# Advanced Algorithms - Wells Assignment Project

Piotr Krzemiński 305785
Zofia Wrona 305803

## 1 Problem Description

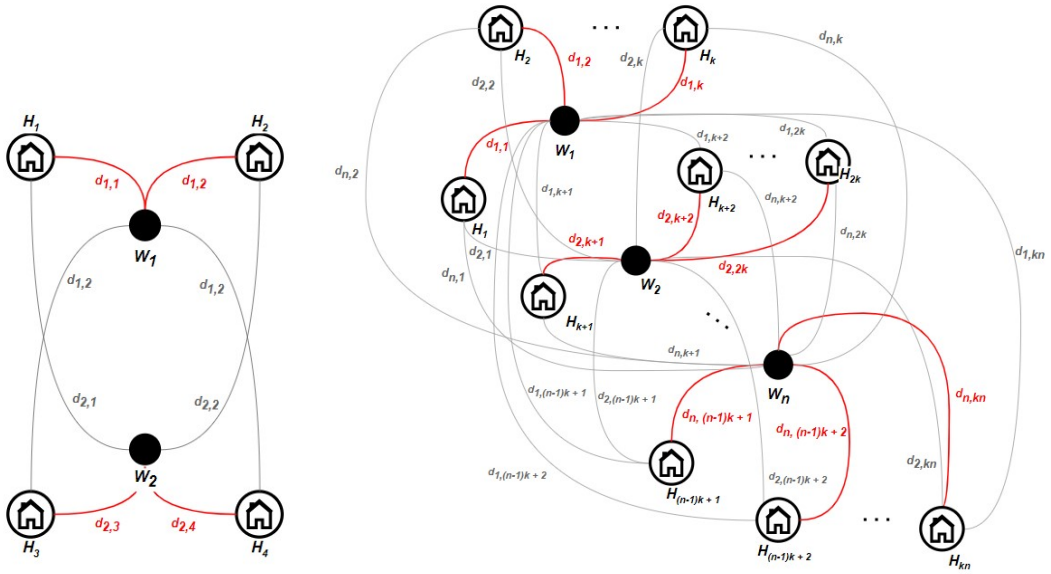
The Wells Problem is a combinatorial optimization problem with applications in fields such as water supply and facility location. The task to be solved has been formulated in the following way:

> *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:

- $W = W_1, W_2, ..., W_n$ is the set of wells, $H = \{H_1, H_2, ..., H_{kn}\}$ 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.

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

## 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.

| | $\mathbf{W}_1$ | $\mathbf{W}_1$ | $\mathbf{W}_2$ | $\mathbf{W}_2$ | ... | $\mathbf{W}_n$ | $\mathbf{W}_n$ |
|---|---|---|---|---|---|---|---|
| $\mathbf{H}_1$ | $-d1, 1$ | $-d1, 1$ | $-d2, 1$ | $-d2, 1$ | ... | $-dn, 1$ | $-dn, 1$ |
| $\mathbf{H}_2$ | $-d1, 2$ | $-d1, 2$ | $-d2, 2$ | $-d2, 2$ | ... | $-dn, 2$ | $-dn, 2$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ | $\vdots$ |
| $\mathbf{H}_{kn}$ | $-d1, kn$ | $-d1, kn$ | $-d2, kn$ | $-d2, kn$ | ... | $-dn, kn$ | $-dn, kn$ |

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. ***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.
2. ***Initialize empty matching***: Initialize an empty graph matching $M$.
3. ***Initial Feasible Labeling***: For each well, assign a label $l$ equal to the **negative** value of the largest distance. For each house, assign a label $l$ equal to 0. This creates an initial feasible labeling ( **feasible labeling** means labeling in which sum of the labels for each pair of houses and wells is greater or equal than the weight of the edge by which they are connected).

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***: Place houses traversed in this walk in a subset $S \subseteq H$. Place wells traversed in this walk in a subset $T \subseteq W'$. 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:

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

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

Throughout the algorithm all key variables were stored inside the **Graph** object. It is important to note that the graphs were passed between functions by reference (excluding the duplicate_wells function), therefore returning a graph variable and assigning it a new name has O(1) complexiy as no copy is created.

The graph variable stores the following parameters:

1. **n, k, wells_coordinates, houses_coordinates** - values read from the input file that define a graph.

2. **cost_matrix** - array that stores the costs (negative distances) between wells and houses.

3. **label_well, label_house** - labels of wells and houses

4. **queue, previous_well** - two variables that store backtracking information (used to mimic alternating tree in lower time complexity).

5. **write, read** - values used to reduce unmatched nodes search time when constructing alternating paths.

6. **S, T** - two sets that store information whether a given well/house was visited when constructing the alternating tree.

7. **slack, slack_matching_well** - first variable stores the difference between a well and all houses connected to it. Second variable stores the well that, given a house, has an edge of incidence value equal to one (zero slack, edge inside matching). Using these values we can mimic building equivalence_graph and reduce time complexity of computing delta in label_modification function.

With that in mind, let us proceed to the pseudocode of the algorithm.

**Algorithm 1:** Modified Hungarian Pseudocode:

**Input: input_file** - a text file with graph data with n *wells* and kn *houses*

```
def hungarian_algorithm(input_file):
    # Step 0: Read graph from file
    graph - read_input(input_file)
    # Step 1: Wells duplication
    G' = duplicate_wells(graph)
    # Step 2: Initialize empty matching
    M - empty matching
    # Step 3: Initial Feasible labeling
    G' = initial_labeling(G')
    # Step 4: Optimal Assignment Check
    while not optimal_assignment_check(Gl, M) do
        # Step 5: Reset Alternating Tree
        clean_alternating_tree(G')
        # Step 6: Equality Graph
        root = find_root_of_alternating_tree(G')
        # Step 7: Equality Graph
        Gl = equality_graph(G', root)
        is_augmenting = False
        while True do
            # Step 8: Construct Augmenting Path
            well, house, is_augmenting = find_augmenting_paths(Gl, M)
            if not is_augmenting then
                # Step 9: Label Modification
                G' = label_modification(G')
                # Step 10: Refine alternating tree
                well, house, is_augmenting = refine_alternating_tree(G', M)
            end
            if is_augmenting then
                break
            end
        end
        if is_augmenting then
            # Step 11: Matching Modification
            M = matching_modification(path, M)
        end
    end
    return graph, M
```

**Algorithm 2:** Wells duplication:

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

```
def duplicate_wells(graph):
    edges - empty list of edges
    wells - empty list of wells
    for i = 0 to graph.k - 1 do
        edges.add(graph.edges)
        wells.add(graph.wells)
    end
    G' = Graph.from(wells, edges, graph.houses)
    return G';
```

**Algorithm 3:** Initial Labeling Pseodocode:

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

```
def initial_labeling(G'):
    for i = 0 to G'.n - 1 do
        G'.wells[i].label = 0
        G'.houses[i].label = max(G'.houses[i].edges.weights)
    end
    return G'
```

**Algorithm 4:** Equality Graph Pseudocode:

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

```
def equality_graph(G'):
    G_l - empty graph
    G_l.add_wells(G'.wells)
    G_l.add_houses(G'.houses)
    for i = 0 to G'.n^2 − 1 do
        if G'.edges[i].weight == G'.edges[i].src.label + G'.edges[i].dst.label then
            G_l.add_edge(G'.edges[i])
        end
    end
    return G_l
```

**Algorithm 5:** Construct Augmenting Path Pseudocode:

**Input:** *starting_node* - a node from which to start building paths.
*M* - a matching of $G'$

```
def find_augmenting_paths(starting_node, matching):
    # Create a queue for BFS search
    queue = [(starting_node, [starting_node], False)]
    alternating_path = [];
    while queue is not empty do
        node, path = queue.pop(0)
        # Search for unvisited neighbors to extend path
        new_paths = find_next_paths(node, path)
        if new_paths.empty() then
            if is_augmenting(path, matching) then
                return path, True
            end
            # Finds longest alternating path for most changes in label modification step
            else if size(path) > size(alternating_path) then
                alternating_path = path
            end
        end
    end
    return alternating_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 = path.houses
3      T = path.wells
4      W \ T = G'.wells.filter_out(T)
5      deltas = []
6      for Edge e in G'.edges do
7          if e ∈ path then
8              deltas.add(e.house.label + e.well.label - e.weight)
9          end
10     end
11     Δ = min(deltas)
12     for Edge e in G'.edges do
13         if e in S then
14             e.weight -= Δ
15         end
16         if e in T then
17             e.weight += Δ
18         end
19     end
20     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
9      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 House h in G_l.houses do
3          if not M.contains_any(h.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 of 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 alternating path, it will have the value $\Delta$ 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.**
A **feasible labeling** of vertices from $W$' and $H$ assigns a label $l(v)$ where $v \in W' \cup H$, such that $l(H_i) + l(W_j) \leq d_{i,j}$. 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 $v_i, v_j$ denote nodes i and j that are connected by an edge $e$ with a weight $w(e) = d_{i,j}$. With this, we have the following inequality:

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

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

$$cost(M) = \sum_{e \in E_l} w(e) = \sum_{e \in E_l} (l(v_i) + l(v_j)) = \sum_{v \in V'} l(v)$$

So $cost(M') \leq cost(M)$ 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.
Furthermore, let us note that after modification of the labelling, it still remains feasible. Assume that in the beginning of some iteration $l(v_i) + l(v_j) \geq w(e)$. Let us consider the following cases:

1. $v_i \in S$ and $v_j \in T \longrightarrow$ label modification would be $l(v_i) - \Delta$ and $l(v_j) + \Delta$, so:
$$l(v_i) + \Delta + l(v_j) - \Delta = l(v_i) + l(v_j) \geq w(e)$$
by the assumption from the beginning of iteration.

2. $v_i \in S$ and $v_j \notin T \longrightarrow$ label modification would be $l(v_i) - \Delta$ and $l(v_j)$ remains the same, so:
$$l(v_i) - \Delta + l(v_j)$$
but recall that $\Delta = min(l(v_k) + l(v_n) - w(e_{k,n}))$ for some $v_k \in S$ and $v_n \notin T$. Therefore the following also holds:
$$l(v_i) + l(v_j) - w(e) \geq l(v_k) + l(v_n) - w(e_{k,n})$$
which by moving $w(e)$ to the right-hand side and the $l(v_k) + l(v_n) - w(e_{k,n})$ to the left-hand side, gives us:
$$l(v_i) + l(v_j) - (l(v_k) + l(v_n) - w(e_{k,n})) \geq w(e) \equiv l(v_i) + l(v_j) - \Delta \geq w(e)$$

3. $v_i \notin S$ and $v_j \in T \longrightarrow$ label modification would be $l(v_j) + \Delta$ and $l(v_i)$ remains the same, so:
$$l(v_i) + l(v_j) + \Delta$$
Therefore, since we enlarge value of label $l(v_j)$:
$$l(v_i) + l(v_j) \geq w(e) \Rightarrow l(v_i) + l(v_j) + \Delta \geq w(e)$$

4. $v_i \notin S$ and $v_j \notin T \longrightarrow$ labeling remains unchanged.

Therefore, selection of $\Delta$ ensures that the labeling will always be feasible.

## 3.2 Complexity

Concerning the complexity, the *Hungarian Algorithm* has a theoretical worst-case time complexity of $O(|V'|^4) = O((kn + kn)^4) = O((kn)^4)$. 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 $O((kn)^2 + (kn)^3) = O((kn)^3)$ as delta has to be computed for up to $kn \times kn$ edges, each edge has to check if it is inside two vertex sets - up to $kn$ vertices in $S$ and $kn$ vertices in W_minus_T, 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 each iteration loop is $O((kn)^3)$ so the total algorithm complexity is $O(kn \times (kn)^3) = O((kn)^4) = O(|V'|^4)$.

## 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**

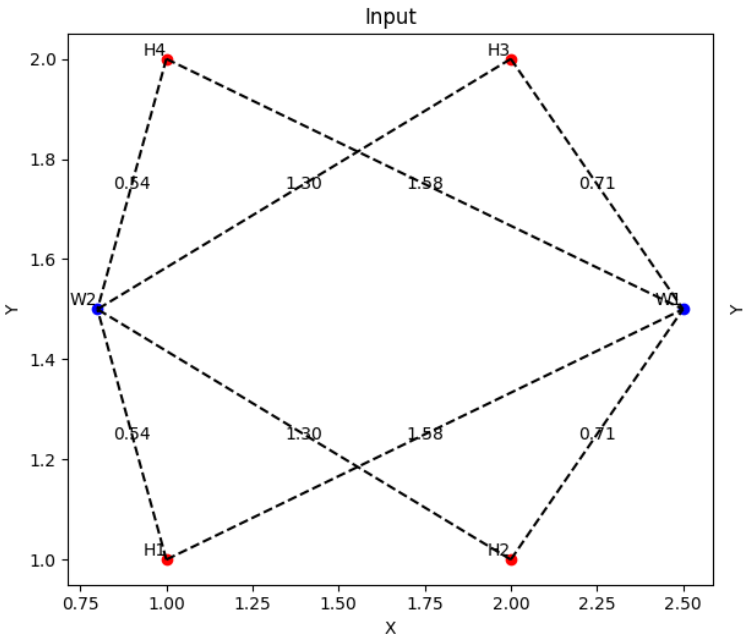



Figure 2: Input example

**2. Initial Labeling and Wells Duplication** Each well is now replaced with 2 duplicates with the same label as the original well.

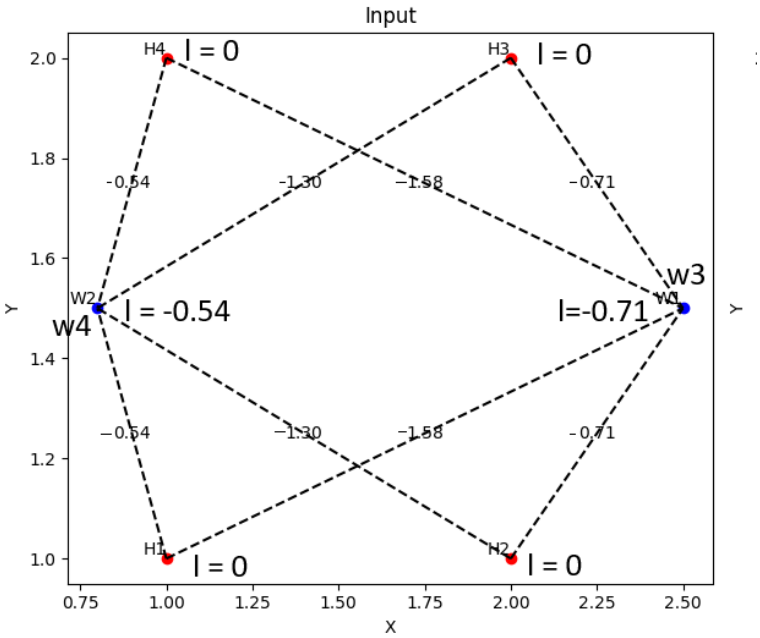



Figure 3: Initial labeling and wells duplication

**3. Equality graph** Construct an equality graph on duplicated wells and houses.

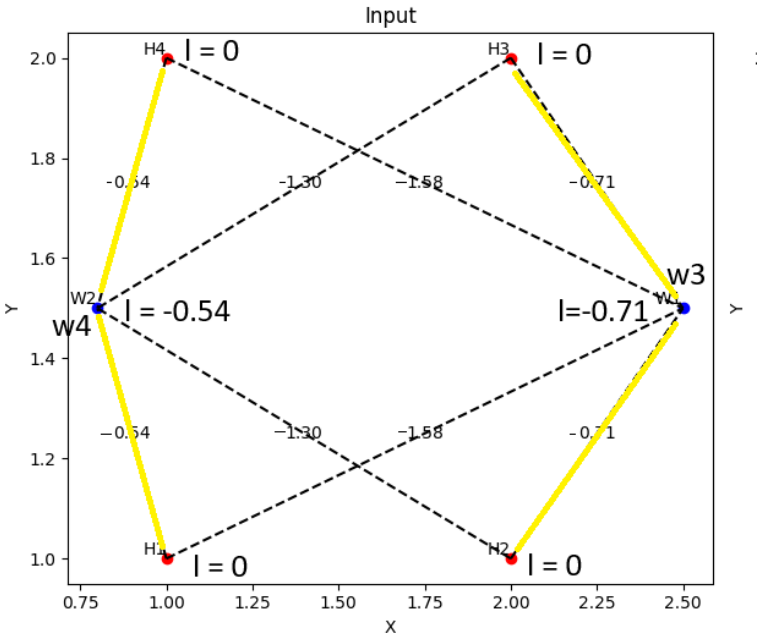



Figure 4: Equality graph

**4.1. First augmenting path** The augmenting path was found W1 -> H3, and because the edge was not in the matching, it is now in the matching. The equality graph stays the same.

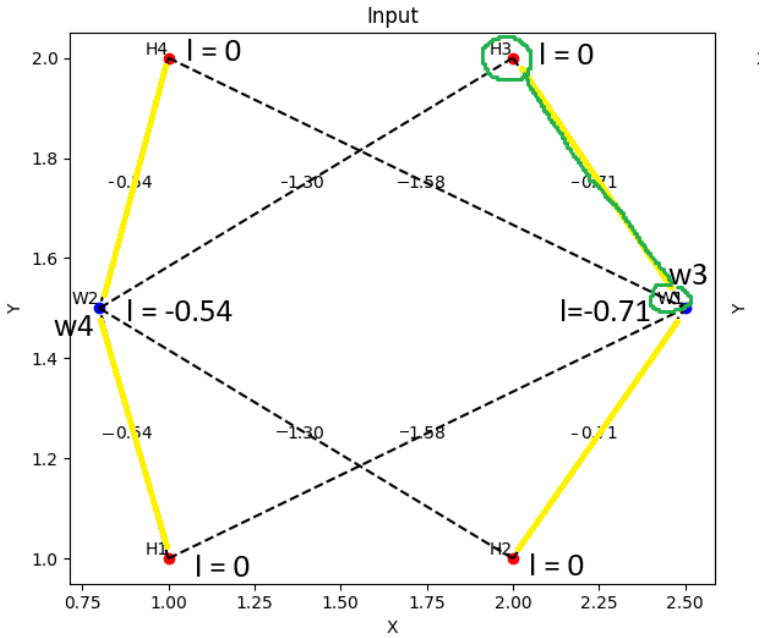



Figure 5: First augmenting path

**4.2. Second augmenting path** The augmenting path was found W2 -> H4, and because the edge was not in the matching, it is now in the matching. The equality graph stays the same.

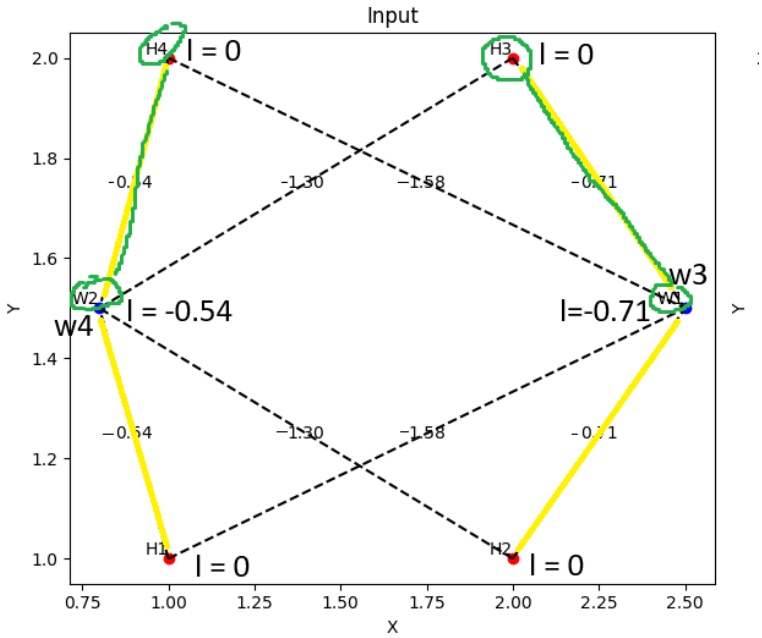



Figure 6: Second augmenting path

**4.3. Third augmenting path** The augmenting path was found W3 -> H2, and because the edge was not in the matching, it is now in the matching. The equality graph stays the same.

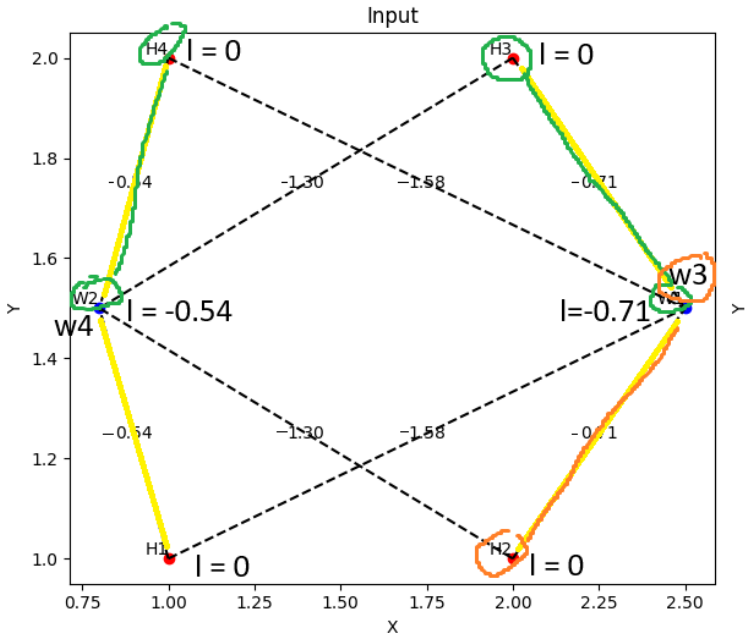



Figure 7: Third augmenting path

**4.4. Fourth augmenting path** The augmenting path was found W4 -> H1, and because the edge was not in the matching, it is now in the matching. The equality graph stays the same. In addition, optimal assignment check now passes as each well has an edge in the matching.

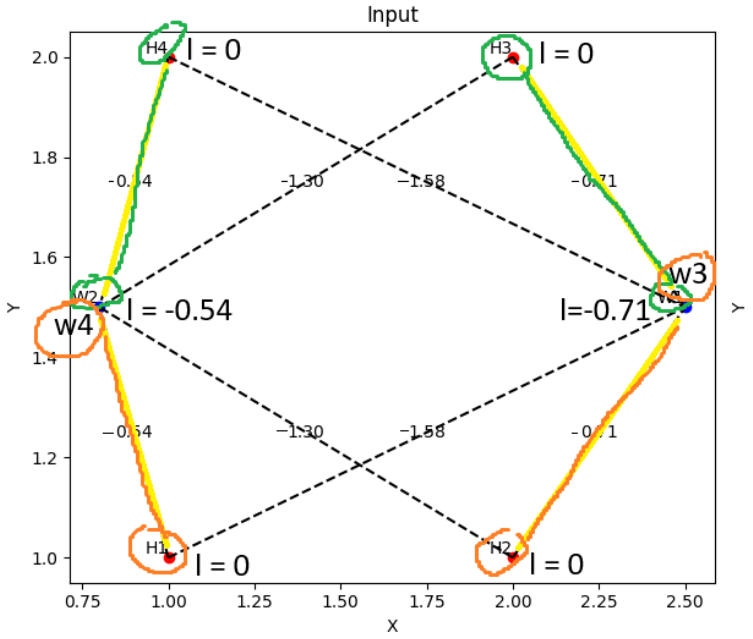



Figure 8: Fourth augmenting path

**5. Output** The original wells are assigned the duplicates' houses from the matching. Algorithm terminates.

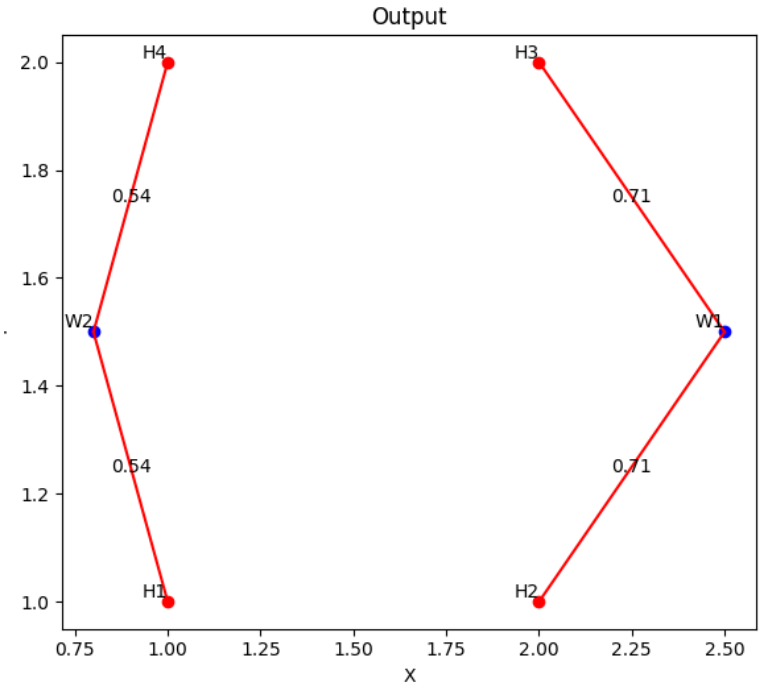



Figure 9: Output perfect matching