Graphs and Greedy Algorithms

M269 Tutorial Resources 2022

Phil Molyneux

14 January 2023

M269 Tutorial Resources — Graph & Greedy Algorithms

Introduction (1)

- The following slides and notes are intended to provide some tutorial resources for the Graphs and Greedy algorithms material in M269
- There are also many worked examples using recursion and can be used for the recursion part of M269
- Material covered:
- Session on M269 Graph, Greedy & DP Algorithms
- Graph definitions and representations
- Python: List comprehensions, Named Tuples
- Topological Sort for directed acyclic graphs
- Dijkstra's Shortest Path Algorithm
- Prim's Minimum Spanning Tree Algorithm
- Greedy algorithms: Interval Scheduling
- Note that there is far more material here than could be covered in a single tutorial session

M269 Tutorial Resources — Graph & Greedy Algorithms

Introduction (2)

- **Materials**
- The file with name ending `.beamer.pdf` is the slides
- There is a version of that file in a folder named `AdobeConnect` scaled for Adobe Connect presentation — note that the links in the PDF file are trashed by the Adobe Connect conversion process
- The file with name ending `.article.pdf` is the notes version. Produced from the same sources, contains:
  - Table of Contents expanded
  - Bibliography (with back references)
  - Index of Python code
  - Index of diagrams
- The folders `DiagramsPDF` and `DiagramsPNG` contain PDF and PNG versions of the diagrams with tight pounding boxes (or equivalent) — suitable for inclusion in PowerPoint. Any transparent backgrounds have been preserved.
- Python code files — the line numbers in the slides or

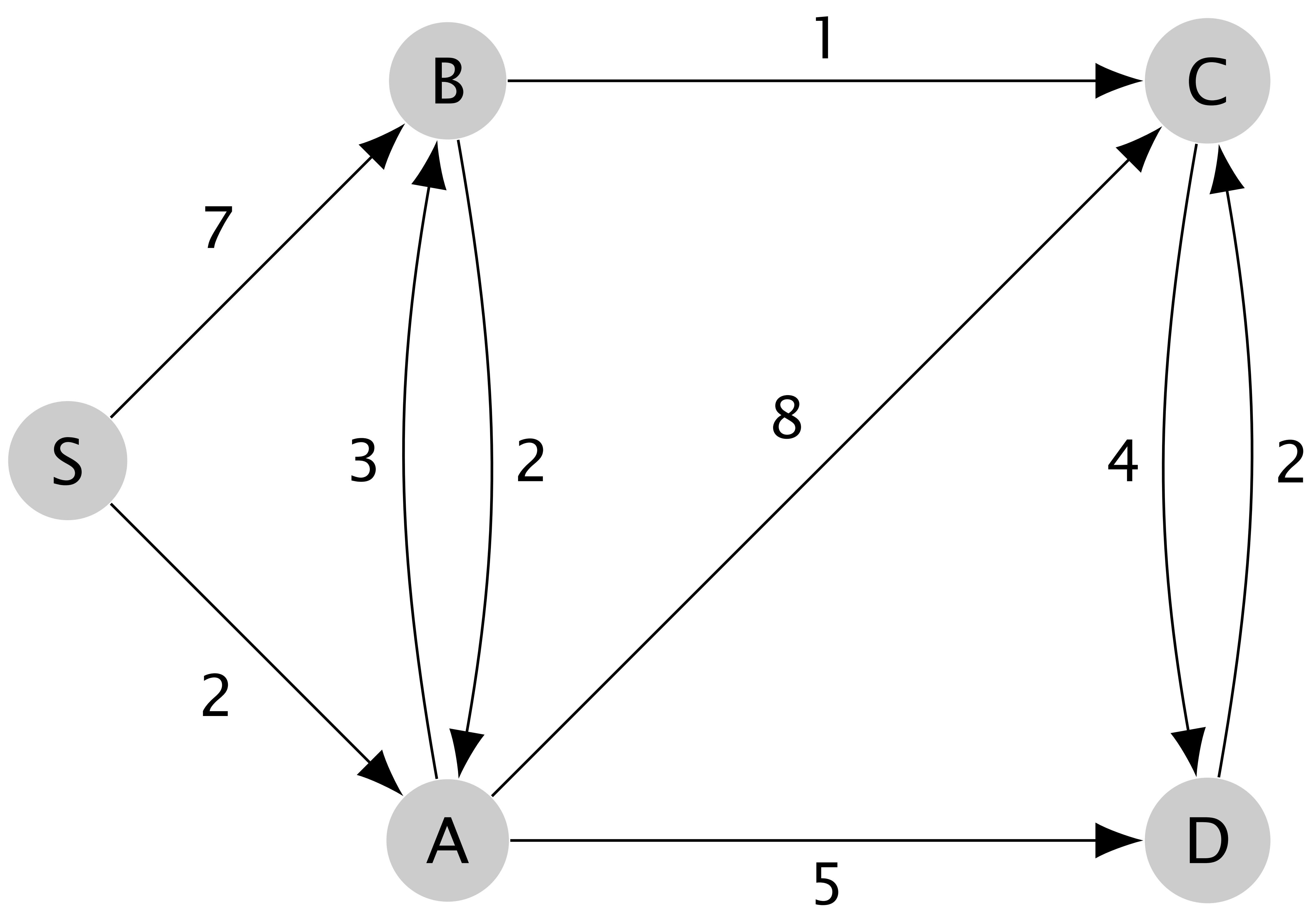
Graph Algorithms

Definitions

- A *Graph*, G, consists of a pair: a set of *vertices*, V, and a set of *edges*, E, where an edge (u, v) represents a connection between two vertices, u and v
- Equivalently, a graph is a set of objects together with a relation over that set
- Edges may have *direction* — that is, the relation is not symmetric — a graph with directed edges is called a *digraph*
- Informally, graphs are represented as diagrams (see below)
- If $G = (V, E)$ is a *weighted digraph* then there is a function $w :: E \to \mathbb{R}$ which maps edges to real numbers.
- If $e = (u, v)$ we write $w(u, v)$ for $w(e)$

Graph Algorithms

Example Digraph egDigraph



Graph Algorithms

Graph Representation

- What operations do we want on graphs ?
- How can we implement a representation of graphs and the operations efficiently ?
- Common representations
  - Adjacency list — a linear structure holds every vertex together with a list of successor vertices and the weights of the successor edges.
  - Adjacency matrix — 2 dimensional array of values of dimension $|V| \times |V|$ where both coordinates u and v are vertices and the entry (u, v) is the weight of the edge (if it exists)
- Additional points:
  - A vertex may have other data: name, label with data (shortest path predecessors, distance, . . .)
  - An edge may have other data: weight, status (on shortest path, minimum spanning tree, . . .)

Graph Algprithms

Activity 1 Graph Operations

- In the space below give a graph operation indicating whether it is a creator, inspector or modifier and give its pre and post conditions

Go to Answer

Graph Algprithms

Answer 1 Graph Operations

- Answer 1 Graph Operations — see next slide

Go to Activity

Graph Algorithms

Graph Operations 01

- *emptyGraph* returns an empty graph
- *mkGraph* takes a list of vertices, and a list of edges and returns a graph
- *isEmptyGraph* takes a graph and returns True if and only if the graph is empty.
- *vertices* takes a graph and returns the vertices
- *edges* takes a graph and returns the edges
- *succLists* takes a graph and returns a list of pairs of vertices and lists of successor edges
- *predLists* takes a graph and returns a list of pairs of vertices and lists of predecessor edges
- *startVertices* takes a graph and returns a list of vertices with no predecessors
- *endVertices* takes a graph and returns a list of vertices with no successors

Graph Algorithms

Graph Operations 02

- *removeVertex* takes a vertex and a graph and returns a graph with the vertex removed.
- Further service functions:
  - *esRemoveV* takes a vertex and a list of edges and returns the list of edges with the vertex removed.
  - *esStartV* takes a vertex and a list of edges and returns the list of edges where the given vertex is the start of an edge
  - *esEndV* takes a vertex and a list of edges and returns the list of edges where the given vertex is the end of an edge

Graph Algorithms

Graph Representation 01

- `Adjacency matrix` Assign a unique label to each vertex and construct an $n \times n$ matrix of values in which (i, j) is x if $(i, j) \in E$ and x is its label, (i, i) is 0 and all other entries are ∞
- The adjacency matrix for the previous example digraph is:

	S	A	B	C	D
S	0	2	7	∞	∞
A	∞	0	3	8	5
B	∞	2	0	1	∞
C	∞	∞	∞	0	4
D	∞	∞	∞	2	0

Graph Algorithms

Graph Representation 02

- The explicit adjacency list or matrix representations are biased towards the procedural view of programming.
- A functional view looks for an *inductive* definition (as we had with trees)
- Functional view:
  - A graph is either the empty graph or
  - a graph extended by a new node v together with its label and with edges to those of v's successors and predecessors *that are already in the graph*
- See FGL — A Functional Graph Library and Erwig (2001)
- M269 Python examples use *adjacency lists* to represent graphs.
- The Haskell examples in these notes use a simple (but inefficient) representation to illustrate the algorithms.

Algorithm Descriptions & Implementations

Overview

- The algorithms are described in a mix of Structured English, Python and Haskell
- The Python and Haskell code does not use any advanced features but may use some features not mentioned in M269
- In Python the code may use:
  - List comprehensions (tutorial), List comprehensions (reference) — a neat way of expressing iterations over a list, came from Miranda
  - Named tuples — a *Factory Function* for tuple with named fields — *quick & dirty* objects
- The Haskell syntax is defined as it is used — novel concepts may be:
  - Algebraic Data Types — just name your user defined data type and name its elements — magic!
  - Explicit type specifications — Haskell has a very powerful type system that can help spot errors.
  - List comprehensions — as above

List Comprehensions

Python

- **List Comprehensions** provide a concise way of performing calculations over lists (or other iterables)
- Example: Square the even numbers between 0 and 9

```
Python3>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
Python3>>> [(x,y) for x in range(4)
...               for y in range(4)
...               if x % 2 == 0
...                  and y % 3 == 0]
[(0, 0), (0, 3), (2, 0), (2, 3)]
Python3>>>
```

- In general

```
  [expr for target1 in iterable1 if cond1
        for target2 in iterable2 if cond2 ...
        for targetN in iterableN if condN ]
```

- Lots example usage in the algorithms below

List Comprehensions

Haskell

- **List Comprehensions** provide a concise way of performing calculations over lists
- Example: Square the even numbers between 0 and 9

```
GHCi> [x^2 | x <- [0..9], x `mod` 2 == 0]
[0,4,16,36,64]
GHCi>
```

- In general

```
[expr | qual1, qual2,..., qualN]
```

- The qualifiers `qual` can be
  - Generators `pattern <- list`
  - Boolean guards — acting as filters
  - Local declarations with `let decls` for use in `expr` and later generators and boolean guards

List Comprehension Exercises

Activity 2 (a) Stop Words Filter

- Stop words are the most common words that most search engines avoid: `'a','an','the','that',...`
- Using list comprehensions, write a function `filterStopWords` that takes a list of words and filters out the stop words
- Here is the initial code

```
sentence \
 = "the␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog"

words = sentence.split()

wordsTest \
 = (words == ['the', 'quick', 'brown'
             , 'fox', 'jumps', 'over'
             , 'the', 'lazy', 'dog'])

stopWords \
 = ['a','an','the','that']
```

Go to Answer

List Comprehension Exercises

Activity 2 (a) Stop Words Filter

```
sentence \
 = "the␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog"

words = sentence.split()

wordsTest \
 = (words == ['the', 'quick', 'brown'
             , 'fox', 'jumps', 'over'
             , 'the', 'lazy', 'dog'])

stopWords \
 = ['a','an','the','that']
```

- Notice the Python Explicit line joining with (\<nl>) and Python Implicit line joining with ((...))
- The backslash (\) must be followed by an end of line character (<nl>)
- The ('␣') symbol represents a space (see Unicode U+2423 Open Box)

Go to Answer

List Comprehension Exercises

Activity 2 (b) Transpose Matrix

- A matrix can be represented as a list of rows of numbers
- We *transpose* a matrix by swapping columns and rows
- Here is an example

```
matrixA \
 = [[1, 2, 3, 4]
   ,[5, 6, 7 ,8]
   ,[9, 10, 11, 12]]

matATr \
 = [[1, 5, 9]
   ,[2, 6, 10]
   ,[3, 7, 11]
   ,[4, 8, 12]]
```

- Using list comprehensions, write a function `transMat`, to transpose a matrix

Go to Answer

List Comprehension Exercises

Activity 2 (c) List Pairs in Fair Order

- Write a function which takes a pair of positive integers and outputs a list of all possible pairs in those ranges
- If we do this in the simplest way we get a bias to one argument
- Here is an example of a bias to the second argument

```
yBiasLstTest \
 = (yBiasListing(5,5)
     == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])
```

▸ Go to Answer

List Comprehension Exercises

Activity 2 (c) List Pairs in Fair Order

- Rewrite the function which takes a pair of positive integers and outputs a list of all possible pairs in those ranges
- The output should treat each argument *fairly* — any initial prefix should have roughly the same number of instances of each argument
- Here is an example output

```
fairLstTest \
 = (fairListing(5,5)
     == [(0, 0)
         , (0, 1), (1, 0)
         , (0, 2), (1, 1), (2, 0)
         , (0, 3), (1, 2), (2, 1), (3, 0)
         , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])
```

Go to Answer

List Comprehension Exercises

Activity 2 (c) List Pairs in Fair Order

- Rewrite the function which takes a pair of positive integers and outputs a list of lists of all possible pairs in those ranges
- The output should treat each argument *fairly* — any initial prefix should have roughly the same number of instances of each argument — further, the output should be segment by each initial prefix (see example below)
- Here is an example output

```
fairLstATest \
 = (fairListingA(5,5)
     == [[(0, 0)]
         , [(0, 1), (1, 0)]
         , [(0, 2), (1, 1), (2, 0)]
         , [(0, 3), (1, 2), (2, 1), (3, 0)]
         , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])
```

Go to Answer

List Comprehension Exercises

Answer 2 (a) Stop Words Filter

- Answer 2 (a) Stop Words Filter
- *Write here:*
- Answer 2 continued on next slide

Go to Activity

List Comprehension Exercises

Answer 2 (a) Stop Words Filter

- Answer 2 (a) Stop Words Filter

```python
def filterStopWords(words) :
  nonStopWords \
   = [word for word in words
           if word not in stopWords]
  return nonStopWords


filterStopWordsTest \
 = filterStopWords(words) \
    == ['quick', 'brown', 'fox'
      , 'jumps', 'over', 'lazy', 'dog']
```

Go to Activity

List Comprehension Exercises

Answer 2 (b) Transpose Matrix

- Answer 2 (b) Transpose Matrix
- *Write here:*
- Answer 2 continued on next slide

Go to Activity

List Comprehension Exercises

Answer 2 (b) Transpose Matrix

- Answer 2 (b) Transpose Matrix

```
def transMat(mat) :
  rowLen = len(mat[0])
  matTr \
   = [[row[i] for row in mat] for i in range(rowLen)]
  return matTr

transMatTestA \
 = (transMat(matrixA)
     == matATr)
```

- Note that a list comprehension is a valid expression as a target expression in a list comprehension
- The code assumes every row is of the same length
- Answer 2 continued on next slide

Go to Activity

List Comprehension Exercises

Answer 2 (b) Transpose Matrix

- Note the differences in the list comprehensions below

```
matrixA \
 = [[1, 2, 3, 4]
   ,[5, 6, 7 ,8]
   ,[9, 10, 11, 12]]
```

```
Python3>>> [[row[i] for row in matrixA]
...                 for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
Python3>>> [row[i] for row in matrixA
...                for i in range(4)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Python3>>> [row[i] for i in range(4)
...                for row in matrixA]
[1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12]
Python3>>> [[row[i] for i in range(4)]
...                 for row in matrixA]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Go to Activity

List Comprehension Exercises

Answer 2 (b) Transpose Matrix

- Answer 2 (b) Transpose Matrix
- The Python NumPy package provides functions for N-dimensional array objects
- For transpose see numpy.ndarray.transpose

```
Python3>>> import numpy as np
Python3>>> ar = np.array([[1,2],[3,4]])
Python3>>> ar
array([[1, 2],
       [3, 4]])
Python3>>> arT = ar.transpose()
Python3>>> arT
array([[1, 3],
       [2, 4]])
Python3>>> ar
array([[1, 2],
       [3, 4]])
Python3>>> ar.shape
(2, 2)
```

Go to Activity

List Comprehension Exercises

Answer 2 (c) List Pairs in Fair Order

- Answer 2 (c) List Pairs in Fair Order — first version
- Write here

```
yBiasLstTest \
 = (yBiasListing(5,5)
     == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])
```

Go to Activity

List Comprehension Exercises

Answer 2 (c) List Pairs in Fair Order

- Answer 2 (c) List Pairs in Fair Order
- This is the *obvious* but biased version

```
def yBiasListing(xRng,yRng) :
  yBiasLst \
   = [(x,y) for x in range(xRng)
            for y in range(yRng)]
  return yBiasLst

yBiasLstTest \
 = (yBiasListing(5,5)
     == [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)
        , (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)
        , (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)
        , (3, 0), (3, 1), (3, 2), (3, 3), (3, 4)
        , (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)])
```

Go to Activity

List Comprehension Exercises

Answer 2 (c) List Pairs in Fair Order

- Answer 2 (c) List Pairs in Fair Order — second version
- Write here

```
fairLstTest \
 = (fairListing(5,5)
     == [(0, 0)
         , (0, 1), (1, 0)
         , (0, 2), (1, 1), (2, 0)
         , (0, 3), (1, 2), (2, 1), (3, 0)
         , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)])
```

Go to Activity

List Comprehension Exercises

Answer 2 (c) List Pairs in Fair Order

- Answer 2 (c) List Pairs in Fair Order — second version
- This works by making the sum of the coordinates the same for each prefix

```
def fairListing(xRng,yRng) :
  fairLst \
   = [(x,d-x) for d in range(yRng)
              for x in range(d+1)]
  return fairLst

fairLstTest \
 = (fairListing(5,5)
     == [(0, 0)
        , (0, 1), (1, 0)
        , (0, 2), (1, 1), (2, 0)
        , (0, 3), (1, 2), (2, 1), (3, 0)
        , (0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]
```

Go to Activity

List Comprehension Exercises

Answer 2 (c) List Pairs in Fair Order

- Answer 2 (c) List Pairs in Fair Order — third version
- Write here

```
fairLstATest \
 = (fairListingA(5,5)
     == [[(0, 0)]
         , [(0, 1), (1, 0)]
         , [(0, 2), (1, 1), (2, 0)]
         , [(0, 3), (1, 2), (2, 1), (3, 0)]
         , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])
```

Go to Activity

List Comprehension Exercises

Answer 2 (c) List Pairs in Fair Order

- Answer 2 (c) List Pairs in Fair Order — third version
- The *inner loop* is placed into its own list comprehension

```
def fairListingA(xRng,yRng) :
  fairLstA \
   = [[(x,d-x) for x in range(d+1)]
               for d in range(yRng)]
  return fairLstA

fairLstATest \
 = (fairListingA(5,5)
     == [[(0, 0)]
        , [(0, 1), (1, 0)]
        , [(0, 2), (1, 1), (2, 0)]
        , [(0, 3), (1, 2), (2, 1), (3, 0)]
        , [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]])
```

Go to Activity

Algorithm Descriptions & Implementations

Python & Haskell Tutorials

- Python tutorials:
  - Beginner's Python Tutorial
  - Python Programming
  - Non-Programmer's Tutorial for Python 3
  - Non-Programmer's Tutorial for Python 2.6
- Haskell Tutorials:
  - Haskell Wikibook
  - What I Wish I Knew When Learning Haskell
  - Haskell Meta-tutorial
  - Learn You a Haskell for Great Good
  - Real World Haskell

Graph Representation

Python

```
from collections import namedtuple

Vertex = namedtuple('Vertex'
                    ,['vtxName'])

Edge = namedtuple('Edge'
                  ,['startVtx','endVtx'])
```

- This is from Python/M269TutorialGraphs2020J.py
- Reserved identifiers are shown in this color
- User defined data constructors such as Vertex and Edge are shown in that color
- Vertex is a *named tuple* with named fields — a *quick and dirty* object — recommended by Guido van Rossum
- **Health Warning:** these notes may not be totally consistent with syntax colouring.

Example Graphs

Python

```
ta = Vertex('TA')
tb = Vertex('TB')
tc = Vertex('TC')
td = Vertex('TD')
te = Vertex('TE')
tf = Vertex('TF')
tg = Vertex('TG')
th = Vertex('TH')

eg01Vs = [ta,tb,tc,td,te,tf,tg,th]

eg01Es = [(ta,tb),(tg,tb),(tg,th),(tb,tc)
         ,(tb,tf),(tf,th),(tc,td),(td,te),(te,th)]

eg01Gr = (eg01Vs, eg01Es)

eg02Es = [(ta,tb),(tb,tc),(tc,ta)] # cycles

eg02Gr = ([ta,tb,tc], eg02Es)
```

- Used ordinary tuples for edges here

Graph Service Functions

Python (1)

```
def vertices(gr):
  return gr[0]

def edges(gr):
  return gr[1]

def esStartV(v,es):
  return [edge for edge in es if edge[0] == v]

def esEndV(v,es):
  return [edge for edge in es if edge[1] == v]

def esRemoveV(v,es):
  return [edge for edge in es
                 if edge[0] != v and edge[1] != v]
```

- Choice of service function (or class methods) is a design issue — a bit of a fudge here (to avoid complexity in these notes)

Graph Service Functions

Python (2)

```
def succLists(gr):
  return [(v, esStartV(v, (edges(gr))))
          for v in vertices(gr)]

def predLists(gr):
  return [(v, esEndV(v, (edges(gr))))
          for v in vertices(gr)]

def isEmptyGraph(gr):
  return gr[0] == [] and gr[1] == []

def startVertices(gr):
  return [pLst[0] for pLst in predLists(gr)
                  if pLst[1] == []]

def endVertices(gr):
  return [sLst[0] for sLst in succLists(gr)
                  if sLst[1] == []]
```

Graph Service Functions

Python (3)

```
74 def removeVertex(v, gr):
75   vs = gr[0]
76   vs1 = vs[:]
77   if v in vs1:
78     vs1.remove(v)
79   es = gr[1]
80   es1 = esRemoveV(v,es)
81   return (vs1,es1)
```

- Note that vs1 at line 76 is a (shallow) copy of vs
- If vertices had more structure we might have to write a function to do a proper copy

Python Graph Representation from 21J

Graph Representation Choices

- A graph is a pair of sets of nodes and edges, possibly with information attached to nodes and edges such as labels, weights, durations or distances — this is the mathematical view of graphs
- Algorithms also need to consider representations for the efficiency of the operations — M269 discusses several graph representations:
- *Edge list representation*
- *Adjacency matrix representation*
- *Adjacency list representation*
- The implementation is given for *directed graphs* or *digraphs* and *undirected graphs* using adjacency list representations

Python 21J Adjacency List Representation

DiGraph Class

- The following code is from M269TutorialGraphs2021JDigraph.py which is from m269_digraph.py modified only for layout

```
import networkx
from typing import Hashable

class DiGraph:
  """A directed graph with hashable node objects.

  Edges are between different nodes.
  There's at most one edge from one node to another.
  """
```

DiGraph Class

Constructor, Inspectors

```python
def __init__(self):
  self.out = dict()   # a map of nodes to their out-neighbours

def has_node(self, node: Hashable) -> bool:
  """Return True if and only if the graph has the node."""
  return node in self.out

def has_edge(self, start: Hashable, end: Hashable) -> bool:
  """Return True if and only if edge start -> end exists.

  Preconditions: self.has_node(start) and self.has_node(end)
  """
  return end in self.out[start]
```

DiGraph Class

Add Node,Edge

```python
def add_node(self, node: Hashable) -> None:
    """Add the node to the graph.

    Preconditions: not self.has_node(node)
    """
    self.out[node] = set()

def add_edge(self, start: Hashable, end: Hashable) -> None:
    """Add edge start -> end to the graph.

    If the edge already exists, do nothing.

    Preconditions:
    self.has_node(start) and self.has_node(end) and start != end
    """
    self.out[start].add(end)
```

- Note add is a set method that does not raise an error if the argument is a node already present

DiGraph Class

Remove Node,Edge

```python
def remove_node(self, node: Hashable) -> None:
  """Remove the node and all its attached edges.

  Preconditions: self.has_node(node)
  """
  self.out.pop(node)
  for start in self.out:
    self.remove_edge(start, node)

def remove_edge(self, start: Hashable, end: Hashable) -> None:
  """Remove edge start -> end from the graph.

  If the edge doesn't exist, do nothing.

  Preconditions: self.has_node(start) and self.has_node(end)
  """
  self.out[start].discard(end)
```

- Note `discard` is a `set` method that does not raise an error if the argument is a node that is not present
- `pop` is a `dict` and a `set` operation
- Note this version of `remove_node` has a bug — remove the edges to the node first

DiGraph Class

Get Nodes,Edges

```
  def nodes(self) -> set:
    """Return the graph's nodes."""
    all_nodes = set()
    for node in self.out:
      all_nodes.add(node)
    return all_nodes

  def edges(self) -> set:
    """Return the graph's edges as a set of pairs (start, end)."""
    all_edges = set()
    for start in self.out:
      for end in self.out[start]:
        all_edges.add( (start, end) )
    return all_edges
```

DiGraph Class

Out Neighbours, Degrees

```
def out_neighbours(self, node: Hashable) -> set:
    """Return the out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return set(self.out[node])  # return a copy

def out_degree(self, node: Hashable) -> int:
    """Return the number of out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return len(self.out[node])
```

DiGraph Class

In Neighbours, Degrees

```python
def in_neighbours(self, node: Hashable) -> set:
  """Return the in-neighbours of the node.

  Preconditions: self.has_node(node)
  """
  start_nodes = set()
  for start in self.out:
    if self.has_edge(start, node):
      start_nodes.add(start)
  return start_nodes

def in_degree(self, node: Hashable) -> int:
  """Return the number of in-neighbours of the node.

  Preconditions: self.has_node(node)
  """
  return len(self.in_neighbours(node))
```

DiGraph Class

Neighbours, Degree

```python
def neighbours(self, node: Hashable) -> set:
    """Return the in- and out-neighbours of the node.

    Preconditions: self.has_node(node)
    """
    return self.out_neighbours(node).union(self.in_neighbours(node))

def degree(self, node: Hashable) -> int:
    """Return the number of in- and out-going edges of the node.

    Preconditions: self.has_node(node)
    """
    return self.in_degree(node) + self.out_degree(node)
```

DiGraph Class

Draw DiGraph

```
def draw(self) -> None:
  """Draw the graph."""
  if type(self) == DiGraph:
    graph = networkx.DiGraph()
  else:
    graph = networkx.Graph()
  graph.add_nodes_from(self.nodes())
  graph.add_edges_from(self.edges())
  networkx.draw(graph, with_labels=True,
    node_size=1000, node_color='lightblue',
    font_size=12, font_weight='bold')
```

DiGraph Class

Breadth First Search

```python
from collections import deque

def bfs(graph: DiGraph, start: Hashable) -> DiGraph:
  """Return the subgraph traversed by a breadth-first search.

  Preconditions: graph.has_node(start)
  """
  # changes from traversed function noted in comments
  visited = DiGraph()
  visited.add_node(start)
  unprocessed = deque()                          # set -> deque
  for neighbour in graph.out_neighbours(start):
    unprocessed.append( (start, neighbour) )    # add -> append
  while len(unprocessed) > 0:
    edge = unprocessed.popleft()                # pop -> popleft
    previous = edge[0]
    current = edge[1]
    if not visited.has_node(current):
      visited.add_node(current)
      visited.add_edge(previous, current)
      for neighbour in graph.out_neighbours(current):
        unprocessed.append( (current, neighbour) )  # add -> append
  return visited
```

DiGraph Class

Depth First Search

```python
def dfs(graph: DiGraph, start: Hashable) -> DiGraph:
  """Return the subgraph traversed by a depth-first search.

  Preconditions: graph.has_node(start)
  """
  visited = DiGraph()
  visited.add_node(start)
  unprocessed = []                          # deque -> list
  for neighbour in graph.out_neighbours(start):
    unprocessed.append( (start, neighbour) )
  while len(unprocessed) > 0:
    edge = unprocessed.pop()                # popleft -> pop
    previous = edge[0]
    current = edge[1]
    if not visited.has_node(current):
      visited.add_node(current)
      visited.add_edge(previous, current)
      for neighbour in graph.out_neighbours(current):
        unprocessed.append( (current, neighbour) )
  return visited
```

Weighted DiGraph Class

Initial Code, Add Node, Edge

```
import math

class WeightedDiGraph(DiGraph):
  """A weighted directed graph with hashable node objects.

  Edges are between different nodes.
  There's at most one edge from one node to another.
  Edges have weights, which can be floats or integers.
  """

  def add_node(self, node: Hashable) -> None:
    """Add the node to the graph.

    Preconditions: not self.has_node(node)
    """
    self.out[node] = dict() # a map of out-neighbours to weights

  def add_edge(self, start: Hashable, end: Hashable, weight: float) -> None:
    """Add edge start -> end, with the given weight, to the graph.

    If the edge already exists, set its weight.

    Preconditions:
    self.has_node(start) and self.has_node(end) and start != end
    """
    self.out[start][end] = weight
```

Weighted DiGraph Class

Weight, Remove Edge

```python
  def weight(self, start: Hashable, end: Hashable) -> float:
    """Return the weight of edge start -> end or infinity if it doesn't exist

    Preconditions: self.has_node(start) and self.has_node(end)
    """
    if self.has_edge(start, end):
      return self.out[start][end]
    else:
      return math.inf

  def remove_edge(self, start: Hashable, end: Hashable) -> None:
    """Remove edge start -> end from the graph.

    If the edge doesn't exist, do nothing.

    Preconditions: self.has_node(start) and self.has_node(end)
    """
    if self.has_edge(start, end):
      self.out[start].pop(end)
```

Weighted DiGraph Class

Weight, Remove Edge

```python
def edges(self) -> set:
  """Return the graph's edges as a set of triples (start, end, weight)."""
  all_edges = set()
  for start in self.out:
    for (end, weight) in self.out[start].items():
      all_edges.add( (start, end, weight) )
  return all_edges

def out_neighbours(self, node: Hashable) -> set:
  """Return the out-neighbours of the node.

  Preconditions: self.has_node(node)
  """
  return set(self.out[node].keys())
```

Weighted DiGraph Class

Draw

```
  def draw(self) -> None:
    """Draw the graph."""
    if type(self) == WeightedDiGraph:
      graph = networkx.DiGraph()
    else:
      graph = networkx.Graph()
    graph.add_nodes_from(self.nodes())
    for (node1, node2, weight) in self.edges():
      graph.add_edge(node1, node2, w=weight)
    pos = networkx.spring_layout(graph)
    networkx.draw(graph, pos, with_labels=True,
      node_size=1000, node_color='lightblue',
      font_size=12, font_weight='bold')
    networkx.draw_networkx_edge_labels(graph, pos,
      edge_labels=networkx.get_edge_attributes(graph, 'w'))
```

Weighted DiGraph Class

Shortest Path: Dijkstra (1)

```python
from heapq import heappush, heappop

def dijkstra(graph: WeightedDiGraph, start: Hashable) -> WeightedDiGraph:
    """Return a shortest path from start to each reachable node.

    Preconditions:
    - graph.has_node(start)
    - node objects are comparable
    - no weight is negative
    """
    visited = WeightedDiGraph()
    visited.add_node(start)
```

- Shortest Path Dijkstra continued on next slide

Weighted DiGraph Class

Shortest Path: Dijkstra (2)

```
# create min-priority queue of tuples (cost, (A, B, weight))
# cost is total weight from start to B via shortest path to A
unprocessed = []    # min-priority queue
for neighbour in graph.out_neighbours(start):
  weight = graph.weight(start, neighbour)
  heappush(unprocessed, (weight, (start, neighbour, weight)) )

while len(unprocessed) > 0:
  info = heappop(unprocessed)
  cost = info[0]
  edge = info[1]
  previous = edge[0]
  current = edge[1]
  weight = edge[2]

  if not visited.has_node(current):
    visited.add_node(current)
    visited.add_edge(previous, current, weight)
    for neighbour in graph.out_neighbours(current):
      weight = graph.weight(current, neighbour)
      edge = (current, neighbour, weight)
      heappush(unprocessed, (cost + weight, edge) )
return visited
```

Python 21J Adjacency List Representation

Undirected Graph Class

- The following code is from `M269TutorialGraphs2021JUngraph.py` which is from `m269_ungraph.py` modified only for layout

```python
from typing import Hashable

class UndirectedGraph(DiGraph):
  """An undirected graph with hashable node objects.

  There's at most one edge between two different nodes.
  There are no edges between a node and itself.
  """
```

Undirected Graph Class

Add and Remove Edge

```python
    def add_edge(self, node1: Hashable, node2: Hashable) -> None:
        """Add an undirected edge node1-node2 to the graph.

        If the edge already exists, do nothing.

        Preconditions: self.has_node(node1) and self.has_node(node2)
        """
        super().add_edge(node1, node2)
        super().add_edge(node2, node1)

    def remove_edge(self, node1: Hashable, node2: Hashable) -> None:
        """Remove edge node1-node2 from the graph.

        If the edge doesn’t exist, do nothing.

        Preconditions: self.has_node(node1) and self.has_node(node2)
        """
        super().remove_edge(node1, node2)
        super().remove_edge(node2, node1)
```

Undirected Graph Class

Edges, Neighbours

```
def edges(self) -> set:
  """Return the graph's edges as a set of pairs.

  Postconditions: for every edge A-B,
  the output has either (A, B) or (B, A) but not both
  """
  all_edges = set()
  for node1 in self.out:
    for node2 in self.out[node1]:
      if (node2, node1) not in all_edges:
        all_edges.add( (node1, node2) )
  return all_edges

def in_neighbours(self, node: Hashable) -> set:
  """Return all nodes that are adjacent to the node.

  Preconditions: self.has_node(node)
  """
  return self.out_neighbours(node)

def neighbours(self, node: Hashable) -> set:
  """Return all nodes that are adjacent to the node.

  Preconditions: self.has_node(node)
  """
  return self.out_neighbours(node)
```

Undirected Graph Class

In Degree, Degree

```python
def in_degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

    Preconditions: self.has_node(node)
    """
    return self.out_degree(node)

def degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

    Preconditions: self.has_node(node)
    """
    return self.out_degree(node)
```

Weighted Undirected Graph Class

Initial Code

```python
class WeightedUndirectedGraph(WeightedDiGraph):
  """A weighted undirected graph with hashable node objects.

  There's at most one edge between two different nodes.
  There are no edges between a node and itself.
  Edges have weights, which may be integers or floats.
  """
```

Weighted Undirected Graph Class

Add and Remove Edge

```python
def add_edge(self, node1: Hashable, node2: Hashable, weight: float) -> None:
    """Add an edge node1-node2 with the given weight to the graph.

    If the edge already exists, do nothing.

    Preconditions: self.has_node(node1) and self.has_node(node2)
    """
    super().add_edge(node1, node2, weight)
    super().add_edge(node2, node1, weight)

def remove_edge(self, node1: Hashable, node2: Hashable) -> None:
    """Remove edge node1-node2 from the graph.

    If the edge doesn't exist, do nothing.

    Preconditions: self.has_node(node1) and self.has_node(node2)
    """
    super().remove_edge(node1, node2)
    super().remove_edge(node2, node1)
```

Weighted Undirected Graph Class

Edges

```
  def edges(self) -> set:
    """Return the graph's edges as a set of triples (node1, node2, weight).

    Postconditions: for every edge A-B,
    the output has either (A, B, w) or (B, A, w) but not both
    """
    all_edges = set()
    for start in self.out:
      for (end, weight) in self.out[start].items():
        if (end, start, weight) not in all_edges:
          all_edges.add( (start, end, weight) )
    return all_edges
```

Weighted Undirected Graph Class

In Neighbours, Neighbours, In Degree, Degree

```python
def in_neighbours(self, node: Hashable) -> set:
    """Return all nodes that are adjacent to the node.

    Preconditions: self.has_node(node)
    """
    return self.out_neighbours(node)

def neighbours(self, node: Hashable) -> set:
    """Return all nodes that are adjacent to the node.

    Preconditions: self.has_node(node)
    """
    return self.out_neighbours(node)

def in_degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

    Preconditions: self.has_node(node)
    """
    return self.out_degree(node)

def degree(self, node: Hashable) -> int:
    """Return the number of edges attached to the node.

    Preconditions: self.has_node(node)
    """
    return self.out_degree(node)
```

Weighted Undirected Graph Class

Minimum Spanning Tree: Prim (1)

```python
from heapq import heappush, heappop

def prim(graph: WeightedUndirectedGraph, start: Hashable) -> WeightedUndirectedGraph
  """Return a minimum spanning tree of graph, beginning at start.

  Preconditions:
  - graph.has_node(start)
  - graph is connected
  - node objects are comparable
  """
  visited = WeightedUndirectedGraph()
  visited.add_node(start)

  unprocessed = []
  for neighbour in graph.neighbours(start):
    weight = graph.weight(start, neighbour)
    heappush(unprocessed, (weight, start, neighbour) )
```

- Minimum Spanning Tree Prim continued on next slide

Weighted Undirected Graph Class

Minimum Spanning Tree: Prim (2)

```python
while len(unprocessed) > 0:
  edge = heappop(unprocessed)
  weight = edge[0]
  previous = edge[1]
  current = edge[2]
  if not visited.has_node(current):
    visited.add_node(current)
    visited.add_edge(previous, current, weight)
    for neighbour in graph.neighbours(current):
      weight = graph.weight(current, neighbour)
      heappush(unprocessed, (weight, current, neighbour) )
return visited
```

- Note that the *priority queue* heapq does the work of making the next smallest weight edge available — it is always the first element of unprocessed

Drawing Graphs

Weighted DiGraph

- The provided graph code gives two `draw` methods:
- For Weighted DiGraph or Undirected Graph see line 249, slide 55,
- For Unweighted see line 130, slide 49,
- NetworkX is a Python package for the creation, manipulation and study of networks
- Matplotlib is a Python library for creating static, animated, and interactive visualizations
- Matplotlib is used by NetworkX
- Some of the examples in these notes explicitly use `savefig(fname)` from `matplotlib.pyplot` to save the current figure to an external file

  see matplotlib.pyplot.savefig

  see also matplotlib.pyplot.show

Drawing Graphs

NetworkX, Matplotlib

- NetworkX Drawing reference introduction states that it provides basic functionality for visualising graphs but its main aim is to enable graph analysis
- The examples in M269 use the Matplotlib interface commands
- It mentions the tools Cytoscape, Gephi, Graphviz, and for LaTeX typesetting, PGF/TikZ
- All of the packages are big and require reading the documentation — for example, the PGF/TikZ manual is 1321 pages (version 3.1.9a, 11 January 2022) (used in this document for most diagrams)
- You are not expected to learn any of the visualisation software but it may be worth noting some points about the provided `draw` method
- The code for the `draw` method is repeated on line 249, slide 70

Drawing Weighted DiGraph

draw method

```python
def draw(self) -> None:
  """Draw the graph."""
  if type(self) == WeightedDiGraph:
    graph = networkx.DiGraph()
  else:
    graph = networkx.Graph()
  graph.add_nodes_from(self.nodes())
  for (node1, node2, weight) in self.edges():
    graph.add_edge(node1, node2, w=weight)
  pos = networkx.spring_layout(graph)
  networkx.draw(graph, pos, with_labels=True,
    node_size=1000, node_color='lightblue',
    font_size=12, font_weight='bold')
  networkx.draw_networkx_edge_labels(graph, pos,
    edge_labels=networkx.get_edge_attributes(graph, 'w'))
```

- The line numbers are in gray to indicate this is a repeat of the code listing

Draw Method

Spring Layout

```
258    pos = networkx.spring_layout(graph)
```

- spring_layout positions nodes using Fruchterman-Reingold force-directed algorithm
- If several layouts are possible then each run of the program will cycle through possible layouts
- To have reproducible sequences of layout use an explicit seed=n where *n* is some fixed value.
- Code in context at line 258, slide 70,

Draw Method

NetworkX draw function

```
259     networkx.draw(graph, pos, with_labels=True,
260       node_size=1000, node_color='lightblue',
261       font_size=12, font_weight='bold')
```

- draw_networkx draws the graph with Matplotlib with various options
- If pos is not specified a *spring layout* will be computed
- with_labels set to True to draw labels on the nodes
- nodelist, edgelist draw only the specified nodes, edges
- Code in context at line 259, slide 70

Draw Method

NetworkX Draw Edge Labels function

```
259    networkx.draw_networkx_edge_labels(graph, pos,
260      edge_labels=networkx.get_edge_attributes(graph, 'w'))
```

- draw_networkx_edge_labels draws edge labels
- label_pos position of edge label along edge (0=head, 0.5=center, 1=tail)
- Code in context at line 262, slide 70,
- See also draw_networkx_nodes, can take a nodelist
- See also draw_networkx_edges, can take an edgelist

Draw Method

Inline and Externalizing Graphics

- Show the graphic in the Notebook cell with the code

```
%matplotlib inline
```

- Save graphic to PNG format file in current folder

```
import matplotlib.pyplot as plt
```

```
graph = WeightedUndirectedGraph()
```

```
graph.draw()

plt.savefig("M269TMA02Q3bGraphC.png")
```

- savefig in matplotlib.pyplot saves the current figure
- See also savefig in matplotlib.figure

Enumerations

Subsequences, Combinations

- M269 21J TMA02 Part 2 has questions that refer to calculating subsequences (or subsets) and combinations of numbers of elements from a list
- It uses the `combinations` function from the `itertools` module of the Python *Functional Programming Modules*
- It may be useful to review some simple programs that implement the same functions, but less efficiently — it may help understand the concepts
- The following code is in the same Python script as Morse Code `M269BinaryTrees2021JMorseCode.py` (but probably should be with the graph algorithm notes)
- The notes here give example implementations of
    - All subsequences of a list (a surrogate for subsets)
    - Two versions of combinations
- The notes use list comprehensions — a nice alternative to loops or explicit recursion (list comprehension reference)

Enumerations

Subsequences

- Subsequences of a list are all possible subsequences of elements from the list

```
AnPython3>>> subSeqsM([1,2,3])
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

```
def subSeqsM(xs) :
  if xs == [] :
    return [[]]
  else :
    return ([[xs[0]] + rs if b else rs
             for b in [False,True]
             for rs in subSeqsM(xs[1:])])
```

- If the list `xs` is empty there is one subsequence: the empty list
- Otherwise you can choose the first element followed by any of the subsequences of the rest of the list
  or ignore the first element and take any of the subsequences of the rest of the list
- See notes on List Comprehensions in the Graphs notes (mine)

Enumerations

Combinations (1)

- Combinations takes a list and an integer and return all subsequences of the list of that length
- Version using list comprehension instead of map

```
AnPython3>>> combsM01([1,2,3,4,5],3)
[[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3, 5], [1, 4, 5], [2, 3, 4], [2, 3,
```

```
def combsM01(xs, k) :
  if k == 0 :
    return [[]]
  elif xs == [] :
    return []
  else :
    return ([[xs[0]] + ys for ys in combsM01(xs[1:],k-1)]
            + combsM01(xs[1:],k))
```

- If k is 0 then there is one combination, the empty list
- If the list is empty (and $k \geqslant 1$) then there are none
- Otherwise choose the first element followed by (k-1) combinations of the rest of the list

  or ignore the first element and choose k combinations of elements from the rest of the list

Enumerations

Combinations (2)

- Combinations takes a list and an integer and return all subsequences of the list of that length

```
AnPython3>>> combsM([1,2,3,4,5],3)
[[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3, 5], [1, 4, 5], [2, 3, 4], [2, 3,
```

```
def combsM(xs, k) :
  if k == 0 :
    return [[]]
  elif xs == [] :
    return []
  else :
    return (list(map(lambda ys : ([xs[0]] + ys), combsM(xs[1:],k-1)))
            + combsM(xs[1:],k))
```

- Same as the list comprehension version (sort of)
- map takes a function and a list and applies the function to every element of the list
- Here the function is expressed as a lambda expression (an anonymous function)
- We need to convert the result to a list since map creates an iterable (explanation required?)

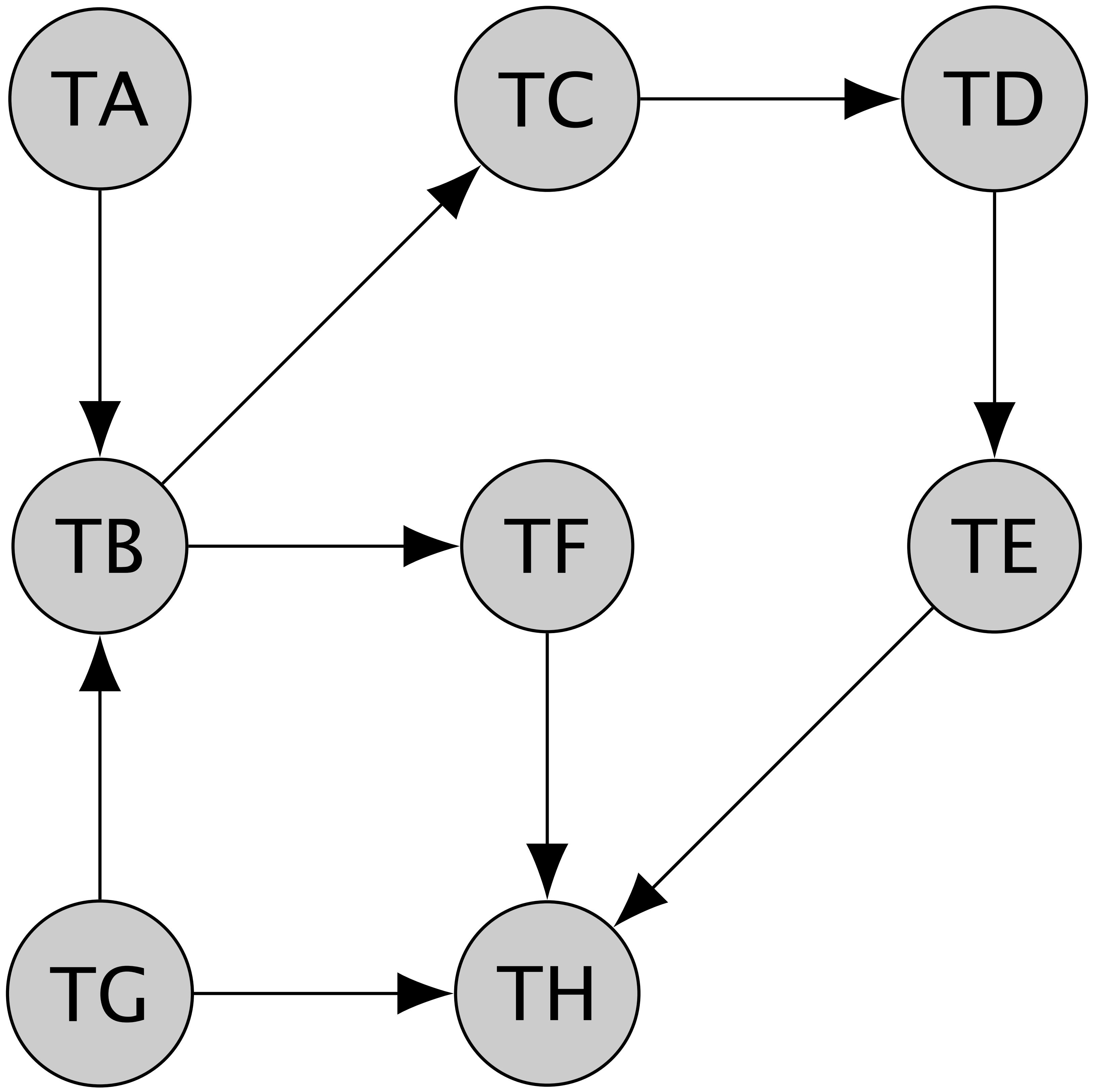
Topological Sort

Definition

- A *topological sort* of a directed acyclic graph (DAG) is a linear ordering of its vertices so that for any directed edge (u, v), u comes before v in the ordering
- See en.wikipedia.org/wiki/Topological_sorting
- A topological ordering is possible for a graph if and only if it is a DAG
- Any DAG has at least one topological ordering
- If a Hamiltonian path exists (a path visiting every node in a graph exactly once) then the graph has exactly one topological ordering

Topological Sort

Example Graph egTopSortGraph



- Find all the topological orderings on this digraph

Topological Sort

Algorithm

- `topSorts` takes a graph, `gr` and returns a list of lists of vertices (all the topological sorts of the graph)
- If the graph is empty, it returns a list containing just the empty list — *Note: not just the empty list*
- Obtain a list of all the start vertices of `gr`
- If the list of start vertices is empty, then the graph has a cycle — so raise an error and stop
- Otherwise for each start vertex, `v`
  - Join it to `ts`
  - where `ts` is one of the topological sorts of `gr` with `v` removed

Topological Sort — Algorithm

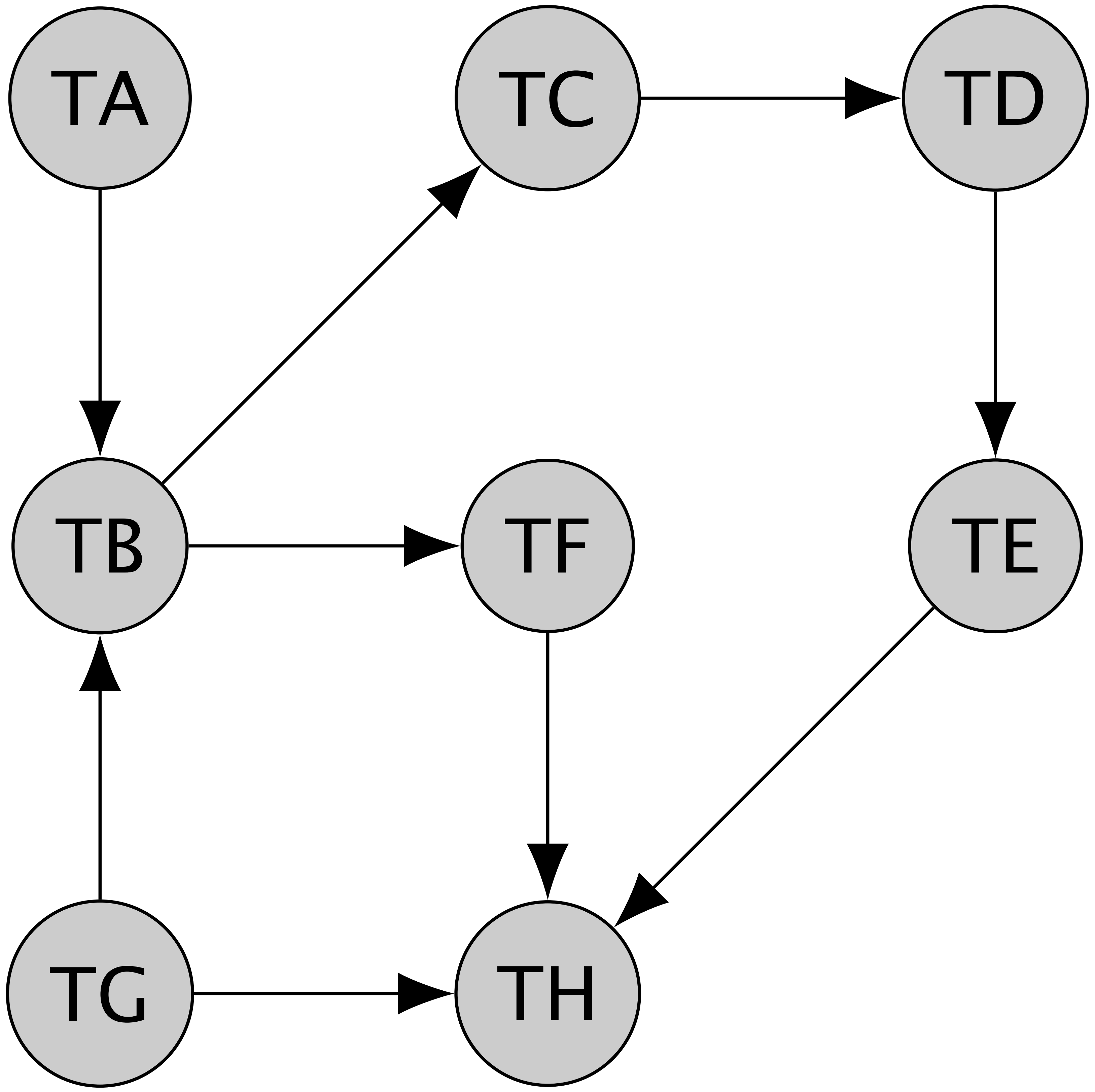
Python

```
def topSorts(gr):
  if isEmptyGraph(gr):
    return [[]]
  elif startVertices(gr) == []:
    raise RuntimeError('Cycle in the graph')
  else:
    return [[v] + ts
            for v in startVertices(gr)
            for ts in topSorts(removeVertex(v,gr))]
```

Topological Sort Example 01

Activity 3 Trace Exercise egTopSortGraph00

- Trace the development of the topological sort algorithm in the following graph



Go to Answer

Topological Sort Example 01

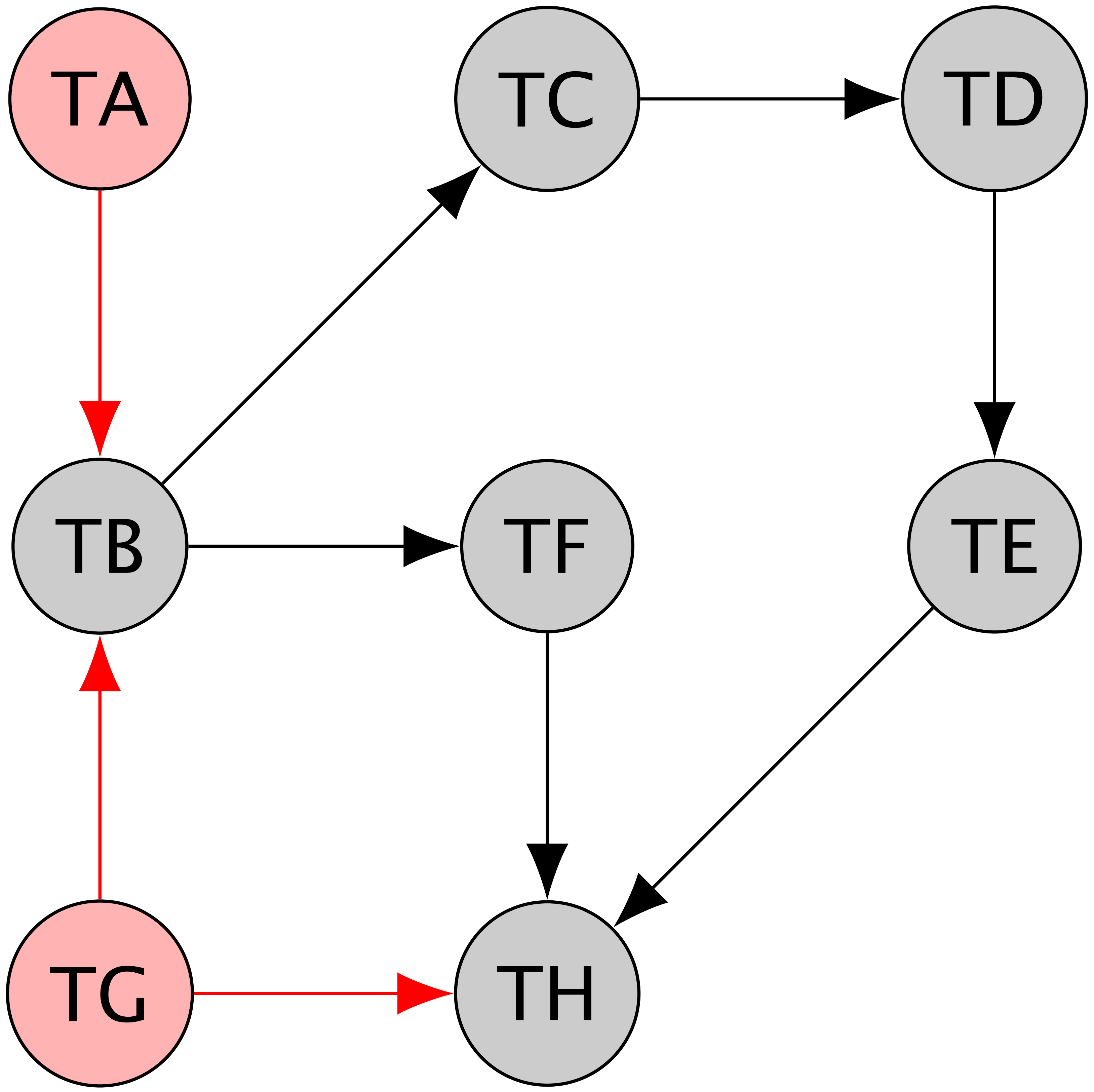
Answer 3 Trace Exercise

- Answer 3 Trace Exercise
- See the following slides

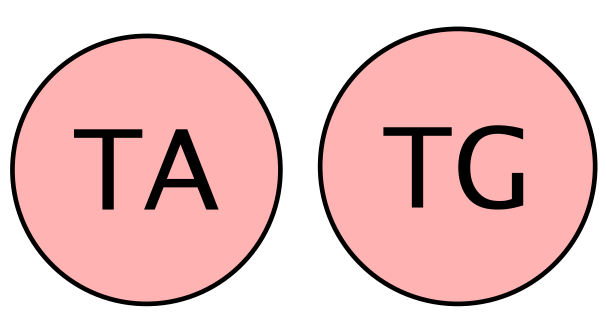
Go to Activity

Topological Sort Example 01

Step 1 Initial Graph egTopSortGraph01

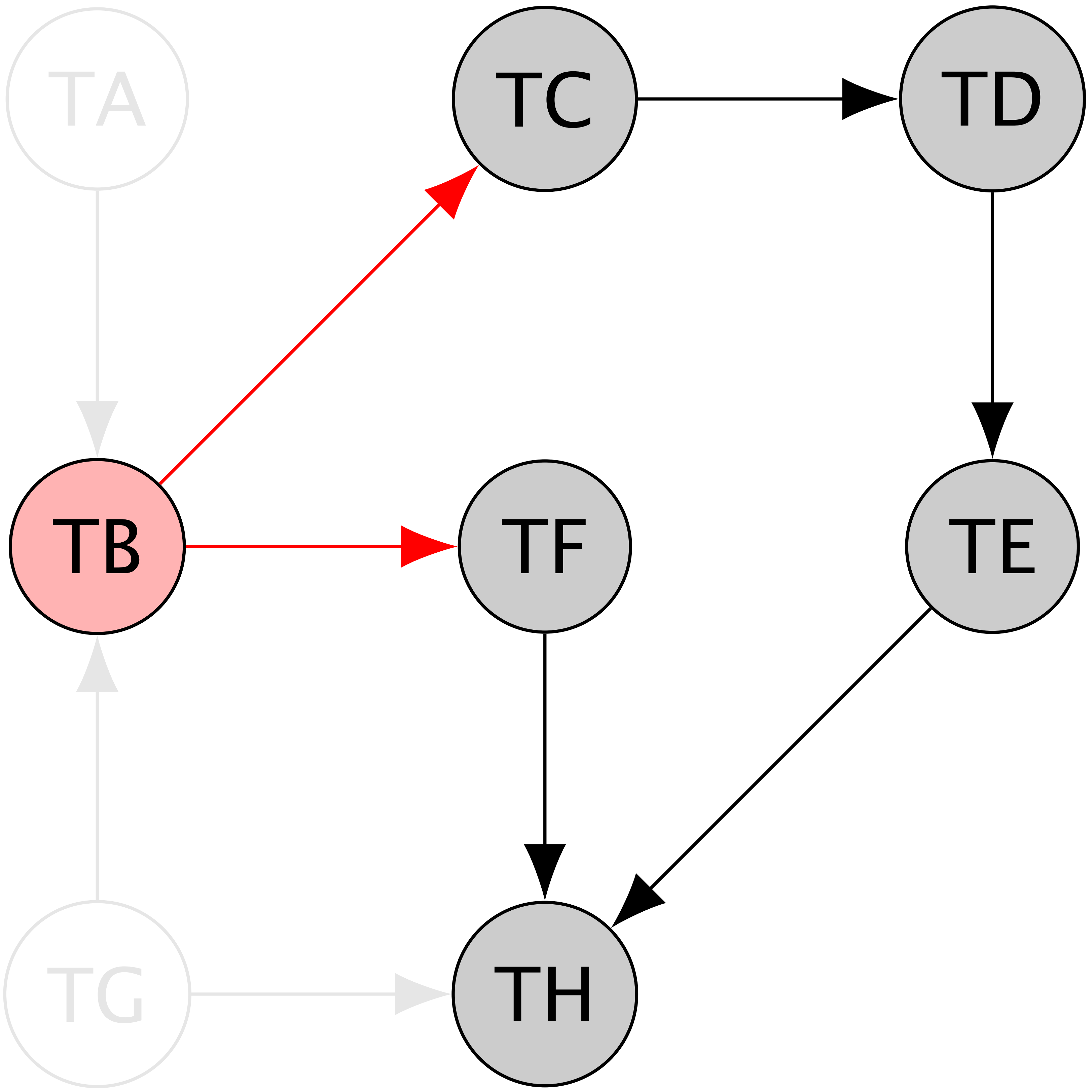


- Start vertices

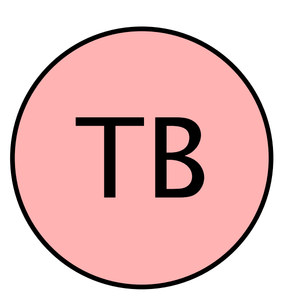


Topological Sort Example 01

Step 2 Remove Vertices TA, TG egTopSortGraph02

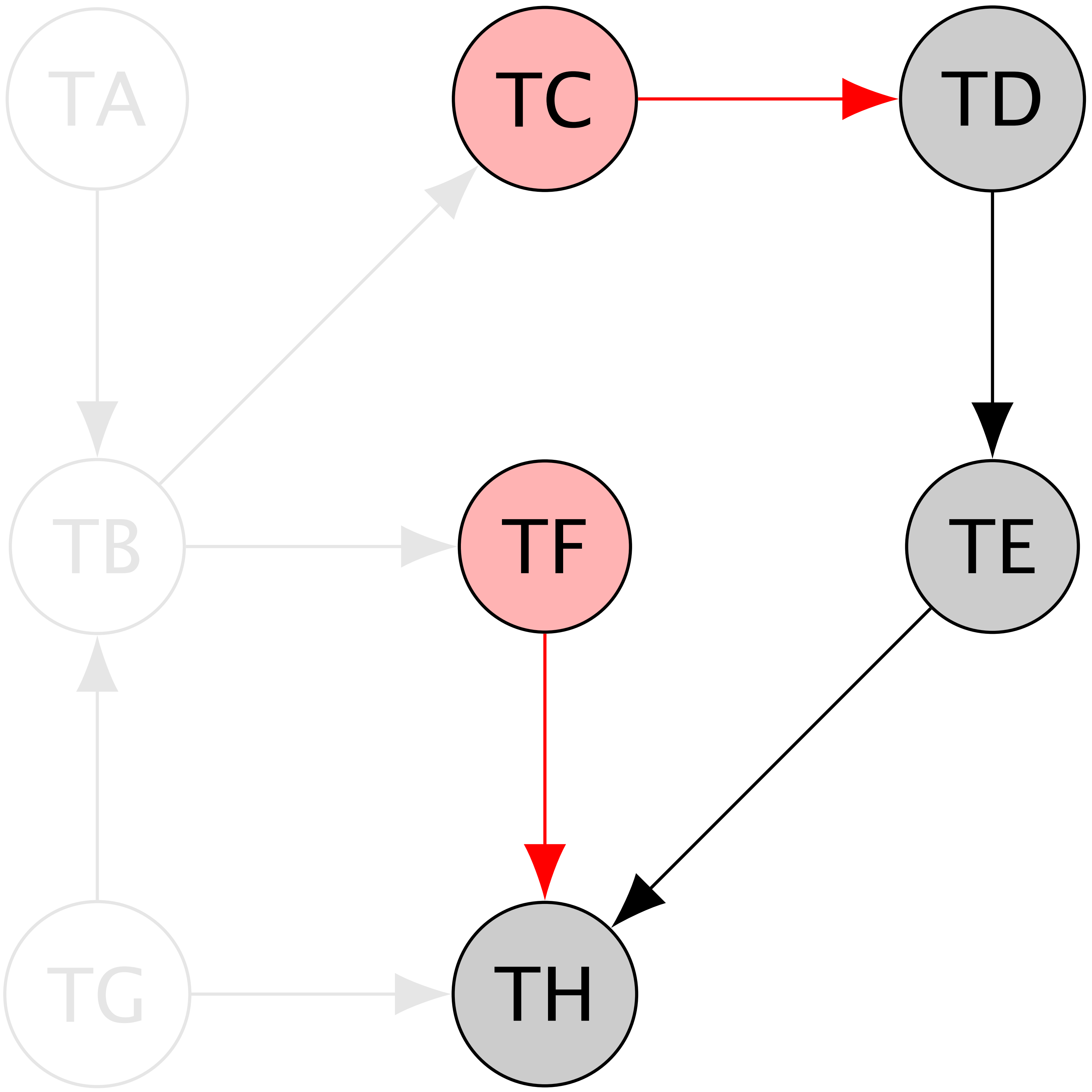


- Start vertices

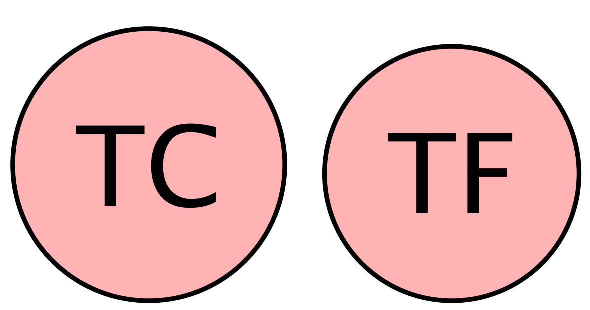


Topological Sort Example 01

Step 3 Remove Vertex TB egTopSortGraph03

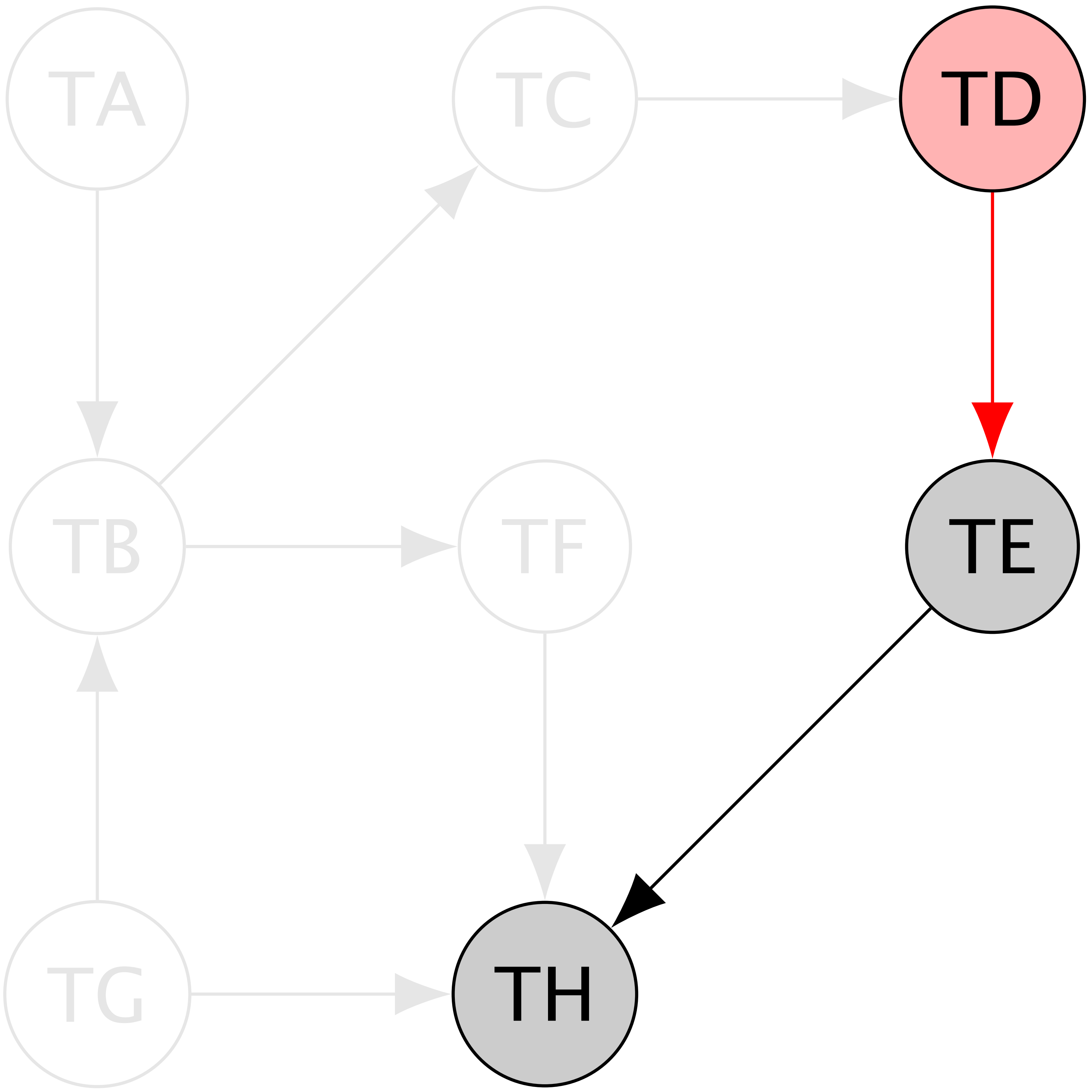


- Start vertices

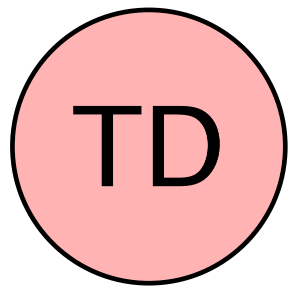


Topological Sort Example 01

Step 4 Remove Vertices TC, TF egTopSortGraph04



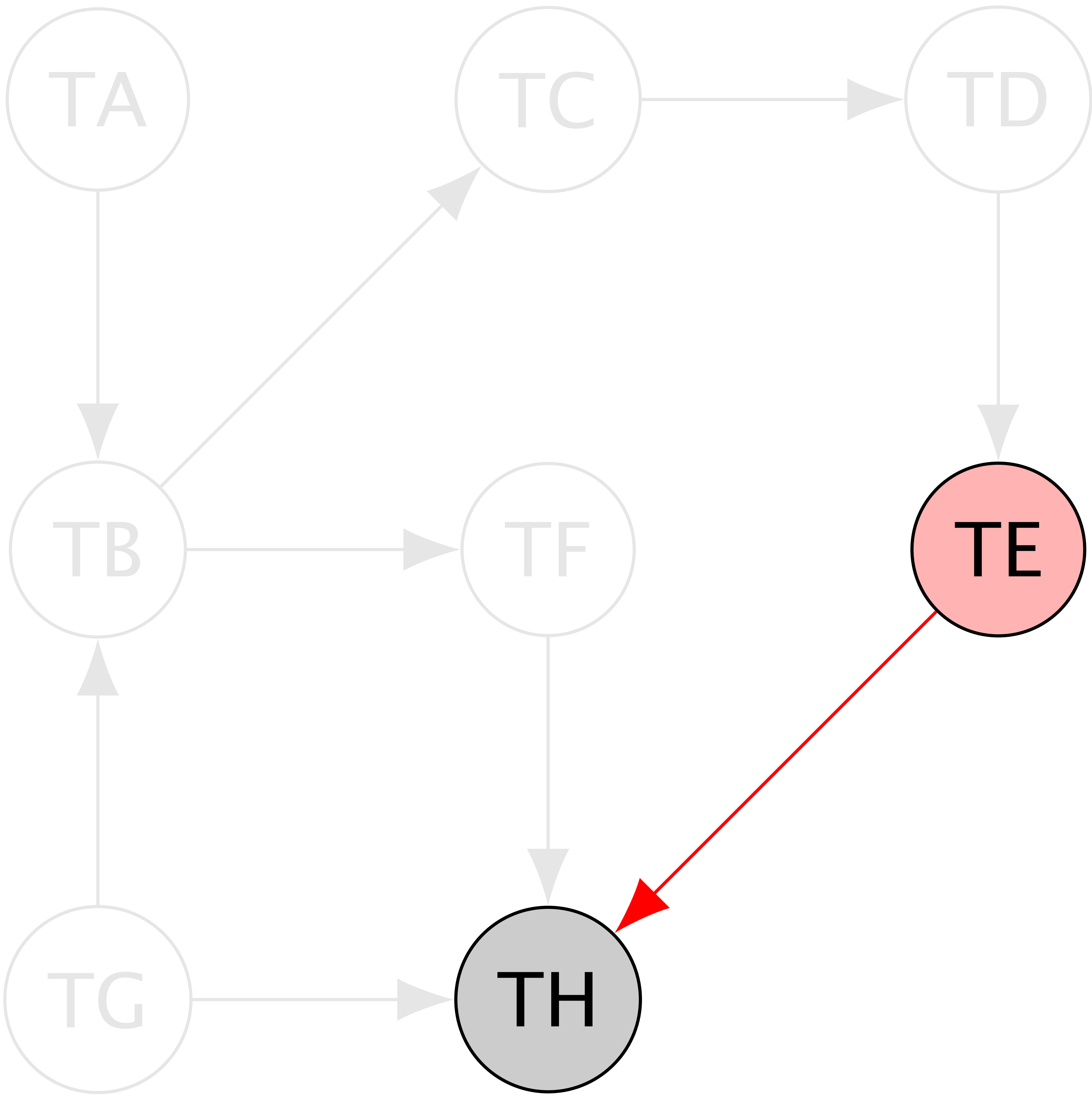
- Start vertices



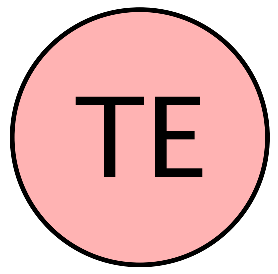
- Note: Step 4 to 6 has 4 combinations (see below)

Topological Sort Example 01

Step 5 Remove Vertex TD egTopSortGraph05

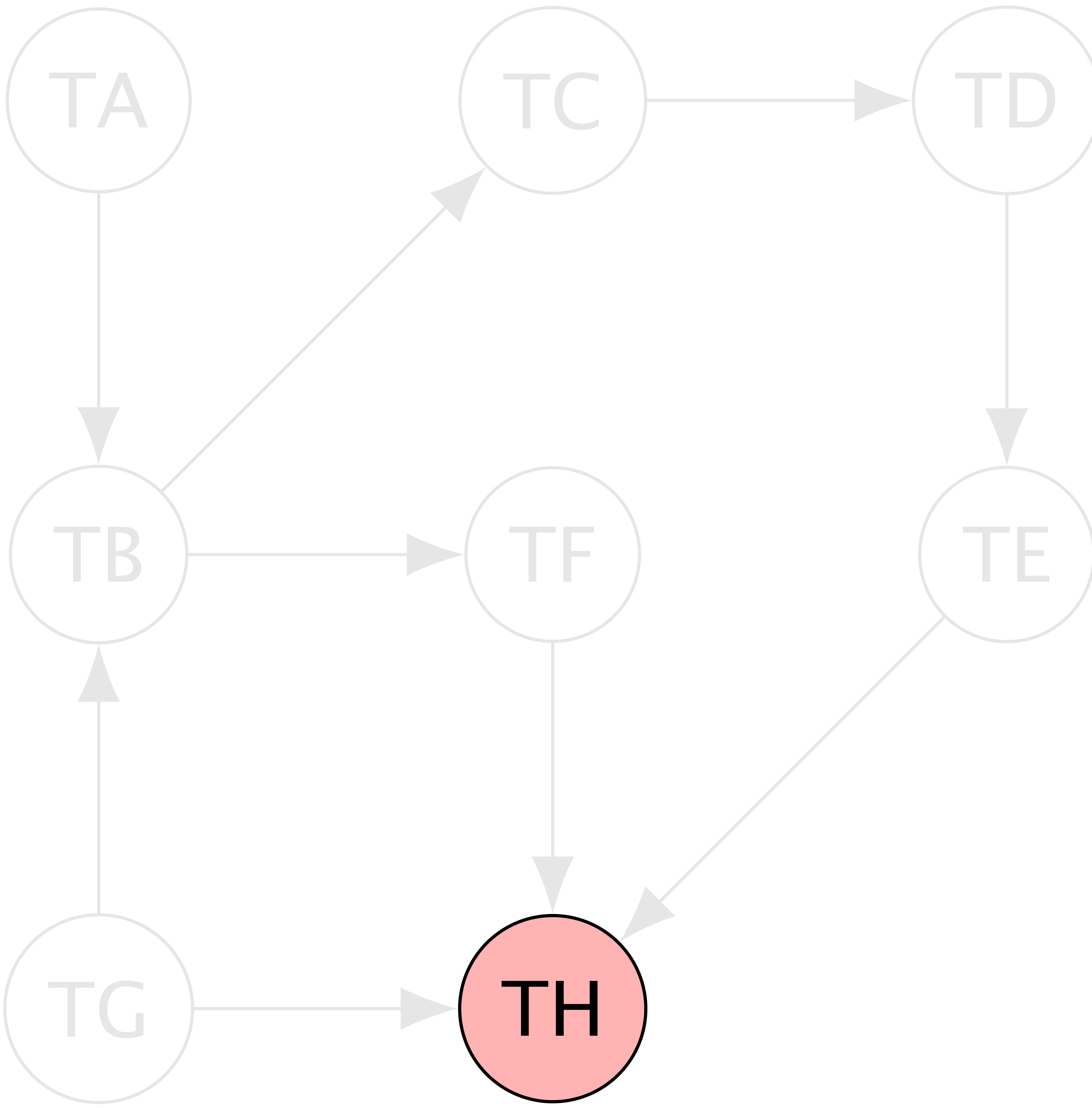


- Start vertices

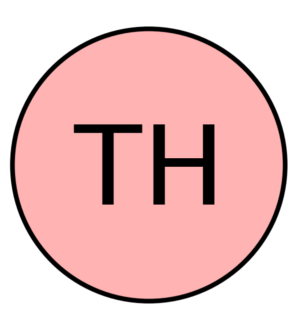


Topological Sort Example 01

Step 6 Remove Vertex TE egTopSortGraph06



- Start vertices



- Step 7 would be the empty graph (not drawn)

Topological Sort — Example

Output — Python

```
topSortsEG01GrTest \
  = (topSorts(eg01Gr)
     == [[ta,tg,tb,tc,td,te,tf,th]
        ,[ta,tg,tb,tc,td,tf,te,th]
        ,[ta,tg,tb,tc,tf,td,te,th]
        ,[ta,tg,tb,tf,tc,td,te,th]
        ,[tg,ta,tb,tc,td,te,tf,th]
        ,[tg,ta,tb,tc,td,tf,te,th]
        ,[tg,ta,tb,tc,tf,td,te,th]
        ,[tg,ta,tb,tf,tc,td,te,th]])
```

- Note how the step 4 to 6 combinations get enumerated
- Note that a vertex `ta` would be displayed as `Vertex(vtxName='TA')`
- Notice the Python Explicit line joining with (`\<nl>`) and Python Implicit line joining with (`(...)`)
- The backslash (`\`) must be followed by an end of line character (`<nl>`)

Dijkstra's Algorithm

Sources

- From https://www.cse.ust.hk/~dekai/271/ (Lecture 10)
- Cormen (2009, chapter 24) — Cormen (2009, page 648) has a footnote explaining the origin of the term *relaxation*
- Sedgewick and Wayne (2011)
- Miller and Ranum (2011, section 7.8)
- *A Functional Graph Library* http://web.engr.oregonstate.edu/~erwig/fgl/
- Rabhi and Lapalme (1999, chapter 7)

Dijkstra's Algorithm

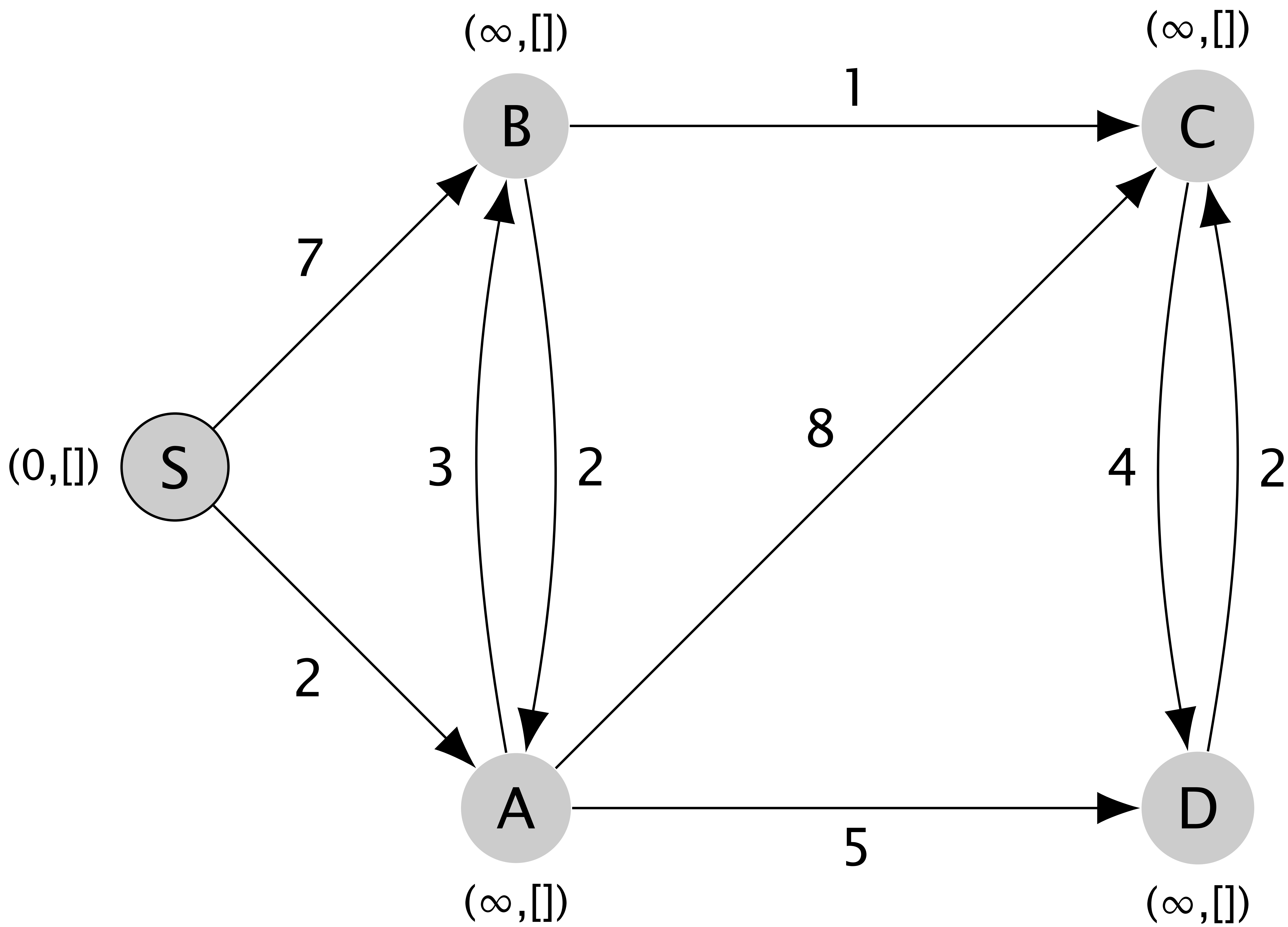
Structured English

```
dijkstra(gr,weight,s)
  for u in vertices(gr)
    dist(u) = Infinity
    label(u) = Temp
  dist(s) = 0
  pred(s) = None
  q = makePriorityQ(vertices(gr))

  while not isEmptyPQ(q)
    u = extractMinPQ(q)
    for v in adj(gr,u)
      if (label(v) == Temp
          and dist(u) + weight(edge(u,v)) < dist(v))
        dist(v) = dist(u) + weight(edge(u,v))
        q = decreaseKeyPQ(q,v,dist(v))
        pred(v) = u
    label(u) = Permanent
```

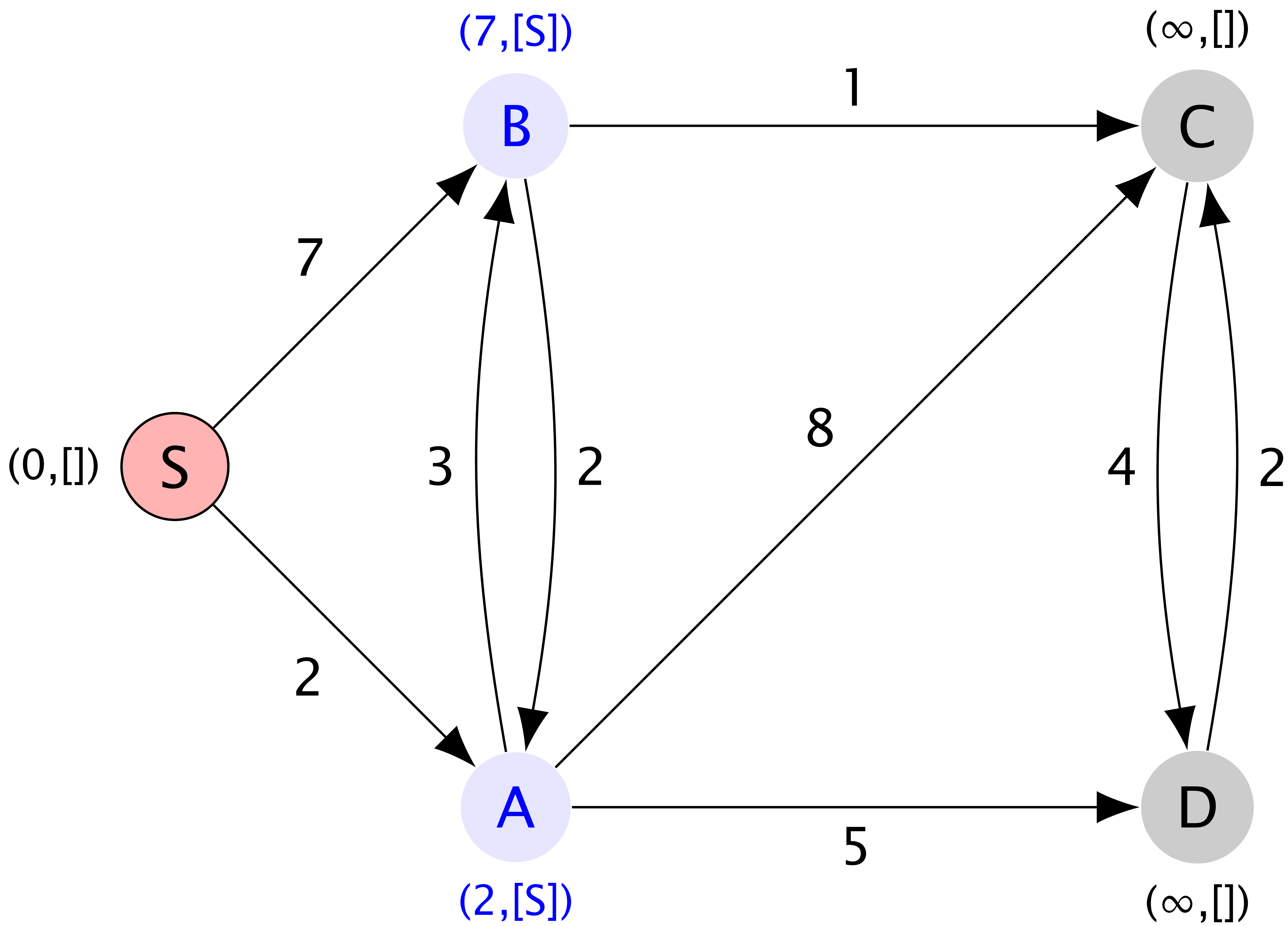
Dijkstra's Algorithm Example 01

Step 0 Initialisation egDijkstraGraph0100



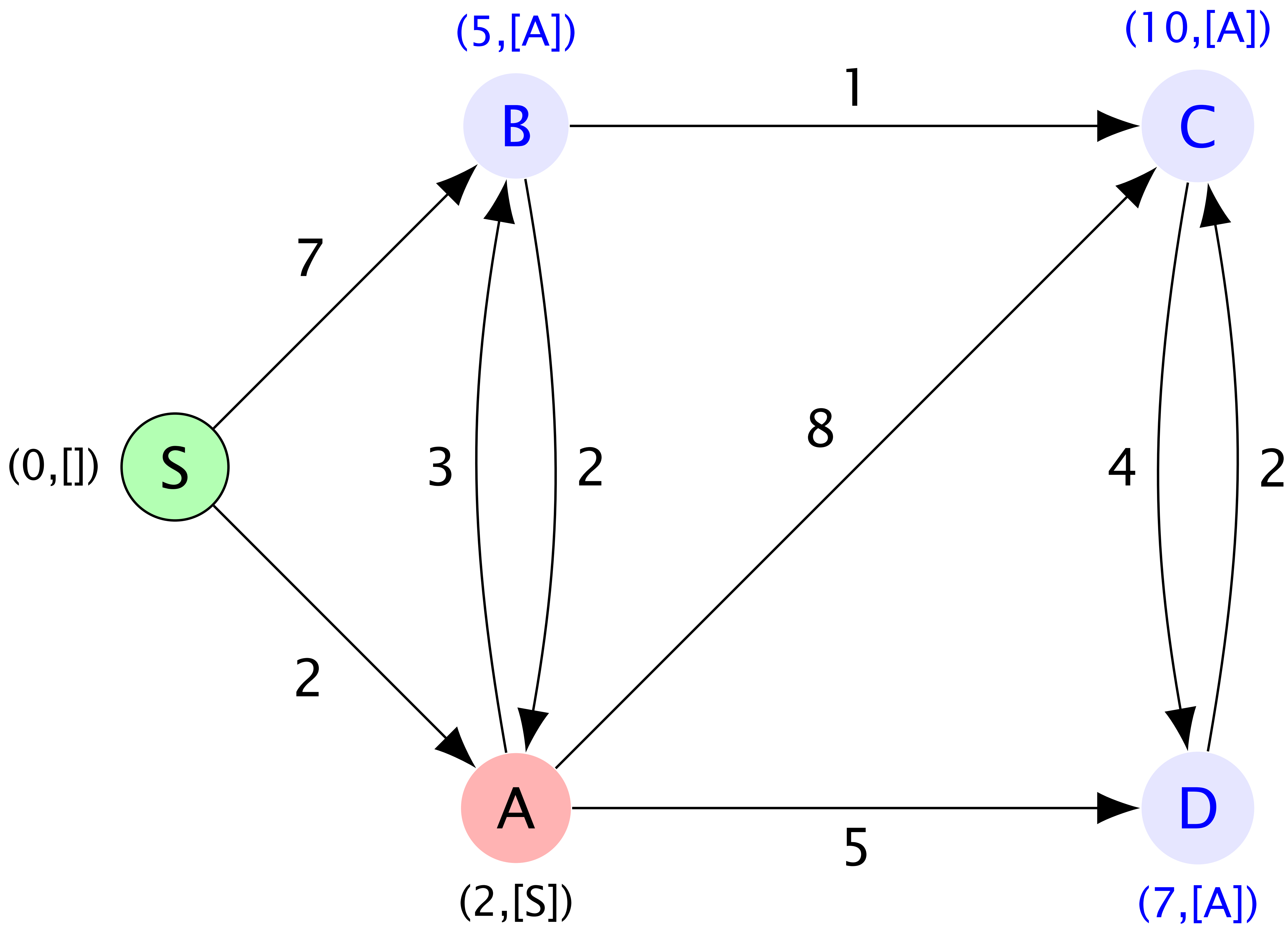
Dijkstra's Algorithm Example

Step 1 Process S egDijkstraGraph0101



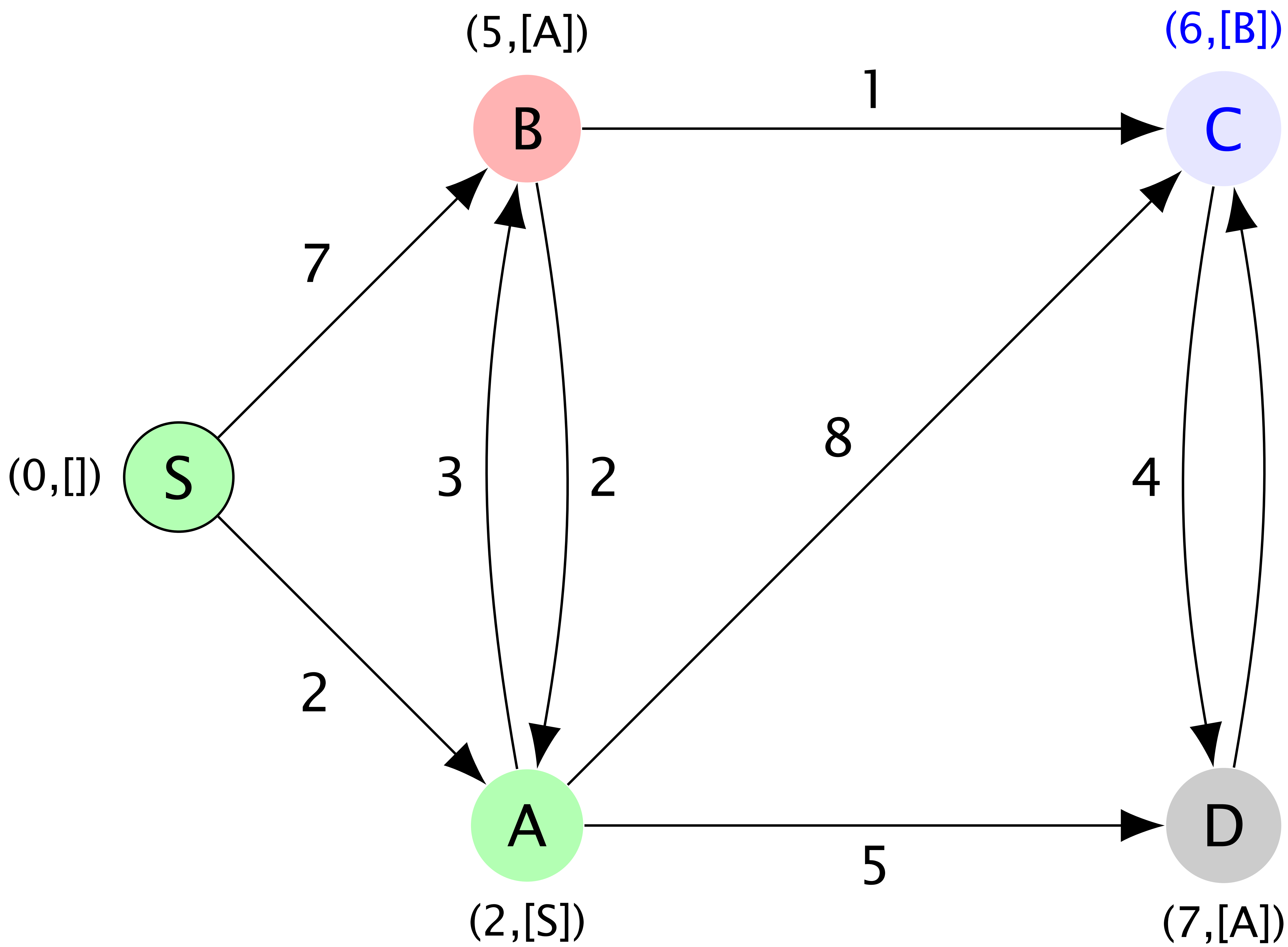
Dijkstra's Algorithm Example

Step 2 Process A egDijkstraGraph0102



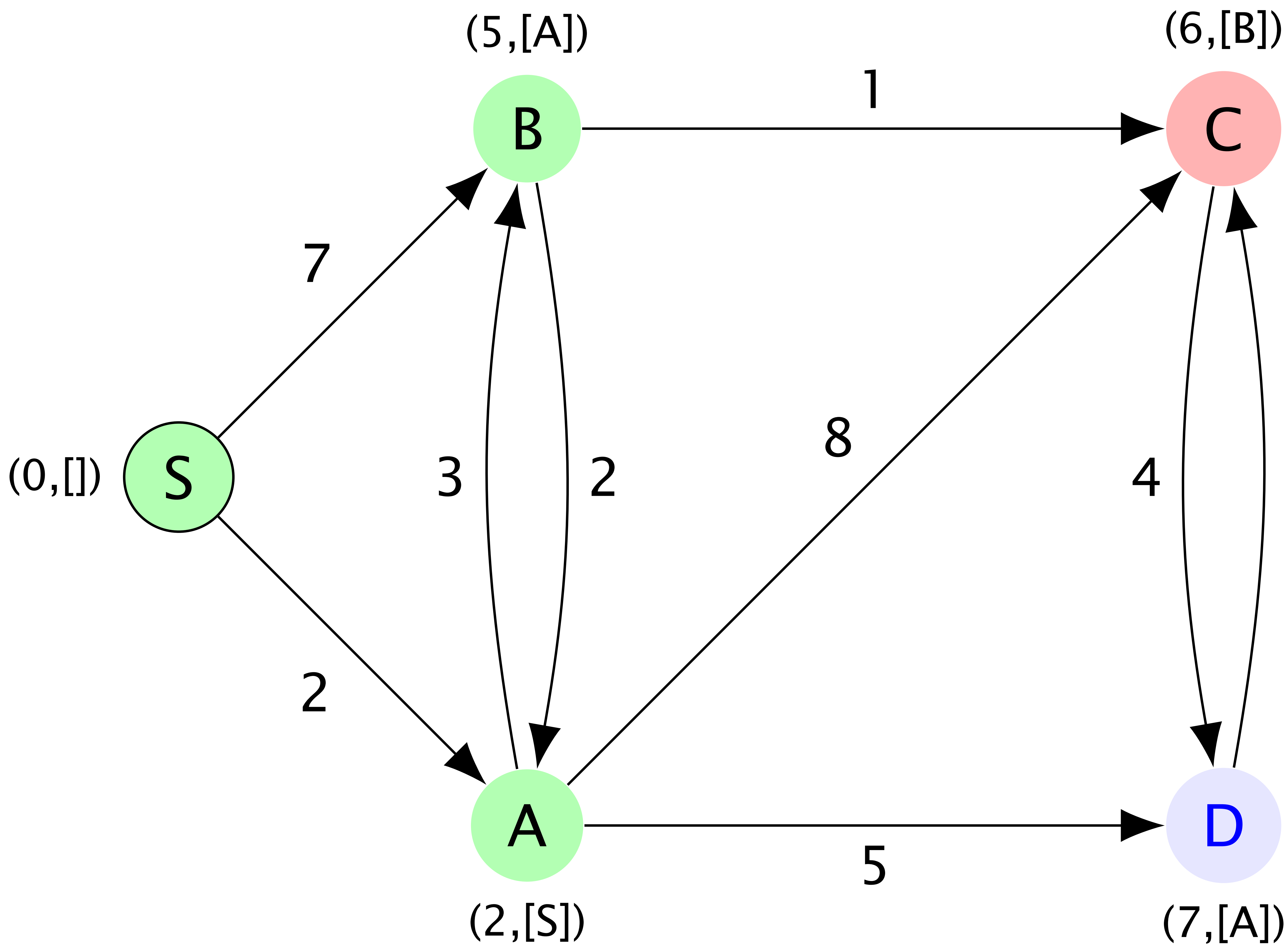
Dijkstra's Algorithm Example

Step 3 Process B egDijkstraGraph0103



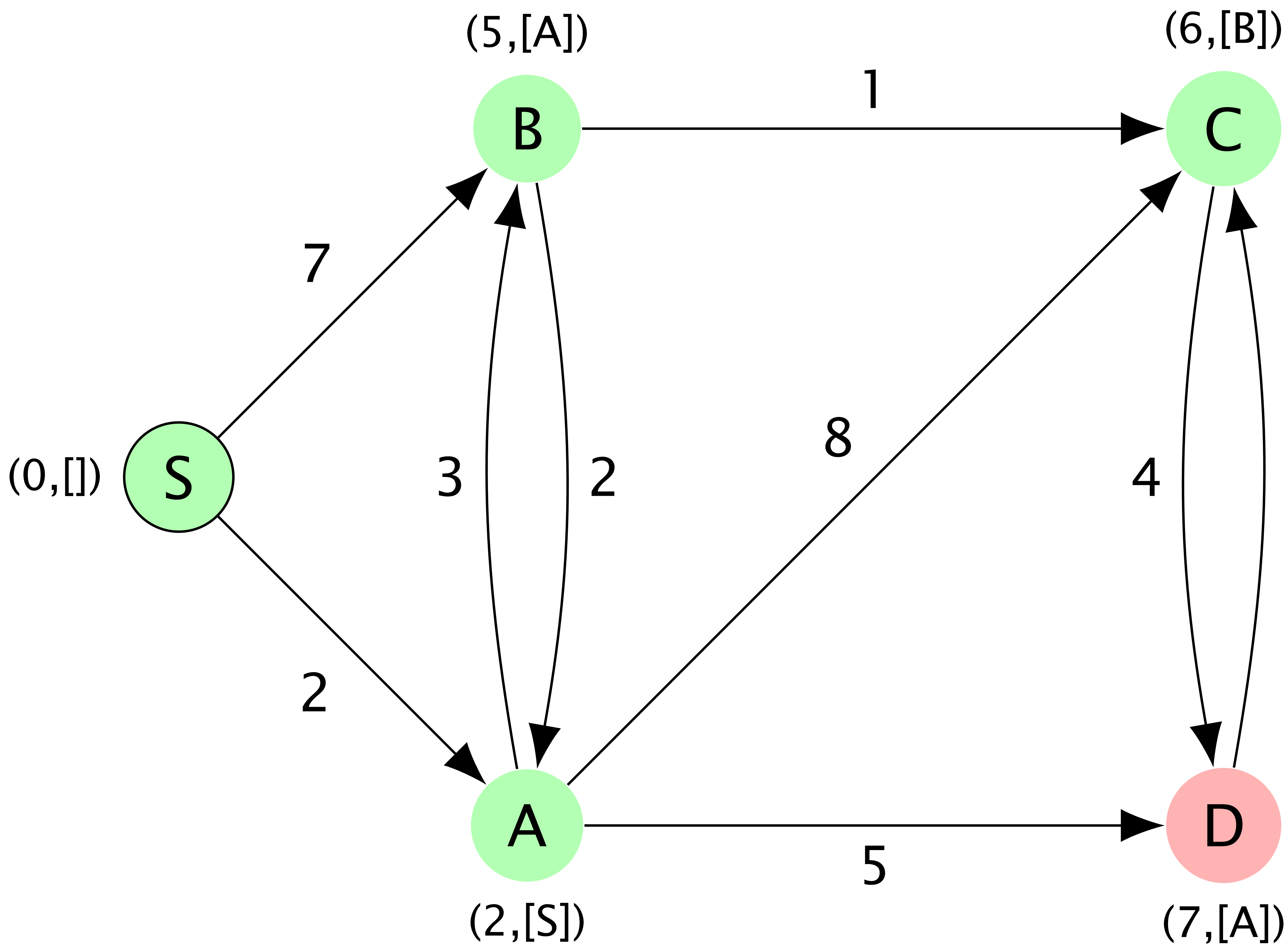
Dijkstra’s Algorithm Example

Step 4 Process C egDijkstraGraph0104



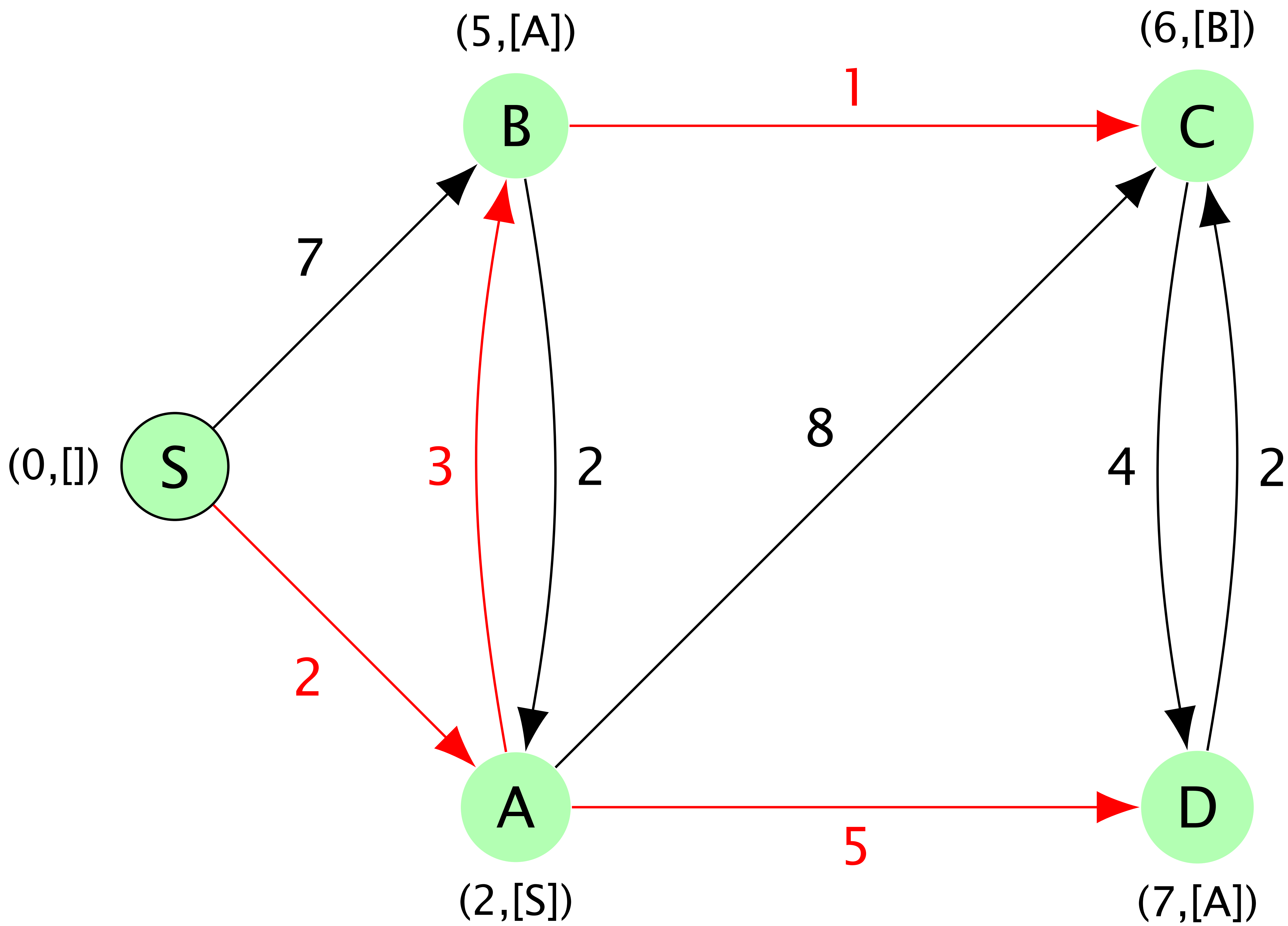
Dijkstra's Algorithm Example

Step 5 Process D egDijkstraGraph0105



Dijkstra's Algorithm Example

Shortest Path Tree Edges egDijkstraGraph0106



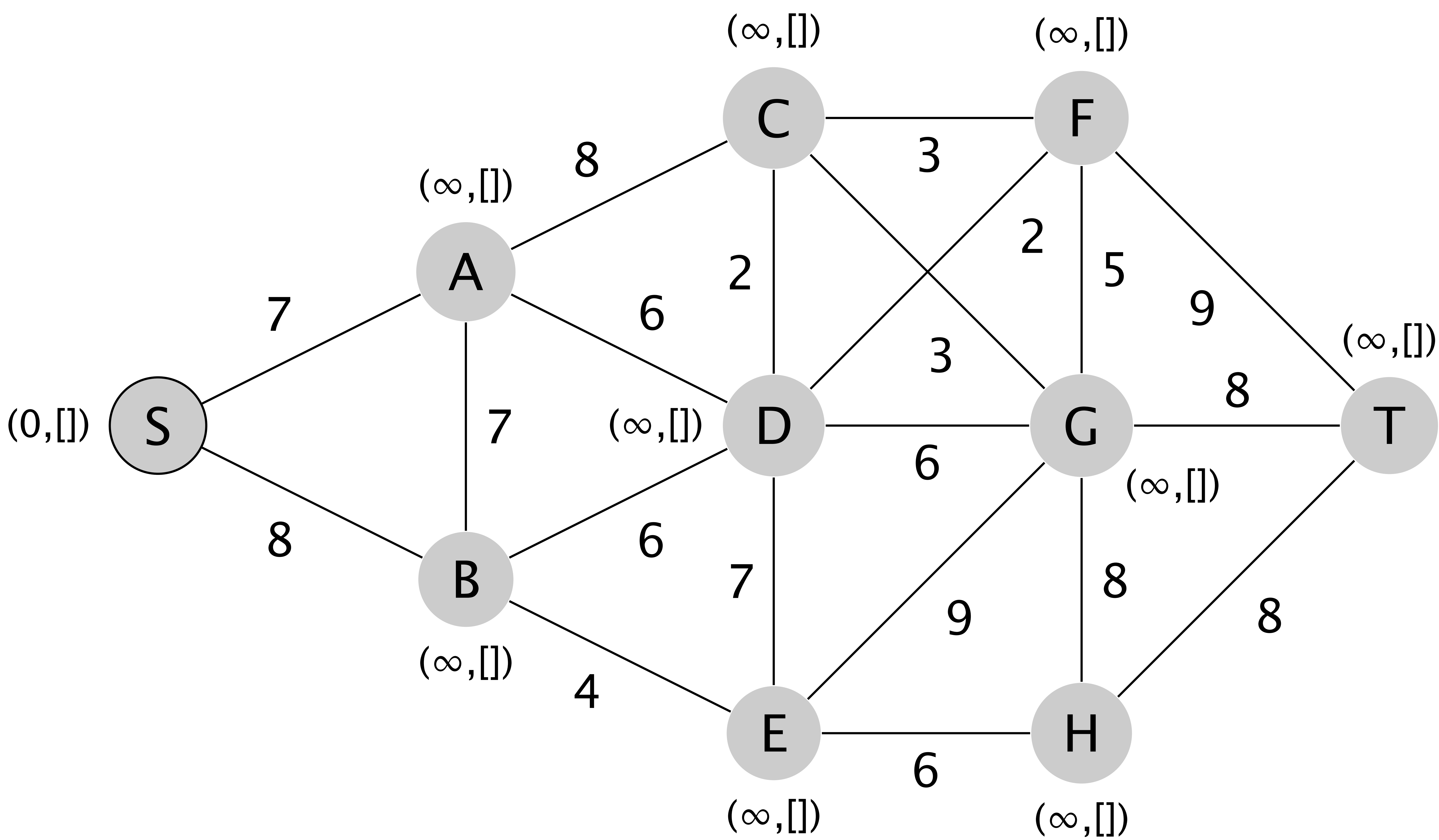
Dijkstra's Algorithm

Further points

- See presentation at http://www.ukuug.org/events/agm2010/ShortestPath.pdf
- The algorithm as given assumes unique shortest paths — what if there are multiple shortest paths ? Modify the algorithm to accommodate this — change the weight on some edge to test this in the above example (change the weight of edge (A,C) to 4, for example)
- Implement a priority queue for Dijkstra's algorithm
- Material essentially comes from Cormen, chp 24

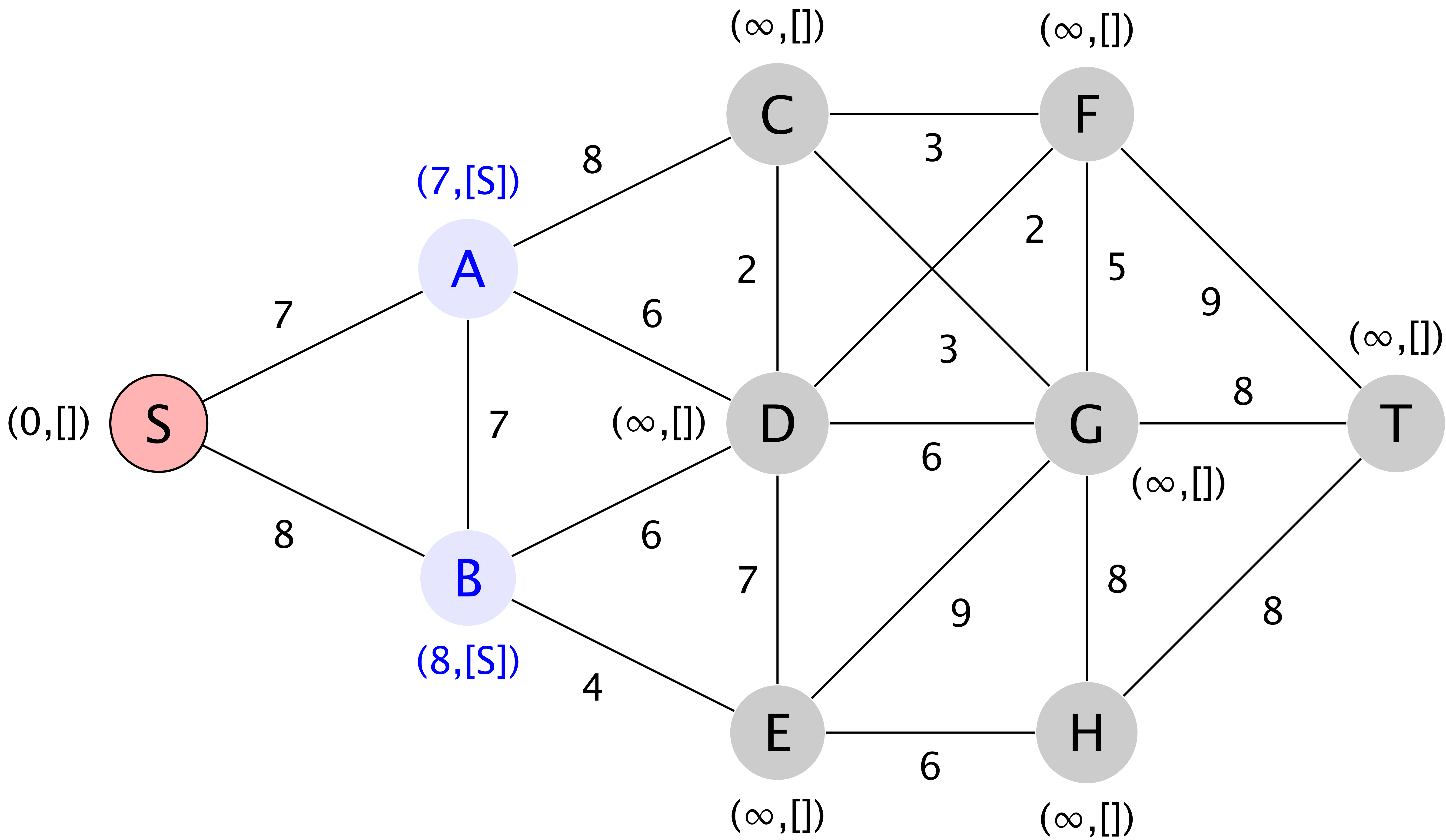
Dijkstra's Algorithm Example 02

Step 0 Initialisation egDijkstraGraph0200



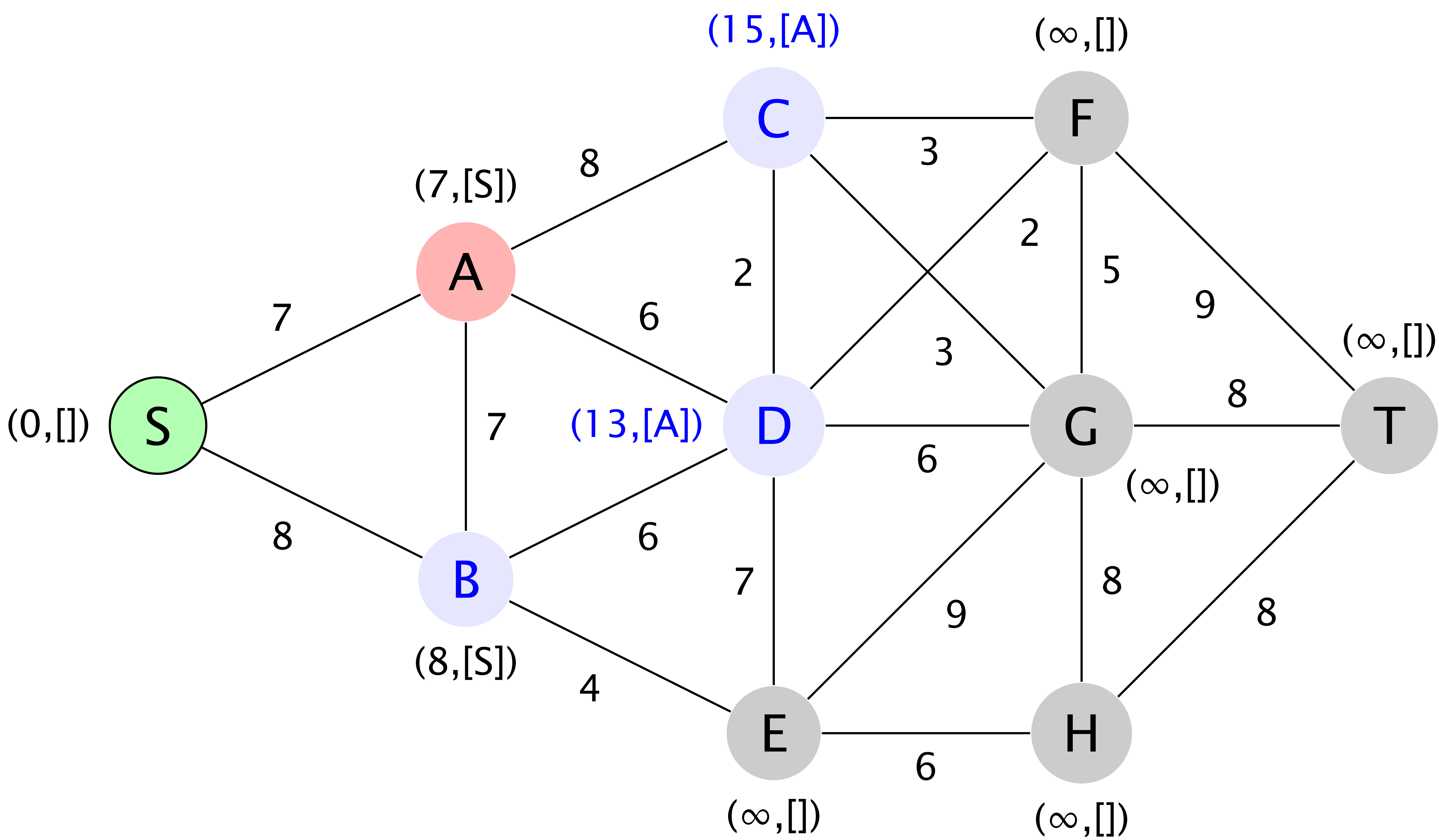
Dijkstra's Algorithm Example 02

Step 1 Process S egDijkstraGraph0201



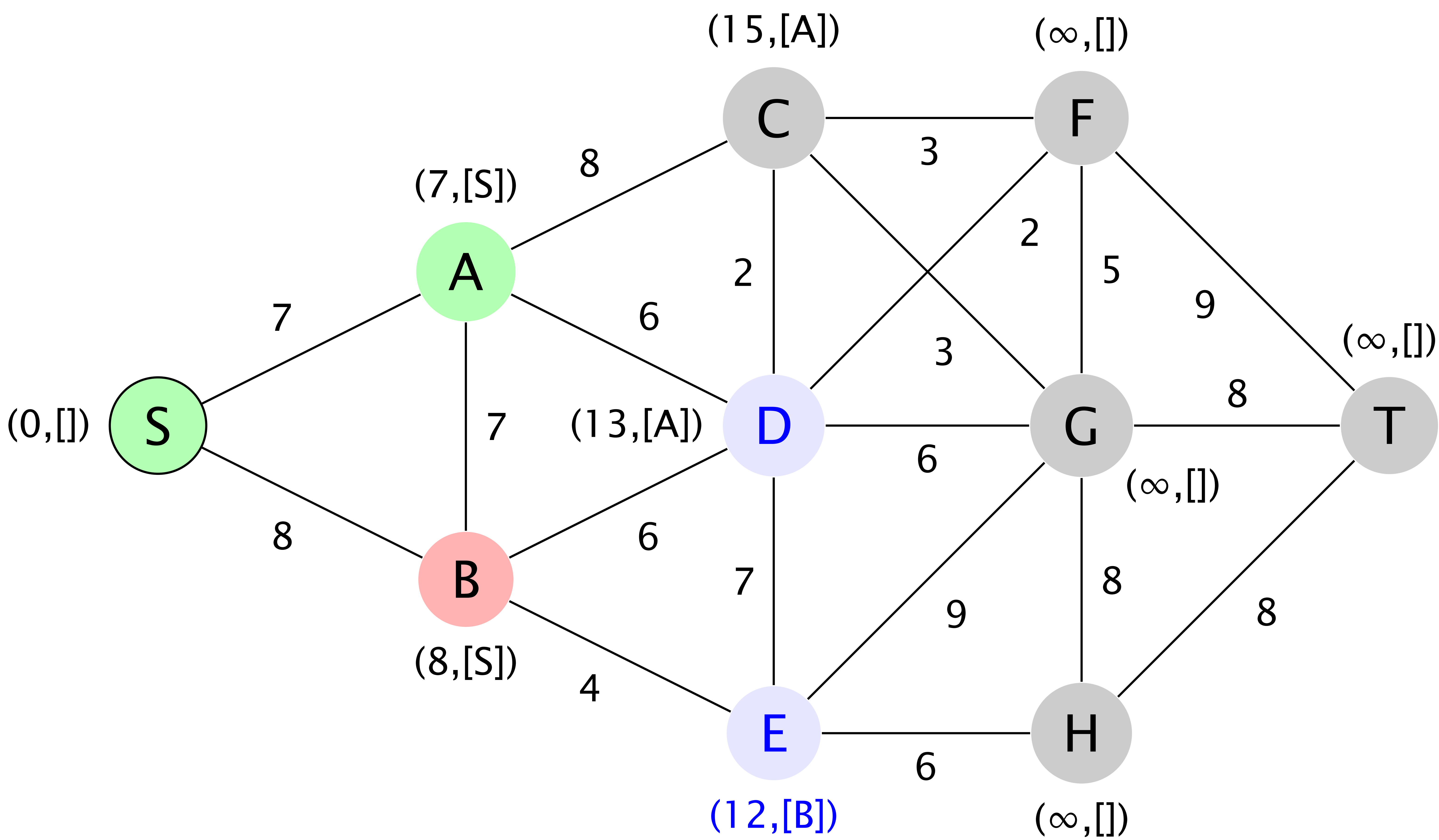
Dijkstra’s Algorithm Example 02

Step 2 Process A egDijkstraGraph0202



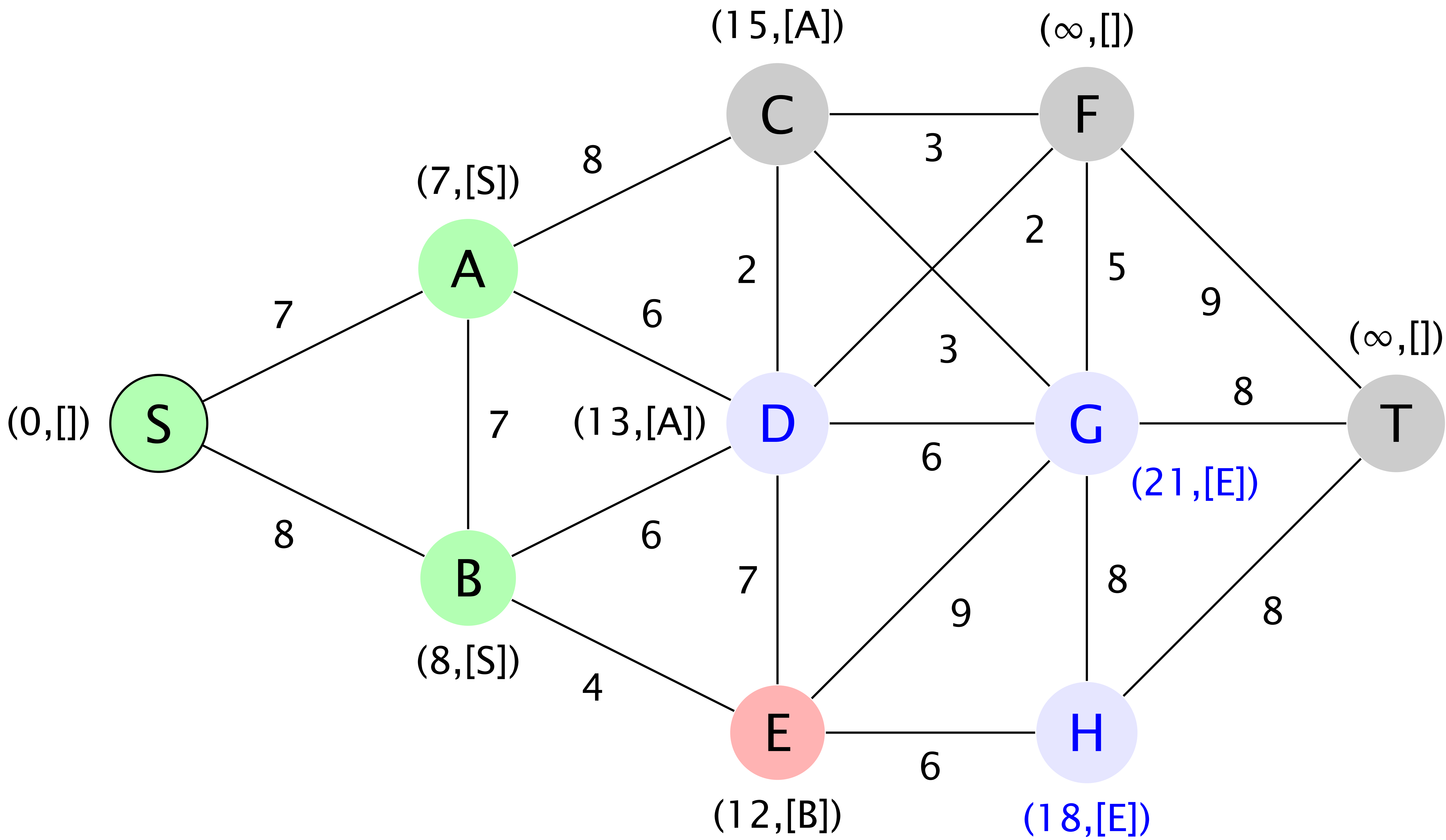
Dijkstra's Algorithm Example 02

Step 3 Process B egDijkstraGraph0203



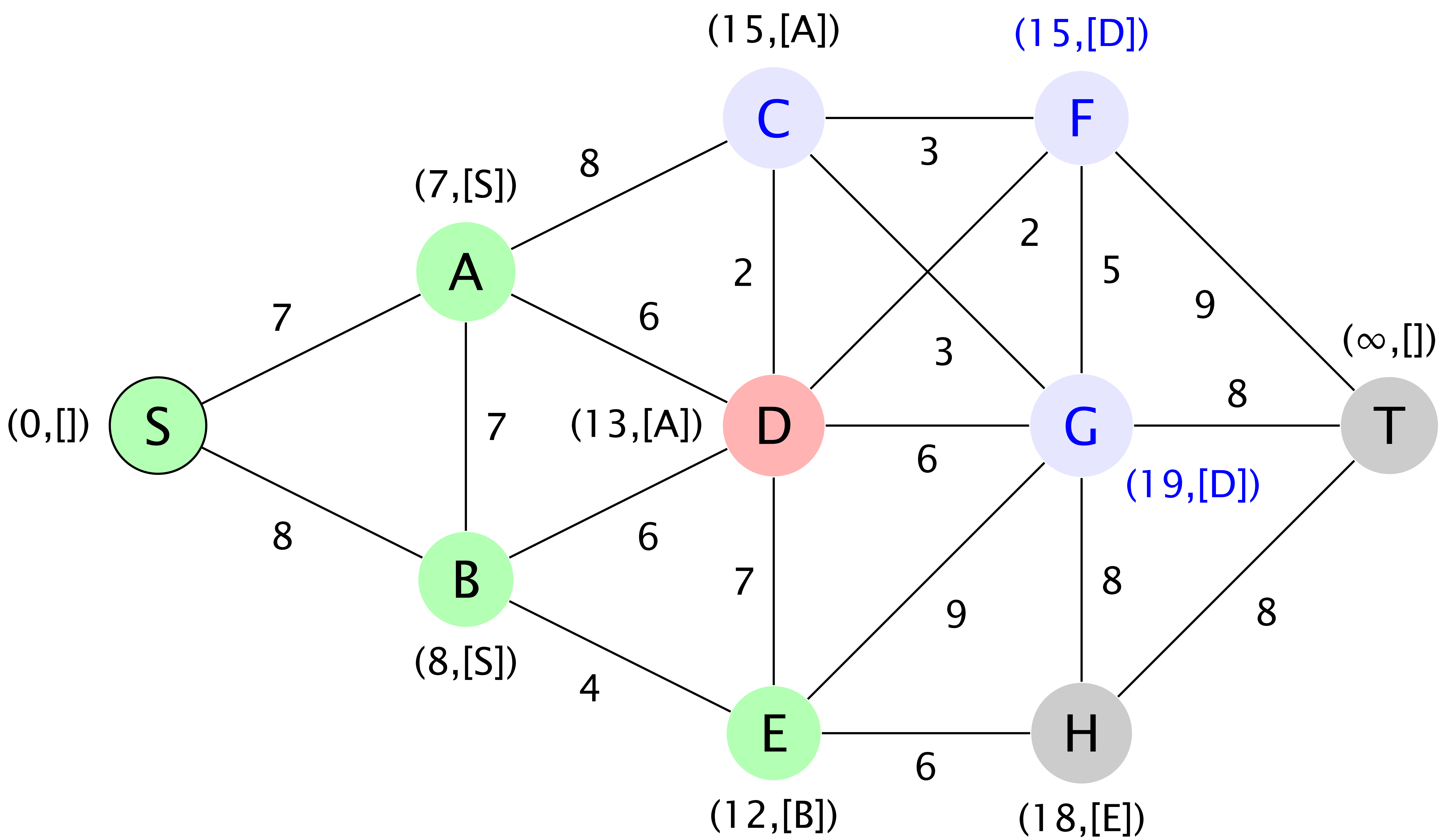
Dijkstra’s Algorithm Example 02

Step 4 Process E egDijkstraGraph0204



Dijkstra's Algorithm Example 02

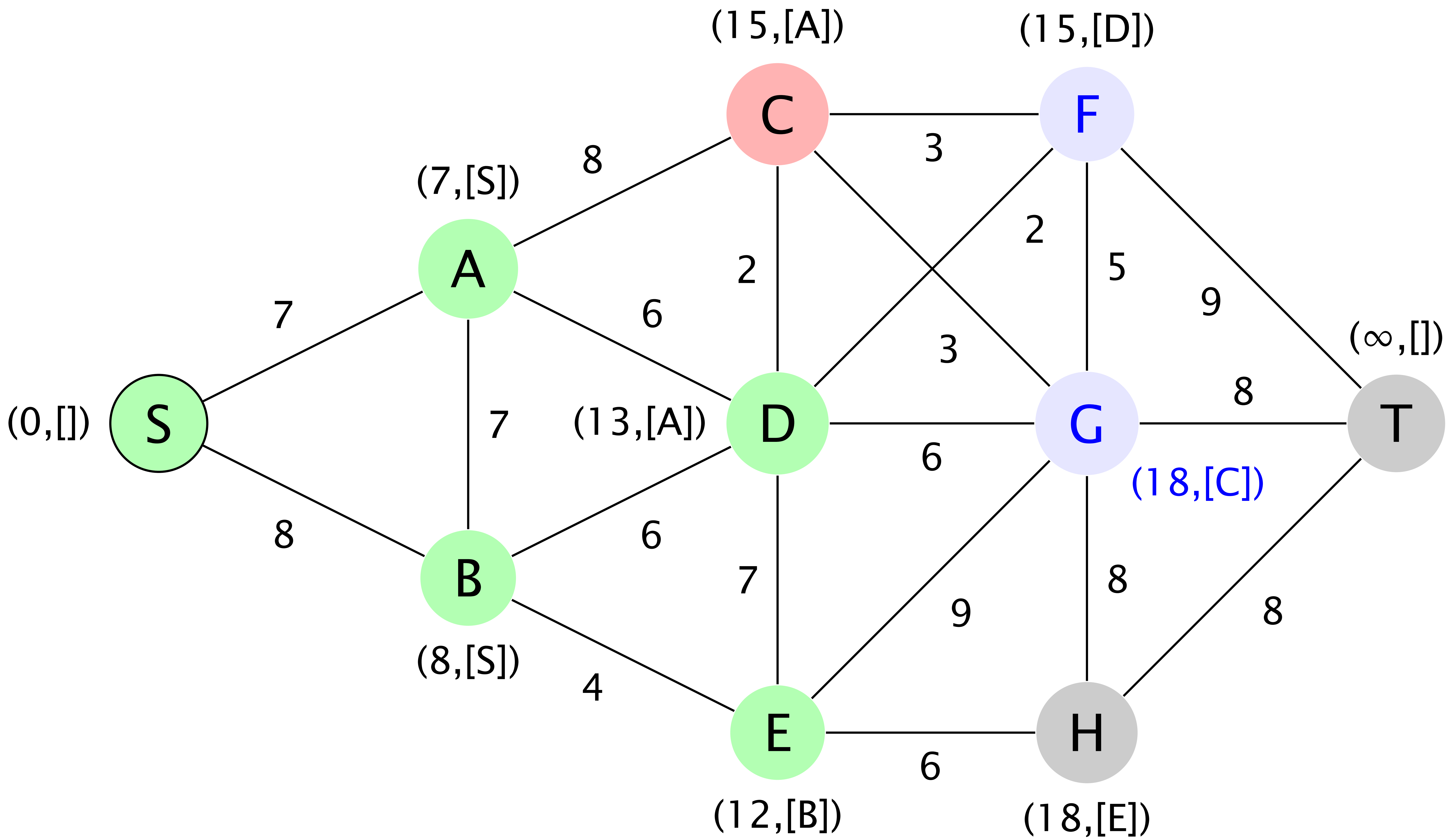
Step 5 Process D egDijkstraGraph0205



- Vertex C should have label (15,[A,D]) if we record multiple shortest routes
- How do we change the algorithm ?

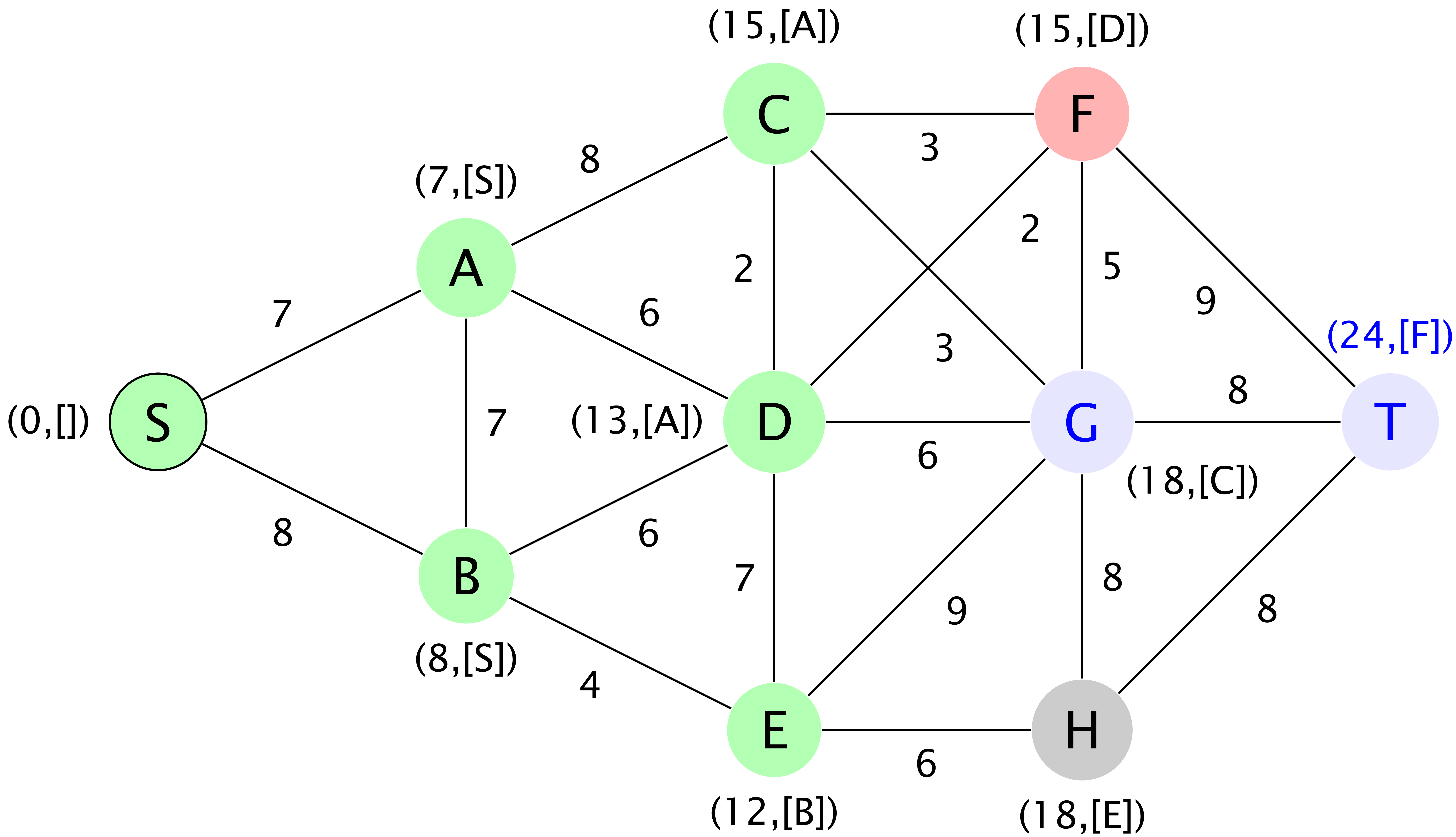
Dijkstra's Algorithm Example 02

Step 6 Process C (or F) egDijkstraGraph0206



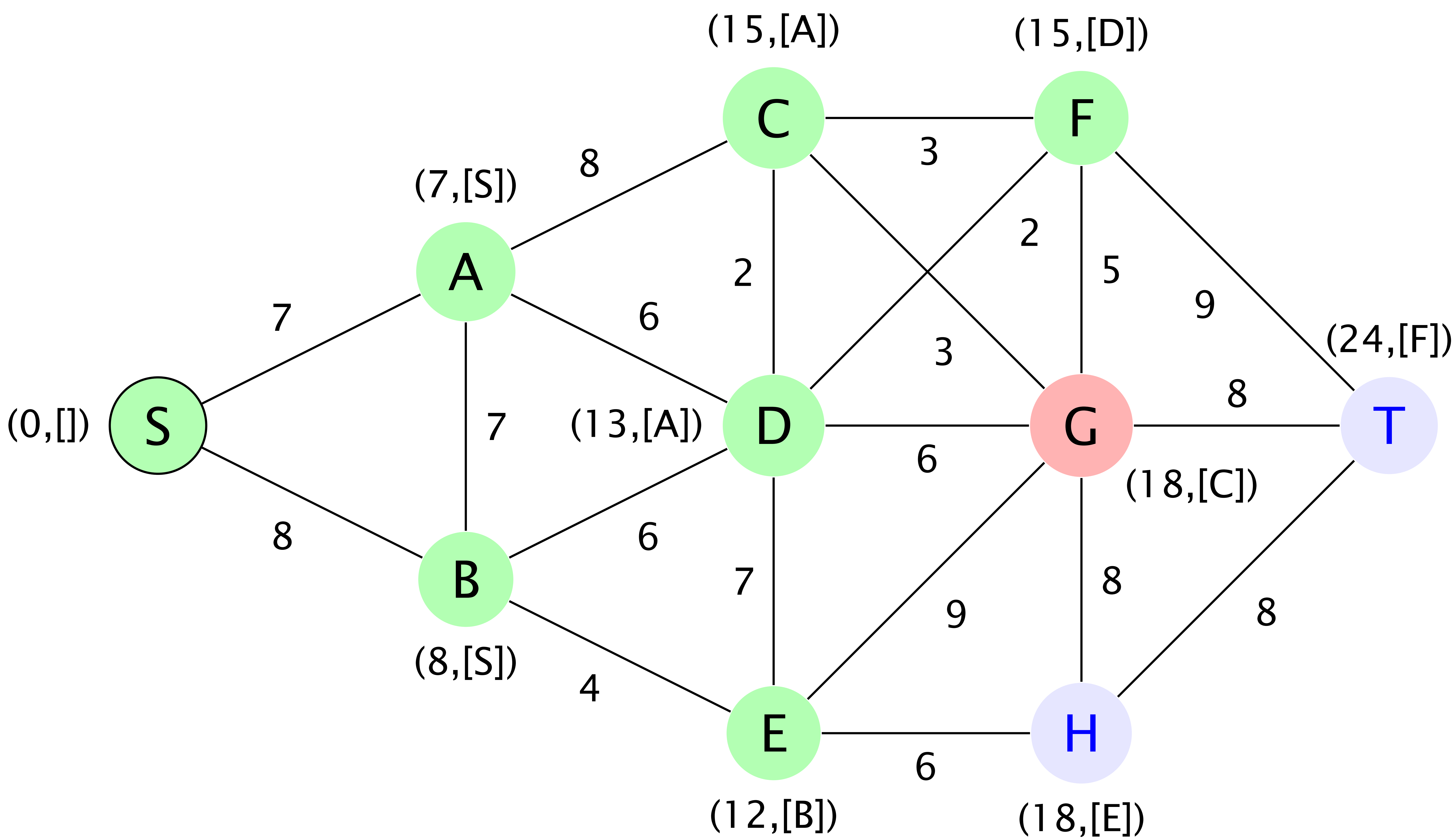
Dijkstra's Algorithm Example 02

Step 7 Process F egDijkstraGraph0207



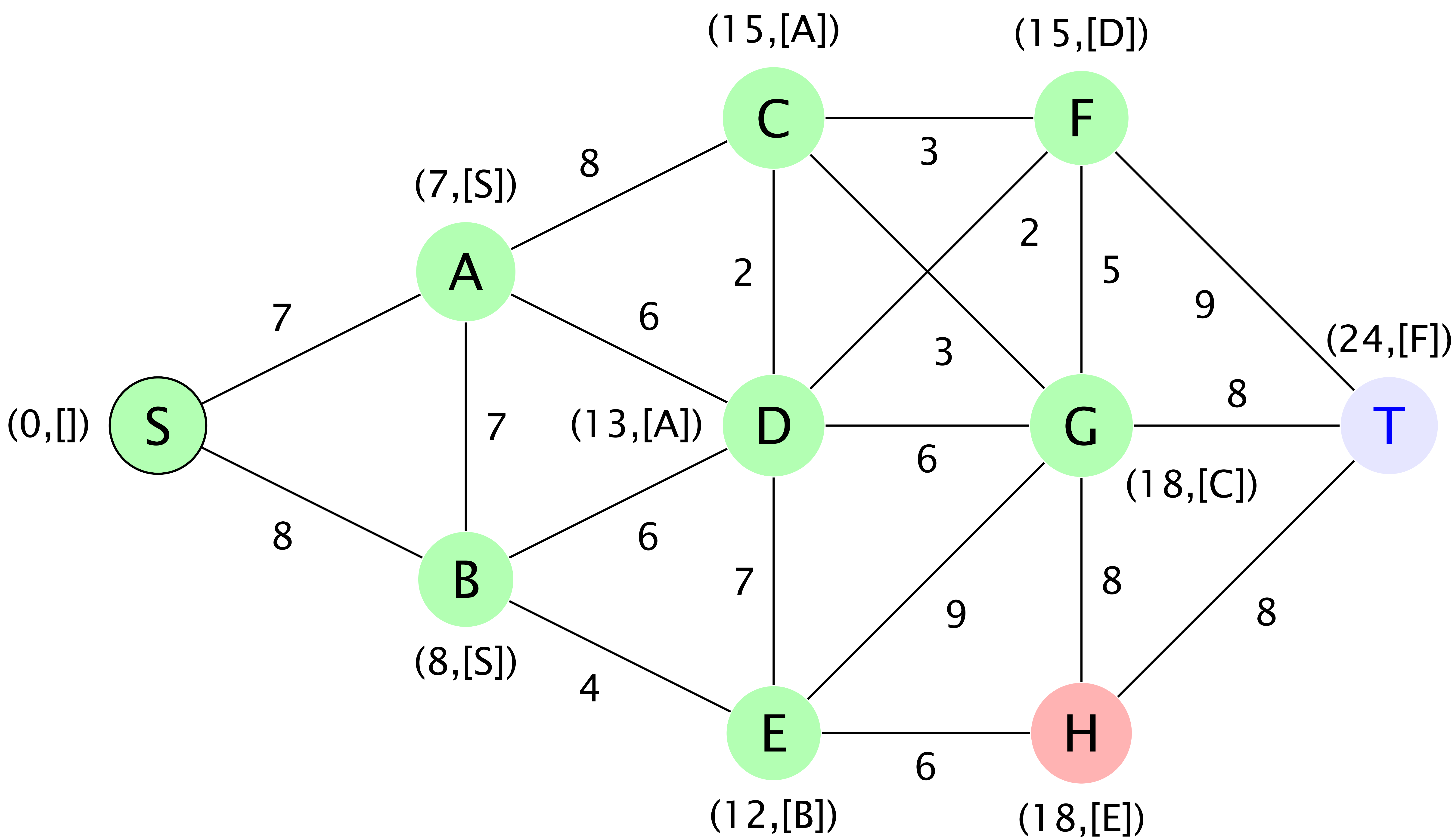
Dijkstra's Algorithm Example 02

Step 8 Process G (or H) egDijkstraGraph0208



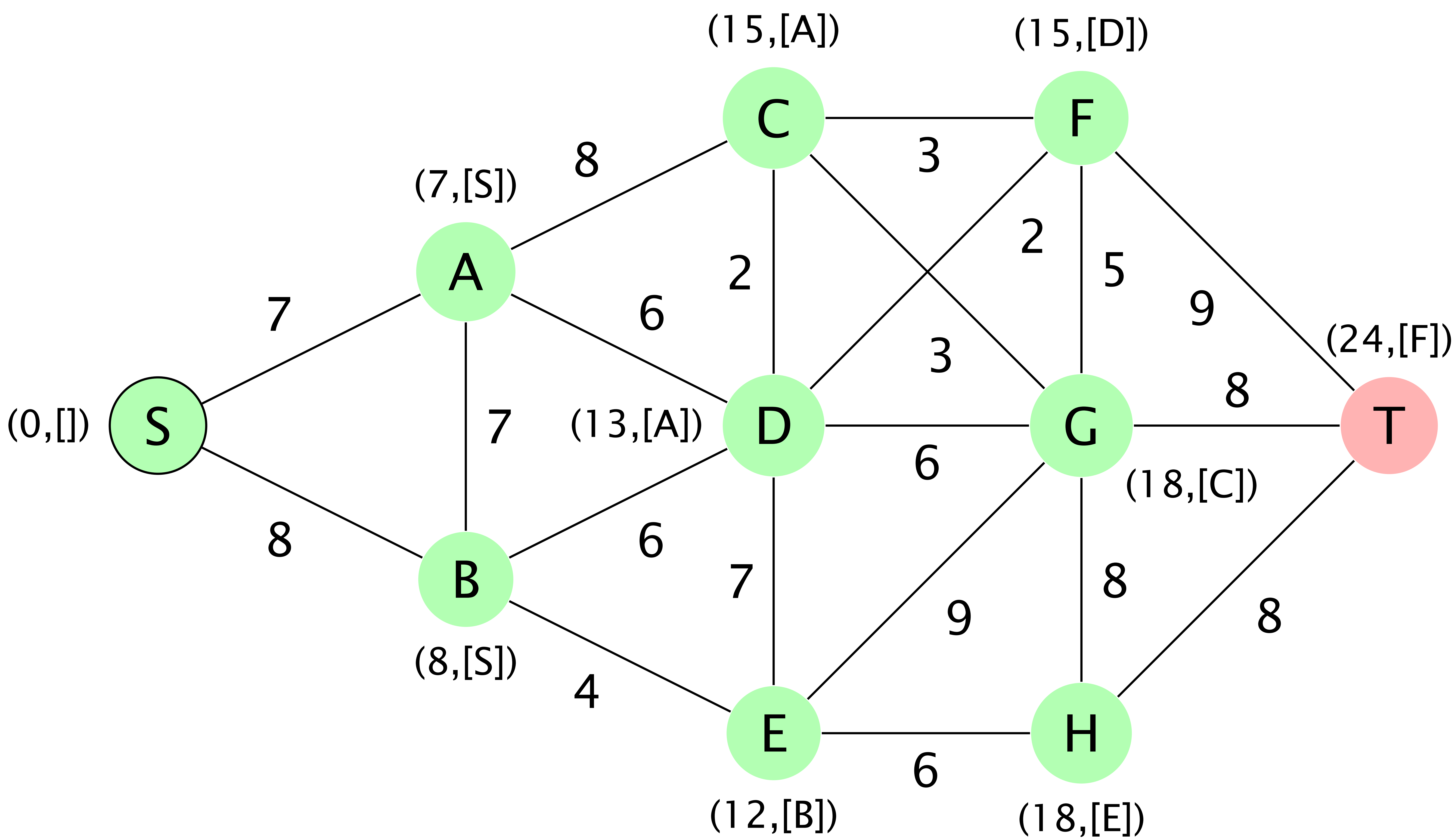
Dijkstra’s Algorithm Example 02

Step 9 Process H egDijkstraGraph0209



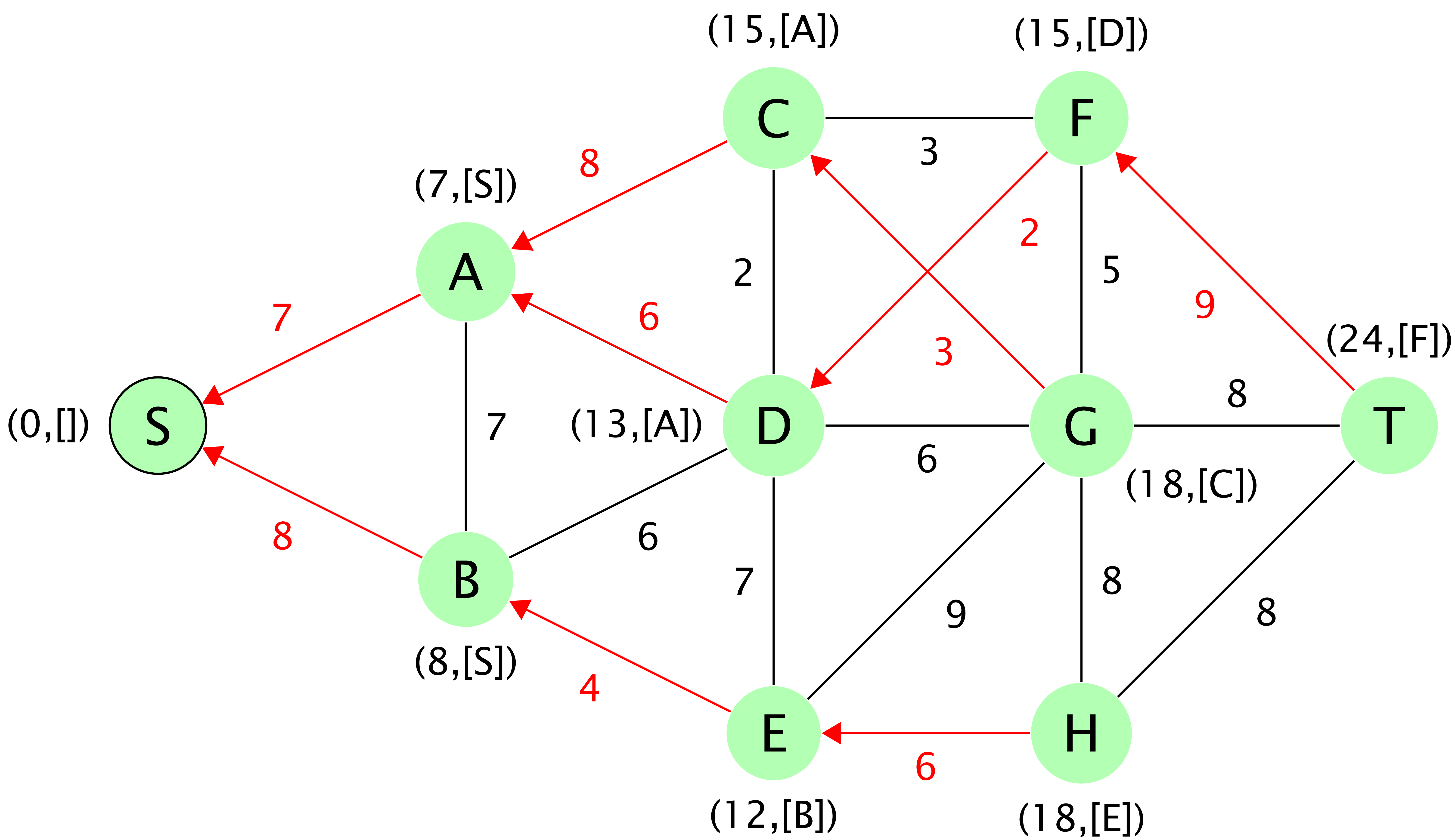
Dijkstra's Algorithm Example 02

Step 10 Process T egDijkstraGraph0210



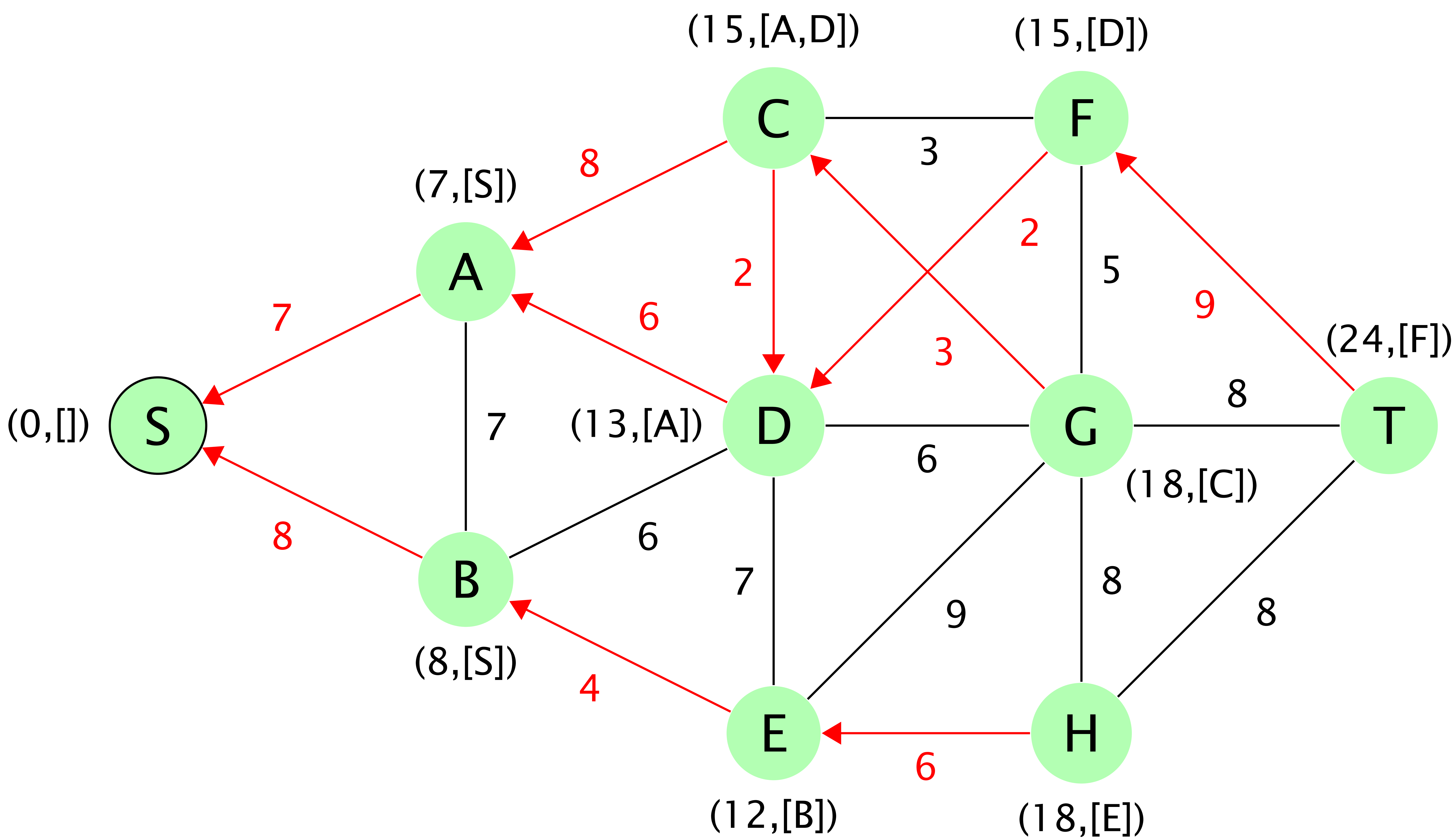
Dijkstra's Algorithm Example 02

Shortest Path Tree egDijkstraGraph02SPT



Dijkstra's Algorithm Example 02

Shortest Path Graph egDijkstraGraph02SPG



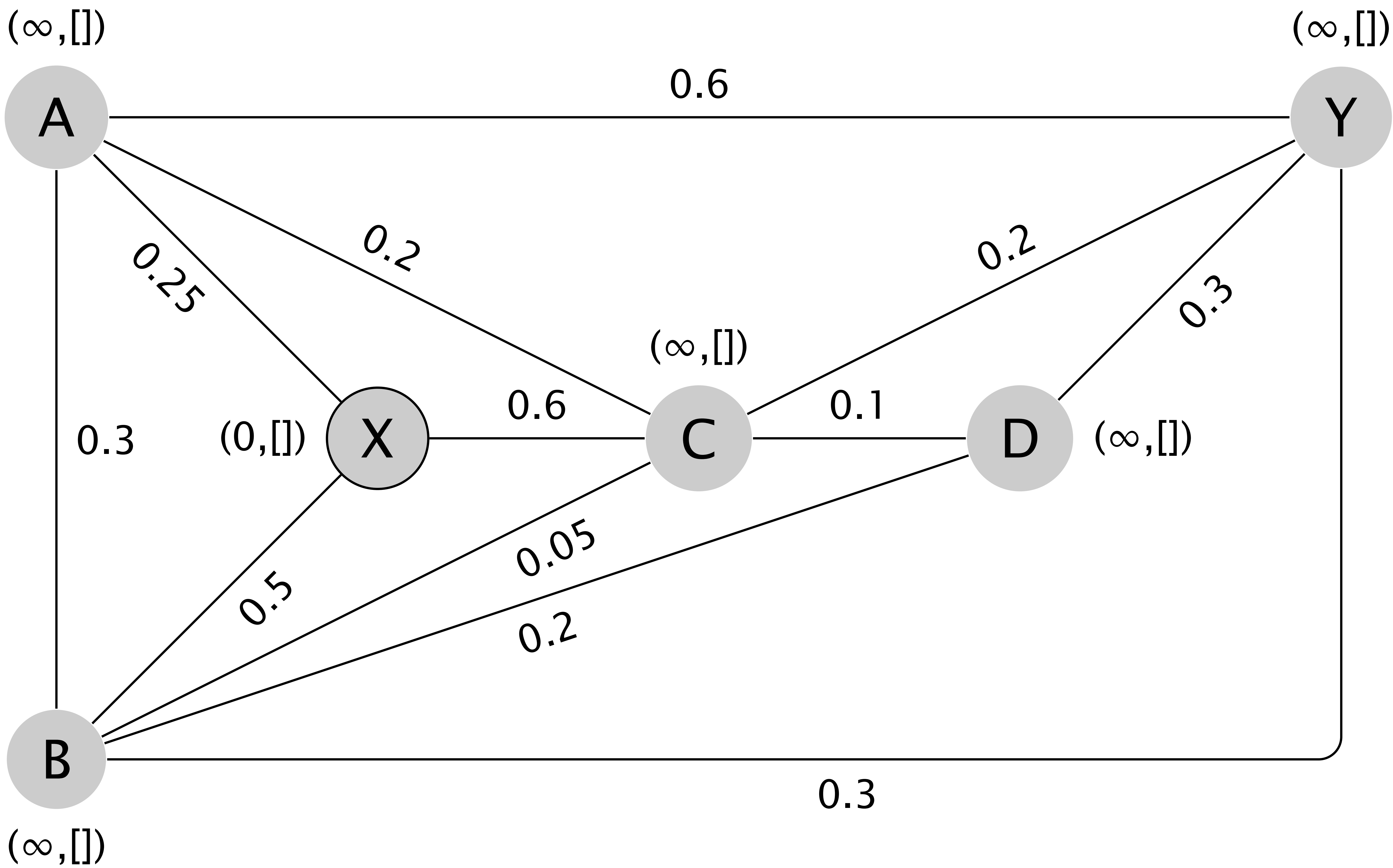
Dijkstra’s Algorithm Example 03

Problem Description

- In the following graph, the weight on each edge represents the probability of failing while traversing the edge
- *Problem:* find the path that maximises the chance of traversing from X to Y

Dijkstra's Algorithm Example 03

Step 0 Initialisation egDijkstraGraph0300



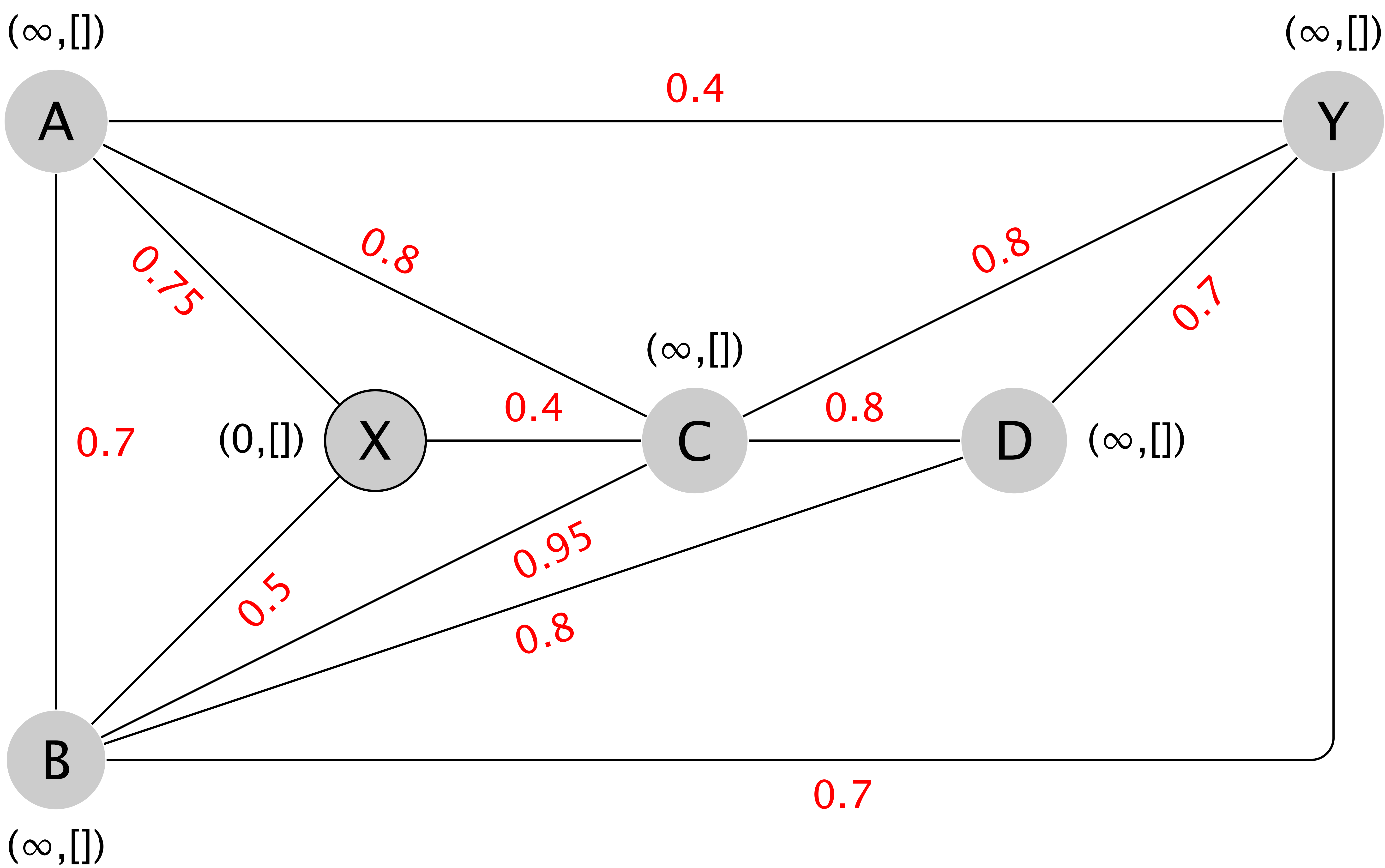
Dijkstra's Algorithm Example 03

Formulation as Shortest Path

- Let $p_{(i,j)}$ be probability of failing on edge (i, j)
- The probability of not failing is $x_{(i,j)} = 1 - p_{(i,j)}$
- Over any path $x_{(i,j)}$ are independent so problem is to maximise probability of not failing $\prod_{(i,j) \in path} x_{(i,j)}$
- Equivalently, if $y_{(i,j)} = \log x_{(i,j)}$ then problem is to maximise $\sum_{(i,j) \in path} y_{(i,j)}$
- Alternatively, since $y_{(i,j)} \in (-\infty, 0]$ as $x_{(i,j)} \in [0, 1]$ then let $z_{(i,j)} = -100 y_{(i,j)}$ and minimise $\sum_{(i,j) \in path} z_{(i,j)}$

Dijkstra's Algorithm Example 03

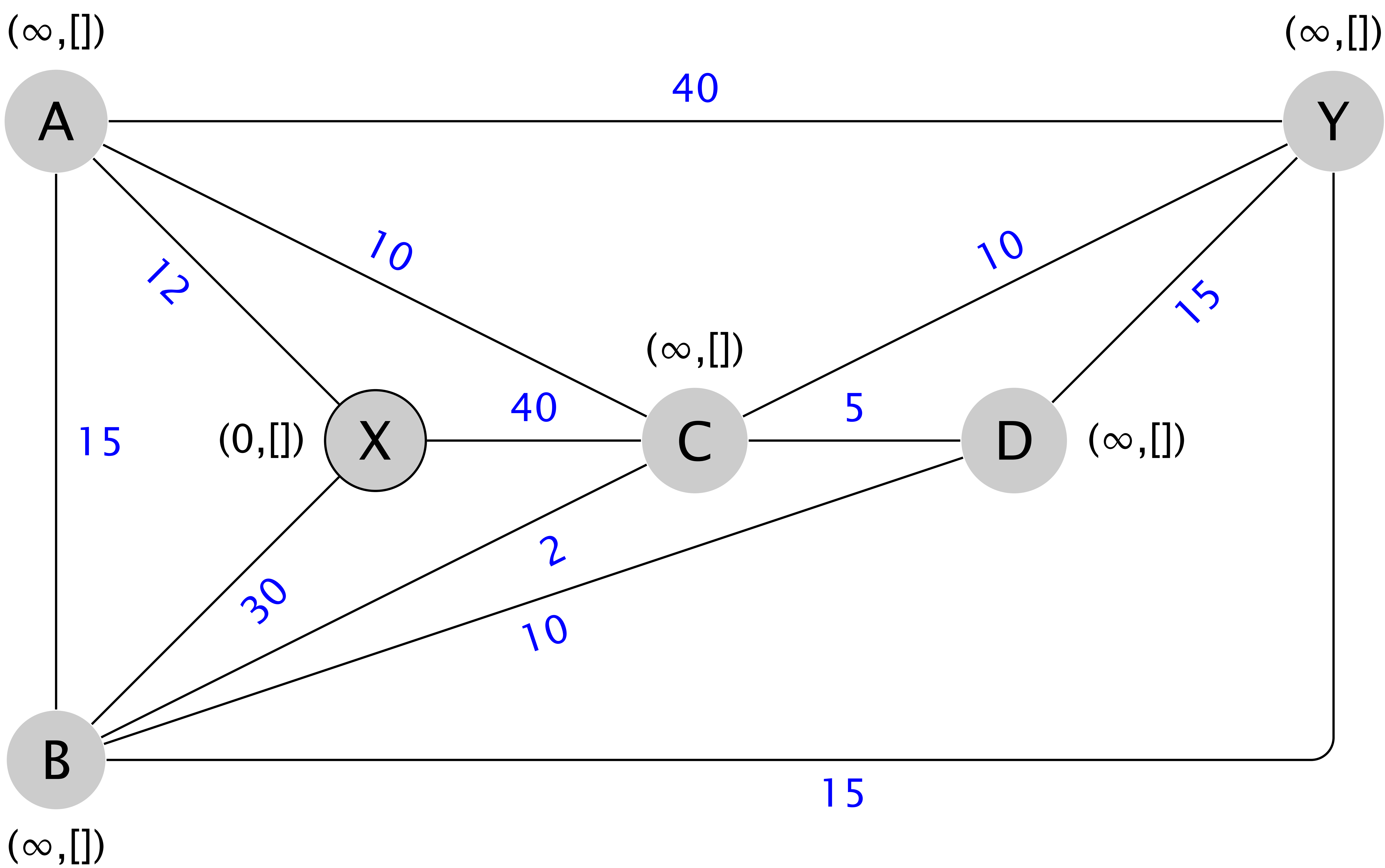
Step 0 Reformulation (a) egDijkstraGraph0300a



- The numbers in red are the probabilities of not failing
- $x_{(i,j)} = 1 - p_{(i,j)}$

Dijkstra's Algorithm Example 03

Step 0 Reformulation (b) egDijkstraGraph0300b



- The numbers in blue are negated scaled logs of $x_{(i,j)}$
- $z_{(i,j)} = -100 \log_{10} x_{(i,j)}$

Prim's Algorithm

Structured English

```
prim(gr,weight,r)
  for u in vertices(gr)
    key(u) = Infinity
    label(u) = Temp
  key(r) = 0
  pred(r) = None
  q = makePriorityQ(vertices(gr))

  while not isEmptyPQ(q)
    u = extractMinPQ(q)
    for v in adj(gr,u)
      if (label(v) == Temp
          and weight(edge(u,v)) < key(v))
        key(v) = weight(edge(u,v))
        q = decreaseKeyPQ(q,v,key(v))
        pred(v) = u
    label(u) = Permanent
```

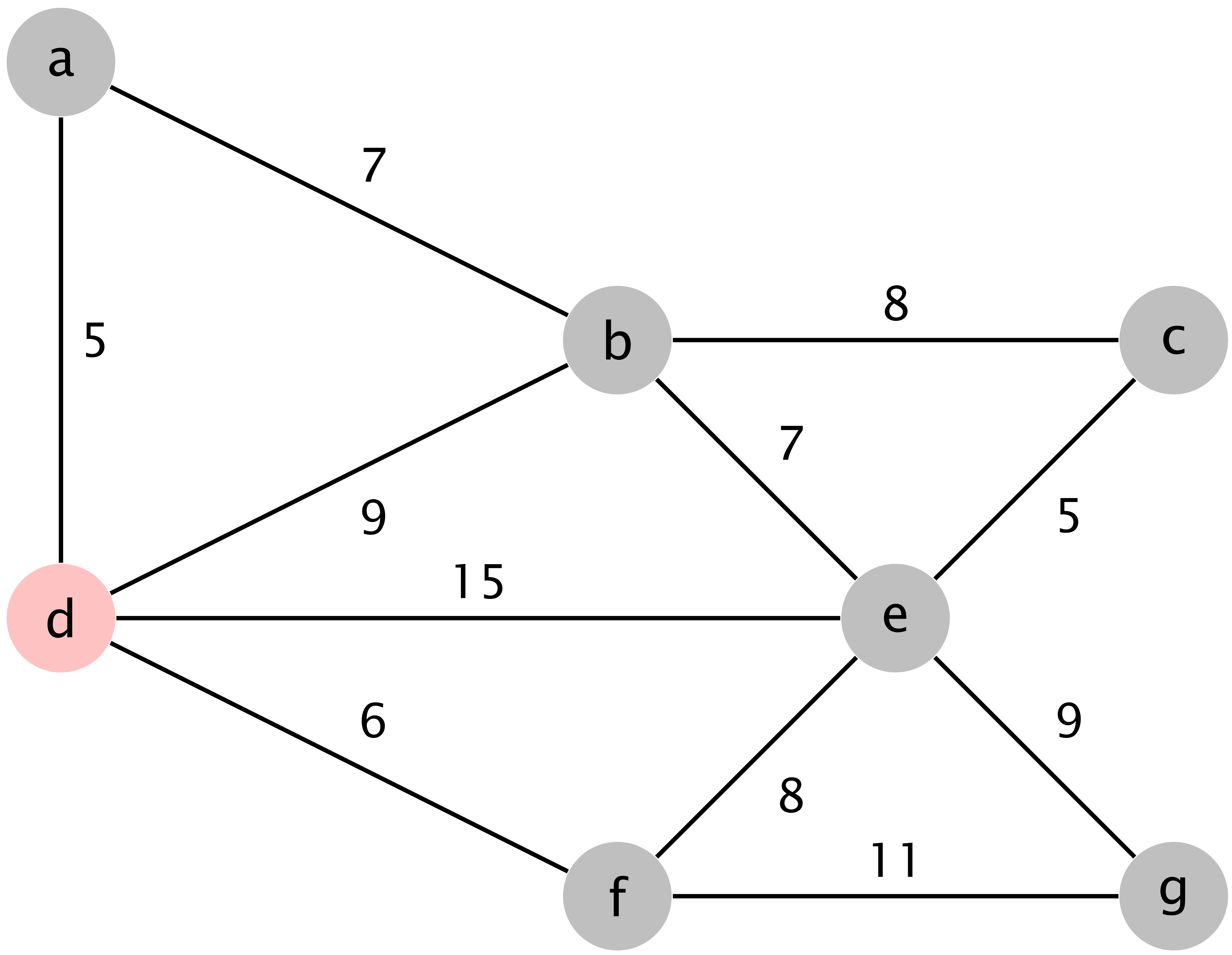
Dijkstra's and Prim's Algorithms

Comparison

- Both are examples of *greedy* algorithms
- They choose the next best edge to add to the permanently labelled set
- The algorithms are very similar
- Process each vertex, v, in turn from a priority queue
- Examine all vertices adjacent to v and perform *relaxation*
- *relaxation* means updating the distances or keys
- For the term *relaxation* see Cormen (2009, page 648) has a footnote explaining the origin of the term *relaxation*

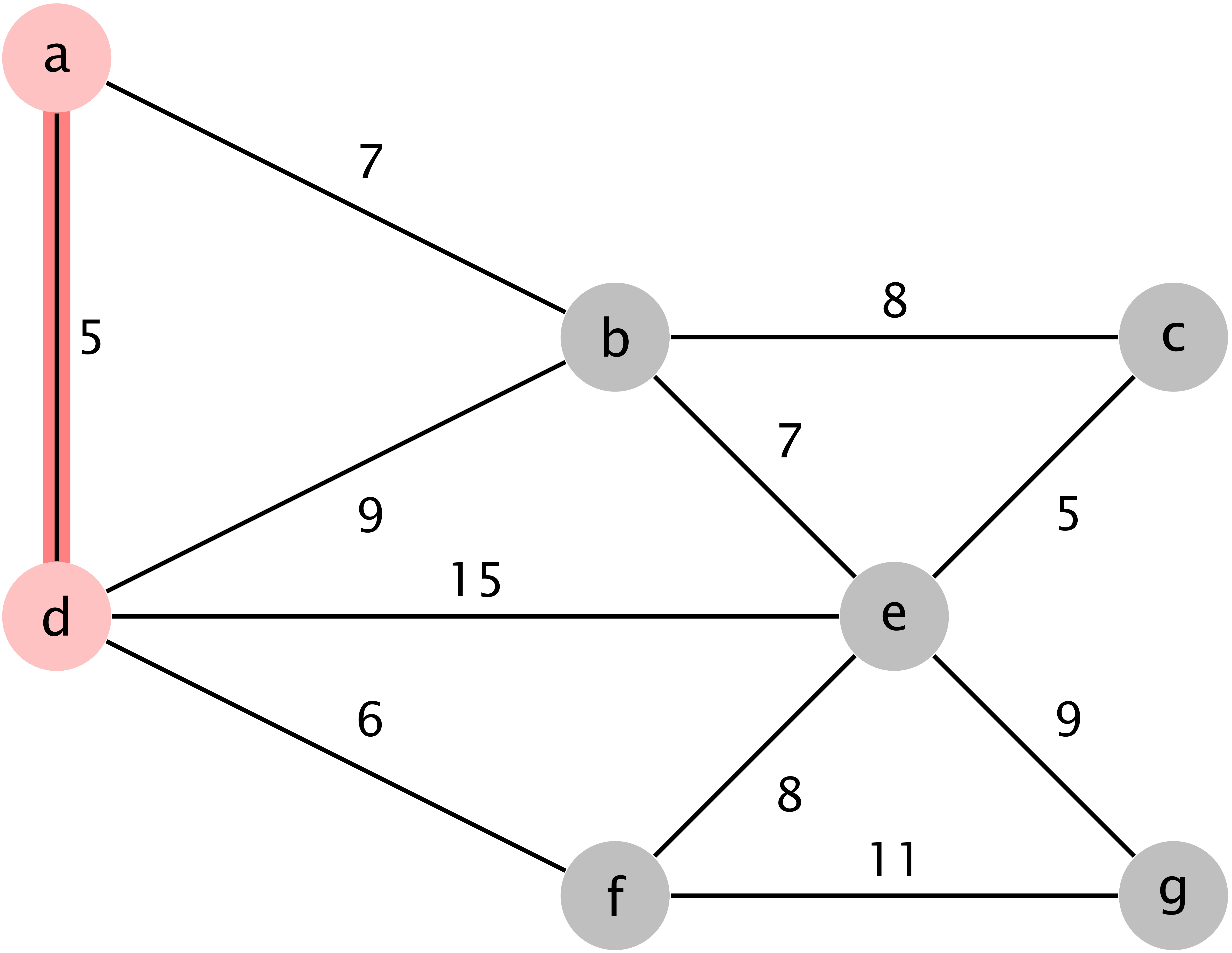
Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



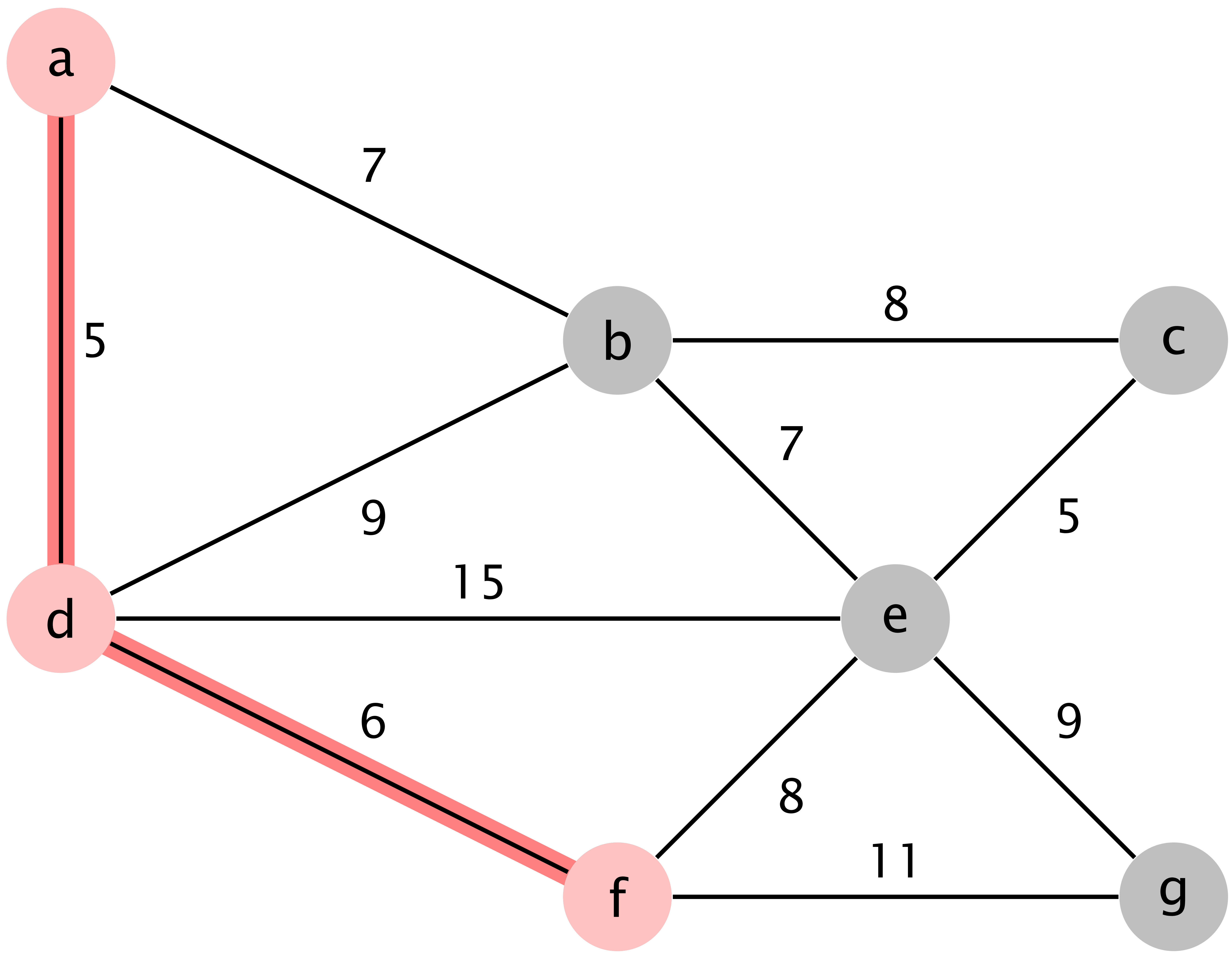
Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



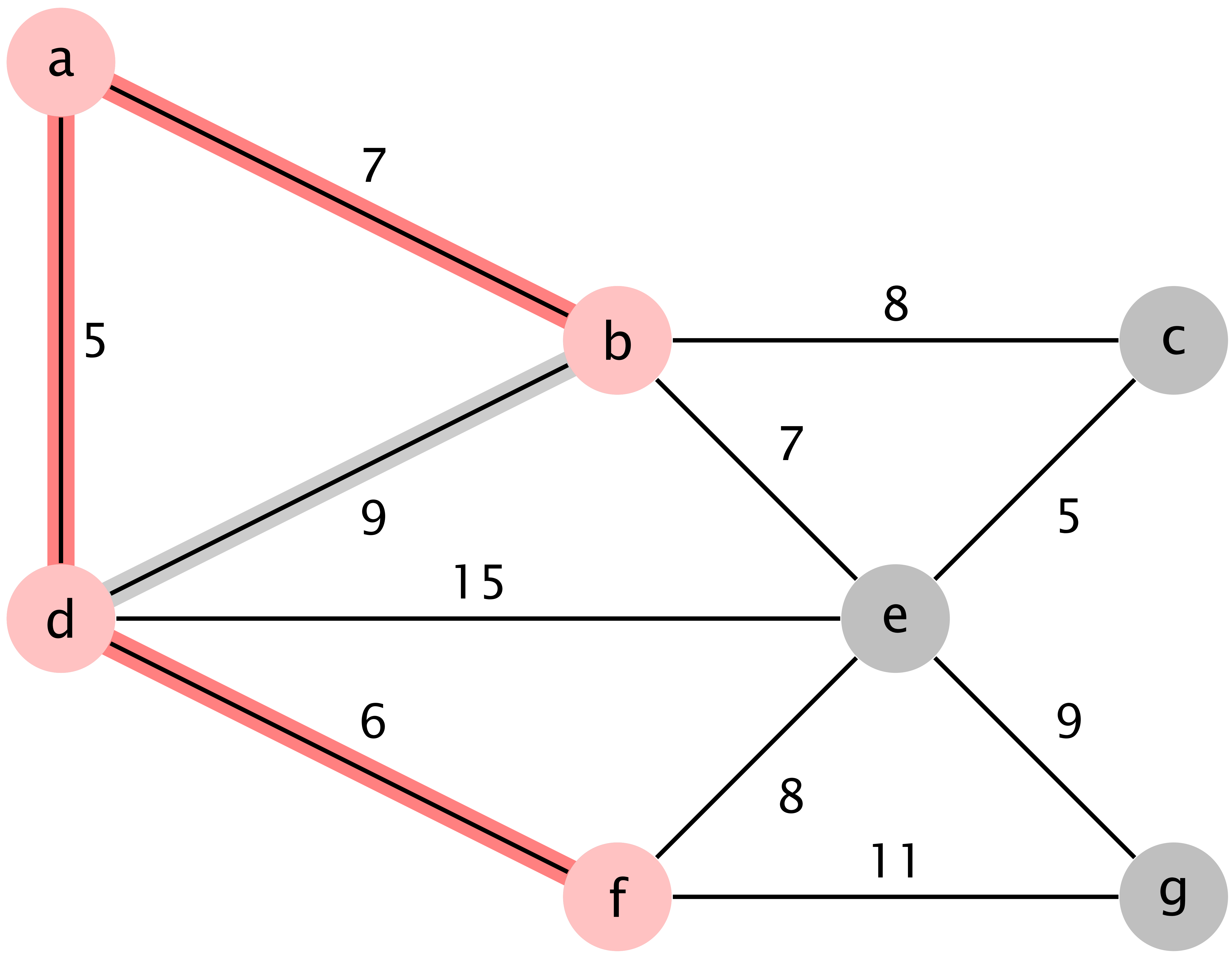
Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



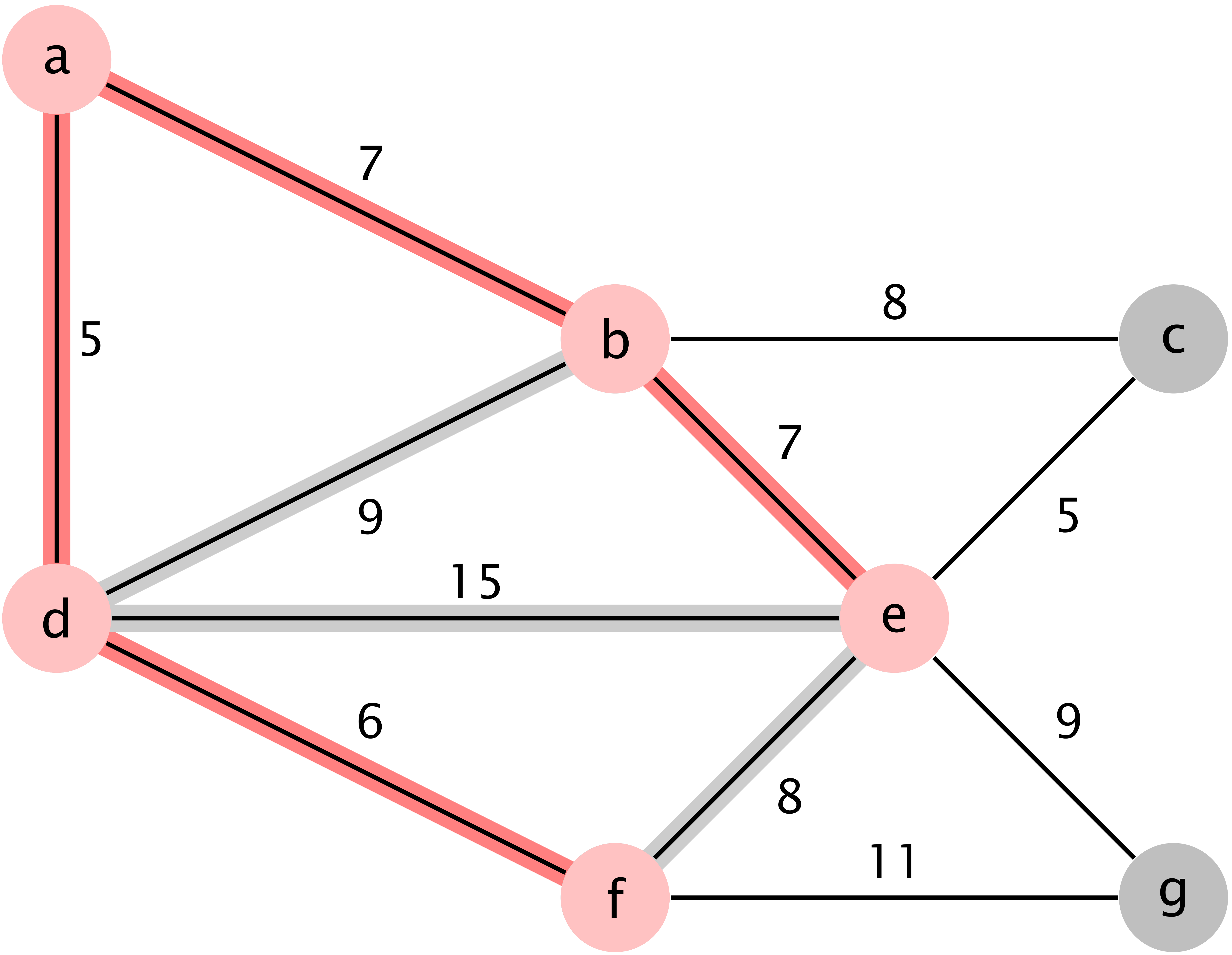
Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



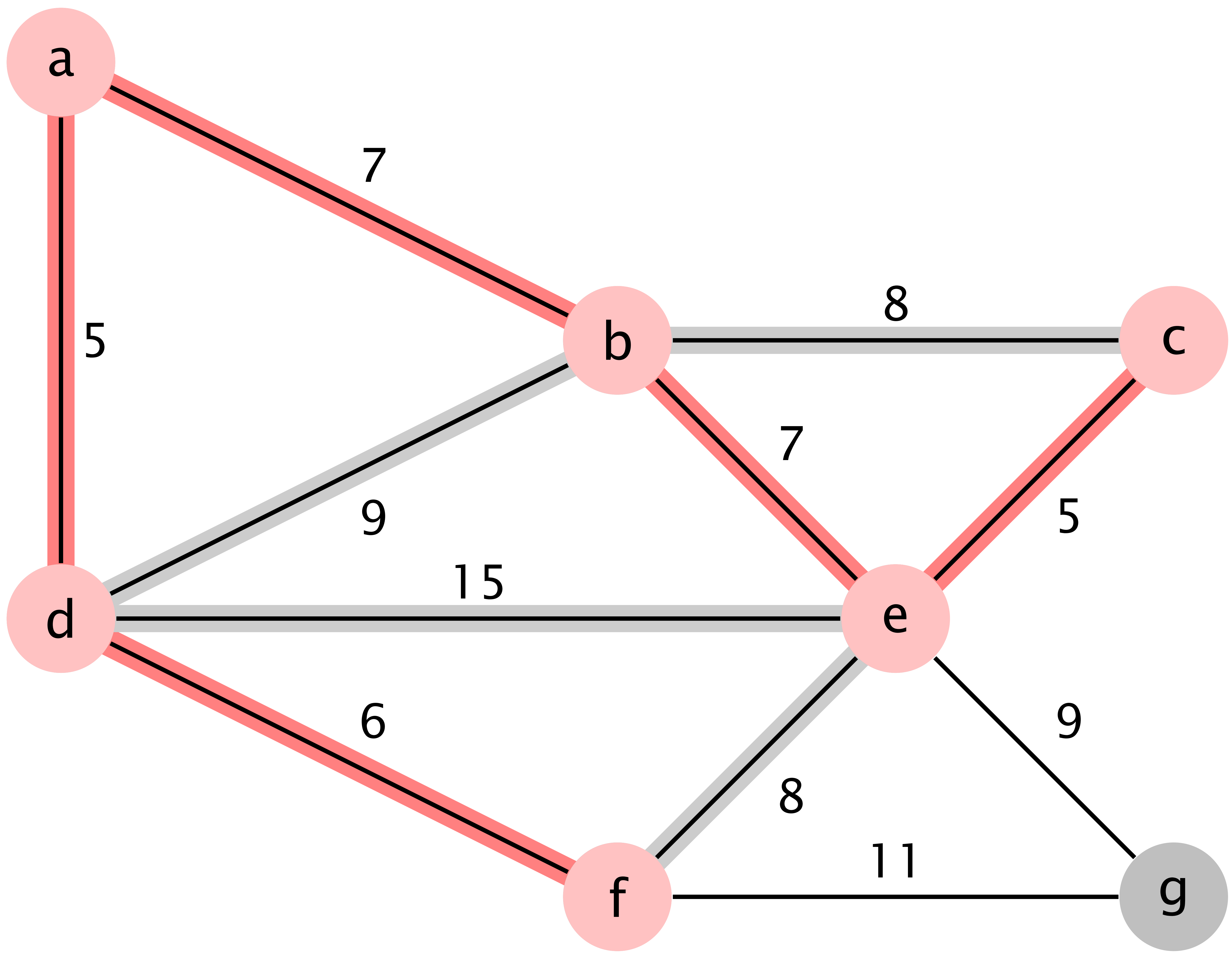
Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



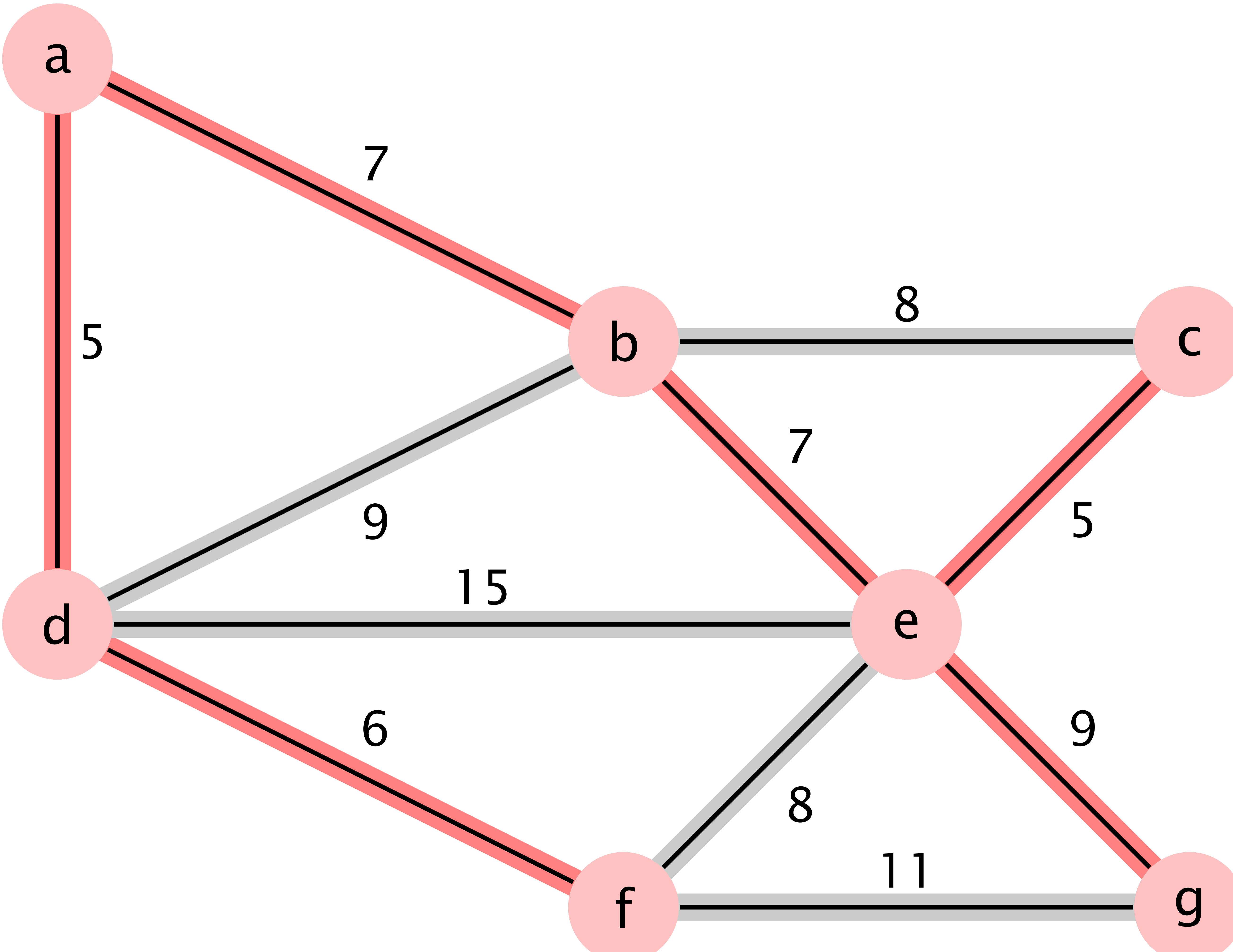
Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



Tutorial Material — Prim's algorithm

Example Graph 01 egPrimGraph00



Greedy Algorithms

Overview

- Greedy algorithms follow the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum
- In general this rarely works — but it does in some cases including
  - Dijkstra's algorithm and A* search algorithm for graph search and shortest path finding
  - Kruskal's algorithm and Prim's algorithm for constructing minimum spanning trees of a given connected graph
  - Interval scheduling or Activity selection problem to find the maximum number of activities that do not clash with each other
- If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods such as dynamic programming.

Greedy Algorithms

Interval Scheduling

- Interval scheduling
- Job j starts at s_j and finishes at f_j
- Two jobs are compatible if they do not overlap
- **Q** What is the maximum subset of mutually compatible jobs?
- **Greedy template** Consider jobs in some order. Take each job provided it is compatible with the ones already taken.
- **Exercise** What orderings can we have ?
- Example from Greedy algorithms: Interval scheduling

Greedy Algorithms

Interval Scheduling

- **Greedy template** Consider jobs in some order. Take each job provided it is compatible with the ones already taken.
- **Earliest start time** Consider jobs in ascending order of start time s_j
- **Earliest finish time** Consider jobs in ascending order of finish time f_j
- **Shortest interval** Consider jobs in ascending order of interval length $f_j + 1 - s_j$
- **Fewest conflicts** For each job, count the number of conflicting jobs c_j and schedule in ascending order of conflicts c_j

Interval Scheduling

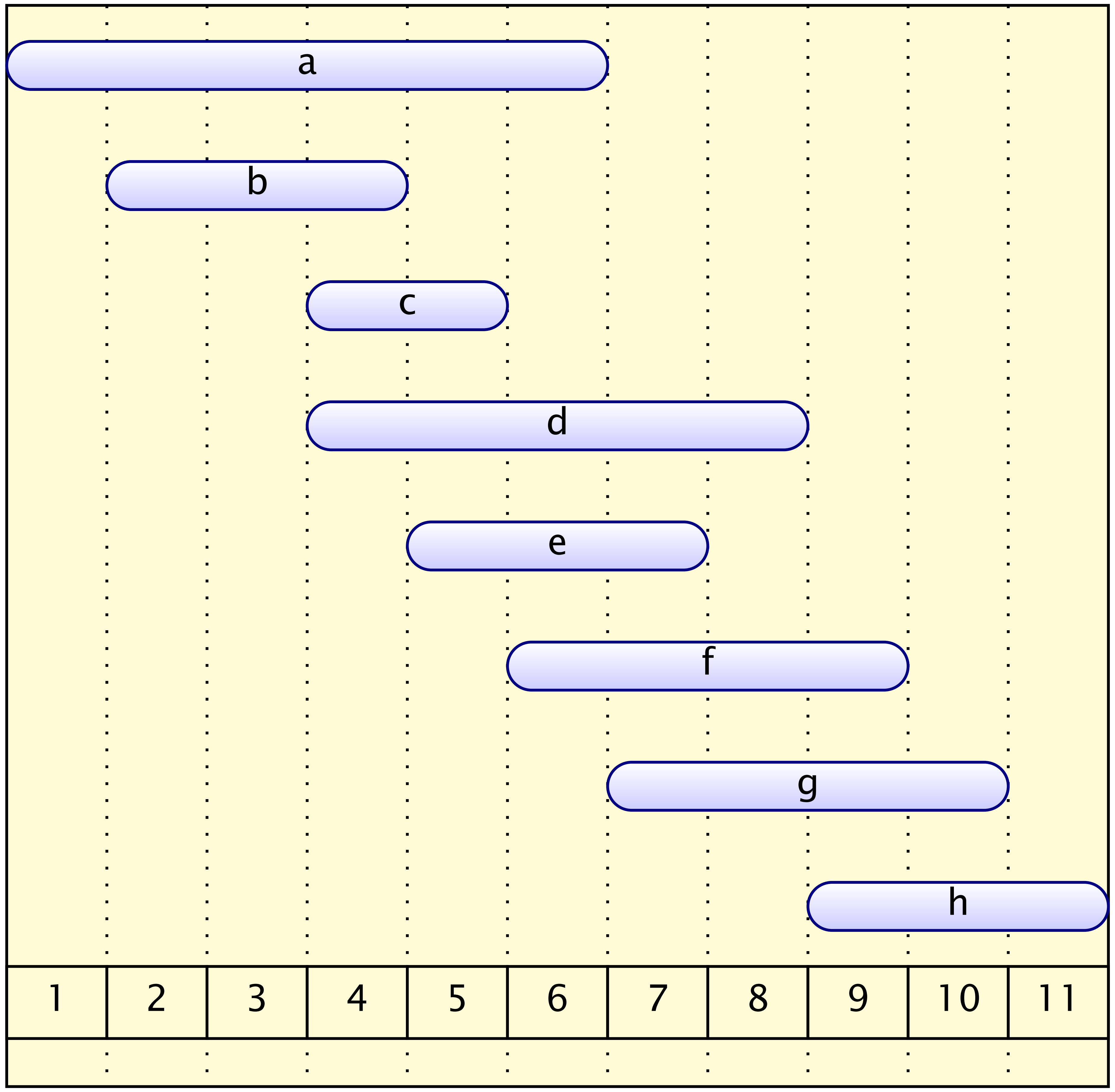
Example

- For the jobs given below, produce an ordering by each of the greedy templates (above) and the schedule produced
- Each triple in the list below means $(\mathtt{name}, s_i, f_i)$ where the times are inclusive

```
jobs
 = [(a,1,6),(b,2,4),(c,4,5),(d,4,8)
   ,(e,5,7),(f,6,9),(g,7,10),(h,9,11)]
```

Interval Scheduling

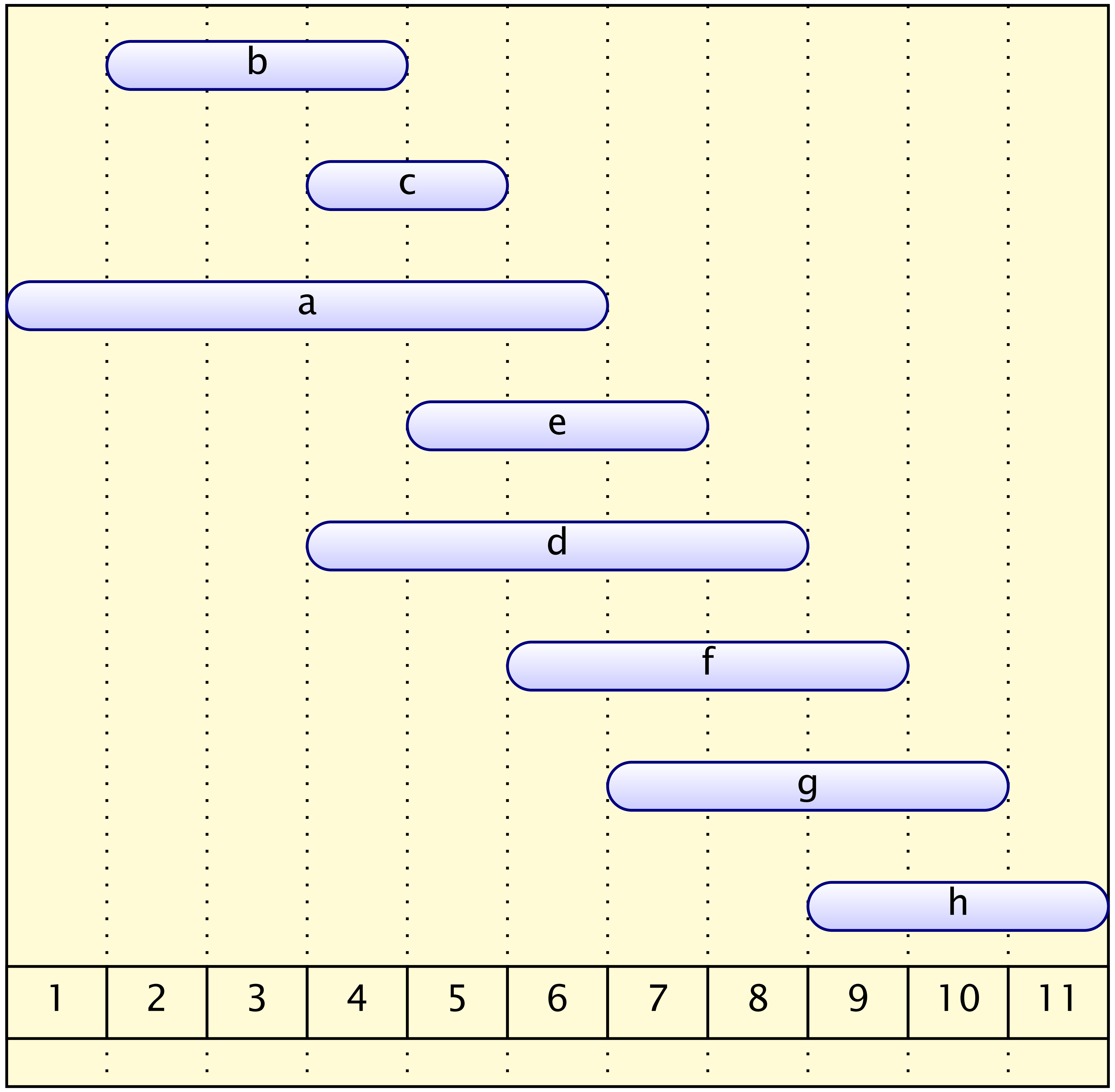
Order by Earliest Start Time egGantt01EST



- Schedule jobs **a**, **g** (2 jobs)

Interval Scheduling

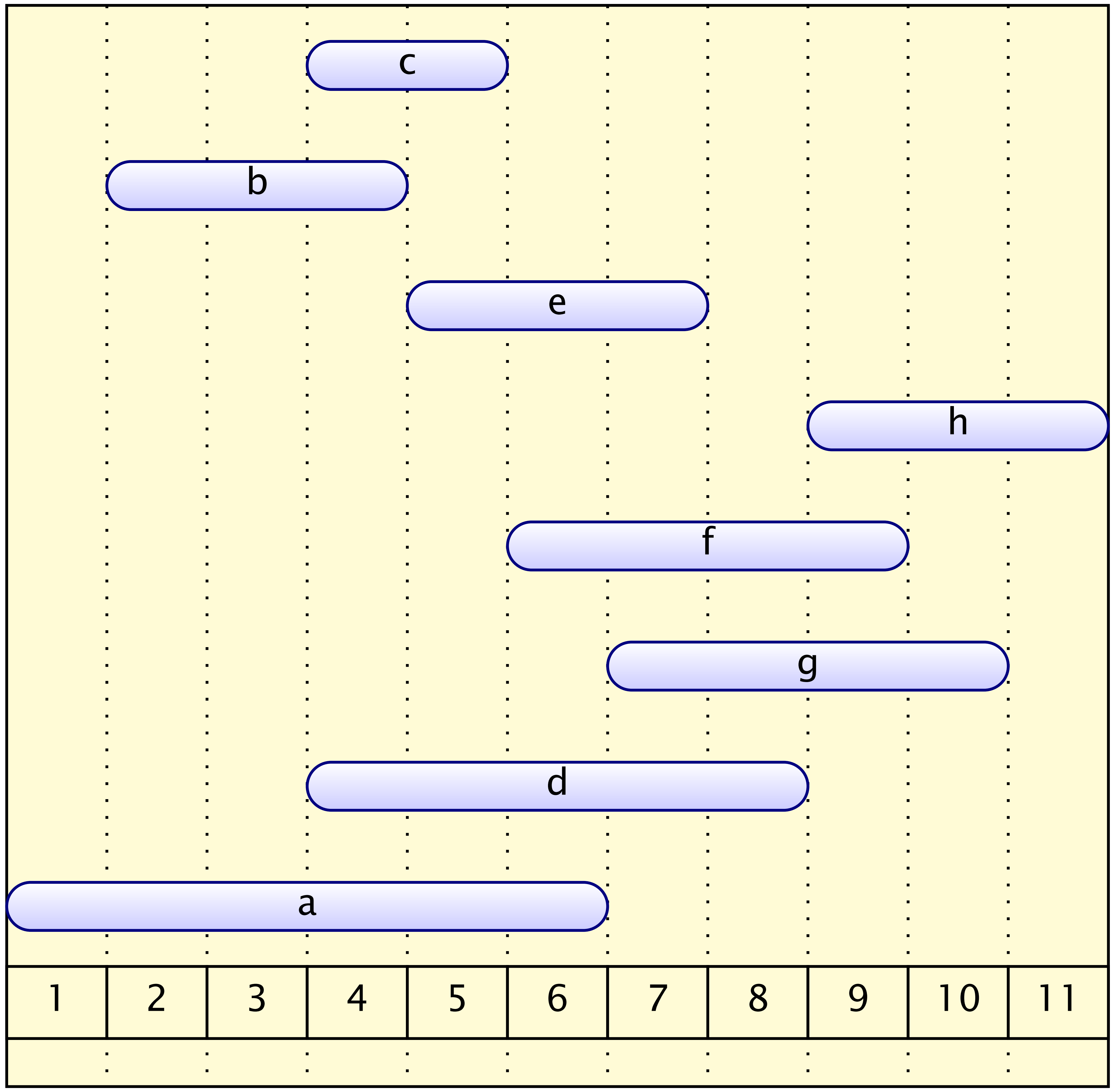
Order by Earliest Finish Time egGantt01EFT



- Schedule jobs **b**, **e**, **h** (3 jobs)

Interval Scheduling

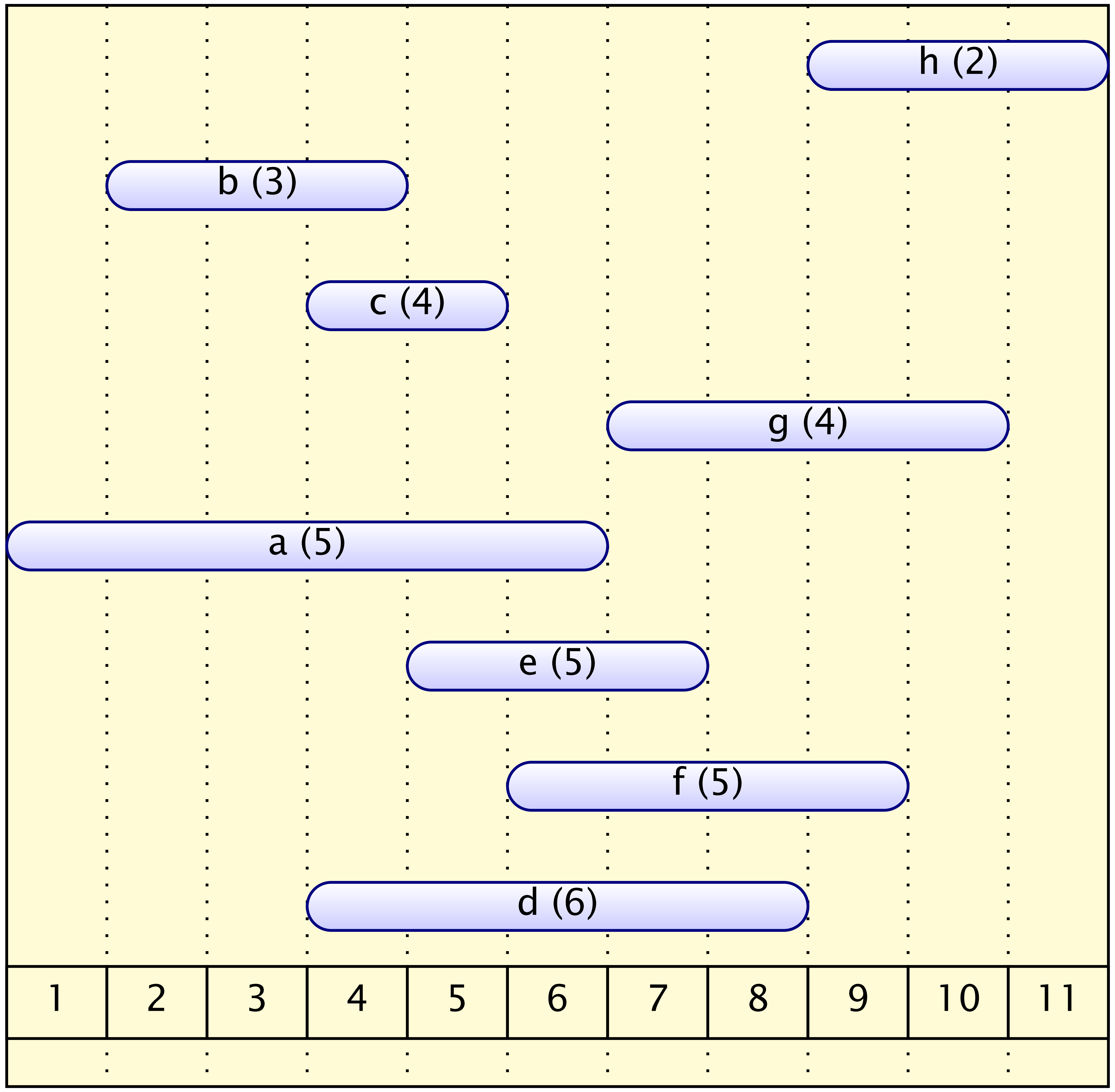
Order by Shortest Interval egGantt01ShortInt



- Schedule jobs **c**, **h** (2 jobs)

Interval Scheduling

Order by Fewest Conflicts egGantt01Conflicts



- Schedule jobs **h**, **b**, **e** (3 jobs)

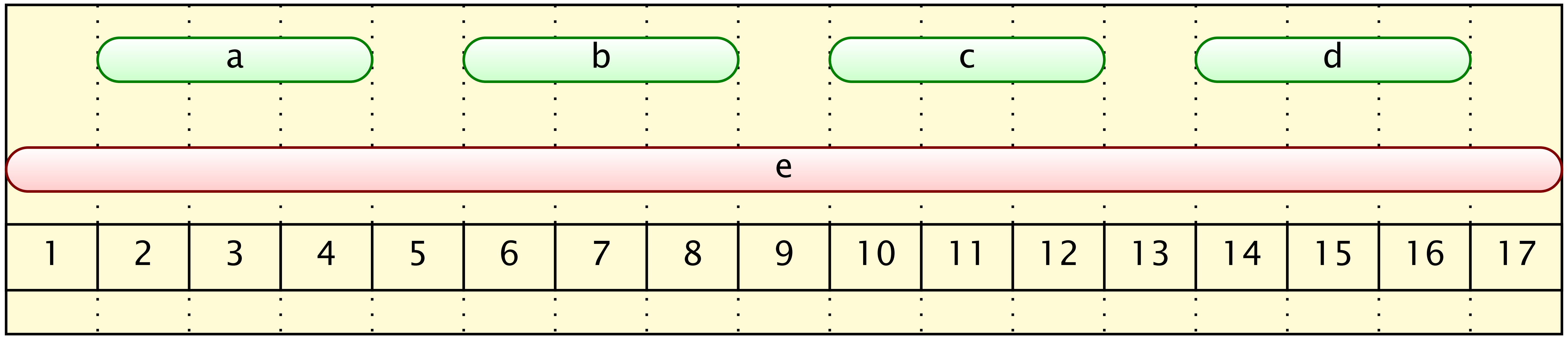
Interval Scheduling

Counter Examples

- For each of the following *Greedy Templates* produce a counter example to show it may not produce the optimal schedule
- **Earliest start time** Consider jobs in ascending order of start time s_j
- **Shortest interval** Consider jobs in ascending order of interval length $f_j + 1 - s_j$
- **Fewest conflicts** For each job, count the number of conflicting jobs c_j and schedule in ascending order of conflicts c_j

Interval Scheduling

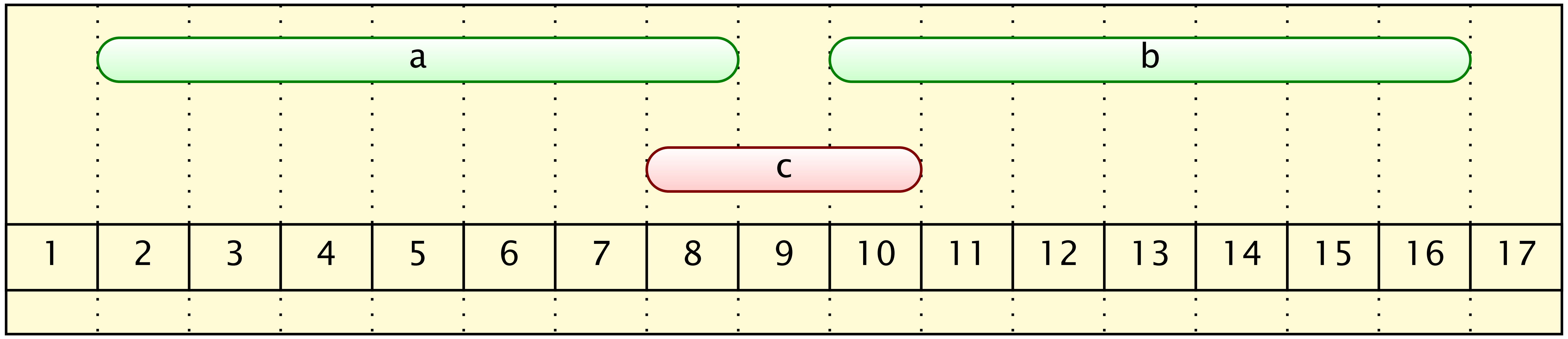
Order by Earliest Start Time — Counter Example egGantt01ESTcntr



- **e** dominates the optimal schedule by starting earlier and overlapping the others

Interval Scheduling

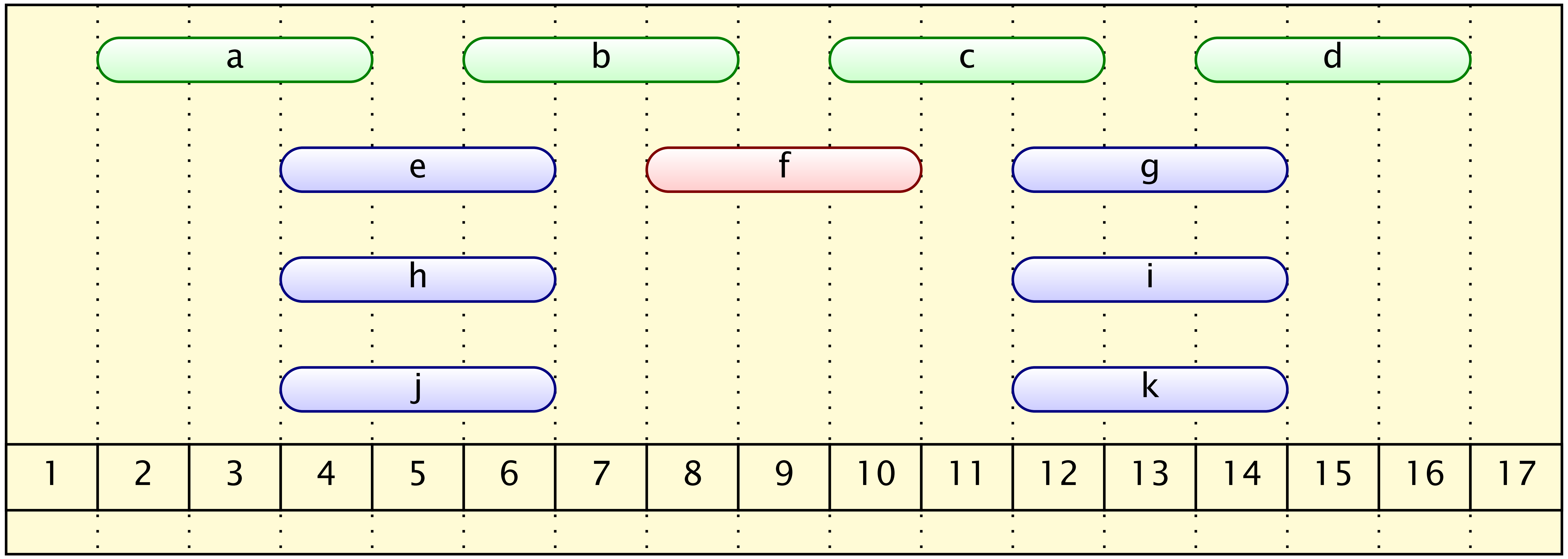
Order by Shortest Interval — Counter Example egGantt01ShortIntCntr



- **c** dominates the optimal schedule y being shorter and overlapping the other two

Interval Scheduling

Order by Fewest Conflicts — Counter Example egGantt01ConflictsCntr



- **f** dominates the optimal schedule by only having two conflicts and overlapping **b** and **c**

Interval Scheduling

Order by Earliest Finish Time — Optimality Proof

- Basic structure of correctness proof:
- Assume that there is an optimal solution that is different from the greedy solution.
- Find the *first* difference between the two solutions.
- Argue that we can exchange the optimal choice for the greedy choice without making the solution worse (although the exchange might not make it better).
- This argument implies by induction that some optimal solution *contains* the entire greedy solution, and therefore *equals* the greedy solution.
- Sometimes, an additional step is required to show no optimal solution *strictly* improves the greedy solution.
- See Jeff Erickson: Algorithms
- Proof also in Interval Scheduling and Greedy Algorithms
- The slides at Kevin Wayne: Greedy Algorithms are from Kleinberg (2013)