ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

INDIAN JOURNAL OF SCIENCE AND TECHNOLOGY
January 2020, Vol 13(02), 229 – 247
DOI: 10.17485/ijst/2020/v13i02/149446,

A Kidney Exchange Matching Application Using the Blossom and Hungarian Algorithms for Pairwise and Multiway Matching

Juan Miguel J. Bawagan*

Institute of Computer Science College of Arts and Sciences (CAS), University of the Philippines Los Baños (UPLB), Laguna 4031, Philippines

Article Type: Article

Article Citation: Juan Miguel J. Bawagan. A kidney exchange matching application using the blossom and Hungarian algorithms for pairwise and multiway matching. *Indian Journal of Science and Technology*. 2020; 13(02), 229–247. DOI: 10.17485/ijst/2020/v013i02/149446

Received date: December 5, 2019

Accepted date: December 19, 2019

***Author for correspondence:**
Juan Miguel J. Bawagan
jjbawagan@up.edu.ph Institute of Computer Science College of Arts and Sciences (CAS), University of the Philippines Los Baños (UPLB), Laguna 4031, Philippines

Abstract

Background/objectives: Patients of kidney failure sometimes have incompatible donors. This study proposes an application to get the best matchings based on scoring data. **Methods:** For pairwise matching, we created a new graph from the original scoring matrix. This graph ensures pairwise matchings. To find an optimal matching, we used the Blossom algorithm. For multiway matching, we interpreted the scoring matrix as an assignment problem. For this, we used the Hungarian algorithm. The application was created using Python, NetworkX, NumPy, and PySimpleGUI. The app uses CSV files as input. **Findings:** Both algorithms made for polynomial run times. Matching is fast and is guaranteed to be optimal. The application itself gets instantaneous results even for large donor-patient matrices. **Application/improvement:** Hospitals or organizations can use this application for their kidney matching programs.

Keywords: Blossom Algorithm, Hungarian Algorithm, Kidney Failure, Donor Matching.

1. Introduction

Chronic kidney disease is the gradual loss of kidney function. This can progress to a state called end-stage renal disease (ESRD). This happens when the kidneys no longer meet the body's needs [1]. At this point, kidney failure is usually permanent. Those who have ESRD can choose from two treatment options: dialysis or transplantation. Dialysis is a treatment that emulates the kidney's functions. Dialysis centers offer machines to filter a patient's blood. Unfortunately, ESRD patients will need dialysis for the rest of their lives. For those who cannot adapt to dialysis, their only hope is transplantation.

In [2] the Philippines, there have been at least 2500 kidney transplants so far. Currently, over 7000 ESRD patients are on waiting lists for cadaver kidneys. In the U.S, there are over 100,000 [3]. Median waiting time is 3.6 years. Because of this long wait time, doctors and patients turned to live-donor transplants.

	patient1	patient2	patient3	patient4	patient5
donor1	0	8	3	7	8
donor2	2	0	4	2	0
donor3	7	5	0	8	9
donor4	6	5	7	0	1
donor5	2	8	8	5	0

FIGURE 1. A sample scoring. Shown here is the matching score (higher is better) for each donor to a patient. If a donor is incompatible to a patient (or wasn't compared at all), the score written is zero.

A healthy person can survive and remain healthy with only one kidney. That person can choose to donate one of their kidneys to a patient. Unfortunately, donors aren't always compatible with their intended recipients. Incompatibilities may be immunological or due to blood type. To address this problem, exchange programs were created. These programs arrange exchanges between two or more incompatible donor-patient pairs.

Let's say that we have two of them, donor-patient X and donor-patient Y. Assuming compatibility, donor X donates to patient Y. Then donor Y donates to patient X. This can be extended to a larger pool of donor-patient pairs. The larger the pool, the higher the chance of getting a compatible exchange. Not only that, we can get a high scoring exchange. The primary criteria for scoring are blood type and tissue type. The Alliance for Paired Kidney Donation (AKPD) also scores for travel distance, age, and antigen mismatches among many others. In the US, AKPD and other organizations have national kidney exchange programs. As of now, there is no kidney exchange program in the Philippines.

Before a kidney exchange, donors are matched to prospective patients. These matches are tested for compatibility. The higher the compatibility, or matching score, the higher the success of an exchange. This would give us an idea if a patient's body will accept the donated kidney. With a pool of matching scores (Figure 1), we could find an optimal matching. This would give us the best donor-patient exchanges.

The goal is to create a free, fast, and easy-to-use application for kidney-exchange matching. Here we provide two proven solutions adapted for pairwise and multiway matching.

2. Pairwise Matching

A pairwise matching is a strict pair-to-pair matching. When donor X is matched to patient Y, donor Y should be matched to patient X (Figure 2).

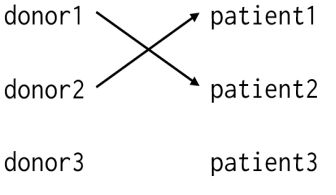


FIGURE 2. Here we see three donor-patient pairs. In pairwise matching, a pair is matched to another pair.

In [4] the past, kidney exchanges were restricted to this scheme. One of the reasons for this is logistics. A pairwise exchange involves four simultaneous surgeries. Increasing to a 3-pair exchange would involve six simultaneous surgeries. Further increases would yield larger logistical problems. So how do we ensure this strict pair-to-pair matching?

2.1. Creating a Graph that Ensures a Pair-to-pair Matching

First, treat each donor-patient pair as a single entity. Instead of matching donors to patients, we match pairs instead. *Caveat*: this assumes that donors are incompatible with their paired patients. Thus, we won't have self-pairings. Each donor-patient pair will be a vertex in our graph G. Next, form the edges. Each edge represents a possible matching between two donor-patient pairs. The higher the edge weight, the better the pairwise matching.

For this application, we form edges (and edge weights) using either:

a) The average score of the pairwise matching (Figure 3). This is the default option.

edge_weight = AVERAGE (score (donorX -> patientY), score (donorY -> patientX))

b) A modified average score. This gives a higher weight to matches that have low variance with each other.

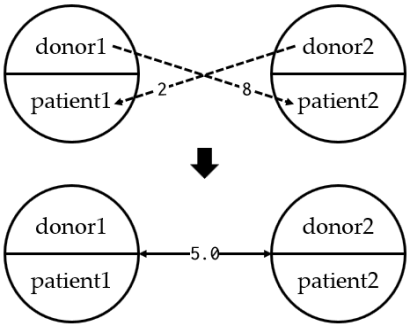


FIGURE 3. Here we see donor-patient pairs treated as vertices. Matching vertices ensures a pairwise matching. The edge between these vertices is the average of their original matching scores (see Figure 1).1

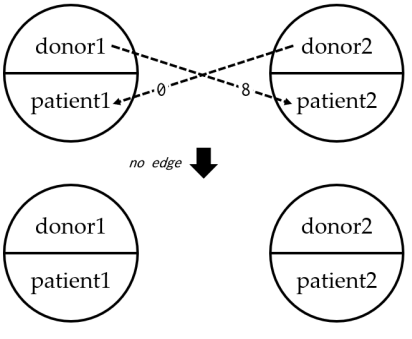


FIGURE 4. There is an option to not produce edges with an incompatible match (zero score). This makes the whole pairwise matching incompatible.

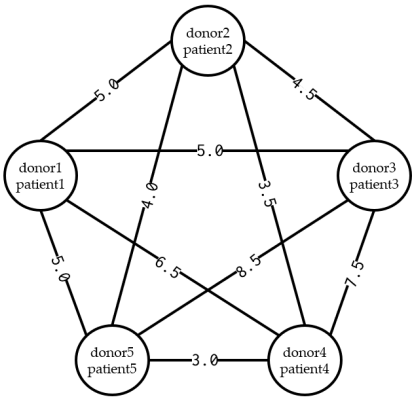


FIGURE 5. Data from Figure 1 converted to a pairwise matching graph using default options.

edge_weight = AVERAGE (score (donorX -> patientY), score (donorY -> patientX)) * [MAX_SCORE - ABS_DIFF (score (donorX -> patientY), score (donorY -> patientX))]

There is an extra option to omit edges if there is an incompatible match in the original scoring (Figure 4). This makes the pairwise matching also incompatible. The user may choose this to make the graph sparser. Otherwise, the app tries to get the best possible matching regardless of incompatibles.

Using the sample scoring from Figure 1 would result in the following graph (Figure 5).

2.2. Edmonds's Blossom Algorithm

We now have a new graph *G* composed of the donor-patient pairs (vertices) and the pairwise scores (edges). Next is to find the best possible matchings. Let's first define the following:

A *matching M* is a subset of edges in G such that none of the edges in M share a common vertex. Matched vertices are vertices in M while unmatched vertices are those not in M.

A *maximal matching* is a matching such that if any edge not in M is added to M, it is no longer a matching (Figure 6).

A *maximum matching* is a matching that contains the largest possible number of edges (or vertices) (Figure 6). This is the largest maximal matching on G.

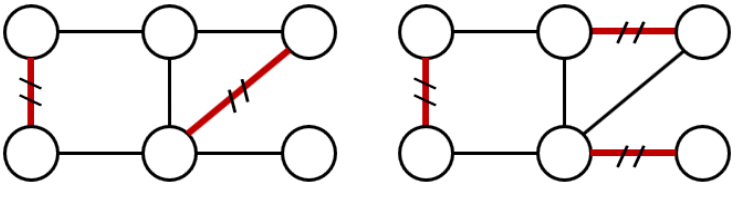


FIGURE 6. A maximal matching (left). Adding another edge would no longer make it a matching. A maximum matching (right). Here the matching contains the largest possible number of edges.

A *maximum weight matching* is a matching in a weighted graph where the sum of the weights is the largest.

As we can see (Figure 5), our problem is now a maximum weight matching problem. We want to get the pairwise matches that give us the best total score. To do this, we'll use *Jack Edmonds's Blossom algorithm* [5]. This algorithm is a bit complex, so we'll try to explain it as simple as possible.

The Blossom algorithm computes for a maximum matching on a general graph G = (V, E). It improves a matching by iteratively finding augmenting paths in G. But what are augmenting paths?

Given a matching M, an *alternating path* is a path whose edges alternate from being in M to not in M. An *augmenting path* is an alternating path that starts and ends on unmatched vertices [6] (also known as free vertices). According to Berge's lemma [7], a matching M is maximum if and only if there is no augmenting path in G. If an augmenting path exists, we can invert it to produce a better matching (Figure 7). The Blossom algorithm finds these paths using a modified Breadth-First Search (BFS). It does this efficiently by finding and contracting *blossoms*.

While doing BFS, the algorithm might encounter odd-length cycles (Figure 8). These are called *blossoms*. Blossoms consist of $2k + 1$ edges where exactly k belongs to M. We call the path from the BFS root to the blossom, the *stem*. The last edge of this stem must always be in M. The algorithm *contracts* these blossoms to form a supernode (Figure 9).

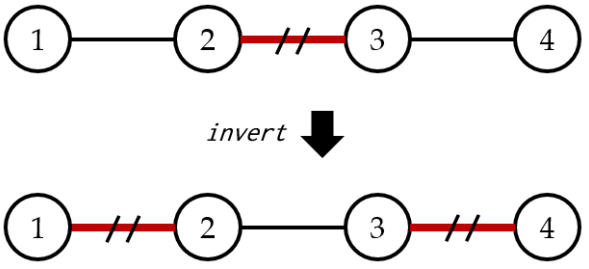


FIGURE 7. An augmenting path (top). The Blossom algorithm finds these paths and inverts them to produce better matchings (bottom).

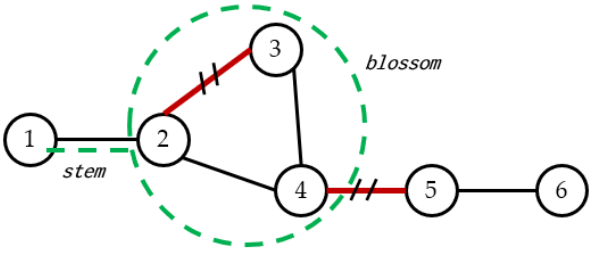


FIGURE 8. A graph with an initial matching (highlighted in red with strokes). Encircled is an odd-length cycle called a blossom. Assuming that vertex 1 is the root of the BFS, edge 1–2 is the stem of this blossom.

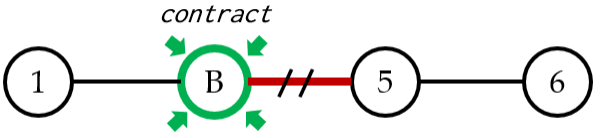


FIGURE 9. Contracting a blossom into a supernode makes the graph smaller.

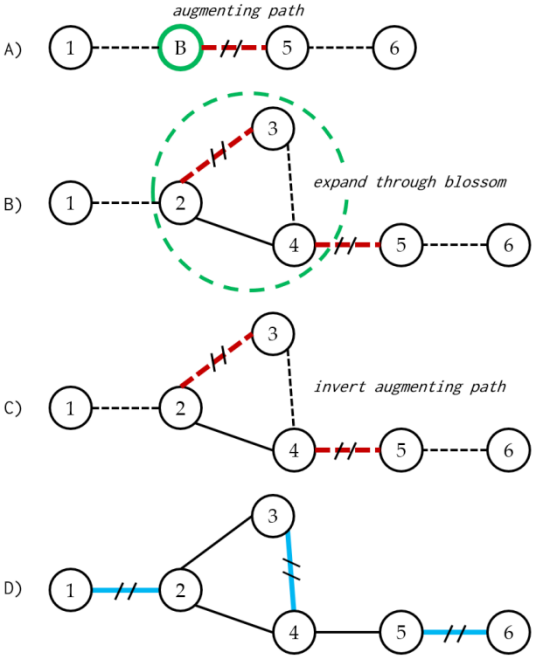


FIGURE 10. A) We find an augmenting path in G' (dotted line). B) The augmenting path is reconstructed through the expanded blossom. C) The augmenting path is inverted. D) A new better matching M'.

This creates a smaller graph G' that makes it easier to find augmenting paths. This is the core aspect of Edmonds's Blossom algorithm.

Once the algorithm does find an augmenting path, it will try to invert it. If the augmenting path includes a supernode, it first expands the supernode. The augmenting path is then reconstructed through the blossom. After this, the augmenting path is inverted to produce a new matching M' (Figure 10). The Blossom algorithm terminates when there are no more augmenting paths in G. For a more detailed explanation.

2.3. Pseudocode of the Blossom Algorithm

```
BEGIN
 WHILE F != ∅ DO  (*set of free nodes F*)
  pick r ∈ F
queue.push(r)      (*BFS queue*)
```

```
 T ← ∅        (*BFS tree T*)
T.add(r)
 WHILE queue != ∅
  v ← queue.pop()
  FOR ALL neighbors w of v DO
  IF w ∉ T AND w matched THEN
T.add(w)
T.add(mate(w))
queue.push(mate(w))
  ELSE IF w ∈ T AND even-length cycle detected THEN
   CONTINUE
  ELSE IF w ∈ T AND odd-length cycle detected THEN
   contract cycle
  ELSE IF w ∈ F THEN
   expand all contracted nodes
   reconstruct augmenting path
   invert augmenting path
END
```

2.4. An Example Using the Blossom Algorithm

Let's start with a simple graph containing four donor-patient pairs. All vertices are unmatched (Figure 11).

Starting from the first unmatched vertex (1), we try to find an augmenting path via BFS. From the root node, we construct a tree of alternating paths (BFS tree). This stops when we reach another unmatched vertex (2). The neighbor (2) is already an unmatched vertex, thus we have an augmenting path. Take note that a single edge containing two unmatched vertices is an augmenting path (Figure 12).

We invert the augmenting path to create a new matching. Now we have edge 1–2 in our matching. Since there are still unmatched vertices, we again search for augmenting paths. We start BFS at the next unmatched vertex (3) (Figure 13).

Vertex 3's neighbor is already matched, so we add the neighbor and its pair to the BFS tree. We check its next neighbor and we see that there is a cycle. This cycle has an odd

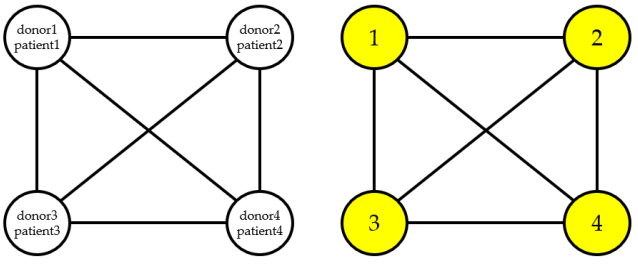


FIGURE 11. A pairwise graph containing four pairs. Unmatched vertices are in yellow.

number of edges thus we found a blossom (Figure 14). We contract the blossom (B) to form a smaller graph G'. From B, we continue doing BFS (Figure 15).

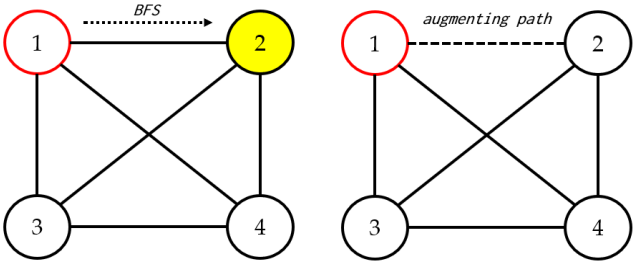


FIGURE 12. Searching for an augmenting path using BFS. The root of the BFS is colored red.

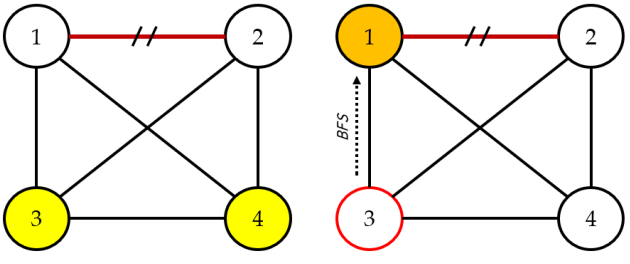


FIGURE 13. Augmenting paths are inverted (left). We repeat the search for augmenting paths (right). Vertex 1 (orange) is already matched.

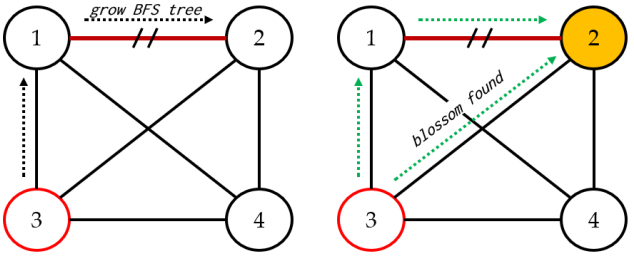


FIGURE 14. BFS tree is augmented (left) and we find a blossom (right).

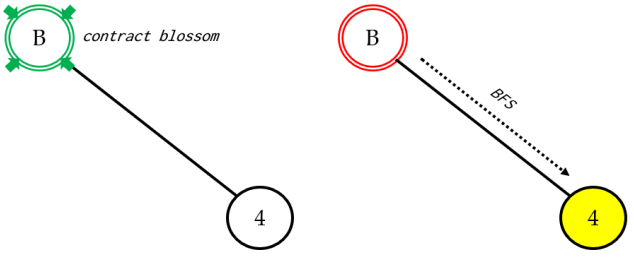


FIGURE 15. Blossom is contracted (left) and the search continues (right).

B's neighbor (4) is an unmatched vertex, thus there is an augmenting path (B-4). The augmenting path runs between the root of the BFS (3) to vertex 4. We expand the blossom to get the whole augmenting path (3–1–2–4) (Figure 16).

We invert the augmenting path. The number of edges in the new matching is increased by one (1–3, 2–4). Thus, we have an improved matching M'. Since there are no more unmatched vertices, the algorithm terminates. We have found the largest possible matching (Figure 17).

2.5. Running Time and Modifications for Weighted Graphs

Let n be the number of vertices and m the number of edges. In Jack Edmonds's original paper, he calculated the running time of the Blossom algorithm as . The BFS implementation by Riedl runs at .

As you may have seen, the Blossom algorithm finds a matching in unweighted graphs. There are various implementations for it to work on weighted graphs. But the core aspect remains the same.

An example implementation:

Start with an empty matching. In each stage, find an augmenting path with the largest weight increase. After k stages, we'll have a matching of maximum weight among matchings of size k.

A more efficient approach uses the primal-dual method. This is taken from the duality theory of linear programming. Galil derived the running time as . Our application uses this. For more information, see Galil, 1986.

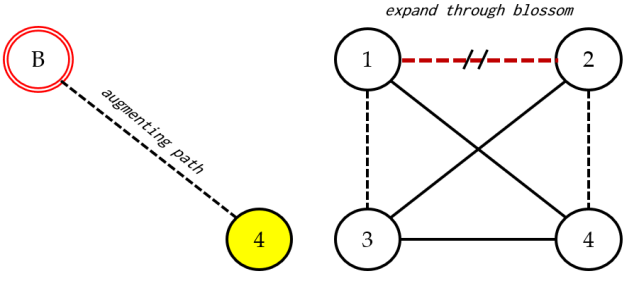


FIGURE 16. An augmenting path is found (left) and is expanded through the blossom (right).

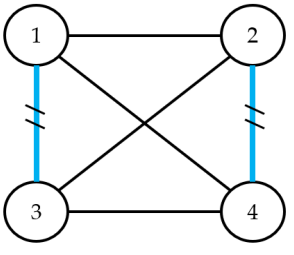


FIGURE 17. A new maximal matching is found.

3. Multiway Matching

Unlike pairwise matching, multiway matching has no pair-to-pair restrictions. If donor X is matched to patient Y, donor Y may or may not be matched to patient X (Figure 18).

This produces a higher matching score at the cost of logistical problems. Long chains can occur which would make simultaneous operations improbable. In 2007, a rare six-way simultaneous operation did occur and was a success [8]. The hospital needed six surgical teams to do the operation.

If simultaneity is not an issue, multiway matching reduces risk from broken links. In pairwise matching, if a pair becomes unavailable so does its matched pair. In multiway matching, that pair can be replaced since it's non-simultaneous.

3.1. Multiway Matching as an Assignment Problem

In multiway matching, we can simply assign a donor to a patient. With the priority being, getting the best total score. This is the assignment problem.

The assignment problem is also known as minimum weight matching in bipartite graphs. Here we have a matrix C. Each cell, C [i, j], is the cost of matching vertex i (donors) to vertex j (patients). Our original sample scoring (Figure 19) is a good example of this. The goal is to find an assignment of the donors to patients of minimal cost [9]. We'll invert our scoring matrix to fit the problem. Where each cell's new value C[i][j] = largest (C) – C[i][j] (Figure 20).

3.2. The Hungarian Algorithm

We can actually use the Blossom algorithm for the assignment problem. Since the graph is bipartite, there will be no blossoms. But for this problem, we'll use a simpler, specialized method, the Hungarian algorithm.

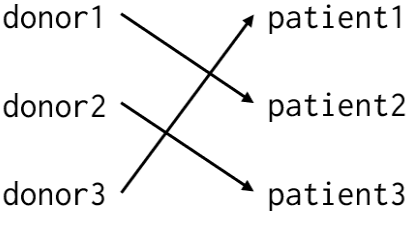


FIGURE 18. An example of a multiway matching.

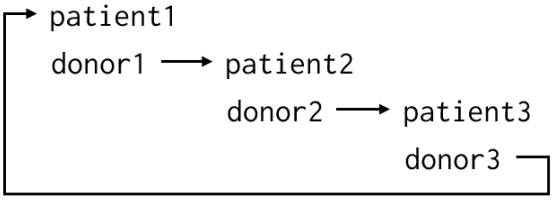


FIGURE 19. A multiway matching cycle/chain.

original

	P1	P2	P3	P4	P5
D1	0	8	3	7	8
D2	2	0	4	2	0
D3	7	5	0	8	9
D4	6	5	7	0	1
D5	2	8	8	5	0

inverted

	P1	P2	P3	P4	P5
D1	9	1	6	2	1
D2	7	9	5	7	9
D3	2	4	9	1	0
D4	3	4	2	9	8
D5	7	1	1	4	9

FIGURE 20. The original scoring from Figure 1 (top) and the inverted matrix (bottom). The largest value in the original scoring is 9. To invert, we subtract 9 with the values in the original cell.

The Hungarian algorithm (or method) was developed by Harold Kuhn in 1955. It solves the assignment problem in polynomial time. Efficient implementations run at . To use this, it requires that the number of rows is equal to the number of columns. This is also known as a balanced assignment. If they are not equal, we create dummy rows or columns with zero values.

The algorithm [10] goes as follows:

Phase 1: Row and column reductions

Step 1: Find the minimum of each row and subtract from each row the minimum value (row reduction).

Step 2: Find the minimum of each column and subtract from each column the minimum value (column reduction).

Phase 2: Optimization of the problem

Step 1: Draw a minimum number of lines to cover all the zeros of the matrix. Entries that are drawn over by lines will be ignored.

a) *Row scanning*:

- Starting from the first row, ask the question: "Is there exactly one zero in that row?" If yes, mark that zero entry and draw a vertical line passing through that zero. Otherwise, skip that row.

- After scanning the last row, check whether all the zeroes are covered with lines.
  If yes, go to step 2.
  Otherwise, do column scanning.

b) *Column scanning*:

- Starting from the first column, ask the question: "Is there exactly one zero in that column?"
  If yes, mark that zero entry and draw a horizontal line passing through that zero.
  Otherwise, skip that column.
- After scanning the last column, check whether all the zeroes are covered with lines.
  If yes, go to step 2.
  Otherwise, select the zeroes "diagonally" opposite with each other (unmarked zeroes that are of not the same column and row; diagonal selection). This means that there is more than one optimal solution.

Diagonal selection:
Starting from the first row, mark a zero (there should be multiple zeroes in the row) and mark a vertical line from that zero.
Mark a zero in the next row of a different column and mark a vertical line from that zero.
Repeat until all rows are done. All zeroes should be covered. We can get the other optimal solutions by marking different zeroes.

Step 2: Check whether the number of marked zeroes is equal to the number of rows of the matrix. If yes, go to step 4, otherwise, go to step 3.

Step 3: Identify the minimum value of the undeleted cell values.

a) Add the minimum undeleted cell value at the intersection points of the current matrix.
b) Subtract the minimum undeleted cell values from all the undeleted cell values.
c) Go back to step 1.

Step 4: We're done. The marked zeroes show the row-column position of the optimal solution (in the original matrix).

3.3. An Example Using the Hungarian Algorithm

Let's use the inverted matrix from Figure 20 for this example. D1, D2,…, D5 are donors and P1, P2,…, P5 are patients.

Phase 1:

Step 1: Row reduction. We find the minimum in each row. Then subtract each row minimum to that row (Figure 21).

Step 2: Column reduction. We find the minimum in each column. Then subtract each column minimum to that column (Figure 22).

Phase 2: Optimization of the problem

	P1	P2	P3	P4	P5	min
D1	9	1	6	2	1	**1**
D2	7	9	5	7	9	**5**
D3	2	4	9	1	0	**0**
D4	3	4	2	9	8	**2**
D5	7	1	1	4	9	**1**

	P1	P2	P3	P4	P5
D1	8	0	5	1	0
D2	2	4	0	2	4
D3	2	4	9	1	0
D4	1	2	0	7	6
D5	6	0	0	3	8

FIGURE 21. Row reduction.

	P1	P2	P3	P4	P5
D1	8	0	5	1	0
D2	2	4	0	2	4
D3	2	4	9	1	0
D4	1	2	0	7	6
D5	6	0	0	3	8
min	**1**	**0**	**0**	**1**	**0**

	P1	P2	P3	P4	P5
D1	7	0	5	0	0
D2	1	4	0	1	4
D3	1	4	9	0	0
D4	0	2	0	6	6
D5	5	0	0	2	8

FIGURE 22. Column reduction.

Step 1a. Row scanning. For each row, if there is only one zero, mark that zero then draw a line over that column. Otherwise, skip that row (Figure 23).

Step 1b. Since not all zeros are covered, we do column scanning. For each column, if there is only one zero, mark that zero then draw a line over that row. Otherwise, skip that column. Take note that we ignore columns marked by row scanning. In this example, we skipped both columns (Figure 24).

Since not all zeros are still covered, we do an additional step called diagonal selection. For each row, we arbitrarily pick a zero then draw a vertical line. We repeat this but for zeros of a different column (Figure 25).

Step 2. We check if the number of marked zeros is equal to the number of rows (Figure 26). Since they are equal, we go to step 4.

Step 4. We are done. The marked zeros show the optimal matches in the original uninverted matrix (Figure 27).

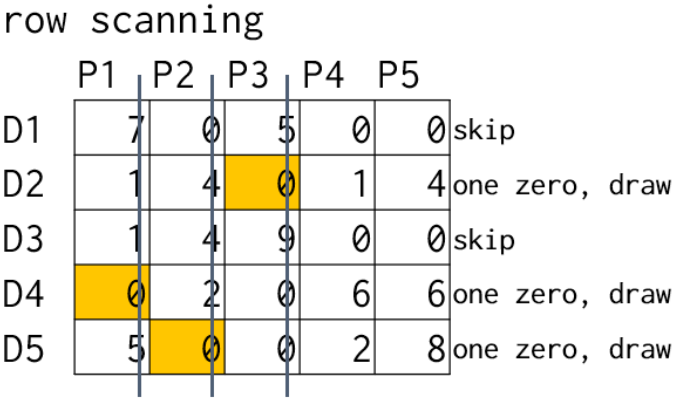


FIGURE 23. Row scanning. Columns that are drawn over will be ignored in the next step.

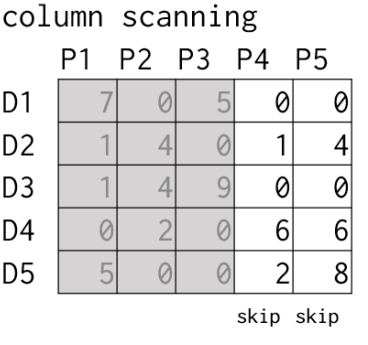


FIGURE 24. Column scanning. No rows were drawn over since both columns have multiple zeros.

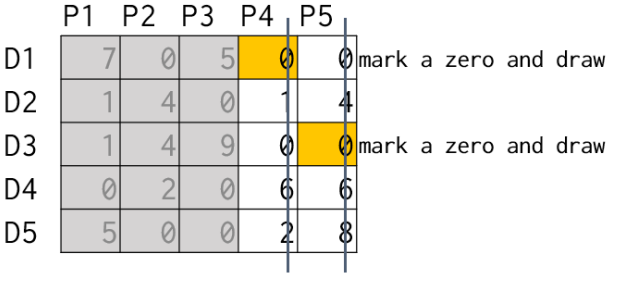


FIGURE 25. Diagonal selection.

	P1	P2	P3	P4	P5
D1	7	0	5	0	0
D2	1	4	0	1	4
D3	1	4	9	0	0
D4	0	2	0	6	6
D5	5	0	0	2	8

FIGURE 26. The number of marked zeros is equal to the number of rows.

	P1	P2	P3	P4	P5
D1	0	8	3	7	8
D2	2	0	4	2	0
D3	7	5	0	8	9
D4	6	5	7	0	1
D5	2	8	8	5	0

Donor	Patient	Score
donor1	patient4	7
donor2	patient3	4
donor3	patient5	9
donor4	patient1	6
donor5	patient2	8
	Total	34

FIGURE 27. The optimal matches in the original matrix (top) and the matches (below).

5. The Application

The app was created using:

- Python 3 for the base code
- NumPy for its fast N-dimensional array object implementation
- NetworkX 2.4 for its graph implementation and max_weight_matching algorithm (Blossom algorithm).
- PySimpleGUI 4.7.0 for the fast and robust GUI

This app uses CSV (comma-separated values) files as input. The CSV format is perfect since scores are in matrix form (Figure 1). We write patient names in the column headers and donor names as row headers. Each donor-patient matching score is written in the corresponding cell. Popular applications like MS Excel or Google Spreadsheets already provide an easy way to write these. For saving results, the app uses the CSV format as well.

Current features:

- CSV file opening and saving – with basic CSV format checking
- Score matrix view – a view of the opened CSV file. Currently limited to 8 columns due to PySimple GUI limitations
- Choose a matching option – pairwise or multiway matching
- For pairwise matching, there is an option for using regular averaging or modified, and to not add edges with zero weight to make the graph sparser
- Matching results view – a simple view showing the results

Sample tests and results:

1) Using test data from Figure 1. 5 donor-patient pairs with complete data (Figures 28 and 29).

2) 8 donor-patient pairs with incomplete data (Figures 30 and 31).

3) 15 donor-patient pairs with complete scorings. Even with a large matrix (15 × 15 scorings), the app ran this almost instantaneously (Figure 32).

6. Conclusion

Overall, this app provides a fast and easy way to match donors to patients. The algorithms used here are tested and proven to be fast and optimal. Run times for the app itself were

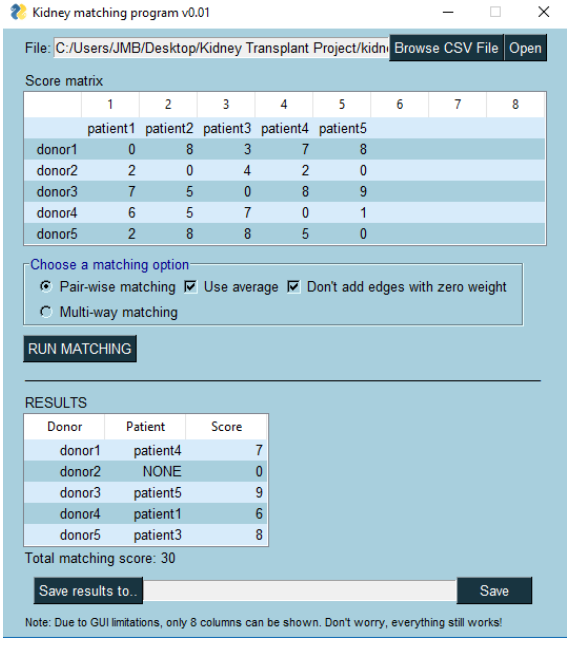


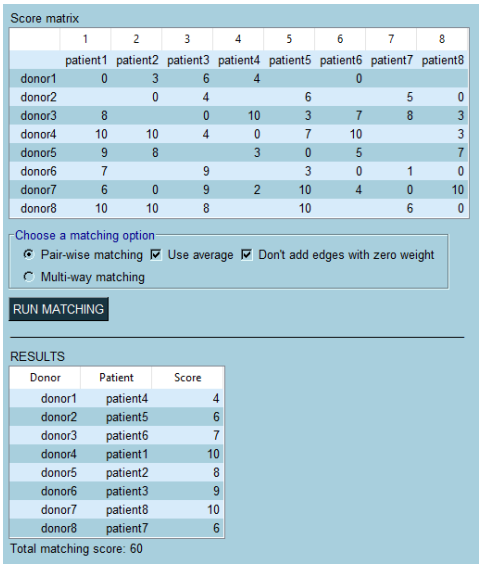
FIGURE 28. The kidney matching app using the matrix from Figure 1 as input. Shown here are pairwise matching results.

RESULTS

Donor	Patient	Score
donor1	patient4	7
donor2	patient3	4
donor3	patient5	9
donor4	patient1	6
donor5	patient2	8

Total matching score: 34

FIGURE 29. Using the matrix from Figure 1. Shown here are the multiway matching results.



	1	2	3	4	5	6	7	8
	patient1	patient2	patient3	patient4	patient5	patient6	patient7	patient8
donor1	0	3	6	4		0		
donor2		0	4		6		5	0
donor3	8		0	10	3	7	8	3
donor4	10	10	4	0	7	10		3
donor5	9	8		3	0	5		7
donor6	7		9		3	0	1	0
donor7	6	0	9	2	10	4	0	10
donor8	10	10	8		10		6	0

Donor	Patient	Score
donor1	patient4	4
donor2	patient5	6
donor3	patient6	7
donor4	patient1	10
donor5	patient2	8
donor6	patient3	9
donor7	patient8	10
donor8	patient7	6

FIGURE 30. Pairwise results for an incomplete 8 donor-patient matrix.

RESULTS

Donor	Patient	Score
donor1	patient3	6
donor2	patient7	5
donor3	patient4	10
donor4	patient6	10
donor5	patient2	8
donor6	patient1	7
donor7	patient8	10
donor8	patient5	10

Total matching score: 66

FIGURE 31. Multiway matching results for an incomplete 8 donor-patient matrix.

RESULTS

Donor	Patient	Score
donor1	patient8	10
donor2	patient14	8
donor3	patient11	9
donor4	patient12	9
donor5	NONE	0
donor6	patient10	5
donor7	patient9	9
donor8	patient1	10
donor9	patient7	7
donor10	patient6	10
donor11	patient3	10
donor12	patient4	5
donor13	patient15	10
donor14	patient2	8
donor15	patient13	10

Total matching score: 120

RESULTS

Donor	Patient	Score
donor1	patient10	10
donor2	patient9	9
donor3	patient14	9
donor4	patient8	10
donor5	patient12	8
donor6	patient3	10
donor7	patient2	9
donor8	patient1	10
donor9	patient4	9
donor10	patient7	9
donor11	patient5	10
donor12	patient11	10
donor13	patient15	10
donor14	patient6	10
donor15	patient13	10

Total matching score: 143

FIGURE 32. Pairwise (left) and multiway (right) matching results on a 15 donor-patient pairs test.

fast. The app had no trouble for even large and unlikely scenarios, e.g. a 20 × 20 patient-donor matrix. With these in mind, new kidney exchange programs may integrate this app for their use.

Future improvements are still being planned for this app. We hope that this app gets real-world use. The app and test cases are available at bit.ly/2KPXUAX or just email me at jjbawagan@up.edu.ph. Feedback is most welcome.

References

1. End-stage renal disease. https://www.health.harvard.edu/a_to_z/end-stage-renal-disease-a-to-z. Date accessed: 12/2018.
2. National Kidney and Transplant Institute. http://www.nkti.gov.ph/index.php/11-services/654-kidney-transplantation. Date accessed: 2019.
3. Kidney disease: the basics. https://www.kidney.org/news/newsroom/factsheets/KidneyDiseaseBasics. Date accessed: 2019.
4. Roth AlE, Sonmez T, Utku Unver M. Pairwise kidney exchange. *Journal of Economic Theory*. 2005; 125(2), 151–188. doi: 10.3386/w10698.
5. Edmonds J. Paths, trees, and flowers. *Canadian Journal of Mathematics*. 1965; 17, 449–467. doi:10.4153/CJM-1965-045-4.
6. Matching in general graphs. http://www.cs.tau.ac.il/~zwick/grad-algo-06/match.pdf. Date accessed: 2006.
7. Karp R. Edmonds's non-bipartite matching algorithm. CS294-5: great algorithms. U.C. Berkeley. 2006, 1–9. http://webdocs.cs.ualberta.ca/~hayward/304/edmondsmatching.pdf.

8. Kruskal's algorithm. Technische Universität München. 2016. https://www-m9.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/index_en.html. Date accessed: 2016.
9. Munkres J. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*. 1957; 5(1), 32–38. https://doi.org/10.1137/0105003.
10. Assignment problem [easy steps to solve – Hungarian method with optimal solution]. https://www.youtube.com/watch?v=rrfFTdO2Z7I. Date accessed: 27/09.2016.