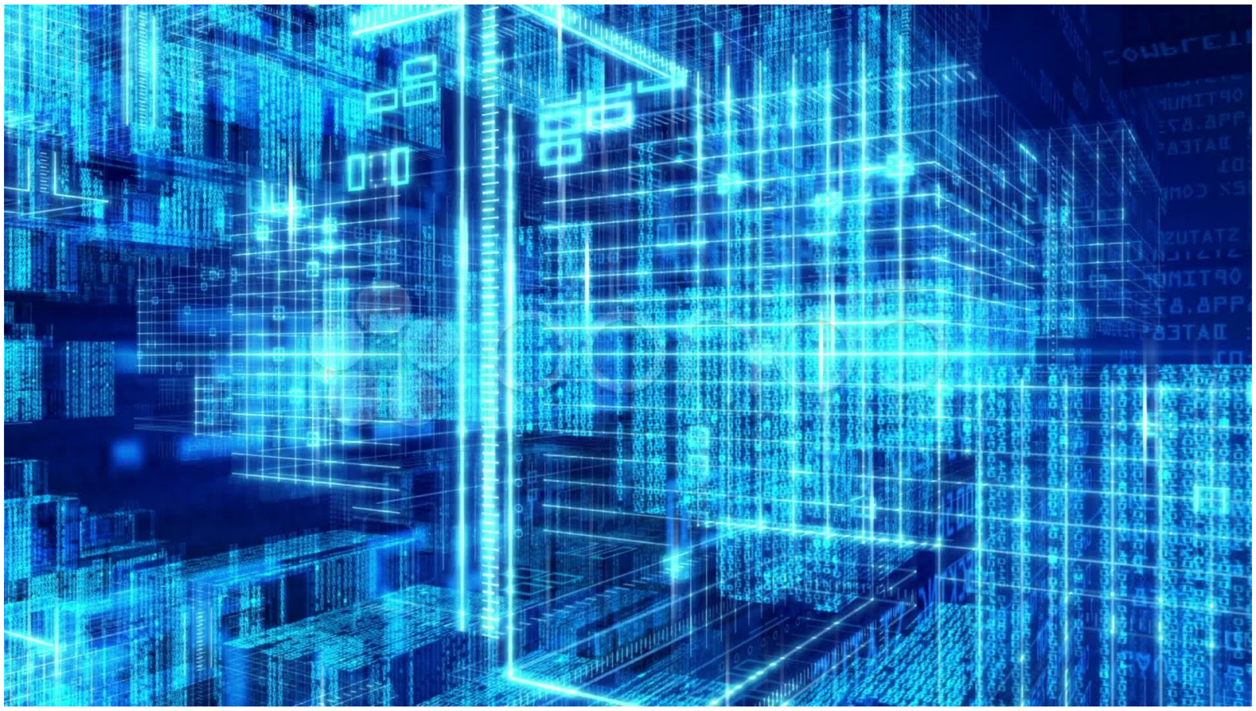
Grid Computing (Scheduling)

Virtual machine placement using Ant Colony, Artificial Bee Colony, FireFly, Tabu Search

Mohamad Reza Jebeli Haji Abadi



Introduction

This report reviews the result of research and works that have been done during the internship program of IPM. The purpose of this program was to study different optimization algorithms and apply them to the VM placement problem.

For each algorithm i declared 3 main class for designing. Problem, State, Search. Problem and State classes are approximately the same for designing of each algorithm. So the optimization algorithm designed in Search class for each step.

Fitness function explanation

Fitness function that i named it to cost() in ant colony optimization and named it to f() in other algorithms is a function that gives a solution as an argument, and give a number to each solution based on how much this solution is good.

First of all i move 1 time on solution array and compute busy part of each cluster. For this purpose i just compute busy cores and don't consider ram. Then i begin to compute the utilization of each cluster that means $\frac{busy\ cores}{all\ cores}$.

Fitness function return value is :

$$Util(C0)^{2} + Util(C0)^{2} + \ldots + Util(C0)^{2}$$

So All Algorithms try to place jobs to fewer clusters to maximize the fitness value.

1. Ant Colony Optimization

Ant colony optimization (ACO) is a population-based metaheuristic that can be used to find approximate solutions to difficult optimization problems.

In ACO, a set of software agents called *artificial ants* search for good solutions to a given optimization problem.

The solution construction process is stochastic and is biased by a *pheromone model*, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants.

Formulas:

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

$$\Delta\tau_{ij} = \sum_{k=1}^{l} \Delta\tau_{ij}^{k}$$

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

$$p_{ij}^{k} = \begin{cases} \dfrac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{s \in allowed_k} [\tau_{is}]^{\alpha} \cdot [\eta_{is}]^{\beta}} & j \in allowed_k \\ 0 & \text{otherwise} \end{cases}$$

$\Delta\tau_{ij}$: *total amount of pheromone increase on edge i , j*

$\Delta\tau^{k}_{ij}$: *total amount of pheromone increase on edge i , j by kth ant*

L_k : *the length of kth ant tour*

Q : *constant*

ρ : *pheromone evaporation rate*

τij : *pheromone amount on i,j edge*

Ant Colony Optimization for Job Scheduling Problem

Each ant is a solution. Solution is a 1D array represents each job assign to which cluster. For example solution[i] = j represents that ith job assign to jth cluster. Jobs numbered from 0 to (number of jobs - 1) and clusters numbered from 0 to (number of clusters - 1). So the length of solution array is according to number of jobs and each cell has a value between 0 to number of clusters. If solution[i] = len(clusters) means that ith job doesn't assign to any clusters.

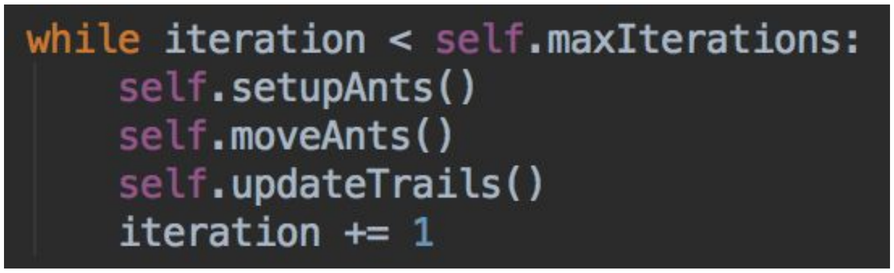
At the first time that algorithm runs and ants want to find their solution, They go just according to heuristic array that designed for this purpose that ants don't choose no clusters. After all ants found their solution, we start to make changes in pheromone array according to solutions. It means that for each solution we look at each job that if job i assigned to cluster j, we should add a constant number to pheromone[i][j] cell. After that each ant starts to find another solution according to pheromone and heuristic array and this sequence repeats and we reach to a state that all ants goes convergent.

Important Variables :

numOfAnts :	number of ants that searching for best solution
ants :	each ant represents a solution. Ants is an array of my ants.
Clusters	an array that keeps informations of all clusters that i have
Jobs	an array that keeps informations of all request that i have
heuristic	heuristic[i][j] represents that is a good assigning that i assign job[i] to Cluster[j] regarding to experience.
Pheromone	pheromone[i][j] represents that how much pheromone is on i,j path This array at first fills with pheremoneInitialVal variable
pheremoneInitialVal :	At first fills pheromone array.

Code explanation:

After declaring clusters and jobs, and initializing pheromone array and heuristic array, i have a main while that repeats maxIteration times.



```
while iteration < self.maxIterations:
    self.setupAnts()
    self.moveAnts()
    self.updateTrails()
    iteration += 1
```

I have 3 basic function that i want to explain them :

setupAnts() :

In this function all ants assign their first job.

moveAnts() :

In this function all ants assign their other jobs according to last formula i explained that pheromone[i][j] used instead of τ_{ij} and heuristic[i][j] used instead of η_{ij} .

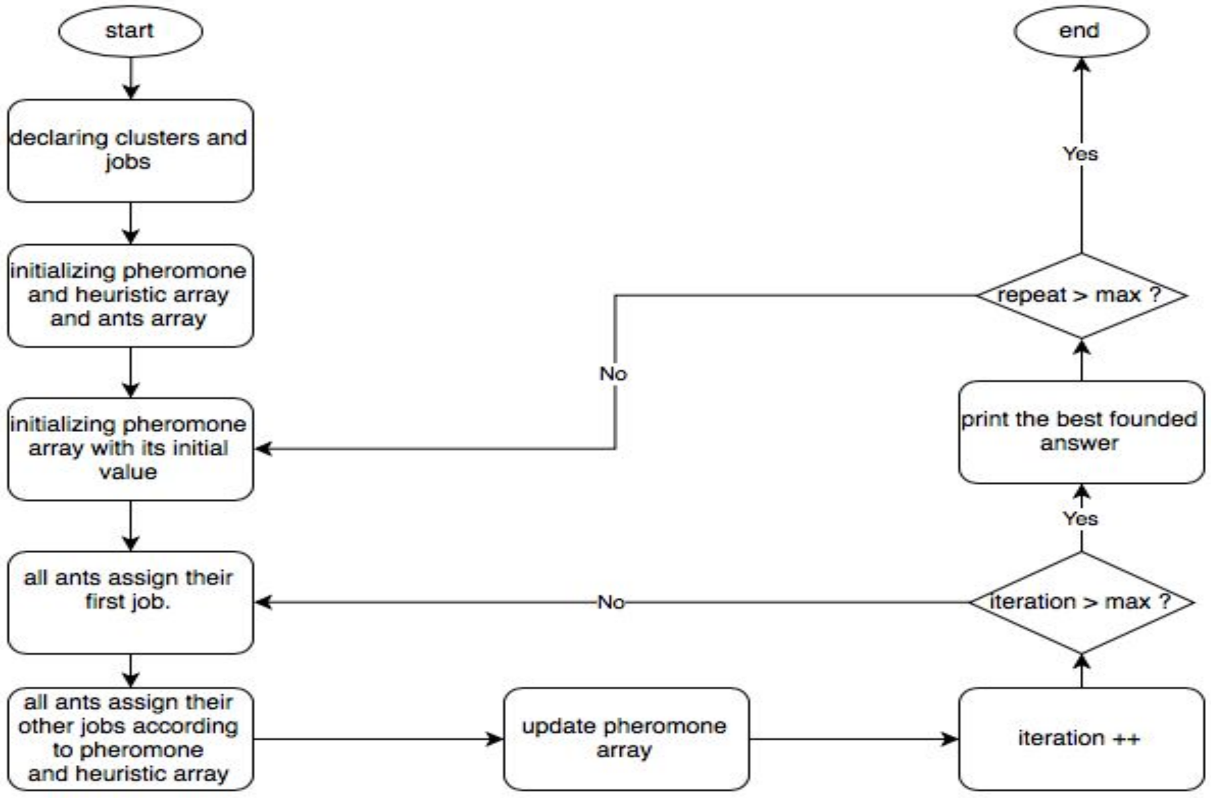
$\alpha = 3$

$\beta = 1$

updateTrails() :

I set evaporation rate to ½ that means at each running of this function all cells of pheromone array divided by 2, then i move on my solution and add pheromone to pheromone[i][j] when solution [i]= j but how much pheromone should i add? It's according to solution value.

Flow chart


start
declaring clusters and jobs
initializing pheromone and heuristic array and ants array
initializing pheromone array with its initial value
all ants assign their first job.
all ants assign their other jobs according to pheromone and heuristic array
update pheromone array
iteration ++
iteration > max ?
No
Yes
print the best founded answer
repeat > max ?
No
Yes
end

2. Artificial Bee Colony

In the **ABC** model, the colony consists of three groups of bees: employed bees, onlookers and scouts. It is assumed that there is only one artificial employed bee for each food source. In other words, the number of employed bees in the colony is equal to the number of food sources around the hive. Employed bees go to their food source and come back to hive and dance on this area. The employed bee whose food source has been abandoned becomes a scout and starts to search for finding a new food source. Onlookers watch the dances of employed bees and choose food sources depending on dances.

In ABC, a population based algorithm, the position of a food source represents a possible solution to the optimization problem and the nectar amount of a food source corresponds to the quality (fitness) of the associated solution.

Main steps of algorithm:

Initial food sources are produced for all employed bees

- REPEAT
    - Each employed bee goes to a food source in her memory and determines a closest source, then evaluates its nectar amount and dances in the hive
    - Each onlooker watches the dance of employed bees and chooses one of their sources depending on the dances, and then goes to that source. After choosing a neighbour around that, she evaluates its nectar amount.
    - Abandoned food sources are determined and are replaced with the new food sources discovered by scouts.
    - The best food source found so far is registered.
- UNTIL (requirements are met)

Formulas:

$$p_i = \frac{\text{fit}_i}{\sum_{n=1}^{\text{SN}} \text{fit}_n},$$

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}),$$

p_i : *probability that onlooker bees choose ith solution*

fit_i : *fitness value for solution i*

About second formula :

In each cycle each employed bee goes to his last solution(food) and decides to find   a better solution around his last solution according to the second formula. If the second food is better than last food bee choose second one neither keeps the last one.

v_{ij} : *new cluster assigning for ith employed bee and jth job*

x_{ij} : *old cluster for ith bee and jth job*

Φ : *learning factor to determine distance between new and old solutions*

Artificial Bee Colony for Job Scheduling Problem

Methods in Problem class:

randomState() : generates a random solution. Use for scout bees

f() : gives an state(solution) as an argument and give a number to its value.

equal : check the equality of 2 solution.

initialStates() : returns n random solution. Use for initializing employed bees.

nextStates() : receives all employed bees solution and produce new solutions near previous using mutant method in State class. Then if new solution was better than old keep it otherwise omit it.

assign() : receives a solution and an action to assign job i to cluster clusterIndex. If it is possible do it , otherwise do another possible assigning.

Methods in State class:

mutant() : receives 2 solution. For each job if both of them assigned to the same cluster, new solution assigns :

90% -> to that cluster

10% -> random

If didn’t assign the same cluster :

30% -> to cluster of first solution

35% -> to cluster of second solution

25% -> random

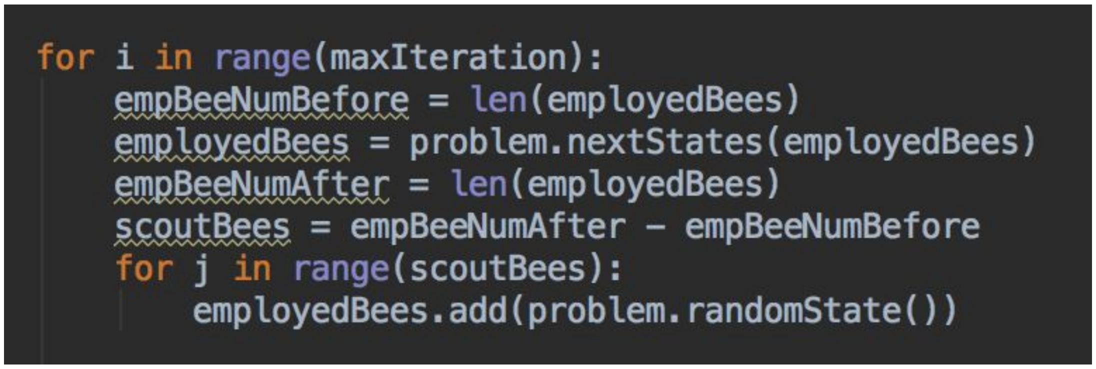
Search class :

First of all, all employed bees initiates randomly. Then we have a big loop that employed bees find their next solutions that which may be the same as before then scout bees find their random solution. If a employed bee time to live goes to 0 it becomes to a scout bee. At each cycle we should keep the best solution that we found until yet.

Important Variables :

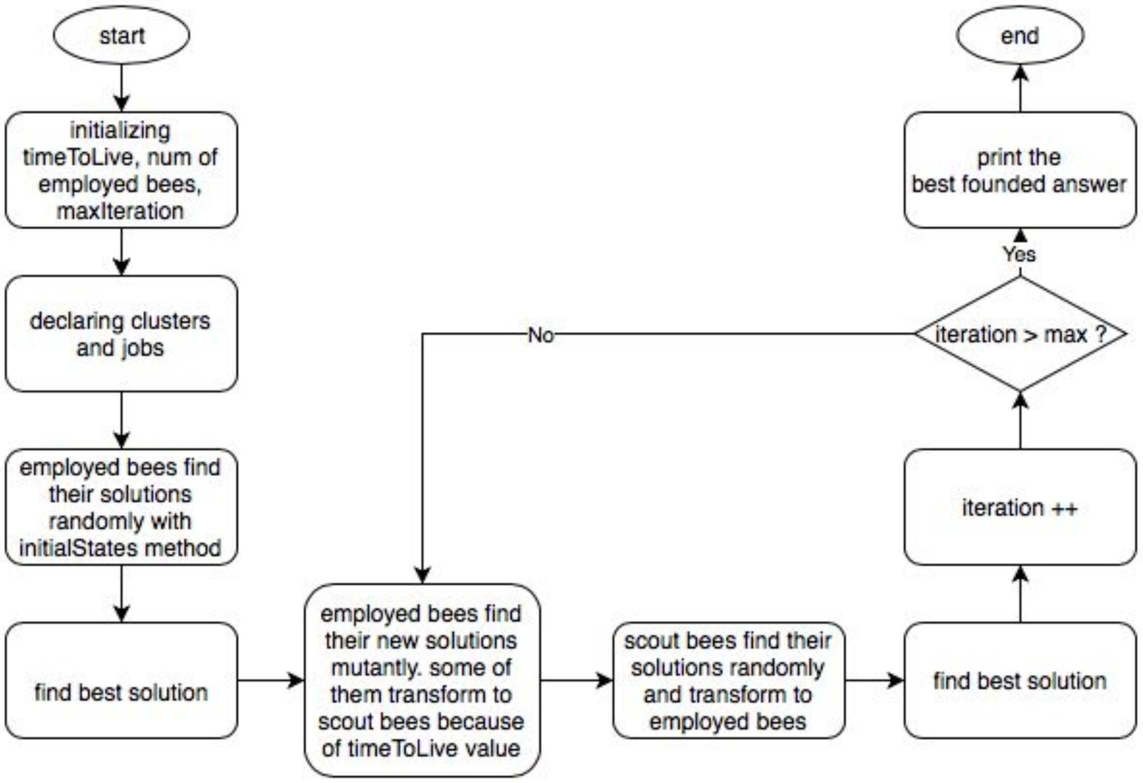
numOfEmployedBees :	number of employed bees.
employedBees :	an array that keeps my solutions that gained from employed bees
Clusters	an array that keeps informations of all clusters that i have
Jobs	an array that keeps informations of all request that i have
timeToLive	if an employed bee goes to a solution more than timeToLive times.
	He becomes to a scout bee and has to find a random solution

Main part of Code :



```
for i in range(maxIteration):
    empBeeNumBefore = len(employedBees)
    employedBees = problem.nextStates(employedBees)
    empBeeNumAfter = len(employedBees)
    scoutBees = empBeeNumAfter - empBeeNumBefore
    for j in range(scoutBees):
        employedBees.add(problem.randomState())
```

Flow Chart



start
initializing timeToLive, num of employed bees, maxIteration
declaring clusters and jobs
employed bees find their solutions randomly with initialStates method
find best solution
employed bees find their new solutions mutantly. some of them transform to scout bees because of timeToLive value
scout bees find their solutions randomly and transform to employed bees
find best solution
iteration ++
iteration > max ?
No
Yes
print the best founded answer
end

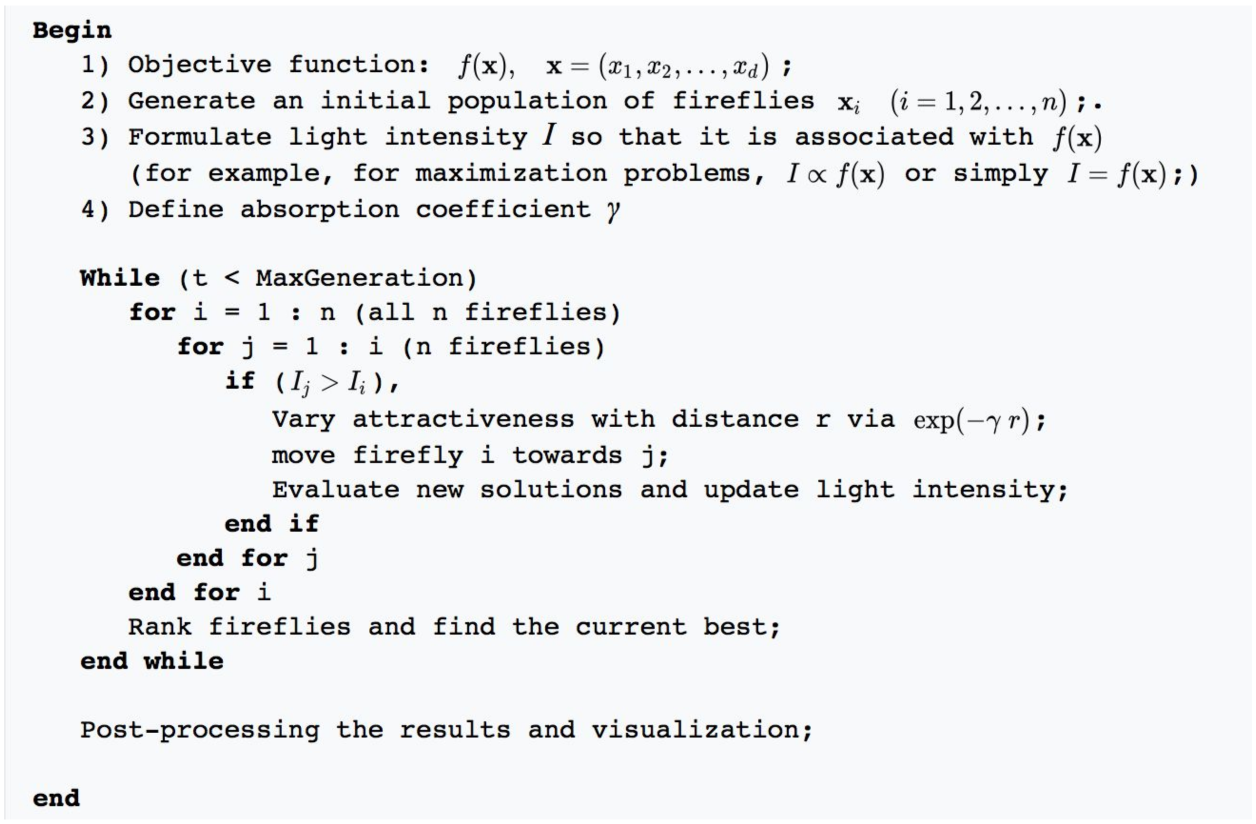
3. FireFly

The primary purpose for a firefly's flash is to act as a signal system to attract other fireflies. Xin-She Yang formulated this firefly algorithm by assuming:

1. All fireflies are unisexual, so that any individual firefly will be attracted to all other fireflies;
2. Attractiveness is proportional to their brightness, and for any two fireflies, the less bright one will be attracted by (and thus move towards) the brighter one; however, the intensity (apparent brightness) decrease as their mutual distance increases;
3. If there are no fireflies brighter than a given firefly, it will move randomly.

The brightness should be associated with the objective function.

Main Steps of algorithm :



```
Begin
   1) Objective function:  $f(\mathbf{x})$,   $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ ;
   2) Generate an initial population of fireflies $\mathbf{x}_i$  $(i = 1, 2, \ldots, n)$ ;.
   3) Formulate light intensity $I$ so that it is associated with $f(\mathbf{x})$
      (for example, for maximization problems, $I \propto f(\mathbf{x})$ or simply $I = f(\mathbf{x})$;)
   4) Define absorption coefficient $\gamma$

   While (t < MaxGeneration)
      for i = 1 : n (all n fireflies)
         for j = 1 : i (n fireflies)
            if ($I_j > I_i$),
               Vary attractiveness with distance r via $\exp(-\gamma\, r)$;
               move firefly i towards j;
               Evaluate new solutions and update light intensity;
            end if
         end for j
      end for i
      Rank fireflies and find the current best;
   end while

   Post-processing the results and visualization;

end
```

Firefly Optimization for Job Scheduling Problem

Methods in Problem class:

f() : gives an state(solution) as an argument and give a number to its value.

equal : check the equality of 2 solution.

initialStates() : returns n random solution. Use for initializing fireflies.

assign() : receives a solution and an action to assign job i to cluster clusterIndex.

If it is possible do it , otherwise do another possible assigning.

Methods in State class:

moveFirefly() : receives a solution. And try to approach current solution to that solution with this formula. For each job if both of the them assigned to the same cluster, new solution assigns :

90% -> to that cluster

10% -> random

If didn't assign the same cluster :

30% -> to cluster of first solution

35% -> to cluster of second solution

25% -> random

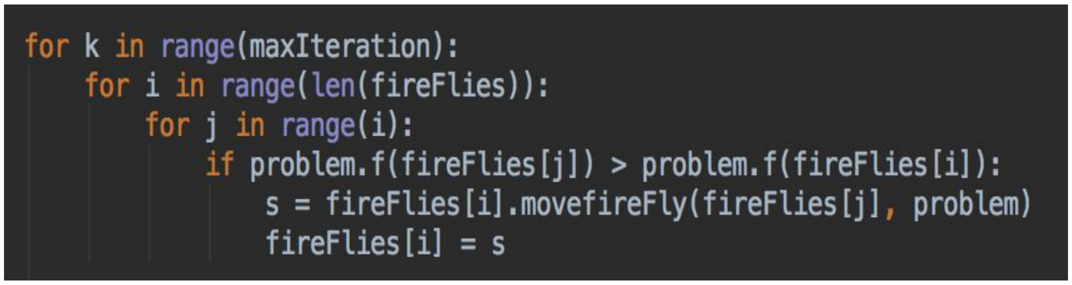
Search Class :

First of all, all fireflies initiates randomly. Then we have a big loop that each firefly try to go near to a better firefly according to moveFirefly method. Then we should find the best solution until yet.

Important Variables:

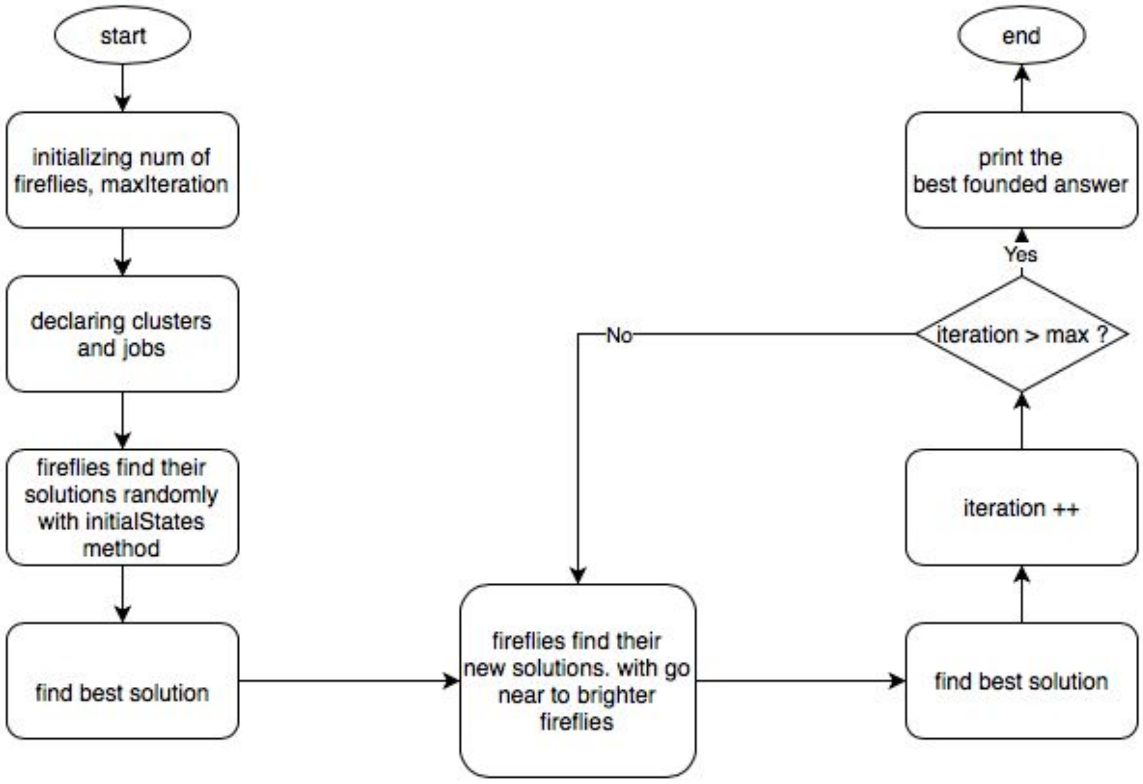
numOfFireflies :	number of fireflies.
fireflies :	an array that keeps my solutions that gained from fireflies
Clusters	an array that keeps informations of all clusters that i have
Jobs	an array that keeps informations of all request that i have

Main part of Code :



```
for k in range(maxIteration):
    for i in range(len(fireFlies)):
        for j in range(i):
            if problem.f(fireFlies[j]) > problem.f(fireFlies[i]):
                s = fireFlies[i].movefireFly(fireFlies[j], problem)
                fireFlies[i] = s
```

Flow Chart


start
initializing num of fireflies, maxIteration
declaring clusters and jobs
fireflies find their solutions randomly with initialStates method
find best solution
fireflies find their new solutions. with go near to brighter fireflies
find best solution
iteration ++
iteration > max ?
No
Yes
print the best founded answer
end

4. Tabu Search

Tabu search is an optimization algorithm that is most likely hill climbing algorithm. But it has some efficient points that almost result better than hill climbing algorithm. For example we don't have to get stuck in local maximum states. In hill climbing optimization we have a current state and some neighbours. Each neighbour has an fitness value, we search and choose the best neighbour that have the maximum fitness value. But when all neighbours don't have value more than current state algorithm stop and return current state as the best possible solution. So it is possible that the best answer be elsewhere.

Tabu search try to fix this problem and make hill climbing more efficient with some rules:

1) It keeps a limited close list to doesn't go to seen states.
2) If in solution 1 bit is changed in t cycle before we don't have permission to change it.
3) If all neighbour states are worse than current state and we can go to a better state than the best state that we've seen until yet allow it, otherwise we go to the best neighbour even it's worse than current.
4) Add a frequency base method to algorithm that means each bit of solution that has changed more has lower probability to choose to change. So the fitness function update to this :

$$f'(state) = f(state) - k * frequency[b_n]$$

$k \textit{ is a double number between } 0\ ,\ 1$

$\textit{frequency is an array to keep how much each bit has changed}$

$\textit{now we use } f' \textit{ instead of } f \textit{ to compare states.}$

Tabu Search for Job Scheduling Problem

In this implementation i added a new class named action. Each action means a change in solution that a job assign to another cluster.

Methods in Problem class:

initialState() : generates a random solution for first state

f() : gives an state(solution) as an argument and give a number to its value.

equa() : check the equality of 2 solution.

actions() : returns all possible actions for recieved state

result() : gives an action and a state and shows the result that action apply to that state

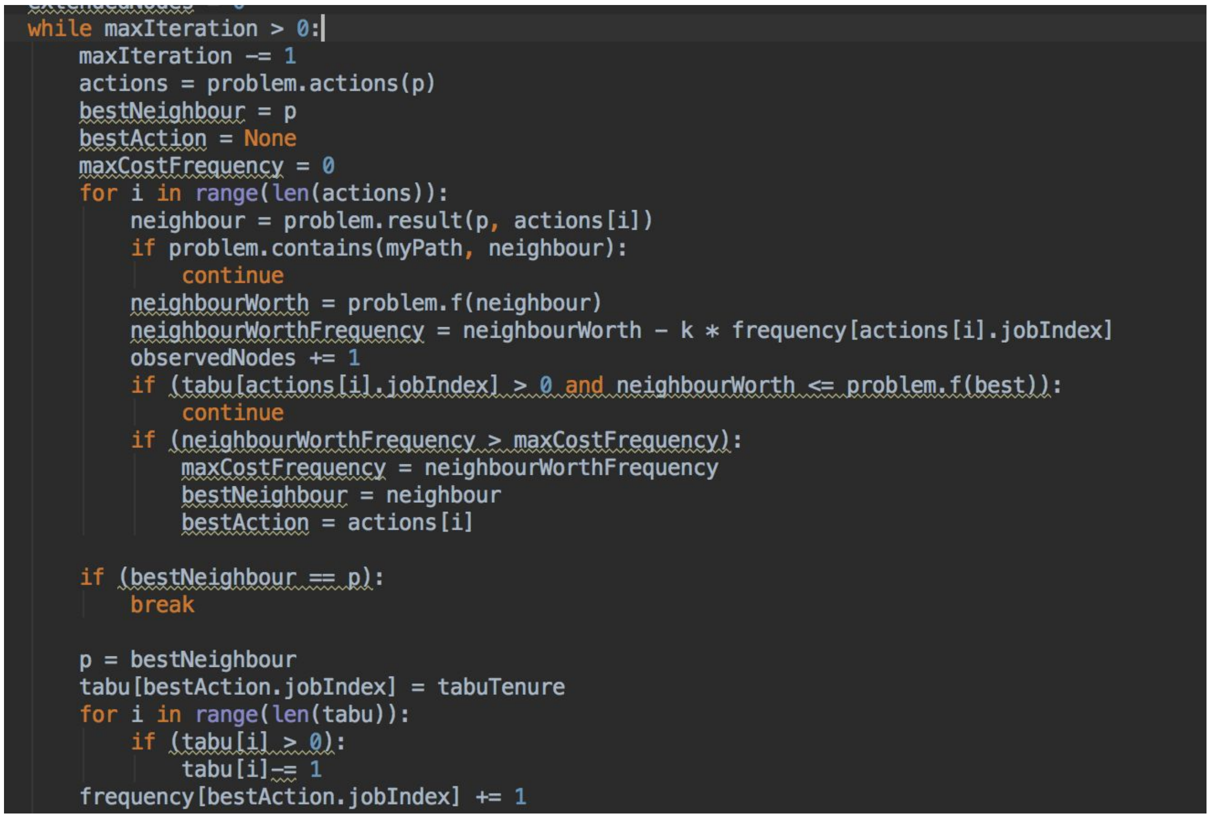
Search class :

Find current state with initialState method, then declaring 2 array named tabu and frequency. Tabu array shows that can we change that job or not and frequency array keeps the value of changed each bit in running time of algorithm. Then we enter a big while and repeat it maxIteration times. Compute actions of current state and then compute the neighbours. If we have this neighbour in our close list ignore it (rule 1), then we compute fitness value of neighbour and after that compute fitness value with frequency effect (rule 4). If we don’t allow to use this neighbour according to tabu array and it’s worse than the best state ignore it. Then we choose the best neighbour we have. Update tabu array so that tabu[i] set to tabuTenure value when i is the jobIndex of chosen neighbour and then increase all other cells of tabu array. Also increase frequency[i] when i is the job index of chose neighbour.

Important Variables:

myPath :	saves all states that we pass to reach the goal.
myActions :	saves all actions that we do to reach the goal.
Clusters	an array that keeps informations of all clusters that i have
Jobs	an array that keeps informations of all request that i have
tabu	explained
Frequency	explained
tabuTenure	tabu array fills with tabuTenure value

Main part of code :

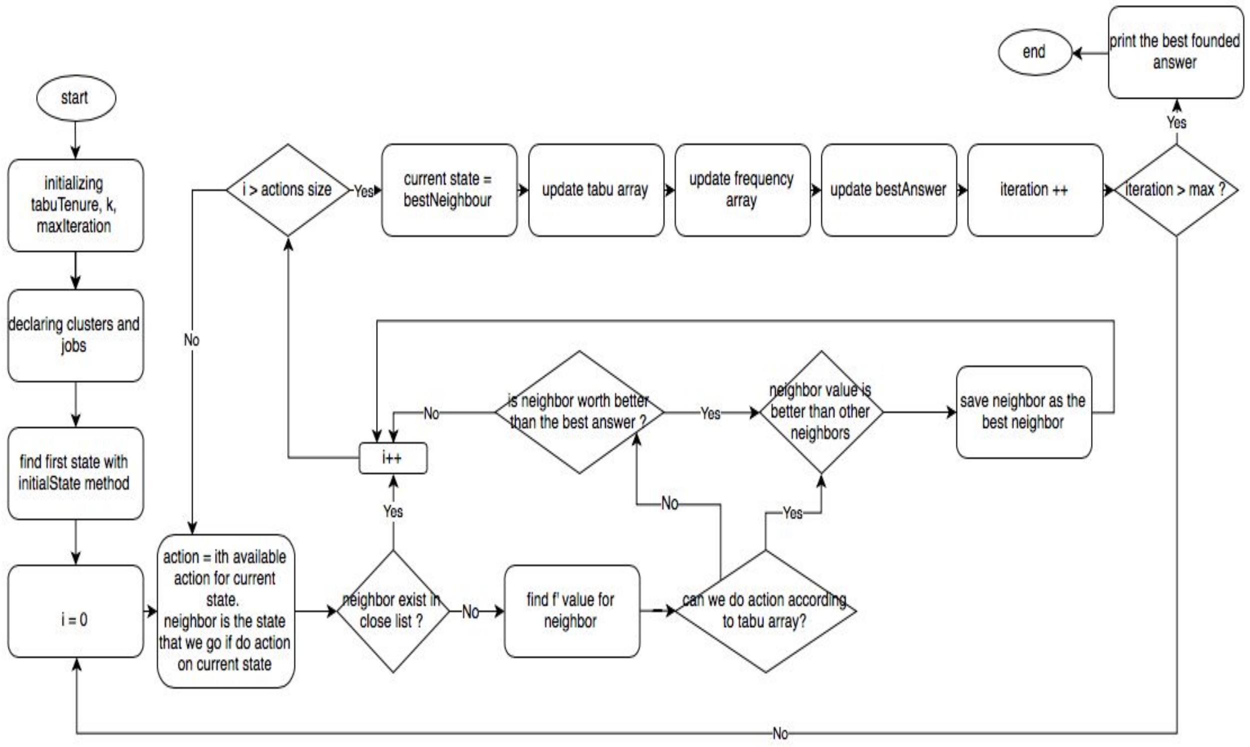


```
while maxIteration > 0:
    maxIteration -= 1
    actions = problem.actions(p)
    bestNeighbour = p
    bestAction = None
    maxCostFrequency = 0
    for i in range(len(actions)):
        neighbour = problem.result(p, actions[i])
        if problem.contains(myPath, neighbour):
            continue
        neighbourWorth = problem.f(neighbour)
        neighbourWorthFrequency = neighbourWorth - k * frequency[actions[i].jobIndex]
        observedNodes += 1
        if (tabu[actions[i].jobIndex] > 0 and neighbourWorth <= problem.f(best)):
            continue
        if (neighbourWorthFrequency > maxCostFrequency):
            maxCostFrequency = neighbourWorthFrequency
            bestNeighbour = neighbour
            bestAction = actions[i]

    if (bestNeighbour == p):
        break

    p = bestNeighbour
    tabu[bestAction.jobIndex] = tabuTenure
    for i in range(len(tabu)):
        if (tabu[i] > 0):
            tabu[i]-= 1
    frequency[bestAction.jobIndex] += 1
```

Flow Chart



start
initializing tabuTenure, k, maxIteration
declaring clusters and jobs
find first state with initialState method
i = 0
action = ith available action for current state. neighbor is the state that we go if do action on current state
neighbor exist in close list ?
Yes
No
i++
find f' value for neighbor
can we do action according to tabu array?
No
Yes
is neighbor worth better than the best answer ?
No
Yes
neighbor value is better than other neighbors
save neighbor as the best neighbor
i > actions size
No
Yes
current state = bestNeighbour
update tabu array
update frequency array
update bestAnswer
iteration ++
iteration > max ?
Yes
No
print the best founded answer
end