
# Connectionist and Evolutionary Systems: ACO

# Final Project: *UCSP*

Dmitry K.

May 12, 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.

# Contents

# 1 Problem

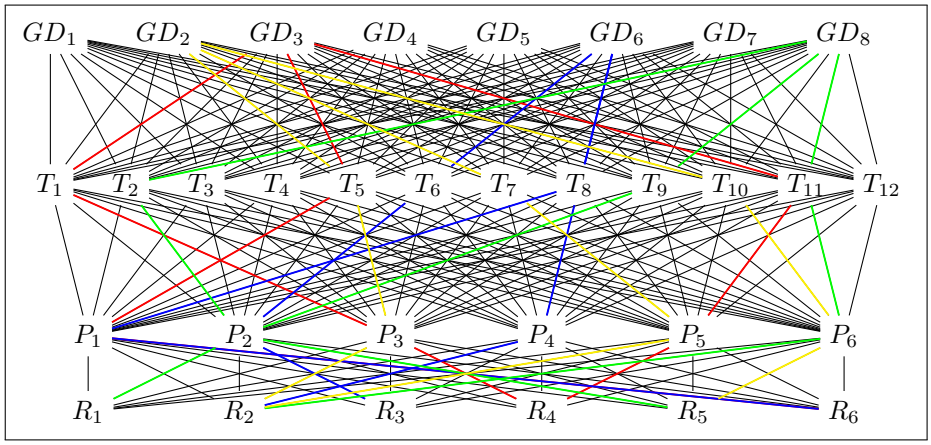


Figure 1: Problem graph schematic, representing **G**roups, **D**isciplines, **T**ime/day, **P**rofessors, Class**R**ooms.

## 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.

```
  -- 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*.

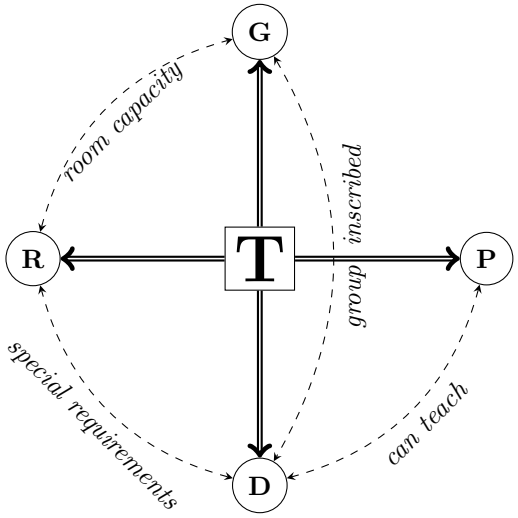


Figure 2: *Class* structure.

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) :: *
newtype Node (r :: Role) = Node (String, [RoleValue r])
nodeId (Node (id, _)) = id
instance Eq   (Node r) where (≡)       = (≡)       `on` nodeId
instance Ord  (Node r) where compare = compare `on` nodeId
  -- — — — — — — — — — — — — — — — — — —
class HasDomain a v | a → v
  where domain      :: a → Set v
        domainPower :: a → Int
type RoleDomain r = HasDomain (Role' r) (RoleValue r)
mkNodes :: RoleDomain r ⇒ String → Role' r → [Node r]
mkNodes name = map Node
             ∘ zip (map ((name++) ∘ show) [1..])
             ∘ 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.

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Eq}\ t, \mathit{Ord}\ t, \mathit{Enum}\ t, \mathit{Bounded}\ t) \Rightarrow \\
&\quad \mathit{DiscreteTime}\ t\ \textbf{where}\ \mathit{timeQuantum} :: t \quad \to \mathit{Int} \\
&\qquad\qquad\qquad\qquad\quad \mathit{toMinutes} \quad :: t \quad \to \mathit{Int} \\
&\qquad\qquad\qquad\qquad\quad \mathit{fromMinutes} \ :: \mathit{Int} \to \mathit{Maybe}\ t \\
&\textbf{class}\ (\mathit{DiscreteTime}\ t, \mathit{Enum}\ d, \mathit{Bounded}\ d) \Rightarrow \\
&\quad \mathit{Timetable}\ tt\ t\ d\ ev \mid tt \to t \\
&\qquad\qquad\qquad\qquad\quad , tt \to d \\
&\qquad\qquad\qquad\qquad\quad , tt \to ev \\
&\quad \textbf{where}\ \mathit{listEvents} \ :: tt \qquad\qquad \to [((d,t), ev)] \\
&\qquad\quad \mathit{newTTable} :: [((d,t), ev)] \to tt \\
&\qquad\quad \mathit{eventsOn} \ :: tt \to d \qquad\quad \to [(t, ev)] \\
&\qquad\quad \mathit{eventsAt} \ :: tt \to t \qquad\quad \to [(d, ev)] \\
&\qquad\quad \mathit{eventAt} \ :: tt \to d \to t \ \to \mathit{Maybe}\ ev
\end{aligned}
$$

## 1.3 Graph Edges

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

$\forall a \in \text{Layer}_A$
$\forall b \in \text{Layer}_B$
if $\text{Layer}_A$ and $\text{Layer}_B$ are neighbors
$\exists$ an edge between $a$ and $b$.

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*.

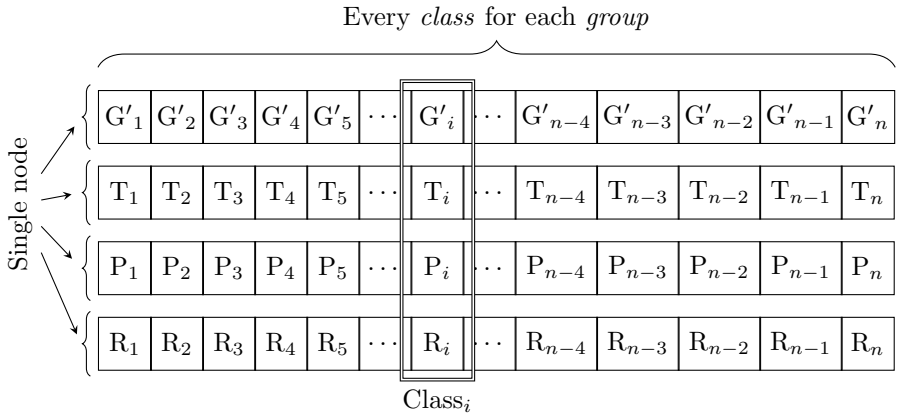


Figure 4: *Route* decomposition.

# 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 $\times$ 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 $g$;

$N_\Sigma = \sum_{g \in G} \sum_{d \in D_g} N_d^g$ — 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 $N_\Sigma$, 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 = Discipline { 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 * 21 + 30

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
                    }
  -- — — — — — — — — — — — — — — — — — —
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
  roleName :: Role' r → String
  mbRole   :: Role' r → PartClass → Maybe (RoleValue r)
  classRole :: Role' r → Class → RoleValue r
instance RoleExtra Groups     where roleIx _    = 0
                                    roleName _  = "Groups"
                                    mbRole _ r  = (,) <$>
                                                  mbGroup r <*>
                                                  mbDiscipline r
                                    classRole _ = classGroup &&&
                                                  classDiscipline
instance RoleExtra DayTime    where roleIx _    = 1
                                    roleName _  = "DayTime"
                                    mbRole _    = mbDayTime
                                    classRole _ = classDay &&&
                                                  classBegins
instance RoleExtra Professors  where roleIx _    = 2
                                    roleName _  = "Professors"
                                    mbRole _    = mbProfessor
                                    classRole _ = classProfessor
instance RoleExtra Classrooms where roleIx _    = 3
                                    roleName _  = "Classrooms"
                                    mbRole _    = mbRoom
                                    classRole _ = classRoom
instance (RoleExtra r) ⇒ Show (Role' r) where show = roleName
```

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]
                    , mbGroupsNode        :: !(Maybe (Node Groups))
                    , mbDayTimeNode       :: !(Maybe (Node DayTime))
                    , mbProfessorsNode    :: !(Maybe (Node Professors))
                    , mbRoomsNode         :: !(Maybe (Node Classrooms))
                    , assessHistory       :: ![InUnitInterval]
                    }
hasDisciplines = isJust ∘ mbGroupsNode
hasGroups      = isJust ∘ mbGroupsNode
hasProfessors  = isJust ∘ mbProfessorsNode
hasRooms       = isJust ∘ mbRoomsNode
hasDayTime     = isJust ∘ mbDayTimeNode
emptyRoute = Route [] Nothing Nothing Nothing Nothing []
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  {
    mbGroupsNode = Just n,
    routeParts       = updRoute' (λpc (g, d) → pc { mbGroup = Just g
                                                  , mbDiscipline = Just d
                                                  }) n r
                    }
instance UpdRoute DayTime where
  updRoute n r = r  {
    mbDayTimeNode = Just n,
    routeParts = updRoute' (λpc x → pc { mbDayTime = Just x }) n r
    }
instance UpdRoute Professors where
  updRoute n r = r  {
    mbProfessorsNode = Just n,
    routeParts = updRoute' (λpc x → pc { mbProfessor = Just x }) n r
```

```
    }
instance UpdRoute Classrooms where
  updRoute n r = r  {
    mbRoomsNode = Just n,
    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 forRole
  where forRole = 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 ForRole v = ∀r.(RoleExtra r) ⇒ ForRole (Role' r) [v r]
forRole vs = ForRole Role' vs
type SomeObligations = [ForRole Obligation]
type SomePreferences = [ForRole Preference]
  -- — — — — — — — — — — — — — — — — — — –
mean [] = 0
mean xs = sum xs / fromIntegral (length xs)
uncurry3 :: (a → b → c → d) → (a, b, c) → d
uncurry3 f (a, b, c) = f a b c
assessPart :: SomeObligations → SomePreferences
           → PartClass        → InUnitInterval
assessPart obligations preferences pc =
```

```
  inUnitInterval' $ if all satisfies obligations
                    then mean $ concatMap assess preferences
                    else 0
  where satisfies (ForRole r os) = case r `mbRole` pc of
          Just rr ->  all (fromMaybe False
                             ∘ ($pc) ∘ ($rr)
                             ∘ assessObligation
                           ) os
          Nothing -> True
        assess (ForRole r ps) = fromMaybe [] $
          do dt ← mbDayTime pc
             di ← mbDiscipline pc
             rv ← mbRole r pc
             return $ map (fromUnitInterval
                             ∘ ($(rv, di, dt))
                             ∘ uncurry3
                             ∘ assessPreference
                           ) ps
η :: SomeObligations → SomePreferences → Route → InUnitInterval
η obligations preferences route = fromJust ∘ inUnitInterval $
  if timeConsistent route ∧ notElem 0 assessed then mean assessed
                                               else 0
  where assessed  =  fromUnitInterval ∘ assessPart obligations preferences
                 <$> routeParts route
```

### 3.3 ACO

```
data ParamsACO = ParamsACO {α   :: Float
                           , β   :: Float
                           , 𝒬   :: Float
                           , 𝒬₀  :: Float
                           , ρ   :: Float
                           }
type RelationsACO = (SomeObligations, SomePreferences)
data PopulationACO = ∀gen.RandomGen gen ⇒
     GenPopulation Int GenUnique (IORef gen)
type GenUnique = Bool
data ACO = ACO {setup          :: ParamsACO
               , relationsACO  :: RelationsACO
               , populationACO :: PopulationACO
               }
  -- — — — — — — — — — — — — — — — — — — —
newtype Pheromone = Pheromone Float deriving (Show, Eq, Ord)
```

```
pheromoneQuantity (Pheromone n) = n

mapPheromone f (Pheromone n) = Pheromone (f n)
mapPheromone2 f (Pheromone x) (Pheromone y) = Pheromone (f x y)
instance Num Pheromone where (+) = mapPheromone2 (+)
                             (-) = mapPheromone2 (-)
                             (*) = mapPheromone2 (*)
                             abs    = mapPheromone abs
                             signum = mapPheromone signum
                             fromInteger = Pheromone ∘ fromInteger
```

### 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 PheromoneBetween = Map (AnyNode, AnyNode) Pheromone
type PheromoneCache   = Map (AnyNode, AnyNode) (IORef Pheromone)
data Graph = Graph { groupsNodes      :: NodeSet Groups
                   , temporalNodes    :: NodeSet DayTime
                   , professorsNodes  :: NodeSet Professors
                   , classroomsNodes :: NodeSet Classrooms
                   , pheromoneCache :: PheromoneCache
                   }
currentPheromone :: Graph → IO PheromoneBetween
currentPheromone = mapM readIORef ∘ pheromoneCache

phKey (x, y) = (x `min` y, x `max` y)

  -- Lazy update
updPheromone ::  Graph
             → (AnyNode, AnyNode)
             → (Pheromone → Pheromone)
             → IO ()
updPheromone g k upd =
  case phKey k `Map.lookup` pheromoneCache g of
    Just ref  → modifyIORef ref upd
    _         → error $ "no pheromone cache for " ++ show k

data ExecACO = ExecACO { exACO  :: ACO
                       , exGraph :: Graph
                       , exRuns  :: IORef Int
                       }
  -- — — — — — — — — — — — — — — — — — —
data AnyNode = ∀r.(Typeable r, RoleExtra r) ⇒
```

```
      AnyNode (Role' r) (Node r)
nodeRoleIx (AnyNode r _)      = roleIx r
nodeId'    (AnyNode _ n)      = nodeId n
nodeId'' = nodeRoleIx &&& nodeId'

anyNode n = AnyNode Role' n

instance Eq   AnyNode where (≡)       = (≡)       `on` nodeId''
instance Ord  AnyNode where compare = compare `on` nodeId''

instance Show AnyNode where
  show (AnyNode r n) = "Node-" ++ show r ++ ":" ++ nodeId n

routeNodes :: Route → [AnyNode]
routeNodes r = mapMaybe ($r) [packNode ∘ mbRoomsNode
                             , packNode ∘ mbProfessorsNode
                             , packNode ∘ mbDayTimeNode
                             , packNode ∘ mbGroupsNode
                             ]
  where packNode :: (Typeable r, RoleExtra r) ⇒
                    Maybe (Node r) → Maybe AnyNode
        packNode = fmap (AnyNode Role')
```

### 3.3.2 Evaluation

Route *probabilistic evaluation* function:

```
ξ ::          ACO → PheromoneBetween → [Route]
  →           [(InUnitInterval, Route)]
ξ aco ph rs =
  first (fromJust ∘ inUnitInterval ∘ (/psum))
    <$>       zip ps rs'
  where (rs', ps) = unzip $ map p rs
        psum = sum ps
        p r    = let (r', η') = assessRoute' r
                 in (r', τ^α · η'^β)
        assessRoute' r = let v = uncurry η (relationsACO aco) r
                         in (r { assessHistory = v : assessHistory r }
                            , fromUnitInterval v
                            )
        τ = case routeNodes r of
              x : y : _ → maybe Q0 pheromoneQuantity
                                $ phKey (x, y) `Map.lookup` ph
              _         → Q0
```

Pheromone secretion for each neighboring nodes pair in a route:

```
Δτr :: ACO → Route → [((AnyNode, AnyNode), Pheromone)]
Δτr aco r =
```

```
  let edgs = lPairs $ routeNodes r
      hist = assessHistory r
      w    = Q / sum (map fromUnitInterval hist)
      weight = Pheromone ∘ (*w) ∘ fromUnitInterval
  in if length edgs ≢ length hist
     then error "[BUG] wrong assess history length"
     else edgs `zip` map weight hist
lPairs (x0 : x1 : xs) = (x0, x1) : lPairs (x1 : xs)
lPairs _              = []
```

Pheromone update (secretion and vaporization):

```
Δ~τ :: ExecACO → [Route] → IO ()
Δ~τ ExecACO {exACO = aco, exGraph = graph} rs = forM_ rs update
                                                ≫ vaporize
  where update r = sequence_ $ do (i, ph) ← Δτ_r aco r
                                  [updPheromone graph i (+ ph · ρ)]
        vaporize = sequence_ $ do ref ← Map.elems $
                                        pheromoneCache graph
                                  [modifyIORef' ref (·(1 − ρ))]   -- strict
```

### 3.3.3 Execution

```
type StopCriteria = ExecACO → IO Bool
execACO :: ExecACO → StopCriteria → IO [Route]
execACO ex@ExecACO {exACO = aco, exGraph = graph} stop = result
  where
```

1. Generate initial population by selectting randomly some nodes at *Group–Discipline* layer:

```
initialPopulation = case populationACO aco of
  GenPopulation size unique genRef → do
    gen ← readIORef genRef
    let rand' = if unique then randChoosesUnique
                          else randChoices
        gdNodes = Set.toList $ groupsNodes graph
        (g, rand) = rand' gen size gdNodes
    writeIORef genRef g
    putStrLn "Generated Initial Population"  -- DEBUG
    return $ map (`updRoute` emptyRoute) rand
```

2. ∀ layer, route **do** :

(a) Generate all the possible *route* continuations by updating the *route* with *layer*'s nodes.

(b) Assess continuations with $\xi$ and select the node according to the assessed probabilities.

```
nextRoute :: (UpdRoute r, RandomGen gen) ⇒
             PheromoneBetween → NodeSet r → gen
          → Route → (gen, Route)
nextRoute ph nset gen r =
  let candidates = (`updRoute`r) <$> Set.toList nset
      evCandidates = ξ aco ph candidates
  in second snd $ randCoiceWithProb gen fst evCandidates

nextRoutes _ acc _ g [] = (g, acc)
nextRoutes ph acc nset gen (r : rs) =
  let (g', next) = nextRoute ph nset gen r
  in nextRoutes ph (next : acc) nset g' rs

routesIO = do ph ← currentPheromone graph
              g0 ← getStdGen
              p0 ← initialPopulation

              let next :: (RandomGen gen, UpdRoute r) ⇒
                          (Graph → NodeSet r) → gen
                       → [Route] → (gen, [Route])
                  next      = nextRoutes ph [] ∘ ($graph)

                  (g1, p1) = next temporalNodes g0 p0
                  (g2, p2) = next professorsNodes g1 p1
                  (g3, p3) = next classroomsNodes g2 p3

              setStdGen g3
              putStrLn "routesIO"  -- DEBUG
              return p3
```

3. Update pheromone and counter:

```
updateStates rs = do (Δ~τ) ex rs
                     exRuns ex `modifyIORef` (+1)
```

4. **Return** best routes **if** *stop criteria* applies, **go to 1** otherwise.

```
result = do routes ← routesIO
            updateStates routes
            stop' ← stop ex
            if stop' then return routes
                     else execACO ex stop
```

```
randChoice gen xs =
  if null xs then error "randChoice: empty list"
             else first (xs!!) $ randomR (0, length xs - 1) gen

randChoices gen count xs =
  if length xs < count
  then randChoices gen (length xs) xs
  else swap $ foldr rand ([], gen) [1 .. count]
       where rand _ (acc, g) = first (:acc) $ randChoice g xs

randUniqueIndices gen count length =
  let ixSet  = Set.fromList [1 .. length]
  in if count > length then error "randUniqueIndices: count > length"
                       else  inner gen ixSet count []
  where inner g _ 0 acc = (g, acc)
        inner g s c acc = let (i, g') = randomR (0, Set.size s - 1) g
                              v       = Set.elemAt i s
                              s'      = Set.delete v s
                          in inner g' s' (c - 1) (v : acc)

randChoosesUnique gen count xs =
  if length xs < count
    then randChoosesUnique gen (length xs) xs
    else second (map (xs!!)) $ randUniqueIndices gen count (length xs)

randCoiceWithProb :: (RandomGen gen) =>
                     gen -> (a -> InUnitInterval) -> [a] -> (gen, a)
randCoiceWithProb gen probOf xs =
  let (r, g') = random gen
      accumulating acc _ [] = acc
      accumulating acc f (x : xs) = case f acc x of
        Just acc' -> accumulating acc' f xs
        _         -> acc
      f (p, _) x = let p' = p + fromUnitInterval (probOf x)
                   in if p' > r  then Nothing
                                 else Just (p', x)
  in (g', snd $ accumulating (0, head xs) f (tail xs))
```

### 3.3.4 Creation

Here follows creation of an '*ExecACO*' instance.

```
newExecACO :: ( HasDomain GroupsData Group
              , RoleDomain DayTime
              , RoleDomain Professors
              , RoleDomain Classrooms
```

```
                  ) ⇒
                  ACO → IO ExecACO
newExecACO aco = do
  let gs = mkNodes "G" (Role' :: Role' Groups)
      ts = mkNodes "T" (Role' :: Role' DayTime)
      ps = mkNodes "P" (Role' :: Role' Professors)
      rs = mkNodes "R" (Role' :: Role' Classrooms)
      ks = concat     [ pairs gs ts
                      , pairs ts ps
                      , pairs ps rs
                      ]
      pairs xs ys = do x ← xs
                       y ← ys
                       return (anyNode x, anyNode y)
  cache ← sequence $ do k ← ks
                        [(, ) k <$> newIORef (Pheromone Q0)]
  let graph = Graph (Set.fromList gs)
                    (Set.fromList ts)
                    (Set.fromList ps)
                    (Set.fromList rs)
                    (Map.fromList cache)
  countRef ← newIORef 0
  return $ ExecACO aco graph countRef
```

*Groups* nodes are created respecting groups' disciplines.

```
data GroupsData = GroupsData
instance (HasDomain GroupsData Group) ⇒
  HasDomain (Role' Groups) (Group, Discipline)
  where domain _  =  Set.unions
                  $  (λg → Set.map ((, ) g) $ groupDisciplines g)
                 <$> Set.toList (domain GroupsData)
        domainPower = Set.size ∘ domain
```

*Day–Time* domain is determined by the *timetable*.

```
instance HasDomain (Role' DayTime) (Day, Time)
  where domainPower _ = 6 ∗ (timeDMax + 1)
        domain _ = Set.fromList $ do d ← [minBound . .]
                                     t ← [minBound . . maxBound]
                                     return (d, t)
```

# 4 Tests №1

Here are presented some *ACO* test runs with the following data.

## 4.1 Data

### 4.1.1 Disciplines

Here are defined some 10 disciplines.

```
labComputers = Requirement "Computer Lab"
labElectronics = Requirement "Electronics Lab"

dA1 = Discipline "A1" 60 ∅
dA2 = Discipline "A2" 60 ∅
dA3 = Discipline "A3" 60 ∅
dA4 = Discipline "A4" 60 ∅

dB1 = Discipline "B1" 30 ∅
dB2 = Discipline "B2" 30 ∅
dB3 = Discipline "B3" 90 ∅

dC1 = Discipline "C1" 60 [labComputers]
dC2 = Discipline "C2" 60 [labComputers]
dC3 = Discipline "C3" 60 [labElectronics]

dataDisciplines = [ dA1, dA2, dA3, dA4
                  , dB1, dB2, dB3
                  , dC1, dC2, dC3
                  ]
```

### 4.1.2 Groups

Here are defined some 7 groups.

```
dataGroups = [ Group "1-1" 20 [dA1, dA2, dC2]
             , Group "1-2" 18 [dA1, dA2, dC2]
             , Group "2-1" 14 [dA1, dA3, dC3, dB1]
             , Group "2-2" 15 [dA1, dA3, dC3, dB2]
             , Group "3-1" 12 [dC1, dA1, dA2]
             , Group "3-2" 11 [dC1, dA1, dA2]
             , Group "4"   40 [dB3, dA4]
             ]

instance HasDomain GroupsData Group
  where domainPower _ = length dataGroups
        domain _ = Set.fromList dataGroups
```

### 4.1.3 Professors

Here are defined some 4 professors.

```
dataProfessors = [Professor "01" [dA1, dA2, dC1, dC2]
                 ,Professor "02" [dA2, dA3, dC2, dC3]
                 ,Professor "03" [dA1, dA2, dA3, dA4]
                 ,Professor "04" [dB1, dB2, dB3]
                 ]
instance HasDomain (Role' Professors) Professor
  where domainPower _ = length dataProfessors
        domain _ = Set.fromList dataProfessors
```

### 4.1.4 Classrooms

Here are defined some 5 classrooms.

```
dataRooms = [Classroom "A01" 20 ∅
            ,Classroom "A02" 20 ∅
            ,Classroom "LC1" 20 $ set [labComputers]
            ,Classroom "LE1" 15 $ set [labElectronics]
            ,Classroom "B01" 50 ∅
            ]
instance HasDomain (Role' Classrooms) Classroom
  where domainPower _ = length dataRooms
        domain _ = Set.fromList dataRooms
```

## 4.2 Tests Execution

### 4.2.1 Stop Criteria

The algorithm should stop after given number of iterations run.

```
stop n ex = do n' ← readIORef $ exRuns ex
               putStrLn "Stop?"
               return $ n' ⩾ n
```

### 4.2.2 ACO Setup

Here follow different $ACO$ parameters configuration parts.

```
acoParams1 = ParamsACO {α   = 0.14
                       ,β   = 0.16
                       ,𝒬   = 10
                       ,𝒬₀ = 0.1
```

```
                                  , ρ    = 0.5
                                  }
  -- — — — — — — — — — — — — — — — — — — — —
mkAcoPop aPop = do gen ← newStdGen
                   ref  ← newIORef gen
                   return $ aPop ref
acoPopulation1 = GenPopulation (domainPower (Role' :: Role' Groups))
                               True
  -- — — — — — — — — — — — — — — — — — — — —
obligations = [forRole [professorCanTeach]
              ,forRole [roomSatisfies]
              ]
relations = (,) obligations
aco params preferences = ACO params (relations preferences)
```

Test params:

```
aco' = aco acoParams1 [] <$> mkAcoPop acoPopulation1
exec = newExecACO =≪ aco'
run = flip execACO (stop 100) =≪ exec
```

FAIL

it doesn't work