A Heuristic for the Time Constrained Asymmetric Linear Sum Assignment Problem

Peter Brown [1], **Yuedong Yang** [2], **Yaoqi Zhou** [1,2], **Wayne Pullan** [1,*]
[1]School of ICT and [2]Institute for Glycomics, Griffith University, Gold Coast, QLD 4222, Australia.

Abstract

The Linear Sum Assignment Problem is a fundamental combinatorial optimisation problem and can be broadly defined as: given an $n \times m, m \geq n$ benefit matrix $B = (b_{ij})$, matching each row to a different column so that the sum of entries at the row-column intersections is maximised. This paper describes the application of a new fast heuristic algorithm, Asymmetric Greedy Search, to the asymmetric version ($n \neq m$) of the linear sum assignment problem. Extensive computational experiments, using a range of model graphs demonstrate the effectiveness of the algorithm. The heuristic was also incorporated within an algorithm for the non-sequential protein structure matching problem where non-sequential alignment between two proteins, normally of different numbers of amino acids, needs to be maximised.

Contact: w.pullan@griffith.edu.au

Introduction

Linear Sum Assignment Problems (LSAPs) are classical combinatorial optimisation problems that occur in many applications. Basically, the maximisation variant of LSAP arises when, given a matrix of assignment benefits, the requirement is to assign the entity represented by a row of the benefit matrix to the entity represented by a column of the benefit matrix such that all rows are assigned to a single column, at most one row is assigned to a column and the sum of the assignment benefits is maximised. Note that the maximisation variant of LSAP can easily be transformed into a minimisation variant by replacing the benefit matrix with a cost matrix C where element c_{ij} of C is equal to $A - b_{ij}$ with $A = max_{i,j}(c_{ij})$ and the goal is to minimise the total cost of the assignments. As a simple example of the LSAP, the rows of the cost matrix C could represent workers, the columns could represent tasks and each element c_{ij} of C is the cost of assigning worker i to task j.

Formally, given a $n \times m$ benefit matrix B with elements $b_{ij}, i \in I = \{1, \ldots, n\}, j \in J = \{1, \ldots, m\}$ and $m \geq n$ the maximisation variant of the asymmetric LSAP can be defined per Equation 1.

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in I} \sum_{j \in J} b_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{i \in I} x_{ij} \leq 1 \qquad \forall j \in J \\ & \sum_{j \in J} x_{ij} \leq 1 \qquad \forall i \in I \\ & x_{ij} \in \{0, 1\} \qquad \forall i \in I, \forall j \in J \end{aligned} \tag{1}$$

When $n = m$ the problem is referred to as the symmetric LSAP and all columns are assigned to a row but, when $m > n$, the problem is the asymmetric LSAP and there will be unassigned columns in the final solution.

Linear sum assignment problems can be found in fields as varied as bioinformatics (Konovalov, Litow, and Bajema 2005; Zaslavskiy, Bach, and Vert 2009; Milenković et al. 2010), computer vision (Cheng et al. 1996; 2001; Lowe 2004) as well as subproblems within a number of other combinatorial optimisation problems.

One of the first exact algorithms to solve the LSAP was the Hungarian Algorithm (Kuhn 1955) however many other exact algorithms have subsequently been developed (Burkard, Dell'Amico, and Martello 2012). These include the Auction Algorithm (Bertsekas 1988) which is currently considered to be one of the fastest algorithms for finding the optimal solution to the assignment problem. However, for large-scale or complex instances of the assignment problem, exact algorithms are not viable options, particularly within time-constrained applications.

Heuristic algorithms have been applied to many different optimisation problems and most have the characteristics that, when compared to exact methods, they are fast, normally start with a viable solution and iteratively make improvements so at any point in time a viable solution is available. Thus they are suitable for time constrained LSAP applications, particularly when a solution is required that is very close to but not necessarily optimal.

A recent LSAP heuristic algorithm, Deep Greedy Switching (DGS) (Naiem and El-Beltagy 2013) was de-

*to whom correspondence should be addressed

veloped to solve the centralised peer to peer live streaming problem which requires solving instances of the assignment problem with greater than 10,000 persons and objects in less than one minute. The Asymmetric Greedy Search (AGS) algorithm developed in this study shares some similarities with DGS but was developed to solve the asymmetric LSAP as a component of a non-sequential protein structure alignment algorithm where, because of the number of protein pairs to be aligned, time is of the essence and very good solutions rather than an optimal solution are required.

The remainder of this paper is organised as follows: in Section 2, the AGS algorithm developed in this study is presented and discussed while Section 3 contains experimental results for this algorithm as compared to the results attained by the Hungarian and Auction algorithms on model LSAP problems in addition to an evaluation of the results of using AGS for the protein structure alignment problem. Finally, Section 4 contains a conclusion that summarises this paper and discusses opportunities for future research.

Algorithm

AGS is an algorithm for quickly finding solutions to the LSAP problem that are very close to optimal. The details of AGS are shown in Algorithm 1 where, after first generating an initial solution (line 1), the algorithm creates, for each row, the best row $\{R, R^b\}$ and column $\{C, C^b\}$ vectors of possible swaps (lines $2-3$) and performs these in largest benefit to smallest benefit order until there are none remaining which improve the total benefit (lines $5-11$) of the solution. During this improvement phase, the only updates to the best row $\{R, R^b\}$ and column $\{C, C^b\}$ benefit vectors are those for the row(s) involved in the swap. When there is no benefit improvement possible, $\{R, R^b\}, \{C, C^b\}$ are regenerated (lines $12-13$) and the improvement phase (lines $5-11$) is repeated. The algorithm terminates when there are no improving swaps available after $\{R, R^b\}, \{C, C^b\}$ have been regenerated (lines $12-13$).

The functions used in Algorithm AGS are:

- $Initial(n, m)$: Creates the initial assignment vector V by sequentially assigning each of the n rows to the unassigned column which has the maximum benefit for this row.
- $BestRowSwap(B, V)$: Using the assignment vector V, creates the row swap vector R which, for each row, gives the best row to swap column assignments with. Also creates R^b which contains the benefit values obtained by performing the associated R row swap.
- $BestColSwap(B, V)$: Using the assignment vector V, creates the column swap vector C which identifies the best unused column to replace the current column assignment for each row. C^b contains the benefit values obtained by performing the associated column swap for each row.
- $RowSwap(B, V, r)$: For r and $r_s = R_r$, $V_r \leftrightarrow V_{r_s}$ and updates the $R_r, R^b_r, R_{r_s}, R^b_{r_s}, C_r, C^b_r, C_{r_s}, C^b_{r_s}$ values.

Algorithm 1: AGS : Addresses the maximisation LSAP problem.

Input: The $n \times m$ benefit matrix B.
Output: The n length assignment vector V.
Data: R is the best row swap vector, R^b is the benefit values for R, C is the best unused column swap vector and C^b is the benefit values for C.

```
1  V ← Initial (n, m)
2  {R, R^b} ← BestRowSwap (B, V)
3  {C, C^b} ← BestColSwap (B, V)
4  while max_{i∈I}(R^b_i) > 0 ∨ max_{i∈I}(C^b_i) > 0 do
5      while max_{i∈I}(R^b_i) > 0 ∨ max_{i∈I}(C^b_i) > 0 do
6          if max_{i∈I}(R^b_i) > max_{i∈I}(C^b_i) then
7              r ← arg max_{i∈I}(R^b_i)
8              {R, R^b, C, C^b} ← RowSwap (B, V, r)
9          else
10             r ← arg max_{i∈I}(C^b_i)
11             {R, R^b, C, C^b} ← ColSwap (B, V, r)
12     {R, R^b} ← BestRowSwap (B, V)
13     {C, C^b} ← BestColSwap (B, V)
14 return V
```

- $ColSwap(B, V, r)$: $V_r \leftarrow C_r$ and updates the R_r, R^b_r, C_r, C^b_r values.

For AGS, the basic mode of evaluating the total effect of swapping rows ($BestRowSwap$) is $\mathcal{O}(n^2)$ and swapping unused columns ($BestColSwap$) is $\mathcal{O}(n(m-n))$ so these operations need to be minimised. However the improvement phase of AGS (lines $5-11$) only performs, at each iteration, the updates for the rows actually involved in the row and column swaps so these time complexities reduce to $\mathcal{O}(2(n-1)) + \mathcal{O}(2(m-n))$ and $\mathcal{O}(m-n)$ respectively.

Computational Experiments

The major objectives of these computational experiments were to evaluate the performance of AGS as compared to exact algorithms Hungarian (Kuhn 1955) and Auction (Bertsekas 1988), and the heuristic algorithm DGS (Naiem and El-Beltagy 2013) on model LSAP problem instances. In addition, results are presented giving a comparison between a sequential protein structure alignment algorithm (incorporating dynamic programming) and a non-sequential protein structure alignment algorithm (replacing dynamic programming with AGS).

As a basic guide for future comparisons with these results, all experiments were performed on a computer that, when executing the DIMACS Machine Benchmark[1] required 0.17 CPU seconds for r300.5, 1.08 CPU

[1] *dmclique*, ftp://dimacs.rutgers.edu in directory /pub/dsj/clique

seconds for r400.5 and 4.12 CPU seconds for r500.5.

LSAP Model Problems

Three model LSAP benchmarks were utilised: RAND and GEOM as used by (Buš and Tvrdík 2009), and a new LSAP model instance benchmark developed in this study, DMON (**D**eterministic **MO**del Be**N**chmark).

- RAND: The b_{ij} values are random integer values that are uniformly distributed between 1 and C where C is a constant input value.
- GEOM: First $2n$ points are distributed randomly on a 2D graph square of dimensions $[0, C] \times [0, C]$ where C is a constant input value. Based on these geometric vertices, each b_{ij} value is set as the Euclidean distance between point i from the first n generated points and point j from the second n generated points.
- DMON: With regards to the RAND and GEOM benchmarks, the use of randomly generated integer values makes it very difficult to guarantee exact duplication of instances across different computers and software systems. A second disadvantage is that benchmarks that require a complete benefit matrix to be initially generated and stored, such as RAND and GEOM, are not scaleable to very large LSAP problems. To overcome these issues and allow for very large LSAP model instances to be created we use Equation 2 to calculate b_{ij} ($i \in I$, $j \in J$) rather than storing a copy of the DMON benefit matrix.

$$\begin{aligned} \alpha &= i + 1 \\ \delta &= j + 1 \\ \sigma &= \alpha \times m + \delta \\ b_{ij} &= (\sigma + \alpha^2 \times \delta^2) \mod (m + 1) + 1 \end{aligned} \tag{2}$$

We observed that to formulate a non-trivial benchmark benefit matrix, the range of b_{ij} values in each row should be small and the number of duplicates minimised. The sub-expression σ of Equation 2 creates an offset value for each b_{ij} and then a factor which is unique for each b_{ij} is added. The total of these two is then converted to the range $1 \ldots m + 1$. A 50×50 benefit matrix generated using Equation 2 is presented in Figure 1, where values are grey scale coded in the range $1 \ldots 51$ and there appears to be no discernible pattern in the b_{ij} values.

For all experiments involving the RAND and GEOM benchmarks, 100 trials of each algorithm were performed using 100 different RAND and GEOM benefit matrices and the average of each result are presented in the tables. For all tables: presented CPU time is in seconds, the $Benefit\%$ column is the final total benefit attained by AGS expressed as a percentage of that attained by the exact (Hungarian / Auction) algorithms, and R is the ratio of the Auction CPU time to the AGS/DGS CPU time giving a relative measure of AGS/DGS performance. In all trials, the final total benefit results obtained by Hungarian and Auction were identical (but the actual assignments could differ).

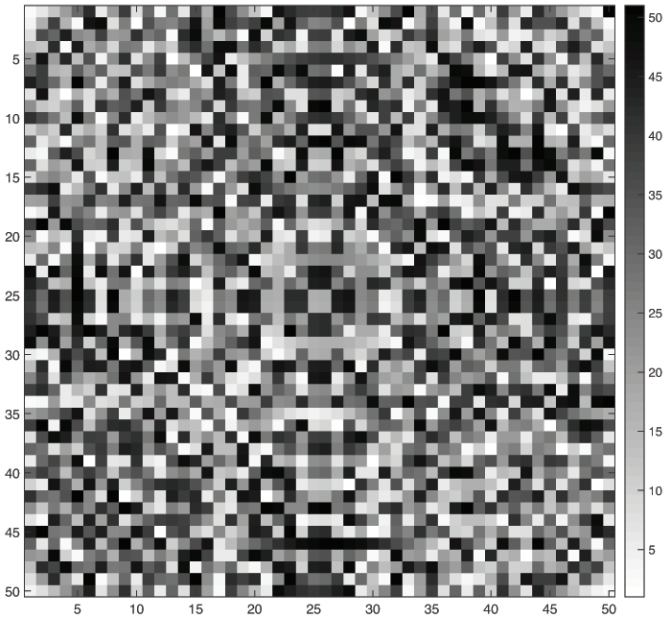

Figure 1: Visualisation of a 50×50 DMON benefit matrix generated by Equation 2 where values are grey scale coded in the range $1 \ldots 51$.

Robustness of AGS. Table 1 shows the performance of AGS on GEOM instances with regard to the initial starting solution (Steps is the number of iterations of the while loop at line 5 of the AGS algorithm shown in Algorithm 1). The initial assignment vector V was generated as follows for each row of the table: $GreedyMin$, a greedy method was used which assigned the column of minimum benefit to each row; $Random$, the initial assignment of rows to columns was performed uniformly randomly; $GreedyMax$, a greedy method was used which assigned the column of maximum benefit to each row. While the amount of computation required varied with each starting point, AGS attained almost identical final results which demonstrates the robustness of the algorithm.

AGS, Hungarian and Auction for RAND and GEOM with $C = 1,000$ and n varying. Tables 2 and 3 present the results attained by the Hungarian,

Table 1: Average results for different AGS $Initial(n, m)$ methods for 100 GEOM instances with $n = 3,000$ and $C = 1,000$.

Initial Method	Start Benefit%	Steps	Final Benefit%	CPU
$GreedyMin$	5.38	36,612	99.98	2.38
$Random$	71.23	31,102	99.99	2.04
$GreedyMax$	99.49	15,901	99.99	1.13

Auction and AGS algorithms for instances of the RAND and GEOM benchmarks. As expected, AGS used less CPU time than the two exact algorithms for all instances and, for RAND, the lowest accuracy achieved by AGS was 99.74% (average = 99.93%) and for GEOM, 99.98% (average = 99.99%) of that attained by Hungarian / Auction, with an average R of 9.59 and 71.00 for RAND and GEOM respectively. Of note from these tables is that Hungarian was slower than Auction for RAND but faster than Auction for GEOM. This is consistent with previous results (Ramakrishnan, Karmarkar, and Kamath 1993) where the Auction algorithm performed poorly over the more complex GEOM problems.

Table 2: Comparative average results for Hungarian, Auction and AGS using 100 different RAND instances for each n with $C = 1,000$.

n	Hungarian	Auction	*AGS*		
	CPU	CPU	CPU	Benefit%	R
1,000	0.05	0.20	0.03	99.74	7.64
2,000	0.58	1.46	0.14	99.83	10.43
3,000	7.49	4.37	0.33	99.87	12.01
4,000	24.34	7.56	0.69	99.88	10.95
5,000	42.64	14.55	1.06	99.90	12.54
6,000	62.08	19.57	1.59	99.91	12.30
7,000	83.02	25.35	2.27	99.93	11.17
8,000	105.78	32.09	3.05	99.93	10.52
9,000	130.47	37.55	4.26	99.94	8.81
10,000	157.40	43.08	4.54	99.95	8.68
11,000	185.10	50.34	5.50	99.95	9.15
12,000	217.45	59.67	6.86	99.95	8.70
13,000	279.17	78.96	10.16	99.96	7.77
14,000	320.08	90.52	12.15	99.96	7.45
15,000	364.30	101.68	15.56	99.96	5.97
16,000	406.33	120.52	16.22	99.96	7.43
17,000	434.87	153.70	15.73	99.96	9.77
18,000	486.01	178.63	16.18	99.97	11.04
19,000	536.89	193.66	18.59	99.97	10.42
20,000	580.92	201.02	21.23	99.97	9.04

Table 3: Comparative average results for Hungarian, Auction and AGS using 100 different GEOM instances for each n with $C = 1,000$.

n	Hungarian	Auction	*AGS*		
	CPU	CPU	CPU	Benefit%	R
1,000	0.46	5.61	0.07	99.99	80.14
2,000	3.38	29.21	0.46	99.99	63.50
3,000	11.20	73.08	1.13	99.99	64.67
4,000	26.24	140.65	2.08	99.99	67.62
5,000	52.05	244.63	3.92	99.99	62.41
6,000	94.41	388.73	5.19	99.99	74.90
7,000	152.62	544.74	7.77	99.99	70.11
8,000	226.30	749.74	10.35	99.99	72.44
9,000	324.41	958.90	13.29	99.98	72.15
10,000	421.55	1,241.66	15.13	99.98	82.07

AGS, Hungarian and Auction for RAND and GEOM with $n = 10,000$ and C varying. Tables 4, 5 and Figure 2 show the results for Hungarian, Auction and AGS on RAND and GEOM problems with $n = 10,000$ and varying C. For the AGS algorithm there is little impact on CPU time by varying C for the RAND problem and a very small linear increase in CPU time for GEOM. For the Auction algorithm, the CPU time required for the RAND problem remained relatively constant as C increased while, for the GEOM problem, the CPU time required increased linearly in proportion to C. However, for the Hungarian algorithm, the CPU time on the RAND problem initially increased as C increased but then started decreasing at $C = 3000$ and remained relatively constant for $C \geq 5000$.

From further investigation, presented in Figure 3 showing the Hungarian algorithm CPU time expressed as a percentage of the maximum CPU time for each of the $n = 2500$, 5000 and 10000 RAND instances, it can be seen that maximum CPU usage occurs at C values of 600, 1200 and 2400 respectively. The ratio n/C for these three cases is constant at $4.1\dot{6}$. As to why this transition seems to occur at $n/C = 4.1\dot{6}$ is currently unresolved and is possibly a side-effect of our implementation. For the GEOM problem, the CPU time requirements of the Hungarian algorithm tended to slowly increase with C.

AGS and DGS for RAND and GEOM instances. Tables 6 and 7 present the results attained by DGS and AGS algorithms for low-cost RAND and GEOM benchmark instances ($C = 1,000$) while Tables 8 and 9

Table 4: Comparative average results for Hungarian, Auction and AGS using 100 different RAND instances for each C with $n = 10,000$ and varying C.

C	Hungarian	Auction	*AGS*		
	CPU	CPU	CPU	Benefit%	R
100	14.24	63.74	5.52	99.99	11.55
200	27.90	47.41	5.32	99.98	8.91
300	42.46	40.23	4.72	99.97	8.52
400	56.84	35.31	6.11	99.95	5.78
500	74.40	38.44	5.19	99.95	7.41
600	92.94	53.75	5.36	99.95	10.03
700	103.11	39.95	4.81	99.95	8.31
800	119.97	39.23	4.81	99.95	8.16
900	140.10	42.91	4.78	99.95	8.98
1,000	157.48	38.48	4.39	99.95	8.77
2,000	391.99	64.96	5.10	99.94	12.74
3,000	423.14	71.72	5.21	99.94	13.77
4,000	134.48	62.56	5.68	99.94	11.01
5,000	48.72	53.13	5.18	99.94	10.26
6,000	34.43	48.99	5.85	99.94	8.37
7,000	27.66	46.33	6.20	99.94	7.47
8,000	23.76	40.07	6.50	99.94	6.88
9,000	21.66	39.40	5.73	99.93	6.87
10,000	20.01	36.67	5.85	99.94	6.27

present the results attained by DGS and AGS algorithms for high-cost RAND and GEOM benchmark instances ($C = 100,000$). For these tables, the benefit and R values of DGS are those detailed in (Naiem and El-Beltagy 2013). These results show that both heuristic methods, AGS and DGS, attain comparable results in terms of accuracy on the RAND instances with a maximum overall benefit difference of 0.1%, which could be a consequence of differences in the randomly generated benefit matrices. For GEOM instances, AGS attained increased or equivalent accuracy to DGS in all cases. On average, across both RAND and GEOM instances, AGS is significantly faster than DGS.

Table 5: Comparative average results for Hungarian, Auction and AGS for 100 different GEOM instances for each C with $n = 10,000$ and varying C.

C	Hungarian	Auction	*AGS*		
	CPU	CPU	CPU	Benefit%	R
100	452.24	608.56	7.91	99.86	76.94
200	453.83	681.04	10.02	99.93	67.97
300	441.47	694.26	11.55	99.95	60.11
400	434.24	747.20	13.05	99.96	57.26
500	444.75	828.85	13.94	99.97	59.46
600	447.33	931.96	13.73	99.97	67.88
700	398.04	996.25	13.63	99.98	73.09
800	431.71	1,076.81	14.24	99.98	75.62
900	424.66	1,062.85	14.81	99.98	71.77
1,000	421.55	1,241.66	15.13	99.98	82.07
2,000	448.59	1,897.64	18.44	99.98	102.91
3,000	462.59	2,600.11	19.33	99.98	134.51
4,000	463.05	3,184.01	20.23	99.98	157.39
5,000	460.85	3,736.96	20.53	99.98	182.02
6,000	459.39	4,246.47	20.95	99.98	202.70
7,000	465.12	4,911.01	21.97	99.98	223.53
8,000	463.34	5,381.95	22.29	99.98	241.45
9,000	464.71	5,878.19	22.43	99.98	262.07
10,000	466.86	6,380.66	22.68	99.98	281.33

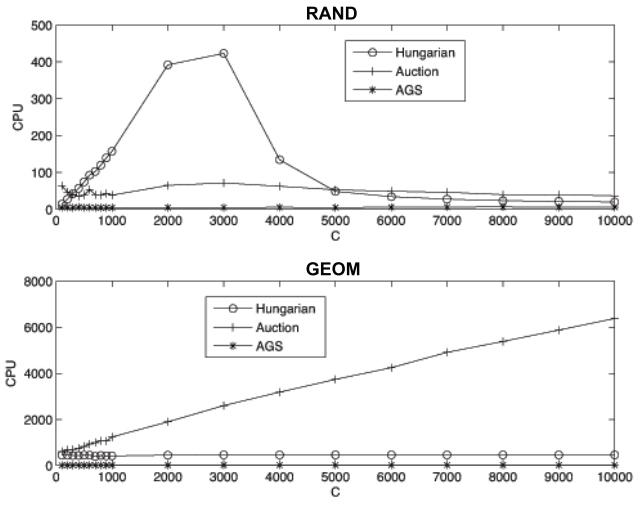

Figure 2: CPU time for Hungarian, Auction and AGS on RAND and GEOM problems as C varies.

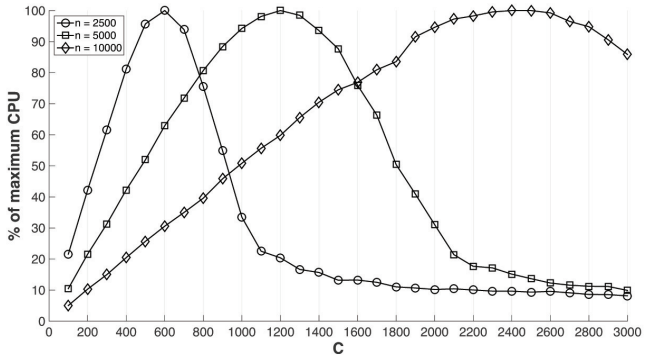

Figure 3: CPU time as a % of maximum CPU time for Hungarian on RAND problems with $n = 2,500$, $n = 5,000$, and $n = 10,000$ as C varies.

AGS and Auction for DMON instances. Tables 10, 11 and 12 present the results attained by Auction and AGS algorithms for instances of the DMON benchmark with low-range n (1,000 - 10,000), mid-range n (11,000 - 25,000) and high-range n (26,000 - 50,000) respectively. Because of excessive memory requirements regarding storage of the reduced benefit matrix, the Hungarian algorithm was applied only to low-range instances of the DMON benchmark. For the purposes of future comparison, the final benefit values achieved by both methods are also presented. These results demonstrate AGS using significantly less CPU time for all instances, with an average R of 10.19. The lowest accuracy achieved by AGS for these experiments

Table 6: Average results for DGS and AGS using 100 different RAND instances for each n with $C = 1,000$.

n	*DGS*		*AGS*	
	Benefit%	R	Benefit%	R
1,000	99.83	5.00	99.73	7.64
3,000	99.92	1.05	99.87	12.01
5,000	99.94	1.22	99.90	12.54
10,000	99.97	0.81	99.95	8.68
15,000	99.98	0.57	99.96	5.97
20,000	99.98	0.70	99.97	9.04

Table 7: Average results for DGS and AGS for 100 different GEOM instances for each n with $C = 1,000$.

n	*DGS*		*AGS*	
	Benefit%	R	Benefit%	R
1,000	99.99	90.00	99.99	80.14
3,000	99.98	9.66	99.99	64.67
5,000	99.98	17.40	99.99	62.41

Table 8: Average results for DGS and AGS for 100 different RAND instances for each n with $C = 100,000$.

n	*DGS*		*AGS*	
	Benefit%	R	Benefit%	R
1,000	99.83	7.50	99.73	12.24
3,000	99.92	2.90	99.87	9.05
5,000	99.94	2.06	99.90	6.37
10,000	99.97	1.49	99.93	5.63
15,000	99.98	0.89	99.95	4.73
20,000	99.99	0.53	99.96	4.82

Table 9: Average results for DGS and AGS for 100 different GEOM instances for each n with $C = 100,000$.

n	*DGS*		*AGS*	
	Benefit%	R	Benefit%	R
1,000	100.00	44.92	100.00	935.61
3,000	100.00	122.93	100.00	1,056.58
5,000	100.00	182.89	100.00	1,188.86

was 99.63% (average = 99.95%) of that attained by Auction.

Table 10: Comparative results for Hungarian, Auction and AGS on DMON instances with n varying between 1,000 and 10,000.

n	Hungarian	Auction	Total	*AGS*		
	CPU	CPU	Benefit	CPU	Benefit (%)	R
1,000	1.65	1.58	996,706	0.56	993,065 (99.63%)	2.82
2,000	11.48	8.67	3,993,253	1.69	3,984,897 (99.79%)	5.13
3,000	8.57	14.69	8,998,414	3.87	8,986,628 (99.87%)	3.80
4,000	19.30	29.67	15,998,014	7.63	15,980,223 (99.89%)	3.89
5,000	118.13	66.40	24,993,592	8.44	24,975,085 (99.93%)	7.87
6,000	324.18	124.92	35,985,644	16.08	35,953,524 (99.91%)	7.77
7,000	65.44	135.80	48,996,904	29.74	48,948,716 (99.90%)	4.57
8,000	951.88	240.76	63,969,932	21.09	63,930,416 (99.94%)	11.42
9,000	111.20	121.12	80,995,627	24.44	80,960,770 (99.96%)	4.96
10,000	1,601.91	382.97	99,974,984	48.86	99,904,388 (99.93%)	7.84

Table 11: Comparative results for Auction and AGS on DMON instances with n varying between 11,000 and 25,000.

n	*Auction*		*AGS*		
	CPU	Benefit	CPU	Benefit (%)	R
11,000	456	120,962,633	87	120,842,782 (99.90%)	5.19
12,000	673	143,978,556	53	143,913,552 (99.95%)	12.70
13,000	492	168,994,102	63	168,922,715 (99.96%)	7.80
14,000	887	195,948,573	71	195,863,349 (99.96%)	12.38
15,000	1,038	224,979,044	62	224,914,164 (99.97%)	16.67
16,000	583	255,992,966	85	255,907,699 (99.97%)	6.81
17,000	1,389	288,961,551	119	288,836,553 (99.96%)	11.63
18,000	1,113	323,950,713	167	323,783,876 (99.95%)	6.65
19,000	1,296	360,991,656	163	360,848,784 (99.96%)	7.91
20,000	1,849	399,918,248	154	399,778,696 (99.97%)	11.96
21,000	1,918	440,990,640	166	440,859,758 (99.97%)	11.53
22,000	2,500	483,944,220	211	483,776,200 (99.97%)	11.85
23,000	4,134	528,844,215	309	528,570,542 (99.95%)	13.37
24,000	2,431	575,989,031	219	575,827,601 (99.97%)	11.07
25,000	3,293	624,945,215	368	624,652,797 (99.95%)	8.93

Non-sequential Protein Structure Alignment using AGS

Proteins are large biological molecules that exist within cells and are the primary enablers for the execution of cellular functionality. Proteins consist of a sequence of amino acids and, as only 20 unique amino acids exist within proteins, there are 20^n different combinations possible for an n amino acid protein. All protein amino acids consisting of a backbone, centred on Carbon Alpha (C_α) atoms, and various side chains which make each amino acid chemically different. The assembly of amino acids comprising a protein is known as its sequence which plays a large role in determining the three-dimensional structure (or shape) of the protein and it is this structure which largely determines the proteins function.

Protein structure alignment is a commonly used technique to identify commonality of function between proteins and addresses the following problem: Given two protein structures P_A and P_B, using some structural metric, determine the three-dimensional alignment of P_A with P_B that maximises that metric's score. All currently developed protein structure alignment algorithms fall into two broad categories: Sequential and Non-Sequential. Sequential structure alignments are performed with the following two constraints: *Exclusivity*, where an amino acid must be aligned with at most one other amino acid; and *Ordering*, where amino acid alignments must not cross-over (residue alignment (i_A, j_A) in P_A mapped to (i_B, j_B) in P_B is valid if and only if $i_A < j_A$ and $i_B < j_B$) (Strickland, Barnes, and Sokol 2005). However, in some cases, an alteration or rearrangement of protein topology through

Table 12: Comparative results for Auction and AGS on DMON instances with n varying between 26,000 and 50,000.

n	*Auction*		*AGS*		
	CPU	Benefit	CPU	Benefit (%)	R
26,000	3,398	675,919,163	747	675,323,611 (99.91%)	4.55
27,000	4,842	728,846,073	548	728,440,681 (99.94%)	8.84
28,000	2,905	783,987,259	790	783,408,254 (99.93%)	3.67
29,000	5,578	840,921,695	456	840,585,721 (99.96%)	12.21
30,000	6,184	899,942,579	407	899,665,764 (99.97%)	15.16
31,000	4,689	960,921,032	333	960,707,956 (99.98%)	14.06
32,000	3,647	1,023,960,294	368	1,023,738,414 (99.98%)	9.89
33,000	5,023	1,088,895,496	429	1,088,613,346 (99.97%)	11.70
34,000	5,349	1,155,909,957	661	1,155,443,340 (99.96%)	8.09
35,000	5,084	1,224,925,942	455	1,224,653,756 (99.98%)	11.17
36,000	7,527	1,295,842,361	534	1,295,495,016 (99.97%)	14.09
37,000	8,299	1,368,900,563	1,180	1,368,175,590 (99.95%)	7.03
38,000	7,598	1,443,842,500	469	1,443,566,569 (99.98%)	16.19
39,000	9,401	1,520,911,693	909	1,520,384,690 (99.97%)	10.34
40,000	13,223	1,599,776,977	656	1,599,403,008 (99.98%)	20.15
41,000	9,008	1,680,845,908	720	1,680,433,381 (99.98%)	12.51
42,000	11,051	1,763,894,612	605	1,763,577,950 (99.98%)	18.25
43,000	9,574	1,848,936,385	812	1,848,503,543 (99.98%)	11.78
44,000	7,534	1,935,905,441	1,425	1,935,100,926 (99.96%)	5.29
45,000	10,257	2,024,914,185	900	2,024,432,377 (99.98%)	11.39
46,000	17,797	2,115,881,447	1,215	2,115,251,603 (99.97%)	14.64
47,000	8,346	2,208,941,452	1,000	2,208,429,219 (99.98%)	8.34
48,000	12,205	2,303,894,372	888	2,303,463,349 (99.98%)	13.73
49,000	12,924	2,400,907,263	953	2,400,471,687 (99.98%)	13.55
50,000	16,392	2,499,863,783	1,008	2,499,360,309 (99.98%)	16.25

Figure 4: Structure alignments of protein pair 1PUGB–1PKPA2, includes sequential alignment from SPalign (left) and non-sequential alignment from SPalignNS (right).

Table 13: Alignment performance comparison for the protein pair 1PUGB–1PKPA2 showing the number of aligned residues (Nali), root mean square deviation (RMSD), structure overlap (SO) and the total CPU time.

	Nali	RMSD (Å)	SO (%)	CPU (s)
SPalign	41	2.75	44.59	0.46
SPalignNS	49	1.70	66.22	0.88

either evolutionary means or circular / non-cyclic permutations can create proteins that may be structurally similar but have different ordering of residue sequences. Non-Sequential structure alignments are required in this situation and, while they retain the Exclusivity constraint, they are not restricted by the Ordering constraint.

The non-sequential protein structure alignment tool SPalignNS was created from the sequential protein structure alignment tool SPalign (Yang et al. 2012) by replacing the dynamic programming (which is restricted to sequential alignments) component with the AGS algorithm. In terms of aligning a pair of protein structures non-sequentially, we generate a benefit matrix using SP-score (Yang et al. 2012) as the objective function for each residue from P_A aligned with each residue from P_B. Attained assignment solutions then directly represent the set of individual protein-protein pairs of residue alignments.

The performance of SPalignNS was compared with the SPalign (sequential) results utilising metrics of the number of aligned amino acids (Nali), Root Mean Square Deviation (RMSD), and Structure Overlap (SO). RMSD is calculated using the amino acid alignment mappings, and provides an overall indication of how well the matched aligned residues contribute to the three-dimensional structure similarity. However this metric has a dependance on the alignment length (Guyon and Tuffery 2014). SO is defined as the percentage value of all aligned residues that are within 3.5Å of each other after superimposition. The overall protein structure alignment goal is to minimise RMSD (geometric divergence) whilst maximising Nali and SO (coverage).

We applied both SPalign and SPalignNS to the protein pair, 1PUGB–1PKPA2 that are related to each other structurally by multiple loop permutation (Dai and Zhou 2011). Comparative metrics for this alignment can be found in Table 13 and it is pictorially shown in Figure 4 for SPalign (left) and SPalignNS (right). It can be seen that the sequential alignment has only successfully one of the three central β-sheets. By aligning non-sequentially with SPalignNS it can be seen from the right portion of the figure that all of the central β-sheets have been matched to a corresponding β-sheets. This yields a significant improvement of SO, increasing by over 20% for a total of 66.22%. Further details regarding the performance of SPalignNS are available in (Brown et al. 2015).

Conclusion

In this paper, a new heuristic algorithm was proposed for the asymmetric linear sum assignment problem. Furthermore we propose a new method for deterministically generating model benefit matrices for evaluating future LSAP algorithms. The results obtained by this algorithm was evaluated against exact solutions obtained on model benefit matrices having up to $50,000$ rows and columns.

The LSAP algorithm implemented in this paper was targeted more at efficiency while still obtaining results which are very close to the optimal result. Typically the results obtained were within 0.1% of optimal and the speed-up generally at least a factor of 10. Future research directions include exploring other variants of LSAP and developing parallel implementations of AGS.

Acknowledgement

The authors would like to thank the anonymous referees whose questions and suggestions resulted in a considerable improvement of this paper.

References

Bertsekas, D. 1988. The auction algorithm; a distributed relaxation method for the assignment problem. *Annals of Operations Research* 14(1):105–123.

Brown, P.; Pullan, W.; Yang, Y.; and Zhou, Y. 2015. Fast and accurate non-sequential protein structure alignment using a new asymmetric linear sum assignment heuristic. *Bioinformatics* btv580.

Burkard, R.; Dell'Amico, M.; and Martello, S. 2012. *Assignment Problems*. Philadelphia, PA 19104-2688 USA: Society for Industrial and Applied Mathematics, revised reprint edition.

Buš, L., and Tvrdík, P. 2009. Towards auction algorithms for large dense assignment problems. *Computational Optimization and Applications* 43(3):411–436.

Cheng, Y.; Wu, V.; Collins, R.; Hanson, A.; and Riseman, E. t. 1996. Maximum-weight bipartite matching technique and its application in image feature matching. In *SPIE Conference on Visual Communication and Image Processing*, 1358–1379.

Cheng, Y.; Wang, X.; Collins, R.; Riseman, E.; and Hanson, A. 2001. Three-dimensional reconstruction of points and lines with unknown correspondence across images. *International Journal of Computer Vision* 45(2):129–156.

Dai, L., and Zhou, Y. 2011. Characterizing the existing and potential structural space of proteins by large-scale multiple loop permutations. *Journal of Molecular Biology* 408(3):585–595.

Guyon, F., and Tuffery, P. 2014. Fast protein fragment similarity scoring using a binet-cauchy kernel. *Bioinformatics* 30(6):784–791.

Konovalov, D.; Litow, B.; and Bajema, N. 2005. Partition-distance via the assignment problem. *Bioinformatics* 21(10):2463–2468.

Kuhn, H. W. 1955. The hungarian method for the assignment problem. *Naval research logistics quarterly* 2(1-2):83–97.

Lowe, D. G. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision* 60(2):91–110.

Milenković, T.; Ng, W. L.; Hayes, W.; and Pržulj, N. 2010. Optimal network alignment with graphlet degree vectors. *Cancer informatics* 9:121.

Naiem, A., and El-Beltagy, M. 2013. On the optimality and speed of the deep greedy switching algorithm for linear assignment problems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, 1828–1837. IEEE.

Ramakrishnan, K.; Karmarkar, N.; and Kamath, A. 1993. An approximate dual projective algorithm for solving assignment problems. *Network Flows and Matching: First DIMACS Implementation Challenge* 12:431–451.

Strickland, D. M.; Barnes, E.; and Sokol, J. S. 2005. Optimal protein structure alignment using maximum cliques. *Operations research* 53(3):389–402.

Yang, Y.; Zhan, J.; Zhao, H.; and Zhou, Y. 2012. A new size-independent score for pairwise protein structure alignment and its application to structure classification and nucleic-acid binding prediction. *Proteins* 80:2080–2088.

Zaslavskiy, M.; Bach, F.; and Vert, J.-P. 2009. Global alignment of protein–protein interaction networks by graph matching methods. *Bioinformatics* 25(12):1259–1267.