

Advanced Algorithms - Wells Assignment Project

Piotr Krzemiński 305785
Zofia Wrona 305803

1 Problem Description

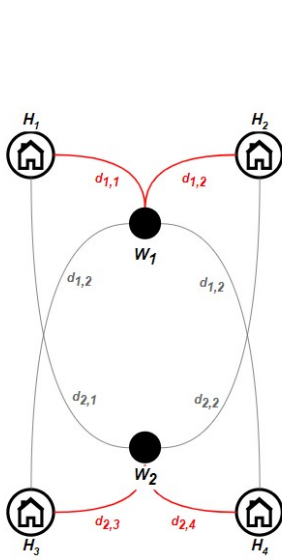
The Wells Problem is a combinatorial optimization problem with applications in fields such as water supply and facility location. The task to be solved has been formulated in the following way:

Design and implement an algorithm for finding the cheapest (total sum of costs) way to provide houses with water. There are given n wells and kn houses (points on a plane), each well provides k houses with water. The cost of providing a house with water from a well is the Euclidean distance between the well and the house.

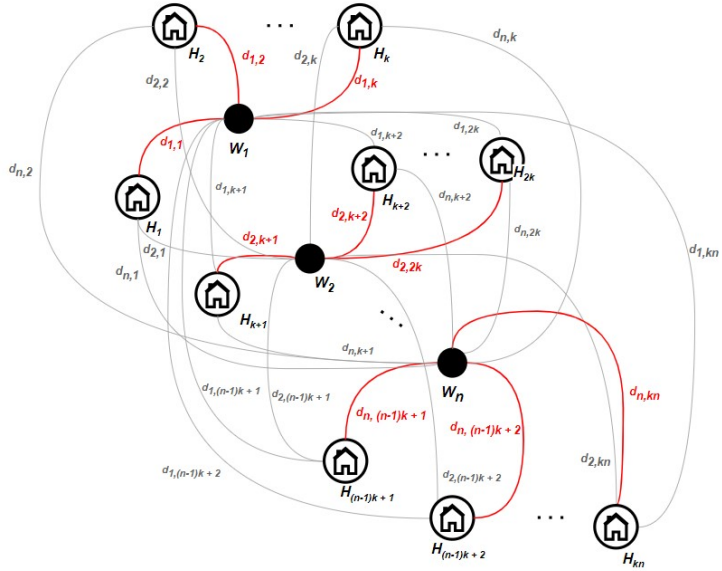
To summarize, the objective of the task is to **minimize the water supply cost** in such a way that each house is connected to **exactly one well** and each well supplies water to **exactly k houses**

1.1 Mathematical Representation

The problem can be viewed as a vertex-labeled graph $G = (V, E)$, where V is a set of vertices representing houses and wells and E is the set of edges defining the connections between each house and each well. The task is to determine a subset of edges (a matching) M such that the sum of the weights of the matched edges is minimal, each well is of degree exactly k and each house is of degree exactly one. Figure 1a depicts a simple example of a problem defined for 4 houses and 2 wells, whereas Figure 1b illustrates its generalized form.



(a) Simple problem graph



(b) Generalized problem graph

Additional notation:

- $W = W_1, W_2, \dots, W_n$ is the set of wells, $H = \{H_1, H_2, \dots, H_{kn}\}$ is the set of houses. Collectively, they form a set of graph vertices $V = W \cup H$ with $n + kn$ members. Each vertex is characterized by a coordinate tuple (x,y).
- $E_{i,j} = (W_i, H_j)$ is the edge connecting the well W_i and the house H_j . Each edge has a **weight** $w_{i,j} = -d_{i,j}$, where $d_{i,j}$ is the **euclidean distance** between the well W_i and the house H_j . The set of all edges forms the set E with $nkn = kn^2$ members.
- l_i is the **label of the vertex i**. It is an assigned value for a vertex that stores problem-specific information.
- $x_{i,j}$ is the **incidence value**. It is an assigned value for a vertex, a boolean that represents whether a given edge is in the matching.

$$x_{i,j} = \begin{cases} 1, & \text{if } E_{i,j} \in M \\ 0, & \text{otherwise} \end{cases}$$

2 Solution description

The algorithm used to solve the problem is based on the **Hungarian Algorithm**.

2.1 Hungarian Algorithm

The *Hungarian Algorithm* is an algorithm aimed at solving assignment problems. It is particularly effective for problems where **each element must be assigned to exactly one element in another set**, and the goal is to maximize the total cost of the assignments.

2.1.1 Hungarian Algorithm – Wells Problem

The *Hungarian Algorithm* can be adapted to solve the Wells Problem and find the optimal assignment of houses to wells. Firstly, it is important to emphasize that since the Hungarian algorithm maximizes the total cost, each edge weight is assigned the negative value of the distance. Second, since the number of wells does not necessarily equal the number of houses (which is the case for all $k \neq 1$), each well has to be duplicated k times to avoid the one-to-one assignment constraint. Matrix 1 showcases a generalized representation of the proposed transformed graph.

	W_1	W_1	W_2	W_2	\dots	W_n	W_n
H_1	$-d1, 1$	$-d1, 1$	$-d2, 1$	$-d2, 1$	\dots	$-dn, 1$	$-dn, 1$
H_2	$-d1, 2$	$-d1, 2$	$-d2, 2$	$-d2, 2$	\dots	$-dn, 2$	$-dn, 2$
\vdots	\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
H_{kn}	$-d1, kn$	$-d1, kn$	$-d2, kn$	$-d2, kn$	\dots	$-dn, kn$	$-dn, kn$

Table 1: Laplacian representation of wells and houses graph

Output assignment for each well is then defined as the collection of houses assigned to any of the duplicates of the given well. The algorithm follows the general description:

1. **Initialize empty matching:** Start with an empty graph matching M .
2. **Wells duplication:** Given a graph $G = (V, E) : V = W \cup H \wedge \forall W_i \in W, \forall H_j \in H, E = \{E_{i,j} | E_{i,j} = (W_i, H_j)\}$ with n wells, kn houses, and kn^2 edges, create a graph $G' = (V', E') : (V' = W' \cup H) \wedge (\forall W'_i \in W' \forall H_j \in H E' = \{E'_{i,j} | E'_{i,j} = (W'_i, H_j)\})$ with kn wells, kn houses, and $(kn)^2$ edges. The new sets of wells W' and edges E' contain k duplicates of every element from the sets W and E , respectively. This modification allows for the possibility of connecting each house to the same well, despite the one-to-one assignment constraint inherent in the basic Hungarian algorithm.
3. **Initial Feasible Labeling:** For each house, assign a label l equal to the maximum of the weights of the connected edges. For each well, assign a label l equal to 0. This creates an initial feasible labeling.

4. **Equality graph:** Given a graph $G' = (V', E')$, construct an equality graph $G_l = (V', E'_l)$. The equality graph contains a subset of edges such that $w_{i,j} = l(i) + l(j)$ where i, j - any two nodes from V' .
5. **Construct Alternating Augmenting Path:** Select the first house $H_j \in H$ (lowest j -index), for which all connected edges have incidence value $x_{i,j}$ equal to 0. Simply said, select the first house that has no connected edge in the matching M . Starting from this house, create every possible longest walk (without loops) on the graph G_l such that the traversed edges alternate in the incidence value (that is, the incidence values follow the pattern 0-1-0-1-...). If any of the walks end on a node that has no connected edge in the matching M (**augmenting path**), go to the **Matching Modification** step. If no walk with this property exists, go to the **Label Modification** step.
6. **Label modification:** Select the longest walk. Place houses traversed in this walk in a subset $S \in H$. Place wells traversed in this walk in a subset $T \in W'$. For every edge connecting any node $s \in S$ with any node $y \in W' \setminus T$, compute the value $\delta = l(s) + l(y) - w(s, y)$. Let $\Delta = \min(\delta_{s,y})$ be the minimum value out of all computed $\delta_{s,y}$. Improve the labeling l to l' for every vertex v based on the following rules:

$$l'_v = \begin{cases} l'(v) = l(v) - \Delta, & \text{if } v \in S \\ l'(v) = l(v) + \Delta, & \text{if } v \in T \\ l(v), & \text{otherwise} \end{cases}$$

7. **Matching Modification:** Starting from the initial node, traverse the discovered augmenting path. For every edge passed, swap its incidence value. In other words, every edge in the path that is in the matching should be removed, and every edge that is not in the matching should be added to it.
8. **Optimal Assignment Check:** If every vertex has exactly one connected matched edge from E_l , the assignment is optimal. Return the matching as the output. Otherwise, go back to the **Construct Alternating Augmenting Path** step.

2.2 Pseudocode

First, let's list the built-in functions that have been used in the pseudo-code. The description of their implementation was omitted since it is trivial.

- $shape(M)$ - function returns dimensions of matrix M .
- $zeros_matrix((row_length, col_length))$ - function creates a matrix (allocates memory) of the size $row_length \times col_length$ filled with zeros.
- $len(L)$ - function returns the length of the list L .
- $min(L)$ - function returns a minimum value from the list L .

Secondly, let's describe the input to the function. It is a graph object with attributes *vertices* - a list of length $(n + kn)$ with coordinates of houses and wells, and *edges* - a list of length kn with Euclidean distances between houses and wells, with input data as defined in the section 1.1). For simplicity, let's also define some helper functions of this object:

- $well(i)$ - function returns the (x,y) coordinate pair of the well W_i . This function is equivalent to calling $graph.vertices[i]$
- $house(i)$ - function returns the (x,y) coordinate pair of the house H_i . This function is equivalent to calling $graph.vertices[n + i]$
- $distance(i, j)$ - function returns the distance between the well W_i and the house H_j . This function is equivalent to calling $graph.edges[k*i + j]$

Let us now proceed to the pseudo-code of the actual algorithm.

Algorithm 1: Modified Hungarian Pseudocode:

Input: **graph** - a graph object with *vertices* and *edges*

```
1 def hungarian_algorithm(graph):
2   # Step 1: Initialize empty matching
3   M - empty matching
4   # Step 2: Wells duplication
5   G' = duplicate_wells(graph)
6   # Step 3: Initial Feasible labeling
7   G' = initial_labeling(G')
8   while True do
9     # Step 4: Equality Graph
10    Gl = equality_graph(G')
11    # Step 5: Construct Augmenting Path
12    path, is_augmenting = find_augmenting_path(Gl, M)
13    if not is_augmenting then
14      # Step 6: Label Modification
15      G' = label_modification(Gl, path)
16      continue
17    end
18    # Step 7: Matching Modification
19    M = matching_modification(path, M)
20    # Step 8: Optimal Assignment Check
21    if optimal_assignment_check(Gl, M) then
22      break
23    end
24  end
25  return M
```

Algorithm 2: Wells duplication:

Input: **graph** - a graph object with *vertices* and *edges*

```
1 def duplicate_wells(graph):
2   G' - empty graph
3   for i = 0 to graph.k - 1 do
4     G'.add_edges(graph.edges)
5     G'.add_wells(graph.wells)
6   end
7   G'.add_houses(graph.houses)
8   return G';
```

Algorithm 3: Initial Labeling Pseudocode:

Input: G' - an extended graph object with *vertices* and *edges*

```
1 def initial_labeling( $G'$ ):
2   for  $i = 0$  to  $G'.kn - 1$  do
3      $G'.houses[i].label = \max(G'.houses[i].edges.weights)$ 
4      $G'.wells[i].label = 0$ 
5   end
6   return  $G'$ 
```

Algorithm 4: Equality Graph Pseudocode:

Input: G' - an extended graph object with *vertices* and *edges*

```
1 def equality_graph( $G'$ ):
2    $G_l$  - empty graph
3   for  $i = 0$  to  $(G'.kn)^2 - 1$  do
4     if  $G'.edges[i].weight == G'.edges[i].src.label + G'.edges[i].dst$  then
5        $G_l.add\_edge(G'.edges[i])$ 
6     end
7   end
8    $G_l.add\_houses(G'.houses)$ 
9    $G_l.add\_wells(G'.wells)$ 
10  return  $G_l$ 
```

Algorithm 5: Construct Augmenting Path Pseudocode:

Input: G_l - an equality graph of G' .
 M - a matching of G'

```
1 def find_augmenting_path( $G_l, M$ ):
2    $G_l$  - empty graph
3   for  $i = 0$  to  $G'.kn - 1$  do
4     if not  $M.contains\_any(G'.houses[i].edges)$  then
5       starting_node =  $G'.houses[i]$ 
6       for alternating_path  $p$  in  $alternating\_paths(starting\_node)$  do
7         if  $is\_augmenting(alternating\_path)$  then
8           return  $p, True$ 
9         end
10      end
11      return  $p, False$ 
12    end
13  end
14  return empty_path, False;
```

Algorithm 6: Label Modification Pseudocode:

Input: G' - an extended graph object with *vertices* and *edges*
 $path$ - an alternating path on the graph G'

```
1 def label_modification( $G', path$ ):
2   S - empty set of vertices
3    $W \setminus T = G'.wells$ 
4   for House  $h$  in  $path$  do
5     S.add( $h$ )
6   end
7   for Well  $w$  in  $path$  do
8      $W \setminus T.remove(w)$ 
9   end
10  deltas = []
11  for Edge  $e$  in  $G'.edges$  do
12    if ( $e.src$  in  $S \vee e.src$  in  $W \setminus T$ )  $\wedge$  ( $e.dst$  in  $S \vee e.dst$  in  $W \setminus T$ ) then
13      deltas.add( $e.src.label + e.dst.label - e.weight$ )
14    end
15  end
16   $\Delta = \min(deltas)$ 
17  for Edge  $e$  in  $G'.edges$  do
18    if  $e$  in  $S$  then
19       $e.weight -= \Delta$ 
20    end
21    if  $e$  in  $T$  then
22       $e.weight += \Delta$ 
23    end
24  end
25  return  $G'$ 
```

Algorithm 7: Matching Modification Pseudocode:

Input: $path$ - an alternating path on the graph G'
 M - a matching of G'

```
1 def matching_modification( $path, M$ ):
2   for Edge  $e$  in  $path$  do
3     if  $e$  in  $M$  then
4        $M.remove(e)$ ;
5     end
6     else
7        $M.add(e)$ ;
8     end
9   end
10  return  $M$ 
```

Algorithm 8: Optimal Assignment Check Pseudocode:

Input: G_l - an equality graph of G'
 M - a matching of G'

```
1 def optimal_assignment_check( $G_l, M$ ):
2   for Vertex  $v$  in  $G_l.vertices$  do
3     if not  $M.contains\_one(v.edges)$  then
4       return False
5     end
6   end
7   return True
```

3 Correctness and complexity

The correctness and complexity of the proposed algorithm have been analyzed as follows:

3.1 Correctness

The proposed algorithm can be considered correct, specifically by analyzing that it terminates and that the output contains the optimal solution.

3.1.1 Termination of the algorithm

To prove the termination of the algorithm, it is enough to notice the following:

1. At first, we start with an initial feasible labeling, some equality graph G_l , and an empty matching M , with no edges with a 1-incidence value.
2. In each iteration, either the **Label modification** or **Matching duplication** occurs.
3. **Label modification** finds minimum value $\Delta = \min(l(i) + l(j) - w(i, j))$ for some edge $e_{i,j}$. This edge, by the description of this step, has one of the nodes inside the alternating path and the other outside. Therefore, assuming it is the node i that is inside the augmenting path, it will have the value Δ subtracted from the label, $l'(i) = l(i) - \Delta$. The other node will have the label value unchanged $l'(j) = l(j)$. Then, the edge between these two nodes will satisfy the graph equality condition and will appear in the graph G_l in the next iteration. Furthermore, at no point in the execution of the algorithm is the number of the edges in G_l being lowered, so this step is guaranteed to terminate.
4. **Matching modification** occurs when the augmenting path is found. By the definition, this step contains an odd amount of edges with a 0-incidence value and an even amount of edges with a 1-incidence value. Furthermore, the number of 0-incidence value edges is always one greater than 1-incidence value edges. Swapping the incidence values therefore increments the total amount of edges with a 1-incidence value, and since the number of edges in the graph is finite, this step is guaranteed to terminate.
5. **Optimal Assignment Check** terminates as soon as there are exactly kn -matched edges (edges with 1-incidence value). This is guaranteed to happen as initially, the algorithm starts with 0 such edges, and then this count gets incremented inside the **Matching modification** step.

In total, every step of the algorithm eventually terminates, therefore the algorithm is guaranteed to terminate.

3.1.2 Optimality of the solution

To prove the optimality of the solution, it is enough to prove the following statement: **A feasible labeling on a perfect matching returns a maximum-weighted matching.**

Suppose each edge $e \in E'_M$ from the matching M on the bipartite graph $G' = (V', E')$ connects two vertices, and every vertex $v \in V'$ is covered exactly once. Let $w(e)$ denote the weight of edge e , and e_i, e_j denote nodes i and j respectively, that are connected by edge e . With this, we have the following inequality:

$$\text{cost}(M') = \sum_{e \in E'_M} w(e) \leq \sum_{e \in E'_M} (l(e_i) + l(e_j)) = \sum_{v \in V'} l(v)$$

in which M' is any perfect matching on G' generated randomly. This means that $\sum_{v \in V'} l(v)$ is the upper bound of the cost of any perfect matching over G' . Now, let M be a perfect match on the equality graph G_l of G' , Then

$$\text{cost}(M) = \sum_{e \in E_l} w(e) = \sum_{e \in E_l} (l(e_i) + l(e_j)) = \sum_{v \in V'} l(v)$$

So $\text{cost}(M') \leq \text{cost}(M)$ and M is optimal, maximum-weight matching.

Our proposed algorithm operates on a bipartite graph G' with computed initial feasible labeling, and attempts to create a perfect matching on the equality graph G_l of G' , so any created perfect matching on this equality graph will be the perfect matching on G' and subsequently, the optimal solution to the problem.

3.2 Complexity

Concerning the complexity, the *Hungarian Algorithm* has a theoretical worst-case time complexity of $O(|V'|^3) = O((2 \times kn)^3) = O((kn)^3)$. In particular, analyzing each step separately:

1. **Initialize empty matching** has a complexity equal to $O(1)$ since it creates an empty object.
2. **Wells duplication** has the complexity equal to $O((n + kn^2) \times k) = O(kn + (kn)^2) = O((kn)^2)$ since n wells and kn^2 edges have to be duplicated k times.
3. **Initial Feasible Labeling** has the time complexity $O((kn) \times n + kn) = O(nk^2)$ since we have to compute the maximum value for each house (kn total) that has exactly n wells connected, plus we have to set kn wells' labels to 0.
4. **Equality Graph** in each iteration has the time complexity $O((kn \times kn))$ since it is necessary to check the equality condition for each edge, and there are kn wells and kn houses.
5. **Construct Alternating Augmenting Path** is a step that has complexity $O(kn + (kn)^2)$. It takes up to $O(kn)$ time to find a vertex to begin the traverse, and then the maximum number of visited nodes can never surpass $O((kn)^2)$ - once for every possible combination.
6. **Label modification** has time complexity $O((kn)^2 + (kn)^2) = O((kn)^2)$ as delta has to be computed for up to $kn \times kn$ edges, and then all edges have to be updated.
7. **Matching Modification** has time complexity $O(2(kn))$ at the theoretical maximum path length cannot exceed the number of vertices.
8. **Optimal Assignment Check:** has time complexity $O(2(kn))$ as each edge ($2kn$ edges) has to check if it has exactly one edge connected in the matching.
9. **Iterations:** Steps 4-8 have to be computed up to kn times (once per each house), increasing their corresponding time complexity by kn . The maximum time complexity for the iteration loop is therefore $O(kn \times (kn)^2) = O((kn)^3) = O(|V'|^3)$.

4 Input and Output

The input, accepted by the system, is given in the *.txt* file that specifies the Cartesian coordinates of houses and wells in the form of tuples. In the first line, the file contains a declaration of the number of houses and wells ($n \ k$ - where n is the number of wells and k is the number of houses per well). Then, the coordinates of wells and houses are provided, where each pair of coordinates is given in a new line. At the beginning are n lines with coordinates of wells and then there are kn lines with coordinates of houses. The x and y coordinates are separated using a colon. For example, let us consider the following input file:

```
2 2
2.5,1.5
0.8,1.5
1,1
2,1
2,2
1,2
```

The given input specifies 4 houses with respective coordinates: $H_1 = (1,1)$, $H_2 = (2,1)$, $H_3 = (2,2)$, $H_4 = (1,2)$ and wells with coordinates: $W_1 = (2.5,1.5)$, $W_2 = (0.8,1.5)$. In this case, each well is going to supply 2 houses, as specified in the third line of the input file.

The produced output is going to be stored in the *output.txt* file and will contain the assignment of wells to houses (each specified in a new line), as well as the final minimized cost. For the above example, the *output.txt* would have a following content:

```
W1(2.5,1.5) -> H2(2,1),H3(2,2)
W2(0.8,1.5) -> H1(1,1),H4(1,2)
Total Cost: 2.49
```


5 Example

1. Input:

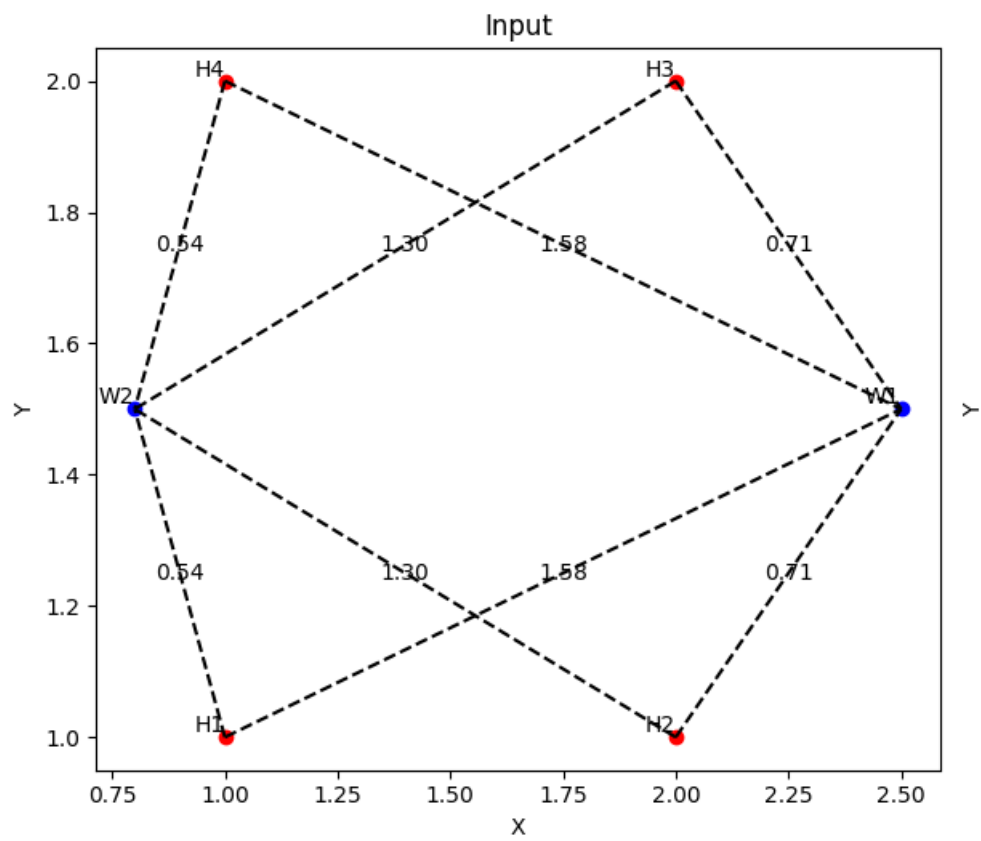


Figure 2: Input example

2. Output

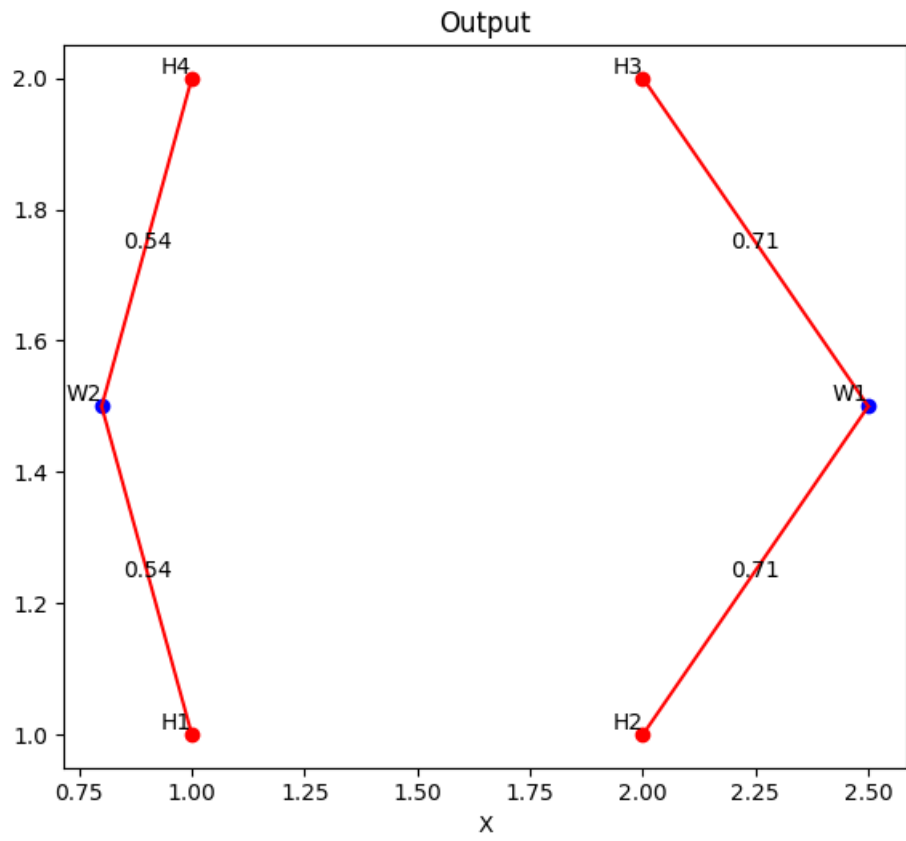


Figure 3: Ouput example