**Implementation of:**

**INTELLIGENT WATER DROPS ALGORITHM**

**By,**

**Iqra Siddiqui and Sara Intikhab**

Habib University

CS – 102

Section: 05

|  |  |  |
| --- | --- | --- |
| S.No. | Topic | Page No. |
| 1 | Abstract` | 3 |
| 2 | Introduction | 4 |
| 3 | Algorithm Description | 5-10 |
| 4 | Algorithm Applications | 11 |
| 5 | Implementation Details | 12 |
| 6 | Performance Evaluation | 17 |
| 7 | Conclusion | 18 |
| 8 | References | 19 |

# Contents

# 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 fining 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) = −∞. 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 *Nc* 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 *ρn* , which is a small positive number less than one, is set as *ρn* = 0.9. The global soil updating parameter *ρiwd* which is chosen from [0, 1], is set as *ρiwd* = 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 *Vc (IWD)*, which is initially empty: *Vc(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.
   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)*:

Such that,

****And

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

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

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

* 1. For the IWD moving on the path from node i to j, compute the soil Δsoil(i,j) that the IWD loads from the path by:

Where,

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

* 1. Update the soil *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:

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

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

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

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

1. Update the total best solution TTB by the current iteration-best solution TIB using
2. Increment the iteration number by
3. 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).

# “

**Complexity Analysis of IWD Algorithm**

Input: un-weighted graph

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

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

*Assuming V<E and V=Niwd*

Total Complexity = O(E)\*O(Niwd)\*O(itermax)

O(E\*Niwd\*itermax) or O(E\*V\*itermax)

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

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

# 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)

|  |  |
| --- | --- |
| Summary | |
| Helper Functions | **Description** |
| initializedIWD() | It takes the number of iwds, initial soil and initial velocity and returns a tuple iwd visited node list, the amount of soil it contains and its velocity. *O(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(number of nodes in graph \* length of visited list of iwd) i.e. O(V\*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(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 |

# Implementation Details

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.

# 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(itermax)\*O(*initializeIWD()*)  = O(itermax)\*O(Niwd) | 3 |
| 3 | O(Niwd) |  |
| 4 | O(Niwd) |  |
| 5 | O(V)\*O(E)\*O(Niwd)\*O(itermax) | 10 |
| 6 | O(len(quality))\*O(itermax) | 2 |
| 7 | O(len(visit))\*O(itermax) | 2 |
| 8 | O(1) \*O(itermax) |  |
| 9 | O(1) \*O(itermax) | 1 |
| 10 | O(1) | 1 |

*Assuming V<E and V=Niwd*

Total Complexity = O (V)\*O(E)\*O(Niwd)\*O(itermax)

= O(V2E\* itermax)

*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.

Total Space Complexity = 18 + 3 + 10+ 2+ 2 +1 +1 = 37 Words

# 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