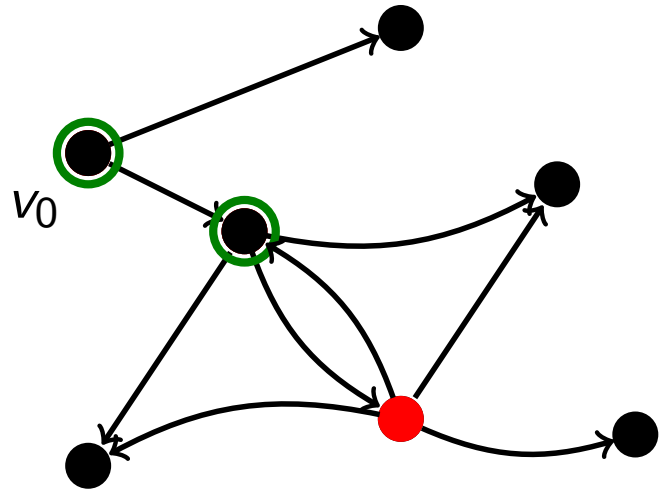
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).

Online Graph Exploration

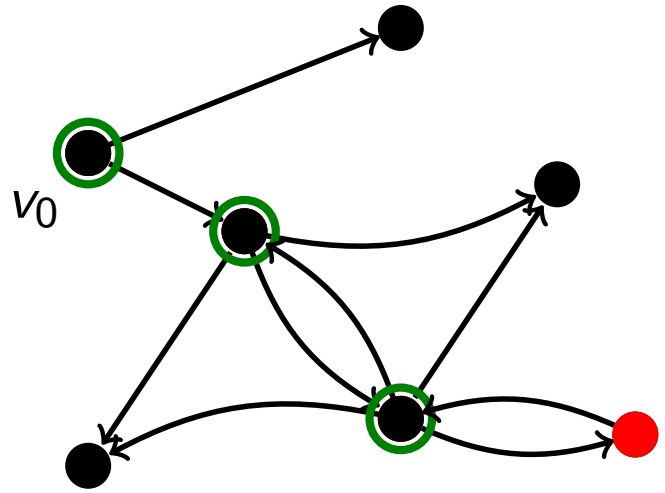
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.

Online Graph Exploration

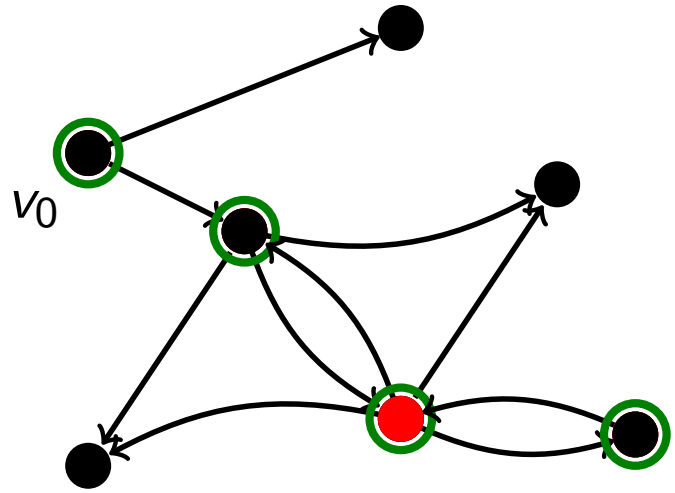
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

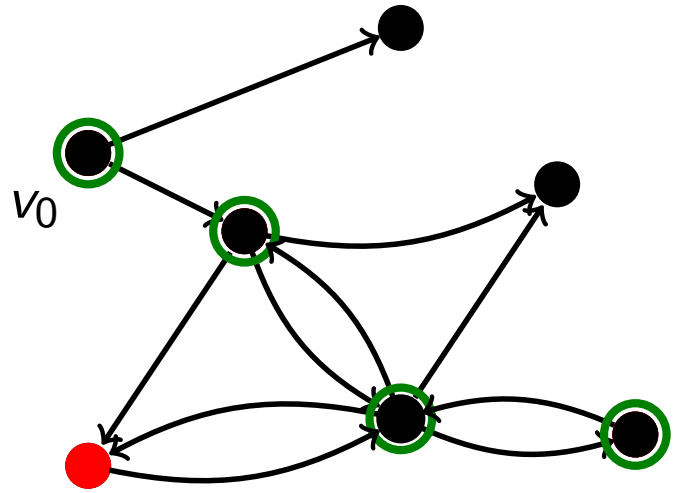
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

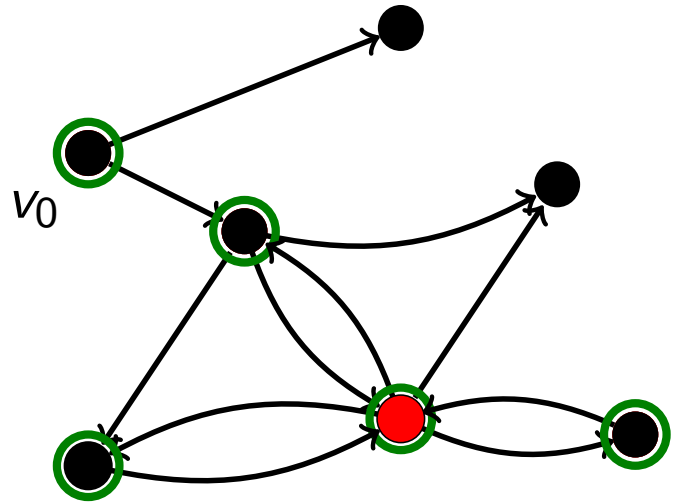
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

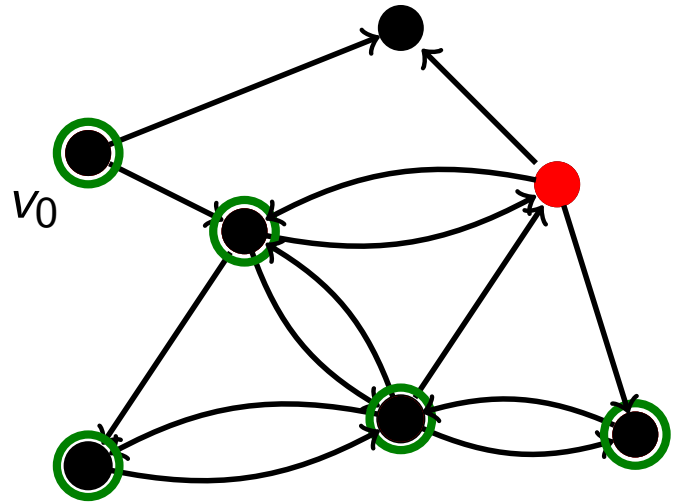
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

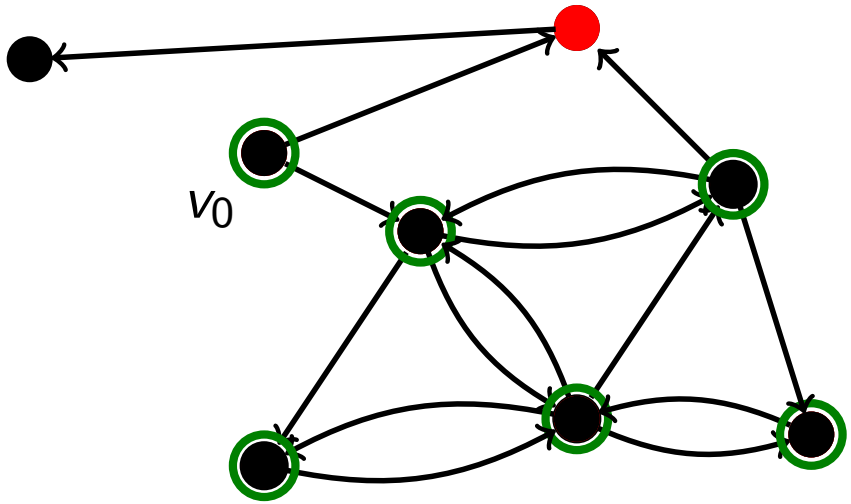
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

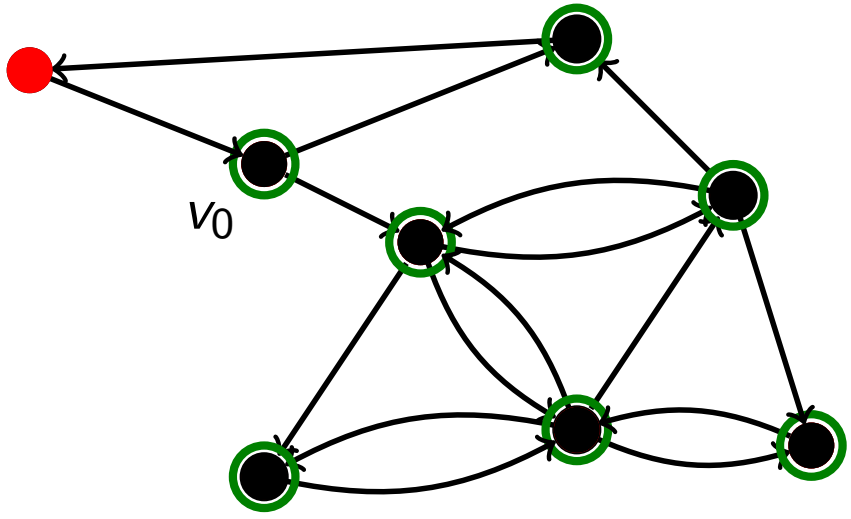
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

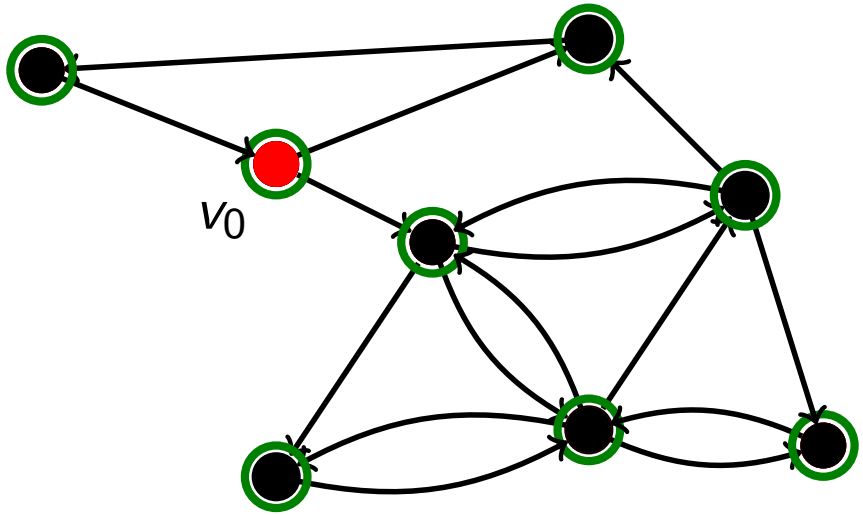
Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

Online Graph Exploration

Task: find a shortest tour visiting every vertex of an unknown graph G at least once.



- Algorithm starts at v_0.
- Visiting a vertex reveals reachable vertices (outgoing edges).
- Algorithm recognizes already seen vertices.
- G is strongly connected.

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

Variations of the problem:

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

Variations of the problem:

- Compute a shortest cyclic tour that visits every vertex.

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

Variations of the problem:

- Compute a shortest cyclic tour that visits every vertex.
- Compute a shortest non-cyclic tour that visits every vertex.

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

Variations of the problem:

- Compute a shortest cyclic tour that visits every vertex.
- Compute a shortest non-cyclic tour that visits every vertex.
- Compute a shortest path to a specified vertex. (Treasure Hunt Problem)

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

Variations of the problem:

- Compute a shortest cyclic tour that visits every vertex.
- Compute a shortest non-cyclic tour that visits every vertex.
- Compute a shortest path to a specified vertex. (Treasure Hunt Problem)
- Many other variations in the number of agents or their abilities/perception.

History of the Problem

Kalyanasundaram et al. introduced the graph exploration problem as an online version of the TSP (ICALP, 1993).

The *fixed-graph scenario* defines the senses of the algorithm:

- Vertices have unique labels.
- Algorithm sees reachable vertices and their labels.

Variations of the problem:

- Compute a shortest cyclic tour that visits every vertex.
- Compute a shortest non-cyclic tour that visits every vertex.
- Compute a shortest path to a specified vertex. (Treasure Hunt Problem)
- Many other variations in the number of agents or their abilities/perception.

Advice Complexity

Classical online setting focuses on the competitive ratio.

Problem:
Some problems are not competitive.
Online problems are hard to compare.

Advice Complexity

Classical online setting focuses on the competitive ratio.

Problem:
Some problems are not competitive.
Online problems are hard to compare.

Dobrev et al. (SOFSEM, 2008) introduced an extension:
Advice Complexity
More refined model introduced by Hromkovic et al. (MFCS 2010) and Böckenhauer et al. (ISAAC 2009).

Advice Complexity

Classical online setting focuses on the competitive ratio.

Problem:
Some problems are not competitive.
Online problems are hard to compare.

Dobrev et al. (SOFSEM, 2008) introduced an extension:
Advice Complexity
More refined model introduced by Hromkovic et al. (MFCS 2010) and Böckenhauer et al. (ISAAC 2009).

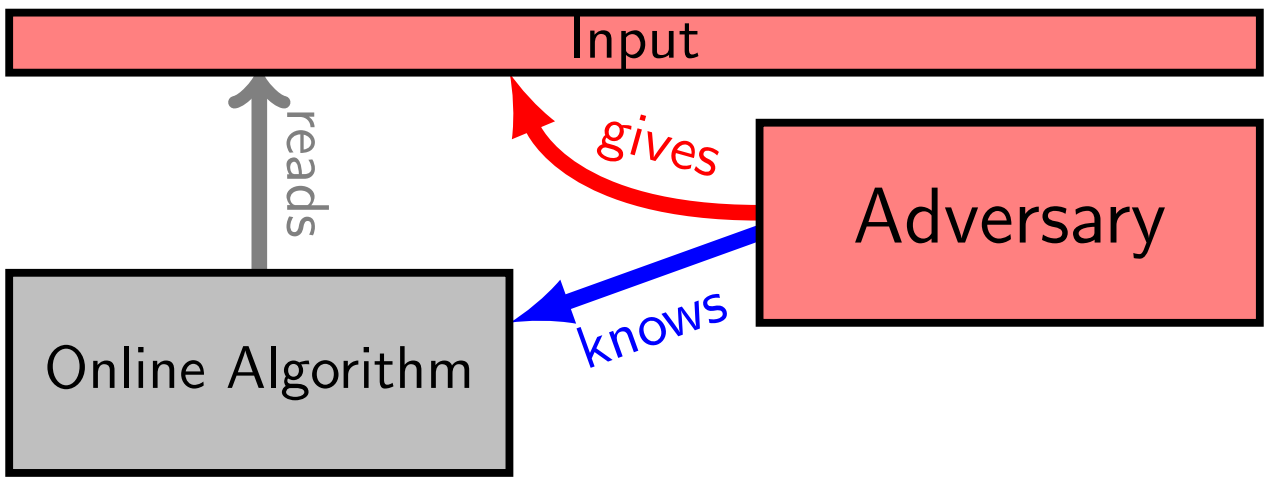
Algorithm can "buy" information about the future.

The Oracle

Algorithm and oracle are a team and play against an adversary.

The classical setting:

- The algorithm receives the input step by step.
- Adversary knows algorithm and prepares worst-case input.

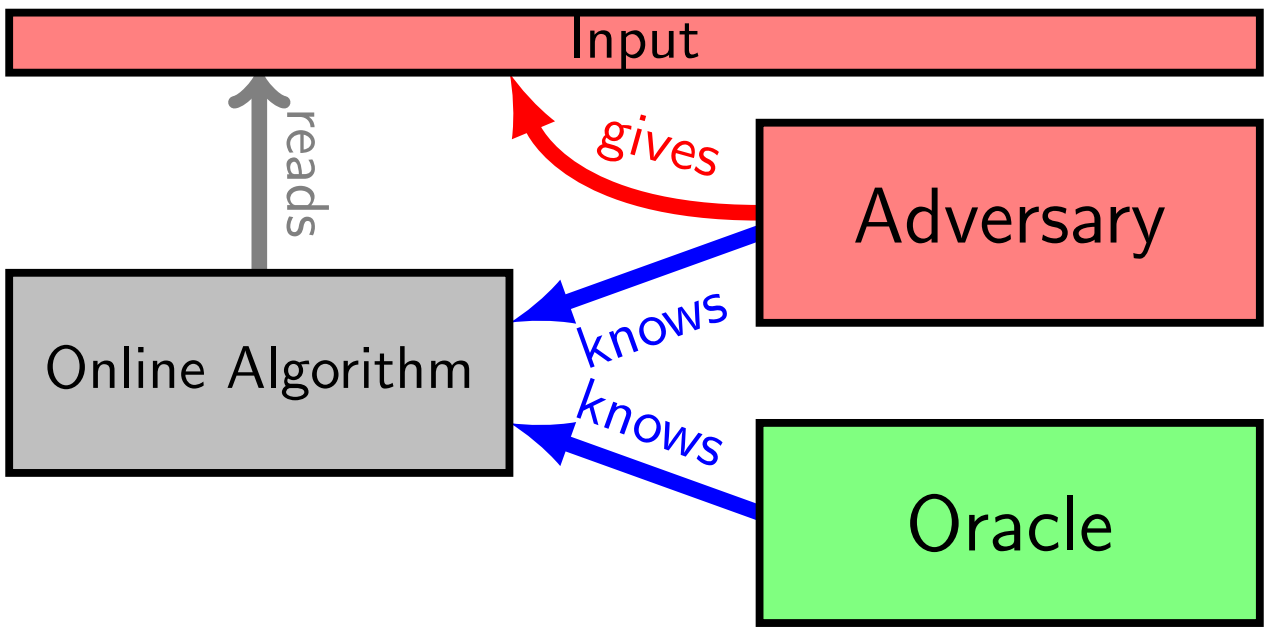


The Oracle

Algorithm and oracle are a team and play against an adversary.

The new setting:

- Oracle knows the input and prepares some information.
- Algorithm can access the information at any time.

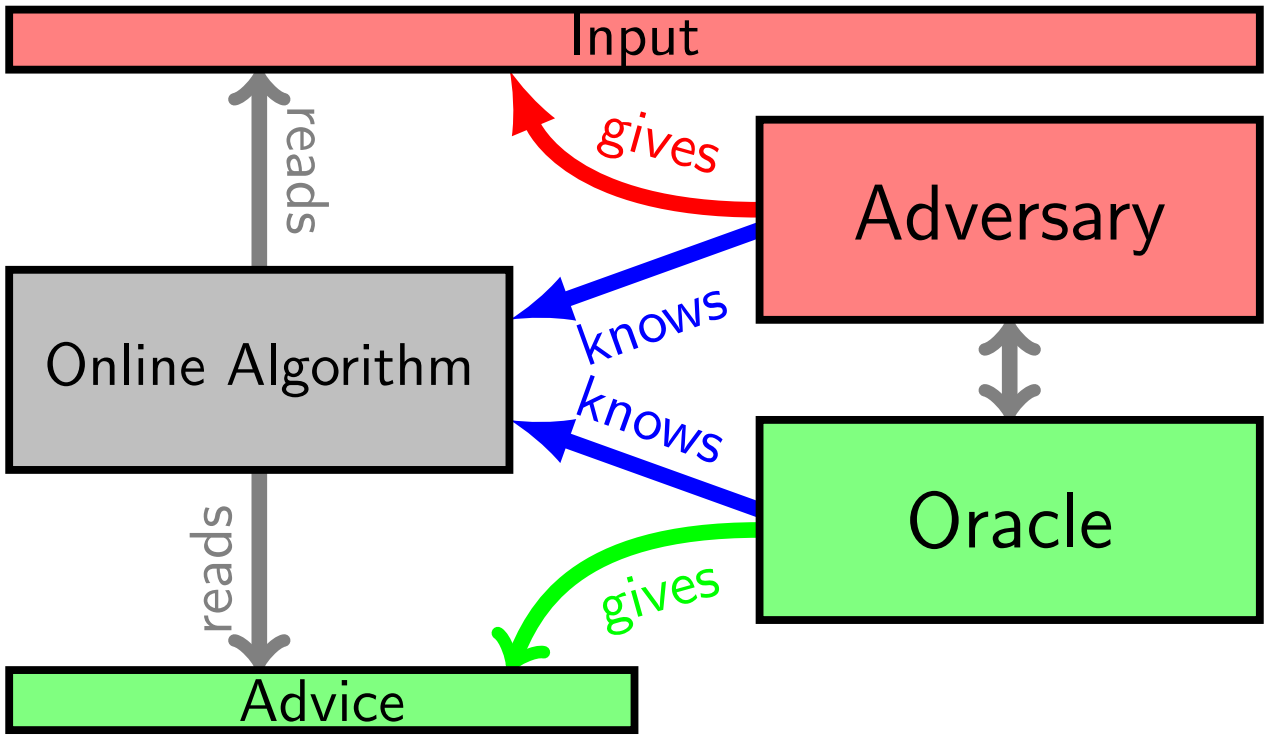


The Oracle

Algorithm and oracle are a team and play against an adversary.

The new setting:

- Oracle knows the input and prepares some information.
- Algorithm can access the information at any time.

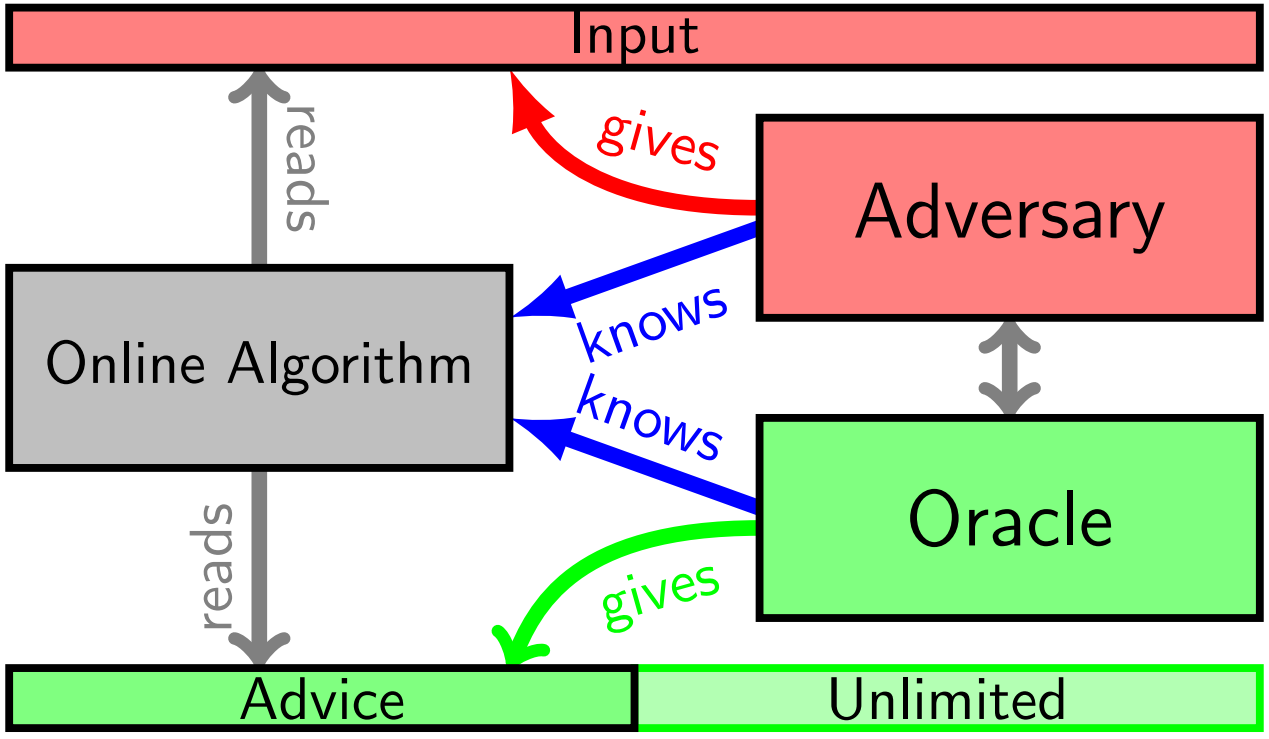


The Oracle

Algorithm and oracle are a team and play against an adversary.

The new setting:

- Oracle knows the input and prepares some information.
- Algorithm can access the information at any time.



Problem and Setting Together

- Adversary constructs G.
- Oracle knows G and the optimal exploration tour S^*.
- Algorithm does not know G.
- Visiting a vertex for the first time reveals its outgoing edges.
- Algorithm can access the advice tape at any time to get information.

Task: compute a shortest tour that visits every vertex at least once.

Problem and Setting Together

- Adversary constructs G.
- Oracle knows G and the optimal exploration tour S^*.
- Algorithm does not know G.
- Visiting a vertex for the first time reveals its outgoing edges.
- Algorithm can access the advice tape at any time to get information.

Task: compute a shortest tour that visits every vertex at least once.

The best known upper bound for the number of needed bits was $\mathcal{O}(n \log n)$.
Idea: Encode vertex labels in the order they are visited in S^*.

Problem and Setting Together

- Adversary constructs G.
- Oracle knows G and the optimal exploration tour S^*.
- Algorithm does not know G.
- Visiting a vertex for the first time reveals its outgoing edges.
- Algorithm can access the advice tape at any time to get information.

Task: compute a shortest tour that visits every vertex at least once.

The best known upper bound for the number of needed bits was $\mathcal{O}(n \log n)$.
Idea: Encode vertex labels in the order they are visited in S^*.

We will show an algorithm that needs at most $\mathcal{O}(m)$ bits of advice.

Edges

For sake of simplicity, we assume that every vertex in G has at most two incoming edges and at most two outgoing edges.
$G = (V, E)$ is strongly connected with $|V| = n$ and $|E| = m$.

The algorithm misses information about the incoming edges.

Edges

For sake of simplicity, we assume that every vertex in G has at most two incoming edges and at most two outgoing edges.
$G = (V, E)$ is strongly connected with $|V| = n$ and $|E| = m$.

The algorithm misses information about the incoming edges.

Algorithm asks if a new vertex has one or two incoming edges.
(G is strongly connected)

Edges

For sake of simplicity, we assume that every vertex in G has at most two incoming edges and at most two outgoing edges.
$G = (V, E)$ is strongly connected with $|V| = n$ and $|E| = m$.

The algorithm misses information about the incoming edges.

Algorithm asks if a new vertex has one or two incoming edges.
(G is strongly connected)
$\rightarrow n$ bits of advice.

Edges

For sake of simplicity, we assume that every vertex in G has at most two incoming edges and at most two outgoing edges.
$G = (V, E)$ is strongly connected with $|V| = n$ and $|E| = m$.

The algorithm misses information about the incoming edges.

Algorithm asks if a new vertex has one or two incoming edges.
(G is strongly connected)
$\rightarrow n$ bits of advice.

The edges are classified with respect to S^*:

- E_0: Set of edges that are never used in S^*.
- E_1: Set of edges that are used precisely once in S^*.
- E_{Multi}: Set of edges that are used multiple times in S^*.

Edges

For sake of simplicity, we assume that every vertex in G has at most two incoming edges and at most two outgoing edges.
$G = (V, E)$ is strongly connected with $|V| = n$ and $|E| = m$.

The algorithm misses information about the incoming edges.

Algorithm asks if a new vertex has one or two incoming edges.
(G is strongly connected)
$\rightarrow n$ bits of advice.

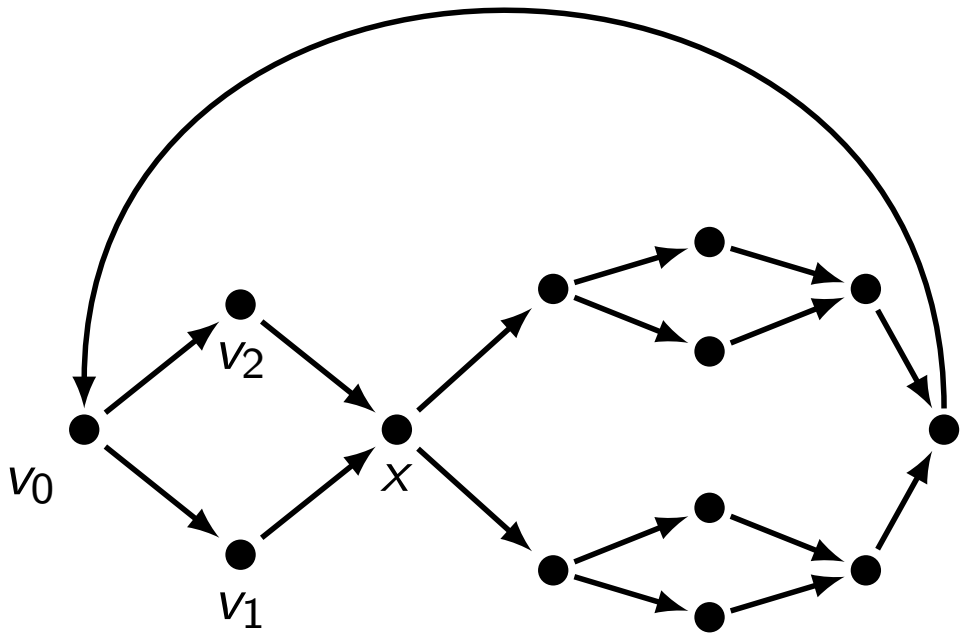
The edges are classified with respect to S^*:

- E_0: Set of edges that are never used in S^*.
- E_1: Set of edges that are used precisely once in S^*.
- E_{Multi}: Set of edges that are used multiple times in S^*.

$\rightarrow \log(3) \cdot m$ bits of advice.

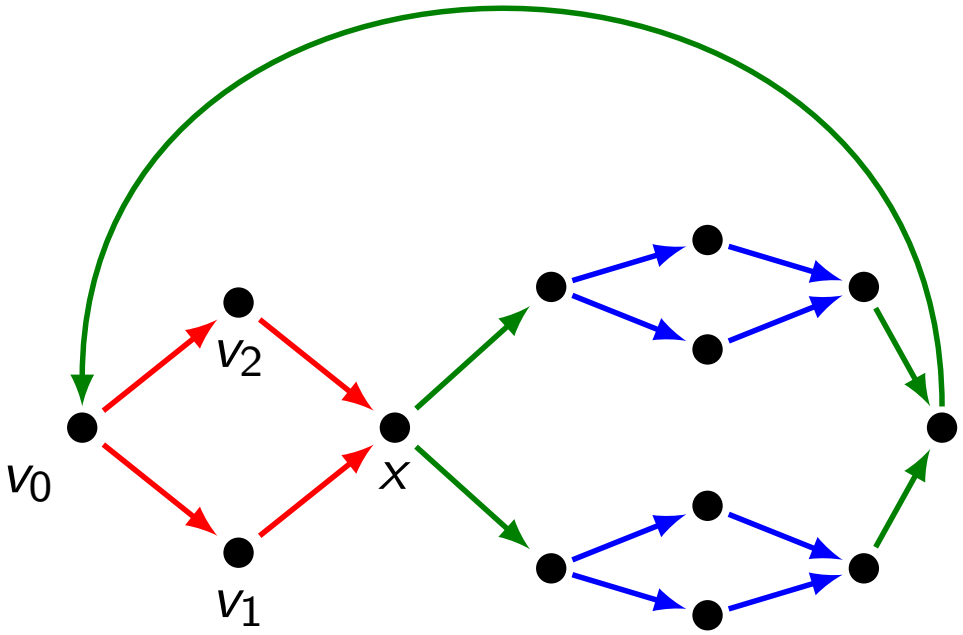
The Optimal Solution S^*

Is S^* or the classification into the sets E_0, E_1, E_{Multi} unique?



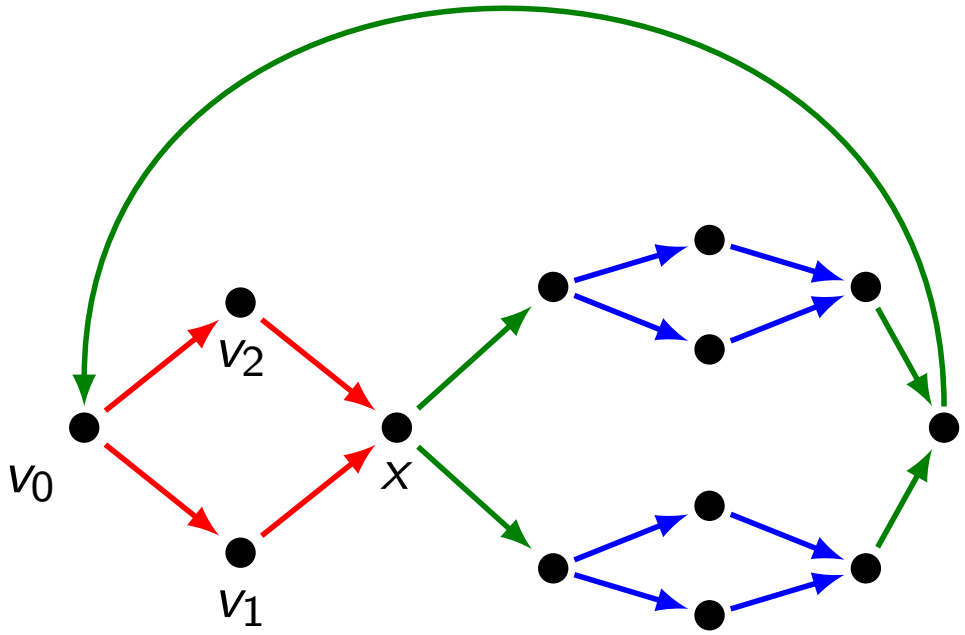
The Optimal Solution S^*

Is S^* or the classification into the sets E_0, E_1, E_{Multi} unique? No



The Optimal Solution S^*

Is S^* or the classification into the sets E_0, E_1, E_{Multi} unique? No

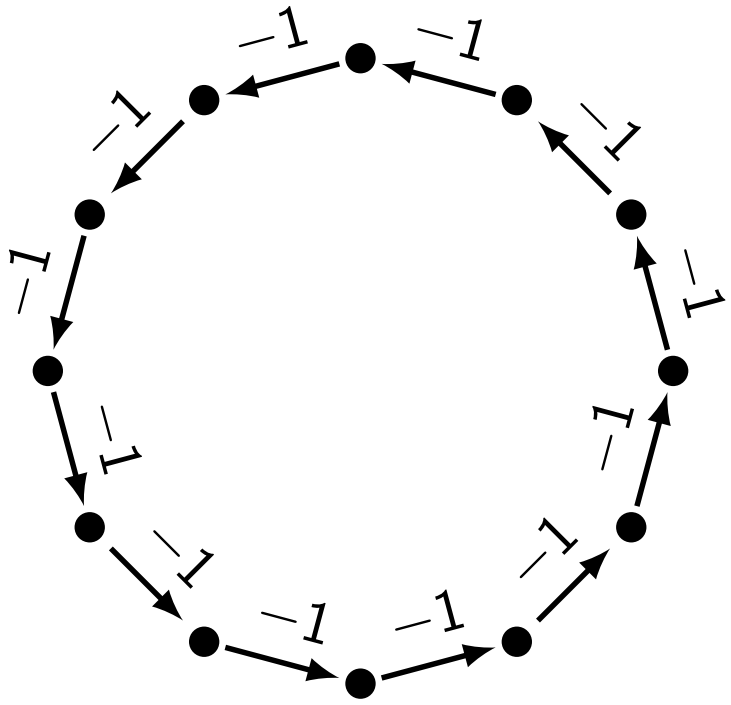


But the oracle can choose an optimal solution S^* that fulfills certain properties.

Properties of S^*

The oracle fixes an optimal solution such that the edges from E_{Multi} do not form a cycle.

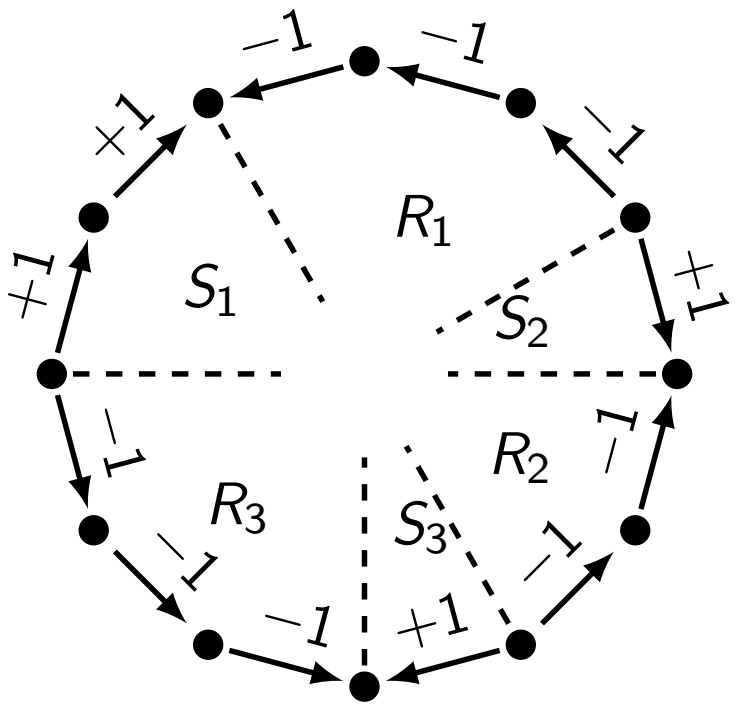
Obviously the edges from E_{Multi} cannot form a directed cycle.



Properties of S^*

The oracle fixes an optimal solution such that the edges from E_{Multi} do not form a cycle.

Even in the underlying undirected graph.



R_i and S_i are paths of edges from E_{Multi} to the same vertex.

The Graph Structure

The graph $F = (V, E_{Multi})$ is a forest.
Even the underlying undirected graph of F is cycle-free.

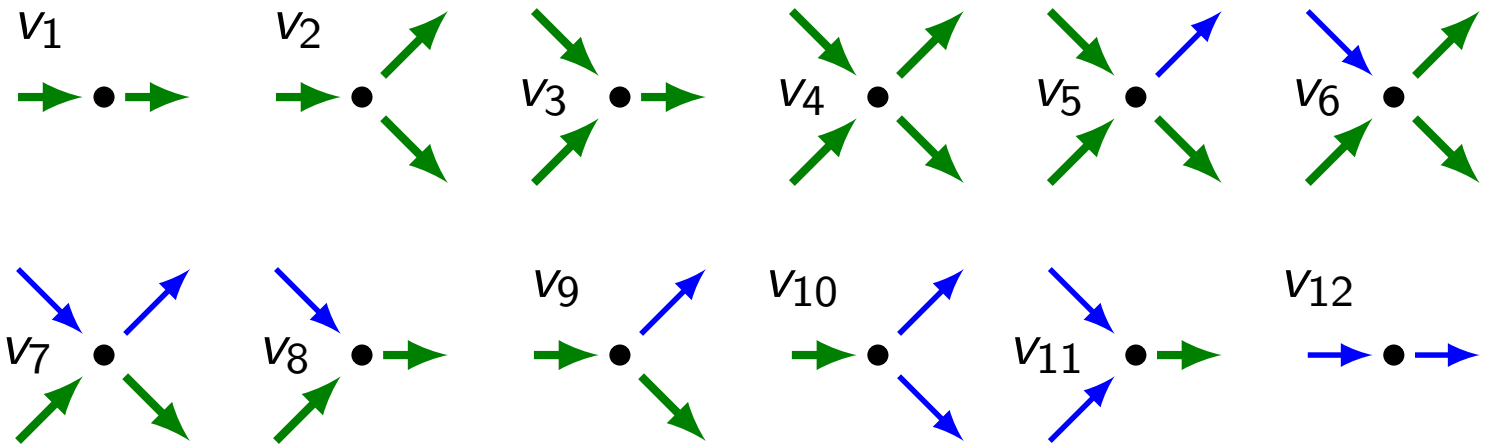
The Graph $G_{S^*} = (V, E_1 \cup E_{Multi})$ contains cycles.
Thus, every cycle contains at least one edge from E_1.

The Graph Structure

The graph $F = (V, E_{Multi})$ is a forest.
Even the underlying undirected graph of F is cycle-free.

The Graph $G_{S^*} = (V, E_1 \cup E_{Multi})$ contains cycles.
Thus, every cycle contains at least one edge from E_1.

There are only twelve edge configurations possible:



Rough Idea

A vertex that is visited x times in S^* is part of $x - 1$ interchangeable cycles.

Last traversal is part of the "main" cycle that starts and ends at v_0.

The interchangeable cycles can be traversed in any order.
The algorithm needs to be careful:

Rough Idea

A vertex that is visited x times in S^* is part of $x - 1$ interchangeable cycles.

Last traversal is part of the "main" cycle that starts and ends at v_0.

The interchangeable cycles can be traversed in any order.
The algorithm needs to be careful:

- Finish all interchangeable cycles before continuing the "main" cycle.

Rough Idea

A vertex that is visited x times in S^* is part of $x - 1$ interchangeable cycles.

Last traversal is part of the "main" cycle that starts and ends at v_0.

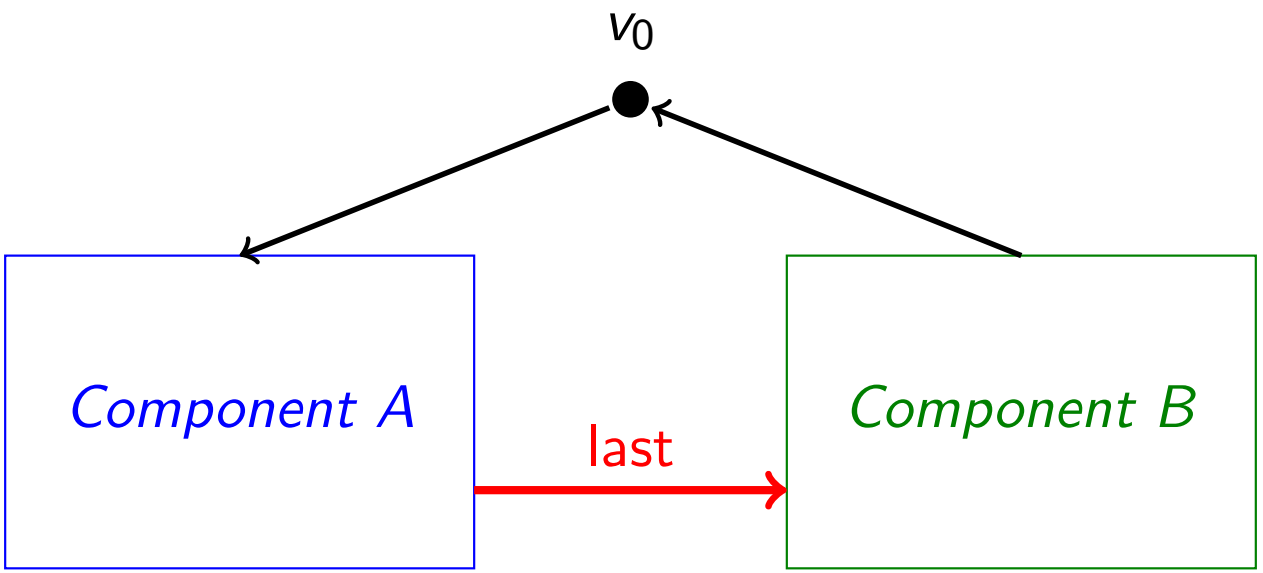
The interchangeable cycles can be traversed in any order.
The algorithm needs to be careful:

- Finish all interchangeable cycles before continuing the "main" cycle.
- Edges cannot be traversed more often than in S^*.

What can go wrong?

The algorithm needs to avoid to cut off components.

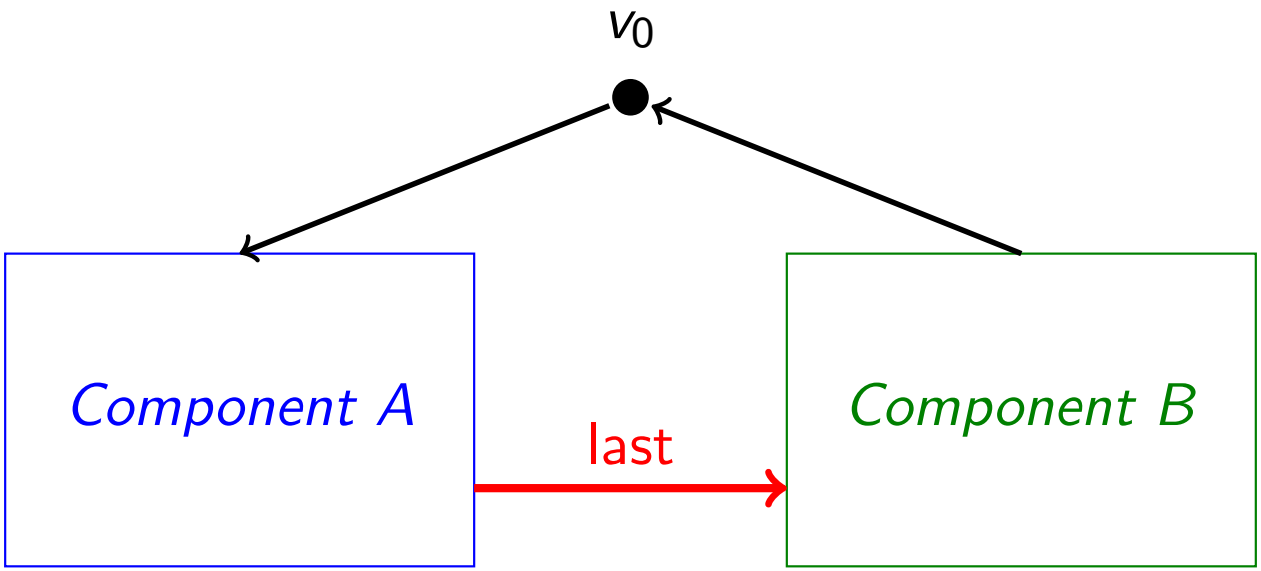
Edges that are used after some component is completely explored are called *last*. (These edges continue the "main" cycle)



What can go wrong?

The algorithm needs to avoid to cut off components.

Edges that are used after some component is completely explored are called *last*. (These edges continue the "main" cycle)



The algorithm asks for every vertex once, which of the two outgoing edges is *last*. $\rightarrow$ n bits of advice.

Number of Traversals $\#_{S^*}$

$2n + \log(3)m$ bits of advice tell the algorithm which edges are used precisely once, never, more than once and which edges are *last*.

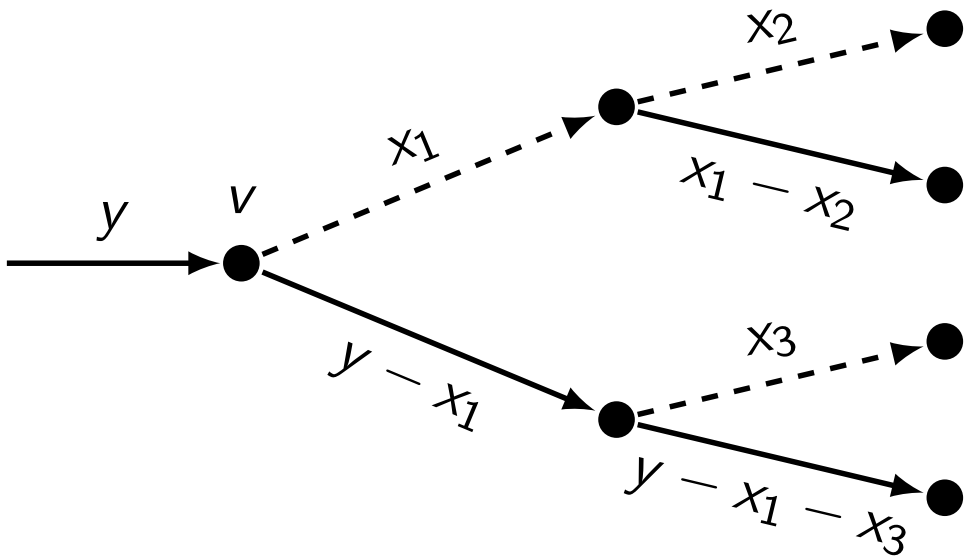
Problem: $\#_{S^*}(e)$ for an edge $e \in E_{Multi}$ is unknown.

Number of Traversals $\#_{S^*}$

$2n + \log(3)m$ bits of advice tell the algorithm which edges are used precisely once, never, more than once and which edges are *last*.

Problem: $\#_{S^*}(e)$ for an edge $e \in E_{Multi}$ is unknown.

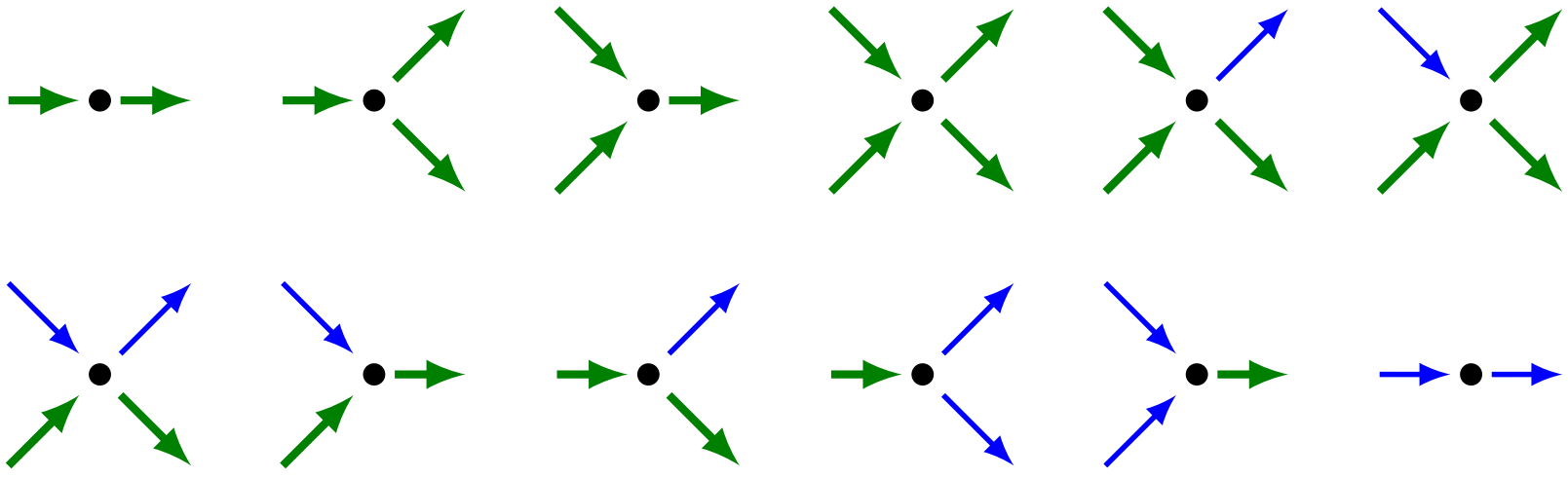
The numbers of traversals influence each other.



If $\#_{S^*}(e)$ is known, the algorithm will not make any mistake.

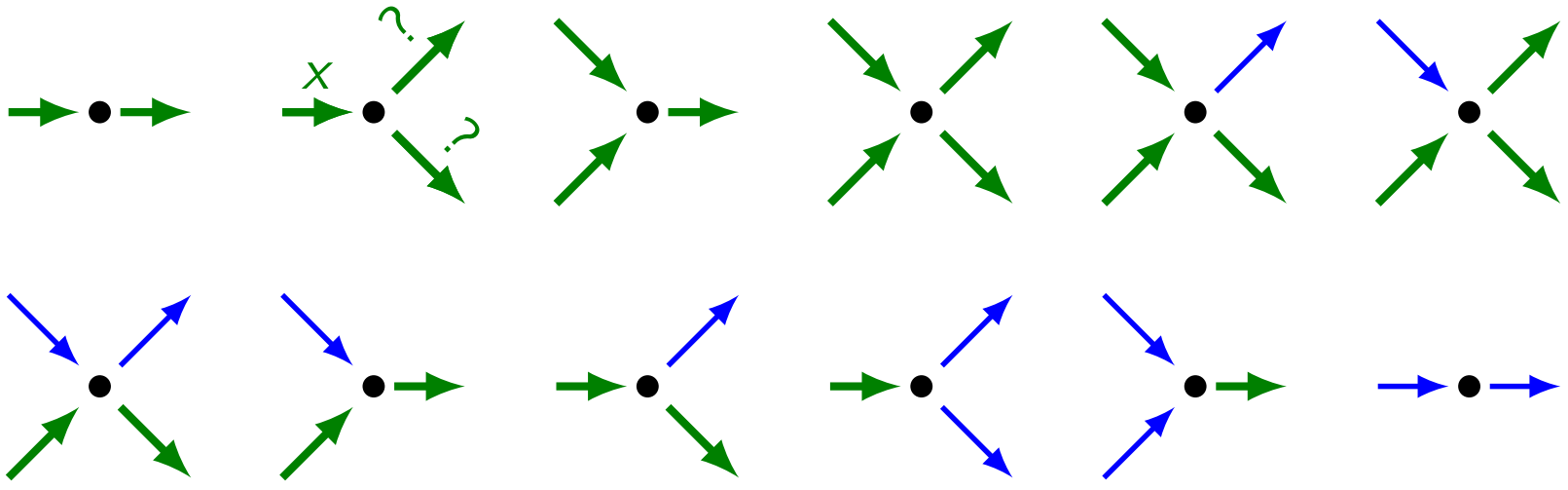
Heavy and Light Edges

Remember the edge configuration for $G_{S^*} = (V, E_1 \cup E_{Multi})$.



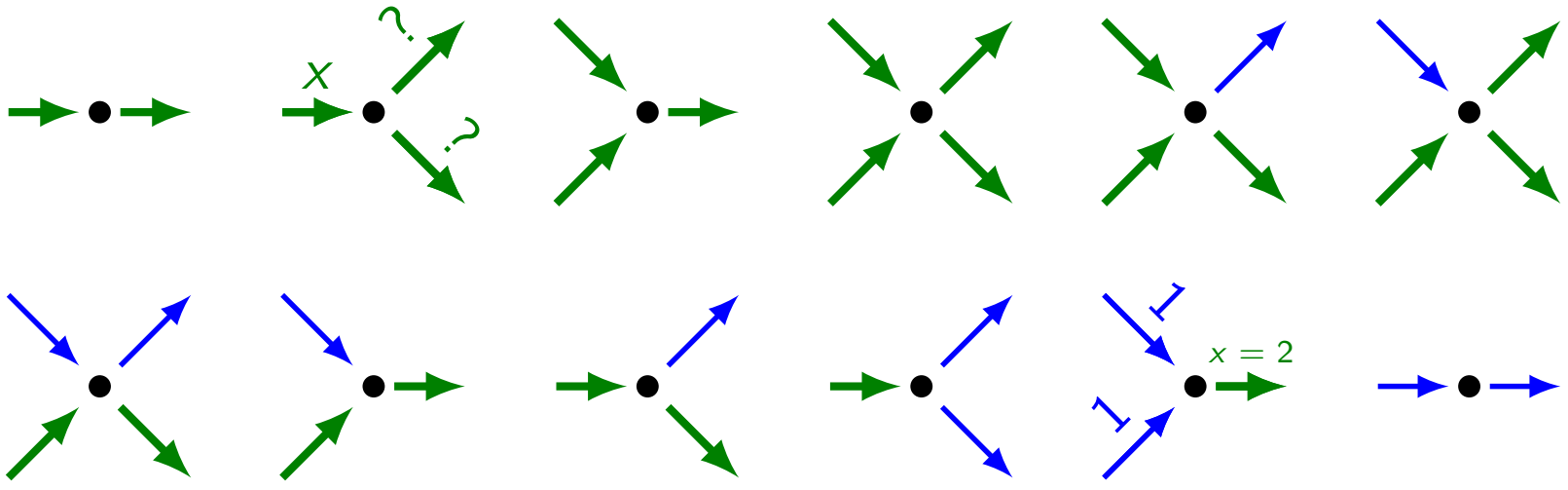
Heavy and Light Edges

Remember the edge configuration for $G_{S^*} = (V, E_1 \cup E_{Multi})$.



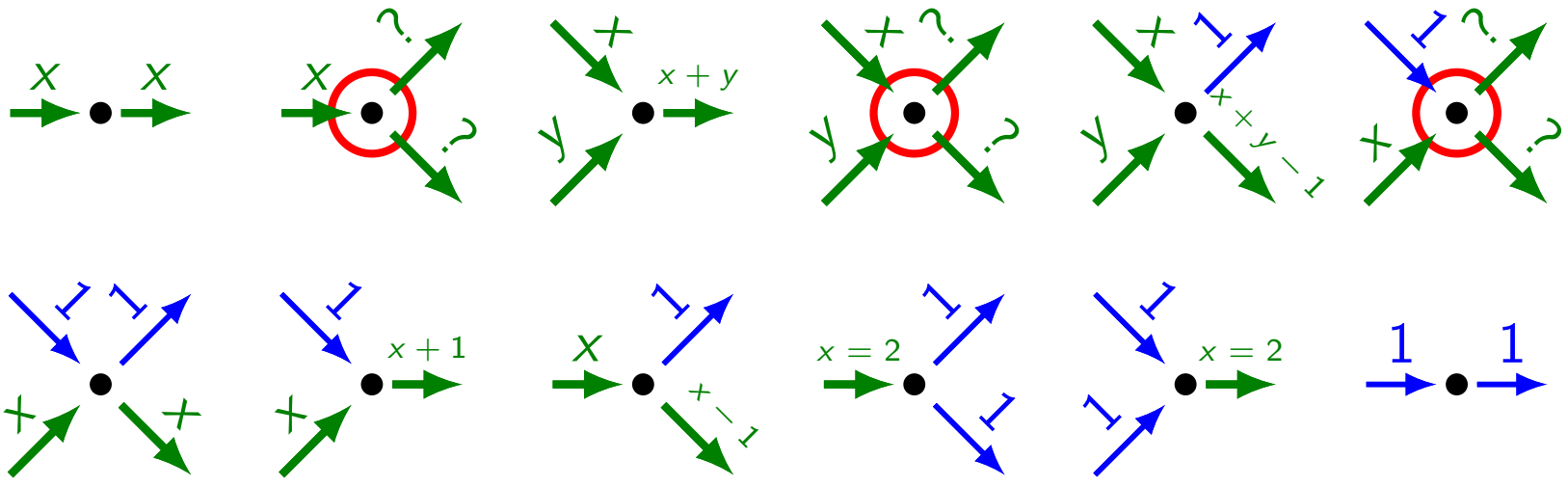
Heavy and Light Edges

Remember the edge configuration for $G_{S^*} = (V, E_1 \cup E_{Multi})$.



Heavy and Light Edges

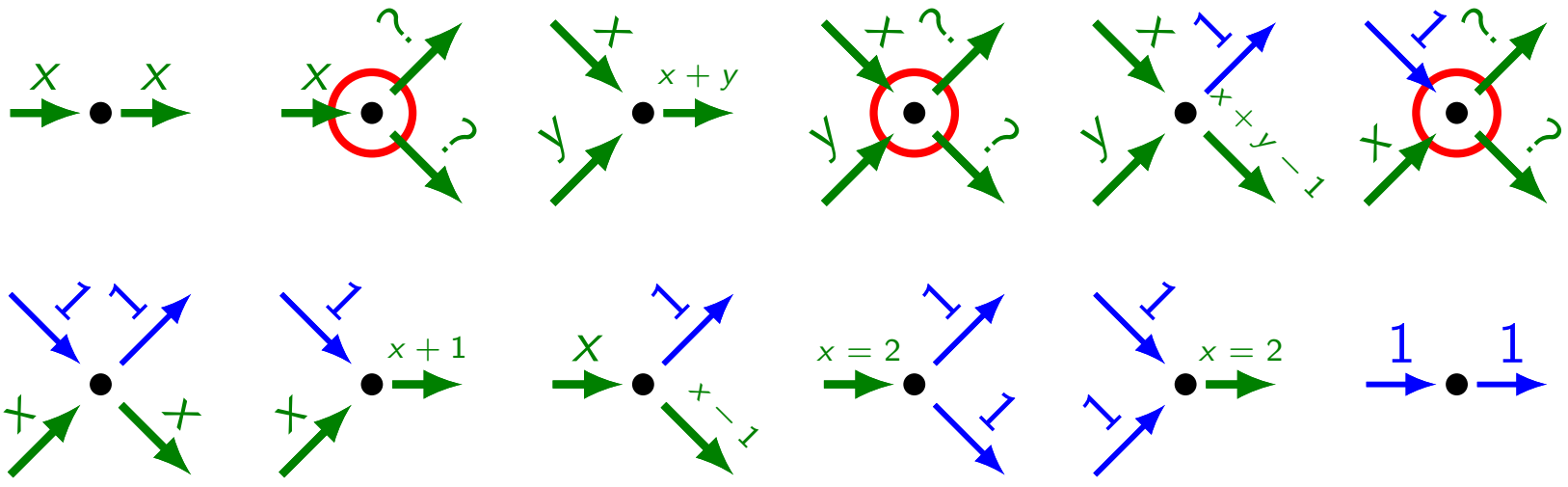
Remember the edge configuration for $G_{S^*} = (V, E_1 \cup E_{Multi})$.



Advice is needed if algo can choose between two edges from E_{Multi}.

Heavy and Light Edges

Remember the edge configuration for $G_{S^*} = (V, E_1 \cup E_{Multi})$.

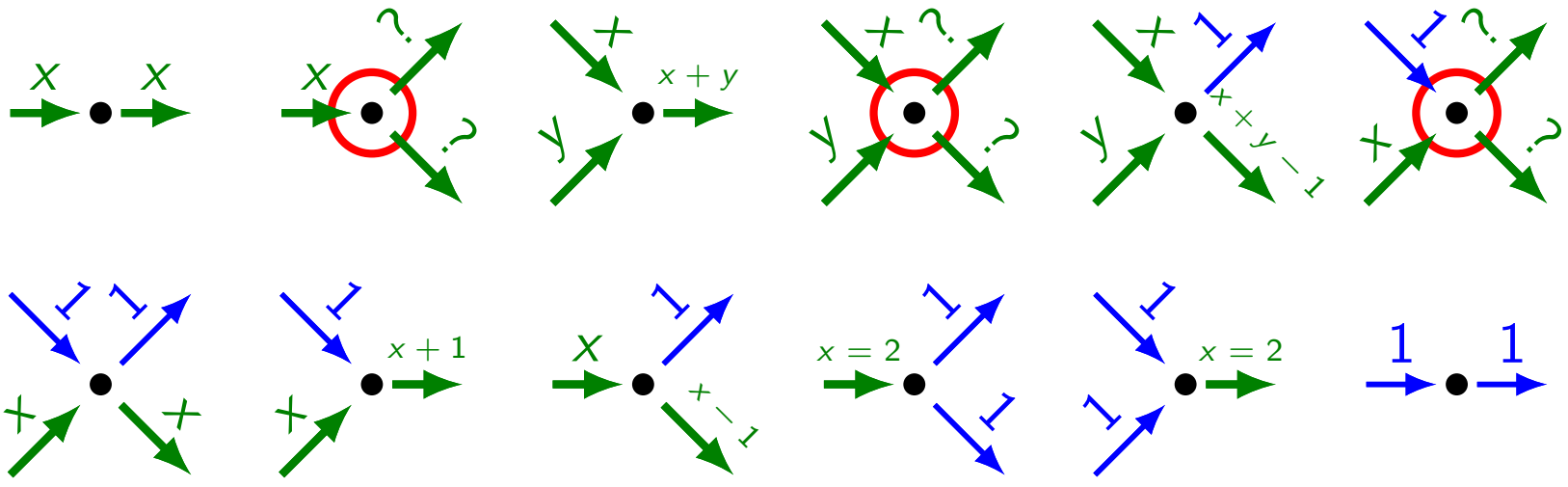


Advice is needed if algo can choose between two edges from E_{Multi}.

Ask which edge is less often used: $\#_{S^*}(e_1) \leq \#_{S^*}(e_2)$?

Heavy and Light Edges

Remember the edge configuration for $G_{S^*} = (V, E_1 \cup E_{Multi})$.



Advice is needed if algo can choose between two edges from E_{Multi}.

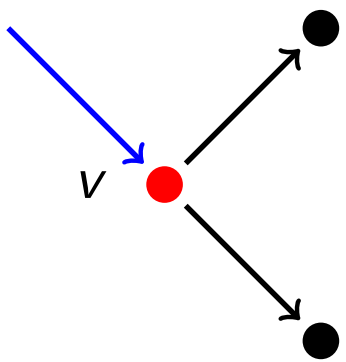
Ask which edge is less often used: $\#_{S^*}(e_1) \leq \#_{S^*}(e_2)$?

The less often used edge is called *light*.
Additionally, the algorithm asks for its precise number of traversals.

Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

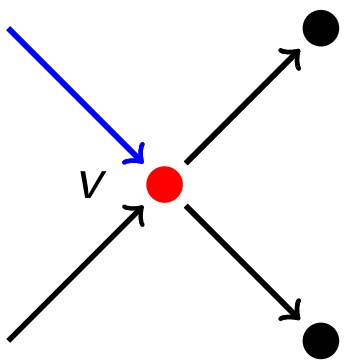
The algorithm enters a vertex v for the first time.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

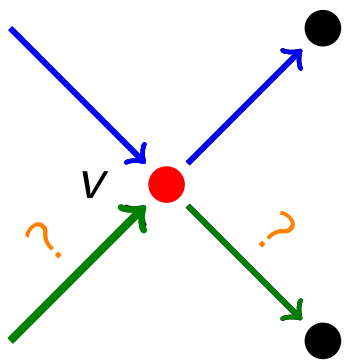
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

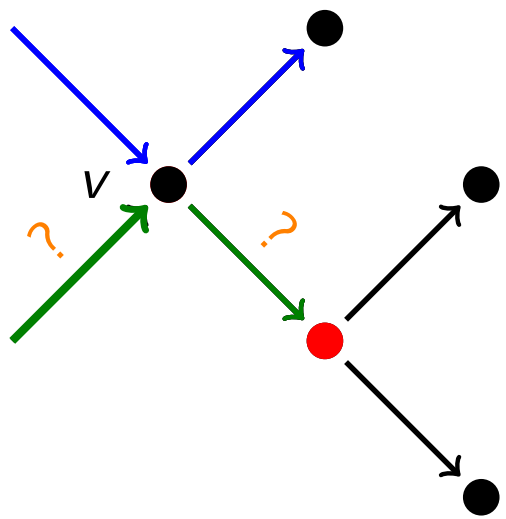
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

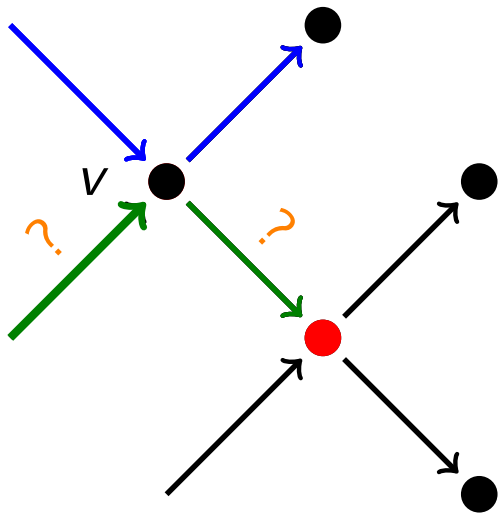
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

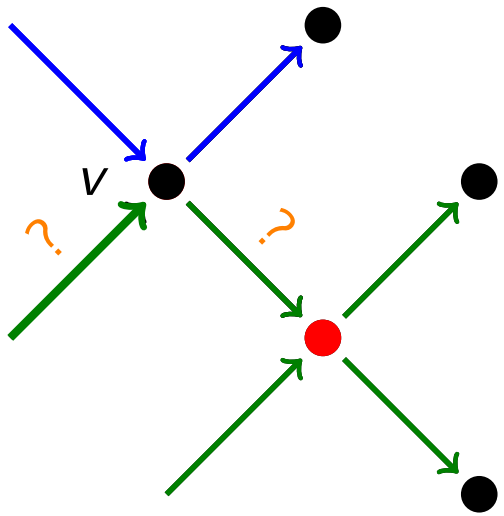
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

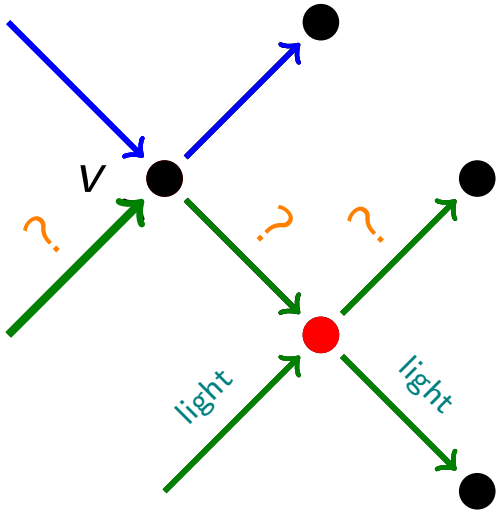
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

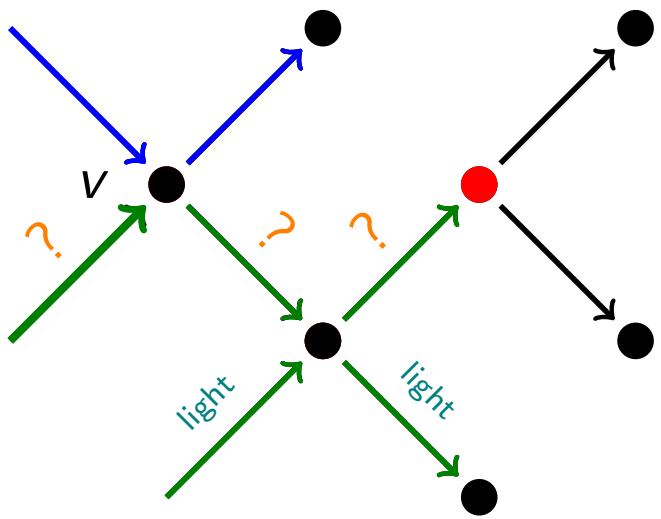
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

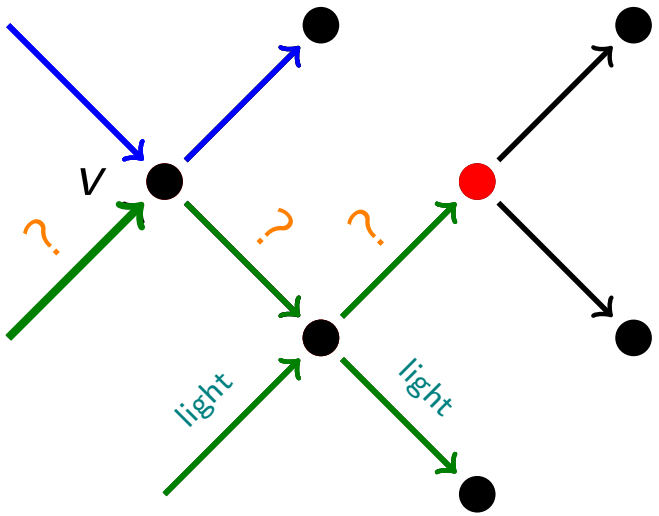
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

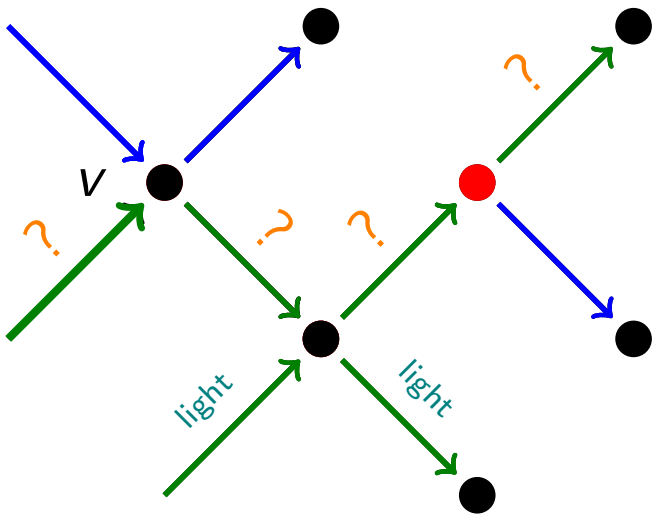
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

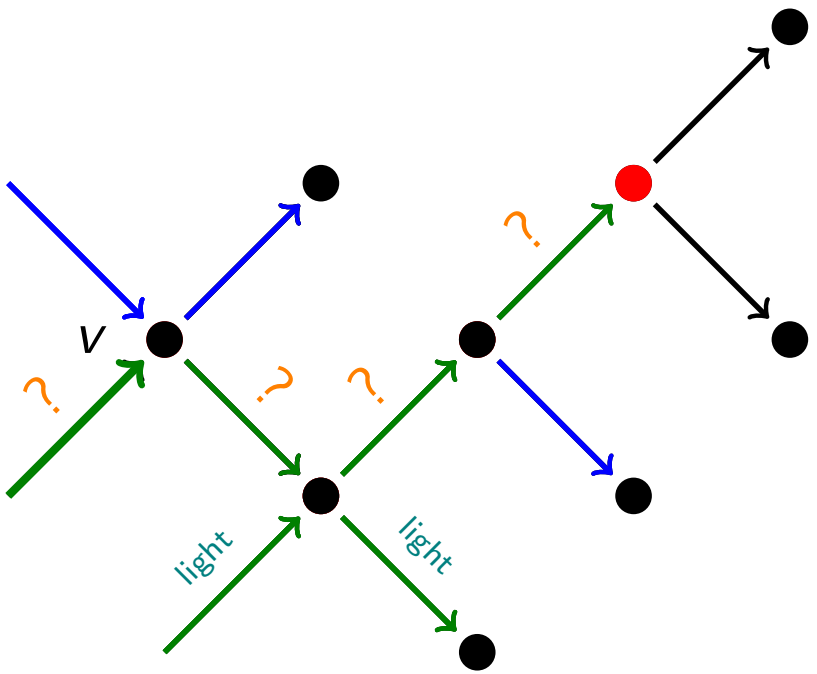
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

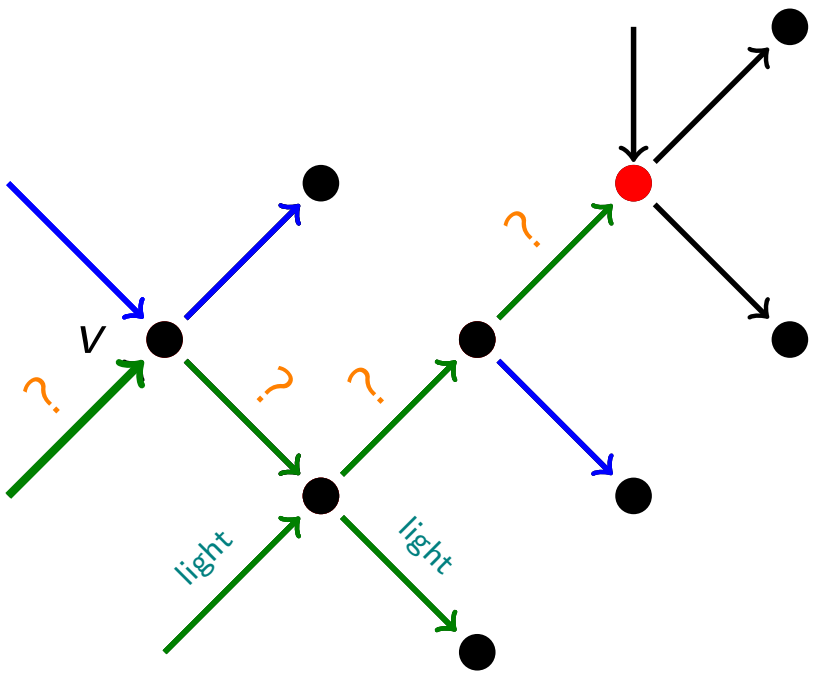
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

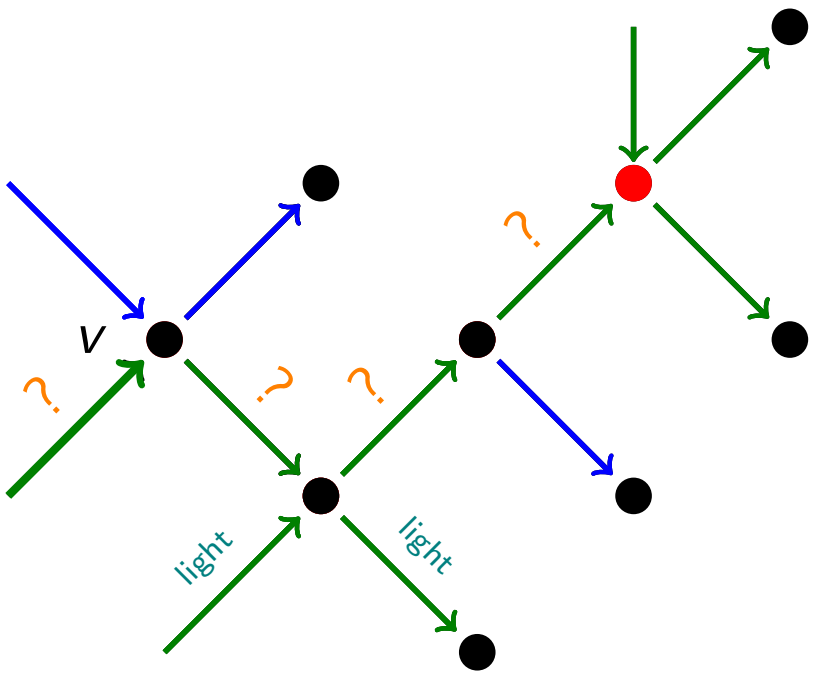
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

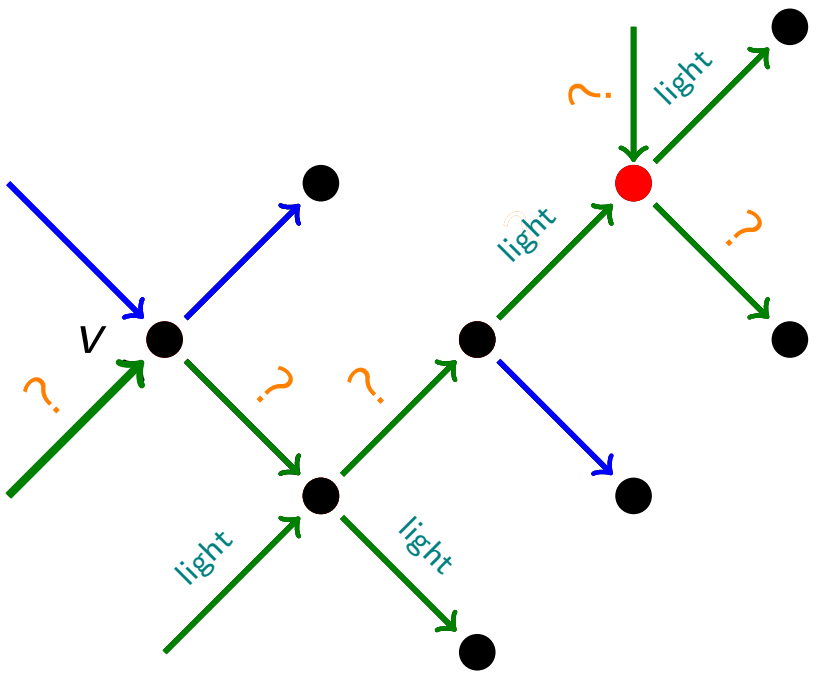
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

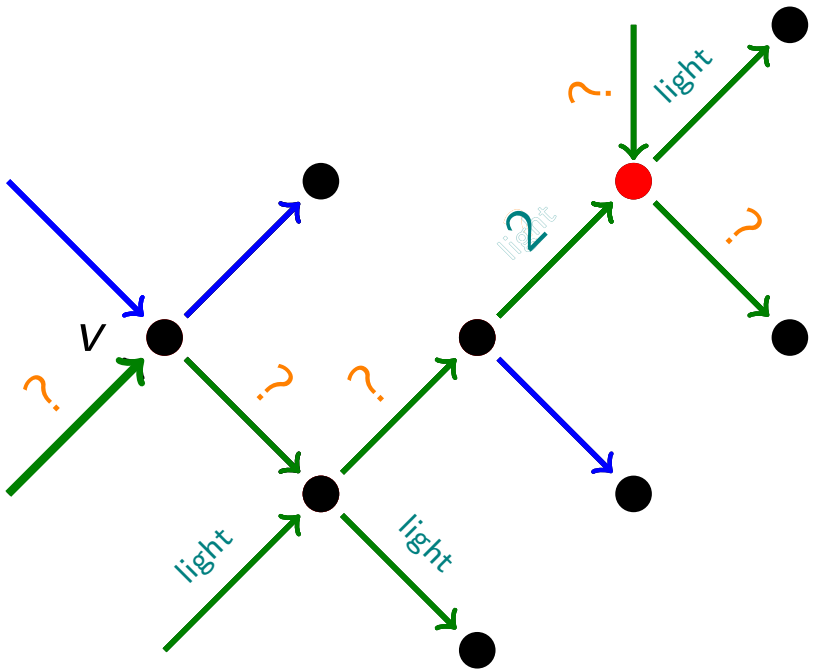
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

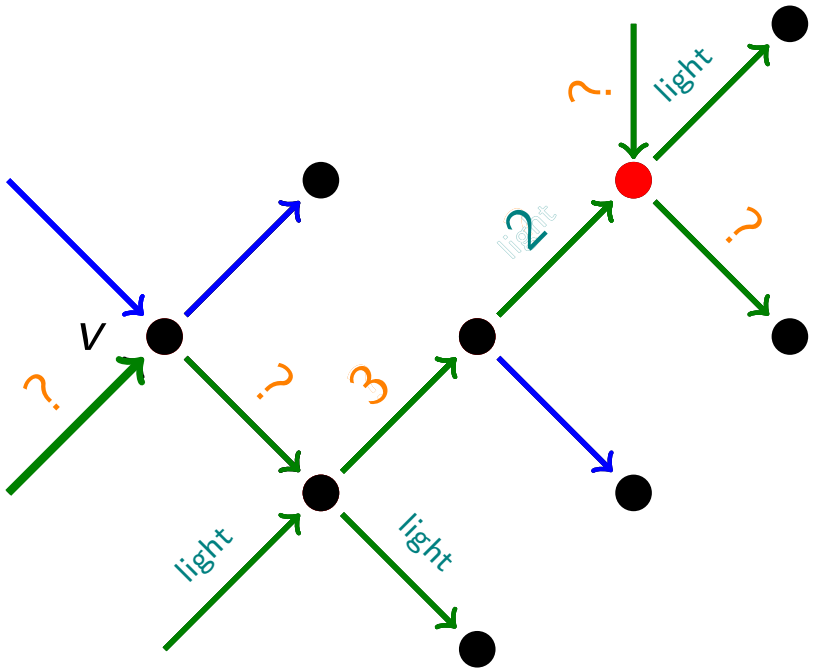
Algorithm computes the number of traversals for the traversed path.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

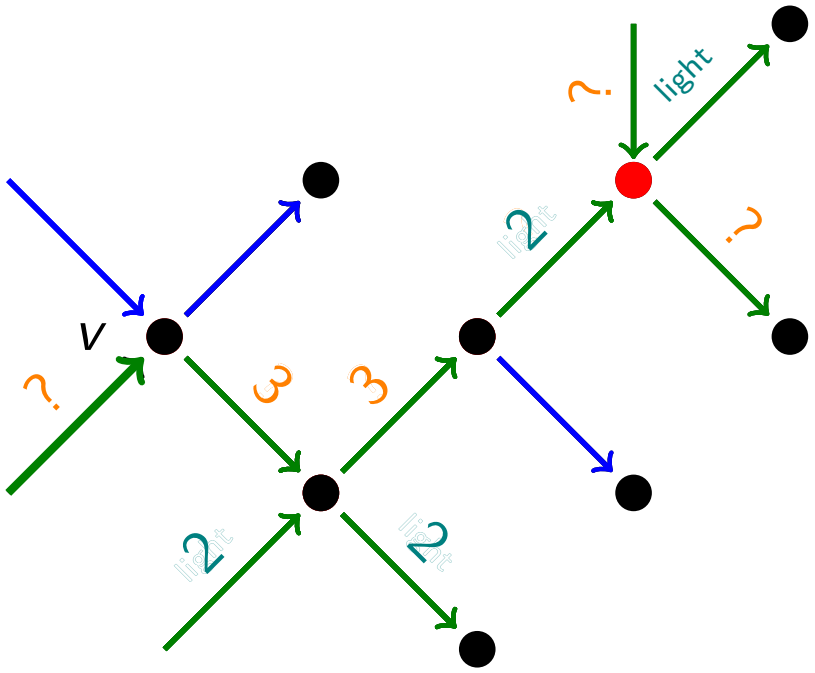
Algorithm computes the number of traversals for the traversed path.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

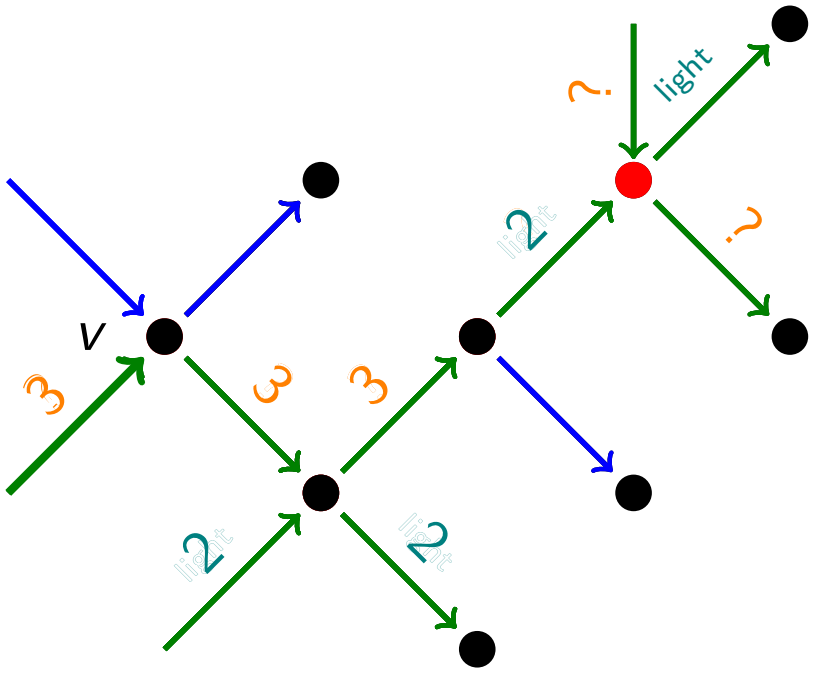
Algorithm computes the number of traversals for the traversed path.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

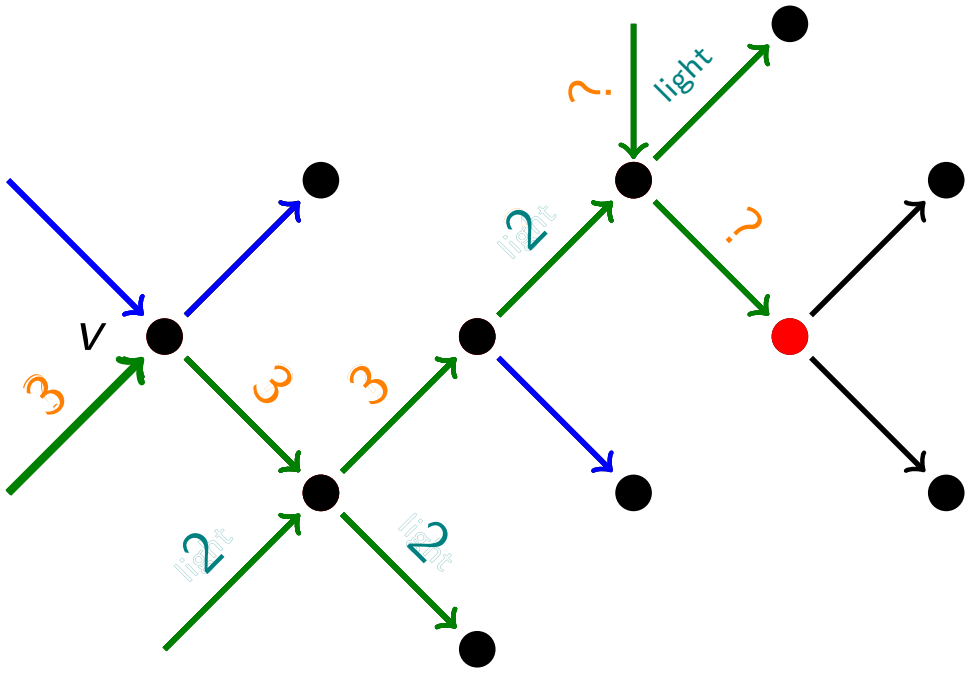
Algorithm computes the number of traversals for the traversed path.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

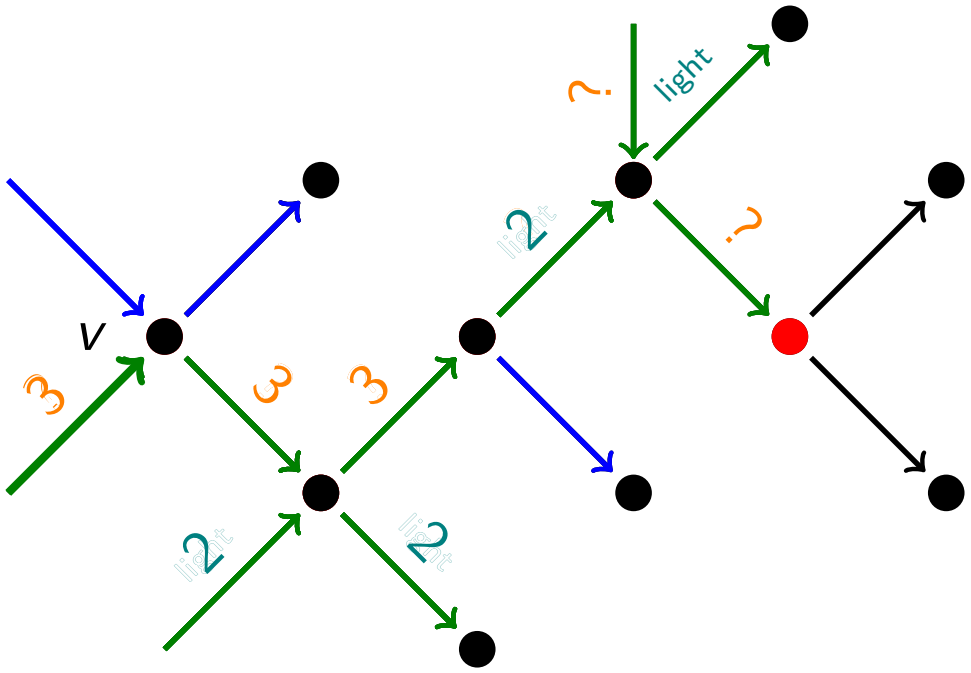
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

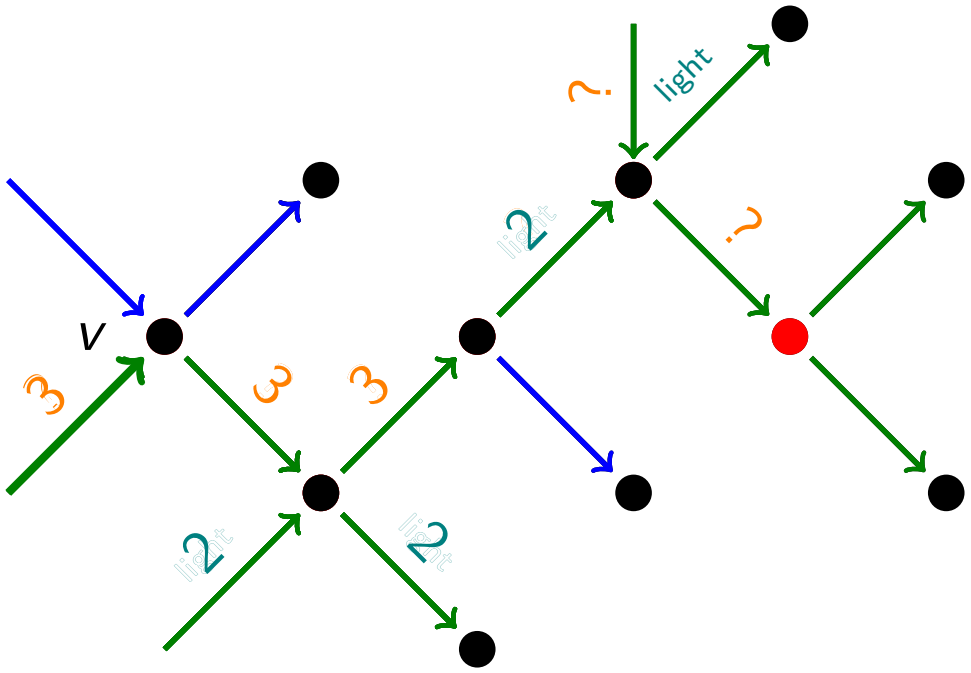
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

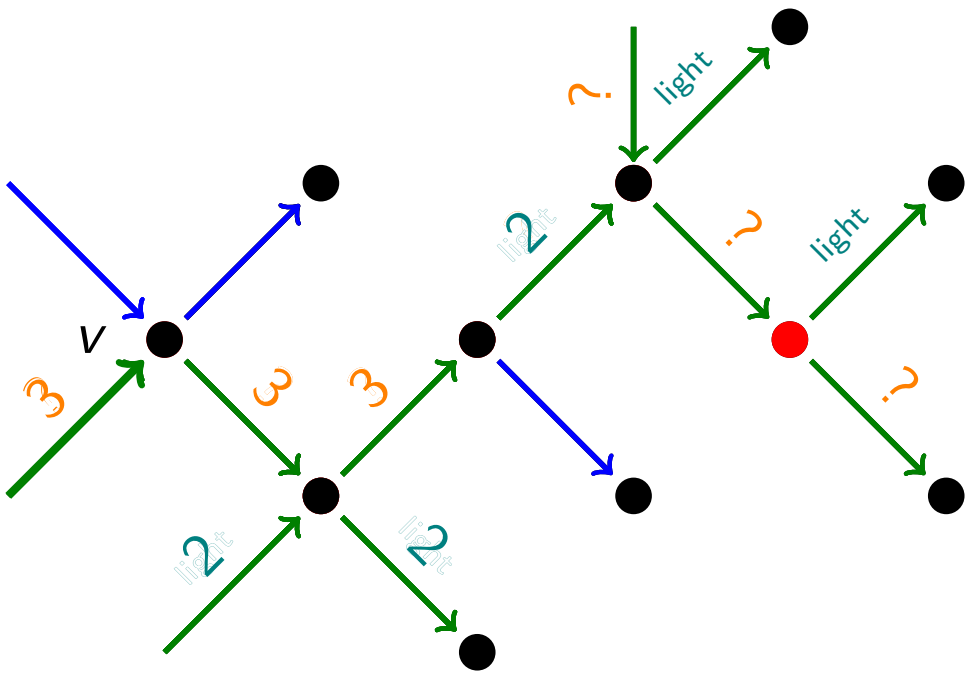
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

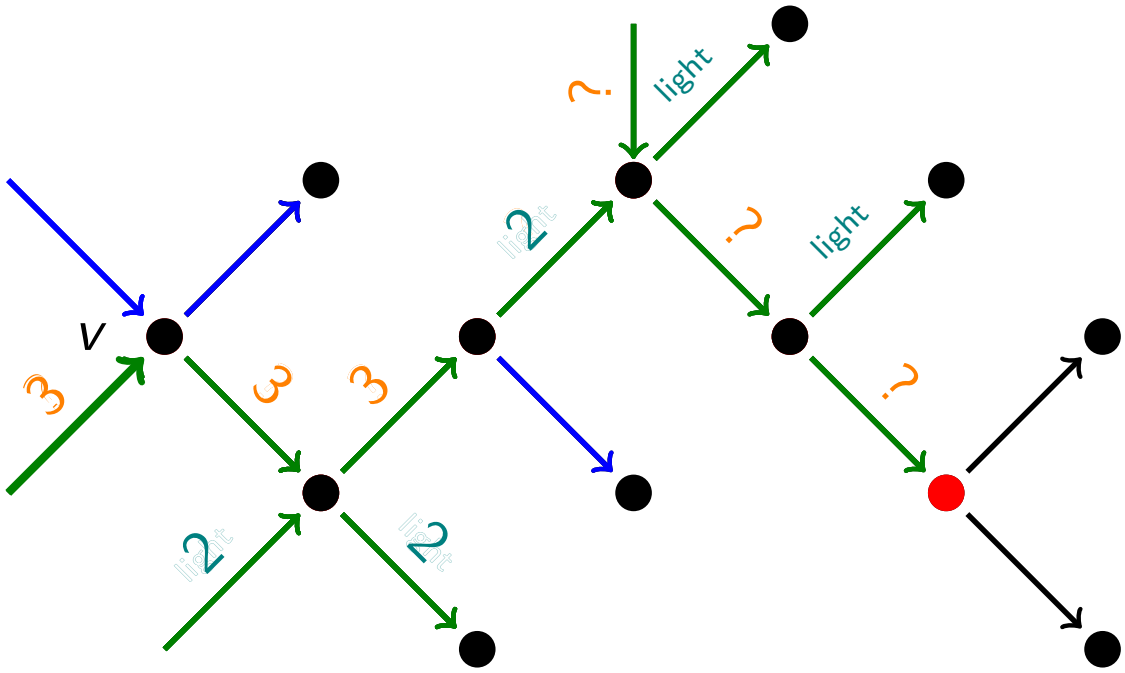
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

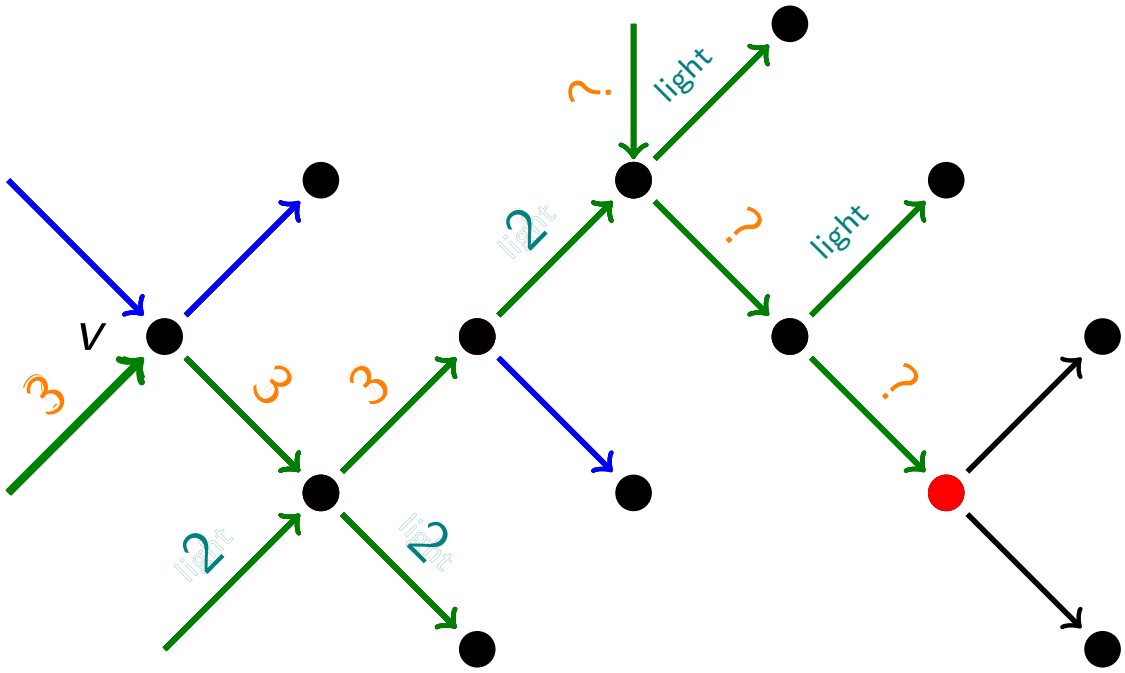
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

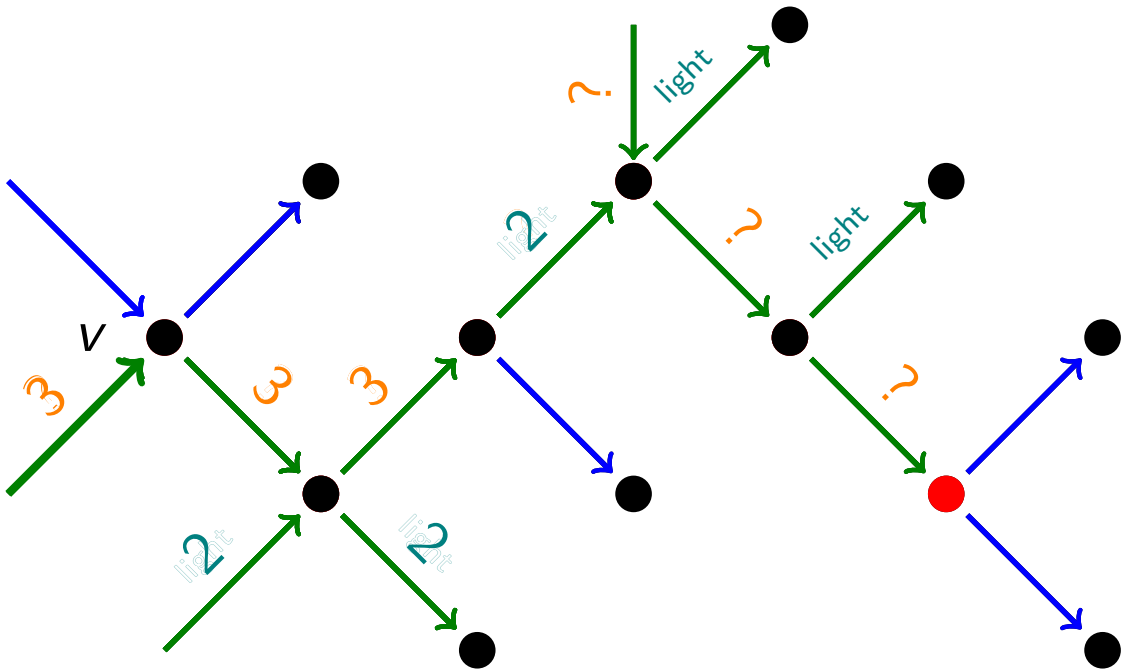
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

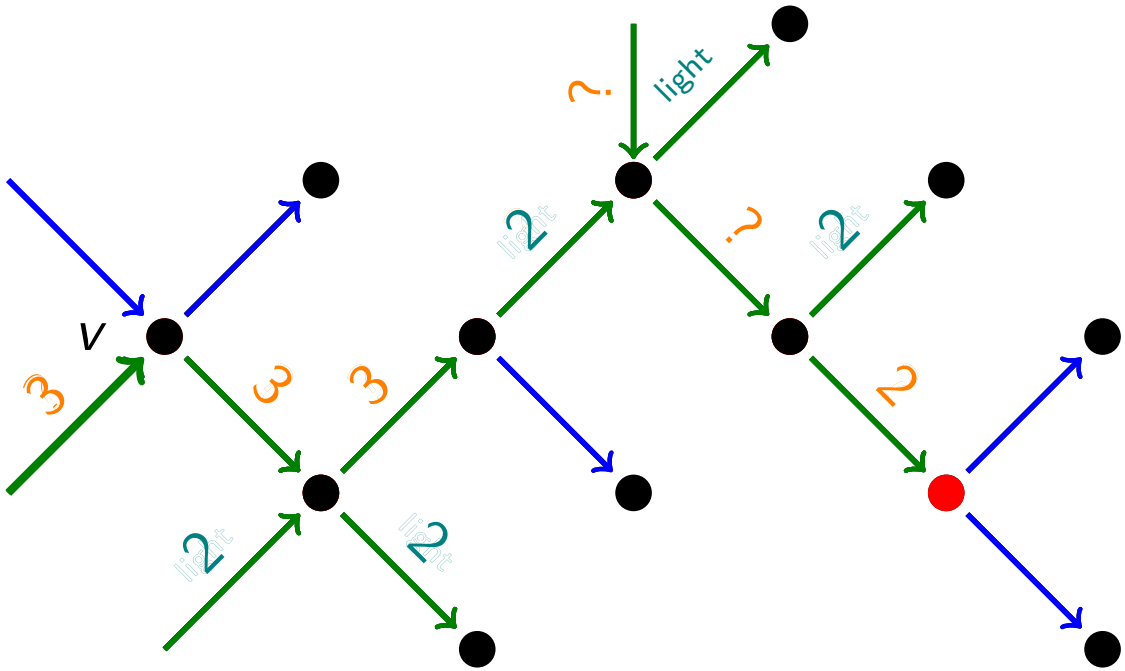
incoming edges? E_1, E_{Multi}? *heavy*, *light*? follows *heavy* edge.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

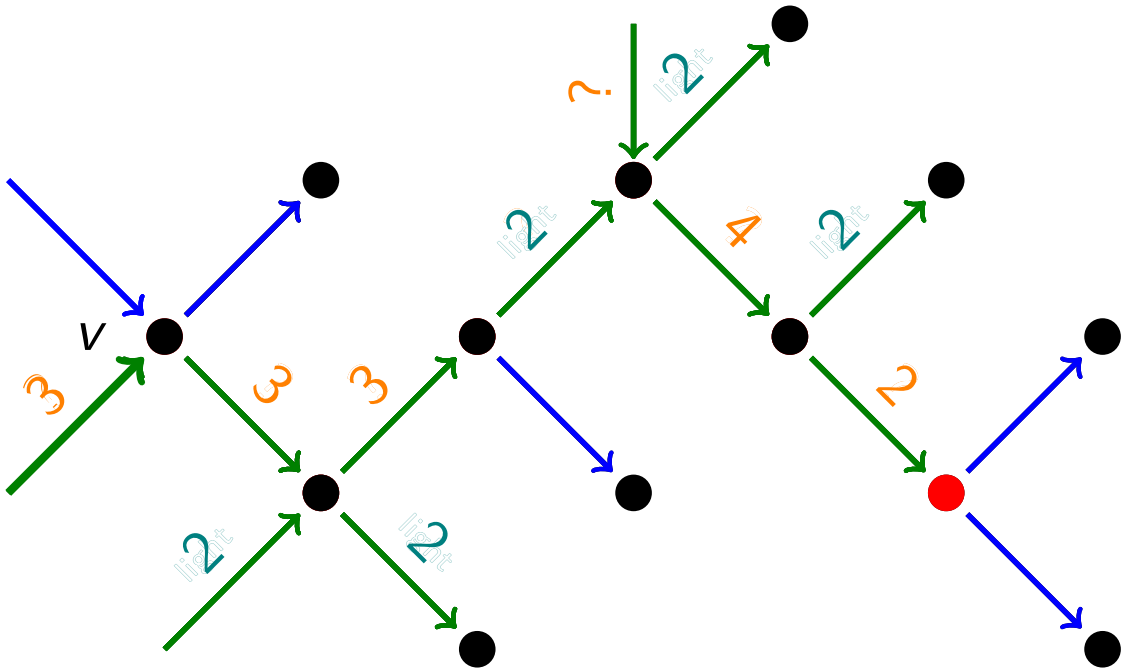
Algorithm computes the number of traversals for the traversed path.



Traversing the Edges from E_{Multi}

The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

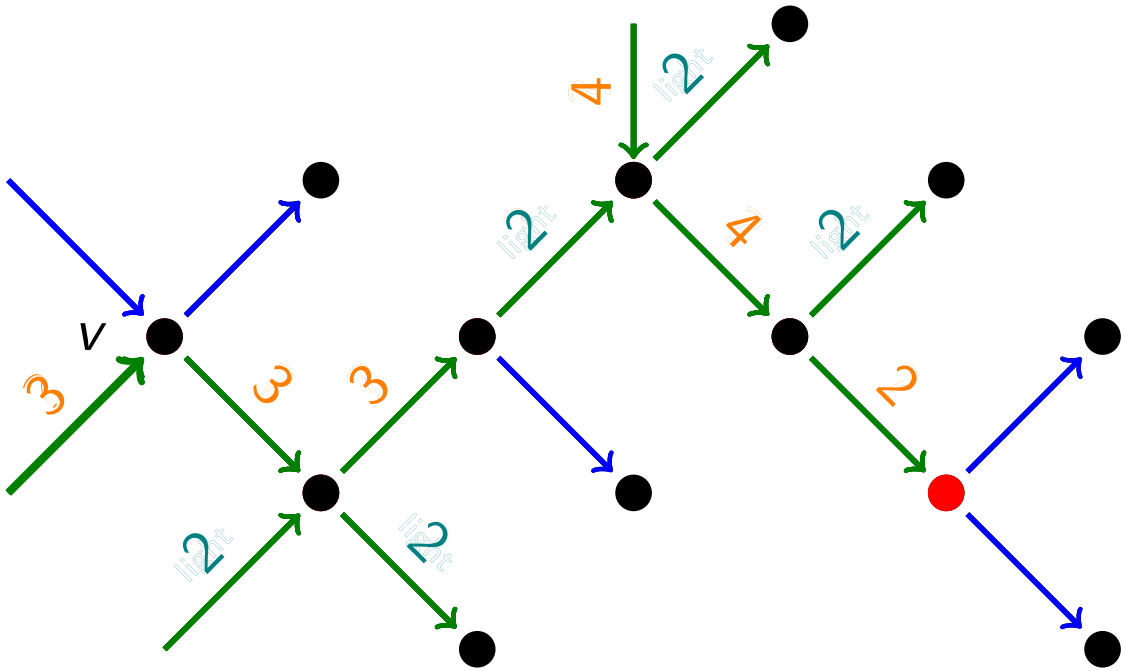
Algorithm computes the number of traversals for the traversed path.



Traversing the Edges from E_{Multi}

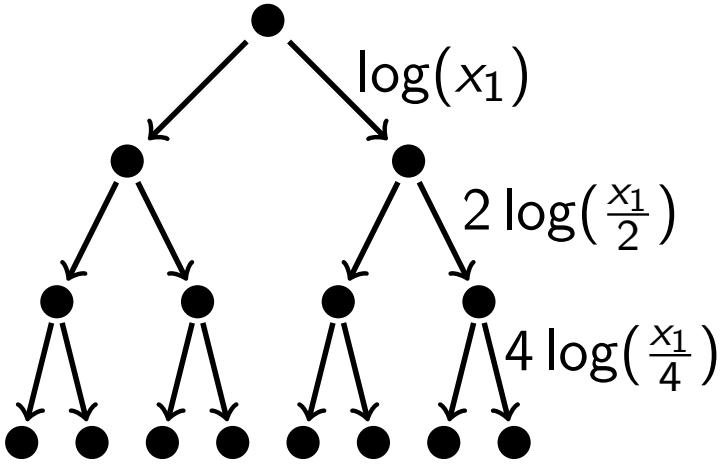
The algorithm follows the path of *heavy* edges.
(Relatively large and unknown number of traversals)

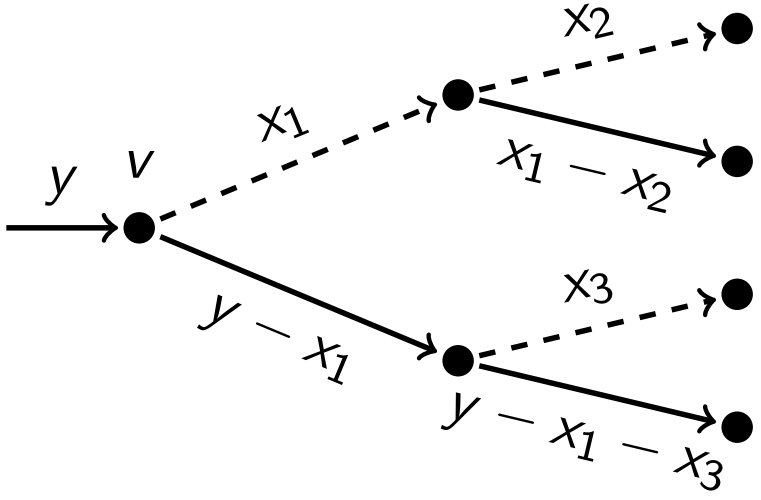
Algorithm computes the number of traversals for the traversed path.



Advice Bits for *light* Edges

For every pair of multi-edges, the algorithm asks for the *light* edge and its number of traversals.

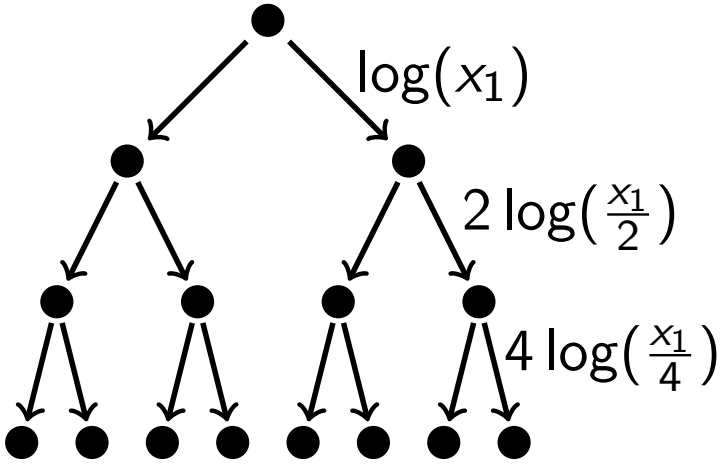


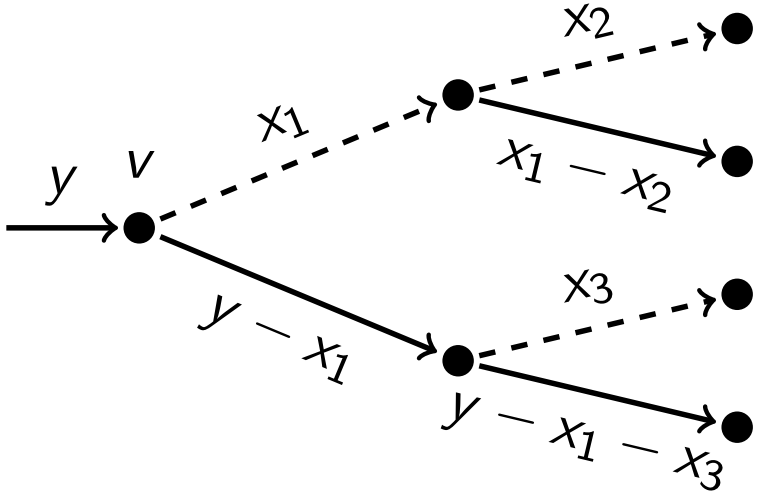


$$g(y) = \max_{2 \le x \le \frac{y}{2}} \{log(x) + 2 \log \log(x)\}$$

Advice Bits for *light* Edges

For every pair of multi-edges, the algorithm asks for the *light* edge and its number of traversals.
Repeatedly...

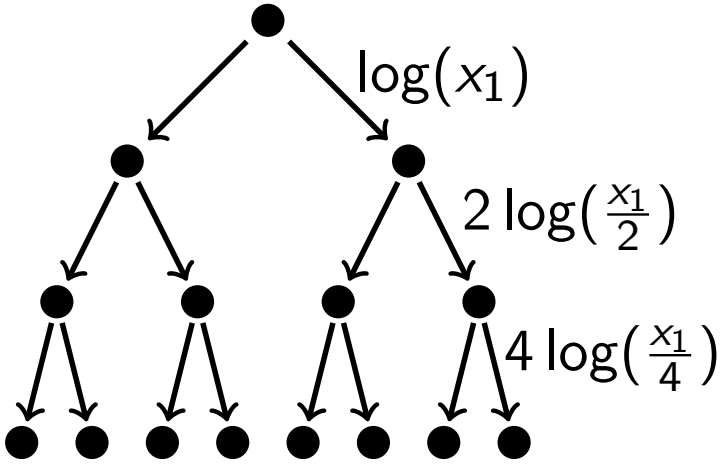


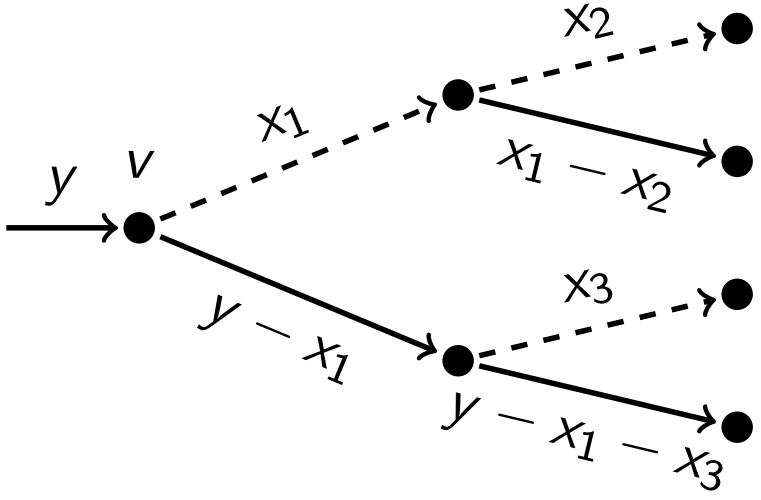


$$g(y) = \max_{2 \le x \le \frac{y}{2}} \{log(x) + 2 \log\log(x) + g(y-x) + g(x)\}$$

Advice Bits for *light* Edges

For every pair of multi-edges, the algorithm asks for the *light* edge and its number of traversals.
Repeatedly...





$$g(y) = \max_{2 \le x \le \frac{y}{2}} \{log(x) + 2 \log\log(x) + g(y-x) + g(x)\}$$

Per induction: $g(y) \le \frac{5}{2}y$

Summarizing the Advice Bits

- Incoming edges: n

Summarizing the Advice Bits

- Incoming edges: n
- Edges in E_0, E_1 or E_{Multi}: $\log(3) \cdot m$

Summarizing the Advice Bits

- Incoming edges: n
- Edges in E_0, E_1 or E_{Multi}: $\log(3) \cdot m$
- Last edges: n

Summarizing the Advice Bits

- Incoming edges: n
- Edges in E_0, E_1 or E_{Multi}: $\log(3) \cdot m$
- Last edges: n
- Multi-edges *heavy* or *light*: $2n$

Summarizing the Advice Bits

- Incoming edges: n
- Edges in E_0, E_1 or E_{Multi}: $\log(3) \cdot m$
- Last edges: n
- Multi-edges *heavy* or *light*: $2n$
- $\#_{S^*}$ for all *light* edges: $\sum_{T \in F} 2g(\max(\#_{S^*}))$

Summarizing the Advice Bits

- Incoming edges: n
- Edges in E_0, E_1 or E_{Multi}: $\log(3) \cdot m$
- Last edges: n
- Multi-edges *heavy* or *light*: $2n$
- $\#s^*$ for all *light* edges: $\sum_{T \in F} 2g(\max(\#s^*))$

$$\sum_{T \in F} 2g(\max(\#s^*) \leq \sum_{T \in F} 5 \max(\#s^*) \leq 5m$$

Summarizing the Advice Bits

- Incoming edges: n
- Edges in E_0, E_1 or E_{Multi}: $\log(3) \cdot m$
- Last edges: n
- Multi-edges *heavy* or *light*: $2n$
- $\#_{S^*}$ for all *light* edges: $\sum_{T \in F} 2g(\max(\#_{S^*}))$

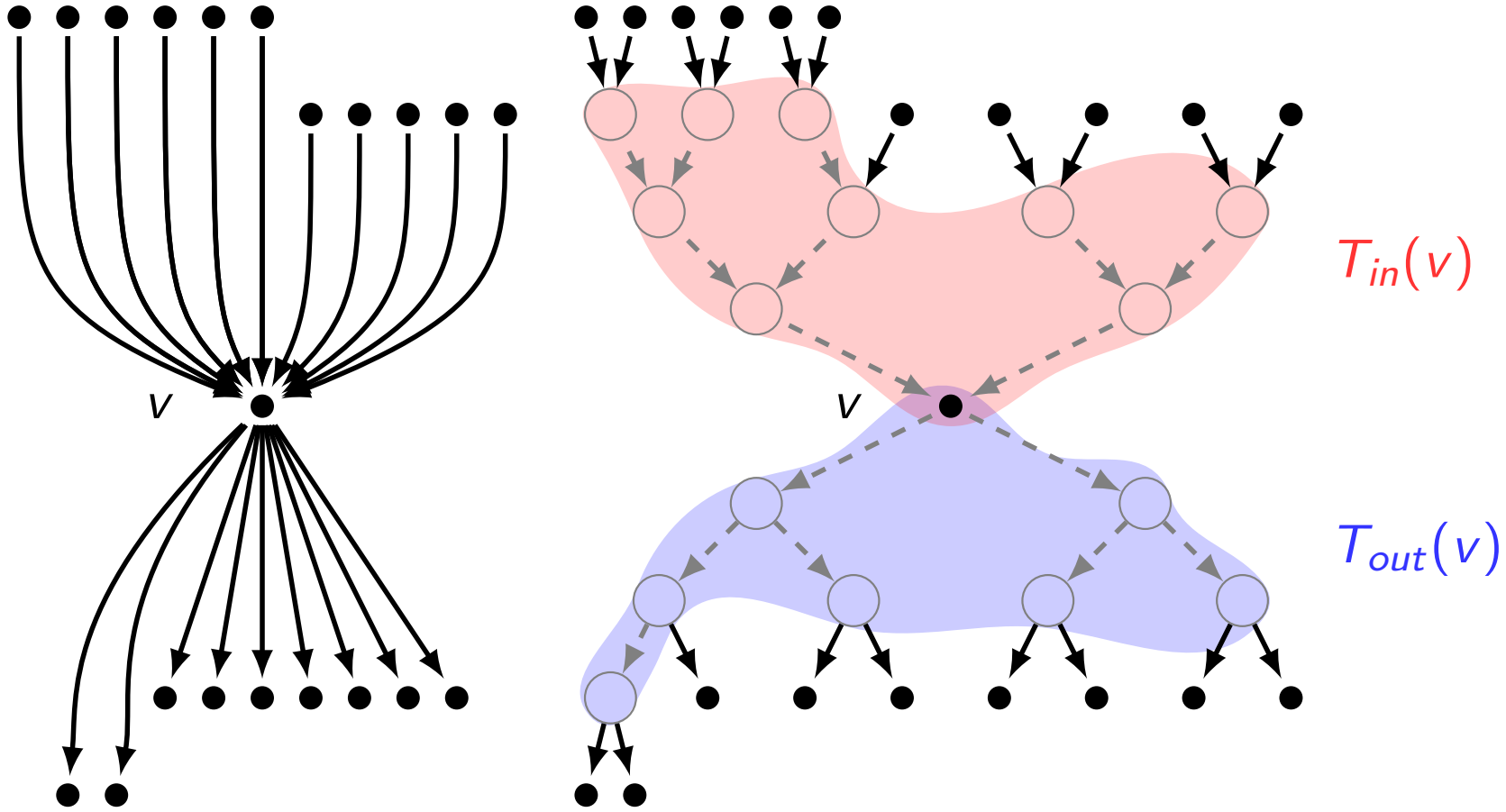
$$\sum_{T \in F} 2g(\max(\#_{S^*}) \leq \sum_{T \in F} 5 \max(\#_{S^*}) \leq 5m$$

Theorem

There exists an online algorithm which solves the graph exploration problem using $4n + (\log(3) + 5)m$ bits of advice on a given unknown directed graph $G = (V, E)$ with in- and outdegree bounded by 2.

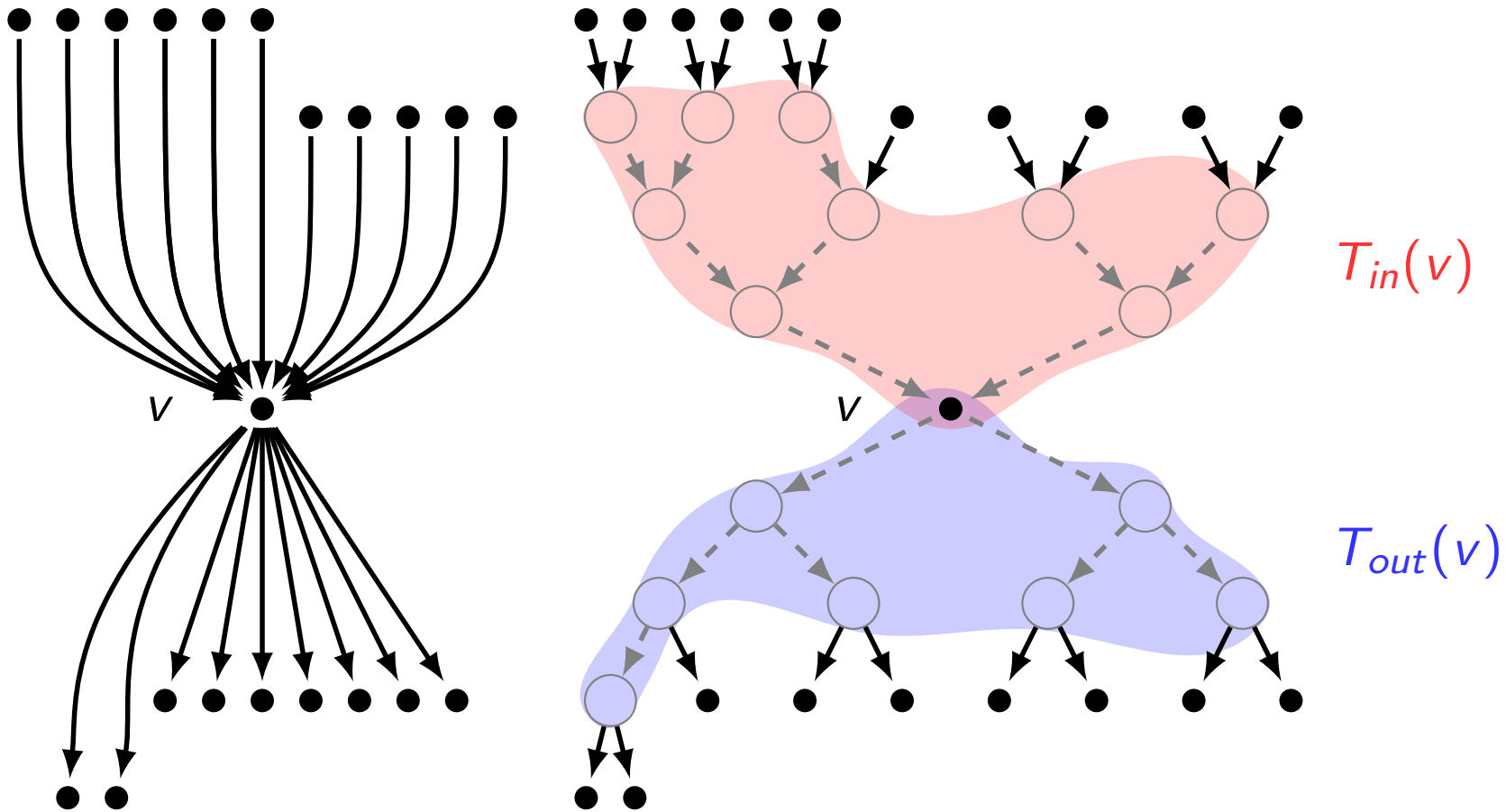
Unbounded Degree Graphs

Oracle and algorithm agree on a transformation of the graph.
Virtual vertices and edges are added to reduce the vertex degree.



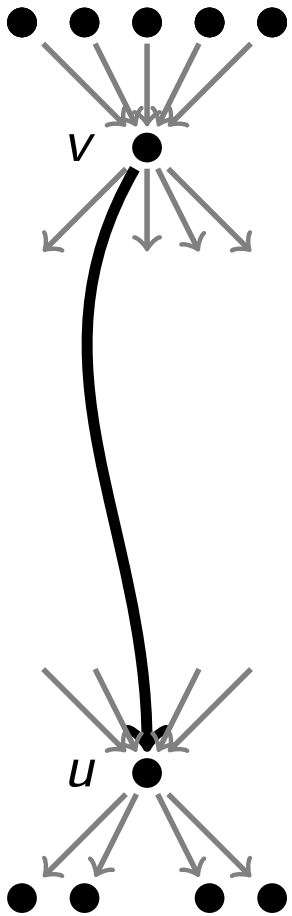
Unbounded Degree Graphs

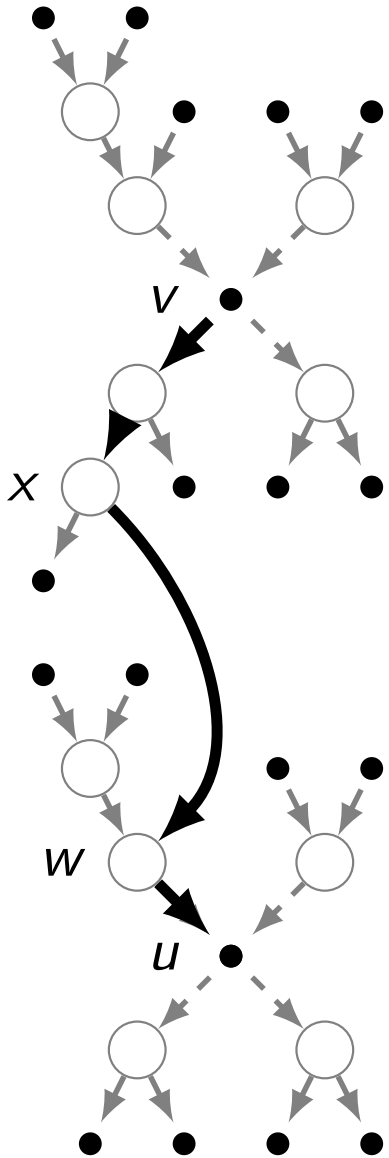
Oracle and algorithm agree on a transformation of the graph.
Virtual vertices and edges are added to reduce the vertex degree.



The virtual edges are always in E_{Multi}.

Edges after Transforming the Graph





Lower Bounds with Advice

A technique to prove lower bounds is the partition tree:

Try to find a family of instances $\mathcal{I}$ with the same prefix.
These instances should require different decisions to compute the optimal solution.

The algorithm should not be able to distinguish the instances until it is too late.

Lower Bounds with Advice

A technique to prove lower bounds is the partition tree:

Try to find a family of instances $\mathcal{I}$ with the same prefix.
These instances should require different decisions to compute the optimal solution.

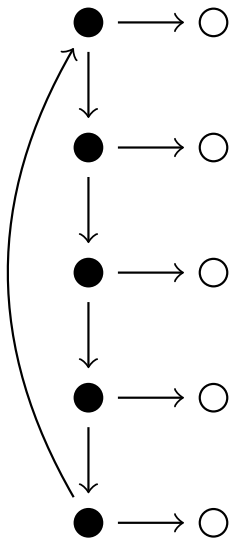
The algorithm should not be able to distinguish the instances until it is too late.

The main argument is that every instance needs a different advice string.
Thus, every algorithm needs at least

$$|\mathcal{I}| = 2^{\log_2(|\mathcal{I}|)} > 2^{b(|\mathcal{I}|)}.$$

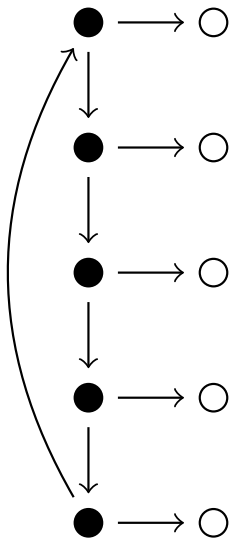
Construction

An instance from $\mathcal{I}$ consists of two parts.
Part A consists of cycles with length $x = 5$.



Construction

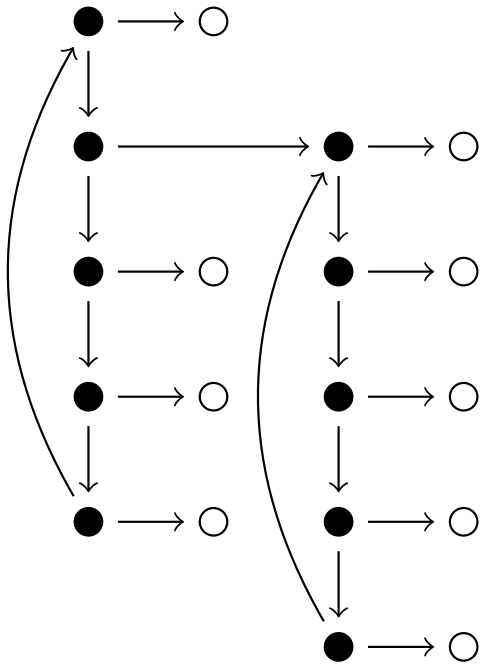
An instance from $\mathcal{I}$ consists of two parts.
Part A consists of cycles with length $x = 5$.



There is only one correct exit into the next cycle.

Construction

An instance from $\mathcal{I}$ consists of two parts.
Part A consists of cycles with length $x = 5$.



There is only one correct exit into the next cycle.
White vertices are for the second part.

Numbers for the First Part

For each of the k cycles, we need to find the cycle and the exit:

$$k\left(x - 1 + \log\left(x\right)\right)$$

Numbers for the First Part

For each of the k cycles, we need to find the cycle and the exit:

$$k\left(x-1+\log\left(x\right)\right)$$

Each cycle generates $x-1$ white vertices.
The last cycle generates x white vertices.

The size of second part B is depends on the parameters of part A:

$$|B| = k(x-1)+1$$

Numbers for the First Part

For each of the k cycles, we need to find the cycle and the exit:

$$k\left(x-1+\log\left(x\right)\right)$$

Each cycle generates $x-1$ white vertices.
The last cycle generates x white vertices.

The size of second part B is depends on the parameters of part A:

$$|B| = k(x-1)+1$$

It is important that this number is a power of two:

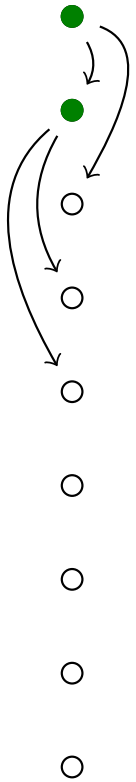
$$|B| = 2^z + 1$$

Second Part

It is important that all choices look the same to give no information to the algorithm.

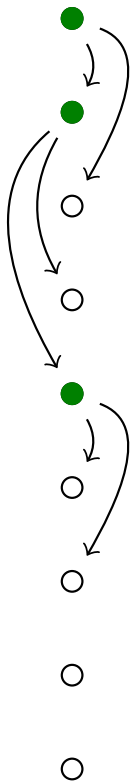
Second Part

It is important that all choices look the same to give no information to the algorithm.



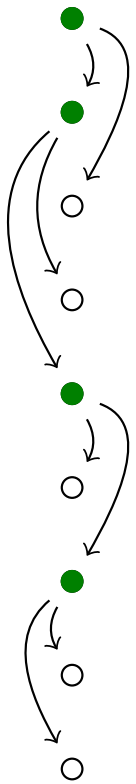
Second Part

It is important that all choices look the same to give no information to the algorithm.



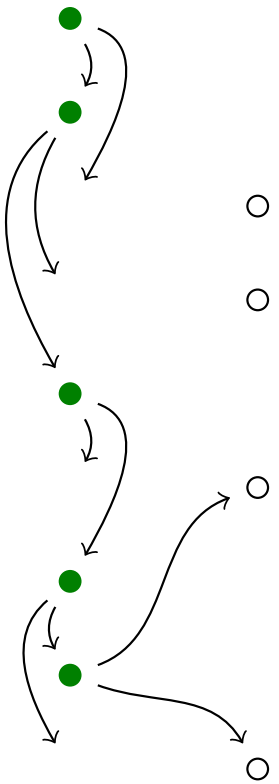
Second Part

It is important that all choices look the same to give no information to the algorithm.



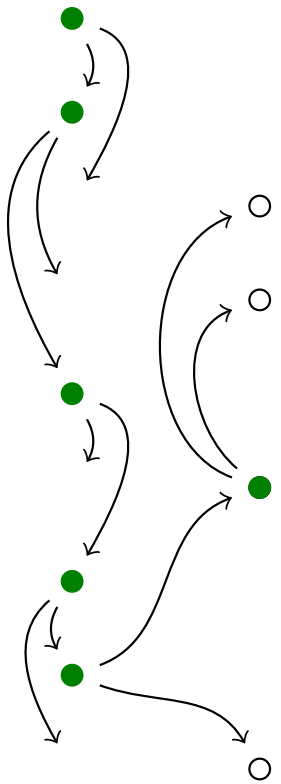
Second Part

It is important that all choices look the same to give no information to the algorithm.



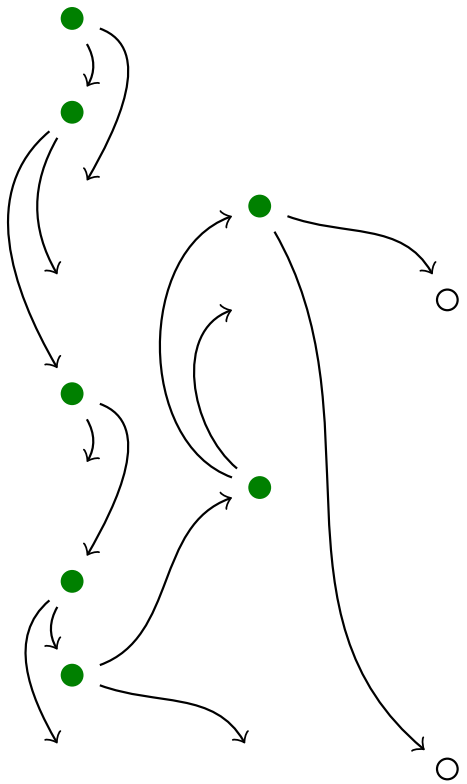
Second Part

It is important that all choices look the same to give no information to the algorithm.



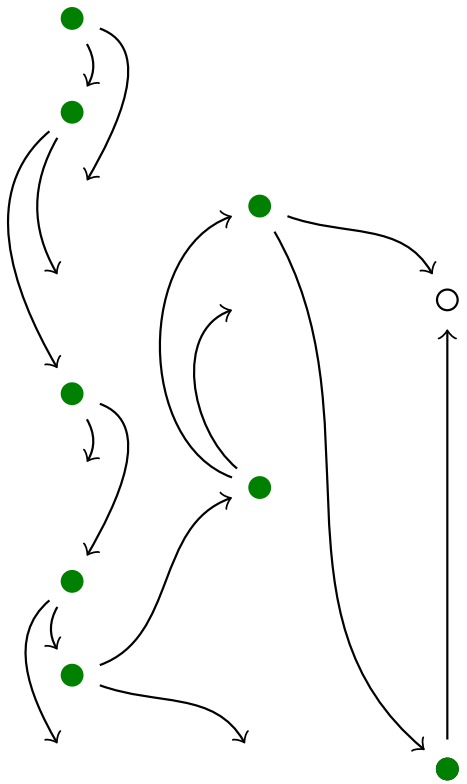
Second Part

It is important that all choices look the same to give no information to the algorithm.



Second Part

It is important that all choices look the same to give no information to the algorithm.



For each vertex we have a binary decision (not the last one)

Advice complexity

The number of vertices in part A and B is:

$$n = kx + k(x - 1) + 1$$

Advice complexity

The number of vertices in part A and B is:

$$n = kx + k(x-1) + 1$$

The number of needed advice for part A and B is:

$$k(x - 1 + \log(x)) + k(x-1) = k(2x + \log(x) - 2)$$

Advice complexity

The number of vertices in part A and B is:

$$n = kx + k(x-1) + 1 = 2kx - k + 1$$

The number of needed advice for part A and B is:

$$k(x - 1 + \log(x)) + k(x-1) = k(2x + \log(x) - 2)$$

For $\log(x) \geq 2$ we can make the following estimation:

$$k(2x + \log(x) - 2) \geq 2kx$$

Advice complexity

The number of vertices in part A and B is:

$$n = kx + k(x-1) + 1 = 2kx - k + 1$$

The number of needed advice for part A and B is:

$$k(x - 1 + \log(x)) + k(x-1) = k(2x + \log(x) - 2)$$

For $\log(x) \geq 2$ we can make the following estimation:

$$k(2x + \log(x) - 2) \geq 2kx$$

Thus, we need slightly more than n bits of advice to solve the lower bound construction optimal.

Results

$n \log(n)$ is the known upper bound for the advice complexity.

Improvement for sparse graphs:

graph model	**exploration**	**upper bound**
directed	cyclic	$2n + 23m$
directed	path	$2n + 23m + \lceil \log(n) \rceil$
undirected	cyclic	$\log(6)(n + m) + 42m$
undirected	path	$\log(6)(n + m) + 42m + \lceil \log(n) \rceil$

The input graph can be weighted or unweighted.
The algorithm works mainly with the number of traversals.

Results

$n \log(n)$ is the known upper bound for the advice complexity.

Improvement for sparse graphs:

graph model	**exploration**	**upper bound**
directed	cyclic	$2n + 23m$
directed	path	$2n + 23m + \lceil \log(n) \rceil$
undirected	cyclic	$\log(6)(n + m) + 42m$
undirected	path	$\log(6)(n + m) + 42m + \lceil \log(n) \rceil$

The input graph can be weighted or unweighted.
The algorithm works mainly with the number of traversals.

The lower bound is slightly larger than n.

Future Work

The newest result:

Theorem

There exists an online algorithm with advice that uses $5.25n$ bits of advice to solve the graph exploration problem on directed graphs with a bounded outdegree of two.

The lower bound construction is a directed graphs with a bounded outdegree of two.

Future Work

The newest result:

Theorem

There exists an online algorithm with advice that uses $5.25n$ bits of advice to solve the graph exploration problem on directed graphs with a bounded outdegree of two.

The lower bound construction is a directed graphs with a bounded outdegree of two.

Thanks for your attention!