Tecnológico de Monterrey

Connectionist and Evolutionary Systems: ACO

Final Project: *UCSP*

Dmitry K.

May 9, 2016

Abstract

The *University Classes Schedule Problem* (**UCSP**) consists in finding all the *required disciplines* for each *group* at some academic period. It doesn't really matter whether the *disciplines* are chosen by the students or assigned by the institution. Anyway, the **primary task** for the "ants" is to encounter **valid** configurations of *classes*, such that provide exactly the *required time* of each *required discipline* for each *group*. The **secondary task** is to encounter the solution, that provides the best *satisfaction* by the represented persons and the institution.

1 Problem

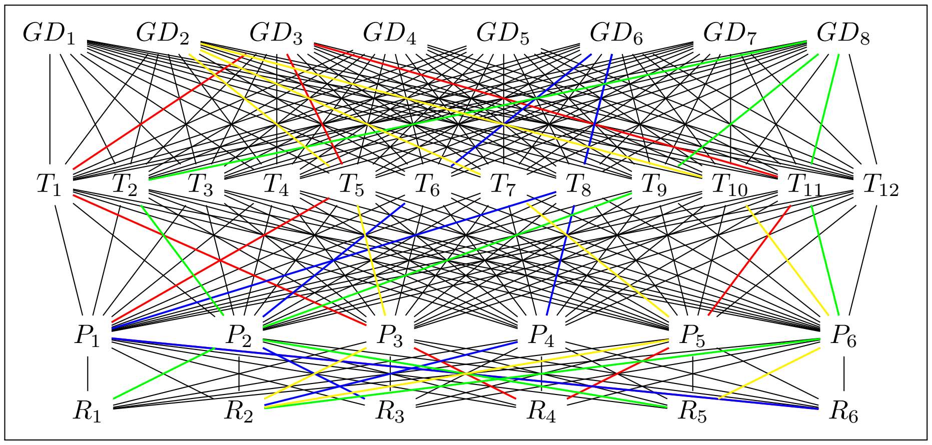


Figure 1: Problem graph schematic, representing **G**roups, **D**isciplines, **T**ime/day, **P**rofessors, Class**R**ooms.

1.1 Classes

A *class* is an event, that links together the following types of entities, denoted as *roles*:

1. group-discipline pairs
2. day/time
3. professors
4. classrooms

Each of the roles must have a finite and non-empty domain, therefore ensuring finite number of unique permutations.

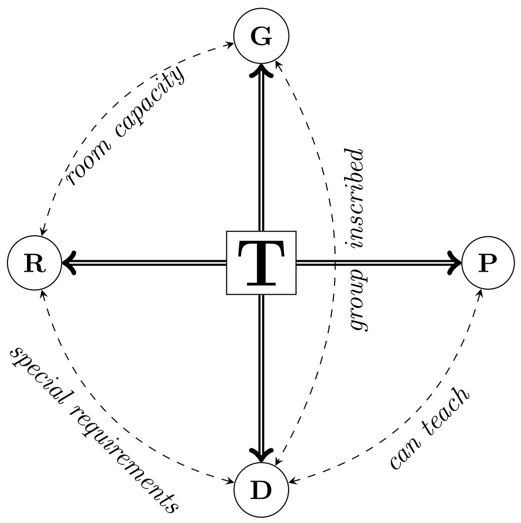


Figure 2: *Class* structure.

```
  -- Used as kind (see data type promotion)
data Role = Groups | DayTime | Professors | Classrooms deriving Typeable
  -- 'Role' kind container
data Role' (r :: Role) = Role' deriving Typeable
```

1.2 Graph Nodes

The problem graph nodes are different *permutations* of *role domains.* They are grouped into *layers*, depending on the corresponding *role*.

The nodes at some layer have exactly the same underlying size and it's the power of it's domain set.

```
type family RoleValue (r :: Role) :: *
class HasDomain a v | a → v
  where domain      :: a → Set v
        domainPower :: a → Int
newtype Node (r :: Role) = Node [RoleValue r]
mkNodes :: HasDomain (Role' r) (RoleValue r) ⇒
           Role' r → [Node r]
mkNodes = map Node ∘ permutations ∘ Set.toList ∘ domain
```

1.2.1 Timetable

A *timetable* holds schedule for one week, that repeats throughout the academic period. The *timetable* is actually a table: the columns represent days of week;

the rows — discrete time intervals. Actual timetable structure may vary, as can be seen in figure 3.

	Mon	Tue	Wed	Thu	Fri	Sat
08:30 – 09:00						
09:00 – 09:30						
09:30 – 10:00						
10:00 – 10:30						
10:30 – 11:00						
11:00 – 11:30						
11:30 – 12:00						
⋮ ⋮						

(a) Timetable without recesses.

	Mon	Tue	Wed	Thu	Fri	Sat
08:30 – 09:10						
09:15 – 09:55						
10:05 – 10:45						
10:50 – 11:30						
11:40 – 12:20						
12:25 – 13:05						
13:15 – 13:55						
⋮ ⋮						

(b) Timetable with recesses.

Figure 3: Possible *timetable* structures.

```
class (Eq t, Ord t, Enum t, Bounded t) ⇒
  DiscreteTime t where timeQuantum :: t   → Int
                       toMinutes   :: t   → Int
                       fromMinutes :: Int → Maybe t

class (DiscreteTime t, Enum d, Bounded d) ⇒
  Timetable tt t d ev | tt → t
                      , tt → d
                      , tt → ev
  where listEvents :: tt                → [((d, t), ev)]
        newTTable  :: [((d, t), ev)] → tt
        eventsOn   :: tt → d           → [(t, ev)]
        eventsAt   :: tt → t           → [(d, ev)]
        eventAt    :: tt → d → t      → Maybe ev
```

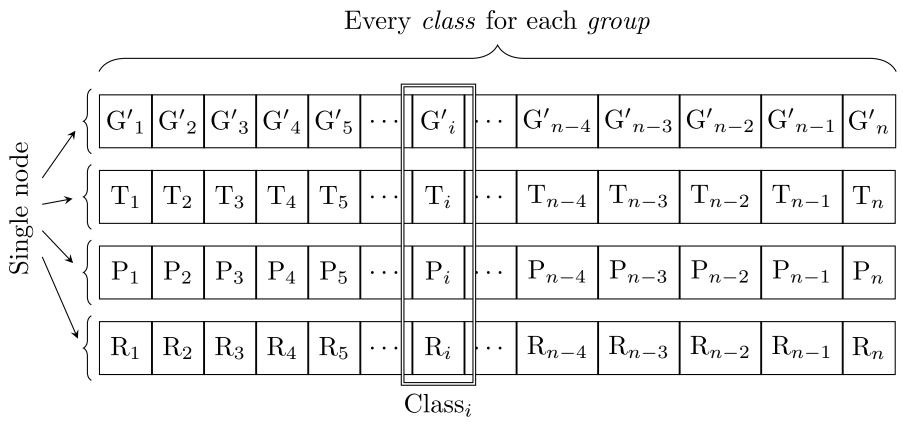
1.3 Graph Edges

The edges are possible routes, that can be taken by an "ant". They connect nodes, belonging to *different layers*.

$$\begin{aligned}
&\forall a \in \text{Layer}_A \\
&\forall b \in \text{Layer}_B \\
&\quad \text{if Layer}_A \text{ and Layer}_B \text{ are neighbors} \\
&\qquad \exists \text{ an edge between } a \text{ and } b.
\end{aligned}$$

A selection of some sub-route, connecting some nodes A_i and B_j (from some layers A and B) means that the ant "proposes" a (partial) solution, that is described by the nodes' underlying values. The "ant" agent must be capable of selecting exactly one node of each role. The selection order doesn't matter.

A complete route (through all the layers) describes a *solution candidate*: some schedule, that holds a list of *classes*.



Every *class* for each *group*
Single node
G'_1 G'_2 G'_3 G'_4 G'_5 $\cdots$ G'_i $\cdots$ G'_{n-4} G'_{n-3} G'_{n-2} G'_{n-1} G'_n
T_1 T_2 T_3 T_4 T_5 $\cdots$ T_i $\cdots$ T_{n-4} T_{n-3} T_{n-2} T_{n-1} T_n
P_1 P_2 P_3 P_4 P_5 $\cdots$ P_i $\cdots$ P_{n-4} P_{n-3} P_{n-2} P_{n-1} P_n
R_1 R_2 R_3 R_4 R_5 $\cdots$ R_i $\cdots$ R_{n-4} R_{n-3} R_{n-2} R_{n-1} R_n
Class_i

Figure 4: *Route* decomposition.

2 Formalization

Let's denote

N_G — number of groups;

N_P — number of professors;

N_R — number of classrooms;

N_D — number of disciplines;

N_T — number of *time periods* per week:
number of *time periods* per day × number of *days*;

N_d^g — number of *time periods* of discipline d, assigned for group g;

$G = \{g_i\}_{i=1}^{N_G}$ — set of groups;

$D = \{d_i\}_{i=1}^{N_D}$ — set of disciplines;

$P = \{p_i\}_{i=1}^{N_P}$ — set of professors;

$R = \{r_i\}_{i=1}^{N_R}$ — set of classrooms;

$D_g = \{d \mid N_d^g \neq 0\}_{d \in D}$ — set of disciplines, assigned to group g;

$N_\Sigma = \sum\limits_{g \in G} \sum\limits_{d \in D_g} N_d^g$ — total number of classes time periods per week.

2.1 Problem Dimensions

2.1.1 Groups and Disciplines

Let G' be a list of pairs $\langle \text{group}, \text{discipline} \rangle$ of length N_Σ, such that $\forall \langle g, d \rangle \in G' \implies \text{count}_{G'}(\langle g, d \rangle) = N_d^g$. There are $N_\Sigma!$ unique permutations.

2.1.2 Professors and Classrooms

With no optimization applied, exists $\binom{N_\Sigma + N - 1}{N_\Sigma - 1}$ (combinations with repetitions), where $N = N_P$ or N_R.

Some invalid instances can be discarded, such that, for example, don't have enough professors capable of teaching some discipline; or classrooms configurations that won't fit all the students etc.

2.1.3 Day and Time

In general case, any day and time may be assigned for any class period, including repetitions, that yields $\binom{N_\Sigma + N_T - 1}{N_\Sigma - 1}$ possible combinations.

This number may be diminished by

- joining class periods;
- requiring a minimum entropy.

Total combinations (worst case):

$$\binom{N_\Sigma + N_P - 1}{N_\Sigma - 1} \binom{N_\Sigma + N_R - 1}{N_\Sigma - 1} \binom{N_\Sigma + N_T - 1}{N_\Sigma - 1} N_\Sigma! \tag{1}$$

2.2 Assessing Candidates

$$\eta = \eta(\{r_i\}_{i=1}^{n-1}, r_n) = \begin{cases} 0 & \text{if any restriction is broken} \\ \text{pref}(\{r_i\}_{i=1}^{n}) & \text{otherwise} \end{cases} \tag{2}$$

where r_i is some some sub-route.

2.2.1 Restrictions

There are two kinds of restrictions: over *time* and over *capabilities.*

Time restriction require the schedule to be *time consistent*: no group, professor and classroom can have two different classes, assigned at the same day/time. The capabilities represent:

Group:	Disciplines needed (searched).
Professors:	Known disciplines (that can be taught).
Classrooms:	Special requirements (labs etc.); students capacity.

Note: group capabilities are incorporated into nodes generation.

2.2.2 Preferences

Preferences create an order over *valid candidates*, that permits the algorithm to optimize them. The preferences might vary for each entity (group, professor, classroom), but they all must have a form of function:

$$\text{pref}'[E] : \langle \text{discipline}, \text{day/time} \rangle \mapsto [0, 1]$$

The preference value for a *complete route*:

$$\text{pref}(r) = \frac{\text{pref}'[G](r) + \text{pref}'[P](r) + \text{pref}'[R](r)}{3}$$

3 Implementation

3.1 Entities

Here follows definition of the input data, as stated in Section 2.

```
data Discipline = Dicipline { disciplineId   :: String
                            , disciplineTime :: Int
                            , disciplineReqs :: Set Requirement
                            }
newtype Requirement = Requirement String
  deriving (Show, Eq, Ord)
instance Show Discipline where show    = disciplineId
instance Eq   Discipline where (≡)     = (≡)     ‘on‘ disciplineId
instance Ord  Discipline where compare = compare ‘on‘ disciplineId
```

```
data Group = Group { groupId          :: String
                   , groupSize        :: Int
                   , groupDisciplines :: Set Discipline
                   }
instance Show Group where show    = groupId
instance Eq   Group where (≡)     = (≡)     ‘on‘ groupId
instance Ord  Group where compare = compare ‘on‘ groupId
```

```
data Professor = Professor { professorId :: String
                           , canTeach    :: Set Discipline
                           }
instance Show Professor where show    = professorId
instance Eq   Professor where (≡)     = (≡)     ‘on‘ professorId
instance Ord  Professor where compare = compare ‘on‘ professorId
```

```
data Classroom = Classroom { roomId        :: String
                           , roomCapacity  :: Int
                           , roomEquipment :: Set Requirement
                           }
instance Show Classroom where show    = roomId
instance Eq   Classroom where (≡)     = (≡)     ‘on‘ roomId
instance Ord  Classroom where compare = compare ‘on‘ roomId
```

3.1.1 Timetable

Timetable is defined over *Mon–Sat*, from *8:00* till *22:00* with *30 minutes* discretization.

```
newtype Time = Time Int
  deriving (Eq, Ord)
timeQ   = 30
timeMin = 60 ∗ 8
timeMax = 60 ∗ 22
timeDMin = 0
timeDMax = (timeMax − timeMin) ‘quot‘ timeQ
instance Enum Time where
```

```
  fromEnum (Time t) = t
  toEnum i = if   i ⩾ timeDMin
     ∧            i ⩽ timeDMax
            then Time i
            else error $ "wrong discrete time: " ++ show i
instance Bounded Time where minBound = Time timeDMin
                            maxBound = Time timeDMax
instance DiscreteTime Time where
  toMinutes (Time t) = timeMin + timeQ ∗ t
  timeQuantum _  = 30
  fromMinutes  m = if m ⩾ timeMin
                    ∧ m ⩽ timeMax
                    ∧ m `rem` timeQ ≡ 0
                   then Just ∘ Time $ (m − timeMin) `quot` timeQ
                   else Nothing
  -- — — — — — — — — — — — — — — — — — —
  -- redefined 'System.Time.Day' — no 'Sunday'
data Day = Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday
  deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)
  -- — — — — — — — — — — — — — — — — — —
type DaySchedule = Map Time Class
newtype WeekSchedule = WeekSchedule (Map Day DaySchedule)
groupWith′ :: (Ord k) ⇒ (a → k) → (a → v) → [a] → Map k [v]
groupWith′ f g es =
  let groupIn []       = id
      groupIn (x : xs) = Map.insertWith (++) (f x) [g x]
  in es `groupIn` Map.empty
instance Timetable WeekSchedule Time Day Class where
  listEvents (WeekSchedule ws) = do
    (day, classes)    ← Map.assocs ws
    (time, class′)    ← Map.assocs classes
    return ((day, time), class′)
  newTTable = WeekSchedule ∘ Map.map Map.fromList
                           ∘ groupWith′ (fst ∘ fst)
                                        (first snd)
```

3.1.2 Classes

A *Class* entity links a *discipline*, *group*, *professor*, *classroom* and some *day-time*.

```
data Class = Class { classDiscipline :: Discipline
                   , classGroup      :: Group
```

```
                          , classProfessor :: Professor
                          , classRoom      :: Classroom
                          , classDay       :: Day
                          , classBegins    :: Time
                          }
 -- — — — — — — — — — — — — — — — — — —
 -- buildclasses :: Node DayTime
 -- -¿ Node Groups
 -- -¿ Node Professors
 -- -¿ Node Classrooms
 -- -¿ [Class]
 -- buildClasses (Node dts) (Node grs) (Node prs) (Node crs) =
 -- let l = length dts
 -- ls = [length grs, length prs, length crs]
 -- in if (l /= ) ‘any‘ ls
 -- then error ”wrongdimensions : ” + +show(l : ls)
 -- else do ((d,t), (gr,di), pr, cr) ¡- zip4 dts grs prs crs
 -- return Class  classDiscipline = di
 -- , classGroup = gr
 -- , classProfessor = pr
 -- , classRoom = cr
 -- , classDay = d
 -- , classBegins = t
 --
 -- — — — — — — — — — — — — — — — — — —
type instance RoleValue DayTime    = (Day, Time)
type instance RoleValue Groups     = (Group, Discipline)
type instance RoleValue Professors = Professor
type instance RoleValue Classrooms = Classroom
 -- — — — — — — — — — — — — — — — — — -
class RoleExtra (r :: Role) where
  roleIx    :: Role′ r → Int
  mbRole    :: Role′ r → PartClass → Maybe (RoleValue r)
  classRole :: Role′ r → Class → RoleValue r
instance RoleExtra Groups     where roleIx _      = 0
                                    mbRole _ r    = (, ) <$>
                                                    mbGroup r <∗>
                                                    mbDiscipline r
                                    classRole _   = classGroup &&&
                                                    classDiscipline
instance RoleExtra DayTime    where roleIx _      = 1
                                    mbRole _      = mbDayTime
                                    classRole _   = classDay &&&
                                                    classBegins
instance RoleExtra Professors  where roleIx _     = 2
                                     mbRole _     = mbProfessor
                                     classRole _  = classProfessor
instance RoleExtra Classrooms where roleIx _      = 3
                                     mbRole _     = mbRoom
                                     classRole _  = classRoom
```

Meanwhile a **PartClass** stands for a partially defined *Class* and a *Route* — for a sequence of *PartClasses*.

```
data PartClass = PartClass { mbDiscipline :: Maybe Discipline
                           , mbGroup      :: Maybe Group
                           , mbProfessor  :: Maybe Professor
                           , mbRoom       :: Maybe Classroom
                           , mbDayTime    :: Maybe (Day, Time)
                           }
toFullClass r = do  di     ← mbDiscipline r
                    g      ← mbGroup r
                    p      ← mbProfessor r
                    cr     ← mbRoom r
                    (d, t) ← mbDayTime r
                    return $ Class di g p cr d t
  -- — — — — — — — — — — — — — — — — — –
data Route = Route { routeParts      :: [PartClass]
                   , hasDisciplines  :: Bool
                   , hasGroups       :: Bool
                   , hasProfessors   :: Bool
                   , hasRooms        :: Bool
                   , hasDayTime      :: Bool
                   }
class UpdRoute (r :: Role) where updRoute :: Node r → Route → Route
updRoute′ upd (Node xs) r =
    do (pc, x) ← routeParts r ‘zip‘ xs
       [upd pc x]
instance UpdRoute Groups where
  updRoute n r = r  {
    hasDisciplines    = True,
    hasGroups         = True,
    routeParts        = updRoute′ (λpc (g, d) → pc { mbGroup = Just g
                                                   , mbDiscipline = Just d
                                                   }) n r
                    }
instance UpdRoute DayTime where
```

```
  updRoute n r = r {
    hasDayTime = True,
    routeParts = updRoute′ (λpc x → pc {mbDayTime = Just x}) n r
    }
instance UpdRoute Professors where
  updRoute n r = r {
    hasProfessors = True,
    routeParts = updRoute′ (λpc x → pc {mbProfessor = Just x}) n r
    }
instance UpdRoute Classrooms where
  updRoute n r = r {
    hasRooms = True,
    routeParts = updRoute′ (λpc x → pc {mbRoom = Just x}) n r
    }
```

3.2 Relations

3.2.1 Restrictions

Classes must be *time consistent* for each *group*, *professor* and *classroom*.

```
timeConsistent :: Route → Bool
timeConsistent r =
  let test :: (Ord a) ⇒ (Route → Bool) → (PartClass → a) → Maybe Bool
      test b sel = if b r  then  timeConsistent′ (routeParts r) sel
                                   <|> Just False
                           else Nothing
      bs = [test hasGroups mbGroup
           ,test hasProfessors mbProfessor
           ,test hasRooms mbRoom
           ]
  in hasDayTime r ∧ fromMaybe False (foldr (<|>) Nothing bs)
timeConsistent′ :: (Ord a) ⇒ [PartClass] → (PartClass → a)
                           → Maybe Bool
timeConsistent′ pcs select = foldr f Nothing byRole
  where byRole = groupWith select pcs
    f xs acc = (∨) <$> acc <∗> timeIntersect xs
mbAllJust :: [Maybe a] → Maybe [a]
mbAllJust l = inner l []
  where inner (Just x : xs) acc = inner xs (x : acc)
        inner []            acc = Just acc
        inner _             _   = Nothing
timeIntersect :: [PartClass] → Maybe Bool
timeIntersect = fmap hasRepetitions ◦ mbAllJust ◦ map mbDayTime
```

```
hasRepetitions (x : xs) = x ∈ xs ∨ hasRepetitions xs
hasRepetitions []       = False
```

Obligations:

```
data Obligation (r :: Role) = Obligation {
  obligationName   :: String
  , assessObligation :: RoleValue r → PartClass → Maybe Bool
  }
professorCanTeach :: Obligation Professors
professorCanTeach = Obligation "Can teach"
                    $ λp c → fmap (∈ canTeach p) (mbDiscipline c)
roomSatisfies :: Obligation Classrooms
roomSatisfies = Obligation "Room Capacity and Special Requirements"
              $ λr c → do gr ← mbGroup c
                          di ← mbDiscipline c
                          return $ roomCapacity r ⩾ groupSize gr
                             ∧ all (∈ roomEquipment r)
                                   (disciplineReqs di)
```

3.2.2 Preferences

```
data Preference (r :: Role) = Preference {
  preferenceName   :: String
  , assessPreference ::  RoleValue r  → Discipline
                       → (Day, Time) → InUnitInterval
  }
  -- ––––––––––––––––––––––––––––––––––––––––
newtype InUnitInterval = InUnitInterval Float
inUnitInterval n = if 0 ⩽ n ∧ n ⩽ 1
                   then Just $ InUnitInterval n
                   else Nothing
inUnitInterval′ = fromJust ∘ inUnitInterval
fromUnitInterval (InUnitInterval n) = n
```

3.2.3 Assessment

```
data ByRole v = ∀r.(RoleExtra r) ⇒ ByRole (Role′ r) [v r]
type SomeObligations = ByRole Obligation
type SomePreferences = ByRole Preference
  -- ––––––––––––––––––––––––––––––––––––––
```

```
assessPart :: SomeObligations → SomePreferences
           → PartClass       → InUnitInterval
assessPart obligations preferences pc =
  inUnitInterval′ $ if satisfies obligations
                    then mean $ assess preferences
                    else 0
  where satisfies (ByRole r os) = case r ‘mbRole‘ pc of
            Just rr → all (fromMaybe False
                          ∘ ($pc) ∘ ($rr)
                          ∘ assessObligation
                          ) os
            Nothing → True
          mean xs = sum xs / fromIntegral (length xs)
          assess _ = [ ]
η :: SomeObligations → SomePreferences → Route → InUnitInterval
η obligations preferences route = ⊥
  where isValid = timeConsistent


η obligations preferences route =
let satisfies c (ByRole r os) = all (  ($c) ∘ ($r ‘routeRole‘ c)
                                     ∘ assessObligation
                                    ) os
  mean xs = sum xs / fromIntegral (length xs)
  assess (ByRole r ps) c = map (  fromUnitInterval
                                ∘ ($(classDay c, classBegins c))
                                ∘ ($classDiscipline c)
                                ∘ ($r ‘routeRole‘ c)
                                ∘ assessPreference
                               ) ps
in inUnitInterval′ $ if route ‘satisfies‘ obligations
                     then mean $ preferences ‘assess‘ route
                     else 0
```

3.3 ACO

```
data SetupACO = SetupACO {α :: Float
                         , β :: Float
                         , 𝒬 :: Float
                         , ρ :: Float
                         }
newtype Pheromone = Pheromone Float
data NodesACO = NodesACO ()
type RelationsACO = (SomeObligations, SomePreferences)
```

```
data ACO = AO { setupACO      :: SetupACO
              , relationsACO :: RelationsACO
              }
```

3.3.1 Graph

The **problem graph** is defined by the nodes of each *role*; while the edges hold the *pheromone*. If the memory permits it, the graph should hold all the permutations of *role*s domains.

```
type NodeSet r = Set (Node r)
type NodeKey = (AnyRole, String)
type PheromoneBetween = Map (AnyRole, AnyRole) Pheromone
data Graph = Graph { groupsNodes       :: NodeSet Groups
                   , temporalNodes     :: NodeSet DayTime
                   , professorsNodes   :: NodeSet Professors
                   , classroomsNodes   :: NodeSet Classrooms
                   , currentPheromone  :: IORef PheromoneBetween
                   }
  -- —————————————————
data AnyRole = ∀r.(Typeable r, RoleExtra r) ⇒ AnyRole (Role′ r)
roleIx′ (AnyRole r) = roleIx r
instance Eq  AnyRole where (≡)     = (≡)     ‘on‘ roleIx′
instance Ord AnyRole where compare = compare ‘on‘ roleIx′
```

3.3.2 Evaluation

Route *probabilistic evaluation* function:

```
evalRoutes :: ACO → PheromoneBetween → [Route]
           → [(InUnitInterval, Route)]
evalRoutes aco ph rs  =  first (fromJust ∘ inUnitInterval ∘ (/psum))
                         <$> zip ps rs
  where ps   = map p rs
        psum = sum ps
        p r  = (τ r)^α · (η′ r)^β
        η′ = fromUnitInterval ∘ uncurry η (relationsACO aco)
        τ r = ⊥   -- TODO
```