Quality and performance of different implementations of Kuhn-Munkres and Jonker-Volgenant algorithms.

Bogusz Jelinski, 2024, Nov 7th

Abstract

Kuhn-Munkres and Jonker & Volgenant algorithms, which solve assignment problem, have been implemented in different programming languages, the quality and performance of implementations vary significantly and depends on the content of the cost matrix, both sparsity and size wise. The purpose is to show these differences with experiments based on random generated examples of huge models with millions of variables, with integer cost matrix. C/C++, Rust and Python have been verified, fourteen implementations altogether, and compared with GLPK solver. Some implementations should be avoided when models are big and performance is a critical factor. LAPJV is not always the best performer, greedy heuristic can give optimal solution much faster. The source code of the test is available in GitHub for reproducibility.

Introduction

The general assignment problem [1] has many applications in industry and society e.g. logistics, workforce allocation, sports scheduling and computer resource allocation. A taxi company could define the following quadratic assignment model:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \rightarrow MIN \tag{1}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \; for \; j = 1, \dots, n \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \; for \; i = 1, \dots, n \tag{3}$$

$$x_{ij} \in \{0,1\} \; for \; i,j = 1, \dots, n \tag{4}$$

The model can have millions of variables, thousands of customers to be assigned to thousands of cabs. You could use integer solver from cvxopt.glpk [6], the above model has to be reflected in form of matrices and vectors, an example in Python:

```
c = matrix(cost, tc='d')
arr = np.zeros((n*2, n*n))
for i in range(0,n):
    for j in range(0,n):
        arr[i][n*i+j]=1.0
        arr[n+i][n*j+i]=1.0
a=matrix(arr, tc='d')
g=matrix([ [0 for x in range(n*n)] ], tc='d')
b=matrix(1*np.ones(n*2))
h=matrix(0*np.ones(1))
I=set(range(n*n))
B=set(range(n*n))
(status,x) = ilp(c,g.T,h,a,b,I,B)
```

GLPK is not fast enough to solve problems of n=1000 in a satisfactory time, transportation in today's cities needs at least an order of magnitude more. There are faster algorithms based on maximum flow and shortest paths. First Kuhn [2,3] proposed a solution based on work of two Hungarian mathematicians: Dénes Kőnig [4] and Jenő Egerváry [5] (in fact Carl Gustav Jacobi found it earlier), which was revised by Munkres [6], Lawler [7], Bertsekas [8] and many others. In 1987 Jonker & Volgenant [9] came with their shortest path implementation named LAPJV, another variant of the Hungarian method. We are going to compare fourteen implementations, see Table.1. The type description is based on authors view of the affiliation.

Label	Source	Type	Language	Last modified
GLPK	[10]	Branch&bound	Python	2012 Jun
HUN1	[11]	Hungarian	Rust	2020 Jun
HUN2	[12]	Hungarian	Rust	2024 Oct
HUN3	[13]	Hungarian	C	2018 Dec
HUN4	[14]	Hungarian	C	2002 Sep
HUN5	[15]	Hungarian	C++	2016 May
HUN6	[16]	Hungarian	C++	2018 Dec
HUN7	[17]	Hungarian	C++	2021 Apr
HUN8	[18]	Hungarian	Python	2023 May
JV1	[19]	Jonker & Volgenant	Python	2024 Jul
JV2	[20]	Jonker & Volgenant	C++	2024 Jun
JV3	[21]	Jonker & Volgenant	Rust	2021 Dec
JV4	[22]	Jonker & Volgenant	Python	2024 Aug
JV5	[23]	Jonker & Volgenant	Python	2020 Aug
JV6	[24]	Jonker & Volgenant	Python	2023 Dec
LCM	[25]	Greedy	Rust	-

Table 1. Sources of implementations and their characteristics

Experiments

All implementations have been checked (see source code [25]) with the same cost (rating) matrices. The goal was to reach the minimum, some implementations have also the maximum alternative. The scope was not to check all implementations available but to have a sufficient overview. Three dimensions and two integer ranges have been checked, six scenarios altogether for all implementations, although a few values could not be achieved due to an error or too lengthy computation. Only time of computation has been measured, data transfer time is not included. Average is based on five iterations. Tests were executed on 8th generation Intel processor. The ranges were based on real usage scenario – cost matrix depicting taxi distances in a medium sized city, measured in minutes (maximum 30) and seconds (maximum 1800). The results are shown in Table 2., three algorithms were tested more extensively with bigger matrices, see Table 3.

Label	2000x2000		1000x2000		500x8000	
	0..30	**10..1800**	**0..30**	**10..1800**	**0..30**	**10..1800**
HUN1	262	300010	26	34579	42	87
HUN2	23840	628	17269	14729	-	-
HUN4	23	765	30	291	984	1115
HUN5	210	5577	40	270	76	(89)
HUN6	55695	266771	94	732511	2419	-
HUN7	688	596453	(106)	(35409)	(2366)	(2605)
HUN8	23645	614993	2523	1499904	41038	-
JV1	219	255	125	258	-	-
JV2	37	111	67	247	1202	1123
JV3	31	77	59	234	1162	1330
JV4	231	440	234	424	-	-
JV5	226	357	229	365	-	-
LCM	(4)	(1478)	(2)	715	(0)	315

Table 2. Average execution times, depending on size and content of cost matrix [ms]

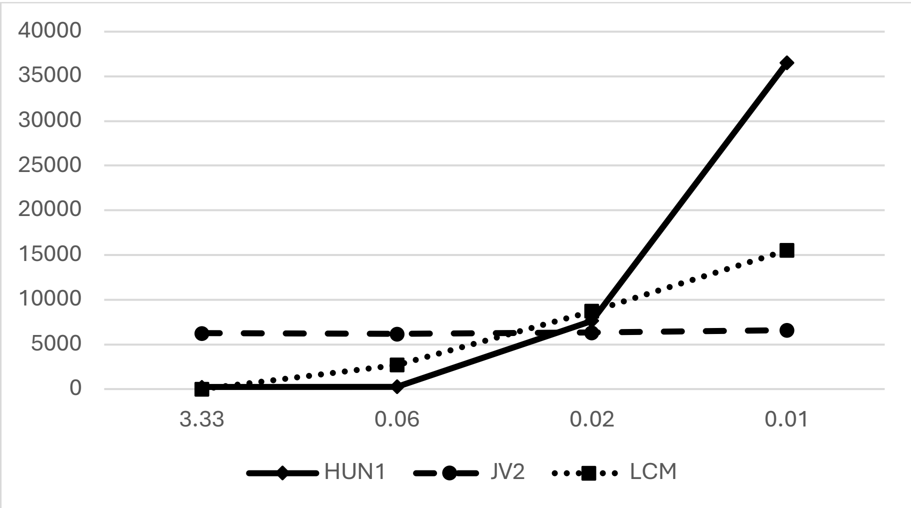

Figure 1. Impact of sparsity in [%] on execution time [ms], same size 1000x16000

	sparse (50%)			0..30			10..1800		
1000 * n	**HUN1**	**JV2**	**LCM**	**HUN1**	**JV2**	**LCM**	**HUN1**	**JV2**	**LCM**
1000	13	9	0	106	11	(1)	25557	18	(274)
2000	23	47	0	26	67	2	34579	247	(715)
4000	42	230	0	63	302	3	24533	485	(1238)
8000	86	1654	0	115	1151	2	1748	1197	(1772)
16000	229	9322	0	235	6249	2	281	6215	1615
24000	253	21423	0	324	14440	2	363	14422	1410
32000	431	50634	0	547	25757	3	589	25424	1496

Table 3. Impact of matrix size on execution time [ms], three cost ranges

Three implementations listed in Table 1. are not shown in Table 2. It takes GLPK 5min to complete a 1000x1000 task. HUN3 is excluded because it fails even with quadratic cost matrices of n=1000 and gives nonoptimal solutions (marked with brackets) with smaller rectangular ones, the same case with HUN7. HUN5 gave invalid solutions in 500x8000/dense scenario - duplicated rows. JV6 gave invalid results. Some authors mention drawbacks in their documentation e.g. "*seems to hang on certain extreme (degenerate?) inputs, especially when the rating matrix is non-square*" ([26]) sometimes it is the software that hints this constraint like JV1 "*matrix must be a square 2D numpy array*" or JV3 "*matrix is not square*". HUN2 requires that "*number of rows must not be larger than number of columns*". One can meet this requirement by adding columns with excessive costs. One author (HUN1) mistrusts his own well-performing implementation and refers to a comparable one (HUN2), the error was due to very small matrix size that was tested.
The best performing implementations, HUN1 and JV2 were given two tests more in order to map their sensitivity to different sparsity (Figure 1) and the size of cost matrix (Table 3). JV3 gave results similar to JV2 but needed a quadratic matrix.

Conclusions

Interesting conclusions can be drawn from the experiments:

- the difference in performance between implementations, even within the same language, is huge.
- the content of cost matrix, the range of values (sparse vs dense) has a vast impact on computation time, the denser the matrix, the longer time is. The more diverse the values, the longer the time.
- the same applies to size of the matrix – some implementations perform better with a square matrix and some with a strongly asymmetric one, which might suggest using multiple ones in an application.
- changing from square to rectangle by reducing the size by half increases the computation time in some implementations, decreases in others.
- a twenty-year-old implementation performs much better than new ones with multithreading.
- low-cost, greedy heuristic gave a solution very close to optimal. The increased cost was equal to average value of four-five cells, both in "minutes" and "seconds" scenarios, circa 0.3% of a randomly selected solution. It gave optimal solution (or the difference was 1/10 of one cell) in rectangular examples for obvious reason – more available minimum values.
- the low-cost heuristic was slower than optimal solutions in the "seconds" data range, and optimal solutions run so fast in the "minute" one, that there is no reason to use LCM – it took HUN1 80ns to complete a 500x15000 case.
- Jonker & Volgenant claimed their "*algorithm to be uniformly faster than the best algorithms from the literature*" (Kuhn-Munkres and derivatives), but it is not true for more asymmetric matrices. The behavior of algorithms is interesting – increasing size requires more time for JV2 and less for HUN1 in the denser scenario.

- if the range of values is small (minutes as the unit of time measurement for a taxi), like in the Kabina taxi dispatcher ([27]), HUN1 is the winner. The work on Kabina open-source dispatcher inspired the research here.
- while benchmarking a solver one should check the result too, not just elapsed time like here [28].

References

[1] Cattrysse DG, Van LN Wassenhove (1992) A survey of algorithms for the generalized assignment problem. Eur J Oper Res 60:260–272

[2] Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly 2(2) (1955) 8397

[3] Kuhn, H.W.: Variants of the hungarian method for assignment problems. Naval Research Logistics Quarterly, 3(253-258) (1956)

[4] Konig, D.: Uber graphen und ihreanwendung aufdeterminantentheorie und mengenlehre. Mathematische Annalen 77(4) (1916) 453465

[5] Egervary, J.: Matrixok kombinatorius tulajdonsagairol [on combinatorial properties of matrices]. Matematikais

[6] Munkres, J.: Algorithms for the assignment and transportation problems. Journal of the Society for Industrial and Applied Mathematics 5(1) (1957) 3238

[7] Lawler, E. L.: Combinatorial Optimization: Networks and Matroids. New York: Holt, Rinehart & Winston 1976.

[8] Bertsekas, D.P.: A new algorithm for the assignment problem. Mathematical Programming 21(1) (1981) 152171 Fizikai Lapok (in Hungarian) 38 (1931) 1628

[9] Bourgeois, F., Lassalle, J.C.: An extension of the munkres algorithm for the assignment problem to rectangular matrices. Communications of the Acm 14 (1971) 802804

[10] https://www.gnu.org/software/glpk/

[11] https://crates.io/crates/hungarian

[12] https://crates.io/crates/pathfinding/4.3.1

[13] https://github.com/xg590/munkres

[14] https://ranger.uta.edu/~weems/NOTES5311/hungarian.c

[15] https://github.com/mcximing/hungarian-algorithm-cpp

[16] https://github.com/phoemur/hungarian_algorithm/tree/master

[17] https://github.com/aaron-michaux/munkres-algorithm.git

[18] https://software.clapper.org/munkres/

[19] https://github.com/src-d/lapjv

[20] https://github.com/yongyanghz/LAPJV-algorithm-c

[21] https://crates.io/crates/lapjv/0.2.1

[22] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html

[23] https://github.com/jdmoorman/laptools

[24] https://github.com/cheind/py-lapsolver

[25] https://github.com/boguszjelinski/munkres.git

[26] http://robotics.stanford.edu/~gerkey/tools/hungarian.html

[27] https://github.com/boguszjelinski/kern

[28] https://github.com/berhane/LAP-solvers