Last update: December 13, 2023 Translated From: e-maxx.ru

Hungarian algorithm for solving the assignment problem

Statement of the assignment problem

There are several standard formulations of the assignment problem (all of which are essentially equivalent). Here are some of them:

- There are n jobs and n workers. Each worker specifies the amount of money they expect for a particular job. Each worker can be assigned to only one job. The objective is to assign jobs to workers in a way that minimizes the total cost.
- Given an $n \times n$ matrix A, the task is to select one number from each row such that exactly one number is chosen from each column, and the sum of the selected numbers is minimized.
- Given an $n \times n$ matrix A, the task is to find a permutation p of length n such that the value $\sum A[i]\,[p[i]]$ is minimized.
- Consider a complete bipartite graph with n vertices per part, where each edge is assigned a weight. The objective is to find a perfect matching with the minimum total weight.

It is important to note that all the above scenarios are "**square**" problems, meaning both dimensions are always equal to n. In practice, similar "**rectangular**" formulations are often encountered, where n is not equal to m, and the task is to select $\min(n, m)$ elements. However, it can be observed that a "rectangular" problem can always be transformed into a "square" problem by adding rows or columns with zero or infinite values, respectively.

We also note that by analogy with the search for a **minimum** solution, one can also pose the problem of finding a **maximum** solution. However, these two problems are equivalent to each other: it is enough to multiply all the weights by -1.

Hungarian algorithm

Historical reference

The algorithm was developed and published by Harold **Kuhn** in 1955. Kuhn himself gave it the name "Hungarian" because it was based on the earlier work by Hungarian mathematicians Dénes Kőnig and Jenő Egerváry.
In 1957, James **Munkres** showed that this algorithm runs in (strictly) polynomial time, independently from the cost.
Therefore, in literature, this algorithm is known not only as the "Hungarian", but also as the "Kuhn-Mankres algorithm" or "Mankres algorithm".

However, it was recently discovered in 2006 that the same algorithm was invented **a century before Kuhn** by the German mathematician Carl Gustav **Jacobi**. His work, *About the research of the order of a system of arbitrary ordinary differential equations*, which was published posthumously in 1890, contained, among other findings, a polynomial algorithm for solving the assignment problem. Unfortunately, since the publication was in Latin, it went unnoticed among mathematicians.

It is also worth noting that Kuhn's original algorithm had an asymptotic complexity of $\mathcal{O}(n^4)$, and only later Jack **Edmonds** and Richard **Karp** (and independently **Tomizawa**) showed how to improve it to an asymptotic complexity of $\mathcal{O}(n^3)$.

The $\mathcal{O}(n^4)$ algorithm

To avoid ambiguity, we note right away that we are mainly concerned with the assignment problem in a matrix formulation (i.e., given a matrix A, you need to select n cells from it that are in different rows and columns). We index arrays starting with 1, i.e., for example, a matrix A has indices $A[1 \dots n][1 \dots n]$.

We will also assume that all numbers in matrix A are **non-negative** (if this is not the case, you can always make the matrix non-negative by adding some constant to all numbers).

Let's call a **potential** two arbitrary arrays of numbers $u[1 \dots n]$ and $v[1 \dots n]$, such that the following condition is satisfied:

$$u[i] + v[j] \le A[i][j], \quad i = 1 \dots n,\ j = 1 \dots n$$

(As you can see, $u[i]$ corresponds to the i-th row, and $v[j]$ corresponds to the j-th column of the matrix).

Let's call **the value f of the potential** the sum of its elements:

$$f = \sum_{i=1}^{n} u[i] + \sum_{j=1}^{n} v[j].$$

On one hand, it is easy to see that the cost of the desired solution sol **is not less than** the value of any potential.

Lemma. $sol \ge f$.

Proof

The desired solution of the problem consists of n cells of the matrix A, so $u[i] + v[j] \le A[i][j]$ for each of them. Since all the elements in sol are in different rows and columns, summing these inequalities over all the selected $A[i][j]$, you get f on the left side of the inequality, and sol on the right side.

On the other hand, it turns out that there is always a solution and a potential that turns this inequality into **equality**. The Hungarian algorithm described below will be a constructive proof of this fact. For now, let's just pay attention to the fact that if any solution has a cost equal to any potential, then this solution is **optimal**.

Let's fix some potential. Let's call an edge (i, j) **rigid** if $u[i] + v[j] = A[i][j]$.

Recall an alternative formulation of the assignment problem, using a bipartite graph. Denote with H a bipartite graph composed only of rigid edges. The Hungarian algorithm will maintain, for the current potential, **the maximum-number-of-edges matching** M of the graph H. As soon as M contains n edges, then the solution to the problem will be just M (after all, it will be a solution whose cost coincides with the value of a potential).

Let's proceed directly to **the description of the algorithm**.

Step 1. At the beginning, the potential is assumed to be zero ($u[i] = v[i] = 0$ for all i), and the matching M is assumed to be empty.

Step 2. Further, at each step of the algorithm, we try, without changing the potential, to increase the cardinality of the current matching M by one (recall that the matching is searched in the graph of rigid edges H). To do this, the usual Kuhn Algorithm for finding the maximum matching in bipartite graphs is used. Let us recall the algorithm here. All edges of the matching M are oriented in the direction from the right part to the left one, and all other edges of the graph H are oriented in the opposite direction.

Recall (from the terminology of searching for matchings) that a vertex is called saturated if an edge of the current matching is adjacent to it. A vertex that is not adjacent to any edge of the current matching is called unsaturated. A path of odd length, in which the first edge does not belong to the matching, and for all subsequent edges there is an alternating belonging to the matching (belongs/does not belong) - is called an augmenting path. From all unsaturated vertices in the left part, a depth-first or breadth-first traversal is started. If, as a result of the search, it was possible to reach an unsaturated vertex of the right part, we have found an augmenting path from the left part to the right one. If we include odd edges of the path and remove the even ones in the matching (i.e. include the first edge in the matching, exclude the second, include the third, etc.), then we will increase the matching cardinality by one.

If there was no augmenting path, then the current matching M is maximal in the graph H.

Step 3. If at the current step, it is not possible to increase the cardinality of the current matching, then a recalculation of the potential is performed in such a way that, at the next steps, there will be more opportunities to increase the matching.

Denote by Z_1 the set of vertices of the left part that were visited during the last traversal of Kuhn's algorithm, and through Z_2 the set of visited vertices of the right part.

Let's calculate the value Δ:

$$\Delta = \min_{i \in Z_1,\, j \notin Z_2} A[i][j] - u[i] - v[j].$$

Lemma. $\Delta > 0$.

Proof

Suppose $\Delta = 0$. Then there exists a rigid edge (i, j) with $i \in Z_1$ and $j \notin Z_2$. It follows that the edge (i, j) must be oriented from the right part to the left one, i.e. (i, j) must be included in the matching M. However, this is impossible, because we could not get to the saturated vertex i except by going along the edge from j to i. So $\Delta > 0$.

Now let's **recalculate the potential** in this way:

- for all vertices $i \in Z_1$, do $u[i] \leftarrow u[i] + \Delta$,
- for all vertices $j \in Z_2$, do $v[j] \leftarrow v[j] - \Delta$.

Lemma. The resulting potential is still a correct potential.

Proof

We will show that, after recalculation, $u[i] + v[j] \leq A[i][j]$ for all i, j. For all the elements of A with $i \in Z_1$ and $j \in Z_2$, the sum $u[i] + v[j]$ does not change, so the inequality remains true. For all the elements with $i \notin Z_1$ and $j \in Z_2$, the sum $u[i] + v[j]$ decreases by Δ, so the inequality is still true. For the other elements whose $i \in Z_1$ and $j \notin Z_2$, the sum increases, but the inequality is still preserved, since the value Δ is, by definition, the maximum increase that does not change the inequality.

Lemma. The old matching M of rigid edges is valid, i.e. all edges of the matching will remain rigid.

Proof

For some rigid edge (i, j) to stop being rigid as a result of a change in potential, it is necessary that equality $u[i] + v[j] = A[i][j]$ turns into inequality $u[i] + v[j] < A[i][j]$. However, this can happen only when $i \notin Z_1$ and $j \in Z_2$. But $i \notin Z_1$ implies that the edge (i, j) could not be a matching edge.

Lemma. After each recalculation of the potential, the number of vertices reachable by the traversal, i.e. $|Z_1| + |Z_2|$, strictly increases.

Proof

First, note that any vertex that was reachable before recalculation, is still reachable. Indeed, if some vertex is reachable, then there is some path from reachable vertices to it, starting from the unsaturated vertex of the left part; since for edges of the form $(i, j),\ i \in Z_1,\ j \in Z_2$ the sum $u[i] + v[j]$ does not change, this entire path will be preserved after changing the potential. Secondly, we show that after a recalculation, at least one new vertex will be reachable. This follows from the definition of Δ: the edge (i, j) which Δ refers to will become rigid, so vertex j will be reachable from vertex i.

Due to the last lemma, **no more than n potential recalculations can occur** before an augmenting path is found and the matching cardinality of M is increased. Thus, sooner or later, a potential that corresponds to a perfect matching M^* will be found, and M^* will be the answer to the problem. If we talk about the complexity of the algorithm, then it is $\mathcal{O}(n^4)$: in total there should be at most n increases in matching, before each of which there are no more than n potential recalculations, each of which is performed in time $\mathcal{O}(n^2)$.

We will not give the implementation for the $\mathcal{O}(n^4)$ algorithm here, since it will turn out to be no shorter than the implementation for the $\mathcal{O}(n^3)$ one, described below.

The $\mathcal{O}(n^3)$ algorithm

Now let's learn how to implement the same algorithm in $\mathcal{O}(n^3)$ (for rectangular problems $n \times m$, $\mathcal{O}(n^2 m)$).

The key idea is to **consider matrix rows one by one**, and not all at once. Thus, the algorithm described above will take the following form:

1. Consider the next row of the matrix A.
2. While there is no increasing path starting in this row, recalculate the potential.
3. As soon as an augmenting path is found, propagate the matching along it (thus including the last edge in the matching), and restart from step 1 (to consider the next line).

To achieve the required complexity, it is necessary to implement steps 2-3, which are performed for each row of the matrix, in time $\mathcal{O}(n^2)$ (for rectangular problems in $\mathcal{O}(nm)$).

To do this, recall two facts proved above:

- With a change in the potential, the vertices that were reachable by Kuhn's traversal will remain reachable.
- In total, only $\mathcal{O}(n)$ recalculations of the potential could occur before an augmenting path was found.

From this follow these **key ideas** that allow us to achieve the required complexity:

- To check for the presence of an augmenting path, there is no need to start the Kuhn traversal again after each potential recalculation. Instead, you can make the Kuhn traversal in an **iterative form**: after each recalculation of the potential, look at the added rigid edges and, if their left ends were reachable, mark their right ends reachable as well and continue the traversal from them.
- Developing this idea further, we can present the algorithm as follows: at each step of the loop, the potential is recalculated. Subsequently, a column that has become reachable is identified (which will always exist as new reachable vertices emerge after every potential recalculation). If the column is unsaturated, an augmenting chain is discovered. Conversely, if the column is saturated, the matching row also becomes reachable.
- To quickly recalculate the potential (faster than the $\mathcal{O}(n^2)$ naive version), you need to maintain auxiliary minima for each of the columns:

  $$minv[j] = \min_{i \in Z_1} A[i][j] - u[i] - v[j].$$

  It's easy to see that the desired value Δ is expressed in terms of them as follows:

  $$\Delta = \min_{j \notin Z_2} minv[j].$$

  Thus, finding Δ can now be done in $\mathcal{O}(n)$.

  It is necessary to update the array $minv$ when new visited rows appear. This can be done in $\mathcal{O}(n)$ for the added row (which adds up over all rows to $\mathcal{O}(n^2)$). It is also necessary to update the array $minv$ when recalculating the potential, which is also done in time $\mathcal{O}(n)$ ($minv$ changes only for columns that have not yet been reached: namely, it decreases by Δ).

Thus, the algorithm takes the following form: in the outer loop, we consider matrix rows one by one. Each row is processed in time $\mathcal{O}(n^2)$, since only $\mathcal{O}(n)$ potential recalculations could occur (each in time $\mathcal{O}(n)$), and the array $minv$ is maintained in time $\mathcal{O}(n^2)$; Kuhn's algorithm will work in time $\mathcal{O}(n^2)$ (since it is presented in the form of $\mathcal{O}(n)$ iterations, each of which visits a new column).

The resulting complexity is $\mathcal{O}(n^3)$ or, if the problem is rectangular, $\mathcal{O}(n^2 m)$.

Implementation of the Hungarian algorithm

The implementation below was developed by **Andrey Lopatin** several years ago. It is distinguished by amazing conciseness: the entire algorithm consists of **30 lines of code**.

The implementation finds a solution for the rectangular matrix $A[1 \dots n][1 \dots m]$, where $n \le m$. The matrix is 1-based for convenience and code brevity: this implementation introduces a dummy zero row and zero column, which allows us to write many cycles in a general form, without additional checks.

Arrays $u[0 \dots n]$ and $v[0 \dots m]$ store potential. Initially, they are set to zero, which is consistent with a matrix of zero rows (Note that it is unimportant for this implementation whether or not the matrix A

contains negative numbers).

The array $p[0 \dots m]$ contains a matching: for each column $j = 1 \dots m$, it stores the number $p[j]$ of the selected row (or 0 if nothing has been selected yet). For the convenience of implementation, $p[0]$ is assumed to be equal to the number of the current row.

The array $minv[1 \dots m]$ contains, for each column j, the auxiliary minima necessary for a quick recalculation of the potential, as described above.

The array $way[1 \dots m]$ contains information about where these minimums are reached so that we can later reconstruct the augmenting path. Note that, to reconstruct the path, it is sufficient to store only column values, since the row numbers can be taken from the matching (i.e., from the array p). Thus, $way[j]$, for each column j, contains the number of the previous column in the path (or 0 if there is none).

The algorithm itself is an outer **loop through the rows of the matrix**, inside which the i-th row of the matrix is considered. The first *do-while* loop runs until a free column $j0$ is found. Each iteration of the loop marks visited a new column with the number $j0$ (calculated at the last iteration; and initially equal to zero - i.e. we start from a dummy column), as well as a new row $i0$ - adjacent to it in the matching (i.e. $p[j0]$; and initially when $j0 = 0$ the i-th row is taken). Due to the appearance of a new visited row $i0$, you need to recalculate the array $minv$ and Δ accordingly. If Δ is updated, then the column $j1$ becomes the minimum that has been reached (note that with such an implementation Δ could turn out to be equal to zero, which means that the potential cannot be changed at the current step: there is already a new reachable column). After that, the potential and the $minv$ array are recalculated. At the end of the "do-while" loop, we found an augmenting path ending in a column $j0$ that can be "unrolled" using the ancestor array way.

The constant `INF` is "infinity", i.e. some number, obviously greater than all possible numbers in the input matrix A.

```
vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
for (int i=1; i<=n; ++i) {
    p[0] = i;
    int j0 = 0;
    vector<int> minv (m+1, INF);
    vector<bool> used (m+1, false);
    do {
        used[j0] = true;
        int i0 = p[j0],  delta = INF,  j1;
        for (int j=1; j<=m; ++j)
            if (!used[j]) {
                int cur = A[i0][j]-u[i0]-v[j];
                if (cur < minv[j])
                    minv[j] = cur,  way[j] = j0;
                if (minv[j] < delta)
                    delta = minv[j],  j1 = j;
            }
        for (int j=0; j<=m; ++j)
            if (used[j])
                u[p[j]] += delta,  v[j] -= delta;
            else
                minv[j] -= delta;
        j0 = j1;
```

```
    } while (p[j0] != 0);
    do {
        int j1 = way[j0];
        p[j0] = p[j1];
        j0 = j1;
    } while (j0);
}
```

To restore the answer in a more familiar form, i.e. finding for each row $i = 1 \dots n$ the number $ans[i]$ of the column selected in it, can be done as follows:

```
vector<int> ans (n+1);
for (int j=1; j<=m; ++j)
    ans[p[j]] = j;
```

The cost of the matching can simply be taken as the potential of the zero column (taken with the opposite sign). Indeed, as you can see from the code, $-v[0]$ contains the sum of all the values of Δ, i.e. total change in potential. Although several values of $u[i]$ and $v[j]$ could change at once, the total change in the potential is exactly equal to Δ, since until there is an augmenting path, the number of reachable rows is exactly one more than the number of the reachable columns (only the current row i does not have a "pair" in the form of a visited column):

```
int cost = -v[0];
```

Connection to the Successive Shortest Path Algorithm

The Hungarian algorithm can be seen as the Successive Shortest Path Algorithm, adapted for the assignment problem. Without going into the details, let's provide an intuition regarding the connection between them.

The Successive Path algorithm uses a modified version of Johnson's algorithm as reweighting technique. This one is divided into four steps:

- Use the Bellman-Ford algorithm, starting from the sink s and, for each node, find the minimum weight $h(v)$ of a path from s to v.

For every step of the main algorithm:

- Reweight the edges of the original graph in this way: $w(u, v) \leftarrow w(u, v) + h(u) - h(v)$.
- Use Dijkstra's algorithm to find the shortest-paths subgraph of the original network.
- Update potentials for the next iteration.

Given this description, we can observe that there is a strong analogy between $h(v)$ and potentials: it can be checked that they are equal up to a constant offset. In addition, it can be shown that, after reweighting, the set of all zero-weight edges represents the shortest-path subgraph where the main algorithm tries to increase the flow. This also happens in the Hungarian algorithm: we create a subgraph made of rigid edges (the ones for which the quantity $A[i][j] - u[i] - v[j]$ is zero), and we try to increase the size of the matching.

In step 4, all the $h(v)$ are updated: every time we modify the flow network, we should guarantee that the distances from the source are correct (otherwise, in the next iteration, Dijkstra's algorithm might fail). This sounds like the update performed on the potentials, but in this case, they are not equally incremented.

To deepen the understanding of potentials, refer to this article.

Task examples

Here are a few examples related to the assignment problem, from very trivial to less obvious tasks:

- Given a bipartite graph, it is required to find in it **the maximum matching with the minimum weight** (i.e., first of all, the size of the matching is maximized, and secondly, its cost is minimized).
  To solve it, we simply build an assignment problem, putting the number "infinity" in place of the missing edges. After that, we solve the problem with the Hungarian algorithm, and remove edges of infinite weight from the answer (they could enter the answer if the problem does not have a solution in the form of a perfect matching).
- Given a bipartite graph, it is required to find in it **the maximum matching with the maximum weight**.
  The solution is again obvious, all weights must be multiplied by minus one.
- The task of **detecting moving objects in images**: two images were taken, as a result of which two sets of coordinates were obtained. It is required to correlate the objects in the first and second images, i.e. determine for each point of the second image, which point of the first image it corresponded to. In this case, it is required to minimize the sum of distances between the compared points (i.e., we are looking for a solution in which the objects have taken the shortest path in total).
  To solve, we simply build and solve an assignment problem, where the weights of the edges are the Euclidean distances between points.
- The task of **detecting moving objects by locators**: there are two locators that can't determine the position of an object in space, but only its direction. Both locators (located at different points) received information in the form of n such directions. It is required to determine the position of objects, i.e. determine the expected positions of objects and their corresponding pairs of directions in such a way that the sum of distances from objects to direction rays is minimized.
  Solution: again, we simply build and solve the assignment problem, where the vertices of the left part are the n directions from the first locator, the vertices of the right part are the n directions from the second locator, and the weights of the edges are the distances between the corresponding rays.
- Covering a **directed acyclic graph with paths**: given a directed acyclic graph, it is required to find the smallest number of paths (if equal, with the smallest total weight) so that each vertex of the graph lies in exactly one path.
  The solution is to build the corresponding bipartite graph from the given graph and find the maximum matching of the minimum weight in it. See separate article for more details.
- **Tree coloring book**. Given a tree in which each vertex, except for leaves, has exactly $k-1$ children. It is required to choose for each vertex one of the k colors available so that no two

adjacent vertices have the same color. In addition, for each vertex and each color, the cost of painting this vertex with this color is known, and it is required to minimize the total cost.
To solve this problem, we use dynamic programming. Namely, let's learn how to calculate the value $d[v][c]$, where v is the vertex number, c is the color number, and the value $d[v][c]$ itself is the minimum cost needed to color all the vertices in the subtree rooted at v, and the vertex v itself with color c. To calculate such a value $d[v][c]$, it is necessary to distribute the remaining $k-1$ colors among the children of the vertex v, and for this, it is necessary to build and solve the assignment problem (in which the vertices of the left part are colors, the vertices of the right part are children, and the weights of the edges are the corresponding values of d).
Thus, each value $d[v][c]$ is calculated using the solution of the assignment problem, which ultimately gives the asymptotic $\mathcal{O}(nk^4)$.

- If, in the assignment problem, the weights are not on the edges, but on the vertices, and only **on the vertices of the same part**, then it's not necessary to use the Hungarian algorithm: just sort the vertices by weight and run the usual Kuhn algorithm (for more details, see a separate article).
- Consider the following **special case**. Let each vertex of the left part be assigned some number $\alpha[i]$, and each vertex of the right part $\beta[j]$. Let the weight of any edge (i, j) be equal to $\alpha[i] \cdot \beta[j]$ (the numbers $\alpha[i]$ and $\beta[j]$ are known). Solve the assignment problem.
To solve it without the Hungarian algorithm, we first consider the case when both parts have two vertices. In this case, as you can easily see, it is better to connect the vertices in the reverse order: connect the vertex with the smaller $\alpha[i]$ to the vertex with the larger $\beta[j]$. This rule can be easily generalized to an arbitrary number of vertices: you need to sort the vertices of the first part in increasing order of $\alpha[i]$ values, the second part in decreasing order of $\beta[j]$ values, and connect the vertices in pairs in that order. Thus, we obtain a solution with complexity of $\mathcal{O}(n \log n)$.
- **The Problem of Potentials**. Given a matrix $A[1 \dots n][1 \dots m]$, it is required to find two arrays $u[1 \dots n]$ and $v[1 \dots m]$ such that, for any i and j, $u[i] + v[j] \le a[i][j]$ and the sum of elements of arrays u and v is maximum.
Knowing the Hungarian algorithm, the solution to this problem will not be difficult: the Hungarian algorithm just finds such a potential u, v that satisfies the condition of the problem. On the other hand, without knowledge of the Hungarian algorithm, it seems almost impossible to solve such a problem.

> **Remark**
>
> This task is also called the **dual problem** of the assignment problem: minimizing the total cost of the assignment is equivalent to maximizing the sum of the potentials.

Literature

- Ravindra Ahuja, Thomas Magnanti, James Orlin. Network Flows [1993]
- Harold Kuhn. The Hungarian Method for the Assignment Problem [1955]
- James Munkres. Algorithms for Assignment and Transportation Problems [1957]

Practice Problems

- UVA - Crime Wave - The Sequel
- UVA - Warehouse
- SGU - Beloved Sons
- UVA - The Great Wall Game
- UVA - Jogging Trails