# Overcoming non Distributivity

## A Case Study through Functional Programming

Juan C Saenz-Carrasco and Mike Stannett

# Agenda

Introduction

Definitions

The Problem

A Functional Approach

Applications

Conclusion

# Introduction

In order to solve path finding problems, we should take care about some properties prior the computation of the corresponding algorithm.

Some of such properties are:

- associativity
- distributivity

Same goes for the definitions of the operators involved in the (host) algorithm

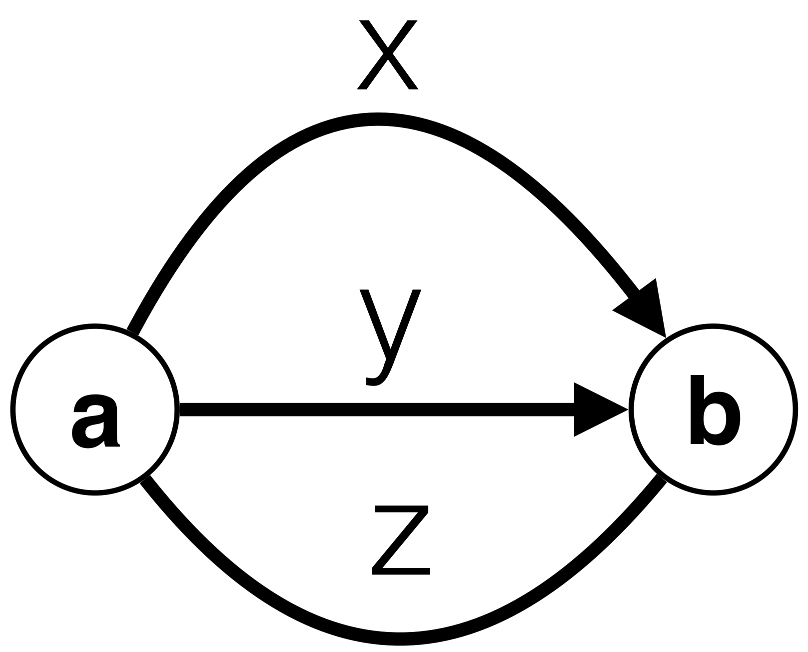
# Definitions: Path Addition

# Definitions: Path Addition

We consider *addition* to the computation of the labels (or weights) of two or more edges or paths:

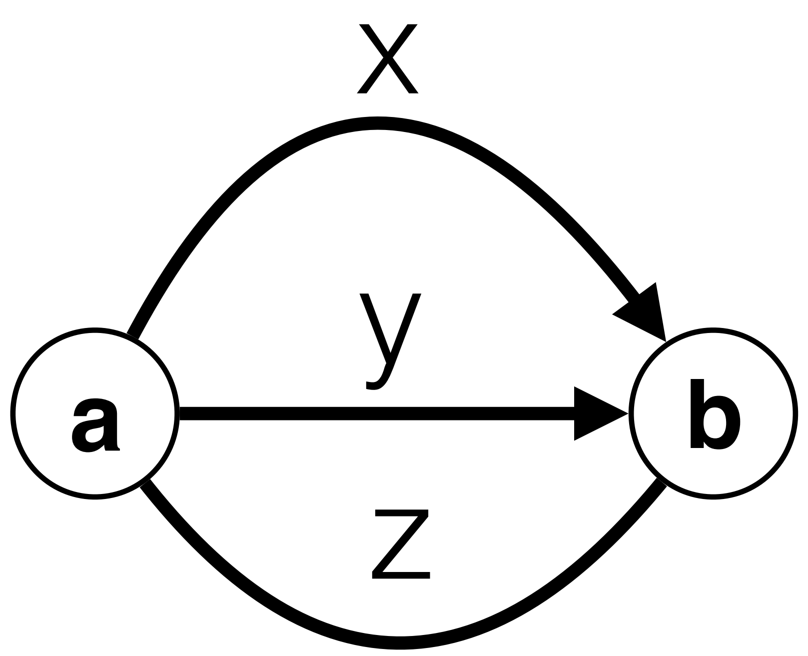
# Definitions: Path Addition

We consider *addition* to the computation of the labels (or weights) of two or more edges or paths:



# Definitions: Path Addition

We consider *addition* to the computation of the labels (or weights) of two or more edges or paths:



The addition of paths from `a` to `b` can be denoted as `x` ⊕ `y` ⊕ `z`, provided the definition for ⊕.

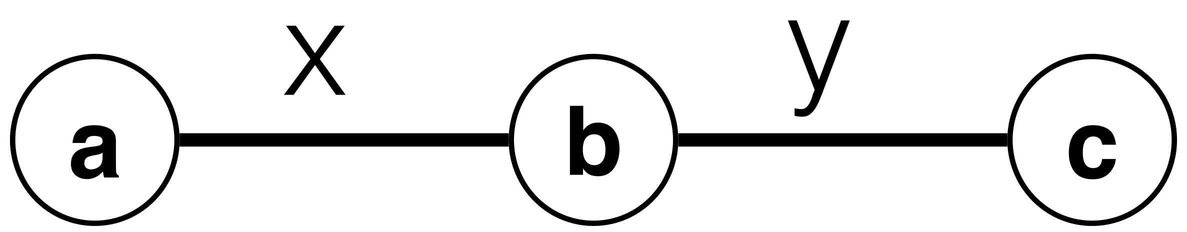
# Definitions: Path Multiplication

# Definitions: Path Multiplication

We consider *multiplication* to the computation of the labels (or weights) of two or more consecutive edges or paths:

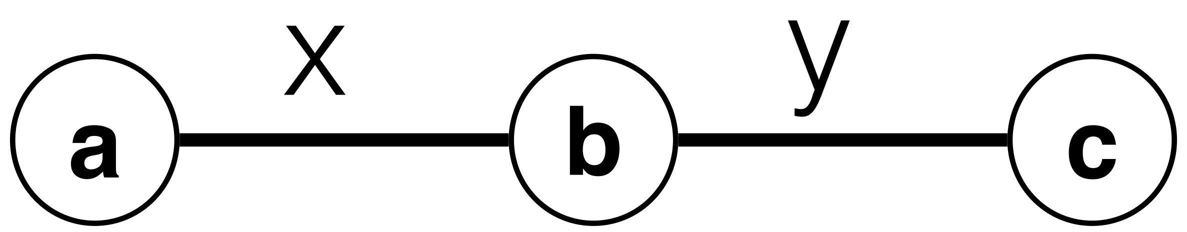
# Definitions: Path Multiplication

We consider *multiplication* to the computation of the labels (or weights) of two or more consecutive edges or paths:



# Definitions: Path Multiplication

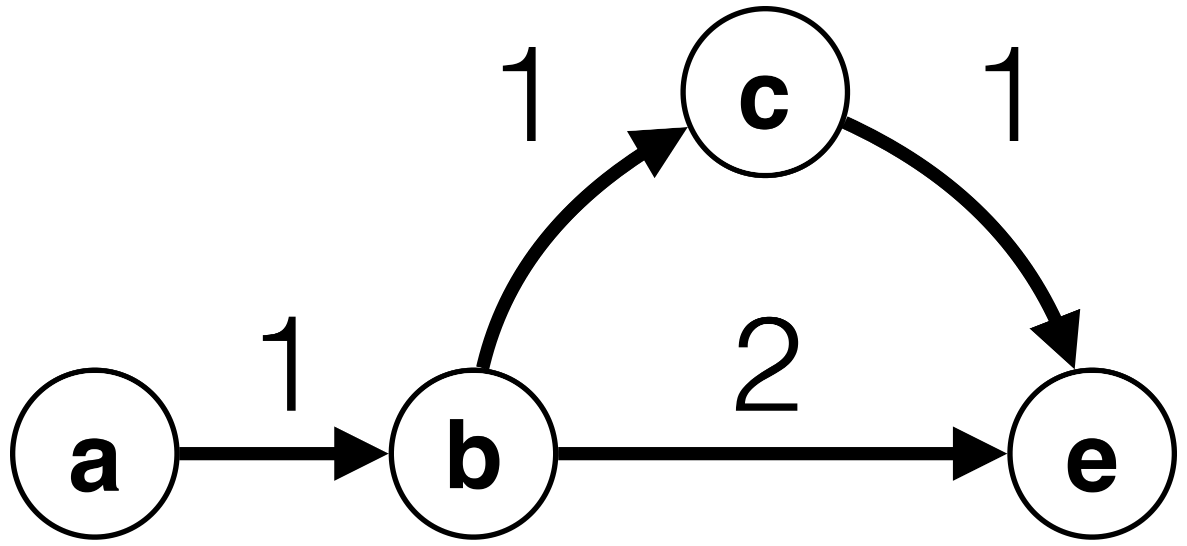
We consider *multiplication* to the computation of the labels (or weights) of two or more consecutive edges or paths:



The multiplication of paths from `a` to `c` can be denoted as `x` ⊗ `y`, provided the definition for ⊗.

# Example

Let us compute the maximum capacity problem for the following graph, where operators $\otimes_1$ = minimum (or ↓) and $\oplus_1$ = maximum (or ↑)



Now, the maximum capacity from a to e is 1 no matter which path from a to e is selected. That is, we have a *tie*

# Example (cont’d)

Now, we can incorporate another criterion to break such a tie, let’s say that we pick the shortest distance, implying $\otimes_2$ = arithmetic addition (+) and $\oplus_2$ = minimum ($\downarrow$).

That is, now we have that:

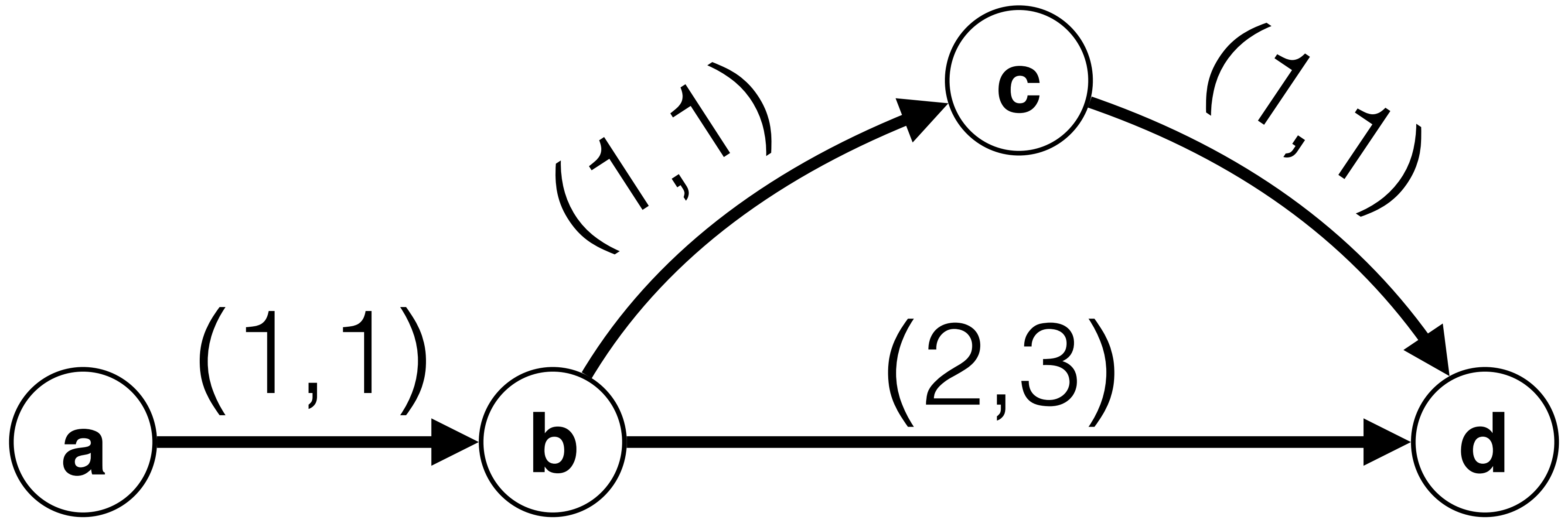
$\otimes = (\otimes 1, \otimes 2)$ and $\oplus = (\oplus 1, \oplus 2)$

in other words,

$\otimes = (\downarrow 1, +2)$ and $\oplus = (\uparrow 1, \downarrow 2)$

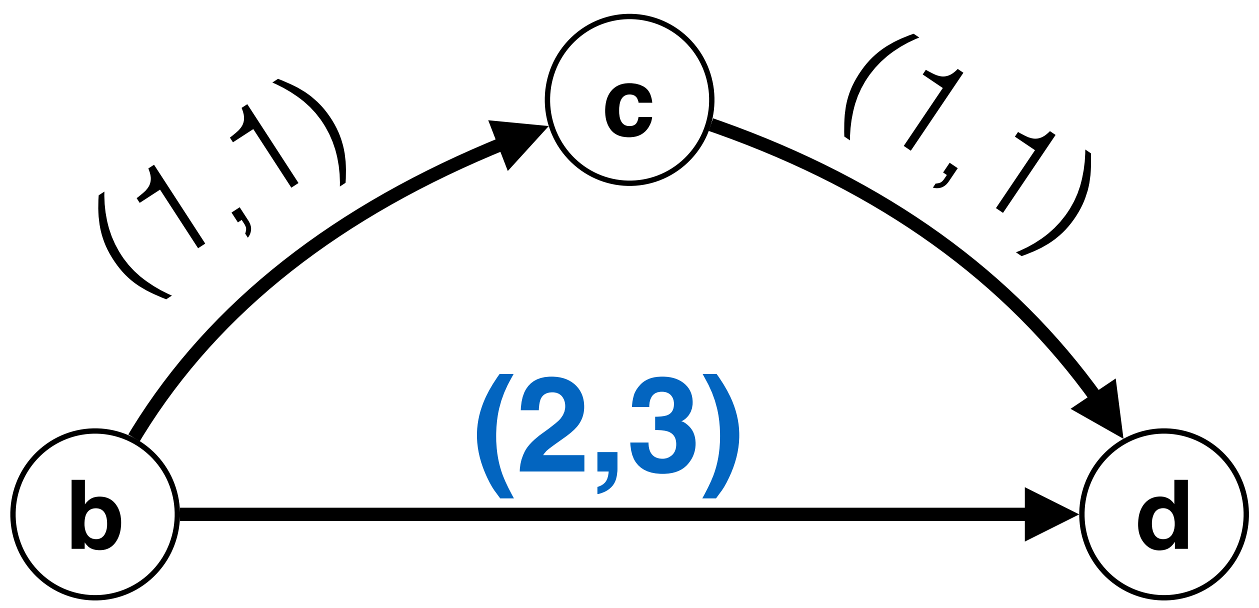
# Example (cont'd)

Also, we add the corresponding values for the new criterion as the second element in the pair-labels over the edges. That is, a pair $(v_j,v_k)$ defines $v_j$ as the valid elements for maximum capacity and $v_k$ as the valid elements for shortest distance.

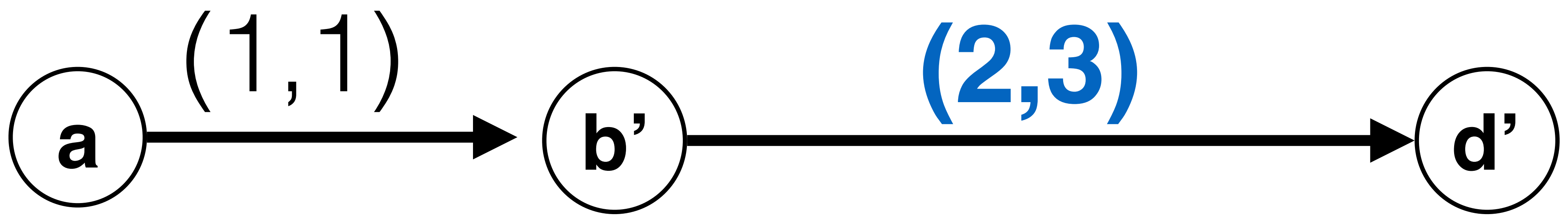


# The Problem

Computing the maximum capacity again, yields to a non optimal solution, that is,



the partial result, being (2,3) as (maximum capacity, shortest distance) leads to (1,4) instead of (1,3) in the final computation



# The Problem (cont'd)

Algebraically, we can represent the above as follows:

$$(1,1) \otimes [\ (1,1) \otimes (1,1) \oplus (2,3)\ ] \ = \ (1,1) \otimes (1,1) \otimes (1,1) \oplus (1,1) \otimes (2,3)$$

$$(1,1) \otimes [\ (1,2) \oplus (2,3)\ ] \ = \ (1,1) \otimes (1,2) \oplus (1,4)$$

$$(1,1) \otimes [(2,3)] \ = \ (1,3) \oplus (1,4)$$

$$(1,4) \ \neq \ (1,3)$$

# Fun Approach: List of Pairs

Preserving local optimal **and** “potential” optimal results along the computation in a list, allows to compute the global optimal. The conditions are:

storing the elements (pairs) preserving the following relation:

$$(x_1, y_1)\ R\ (x_2, y_2) \rightarrow x_1 > x_2 \wedge y_1 > y_2 \quad \vee$$

$$(x_2, y_2)\ R\ (x_1, y_1) \rightarrow x_2 > x_1 \wedge y_2 > y_1$$

otherwise simply store the greatest x-tuple

# List of Pairs applied

let us denoted the list notation with **{}**

$$(1,1) \otimes [\ (1,1) \otimes (1,1) \oplus (2,3)\ ] \rightarrow (1,1) \otimes [\ \boldsymbol{\{}(1,2)\boldsymbol{\}} \oplus (2,3)\ ]$$

$$\rightarrow (1,1) \otimes [\ \boldsymbol{\{}(2,3),(1,2)\boldsymbol{\}}\ ] \rightarrow (1,1) \otimes [\ \boldsymbol{\{}(2,3),(1,2)\boldsymbol{\}}\ ]$$

$$\rightarrow \boldsymbol{\{}\ (1,3)\ \boldsymbol{\}}$$

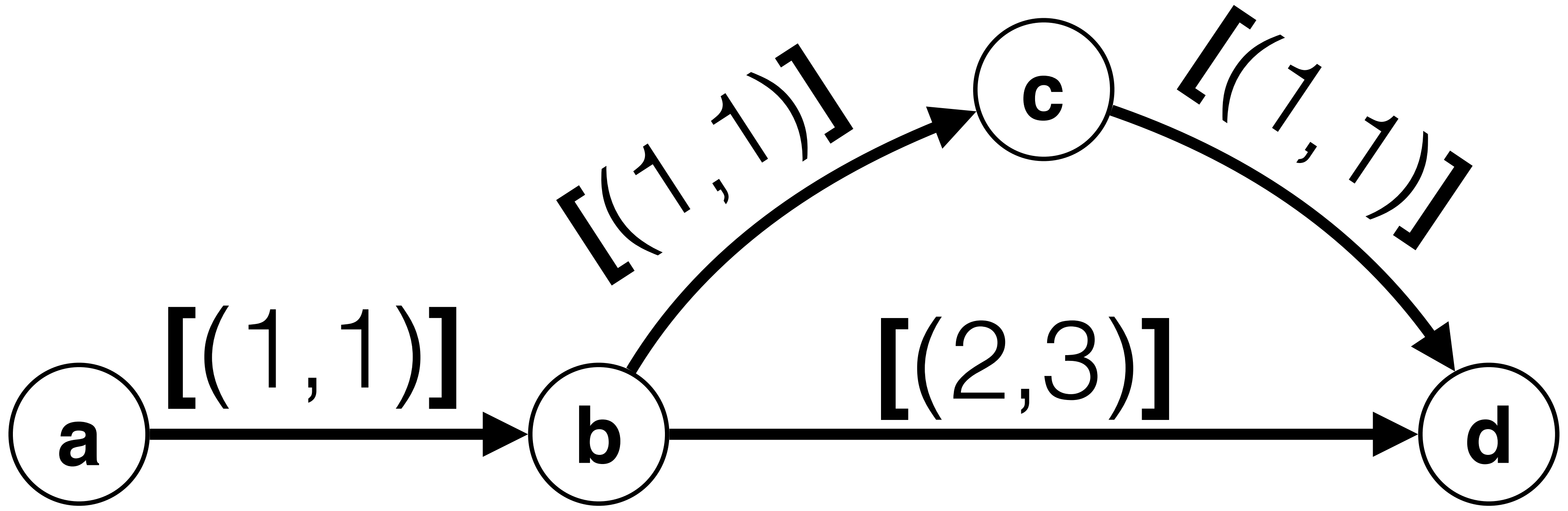
the global optimal as expected !!

# implementing MC-SD

We start with the types, calling join for ⊗ and choose for ⊕

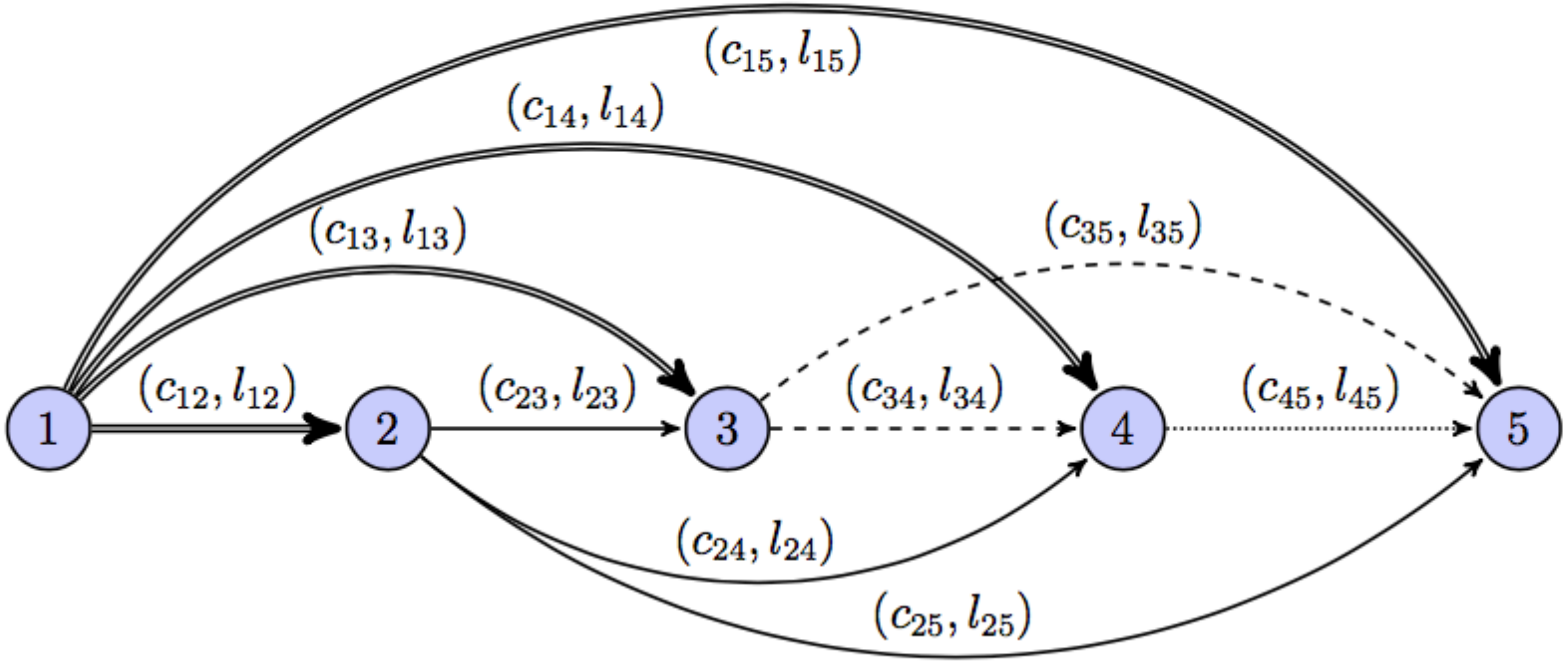
```
type BSP = [(Capacity,Distance)]
join     :: BSP → BSP → BSP
choose   :: BSP → BSP → BSP
```

since the functions should work either edges or paths, we turn every edge label into a singleton-path prior any computation

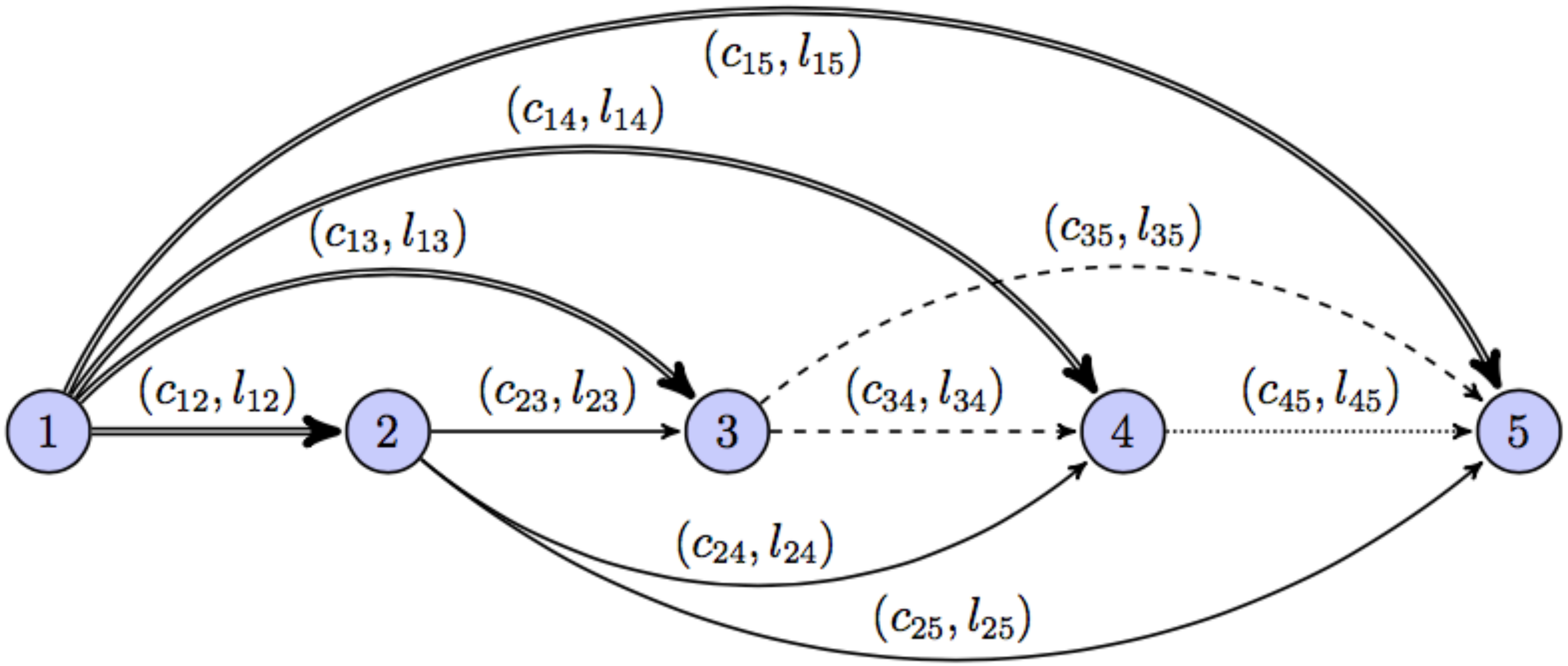


# Application: Single Source on DAGs

# Application: Single Source on DAGs

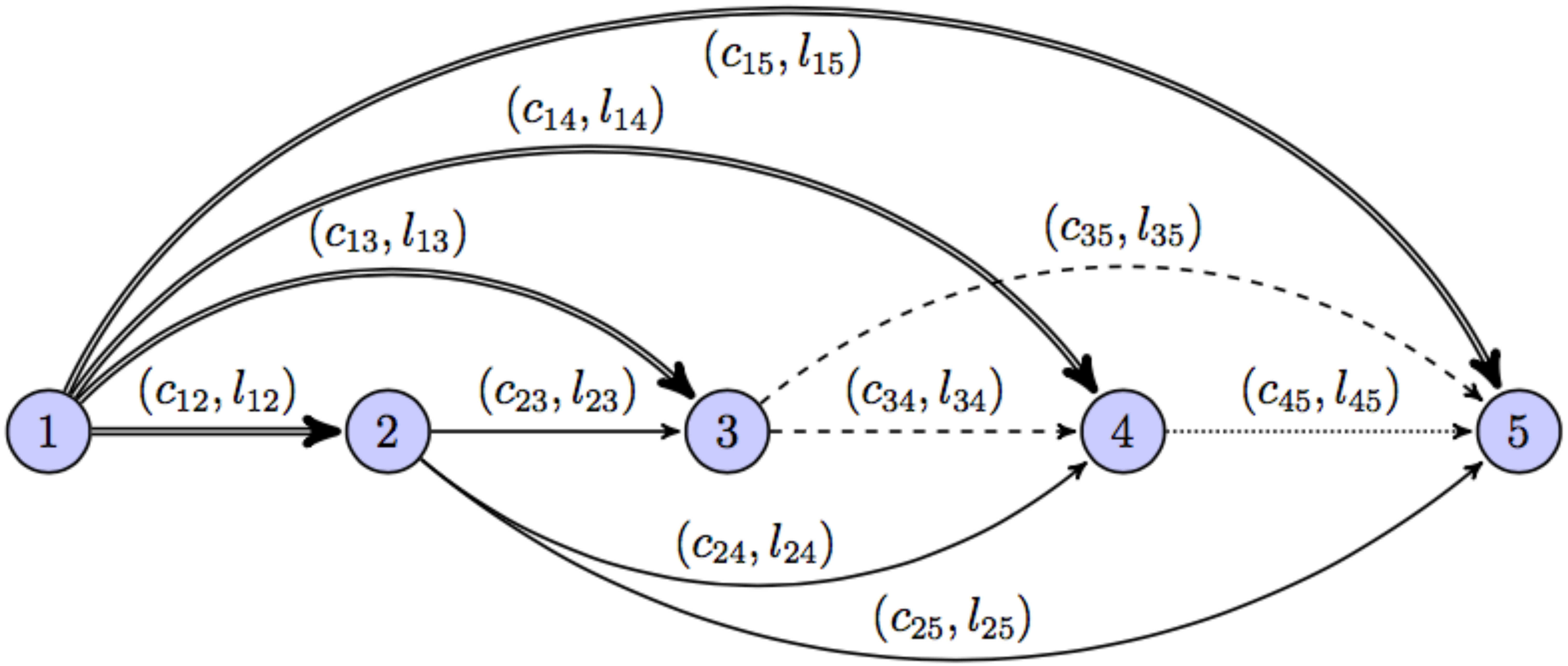


# Application: Single Source on DAGs



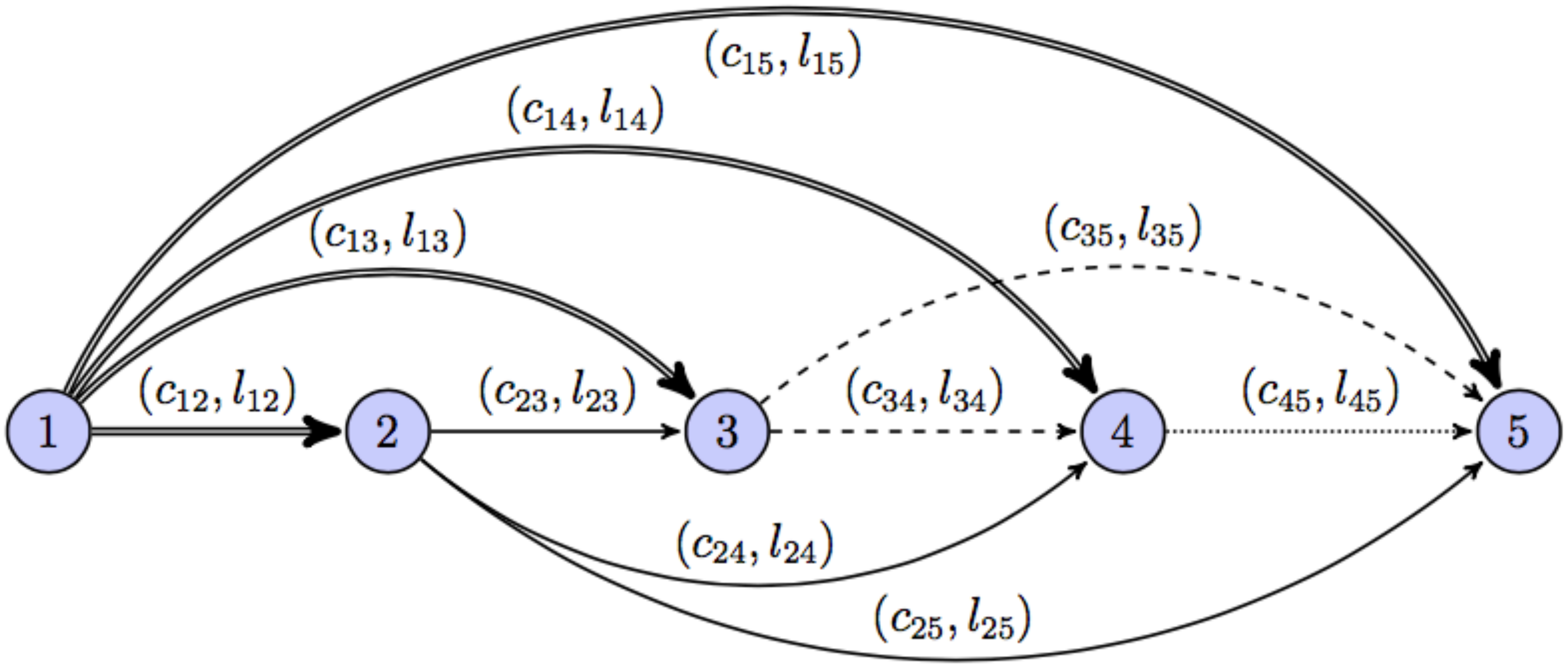
$(c_{45}, l_{45})$

# Application: Single Source on DAGs



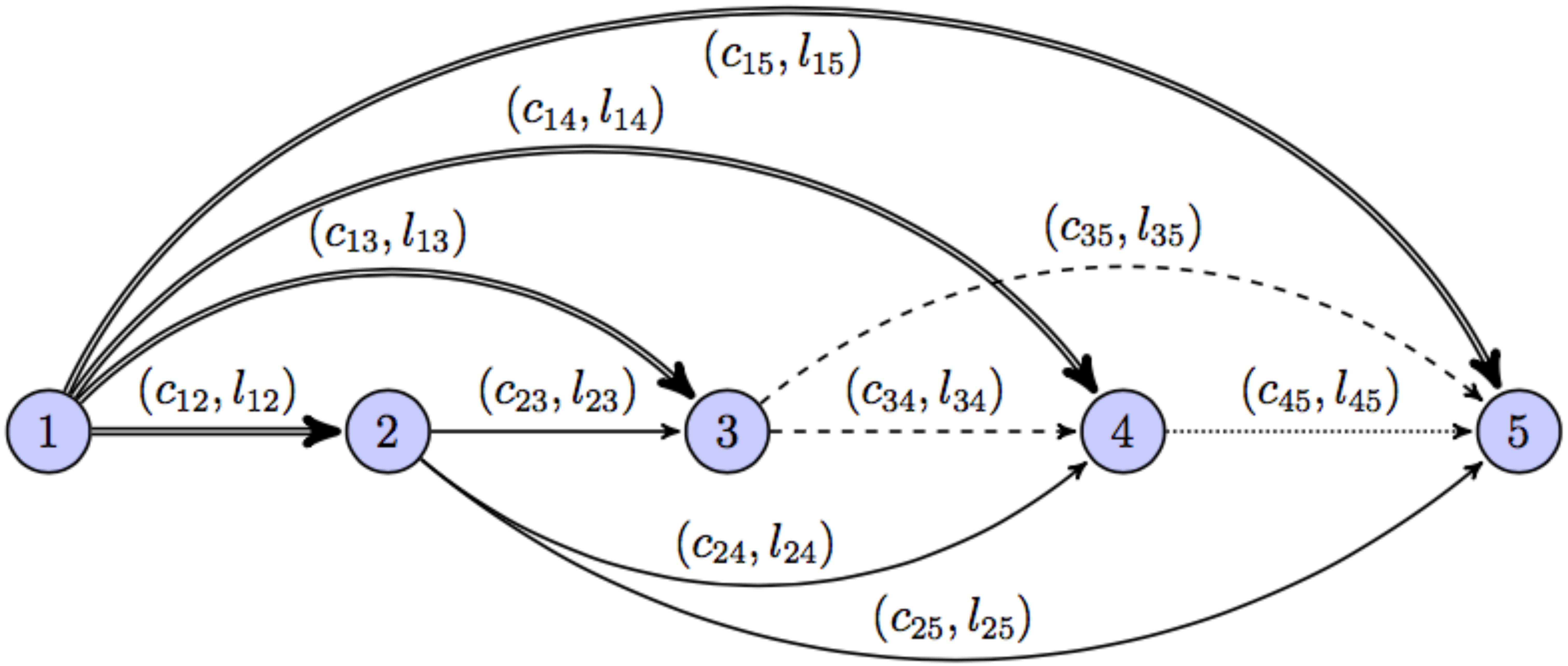
$(c_{45}, l_{45}) \otimes \mathit{sol}_5$

# Application: Single Source on DAGs



$(c_{45}, l_{45}) \otimes \mathit{sol}_5$ $\mathrm{sol}_4$

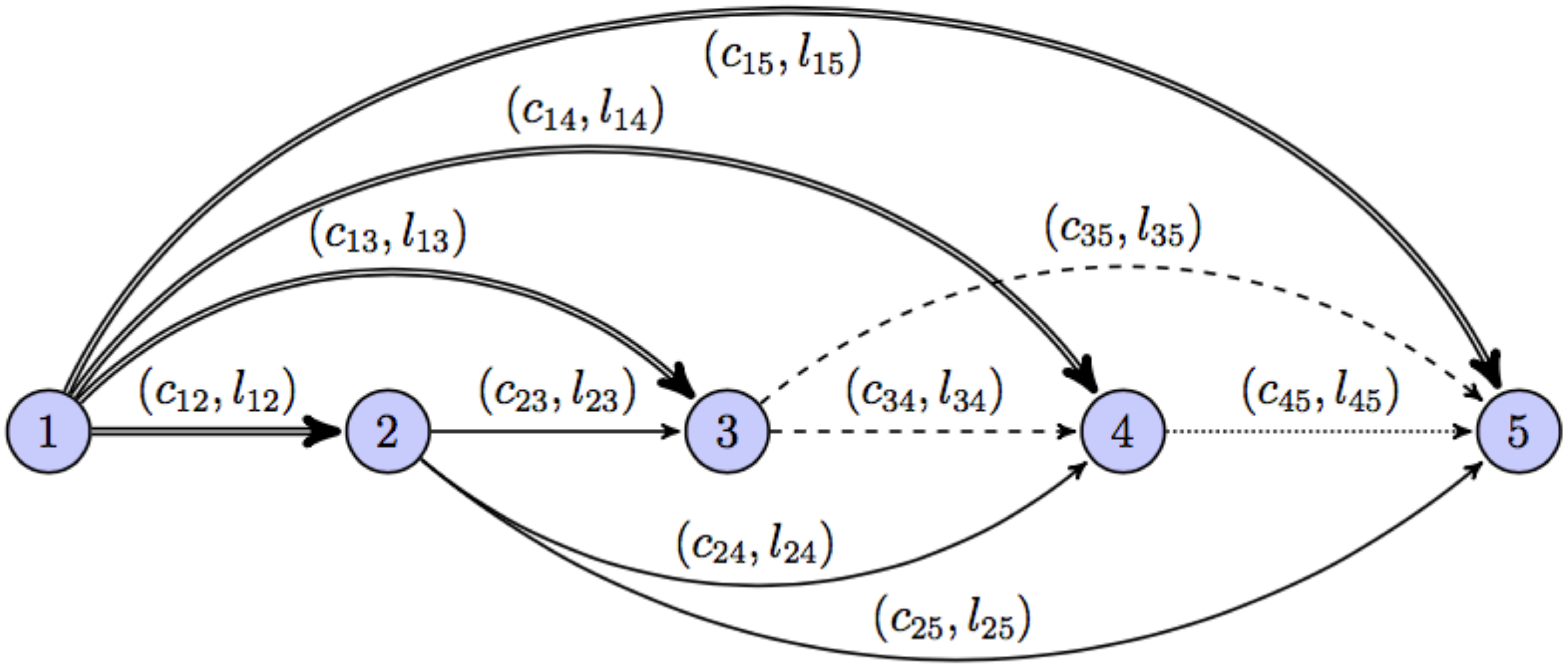
# Application: Single Source on DAGs



$(c_{45}, l_{45}) \otimes sol_5$ $sol_4$

$sol_5 = (\infty, 0)$

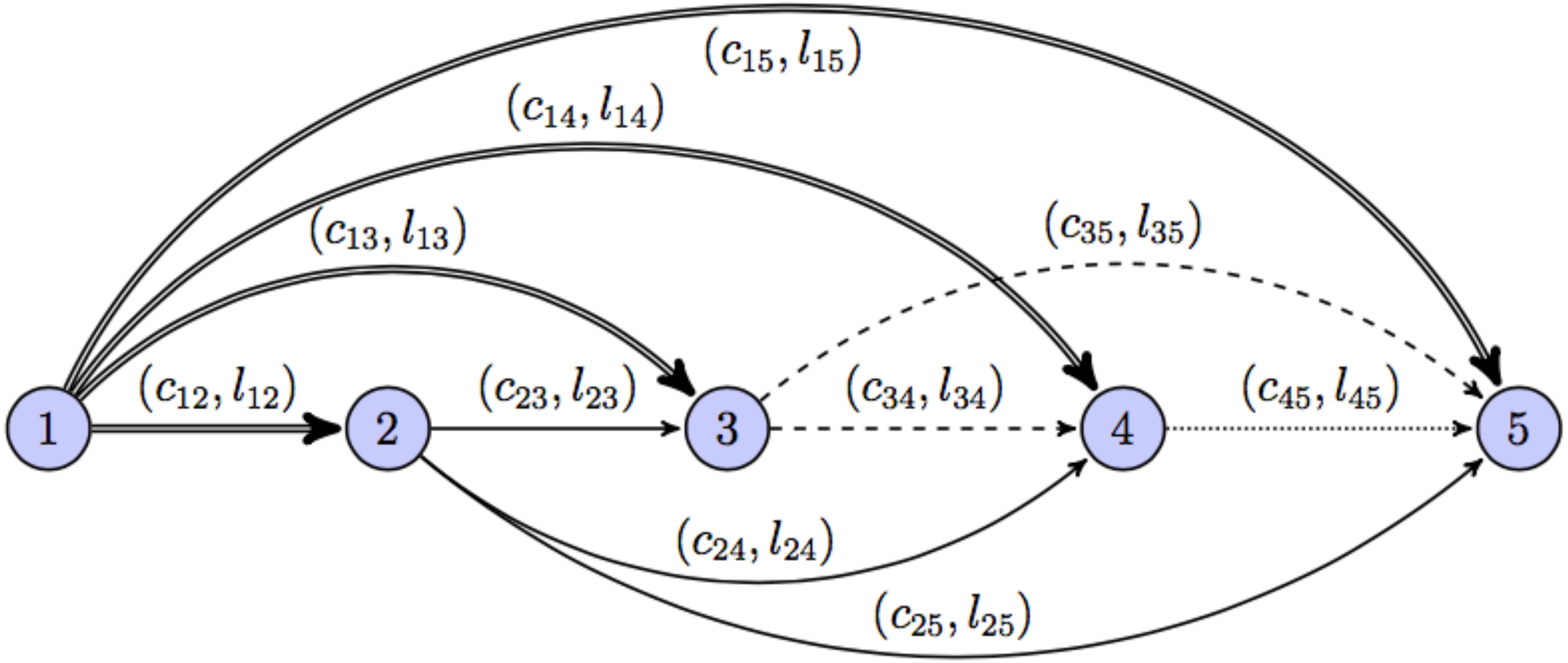
# Application: Single Source on DAGs



$(c_{45},l_{45}) \otimes sol_5$ $sol_4$

$sol_5 = (\infty,0)$ ?

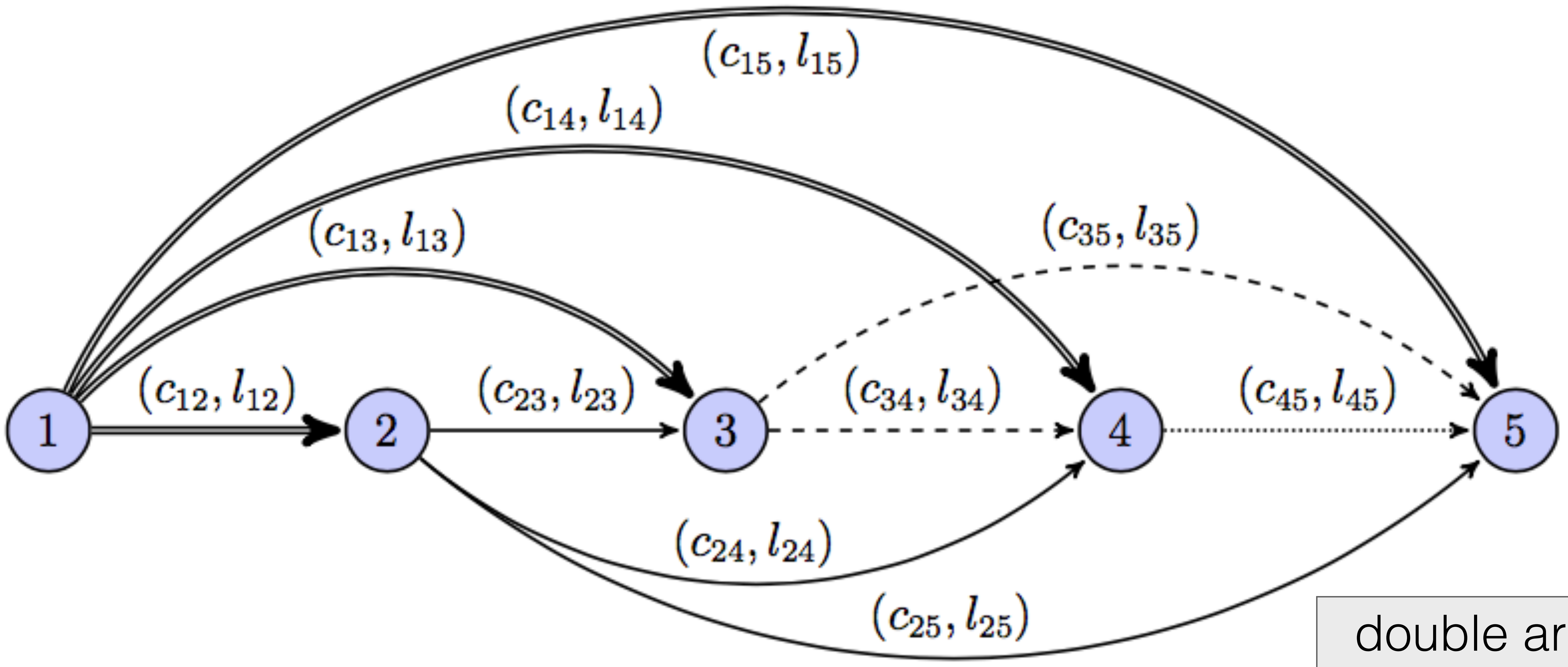
# Application: Single Source on DAGs



$(c_{45}, l_{45}) \otimes sol_5$ $sol_4$

$sol_5 = (\infty, 0)$ ? *unit for* $\otimes$

# Application: Single Source on DAGs

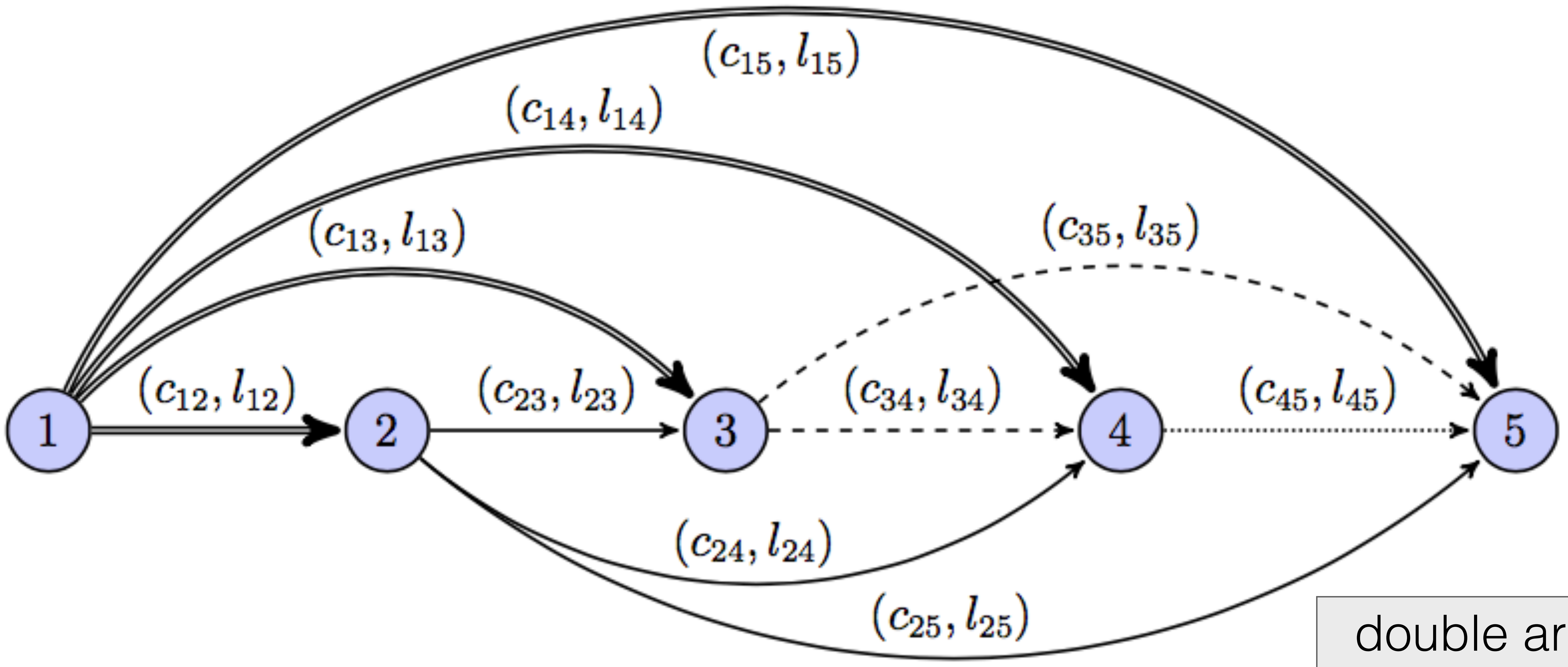


$(c_{45}, l_{45}) \otimes sol_5$ $sol_4$

$sol_5 = (\infty, 0)$ **?** *unit for* $\otimes$

| double arrow | single arrow ($sol_2$) | dashed arrow ($sol_3$) |
|---|---|---|
| $\oplus$ | $\oplus$ | $\oplus$ |
| $(c_{15}, l_{15}) \otimes sol_5$ | $(c_{25}, l_{25}) \otimes sol_5$ | $(c_{35}, l_{35}) \otimes sol_5$ |
| $(c_{14}, l_{14}) \otimes sol_4$ | $(c_{24}, l_{24}) \otimes sol_4$ | $(c_{34}, l_{34}) \otimes sol_4$ |
| $(c_{13}, l_{13}) \otimes sol_3$ | $(c_{23}, l_{23}) \otimes sol_3$ | |
| $(c_{12}, l_{12}) \otimes sol_2$ | | |

# Application: Single Source on DAGs



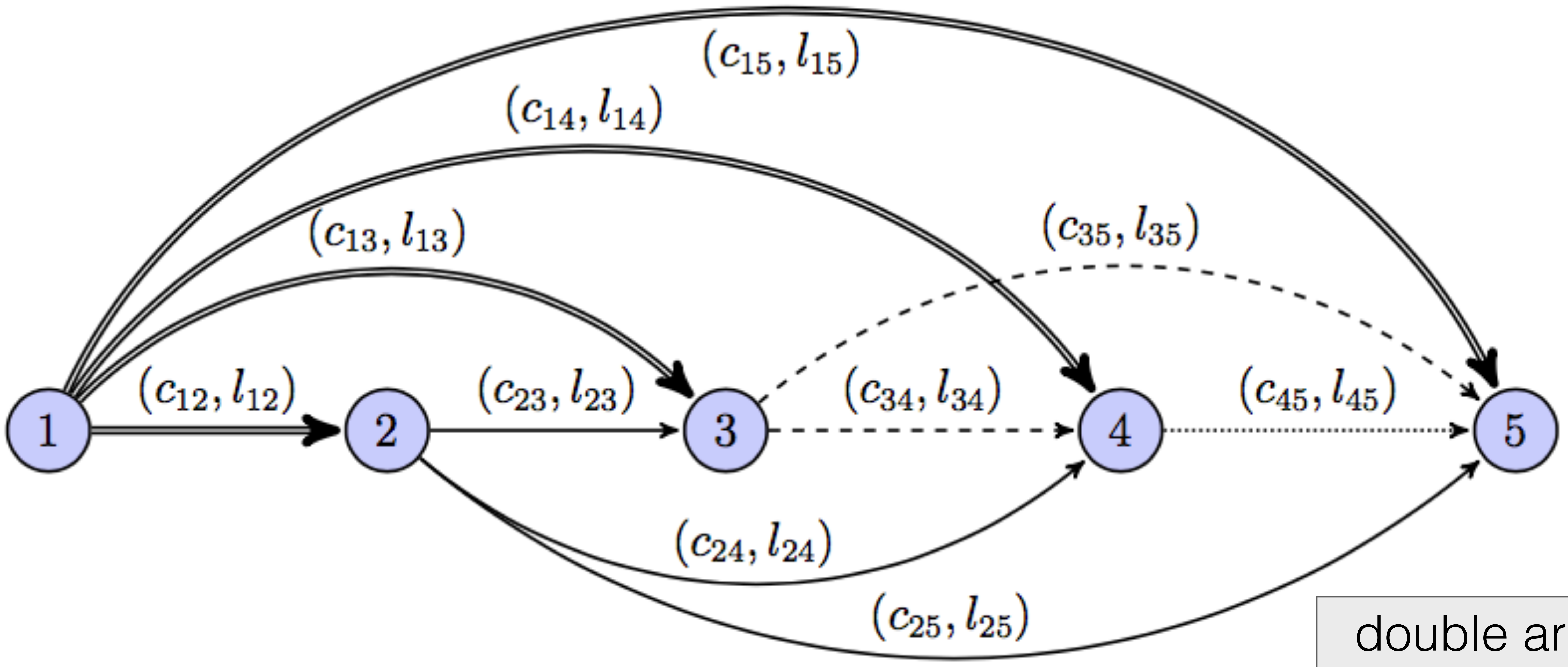
$(c_{45}, l_{45}) \otimes sol_5$ $sol_4$

$sol_5 = (\infty, 0)$ **?** *unit for* $\otimes$

| double arrow | single arrow ($sol_2$) | dashed arrow ($sol_3$) |
|---|---|---|
| $\oplus$ | $\oplus$ | $\oplus$ |
| $(c_{15}, l_{15}) \otimes sol_5$ | $(c_{25}, l_{25}) \otimes sol_5$ | $(c_{35}, l_{35}) \otimes sol_5$ |
| $(c_{14}, l_{14}) \otimes sol_4$ | $(c_{24}, l_{24}) \otimes sol_4$ | $(c_{34}, l_{34}) \otimes sol_4$ |
| $(c_{13}, l_{13}) \otimes sol_3$ | $(c_{23}, l_{23}) \otimes sol_3$ | |
| $(c_{12}, l_{12}) \otimes sol_2$ | | |

sounds familiar **?**

# Application: Single Source on DAGs



$(c_{45}, l_{45}) \otimes sol_5$ $sol_4$

$sol_5 = (\infty, 0)$ **?** *unit for* $\otimes$

| double arrow | single arrow ($sol_2$) | dashed arrow ($sol_3$) |
|---|---|---|
| $\oplus$ | $\oplus$ | $\oplus$ |
| $(c_{15}, l_{15}) \otimes sol_5$ | $(c_{25}, l_{25}) \otimes sol_5$ | $(c_{35}, l_{35}) \otimes sol_5$ |
| $(c_{14}, l_{14}) \otimes sol_4$ | $(c_{24}, l_{24}) \otimes sol_4$ | $(c_{34}, l_{34}) \otimes sol_4$ |
| $(c_{13}, l_{13}) \otimes sol_3$ | $(c_{23}, l_{23}) \otimes sol_3$ | |
| $(c_{12}, l_{12}) \otimes sol_2$ | | |

**That's right: Dynamic Programming !!!**

sounds familiar **?**

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

**:: [(Capacity, Distance)]**

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

**:: [(Capacity, Distance)]**

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

**:: [(Capacity, Distance)]**

**unit for ⊕, that is [(0,∞)]**

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

**:: [(Capacity, Distance)]**

**unit for ⊕, that is [(0,∞)]**

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

**:: [(Capacity, Distance)]**

**unit for ⊕, that is [(0,∞)]**

**⊕, that is ($\oplus_1$,$\oplus_2$)**

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

:: [(Capacity, Distance)]

unit for $\oplus$, that is $[(0,\infty)]$

$\oplus$, that is $(\oplus_1,\oplus_2)$

# Application: All Pairs (square matrix)

As an example of full (dense) connected graph, including cycles, we can recurre to the Floyd-Roy-Warshall algorithm (all-pairs shortest path)

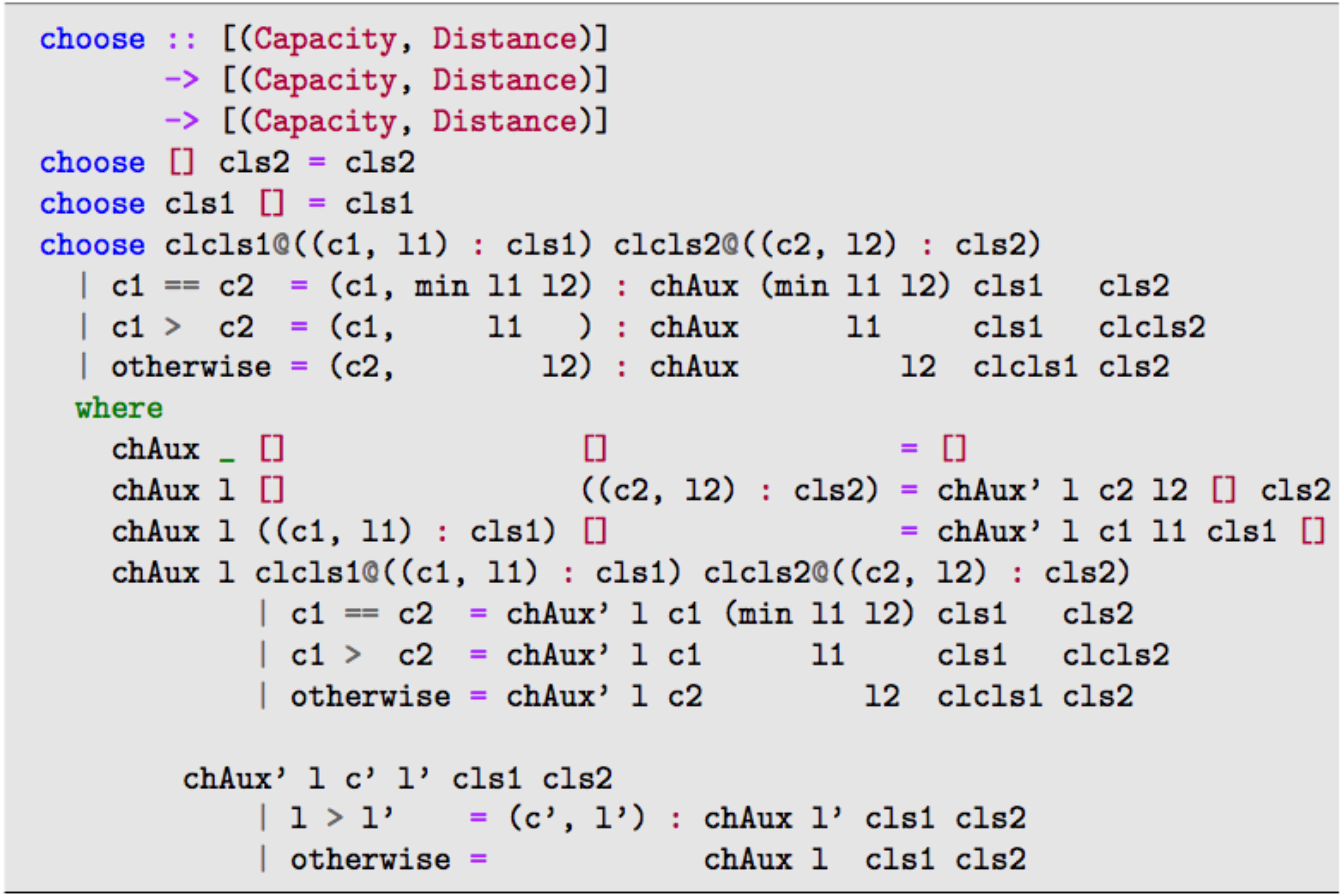
```
for each (i,j) in N x N:
 if (i,j) is in E then:
   d(i,j) := w(i,j)
 else:
   d(i,j) := 0

for each k in N:
 for each i in N:
  for each j in N:
   d(i,j) := min{d(i,j),d(i,k) + d(k,j)}
```

:: [(Capacity, Distance)]

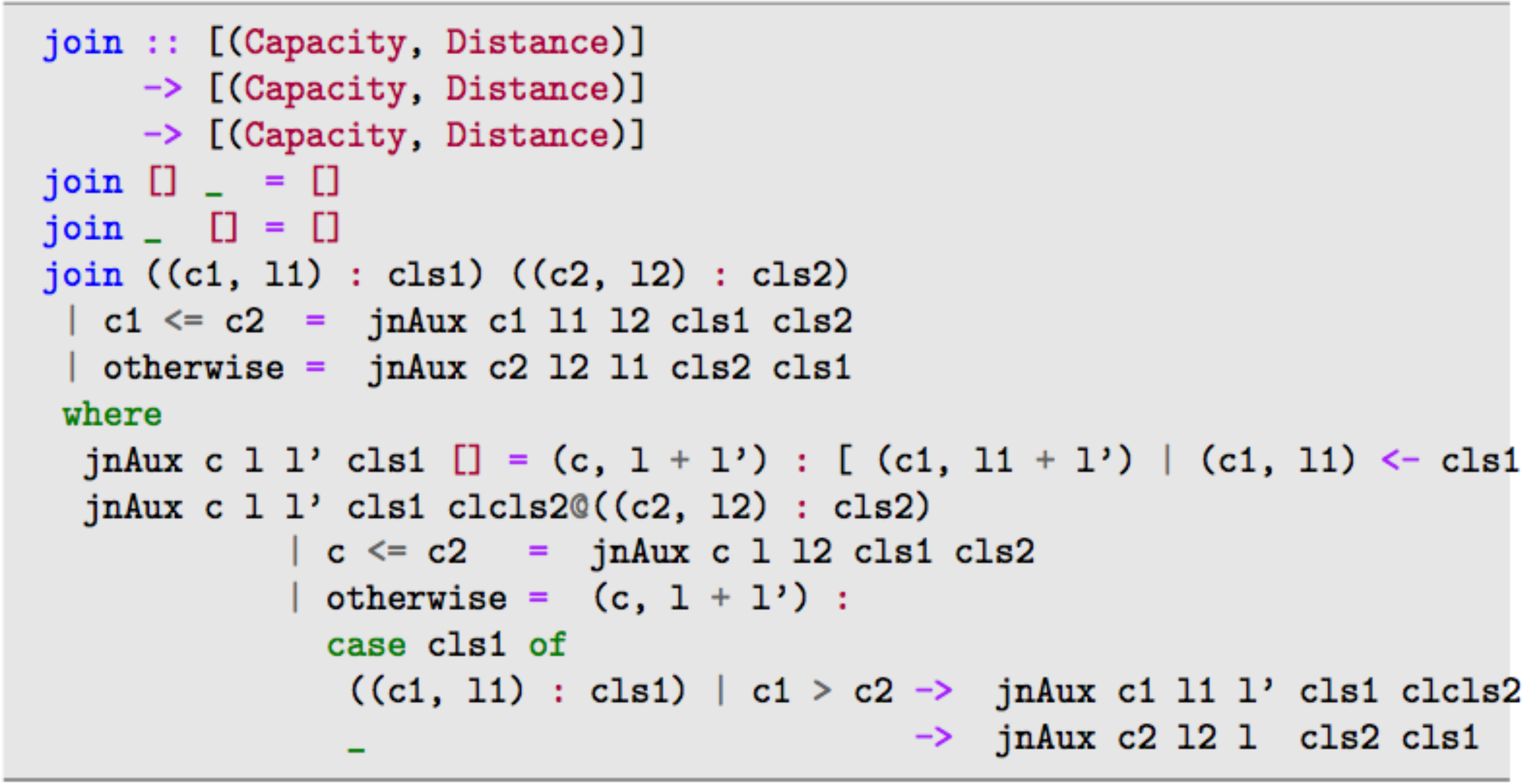
unit for ⊕, that is [(0,∞)]

⊕, that is ($\oplus_1$,$\oplus_2$)

⊗, that is ($\otimes_1$,$\otimes_2$)



```haskell
choose :: [(Capacity, Distance)]
       -> [(Capacity, Distance)]
       -> [(Capacity, Distance)]
choose [] cls2 = cls2
choose cls1 [] = cls1
choose clcls1@((c1, l1) : cls1) clcls2@((c2, l2) : cls2)
  | c1 == c2  = (c1, min l1 l2) : chAux (min l1 l2) cls1   cls2
  | c1 >  c2  = (c1,     l1    ) : chAux      l1      cls1   clcls2
  | otherwise = (c2,         l2) : chAux         l2  clcls1 cls2
  where
    chAux _ []                 []                = []
    chAux l []                 ((c2, l2) : cls2) = chAux' l c2 l2 [] cls2
    chAux l ((c1, l1) : cls1) []                 = chAux' l c1 l1 cls1 []
    chAux l clcls1@((c1, l1) : cls1) clcls2@((c2, l2) : cls2)
            | c1 == c2  = chAux' l c1 (min l1 l2) cls1   cls2
            | c1 >  c2  = chAux' l c1      l1      cls1   clcls2
            | otherwise = chAux' l c2         l2  clcls1 cls2

        chAux' l c' l' cls1 cls2
            | l > l'     = (c', l') : chAux l' cls1 cls2
            | otherwise =              chAux l  cls1 cls2
```



```haskell
join :: [(Capacity, Distance)]
     -> [(Capacity, Distance)]
     -> [(Capacity, Distance)]
join [] _  = []
join _  [] = []
join ((c1, l1) : cls1) ((c2, l2) : cls2)
 | c1 <= c2  =  jnAux c1 l1 l2 cls1 cls2
 | otherwise =  jnAux c2 l2 l1 cls2 cls1
 where
  jnAux c l l’ cls1 [] = (c, l + l’) : [ (c1, l1 + l’) | (c1, l1) <- cls1 ]
  jnAux c l l’ cls1 clcls2@((c2, l2) : cls2)
            | c <= c2    =  jnAux c l l2 cls1 cls2
            | otherwise =  (c, l + l’) :
              case cls1 of
               ((c1, l1) : cls1) | c1 > c2 ->  jnAux c1 l1 l’ cls1 clcls2
               _                           ->  jnAux c2 l2 l  cls2 cls1
```

# What is next?

- Include the analysis on Knapsack problem, where the ⊗ takes the weight of the bag as an argument
- Include an analysis on the lazy and strict evaluations
- Monadic implementation (Floyd-Roy-Warshall algorithm)

http://staffwww.dcs.shef.ac.uk/people/J.Saenz_Carrasco/