~~(本会議にて公開を予定)~~

グラフ理論と ネットワークのアルゴリズムの基礎

2013/11/28(木)

伊波 源太 (Genta IHA) <genta@ate-mahoroba.jp>

(株)まほろば工房 主幹工芸士

Agenda

1. 導入
2. 計算量 (オーダ記法)
3. 最短経路問題 (ダイクストラ法)
4. 最小木問題 (プリム法)

導入 – なぜ、グラフ理論か?

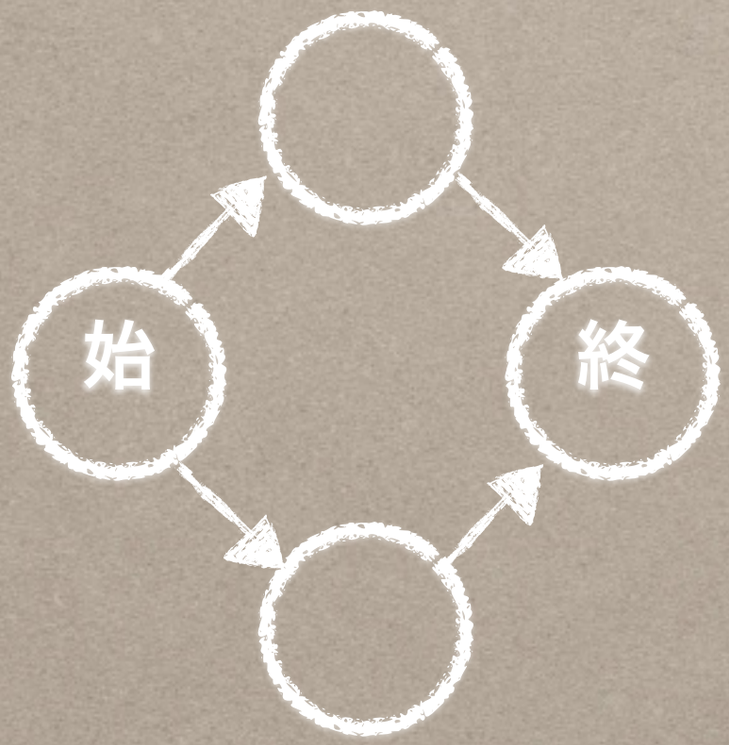
- 動機: 人それぞれ?
  - ルータの気持ちになって考えてみたい
  - 勉強してみたい
  - もし世界が滅亡しても、自力でInternetを再生したい
- 自力で問題を解けることが重要!
- 最短経路問題、最小木問題、計算量

最短経路の計算 (素朴な方法)

- ネットワークの最短経路を計算する
- 全ての組み合わせを列挙して、
  - 始点、終点を通るものを抽出し、
  - そのなかで、最もコストが低いものを選ぶ
- まったく簡単だ
- 複雑なアルゴリズムなんて必要ないのでは?

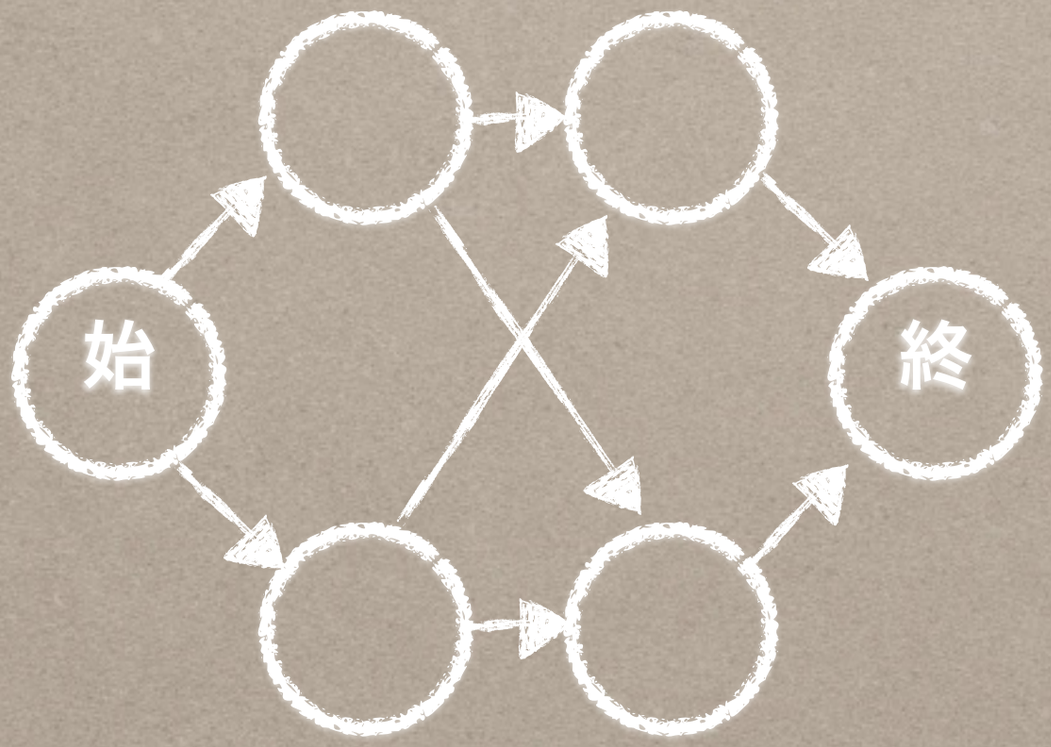
計算してみよう! (1/5)

- ノード数4の場合の例: 全経路は2通り



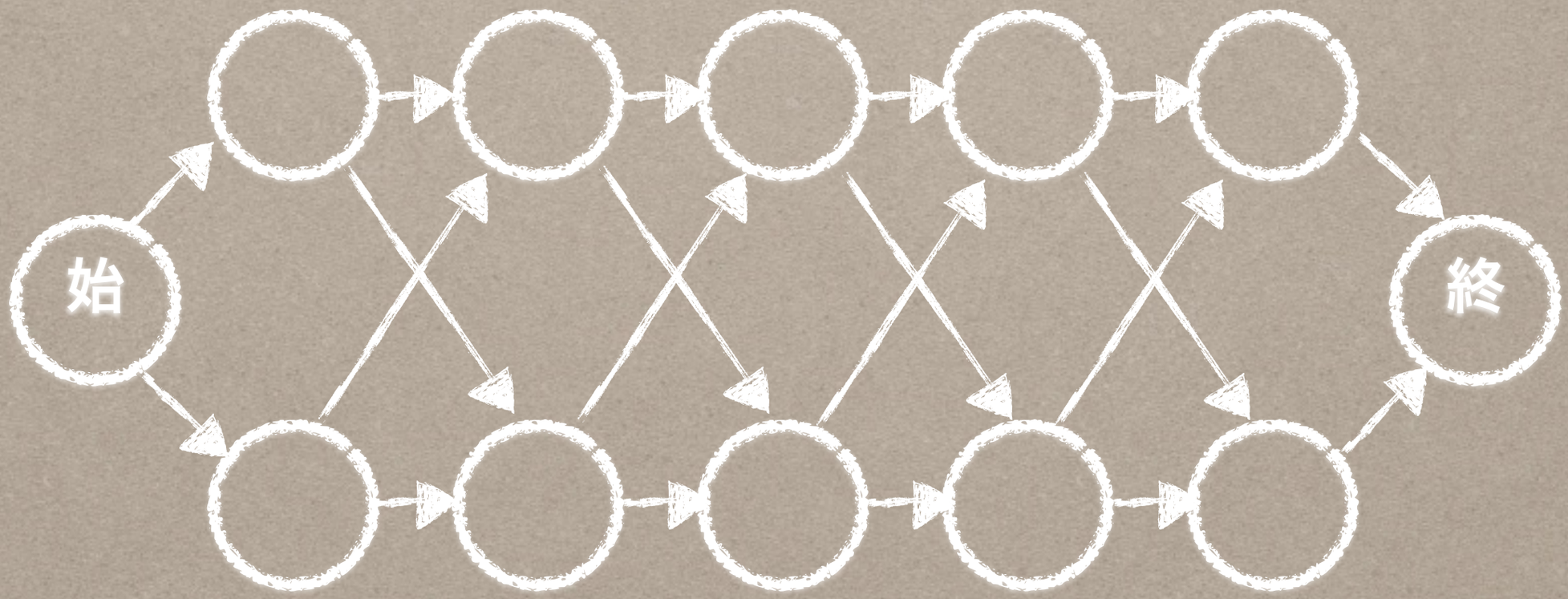
計算してみよう! (2/5)

- ノード数6の場合の例: 全経路は4通り (2^2通り)



計算してみよう! (3/5)

- ノード数12の場合の例: 全経路は32通り (2^5通り)



計算してみよう! (4/5)

- ノード数22の場合: 2^{10}通り (=1,024通り)
- ノード数42の場合: 2^{20}通り (=1,048,576通り)
- ノード数102の場合: 2^{50}通り

  (= 1,125,899,906,842,624通り)
- ノード数202の場合: 2^{100}通り

  (=1,267,650,600,228,229,401,496,703,205,376通り)

計算してみよう! (5/5)

(本会議にて公開を予定)

- 適切なアルゴリズムが必要

Agenda

1. 導入
2. 計算量 (オーダ記法)
3. 最短経路問題 (ダイクストラ法)
4. 最小木問題 (プリム法)

2. 計算量 (オーダ記法)

- 実際に計算をはじめる前に、規模見積もりをすることができる
  - どのくらいの処理時間がかかるか (時間計算量)
  - どのくらいメモリが必要か (空間計算量)
- 入力するデータの量と、アルゴリズムによって見積もることができる

時間計算量

- 計算量の「増え方」の傾向を表す(オーダ記法)
  - O(1)
  - O(log n) – 平衡二分木の探索
  - O(n) – 配列の線形探索など
  - $O(n^2)$ – ダイクストラ法, プリム法
  - $O(2^n)$ – グラフの全数探索の例
- 計算量の「増え方」に着目する
  - 係数は無視してよい (O(2n) = O(n))
  - nの規模が大きくなってきたときの規模に着目 (O(1) + O(n) = O(n))

Agenda

1. 導入
2. 計算量 (オーダ記法)
3. 最短経路問題 (ダイクストラ法)
4. 最小木問題 (プリム法)

3. 最短経路問題 (ダイクストラ法)

- ダイクストラ法でできること
  - グラフ、始点ノードが与えられた時、始点ノードから全ノードに対する「コストが最小になるような路(Path)」を求めることができる
- ダイクストラ法の利用シーン
  - 始点ノードから、すべてのノードへの最短経路(コスト最小な路)を求める
  - OSPF, STP, etc...
- 有向グラフを例に説明します
  - ダイクストラ法は無向グラフにも適用可能です

ダイクストラ法 (概要)

入力: (有向グラフ, 始点ノード)

(1) 初期化する (グラフを初期状態にする)

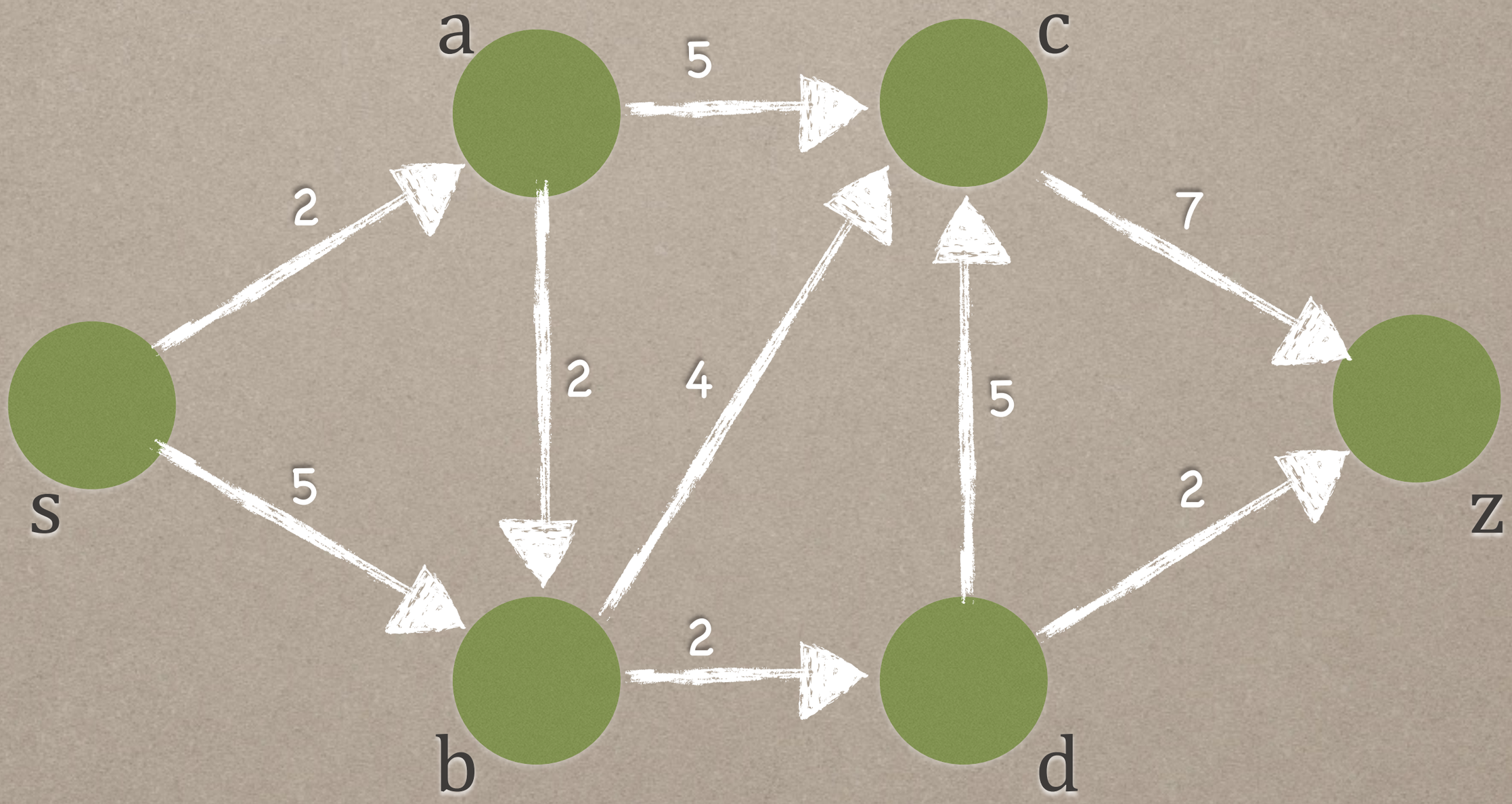
(2) 「未処理」のノードのうち、最小コストのノードに注目する

(3) 隣接ノードのコストをアップデートする

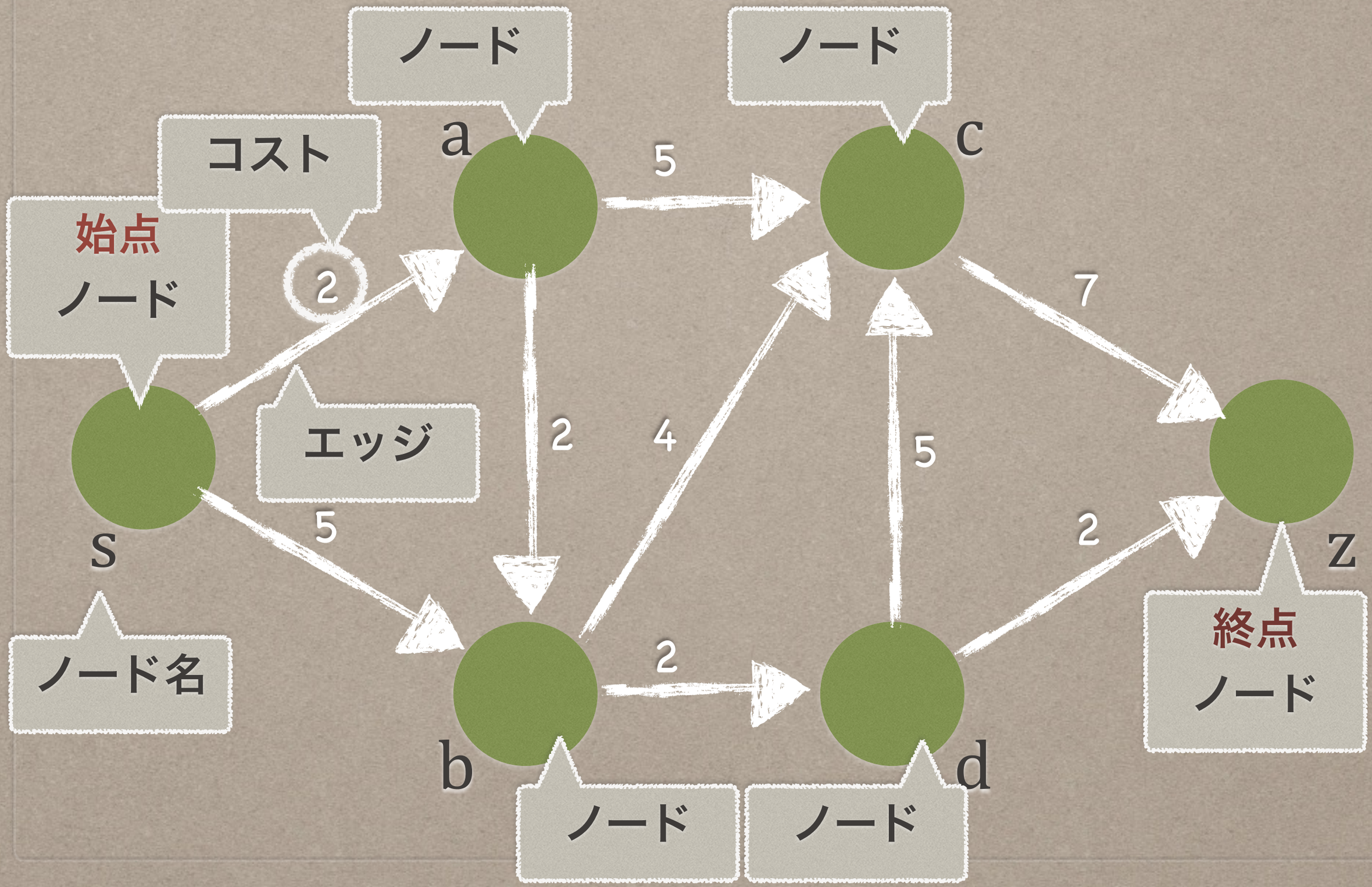
(4) 「未処理」ノードが無くなるまで、(2)〜(3)を繰り返す

(5) 最短経路を出力する

ダイクストラ法 (イメージ) 1/16

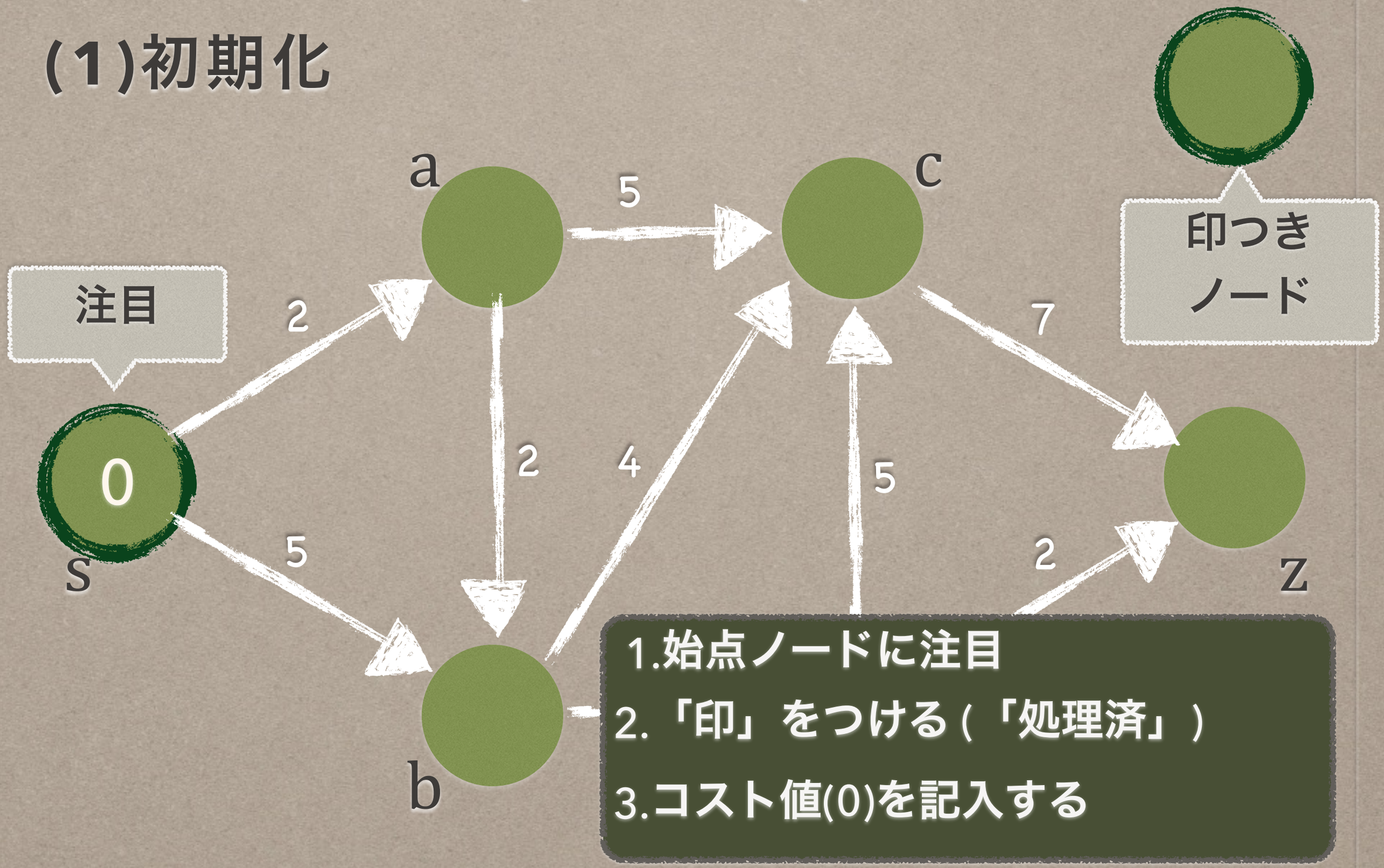


ダイクストラ法 (イメージ) 2/16



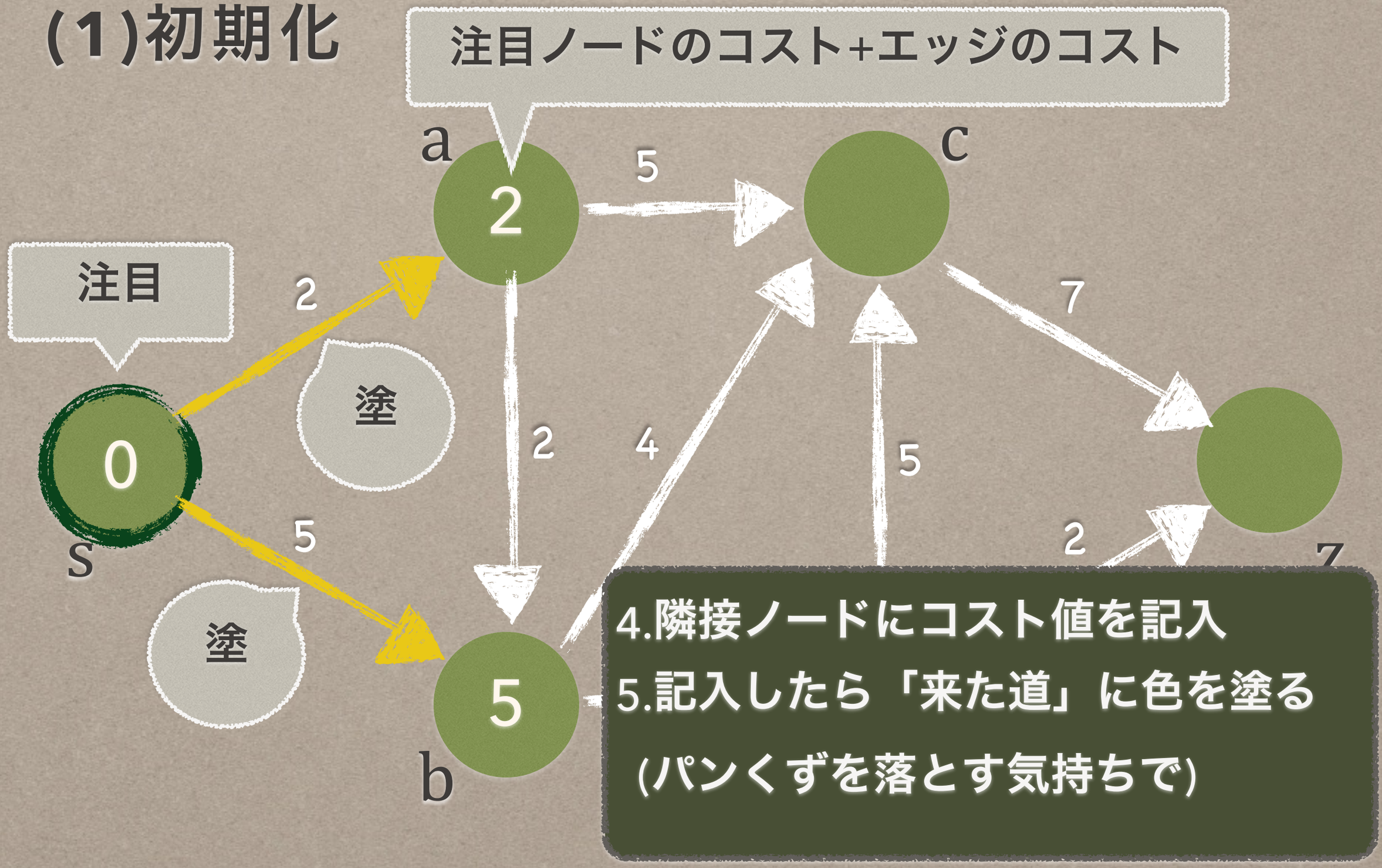
ダイクストラ法 (イメージ) 3/16

(1)初期化



ダイクストラ法 (イメージ) 4/16

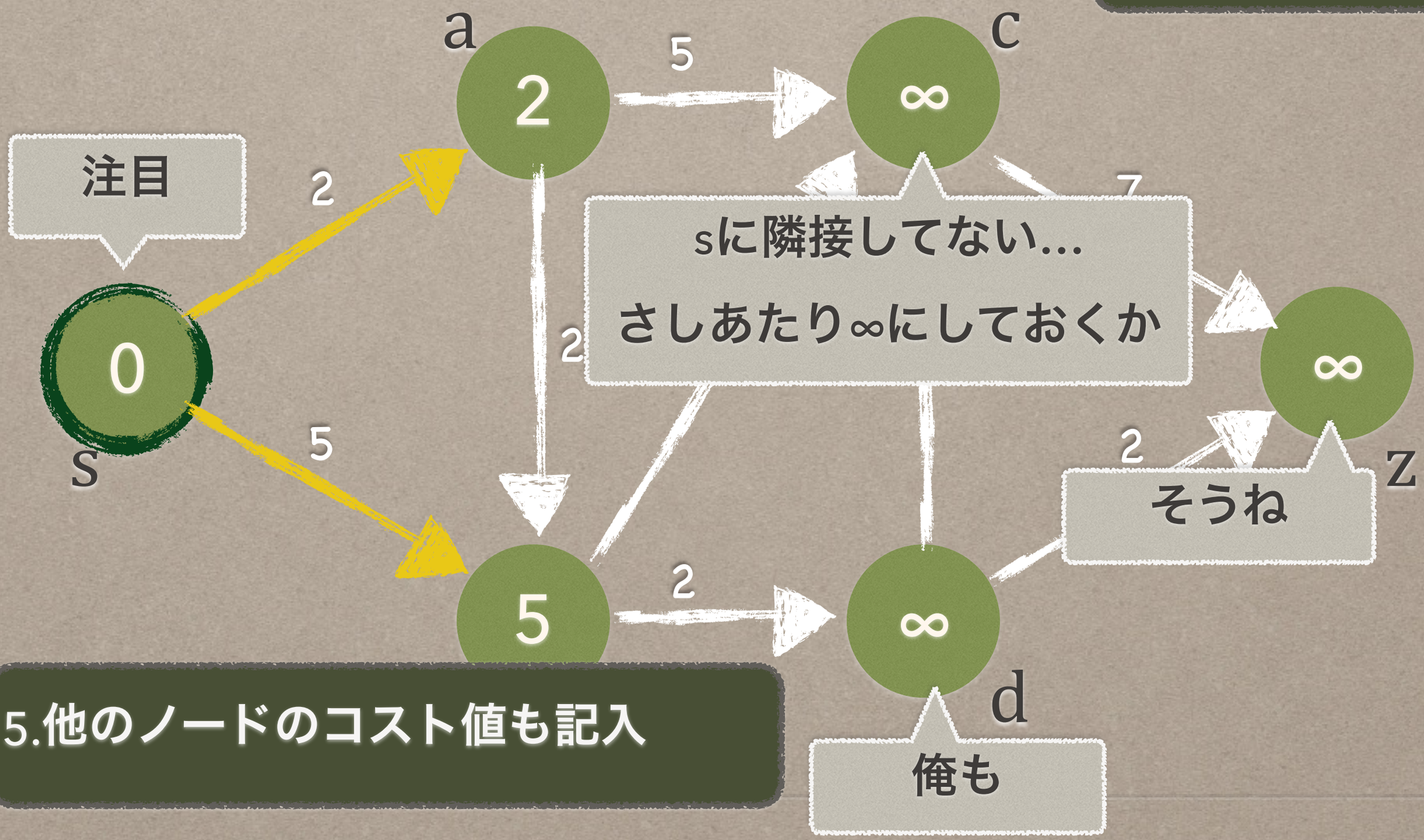
(1)初期化



ダイクストラ法 (イメージ) 5/16

(1)初期化

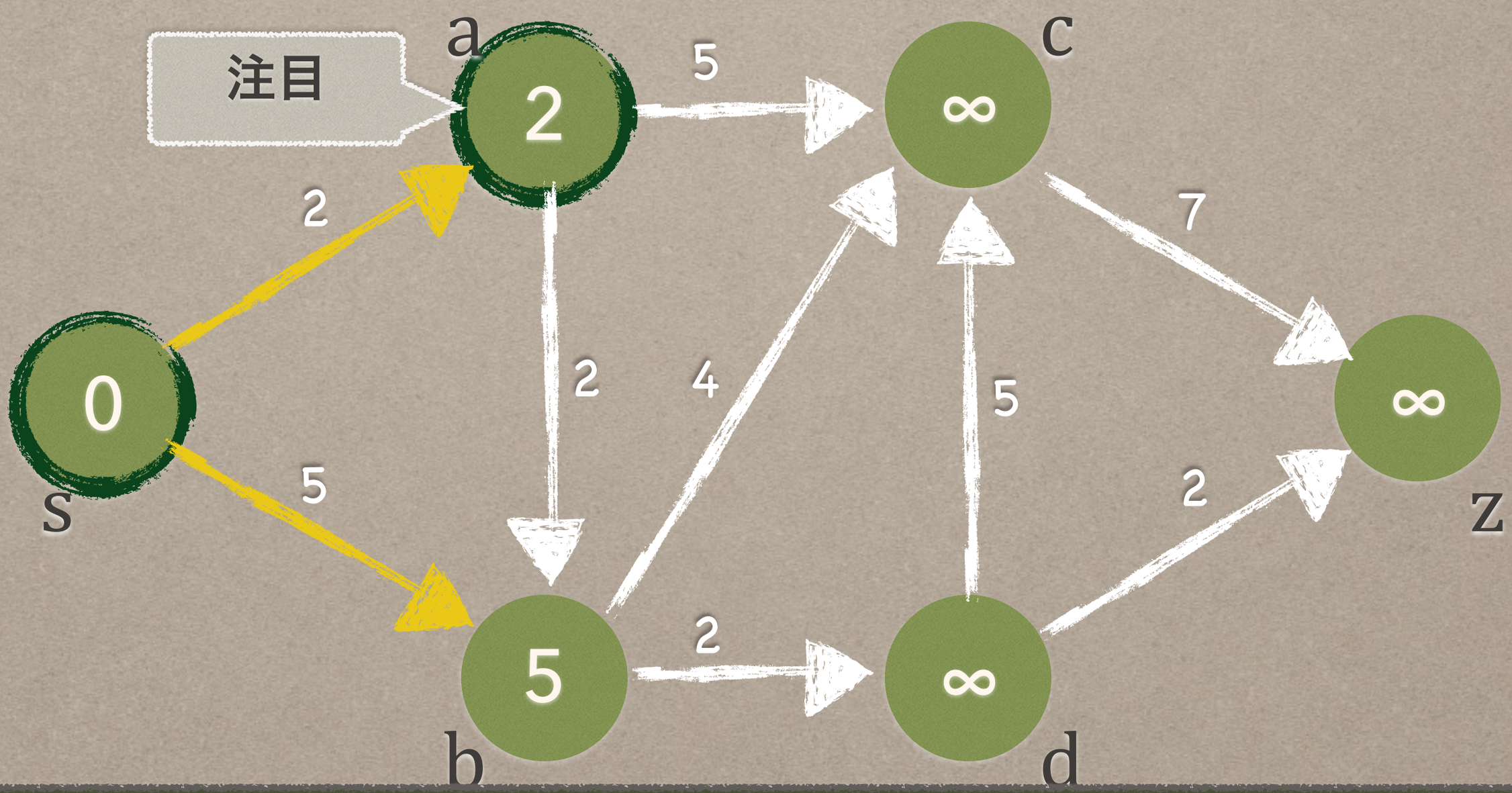
初期化は完了!



5.他のノードのコスト値も記入

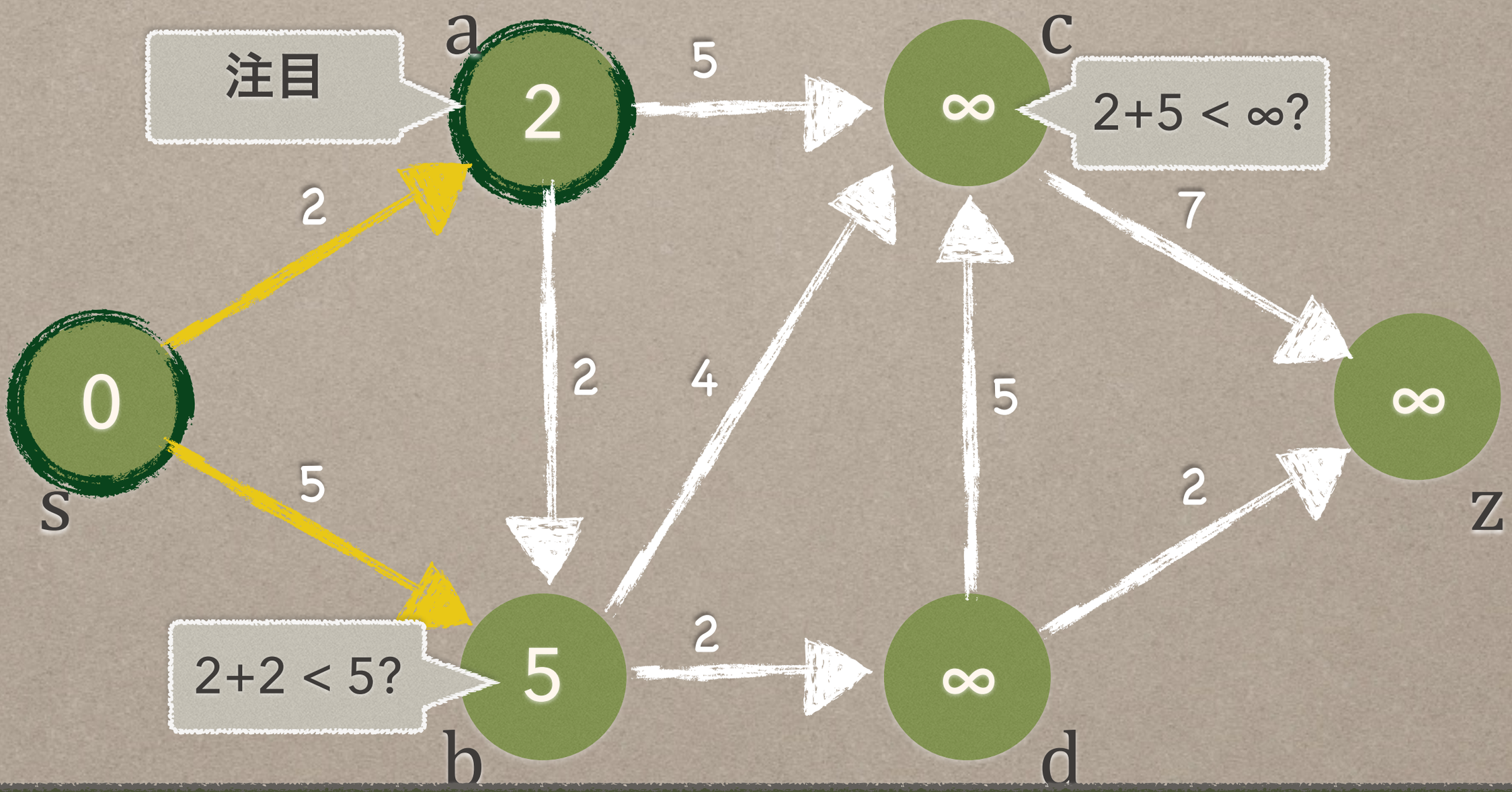
ダイクストラ法 (イメージ) 6/16

(2)印なし最小ノードを選択



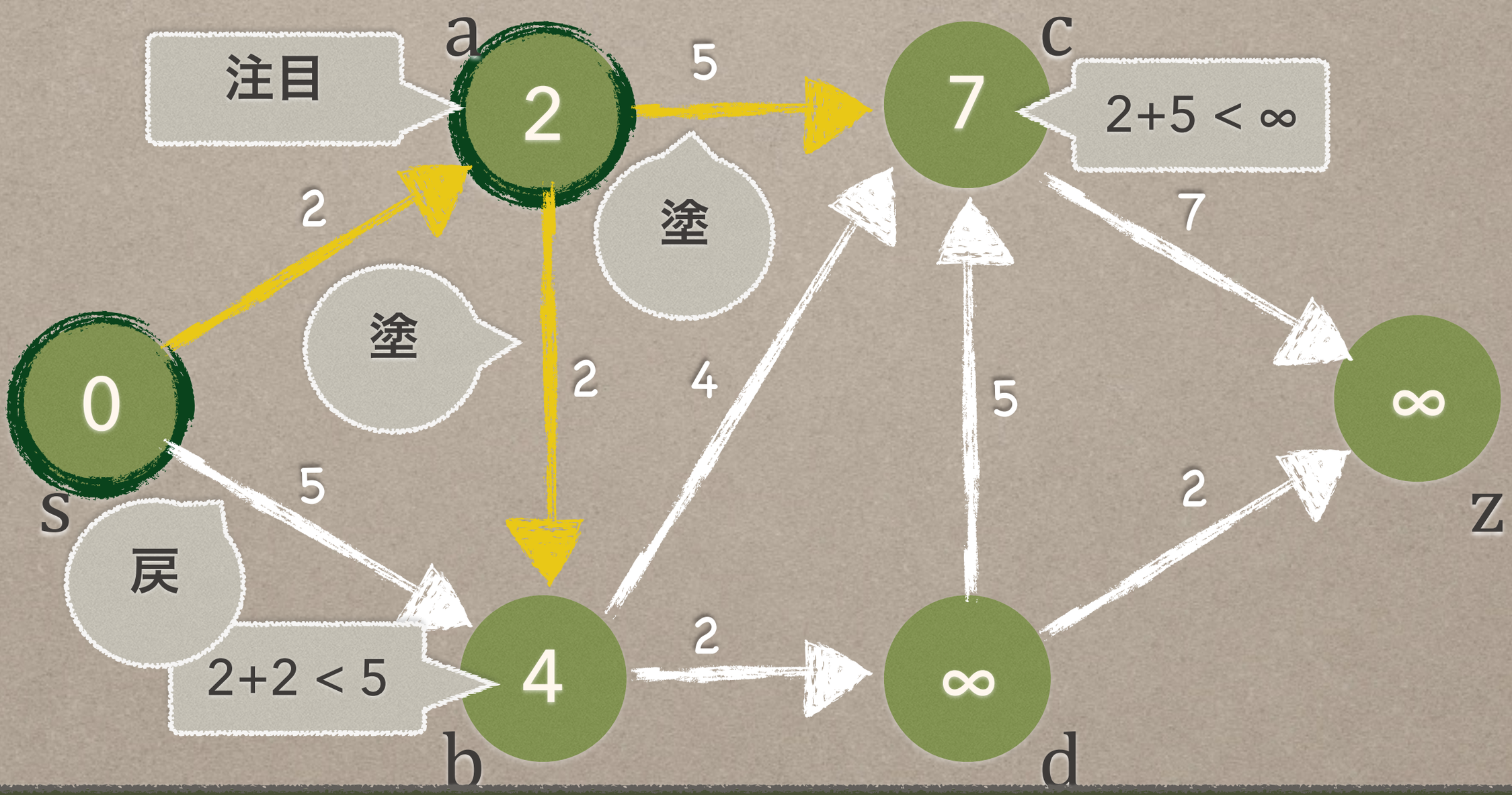
1.「印なし」のうち、コスト最小のノードに注目、印をつける

ダイクストラ法 (イメージ) 7/16
(3)隣接ノードを「アップデート」



1.隣接ノードのコスト値を「比較」

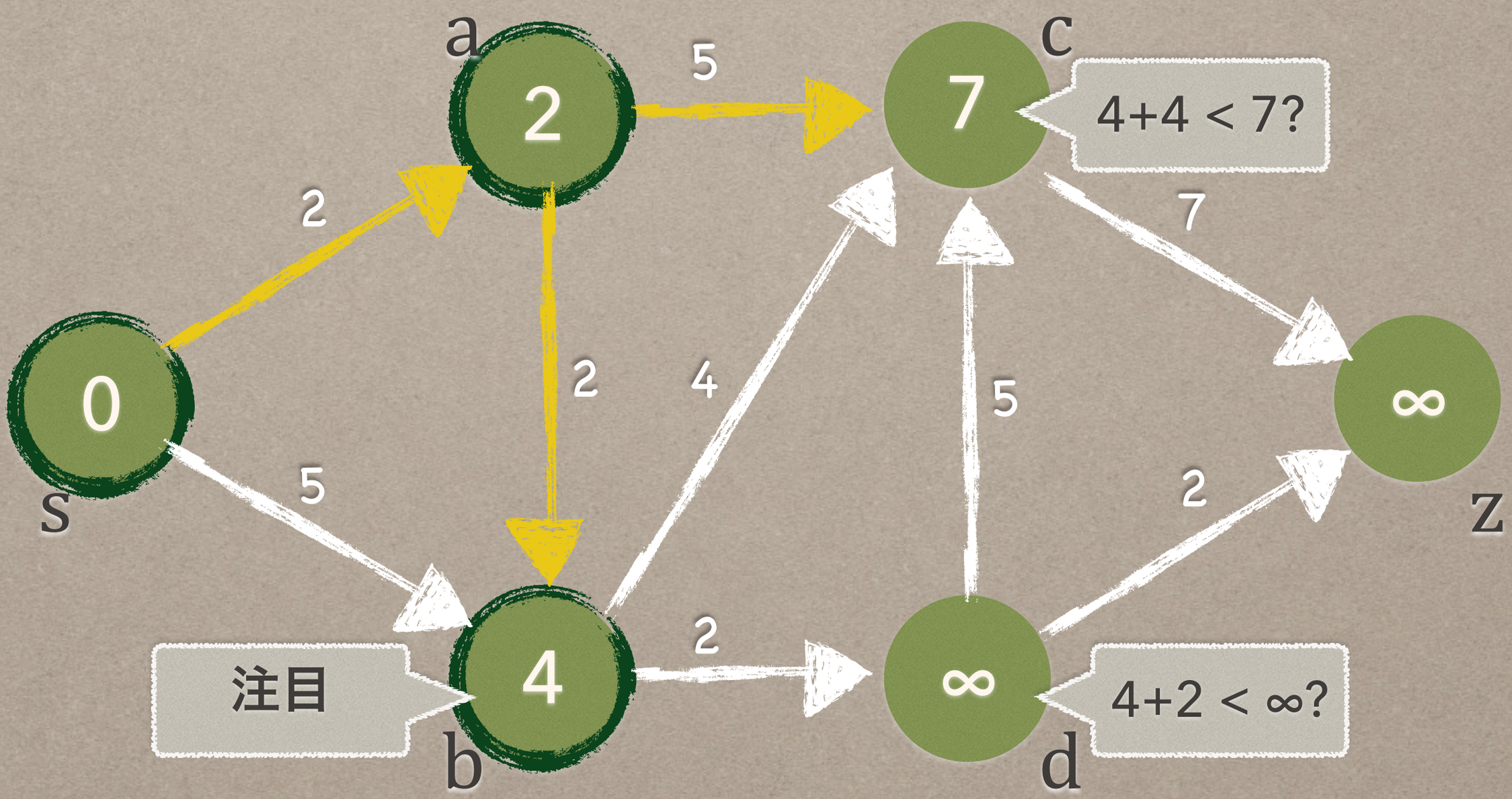
ダイクストラ法 (イメージ) 8/16
(3)隣接ノードを「アップデート」



2.コストが小さくなるなら更新して、色を塗る

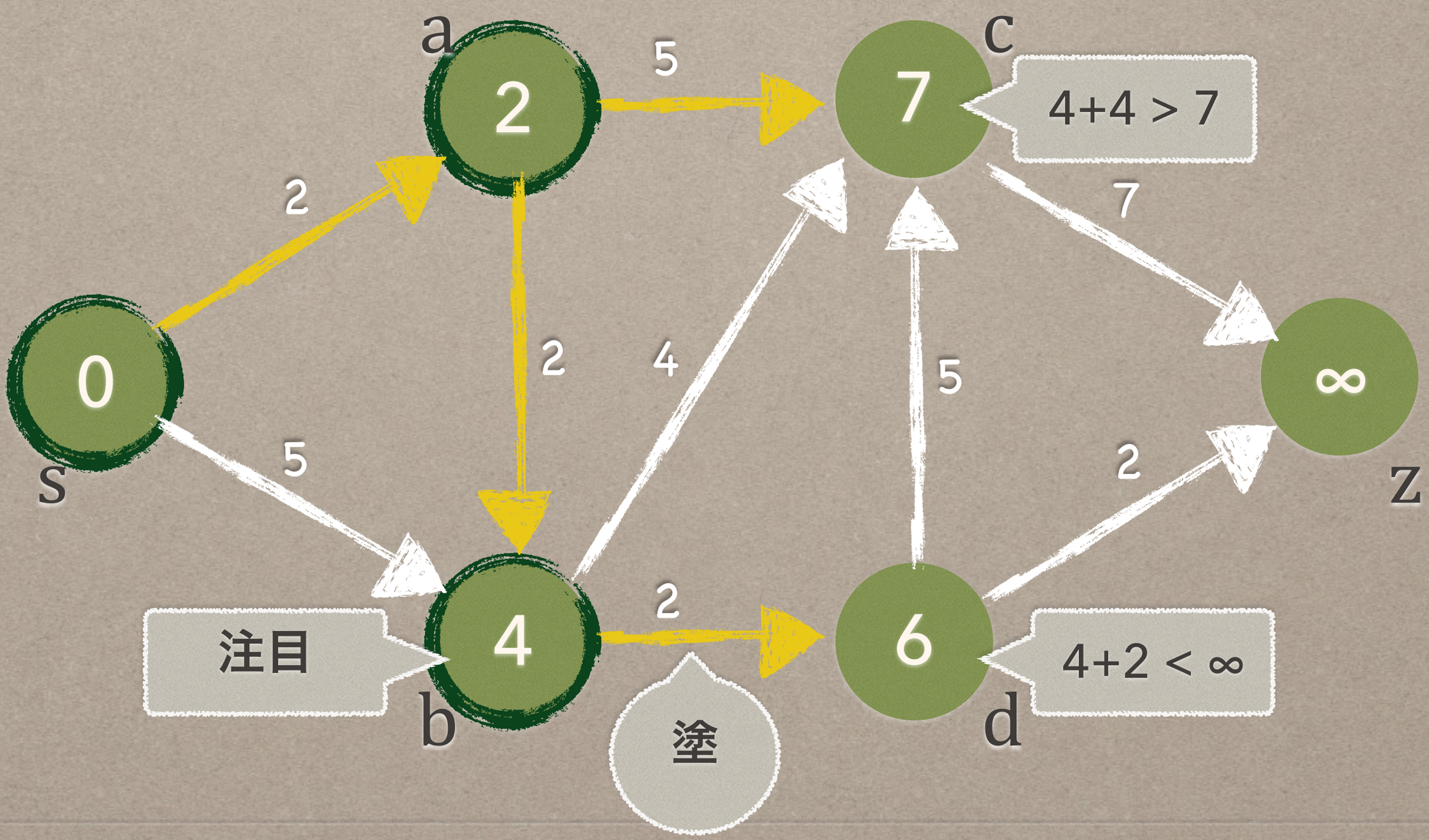
ダイクストラ法 (イメージ) 9/16

(2)印づけ、(3)アップデートを繰り返す



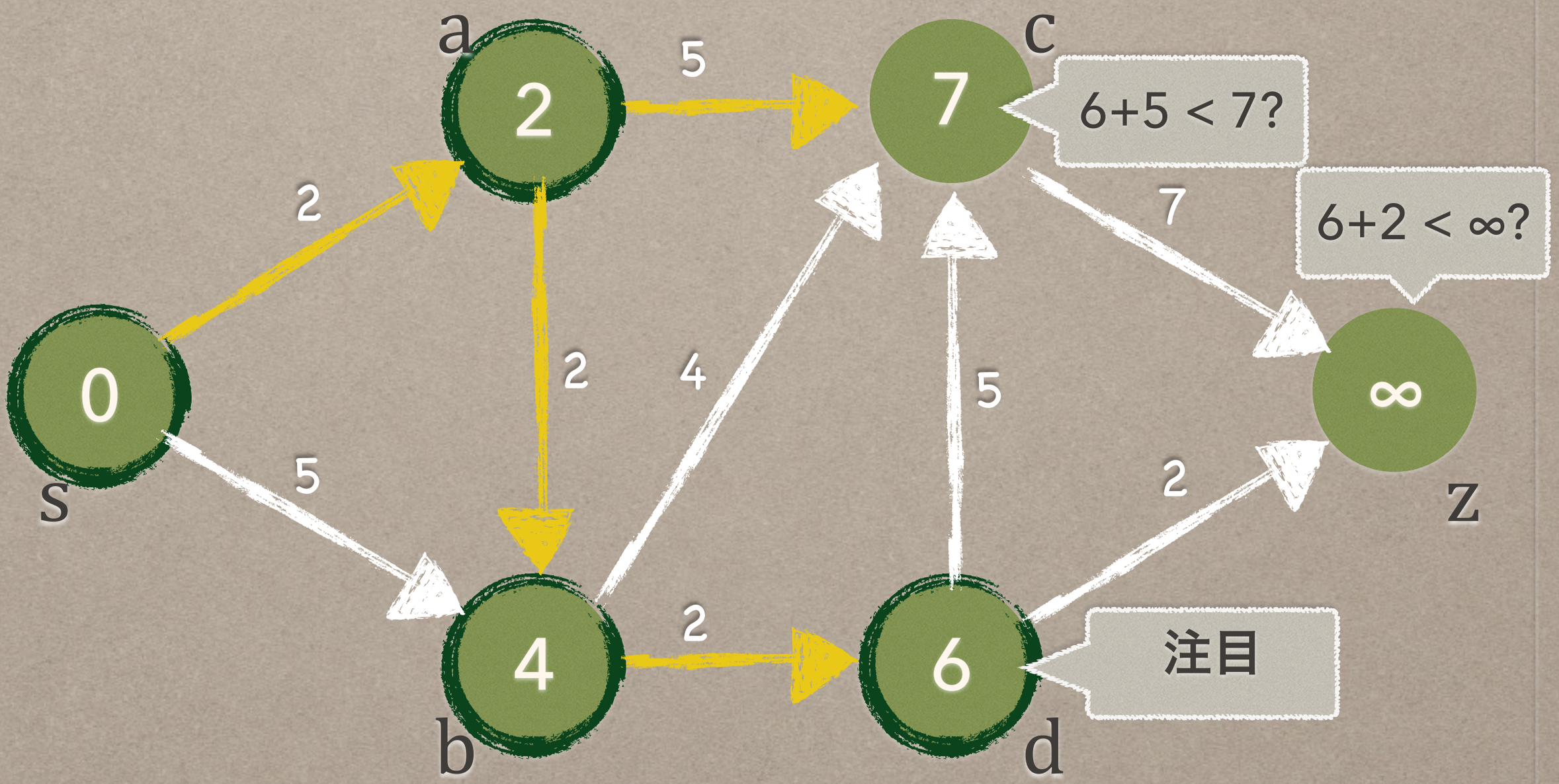
ダイクストラ法 (イメージ) 10/16

(2)印づけ、(3)アップデートを繰り返す



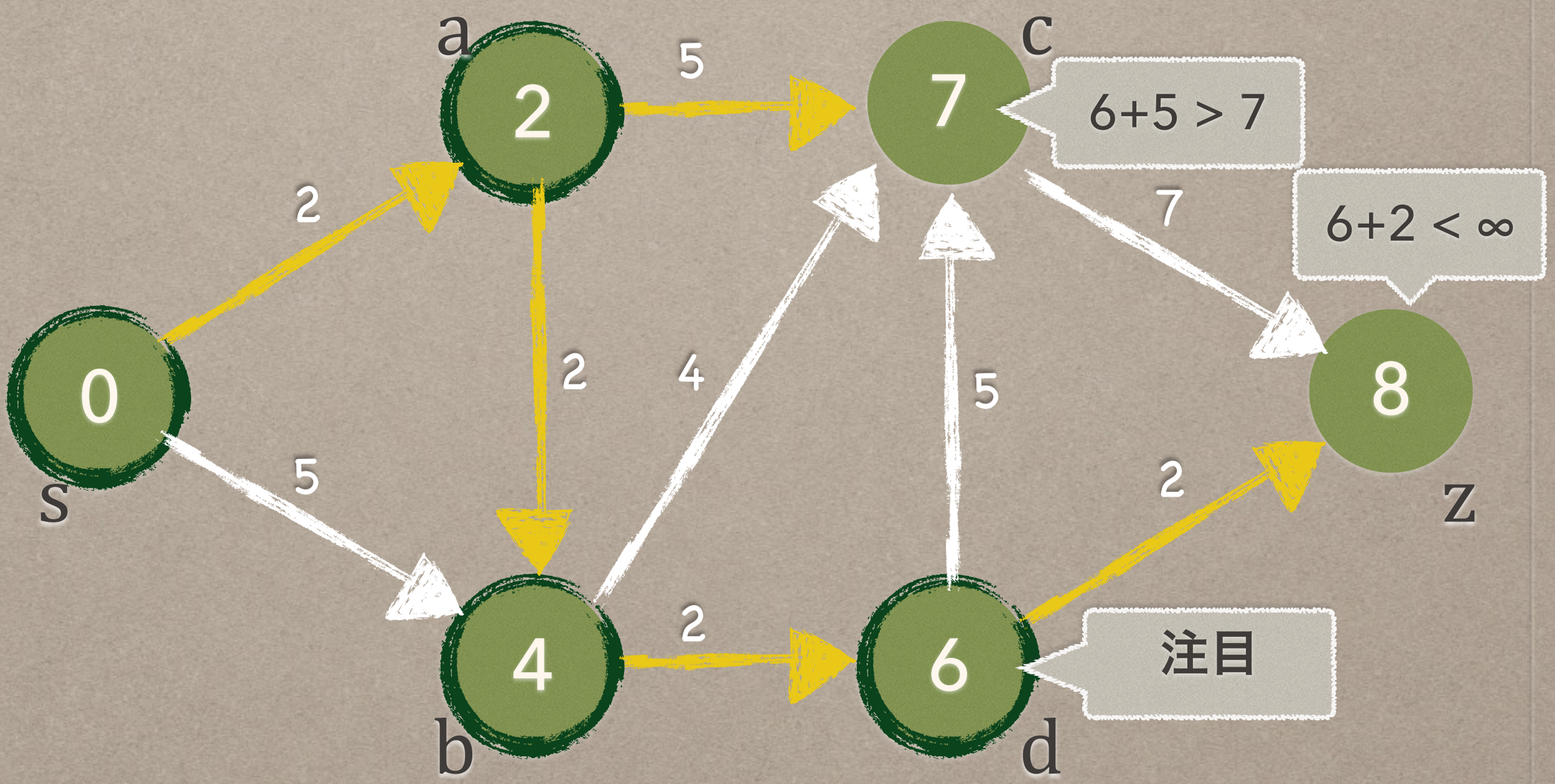
ダイクストラ法 (イメージ) 11/16

(2)印づけ、(3)アップデートを繰り返す



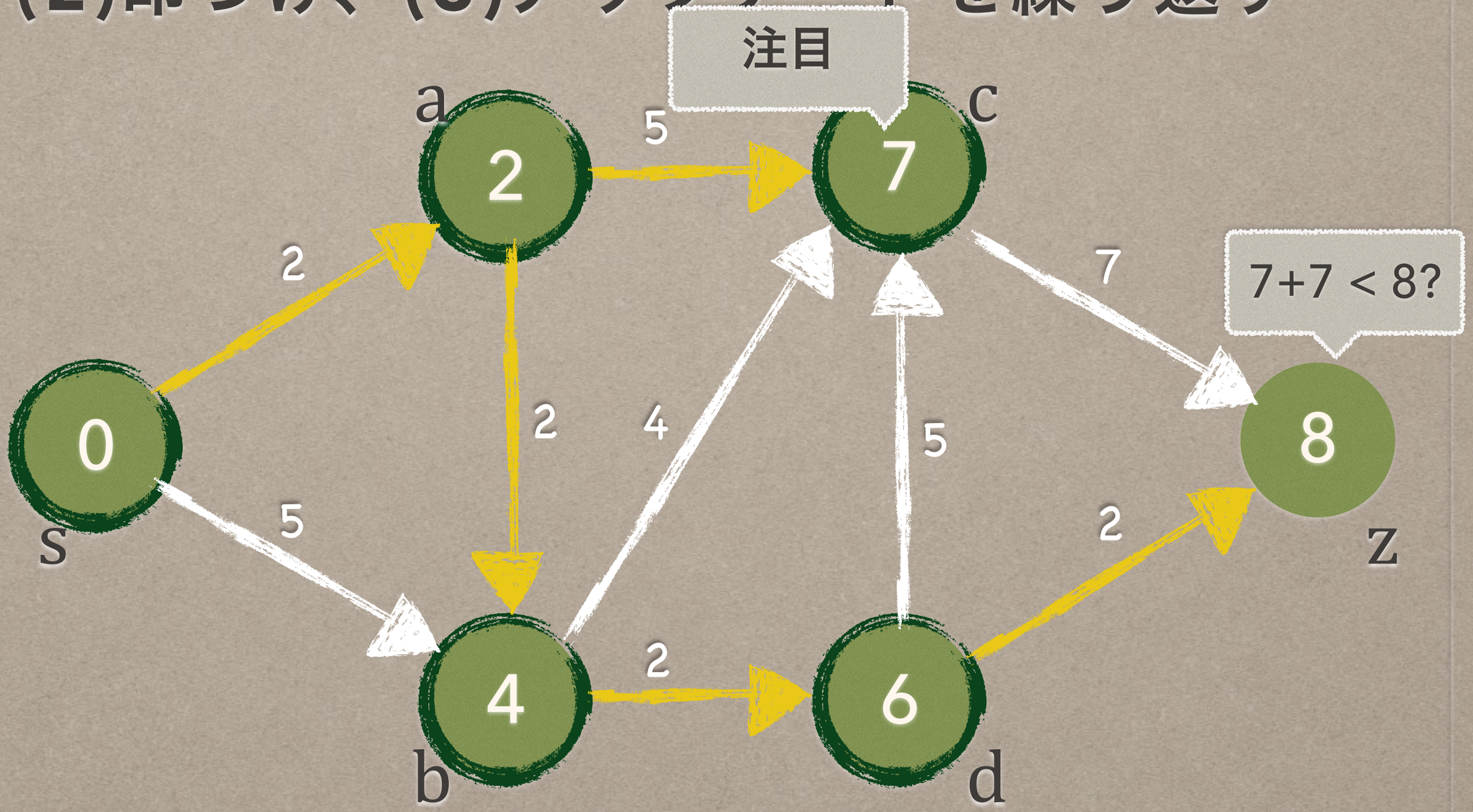
ダイクストラ法 (イメージ) 12/16

(2)印づけ、(3)アップデートを繰り返す



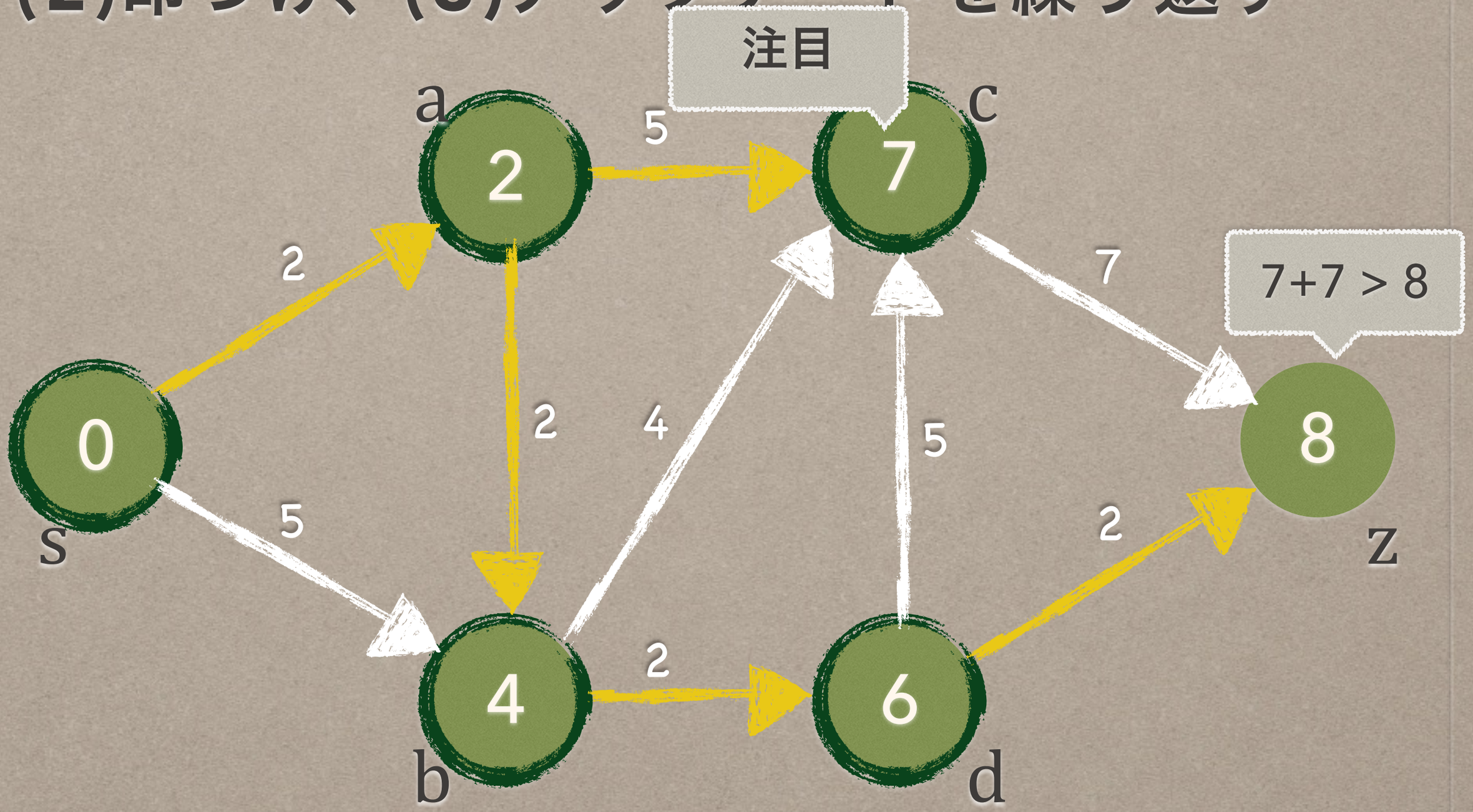
ダイクストラ法 (イメージ) 13/16

(2)印づけ、(3)アップデートを繰り返す



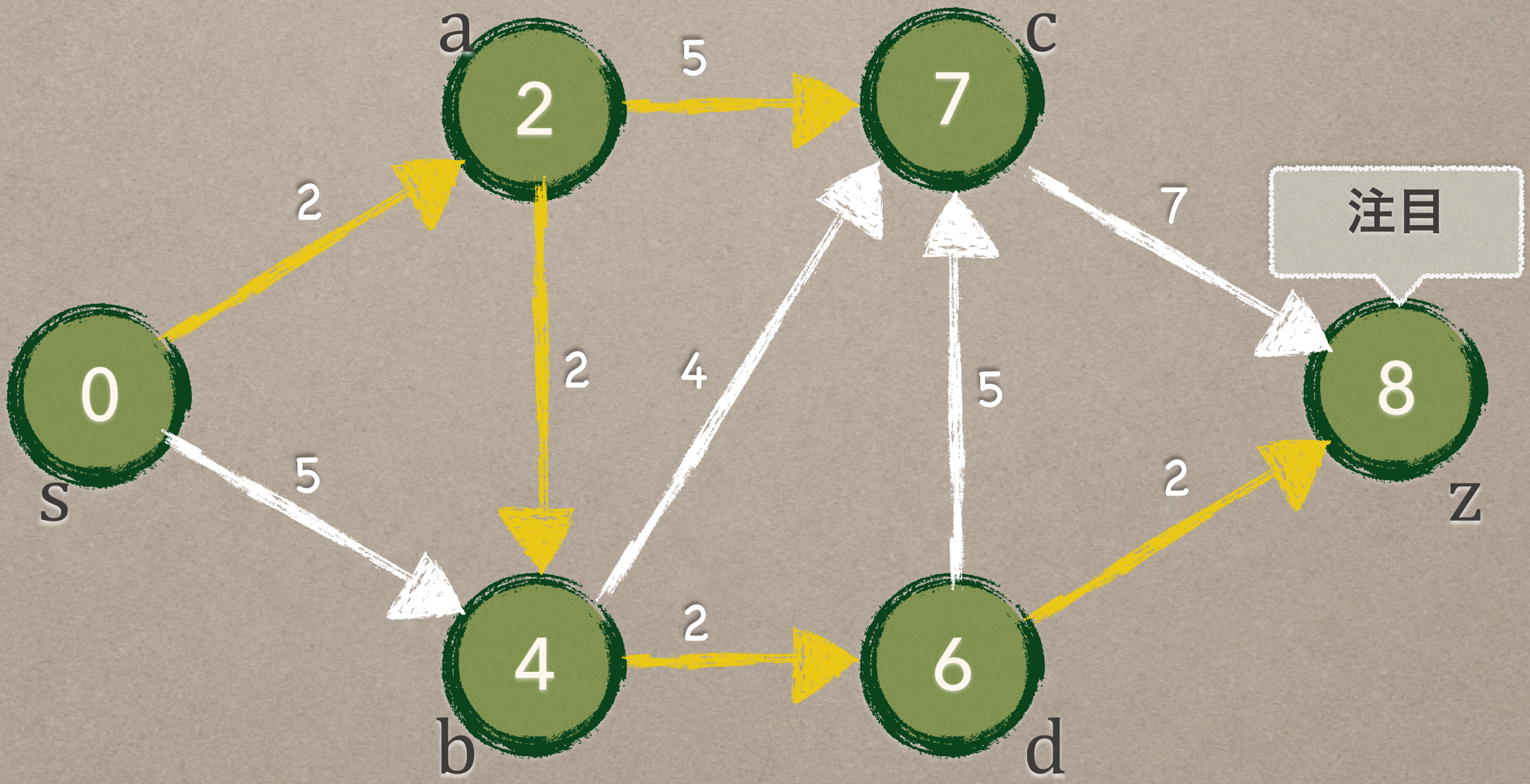
ダイクストラ法 (イメージ) 14/16

(2)印づけ、(3)アップデートを繰り返す

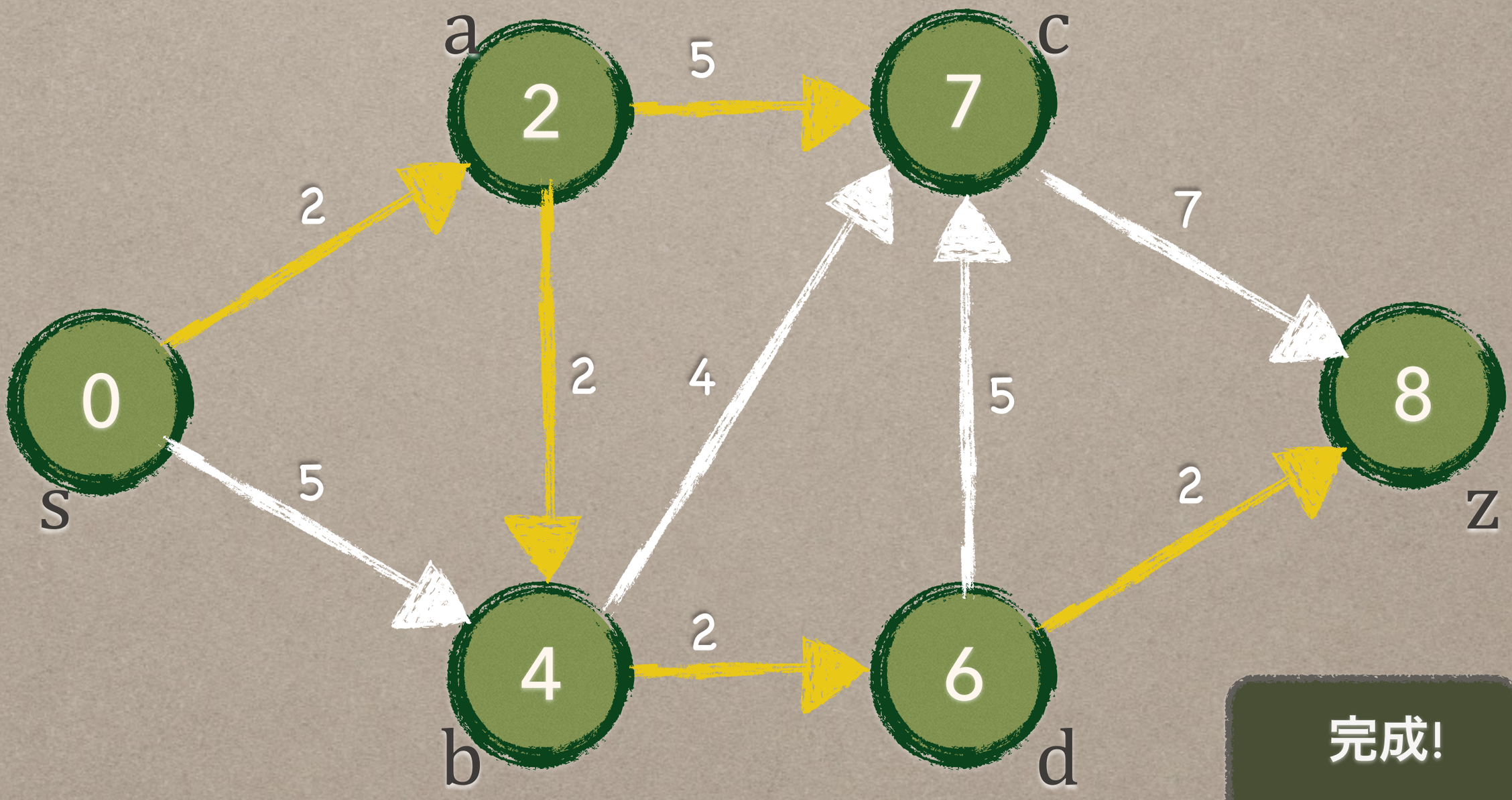


ダイクストラ法 (イメージ) 15/16

(2)印づけ、(3)アップデートを繰り返す



ダイクストラ法 (イメージ) 16/16
全員に印がついたら完成



ダイクストラ法 (まとめ)

入力: (有向グラフ, 始点ノード)

(1) 初期化する

(2) 「印」なしの最小コストのノードに注目、印をつける

(3) 隣接ノードをアップデート

- コストを評価し、「更新」できるならエッジに着色
- コスト = 注目ノードのコスト + エッジ(来た道)のコスト

(4) (2)〜(3)を繰り返す

(5) 「色が塗られた路」を最短経路として出力

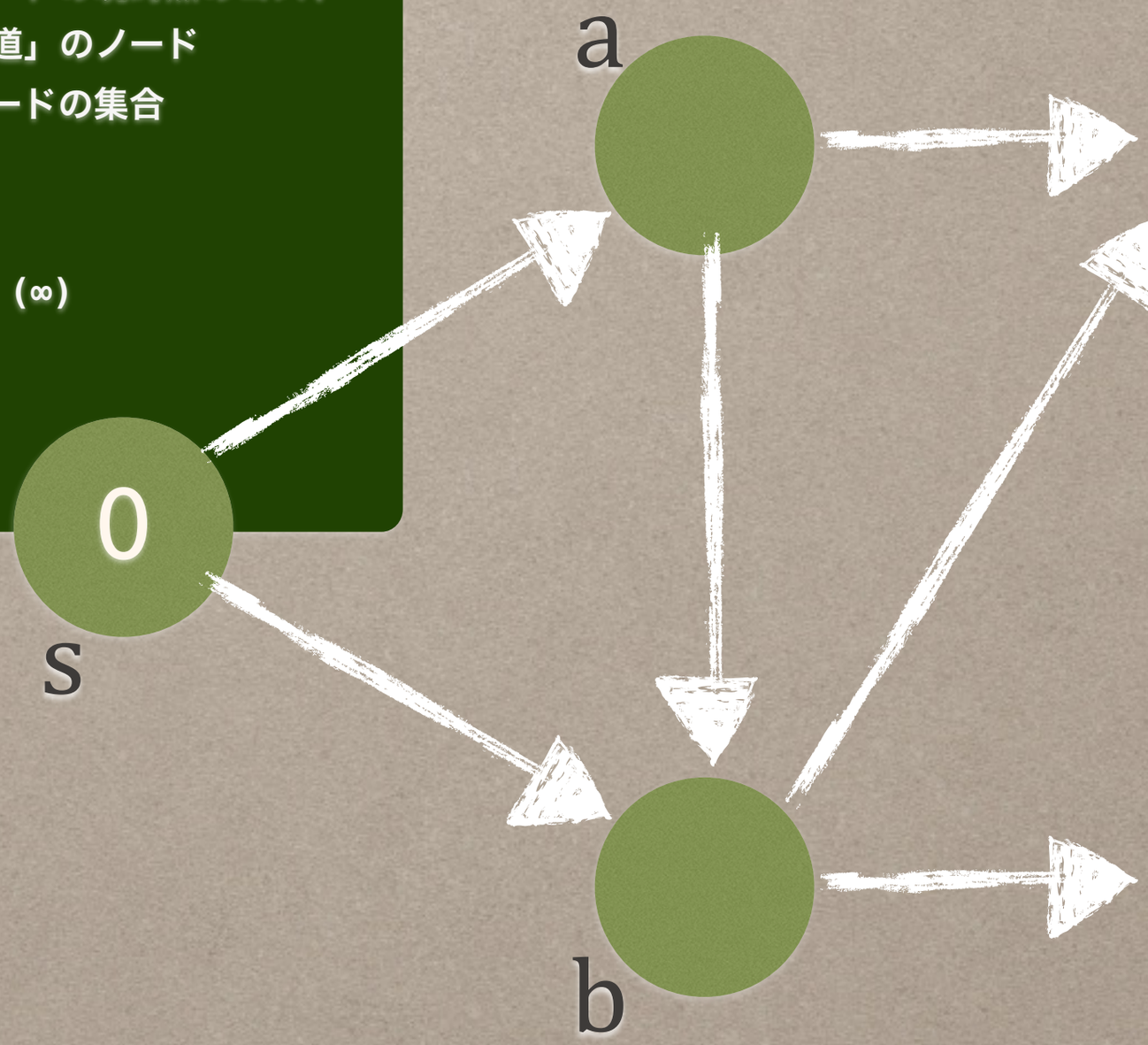
最小コストとなる
「来た道」だけを選ぶ

ダイクストラ法の実装例 (1/5)

ノードのデータ表現(1)

```
class Node
  attr_accessor :name          # :s や :a, :z など
  attr_accessor :cost          # このノードの現時点のコスト
  attr_accessor :predecessor   # 「来た道」のノード
  attr_accessor :neighbors     # 隣接ノードの集合

  def initialize(name)
    @name = name
    @cost = Float::INFINITY    # 無限大 (∞)
    @predecessor = nil
    @neighbors = Hash.new
  end
end
```

```
# s の初期化の例
s = Node.new(:s)
s.cost = 0
s.predecessor = nil
s.neighbors[:a] = a
s.neighbors[:b] = b
```

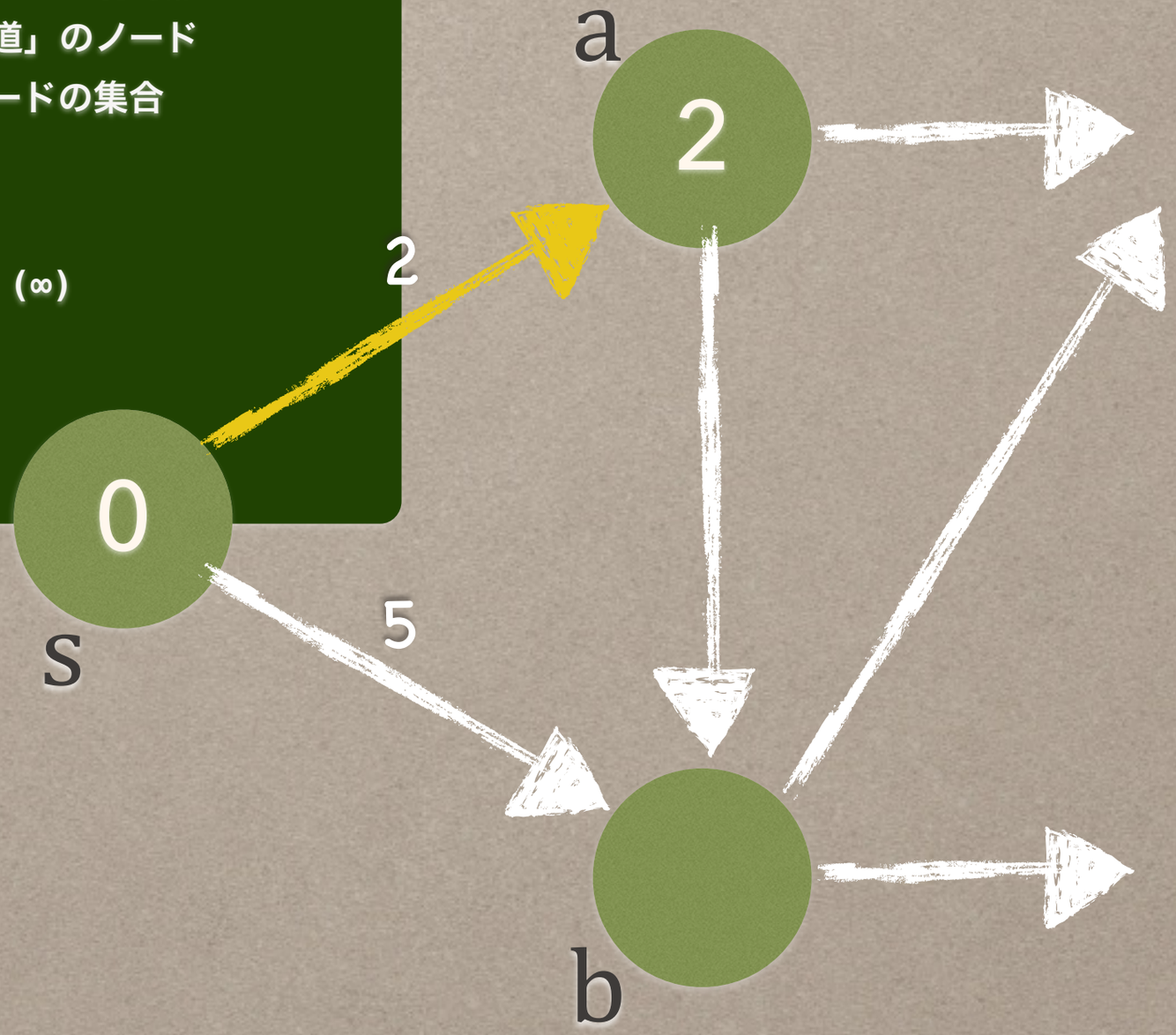


ダイクストラ法の実装例 (2/5)

ノードのデータ表現(2)

```
class Node
  attr_accessor :name          # :s や :a, :z など
  attr_accessor :cost          # このノードの現時点のコスト
  attr_accessor :predecessor   # 「来た道」のノード
  attr_accessor :neighbors     # 隣接ノードの集合

  def initialize(name)
    @name = name
    @cost = Float::INFINITY    # 無限大 (∞)
    @predecessor = nil
    @neighbors = Hash.new
  end
end
```

```
# a の初期化の例
a = Node.new(:s)
a.cost = 2
a.predecessor = s     # 「色を塗る」
a.neighbors[:b] = b
a.neighbors[:c] = c
```



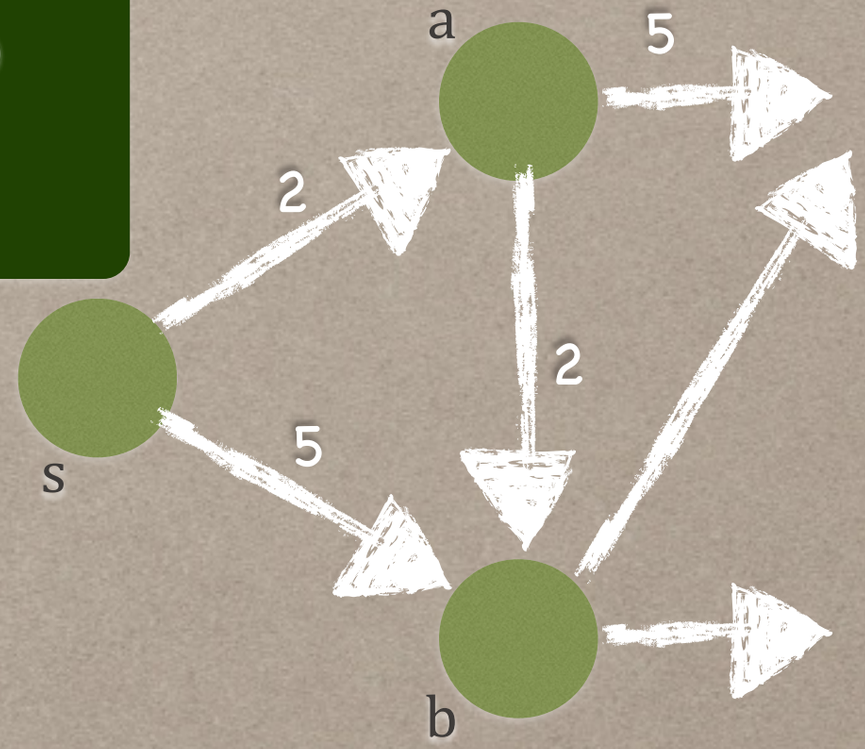
ダイクストラ法の実装例 (3/5)

グラフのデータ表現

```
class Graph
  attr_accessor :nodes    # このグラフの全ノード
  attr_accessor :edges    # このグラフの全エッジ

  def initialize(param)
    @nodes = Hash.new
    @edges = EdgeCollection.new  # ハッシュマップの入れ子クラス
    param.each do |src_name, dest_name, cost|
      node_src = @nodes[src_name] ||= Node.new(src_name)
      node_dest = @nodes[dest_name] ||= Node.new(dest_name)

      edge = @edges[src_name, dest_name] = Edge.new(cost)
      node_src.neighbors[node_dest.name] = edge
    end
  end
end
```

```
class Edge
  attr_accessor :cost

  def initialize(cost)
    @cost = cost
  end
end
```

```
# グラフの初期化例
graph = Graph.new([[:s, :a, 2],
                   [:s, :b, 5],
                   [:a, :b, 2],
                   [:a, :c, 5],
                    …… (略) ……
                   [:d, :z, 2]])
```



ダイクストラ法の実装例 (4/5)

アルゴリズム(1)

```
class Dijkstra
  def initialize(graph)
    @graph = graph
  end

  def calc(s)  # s は始点ノード
    all_nodes = @graph.nodes.values   # 全ノード
    unvisited_nodes = all_nodes - [s] # 印なしのノード群(sのみ確定済)

    # (1) 初期化
    s.cost = 0
    s.predecessor = nil

    # s 以外の全ノードのコストを設定
    unvisited_nodes.each do |node|
      node.cost = cost(s, node)
      node.predecessor = s    # 路に「色を塗る」
    end

    # …… (続く)
```

ダイクストラ法の実装例 (5/5)

アルゴリズム(2)

```
# ……（続き）
# 全てのノードに印が付くまで(2)〜(3)を繰り返す
while !unvisited_nodes.empty?
  # (2) 無印のうち、コストが最小のノードに注目(currentとする)
  current = unvisited_nodes.min {|x, y| x.cost <=> y.cost }
  unvisited_nodes.delete(current)  # ノードに「印をつける」

  # (3) 注目ノードの隣接ノード(neighbors)のコストをアップデート
  current.neighbors.each_key do |name|
    neighbor = @graph.nodes[name]
    cost_from_current = cost(current, neighbor)

    # currentからの方が近い場合、neighborのコストをアップデート
    if cost_from_current < neighbor.cost
      neighbor.cost = cost_from_current
      neighbor.predecessor = current
    end
  end
 end
end
```

```
# コスト評価関数(ダイクストラ法)
def cost(from, to)
  edge = \
    @graph.edges[from, to]

  if edge
    from.cost + edge.cost
  else
    # 両者を接続するエッジが
    # ない場合、無限大を返す
    Float::INFINITY
  end
end
```

Agenda

1. 導入
2. 計算量 (オーダ記法)
3. 最短経路問題 (ダイクストラ法)
4. 最小木問題 (プリム法)

4. 最小木問題 (プリム法)

- プリム法でできること
  - グラフと始点ノードが与えられた時、「全ノードを最小のコストでつなぐ木」を求めることができる
- プリム法の利用シーン
  - 全ノードを、なるべく安く繋ぎたい(コスト=お金とした場合)
- 有向グラフを例に説明します
  - ダイクストラ法と同様、無向グラフにも適用可能です

プリム法 (概要)

入力: (有向グラフ, 始点ノード)

(1) 初期化する

(2) 「印」なしの最小コストのノードに注目、印をつける

(3) 隣接ノードをアップデート

- コストを評価し、「更新」できるならエッジに着色
- コスト = ~~注目ノードのコスト +~~ エッジ(来た道)のコスト

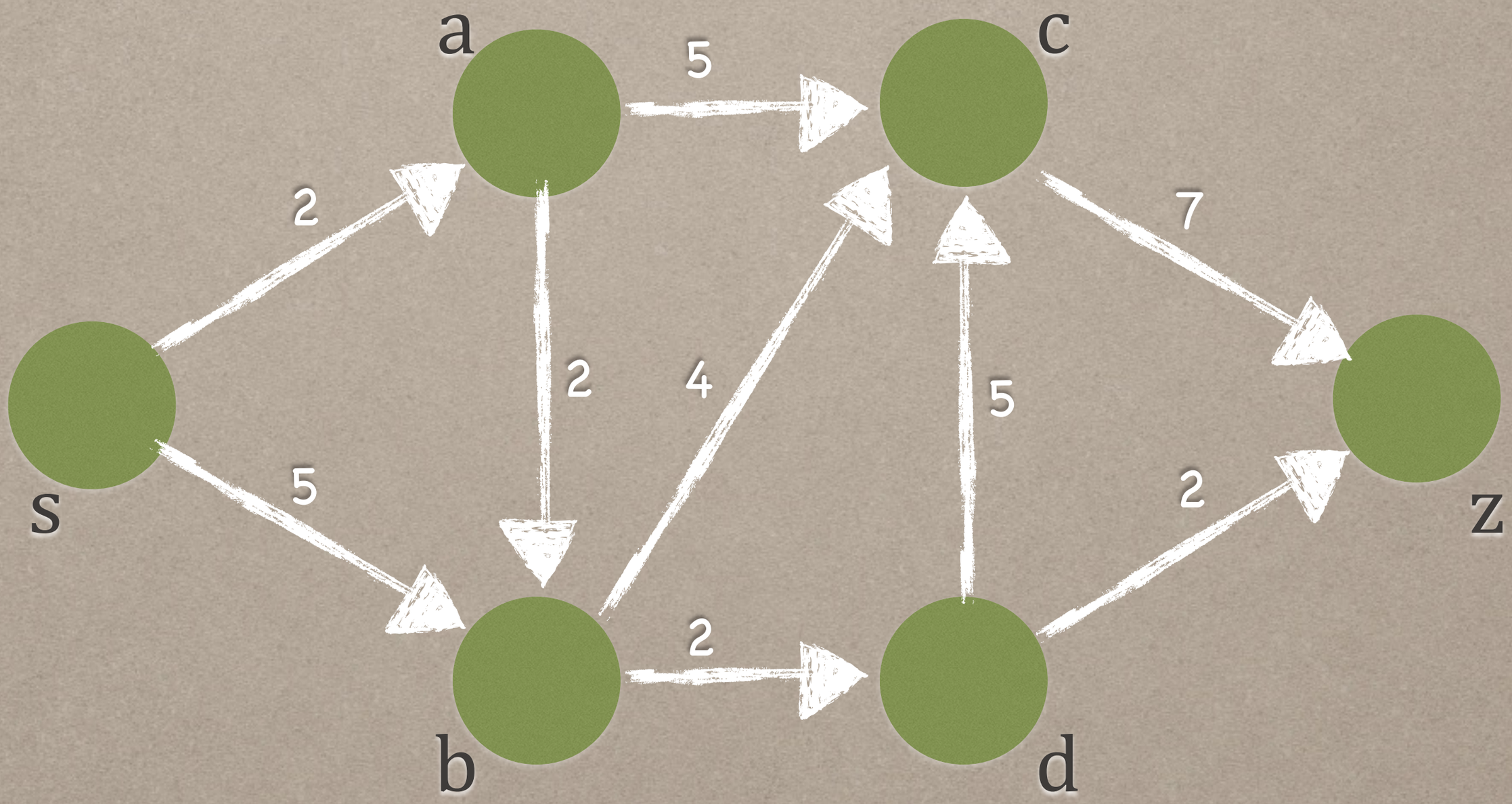
(4) (2)〜(3)を繰り返す

(5) 「色が塗られた路」を最小全域木として出力

コストの評価方法
だけが異なる

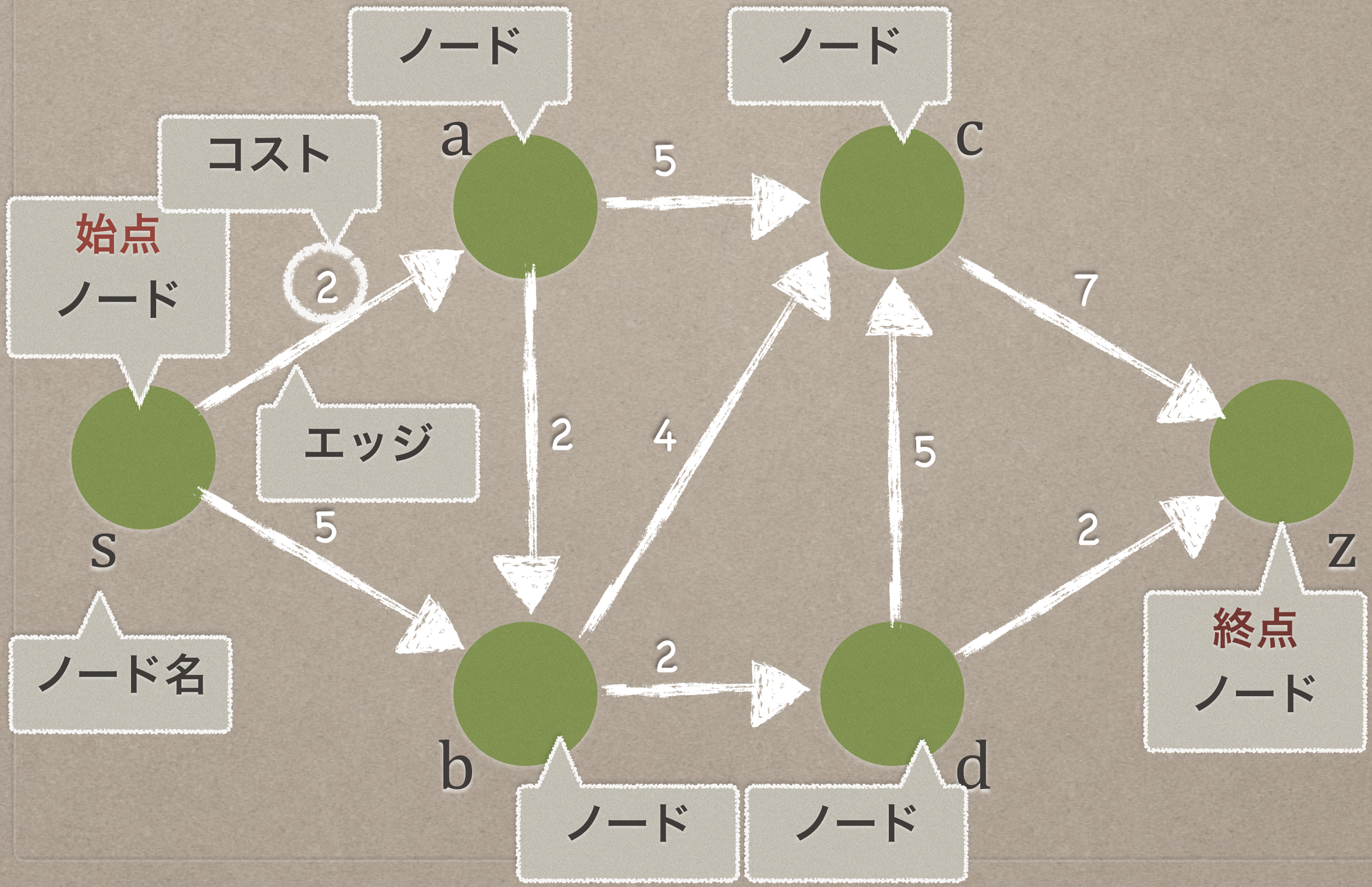
プリム法 (イメージ) 1/16

ダイクストラ法と同じ



プリム法 (イメージ) 2/16

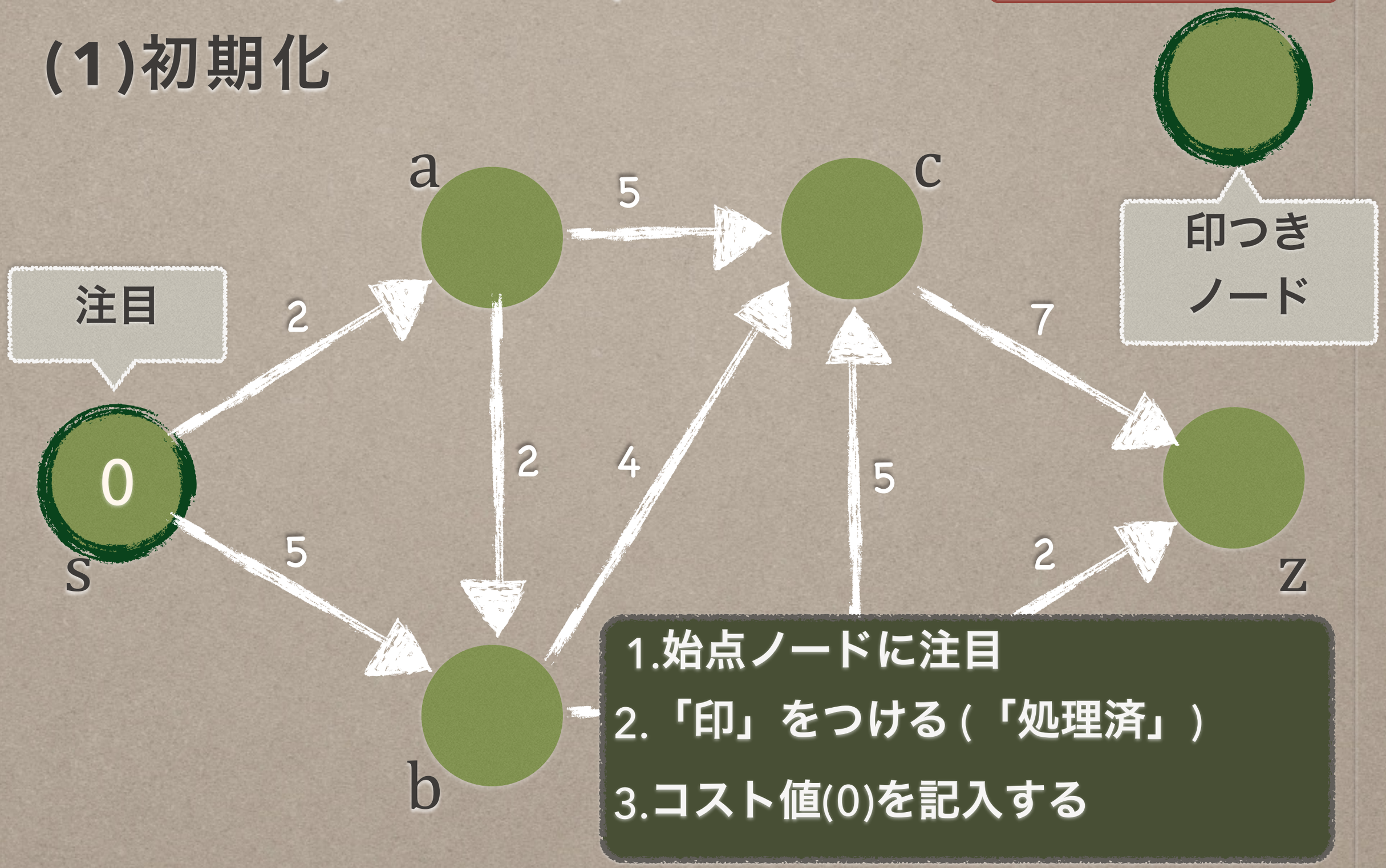
ダイクストラ法と同じ



プリム法 (イメージ) 3/16

(1)初期化

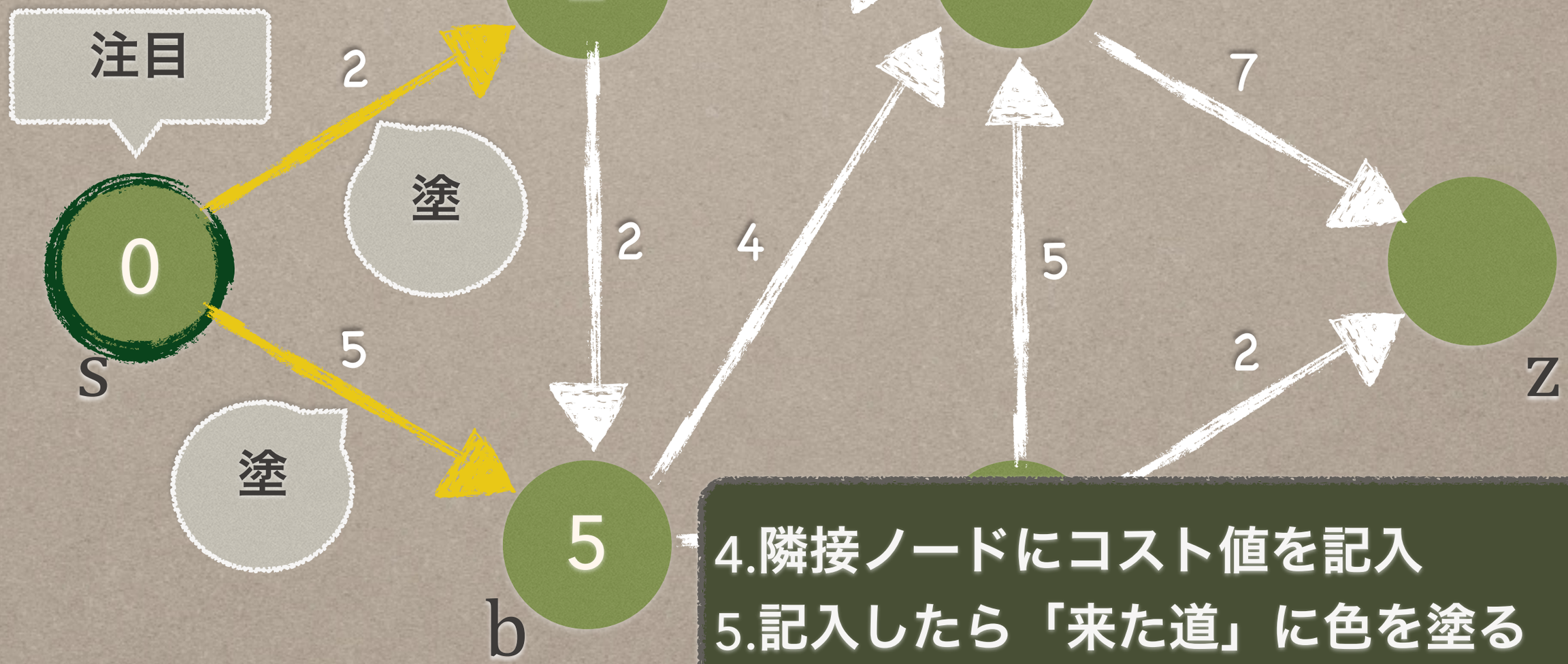
ダイクストラ法と同じ



プリム法 (イメージ) 4/16

(1)初期化

~~注目ノードのコスト+エッジのコスト~~



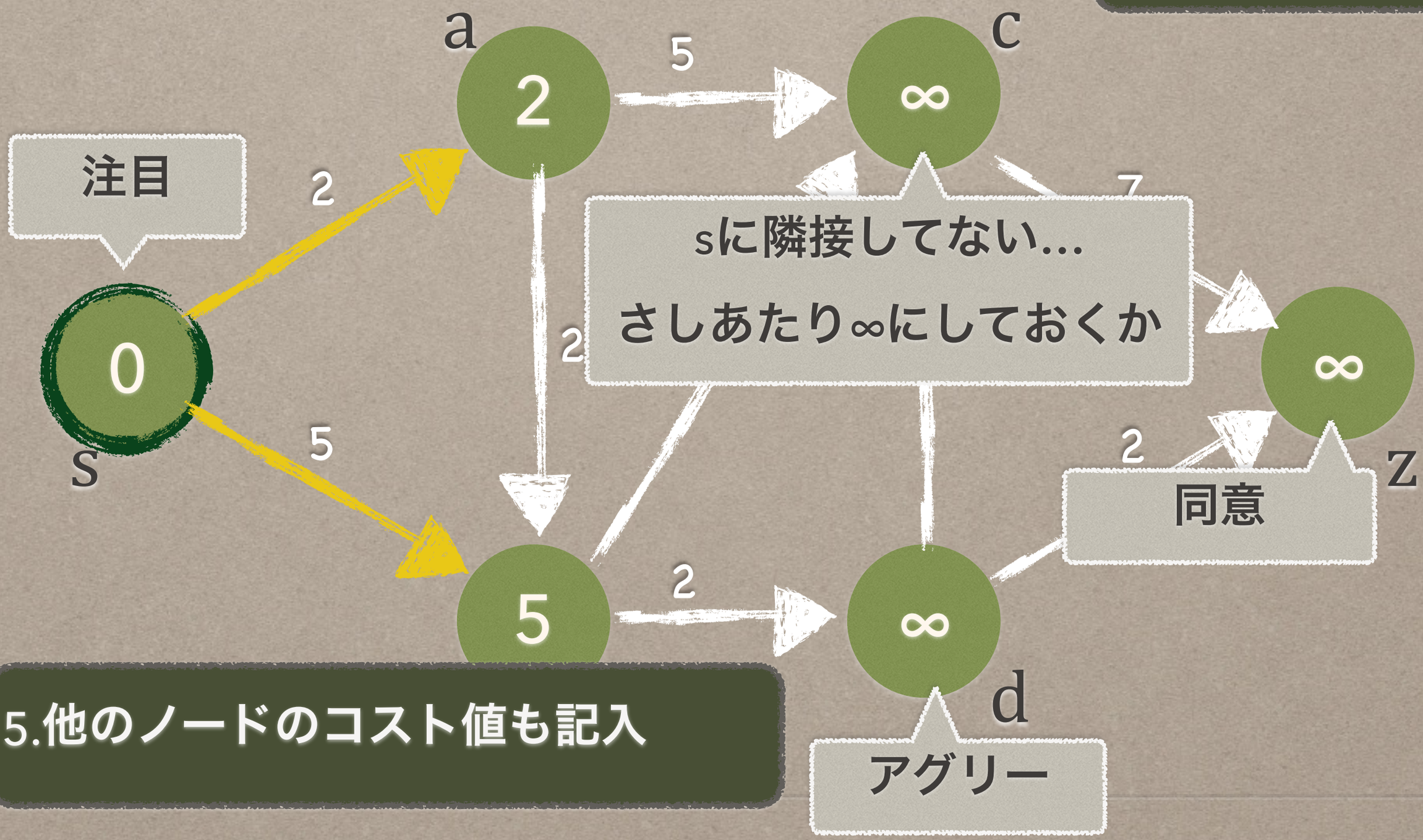
4.隣接ノードにコスト値を記入
5.記入したら「来た道」に色を塗る

プリム法 (イメージ) 5/16

(1)初期化

ダイクストラ法と同じ

初期化は完了!

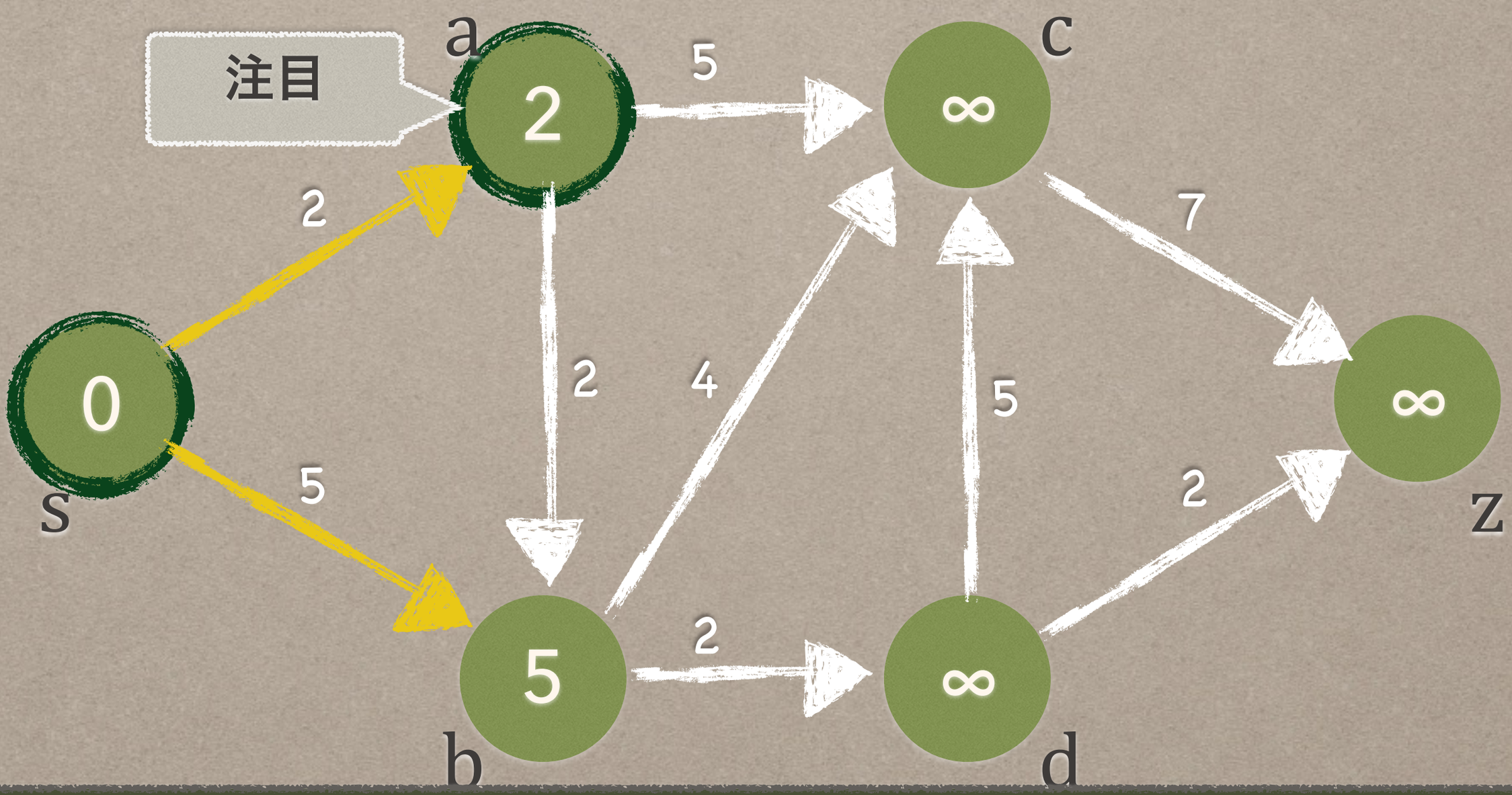


5.他のノードのコスト値も記入

プリム法 (イメージ) 6/16

(2)印なし最小ノードを選択

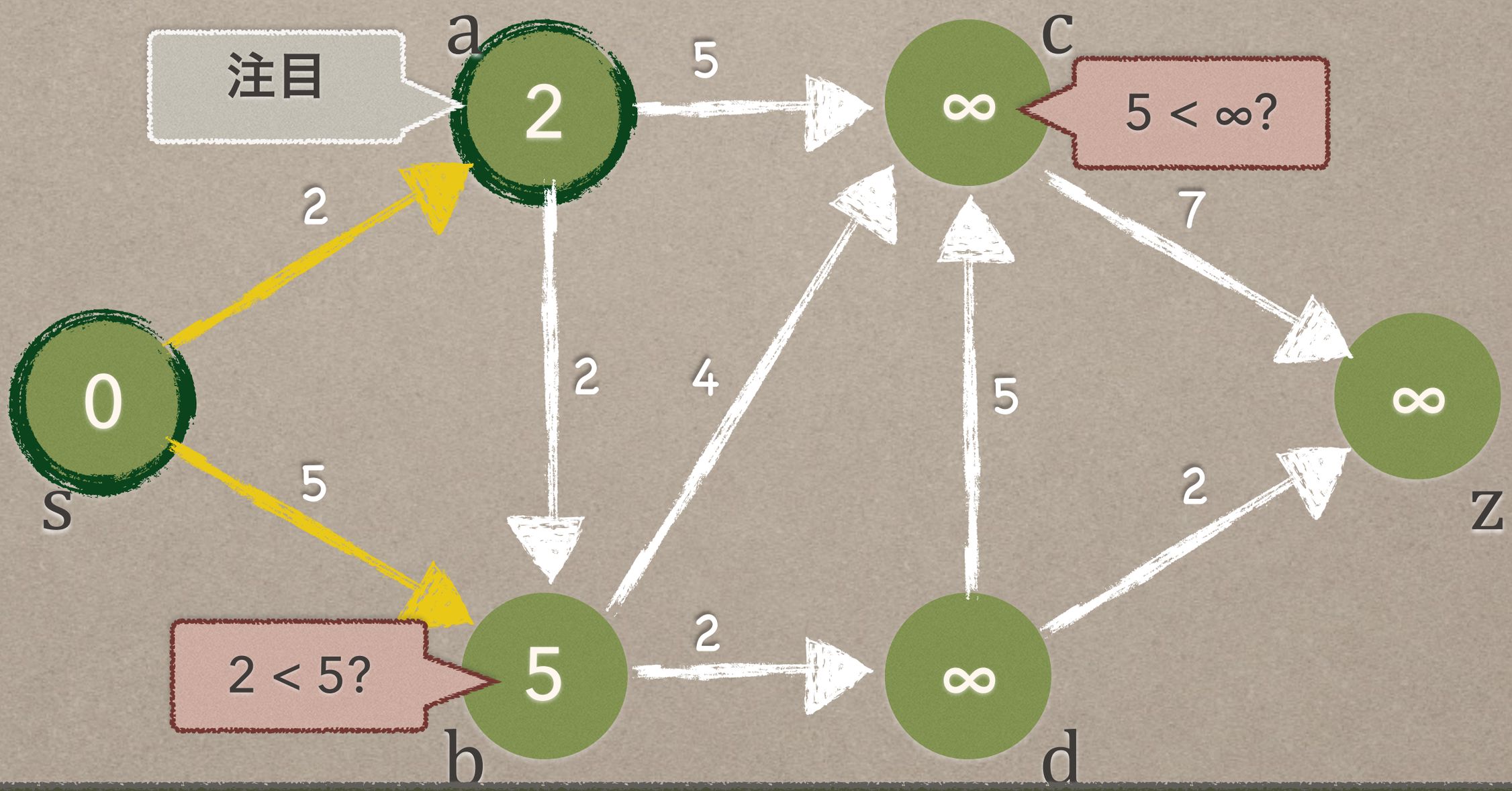
ダイクストラ法と同じ



1.「印なし」のうち、コスト最小のノードに注目、印をつける

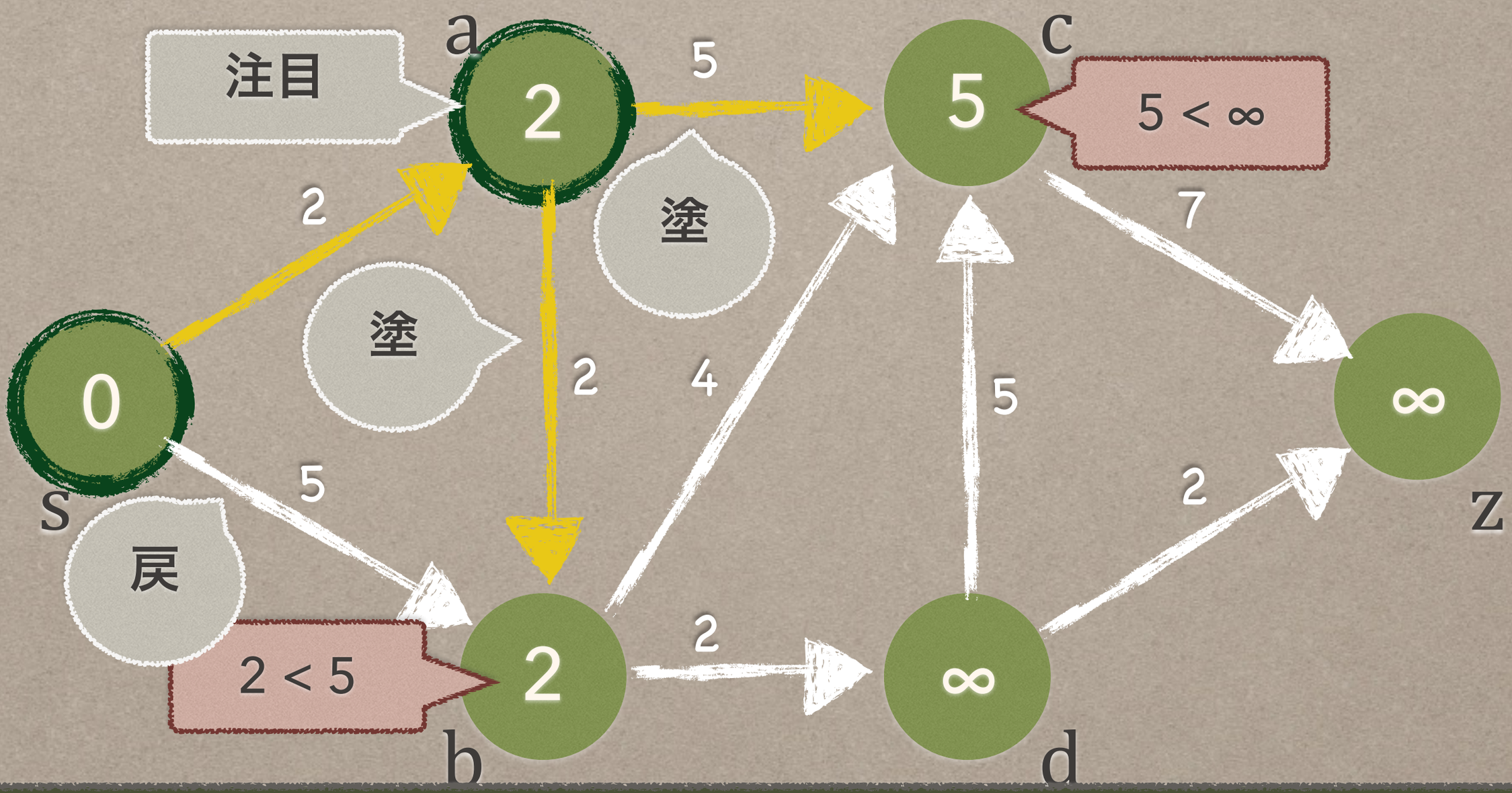
プリム法 (イメージ) 7/16

(3)隣接ノードを「アップデート」



1.隣接ノードのコスト値を「比較」

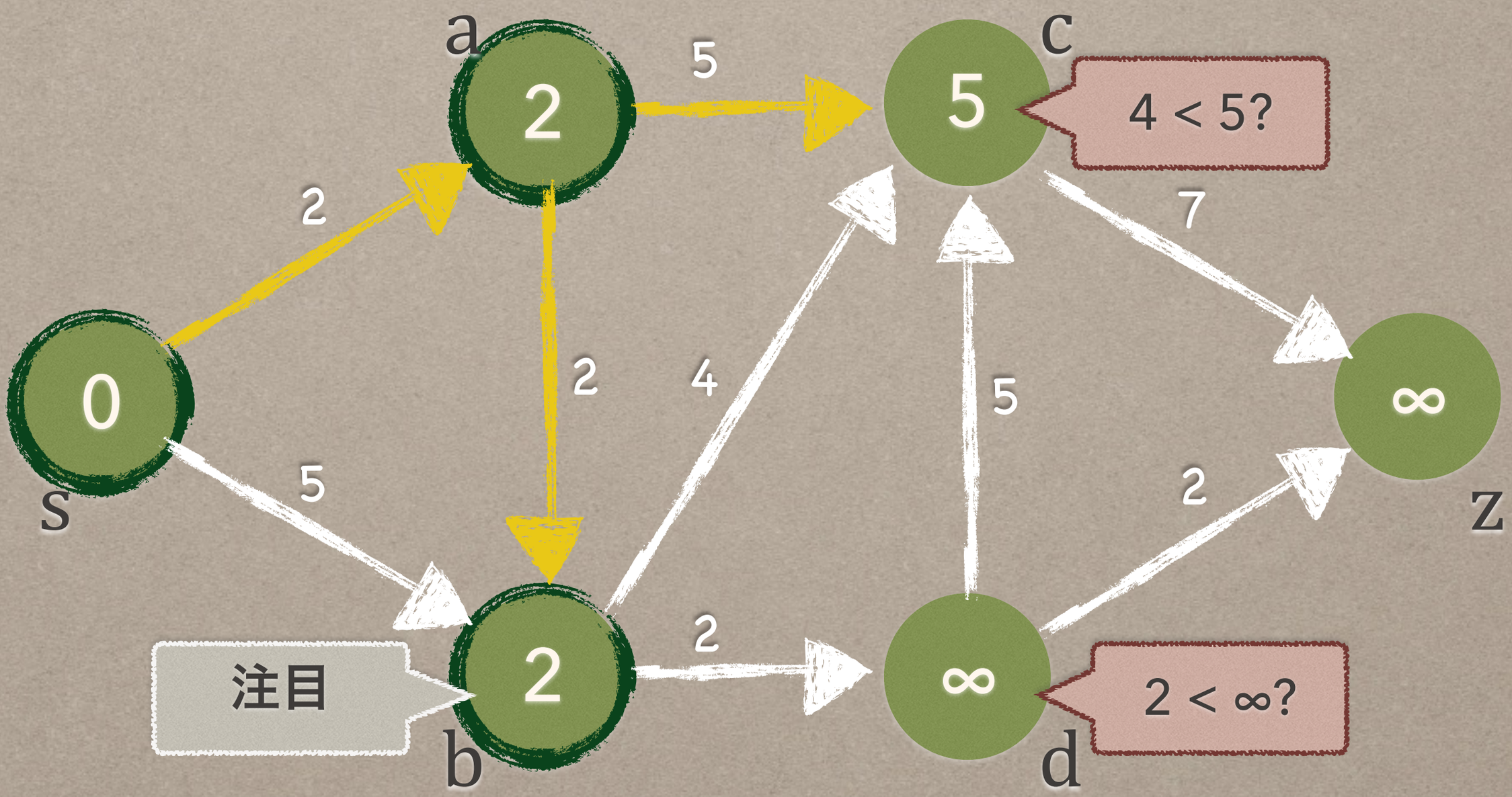
プリム法 (イメージ) 8/16
(3)隣接ノードを「アップデート」



2.コストが小さくなるなら更新して、色を塗る

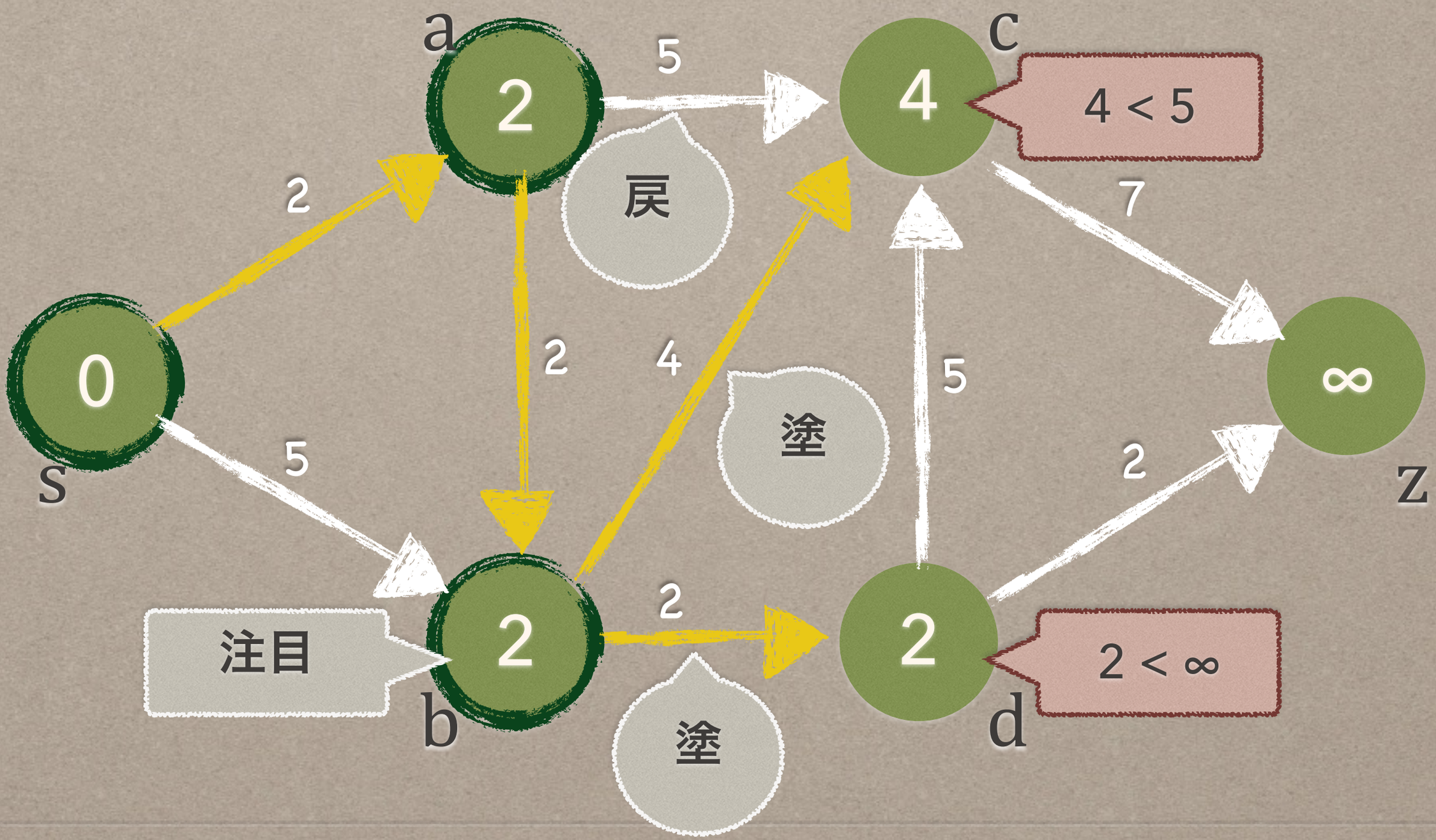
プリム法 (イメージ) 9/16

(2)印づけ、(3)アップデートを繰り返す



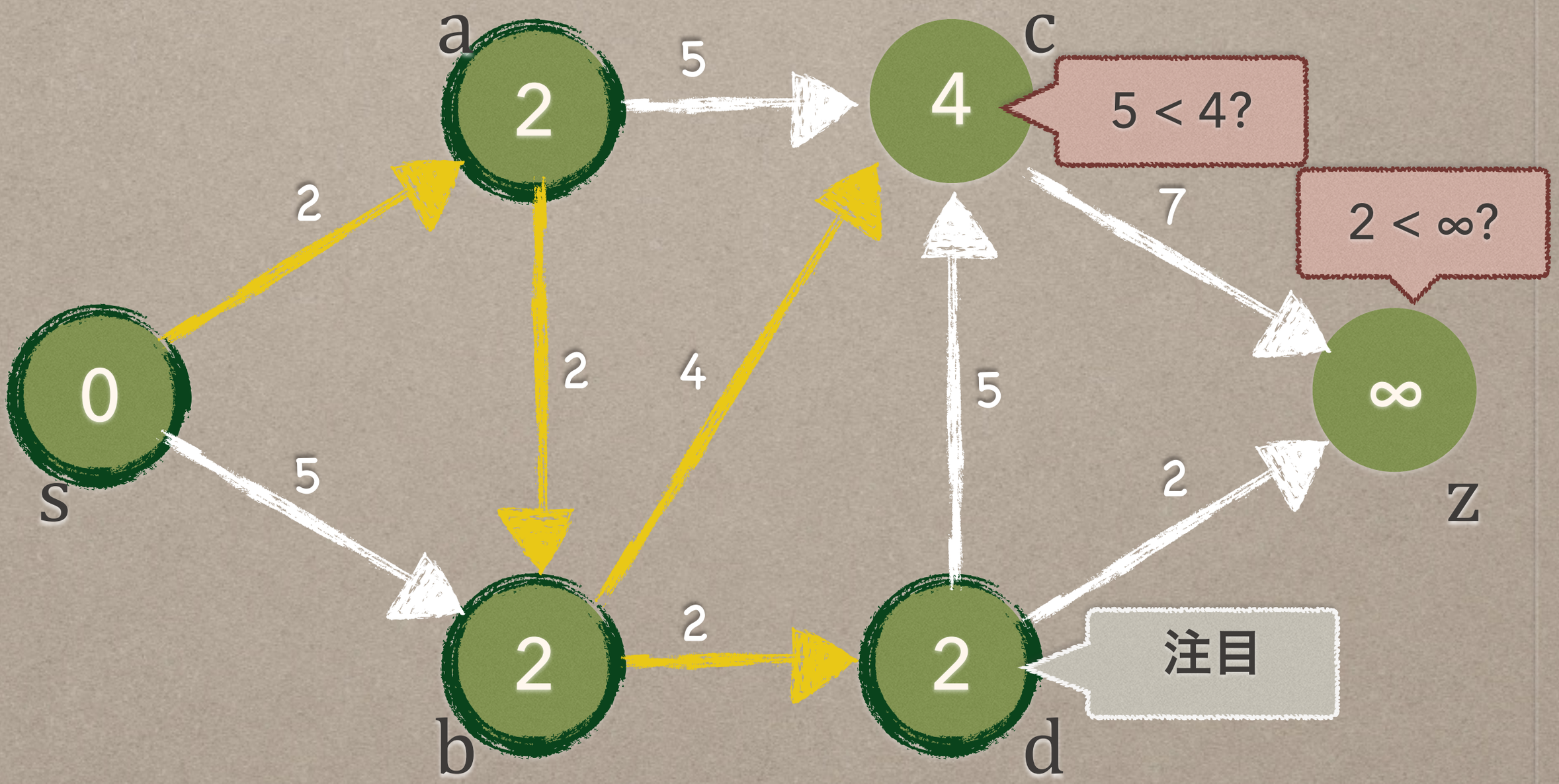
プリム法 (イメージ) 10/16

(2)印づけ、(3)アップデートを繰り返す



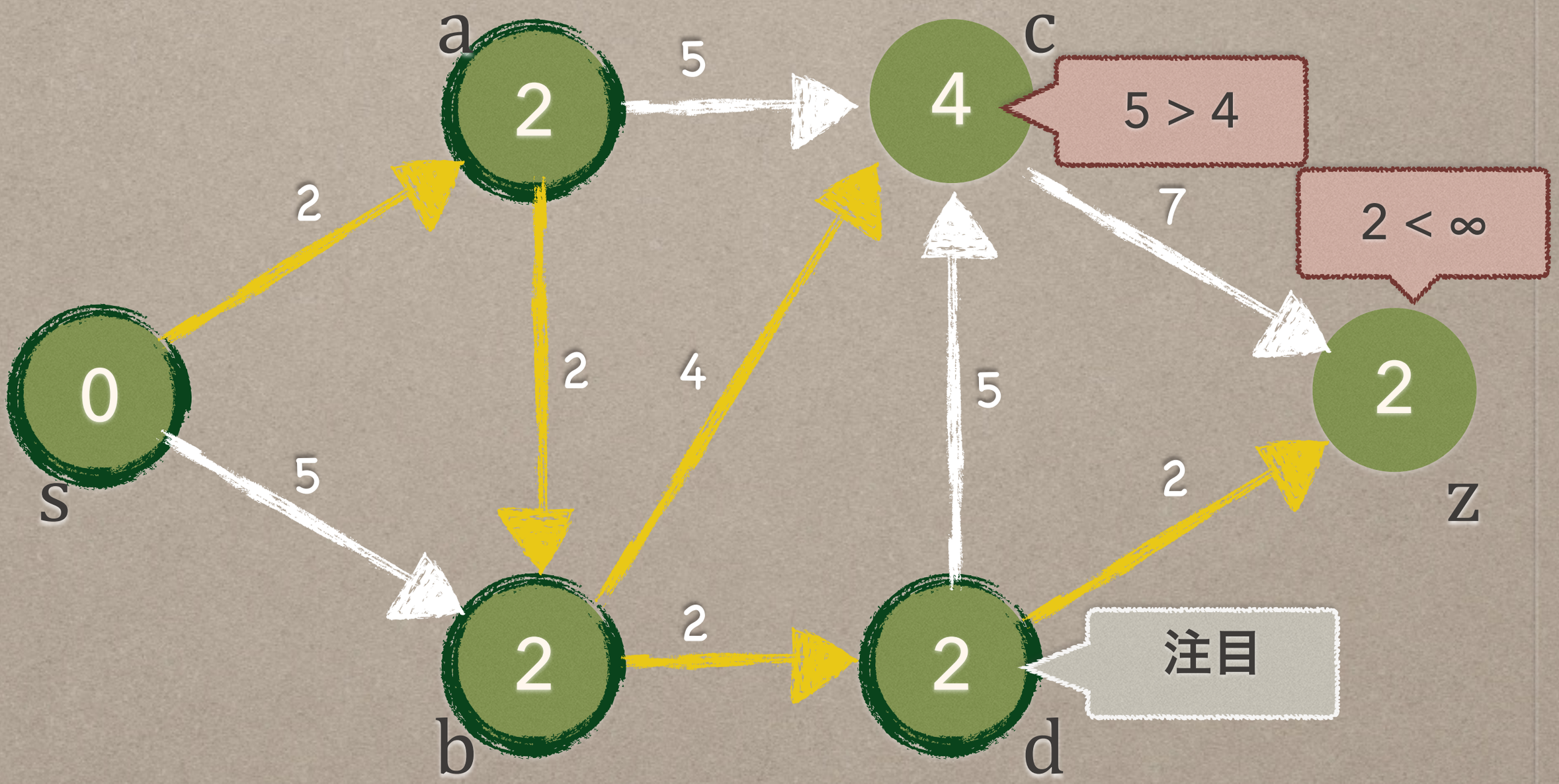
プリム法 (イメージ) 11/16

(2)印づけ、(3)アップデートを繰り返す



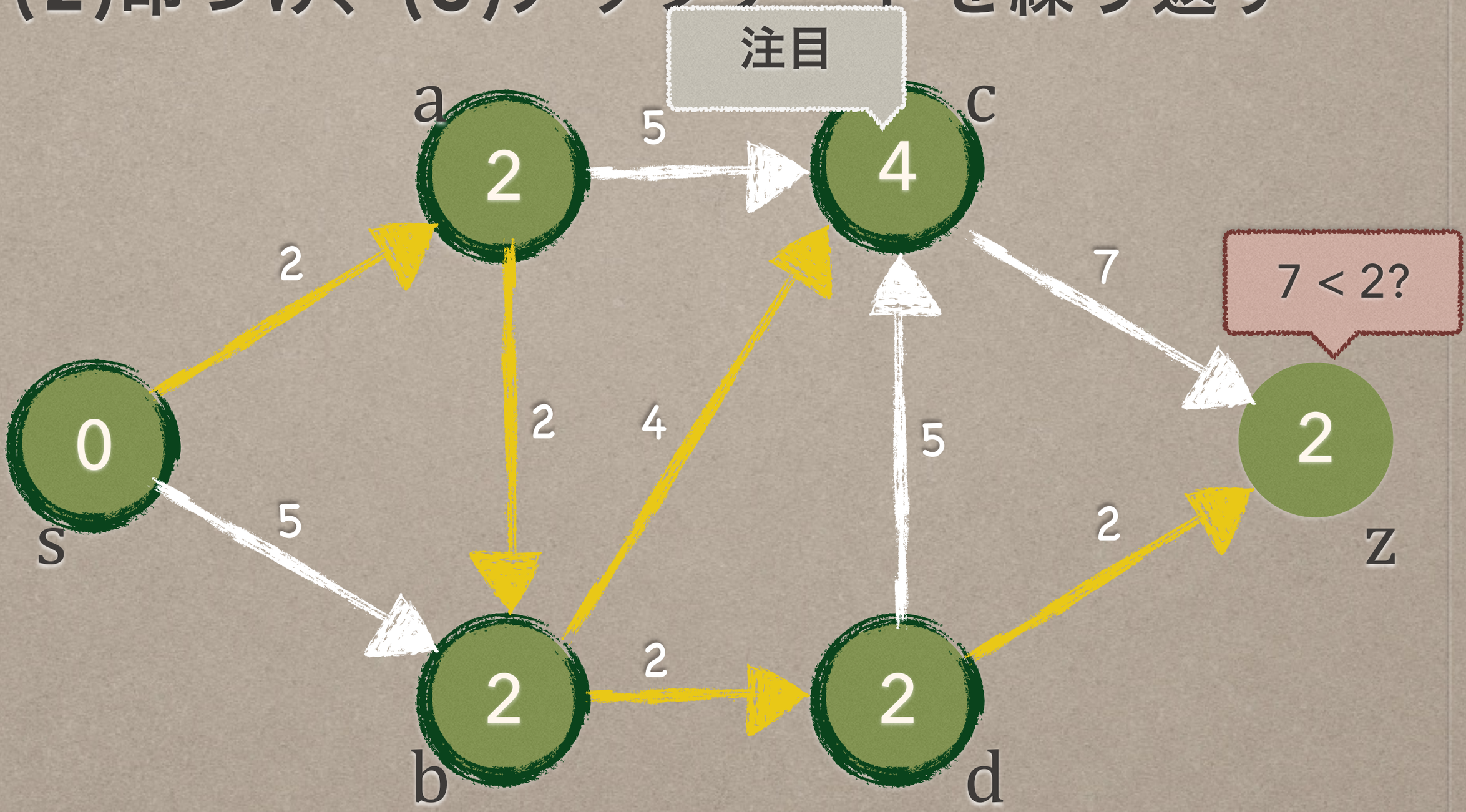
プリム法 (イメージ) 12/16

(2)印づけ、(3)アップデートを繰り返す



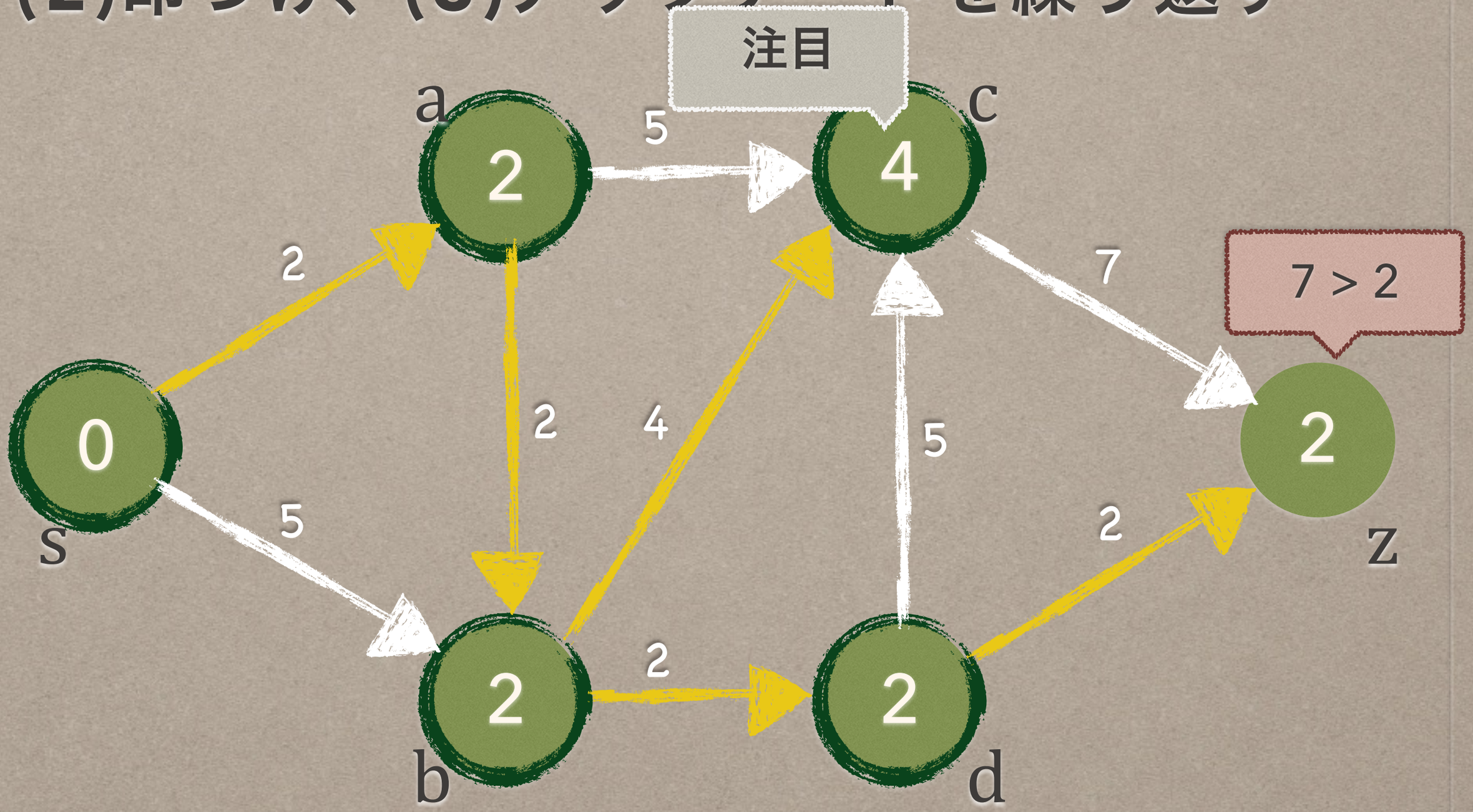
プリム法 (イメージ) 13/16

(2)印づけ、(3)アップデートを繰り返す



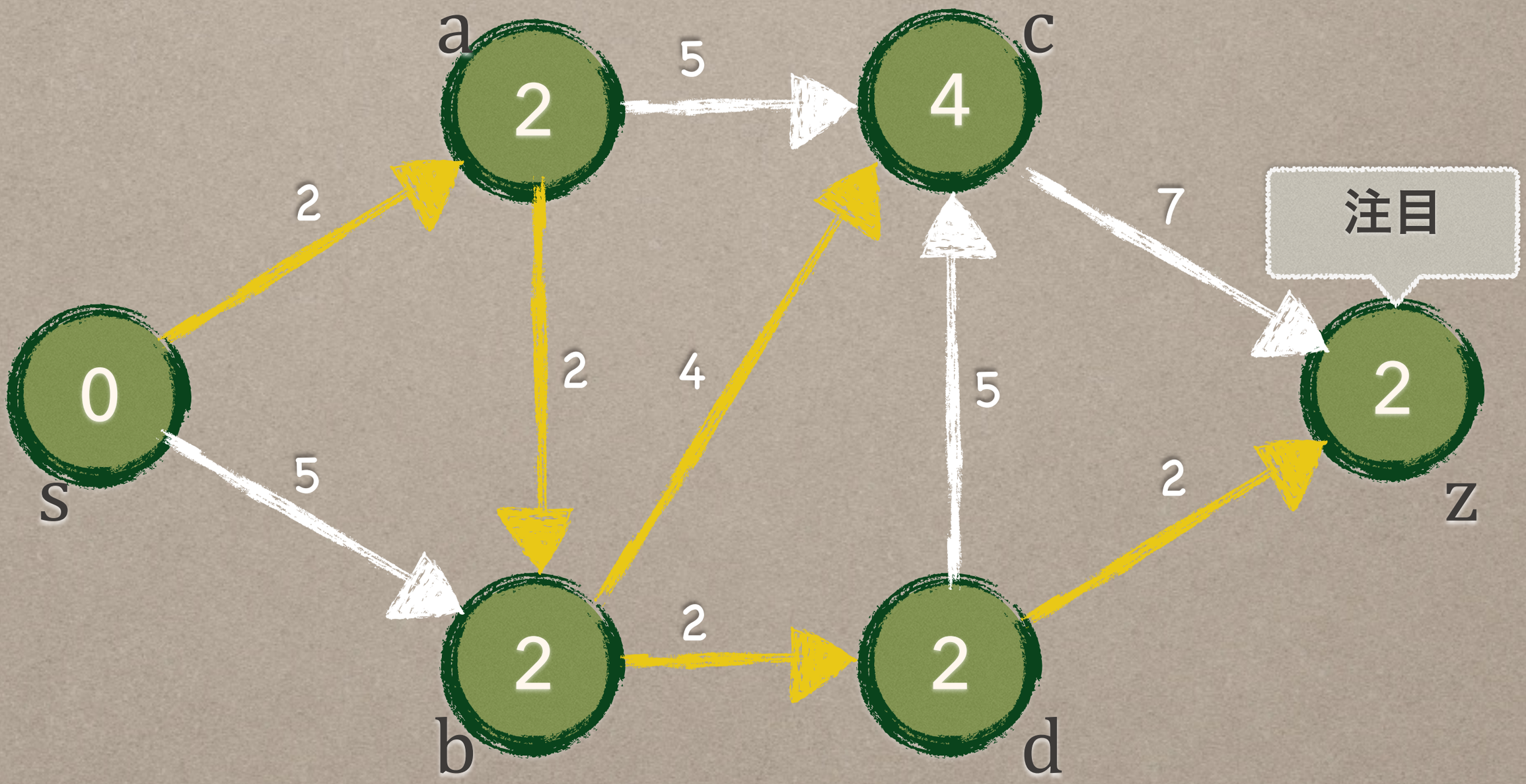
プリム法 (イメージ) 14/16

(2)印づけ、(3)アップデートを繰り返す

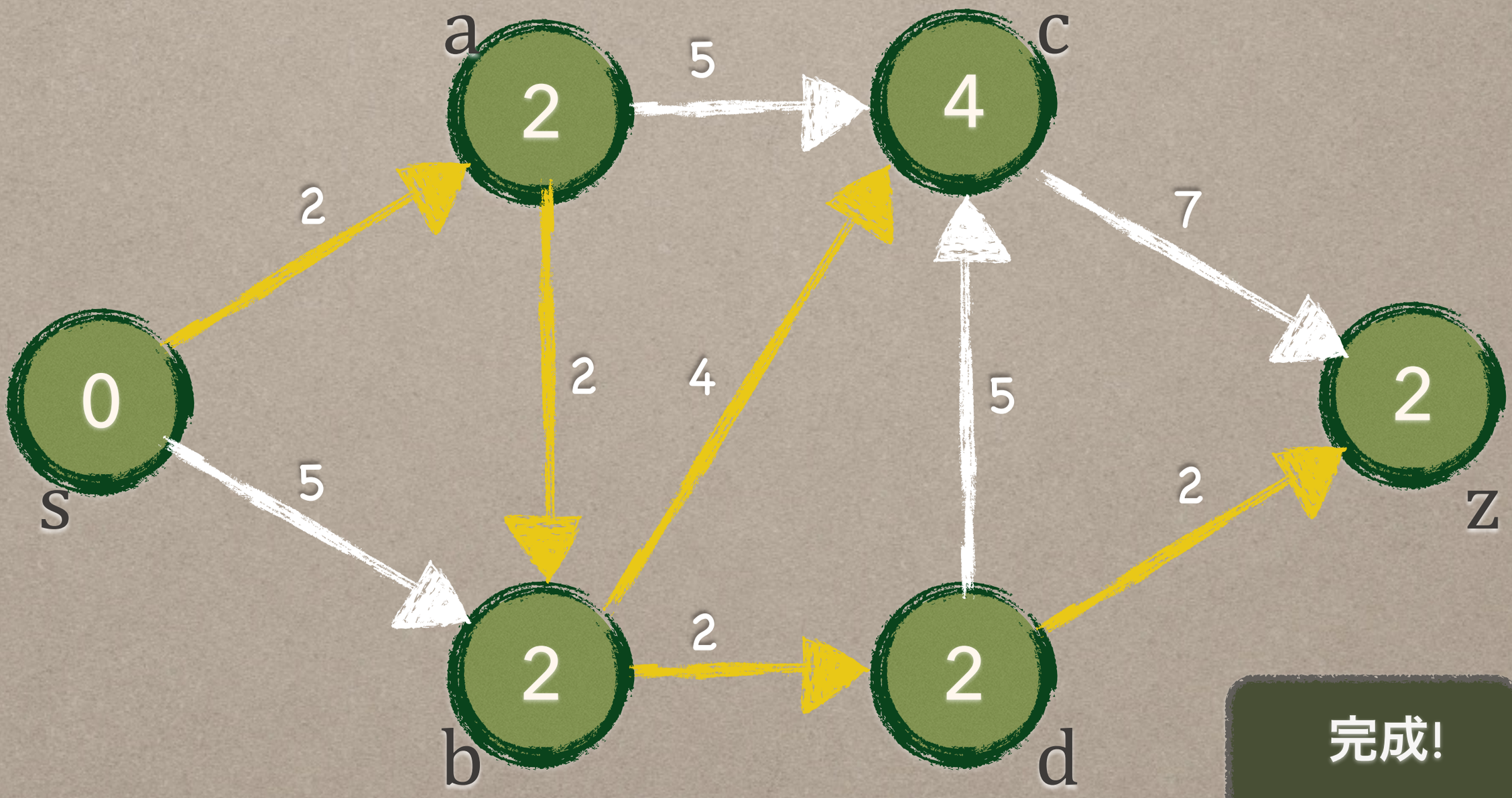


プリム法 (イメージ) 15/16

(2)印づけ、(3)アップデートを繰り返す



プリム法 (イメージ) 16/16
全員に印がついたら完成



完成!

プリム法 (まとめ)

入力: (有向グラフ, 始点ノード)

(1) 初期化する

(2) 「印」なしの最小コストのノードに注目、印をつける

(3) 隣接ノードをアップデート

- コストを評価し、「更新」できるならエッジに着色
- コスト = エッジ(来た道)のコスト

差分ここだけ

(4) (2)〜(3)を繰り返す

(5) 「色が塗られた路」を最小全域木として出力

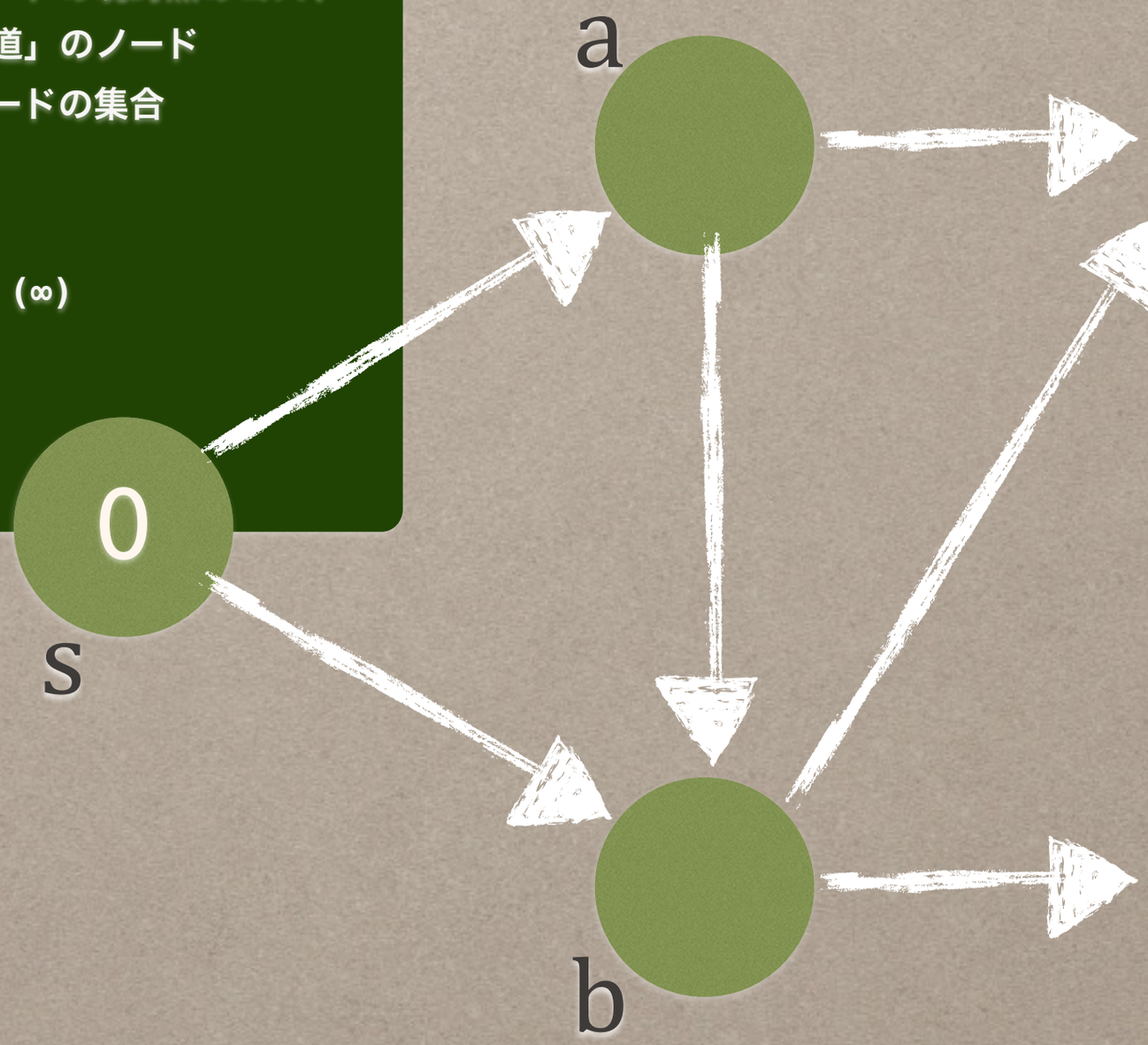
プリム法の実装例 (1/5)

ダイクストラ法と同じ

ノードのデータ表現(1)

```
class Node
  attr_accessor :name           # :s や :a, :z など
  attr_accessor :cost           # このノードの現時点のコスト
  attr_accessor :predecessor    # 「来た道」のノード
  attr_accessor :neighbors      # 隣接ノードの集合

  def initialize(name)
    @name = name
    @cost = Float::INFINITY     # 無限大 (∞)
    @predecessor = nil
    @neighbors = Hash.new
  end
end
```

```
# s の初期化の例
s = Node.new(:s)
s.cost = 0
s.predecessor = nil
s.neighbors[:a] = a
s.neighbors[:b] = b
```



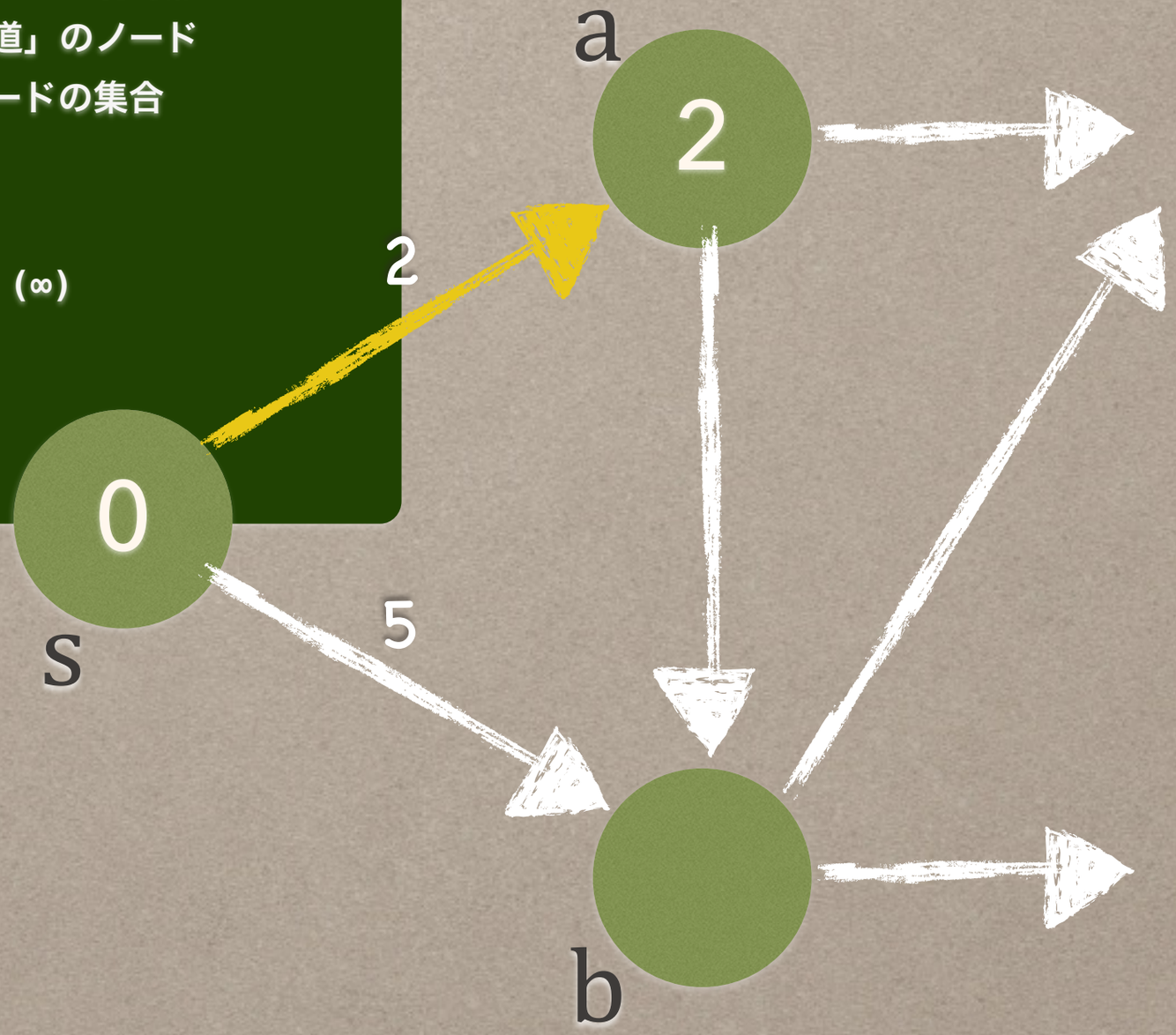
プリム法の実装例 (2/5)

ダイクストラ法と同じ

ノードのデータ表現(2)

```
class Node
  attr_accessor :name           # :s や :a, :z など
  attr_accessor :cost           # このノードの現時点のコスト
  attr_accessor :predecessor    # 「来た道」のノード
  attr_accessor :neighbors      # 隣接ノードの集合

  def initialize(name)
    @name = name
    @cost = Float::INFINITY     # 無限大 (∞)
    @predecessor = nil
    @neighbors = Hash.new
  end
end
```

```
# a の初期化の例
a = Node.new(:s)
a.cost = 2
a.predecessor = s     # 「色を塗る」
a.neighbors[:b] = b
a.neighbors[:c] = c
```



プリム法の実装例 (3/5)

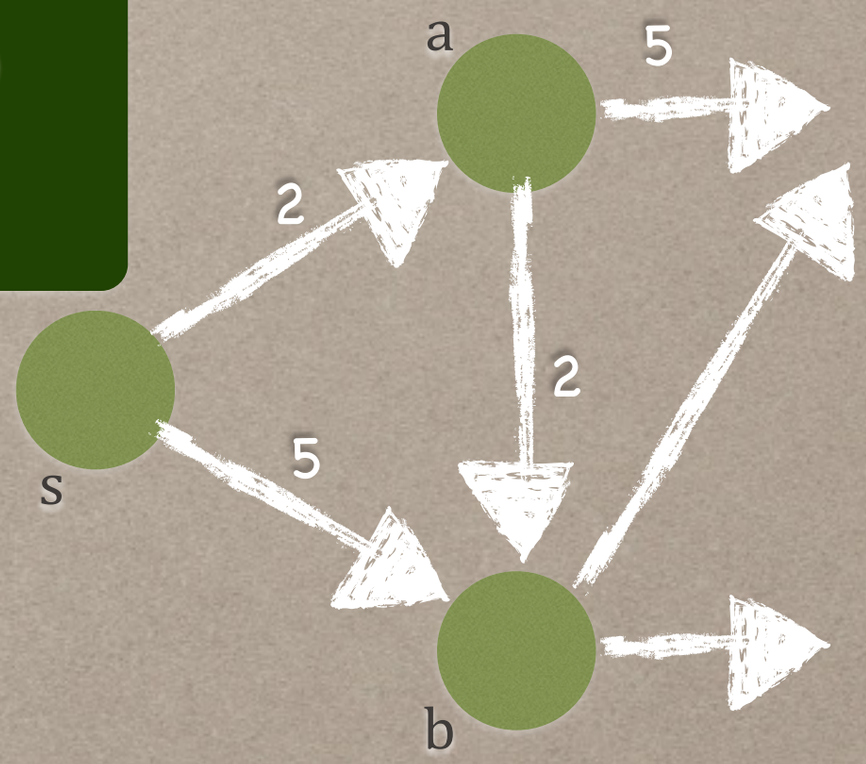
ダイクストラ法と同じ

グラフのデータ表現

```
class Graph
  attr_accessor :nodes    # このグラフの全ノード
  attr_accessor :edges    # このグラフの全エッジ

  def initialize(param)
    @nodes = Hash.new
    @edges = EdgeCollection.new  # ハッシュマップの入れ子クラス
    param.each do |src_name, dest_name, cost|
      node_src = @nodes[src_name] ||= Node.new(src_name)
      node_dest = @nodes[dest_name] ||= Node.new(dest_name)

      edge = @edges[src_name, dest_name] = Edge.new(cost)
      node_src.neighbors[node_dest.name] = edge
    end
  end
end
```

```
class Edge
  attr_accessor :cost

  def initialize(cost)
    @cost = cost
  end
end
```

```
# グラフの初期化例
graph = Graph.new([[:s, :a, 2],
                   [:s, :b, 5],
                   [:a, :b, 2],
                   [:a, :c, 5],
                    …… (略) ……
                   [:d, :z, 2]])
```



プリム法の実装例 (4/5)

ダイクストラ法と同じ

アルゴリズム(1)

```
class Prim
  def initialize(graph)
    @graph = graph
  end

  def calc(s)  # s は始点ノード
    all_nodes = @graph.nodes.values   # 全ノード
    unvisited_nodes = all_nodes - [s] # 印なしのノード群(sのみ確定済)

    # (1) 初期化
    s.cost = 0
    s.predecessor = nil

    # s 以外の全ノードのコストを設定
    unvisited_nodes.each do |node|
      node.cost = cost(s, node)
      node.predecessor = s    # 路に「色を塗る」
    end

    # …… (続く)
```

プリム法の実装例 (5/5)

アルゴリズム(2)

```
# ……（続き）
# 全てのノードに印が付くまで(2)〜(3)を繰り返す
while !unvisited_nodes.empty?
  # (2) 無印のうち、コストが最小のノードに注目(currentとする)
  current = unvisited_nodes.min {|x, y| x.cost <=> y.cost }
  unvisited_nodes.delete(current)  # ノードに「印をつける」

  # (3) 注目ノードの隣接ノード(neighbors)のコストをアップデート
  current.neighbors.each_key do |name|
    neighbor = @graph.nodes[name]
    cost_from_current = cost(current, neighbor)

    # currentからの方が近い場合、neighborのコストをアップデート
    if cost_from_current < neighbor.cost
      neighbor.cost = cost_from_current
      neighbor.predecessor = current
    end
  end
end
end
```

ダイクストラ法と同じ

```
# コスト評価関数(プリム法)
def cost(from, to)
  edge = \
    @graph.edges[from, to]

  if edge
    # from.cost + edge.cost
    edge.cost  # 差分
  else
    # 両者を接続するエッジが
    # ない場合、無限大を返す
    Float::INFINITY
  end
end
```

差分