



Report
IMPLEMENTATION OF
INTELLIGENT WATER DROPS ALGORITHM

By,
Team- ISI
(Iqra Siddiqui and Sara Intikhab)

Habib University

CS – 102

Section: 05

Contents

S.NO.	TOPIC	PAGE NO.
1	Abstract`	3
2	Introduction	4
3	Algorithm Description	5-9
4	Algorithm Flowchart	10
5	Algorithm Applications	11
6	Implementation Details	12-16
7	Validation of Output	17-19
8	Theoretical Complexity Analysis	20-22
9	Performance Evaluation	23- 25
10	Experimental Analysis	26- 30
11	Theoretical Analysis VS Experimental Analysis	31
12	Further Room for Optimization	32- 33
13	Conclusion	33
14	References	34

Abstract

The nature provides signs of inspiration for developing new intelligent systems. Intelligent Water Drops Algorithm is a product of this inspiration. It is a swarm based algorithm which is efficient in solving combinatorial and optimization problems. The algorithm uses number of intelligent water drops (IWD) which develops their solution incrementally and the best quality solution is chosen after all iterations. This quality is decided on the basis of soil and velocity that an IWD carries/ removes. After enough iterations of the IWD algorithm, the IWDs find the good paths that are decoded to good solutions of the problem.

This report provides an analysis on the python implementation of this algorithm and concludes the findings obtained while implementing it. These finding includes the algorithm complexities along with its time and space efficiency. We have also provided the algorithm essentials in this report including its flowchart, performance evaluations and its applications. This report also contains implementation details which will serve to make the IWD algorithm implementation easy to understand in python. Additionally, python implementation has been done using simpler data structures and taking consideration of weighted as well as un-weighted graph as input.

IWD Algorithm is an algorithm that can solve complex problems with great efficiency and can be modified easily according to the problem. This implementation is a minute contribution in opening room for more modifications and advancements in this algorithm and other similar nature inspired algorithms that can revolutionized the age of computing in near future.

Introduction

Intelligent Water Drops (IWD) Algorithm is a Swarm- based optimization algorithm which is proposed by Shah Hosseini in 2007 proceedings of CEC i.e. Congress on Evolutionary Computation (Camacho-Villalón, Dorigo , & Stützle , 2019).

This algorithm is inspired by the natural swarms that exist in nature. Examples of such swarms can be ant colonies, bee colonies, rivers, etc. Intelligent Water Drops (IWD) is a way to compute a way finding technique within a swarm. This way finding technique is based on the dynamic actions of a river system and the reactions that happen within each droplet for it to find the optimum path. “The solutions are incrementally constructed by the IWD algorithm. Therefore, the IWD algorithm is a population-based constructive optimization algorithm” (Hosseini, 2009). This algorithm has been used to counter problems like knapsack problem or travelling salesman problem.

IWD Algorithm is inspired by the flow of Natural River. A natural river often finds good paths among lots of possible paths in its ways from the source to destination. These near optimal or optimal paths are obtained by the actions and reactions that occur among the water drops and the water drops with the riverbeds (Shah-Hosseini). Each water droplet in the river goes through its own obstacles to reach a particular ocean, lake or sea (destination). These obstacles vary from gravitational force of the earth, to the density of soil, and to the velocity of the droplet itself. This algorithm follows the same process and the optimum path is selected on the basis of soil and velocity of the water droplet. In the coming sections, we will determine how optimum path is taken and why it is the best path.

Algorithm Description

The IWD algorithm is a step in the direction to model a few actions that happen in natural rivers and then to implement them in a form of an algorithm. In the IWD algorithm, IWDs are created with two main properties:

- Velocity
- Soil

A path with less soil lets the IWD become faster than a path with more soil in its route from source to destination. Therefore, paths with lower soils have higher chance to be selected by the IWD. (Shah-Hosseini)

The problem is expressed in the form of an undirected graph (N, E) where N is the nodes and E is the edges. The graph represent the environment for every IWD and they are spread randomly on the nodes of the graph.

Each IWD begins constructing its solution gradually by travelling on the nodes of the graph along the edges until an IWD finally completes its solution.

One iteration of the algorithm is complete when all IWDs have completed their solutions.

After each iteration, the iteration-best solution TIB is found and it is used to update the total-best solution TTB.

The amount of soil on the edges of the iteration-best solution TIB is reduced based on the goodness (quality) of the solution. Then, the algorithm begins another iteration with new IWDs but with the same soils on the paths of the graph and the whole process is repeated.

The algorithm stops when it reaches the maximum number of iterations or the total-best solution TTB reaches the expected quality (Shah-Hosseini).

The IWD algorithm is specified in the following steps: (Shah-Hosseini)

1. *Initialisation of static parameters.* The graph (N, E) of the problem is given to the algorithm. The quality of the total-best solution TTB is initially set to the worst value: $q(TTB) = -\infty$. The maximum number of iterations *itermax* is specified by the user. The iteration count *itercount* is set to zero. The number of water drops *NIWD* is set to a positive integer value, which is usually set to the number of nodes N_c of the graph. For velocity updating, the parameters are $1\ av = 0.01$, $bv = 1$ and $cv = 1$. For soil updating, $1\ as = 0.01$, $bs = 1$ and $cs = 1$. The local soil updating parameter ρ_n , which is a small positive number less than one, is set as $\rho_n = 0.9$. The global soil updating parameter ρ_{iwd} which is chosen from $[0, 1]$, is set as $\rho_{iwd} = 0.9$. Moreover, the initial soil on each path (edge) is denoted by the constant *InitSoil* such that the soil of the path between every two nodes i and j is set by $soil(i,j) = InitSoil$. The initial velocity of each IWD is set to *InitVel*. Both parameters *InitSoil* and *InitVel* are user selected and they should be tuned experimentally for the application. Here, $InitSoil = 10000$ and $InitVel = 200$.
2. *Initialisation of dynamic parameters.* Every IWD has a visited node list $V_c(IWD)$, which is initially empty: $V_c(IWD) = \{\}$. Each IWD's velocity is set to *InitVel*. All IWDs are set to have zero amount of soil.
3. Spread the IWDs randomly on the nodes of the graph as their first visited nodes.
4. Update the visited node list of each IWD to include the nodes just visited

5. Repeat Steps 5.1 to 5.4 for those IWDs with partial solutions.

5.1. For the IWD residing in node i , choose the next node j , which does not violate any constraints of the problem and is not in the visited node list $()_{vc}$ IWD of the IWD, using the following probability $pi(j)$:

$$pi(j) = \frac{f(soil(i, j))}{\sum_{k \neq vc(IWD)} f(soil(i, k))}$$

Such that,

$$f(soil(i, j)) = \frac{1}{\epsilon + g(soil(i, j))}$$

And

$$g(soil(i, j)) = \begin{cases} soil(i, j) & \text{if } \min_{l \in vc(IWD)} (soil(i, l)) \geq 0 \\ soil(i, j) - \min_{l \in vc(IWD)} (soil(i, l)) & \text{else} \end{cases}$$

Then, add newly visited node j to the list $Vc(IWD)$

5.2. For each IWD moving from node i to node j , update its velocity $vel(t)$ by:

$$vel(t + 1) = vel(t) + \frac{av}{bv + cv * soil^2(i, j)}$$

Where $vel(t+1)$ is the updated velocity of the IWD

5.3. For the IWD moving on the path from node i to j , compute the soil $\Delta\text{soil}(i,j)$ that the IWD loads from the path by:

$$\Delta\text{soil}(i,j) = \frac{as}{bs + cs * \text{time}^2(i,j; \text{vel}(t+1))}$$

Where,

$$\text{time}(i,j; \text{vel}(t+1)) = \frac{HUD(j)}{\text{vel}(t+1)}$$

And the heuristic undesirability $HUD(j)$ is defined appropriately for the given problem.

5.4. Update the soil $\text{soil}(i,j)$ of the path from node i to j traversed by that IWD and also update the soil that the IWD carries soilIWD by:

$$\begin{aligned}\text{soil}(i,j) &= (1 - pn) * \text{soil}(i,j) - pn * \Delta\text{soil}(i,j) \\ \text{soilIWD} &= \text{soilIWD} + \Delta\text{soil}(i,j)\end{aligned}$$

6. Find the iteration-best solution TIB from all the solutions TIWD found by the IWDs using

$$TIB = \arg \max_{\forall TIWD} (q(TIWD))$$

Where, function $q(.)$ gives the quality of the solution.

7. Update the soils on the paths that form the current iteration-best solution TIB by

$$\text{soil}(i,j) = (1 + piwd) * \text{soil}(i,j) - piwd * \frac{1}{NIB - 1} * \text{soilIWD} \quad \forall (i,j) \in TIB$$

Where, NIB is the number of nodes in the solution TIB.

8. Update the total best solution TTB by the current iteration-best solution TIB using

$$TTB = TTB \quad \text{if } q(TTB) \geq q(TIB)$$

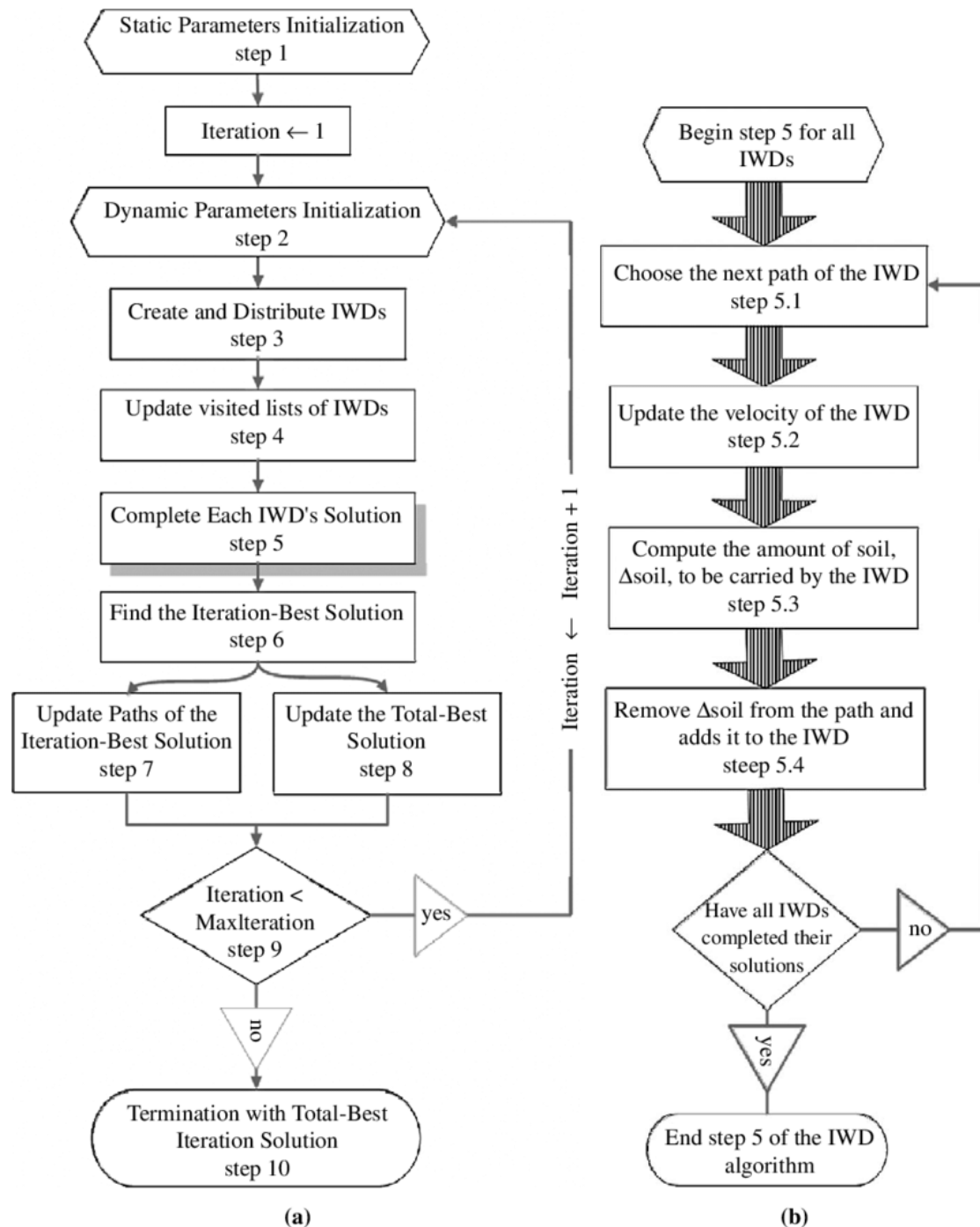
$$TTB = TIB \quad \text{otherwise}$$

9. Increment the iteration number by

$$itercount = itercount + 1 \quad \text{if } itercount < itermax, \text{ then goto Step 2}$$

10. The algorithm stops here with the total-best solution TTB.

Flowchart of IWD Algorithm



“

The flowchart of the proposed IWD algorithm

(a) The flowchart of the main steps of the IWD algorithm

(b) A detailed flowchart of the sub-steps of step 5 of the IWD algorithm' (Shah-Hosseini, 2008).

Algorithm Application

IWD algorithm has been successfully to perform combinatorial and function optimization problems (O Alijla, Wong, Lim, Khader, & Betar, 2014).

For example:

- Travelling Salesman problem
- Multiple knapsack
- Rough set feature subset selection
- Optimum Reservoir operation (Sarani & Dariane, 2013)
- Workflow scheduling problem in Cloud Computing (Kalra & Singh, 2017)

Implementation Details

Summary	
Helper Functions	Description
initializedIWD()	It takes the number of iwd, initial soil and initial velocity and returns a tuple iwd visited node list, the amount of soil it contains and its velocity. $O(\text{Number of Iwds})$
g_soil()	It takes the visited node list of the iwd in process, the node on which this iwd is currently on; i, the node whose probability is being calculated; j and the amount of soil which exist on the path from i to j and returns a float value. $O(\text{number of nodes in graph} * \text{length of visited list of iwd})$ i.e. $O(V * \text{len(visited)})$
f_soil()	It takes the same parameter as g_soil(), uses g_soil() in its calculation and returns a float value $O(O(g_soil))$
probabilityJ()	It takes the same parameter as f_soil(), uses f_soil() in its calculation and returns a float value. $O((f_soil)*V)$
HUD()	It takes the node on which iwd is currently on; i, the next node; j and the amount of soil which exist on the path from i to j and returns a float value. $O(1)$
time()	It takes the square of HUD(), iwd is currently on; i, the next node; j and the velocity of iwd as it traverse on the path from i to j and returns a float value. $O(1)$
q()	It takes the visited node list of iwd and amount of soil on path as parameter and returns a float value. $O(\text{len(visited)})$
Others	
soil	Data structure: dictionary with key as node, value as a dictionary with key as node having edge with that node and value as soil on this edge.
weight	Data structure: dictionary with key as node, value as a dictionary with key as node having edge with that node and value as edge weight.
soiliwd	Data structure: dictionary with key as iwd, value as its soil.
veliwd	Data structure: dictionary with key as iwd, value as its velocity.
visitiwd	Data structure: dictionary with key as iwd, value as list having all the visited nodes of the iwd.
probability	Data structure: dictionary with key as node j, value as its probability of being selected as the next node
quality	Data structure: list
Ttb	Data structure: list
Output	
result	Data Structure: list containing Ttb and its quality as elements

The IWD algorithm takes a graph $G(N, E)$ as input in order to find the optimal solution. In this implementation, we have used the adjacency list/ adjacency map representation of graph. IWD works on an undirected graph. It may be weighted or un-weighted. We have provided two separate codes for the un-weighted, undirected graph and weighted, undirected graph in order to highlight the minute difference that weights can have on implementation. The two functions are held separate and named *iwd()* for the unweighted graph and *iwd_weighted()* for the weighted graph respectively.

Tour implementation is divided into ten steps. These steps are same as in the IWD algorithm.

In the first step, we initialize the static parameters of our algorithm. The data structures which has been used for static parameters are integers. Since IWD algorithm uses a lot of static parameters i.e. constants therefore instead of storing it in a variable, we have directly used their values and has provided the details as comment in the implementation, in order to reduce the space complexity of the code.

In the second, third and fourth steps, we are basically initializing the dynamic parameters which includes the iwd visited node list, the amount of soil it contains and its velocity. In case of the weighted graph, we have a nested dictionary named *weight* that holds key as the node of the graph and its value is a dictionary whose keys are the nodes having an edge with that node and value as the edge weight. For the sake of simplification and ease of understanding we have made another function to perform step two, three and four with the name *initializeIWD()* that takes the number of iwds, initial soil and initial velocity which are user selected for every problem. This function returns a tuple iwd visited node list, the amount of soil it contains and its velocity which is later unpacked. The data structure used for visited node is a list, for soil and velocity it is a dictionary in which the key is the iwd and its value is soil in soil dictionary and velocity in velocity dictionary.

In next step: Step five, we will be performing the mathematic calculations and computations to generate the quality of the iwd whose solution is currently in process. It is divided into 4 parts for ease and reduction of complexity. In the first part of step five, we select a node which is to be visited next and that is also not in the visited list. For this purpose we calculate the probability of all the nodes that can be visited from i using another function *probabilityJ()*. This function takes the visited node list of the iwd in process, the node on which this iwd is currently on; i, the node whose probability is being calculated; j and the amount of soil which exist on the path from i to j. The function calls another function *f_soil()* which takes the same parameters as *probabilityJ()*. This function calls another function *g_soil()* as per the formulae provided in the IWD algorithm. This function evaluates and returns the amount of soil on path from i to j which is later used to generate the value of *f_soil()* and later the probability of j. All the formulae used in this calculations are provided in the IWD algorithm and we have just made the python analogue of the same formulae to generate results. After all this, the function *probabilityJ()* returns a float value of probability which is later being stored in a dictionary named *probability* with key as j.

Moreover the selected j (the next node to visit) should satisfy the constraints of the problem which is being solved using the algorithm along with having maximum probability. In this implementation, we have taken a dummy constraint that j should satisfy i.e. j should have probability sum less than a random number to be ideal for being selected.

After j has been selected, we update the velocity of the iwd as it moves from the current node to j which is the next part of Step 5. Similarly in the third part of Step 5, we update the amount of soil that the iwd has carried while traversing from i to j. This carried soil amount (*ds*) is calculated using the formula provided in algorithm. This formula along with some static parameters, uses time and HUD. The time is simply the amount of traversal time of the path. In order to calculate this we have made another function *time()*. This function takes the current

node i , next node j , the velocity of the iwd which is traversing on this path, and the heuristic undesirability of this path and returns a float value which is the traversal time of the path. Heuristic Undesirability (HUD) varies from problem to problem. For example, in case of solving TSP, the HUD is the distance between cities. We have store *HUD* in another function to make it prominent and equal to the amount of soil of this path. As greater the amount of soil on path, less preferred it is by the iwd. In case of weighted graph, this *HUD* equals the edge weight of that path. After all this, we have an update ds which is a value of float data type.

In the last part of step 5, we update the soil of the path traversed using the formula on the basis of the current soil and the soil carried by the iwd (ds). The soil which iwd currently have, after the traversal is also updated. Using all these factors, we find the quality of the solution path that this iwd has been successful to traverse. In order to calculate the quality we have made another function $q()$ that takes visited node list of the iwd and the soil of the path. The function finds the quality on the basis of the formula provided in algorithm and returns a float value. This float value is appended in a list named *quality* which is made to keep track of the quality of each iwd solution which will be used later to find the best solution.

The sixth step of our algorithm chooses the iteration-best solution which is basically the iwd that has provided the maximum quality. Since we have taken iwds equal to the number of nodes in graph for simplicity, thus the index of the quality is basically the iwd number whose quality is at that index. Using this fact, we find the best iwd which has provided the maximum quality solution.

In Step seven, we update the soil of the iteration best solution path. The amount of soil between the node on which the best iwd has initially reside on i.e. i and the node on which the best iwd is currently residing i.e. j .

In step eight we update the best solution with the current best solution. The best solution is held in list named *Ttb*. This update is done on the basis of conditions provided in the algorithm.

In step nine, we go back to step two of the algorithm which is starting from initialization of dynamic parameters. These steps are repeated until the maximum iteration count is reached. After the maximum iterations has been achieved, the *Ttb* contains the best solution and algorithm returns it.

The output can be provided on the basis of the problem and what it demands. As after running the algorithm we have the total best path, the quality of the path and the iwd which has traced this optimal path. In our implementation we have provided the output in the form of a list. The first element of this list is the total best path and the second element of list is the quality of this path.

Validation of Output

In order to verify that the output of our implementation is correct, we used the python built in library of Intelligent Water Drop algorithm. This library is named as IWD and has two modules. Module 1 is the “*iwd*” which is a simple implementation of the weighted graph input iwd algorithm. Module 2 is “*miwd*” which is the advancement/ modification of *iwd*.

We use a weighted graph as input and generated output by using the IWD library as well as by using our implementation of the algorithm.

- **Using Python Library**

The input in this case was a distance matrix which is a weighted edge list representation of a graph. And the number of nodes in the graph.

For validation, we used these input values:

```
distance_matrix= {0: {0: 0,1: 34,2: 56,3: 79},1: {0: 45,1: 0,2: 13,3: 77},2: {0: 89,1: 56,2: 0,3: 75},3: {0: 46,1: 48,2: 31,3: 0}}
```

```
number_of_nodes= 4
```

According to the usage documentation of this library, (INTELLIGENT WATER

```
from IWD import iwd
parameters = iwd.parameters_initialization()
distance_matrix={0: {0: 0,1: 34,2: 56,3: 79},1: {0: 45,1: 0,2: 13,3: 77},2: {0: 89,1: 56,2: 0,3: 75},3: {
number_of_nodes=4
parameters.initialize_graph(number_of_nodes, distance_matrix)

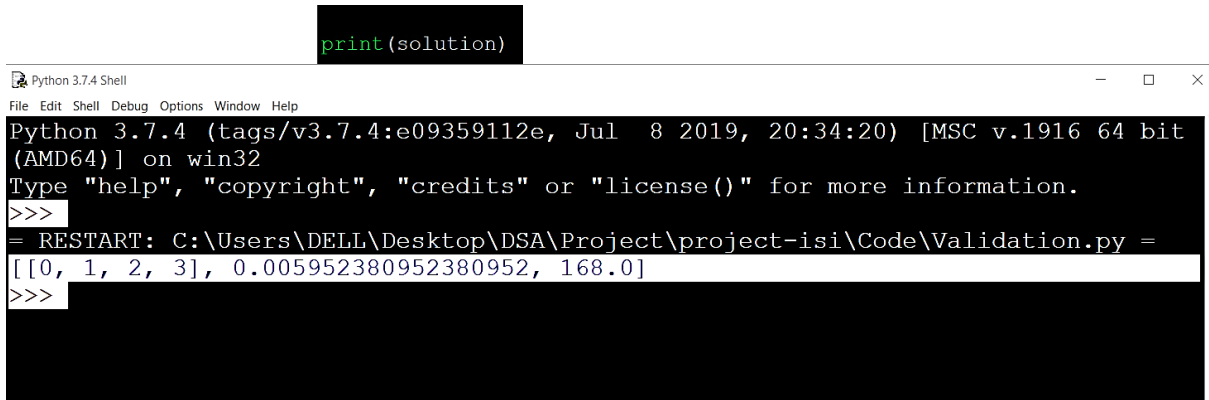
solution = iwd.compute(parameters.parameter_list)
```

DROPS MODULE), we generated the solution on these input values as follow:

On printing the solution

We got,

```
print(solution)
```



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\DELL\Desktop\DSA\Project\project-isi\Code\Validation.py =
[[0, 1, 2, 3], 0.005952380952380952, 168.0]
>>>
```

The first element of this output list is the total best solution, second element is the quality and the third element is gives the ratio of 1 to the quality.

- **Using own Implementation**

We executed our iwd implementation on the same set of inputs.

In this case, the input was the adjacency representation of the same graph as we used for the previous part. This variation was due to the variation in the data structures that our implementation and the library module uses.

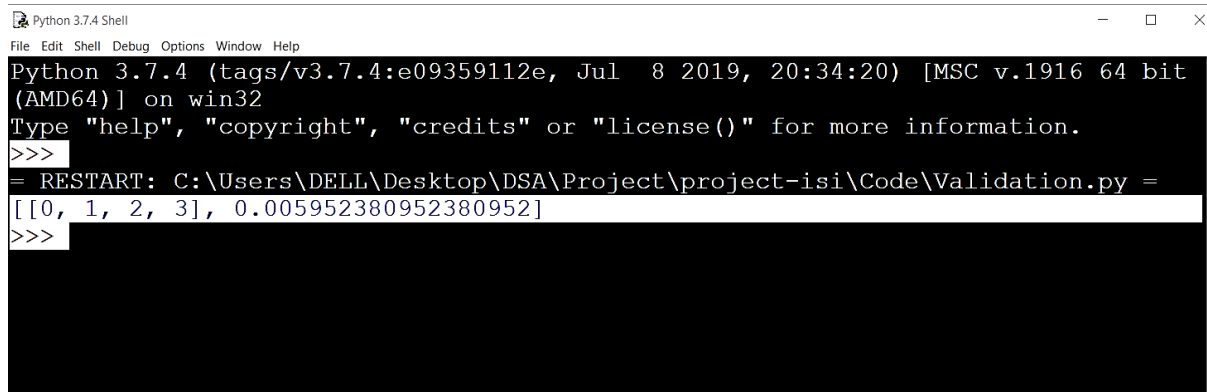
i.e.

```
graph= {0: [(0,0),(1,34),(2,56),(3, 79)],1: [(0,45),(1, 0),(2, 13),(3, 77)],2: [(0, 89),(1, 56),(2, 0),(3, 75)],3: [(0, 46),(1, 48),(2, 31),(3, 0)]}
```

Then we called the function on our graph

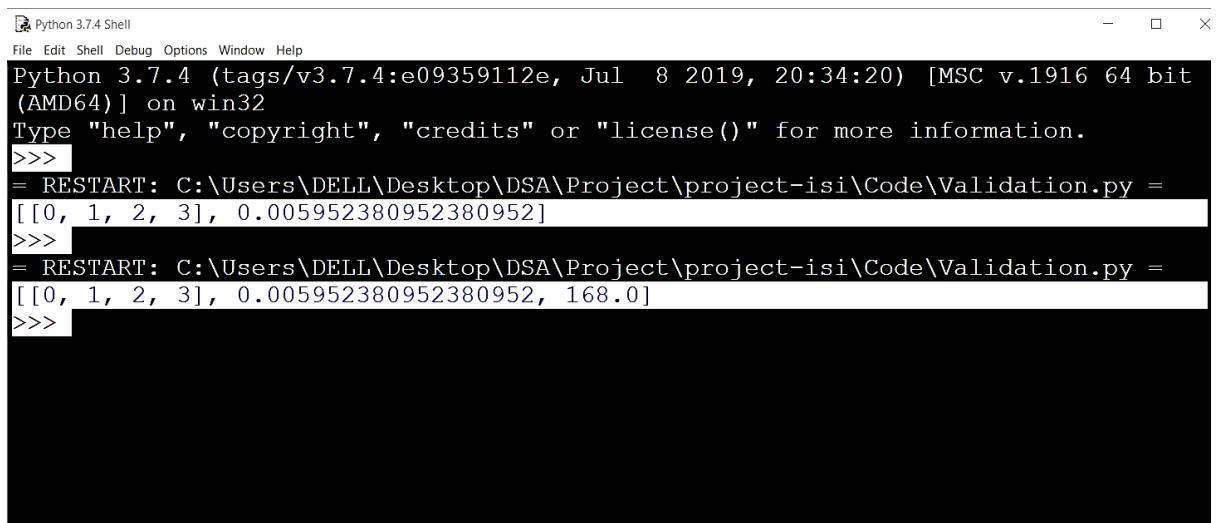
```
graph={0: [(0,0), (1,34), (2,56), (3, 79)],1: [(0,45), (1, 0), (2, 13), (3, 77)],2: [(0, 89), (1, 56), (2, 0), (3,
print(iwd_weighted(graph))
```

This provided the following output:



```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\DELL\Desktop\DSA\Project\project-isi\Code\Validation.py =
[[0, 1, 2, 3], 0.005952380952380952]
>>>
```

Hence, when we compare the outputs from both methods, they were exactly same except for the fact that our output provides just the total best path and quality and does not provide the ratio of 1 to the quality.



```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\DELL\Desktop\DSA\Project\project-isi\Code\Validation.py =
[[0, 1, 2, 3], 0.005952380952380952]
>>>
= RESTART: C:\Users\DELL\Desktop\DSA\Project\project-isi\Code\Validation.py =
[[0, 1, 2, 3], 0.005952380952380952, 168.0]
>>>
```

This validates the output of our implementation.

Theoretical Complexity Analysis of IWD Algorithm

Input: un-weighted graph

(V=number of nodes in graph, E= number of edges in graph, Niwd=number of iwd's)

Step No.	Complexity
1	$O(V^2 * E)$
2	$O(\text{itermax}) * O(\text{initializeIWD}())$
	$O(\text{itermax}) * O(\text{Niwd})$
3	$O(\text{Niwd})$
4	$O(\text{Niwd})$
5	$O(E) * O(\text{Niwd}) * O(\text{itermax})$
6	$O(\text{len(quality)}) * O(\text{itermax})$
7	$O(\text{len(visit)}) * O(\text{itermax})$
8	$O(1) * O(\text{itermax})$
9	$O(1) * O(\text{itermax})$
10	$O(1)$

Assuming $V < E$ and $V = \text{Niwd}$

Total Complexity = $O(E) * O(\text{Niwd}) * O(\text{itermax})$

$O(E * \text{Niwd} * \text{itermax})$ or $O(E * V * \text{itermax})$

For further details, see the File “Complexity Analysis Unweighted Graph.py” on Git -

> Code.

We examined IWD algorithm best case and worst case. It can be concluded that there is no best case for this algorithm as IWD algorithm has the property of convergence and thus provide optimum result when the iterations are sufficiently large (Shah-Hosseini). Therefore, its best case time complexity is same as its worst case time complexity.

Input: weighted graph

(V=number of nodes in graph, E= number of edges in graph, Niwd=number of iwd)

Step No.	Complexity
1	$O(V^2 * E)$
2	$O(\text{itermax}) * O(\text{initializeIWD}())$
	$O(\text{itermax}) * O(\text{Niwd})$
3	$O(\text{Niwd})$
4	$O(\text{Niwd})$
5	$O(E) * O(\text{Niwd}) * O(\text{itermax})$
6	$O(\text{len}(\text{quality})) * O(\text{itermax})$
7	$O(\text{len}(\text{visit})) * O(\text{itermax})$
8	$O(1) * O(\text{itermax})$
9	$O(1) * O(\text{itermax})$
10	$O(1)$

Assuming $V < E$ and $V = \text{Niwd}$

Total Complexity = $O(E) * O(\text{Niwd}) * O(\text{itermax})$

$O(E \cdot N_{wd} \cdot \text{itermax})$ or $O(E \cdot V \cdot \text{itermax})$

For further details, see the File “Complexity Analysis Weighted Graph.py” on Git -> Code.

We examined IWD algorithm best case and worst case. It can be concluded that there is no best case for this algorithm as IWD algorithm has the property of convergence and thus provide optimum result when the iterations are sufficiently large (Shah-Hosseini). Therefore, its best case time complexity is same as its worst case time complexity.

Performance Evaluation

In this section, we wish to analyze the performance of IWD Algorithm. For this purpose we will have to take one of its implementation in account. Here we will consider our own python implementation of IWD Algorithm which can be found in the “Code” Folder on Git.

Let us first assume that we are representing the graph G in the form of an adjacency list representation or adjacency map structure. This data structure makes it easier to update the soil on the paths. We also assume that the edge weights can be manipulated in constant time.

We will begin by evaluating the time complexity of our implementation. (V=number of nodes in graph, E= number of edges in graph, Niwd=number of iwds)

Input: un-weighted graph

Step No.	Run Time Complexity	Space Complexity (words)
1	$O(V^2 * E)$	18
2	$O(\text{itermax}) * O(\text{initializeIWD}())$ $= O(\text{itermax}) * O(\text{Niwd})$	3
3	$O(\text{Niwd})$	
4	$O(\text{Niwd})$	
5	$O(V) * O(E) * O(\text{Niwd}) * O(\text{itermax})$	10
6	$O(\text{len}(\text{quality})) * O(\text{itermax})$	2
7	$O(\text{len}(\text{visit})) * O(\text{itermax})$	2
8	$O(1) * O(\text{itermax})$	
9	$O(1) * O(\text{itermax})$	1
10	$O(1)$	1

Assuming $V < E$ and $V = Niwd$

$$\begin{aligned}\text{Total Complexity} &= O(V) * O(E) * O(Niwd) * O(\text{itermax}) \\ &= O(V^2 E * \text{itermax})\end{aligned}$$

For further details, see the File “Complexity Analysis Unweighted Graph.py” on Git -
> Code.

In evaluating the space complexity, we will just consider the data space.

$$\text{Total Space Complexity} = 18 + 3 + 10 + 2 + 2 + 1 + 1 = 37 \text{ Words}$$

Input: weighted graph

Step No.	Run Time Complexity	Space Complexity (words)
1	$O(V^2 * E)$	19
2	$O(\text{itermax}) * O(\text{initializeIWD}())$ $= O(\text{itermax}) * O(Niwd)$	3
3	$O(Niwd)$	
4	$O(Niwd)$	
5	$O(V) * O(E) * O(Niwd) * O(\text{itermax})$	10
6	$O(\text{len}(\text{quality})) * O(\text{itermax})$	2
7	$O(\text{len}(\text{visit})) * O(\text{itermax})$	2

8	$O(1) * O(\text{itermax})$	
9	$O(1) * O(\text{itermax})$	1
10	$O(1)$	1

Assuming $V < E$ and $V = Niwd$

$$\begin{aligned} \text{Total Complexity} &= O(V) * O(E) * O(Niwd) * O(\text{itermax}) \\ &= O(V^2 E * \text{itermax}) \end{aligned}$$

For further details, see the File “Complexity Analysis Weighted Graph.py” on Git -> Code.

In evaluating the space complexity, we will just consider the data space.

$$\text{Total Space Complexity} = 18 + 3 + 10 + 2 + 2 + 1 + 1 = 38 \text{ Words}$$

Experimental Analysis of IWD Algorithm

We have used both implementations of IWD i.e. the input with unweighted graph and the weighted graph to analyze the relation of the input size that is the number of nodes in graph and the execution time of code. We used dynamic input for the purpose of analysis. The analysis was done using timeit function of timeit module. The summary of input sizes that were selected in experiment and their time obtained is provided in the table below.

In this experiment the number of iwd's (NIWDS) were taken equivalent to the number of nodes in graph. The input size is the number of nodes in graph. Let it be n . The number of edges in the graph are $n*(n-1)$.

Input: Unweighted graph

S #	Input Size	Time	
		secs	milliseconds
1	8	0.031341	31.3409
2	10	0.048045	48.0448
3	13	0.032159	32.1594
4	16	0.022832	22.8322
5	24	0.023486	23.4862
6	28	0.041764	41.7645
7	26	0.045471	45.4713
8	35	0.030372	30.3721
9	39	0.047076	47.0756
10	34	0.041056	41.0565
11	44	0.030821	30.821
12	45	0.027747	27.7468
13	53	0.078041	78.0411
14	52	0.018221	18.2214
15	53	0.018454	18.454
16	70	0.075677	75.6771
17	80	0.075787	75.7874
18	84	0.03649	36.4899
19	87	0.056093	56.0932
20	93	0.045705	45.7052
21	95	0.062425	62.425
22	96	0.060083	60.0833
23	100	0.035759	35.7595

Figure 1: The table summarizes the input values that were taken randomly and dynamically and their corresponding execution time in seconds and milliseconds obtain from applying timeit.timeit() function.

The results are summarized in the scatterplot shown below.

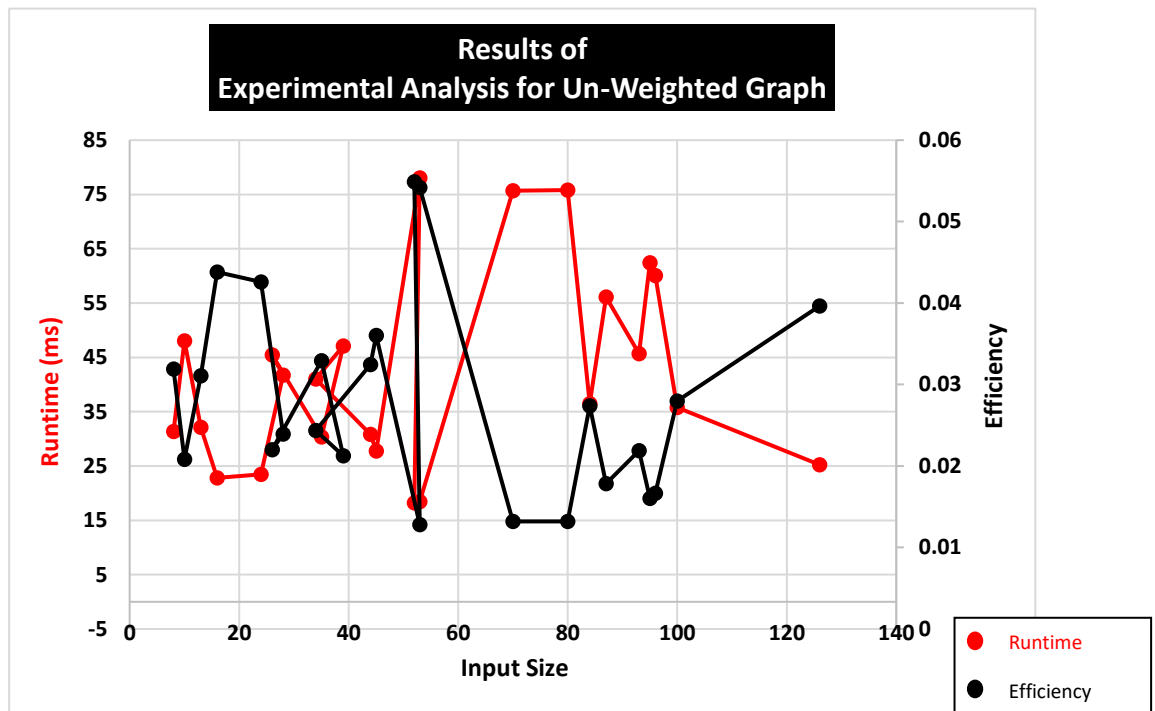


Figure 2: The red plots shows the runtime. A red dot (n,t) represents the runtime of input size n. The black plots shows the efficiency. A black dot (n,e) represents the efficiency of input size n. The input size is the number of nodes in the graph on which we executed the iwd.

We know that efficiency of an algorithm is inversely proportional to its run time. Thus, the efficiency of the input sizes is obtained by $1/t$ where t is the corresponding runtime for that input size.

It can be concluded from the results that (*consider fig. 2*)

- The IWD presented maximum runtime when number of nodes selected were 53 which is 78 ms (*consider red*). Thus minimum efficiency. (*consider black*)
- The IWD presented minimum runtime when number of nodes selected were 52 which is 18.2 ms (*consider red*). Thus maximum efficiency. (*consider black*)

Input: Weighted graph

In this experiment, the weights here are taken as random numbers. Methodology used for dynamic edges and nodes is same as for the unweighted graph.

Weighted graph			
S #	Input Size	Time	
		secs	millisecs
1	13	0.017576	17.5763
2	11	0.05812	58.1201
3	17	0.030089	30.0885
4	28	0.033287	33.2865
5	30	0.068033	68.0331
6	21	0.029913	29.913
7	32	0.156086	156.0856
8	31	0.031217	31.2167
9	47	0.0539	53.8995
10	44	0.050744	50.7444
11	52	0.022428	22.4278
12	54	0.019097	19.0973
13	51	0.053365	53.3654
14	61	0.033652	33.6521
15	73	0.052515	52.5149
16	81	0.079252	79.2523
17	82	0.036319	36.3187
18	89	0.038586	38.5857
19	91	0.071954	71.9536
20	95	0.054746	54.7455
21	92	0.034743	34.743
22	94	0.03396	33.9602
23	106	0.051919	51.9187
24	100	0.056098	56.0977

Figure 3: The table summarizes the input values that were taken randomly and dynamically and their corresponding execution time in seconds and milliseconds obtain from applying `timeit.timeit()` function.

The results are summarized in the scatterplot shown below.

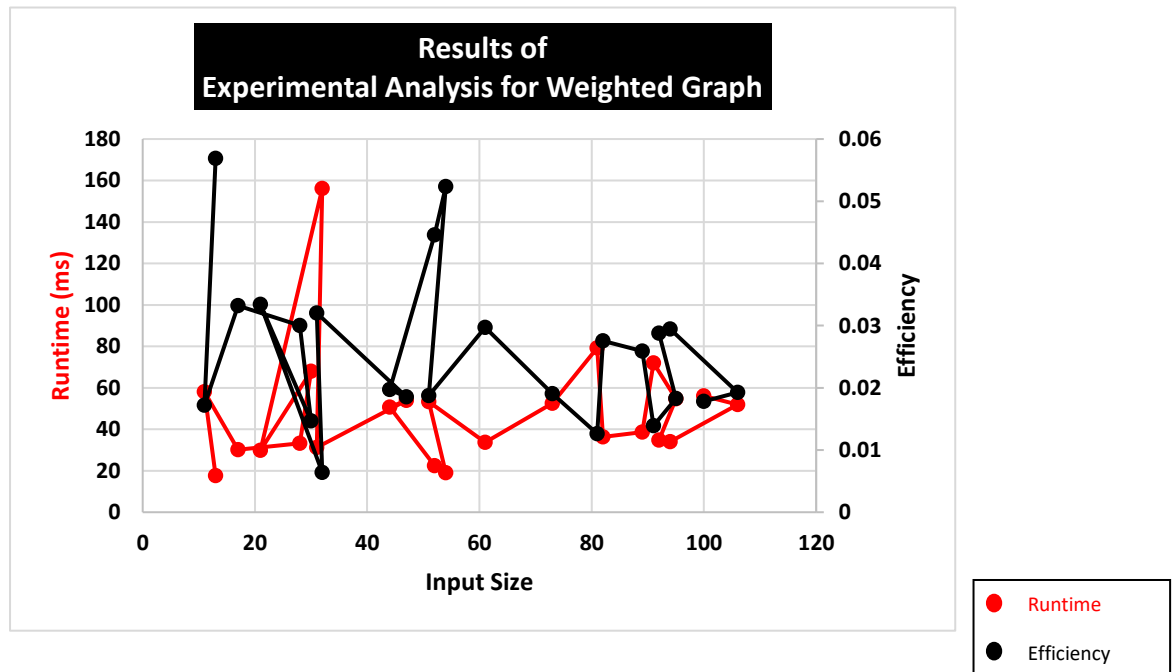


Figure 4: The red plots shows the runtime. A red dot (n,t) represents the runtime of input size n . The black plots shows the efficiency. A black dot (n,e) represents the efficiency of input size n . The input size is the number of nodes in the graph on which we executed the iwd.

We know that efficiency of an algorithm is inversely proportional to its run time. Thus, the efficiency of the input sizes is obtained by $1/t$ where t is the corresponding runtime for that input size.

It can be concluded from the results that (*consider fig. 2*)

- The IWD presented maximum runtime when number of nodes selected were 32 which is 156 ms (*consider red*). Thus minimum efficiency. (*consider black*)

- The IWD presented minimum runtime when number of nodes selected were 13 which is 17 ms (*consider red*). Thus maximum efficiency. (*consider black*)

Theoretical Complexity Analysis VS Experimental Analysis

The result for our theoretical runtime analysis of implementation came out to be:

$$O(V^2E * \text{itermax})$$

We can validate it by comparing to the results of our experimental analysis.

As it can be seen from figure 2 and figure 4, that the runtime was dependent on the input size which is basically the number of nodes in or graph. The runtime, hence the efficiency was varying with variations in our input sizes.

For example, an input size of 13 has a runtime of 17.5 while an input size of 21 has a runtime of 29.9.

It is important note, that the *itermax* was kept constant in this experiment. We were varying the V (input size) and E (i.e. number of edges which in this experiment were varied by a factor $V*(V-1)$)

Therefore, it can said that the complexity is dependent on V or V^2 , and E when the *itermax* is constant.

Thus, it validates our theoretical complexity analysis that our Worst case complexity of this implementation is

$$O(V^2E * \text{itermax})$$

Further Room for Optimization:

As evaluated in the previous section, there is a need to reduce the runtime complexity of our implementation for better and efficient execution. We suggest two ways for optimizing our implementation in this section.

- **Way 1: To optimize the hot spot function**

For this purpose we need to find the hot spot functions of our implementation that can be modified in order to optimize our code runtime. For finding the hot spot functions, we used the *profile* function.

For further details regarding how we have find the hot spot function, see the File “Hot spot function” on Git -> Code.

The results were as follows:

```
1335007 function calls in 9.953 seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
110000	0.438	0.000	0.438	0.000	:0(append)
1	0.000	0.000	9.953	9.953	:0(exec)
1000	0.000	0.000	0.000	0.000	:0(index)
1001	0.016	0.000	0.016	0.000	:0(keys)
11001	0.031	0.000	0.031	0.000	:0(len)
1000	0.000	0.000	0.000	0.000	:0(max)
10000	0.062	0.000	0.062	0.000	:0(random)
1	0.000	0.000	0.000	0.000	:0(setprofile)
1	0.000	0.000	9.953	9.953	<string>:1(<module>)
1	0.859	0.859	9.953	9.953	Hot Spot Function.py:101(IWD)
1000	0.016	0.000	0.047	0.000	Hot Spot Function.py:38(initializeIWD)
540000	2.875	0.000	2.875	0.000	Hot Spot Function.py:54(g_soil)
540000	3.297	0.000	6.172	0.000	Hot Spot Function.py:65(f_soil)
90000	2.281	0.000	8.453	0.000	Hot Spot Function.py:69(probabilityJ)
10000	0.047	0.000	0.047	0.000	Hot Spot Function.py:77(HUD)
10000	0.000	0.000	0.000	0.000	Hot Spot Function.py:81(time)
10000	0.031	0.000	0.062	0.000	Hot Spot Function.py:84(q)
1	0.000	0.000	9.953	9.953	profile:0(IWD(graph))
0	0.000		0.000		profile:0(profiler)

This shows that the *f_soil()* is our hotspot function as it has the maximum total time i.e. 3.267 seconds.

Thus, changes can be made in $f_{soil}()$ function which will eventually reduce the total time of our code.

- **Way 2: To minimize the *itermax***

Another step that can be taken to reduce the overall execution time is to select *itermax* as 10 or 100 instead of 1000. This can work as instead of repeating our entire code for 1000 times, we will be repeating it just for 10 or 100 times which will have an immense impact.

Conclusion

IWD is an optimization algorithm that makes use of water droplets of a river to find the solution to a given problem. The algorithm creates different routes by moving on the undirected graph representation and finds the shortest path amongst the obtained solutions. With each iteration, the IWD gains some velocity and removes some soil from the path it flows on.

Travelling salesman problem, multiple knapsack problem, and the n-queen puzzle are the three problems for which this algorithm has been used till now. There is some difference in the HUD of the algorithm based on different problems but the general outlook is the same.

This algorithm is good for finding optimal solutions of good quality. This algorithm is also representative of the fact that nature is the best teacher for “designing and inventing swarm-based optimization algorithms” (Hosseini, 2009).

References

- Camacho-Villalón, C. L., Dorigo, M., & Stützle, T. (2019). *The intelligent water drops algorithm: why it cannot be considered a novel algorithm*. SpringerLink.
- A Survey on Intelligent Water Drop Algorithm. (2014). *Researchgate*.
- Kalra, M., & Singh, S. (2017). *Application of intelligent water drops algorithm to workflow scheduling in cloud environment*. IEEE.
- O Alijla, B., Wong, L.-P., Lim, C. P., Khader, A. T., & Betar, M. A. (2014). *A modified Intelligent Water Drops algorithm and its application to optimization problems*. ScienceDirect.
- Sarani, S., & Dariane, A. (2013). Application of Intelligent Water Drops Algorithm in Reservoir Operation. *SpringerLink*.
- Shah-Hosseini, H. (2008). Intelligent water drops algorithm. *International Journal Of Intelligent Computing And Cybernetics*, 1(2), 193-212. doi: 10.1108/17563780810874717
- Hosseini, H. (2009). The intelligent water drops algorithm: a nature-inspired swarm-based optimization algorithm. *International Journal Of Bio-Inspired Computation*, 1(1/2), 71. doi: 10.1504/ijbic.2009.022775
- Shah-Hosseini, H. (n.d.). The intelligent water drops algorithm: a nature-inspired swarm-based optimization algorithm . In *Int. J. Bio-Inspired Computation*, . Tehran.
- Shah-Hosseini, H. (2008). Intelligent water drops algorithm. *International Journal Of Intelligent Computing And Cybernetics*, 1(2), 193-212. doi: 10.1108/17563780810874717