Hungarian algorithm

From Wikipedia, the free encyclopedia

The **Hungarian method** is a combinatorial optimization algorithm which solves the assignment problem in polynomial time and which anticipated later primal-dual methods. It was developed and published by Harold Kuhn in 1955, 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.

James Munkres reviewed the algorithm in 1957 and observed that it is (strongly) polynomial. Since then the algorithm has been known also as **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. Ford and Fulkerson extended the method to general transportation problems. In 2006, it was discovered that Carl Gustav Jacobi had solved the assignment problem in the 19th century, and published posthumously in 1890 in Latin.[1]

Contents

Laymen's Explanation

Say you have three workers: **Jim**, **Steve** & **Alan**. You need to have one of them clean the bathroom, another sweep the floors & the third wash the windows. What's the best (minimum-cost) way to assign the jobs? First we need a matrix of the costs of the workers doing the jobs.

	Clean bathroom	**Sweep floors**	**Wash Windows**
Jim	$1	$2	$3
Steve	$3	$3	$3
Alan	$3	$3	$2

Then the Hungarian algorithm, when applied to the above table would give us the minimum cost it can be done with: Jim cleans the bathroom, Steve sweeps the floors and Alan washes the windows.

Setting

We are given a nonnegative *n*×*n* matrix, where the element in the *i*-th row and *j*-th column represents the

cost of assigning the *i*-th job to the *j*-th worker. We have to find an assignment of the jobs to the workers that has minimum cost.

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 minimum cost.

Let us call a function $y : (S \cup T) \mapsto \mathbb{Q}$ 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 $\sum_{v \in S \cup T} y(v)$. It can be seen that the cost of each perfect matching is at least the value of each potential. The Hungarian method finds a perfect matching and a potential with equal cost/value which proves the optimality of both. In fact it 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*.

The algorithm in terms of bipartite graphs

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 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. The graph G_y changes, but it still contains *M*. 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 needed for

a potential change is $O(n^2)$.

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

$$\begin{bmatrix} a1 & a2 & a3 & a4 \\ b1 & b2 & b3 & b4 \\ c1 & c2 & c3 & c4 \\ d1 & d2 & d3 & d4 \end{bmatrix}$$

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.

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 the other elements 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. 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. This is illustrated below.

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

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

Sometimes it may turn out that the matrix at this stage cannot be used for assigning, as is the case in 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 to assign then the procedure described below must be

followed.

Initially assign as many tasks as possible then do the following (assign tasks in rows 2, 3 and 4)

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

Mark all rows having no assignments (row 1). Then mark all columns having zeros in that row (column 1). Then mark all rows having assignments in the given column (row 3). Repeat this till a closed loop is obtained.

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

From the elements that are left, find the lowest value. Subtract this from all elements that are not struck. Add this to elements that are present at the intersection of two lines. Leave other elements unchanged. Now assign the tasks using above rules. Repeat the procedure till an assignment is possible.

Basically you find the second minimum cost among the two rows. The procedure is repeated until you are able to distinguish among the workers in terms of least cost.

Bibliography

- Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, **2**:83–97, 1955. Kuhn's original publication.
- Harold W. Kuhn, "Variants of the Hungarian method for assignment problems", *Naval Research Logistics Quarterly*, **3**: 253–258, 1956.
- J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society of Industrial and Applied Mathematics*, **5**(1):32–38, 1957 March.
- M. Fischetti, "Lezioni di Ricerca Operativa", Edizioni Libreria Progetto Padova, Italia, 1995.
- R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice Hall, 1993.

References

1. ^ http://www.lix.polytechnique.fr/~ollivier/JACOBI/jacobiEngl.htm

External links

- Mordecai J. Golin, Bipartite Matching and the Hungarian Method, Course Notes, Hong Kong University of Science and Technology.
- R. A. Pilgrim, *Munkres' Assignment Algorithm. Modified for Rectangular Matrices*, Course notes, Murray State University.
  - Or: Step-by-step description of algorithm
- Mike Dawes, *The Optimal Assignment Problem*, Course notes, University of Western Ontario.
- On Kuhn's Hungarian Method - A tribute from Hungary, András Frank, Egervary Research Group, Pazmany P. setany 1/C, H1117, Budapest, Hungary.

Implementations

(Note that not all of these satisfy the $O(n^3)$ time constraint.)

- Python implementation
- Ruby implementation with unit tests
- Online interactive implementation Please note that this implements a variant of the algorithm as described above.
- Graphical implementation with options (Java applet)
- Serial and parallel implementations.
- Implementation in Matlab and C
- Perl implementation
- Lisp implementation
- C++ implementation
- Another C++ implementation with unit tests
- Java implementation (GPLv3)
- Another Java implementation with JUnit tests (Apache 2.0)
- Serial and parallel implementations.

Retrieved from "http://en.wikipedia.org/wiki/Hungarian_algorithm"
Categories: Matching | Combinatorial optimization

Munkres' Assignment Algorithm

Modified for Rectangular Matrices

Assignment Problem - Let *C* be an *nxn* matrix representing the costs of each of *n* workers to perform any of *n* jobs. The assignment problem is to assign jobs to workers so as to minimize the total cost. Since each worker can perform only one job and each job can be assigned to only one worker the assignments constitute an *independent set* of the matrix *C*.

$$C(i,j) = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix} \end{array}$$

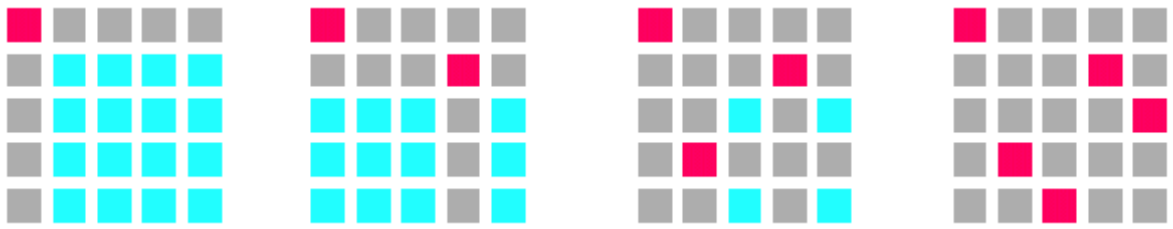
```
Workers = {a,b,c,d}
   Jobs = {p,q,r,s}

An arbitrary assignment
  A = {(a,q),(b,s),(c,r),(d,p)}

Total cost = 23
```

An arbitrary assignment is shown above in which worker *a* is assigned job *q*, worker *b* is assigned job *s* and so on. The total cost of this assignment is *23*. Can you find a lower cost assignment? Can you find the minimal cost assignment? Remember that each assignment must be unique in its row and column.

A brute-force algorithm for solving the assignment problem involves generating all independent sets of the matrix *C*, computing the total costs of each assignment and a search of all assignment to find a minimal-sum independent set. The complexity of this method is driven by the number of independent assignments possible in an *nxn* matrix. There are *n* choices for the first assignment, *n-1* choices for the second assignment and so on, giving n! possible assignment sets. Therefore, this approach has, at least, an exponential runtime complexity.



As each assignment is chosen that row and column are eliminated from consideration. The question is raised as to whether there is a better algorithm. In fact there exists a polynomial runtime complexity algorithm for solving the assignment problem developed by James Munkre's in the late 1950's despite the fact that some references still describe this as a problem of exponential complexity.

The following 6-step algorithm is a modified form of the original Munkres' Assignment Algorithm (sometimes referred to as the Hungarian Algorithm). This algorithm describes to the manual manipulation of a two-dimensional matrix by starring and priming zeros and by covering and uncovering rows and columns. This is because, at the time of publication (1957), few people had access to a computer and the algorithm was exercised by hand.

Step 0: *Create an nxm matrix called the cost matrix in which each element represents the cost of assigning*
one of n workers to one of m jobs. Rotate the matrix so that there are at least as many columns as rows and
let k=min(n,m).

Step 1: *For each row of the matrix, find the smallest element and subtract it from every element in its row. Go to Step 2.*

Step 2: *Find a zero (Z) in the resulting matrix. If there is no starred zero in its row or column, star Z. Repeat*
for each element in the matrix. Go to Step 3.

Step 3: *Cover each column containing a starred zero. If K columns are covered, the starred zeros describe*
a complete set of unique assignments. In this case, Go to DONE, otherwise, Go to Step 4.

Step 4: *Find a noncovered zero and prime it. If there is no starred zero in the row containing this primed zero, Go to Step 5. Otherwise, cover this row and uncover the column containing the starred zero. Continue*
in this manner until there are no uncovered zeros left. Save the smallest uncovered value and Go to Step 6.

Step 5: *Construct a series of alternating primed and starred zeros as follows. Let Z0 represent the uncovered primed zero found in Step 4. Let Z1 denote the starred zero in the column of Z0 (if any). Let Z2 denote the primed zero in the row of Z1 (there will always be one). Continue until the series terminates at a*
primed zero that has no starred zero in its column. Unstar each starred zero of the series, star each primed
zero of the series, erase all primes and uncover every line in the matrix. Return to Step 3.

Step 6: *Add the value found in Step 4 to every element of each covered row, and subtract it from every element of each uncovered column. Return to Step 4 without altering any stars, primes, or covered lines.*

DONE: *Assignment pairs are indicated by the positions of the starred zeros in the cost matrix. If C(i,j) is a starred zero, then the element associated with row i is assigned to the element associated with column j.*
Some of these descriptions require careful interpretation. In Step 4, for example, the possible situations are, that there is a noncovered zero which get primed and if there is no starred zero in its row the program goes onto Step 5. The other possible way out of Step 4 is that there are no noncovered zeros at all, in which case the program goes to Step 6.

At first it may seem that the erratic nature of this algorithm would make its implementation difficult. However, we can apply a few general rules of programming style to simplify this problem. The same rules can be applied to any step-algorithm.

Good Programming Style and Design Practices
1. Strive to create readable source code through the use of blank lines, comments and spacing.
2. Use consistent naming conventions, for variable and constant identifiers and subprograms.
3. Use consistent indentation and always indent the bodies of conditionals and looping constructs.
4. Place logically distinct computations in their own execution blocks or in separate subprograms.
5. Don't use global variables inside subprograms except where such use is clear and improves readability and efficiency.
6. Use local variables where appropriate and try to limit the creation of unnecessary identifiers in the main program.
7. Open I/O files only when needed and close them as soon as they are no longer required.
8. Work to keep the level of nesting of conditionals and loops at a minimum.
9. Use constant identifiers instead of hardwiring for-loop and array ranges in the body of the code with literal
values.
10. When you feel that things are getting out of control, start over. Re-coding is good coding.
By applying Rule 4 to the step-algorithm we decide to make each step its own procedure. Now we can apply Rule 8 by

using a case statement in a loop to control the ordering of step execution.
A old implementation of Munkres - This implementation is