Hungarian algorithm

The **Hungarian method** is a combinatorial optimization algorithm that solves the assignment problem in polynomial time and which anticipated later primal–dual methods. It was developed and published in 1955 by Harold Kuhn, who gave the name "Hungarian method" because the algorithm was largely based on the earlier works of two Hungarian mathematicians: Dénes Kőnig and Jenő Egerváry.[1][2]

James Munkres reviewed the algorithm in 1957 and observed that it is (strongly) polynomial.[3] Since then the algorithm has been known also as the **Kuhn–Munkres algorithm** or **Munkres assignment algorithm**. The time complexity of the original algorithm was $O(n^4)$, however Edmonds and Karp, and independently Tomizawa noticed that it can be modified to achieve an $O(n^3)$ running time.[4][5] One of the most popular $O(n^3)$ variants is the Jonker–Volgenant algorithm.[6] Ford and Fulkerson extended the method to general maximum flow problems in form of the Ford–Fulkerson algorithm. In 2006, it was discovered that Carl Gustav Jacobi had solved the assignment problem in the 19th century, and the solution had been published posthumously in 1890 in Latin.[7]

Contents

The problem

Example

In this simple example there are three workers: Paul, Dave, and Chris. One of them has to clean the bathroom, another sweep the floors and the third washes the windows, but they each demand different pay for the various tasks. The problem is to find the lowest-cost way to assign the jobs. The problem can be represented in a matrix of the costs of the workers doing the jobs. For example:

	Clean bathroom	Sweep floors	Wash windows
Paul	$2	$3	$3
Dave	$3	$2	$3
Chris	$3	$3	$2

The Hungarian method, when applied to the above table, would give the minimum cost: this is $6, achieved by having Paul clean the bathroom, Dave sweep the floors, and Chris wash the windows.

Matrix formulation

In the matrix formulation, we are given a nonnegative $n \times n$ matrix, where the element in the i-th row and j-th column represents the cost of assigning the j-th job to the i-th worker. We have to find an assignment of the jobs to the workers, such that each job is assigned to one worker and each worker is assigned one job, such that the total cost of assignment is minimum.

This can be expressed as permuting the rows and columns of a cost matrix C to minimize the trace of a matrix:

$$\min_{L,R} \operatorname{Tr}(LCR)$$

where L and R are permutation matrices.

If the goal is to find the assignment that yields the *maximum* cost, the problem can be solved by negating the cost matrix C.

Bipartite graph formulation

The algorithm is easier to describe if we formulate the problem using a bipartite graph. We have a complete bipartite graph $G = (S, T; E)$ with n worker vertices (S) and n job vertices (T), and each edge has a nonnegative cost $c(i, j)$. We want to find a perfect matching with a minimum total cost.

The algorithm in terms of bipartite graphs

Let us call a function $y : (S \cup T) \to \mathbb{R}$ a **potential** if $y(i) + y(j) \leq c(i, j)$ for each $i \in S, j \in T$. The *value* of potential y is the sum of the potential over all vertices: $\sum_{v \in S \cup T} y(v)$.

The cost of each perfect matching is at least the value of each potential: the total cost of the matching is the sum of costs of all edges; the cost of each edge is at least the sum of potentials of its endpoints; since the matching is perfect, each vertex is an endpoint of exactly one edge; hence the total cost is at least the total potential.

The Hungarian method finds a perfect matching and a potential such that the matching cost equals the potential value. This proves that both of them are optimal. In fact, the Hungarian method finds a perfect matching of **tight edges**: an edge ij is called tight for a potential y if $y(i) + y(j) = c(i, j)$. Let us denote the subgraph of tight edges by G_y. The cost of a perfect matching in G_y (if there is one) equals the value of y.

During the algorithm we maintain a potential y and an orientation of G_y (denoted by $\overrightarrow{G_y}$) which has the property that the edges oriented from T to S form a matching M. Initially, y is 0 everywhere, and all edges are oriented from S to T (so M is empty). In each step, either we modify y so that its value increases, or modify the orientation to obtain a matching with more edges. We maintain the invariant that all the edges of M are tight. We are done if M is a perfect matching.

In a general step, let $R_S \subseteq S$ and $R_T \subseteq T$ be the vertices not covered by M (so R_S consists of the vertices in S with no incoming edge and R_T consists of the vertices in T with no outgoing edge). Let Z be the set of vertices reachable in $\overrightarrow{G_y}$ from R_S by a directed path only following edges that are tight. This can be computed by breadth-first search.

If $R_T \cap Z$ is nonempty, then reverse the orientation of a directed path in $\overrightarrow{G_y}$ from R_S to R_T. Thus the size of the corresponding matching increases by 1.

If $R_T \cap Z$ is empty, then let

$$\Delta := \min\{c(i, j) - y(i) - y(j) : i \in Z \cap S, j \in T \setminus Z\}.$$

Δ is well defined because at least one such edge ij must exist whenever the matching is not yet of maximum possible size (see the following section); it is positive because there are no tight edges between $Z \cap S$ and $T \setminus Z$. Increase y by Δ on the vertices of $Z \cap S$ and decrease y by Δ on the vertices of $Z \cap T$. The resulting y is still a potential, and although the graph G_y changes, it still contains M (see the next subsections). We orient the new edges from S to T. By the definition of Δ the set Z of vertices reachable from R_S increases (note that the number of tight edges does not necessarily increase).

We repeat these steps until M is a perfect matching, in which case it gives a minimum cost assignment. The running time of this version of the method is $O(n^4)$: M is augmented n times, and in a phase where M is unchanged, there are at most n potential changes (since Z increases every time). The time sufficient for a potential change is $O(n^2)$.

Proof that the algorithm makes progress

We must show that as long as the matching is not of maximum possible size, the algorithm is always able to make progress — that is, to either increase the number of matched edges, or tighten at least one edge. It suffices to show that at least one of the following holds at every step:

- M is of maximum possible size.
- G_y contains an augmenting path.
- G contains a **loose-tailed path**: a path from some vertex in R_S to a vertex in $T \setminus Z$ that consists of any number (possibly zero) of tight edges followed by a single loose edge. The trailing loose edge of a loose-tailed path is thus from $Z \cap S$, guaranteeing that Δ is well defined.

If M is of maximum possible size, we are of course finished. Otherwise, by Berge's lemma, there must exist an augmenting path P with respect to M in the underlying graph G. However, this path may not exist in G_y: Although every even-numbered edge in P is tight by the definition of M, odd-numbered edges may be loose and thus absent from G_y. One endpoint of P is in R_S, the other in R_T; w.l.o.g., suppose it begins in R_S. If every edge on P is tight, then it remains an augmenting path in G_y and we are done. Otherwise, let uv be the first loose edge on P. If $v \notin Z$ then we have found a loose-tailed path and we are done. Otherwise, v is reachable from some other path Q of tight edges from a vertex in R_S. Let P_v be the subpath of P beginning at v and continuing to the end, and let P' be the path formed by travelling along Q until a vertex on P_v is reached, and then continuing to the end of P_v. Observe that P' is an augmenting path in G with at least one fewer loose edge than P. P can be replaced with P' and this reasoning process iterated (formally, using induction on the number of loose edges) until either an augmenting path in G_y or a loose-tailed path in G is found.

Proof that adjusting the potential *y* leaves *M* unchanged

To show that every edge in M remains after adjusting y, it suffices to show that for an arbitrary edge in M, either both of its endpoints, or neither of them, are in Z. To this end let vu be an edge in M from T to S. It is easy to see that if v is in Z then u must be too, since every edge in M is tight. Now suppose, toward contradiction, that $u \in Z$ but $v \notin Z$. u itself cannot be in R_S because it is the endpoint of a matched edge, so there must be some directed path of tight edges from a vertex in R_S to u. This path must avoid v, since that is by assumption not in Z, so the vertex immediately preceding u in this path is some other vertex $v' \in T$. $v'u$ is a tight edge from T to S and is thus in M. But then M contains two edges that share the vertex u, contradicting the fact that M is a matching. Thus every edge in M has either both endpoints or neither endpoint in Z.

Proof that *y* remains a potential

To show that y remains a potential after being adjusted, it suffices to show that no edge has its total potential increased beyond its cost. This is already established for edges in M by the preceding paragraph, so consider an arbitrary edge uv from S to T. If $y(u)$ is increased by Δ, then either $v \in Z \cap T$, in which case $y(v)$ is decreased by Δ, leaving the total potential of the edge unchanged, or $v \in T \setminus Z$, in which case the definition of Δ guarantees that $y(u) + y(v) + \Delta \leq c(u, v)$. Thus y remains a potential.

Matrix interpretation

Given n workers and tasks, and an $n \times n$ matrix containing the cost of assigning each worker to a task, find the cost minimizing assignment.

First the problem is written in the form of a matrix as given below

a1	a2	a3	a4
b1	b2	b3	b4
c1	c2	c3	c4
d1	d2	d3	d4

where a, b, c and d are the workers who have to perform tasks 1, 2, 3 and 4. a1, a2, a3, a4 denote the penalties incurred when worker "a" does task 1, 2, 3, 4 respectively. The same holds true for the other symbols as well. The matrix is square, so each worker can perform only one task.

Step 1

Then we perform row operations on the matrix. To do this, **the lowest of all a_i** (i belonging to 1-4) **is taken and is subtracted from each element in that row.** *This will lead to at least one zero in that row* (We get multiple zeros when there are two equal elements which also happen to be the lowest in that row). **This procedure is repeated for all rows**. *We now have a matrix with at least one zero per row.*

As there are n workers and n tasks, adding or subtracting a fixed number to each item in a row or a column will only change the cost of the assignment by that amount; but the minimum cost assignment under old weights will remain a minimum cost assignment under new weights.

Now we try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is zero. As all weights are non-negative, the assignment will be of minimum cost. This is illustrated below.

0	a2'	a3'	a4'
b1'	b2'	b3'	0
c1'	0	c3'	c4'
d1'	d2'	0	d4'

The zeros that are indicated as 0 are the assigned tasks.

Step 2

Sometimes it may turn out that the matrix at this stage cannot be used for assigning, as is the case for the matrix below.

0	a2'	a3'	a4'
b1'	b2'	b3'	0
0	c2'	c3'	c4'
d1'	0	d3'	d4'

In the above case, no assignment can be made. Note that task 1 is done efficiently by both agent a and c. Both can't be assigned the same task. Also note that no one does task 3 efficiently. To overcome this, we repeat the above procedure for all columns (i.e. **the minimum element in each column is subtracted from all the elements in that column**) and then check if an assignment is possible.

In most situations this will give the result, but if it is still not possible then we need to keep going.

Step 3

All zeros in the matrix must be covered by marking as few rows and/or columns as possible. The following procedure is *one way* to accomplish this:

First, assign as many tasks as possible.

- Row 1 has one zero, so it is assigned. The 0 in row 3 is crossed out because it is in the same column.
- Row 2 has one zero, so it is assigned.
- Row 3's only zero has been crossed out, so nothing is assigned.
- Row 4 has two uncrossed zeros. Either one can be assigned, and the other zero is crossed out.

Alternatively, the 0 in row 3 may be assigned, causing the 0 in row 1 to be crossed instead.

0'	a2'	a3'	a4'
b1'	b2'	b3'	0'
0	c2'	c3'	c4'
d1'	0'	0	d4'

Now to the drawing part.

- Mark all rows having no assignments (row 3).
- Mark all columns having zeros in newly marked row(s) (column 1).
- Mark all rows having assignments in newly marked columns (row 1).
- Repeat the steps outlined in the previous 2 bullets until there are no new rows or columns being marked.

×				
0'	a2'	a3'	a4'	×
b1'	b2'	b3'	0'	
0	c2'	c3'	c4'	×
d1'	0'	0	d4'	

Now draw lines through all marked columns and **unmarked** rows.

×				
0'	a2'	a3'	a4'	×
b1'	b2'	b3'	0'	
0	c2'	c3'	c4'	×
d1'	0'	0	d4'	

The aforementioned detailed description is *just one way* to draw the minimum number of lines to cover all the 0s. Other methods work as well.

Step 4

From the elements that are left, find the lowest value. Subtract this from every unmarked element and add it to every element covered by two lines.

This is equivalent to subtracting a number from all rows which are not crossed and adding the same number to all columns which are crossed. These operations do not change optimal assignments.

Repeat steps 3–4 until an assignment is possible; this is when the minimum number of lines used to cover all the 0s is equal to max(number of people, number of assignments), assuming dummy variables (usually the max cost) are used to fill in when the number of people is greater than the number of assignments.

From Kőnig's theorem,[8] the minimum number of lines (minimum Vertex cover [9]) will be n (the size of maximum matching [10]). Thus, when n lines are required, minimum cost assignment can be found by looking at only zeroes in the matrix.

Bibliography

- R.E. Burkard, M. Dell'Amico, S. Martello: *Assignment Problems* (Revised reprint). SIAM, Philadelphia (PA.) 2012. ISBN 978-1-61197-222-1
- M. Fischetti, "Lezioni di Ricerca Operativa", Edizioni Libreria Progetto Padova, Italia, 1995.
- R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice Hall, 1993.
- S. Martello, "Jeno Egerváry: from the origins of the Hungarian algorithm to satellite communication". Central European Journal of Operational Research 18, 47–58, 2010

References

1. Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, **2**: 83–97, 1955. Kuhn's original publication.
2. Harold W. Kuhn, "Variants of the Hungarian method for assignment problems", *Naval Research Logistics Quarterly*, **3**: 253–258, 1956.
3. J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society for Industrial and Applied Mathematics*, **5**(1):32–38, 1957 March.
4. Edmonds, Jack; Karp, Richard M. (1 April 1972). "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". *Journal of the ACM*. **19** (2): 248–264. doi:10.1145/321694.321699 (https://doi.org/10.1145%2F321694.321699).
5. Tomizawa, N. (1971). "On some techniques useful for solution of transportation network problems". *Networks*. **1** (2): 173–194. doi:10.1002/net.3230010206 (https://doi.org/10.1002%2Fnet.3230010206). ISSN 1097-0037 (https://www.worldcat.org/issn/1097-0037).
6. Jonker, R.; Volgenant, A. (December 1987). "A shortest augmenting path algorithm for dense and sparse linear assignment problems". *Computing*. **38** (4): 325–340. doi:10.1007/BF02278710 (https://doi.org/10.1007%2FBF02278710).
7. https://web.archive.org/web/20151016182619/http://www.lix.polytechnique.fr/~ollivier/JACOBI/presentationIEngl.htm

8. [1] (https://en.wikipedia.org/wiki/K%C5%91nig%27s_theorem_(graph_theory))Konig's theorem
9. [2] (https://en.wikipedia.org/wiki/Vertex_cover)minimum vertex cover
10. [3] (https://en.wikipedia.org/wiki/Matching_(graph_theory))matching

External links

- Bruff, Derek, The Assignment Problem and the Hungarian Method (https://web.archive.org/web/20120105112913/http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf) (matrix formalism).
- Mordecai J. Golin, Bipartite Matching and the Hungarian Method (http://www.cse.ust.hk/~golin/COMP572/Notes/Matching.pdf) (bigraph formalism), Course Notes, Hong Kong University of Science and Technology.
- Hungarian maximum matching algorithm (https://brilliant.org/wiki/hungarian-matching) (both formalisms), in Brilliant website.
- R. A. Pilgrim, *Munkres' Assignment Algorithm. Modified for Rectangular Matrices (http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html)*, Course notes, Murray State University.
- Mike Dawes, *The Optimal Assignment Problem (https://web.archive.org/web/20060812030313/http://www.math.uwo.ca/~mdawes/courses/344/kuhn-munkres.pdf)*, Course notes, University of Western Ontario.
- On Kuhn's Hungarian Method – A tribute from Hungary (http://www.cs.elte.hu/egres/tr/egres-04-14.pdf), András Frank, Egervary Research Group, Pazmany P. setany 1/C, H1117, Budapest, Hungary.
- Lecture: Fundamentals of Operations Research - Assignment Problem - Hungarian Algorithm (https://www.youtube.com/watch?v=BUGIhEecipE), Prof. G. Srinivasan, Department of Management Studies, IIT Madras.
- Extension: Assignment sensitivity analysis (with O(n^4) time complexity) (http://www.roboticsproceedings.org/rss06/p16.html), Liu, Shell.
- Solve any Assignment Problem online (http://www.hungarianalgorithm.com/solve.php), provides a step by step explanation of the Hungarian Algorithm.

Implementations

Note that not all of these satisfy the $O(n^3)$ time complexity, even if they claim so. Some may contain errors, implement the slower $O(n^4)$ algorithm, or have other inefficiencies. In the worst case, a code example linked from Wikipedia could later be modified to include exploit code. Verification and benchmarking is necessary when using such code examples from unknown authors.

- C implementation claiming $O(n^3)$ time complexity (https://github.com/maandree/hungarian-algorithm-n3/blob/master/hungarian.c)
- Java implementation claiming $O(n^3)$ time complexity (https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/optimization/HungarianAlgorithm.java)
- Matlab implementation claiming $O(n^3)$ time complexity (https://github.com/USNavalResearchLaboratory/TrackerComponentLibrary/blob/master/Assignment%20Algorithms/2D%20Assignment/assign2D.m) (public domain)
- Python implementation (http://software.clapper.org/munkres/)
- Ruby implementation with unit tests (https://github.com/evansenter/gene/blob/f515fd73cb9d6a22b4d4b146d70b6c2ec6a5125b/objects/extensions/hungarian.rb)
- C# implementation claiming $O(n^3)$ time complexity (https://github.com/antifriz/hungarian-algorithm-n3)
- D implementation with unit tests (port of a Java version claiming $O(n^3)$) (http://www.fantascienza.net/leonardo/so/hungarian.d)
- Online interactive implementation (http://www.ifors.ms.unimelb.edu.au/tutorial/hungarian/welcome_frame.html)
- Serial and parallel implementations. (http://www.netlib.org/utk/lsi/pcwLSI/text/node220.html)
- Matlab and C (http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=6543)
- Perl implementation (https://metacpan.org/module/Algorithm::Munkres)
- C++ implementation (https://github.com/saebyn/munkres-cpp)
- C++ implementation claiming $O(n^3)$ time complexity (http://dlib.net/optimization.html#max_cost_assignment) (BSD style open source licensed)
- MATLAB implementation (http://www.mathworks.com/matlabcentral/fileexchange/20652-hungarian-algorithm-for-linear-assignment-problems--v2-3-)
- C implementation (https://launchpad.net/lib-bipartite-match)
- JavaScript implementation with unit tests (port of a Java version claiming $O(n^3)$ time complexity) (https://github.com/Gerjo/esoteric/blob/master/Hungarian.js)
- Clue R package proposes an implementation, solve_LSAP (https://cran.r-project.org/web/packages/clue/)

- Node.js implementation on GitHub (https://github.com/addaleax/munkres-js)
- Python implementation in scipy package (https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.linear_sum_assignment.html)
